

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.

```

1 # Create population with random variables within bounds
2 population = np.random.uniform(
3     [b[0] for b in bounds], # Lower bounds
4     [b[1] for b in bounds], # Upper bounds
5     (population_size, dim) # Shape of population
6 )

```

Listing 1: Population Initialization Code

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

```

1 # Obtain fitness of every individual
2 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.

```

1 # Calculate probabilities for biased wheel
2 total_fitness = np.sum(fitness)
3 selection_probs = (1 / (fitness + 1e-6)) / total_fitness
4
5 # Normalize (ensure probabilities add to 1)
6 selection_probs /= np.sum(selection_probs)
7
8 # Select mating pool based on biased wheel
9 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.

```

1 # Create offspring using crossover
2 offspring = []
3 for _ in range((population_size - num_elites) // 2):
4     # Randomly select 2 parents from mating pool
5     p1, p2 = selected[np.random.choice(len(selected), size=2)]
6
7     # Create crossover point and make children based on parents and
8     # crossover point
9     crossover_point = np.random.randint(1, dim)
10    child1 = np.concatenate((p1[:crossover_point], p2[
11        crossover_point:]))
12    child2 = np.concatenate((p2[:crossover_point], p1[
13        crossover_point:]))
14
15    # Add children to list
16    offspring.append(child1)
17    offspring.append(child2)

```

Listing 4: Crossover Code

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

```

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

```

Listing 5: Mutation Code

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

```

1 # Keep track of elites in the generation
2 elite_indices = np.argsort(fitness)[:num_elites]
3 elite_individuals = population[elite_indices]
4
5 # Replace old population with elites and children
6 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.

```

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

```

Listing 7: Exploration Code

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

```

1 # Selection biases towards fitter individuals
2 selection_probs = (1 / (fitness + 1e-6)) / total_fitness
3 selection_probs /= np.sum(selection_probs)
4 selected = population[np.random.choice(len(population), size=
  population_size, p=selection_probs)]
5
6 # Elitism: Retain the best-performing individuals
7 elite_indices = np.argsort(fitness)[:num_elites]
8 elite_individuals = population[elite_indices]
9 population = np.vstack((elite_individuals, np.array(offspring))
  )

```

Listing 8: Exploitation Code

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

```

1 # Convergence is determined by running the algorithm for a fixed
  number of generations
2 for generation in range(generations):
3     # Evaluate fitness for the current population
4     fitness = np.array([objective_function(ind) for ind in
      population])
5
6     # Track the best fitness in the current generation
7     best_individual_gen = population[np.argmin([objective_function(
      ind) for ind in population])]
8     best_fitness_gen = objective_function(best_individual_gen)
9     best_fitness_per_generation.append(best_fitness_gen)
10
11     # Print generation progress
12     print(f"Generation_{generation+1} - Best Solution: {
      best_individual_gen}, Fitness: {best_fitness_gen}")
13
14 # Obtain the final best solution and fitness score after all
  generations
15 best_individual = population[np.argmin([objective_function(ind) for
    ind in population])]
16 best_fitness = objective_function(best_individual)
17 print("\nFinal Best Solution:", best_individual)
18 print("Final Fitness Score:", best_fitness)
19
20 # Plot the fitness over generations to visualize convergence
21 plt.plot(range(1, generations + 1), best_fitness_per_generation)

```

```
22 plt.xlabel("Generation")
23 plt.ylabel("Best_Fitness")
24 plt.title("Best_Fitness_Over_Generations")
25 plt.grid(True)
26 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.

```
1 # Plot the fitness over generations
2 plt.plot(range(1, generations + 1), best_fitness_per_generation)
3 plt.xlabel("Generation")
4 plt.ylabel("Best_Fitness")
5 plt.title("Best_Fitness_Over_Generations")
6 plt.grid(True)
7 plt.show()
```

Listing 10: Fitness Plot Visualization Code

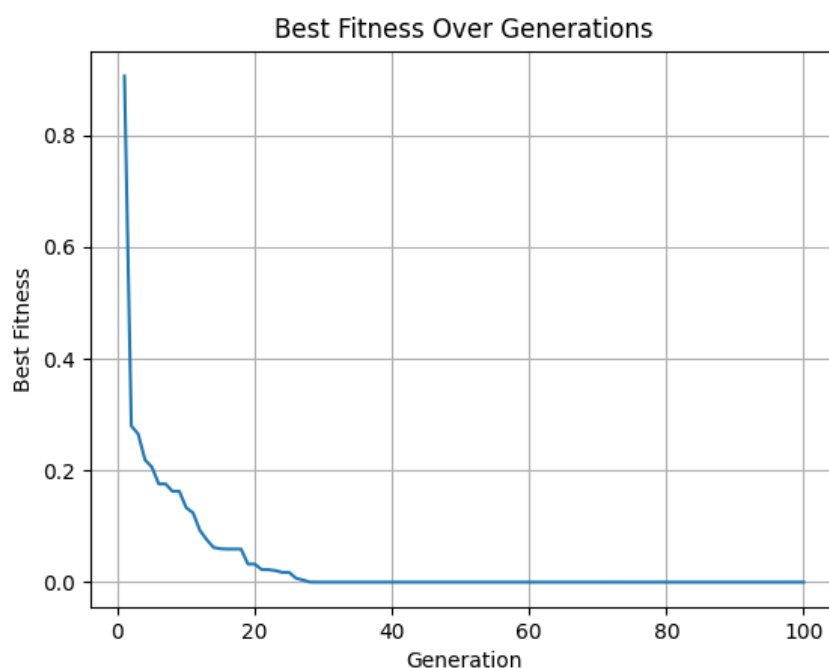


Figure 1: Sample Visualization of Fitness Convergence

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.

```

1  # Tkinter GUI
2  def run_gui():
3      def start_ga():
4          # Retrieve inputs
5          try:
6              objective_choice = int(obj_func_var.get())
7              if objective_choice == 1:
8                  objective_function = sphere_function
9              elif objective_choice == 2:
10                 objective_function = rosenbrock_function
11             elif objective_choice == 3:
12                 objective_function = himmelblau_function
13             elif objective_choice == 4:
14                 objective_function = n3_function
15             elif objective_choice == 5:
16                 objective_function = n5_function
17             elif objective_choice == 6:
18                 func_input = custom_func_entry.get()
19                 objective_function = eval(f"lambda x: {func_input}")
20             else:
21                 messagebox.showerror("Error", "Invalid objective_
22                                     function choice.")
23             return
24
25         num_variables = int(num_vars_entry.get())
26         bounds = []
27         for i in range(num_variables):
28             bounds.append(tuple(map(float, bounds_entries[i].get().
29                                     split(','))))
30
31         population_size = int(pop_size_entry.get())

