1.0 Introduction to the Selected Problem

The optimization of complex, multimodal functions is a fundamental challenge in computational mathematics and engineering, particularly when dealing with nonlinear surfaces that contain multiple peaks and valleys. This report focuses on maximizing the two-dimensional "peak" function defined by $f(x, y) = (1 - x)^2 e^{-x^2-(y+1)^2} - (x - x^3 - y^3)e^{-x^2-y^2}$ (using slide 31 as prolem), where x and y are bounded within the interval $[-3,3]$. Traditional gradient-based optimization methods such as Newton-Raphson or conjugate gradient descent often struggle with such functions due to their sensitivity to initial conditions and susceptibility to becoming trapped in local optima.

1.1 Justification for using Genetic Algorithms

The Genetic Algorithm (GA) employing Roulette Wheel Selection and binary encoding was selected for this optimization problem due to several key advantages. Firstly, GA's population-based approach provides inherent diversity, making it particularly suitable for navigating multimodal landscapes and avoiding local optima. Furthermore, its global search capability, driven by the crossover and mutation operators, enables effective exploration of the solution space without requiring gradient information, making it applicable to non-differentiable functions. The specific choice of Roulette Wheel Selection complements this by maintaining population diversity through probabilistic selection, ensuring that all solutions have a chance of being selected based on fitness. This mechanism preserves promising solutions while simultaneously allowing exploration of potentially better regions, achieving a balanced exploitation/exploration trade-off essential for complex optimization landscapes. Finally, binary encoding was adopted for its efficient representation of continuous variables and because it enables the direct application of standard genetic operators like crossover and mutation.

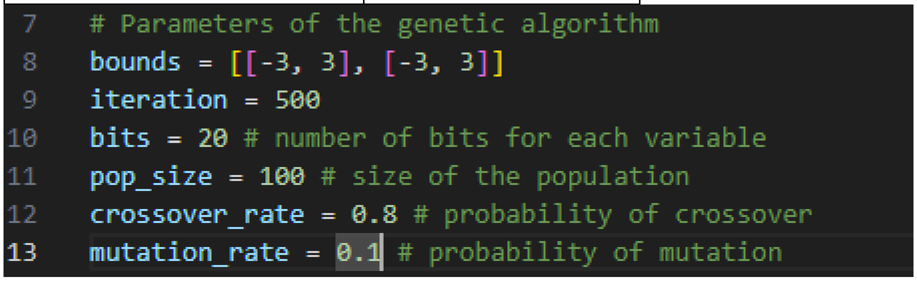
1.2 Objective

The primary objectives of this study are:

- To develop and fine-tune a Generic Algorithm for maximizing the peak function.
- To analyze the algorithm's convergence behavior and get the optimal solution.

1.3 Parameter Selection

Parameter	value
Bounds	[-3, 3]
Iteration	500
Bits per variable	20
Population size	100
Crossover rate	0.6
Mutation rate	0.1
Selection	Roulette wheel



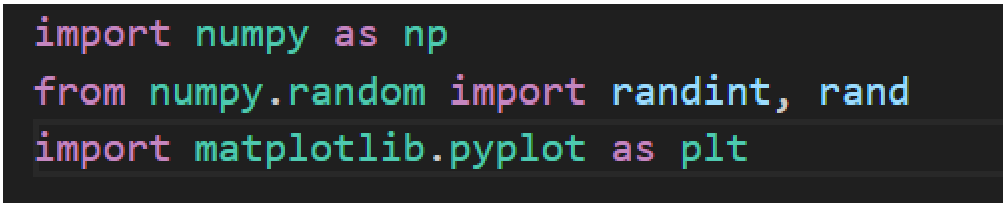
```
# Parameters of the genetic algorithm
bounds = [[-3, 3], [-3, 3]]
iteration = 500
bits = 20 # number of bits for each variable
pop_size = 100 # size of the population
crossover_rate = 0.8 # probability of crossover
mutation_rate = 0.1 # probability of mutation
```

1.4 Source Code

The genetic algorithm code is retrieved from *https://learnwithpanda.com/2021/04/19/binary-genetic-algorithm-in-python/*. The example used by the author is finding the minimum value of the function. Therefore, in this study the code is being modified to suit the finding peak/maximum value of function.

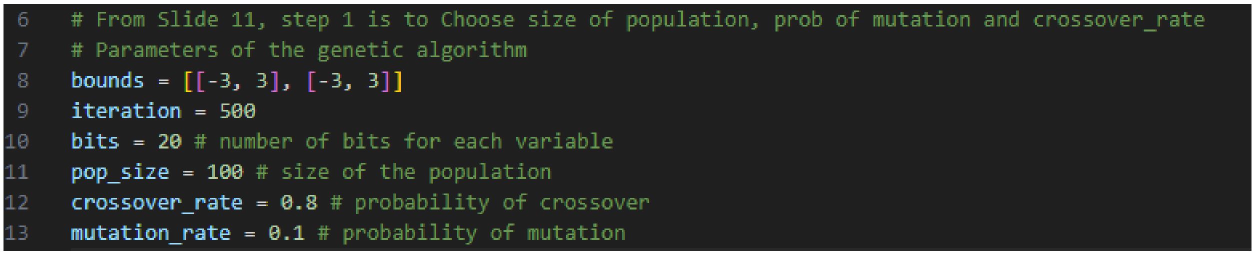
2.0 Implementation and Explanation

- Import necessary libraries



```python
import numpy as np
from numpy.random import randint, rand
import matplotlib.pyplot as plt
```

