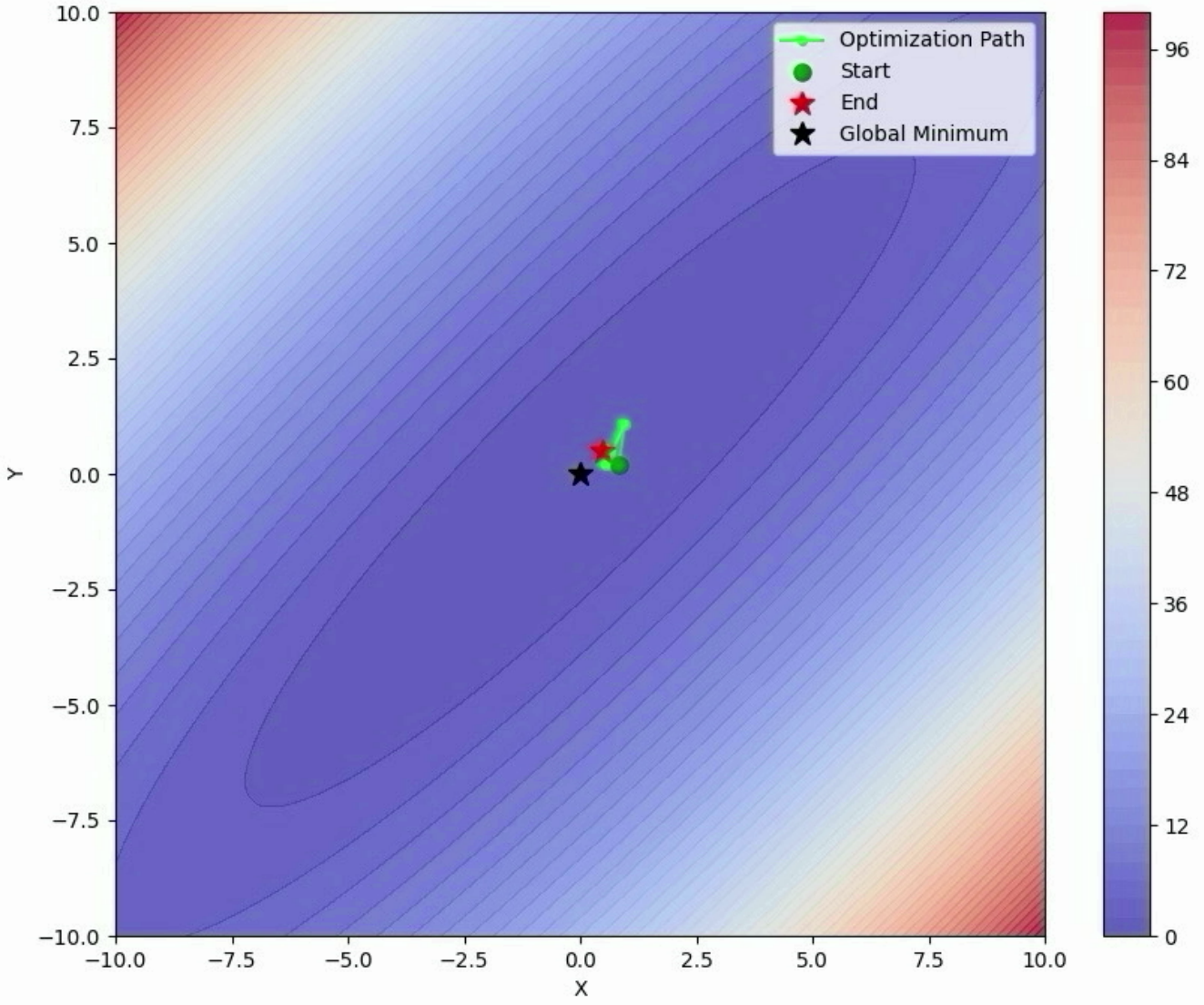


The Role of Genetic Algorithms in Function Optimization

Optimization Algorithms for Function Optimization

Using Genetic Algorithm & Differential Evolution

👥 Team Members

Name	Role	ID
Ali Ezz Ali	Project Manager & Coordinator	931230187
Ahmed Abo Bakr	Lead Genetic Algorithm Developer	931230015
Ahmed Sabry Hamza	UI Developer	931230030
Saeed Nadim Saeed	Literature Reviewer & Documenter	931230139
Shehab Hossam eldien	System Designer & Analyst	931230153
Abdelrahman Ezat	Experiment Lead & Results Analyst	931230162

Table of Contents:

1. Introduction to Optimization Algorithms

Modern real-world optimization problems—ranging from engineering design to financial modeling—are often too difficult for simple analytical methods, so advanced computational approaches are needed. This project focuses on two state-of-the-art population-based algorithms: **Genetic Algorithm (GA)** and **Differential Evolution (DE)**.

Genetic Algorithm (GA) – Detailed Overview

A Genetic Algorithm is a search and optimization technique inspired by the process of natural selection and genetics found in biological systems.

Key Elements:

- **Population:** GA begins with a group of candidate solutions (called chromosomes or individuals).
- **Fitness Function:** Each individual is evaluated according to a fitness function, which measures how good the solution is for the specific problem.
- **Selection:** The algorithm selects the better-performing individuals for reproduction, favoring those with higher fitness.
- **Crossover (Recombination):** Pairs of selected individuals exchange parts of their data (genes) to generate new offspring. This simulates biological reproduction.
- **Mutation:** Small random changes are applied to some individuals to maintain genetic diversity and explore new solutions.
- **Replacement:** Offspring replace some or all members of the previous generation.

Typical GA Parameters:

- *Population Size:* 30 to 200 (commonly 50–100)
- *Crossover Rate:* ~0.6–0.95
- *Mutation Rate:* ~0.001–0.1 (higher rates can help exploration for multimodal problems)
- *Selection Type:* Roulette wheel, tournament, rank-based, etc.
- *Crossover Type:* One-point, two-point, uniform, etc.

Process Summary:

1. Initialize a random population.
2. Evaluate all individuals' fitness.
3. Repeat for a set number of generations or until goal:
    - Select parents based on fitness.
    - Apply crossover and mutation to create offspring.
    - Evaluate the new population.
    - Optionally, carry forward the best solutions (elitism).

Strengths:

- Very flexible, can solve discrete, continuous, and mixed problems.
- Good at escaping local optima due to mutation and population diversity.
- Widely used in engineering, scheduling, machine learning, evolutionary design.

Limitations:

- May require many generations for very large, high-dimensional problems.
- Performance depends on effective parameter tuning and representation.

Differential Evolution (DE) – Detailed Overview

Differential Evolution is a population-based optimization method particularly suited for continuous-valued functions.

Key Elements:

- **Population:** Like GA, DE works with a population of candidate solutions (vectors).
- **Mutation (Vector Differential):** Instead of operating on bits, DE creates a "trial vector" for each target solution by adding the weighted difference between two population members to a third member. (Formula: `v = x_a + F * (x_b - x_c)`)
- **Crossover:** Trial vectors are mixed with target solutions using a crossover probability to generate a child.
- **Selection:** The algorithm compares the child and target; the one with higher fitness survives to the next generation.

Typical DE Parameters:

- *Population Size:* Often 10 times the number of variables, e.g., 50–100.
- *Mutation Factor (F):* Ranges from 0.4 to 1.0 (commonly 0.5–0.9).
- *Crossover Rate (CR):* 0.7–0.95.
- *Strategy Type:* "DE/rand/1/bin" is most frequently used (random base, one differential vector, binomial crossover).

Process Summary:

1. Initialize a random population within variable bounds.
2. For each individual:
    - Mutation: Compute a donor vector via differences plus the scaling factor.
    - Crossover: Mix donor and target using CR to form a trial individual.
    - Selection: Evaluate trial vs. current; the one with better fitness continues.
3. Repeat until stopping criteria are met.

Strengths:

- Very effective for real-valued unconstrained and box-constrained optimization.
- Fewer control parameters than GA, making tuning simpler.
- Robust, handles rugged landscapes and high dimensions well.
- Can converge more quickly than GA for many continuous optimization benchmarks.

Limitations:

- Less suited for discrete or combinatorial problems.
- Like GA, sensitive to parameter choices and may require normalization/scaling.

Why Compare GA and DE?

- Both are **population-based**, using multiple solutions to avoid local minima.
- **GA** is more general, often chosen for problems with complex constraints, discrete settings, and for multipurpose evolutionary tasks.
- **DE** is specialized for real-valued and continuous optimization, typically providing faster convergence and better solution accuracy for such domains.
- Comparing both algorithms on standard benchmark functions gives insight into their relative strengths, convergence patterns, and suitability for a given problem.

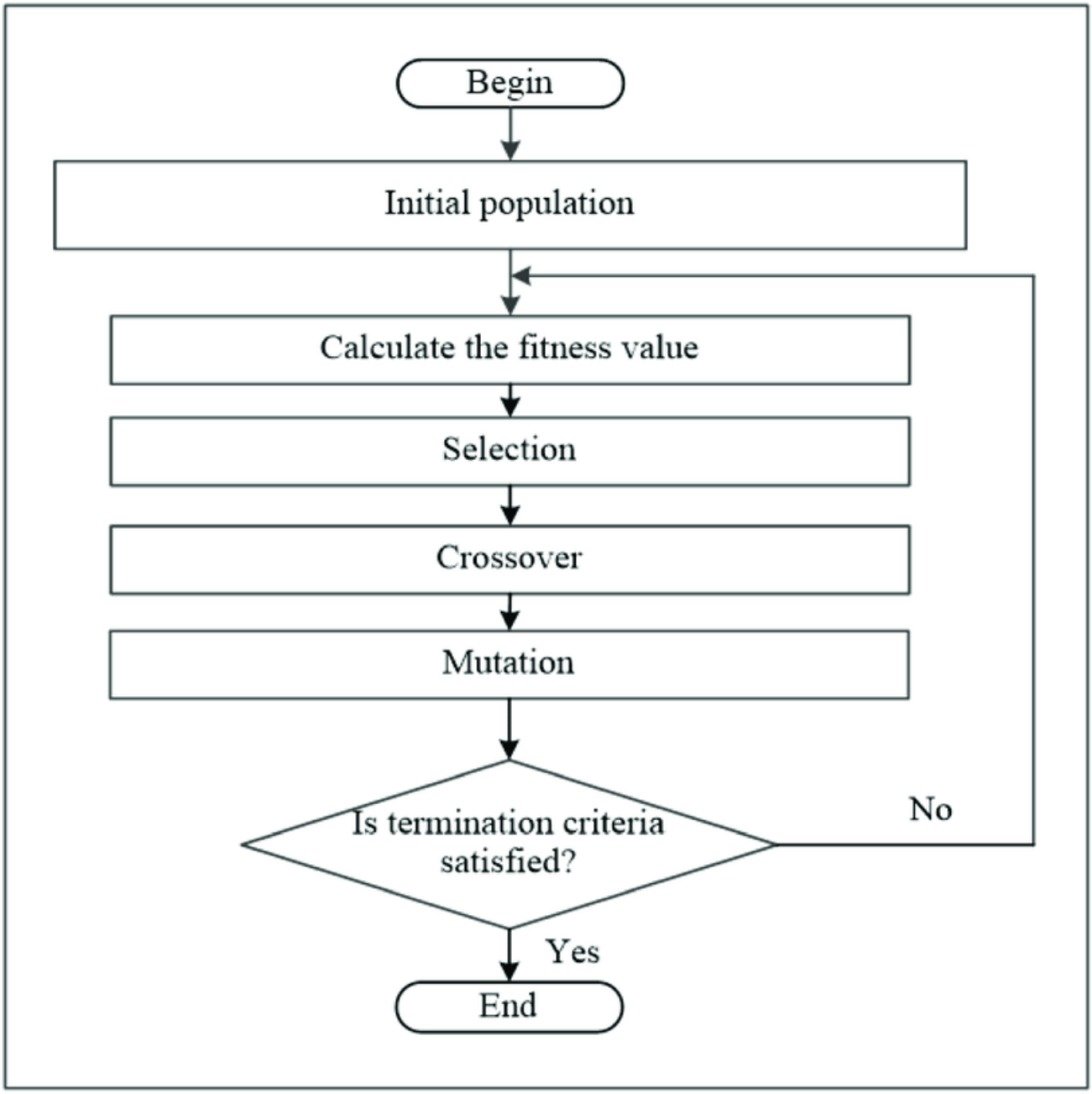
Example Real-World Applications

- **Genetic Algorithm:**
    - Automated scheduling (e.g., university timetables, airline crew rostering)
    - Feature selection in machine learning
    - Engineering design and robot path planning
    - Game AI and strategy evolution
- **Differential Evolution:**
    - Hyperparameter tuning for deep learning models
    - Mechanical design parameter optimization
    - Antenna array synthesis
    - Chemical process parameter fitting

Visualizations

To further clarify, our project includes process flowcharts and block diagrams for both GA and DE (see below), showing all stages from initialization to termination.

Genetic Algorithm Flowchart



Flowchart showing initialization, fitness evaluation, selection, crossover, mutation, and next generation cycle.

Genetic Algorithm Block Diagram

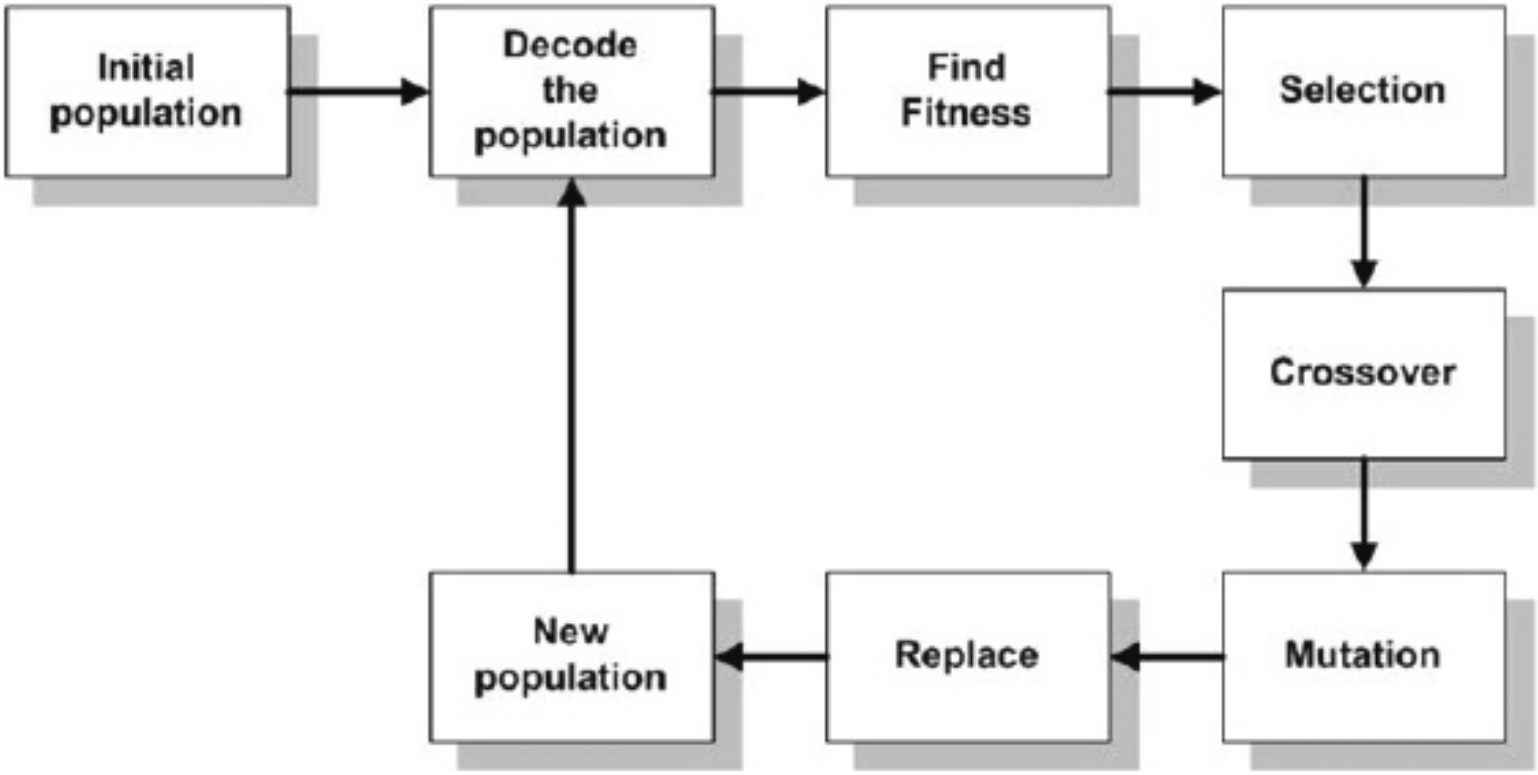
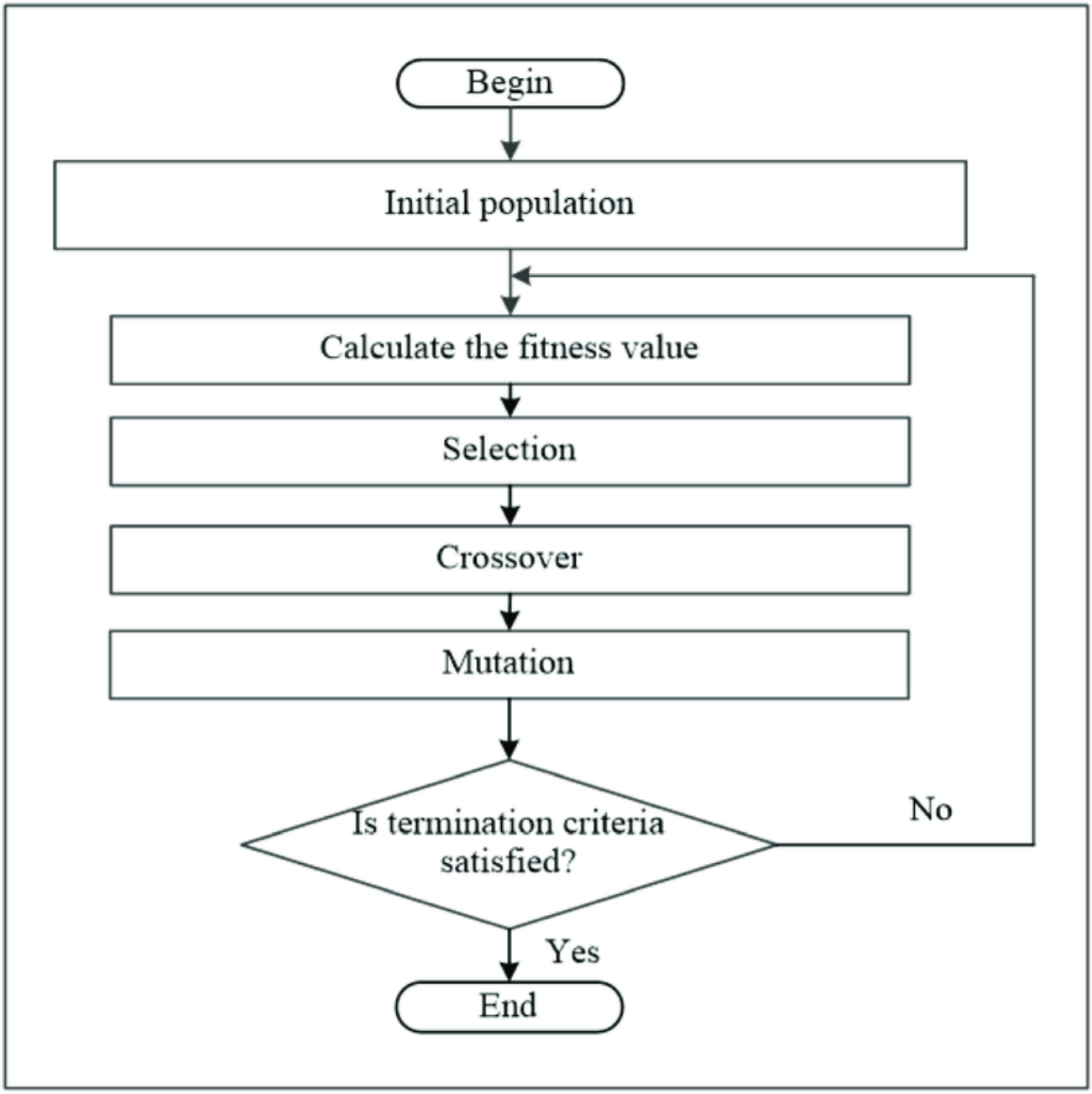


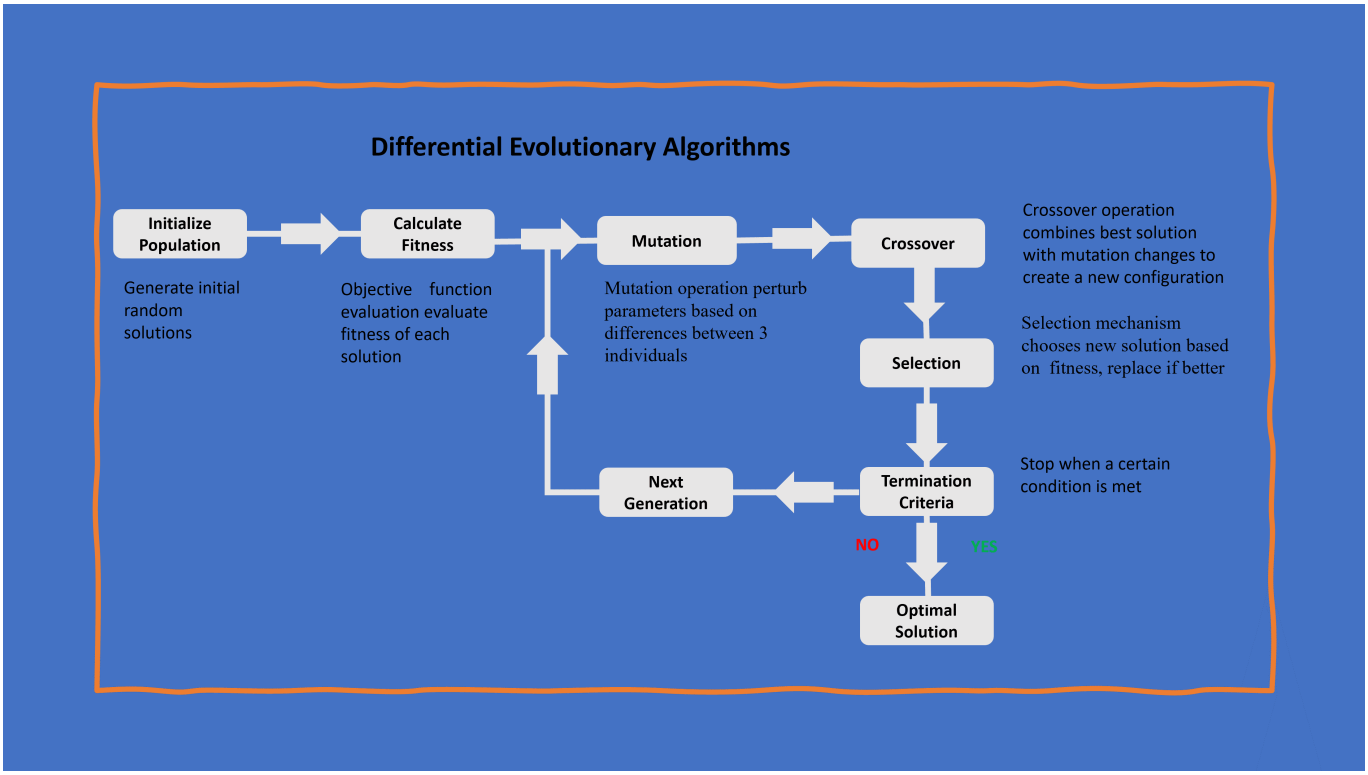
Diagram capturing the information flow of a GA cycle.

Differential Evolution Flowchart



A typical DE cycle highlighting mutation, crossover, selection, and generation update.

Differential Evolution Block Diagram



Visual process of DE, emphasizing trial vector creation and survivor selection.

---

2. Test Functions for Optimization

Benchmark functions offer known minima and challenging landscapes for testing optimization methods, serving as a crucial tool for comparing and analyzing algorithms in a controlled way.

These functions are carefully chosen to represent a wide variety of optimization difficulties:

- **Dimensionality:** Some functions have just two variables, while others have higher dimensions, simulating real-world challenges such as many-parameter engineering or machine learning tasks.
- **Surface Topology:** Convex bowls (easy), long and narrow valleys, deceptive plateaus, and multi-modal surfaces mimic actual problem geometries, making them ideal for testing a method's ability to escape local optima.
- **Complexity Level:** While some functions are considered "easy" for optimizers, others, like Rosenbrock and Zakharov, are famous for trapping algorithms in difficult-to-escape regions.
- **Reproducibility and Analysis:** Because the global minimum is known, benchmark functions help measure not only accuracy but also speed, stability, and reliability across repeated runs.

Why Use Benchmark Functions?

- Allow comparison between optimization algorithms under standardized conditions. This means results are fair and repeatable, not dependent on problem-specific quirks.
- Known global minima provide fair assessment; you can judge how close the solution is, whether the algorithm always finds it, and how quickly convergence occurs.
- Enable controlled experiments that test how different parameters or methods affect optimization performance, without confounding external variables.

Functions Used – Table Summary

Name	Vars	Formula	Overview	Global Min
Booth	2	(x+2y-7)^2 + (2x+y-5)^2	Simple, convex	(1,3)
Matyas	2	0.26(x^2+y^2) - 0.48xy	Bowl, (0,0)	(0,0)
Rosenbrock	4	sum 100(x_{i+1}-x_i^2)^2 + (1-x_i)^2	Long/narrow valley	all x_i=1
Sphere	5	sum x_i^2	Ball at origin	all x_i=0
Zakharov	5	sum x_i^2 + (sum 0.5ix_i)^2 + (sum 0.5ix_i)^4	Complex landscape	all x_i=0

Each function is selected to test different behaviors: Booth and Matyas reward algorithms that converge quickly in smooth environments; Rosenbrock tests the ability to search along tricky ridges; Sphere provides a baseline for uncorrelated variable search; Zakharov pushes algorithms to manage interaction and nonlinearity among variables.

Mathematical Formulas

- **Booth**: $f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$
- **Matyas**: $f(x, y) = 0.26(x^2 + y^2) - 0.48xy$
- **Rosenbrock**: $f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (1-x_i)^2]$
- **Sphere**: $f(x) = \sum x_i^2$
- **Zakharov**: $f(x) = \sum x_i^2 + (\sum 0.5ix_i)^2 + (\sum 0.5ix_i)^4$

Interpreting Results & Visualization

- **Surface/Contour Plots:** Visualizing 2D or 3D functions helps intuitively grasp how algorithms explore, converge, or get trapped in certain landscapes.
- **Best-versus-Average Runs:** Test functions can reveal if an algorithm is robust (performing well across multiple runs) or if it occasionally finds the optimum but often gets stuck.
- **Progress over Generations:** Tracking fitness improvement helps analyze the balance between exploration (diversity) and exploitation (fast convergence) for each algorithm.

Real-World Parallels

Although often synthetic, such functions are analogues for real optimization problems—objectives in engineering, finance, or AI modeling may have similar rugged, deceptive, or plateaued landscapes. Insights from these benchmarks thus guide algorithm selection and tuning for practical applications.

---

3. Code Explanation

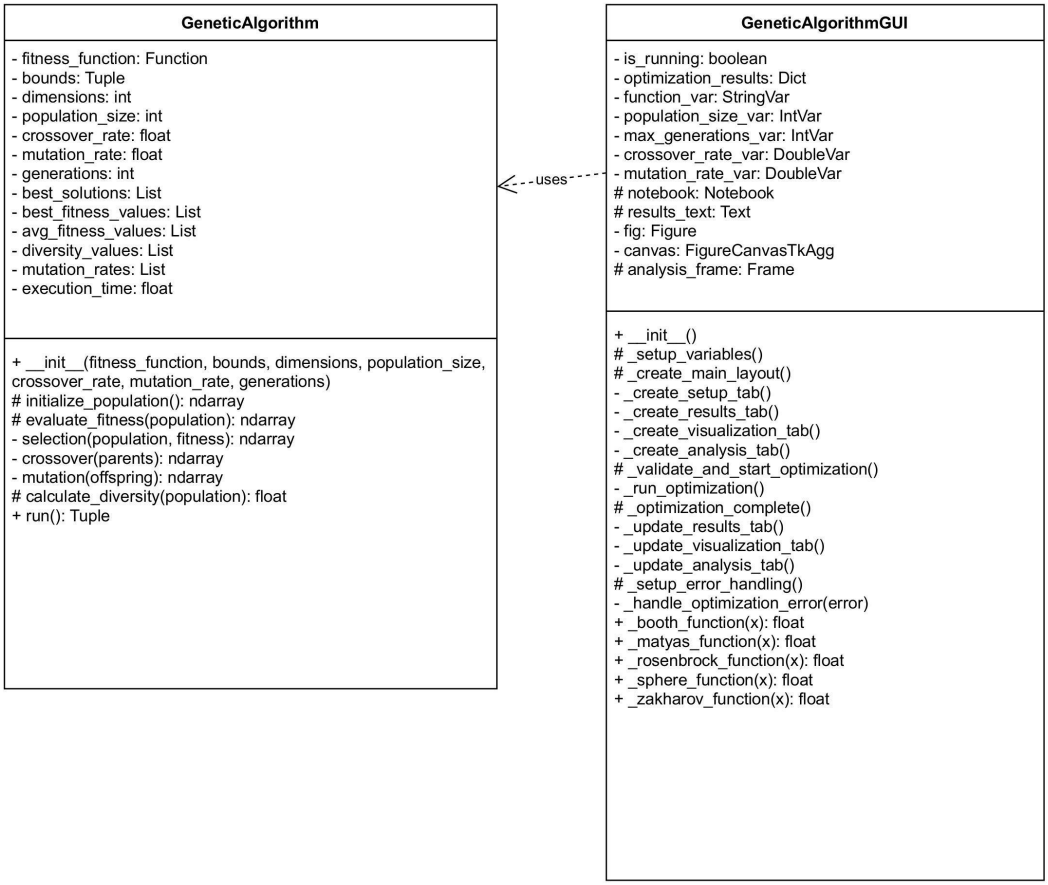
Detailed code explanations and algorithm walkthroughs are provided in a supplementary

In our codebase, *both GA and DE implementations* are modular, allowing function benchmarks to be optimized by either algorithm with appropriate parameters. The code structure provides an interface to select the optimization method and visualize progress.

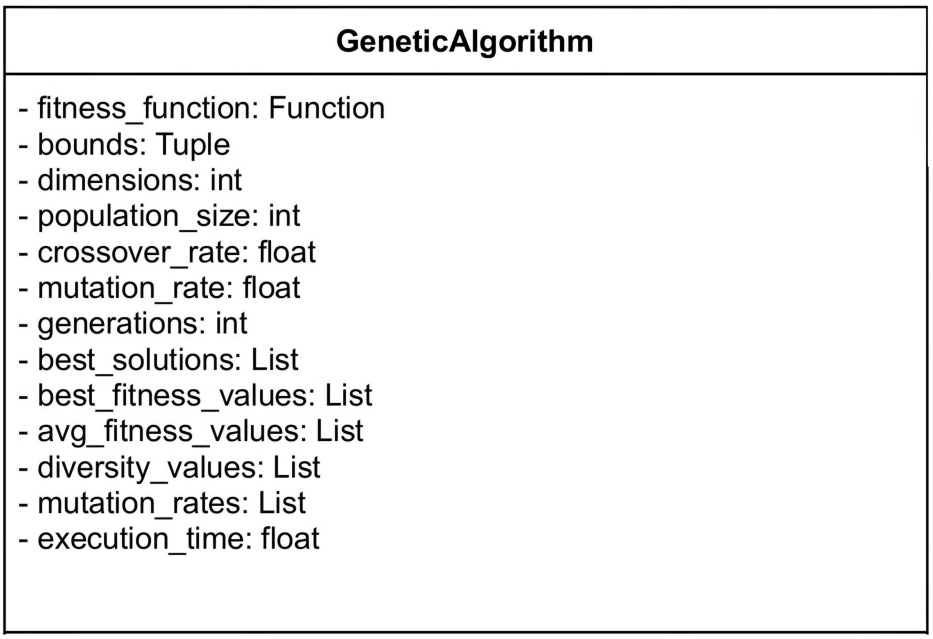
Detailed Code Analysis with Steps

UML OF GeneticAlgorithm Class Diagram

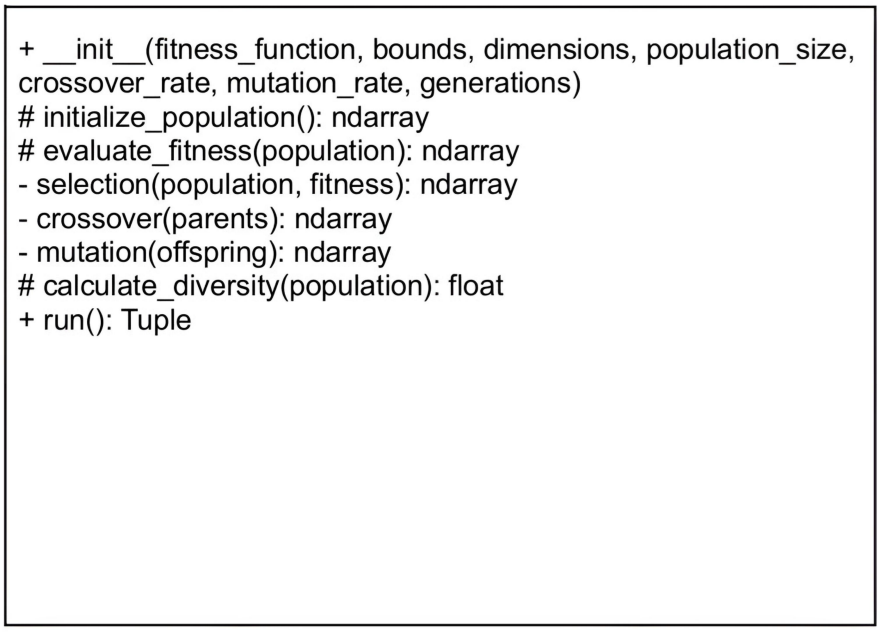
1. GeneticAlgorithm



- **Classes**: 2
- **Attributes**: 20
- **Operations (Methods)**: 31
- **Attributes:**



  - `fitness_function`: Function to evaluate the fitness of individuals. (Initialized in `__init__`)
  - `bounds`: Tuple for the search space bounds. (Initialized in `__init__`)
  - `dimensions`: Number of dimensions in the problem. (Initialized in `__init__`)
  - `population_size`: Size of the population. (Initialized in `__init__`)
  - `crossover_rate`: Rate of crossover. (Initialized in `__init__`)
  - `mutation_rate`: Rate of mutation. (Initialized in `__init__`)
  - `generations`: Number of generations. (Initialized in `__init__`)
  - `best_solutions`: List to store best solutions. (Initialized in `__init__`)
  - `best_fitness_values`: List to store best fitness values. (Initialized in `__init__`)
  - `avg_fitness_values`: List to store average fitness values. (Initialized in `__init__`)
  - `diversity_values`: List to store diversity values. (Initialized in `__init__`)
  - `mutation_rates`: List to store mutation rates. (Initialized in `__init__`)
  - `execution_time`: Time taken for execution. (Initialized in `__init__`)
- **Methods and Steps:**



- `__init__`: Initializes the genetic algorithm.

```
def __init__(self, fitness_function, bounds, dimensions, population_size=100,
crossover_rate=0.8, mutation_rate=0.1, generations=1000):
    ...
```

  - Step 1: Set parameters including fitness function, bounds, dimensions, etc.
- `initialize_population`: Generates a random initial population.

```
def initialize_population(self):
    ...
```

  - Step 2: Generate random individuals within bounds.
- `evaluate_fitness`: Evaluates fitness for each individual.

```
def evaluate_fitness(self, population):
    ...
```

  - Step 3: Compute fitness values for the population.
- `selection`: Performs tournament selection.

```
def selection(self, population, fitness):
    ...
```

  - Step 4: Select individuals based on fitness.
- `crossover`: Performs crossover between parents.

```
def crossover(self, parents):
    ...
```

  - Step 5: Create offspring through crossover.
- `mutation`: Applies mutation to offspring.

```
def mutation(self, offspring):
    ...
```

  - Step 6: Introduce variation via mutation.
- `calculate_diversity`: Calculates population diversity.

```
def calculate_diversity(self, population):
    ...
```

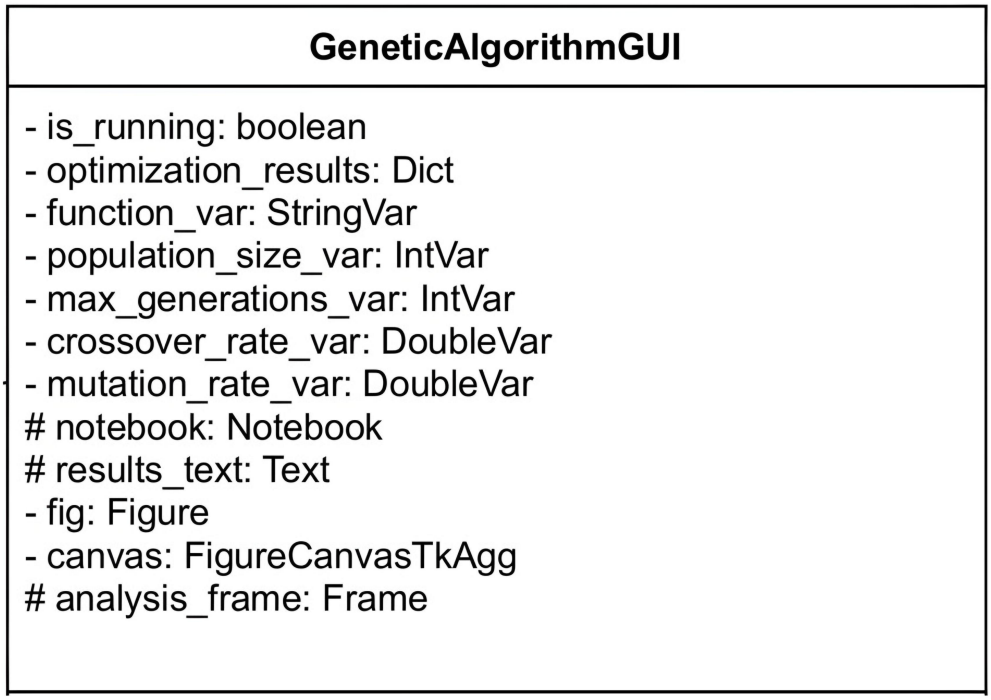
        - Step 7: Measure diversity of the population.
    - `run`: Main loop for optimization.

```
def run(self):
    ...
```

        - Step 8: Execute the genetic algorithm loop.
        - Step 9: Track best solutions and fitness values.
        - Step 10: Record execution time.