```

```

31         generations = int(generations_entry.get())
32         mutation_rate = float(mutation_rate_entry.get())
33
34         # Run GA
35         best_solution, best_fitness = genetic_algorithm(
36             objective_function, bounds, population_size, generations
37             , mutation_rate)
38         messagebox.showinfo(
39             "Best_Solution",
40             f"Best_Solution_Found: {best_solution}\nFinal_Fitness_Score
41             : {best_fitness}"
42         )
43     except Exception as e:
44         messagebox.showerror("Error", f"An_error_occurred: {e}")
45
46 # Tkinter window setup
47 root = tk.Tk()
48 root.title("Genetic_Algorithm_GUI")
49
50 # Objective function selection
51 tk.Label(root, text="Select_Objective_Function").grid(row=0, column
52 =0, columnspan=2)
53 obj_func_var = tk.StringVar(value="1")
54 tk.Radiobutton(root, text="Sphere_Function", variable=obj_func_var,
55 value="1").grid(row=1, column=0)
56 tk.Radiobutton(root, text="Rosenbrock_Function", variable=
57 obj_func_var, value="2").grid(row=2, column=0)
58 tk.Radiobutton(root, text="Himmelblau_Function", variable=
59 obj_func_var, value="3").grid(row=3, column=0)
60 tk.Radiobutton(root, text="N=3_Function", variable=obj_func_var,
61 value="4").grid(row=4, column=0)
62 tk.Radiobutton(root, text="N=5_Function", variable=obj_func_var,
63 value="5").grid(row=5, column=0)
64 tk.Radiobutton(root, text="Custom_Function", variable=obj_func_var,
65 value="6").grid(row=6, column=0)
66 custom_func_entry = tk.Entry(root, width=40)
67 custom_func_entry.grid(row=6, column=1)
68
69 # Number of variables
70 tk.Label(root, text="Number_of_Variables:").grid(row=7, column=0)
71 num_vars_entry = tk.Entry(root)
72 num_vars_entry.grid(row=7, column=1)
73
74 # Scrollable frame for bounds
75 tk.Label(root, text="Enter_Bounds_(min,max_for_each_variable):").
76 grid(row=8, column=0, columnspan=2)
77
78 bounds_frame = ttk.Frame(root)
79 bounds_canvas = tk.Canvas(bounds_frame, height=200) # Set a fixed
80 height
81 bounds_scrollbar = ttk.Scrollbar(bounds_frame, orient="vertical",
82 command=bounds_canvas.yview)
83 bounds_scrollable_frame = ttk.Frame(bounds_canvas)
84
85 bounds_scrollable_frame.bind(
86     "<Configure>",
87     lambda e: bounds_canvas.configure(scrollregion=bounds_canvas.
88         bbox("all"))

```

```

75     )
76
77     bounds_canvas.create_window((0, 0), window=bounds_scrollable_frame,
78                                anchor="nw")
79     bounds_canvas.configure(yscrollcommand=bounds_scrollbar.set)
80
81     bounds_frame.grid(row=9, column=0, columnspan=2, sticky="nsew")
82     bounds_canvas.pack(side="left", fill="both", expand=True)
83     bounds_scrollbar.pack(side="right", fill="y")
84
85     # Create entries for bounds
86     bounds_entries = [tk.Entry(bounds_scrollable_frame) for _ in range
87                       (27)]
88     for i, entry in enumerate(bounds_entries):
89         tk.Label(bounds_scrollable_frame, text=f"Var_{i+1}_Bounds:").
90             grid(row=i, column=0, sticky="w")
91         entry.grid(row=i, column=1)
92
93     # Population size, generations, mutation rate
94     tk.Label(root, text="Population_Size:").grid(row=36, column=0)
95     pop_size_entry = tk.Entry(root)
96     pop_size_entry.insert(0, "50")
97     pop_size_entry.grid(row=36, column=1)
98
99     tk.Label(root, text="Generations:").grid(row=37, column=0)
100    generations_entry = tk.Entry(root)
101    generations_entry.insert(0, "100")
102    generations_entry.grid(row=37, column=1)
103
104    tk.Label(root, text="Mutation_Rate:").grid(row=38, column=0)
105    mutation_rate_entry = tk.Entry(root)
106    mutation_rate_entry.insert(0, "0.1")
107    mutation_rate_entry.grid(row=38, column=1)
108
109    # Start button
110    tk.Button(root, text="Start_Genetic_Algorithm", command=start_ga).
111        grid(row=39, column=0, columnspan=2)
112
113    root.mainloop()
114
115    # Run the GUI
116    if __name__ == "__main__":
117        run_gui()

```

Listing 11: Tkinter GUI Setup Code

Genetic Algorithm GUI

Select Objective Function

☐ Sphere Function

☒ Rosenbrock Function

☐ Himmelblau Function

☐ Custom Function

Number of Variables: 2

Enter Bounds (min,max for each variable):

-5,5

-5,5

Population Size: 50

Generations: 100

Mutation Rate: 0.1

Start Genetic Algorithm

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:

```
1 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:

```
1 pip install numpy matplotlib
```

You can confirm the installation by running:

```
1 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:

```
1 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:

