Simple Genetic Algorithm and Applications

CP 468 Term Project

Group 8

December 6, 2024

Team Members:

Romin Gandhi, Jenish Bharucha, Nakul Patel, Arsh Patel, Dhairya Patel
Paarth Bagga, Devarth Trivedi, Gleb Silin, Emmet Currie, Parker Riches

1. Overview

This term project focuses on implementing a **Simple Genetic Algorithm (SGA)** to solve optimization problems by using specific objective functions and with the additional functionality of implementing an objective function of your choice. Three well-known objective functions were chosen to evaluate the performance of the Simple Genetic Algorithm, followed by 2 custom functions provided by Dr. Ilias S. Kotsireas, Professor.

- **Sphere Function:** $f(x) = \sum_{i=1}^{n} x_i^2$, a straightforward uni modal function used to assess how well the SGA can converge towards the global minimum of [0, 0].
- **Rosenbrock Function:** $f(x) = \sum_{i=1}^{n-1}[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$, a non-convex function used to test the SGA's ability of handling difficult and non-straightforward optimization problems. The functions global minimum is [1,1].
- **Himmelblau Function:** $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$, a multimodal function with several local minima, providing a challenging test for the SGA's capabilities. The function has four global minima at $[3.0, 2.0]$, $[-2.805118, 3.131312]$, $[-3.779310, -3.283186]$, and $[3.584428, -1.848126]$.

Key features that we have implemented in SGA include adjustable parameters (e.g., population size, mutation rate, and number of generations) and the ability to visualize convergence through fitness plots using Python's Matplotlib library. Additionally, a Graphical User Interface (GUI) was developed using Python's Tkinter library, allowing for an smooth function selection and parameter configuration.

2. Key Data Structures in Genetic Algorithm Design

The Simple Genetic Algorithm (GA) has the following data structures implemented:

1. **Population Initialization:** Generating solutions randomly within defined limits.

```python
# Create population with random variables within bounds
population = np.random.uniform(
    [b[0] for b in bounds],  # Lower bounds
    [b[1] for b in bounds],  # Upper bounds
    (population_size, dim)   # Shape of population
)
```

Listing 1: Population Initialization Code

2. **Fitness Evaluation:** Using the objective functions to assess solution quality.

```python
# Obtain fitness of every individual
fitness = np.array([objective_function(ind) for ind in population])
```

Listing 2: Fitness Evaluation Code

3. **Selection:** Utilizing a biased roulette wheel approach to favor fitter individuals.

```
# Calculate probabilities for biased wheel
total_fitness = np.sum(fitness)
selection_probs = (1 / (fitness + 1e-6)) / total_fitness

# Normalize (ensure probabilities add to 1)
selection_probs /= np.sum(selection_probs)

# Select mating pool based on biased wheel
selected = population[np.random.choice(len(population), size=
    population_size, p=selection_probs)]
```

Listing 3: Selection Code

4. **Crossover:** Combining genetic material from two parents to create child.

```
# Create offspring using crossover
offspring = []
for _ in range((population_size - num_elites) // 2):
    # Randomly select 2 parents from mating pool
    p1, p2 = selected[np.random.choice(len(selected), size=2)]

    # Create crossover point and make children based on parents and
        crossover point
    crossover_point = np.random.randint(1, dim)
    child1 = np.concatenate((p1[:crossover_point], p2[
        crossover_point:]))
    child2 = np.concatenate((p2[:crossover_point], p1[
        crossover_point:]))

    # Add children to list
    offspring.append(child1)
    offspring.append(child2)
```

Listing 4: Crossover Code

5. **Mutation:** Introducing random genetic variation to maintain diversity.

```
# Chance to mutate a child
if np.random.rand() < mutation_rate:
    child1 += np.random.uniform(-0.1, 0.1, dim)
if np.random.rand() < mutation_rate:
    child2 += np.random.uniform(-0.1, 0.1, dim)
```

Listing 5: Mutation Code

6. **Elitism:** Preserving the best solutions across generations.

```
# Keep track of elites in the generation
elite_indices = np.argsort(fitness)[:num_elites]
elite_individuals = population[elite_indices]

# Replace old population with elites and children
population = np.vstack((elite_individuals, np.array(offspring)))
```

Listing 6: Elitism Code

7. **Exploration vs. Exploitation:**

   - **Exploration:** Ensuring global search through mutation to introduce diversity.

```python
# Mutation introduces diversity by adding random noise to offspring
if np.random.rand() < mutation_rate:
    child1 += np.random.uniform(-0.1, 0.1, dim)
if np.random.rand() < mutation_rate:
    child2 += np.random.uniform(-0.1, 0.1, dim)
```

Listing 7: Exploration Code

   - **Exploitation:** Refining promising solutions through selection and elitism.

```python
# Selection biases towards fitter individuals
selection_probs = (1 / (fitness + 1e-6)) / total_fitness
selection_probs /= np.sum(selection_probs)
selected = population[np.random.choice(len(population), size=population_size, p=selection_probs)]

# Elitism: Retain the best-performing individuals
elite_indices = np.argsort(fitness)[:num_elites]
elite_individuals = population[elite_indices]
population = np.vstack((elite_individuals, np.array(offspring)))
```

Listing 8: Exploitation Code

8. **Convergence:** Ensuring the algorithm reaches a solution over generations.

```python
# Convergence is determined by running the algorithm for a fixed number of generations
for generation in range(generations):
    # Evaluate fitness for the current population
    fitness = np.array([objective_function(ind) for ind in population])

    # Track the best fitness in the current generation
    best_individual_gen = population[np.argmin([objective_function(ind) for ind in population])]
    best_fitness_gen = objective_function(best_individual_gen)
    best_fitness_per_generation.append(best_fitness_gen)

    # Print generation progress
    print(f"Generation {generation + 1} - Best Solution: {best_individual_gen}, Fitness: {best_fitness_gen}")

# Obtain the final best solution and fitness score after all generations
best_individual = population[np.argmin([objective_function(ind) for ind in population])]
best_fitness = objective_function(best_individual)
print("\nFinal Best Solution:", best_individual)
print("Final Fitness Score:", best_fitness)

# Plot the fitness over generations to visualize convergence
plt.plot(range(1, generations + 1), best_fitness_per_generation)
```

```
plt.xlabel("Generation")
plt.ylabel("Best Fitness")
plt.title("Best Fitness Over Generations")
plt.grid(True)
plt.show()
```

Listing 9: Convergence Code

3. Visualization and User Interface

The implementation of the Simple Genetic Algorithm (SGA) includes both visualization of the best fitness over number of generation and a Graphical User Interface (GUI) to increase usability and provide insights into the algorithm's performance.

3.1 Visualization

To visualize the algorithm's performance, a fitness plot is generated using Matplotlib. The plot tracks the best fitness value across generations, providing a clear representation of the algorithm's convergence.

```
# Plot the fitness over generations
plt.plot(range(1, generations + 1), best_fitness_per_generation)
plt.xlabel("Generation")
plt.ylabel("Best Fitness")
plt.title("Best Fitness Over Generations")
plt.grid(True)
plt.show()
```

Listing 10: Fitness Plot Visualization Code

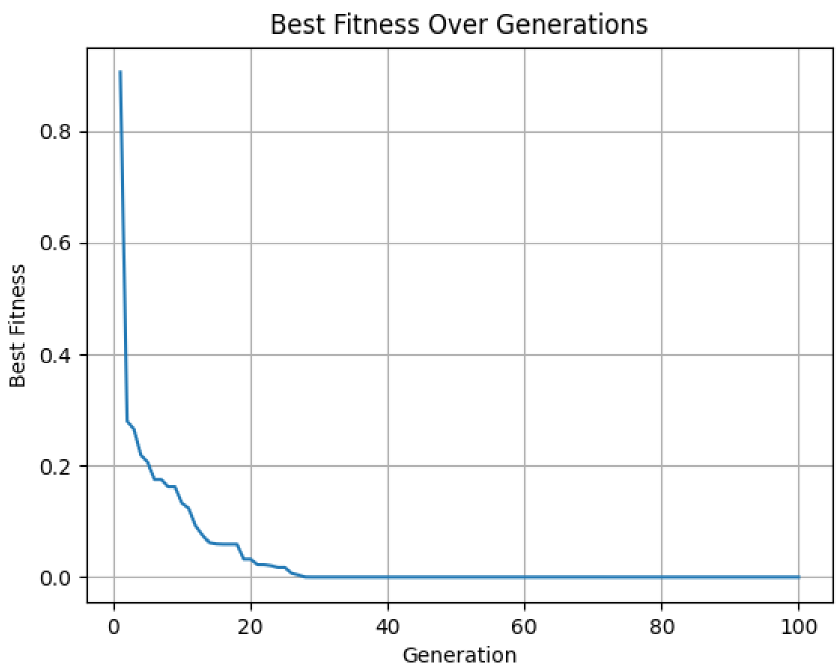

Figure 1: Sample Visualization of Fitness Convergence

Explanation:

- **X-axis:** Represents the generation number.
- **Y-axis:** Represents the best fitness value in each generation.
- **Purpose:** Tracks the convergence of the algorithm and identifies if it is improving or stagnating.