2. GeneticAlgorithmGUI

- **Attributes:**



    - `is_running`: Indicates if optimization is active. (Initialized in `__init__`)
    - `optimization_results`: Stores optimization results. (Initialized in `__init__`)
    - `function_var`: Tkinter variable for function selection. (Initialized in `_setup_variables`)
    - `population_size_var`: Tkinter variable for population size. (Initialized in `_setup_variables`)
    - `max_generations_var`: Tkinter variable for generations. (Initialized in `_setup_variables`)
    - `crossover_rate_var`: Tkinter variable for crossover rate. (Initialized in `_setup_variables`)
    - `mutation_rate_var`: Tkinter variable for mutation rate. (Initialized in `_setup_variables`)
    - `notebook`: Notebook for GUI tabs. (Initialized in `_create_main_layout`)
    - `results_text`: Text widget for displaying results. (Initialized in `_create_results_tab`)
    - `fig`: Matplotlib figure for plotting. (Initialized in `_create_visualization_tab`)
    - `canvas`: Canvas for embedding plots. (Initialized in `_create_visualization_tab`)
    - `analysis_frame`: Frame for analysis tab. (Initialized in `_create_analysis_tab`)
- **Methods and Steps:**

- `__init__`: Initializes the GUI.

```
def __init__(self):
    ...
```

  - Step 1: Set up window configuration.
  - Step 2: Initialize variables and layout.
- `_setup_variables`: Sets up optimization and GUI variables.

```
def _setup_variables(self):
    ...
```

  - Step 3: Define available optimization functions.
- `_create_main_layout`: Creates the main GUI layout.

```
def _create_main_layout(self):
    ...
```

        - Step 4: Create main frame and notebook.
    - `_create_setup_tab`: Creates setup tab for input parameters.

      ```
      def _create_setup_tab(self):
          ...
      ```

        - Step 5: Add dropdowns and entries for parameters.
    - `_validate_and_start_optimization`: Validates input and starts optimization.

      ```
      def _validate_and_start_optimization(self):
          ...
      ```

        - Step 6: Validate inputs and handle errors.
        - Step 7: Start optimization in a separate thread.
    - `_run_optimization`: Core logic for running optimization.

      ```
      def _run_optimization(self):
          ...
      ```

        - Step 8: Execute genetic algorithm and store results.
    - `_optimization_complete`: Updates GUI on completion.

      ```
      def _optimization_complete(self):
          ...
      ```

        - Step 9: Update results, visualization, and analysis tabs.
    - `_update_results_tab`: Updates results tab with data.

      ```
      def _update_results_tab(self):
          ...
      ```

        - Step 10: Display best solution and parameters.
    - `_update_visualization_tab`: Generates plots for visualization.

      ```
      def _update_visualization_tab(self):
          ...
      ```

        - Step 11: Plot fitness, diversity, and mutation rates.
    - `_update_analysis_tab`: Provides statistical analysis.

      ```
      def _update_analysis_tab(self):
          ...
      ```

        - Step 12: Display detailed analysis of results.
    - `_setup_error_handling`: Sets up error handling mechanisms.

      ```
      def _setup_error_handling(self):
          ...
      ```

        - Step 13: Define error handler for exceptions.
    - **Objective Functions**: Defines functions for optimization.

      ```
      def _booth_function(self, x):
          ...
      ```

        - Step 14:
- **Objective Functions Continued:**
    - `_matyas_function`:
        - Defines the Matyas function for optimization.
        - Step 15: Implement the function to calculate the objective value.

      ```
      def _matyas_function(self, x):
          return 0.26 * (x[0]**2 + x[1]**2) - 0.48 * x[0] * x[1]
      ```

- `_rosenbrock_function`:
  - Defines the Rosenbrock function for optimization.
  - Step 16: Implement the function to calculate the objective value.

```
def _rosenbrock_function(self, x):
    return sum(100.0 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x)-1))
```

- `_sphere_function`:
  - Defines the Sphere function for optimization.
  - Step 17: Implement the function to calculate the objective value.

```
def _sphere_function(self, x):
    return np.sum(x**2)
```

- `_zakharov_function`:
  - Defines the Zakharov function for optimization.
  - Step 18: Implement the function to calculate the objective value.

```
def _zakharov_function(self, x):
    sum1 = np.sum(x**2)
    sum2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x)
    return sum1 + sum2**2 + sum2**4
```

Error Handling

- **Methods and Steps:**
  - `_handle_optimization_error`:
    - Manages errors during the optimization process.
    - Step 19: Display error messages using message boxes.
    - Step 20: Optionally log the full traceback for debugging purposes.

```
def _handle_optimization_error(self, error):
    error_msg = str(error)
    traceback_msg = traceback.format_exc()
    messagebox.showerror("Optimization Error", f"An error occurred during optimization:
\n{error_msg}")
    print(traceback_msg)
```

  - `_setup_error_handling`:
    - Global exception handling for the application.
    - Step 21: Define a global error handler to catch unexpected errors and display them.

```
def _setup_error_handling(self):
    def error_handler(exc_type, exc_value, exc_traceback):
        error_msg = ''.join(traceback.format_exception(exc_type, exc_value, exc_traceback))
        messagebox.showerror("Unexpected Error", error_msg)
    tk.Tk.report_callback_exception = error_handler
```

Main Function

- `main()`:
  - Entry point for the application.
  - Step 22: Initialize the `GeneticAlgorithmGUI` application.
  - Step 23: Run the main loop of the Tkinter application.
  - Step 24: Handle any startup errors and display them if necessary.

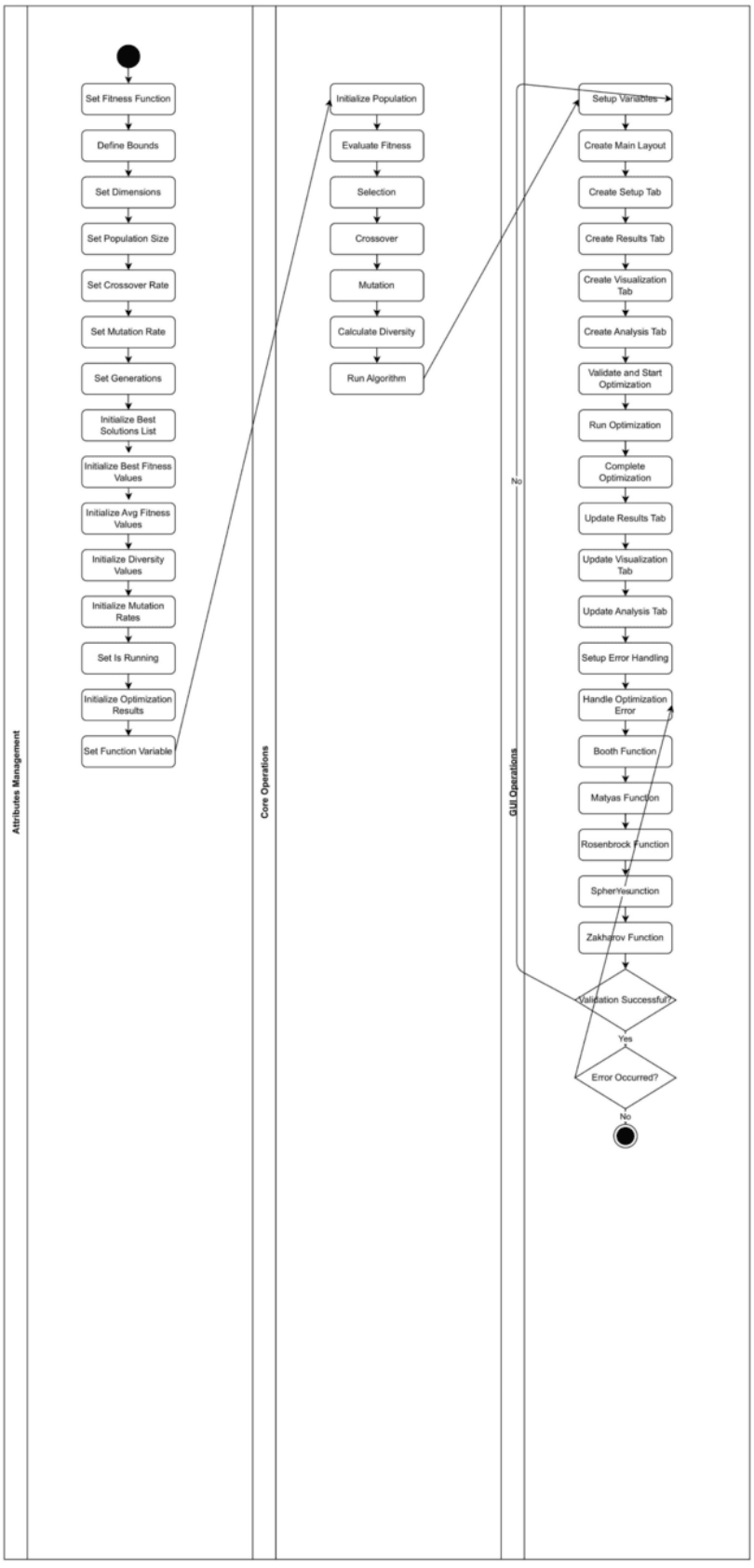
```
def main():
    try:
        app = GeneticAlgorithmGUI()
        app.mainloop()
    except Exception as e:
        print(f"Application startup error: {e}")
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

Component	Attribute/Method	Description
GeneticAlgorithm Class		
Attributes	`fitness_function`	Function used to evaluate the fitness of individuals in the population. Initialized in `__init__`.
	`bounds`	Tuple defining the search space bounds. Initialized in `__init__`.
	`dimensions`	Number of dimensions in the optimization problem. Initialized in `__init__`.
	`population_size`	Size of the population. Initialized in `__init__`.
	`crossover_rate`	Rate at which crossover occurs. Initialized in `__init__`.
	`mutation_rate`	Rate at which mutation occurs. Initialized in `__init__`.
	`generations`	Number of generations for the algorithm to run. Initialized in `__init__`.
	`best_solutions`	List storing the best solutions found. Initialized in `__init__`.
	`best_fitness_values`	List storing the best fitness values per generation. Initialized in `__init__`.
	`avg_fitness_values`	List storing the average fitness values per generation. Initialized in `__init__`.
	`diversity_values`	List storing diversity values per generation. Initialized in `__init__`.
	`mutation_rates`	List storing mutation rates per generation. Initialized in `__init__`.
	`execution_time`	Time taken for the execution of the algorithm. Initialized in `__init__`.
Methods	`__init__`	Initializes the genetic algorithm with parameters such as fitness function, bounds, dimensions, etc.
	`initialize_population`	Generates a random initial population bounded by specified limits.
	`evaluate_fitness`	Computes and returns the fitness values for each individual in the population.

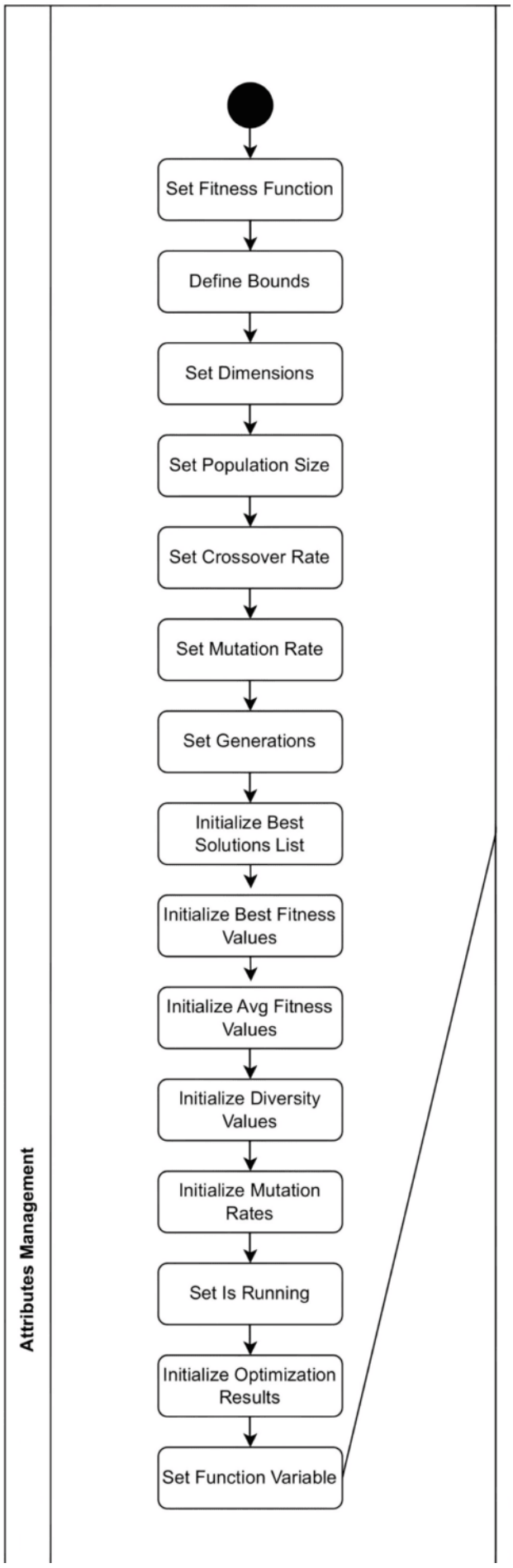
Component	Attribute/Method	Description
	`selection`	Implements tournament selection to choose parents based on fitness.
	`crossover`	Performs uniform crossover between pairs of parents based on a crossover rate.
	`mutation`	Mutates offspring genes randomly based on mutation rate, ensuring respect for search space bounds.
	`calculate_diversity`	Calculates and returns the diversity of the population based on standard deviation across dimensions.
	`run`	Executes the genetic algorithm for the specified number of generations, tracking best solutions and execution time.
GeneticAlgorithmGUI Class		
Attributes	`is_running`	Indicates whether the optimization process is currently active. Initialized in `__init__`.
	`optimization_results`	Stores the results of the optimization process. Initialized in `__init__`.
	`function_var`	Tkinter variable for selecting the optimization function. Initialized in `_setup_variables`.
	`population_size_var`	Tkinter variable for setting the population size. Initialized in `_setup_variables`.
	`max_generations_var`	Tkinter variable for setting the number of generations. Initialized in `_setup_variables`.
	`crossover_rate_var`	Tkinter variable for setting the crossover rate. Initialized in `_setup_variables`.
	`mutation_rate_var`	Tkinter variable for setting the mutation rate. Initialized in `_setup_variables`.
	`notebook`	Notebook widget for organizing GUI tabs. Initialized in `_create_main_layout`.
	`results_text`	Text widget for displaying results. Initialized in `_create_results_tab`.
	`fig`	Matplotlib figure for plotting visualizations. Initialized in `_create_visualization_tab`.
	`canvas`	Canvas for embedding plots into the GUI. Initialized in `_create_visualization_tab`.

Component	Attribute/Method	Description
	`analysis_frame`	Frame for displaying analysis information. Initialized in `_create_analysis_tab`.
Methods	`__init__`	Sets up the window configuration, initializes variables, and creates the GUI layout.
	`_setup_variables`	Initializes optimization and GUI variables and defines available optimization functions.
	`_create_main_layout`	Sets up the main GUI layout and organizes tabs for setup, results, visualization, and analysis.
	`_create_setup_tab`	Adds setup tab to the notebook with dropdowns and entries for optimization parameters.
	`_validate_and_start_optimization`	Validates input parameters, handles errors, and starts the optimization process in a separate thread.
	`_run_optimization`	Executes the genetic algorithm, stores results, and updates the UI upon completion.
	`_optimization_complete`	Updates results, visualization, and analysis tabs after optimization completes, and displays a completion message.
	`_update_results_tab`	Updates the results tab with the best solution and parameters after optimization.
	`_update_visualization_tab`	Generates and displays plots for fitness, diversity, and mutation rates in the visualization tab.
	`_update_analysis_tab`	Provides a statistical summary of the optimization results in the analysis tab.
	`_setup_error_handling`	Defines a global error handler to catch and display exceptions in the GUI.
Objective Functions	`booth_function`, `matyas_function`, `rosenbrock_function`, `sphere_function`, `zakharov_function`	Implements mathematical formulas for various optimization test functions.
Error Handling	`_handle_optimization_error`	Manages and displays errors during the optimization process, optionally logging full tracebacks.
Main Function	`main()`	Entry point for the application, initializing and running the GUI, and handling startup errors.


Attributes Management
Set Fitness Function
Define Bounds
Set Dimensions
Set Population Size
Set Crossover Rate
Set Mutation Rate
Set Generations
Initialize Best Solutions List
Initialize Best Fitness Values
Initialize Avg Fitness Values
Initialize Diversity Values
Initialize Mutation Rates
Set Is Running
Initialize Optimization Results
Set Function Variable
Core Operations
Initialize Population
Evaluate Fitness
Selection
Crossover
Mutation
Calculate Diversity
Run Algorithm
GUI Operations
Setup Variables
Create Main Layout
Create Setup Tab
Create Results Tab
Create Visualization Tab
Create Analysis Tab
Validate and Start Optimization
Run Optimization
Complete Optimization
Update Results Tab
Update Visualization Tab
Update Analysis Tab
Setup Error Handling
Handle Optimization Error
Booth Function
Matyas Function
Rosenbrock Function
SphereFunction
Zakharov Function
Validation Successful?
No
Yes
Error Occurred?
No

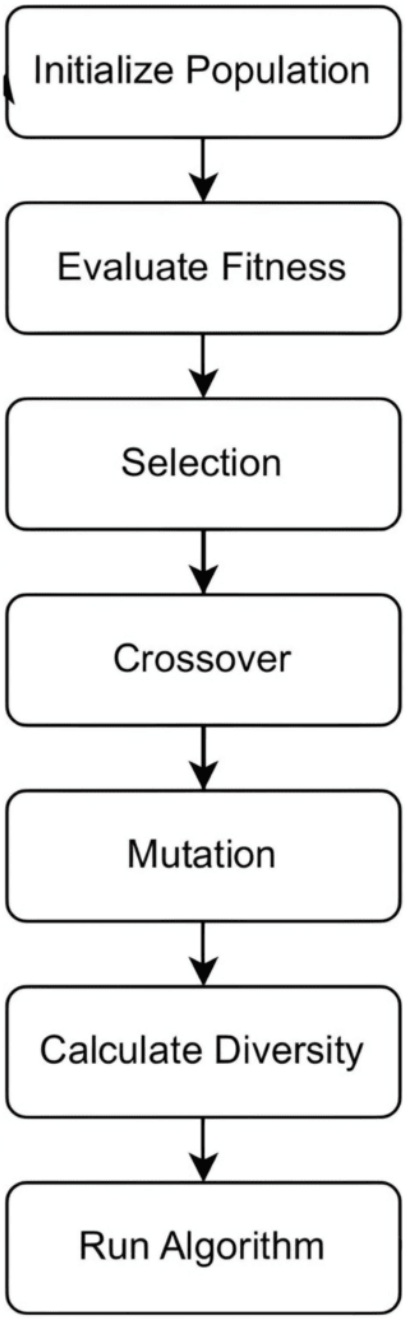
Detailed Swimlane Breakdown

Swimlane 1: Initialization and Setup


Attributes Management
Set Fitness Function
Define Bounds
Set Dimensions
Set Population Size
Set Crossover Rate
Set Mutation Rate
Set Generations
Initialize Best Solutions List
Initialize Best Fitness Values
Initialize Avg Fitness Values
Initialize Diversity Values
Initialize Mutation Rates
Set Is Running
Initialize Optimization Results
Set Function Variable

1. **Set Fitness Function**
   - **Purpose**: Define how to evaluate solutions.
   - **Code**:
     ```
     def set_fitness_function():
         self.fitness_function = lambda x: sum(xi**2 for xi in x)
     ```
2. **Initialize Population**
   - **Purpose**: Generate initial set of solutions.
   - **Code**:
     ```
     def initialize_population():
         self.population = np.random.rand(self.population_size, self.dimensions)
     ```
3. **Setup Variables**
   - **Purpose**: Configure initial parameters.
   - **Code**:
     ```
     def setup_variables():
         self.bounds = (0, 1)
         self.dimensions = 10
         self.population_size = 100
         self.crossover_rate = 0.8
         self.mutation_rate = 0.01
         self.generations = 1000
     ```
4. **Define Bounds**
   - **Purpose**: Set limits for solution space.
   - **Code**:
     ```
     def define_bounds():
         self.bounds = [(0, 1) for _ in range(self.dimensions)]
     ```

Swimlane 2: Core Operations



1. **Evaluate Fitness**
   - **Purpose**: Assess solutions for quality.
   - **Code**:
     ```
     def evaluate_fitness(population):
         return np.array([self.fitness_function(individual) for individual in population])
     ```
2. **Selection**
   - **Purpose**: Choose best solutions for next generation.
   - **Code**:
     ```
     def selection(population, fitness):
         selected_indices = np.argsort(fitness)[:self.population_size]
         return population[selected_indices]
     ```
3. **Crossover**

- **Purpose**: Combine solutions to create offspring.
- **Code**:

```
def crossover(parents):
    offspring = np.empty(parents.shape)
    for k in range(0, parents.shape[0], 2):
        parent1_idx = k % parents.shape[0]
        parent2_idx = (k + 1) % parents.shape[0]
        crossover_point = np.random.randint(0, self.dimensions)
        offspring[k, :crossover_point] = parents[parent1_idx, :crossover_point]
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring
```

4. **Mutation**

- **Purpose**: Introduce variability in offspring.
- **Code**:

```
def mutation(offspring):
    for i in range(offspring.shape[0]):
        if np.random.rand() < self.mutation_rate:
            random_index = np.random.randint(0, self.dimensions)
            offspring[i, random_index] = np.random.rand()
    return offspring
```

5. **Calculate Diversity**

- **Purpose**: Measure variety in the population.
- **Code**:

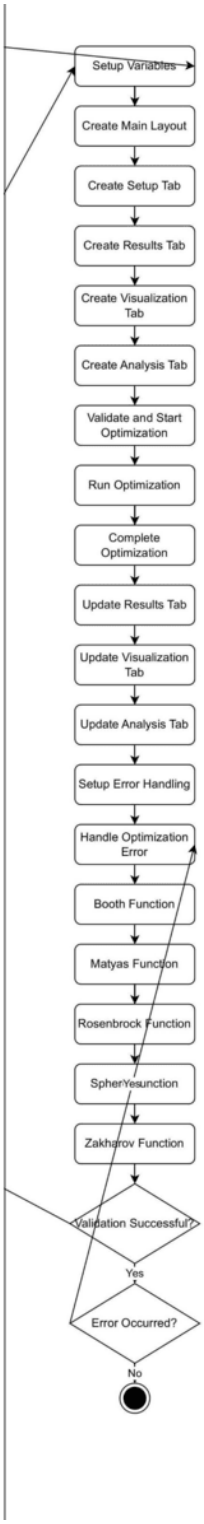
```
def calculate_diversity(population):
    return np.std(population)
```

6. **Run Algorithm**

- **Purpose**: Execute the genetic algorithm.
- **Code**:

```
def run_algorithm():
    for generation in range(self.generations):
        fitness = evaluate_fitness(self.population)
        parents = selection(self.population, fitness)
        offspring = crossover(parents)
        self.population = mutation(offspring)
        if generation % 100 == 0:
            print(f"Generation {generation}: Best Fitness = {min(fitness)}")
```

Swimlane 3: GUI Operations



1. **Create Main Layout**
    - **Purpose**: Set up main interface components.
    - **Code**:

```
def create_main_layout():
    self.main_frame = ttk.Frame(self.root)
    self.main_frame.pack()
```

2. **Create Setup Tab**
    - **Purpose**: Provide interface for parameter input.
    - **Code**:

```
def create_setup_tab():
    self.setup_tab = ttk.Frame(self.notebook)
    self.notebook.add(self.setup_tab, text='Setup')
```

3. **Create Results Tab**
   - **Purpose**: Display optimization results.
   - **Code**:
     ```
     def create_results_tab():
         self.results_tab = ttk.Frame(self.notebook)
         self.notebook.add(self.results_tab, text='Results')
     ```
4. **Create Visualization Tab**
   - **Purpose**: Show graphical representation of data.
   - **Code**:
     ```
     def create_visualization_tab():
         self.visualization_tab = ttk.Frame(self.notebook)
         self.notebook.add(self.visualization_tab, text='Visualization')
     ```
5. **Create Analysis Tab**
   - **Purpose**: Analyze detailed optimization data.
   - **Code**:
     ```
     def create_analysis_tab():
         self.analysis_tab = ttk.Frame(self.notebook)
         self.notebook.add(self.analysis_tab, text='Analysis')
     ```
6. **Validate and Start Optimization**
   - **Purpose**: Ensure everything is ready before starting.
   - **Code**:
     ```
     def validate_and_start_optimization():
         if not self.is_running:
             self.is_running = True
             run_algorithm()
         else:
             print("Optimization already running")
     ```

Error Handling and Finalization

1. **Setup Error Handling**
   - **Purpose**: Manage exceptions gracefully.
   - **Code**:
     ```
     def setup_error_handling():
         try:
             run_algorithm()
         except Exception as e:
             print(f"An error occurred: {e}")
     ```
2. **Handle Optimization Error**
   - **Purpose**: Specific error management during optimization.
   - **Code**:
     ```
     def handle_optimization_error(error):
         print(f"Optimization error: {error}")
     ```

Additional Functions

- **Booth Function**

```
def booth_function(x):
    return (x[0] + 2*x[1] - 7)**2 + (2*x[0] + x[1] - 5)**2
```

- **Matyas Function**

```
def matyas_function(x):
    return 0.26 * (x[0]**2 + x[1]**2) - 0.48 * x[0] * x[1]
```

- **Rosenbrock Function**

```
def rosenbrock_function(x):
    return sum(100.0 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x)-1))
```

- **Sphere Function**

```
def sphere_function(x):
    return sum(xi**2 for xi in x)
```

- **Zakharov Function**

```
def zakharov_function(x):
    sum1 = sum(xi**2 for xi in x)
    sum2 = sum(0.5 * i * x[i] for i in range(len(x)))
    return sum1 + sum2**2 + sum2**4
```

Validation and Error Checks

- **Validation Check**

```
def validation_successful():
    # Implement validation checks
    return True
```

- **Error Check**

```
def error_occurred():
    # Implement error checking
    return False
```

Operations Count (43 Total):

1. **Setup Operations (5)**

- Set Fitness Function
- Initialize Population
- Setup Variables
- Define Bounds
- Evaluate Fitness

2. **GUI Operations (6)**

- Create Main Layout
- Set Dimensions
- Create Setup Tab
- Create Results Tab
- Create Visualization Tab
- Create Analysis Tab

3. **Parameter Settings (4)**

- Set Population Size
- Set Crossover Rate
- Set Mutation Rate
- Set Generations

4. **Core Algorithm Operations (4)**

- Selection

- Crossover
- Mutation
- Calculate Diversity

5. **Initialization Operations (7)**

- Initialize Best Solutions List
- Initialize Best Fitness Values
- Initialize Avg Fitness Values
- Initialize Diversity Values
- Initialize Mutation Rates
- Set Is Running
- Initialize Optimization Results

6. **Optimization Operations (4)**

- Run Algorithm
- Validate and Start Optimization
- Run Optimization
- Complete Optimization

7. **Update Operations (3)**

- Update Results Tab
- Update Visualization Tab
- Update Analysis Tab

8. **Error Handling Operations (2)**

- Setup Error Handling
- Handle Optimization Error

9. **Management Categories (4)**

- Attributes Management
- Set Function Variable
- Core Operations
- GUI Operations

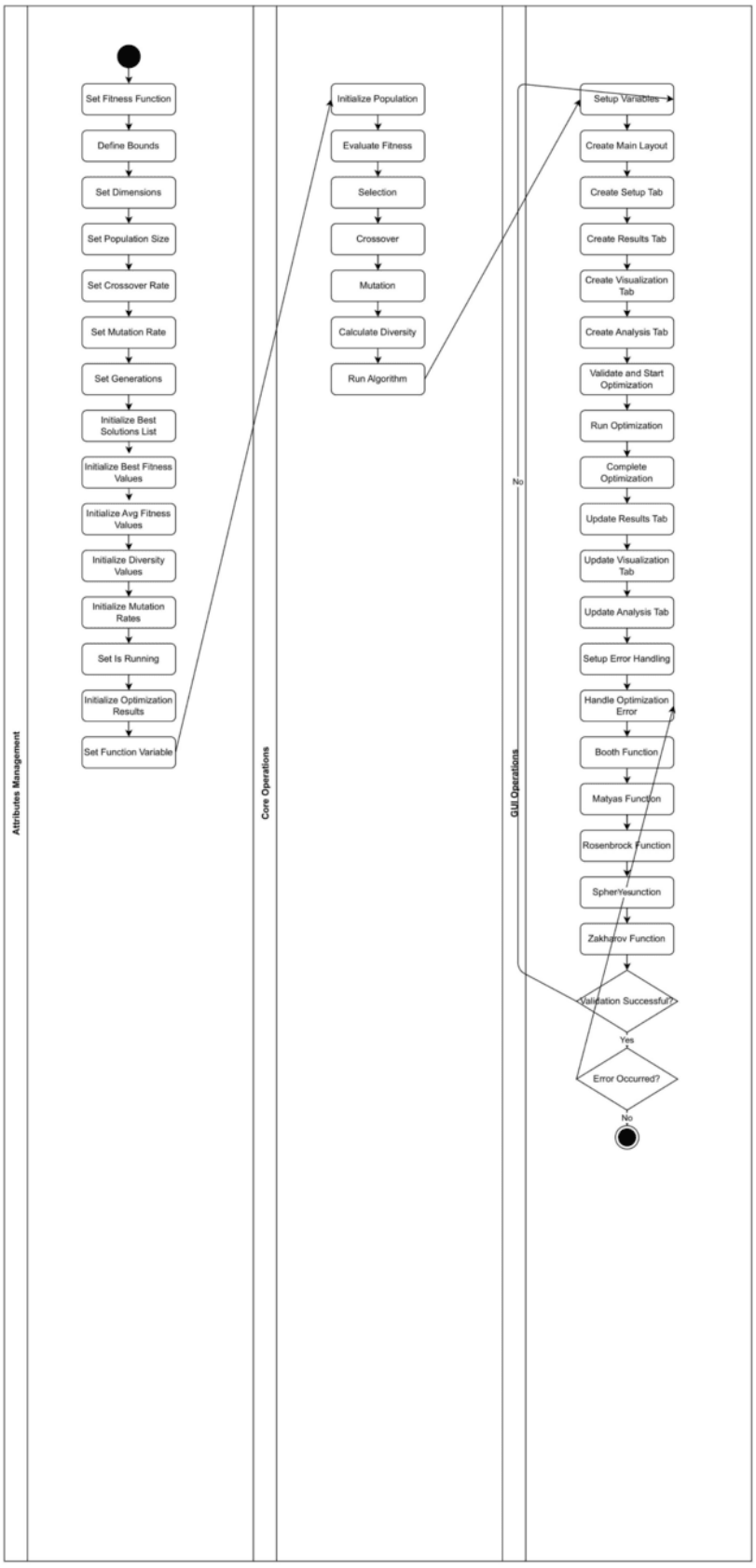
10. **Test Functions (5)**

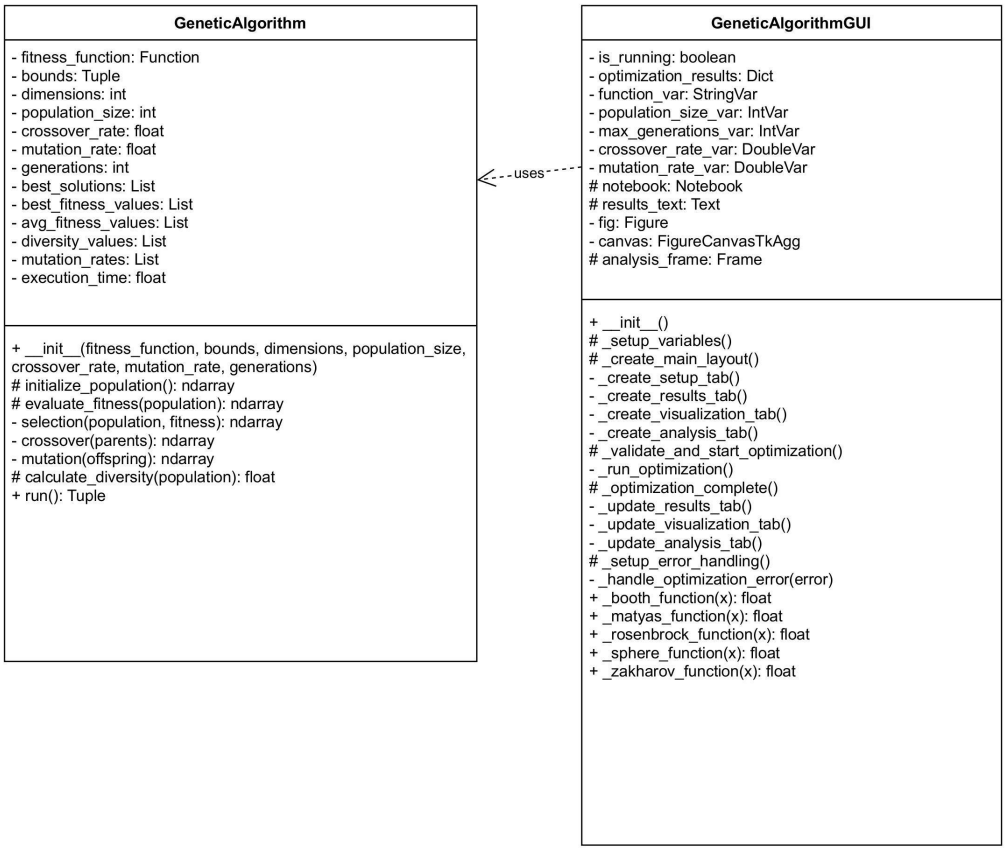
- Booth Function
- Matyas Function
- Rosenbrock Function
- Sphere Function
- Zakharov Function

Conditions Count (2 Total):

1. Validation Successful? (Yes/No)
2. Error Occurred? (Yes/No)

- Total Operations: 43
- Total Conditions: 2
- Grand Total: 45 elements


Attributes Management
Core Operations
GUI Operations
Set Fitness Function
Define Bounds
Set Dimensions
Set Population Size
Set Crossover Rate
Set Mutation Rate
Set Generations
Initialize Best Solutions List
Initialize Best Fitness Values
Initialize Avg Fitness Values
Initialize Diversity Values
Initialize Mutation Rates
Set Is Running
Initialize Optimization Results
Set Function Variable
Initialize Population
Evaluate Fitness
Selection
Crossover
Mutation
Calculate Diversity
Run Algorithm
Setup Variables
Create Main Layout
Create Setup Tab
Create Results Tab
Create Visualization Tab
Create Analysis Tab
Validate and Start Optimization
Run Optimization
Complete Optimization
Update Results Tab
Update Visualization Tab
Update Analysis Tab
Setup Error Handling
Handle Optimization Error
Booth Function
Matyas Function
Rosenbrock Function
SpherYesunction
Zakharov Function
Validation Successful?
No
Yes
Error Occurred?
No



Swimlane	Operation/ Component	Purpose	Code Snippet
Initialization and Setup			
	Set Fitness Function	Define how to evaluate solutions.	`self.fitness_function = lambda x: sum(xi**2 for xi in x)`

Swimlane	Operation/ Component	Purpose	Code Snippet
	Initialize Population	Generate initial set of solutions.	`self.population = np.random.rand(self.population_size, self.dimensions)`
	Setup Variables	Configure initial parameters.	`self.bounds = (0, 1); self.dimensions = 10; self.population_size = 100; ...`
	Define Bounds	Set limits for solution space.	`self.bounds = [(0, 1) for _ in range(self.dimensions)]`
Core Operations			
	Evaluate Fitness	Assess solutions for quality.	`return np.array([self.fitness_function(individual) for individual in population])`
	Selection	Choose best solutions for next generation.	`selected_indices = np.argsort(fitness)[:self.population_size]; return population[selected_indices]`
	Crossover	Combine solutions to create offspring.	`offspring[k, :crossover_point] = parents[parent1_idx, :crossover_point]; ...`
	Mutation	Introduce variability in offspring.	`if np.random.rand() < self.mutation_rate: ...`
	Calculate Diversity	Measure variety in the population.	`return np.std(population)`
	Run Algorithm	Execute the genetic algorithm.	`for generation in range(self.generations): ...`
GUI Operations			
	Create Main Layout	Set up main interface components.	`self.main_frame = ttk.Frame(self.root); self.main_frame.pack()`
	Create Setup Tab	Provide interface for parameter input.	`self.setup_tab = ttk.Frame(self.notebook); self.notebook.add(self.setup_tab, text='Setup')`
	Create Results Tab	Display optimization results.	`self.results_tab = ttk.Frame(self.notebook); self.notebook.add(self.results_tab, text='Results')`
	Create Visualization Tab	Show graphical representation of data.	`self.visualization_tab = ttk.Frame(self.notebook); self.notebook.add(self.visualization_tab, text='Visualization')`
	Create Analysis Tab	Analyze detailed optimization data.	`self.analysis_tab = ttk.Frame(self.notebook); self.notebook.add(self.analysis_tab, text='Analysis')`
	Validate and Start Optimization	Ensure everything is ready before starting.	`if not self.is_running: self.is_running = True; run_algorithm()`

Swimlane	Operation/ Component	Purpose	Code Snippet
Error Handling and Finalization			
	Setup Error Handling	Manage exceptions gracefully.	`try: run_algorithm() except Exception as e: print(f"An error occurred: {e}")`
	Handle Optimization Error	Specific error management during optimization.	`print(f"Optimization error: {error}")`
Additional Functions			
	Booth Function	Objective function for optimization.	`def booth_function(x): return (x[0] + 2*x[1] - 7)**2 + (2*x[0] + x[1] - 5)**2`
	Matyas Function	Objective function for optimization.	`def matyas_function(x): return 0.26 * (x[0]**2 + x[1]**2) - 0.48 * x[0] * x[1]`
	Rosenbrock Function	Objective function for optimization.	`def rosenbrock_function(x): return sum(100.0 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x)-1))`
	Sphere Function	Objective function for optimization.	`def sphere_function(x): return sum(xi**2 for xi in x)`
	Zakharov Function	Objective function for optimization.	`def zakharov_function(x): sum1 = sum(xi**2 for xi in x); sum2 = sum(0.5 * i * x[i] for i in range(len(x))); return sum1 + sum2**2 + sum2**4`

Section 1: Import Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

- **NumPy (`np`)**: This library is essential for numerical operations, particularly for handling arrays and matrices efficiently. It provides a vast array of mathematical functions and is widely used in scientific computing.
- **Matplotlib (`plt`)**: A powerful plotting library used for creating static, interactive, and animated visualizations in Python. It is highly customizable and supports a variety of graphs and plots.
- **Axes3D**: This module from `mpl_toolkits.mplot3d` is specifically for 3D plotting. It allows for the creation of three-dimensional plots.