```
1 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

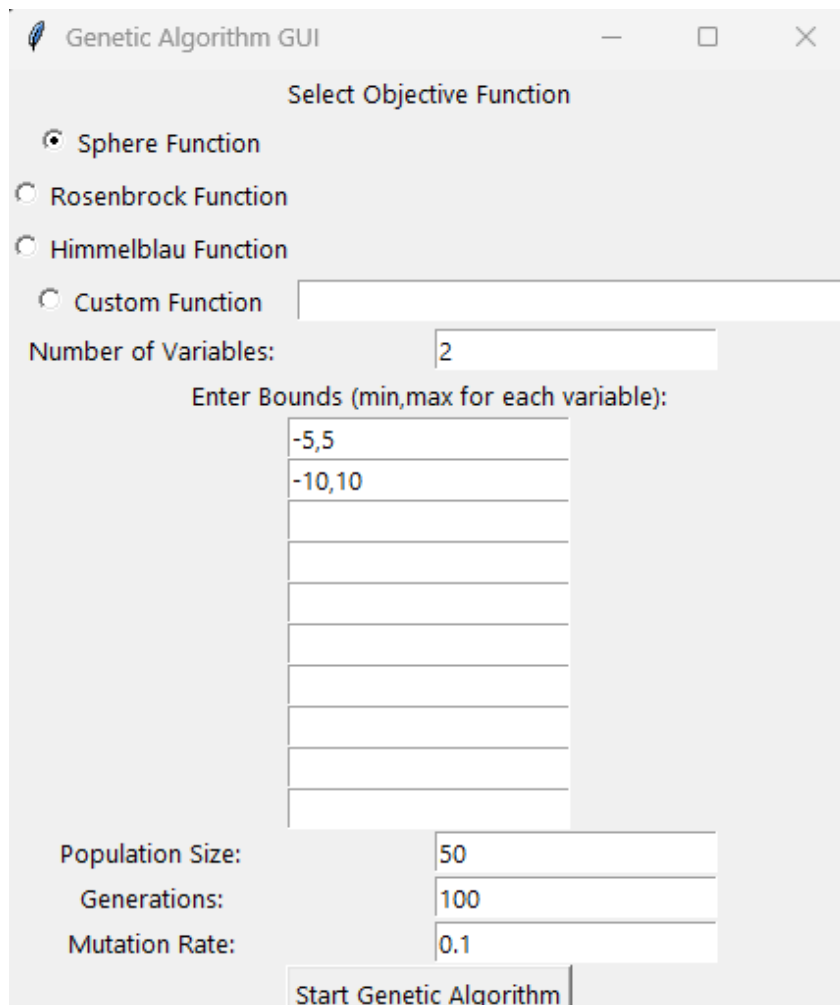
This section presents the results of the Simple Genetic Algorithm (SGA) tested on three classical benchmark objective functions: Sphere Function, Rosenbrock Function, and Himmelblau Function. The results are illustrated through visualizations and table of the search process and the convergence of the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



The screenshot shows a window titled "Genetic Algorithm GUI" with a "Select Objective Function" section. The "Sphere Function" is selected with a radio button. Below this, there are input fields for "Number of Variables:" (set to 2), "Population Size:" (set to 50), "Generations:" (set to 100), and "Mutation Rate:" (set to 0.1). A "Start Genetic Algorithm" button is at the bottom. The "Enter Bounds (min,max for each variable):" section contains a table with two rows of input fields, the first two containing "-5,5" and "-10,10".

Enter Bounds (min,max for each variable):	
-5,5	
-10,10	

Figure 3: Input Process for Sphere Function

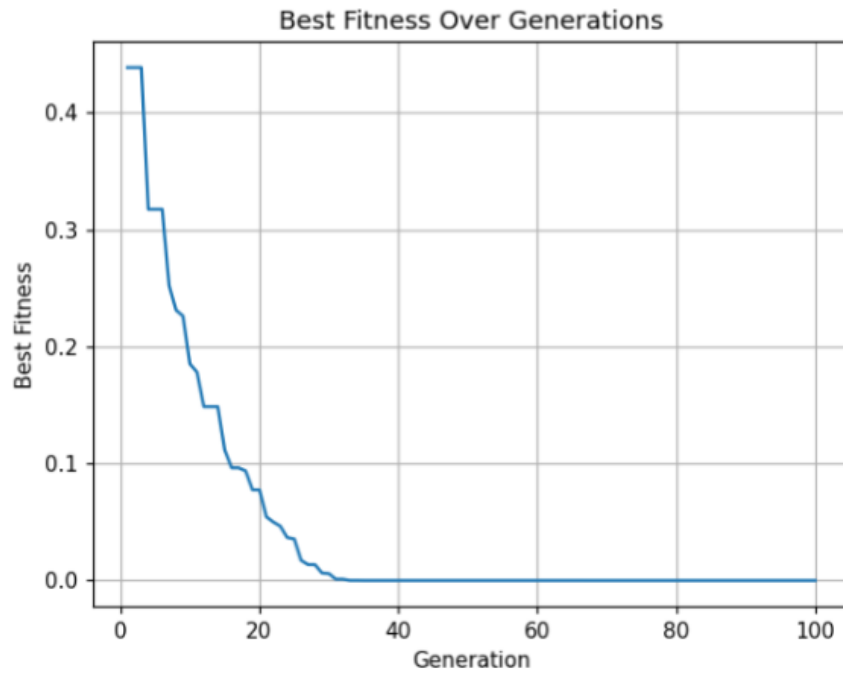


Figure 4: Fitness Visualization for Sphere Function

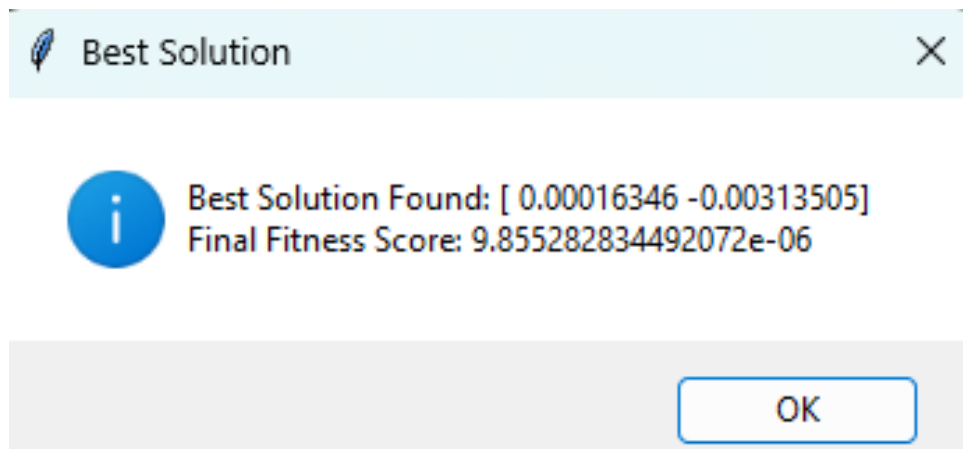


Figure 5: Best Solution and Final Fitness Score

```

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