3.2 User Interface

A graphical user interface (GUI) was developed using Tkinter, making the Genetic Algorithm accessible to users without programming expertise. The interface provides options to:

- Select predefined objective functions or define a custom function.
- Input parameters such as population size, number of generations, and mutation rate.
- Specify bounds for variables dynamically.
- Start the Genetic Algorithm and display the results.

```

# Tkinter GUI
def run_gui():
    def start_ga():
        # Retrieve inputs
        try:
            objective_choice = int(obj_func_var.get())
            if objective_choice == 1:
                objective_function = sphere_function
            elif objective_choice == 2:
                objective_function = rosenbrock_function
            elif objective_choice == 3:
                objective_function = himmelblau_function
            elif objective_choice == 4:
                objective_function = n3_function
            elif objective_choice == 5:
                objective_function = n5_function
            elif objective_choice == 6:
                func_input = custom_func_entry.get()
                objective_function = eval(f"lambda x: {func_input}")
            else:
                messagebox.showerror("Error", "Invalid objective function choice.")
                return

            num_variables = int(num_vars_entry.get())
            bounds = []
            for i in range(num_variables):
                bounds.append(tuple(map(float, bounds_entries[i].get().split(','))))

            population_size = int(pop_size_entry.get())
```

```python
            generations = int(generations_entry.get())
            mutation_rate = float(mutation_rate_entry.get())

            # Run GA
            best_solution, best_fitness  = genetic_algorithm(
                objective_function, bounds, population_size, generations, mutation_rate)
            messagebox.showinfo(
            "Best Solution",
            f"Best Solution Found: {best_solution}\nFinal Fitness Score: {best_fitness}"
        )
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred: {e}")

    # Tkinter window setup
    root = tk.Tk()
    root.title("Genetic Algorithm GUI")

    # Objective function selection
    tk.Label(root, text="Select Objective Function").grid(row=0, column=0, columnspan=2)
    obj_func_var = tk.StringVar(value="1")
    tk.Radiobutton(root, text="Sphere Function", variable=obj_func_var, value="1").grid(row=1, column=0)
    tk.Radiobutton(root, text="Rosenbrock Function", variable=obj_func_var, value="2").grid(row=2, column=0)
    tk.Radiobutton(root, text="Himmelblau Function", variable=obj_func_var, value="3").grid(row=3, column=0)
    tk.Radiobutton(root, text="N=3 Function", variable=obj_func_var, value="4").grid(row=4, column=0)
    tk.Radiobutton(root, text="N=5 Function", variable=obj_func_var, value="5").grid(row=5, column=0)
    tk.Radiobutton(root, text="Custom Function", variable=obj_func_var, value="6").grid(row=6, column=0)
    custom_func_entry = tk.Entry(root, width=40)
    custom_func_entry.grid(row=6, column=1)

    # Number of variables
    tk.Label(root, text="Number of Variables:").grid(row=7, column=0)
    num_vars_entry = tk.Entry(root)
    num_vars_entry.grid(row=7, column=1)

    # Scrollable frame for bounds
    tk.Label(root, text="Enter Bounds (min,max for each variable):").grid(row=8, column=0, columnspan=2)

    bounds_frame = ttk.Frame(root)
    bounds_canvas = tk.Canvas(bounds_frame, height=200)  # Set a fixed height
    bounds_scrollbar = ttk.Scrollbar(bounds_frame, orient="vertical", command=bounds_canvas.yview)
    bounds_scrollable_frame = ttk.Frame(bounds_canvas)

    bounds_scrollable_frame.bind(
        "<Configure>",
        lambda e: bounds_canvas.configure(scrollregion=bounds_canvas.bbox("all"))
```

```python
    )

    bounds_canvas.create_window((0, 0), window=bounds_scrollable_frame,
        anchor="nw")
    bounds_canvas.configure(yscrollcommand=bounds_scrollbar.set)

    bounds_frame.grid(row=9, column=0, columnspan=2, sticky="nsew")
    bounds_canvas.pack(side="left", fill="both", expand=True)
    bounds_scrollbar.pack(side="right", fill="y")

    # Create entries for bounds
    bounds_entries = [tk.Entry(bounds_scrollable_frame) for _ in range
        (27)]
    for i, entry in enumerate(bounds_entries):
        tk.Label(bounds_scrollable_frame, text=f"Var {i+1} Bounds:").
            grid(row=i, column=0, sticky="w")
        entry.grid(row=i, column=1)

    # Population size, generations, mutation rate
    tk.Label(root, text="Population Size:").grid(row=36, column=0)
    pop_size_entry = tk.Entry(root)
    pop_size_entry.insert(0, "50")
    pop_size_entry.grid(row=36, column=1)

    tk.Label(root, text="Generations:").grid(row=37, column=0)
    generations_entry = tk.Entry(root)
    generations_entry.insert(0, "100")
    generations_entry.grid(row=37, column=1)

    tk.Label(root, text="Mutation Rate:").grid(row=38, column=0)
    mutation_rate_entry = tk.Entry(root)
    mutation_rate_entry.insert(0, "0.1")
    mutation_rate_entry.grid(row=38, column=1)

    # Start button
    tk.Button(root, text="Start Genetic Algorithm", command=start_ga).
        grid(row=39, column=0, columnspan=2)

    root.mainloop()

# Run the GUI
if __name__ == "__main__":
    run_gui()
```

Listing 11: Tkinter GUI Setup Code

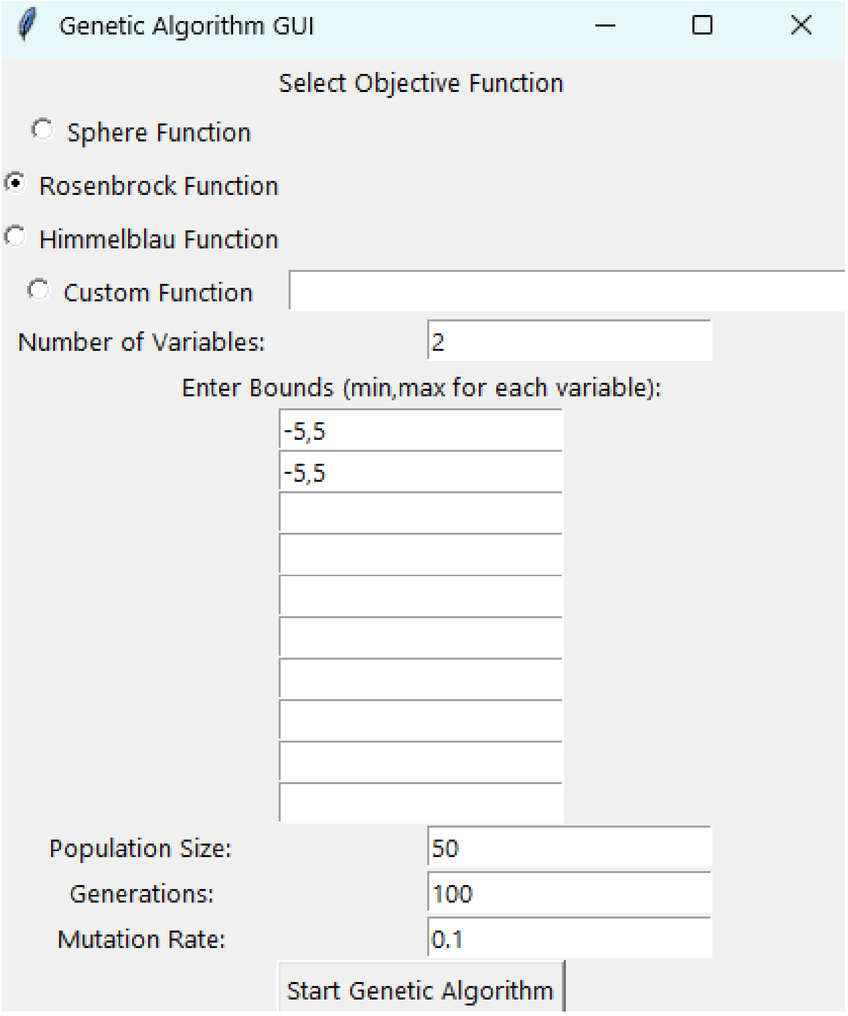

Figure 2: Graphical User Interface for Genetic Algorithm

Explanation:

- The GUI features radio buttons for selecting objective functions and input fields for parameters.
- The "Start Genetic Algorithm" button triggers the computation and displays the best solution found.
- Users can define custom objective functions dynamically using the text field or are given the functionality of hardcoding.