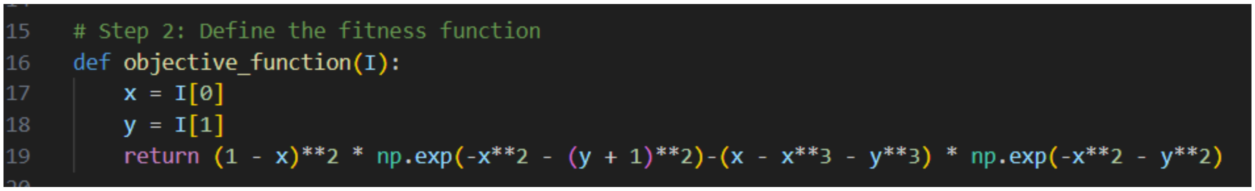
- Initialize the parameters and population



```python
# From Slide 11, step 1 is to Choose size of population, prob of mutation and crossover_rate
# Parameters of the genetic algorithm
bounds = [[-3, 3], [-3, 3]]
iteration = 500
bits = 20 # number of bits for each variable
pop_size = 100 # size of the population
crossover_rate = 0.8 # probability of crossover
mutation_rate = 0.1 # probability of mutation
```

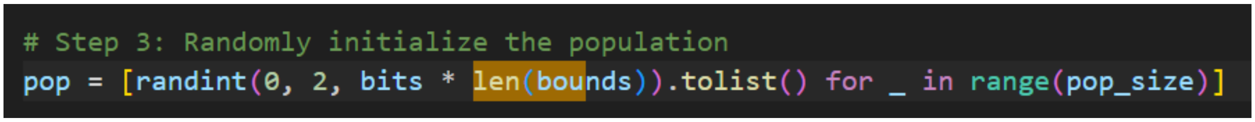
The parameters bounds, iterations, bits per variable, population size, crossover rate, mutation rate are set.

- Define fitness function to measure performance of individual chromosomes.



```python
# Step 2: Define the fitness function
def objective_function(I):
    x = I[0]
    y = I[1]
    return (1 - x)**2 * np.exp(-x**2 - (y + 1)**2)-(x - x**3 - y**3) * np.exp(-x**2 - y**2)
```

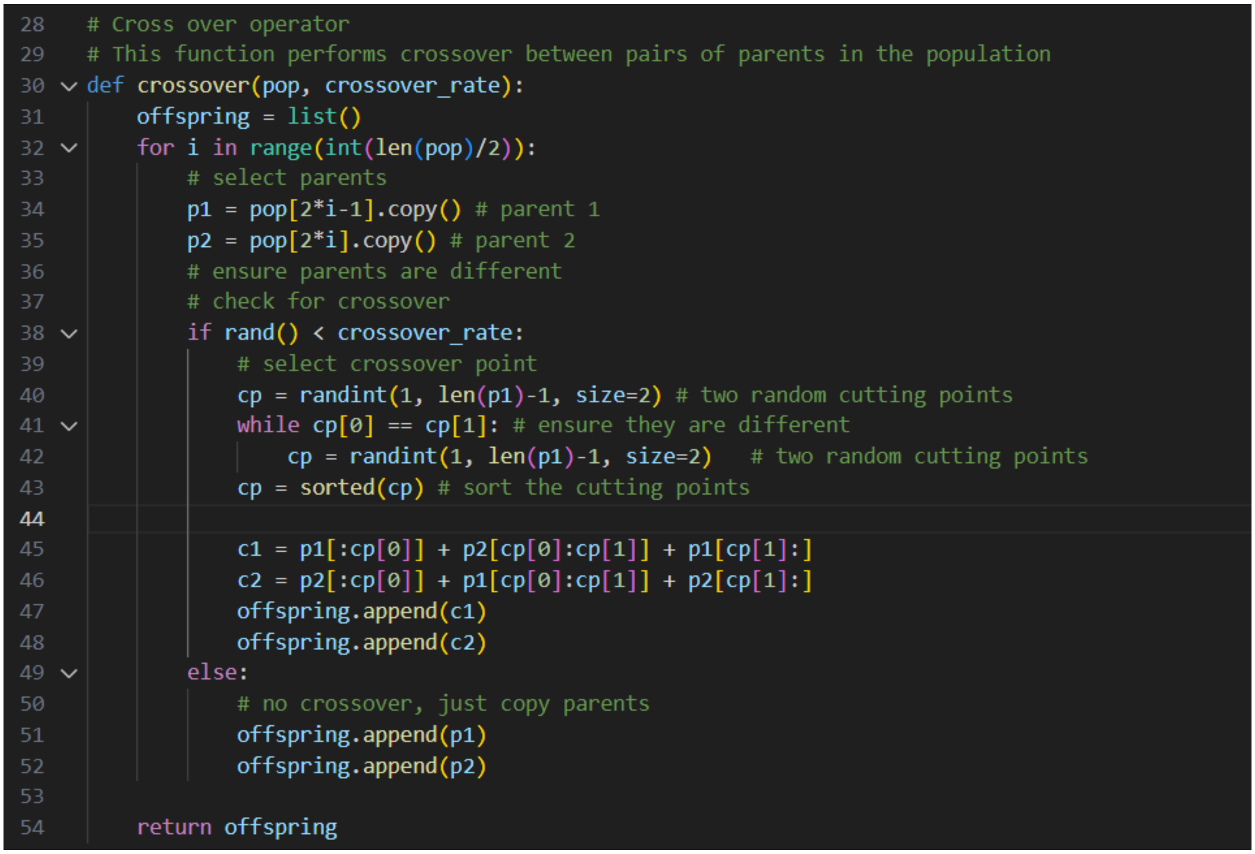
- Randomly generate initial population of chromosomes of sized N.



```python
# Step 3: Randomly initialize the population
pop = [randint(0, 2, bits * len(bounds)).tolist() for _ in range(pop_size)]
```