```

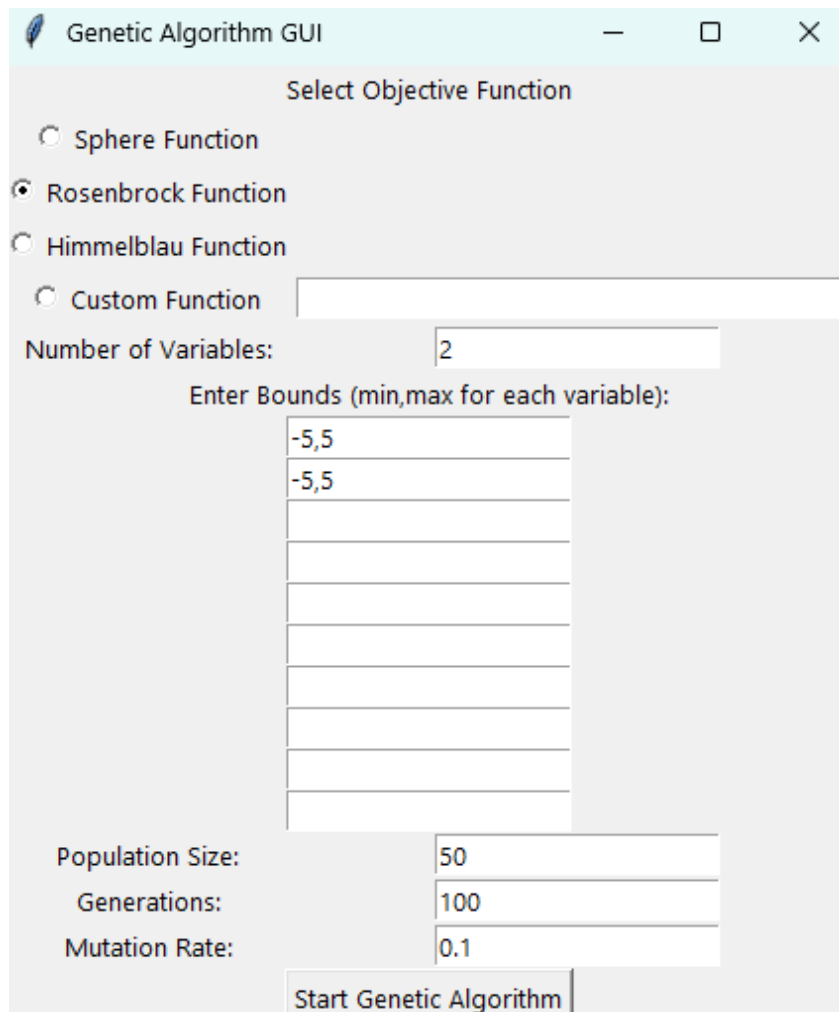
Figure 6: Generation-wise Best Solutions and Fitness Convergence

5.2 Rosenbrock Function

The Rosenbrock function is defined as:

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

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.



The screenshot shows a window titled "Genetic Algorithm GUI" with standard window controls. The "Select Objective Function" section has four radio buttons: "Sphere Function", "Rosenbrock Function" (which is selected), "Himmelblau Function", and "Custom Function". Below this, the "Number of Variables:" is set to 2. The "Enter Bounds (min,max for each variable):" section contains a table with 10 rows. The first two rows are filled with "-5,5", and the remaining eight rows are empty. At the bottom, there are three input fields: "Population Size:" set to 50, "Generations:" set to 100, and "Mutation Rate:" set to 0.1. A "Start Genetic Algorithm" button is located at the bottom right.