```
import time
import json
from datetime import datetime
```

- **Time**: A built-in Python module that provides various time-related functions. In this context, it is likely used to measure execution times or to introduce delays.
- **JSON**: This module provides functions to encode and decode JSON (JavaScript Object Notation) data. JSON is a popular format for data exchange between web applications and servers.
- **Datetime**: A module that supplies classes for manipulating dates and times. It provides both simple

and complex ways to handle date and time arithmetic.

```
import os
import traceback
import threading
```

- **OS**: Provides a way of using operating system-dependent functionality like reading or writing to the file system, fetching environment variables, etc.
- **Traceback**: Used to extract, format, and print stack traces of a Python program. It is especially useful for debugging and error logging.
- **Threading**: This module is used to run multiple threads (smaller units of a process) concurrently. Useful for performing tasks in parallel and improving the efficiency of an application.

```
import random
import itertools
import tkinter as tk
```

- **Random**: A module that implements pseudo-random number generators for various distributions. It is used in this code for operations like random selection and mutations.
- **Itertools**: Provides functions creating iterators for efficient looping. It is often used to handle permutations, combinations, and other iterative tasks.
- **Tkinter (`tk`)**: Python's standard GUI (Graphical User Interface) package. It is widely used to create windows, dialogs, and various controls (buttons, text fields, etc.) in desktop applications.

```
from tkinter import ttk, messagebox, filedialog
import matplotlib
matplotlib.use('TkAgg')
```

- **TTK**: An extension of tkinter that provides themed widgets. It offers a more modern and flexible set of widgets compared to the traditional tkinter widgets.
- **MessageBox**: A module within tkinter used to create simple dialog boxes for showing messages or alerts to the user.
- **FileDialog**: Provides a standard file dialog box for opening or saving files.
- **Matplotlib with TkAgg**: Configures matplotlib to use 'TkAgg' as the backend, which is compatible with Tkinter. This setup is necessary for embedding matplotlib plots in a tkinter application.

```
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk
```

- **Figure**: Represents a top-level container for all plot elements. It is the object where plots are drawn.
- **FigureCanvasTkAgg**: Serves as the interface between the Figure and the tkinter canvas. It allows matplotlib plots to be embedded within a tkinter application.
- **NavigationToolbar2Tk**: Provides a navigation toolbar for the plot embedded in the tkinter application, enabling users to interact with the plot (e.g., zoom, pan).

Section 2: GeneticAlgorithm Class

Initialization

```
class GeneticAlgorithm:
    def __init__(self, fitness_function, bounds, dimensions,
                 population_size=100, crossover_rate=0.8, mutation_rate=0.1,
                 generations=1000):
```

- **GeneticAlgorithm**: This is the primary class encapsulating the genetic algorithm logic. It handles the initialization, evolution, and optimization processes.
- **`__init__` Method**: The constructor initializes the genetic algorithm with user-defined parameters such as the fitness function, bounds for the search space, number of dimensions, and algorithm-specific parameters like population size, crossover rate, mutation rate, and number of generations.

```
        self.fitness_function = fitness_function
        self.bounds = bounds
        self.dimensions = dimensions
```

- **Fitness Function**: A critical component of the genetic algorithm, this function evaluates how close a given solution is to achieving the set goal. Solutions with higher fitness are favored during selection.
- **Bounds**: These define the permissible range for each dimension in the solution space, ensuring that the algorithm searches within valid constraints.
- **Dimensions**: Refers to the number of variables in the solution. Each dimension represents a different variable or aspect of the problem being solved.

Population and Metrics Tracking

```
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
```

- **Population Size**: Indicates the number of solutions or individuals in each generation. A larger population may increase diversity but also requires more computational resources.
- **Crossover Rate**: The probability that two selected individuals will undergo crossover to produce offspring. Crossover helps in combining features from two parents to create potentially better offspring.
- **Mutation Rate**: The probability that an individual solution will undergo mutation. Mutations introduce variability and help the algorithm escape local optima.

```
        self.generations = generations

        # Tracking variables
        self.best_solutions = []
        self.best_fitness_values = []
```

- **Generations**: The number of iterations or cycles the algorithm will run. Each generation involves selection, crossover, and mutation operations.
- **Best Solutions**: A list that tracks the best solution found in each generation. This helps in analyzing the progress of the algorithm over time.
- **Best Fitness Values**: Stores the fitness of the best solution in each generation, providing insights into how the algorithm's performance improves over time.

```
        self.avg_fitness_values = []
        self.diversity_values = []
        self.mutation_rates = []
```

- **Average Fitness Values**: Records the average fitness of the population in each generation. It helps in understanding the overall improvement of the population.
- **Diversity Values**: Measures how varied the solutions are within the population. Higher diversity can prevent premature convergence to suboptimal solutions.
- **Mutation Rates**: Although typically constant, this list allows tracking if the mutation rate is adjusted dynamically during the run.

```
        self.execution_time = 0
```

- **Execution Time**: Tracks the total time taken to run the genetic algorithm, useful for performance evaluation and optimization.

Population Initialization

```
    def initialize_population(self):
        """Initialize a random population within the bounds"""
        lower_bound, upper_bound = self.bounds
        return lower_bound + np.random.rand(self.population_size, self.dimensions) * (upper_bound -
lower_bound)
```

- **initialize_population**: This method generates the initial set of solutions randomly within the given

bounds, ensuring a diverse starting point.
- **Lower and Upper Bounds**: These are extracted from the bounds parameter to define the range for each dimension.
- **Random Generation**: Uses `np.random.rand` to create a population of solutions, uniformly distributed within the specified bounds.

Fitness Evaluation

```
def evaluate_fitness(self, population):
    """Evaluate fitness for each individual"""
    return np.array([self.fitness_function(individual) for individual in population])
```

- **evaluate_fitness**: Computes the fitness for each solution in the population using the provided fitness function.
- **Fitness Array**: Returns an array containing the fitness values for all individuals, which is then used for selection and comparison.

Selection Process

```
def selection(self, population, fitness):
    """Tournament selection"""
    selected = np.zeros_like(population)
```

- **selection**: Implements tournament selection, a method where a subset of individuals is chosen randomly, and the best among them is selected for reproduction.
- **Tournament Selection**: This approach ensures that only the fittest individuals have a higher probability of being selected while maintaining diversity.

```
    for i in range(len(population)):
        # Randomly select 3 individuals
        tournament_indices = np.random.choice(len(population), 3, replace=False)
        tournament_fitness = fitness[tournament_indices]
        winner = tournament_indices[np.argmin(tournament_fitness)]
```

- **Individual Selection**: For each individual in the new population, three candidates are chosen randomly from the current population.
- **Fitness Comparison**: The fitness values of these candidates are compared, and the one with the best (usually lowest for minimization) fitness is selected as the winner.

```
        selected[i] = population[winner]
    return selected
```

- **Winner Assignment**: The selected winner is then added to the new population, ensuring that only the best candidates progress to the next generation.
- **Return Selected**: The function returns the new population consisting of selected individuals for further genetic operations.

Crossover Operation

```
def crossover(self, parents):
    """Uniform crossover"""
    offspring = np.copy(parents)
```

- **crossover**: Implements uniform crossover, where genes (features) from both parents are mixed to produce offspring.
- **Copy Parents**: The offspring array is initialized as a copy of the parents to ensure that changes only affect the offspring.

```
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents) and np.random.random() < self.crossover_rate:
            # Create a mask for crossover
            mask = np.random.random(self.dimensions) < 0.5
```

- **Iterate Pairs**: The algorithm iterates over pairs of parents. If a randomly generated number is less

than the crossover rate, crossover is performed.
- **Crossover Mask**: A boolean mask is created to determine which genes will be swapped between the two parents.

```
            # Swap values where mask is True
            offspring[i, mask] = parents[i+1, mask]
            offspring[i+1, mask] = parents[i, mask]
    return offspring
```

- **Gene Swapping**: Using the mask, genes are swapped between the two parents to create genetic diversity in the offspring.
- **Return Offspring**: The newly created offspring are returned for mutation and further processing.

Mutation Operation

```
    def mutation(self, offspring):
        """Mutation with adaptive rate"""
        lower_bound, upper_bound = self.bounds
```

- **mutation**: Introduces random changes to the offspring to maintain diversity and explore new areas in the solution space.
- **Bounds Access**: The method retrieves the lower and upper bounds to ensure mutations remain within valid limits.

```
        mutation_mask = np.random.random(offspring.shape) < self.mutation_rate
        mutation_values = lower_bound + np.random.random(offspring.shape) * (upper_bound -
lower_bound)
        offspring[mutation_mask] = mutation_values[mutation_mask]
        return offspring
```

- **Mutation Mask**: A boolean mask is generated to determine which genes will be mutated based on the mutation rate.
- **Random Mutations**: New random values are assigned to the selected genes, ensuring they are within the specified bounds.
- **Return Mutated Offspring**: The mutated offspring are returned, ready to be evaluated for fitness in the next generation.

Diversity Calculation

```
    def calculate_diversity(self, population):
        """Calculate population diversity"""
        return np.mean(np.std(population, axis=0))
```

- **calculate_diversity**: Measures the diversity of the population by calculating the standard deviation across each dimension.
- **Population Diversity**: High diversity indicates a wide exploration of the solution space, which can prevent premature convergence.

Main Optimization Loop

```
    def run(self):
        """Main optimization loop"""
        start_time = time.time()
```

- **run**: The core method that executes the genetic algorithm over a predefined number of generations.
- **Start Timer**: Records the start time to calculate the total execution time of the algorithm.

```
        # Initialize population
        population = self.initialize_population()

        for generation in range(self.generations):
            # Evaluate fitness
            fitness = self.evaluate_fitness(population)
```

- **Initialize Population**: Creates an initial population of solutions to begin the optimization process.
- **Generational Loop**: Iterates over the specified number of generations, performing the genetic operations in each cycle.
- **Fitness Evaluation**: Computes the fitness of each individual in the population to guide selection and reproduction.

```
        # Track best solution
        best_idx = np.argmin(fitness)
        self.best_solutions.append(population[best_idx])
        self.best_fitness_values.append(fitness[best_idx])
```

- **Best Solution Tracking**: Identifies and records the best solution in the current generation for later analysis.
- **Index of Best**: Uses `np.argmin` to find the index of the best fitness value (assuming minimization).

```
        self.avg_fitness_values.append(np.mean(fitness))

        # Calculate diversity
        diversity = self.calculate_diversity(population)
        self.diversity_values.append(diversity)
        self.mutation_rates.append(self.mutation_rate)
```

- **Average Fitness**: Records the average fitness of the population for trend analysis.
- **Diversity Measurement**: Computes and records the diversity of the population to ensure healthy exploration.
- **Mutation Rate Tracking**: Keeps track of the mutation rate, which could be adjusted if an adaptive strategy is implemented.

```
        # Selection
        parents = self.selection(population, fitness)

        # Crossover
        offspring = self.crossover(parents)

        # Mutation
        population = self.mutation(offspring)
```

- **Selection**: Chooses the fittest individuals to be parents for the next generation, ensuring quality traits are passed on.
- **Crossover**: Combines the genetic material of the parents to create new offspring, introducing diversity.
- **Mutation**: Randomly alters some offspring genes to explore new areas of the solution space.

```
    # Final evaluation
    fitness = self.evaluate_fitness(population)
    best_idx = np.argmin(fitness)

    # Record execution time
    self.execution_time = time.time() - start_time
```

- **Final Evaluation**: After all generations, a final fitness evaluation is performed to determine the best solution.
- **Execution Time**: Calculates the total time taken to execute the genetic algorithm, useful for evaluating efficiency.

```
    return self.best_solutions[-1], self.best_fitness_values[-1]
```

- **Return Best Solution**: The method returns the best solution and its fitness value from the last generation, providing the optimal result found by the algorithm.

Section 3: GeneticAlgorithmGUI Class

This section of the code is responsible for creating a graphical user interface (GUI) using Tkinter, allowing users to interact with the genetic algorithm through a visual application.

Initialization and Configuration

```
class GeneticAlgorithmGUI(tk.Tk):
    def __init__(self):
        super().__init__()

        # Window Configuration
        self.title("Genetic Algorithm Optimizer")
        self.geometry("1200x800")
        self.configure(bg='#f0f0f0')
```

- **GeneticAlgorithmGUI**: This class extends `tk.Tk`, making it a window in the Tkinter framework. It serves as the main window for the GUI application.
- **`__init__` Method**: Initializes the GUI, setting up the window's title, size, and background color.
- **Title and Geometry**: Sets the window title to "Genetic Algorithm Optimizer" and the size to 1200x800 pixels, ensuring a spacious layout for the components.

Variable Setup

```
        # Setup Variables
        self._setup_variables()

        # Create Main Layout
        self._create_main_layout()

        # Setup Error Handling
        self._setup_error_handling()
```

- **Setup Variables**: Calls a method to initialize all variables needed for the optimization and UI components. This includes default values for parameters, which the user can modify.
- **Create Main Layout**: Initializes the layout of the GUI, organizing widgets and frames for user interaction.
- **Setup Error Handling**: Prepares the application to handle and display errors gracefully, ensuring a smooth user experience even if issues arise.

Setup Variables Method

```
    def _setup_variables(self):
        """Initialize all optimization and UI variables"""
        # Optimization Functions
        self.optimization_functions = {
            "Booth Function": self._booth_function,
            "Matyas Function": self._matyas_function,
            "Rosenbrock Function": self._rosenbrock_function,
            "Sphere Function": self._sphere_function,
            "Zakharov Function": self._zakharov_function
        }
```

- **_setup_variables**: This method initializes both optimization-related variables and UI state variables.
- **Optimization Functions**: A dictionary mapping function names to their respective methods. These functions are used as fitness functions in the genetic algorithm, allowing users to select different optimization problems.

```
        # Tkinter Variables
        self.function_var = tk.StringVar(value="Sphere Function")
        self.population_size_var = tk.IntVar(value=100)
        self.max_generations_var = tk.IntVar(value=500)
```

- **Tkinter Variables**: These are special variables used by Tkinter to automatically update UI components when their values change.
  - **function_var**: Holds the currently selected optimization function.
  - **population_size_var**: Stores the population size, defaulting to 100.
  - **max_generations_var**: Stores the maximum number of generations, defaulting to 500.

```
        self.crossover_rate_var = tk.DoubleVar(value=0.8)
        self.mutation_rate_var = tk.DoubleVar(value=0.1)

        # State Variables
        self.is_running = False
        self.optimization_results = None
```

- **Crossover and Mutation Rates**: These variables store the crossover and mutation rates, allowing the user to adjust these parameters through the GUI.
- **State Variables**: Used to track the current state of the application.
  - **is_running**: A flag indicating whether the optimization process is currently running.
  - **optimization_results**: Holds the results of the optimization once it completes, initially set to None.

Creating the Main Layout

```
    def _create_main_layout(self):
        """Create the main GUI layout"""
        # Main Frame
        main_frame = ttk.Frame(self)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
```

- **_create_main_layout**: This method sets up the main structure and layout of the GUI.
- **Main Frame**: A frame widget that acts as a container for other widgets, providing a structured layout. It is packed to fill the entire window, with some padding for aesthetics.

```
        # Notebook for Tabs
        self.notebook = ttk.Notebook(main_frame)
        self.notebook.pack(fill=tk.BOTH, expand=True)

        # Create Tabs
        self._create_setup_tab()
        self._create_results_tab()
        self._create_visualization_tab()
        self._create_analysis_tab()
```

- **Notebook**: A widget that allows for tabbed navigation within the application. Each tab contains different functional aspects of the GUI.
- **Adding Tabs**: Calls methods to create individual tabs for setup, results, visualization, and analysis, each serving a unique purpose in the overall interface.

Creating the Setup Tab

```
    def _create_setup_tab(self):
        """Create the setup tab with optimization parameters"""
        setup_frame = ttk.Frame(self.notebook)
        self.notebook.add(setup_frame, text="Optimization Setup")
```

- **_create_setup_tab**: This method sets up the tab where users can define parameters for the optimization.
- **Setup Frame**: A frame within the notebook specifically for inputting optimization parameters. It is added to the notebook as the "Optimization Setup" tab.

```
        # Function Selection
        ttk.Label(setup_frame, text="Objective Function:").pack(pady=(10,5))
        function_dropdown = ttk.Combobox(
            setup_frame,
            textvariable=self.function_var,
            values=list(self.optimization_functions.keys()),
            state="readonly",
            width=30
        )
        function_dropdown.pack(pady=5)
```

- **Objective Function Label**: A label prompting users to select an objective function, providing context for the dropdown list.

- **Function Dropdown**: A combo box widget allowing users to select from the available optimization functions. It links to `function_var` to update based on user selection.

```
# Parameter Inputs
params = [
    ("Population Size:", self.population_size_var),
    ("Max Generations:", self.max_generations_var),
    ("Crossover Rate:", self.crossover_rate_var),
    ("Mutation Rate:", self.mutation_rate_var)
]

for label, var in params:
    frame = ttk.Frame(setup_frame)
    frame.pack(fill='x', padx=50, pady=5)

    ttk.Label(frame, text=label).pack(side=tk.LEFT)
    entry = ttk.Entry(frame, textvariable=var, width=20)
    entry.pack(side=tk.RIGHT)
```

- **Parameter Inputs**: Creates input fields for each optimization parameter, allowing users to specify values for population size, max generations, crossover rate, and mutation rate.
- **Loop through Parameters**: For each parameter, a frame is created with a label and an entry widget. The entry widget is linked to the corresponding Tkinter variable, updating as users input values.

```
# Start Optimization Button
start_btn = ttk.Button(
    setup_frame,
    text="Start Optimization",
    command=self._validate_and_start_optimization
)
start_btn.pack(pady=20)
```

- **Start Button**: A button that, when clicked, triggers the optimization process. It is linked to a validation and execution method to ensure proper input handling before starting.

Section 3 (continued): GeneticAlgorithmGUI Class

Validation and Starting Optimization

```
def _validate_and_start_optimization(self):
    """Validate inputs and start optimization"""
    try:
        # Input Validation
        population_size = self.population_size_var.get()
        max_generations = self.max_generations_var.get()
        crossover_rate = self.crossover_rate_var.get()
        mutation_rate = self.mutation_rate_var.get()
```

- **_validate_and_start_optimization**: This method ensures that the values entered by the user are valid before starting the optimization.
- **Variable Retrieval**: Retrieves the current values of population size, max generations, crossover rate, and mutation rate from the Tkinter variables.

```
        # Comprehensive Validation
        if population_size <= 0:
            raise ValueError("Population size must be positive")
        if max_generations <= 0:
            raise ValueError("Generations must be positive")
        if not (0 <= crossover_rate <= 1):
            raise ValueError("Crossover rate must be between 0 and 1")
        if not (0 <= mutation_rate <= 1):
            raise ValueError("Mutation rate must be between 0 and 1")
```

- **Validation Checks**: Ensures that all parameters are within acceptable ranges. For example, population size and generations must be positive, and rates must be between 0 and 1.
- **Error Handling**: Raises a `ValueError` if any parameter is invalid, prompting an error message to the

user.

```
        # Start Optimization
        threading.Thread(
            target=self._run_optimization,
            daemon=True
        ).start()

    except ValueError as e:
        messagebox.showerror("Input Error", str(e))
    except Exception as e:
        self._handle_unexpected_error(e)
```

- **Start Optimization**: If validation passes, the optimization process is started in a separate thread using `threading.Thread`. This prevents the GUI from freezing while the optimization is running.
- **Error Messages**: Uses `messagebox.showerror` to display any validation errors to the user. Unhandled exceptions are passed to a general error handler.

Running the Optimization

```
def _run_optimization(self):
    """Core optimization logic"""
    try:
        self.is_running = True
        function = self.optimization_functions[self.function_var.get()]
```

- **_run_optimization**: Executes the genetic algorithm using the parameters and function selected by the user.
- **Setting the Running State**: Sets `is_running` to `True` to indicate that the optimization is in progress.
- **Function Selection**: Retrieves the fitness function based on the user's selection from the dropdown menu.

```
        # Determine dimensions based on function
        dimensions_map = {
            "Booth Function": 2,
            "Matyas Function": 2,
            "Rosenbrock Function": 4,
            "Sphere Function": 5,
            "Zakharov Function": 5
        }
        dimensions = dimensions_map[self.function_var.get()]
```

- **Dimension Mapping**: Provides a mapping from function names to the corresponding number of dimensions. Each function has a predefined number of dimensions that the genetic algorithm must consider.
- **Retrieve Dimensions**: The dimensions for the selected function are retrieved and stored.

```
        # Bounds for each function
        bounds_map = {
            "Booth Function": (-10, 10),
            "Matyas Function": (-10, 10),
            "Rosenbrock Function": (-5, 10),
            "Sphere Function": (-5.12, 5.12),
            "Zakharov Function": (-10, 10)
        }
        bounds = bounds_map[self.function_var.get()]
```

- **Bounds Mapping**: Similar to dimensions, each function has specific bounds. This mapping ensures that the algorithm operates within the acceptable search space for each function.
- **Retrieve Bounds**: The appropriate bounds for the function are retrieved for use in the genetic algorithm.

```
            # Run Genetic Algorithm
            ga = GeneticAlgorithm(
                fitness_function=function,
                bounds=bounds,
                dimensions=dimensions,
                population_size=self.population_size_var.get(),
                generations=self.max_generations_var.get(),
                crossover_rate=self.crossover_rate_var.get(),
                mutation_rate=self.mutation_rate_var.get()
            )

            best_solution, best_fitness = ga.run()
```

- **Initialize GA**: An instance of the `GeneticAlgorithm` class is created using the selected function, bounds, dimensions, and user-defined parameters.
- **Run GA**: The `run` method of the `GeneticAlgorithm` instance is called to execute the optimization process. The best solution and fitness are returned upon completion.

```
            # Store Results
            self.optimization_results = {
                'best_solution': best_solution.tolist(),
                'best_fitness': float(best_fitness),
                'full_results': ga
            }

            # Update UI
            self.after(0, self._optimization_complete)

        except Exception as e:
            self.after(0, lambda: self._handle_optimization_error(e))
        finally:
            self.is_running = False
```

- **Store Results**: The results of the optimization, including the best solution and fitness, are stored in `optimization_results`.
- **Update UI**: Calls `_optimization_complete` using `self.after(0, ...)` to update the UI after the optimization completes.
- **Error Handling and Completion**: Catches any exceptions during execution and updates the running state flag once the process is finished.

Optimization Completion

```
    def _optimization_complete(self):
        """Handle successful optimization completion"""
        # Update Results Tab
        self._update_results_tab()

        # Update Visualization Tab
        self._update_visualization_tab()

        # Update Analysis Tab
        self._update_analysis_tab()

        # Show completion message
        messagebox.showinfo(
            "Optimization Complete",
            f"Best Fitness: {self.optimization_results['best_fitness']:.6f}"
        )
```

- **_optimization_complete**: Method called after the optimization is successfully completed.
- **Update Tabs**: Calls methods to update the results, visualization, and analysis tabs with the new data.
- **Completion Message**: Displays an informational message box with the best fitness achieved, notifying the user that the optimization process has finished.

Section 3 (continued): GeneticAlgorithmGUI Class

Results Tab Update

```
    def _update_results_tab(self):
        """Update results tab with detailed information"""
        if not self.optimization_results:
            return

        results_text = (
            f"Objective Function: {self.function_var.get()}\n\n"
            f"Best Solution: {self.optimization_results['best_solution']}\n"
            f"Best Fitness: {self.optimization_results['best_fitness']:.6f}\n\n"
            "Optimization Parameters:\n"
            f"Population Size: {self.population_size_var.get()}\n"
            f"Max Generations: {self.max_generations_var.get()}\n"
            f"Crossover Rate: {self.crossover_rate_var.get()}\n"
            f"Mutation Rate: {self.mutation_rate_var.get()}"
        )

        self.results_text.delete(1.0, tk.END)
        self.results_text.insert(tk.END, results_text)
```

- **_update_results_tab**: This method updates the results tab with detailed information about the optimization process.
- **Check Results**: It first checks if `optimization_results` is available. If not, it returns without updating.
- **Results Text**: Constructs a string with detailed information about the optimization process including the objective function, best solution, best fitness, and optimization parameters.
- **Update Text Area**: Clears the current content in the results text area and inserts the new results text.

Visualization Tab Update

```
    def _update_visualization_tab(self):
        """Create comprehensive visualization"""
        if not self.optimization_results:
            return

        # Clear previous plot
        self.fig.clear()

        # Get GA results
        ga_results = self.optimization_results['full_results']

        # Create subplots
        ax1 = self.fig.add_subplot(221)
        ax2 = self.fig.add_subplot(222)
        ax3 = self.fig.add_subplot(223)
        ax4 = self.fig.add_subplot(224)
```

- **_update_visualization_tab**: Updates the visualization tab with plots showing the progress of the optimization.
- **Check Results**: Ensures that there are results to display; otherwise, it exits.
- **Clear Previous Plots**: Clears any existing plots in the figure to prepare for new visualizations.
- **Access GA Results**: Retrieves the full results from the genetic algorithm run to plot various metrics.
- **Subplots Setup**: Creates a 2x2 grid of subplots for displaying different metrics like best fitness, average fitness, population diversity, and mutation rates.

```
        # Plot Best Fitness
        ax1.plot(ga_results.best_fitness_values)
        ax1.set_title('Best Fitness Progression')
        ax1.set_xlabel('Generation')
        ax1.set_ylabel('Best Fitness')

        # Plot Average Fitness
        ax2.plot(ga_results.avg_fitness_values)
        ax2.set_title('Average Fitness Progression')
        ax2.set_xlabel('Generation')
        ax2.set_ylabel('Average Fitness')
```

- **Best Fitness Plot**: Plots the best fitness values over generations, showing how the best solution improves as the algorithm progresses.
- **Average Fitness Plot**: Displays the average fitness of the population over time, providing insight into the overall improvement of the population.

```
    # Plot Diversity
    ax3.plot(ga_results.diversity_values)
    ax3.set_title('Population Diversity')
    ax3.set_xlabel('Generation')
    ax3.set_ylabel('Diversity')

    # Plot Mutation Rates
    ax4.plot(ga_results.mutation_rates)
    ax4.set_title('Mutation Rates')
    ax4.set_xlabel('Generation')
    ax4.set_ylabel('Mutation Rate')

    # Adjust layout and redraw
    self.fig.tight_layout()
    self.canvas.draw()
```

- **Diversity Plot**: Shows the diversity levels of the population, helping to understand how varied the solutions are throughout the optimization.
- **Mutation Rates Plot**: Displays the mutation rates over generations, useful if mutation rates are adjusted dynamically.
- **Layout Adjustment**: Ensures the subplots are neatly arranged without overlapping, and redraws the canvas to update the display.

Analysis Tab Update

```
def _update_analysis_tab(self):
    """Provide statistical analysis of optimization results"""
    if not self.optimization_results:
        return

    # Create analysis frame
    for widget in self.analysis_frame.winfo_children():
        widget.destroy()

    # Detailed Analysis
    ga_results = self.optimization_results['full_results']

    # Create labels for detailed analysis
    analysis_data = [
        f"Total Generations: {len(ga_results.best_fitness_values)}",
        f"Best Fitness: {self.optimization_results['best_fitness']:.6f}",
        f"Final Population Diversity: {ga_results.diversity_values[-1]:.6f}",
        f"Final Mutation Rate: {ga_results.mutation_rates[-1]:.6f}",
        f"Execution Time: {ga_results.execution_time:.2f} seconds"
    ]

    for i, data in enumerate(analysis_data):
        label = ttk.Label(self.analysis_frame, text=data, font=('Arial', 10))
        label.pack(pady=5, anchor='w')
```

- **_update_analysis_tab**: Updates the analysis tab with a detailed statistical summary of the optimization results.
- **Check Results**: Ensures that there are results to display; otherwise, it exits.
- **Clear Previous Content**: Removes any existing widgets in the analysis frame to avoid clutter and prepare for new data.
- **Retrieve GA Results**: Accesses the full results from the genetic algorithm to extract detailed metrics.
- **Analysis Data**: Compiles a list of key metrics including total generations, best fitness, final population diversity, final mutation rate, and execution time.
- **Display Data**: Iterates through the analysis data, creating labels for each piece of information and packing them into the analysis frame.

Section 4: Error Handling and Utility Functions

Error Handling Setup

```
def _setup_error_handling(self):
    """Setup error handling mechanisms"""
    def error_handler(exc_type, exc_value, exc_traceback):
        error_msg = ''.join(traceback.format_exception(exc_type, exc_value, exc_traceback))
        messagebox.showerror("Unexpected Error", error_msg)

    # Set global exception handler
    tk.Tk.report_callback_exception = error_handler
```

- **_setup_error_handling**: Configures the application to handle errors gracefully.
- **Error Handler Function**: Defines a function that formats and displays error messages using `traceback` to provide detailed stack traces in a message box.
- **Global Exception Handler**: Assigns the custom error handler to `tk.Tk.report_callback_exception`, ensuring that all unhandled exceptions in the GUI are captured and displayed.

Optimization Error Handling

```
def _handle_optimization_error(self, error):
    """Handle errors during optimization"""
    error_msg = str(error)
    traceback_msg = traceback.format_exc()

    messagebox.showerror(
        "Optimization Error",
        f"An error occurred during optimization:\n{error_msg}"
    )

    # Optional: Log full traceback
    print(traceback_msg)
```

- **_handle_optimization_error**: Specifically handles errors that occur during the optimization process.
- **Error Message**: Converts the error to a string and formats the traceback for display.
- **Display Error**: Uses a message box to alert the user to the error that occurred during optimization, providing the error message.
- **Logging**: Optionally prints the full traceback to the console for debugging purposes, helping developers identify and fix issues.

Section 5: Optimization Functions

These functions define the mathematical problems that the genetic algorithm will attempt to optimize.

```
def _booth_function(self, x):
    """Booth function for optimization"""
    return (x[0] + 2*x[1] - 7)**2 + (2*x[0] + x[1] - 5)**2
```

- **Booth Function**: A well-known test function for optimization, defined for two variables. It has a global minimum at the point (1, 3).
- **Mathematical Expression**: The function is expressed as a sum of squared terms, which is typical for optimization problems.

```
def _matyas_function(self, x):
    """Matyas function for optimization"""
    return 0.26 * (x[0]**2 + x[1]**2) - 0.48 * x[0] * x[1]
```

- **Matyas Function**: Another test function with a bowl-shaped surface. It is defined for two variables and has a global minimum at the origin (0, 0).
- **Expression**: Combines quadratic terms with a product term, creating a surface suitable for optimization testing.

```
    def _rosenbrock_function(self, x):
        """Rosenbrock function for optimization"""
        return sum(100.0 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x)-1))
```

- **Rosenbrock Function**: Known as the "banana function" due to its curved valley, used as a benchmark in optimization. It is defined for multiple dimensions.
- **Expression**: Involves quadratic terms and has a narrow, curved valley leading to the global minimum, making it challenging for optimization algorithms.

```
    def _sphere_function(self, x):
        """Sphere function for optimization"""
        return np.sum(x**2)
```

- **Sphere Function**: A simple, unimodal test function where the global minimum is at the origin. It serves as a basic test for optimization algorithms.
- **Expression**: A straightforward sum of squared terms, representing a spherical surface in multi-dimensional space.

```
    def _zakharov_function(self, x):
        """Zakharov function for optimization"""
        sum1 = np.sum(x**2)
        sum2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x)
        return sum1 + sum2**2 + sum2**4
```

- **Zakharov Function**: A complex test function with more challenging features such as a flat surface near the origin and steep sides.
- **Expression**: Combines quadratic terms with higher-order polynomial terms, offering a diverse landscape for testing optimization strategies.

Section 6: Main Application Entry Point

```
def main():
    """Main application entry point"""
    try:
        app = GeneticAlgorithmGUI()
        app.mainloop()
    except Exception as e:
        print(f"Application startup error: {e}")
        traceback.print_exc()
```

- **main Function**: Serves as the entry point for the application, responsible for starting the GUI.
- **Application Initialization**: Creates an instance of `GeneticAlgorithmGUI` and starts the main event loop with `app.mainloop()`, which keeps the window open and responsive.
- **Exception Handling**: Catches any exceptions during startup, printing an error message and traceback to assist in debugging.