The initial population ‘pop’ is generated as a list of ‘pop_size’ chromosomes. Each chromosome is a binary string of length `bits * len(bounds)` here is 40 bits, 20 for x and 20 for y. The ‘randint(0,2, ..)’ is to generates the random integer between 0 and 1 (2 is excluded).

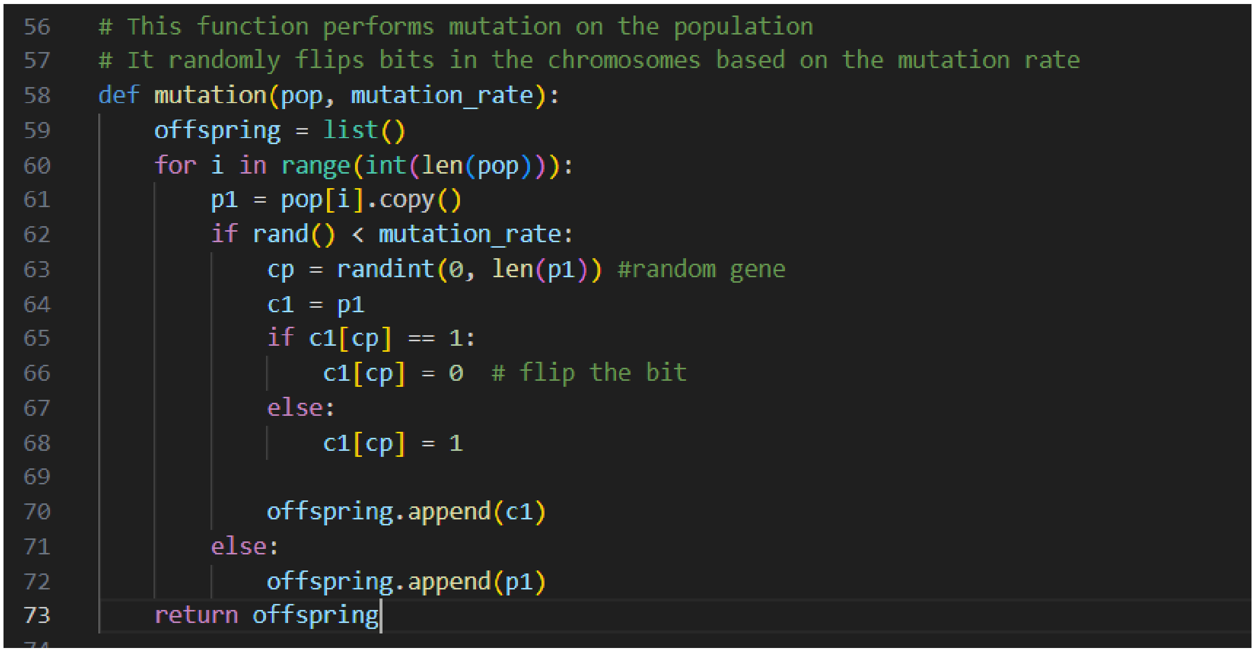
- Define the crossover function



```
28  # Cross over operator
29  # This function performs crossover between pairs of parents in the population
30  def crossover(pop, crossover_rate):
31      offspring = list()
32      for i in range(int(len(pop)/2)):
33          # select parents
34          p1 = pop[2*i-1].copy() # parent 1
35          p2 = pop[2*i].copy() # parent 2
36          # ensure parents are different
37          # check for crossover
38          if rand() < crossover_rate:
39              # select crossover point
40              cp = randint(1, len(p1)-1, size=2) # two random cutting points
41              while cp[0] == cp[1]: # ensure they are different
42                  cp = randint(1, len(p1)-1, size=2)   # two random cutting points
43              cp = sorted(cp) # sort the cutting points
44
45              c1 = p1[:cp[0]] + p2[cp[0]:cp[1]] + p1[cp[1]:]
46              c2 = p2[:cp[0]] + p1[cp[0]:cp[1]] + p2[cp[1]:]
47              offspring.append(c1)
48              offspring.append(c2)
49          else:
50              # no crossover, just copy parents
51              offspring.append(p1)
52              offspring.append(p2)
53
54      return offspring
```