4. Installation, Compilation, and Execution Instructions

The following steps provide clear instructions to set up and run the Genetic Algorithm project:

Prerequisites

Before running the project, ensure you have:

- **Python 3.8 or higher** installed on your system.
- A code editor or IDE (e.g., Visual Studio Code, PyCharm, or Notepad++).
- Basic knowledge of running commands in a terminal or command prompt.

You can check if Python is installed by running the following command in your terminal or command prompt:

```
python --version
```

Installing Required Libraries

The project requires the following Python libraries:

- **numpy**: For mathematical operations.
- **matplotlib**: For visualizing the fitness plot.
- **tkinter**: Pre-installed with Python and used for the GUI.

To install the missing libraries, run the following command:

```
pip install numpy matplotlib
```

You can confirm the installation by running:

```
pip show numpy matplotlib
```

Note: `tkinter` comes pre-installed with Python, so no additional installation is required.

Downloading the Project

- Download the project files (e.g., `genetic_algorithm.py`).
- Place all files in a single folder on your computer.

Running the Program

To execute the Genetic Algorithm and interact with the GUI:

1. Open your terminal or command prompt and navigate to the project folder:

```
    cd path_to_your_project_folder
```

   Replace `path_to_your_project_folder` with the actual path to the folder containing the program.

2. Run the program using Python:

```
    python genetic_algorithm.py
```

3. The GUI will appear. Follow these steps to interact with it:
   - Select an objective function (e.g., Sphere Function).
   - Enter parameters such as the number of variables, bounds, population size, generations, and mutation rate.
   - Click the **Start Genetic Algorithm** button to run the optimization.
   - A popup will display the best solution and fitness score, and a fitness plot will appear.

Example Input

For testing purposes, use the following inputs:

- **Objective Function:** Sphere Function.
- **Number of Variables:** 2.
- **Bounds:** Enter `-5,5` for both variables.
- **Population Size:** 50.
- **Generations:** 100.
- **Mutation Rate:** 0.1.

Troubleshooting

- If `numpy` or `matplotlib` is not installed, ensure you run the `pip install` command as described above.
- If Python is not recognized as a command, add it to your system's PATH variable during installation or reinstall Python.

5. Test Runs on Classical Benchmark Objective Functions

This section presents the results of the Simple Genetic Algorithm (SGA) tested on three classical benchmark objective functions: Sphere Function, Rosenbrock Function, and Himmelblau Function. The results are illustrated through visualizations and table of the search process and the convergence of the algorithm for each function with detailed explanation of the correctness.

5.1 Sphere Function

The Sphere function is defined as:

$$f(x) = \sum_{i=1}^{n} x_i^2$$

where x_i represents the variables of the function. The global minimum is at (0,0). To evaluate the correctness of the SGA in the sphere function, the computed solution and fitness score need to be as close to 0. In some cases the solution and fitness source can achieve a very small value (e.g., 10^{-9}) as seen in the solutions below (figure 4, 5 & 6).