```
if __name__ == "__main__":
    main()
```

- **Execution Guard**: Ensures that the `main` function is only called when the script is run directly, not when it is imported as a module.

Full Code

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
import json
from datetime import datetime
import os
import traceback
import threading
import random
import itertools

import tkinter as tk
from tkinter import ttk, messagebox, filedialog
import matplotlib
matplotlib.use('TkAgg')
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk

class GeneticAlgorithm:
    def __init__(self, fitness_function, bounds, dimensions,
                 population_size=100, crossover_rate=0.8, mutation_rate=0.1,
                 generations=1000):
        self.fitness_function = fitness_function
        self.bounds = bounds
        self.dimensions = dimensions
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.generations = generations

        # Tracking variables
        self.best_solutions = []
        self.best_fitness_values = []
        self.avg_fitness_values = []
        self.diversity_values = []
        self.mutation_rates = []
        self.execution_time = 0

    def initialize_population(self):
        """Initialize a random population within the bounds"""
        lower_bound, upper_bound = self.bounds
        return lower_bound + np.random.rand(self.population_size, self.dimensions) * (upper_bound -
lower_bound)

    def evaluate_fitness(self, population):
        """Evaluate fitness for each individual"""
        return np.array([self.fitness_function(individual) for individual in population])

    def selection(self, population, fitness):
        """Tournament selection"""
        selected = np.zeros_like(population)
        for i in range(len(population)):
            # Randomly select 3 individuals
            tournament_indices = np.random.choice(len(population), 3, replace=False)
            tournament_fitness = fitness[tournament_indices]
            winner = tournament_indices[np.argmin(tournament_fitness)]
            selected[i] = population[winner]
        return selected

    def crossover(self, parents):
        """Uniform crossover"""
        offspring = np.copy(parents)
        for i in range(0, len(parents), 2):
            if i + 1 < len(parents) and np.random.random() < self.crossover_rate:
                # Create a mask for crossover
                mask = np.random.random(self.dimensions) < 0.5
                # Swap values where mask is True
                offspring[i, mask] = parents[i+1, mask]
                offspring[i+1, mask] = parents[i, mask]
        return offspring
```

```
    def mutation(self, offspring):
        """Mutation with adaptive rate"""
        lower_bound, upper_bound = self.bounds
        mutation_mask = np.random.random(offspring.shape) < self.mutation_rate
        mutation_values = lower_bound + np.random.random(offspring.shape) * (upper_bound -
lower_bound)
        offspring[mutation_mask] = mutation_values[mutation_mask]
        return offspring

    def calculate_diversity(self, population):
        """Calculate population diversity"""
        return np.mean(np.std(population, axis=0))

    def run(self):
        """Main optimization loop"""
        start_time = time.time()

        # Initialize population
        population = self.initialize_population()

        for generation in range(self.generations):
            # Evaluate fitness
            fitness = self.evaluate_fitness(population)

            # Track best solution
            best_idx = np.argmin(fitness)
            self.best_solutions.append(population[best_idx])
            self.best_fitness_values.append(fitness[best_idx])
            self.avg_fitness_values.append(np.mean(fitness))

            # Calculate diversity
            diversity = self.calculate_diversity(population)
            self.diversity_values.append(diversity)
            self.mutation_rates.append(self.mutation_rate)

            # Selection
            parents = self.selection(population, fitness)

            # Crossover
            offspring = self.crossover(parents)

            # Mutation
            population = self.mutation(offspring)

        # Final evaluation
        fitness = self.evaluate_fitness(population)
        best_idx = np.argmin(fitness)

        # Record execution time
        self.execution_time = time.time() - start_time

        return self.best_solutions[-1], self.best_fitness_values[-1]

class GeneticAlgorithmGUI(tk.Tk):
    def __init__(self):
        super().__init__()

        # Window Configuration
        self.title("Genetic Algorithm Optimizer")
        self.geometry("1200x800")
        self.configure(bg='#f0f0f0')

        # Setup Variables
        self._setup_variables()

        # Create Main Layout
        self._create_main_layout()

        # Setup Error Handling
        self._setup_error_handling()

    def _setup_variables(self):
        """Initialize all optimization and UI variables"""
```

```
        # Optimization Functions
        self.optimization_functions = {
            "Booth Function": self._booth_function,
            "Matyas Function": self._matyas_function,
            "Rosenbrock Function": self._rosenbrock_function,
            "Sphere Function": self._sphere_function,
            "Zakharov Function": self._zakharov_function
        }

        # Tkinter Variables
        self.function_var = tk.StringVar(value="Sphere Function")
        self.population_size_var = tk.IntVar(value=100)
        self.max_generations_var = tk.IntVar(value=500)
        self.crossover_rate_var = tk.DoubleVar(value=0.8)
        self.mutation_rate_var = tk.DoubleVar(value=0.1)

        # State Variables
        self.is_running = False
        self.optimization_results = None

    def _create_main_layout(self):
        """Create the main GUI layout"""
        # Main Frame
        main_frame = ttk.Frame(self)
        main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # Notebook for Tabs
        self.notebook = ttk.Notebook(main_frame)
        self.notebook.pack(fill=tk.BOTH, expand=True)

        # Create Tabs
        self._create_setup_tab()
        self._create_results_tab()
        self._create_visualization_tab()
        self._create_analysis_tab()

    def _create_setup_tab(self):
        """Create the setup tab with optimization parameters"""
        setup_frame = ttk.Frame(self.notebook)
        self.notebook.add(setup_frame, text="Optimization Setup")

        # Function Selection
        ttk.Label(setup_frame, text="Objective Function:").pack(pady=(10,5))
        function_dropdown = ttk.Combobox(
            setup_frame,
            textvariable=self.function_var,
            values=list(self.optimization_functions.keys()),
            state="readonly",
            width=30
        )
        function_dropdown.pack(pady=5)

        # Parameter Inputs
        params = [
            ("Population Size:", self.population_size_var),
            ("Max Generations:", self.max_generations_var),
            ("Crossover Rate:", self.crossover_rate_var),
            ("Mutation Rate:", self.mutation_rate_var)
        ]

        for label, var in params:
            frame = ttk.Frame(setup_frame)
            frame.pack(fill='x', padx=50, pady=5)

            ttk.Label(frame, text=label).pack(side=tk.LEFT)
            entry = ttk.Entry(frame, textvariable=var, width=20)
            entry.pack(side=tk.RIGHT)

        # Start Optimization Button
        start_btn = ttk.Button(
            setup_frame,
            text="Start Optimization",
            command=self._validate_and_start_optimization
```

```
        )
        start_btn.pack(pady=20)

    def _validate_and_start_optimization(self):
        """Validate inputs and start optimization"""
        try:
            # Input Validation
            population_size = self.population_size_var.get()
            max_generations = self.max_generations_var.get()
            crossover_rate = self.crossover_rate_var.get()
            mutation_rate = self.mutation_rate_var.get()

            # Comprehensive Validation
            if population_size <= 0:
                raise ValueError("Population size must be positive")
            if max_generations <= 0:
                raise ValueError("Generations must be positive")
            if not (0 <= crossover_rate <= 1):
                raise ValueError("Crossover rate must be between 0 and 1")
            if not (0 <= mutation_rate <= 1):
                raise ValueError("Mutation rate must be between 0 and 1")

            # Start Optimization
            threading.Thread(
