Optimization via Gene Expression Algorithms:

```
import numpy as np


def objective_function(x):
    """
    Define the mathematical function to optimize.
    Example: Sphere function f(x) = sum(x^2).
    """
    return np.sum(x**2)


def initialize_population(pop_size, num_genes, bounds):
    """
    Generate an initial population of random genetic sequences.
    """
    return np.random.uniform(bounds[0], bounds[1], size=(pop_size, num_genes))


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


def selection(population, fitness, num_parents):
    """
    Select genetic sequences based on their fitness for reproduction.
    """
    parents = np.zeros((num_parents, population.shape[1]))
    for i in range(num_parents):
        best_idx = np.argmin(fitness)
        parents[i, :] = population[best_idx, :]
        fitness[best_idx] = float('inf')  # Exclude the selected individual
```

```
    return parents

def crossover(parents, offspring_size):
    """
    Perform crossover between selected sequences to produce offspring.
    """
    offspring = np.zeros(offspring_size)
    num_parents = parents.shape[0]
    for i in range(offspring_size[0]):
        parent1_idx = i % num_parents
        parent2_idx = (i + 1) % num_parents
        crossover_point = np.random.randint(1, offspring_size[1])
        offspring[i, :crossover_point] = parents[parent1_idx, :crossover_point]
        offspring[i, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring

def mutation(offspring, mutation_rate, bounds):
    """
    Apply mutation to the offspring to introduce variability.
    """
    for i in range(offspring.shape[0]):
        if np.random.rand() < mutation_rate:
            gene_idx = np.random.randint(0, offspring.shape[1])
            offspring[i, gene_idx] = np.random.uniform(bounds[0], bounds[1])
    return offspring

def gene_expression(genetic_sequences):
    """
    Translate genetic sequences into functional solutions (if needed).
    Here, the genetic sequences directly represent solutions.
    """
```

```
    return genetic_sequences


def gene_expression_algorithm(
    pop_size, num_genes, bounds, num_generations, mutation_rate, crossover_rate
):
    """
    Main implementation of the gene expression algorithm.
    """
    # Initialize population
    population = initialize_population(pop_size, num_genes, bounds)

    # Track the best solution
    best_solution = None
    best_fitness = float('inf')

    for generation in range(num_generations):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

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

        # Selection
        num_parents = int(crossover_rate * pop_size)
        parents = selection(population, fitness, num_parents)

        # Crossover
        offspring_size = (pop_size - parents.shape[0], num_genes)
```

```
        offspring = crossover(parents, offspring_size)

        # Mutation
        offspring = mutation(offspring, mutation_rate, bounds)

        # Gene Expression (if needed)
        offspring = gene_expression(offspring)

        # Create the new population
        population[:parents.shape[0], :] = parents
        population[parents.shape[0]:, :] = offspring



    return best_solution, best_fitness

# Parameters
pop_size = 100  # Population size
num_genes = 5  # Number of genes (dimensionality of the solution space)
bounds = [-10, 10]  # Bounds for the solution space
num_generations = 50  # Number of generations
mutation_rate = 0.1  # Probability of mutation
crossover_rate = 0.5  # Fraction of population involved in crossover

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm(
    pop_size, num_genes, bounds, num_generations, mutation_rate, crossover_rate
)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

Output:

```
Best Solution: [-0.06016522  0.28760975  0.48263338  9.17836477  0.08649596]
Best Fitness: 0.3300982665540435
```