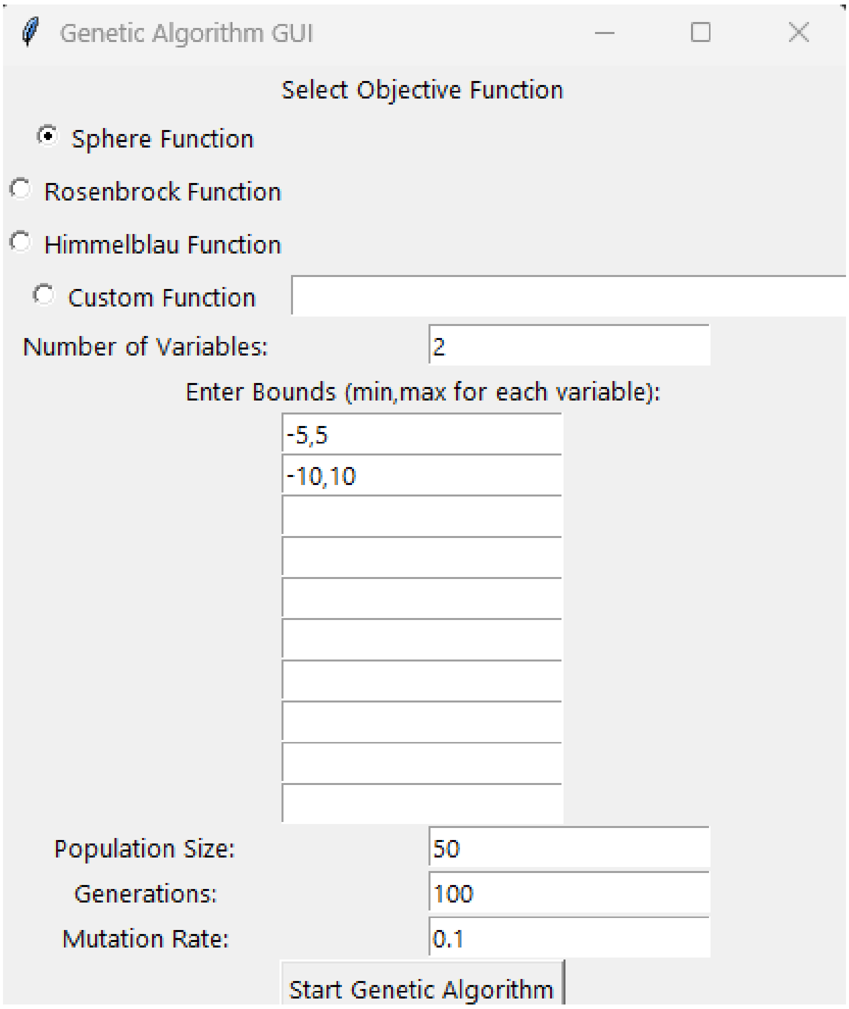

Figure 3: Input Process for Sphere Function

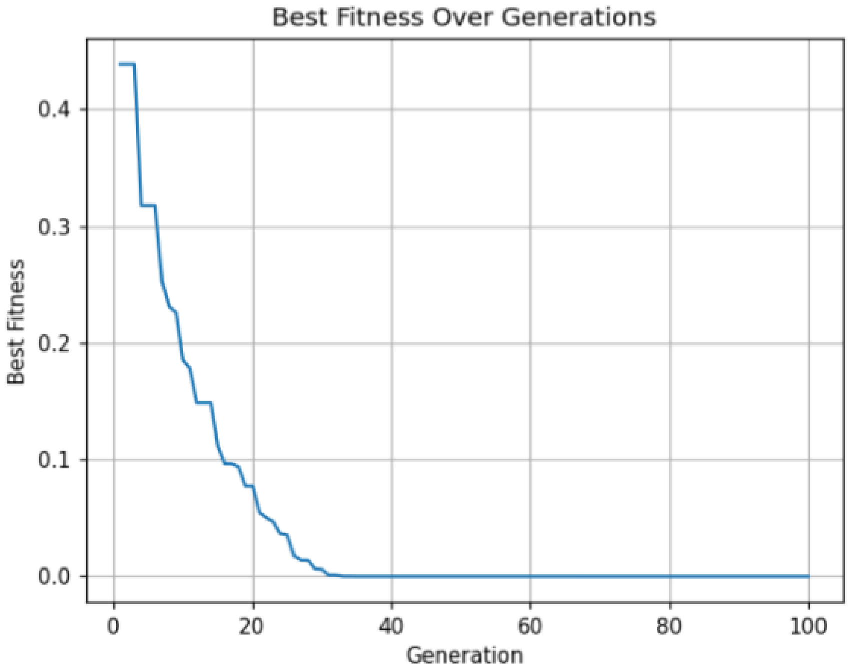

Figure 4: Fitness Visualization for Sphere Function

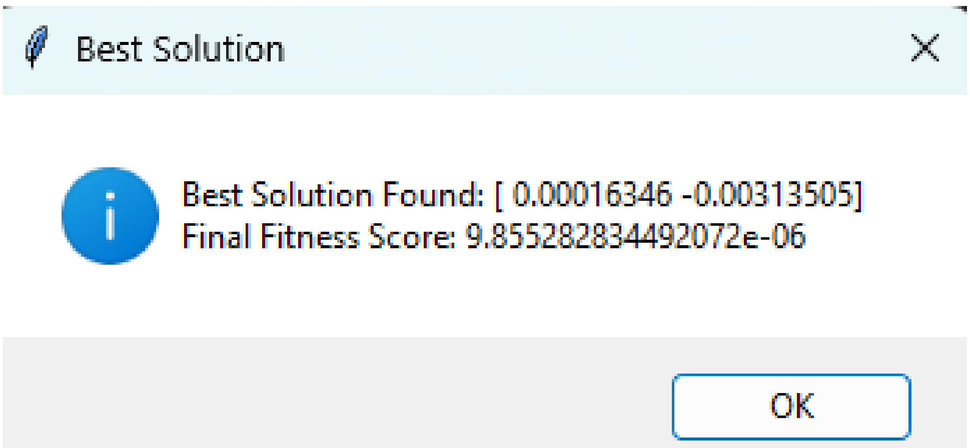

Figure 5: Best Solution and Final Fitness Score

```
Generation 76 - Number of Individuals: 50
Generation 76 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 77 - Number of Individuals: 50
Generation 77 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 78 - Number of Individuals: 50
Generation 78 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 79 - Number of Individuals: 50
Generation 79 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 80 - Number of Individuals: 50
Generation 80 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 81 - Number of Individuals: 50
Generation 81 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 82 - Number of Individuals: 50
Generation 82 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 83 - Number of Individuals: 50
Generation 83 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 84 - Number of Individuals: 50
Generation 84 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 85 - Number of Individuals: 50
Generation 85 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 86 - Number of Individuals: 50
Generation 86 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 87 - Number of Individuals: 50
Generation 87 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 88 - Number of Individuals: 50
Generation 88 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 89 - Number of Individuals: 50
Generation 89 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 90 - Number of Individuals: 50
Generation 90 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 91 - Number of Individuals: 50
Generation 91 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 92 - Number of Individuals: 50
Generation 92 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 93 - Number of Individuals: 50
Generation 93 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 94 - Number of Individuals: 50
Generation 94 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 95 - Number of Individuals: 50
Generation 95 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 96 - Number of Individuals: 50
Generation 96 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 97 - Number of Individuals: 50
Generation 97 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 98 - Number of Individuals: 50
Generation 98 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 99 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06
Generation 100 - Number of Individuals: 50
Generation 100 - Best Solution: [ 0.00016346 -0.00313505], Fitness: 9.855282834492072e-06

Final Best Solution: [ 0.00016346 -0.00313505]
Final Fitness Score: 9.855282834492072e-06
```

Figure 6: Generation-wise Best Solutions and Fitness Convergence

5.2 Rosenbrock Function

The Rosenbrock function is defined as:

$$f(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

The global minimum is at $x = 1$, with $f(x) = 0$. The correctness of the SGA can be proved by having the final best solution as close as possible to [1,1] and the final fitness score as close as to 0. Figure 9 & 10 show that the algorithm did converge close to the global minimum, proving the correctness of our SGA implementation.