Figure 7: Input Process for Rosenbrock Function

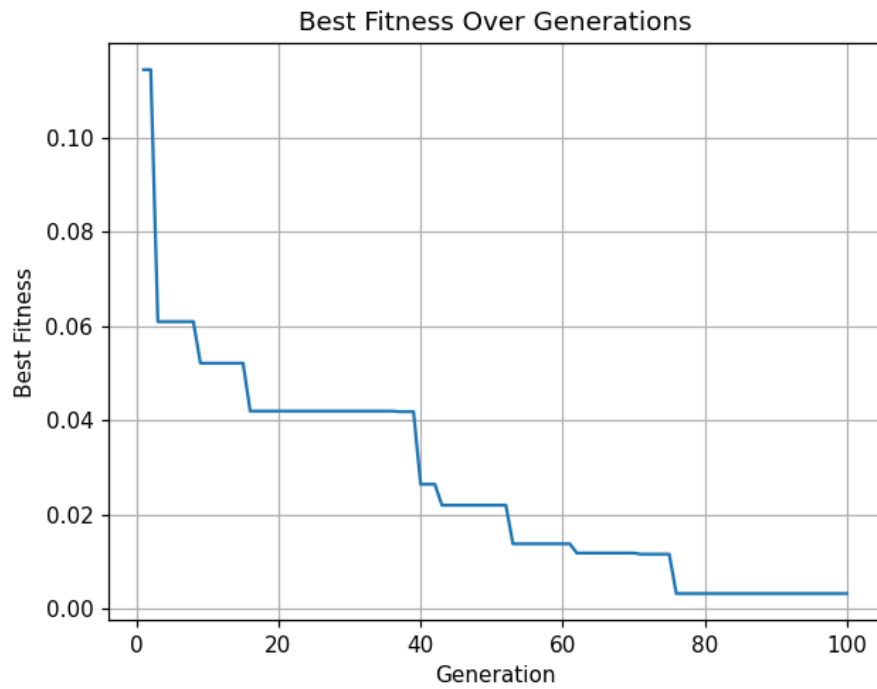


Figure 8: Fitness Visualization for Rosenbrock Function

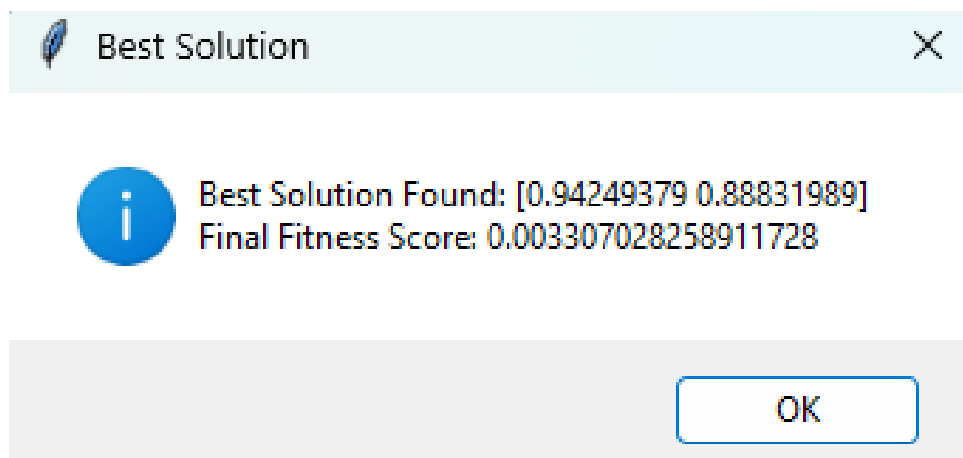


Figure 9: Best Solution and Final Fitness Score


```
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.

The screenshot shows a window titled "Genetic Algorithm GUI" with a light blue header. Below the header, there is a section titled "Select Objective Function" with four radio button options: "Sphere Function", "Rosenbrock Function", "Himmelblau Function" (which is selected), and "Custom Function". Below this, there is a text input field for "Number of Variables:" with the value "2" entered. Underneath, there is a section titled "Enter Bounds (min,max for each variable):" followed by a vertical stack of eight text input fields. The first two fields contain "-3,3" and "-5,5" respectively, while the others are empty. At the bottom, there are three more input fields: "Population Size:" with "50", "Generations:" with "100", and "Mutation Rate:" with "0.1". A "Start Genetic Algorithm" button is located at the bottom right of the form.

Figure 11: Input Process for Himmelblau Function

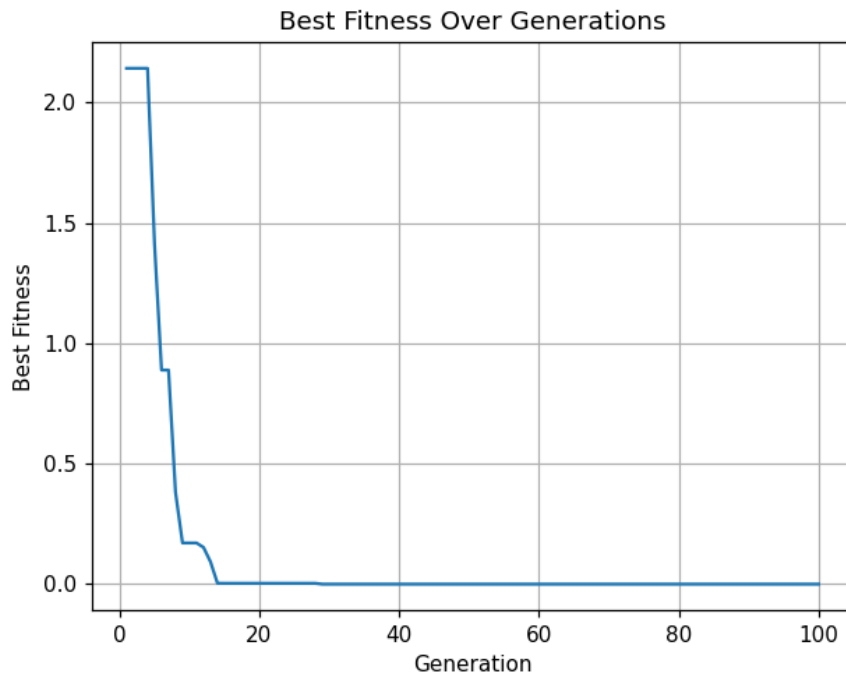


Figure 12: Fitness Visualization for Himmelblau Function

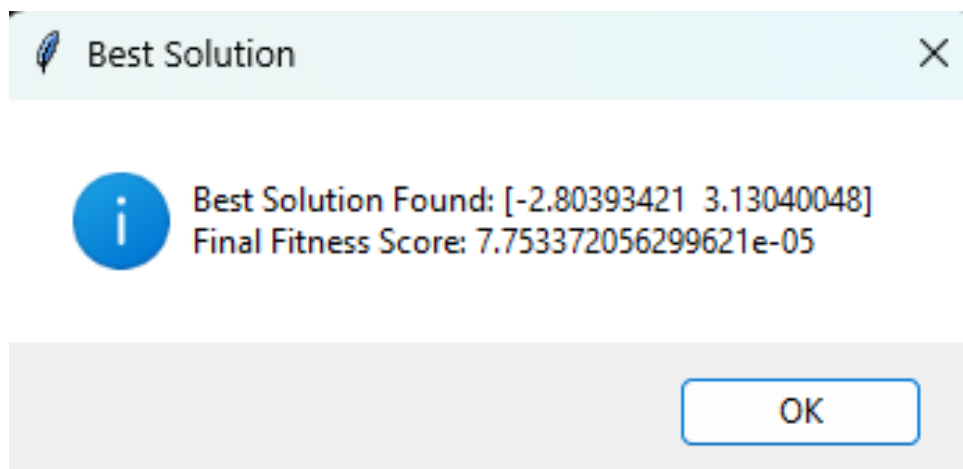


Figure 13: Best Solution and Final Fitness Score