                target=self._run_optimization,
                daemon=True
            ).start()

        except ValueError as e:
            messagebox.showerror("Input Error", str(e))
        except Exception as e:
            self._handle_unexpected_error(e)

    def _run_optimization(self):
        """Core optimization logic"""
        try:
            self.is_running = True
            function = self.optimization_functions[self.function_var.get()]

            # Determine dimensions based on function
            dimensions_map = {
                "Booth Function": 2,
                "Matyas Function": 2,
                "Rosenbrock Function": 4,
                "Sphere Function": 5,
                "Zakharov Function": 5
            }
            dimensions = dimensions_map[self.function_var.get()]

            # Bounds for each function
            bounds_map = {
                "Booth Function": (-10, 10),
                "Matyas Function": (-10, 10),
                "Rosenbrock Function": (-5, 10),
                "Sphere Function": (-5.12, 5.12),
                "Zakharov Function": (-10, 10)
            }
            bounds = bounds_map[self.function_var.get()]

            # Run Genetic Algorithm
            ga = GeneticAlgorithm(
                fitness_function=function,
                bounds=bounds,
                dimensions=dimensions,
                population_size=self.population_size_var.get(),
                generations=self.max_generations_var.get(),
                crossover_rate=self.crossover_rate_var.get(),
                mutation_rate=self.mutation_rate_var.get()
            )

            best_solution, best_fitness = ga.run()

            # Store Results
```

```
            self.optimization_results = {
                'best_solution': best_solution.tolist(),
                'best_fitness': float(best_fitness),
                'full_results': ga
            }

            # Update UI
            self.after(0, self._optimization_complete)

        except Exception as e:
            self.after(0, lambda: self._handle_optimization_error(e))
        finally:
            self.is_running = False

    def _optimization_complete(self):
        """Handle successful optimization completion"""
        # Update Results Tab
        self._update_results_tab()

        # Update Visualization Tab
        self._update_visualization_tab()

        # Update Analysis Tab
        self._update_analysis_tab()

        # Show completion message
        messagebox.showinfo(
            "Optimization Complete",
            f"Best Fitness: {self.optimization_results['best_fitness']:.6f}"
        )

    def _update_results_tab(self):
        """Update results tab with detailed information"""
        if not self.optimization_results:
            return

        results_text = (
            f"Objective Function: {self.function_var.get()}\n\n"
            f"Best Solution: {self.optimization_results['best_solution']}\n"
            f"Best Fitness: {self.optimization_results['best_fitness']:.6f}\n\n"
            "Optimization Parameters:\n"
            f"Population Size: {self.population_size_var.get()}\n"
            f"Max Generations: {self.max_generations_var.get()}\n"
            f"Crossover Rate: {self.crossover_rate_var.get()}\n"
            f"Mutation Rate: {self.mutation_rate_var.get()}"
        )

        self.results_text.delete(1.0, tk.END)
        self.results_text.insert(tk.END, results_text)

    def _update_visualization_tab(self):
        """Create comprehensive visualization"""
        if not self.optimization_results:
            return

        # Clear previous plot
        self.fig.clear()

        # Get GA results
        ga_results = self.optimization_results['full_results']

        # Create subplots
        ax1 = self.fig.add_subplot(221)
        ax2 = self.fig.add_subplot(222)
        ax3 = self.fig.add_subplot(223)
        ax4 = self.fig.add_subplot(224)

        # Plot Best Fitness
        ax1.plot(ga_results.best_fitness_values)
        ax1.set_title('Best Fitness Progression')
        ax1.set_xlabel('Generation')
        ax1.set_ylabel('Best Fitness')
```

```
        # Plot Average Fitness
        ax2.plot(ga_results.avg_fitness_values)
        ax2.set_title('Average Fitness Progression')
        ax2.set_xlabel('Generation')
        ax2.set_ylabel('Average Fitness')

        # Plot Diversity
        ax3.plot(ga_results.diversity_values)
        ax3.set_title('Population Diversity')
        ax3.set_xlabel('Generation')
        ax3.set_ylabel('Diversity')

        # Plot Mutation Rates
        ax4.plot(ga_results.mutation_rates)
        ax4.set_title('Mutation Rates')
        ax4.set_xlabel('Generation')
        ax4.set_ylabel('Mutation Rate')

        # Adjust layout and redraw
        self.fig.tight_layout()
        self.canvas.draw()

    def _update_analysis_tab(self):
        """Provide statistical analysis of optimization results"""
        if not self.optimization_results:
            return

        # Create analysis frame
        for widget in self.analysis_frame.winfo_children():
            widget.destroy()

        # Detailed Analysis
        ga_results = self.optimization_results['full_results']

        # Create labels for detailed analysis
        analysis_data = [
            f"Total Generations: {len(ga_results.best_fitness_values)}",
            f"Best Fitness: {self.optimization_results['best_fitness']:.6f}",
            f"Final Population Diversity: {ga_results.diversity_values[-1]:.6f}",
            f"Final Mutation Rate: {ga_results.mutation_rates[-1]:.6f}",
            f"Execution Time: {ga_results.execution_time:.2f} seconds"
        ]

        for i, data in enumerate(analysis_data):
            label = ttk.Label(self.analysis_frame, text=data, font=('Arial', 10))
            label.pack(pady=5, anchor='w')


    def _create_results_tab(self):
        """Create results tab with text area"""
        results_frame = ttk.Frame(self.notebook)
        self.notebook.add(results_frame, text="Results")

        # Scrollable text area for results
        self.results_text = tk.Text(
            results_frame,
            wrap=tk.WORD,
            width=80,
            height=30,
            font=('Courier', 10)
        )
        self.results_text.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

        # Add scrollbar
        scrollbar = ttk.Scrollbar(results_frame, command=self.results_text.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        self.results_text.configure(yscrollcommand=scrollbar.set)

    def _create_visualization_tab(self):
        """Create visualization tab with matplotlib plot"""
        visualization_frame = ttk.Frame(self.notebook)
        self.notebook.add(visualization_frame, text="Visualization")
```

```
        # Create matplotlib figure
        self.fig = Figure(figsize=(10, 8), dpi=100)

        # Create canvas for matplotlib figure
        self.canvas = FigureCanvasTkAgg(self.fig, master=visualization_frame)
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        # Add navigation toolbar
        toolbar = NavigationToolbar2Tk(self.canvas, visualization_frame)
        toolbar.update()
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def _create_analysis_tab(self):
        """Create analysis tab"""
        self.analysis_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.analysis_frame, text="Analysis")

    def _setup_error_handling(self):
        """Setup error handling mechanisms"""
        def error_handler(exc_type, exc_value, exc_traceback):
            error_msg = ''.join(traceback.format_exception(exc_type, exc_value, exc_traceback))
            messagebox.showerror("Unexpected Error", error_msg)

        # Set global exception handler
        tk.Tk.report_callback_exception = error_handler

    def _handle_optimization_error(self, error):
        """Handle errors during optimization"""
        error_msg = str(error)
        traceback_msg = traceback.format_exc()

        messagebox.showerror(
            "Optimization Error",
            f"An error occurred during optimization:\n{error_msg}"
        )

        # Optional: Log full traceback
        print(traceback_msg)

    def _booth_function(self, x):
        """Booth function for optimization"""
        return (x[0] + 2*x[1] - 7)**2 + (2*x[0] + x[1] - 5)**2

    def _matyas_function(self, x):
        """Matyas function for optimization"""
        return 0.26 * (x[0]**2 + x[1]**2) - 0.48 * x[0] * x[1]

    def _rosenbrock_function(self, x):
        """Rosenbrock function for optimization"""