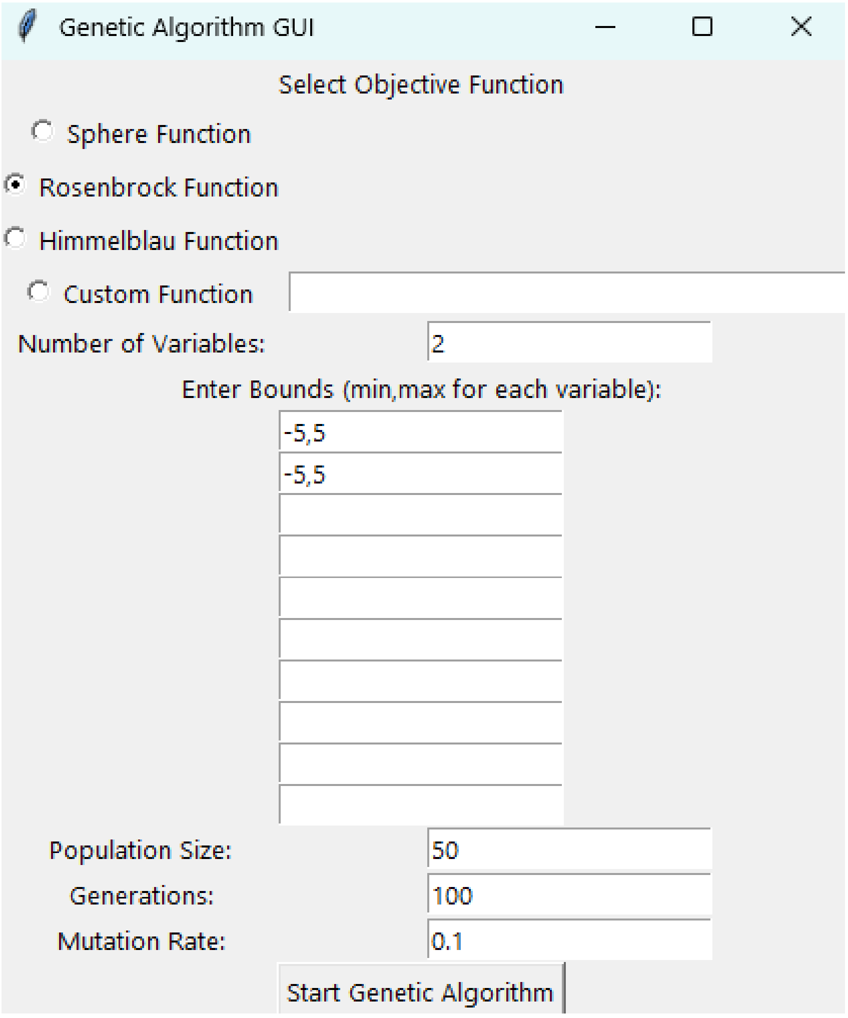

Figure 7: Input Process for Rosenbrock Function

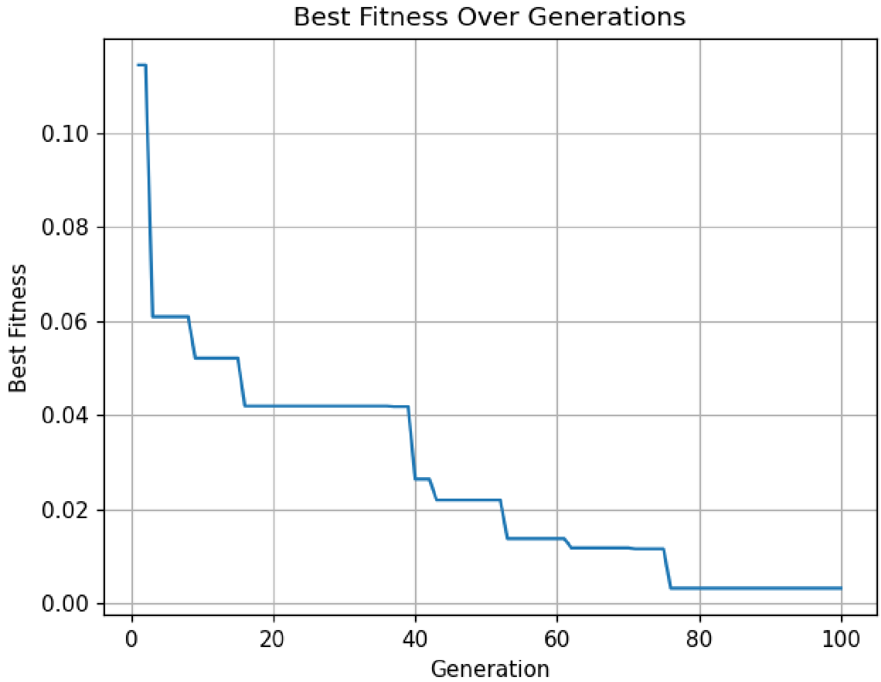

Figure 8: Fitness Visualization for Rosenbrock Function

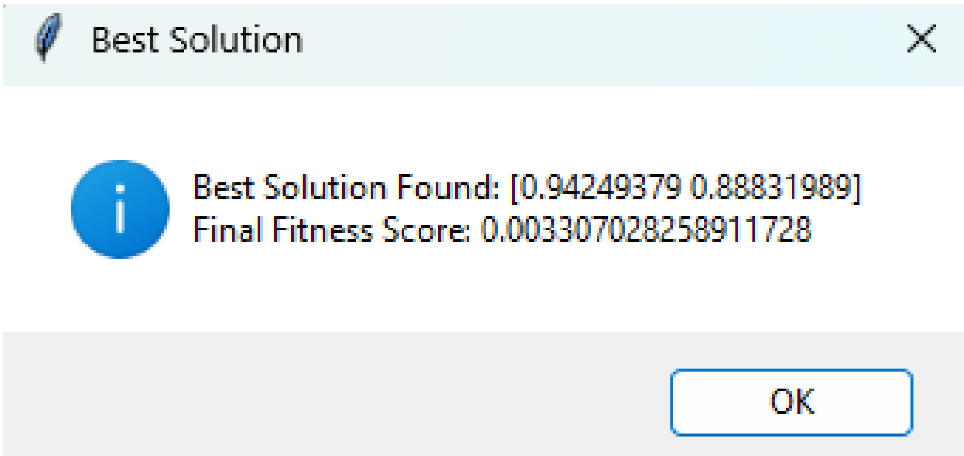

Figure 9: Best Solution and Final Fitness Score

```
Generation 82 - Number of Individuals: 50
Generation 82 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 83 - Number of Individuals: 50
Generation 83 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 84 - Number of Individuals: 50
Generation 84 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 85 - Number of Individuals: 50
Generation 85 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 86 - Number of Individuals: 50
Generation 86 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 87 - Number of Individuals: 50
Generation 87 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 88 - Number of Individuals: 50
Generation 88 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 89 - Number of Individuals: 50
Generation 89 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 90 - Number of Individuals: 50
Generation 90 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 91 - Number of Individuals: 50
Generation 91 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 92 - Number of Individuals: 50
Generation 92 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 93 - Number of Individuals: 50
Generation 93 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 94 - Number of Individuals: 50
Generation 94 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 95 - Number of Individuals: 50
Generation 95 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 96 - Number of Individuals: 50
Generation 96 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 97 - Number of Individuals: 50
Generation 97 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 98 - Number of Individuals: 50
Generation 98 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 99 - Number of Individuals: 50
Generation 99 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728
Generation 100 - Number of Individuals: 50
Generation 100 - Best Solution: [0.94249379 0.88831989], Fitness: 0.003307028258911728

Final Best Solution: [0.94249379 0.88831989]
Final Fitness Score: 0.003307028258911728
```

Figure 10: Generation-wise Best Solutions and Fitness Convergence

5.3 Himmelblau Function

The Himmelblau function is defined as:

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

The function has four global minima:

- $[3, 2]$, where $f(3, 2) = 0$
- $[-2.805118, 3.131312]$, where $f(-2.805118, 3.131312) = 0$
- $[-3.779310, -3.283186]$, where $f(-3.779310, -3.283186) = 0$
- $[3.584428, -1.848126]$, where $f(3.584428, -1.848126) = 0$

To prove the correctness of the SGA, the best solution must converge to 1 of the 4 global minima, that is the final best solution as close as possible to 1 of 4 global minima. The final fitness score must be as close as to 0. Figure 13 & 14 indicate that the algorithm did coverage as close as possible to the global minimum, thus proving the correctness of our SGA implementation.

