Lab - 7

Optimization via Gene Expression Algorithms

```
import numpy as np

# Define the mathematical function to optimize (e.g., 
Sphere function)
def objective_function(x):
    return np.sum(x**2)  # Minimize this function

# Initialize parameters
population_size = 50  # Number of genetic sequences in 
the population
num_genes = 10  # Number of genes in each sequence
mutation_rate = 0.1  # Probability of mutation
crossover_rate = 0.8  # Probability of crossover
generations = 100  # Number of generations
lower_bound, upper_bound = -10, 10  # Solution space 
bounds

# Initialize the population with random genetic sequences
def initialize_population():
    return np.random.uniform(lower_bound, upper_bound, 
(population_size, num_genes))

# Evaluate the fitness of each genetic sequence
def evaluate_fitness(population):
    fitness = np.array([objective_function(individual) 
for individual in population])
    return fitness

# Selection: Select genetic sequences based on fitness 
(tournament selection)
def select_parents(population, fitness):
    selected_parents = []
    for _ in range(population_size):
        candidates = np.random.choice(len(population), 3, 
replace=False)
        best_candidate = 
candidates[np.argmin(fitness[candidates])]
```

```
    selected_parents.append(population[best_candidate])
    return np.array(selected_parents)

# Crossover: Perform crossover between selected sequences
to produce offspring
def crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        parent1 = parents[i]
        parent2 = parents[(i + 1) % len(parents)]
        if np.random.rand() < crossover_rate:
            crossover_point = np.random.randint(1,
num_genes - 1)
            child1 =
np.concatenate((parent1[:crossover_point],
parent2[crossover_point:]))
            child2 =
np.concatenate((parent2[:crossover_point],
parent1[crossover_point:]))
            offspring.extend([child1, child2])
        else:
            offspring.extend([parent1, parent2])
    return np.array(offspring)

# Mutation: Apply mutation to the offspring
def mutate(offspring):
    for individual in offspring:
        if np.random.rand() < mutation_rate:
            gene_to_mutate = np.random.randint(num_genes)
            individual[gene_to_mutate] =
np.random.uniform(lower_bound, upper_bound)
    return offspring

# Gene Expression: Decode the genetic sequence into a
functional solution (identity in this case)
def gene_expression(sequence):
    return sequence

# Main function to execute the Gene Expression Algorithm
def gene_expression_algorithm():
    population = initialize_population()
```

```
    best_solution = None
    best_fitness = float('inf')

    for generation in range(generations):
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution =
population[np.argmin(fitness)]

        # Selection, Crossover, Mutation, and Gene
Expression
        parents = select_parents(population, fitness)
        offspring = crossover(parents)
        mutated_offspring = mutate(offspring)
        population = np.array([gene_expression(ind) for
ind in mutated_offspring])

    return best_solution, best_fitness

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Best Solution: [ 6.02069591e-04  6.33325442e-02  5.84702655e-01 -2.23247895e-01
 -3.99173913e-01  2.32991980e-01 -6.35082961e-01  1.57878812e-01
  4.75012210e-02  2.92306981e-01]
Best Fitness: 1.1253090907692773
```