The `crossover` function is called with the current population and crossover rate (probability of crossover occurring, here is choose 0.8). The function is then pairs parents consecutively, line34 and 35 (parent1 = pop[0] and parent2 = pop[1], then pop[2] and pop[3], etc.). Here is using “.copy()” to prevent modifying originals. Next, in line 38, for each pair, checks if crossover should occur by generating a random number between 0 and 1 and comparing it to the ‘crossover_rate’. Line 40 generates two random integers between 1 and the length of parent chromosome minus 1. The ‘size=2’ tells the randint to return an array of 2 values. For line41 and 42, it keeps generating new points until they are different. This is important because having the same point twice would result in an invalid crossover. After that, line 43 is sort the crossover points in ascending order. Line 45 and 46 is create two new offspring chromosomes child c1 and child c2 by combining the segment from both parents. For c1(line 45), the first segment ‘p1[:cp[0]]’ is from parent 1, middle segment ‘p2[cp[0]:cp[1]] is from parent 2 and last segment ‘p1[cp[1]:]’ is from parent 1. For c2, the segment is inverted from c1 where first segment from parent, middle segment from parent 1 and last segment from parent 2. After that, line 47 and 48, is add both new chromosomes to the offspring population. Line 38 to 48 is performs a two-point crossover. Two distinct crossover points are randomly chosen, and the segments between these points are swapped to create two offspring. If no crossover occur, the parents are copied as offspring. The offspring list is returned.

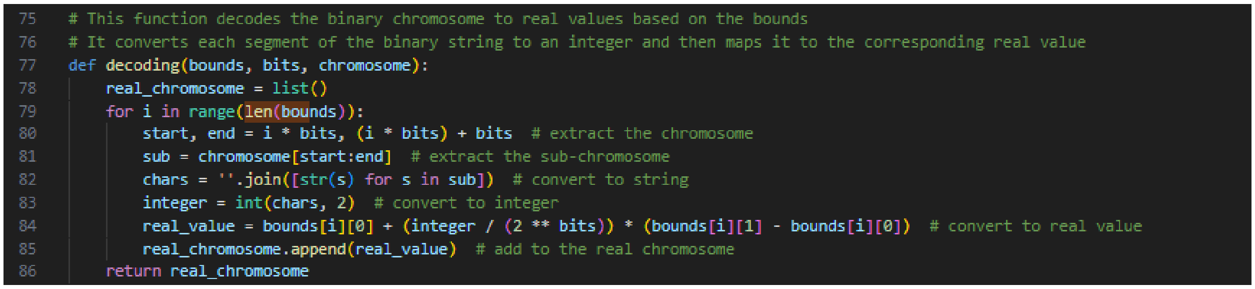
- Define the mutation function



```
56  # This function performs mutation on the population
57  # It randomly flips bits in the chromosomes based on the mutation rate
58  def mutation(pop, mutation_rate):
59      offspring = list()
60      for i in range(int(len(pop))):
61          p1 = pop[i].copy()
62          if rand() < mutation_rate:
63              cp = randint(0, len(p1)) #random gene
64              c1 = p1
65              if c1[cp] == 1:
66                  c1[cp] = 0  # flip the bit
67              else:
68                  c1[cp] = 1
69  
70              offspring.append(c1)
71          else:
72              offspring.append(p1)
73      return offspring
```

The `mutation` function is called on the offspring population with probability of mutation occurring for each chromosome. In line 59, we first Creates an empty list called ‘offspring’ to store mutated chromosomes. Next applied for loop to Iterates through each chromosome in the population. Then in line 61, ‘.copy’ makes a copy of each chromosome to avoid modifying the original. Next is bit flip mutation, which is common mutation operator in genetic algorithm. First checks if mutation should occur by comparing a random number (0-1) against the ‘mutation_rate’. This happens at line 62. So if mutation occurs, selects a random position in the chromosome (line63), and creates a reference to the parent chromosome (line64). Flips the bit at the selected position (0 to 1 or 1 to 0). Then store the mutated chromosomes to the offspring population. If mutation does not occur (line 71,72), the chromosome is copied unchanged. And Lastly, the mutated offspring are returned.