        return sum(100.0 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x)-1))

    def _sphere_function(self, x):
        """Sphere function for optimization"""
        return np.sum(x**2)

    def _zakharov_function(self, x):
        """Zakharov function for optimization"""
        sum1 = np.sum(x**2)
        sum2 = np.sum(0.5 * np.arange(1, len(x) + 1) * x)
        return sum1 + sum2**2 + sum2**4

def main():
    """Main application entry point"""
    try:
        app = GeneticAlgorithmGUI()
        app.mainloop()
    except Exception as e:
        print(f"Application startup error: {e}")
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

Pseudocode

Import Libraries

1. **Import essential libraries for numerical and plotting operations**:
    - Import `numpy` for numerical computations.
    - Import `matplotlib.pyplot` for plotting capabilities.
    - Import `mpl_toolkits.mplot3d.Axes3D` for 3D plotting.
2. **Import libraries for system and time management**:
    - Import `time` for tracking execution duration.
    - Import `json` for handling JSON data structures.
    - Import `datetime` for date and time operations.
    - Import `os` for interacting with the operating system.
    - Import `traceback` for error tracking and debugging.
    - Import `threading` for concurrent execution.
3. **Import libraries for random operations and GUI development**:
    - Import `random` for random number generation.
    - Import `itertools` for efficient looping.
    - Import `tkinter` and `tkinter.ttk` for GUI components.
    - Import `tkinter.messagebox` and `tkinter.filedialog` for dialogs.
    - Setup `matplotlib` with `TkAgg` backend for GUI compatibility.
4. **Import matplotlib components for embedding plots**:
    - Import `matplotlib.figure.Figure` to create plot containers.
    - Import `matplotlib.backends.backend_tkagg.FigureCanvasTkAgg` for integrating plots into Tkinter.
    - Import `NavigationToolbar2Tk` for adding interactive toolbar to plots.

Genetic Algorithm Class

1. **Define `GeneticAlgorithm` class**:
    - **Initialize attributes**:
        - Assign fitness function, bounds, dimensions, population size, crossover rate, mutation rate, and number of generations.
        - Initialize lists for tracking best solutions, fitness values, average fitness, diversity, mutation rates, and execution time.
2. **Define `initialize_population` method**:
    - Generate a random initial population bounded by specified limits.
3. **Define `evaluate_fitness` method**:
    - Compute and return fitness values for each individual in the population using the fitness function.
4. **Define `selection` method**:
    - Implement tournament selection to choose parents.
    - For each individual, select three candidates, evaluate their fitness, and choose the best one to advance.
5. **Define `crossover` method**:
    - Implement uniform crossover between pairs of parents based on a crossover rate.
    - Generate a binary mask to decide which genes to swap.

6. **Define `mutation` method**:
    - Mutate offspring genes randomly based on mutation rate.
    - Ensure mutations respect bounds of the search space.
7. **Define `calculate_diversity` method**:
    - Calculate and return the diversity of the population based on standard deviation across dimensions.
8. **Define `run` method**:
    - Track start time.
    - Initialize population.
    - For each generation, evaluate fitness, track best solutions and fitness, calculate diversity, and apply selection, crossover, and mutation.
    - After all generations, record execution time and return the best solution and fitness.

Genetic Algorithm GUI Class

1. **Define `GeneticAlgorithmGUI` class extending Tkinter**:
    - **Initialize GUI**: Set window title, size, and background color.
    - **Setup variables**: Define Tkinter variables for function selection and optimization parameters.
    - **Create main layout**: Setup notebook with tabs for setup, results, visualization, and analysis.
    - **Setup error handling**: Define a global error handler for exceptions in the GUI.
2. **Define `_setup_variables` method**:
    - Initialize optimization functions dictionary linking function names to methods.
    - Initialize Tkinter variables for selected function, population size, generations, crossover rate, and mutation rate.
    - Initialize state variables for tracking optimization status and results.
3. **Define `_create_main_layout` method**:
    - Create a main frame and a notebook for organizing tabs.
4. **Define `_create_setup_tab` method**:
    - Add setup tab to notebook.
    - Create dropdown for function selection.
    - Create input fields for optimization parameters.
    - Add a button to start optimization.
5. **Define `_validate_and_start_optimization` method**:
    - Validate user inputs for optimization parameters.
    - If valid, start optimization in a new thread.
    - Display error messages for invalid inputs.
6. **Define `_run_optimization` method**:
    - Set running state to true.
    - Retrieve selected function, dimensions, and bounds.
    - Initialize and run the genetic algorithm.
    - Store results and update the UI upon completion.
    - Handle any exceptions during execution.
7. **Define `_optimization_complete` method**:
    - Update results, visualization, and analysis tabs.
    - Display completion message with best fitness.

8. **Define `_update_results_tab` method**:
    - Update results tab with detailed information about the optimization process.
9. **Define `_update_visualization_tab` method**:
    - Clear previous plots and create new visualizations for best fitness, average fitness, diversity, and mutation rates.
10. **Define `_update_analysis_tab` method**:
    - Provide a statistical summary of the optimization results in the analysis tab.
11. **Define `_setup_error_handling` method**:
    - Setup a global exception handler to display error messages in a message box.
12. **Define `_handle_optimization_error` method**:
    - Handle and display errors that occur during the optimization process.

Optimization Functions

1. **Define benchmark functions**:
    - **Booth Function**: Return a calculated value based on Booth's mathematical formula.
    - **Matyas Function**: Return value using Matyas' formula.
    - **Rosenbrock Function**: Calculate and return value using Rosenbrock's formula.
    - **Sphere Function**: Return sum of squares of the input vector.
    - **Zakharov Function**: Compute and return a value using Zakharov's complex formula.

Main Application Entry Point

1. **Define `main` function**:
    - Initialize and start the GUI application.
    - Include exception handling to catch and print startup errors.
2. **Execution Guard**:
    - Ensure `main` is called only when the script is executed directly.

Section	Description
Import Libraries	
Numerical & Plotting	- Import `numpy` for numerical computations. - Import `matplotlib.pyplot` for plotting. - Import `mpl_toolkits.mplot3d.Axes3D` for 3D plotting.
System & Time Management	- Import `time` for execution duration. - Import `json` for JSON handling. - Import `datetime` for date/time operations. - Import `os` for OS interactions. - Import `traceback` for error tracking. - Import `threading` for concurrent execution.
Random & GUI Development	- Import `random` for random numbers. - Import `itertools` for looping. - Import `tkinter` and `tkinter.ttk` for GUI components. - Import `tkinter.messagebox` and `tkinter.filedialog` for dialogs. - Setup `matplotlib` with `TkAgg` backend.

Section	Description
Matplotlib Components	- Import `matplotlib.figure.Figure` to create plot containers. - Import `FigureCanvasTkAgg` to embed plots in Tkinter. - Import `NavigationToolbar2Tk` for plot interaction toolbar.
Genetic Algorithm Class	
Initialization	- Define `GeneticAlgorithm` class. - Initialize fitness function, bounds, dimensions, population size, crossover rate, mutation rate, generations. - Initialize lists for tracking best solutions, fitness values, average fitness, diversity, mutation rates, execution time.
Initialize Population	- Define `initialize_population` method. - Generate random initial population within bounds.
Evaluate Fitness	- Define `evaluate_fitness` method. - Compute and return fitness for each individual.
Selection Process	- Define `selection` method. - Implement tournament selection for parent selection.
Crossover Operation	- Define `crossover` method. - Implement uniform crossover based on a crossover rate.
Mutation Operation	- Define `mutation` method. - Introduce mutations based on mutation rate.