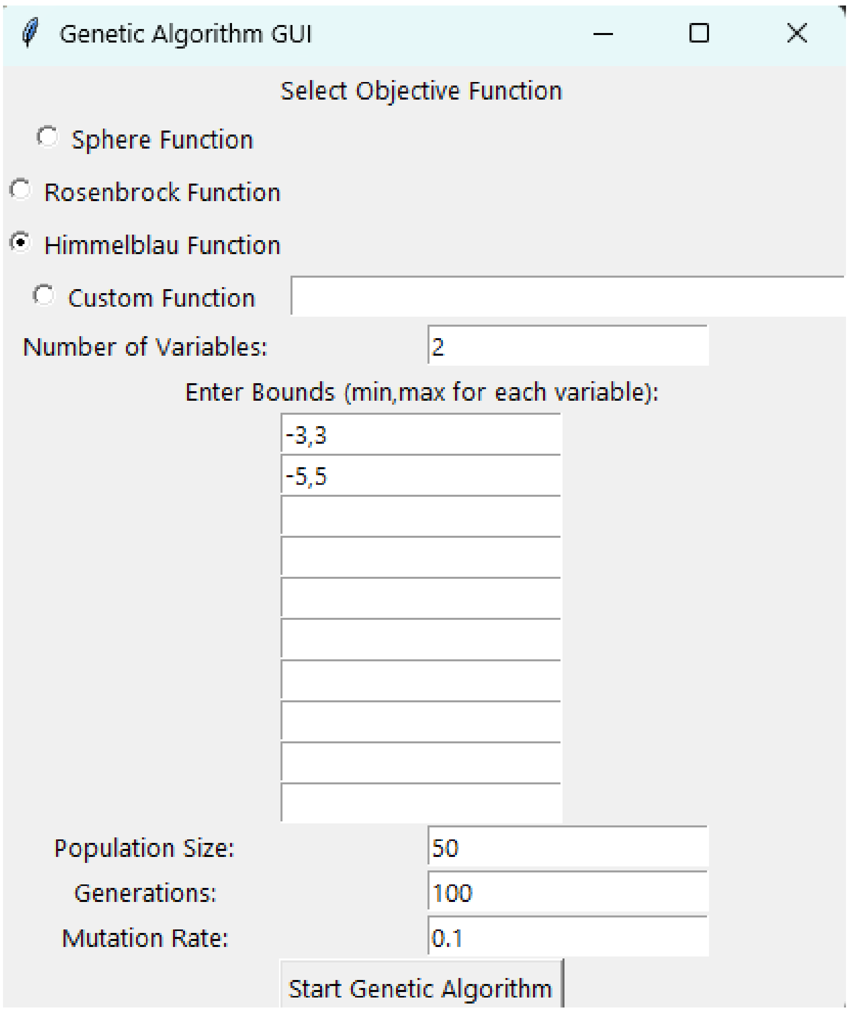

Figure 11: Input Process for Himmelblau Function

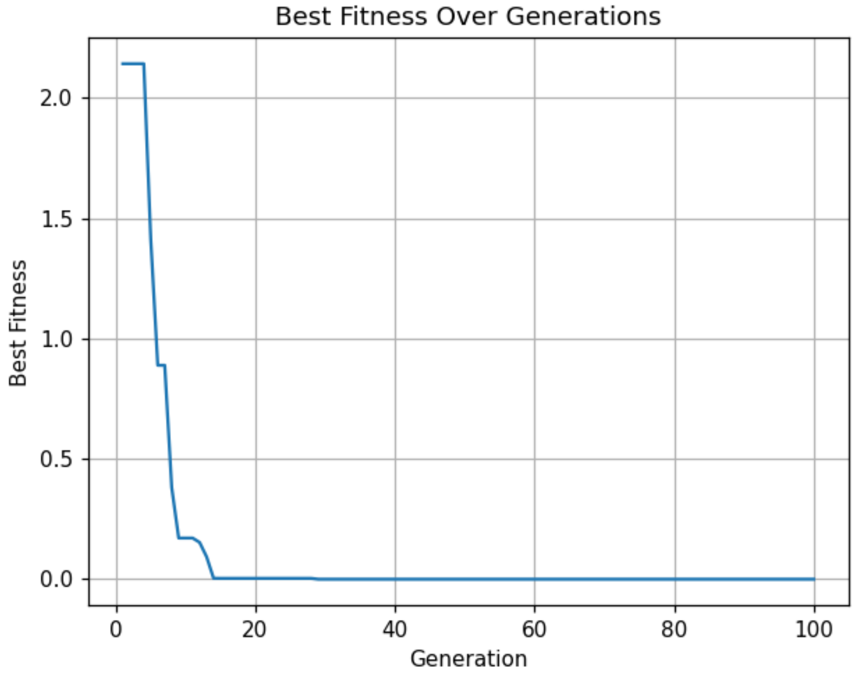

Figure 12: Fitness Visualization for Himmelblau Function

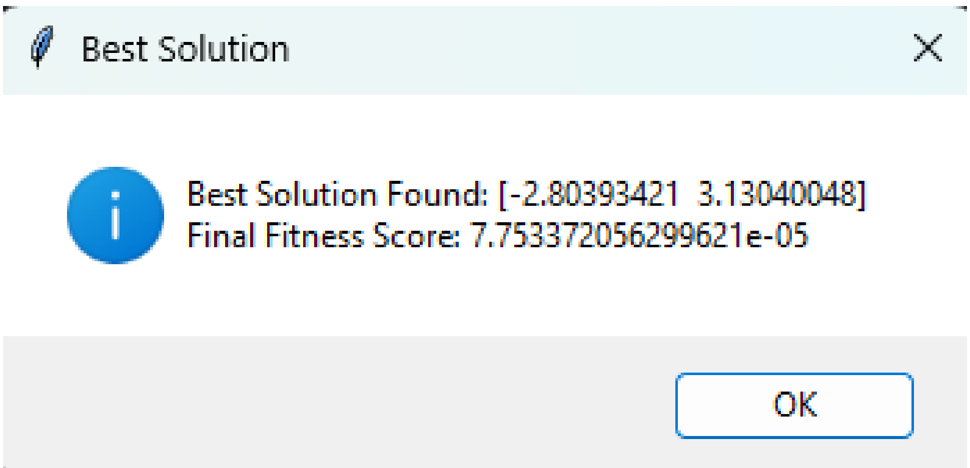

Figure 13: Best Solution and Final Fitness Score

```
Generation 77 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 78 - Number of Individuals: 50
Generation 78 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 79 - Number of Individuals: 50
Generation 79 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 80 - Number of Individuals: 50
Generation 80 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 81 - Number of Individuals: 50
Generation 81 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 82 - Number of Individuals: 50
Generation 82 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 83 - Number of Individuals: 50
Generation 83 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 84 - Number of Individuals: 50
Generation 84 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 85 - Number of Individuals: 50
Generation 85 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 86 - Number of Individuals: 50
Generation 86 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 87 - Number of Individuals: 50
Generation 87 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 88 - Number of Individuals: 50
Generation 88 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 89 - Number of Individuals: 50
Generation 89 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 90 - Number of Individuals: 50
Generation 90 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 91 - Number of Individuals: 50
Generation 91 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 92 - Number of Individuals: 50
Generation 92 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 93 - Number of Individuals: 50
Generation 93 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 94 - Number of Individuals: 50
Generation 94 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 95 - Number of Individuals: 50
Generation 95 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 96 - Number of Individuals: 50
Generation 96 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 97 - Number of Individuals: 50
Generation 97 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 98 - Number of Individuals: 50
Generation 98 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 99 - Number of Individuals: 50
Generation 99 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05
Generation 100 - Number of Individuals: 50
Generation 100 - Best Solution: [-2.80393421  3.13040048], Fitness: 7.753372056299621e-05

Final Best Solution: [-2.80393421  3.13040048]
Final Fitness Score: 7.753372056299621e-05
```

Figure 14: Generation-wise Best Solutions and Fitness Convergence

6. Test Runs on Custom Objective Functions

This section presents the results of 2 given objective functions by Dr. Ilias S. Kotsireas, Professor.

6.1 Custom Function 1

For $n = 3$, the following expression is used:

$$|a_0a_1 + 2b_0b_1 + 2c_0c_1 + 2d_0d_1 + 2e_0e_1 + 2f_0f_1 + 2g_0g_1 + 2h_0h_1 + i_0i_1 + 8|$$

```
def n3_function(x):
    return np.abs(
        x[0] * x[1] +
        2 * x[2] * x[3] +
        2 * x[4] * x[5] +
        2 * x[6] * x[7] +
        2 * x[8] * x[9] +
        2 * x[10] * x[11] +
        2 * x[12] * x[13] +
        2 * x[14] * x[15] +
        x[16] * x[17] +
        8
    )
```

Figure 15: Input Process for Custom 1 Function

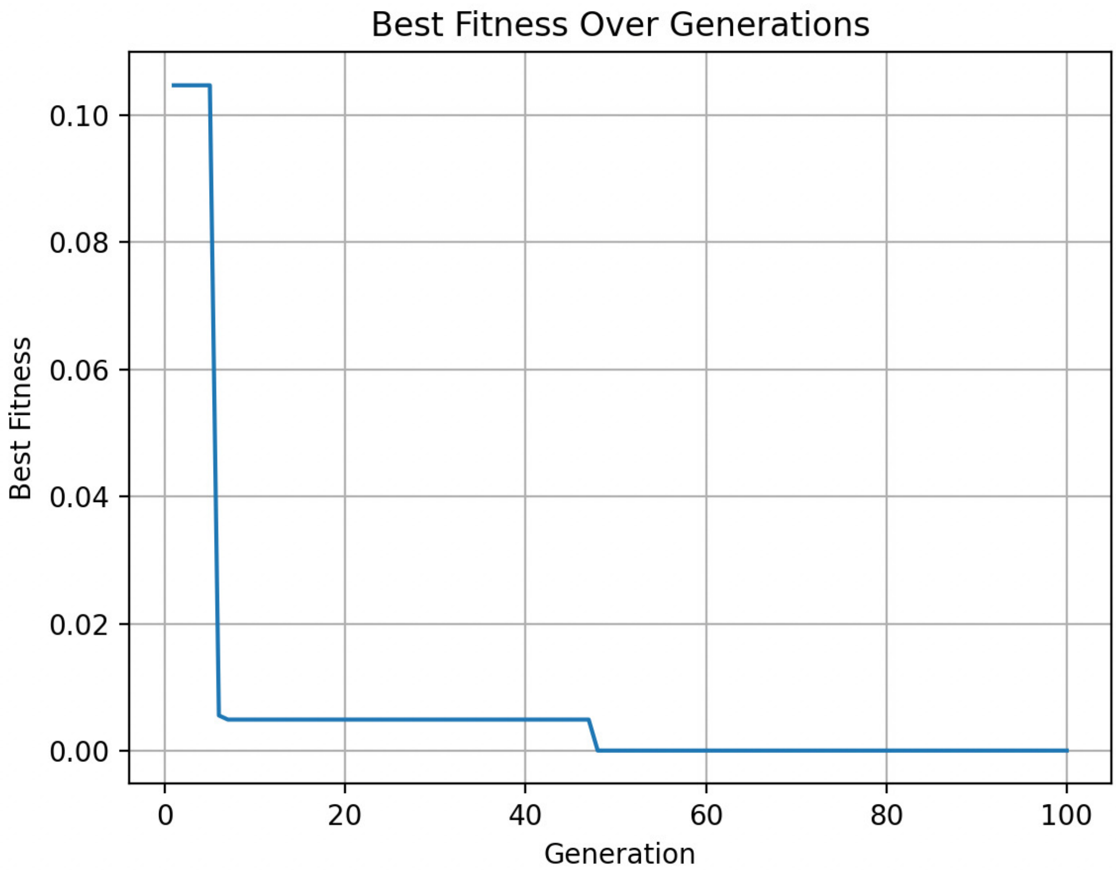

Figure 16: Fitness Visualization for Custom Function 1

```
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 88 - Number of Individuals: 50
Generation 88 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 89 - Number of Individuals: 50
Generation 89 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 90 - Number of Individuals: 50
Generation 90 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 91 - Number of Individuals: 50
Generation 91 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 92 - Number of Individuals: 50
Generation 92 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 93 - Number of Individuals: 50
Generation 93 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 94 - Number of Individuals: 50
Generation 94 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 95 - Number of Individuals: 50
Generation 95 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 96 - Number of Individuals: 50
Generation 96 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 97 - Number of Individuals: 50
Generation 97 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 98 - Number of Individuals: 50
Generation 98 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 99 - Number of Individuals: 50
Generation 99 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05
Generation 100 - Number of Individuals: 50
Generation 100 - Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771], Fitness: 7.476642150461288e-05

Final Best Solution: [ 3.73957    -2.08694621 -3.52091987 -1.09653401  1.61439843  2.27276019
  2.77393251 -4.05561271  1.11510483  0.0496407  -2.66943846 -3.41243944
 -3.94438215 -0.40978811 -1.48804694  4.08048295  0.55774463 -3.89725771]
Final Fitness Score: 7.476642150461288e-05
```