```

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

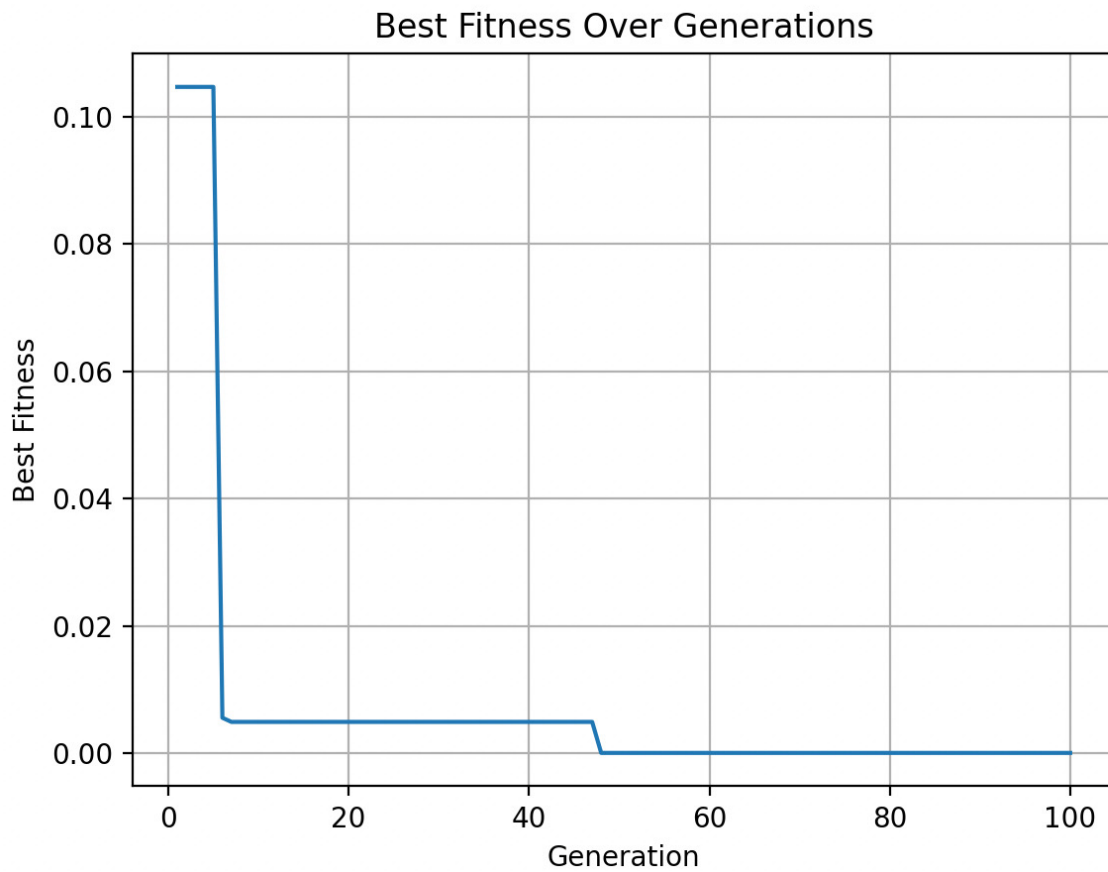


Figure 16: Fitness Visualization for Custom Function 1

Final Best Solution: [3.73957 -2.08694621 -3.52091987 -1.09653401 1.61439843 2.27276019
2.77393251 -4.05616271 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

22

6.2 Custom Function 2

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

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

```

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

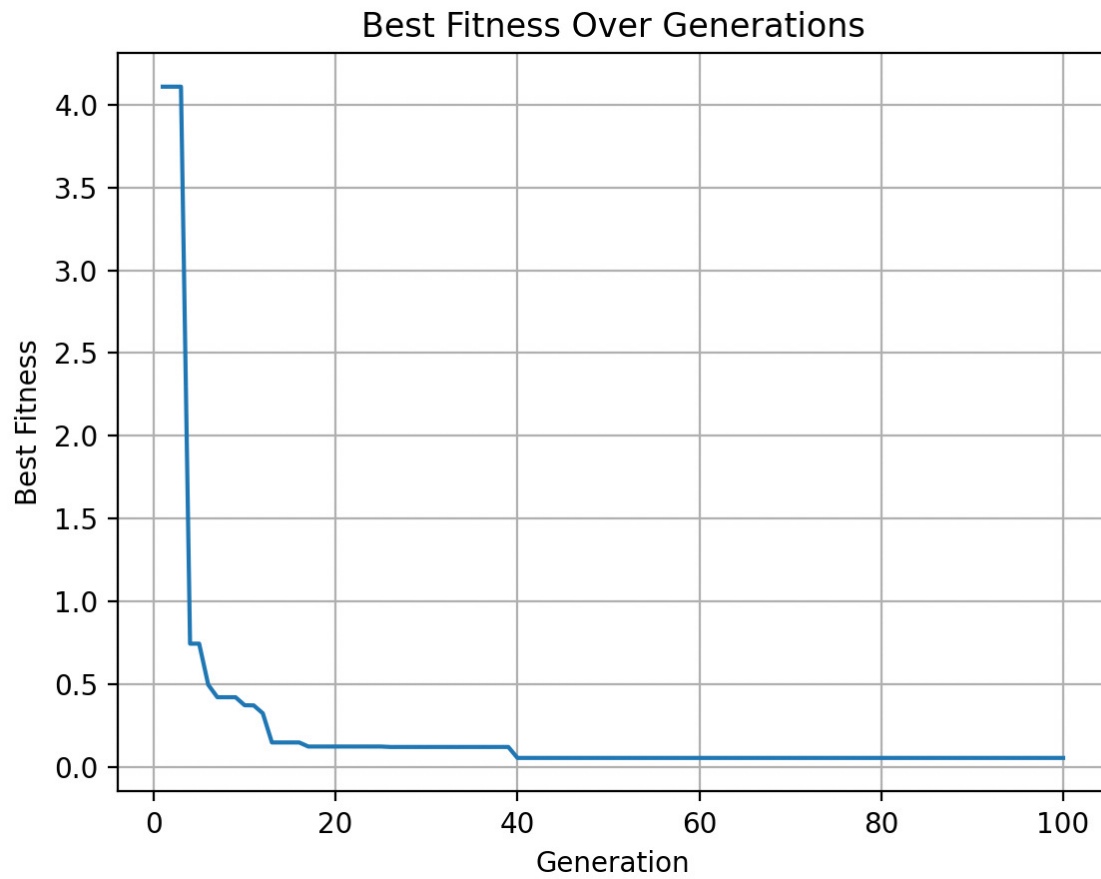


Figure 19: Fitness Visualization for Custom Function 2