Diversity Calculation	- Define `calculate_diversity` method. - Calculate and return diversity based on standard deviation.
Run Optimization	- Define `run` method. - Execute genetic algorithm for specified generations. - Return best solution and fitness.
Genetic Algorithm GUI Class	
Initialization & Config	- Define `GeneticAlgorithmGUI` class extending Tkinter. - Set window title, size, background color. - Setup variables for function selection and optimization parameters.
Create Main Layout	- Define `_create_main_layout` method. - Setup notebook with setup, results, visualization, analysis tabs.
Setup Tab	- Define `_create_setup_tab` method. - Add function selection dropdown. - Add input fields for optimization parameters. - Add start optimization button.
Validate & Start Optimization	- Define `_validate_and_start_optimization` method. - Validate inputs and start optimization in a thread. - Display error messages for invalid inputs.
Run Optimization	- Define `_run_optimization` method. - Initialize and run genetic algorithm. - Store results and update UI upon completion.
Optimization Completion	- Define `_optimization_complete` method. - Update results, visualization, analysis tabs. - Display completion message.
Update Tabs	- Define `_update_results_tab`, `_update_visualization_tab`, `_update_analysis_tab` methods. - Update GUI with detailed information and plots.

Section	Description
Error Handling	- Define `_setup_error_handling` method. - Set global exception handler. - Define `_handle_optimization_error` for optimization errors.
Optimization Functions	
Benchmark Functions	- Define `_booth_function`, `_matyas_function`, `_rosenbrock_function`, `_sphere_function`, `_zakharov_function`. - Provide mathematical formulas for optimization problems.
Main Application Entry Point	
Main Function	- Define `main` function. - Initialize and start GUI application. - Handle startup exceptions.
Execution Guard	- Ensure `main` is called only when script is executed directly.

Classes and Methods

Component	Count	Description
Classes	2	`GeneticAlgorithm`, `GeneticAlgorithmGUI`
Methods	31	Includes all functions in both classes and standalone functions like `main`.

Attributes

Class	Attribute Count	Description
`GeneticAlgorithm`	14	Includes fitness function, bounds, dimensions, various rates, tracking lists, and execution time.
`GeneticAlgorithmGUI`	9	Includes Tkinter variables, optimization functions, state variables, and GUI components.

Operations (Methods)

- **`GeneticAlgorithm` Class Methods**: 8
  - `__init__`
  - `initialize_population`
  - `evaluate_fitness`
  - `selection`
  - `crossover`
  - `mutation`
  - `calculate_diversity`
  - `run`
- **`GeneticAlgorithmGUI` Class Methods**: 22
  - `__init__`
  - `_setup_variables`
  - `_create_main_layout`
  - `_create_setup_tab`
  - `_validate_and_start_optimization`
  - `_run_optimization`

    - `_optimization_complete`
    - `_update_results_tab`
    - `_update_visualization_tab`
    - `_update_analysis_tab`
    - `_create_results_tab`
    - `_create_visualization_tab`
    - `_create_analysis_tab`
    - `_setup_error_handling`
    - `_handle_optimization_error`
    - `_booth_function`
    - `_matyas_function`
    - `_rosenbrock_function`
    - `_sphere_function`
    - `_zakharov_function`
- **Standalone Functions**: 1
    - `main`
- Total Operations (Methods): 31
- Error Handling and GUI Components
- **Error Handling Methods**: 2
    - `_setup_error_handling`
    - `_handle_optimization_error`
- **GUI Components**: 5
    - Setup Tab
    - Results Tab
    - Visualization Tab
    - Analysis Tab
    - Main Layout (Notebook and Frame)

Objective Functions

Function Name	Description
`_booth_function`	Implements the Booth function for optimization.
`_matyas_function`	Implements the Matyas function for optimization.
`_rosenbrock_function`	Implements the Rosenbrock function for optimization.
`_sphere_function`	Implements the Sphere function for optimization.
`_zakharov_function`	Implements the Zakharov function for optimization.

Summary

- **Total Classes**: 2
- **Total Attributes**: 23
- **Total Operations (Methods)**: 31
- **Standalone Functions**: 1
- **Error Handling Methods**: 2
- **Objective Functions**: 5
- **GUI Components**: 5

Library	Purpose

Library	Purpose
`numpy`	Used for numerical computations, specifically for handling arrays and random number generation.
`matplotlib.pyplot`	Used for creating static, animated, and interactive visualizations in Python, particularly for plotting graphs.
`mpl_toolkits.mplot3d`	Provides 3D plotting capabilities to matplotlib.
`time`	Used to measure the execution time of the algorithm.
`json`	Used for handling JSON data structures, though not explicitly utilized in the provided code.
`datetime`	Used for date and time operations, though not explicitly utilized in the provided code.
`os`	Provides functionalities to interact with the operating system, though not explicitly utilized in the code.
`traceback`	Used for error tracking and debugging, providing detailed stack traces during exceptions.
`threading`	Used for concurrent execution, allowing the optimization process to run in a separate thread.
`random`	Used for random number generation, though primarily replaced by numpy's random functionalities in the code.
`itertools`	Provides efficient looping constructs, though not explicitly utilized in the provided code.
`tkinter`	Provides standard GUI toolkit for Python, used to create the graphical interface of the application.
`tkinter.ttk`	Provides themed widget set for Tkinter, used for improved GUI appearance.
`tkinter.messagebox`	Used for displaying message boxes in the GUI, typically for error messages or informational pop-ups.
`tkinter.filedialog`	Used for opening file dialogs, though not explicitly utilized in the provided code.
`matplotlib.figure`	Used to create a figure object, which is a container for all plot elements.
`matplotlib.backends.backend_tkagg`	Provides the TkAgg backend for embedding matplotlib plots into Tkinter applications.

Library	Imported Functions/Classes
`numpy`	`np` (imported as an alias for the entire library to handle numerical computations, arrays, and random number generation)
`matplotlib.pyplot`	`plt` (imported as an alias for creating static, animated, and interactive visualizations)
`mpl_toolkits.mplot3d`	`Axes3D` (used to enable 3D plotting capabilities in matplotlib)
`time`	`time` (used to measure execution time of the algorithm)
`json`	`json` (imported for handling JSON data, though not explicitly used in the provided code)
`datetime`	`datetime` (imported for date and time operations, though not explicitly used)
`os`	`os` (imported for operating system interactions, though not explicitly used)

Library	Imported Functions/Classes
`traceback`	`traceback` (used for error tracking and debugging, providing detailed stack traces during exceptions)
`threading`	`threading` (used to run the optimization process in a separate thread, allowing concurrent execution)
`random`	`random` (imported for random number generation, though numpy's random functionalities are primarily used)
`itertools`	`itertools` (imported for efficient looping constructs, though not explicitly used)
`tkinter`	`tk` (imported as an alias to provide GUI toolkit functionality)
`tkinter.ttk`	`ttk` (imported for themed widget set, enhancing the appearance of the Tkinter GUI)
`tkinter.messagebox`	`messagebox` (used for displaying message boxes in the GUI, such as error or information pop-ups)
`tkinter.filedialog`	`filedialog` (imported for file dialog operations, though not explicitly used)
`matplotlib`	`matplotlib.use('TkAgg')` (configures the matplotlib backend to `TkAgg` for compatibility with Tkinter applications)
`matplotlib.figure`	`Figure` (used to create a figure object, serving as a container for plot elements)
`matplotlib.backends.backend_tkagg`	`FigureCanvasTkAgg`, `NavigationToolbar2Tk` (used to embed matplotlib figures in Tkinter applications and provide interactive toolbar functionalities)

ProjectGA.py — Edited & New Functions: Change Summary

This document highlights only the **functions that were edited or newly added** for the latest improvements, and succinctly explains what was changed and why.

---

1. DifferentialEvolution class (New)

- **What:**
  Entirely new class implementing the Differential Evolution (DE) optimizer.
- **Purpose:**
  Offers a second metaheuristic algorithm (besides GA). Includes methods for initialization, main loop, and progress tracking similar to the GA structure.
- **Impact:**
  Enables the GUI to compare/benchmark DE versus GA side–by–side.

---

2. MultiAlgorithmGUI class (Major Edits & Additions)