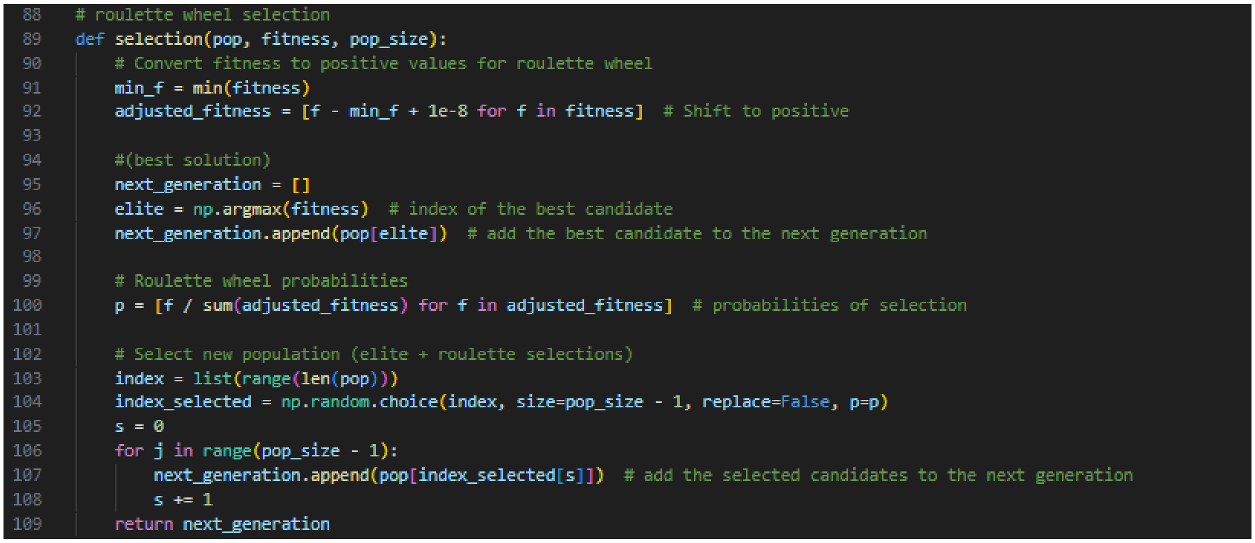
- Define the decoding function



```
75  # This function decodes the binary chromosome to real values based on the bounds
76  # It converts each segment of the binary string to an integer and then maps it to the corresponding real value
77  def decoding(bounds, bits, chromosome):
78      real_chromosome = list()
79      for i in range(len(bounds)):
80          start, end = i * bits, (i * bits) + bits  # extract the chromosome
81          sub = chromosome[start:end]  # extract the sub-chromosome
82          chars = ''.join([str(s) for s in sub])  # convert to string
83          integer = int(chars, 2)  # convert to integer
84          real_value = bounds[i][0] + (integer / (2 ** bits)) * (bounds[i][1] - bounds[i][0])  # convert to real value
85          real_chromosome.append(real_value)  # add to the real chromosome
86      return real_chromosome
```

The decoding function converts the binary chromosome list to a string and then to an integer based on the bounds [-3,3]. Firstly, is initialize the output list call ‘real_chromosome’. Then calculates slice indices for each variable binary segment. Moving on is conversion of binary to integer, where line 81 is extracts binary substring, line 82 is converts to string representation and line 83 is parses binary string to integer. Then we map the integer value within the bounds to ‘real-value’. After calculating the real value from its binary representation, the real value gets appended to the ‘real_chromosome’ list. Once all variables have been processed, the complete list of decoded real values is returned.

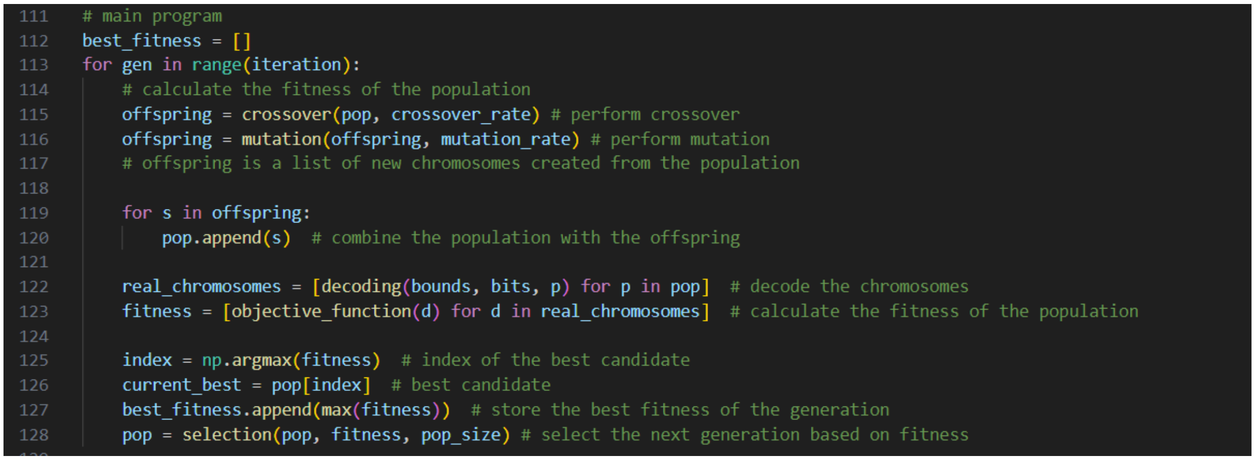
- Define the roulette wheel selection function



```
# roulette wheel selection
def selection(pop, fitness, pop_size):
    # Convert fitness to positive values for roulette wheel
    min_f = min(fitness)
    adjusted_fitness = [f - min_f + 1e-8 for f in fitness]  # Shift to positive

    #(best solution)
    next_generation = []
    elite = np.argmax(fitness)  # index of the best candidate
    next_generation.append(pop[elite])  # add the best candidate to the next generation

    # Roulette wheel probabilities
    p = [f / sum(adjusted_fitness) for f in adjusted_fitness]  # probabilities of selection

    # Select new population (elite + roulette selections)
    index = list(range(len(pop)))
    index_selected = np.random.choice(index, size=pop_size - 1, replace=False, p=p)
    s = 0
    for j in range(pop_size - 1):
        next_generation.append(pop[index_selected[s]])  # add the selected candidates to the next generation
        s += 1
    return next_generation
```

The ‘selection’ function is called with the combined population, their fitness, and the desired population size (100). In line 92, the function first adjusts the fitness to be positive by subtracting the minimum and adding a small epilation to use in the roulette wheel. Then it selects the ‘elite’ the best individual by using ‘.argmax’ (line96) to find the index of the highest fitness value in the fitness array and places it in the next generation. Then in line 100, create selection probabilities for each chromosome in the population. For line 103 to 108, it is used to choose individuals for the next generation. This is done by create a list of indices corresponding to each individual in the population. Then uses NumPy's random selection to choose indices where ‘pop_size-1’ is leaving a room for elite individual, ‘replace=False’ is to ensure no duplicates and ‘p=p’ is contains the selection probabilities based on the fitness value. In line 106 to 108, the code iterates through previously selected indices to build the next generation population, excluding the elite individual that was already added. Lastly, returns the complete next generation list that containing the elite individual best solution and all selected individuals from roulette wheel selection.