Figure 17: Generation-wise Best Solutions and Fitness Convergence

6.2 Custom Function 2

For $n = 5$, the following expression is used:

$$
\begin{aligned}
&\big|a_0a_2 + a_1a_2 + 2b_0b_2 + 2b_1b_2 + 2c_0c_2 + 2c_1c_2 \\
&\quad + 2d_0d_2 + 2d_1d_2 + 2e_0e_2 + 2e_1e_2 + 2f_0f_2 + 2f_1f_2 \\
&\quad + 2g_0g_2 + 2g_1g_2 + 2h_0h_2 + 2h_1h_2 + i_0i_2 + i_1i_2 + 8\big| + \\
&\big|a_0a_2 + a_1a_2 + 2b_0b_2 + 2b_1b_2 + 2c_0c_2 + 2c_1c_2 \\
&\quad + 2d_0d_2 + 2d_1d_2 + 2e_0e_2 + 2e_1e_2 + 2f_0f_2 + 2f_1f_2 \\
&\quad + 2g_0g_2 + 2g_1g_2 + 2h_0h_2 + 2h_1h_2 + i_0i_2 + i_1i_2 + 8\big|
\end{aligned}
$$

```python
def n5_function(x):
    return (
        np.abs(
            x[0] * x[2] +
            x[1] * x[2] +
            2 * x[3] * x[5] +
            2 * x[4] * x[5] +
            2 * x[6] * x[8] +
            2 * x[7] * x[8] +
            2 * x[9] * x[11] +
            2 * x[10] * x[11] +
            2 * x[12] * x[14] +
            2 * x[13] * x[14] +
            2 * x[15] * x[17] +
            2 * x[16] * x[17] +
            2 * x[18] * x[20] +
            2 * x[19] * x[20] +
            2 * x[21] * x[23] +
            2 * x[22] * x[23] +
            x[24] * x[26] +
            x[25] * x[26] +
            8
    ) +
        np.abs (
            x[0] * x[1] +
            x[1] * x[2] +
            2 * x[3] * x[4] +
            2 * x[4] * x[5] +
            2 * x[6] * x[7] +
            2 * x[7] * x[8] +
            2 * x[9] * x[10] +
            2 * x[10] * x[11] +
            2 * x[12] * x[13] +
            2 * x[13] * x[14] +
            2 * x[15] * x[16] +
            2 * x[16] * x[17] +
            2 * x[18] * x[19] +
            2 * x[19] * x[20] +
            2 * x[21] * x[22] +
            2 * x[22] * x[23] +
            x[24] * x[25] +
            x[25] * x[26] +
            8
        )
)
```

Figure 18: Input Process for Custom 2 Function

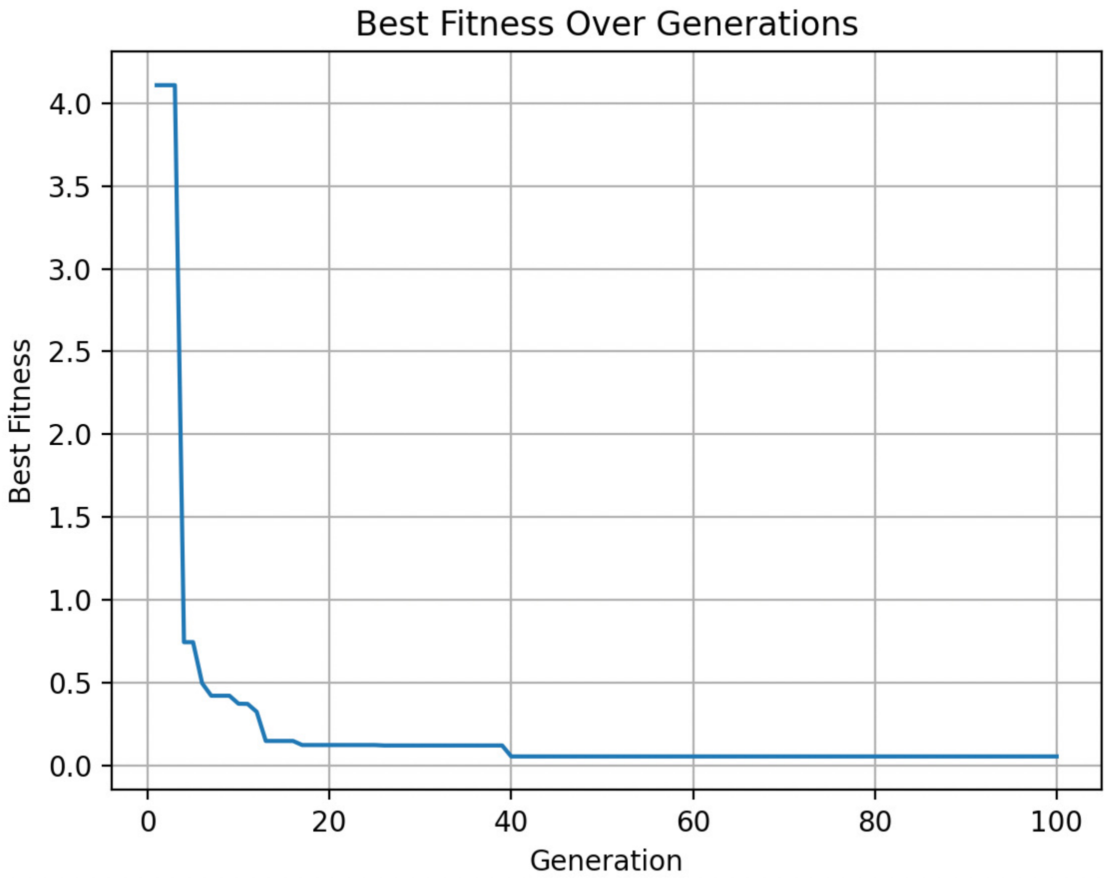

Figure 19: Fitness Visualization for Custom Function 2

```
Generation 92 - Number of Individuals: 50
Generation 92 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 93 - Number of Individuals: 50
Generation 93 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 94 - Number of Individuals: 50
Generation 94 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 95 - Number of Individuals: 50
Generation 95 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 96 - Number of Individuals: 50
Generation 96 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 97 - Number of Individuals: 50
Generation 97 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 98 - Number of Individuals: 50
Generation 98 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 99 - Number of Individuals: 50
Generation 99 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941
Generation 100 - Number of Individuals: 50
Generation 100 - Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915], Fitness: 0.05720223894432941

Final Best Solution: [-2.92102007 -0.7669046  -4.74002546  3.35172244 -2.09571794  3.73434001
 -4.29616027  0.85807323  3.4941988   3.44791261 -0.93956406 -2.52163499
  1.45389247 -3.7080838  -2.55208832  0.84650527  0.06347662  0.20028874
  0.53077053 -0.40027698 -1.06591096 -2.37387511 -2.24382316  0.38749745
 -2.22995049 -1.8837404   1.50379915]
Final Fitness Score: 0.05720223894432941
```

Figure 20: Generation-wise Best Solutions and Fitness Convergence

7. Code

This section includes our detailed SGA implementation code including all the objective functions, custom objective functions, with the Graphical User Interface (GUI) and visualization of the end results.

```
import numpy as np
import matplotlib.pyplot as plt
import tkinter as tk
from tkinter import messagebox, simpledialog, ttk

# Objective functions
def sphere_function(x):
    return np.sum(x**2)

def rosenbrock_function(x):
    return np.sum(100 * (x[1:] - x[:-1]**2)**2 + (1 - x[:-1])**2)

def himmelblau_function(x):
    return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2

def function_(x):
    return np.abs((x[0]*x[1]) + (2*x[2]*x[3]) + (2*x[4]*x[5]) + (2*x
        [6]*x[7]))

def n3_function(x):
    return np.abs(
        x[0] * x[1] +
        2 * x[2] * x[3] +
        2 * x[4] * x[5] +
        2 * x[6] * x[7] +
        2 * x[8] * x[9] +
        2 * x[10] * x[11] +
        2 * x[12] * x[13] +
        2 * x[14] * x[15] +
        x[16] * x[17] +
        8
    )

def n5_function(x):
    return (
        np.abs(
            x[0] * x[2] +
            x[1] * x[2] +
            2 * x[3] * x[5] +
            2 * x[4] * x[5] +
            2 * x[6] * x[8] +
            2 * x[7] * x[8] +
            2 * x[9] * x[11] +
            2 * x[10] * x[11] +
            2 * x[12] * x[14] +
            2 * x[13] * x[14] +
            2 * x[15] * x[17] +
            2 * x[16] * x[17] +
            2 * x[18] * x[20] +
            2 * x[19] * x[20] +
            2 * x[21] * x[23] +
            2 * x[22] * x[23] +
```

```
                x[24] * x[26] +
                x[25] * x[26] +
                8
    ) +
        np.abs (
            x[0] * x[1] +
            x[1] * x[2] +
            2 * x[3] * x[4] +
            2 * x[4] * x[5] +
            2 * x[6] * x[7] +
            2 * x[7] * x[8] +
            2 * x[9] * x[10] +
            2 * x[10] * x[11] +
            2 * x[12] * x[13] +
            2 * x[13] * x[14] +
            2 * x[15] * x[16] +
            2 * x[16] * x[17] +
            2 * x[18] * x[19] +
            2 * x[19] * x[20] +
            2 * x[21] * x[22] +
            2 * x[22] * x[23] +
            x[24] * x[25] +
            x[25] * x[26] +
            8
        )
)

def genetic_algorithm(
    objective_function, bounds, population_size=50, generations=100,
        mutation_rate=0.1
):
    # Number of variables in problem
    dim = len(bounds)

    # Create population with random variables within bounds
    population = np.random.uniform(
        [b[0] for b in bounds],  # Lower bounds
        [b[1] for b in bounds],  # Upper bounds
        (population_size, dim)   # Shape of population
    )

    num_elites = 4
    best_fitness_per_generation = []

    # Iterate through each generation
    for generation in range(generations):
        # Obtain fitness of every individual
        fitness = np.array([objective_function(ind) for ind in
            population])

        # Keep track of elites in the generation
        elite_indices = np.argsort(fitness)[:num_elites]
        elite_individuals = population[elite_indices]

        # Calculate probabilities for biased wheel
        total_fitness = np.sum(fitness)
        selection_probs = (1 / (fitness + 1e-6)) / total_fitness