```

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.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import tkinter as tk
4 from tkinter import messagebox, simpledialog, ttk
5
6 # Objective functions
7 def sphere_function(x):
8     return np.sum(x**2)
9
10 def rosenbrock_function(x):
11     return np.sum(100 * (x[1:] - x[:-1]**2)**2 + (1 - x[:-1])**2)
12
13 def himmelblau_function(x):
14     return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2
15
16 def function_(x):
17     return np.abs((x[0]*x[1]) + (2*x[2]*x[3]) + (2*x[4]*x[5]) + (2*x
18         [6]*x[7]))
19
20 def n3_function(x):
21     return np.abs(
22         x[0] * x[1] +
23         2 * x[2] * x[3] +
24         2 * x[4] * x[5] +
25         2 * x[6] * x[7] +
26         2 * x[8] * x[9] +
27         2 * x[10] * x[11] +
28         2 * x[12] * x[13] +
29         2 * x[14] * x[15] +
30         x[16] * x[17] +
31         8
32     )
33
34 def n5_function(x):
35     return (
36         np.abs(
37             x[0] * x[2] +
38             x[1] * x[2] +
39             2 * x[3] * x[5] +
40             2 * x[4] * x[5] +
41             2 * x[6] * x[8] +
42             2 * x[7] * x[8] +
43             2 * x[9] * x[11] +
44             2 * x[10] * x[11] +
45             2 * x[12] * x[14] +
46             2 * x[13] * x[14] +
47             2 * x[15] * x[17] +
48             2 * x[16] * x[17] +
49             2 * x[18] * x[20] +
50             2 * x[19] * x[20] +
51             2 * x[21] * x[23] +
52             2 * x[22] * x[23] +

```

```

52         x[24] * x[26] +
53         x[25] * x[26] +
54         8
55     ) +
56     np.abs (
57         x[0] * x[1] +
58         x[1] * x[2] +
59         2 * x[3] * x[4] +
60         2 * x[4] * x[5] +
61         2 * x[6] * x[7] +
62         2 * x[7] * x[8] +
63         2 * x[9] * x[10] +
64         2 * x[10] * x[11] +
65         2 * x[12] * x[13] +
66         2 * x[13] * x[14] +
67         2 * x[15] * x[16] +
68         2 * x[16] * x[17] +
69         2 * x[18] * x[19] +
70         2 * x[19] * x[20] +
71         2 * x[21] * x[22] +
72         2 * x[22] * x[23] +
73         x[24] * x[25] +
74         x[25] * x[26] +
75         8
76     )
77 )
78
79 def genetic_algorithm(
80     objective_function, bounds, population_size=50, generations=100,
81     mutation_rate=0.1
82 ):
83     # Number of variables in problem
84     dim = len(bounds)
85
86     # Create population with random variables within bounds
87     population = np.random.uniform(
88         [b[0] for b in bounds], # Lower bounds
89         [b[1] for b in bounds], # Upper bounds
90         (population_size, dim) # Shape of population
91     )
92
93     num_elites = 4
94     best_fitness_per_generation = []
95
96     # Iterate through each generation
97     for generation in range(generations):
98         # Obtain fitness of every individual
99         fitness = np.array([objective_function(ind) for ind in
100             population])
101
102         # Keep track of elites in the generation
103         elite_indices = np.argsort(fitness)[:num_elites]
104         elite_individuals = population[elite_indices]
105
106         # Calculate probabilities for biased wheel
107         total_fitness = np.sum(fitness)
108         selection_probs = (1 / (fitness + 1e-6)) / total_fitness

```

```

108     # Normalize (ensure probabilities add to 1)
109     selection_probs /= np.sum(selection_probs)
110
111     # Select mating pool based on biased wheel
112     selected = population[np.random.choice(len(population), size=
        population_size, p=selection_probs)]
113
114     # Create offspring using crossover and mutation
115     offspring = []
116     for _ in range((population_size - num_elites) // 2):
117         # Randomly select 2 parents from mating pool
118         p1, p2 = selected[np.random.choice(len(selected), size=2)]
119
120         # Create crossover point and make children based on parents
        and crossover point
121         crossover_point = np.random.randint(1, dim)
122         child1 = np.concatenate((p1[:crossover_point], p2[
            crossover_point:]))
123         child2 = np.concatenate((p2[:crossover_point], p1[
            crossover_point:]))
124
125         # Chance to mutate a child
126         if np.random.rand() < mutation_rate:
127             child1 += np.random.uniform(-0.1, 0.1, dim)
128         if np.random.rand() < mutation_rate:
129             child2 += np.random.uniform(-0.1, 0.1, dim)
130
131         # Add children to list
132         offspring.append(child1)
133         offspring.append(child2)
134
135     # Replace old population with elites and children
136     population = np.vstack((elite_individuals, np.array(offspring))
        )
137
138
139     # Print generation and number of individuals
140     print(f"Generation_{generation+1}-_Number_of_Individuals:_{
        len(population)}")
141
142     # Obtain best individual and fitness in current generation
143     best_individual_gen = population[np.argmin([objective_function(
        ind) for ind in population])]
144     best_fitness_gen = objective_function(best_individual_gen)
145     best_fitness_per_generation.append(best_fitness_gen)
146
147     # Print best individual and fitness score in each generation
148     print(f"Generation_{generation+1}-_Best_Solution:_{
        best_individual_gen},_Fitness:_{best_fitness_gen}")
149
150     # Obtain best overall individual and fitness score
151     best_individual = population[np.argmin([objective_function(ind) for
        ind in population])]
152     best_fitness = objective_function(best_individual)
153
154     # Print overall best individual and fitness score
155     print("\nFinal_Best_Solution:", best_individual)
156     print("Final_Fitness_Score:", best_fitness)

```