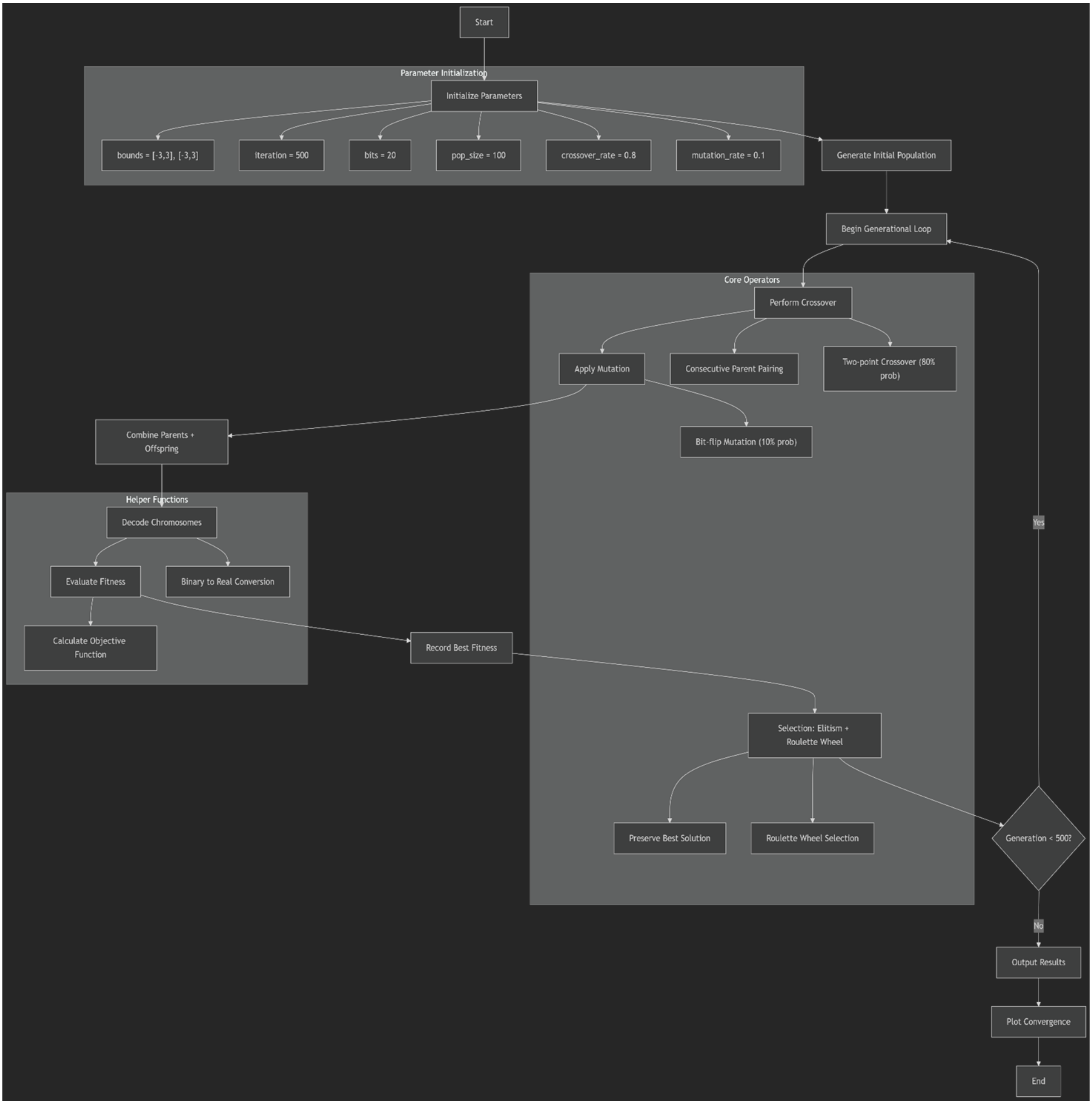
- Main Loop of Genetic Algorithm



```
111  # main program
112  best_fitness = []
113  for gen in range(iteration):
114      # calculate the fitness of the population
115      offspring = crossover(pop, crossover_rate) # perform crossover
116      offspring = mutation(offspring, mutation_rate) # perform mutation
117      # offspring is a list of new chromosomes created from the population
118
119      for s in offspring:
120          pop.append(s)  # combine the population with the offspring
121
122      real_chromosomes = [decoding(bounds, bits, p) for p in pop]  # decode the chromosomes
123      fitness = [objective_function(d) for d in real_chromosomes]  # calculate the fitness of the population
124
125      index = np.argmax(fitness)  # index of the best candidate
126      current_best = pop[index]  # best candidate
127      best_fitness.append(max(fitness))  # store the best fitness of the generation
128      pop = selection(pop, fitness, pop_size) # select the next generation based on fitness
```