```

```
        # Normalize (ensure probabilities add to 1)
        selection_probs /= np.sum(selection_probs)

        # Select mating pool based on biased wheel
        selected = population[np.random.choice(len(population), size=population_size, p=selection_probs)]

        # Create offspring using crossover and mutation
        offspring = []
        for _ in range((population_size - num_elites) // 2):
            # Randomly select 2 parents from mating pool
            p1, p2 = selected[np.random.choice(len(selected), size=2)]

            # Create crossover point and make children based on parents and crossover point
            crossover_point = np.random.randint(1, dim)
            child1 = np.concatenate((p1[:crossover_point], p2[crossover_point:]))
            child2 = np.concatenate((p2[:crossover_point], p1[crossover_point:]))

            # Chance to mutate a child
            if np.random.rand() < mutation_rate:
                child1 += np.random.uniform(-0.1, 0.1, dim)
            if np.random.rand() < mutation_rate:
                child2 += np.random.uniform(-0.1, 0.1, dim)

            # Add children to list
            offspring.append(child1)
            offspring.append(child2)

        # Replace old population with elites and children
        population = np.vstack((elite_individuals, np.array(offspring)))


        # Print generation and number of individuals
        print(f"Generation {generation + 1} - Number of Individuals: {len(population)}")

        # Obtain best individual and fitness in current generation
        best_individual_gen = population[np.argmin([objective_function(ind) for ind in population])]
        best_fitness_gen = objective_function(best_individual_gen)
        best_fitness_per_generation.append(best_fitness_gen)

        # Print best individual and fitness score in each generation
        print(f"Generation {generation + 1} - Best Solution: {best_individual_gen}, Fitness: {best_fitness_gen}")

    # Obtain best overall individual and fitness score
    best_individual = population[np.argmin([objective_function(ind) for ind in population])]
    best_fitness = objective_function(best_individual)

    # Print overall best individual and fitness score
    print("\nFinal Best Solution:", best_individual)
    print("Final Fitness Score:", best_fitness)
```

```python

    # Plot the fitness over generations
    plt.plot(range(1, generations + 1), best_fitness_per_generation)
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness")
    plt.title("Best Fitness Over Generations")
    plt.grid(True)
    plt.show()

    return best_individual, best_fitness

# Tkinter GUI
def run_gui():
    def start_ga():
        # Retrieve inputs
        try:
            objective_choice = int(obj_func_var.get())
            if objective_choice == 1:
                objective_function = sphere_function
            elif objective_choice == 2:
                objective_function = rosenbrock_function
            elif objective_choice == 3:
                objective_function = himmelblau_function
            elif objective_choice == 4:
                objective_function = n3_function
            elif objective_choice == 5:
                objective_function = n5_function
            elif objective_choice == 6:
                func_input = custom_func_entry.get()
                objective_function = eval(f"lambda x: {func_input}")
            else:
                messagebox.showerror("Error", "Invalid objective function choice.")
                return

            num_variables = int(num_vars_entry.get())
            bounds = []
            for i in range(num_variables):
                bounds.append(tuple(map(float, bounds_entries[i].get().split(','))))

            population_size = int(pop_size_entry.get())
            generations = int(generations_entry.get())
            mutation_rate = float(mutation_rate_entry.get())

            # Run GA
            best_solution, best_fitness  = genetic_algorithm(objective_function, bounds, population_size, generations, mutation_rate)
            messagebox.showinfo(
            "Best Solution",
            f"Best Solution Found: {best_solution}\nFinal Fitness Score: {best_fitness}"
        )
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred: {e}")

    # Tkinter window setup
```

```
    root = tk.Tk()
    root.title("Genetic Algorithm GUI")

    # Objective function selection
    tk.Label(root, text="Select Objective Function").grid(row=0, column=0, columnspan=2)
    obj_func_var = tk.StringVar(value="1")
    tk.Radiobutton(root, text="Sphere Function", variable=obj_func_var, value="1").grid(row=1, column=0)
    tk.Radiobutton(root, text="Rosenbrock Function", variable=obj_func_var, value="2").grid(row=2, column=0)
    tk.Radiobutton(root, text="Himmelblau Function", variable=obj_func_var, value="3").grid(row=3, column=0)
    tk.Radiobutton(root, text="N=3 Function", variable=obj_func_var, value="4").grid(row=4, column=0)
    tk.Radiobutton(root, text="N=5 Function", variable=obj_func_var, value="5").grid(row=5, column=0)
    tk.Radiobutton(root, text="Custom Function", variable=obj_func_var, value="6").grid(row=6, column=0)
    custom_func_entry = tk.Entry(root, width=40)
    custom_func_entry.grid(row=6, column=1)

    # Number of variables
    tk.Label(root, text="Number of Variables:").grid(row=7, column=0)
    num_vars_entry = tk.Entry(root)
    num_vars_entry.grid(row=7, column=1)

    # Scrollable frame for bounds
    tk.Label(root, text="Enter Bounds (min,max for each variable):").grid(row=8, column=0, columnspan=2)

    bounds_frame = ttk.Frame(root)
    bounds_canvas = tk.Canvas(bounds_frame, height=200)  # Set a fixed height
    bounds_scrollbar = ttk.Scrollbar(bounds_frame, orient="vertical", command=bounds_canvas.yview)
    bounds_scrollable_frame = ttk.Frame(bounds_canvas)

    bounds_scrollable_frame.bind(
        "<Configure>",
        lambda e: bounds_canvas.configure(scrollregion=bounds_canvas.bbox("all"))
    )

    bounds_canvas.create_window((0, 0), window=bounds_scrollable_frame, anchor="nw")
    bounds_canvas.configure(yscrollcommand=bounds_scrollbar.set)

    bounds_frame.grid(row=9, column=0, columnspan=2, sticky="nsew")
    bounds_canvas.pack(side="left", fill="both", expand=True)
    bounds_scrollbar.pack(side="right", fill="y")

    # Create entries for bounds
    bounds_entries = [tk.Entry(bounds_scrollable_frame) for _ in range(27)]
    for i, entry in enumerate(bounds_entries):
        tk.Label(bounds_scrollable_frame, text=f"Var {i+1} Bounds:").grid(row=i, column=0, sticky="w")
```

```
        entry.grid(row=i, column=1)

    # Population size, generations, mutation rate
    tk.Label(root, text="Population Size:").grid(row=36, column=0)
    pop_size_entry = tk.Entry(root)
    pop_size_entry.insert(0, "50")
    pop_size_entry.grid(row=36, column=1)

    tk.Label(root, text="Generations:").grid(row=37, column=0)
    generations_entry = tk.Entry(root)
    generations_entry.insert(0, "100")
    generations_entry.grid(row=37, column=1)

    tk.Label(root, text="Mutation Rate:").grid(row=38, column=0)
    mutation_rate_entry = tk.Entry(root)
    mutation_rate_entry.insert(0, "0.1")
    mutation_rate_entry.grid(row=38, column=1)

    # Start button
    tk.Button(root, text="Start Genetic Algorithm", command=start_ga).grid(row=39, column=0, columnspan=2)

    root.mainloop()

# Run the GUI
if __name__ == "__main__":
    run_gui()
```

8. Project Resources

Genetic Algorithms in Search, Optimization, and Machine Learning by David E. Goldberg, Addison-Wesley, 1989.