```

157
158     # Plot the fitness over generations
159     plt.plot(range(1, generations + 1), best_fitness_per_generation)
160     plt.xlabel("Generation")
161     plt.ylabel("Best_Fitness")
162     plt.title("Best_Fitness_Over_Generations")
163     plt.grid(True)
164     plt.show()
165
166     return best_individual, best_fitness
167
168 # Tkinter GUI
169 def run_gui():
170     def start_ga():
171         # Retrieve inputs
172         try:
173             objective_choice = int(obj_func_var.get())
174             if objective_choice == 1:
175                 objective_function = sphere_function
176             elif objective_choice == 2:
177                 objective_function = rosenbrock_function
178             elif objective_choice == 3:
179                 objective_function = himmelblau_function
180             elif objective_choice == 4:
181                 objective_function = n3_function
182             elif objective_choice == 5:
183                 objective_function = n5_function
184             elif objective_choice == 6:
185                 func_input = custom_func_entry.get()
186                 objective_function = eval(f"lambda x: {func_input}")
187             else:
188                 messagebox.showerror("Error", "Invalid objective_
189                                     function_choice.")
190                 return
191
192             num_variables = int(num_vars_entry.get())
193             bounds = []
194             for i in range(num_variables):
195                 bounds.append(tuple(map(float, bounds_entries[i].get().
196                                     split(','))))
197
198             population_size = int(pop_size_entry.get())
199             generations = int(generations_entry.get())
200             mutation_rate = float(mutation_rate_entry.get())
201
202             # Run GA
203             best_solution, best_fitness = genetic_algorithm(
204                 objective_function, bounds, population_size, generations
205                 , mutation_rate)
206             messagebox.showinfo(
207                 "Best_Solution",
208                 f"Best_Solution_Found: {best_solution}\nFinal_Fitness_Score
209                 : {best_fitness}"
210             )
211         except Exception as e:
212             messagebox.showerror("Error", f"An error occurred: {e}")
213
214 # Tkinter window setup

```

```

210 root = tk.Tk()
211 root.title("Genetic_Algorithm_GUI")
212
213 # Objective function selection
214 tk.Label(root, text="Select_Objective_Function").grid(row=0, column
    =0, columnspan=2)
215 obj_func_var = tk.StringVar(value="1")
216 tk.Radiobutton(root, text="Sphere_Function", variable=obj_func_var,
    value="1").grid(row=1, column=0)
217 tk.Radiobutton(root, text="Rosenbrock_Function", variable=
    obj_func_var, value="2").grid(row=2, column=0)
218 tk.Radiobutton(root, text="Himmelblau_Function", variable=
    obj_func_var, value="3").grid(row=3, column=0)
219 tk.Radiobutton(root, text="N=3_Function", variable=obj_func_var,
    value="4").grid(row=4, column=0)
220 tk.Radiobutton(root, text="N=5_Function", variable=obj_func_var,
    value="5").grid(row=5, column=0)
221 tk.Radiobutton(root, text="Custom_Function", variable=obj_func_var,
    value="6").grid(row=6, column=0)
222 custom_func_entry = tk.Entry(root, width=40)
223 custom_func_entry.grid(row=6, column=1)
224
225 # Number of variables
226 tk.Label(root, text="Number_of_Variables:").grid(row=7, column=0)
227 num_vars_entry = tk.Entry(root)
228 num_vars_entry.grid(row=7, column=1)
229
230 # Scrollable frame for bounds
231 tk.Label(root, text="Enter_Bounds_(min,max_for_each_variable):").
    grid(row=8, column=0, columnspan=2)
232
233 bounds_frame = ttk.Frame(root)
234 bounds_canvas = tk.Canvas(bounds_frame, height=200) # Set a fixed
    height
235 bounds_scrollbar = ttk.Scrollbar(bounds_frame, orient="vertical",
    command=bounds_canvas.yview)
236 bounds_scrollable_frame = ttk.Frame(bounds_canvas)
237
238 bounds_scrollable_frame.bind(
239     "<Configure>",
240     lambda e: bounds_canvas.configure(scrollregion=bounds_canvas.
        bbox("all"))
241 )
242
243 bounds_canvas.create_window((0, 0), window=bounds_scrollable_frame,
    anchor="nw")
244 bounds_canvas.configure(yscrollcommand=bounds_scrollbar.set)
245
246 bounds_frame.grid(row=9, column=0, columnspan=2, sticky="nsew")
247 bounds_canvas.pack(side="left", fill="both", expand=True)
248 bounds_scrollbar.pack(side="right", fill="y")
249
250 # Create entries for bounds
251 bounds_entries = [tk.Entry(bounds_scrollable_frame) for _ in range
    (27)]
252 for i, entry in enumerate(bounds_entries):
253     tk.Label(bounds_scrollable_frame, text=f"Var_{i+1}_Bounds:").
        grid(row=i, column=0, sticky="w")

```



```
254     entry.grid(row=i, column=1)
255
256     # Population size, generations, mutation rate
257     tk.Label(root, text="Population Size:").grid(row=36, column=0)
258     pop_size_entry = tk.Entry(root)
259     pop_size_entry.insert(0, "50")
260     pop_size_entry.grid(row=36, column=1)
261
262     tk.Label(root, text="Generations:").grid(row=37, column=0)
263     generations_entry = tk.Entry(root)
264     generations_entry.insert(0, "100")
265     generations_entry.grid(row=37, column=1)
266
267     tk.Label(root, text="Mutation Rate:").grid(row=38, column=0)
268     mutation_rate_entry = tk.Entry(root)
269     mutation_rate_entry.insert(0, "0.1")
270     mutation_rate_entry.grid(row=38, column=1)
271
272     # Start button
273     tk.Button(root, text="Start Genetic Algorithm", command=start_ga).
274         grid(row=39, column=0, columnspan=2)
275
276     root.mainloop()
277
278 # Run the GUI
279 if __name__ == "__main__":
280     run_gui()
```

8. Project Resources

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