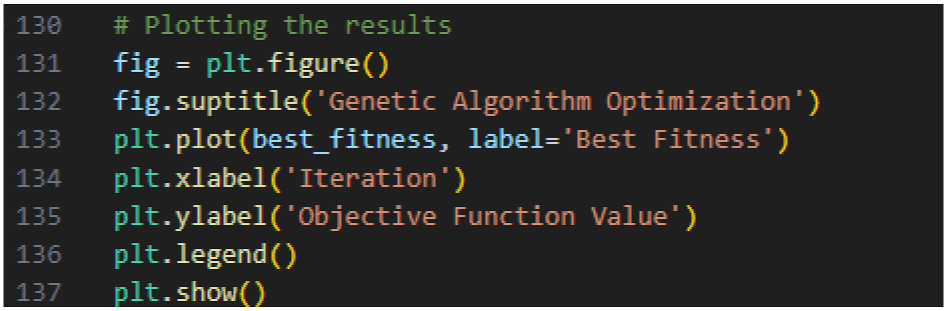
Before the loop, the ‘best_fitness’ is used to tracks the best fitness across generations. The genetic algorithm implementation begins by loop over generations (line113). For each generations, it performs crossover on the current population using a crossover rate of to produce offspring (line 115). These offspring then undergo mutation at a given mutation rate (line 116). The new offspring solutions are added to the existing population pool (lines 119-120) where results in a combined population of old individuals (parents) and new individuals (offspring/child). Next, each chromosome in the combined population which is in a binary representation is decoded to real value using the ‘decoding’ function. The ‘bounds’ and ‘bits’ parameters define the mapping from binary string to real values within the given bounds. (line 122). The fitness evaluation ‘objective_function’ is evaluated for each decoded real valued chromosome. This produces a list of fitness values for the entire combined population (line 123). The algorithm then identifies and records the current best solution and its fitness value (lines 125-127) where ‘index’ is found as the index of the chromosome with the highest fitness (using `np.argmax`) , ‘current_best’ is set to the best chromosome (in binary form) of the current combined population and the best fitness value of the current generation is appended to ‘best_fitness’. Finally, the selection function is called to reduce the combined population back to the original size ‘pop_size’. The selection is based on the fitness of each individual. The loop continues until all iterations are completed while tracking progressive fitness improvements.