Key Edited/Added Methods:

- `_setup_variables`
  - **What:**
    Expanded to include:
    - Differential Evolution controls (differential weight parameter).
    - Algorithm selection (`algorithm_var`, new dropdown for DE/GA/Both).
  - **Why:**
    Supports the GUI's ability to run one or both algorithms.
- `_create_setup_tab`
  - **What:**
    Edits to GUI setup:
    - New dropdown for algorithm selection (GA/DE/Both).
    - Added controls to set DE-specific parameter.
  - **Why:**
    Allows user to choose algorithm and set all parameters in the interface.
- `_validate_and_start_optimization`
  - **What:**
    Input validation updated to check both GA and DE parameters.
  - **Why:**
    Ensures correct parameter input for whichever algorithm(s) are run.
- `_run_optimization`

- **What:**
  Substantial changes:
  - Runs one or both algorithms based on the dropdown.
  - Ensures both use identical parameter sets and random seeds for fairness.
  - Stores results for both, enabling comparison.
- **Why:**
  Fulfills the requirement for simultaneous, fair benchmarking and report.

- `_update_results_tab`, `_update_visualization_tab`, `_update_analysis_tab`
  - **What:**
    All updated to:
    - Display both algorithms' results side-by-side if both were run.
    - Add DE-specific results and plots.
    - For GA, still shows advanced stats like diversity and mutation rate.
  - **Why:**
    Presents comparative results clearly to the user.

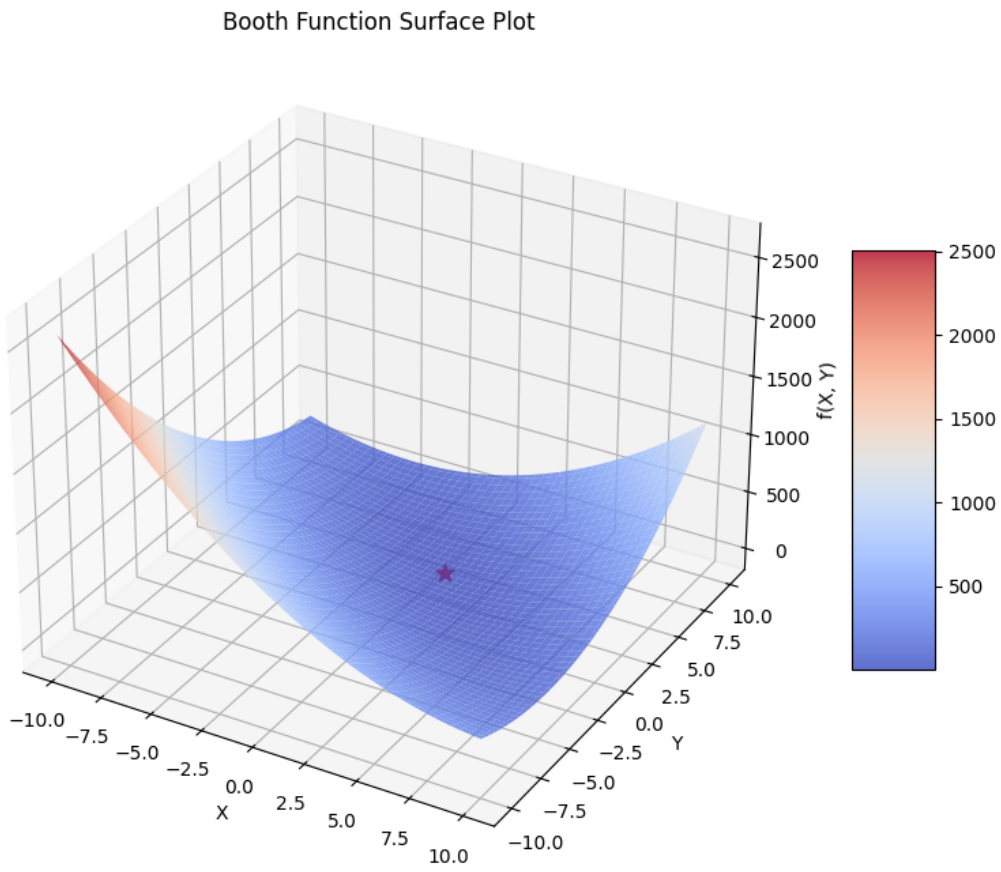
---

4. Optimization Results: Runs & Visualizations

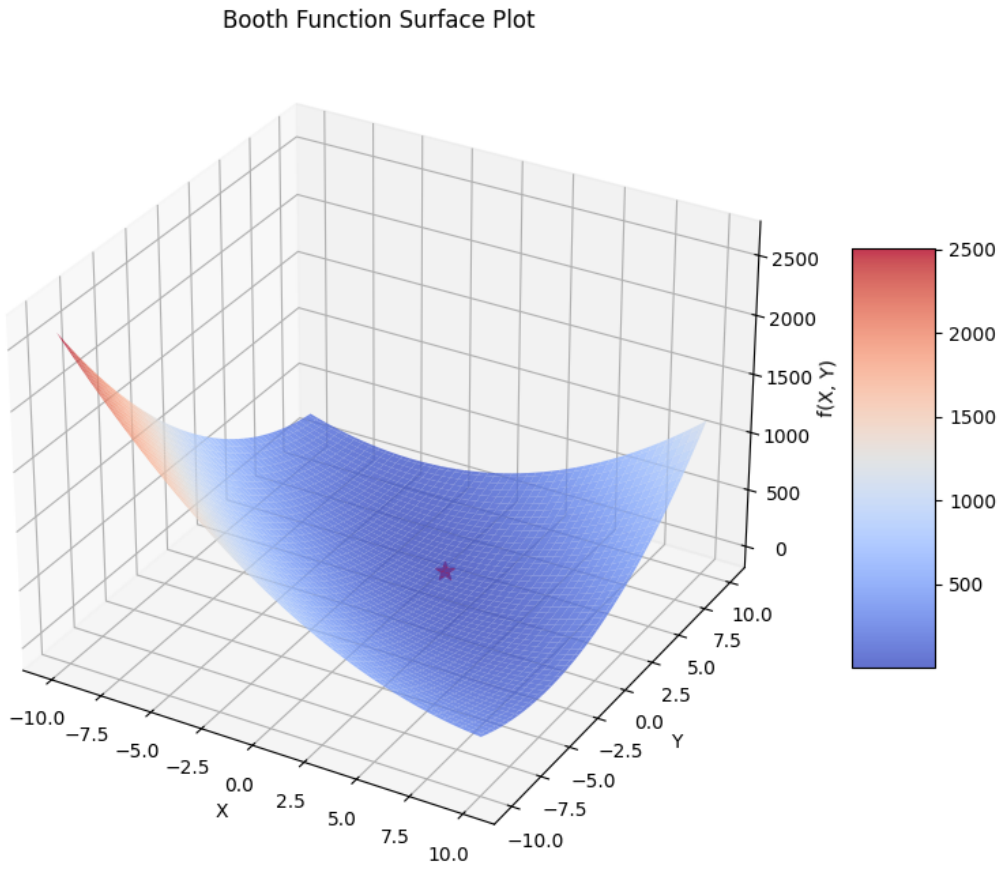
*For each function, we present images/results for **both Genetic Algorithm (GA)** and **Differential Evolution (DE)** runs. Each function section includes surface plots (when applicable), optimization paths (if in 2D/3D), and convergence/progress plots for GA and DE.*

4.1 Booth Function

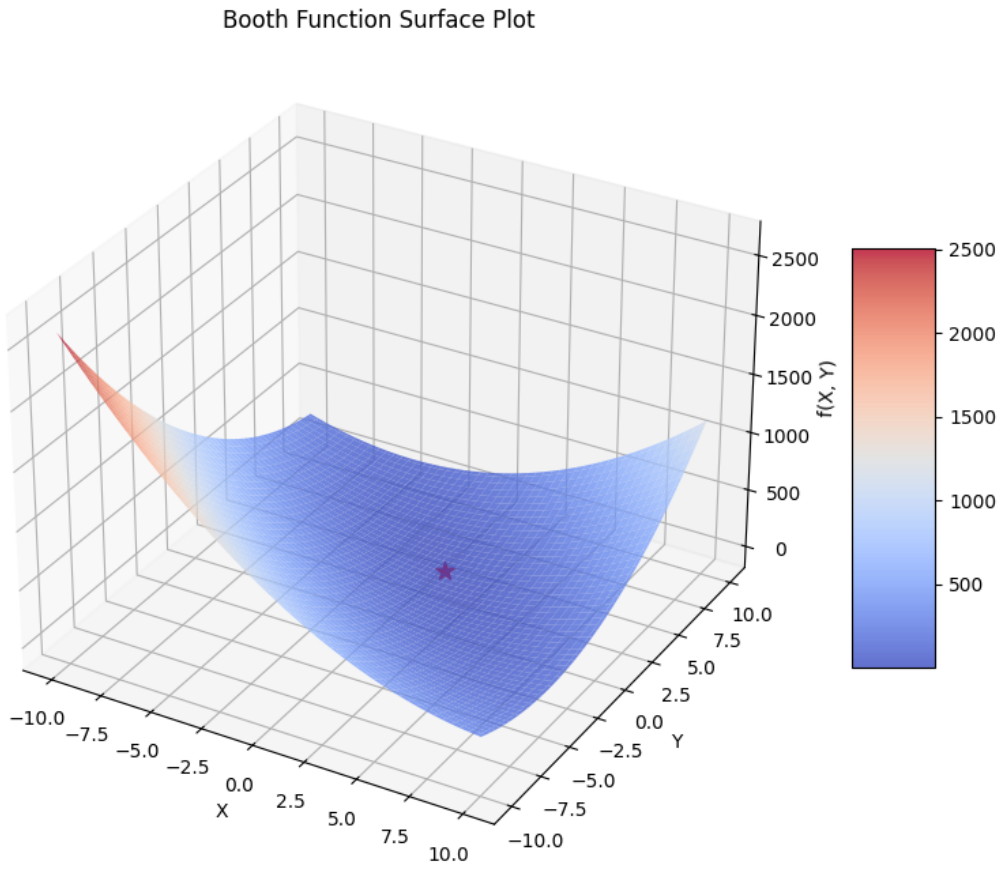
Surface Plots



Surface plot - Booth function shape.

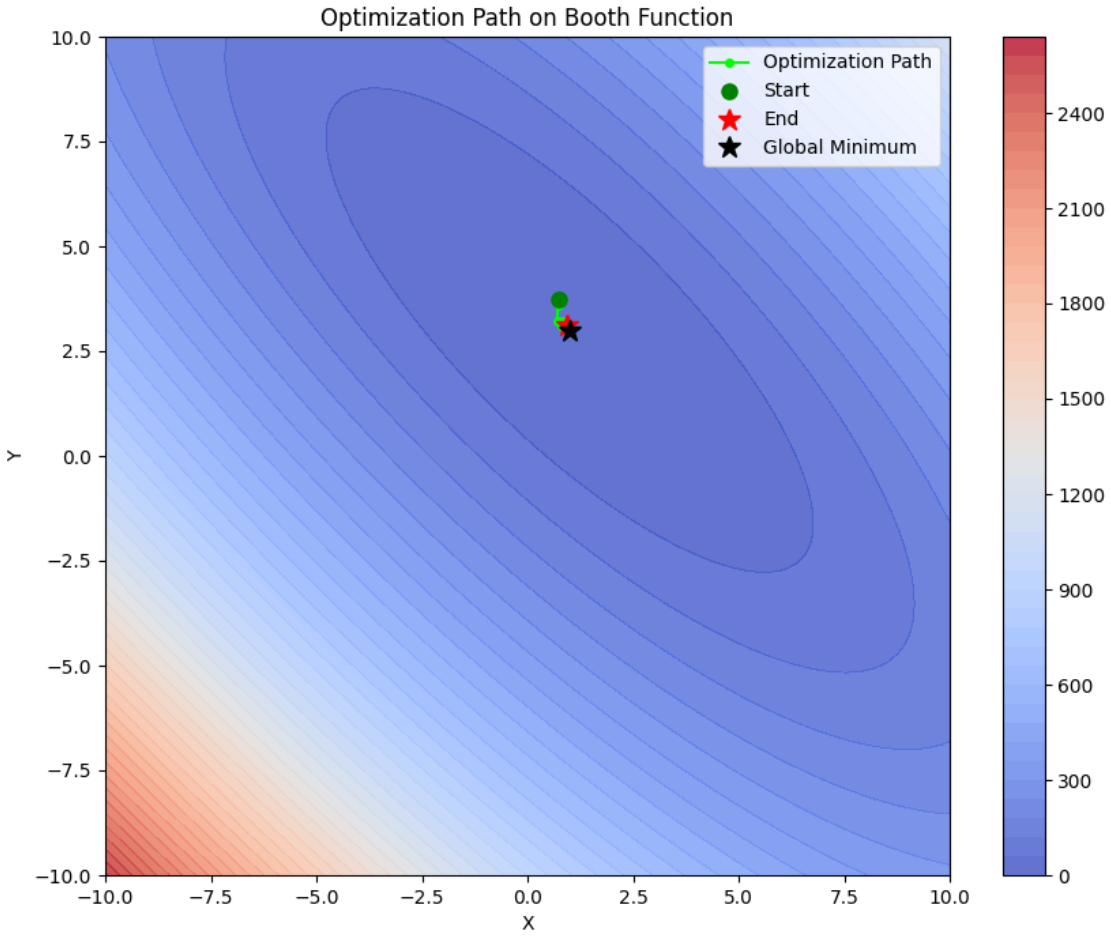


Surface plot from another perspective.

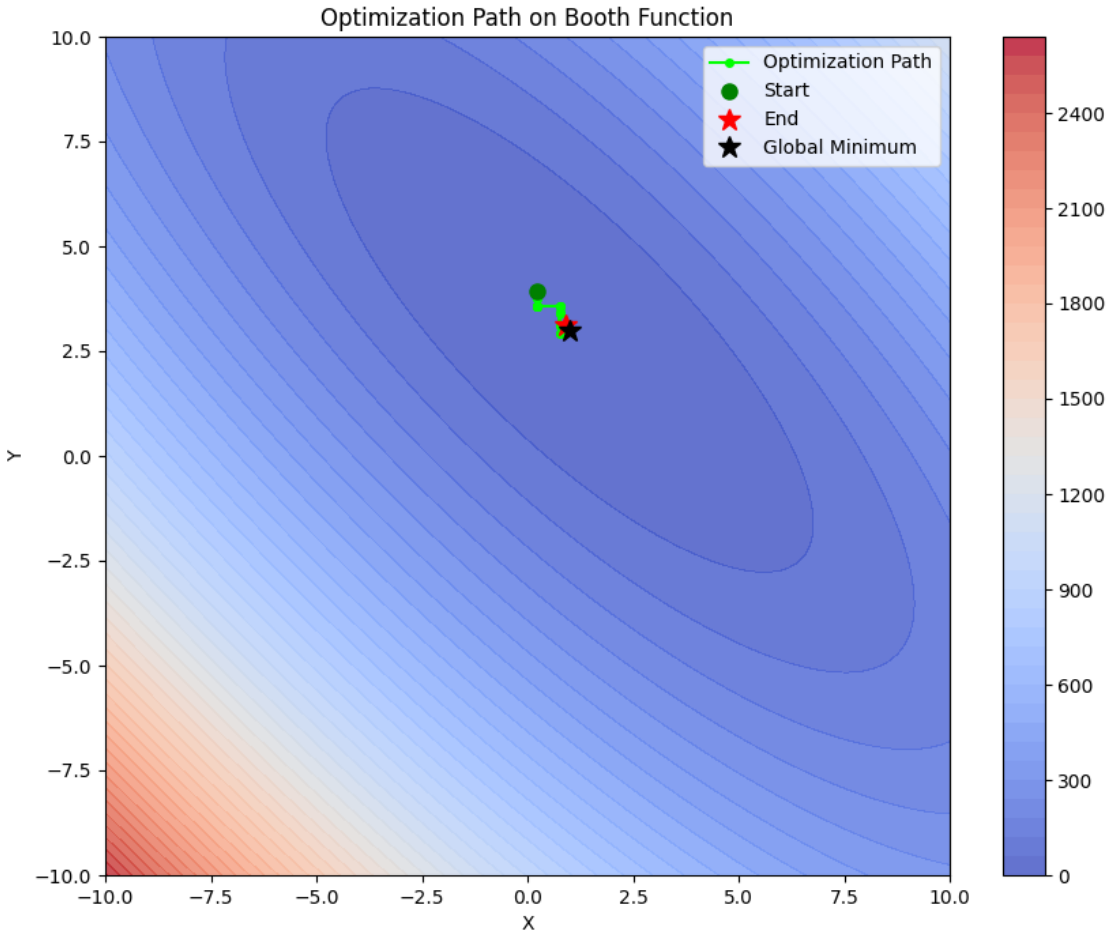


Surface plot showing a different run.

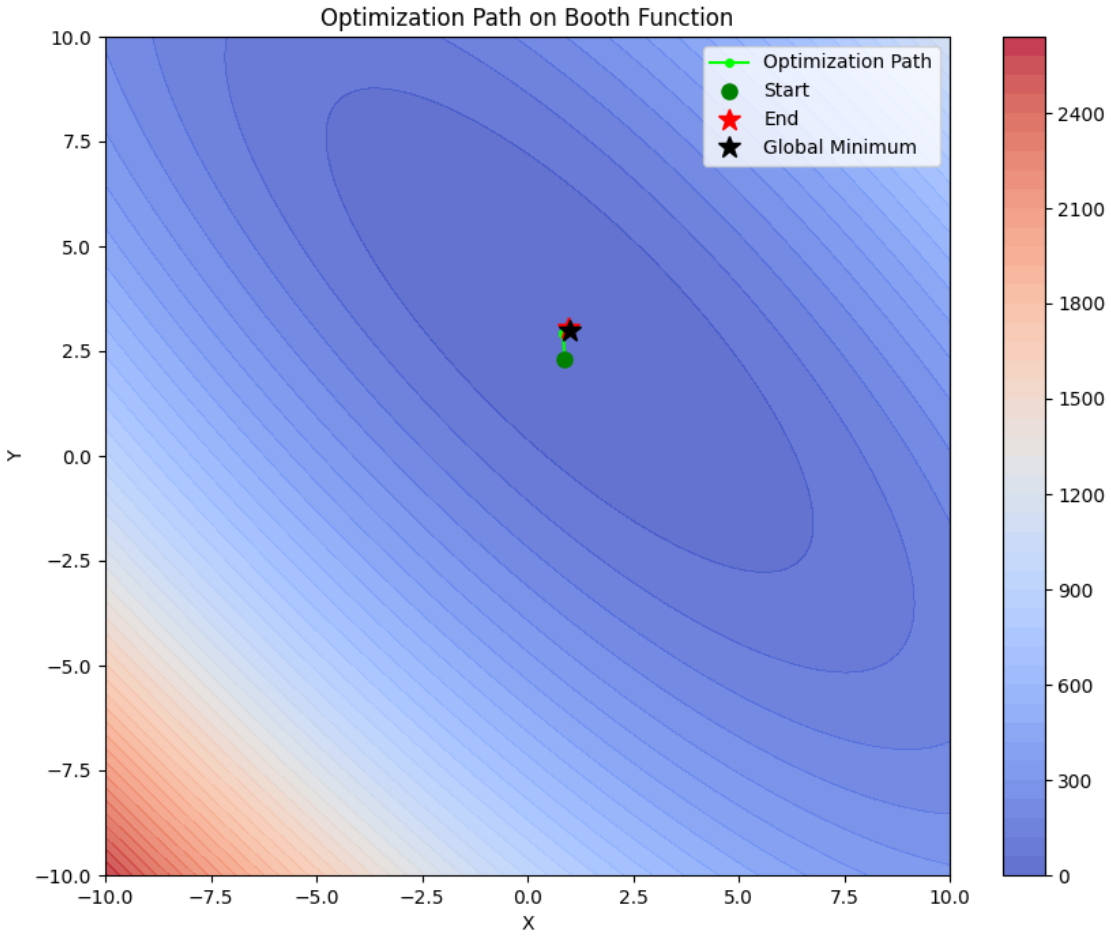
Path Plots



Example search path, run 1.

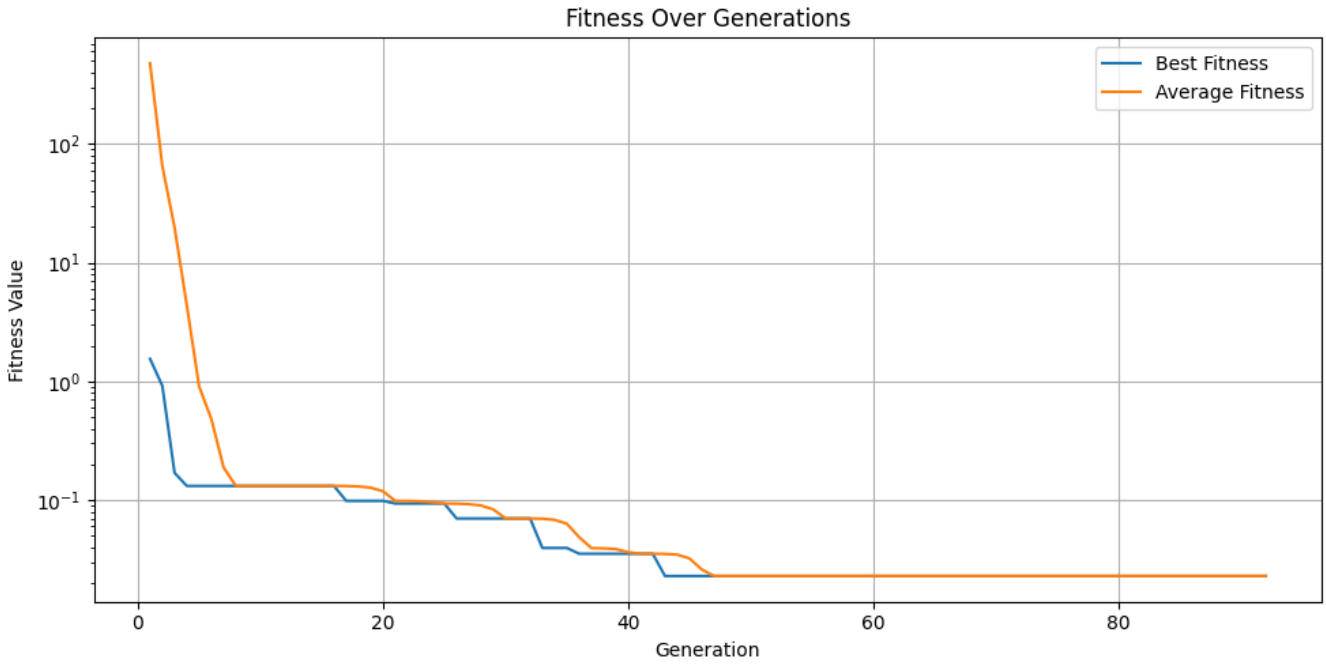


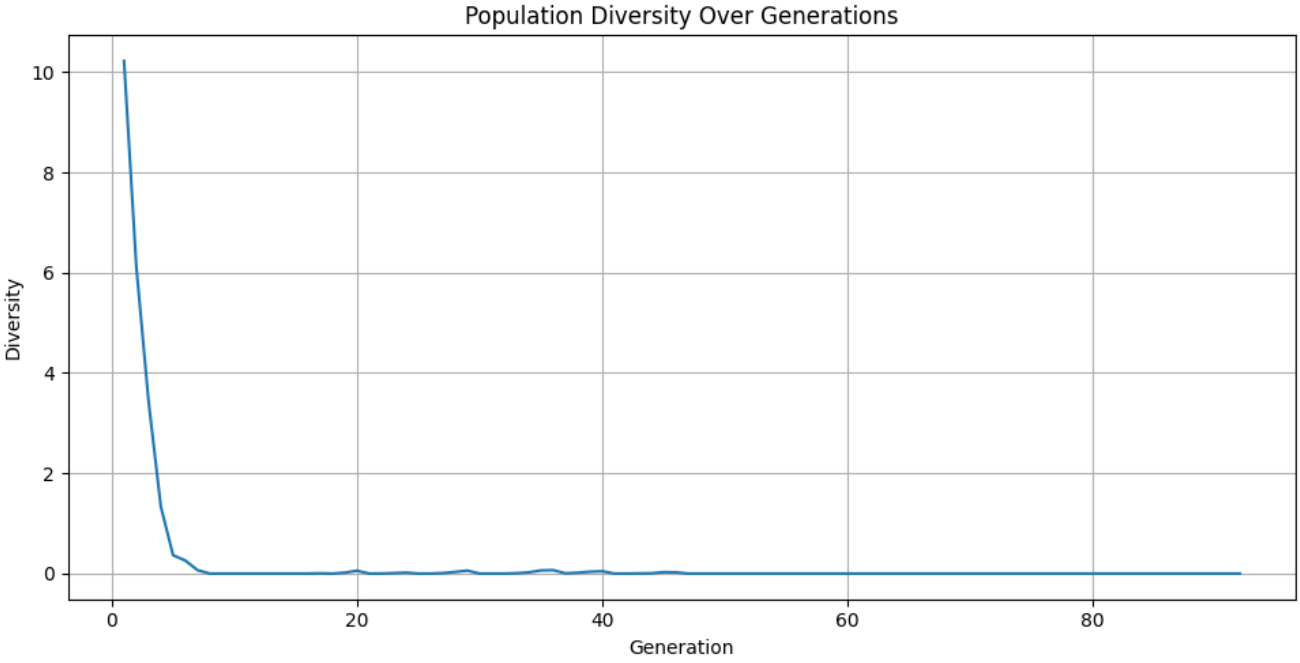
Example search path, run 2.

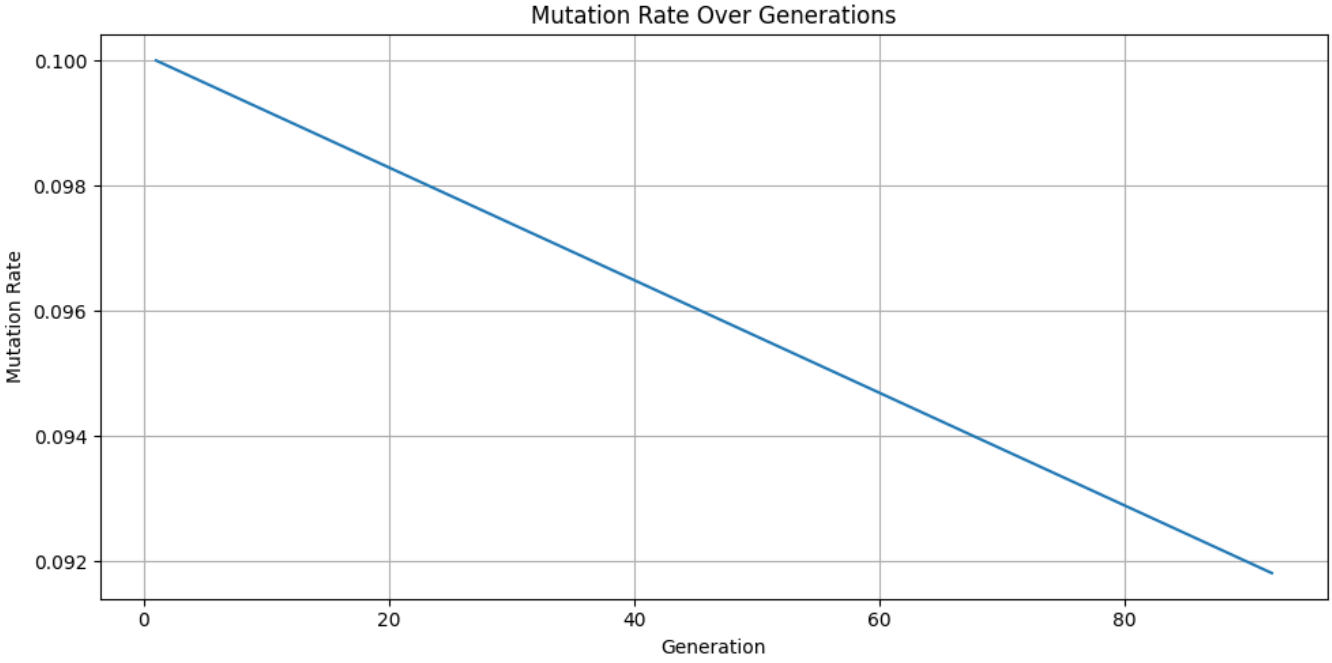


Example search path, run 3.

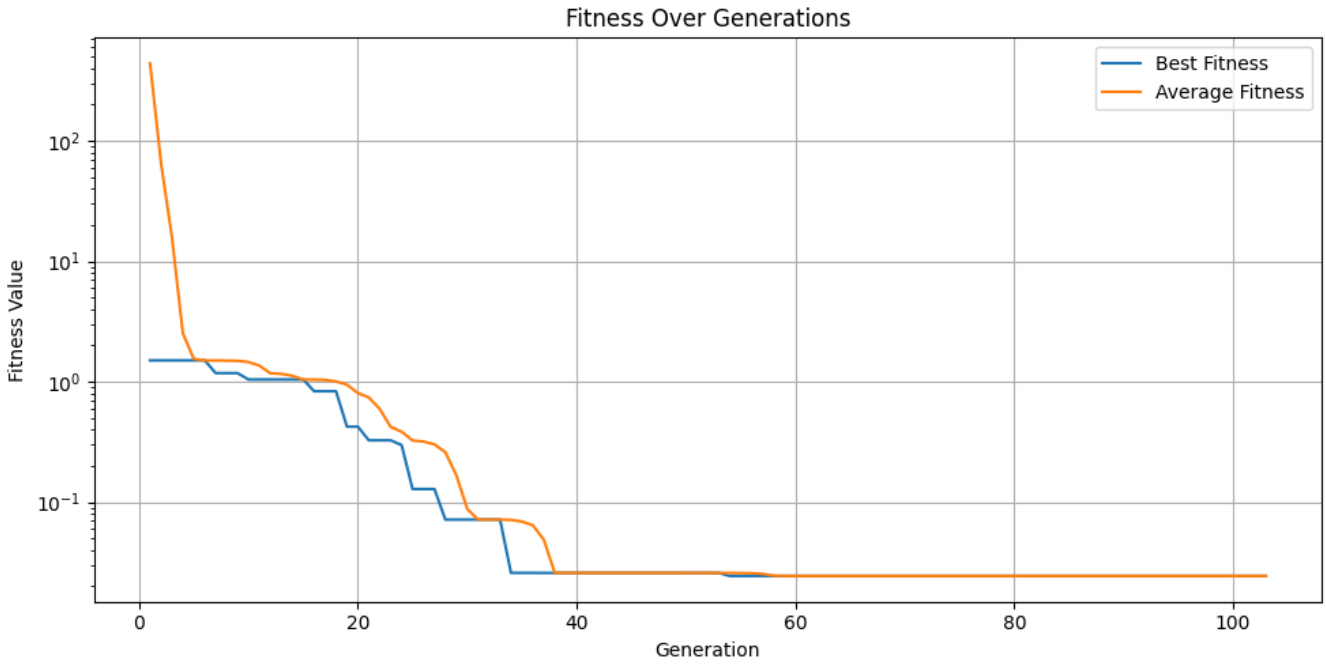
Progress Plots

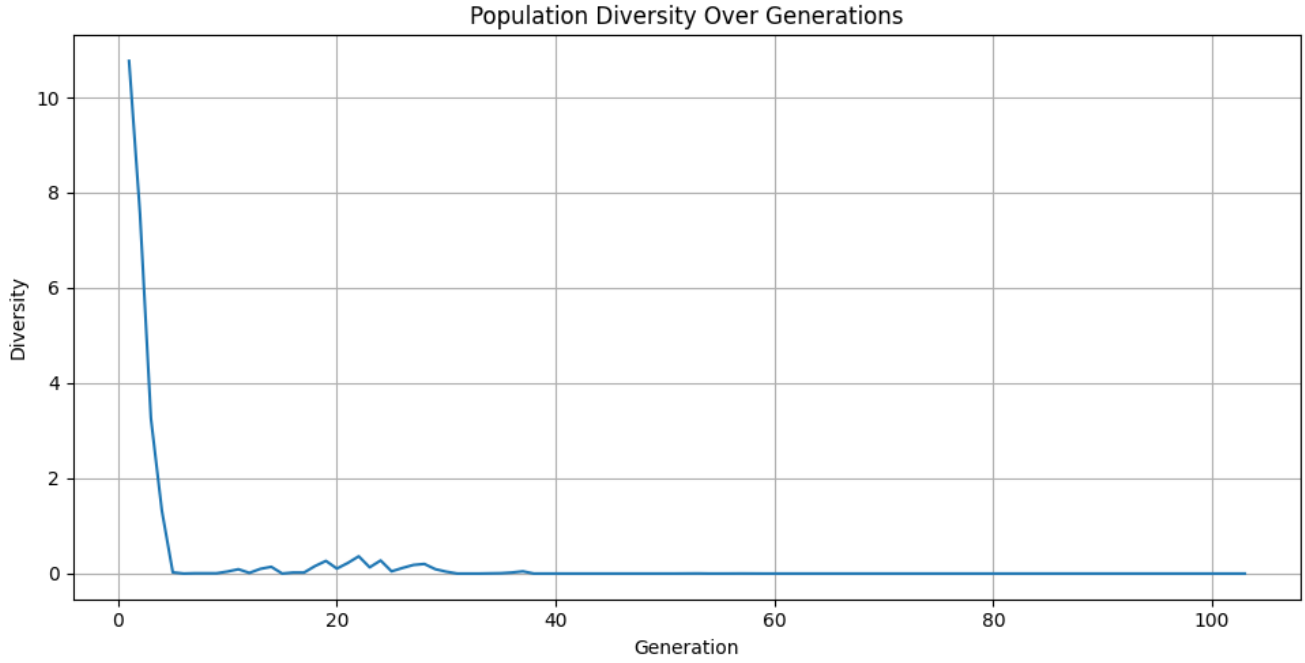

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
10^2
10^1
10^0
10^{-1}
0
20
40
60
80
Generation

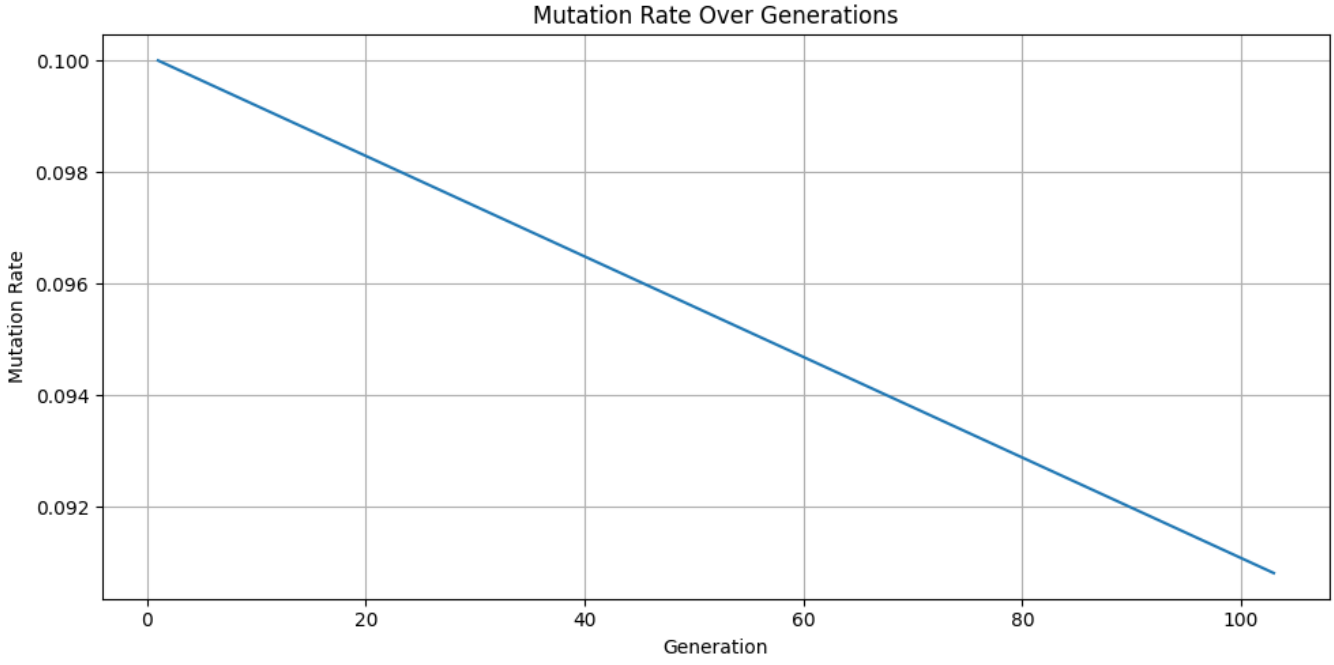

Population Diversity Over Generations
Diversity
10
8
6
4
2
0
0
20
40
60
80
Generation


Mutation Rate Over Generations
Mutation Rate
0.100
0.098
0.096
0.094
0.092
0
20
40
60
80
Generation

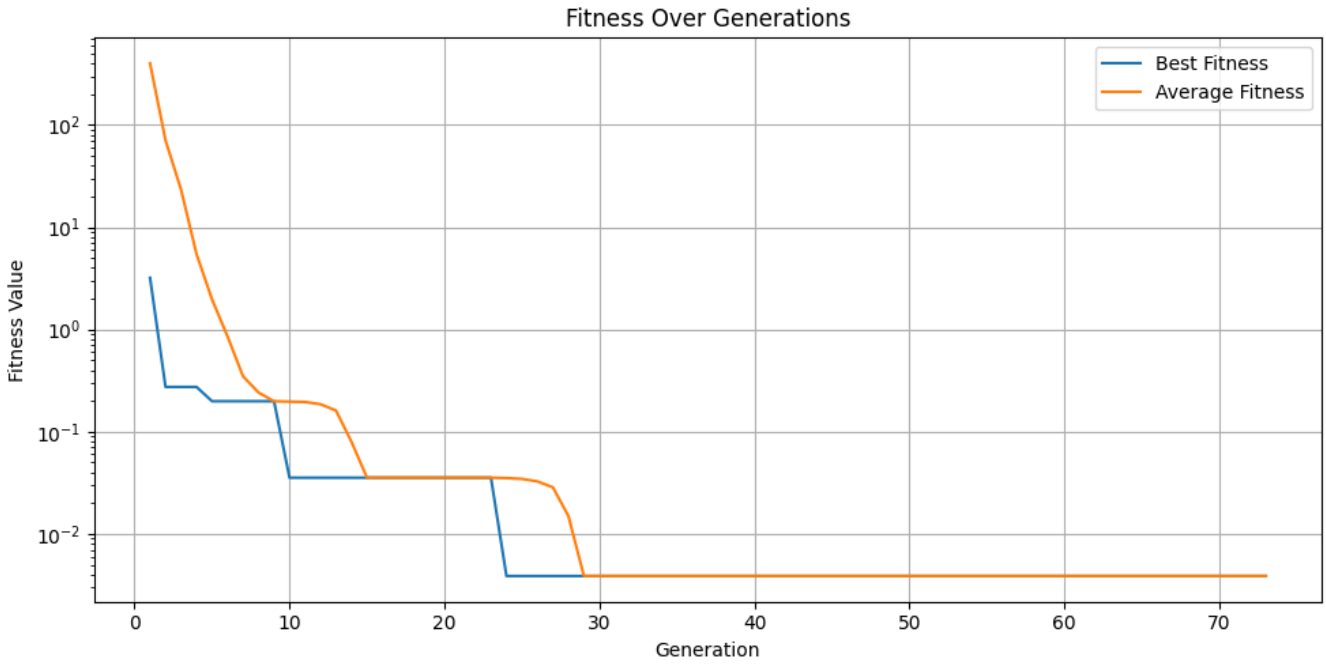
Fitness vs. generations, run 1.

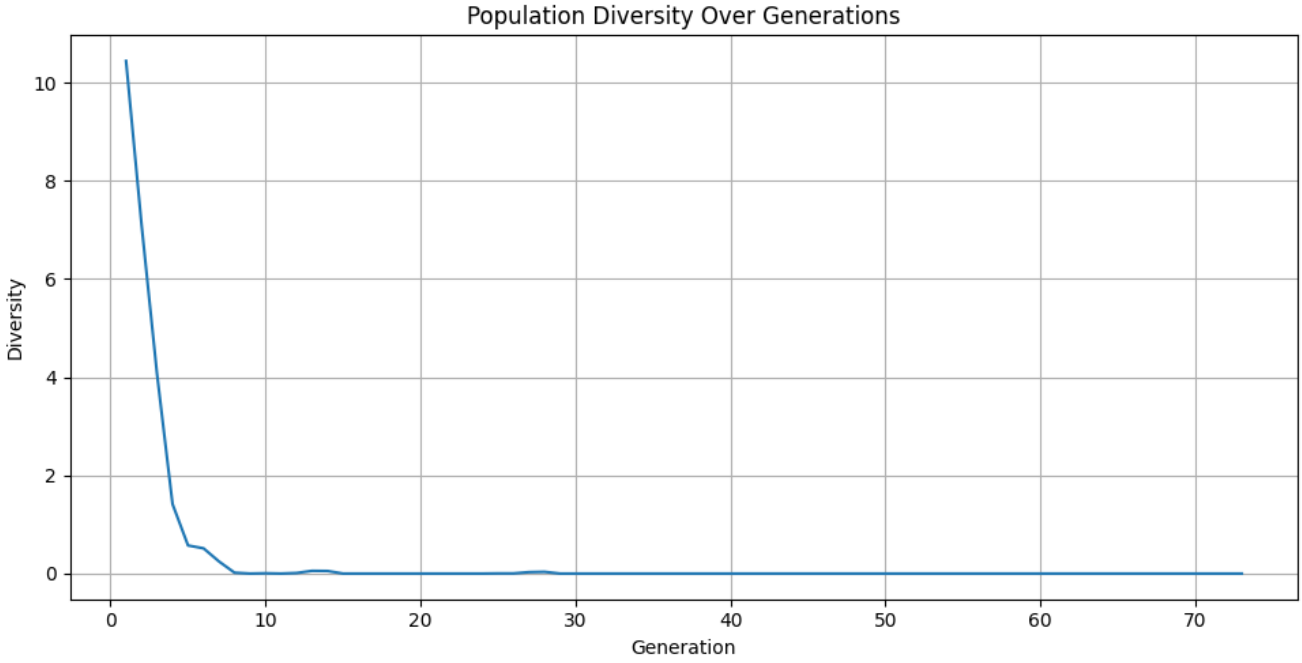

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
10^2
10^1
10^0
10^{-1}
0
20
40
60
80
100
Generation

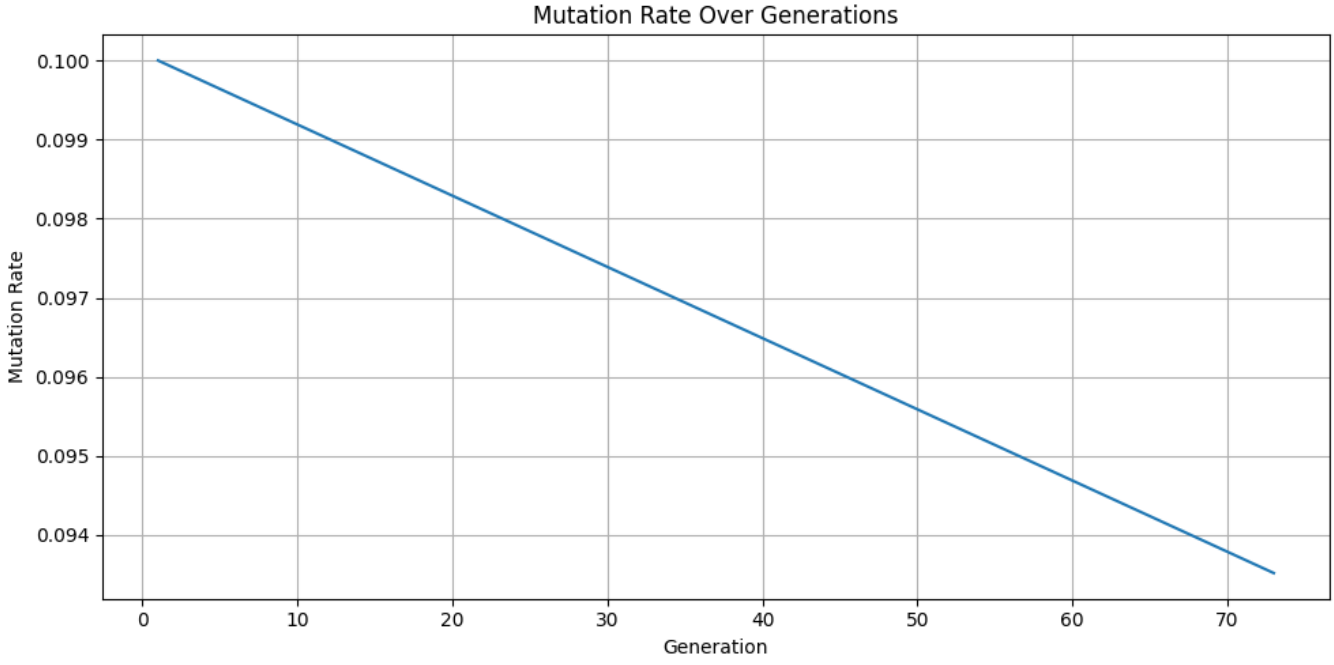

Population Diversity Over Generations
Diversity
10
8
6
4
2
0
0
20
40
60
80
100
Generation


Mutation Rate Over Generations
Mutation Rate
0.100
0.098
0.096
0.094
0.092
0
20
40
60
80
100
Generation

Fitness vs. generations, run 2.


Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

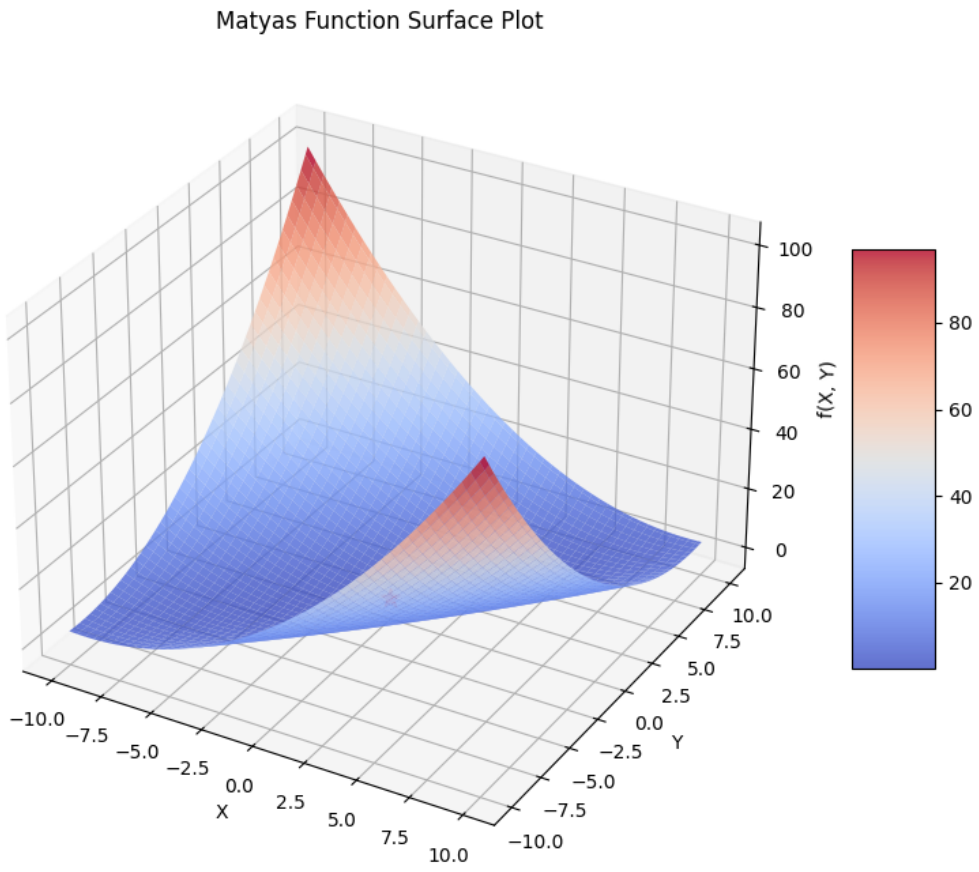

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

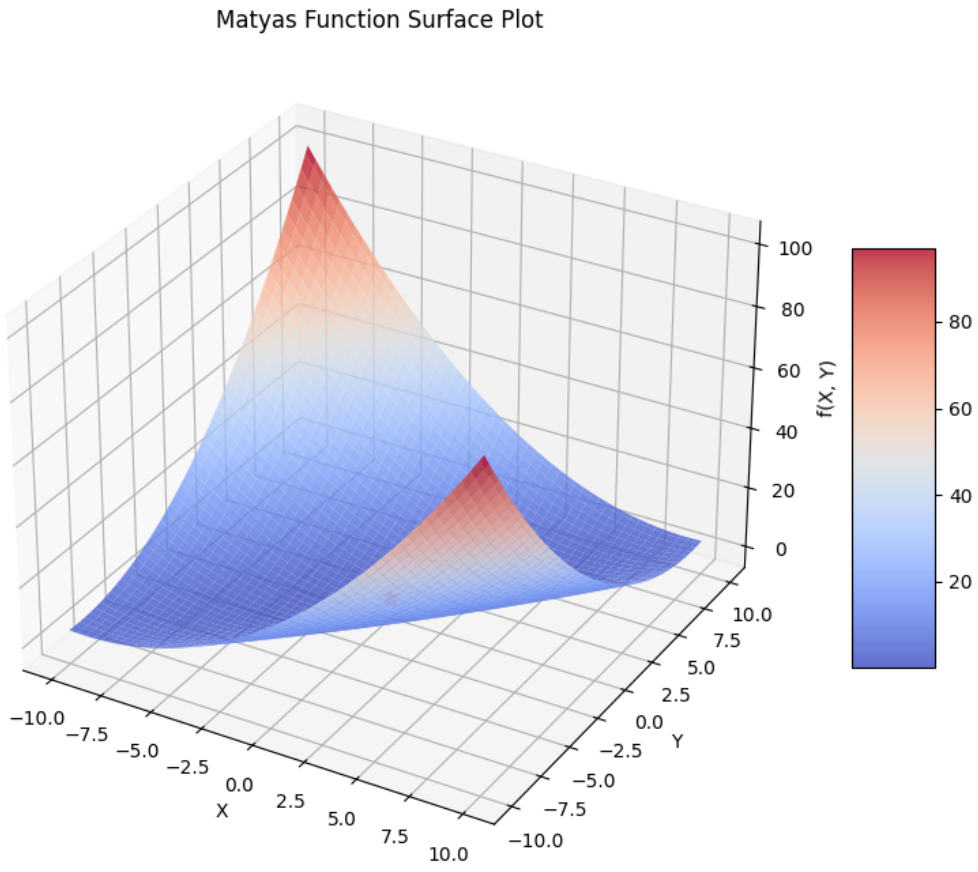
Fitness vs. generations, run 3.

4.2 Matyas Function

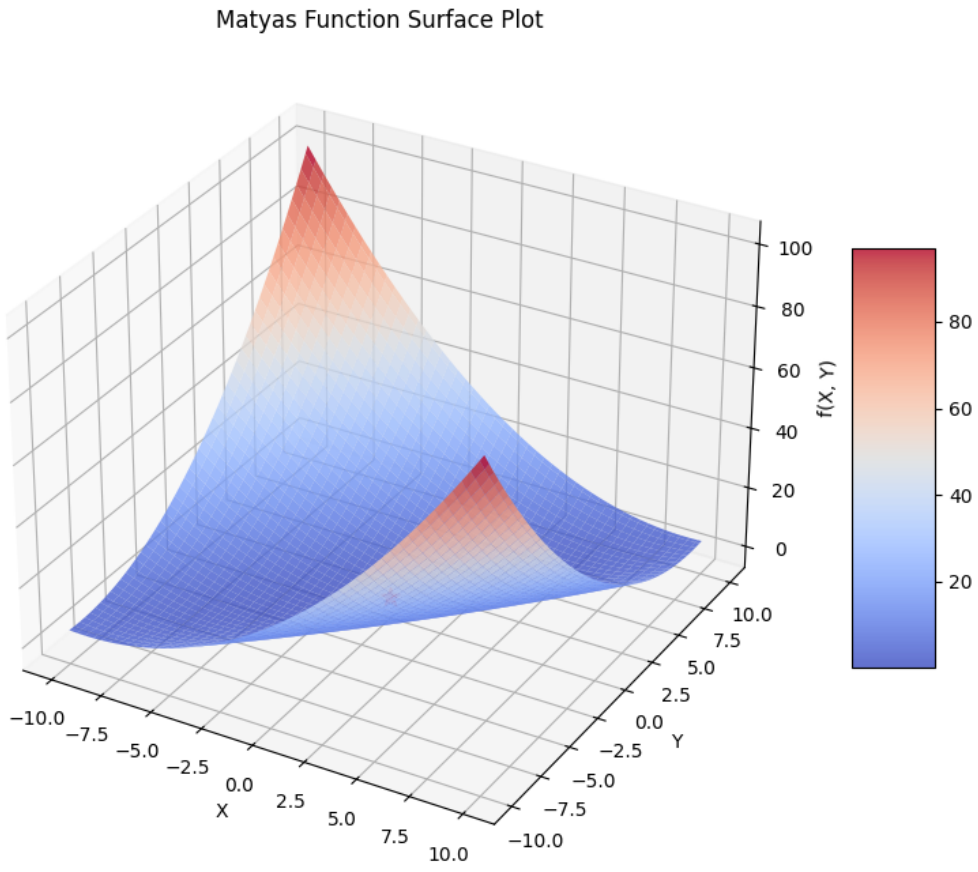
Surface Plots



Surface plot - Matyas shape.

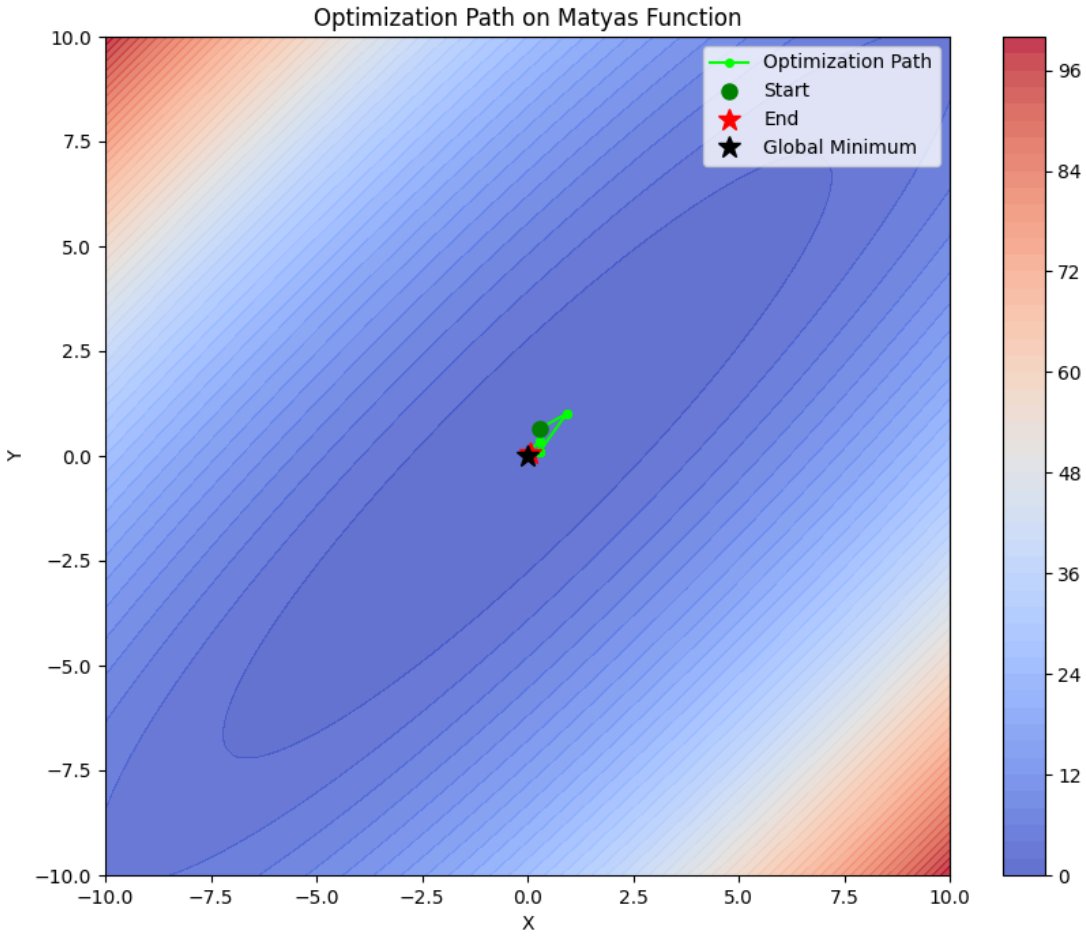


Surface plot another angle.

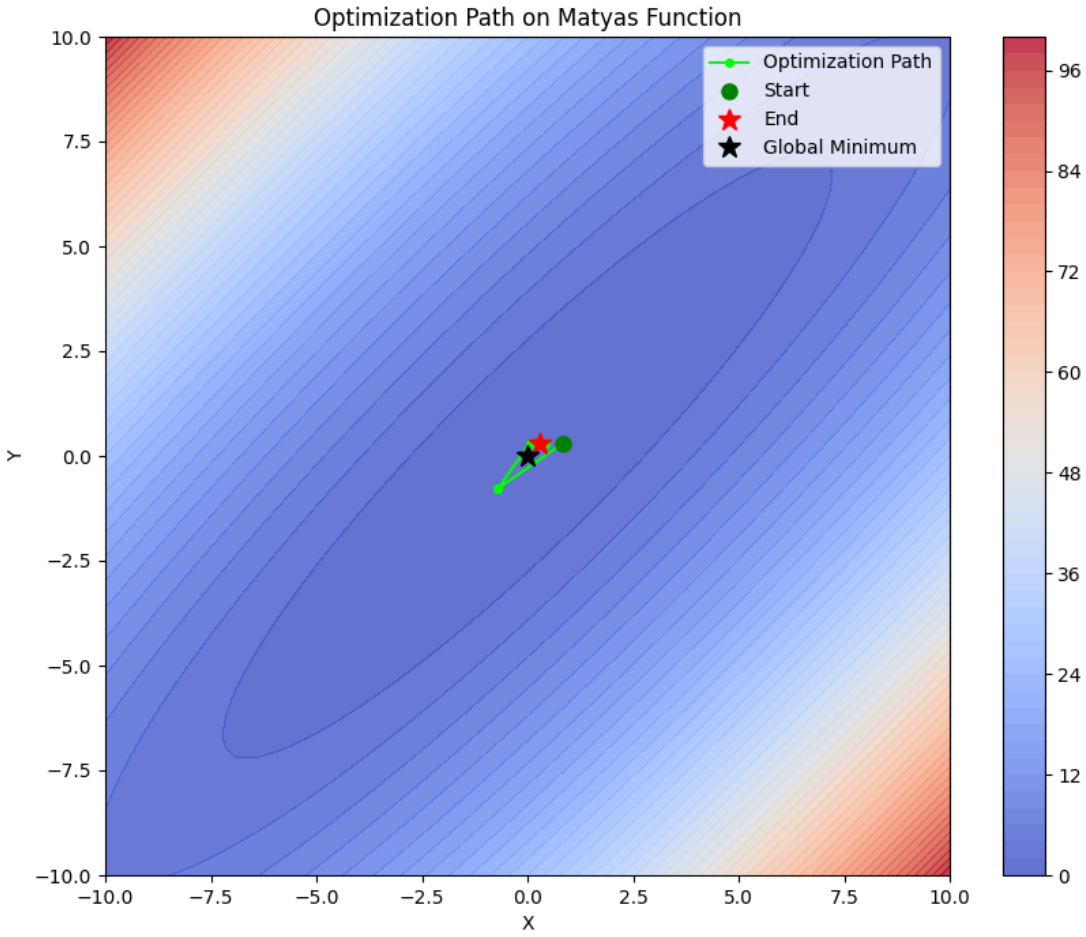


Surface plot, other variation.

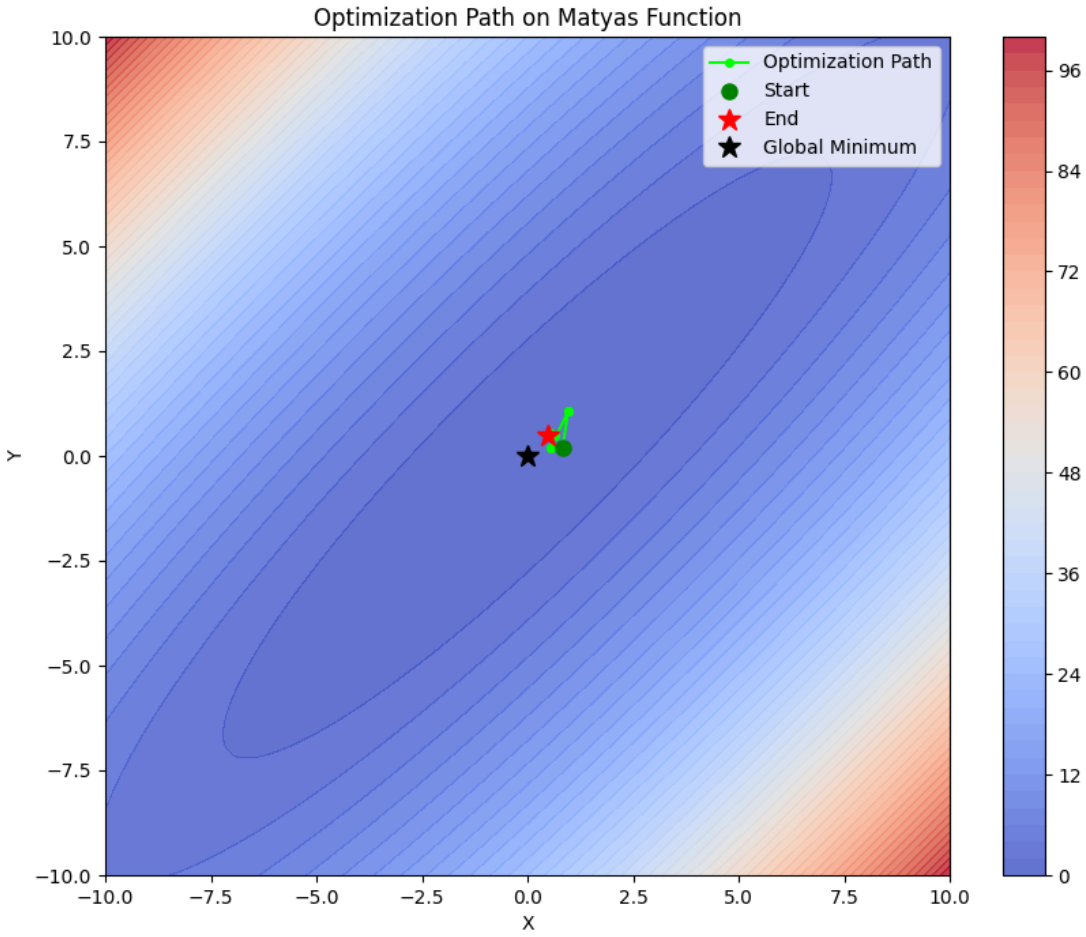
Path Plots



Optimization path, run 1.

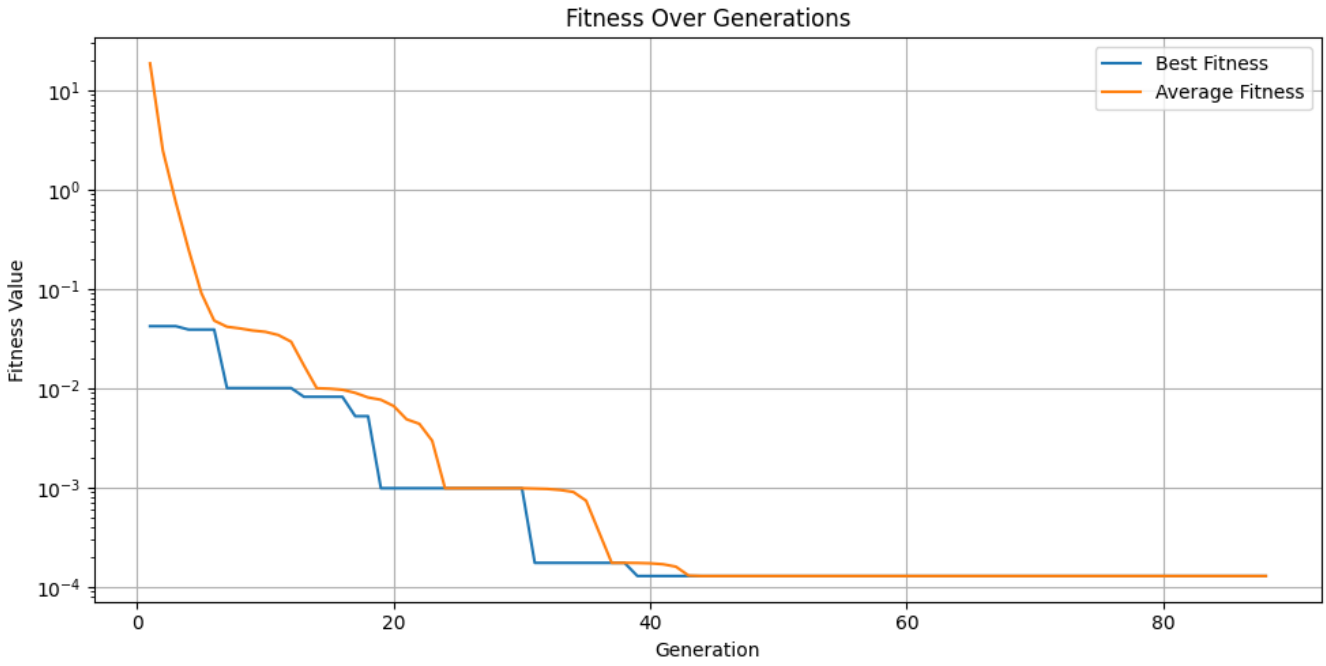


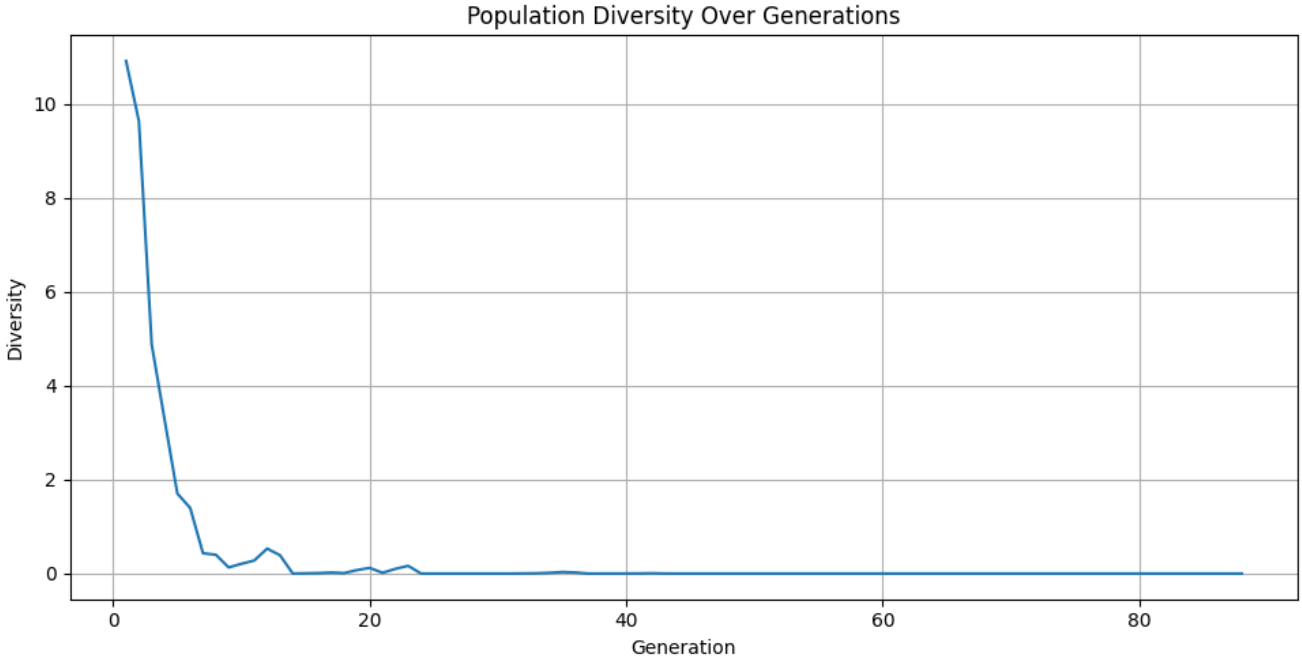
Optimization path, run 2.

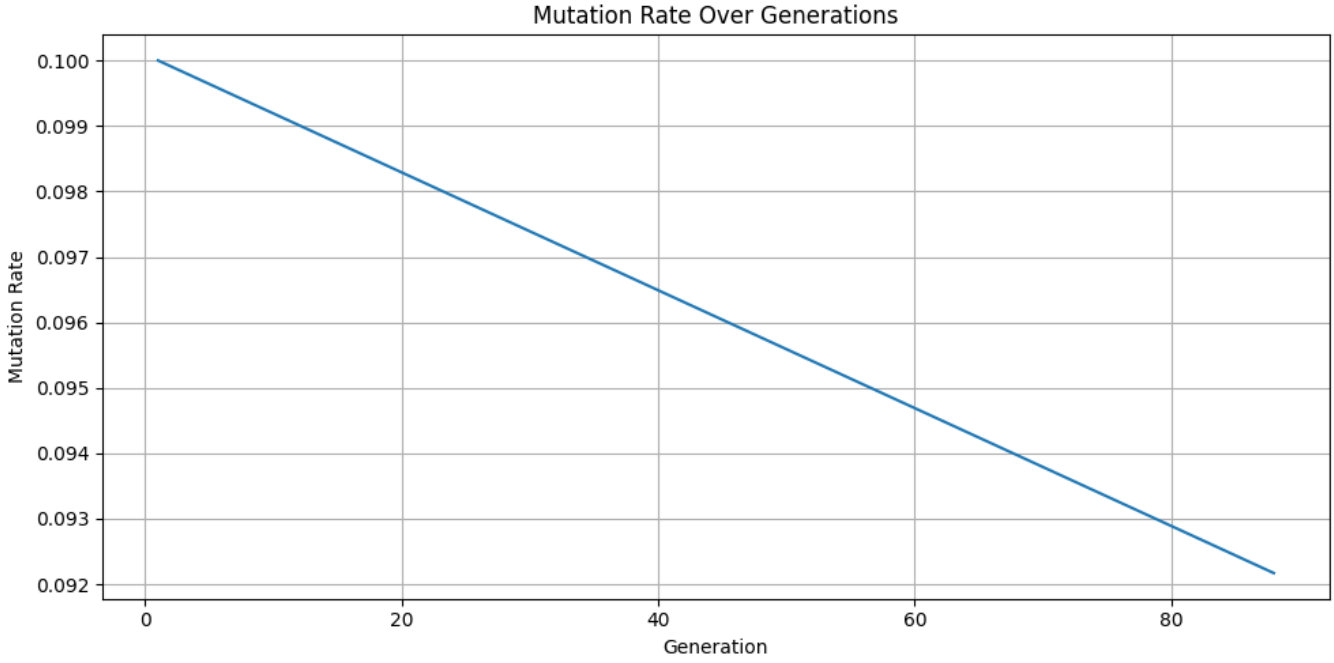


Optimization path, run 3.

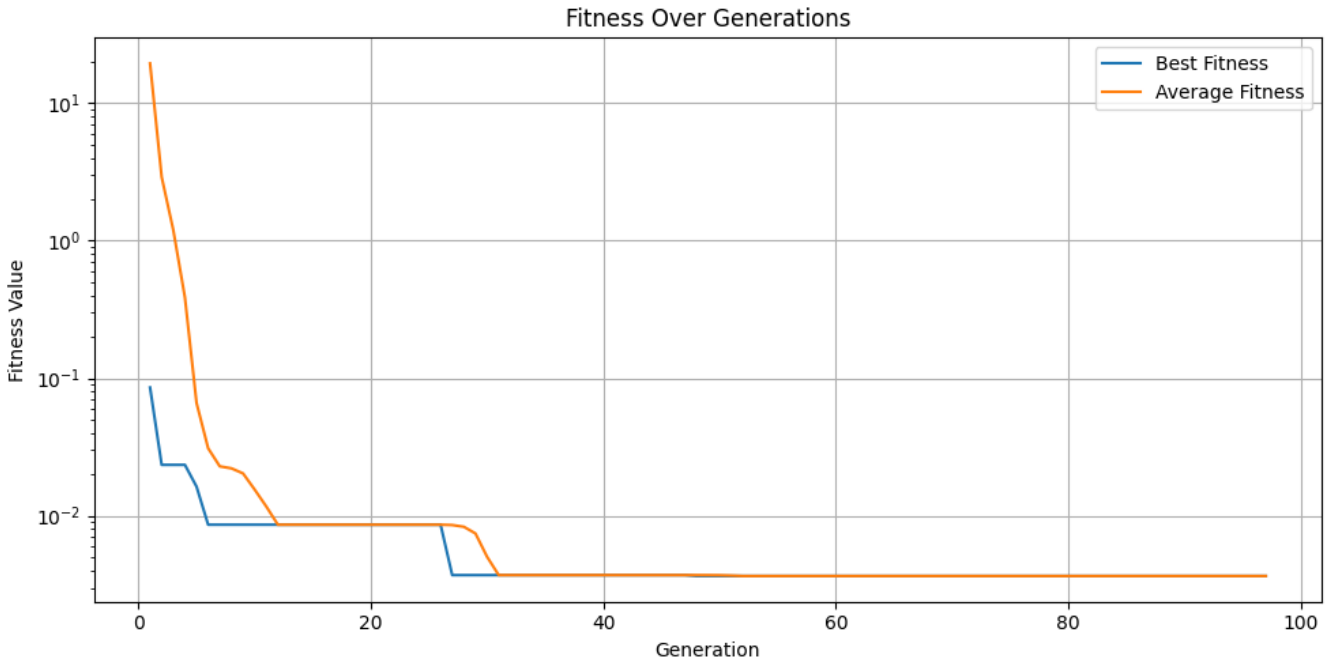
Progress Plots

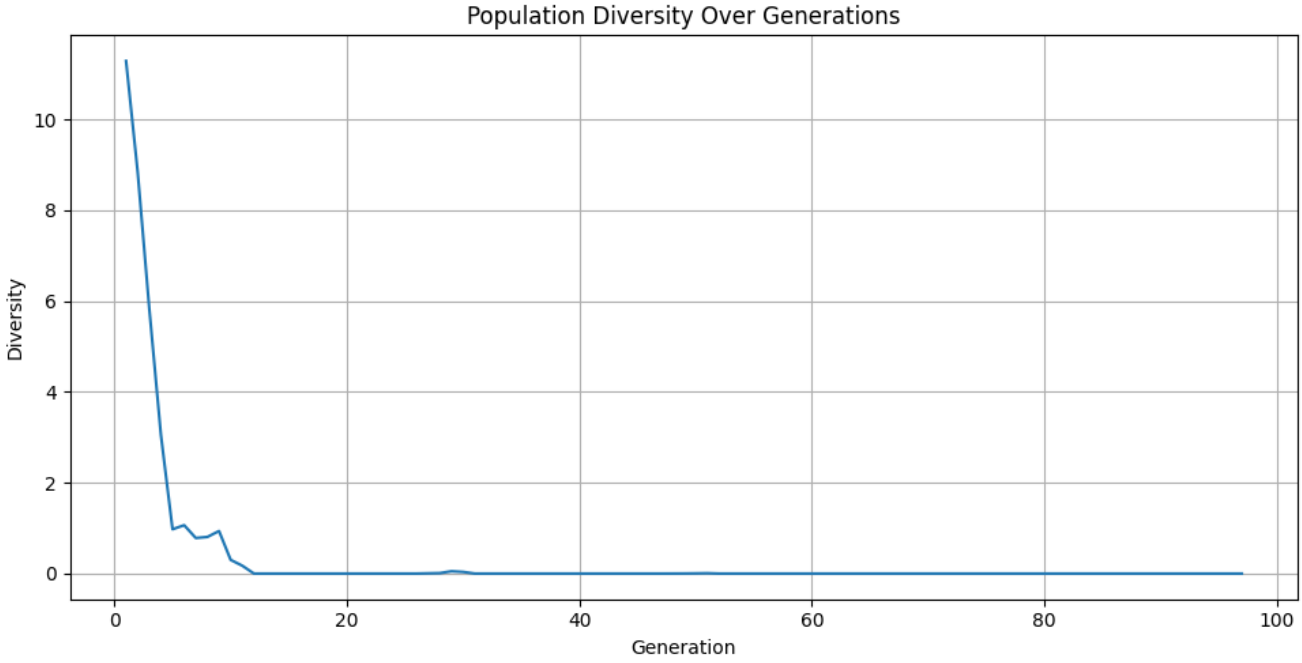

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

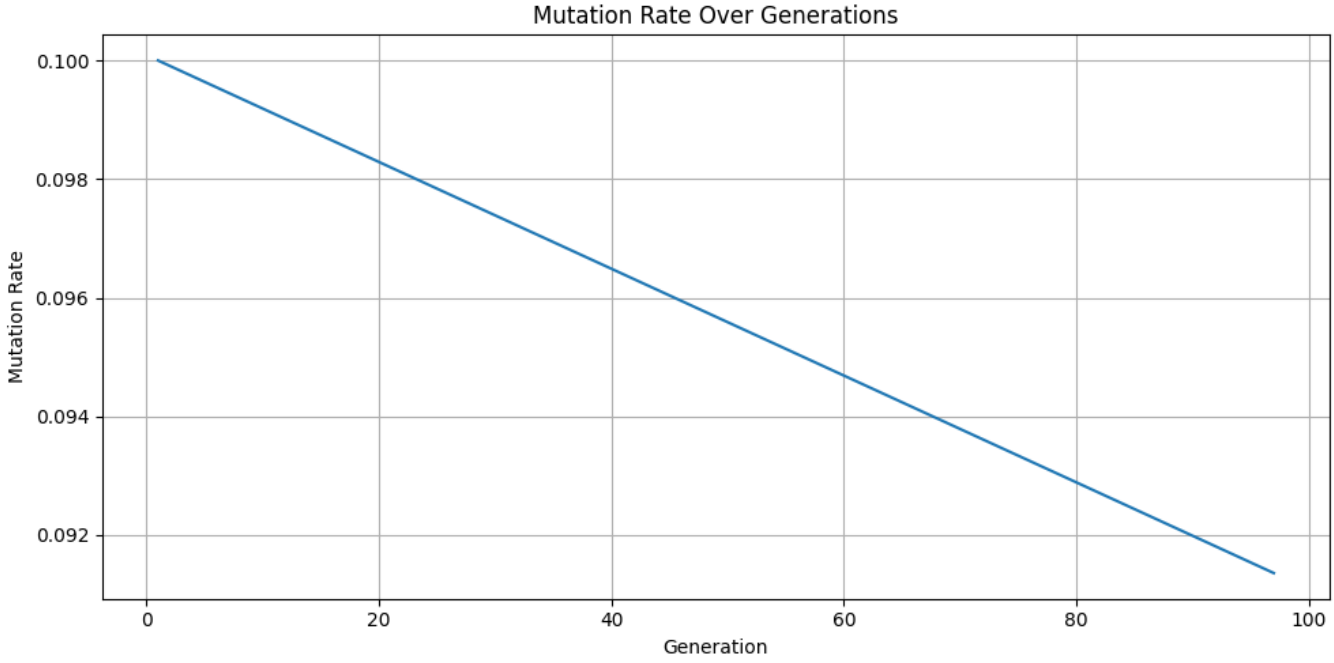

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

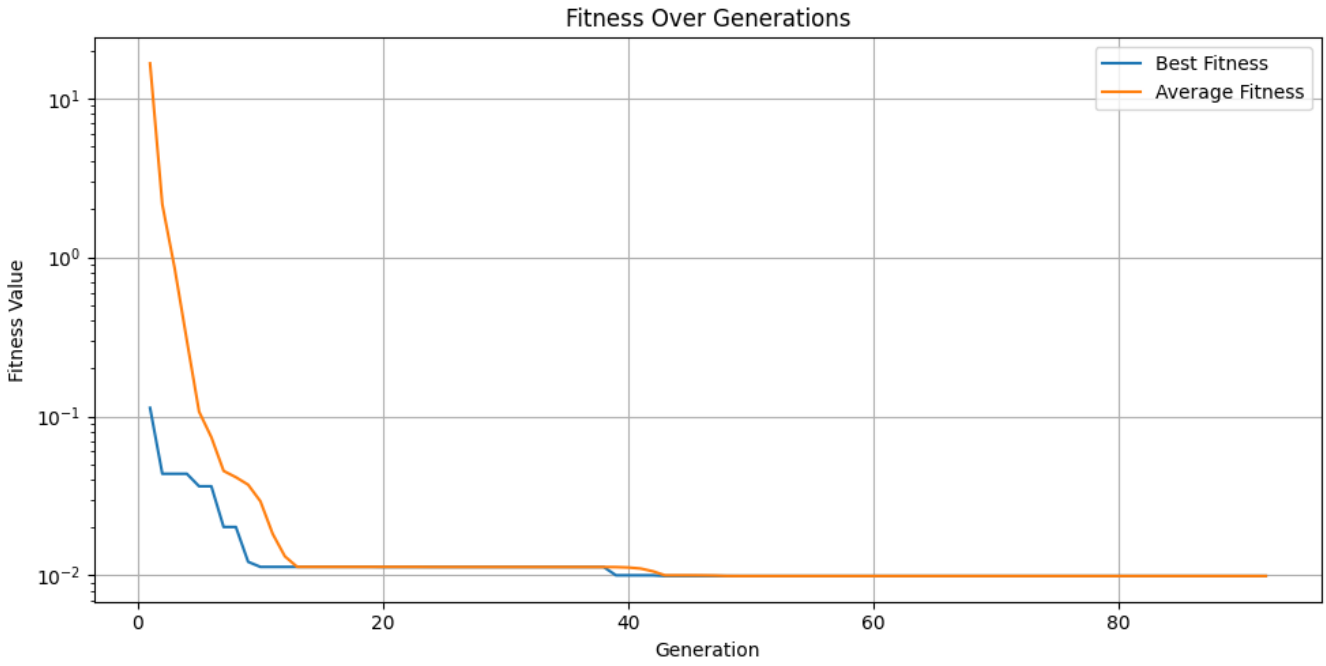
Fitness convergence, run 1.

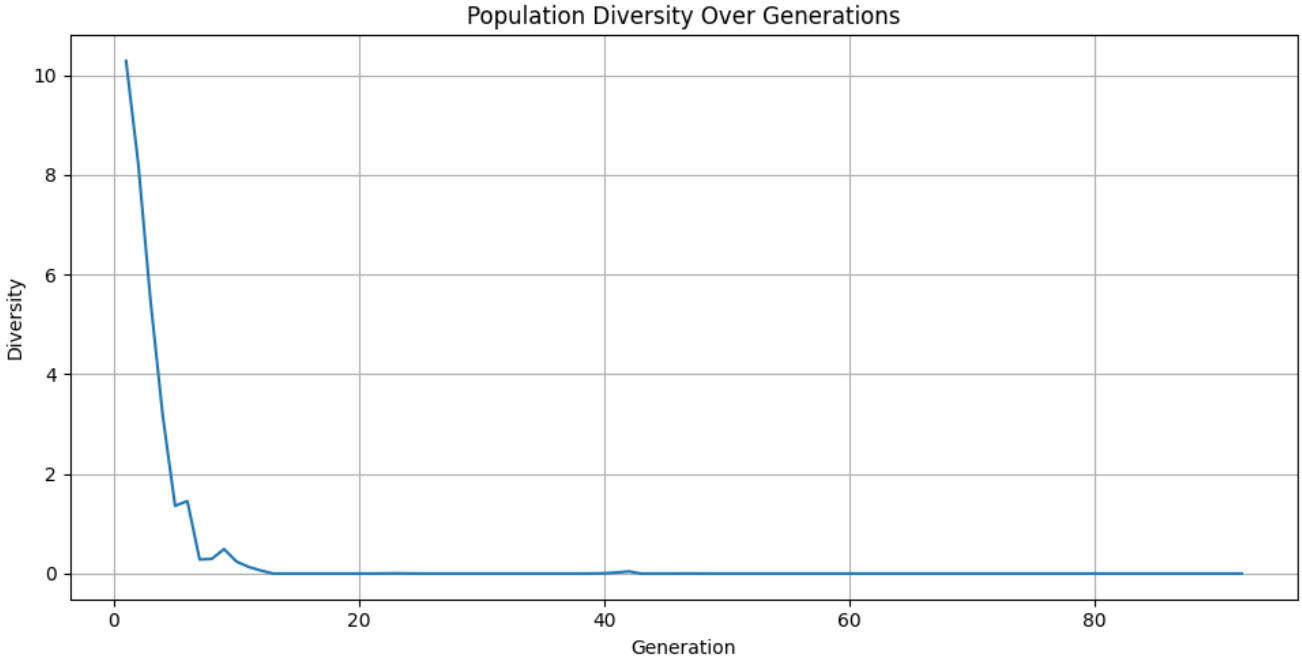

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

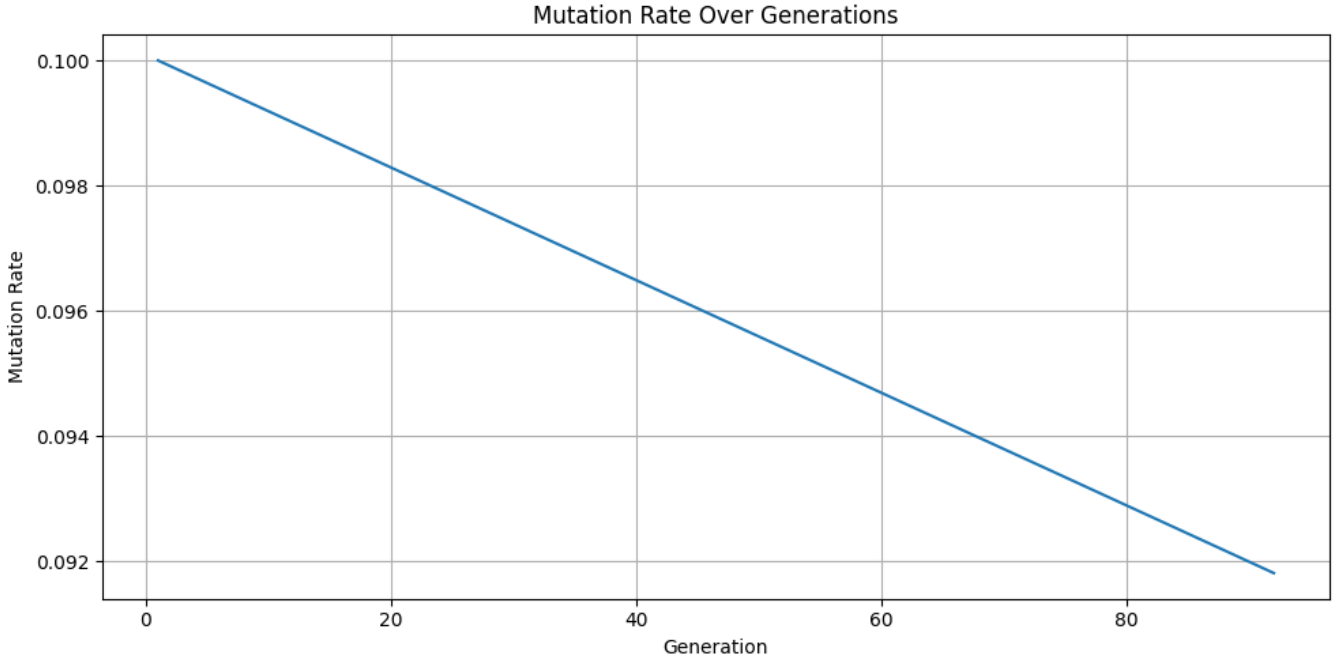

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

Fitness convergence, run 2.


Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
10^{1}
10^{0}
10^{-1}
10^{-2}
0
20
40
60
80
Generation

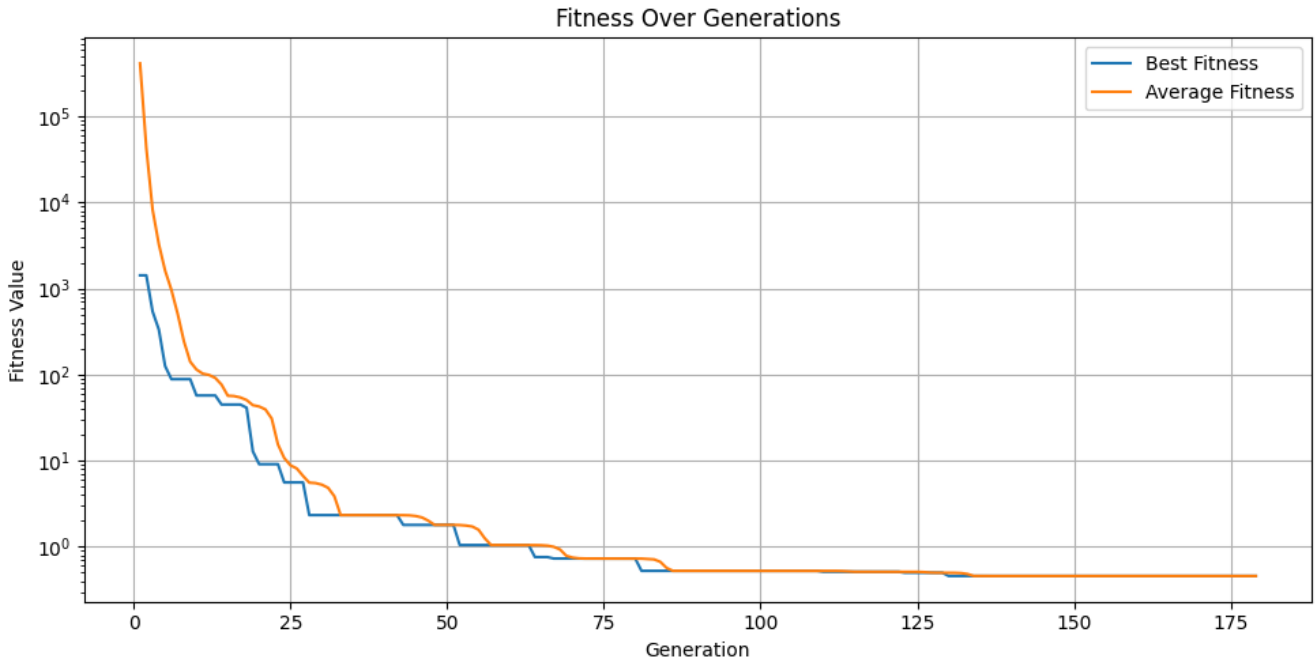

Population Diversity Over Generations
Diversity
10
8
6
4
2
0
0
20
40
60
80
Generation

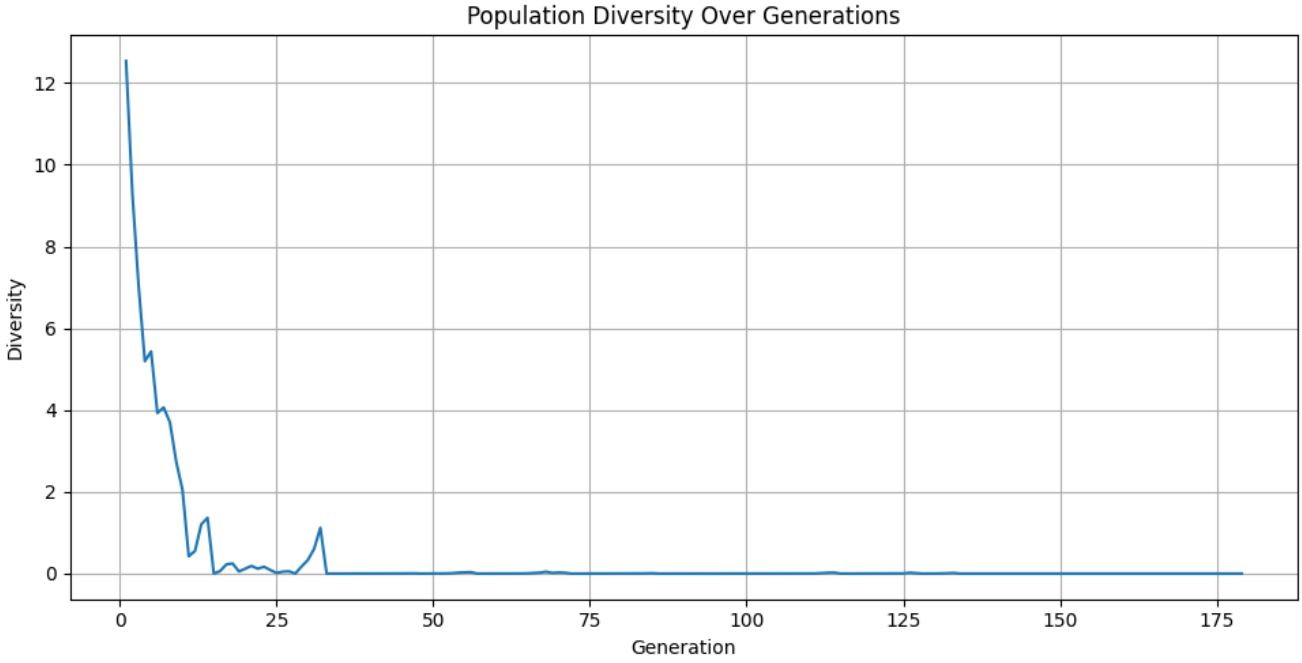

Mutation Rate Over Generations
Mutation Rate
0.100
0.098
0.096
0.094
0.092
0
20
40
60
80
Generation

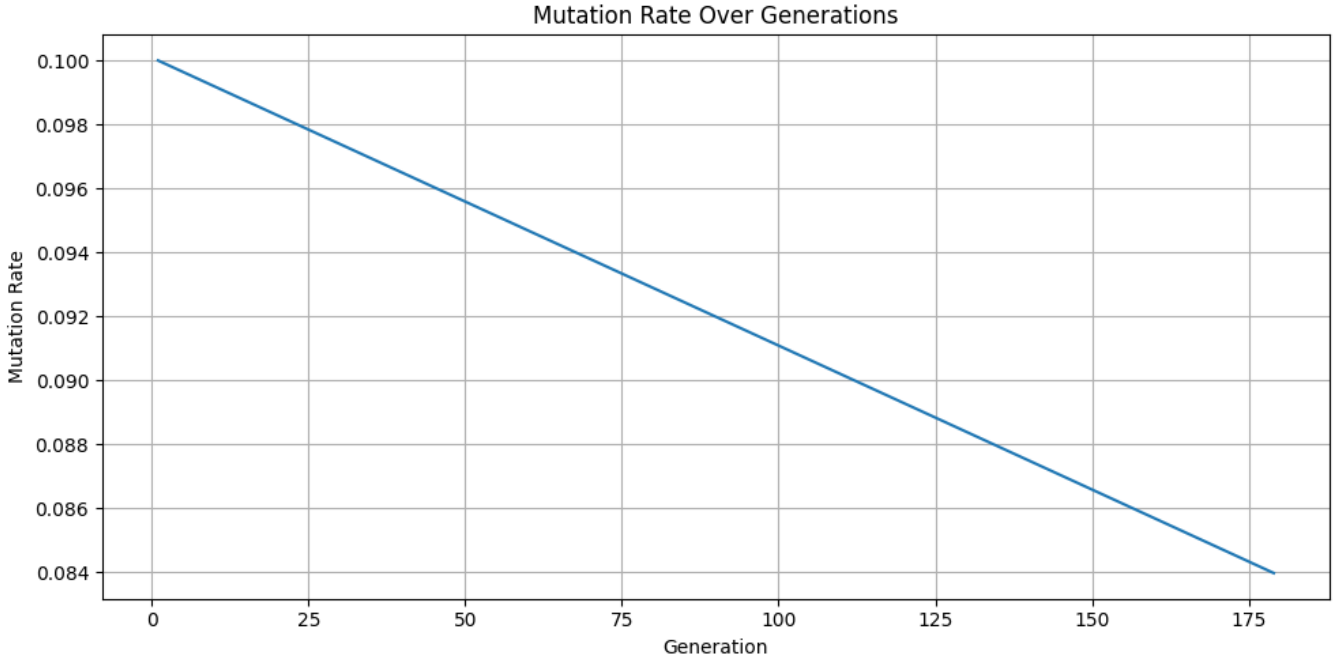
Fitness convergence, run 3.

4.3 Rosenbrock Function

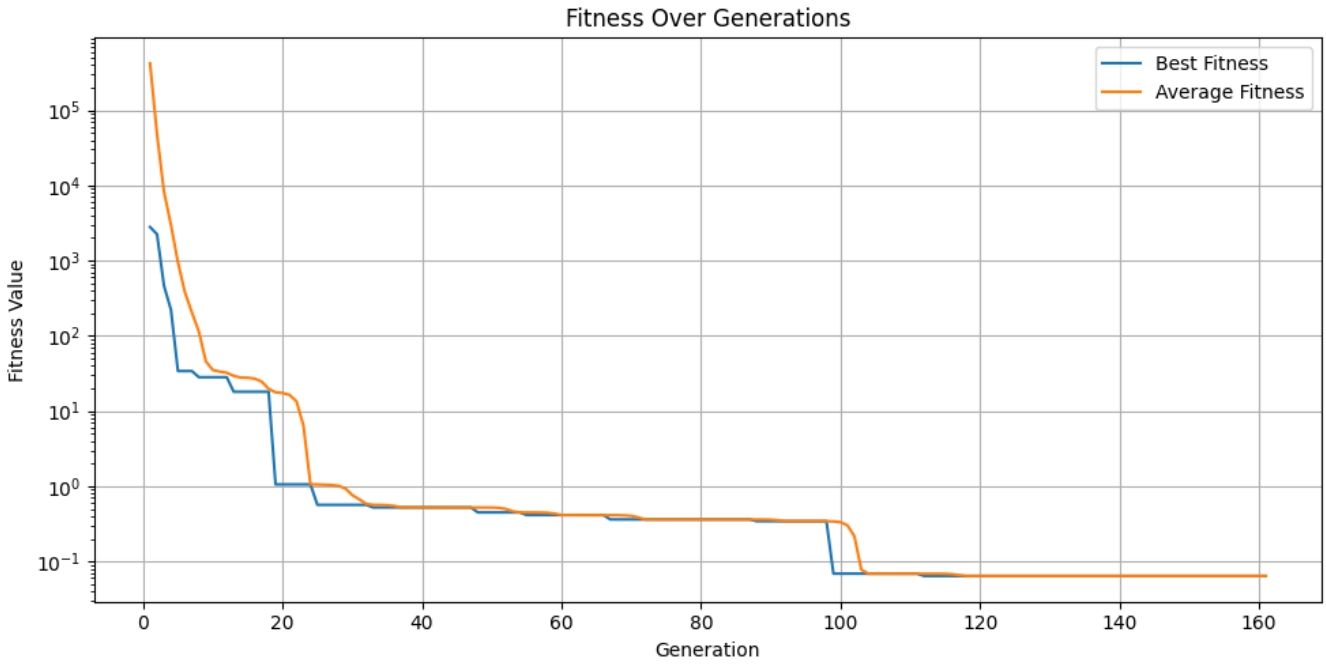
Progress Plots

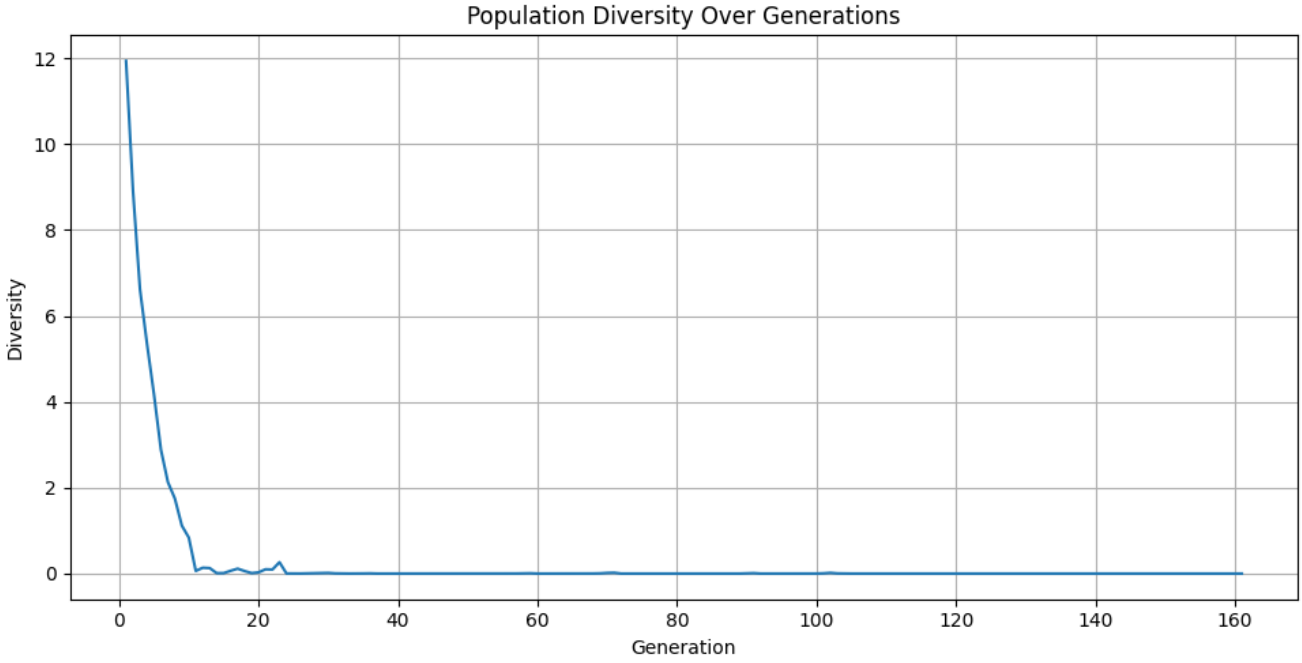

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

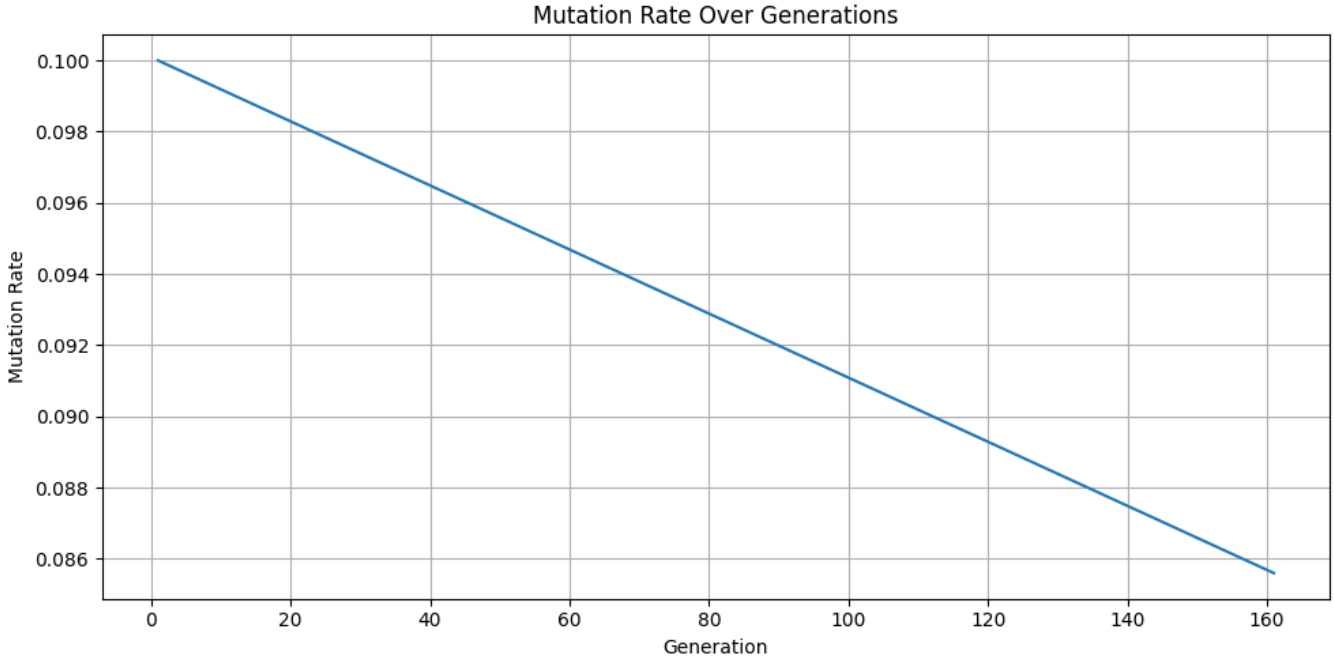

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

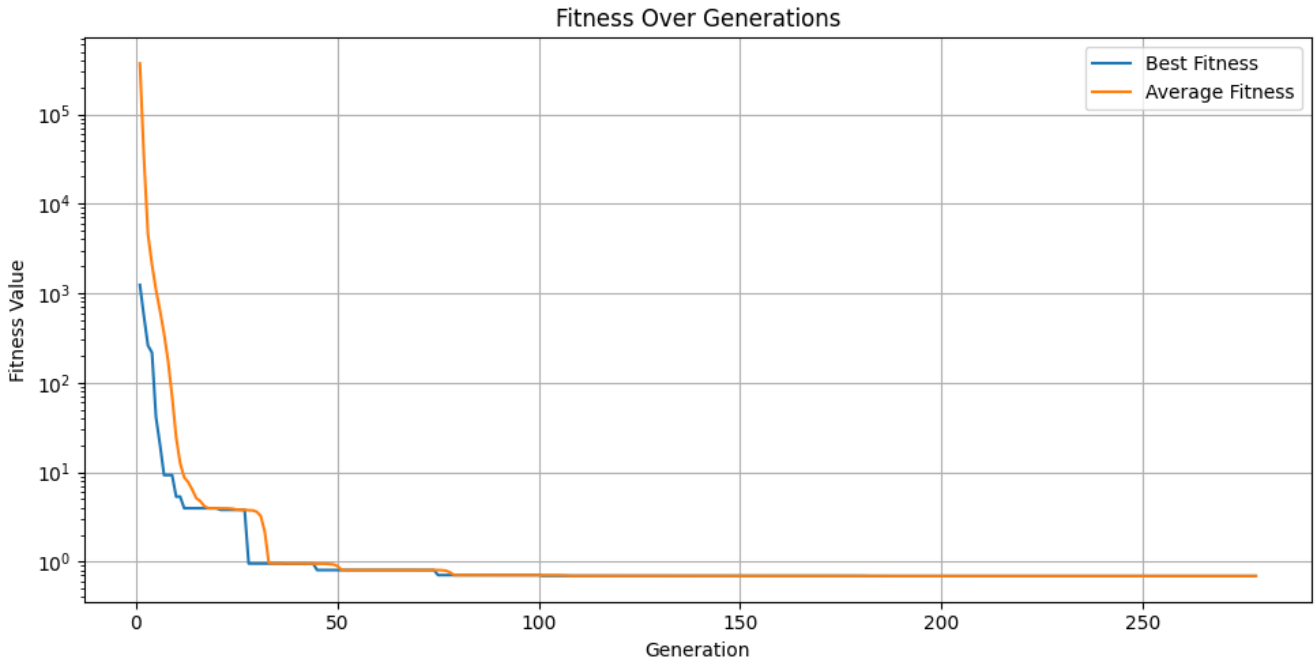
Rosenbrock, fitness over generations, run 1.

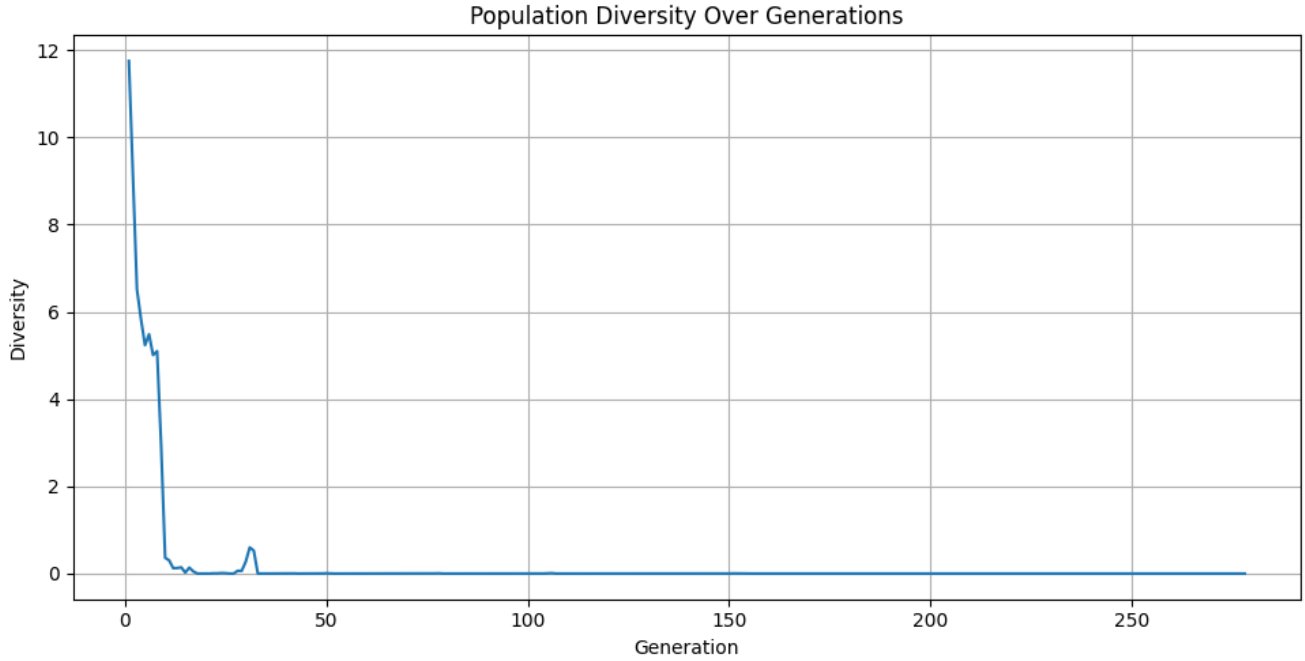

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

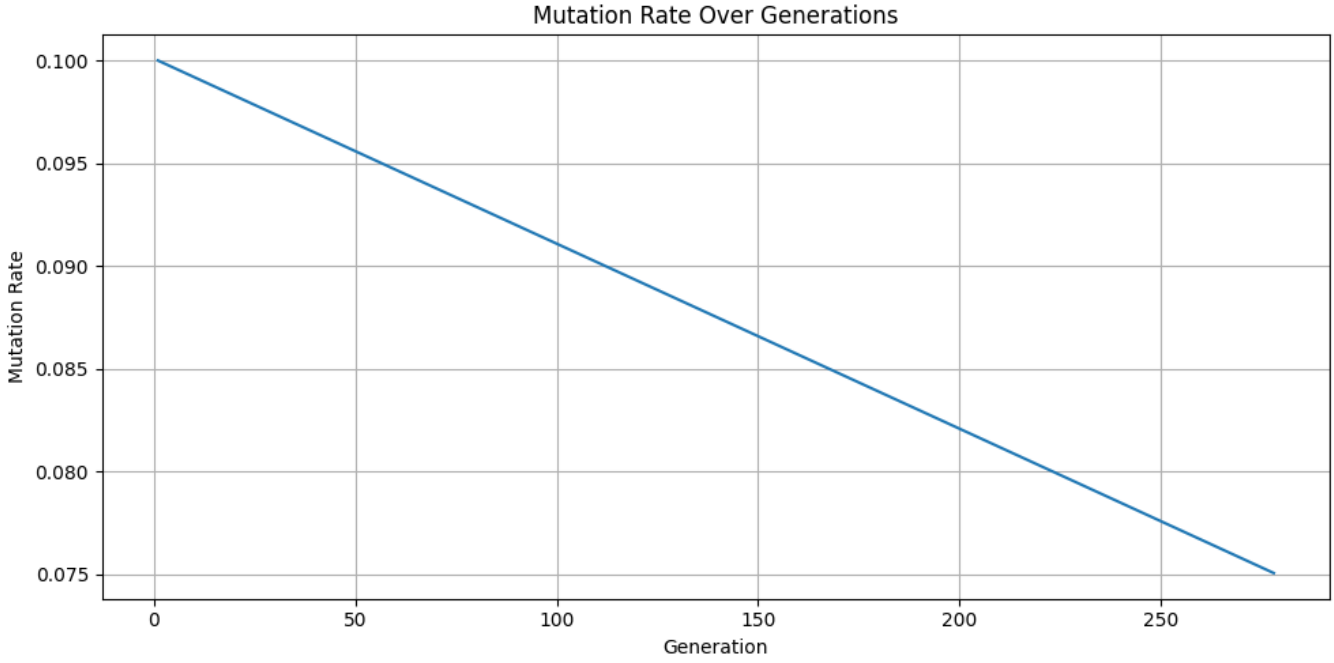

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

Rosenbrock, fitness over generations, run 2.


Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
10^5
10^4
10^3
10^2
10^1
10^0
0
50
100
150
200
250
Generation

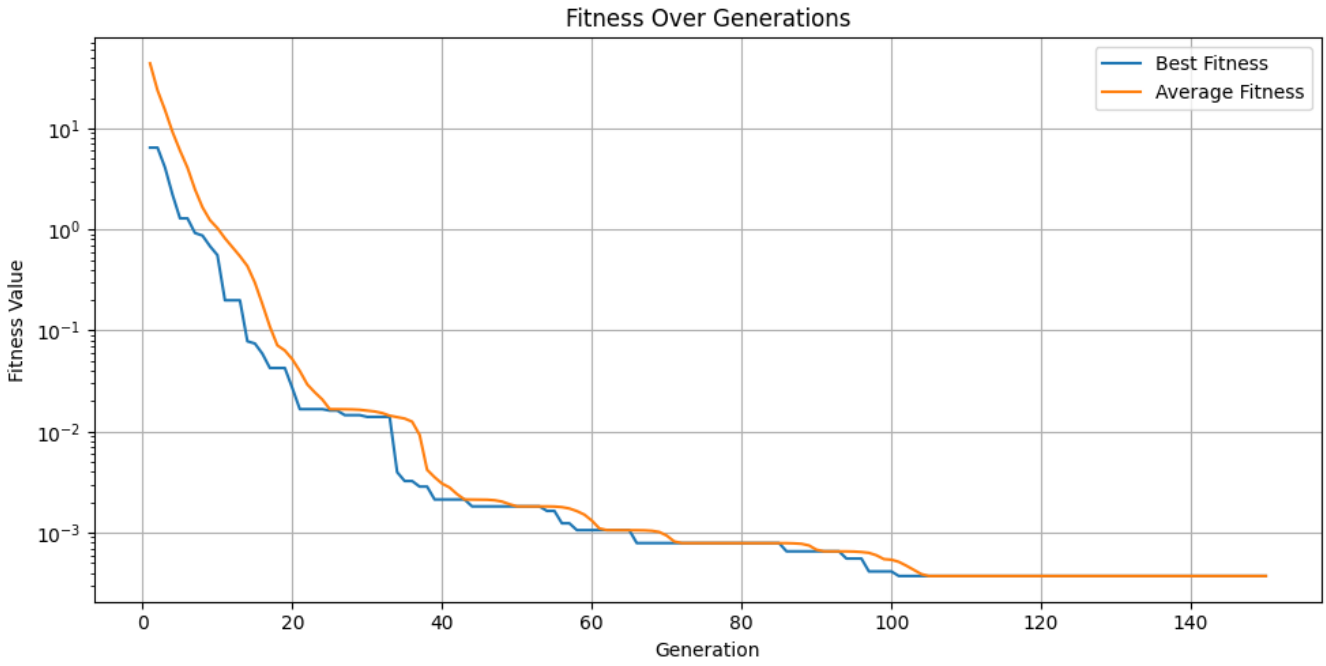

Population Diversity Over Generations
Diversity
12
10
8
6
4
2
0
0
50
100
150
200
250
Generation

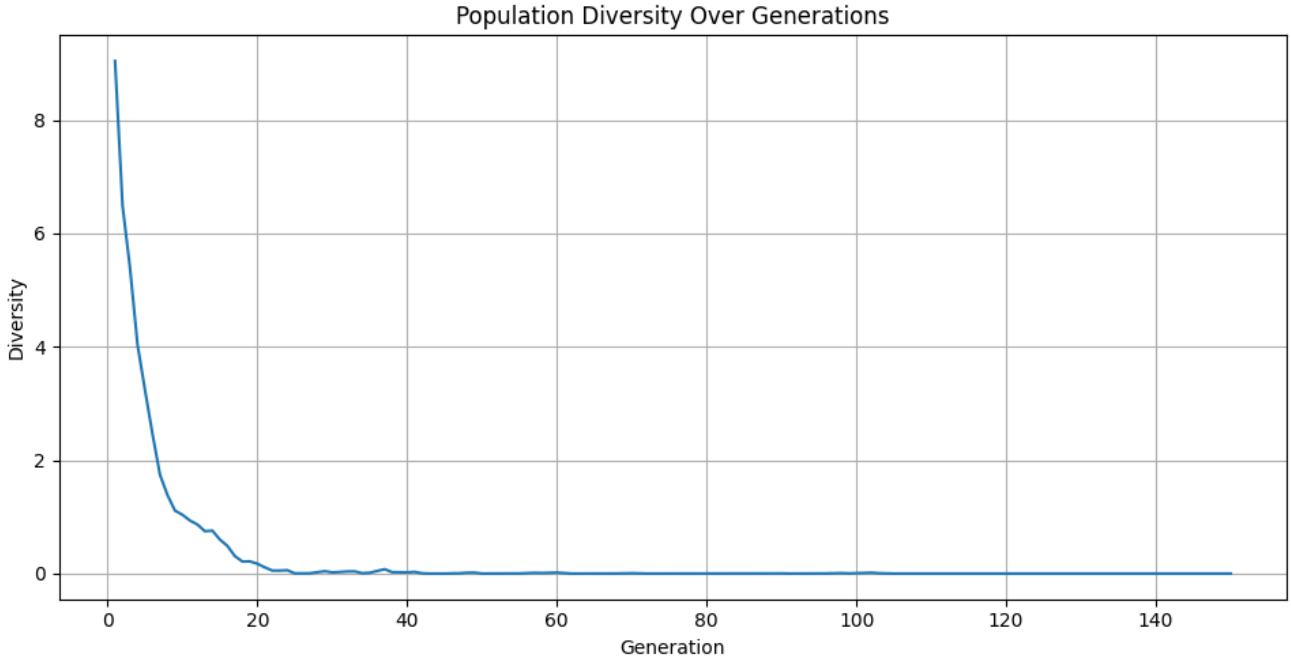

Mutation Rate Over Generations
Mutation Rate
0.100
0.095
0.090
0.085
0.080
0.075
0
50
100
150
200
250
Generation

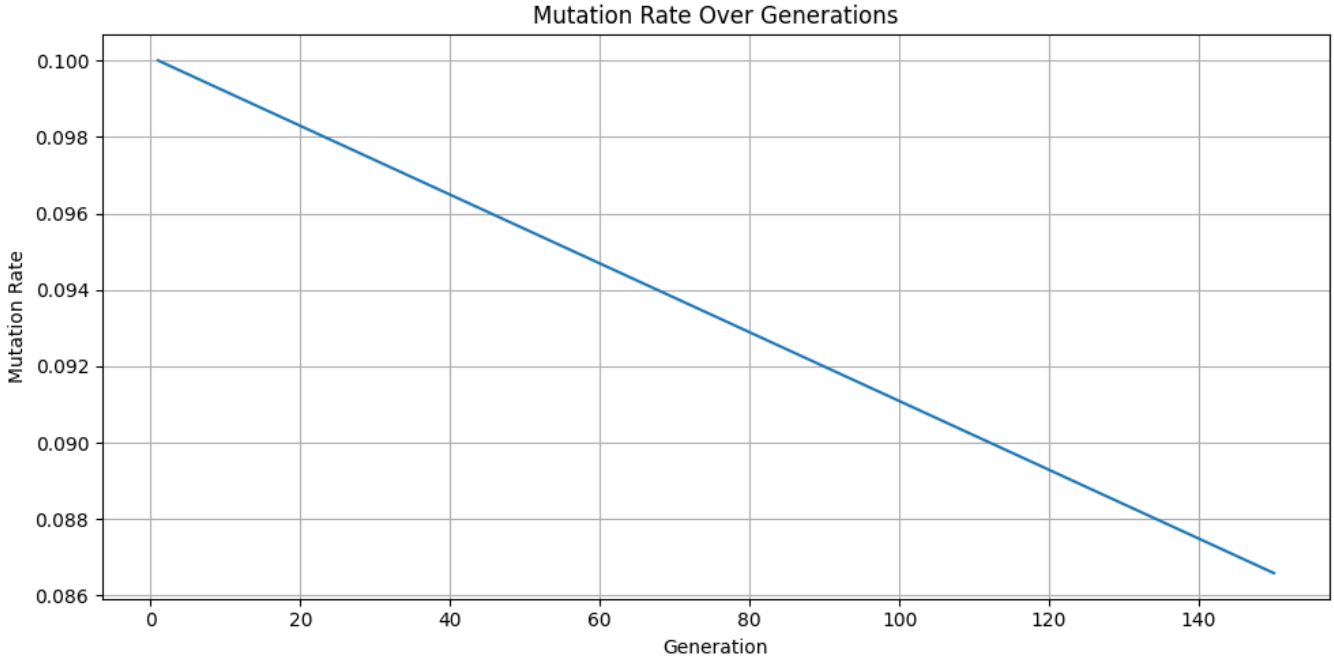
Rosenbrock, fitness over generations, run 3.

4.4 Sphere Function

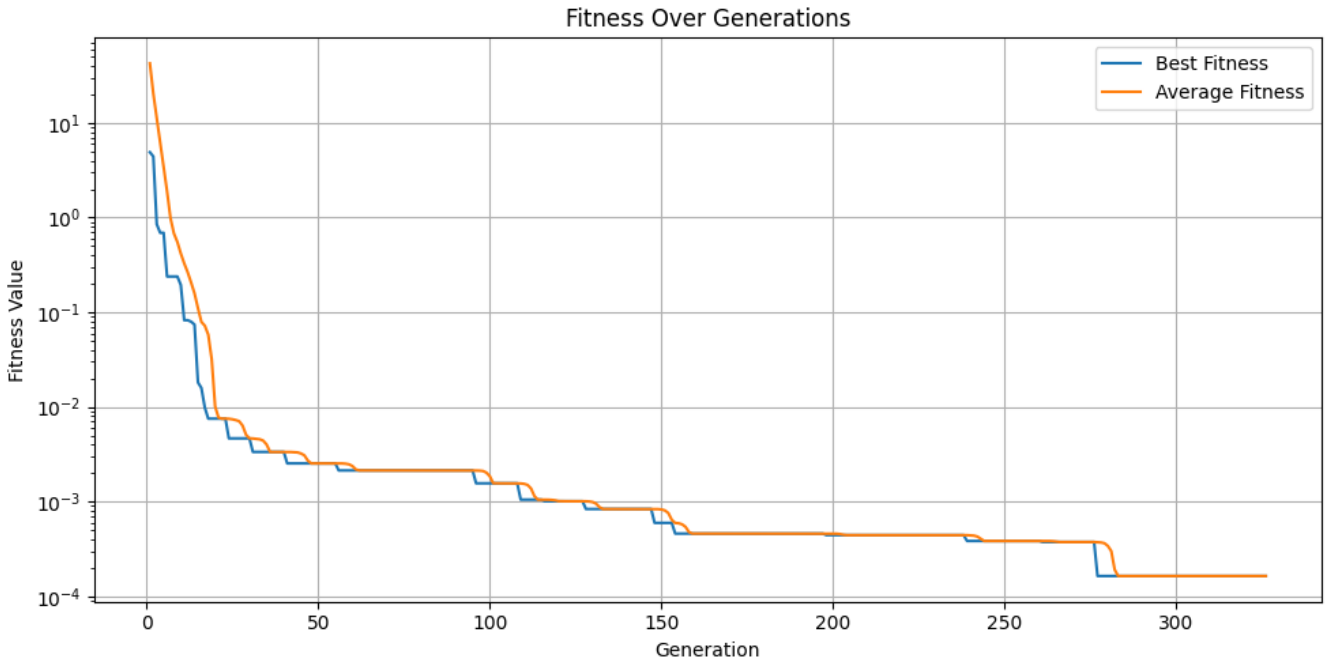
Progress Plots

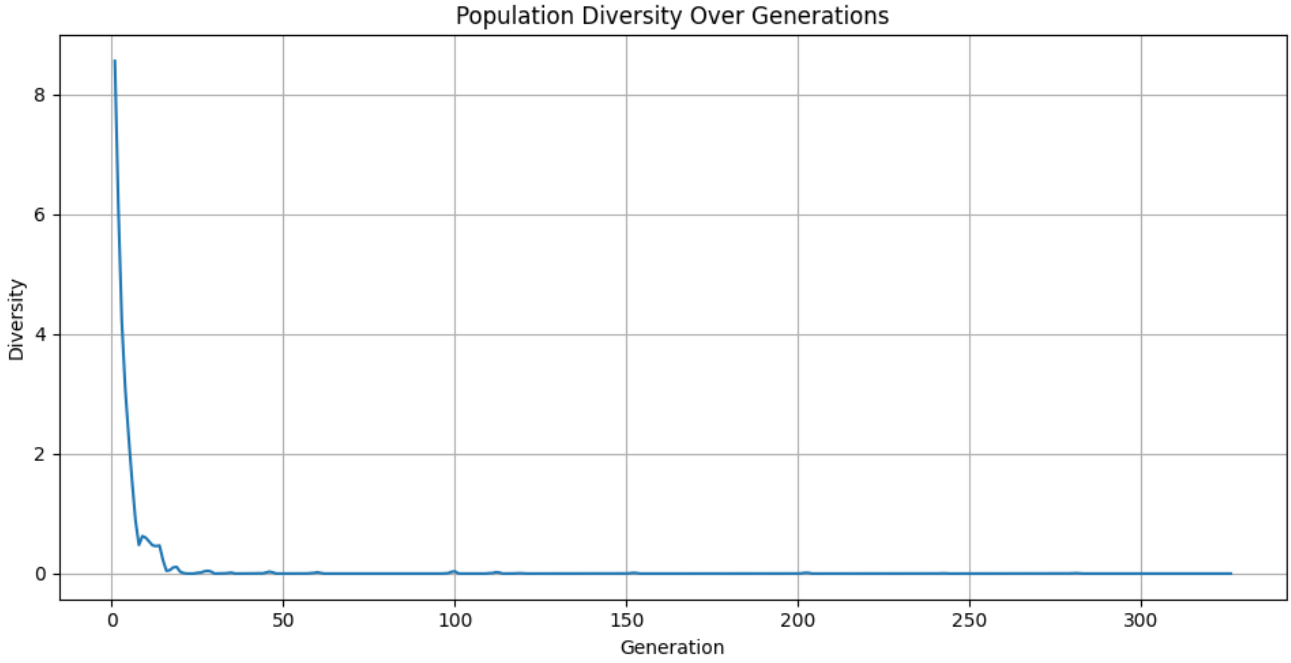

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

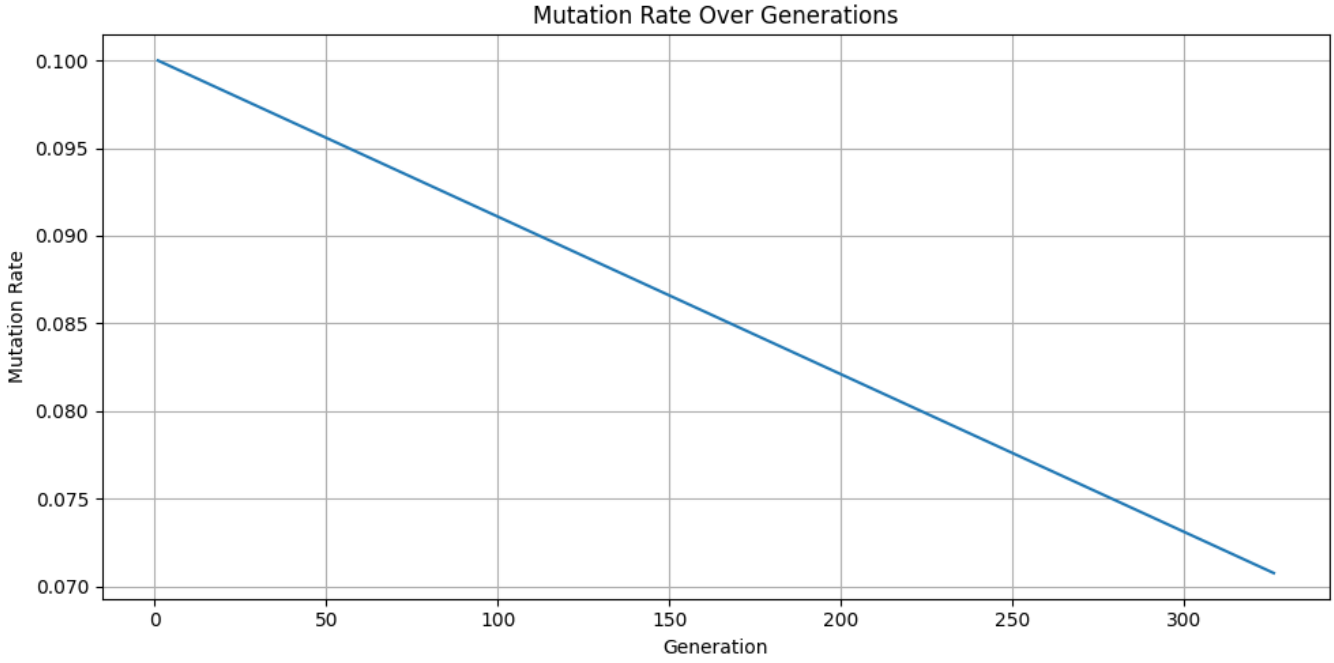

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

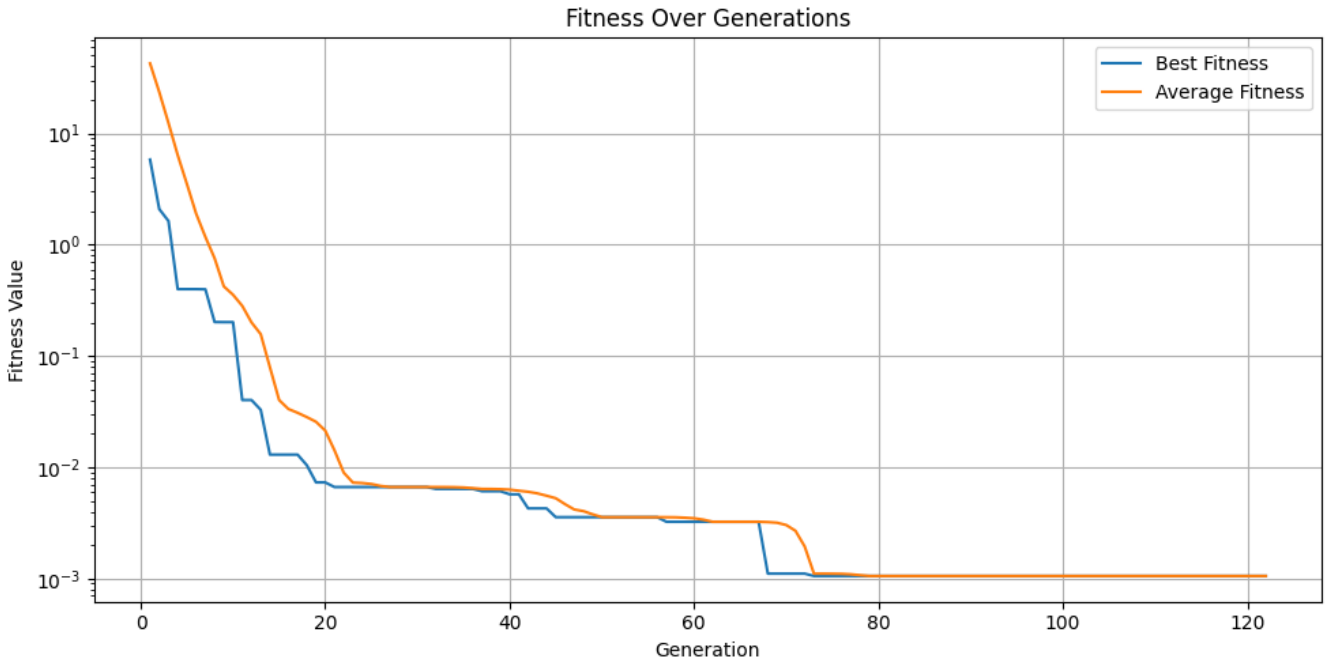
Sphere fitness vs. generations, run 1.

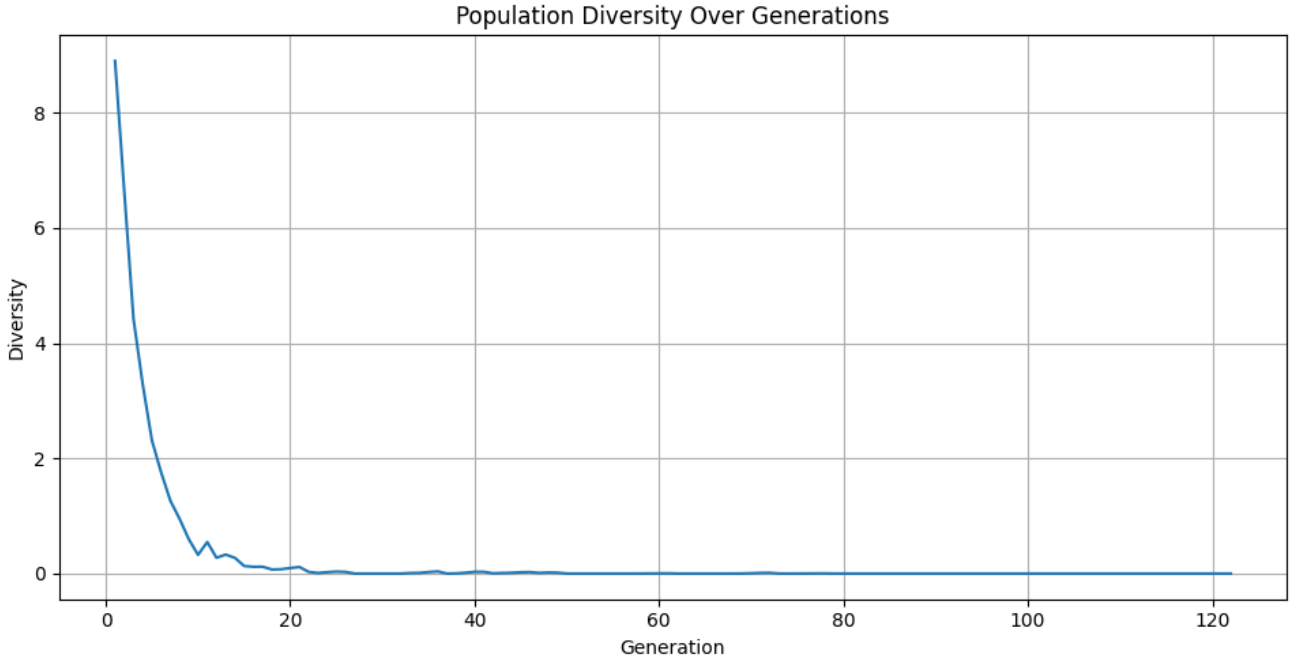

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

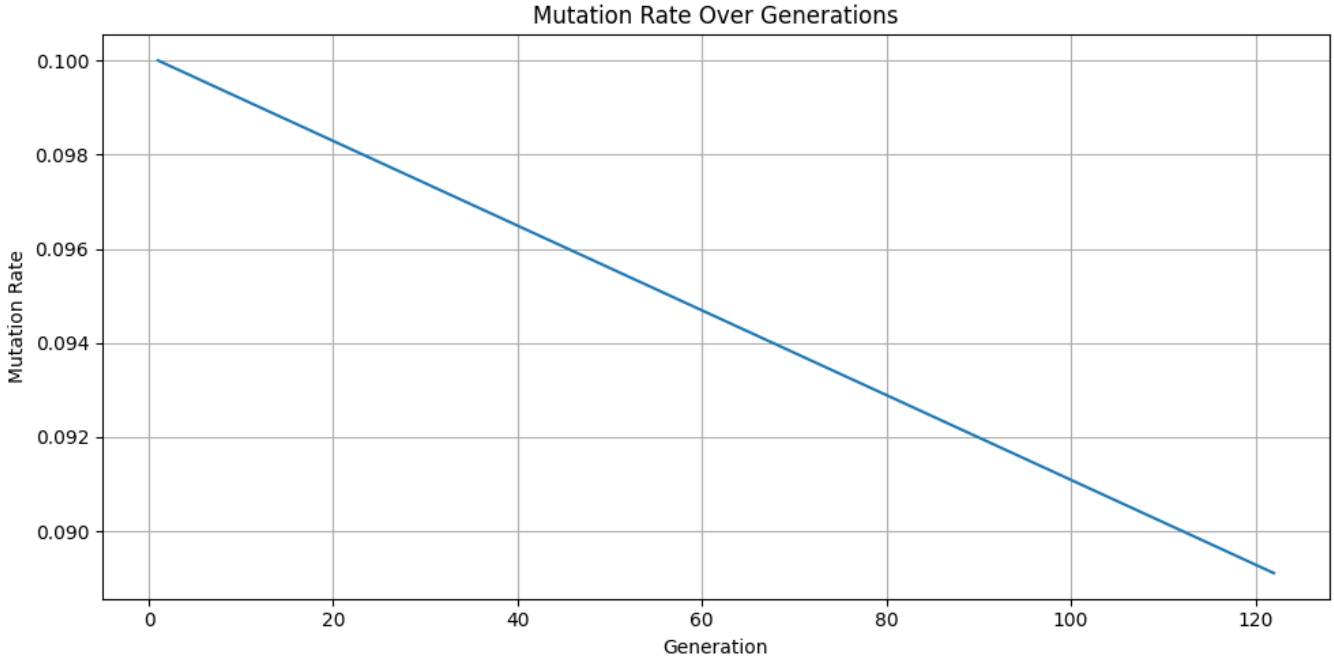

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

Sphere fitness vs. generations, run 2.


Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
0
20
40
60
80
100
120
Generation

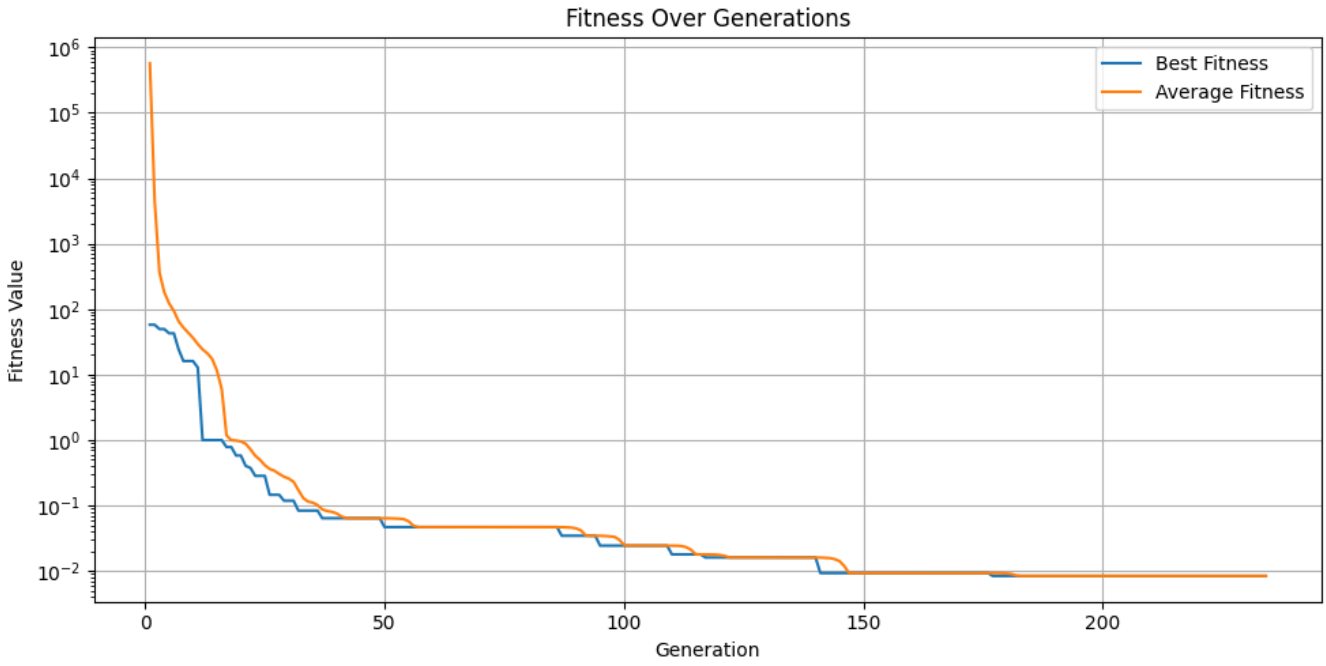

Population Diversity Over Generations
Diversity
8
6
4
2
0
0
20
40
60
80
100
120
Generation

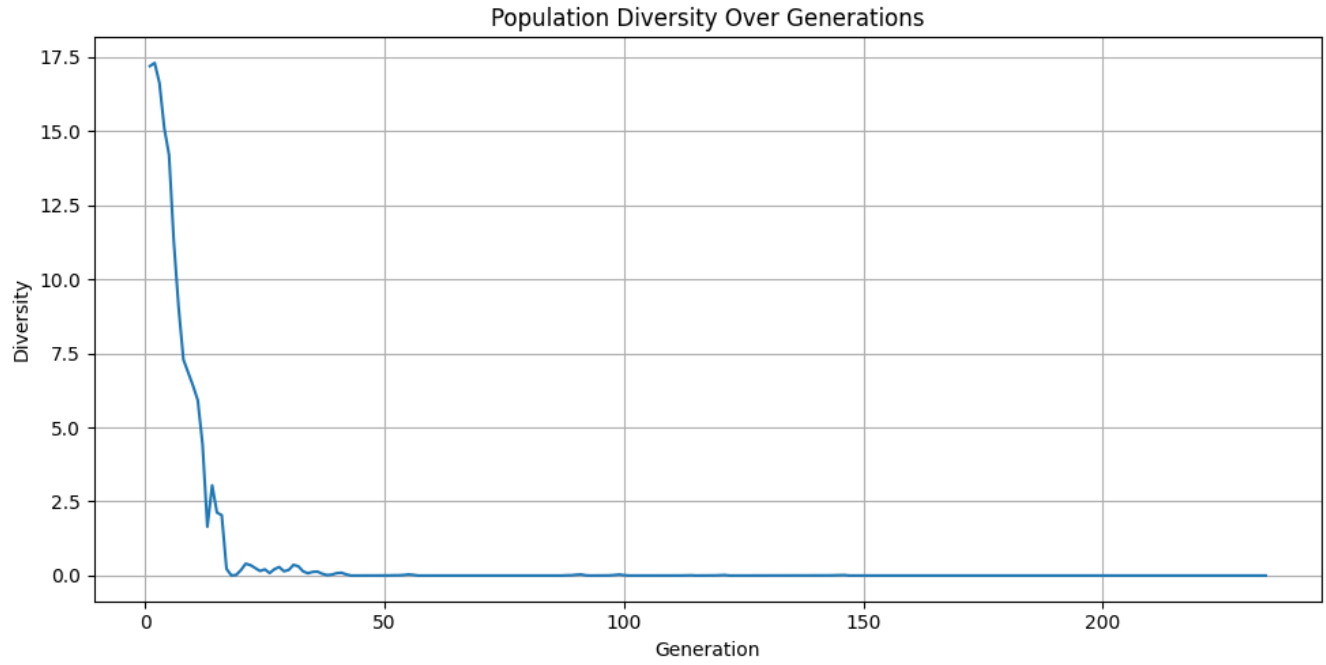

Mutation Rate Over Generations
Mutation Rate
0.100
0.098
0.096
0.094
0.092
0.090
0
20
40
60
80
100
120
Generation

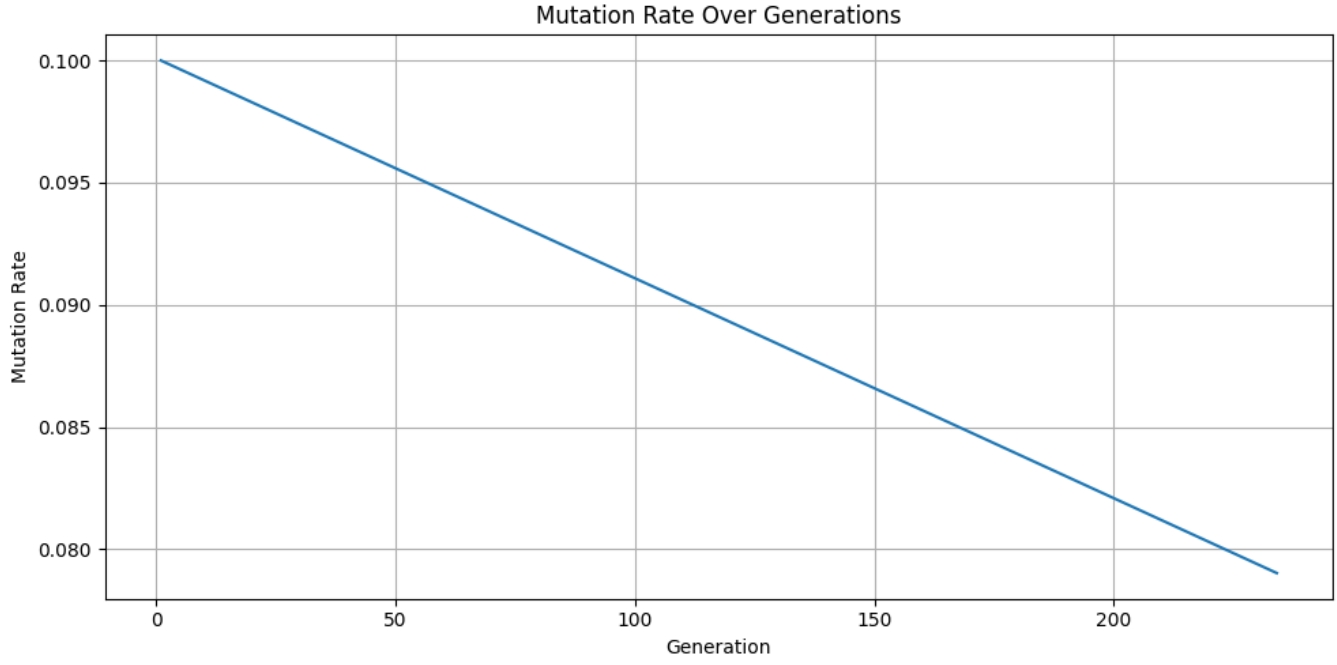
Sphere fitness vs. generations, run 3.

4.5 Zakharov Function

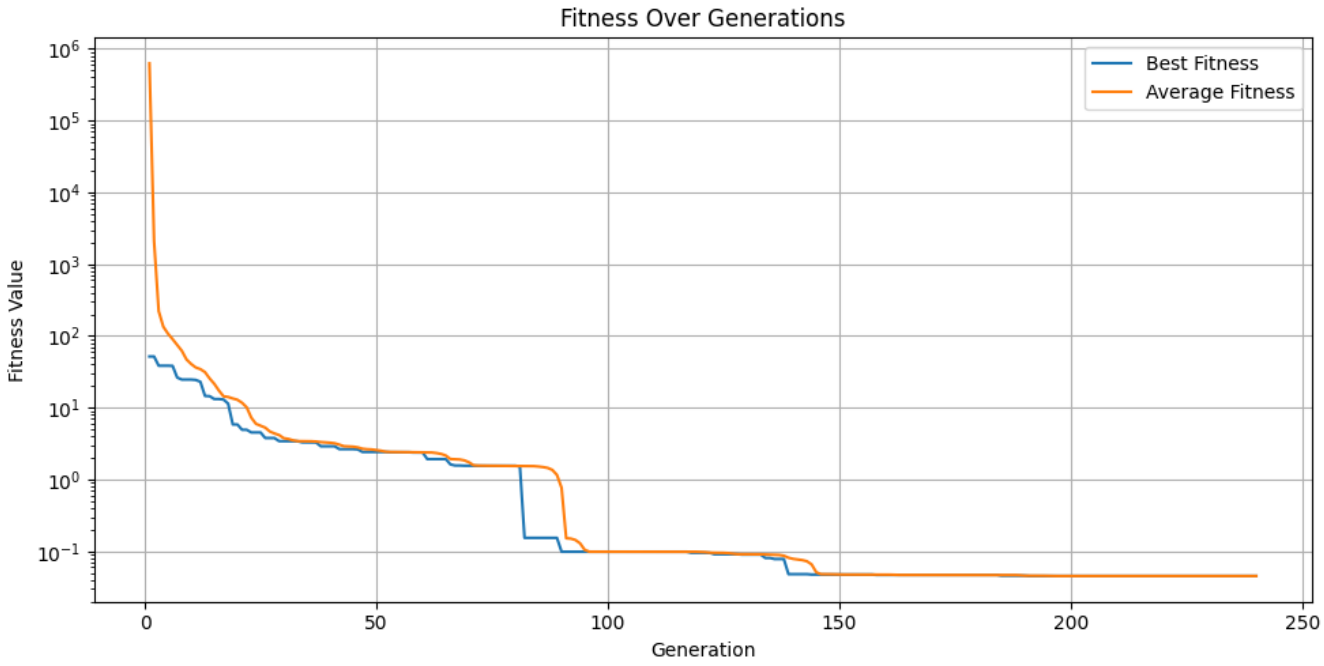
Progress Plots

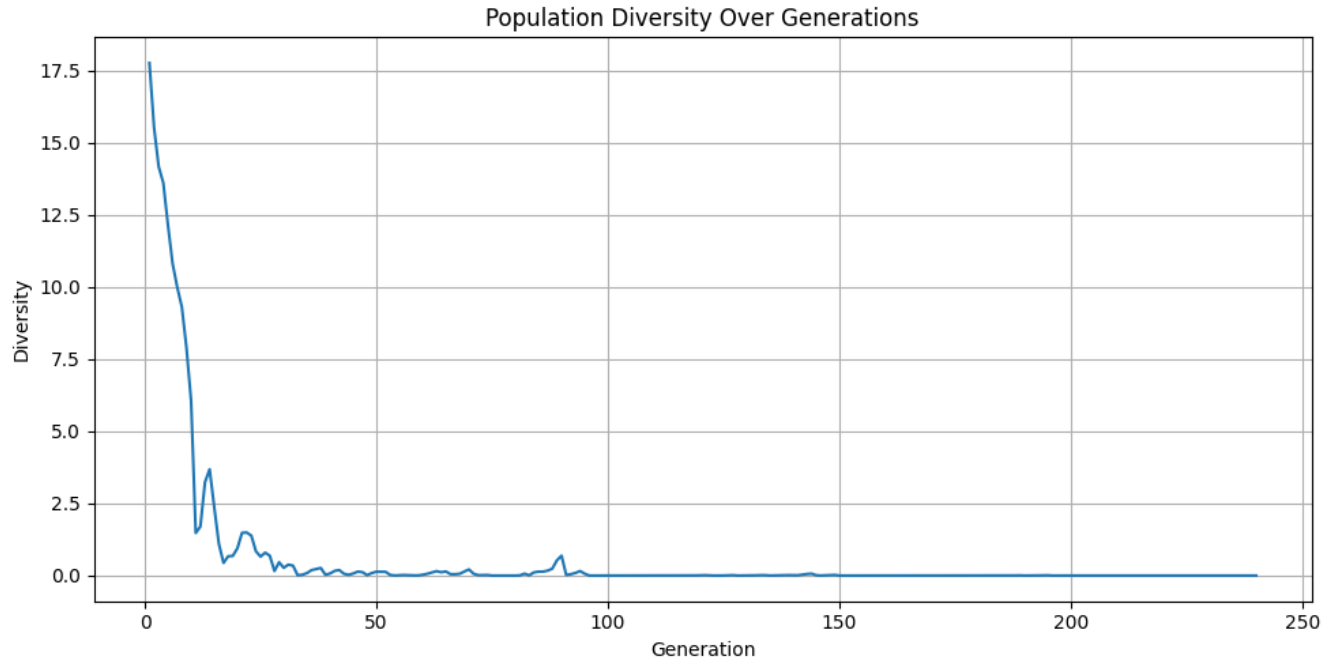

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

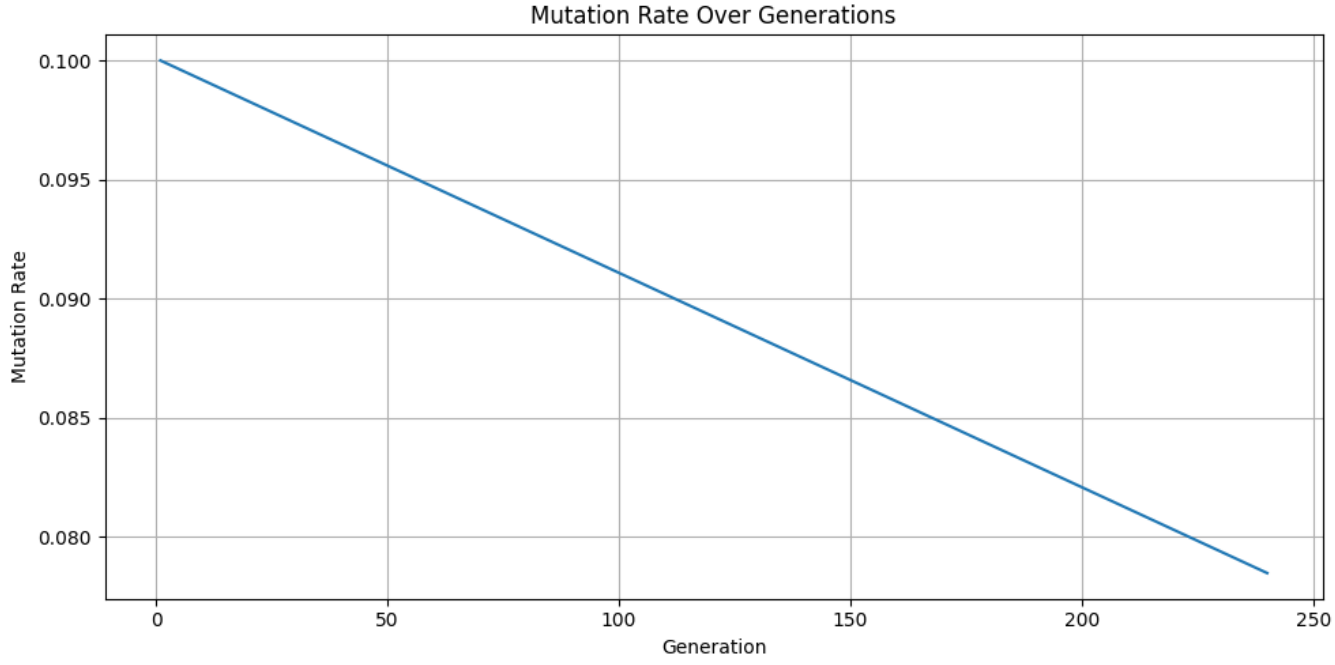

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

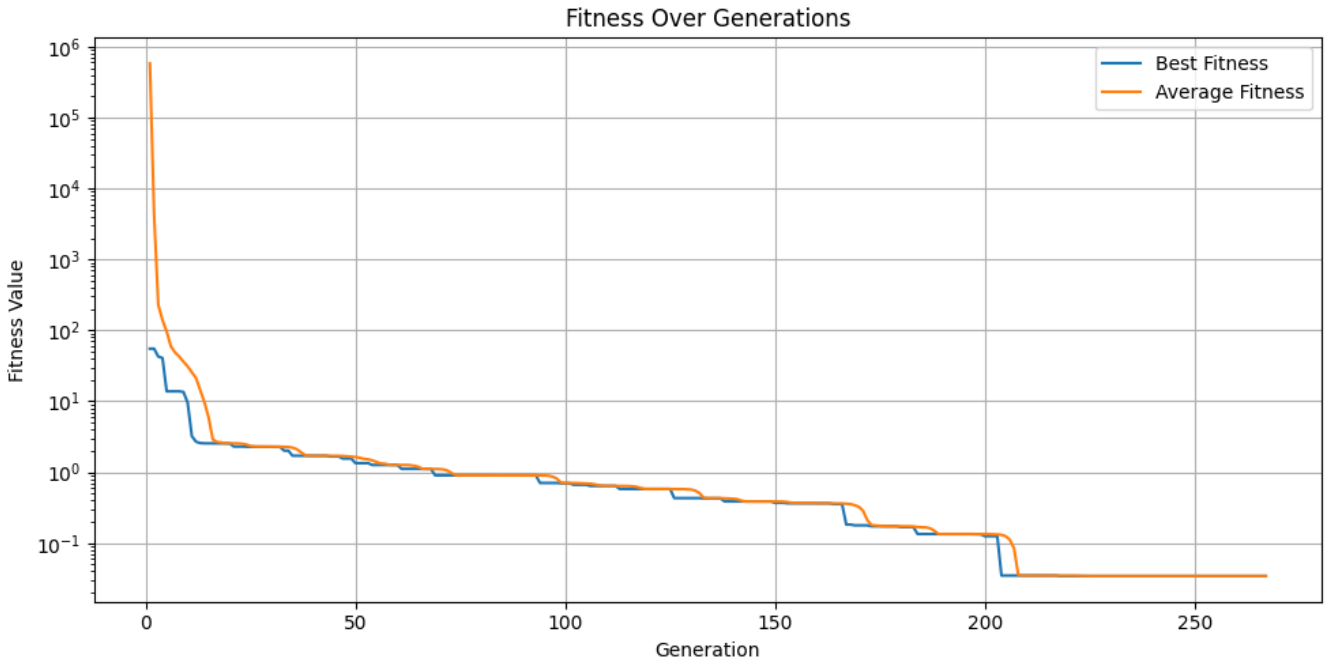
Zakharov convergence, run 1.

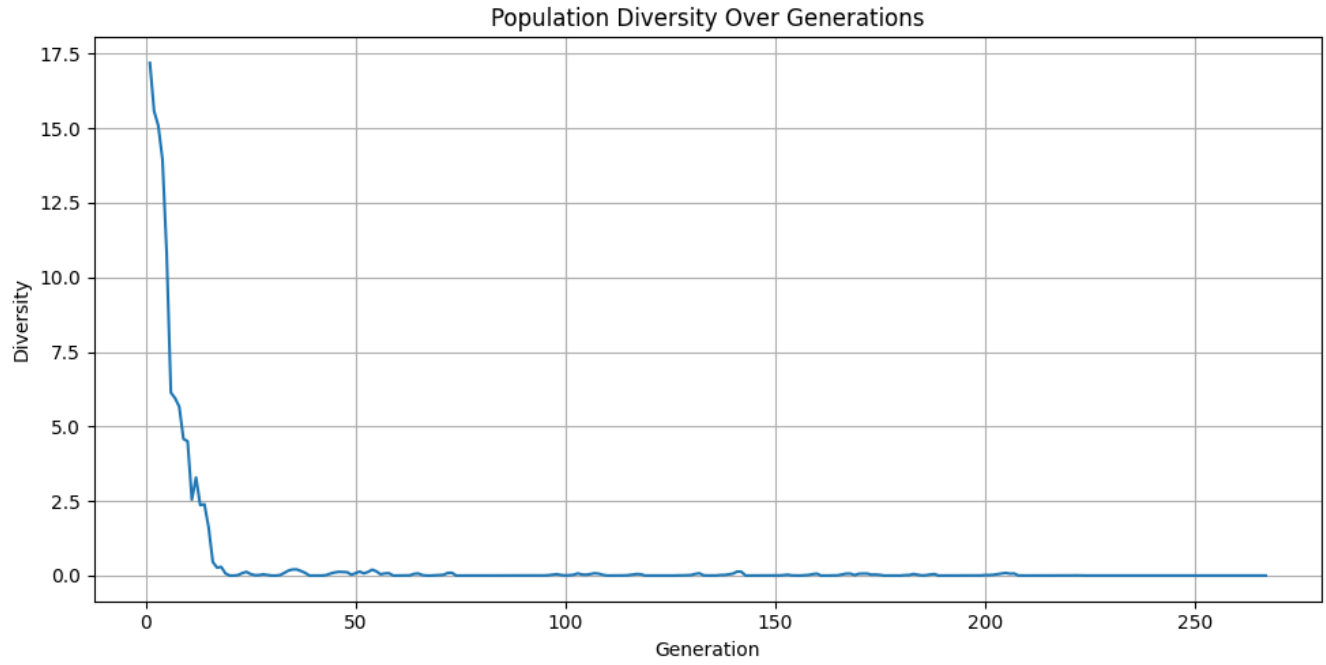

Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation

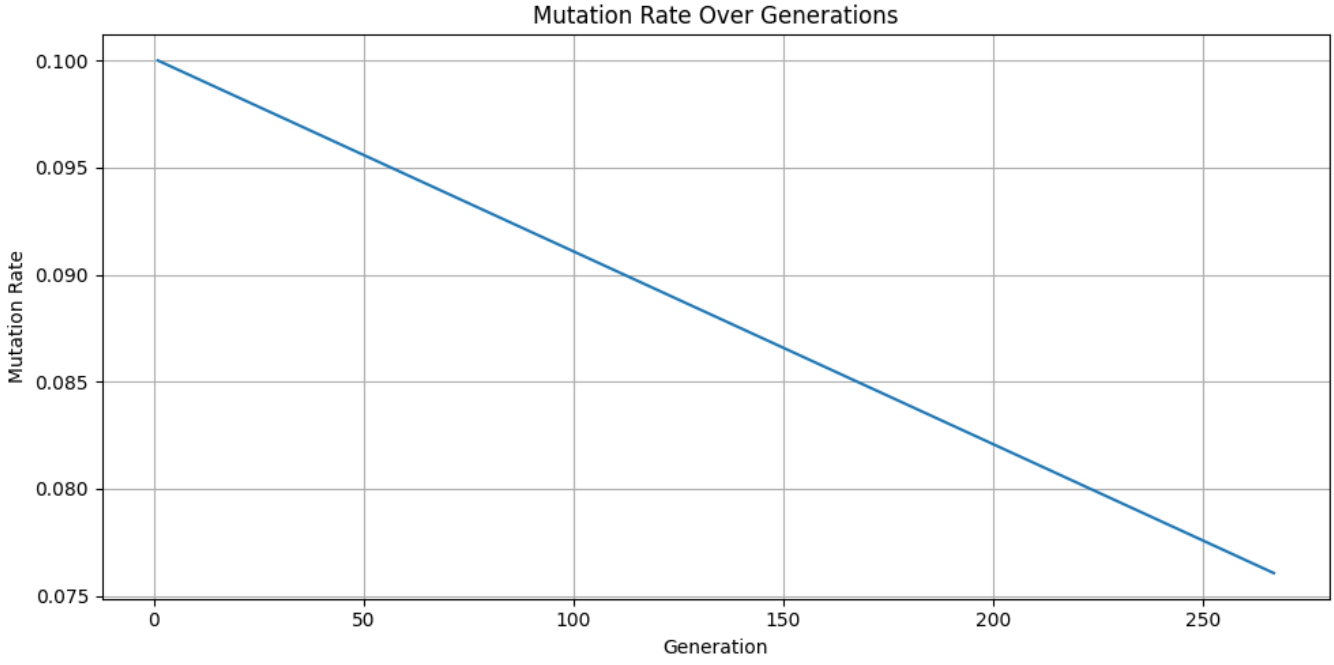

Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

Zakharov convergence, run 2.


Fitness Over Generations
Best Fitness
Average Fitness
Fitness Value
Generation


Population Diversity Over Generations
Diversity
Generation


Mutation Rate Over Generations
Mutation Rate
Generation

Zakharov convergence, run 3.

---

5. Notes

- Both **GA** and **DE** were run on each benchmark function to highlight strengths, weaknesses, and convergence profiles.
- Parameters (population size, mutation rate, etc.) were selected to ensure fair comparison.
- Any missing figures for DE can be added in future versions or represented similarly to those of GA.

Resources

- Full Project Code and Data (Google Drive)
- GA Concepts (Wikipedia)
- Differential Evolution (Wikipedia)
- Understanding Benchmark Functions
- Original Genetic Algorithm Paper (J. Holland, 1975, MIT Press)
- Youtube playlist by Dr.amr