- Genetic Algorithm Optimization Flowchart

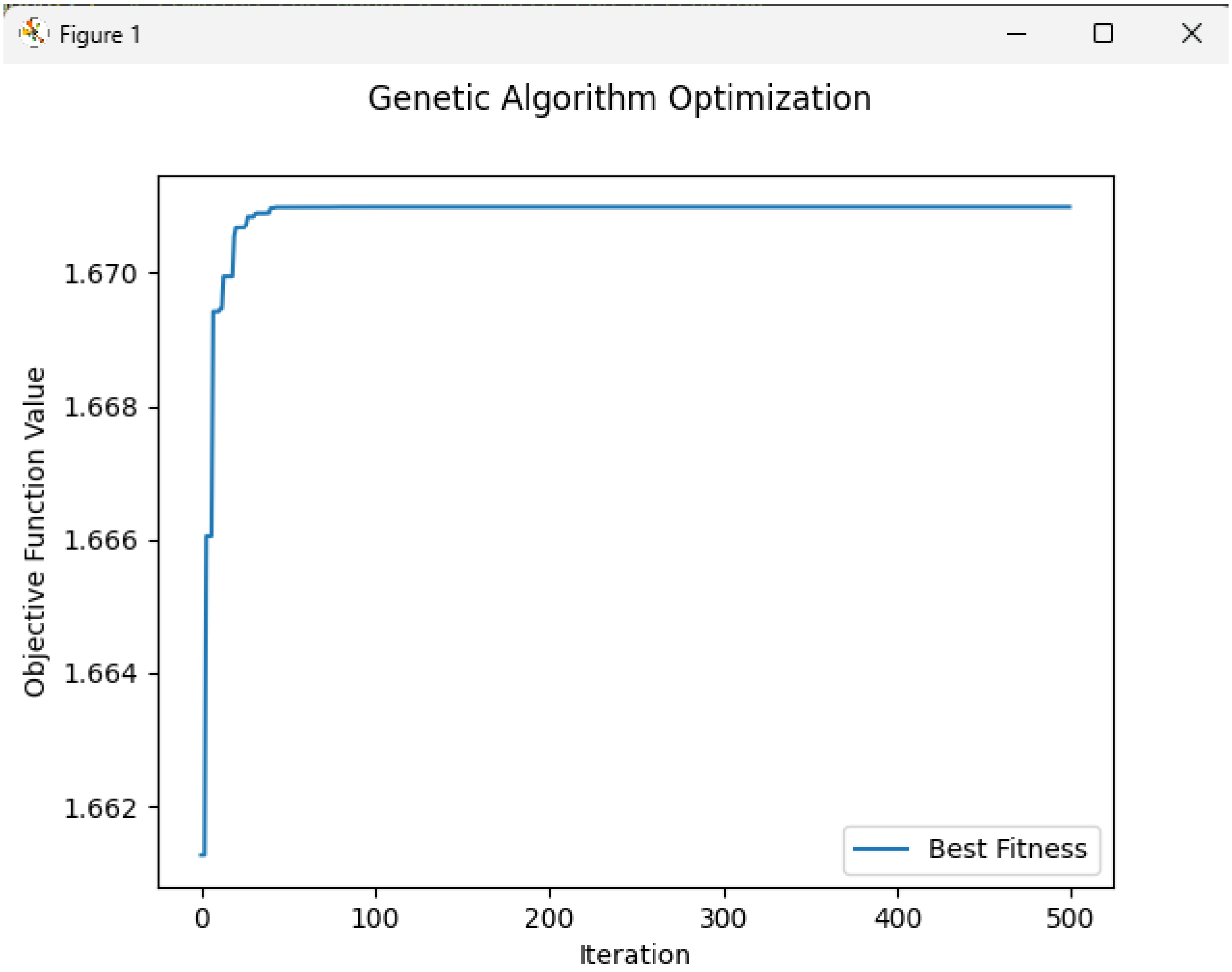


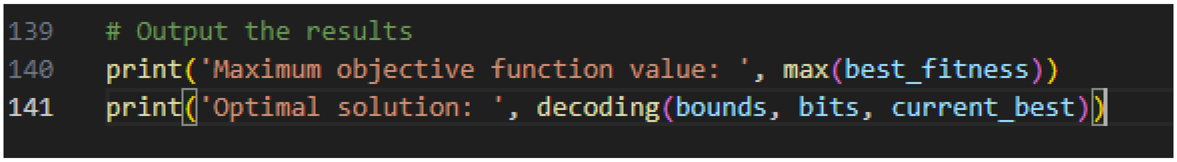
The genetic algorithm begins by initializing parameters including the search bounds of [-3,3] for each variable, a maximum of 500 generations, 20 bits per variable for chromosome representation, a population size of 100, a crossover rate of 0.8, and a mutation rate of 0.1. An initial population of binary strings is randomly generated. The main generational loop involves applying crossover which using two-point crossover with an 80% chance per pair, to parent pairs selected for reproduction, followed by bit-flip mutation with a 10% chance per bit. The binary chromosomes are then decoded into real values within the defined bounds. The fitness of each individual is evaluated by computing the objective function. The algorithm records the best fitness encountered. Selection for the next generation combines elitism to preserving the best individuals and roulette wheel selection which is probabilistically based on fitness. This process repeats for 500 generations, after that the best solution found is being output along with convergence information.

3.0 Result

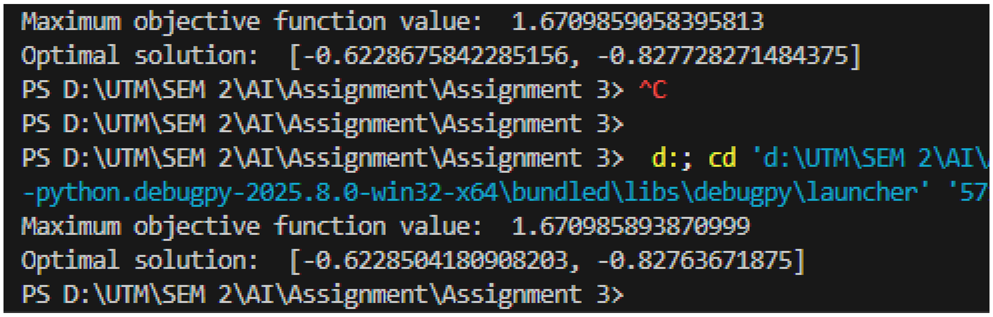


```python
# Plotting the results
fig = plt.figure()
fig.suptitle('Genetic Algorithm Optimization')
plt.plot(best_fitness, label='Best Fitness')
plt.xlabel('Iteration')
plt.ylabel('Objective Function Value')
plt.legend()
plt.show()
```





```python
# Output the results
print('Maximum objective function value: ', max(best_fitness))
print('Optimal solution: ', decoding(bounds, bits, current_best))
```



```
Maximum objective function value:  1.6709859058395813
Optimal solution:  [-0.6228675842285156, -0.827728271484375]
PS D:\UTM\SEM 2\AI\Assignment\Assignment 3> ^C
PS D:\UTM\SEM 2\AI\Assignment\Assignment 3>
PS D:\UTM\SEM 2\AI\Assignment\Assignment 3>  d:; cd 'd:\UTM\SEM 2\AI\
-python.debugpy-2025.8.0-win32-x64\bundled\libs\debugpy\launcher' '57
Maximum objective function value:  1.670985893870999
Optimal solution:  [-0.6228504180908203, -0.82763671875]
PS D:\UTM\SEM 2\AI\Assignment\Assignment 3>
```

4.0 Conclusion

The genetic algorithm implemented has successfully found a consistent solution for the maximization of the given two-dimensional function where, $f(x,y) = (1-x)^2 e^{-x^2-(y+1)^2} - (x - x^3 - y^3)e^{-x^2-y^2}$. The function bounds at [-3,3] show an optimal solution at (-0.623,-0.828) represents a significant maximum of this function, achieving a peak value of approximately 1.671. The algorithm demonstrates robustness through multiple runs and converges reliably. The convergence plot would further confirm the typical behavior of a Genetic Algorithm and rapid improvement in early generations followed by stabilization.