GENE EXPRESSION ALGORITHM

Code:

```
import random
import math

# --- PARAMETERS ---
POPULATION_SIZE = 50
GENE_LENGTH = 30
GENERATIONS = 100
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.7

# Terminals (constants, variable 'x') and Functions
TERMINALS = ['x', '1', '2', '3', '4', '5']
FUNCTIONS = ['+', '-', '*', '/', 'sin', 'cos']

# Target Cost Function (to minimize)
def cost_function(x):
    """ Example cost function to minimize. Replace with your target function. """
    return x**2 - 10 * math.sin(2 * x)

# --- GENE EXPRESSION CLASS ---
class GeneExpression:
    def __init__(self):
        self.gene = self._random_gene()
        self.cached_fitness = None  # To store fitness value

    def _random_gene(self):
        """ Initialize a random gene sequence. """
        return [random.choice(TERMINALS + FUNCTIONS) for _ in range(GENE_LENGTH)]
```

```
def decode_gene(self, x):
    """ Decode the gene into a mathematical expression and evaluate it. """
    stack = []
    for token in self.gene:
        if token in TERMINALS:
            stack.append(float(x) if token == 'x' else float(token))
        elif token in FUNCTIONS:
            if len(stack) >= 1 and token in ['sin', 'cos']:
                arg = stack.pop()
                stack.append(math.sin(arg) if token == 'sin' else math.cos(arg))
            elif len(stack) >= 2:
                b, a = stack.pop(), stack.pop()
                if token == '+': stack.append(a + b)
                elif token == '-': stack.append(a - b)
                elif token == '*': stack.append(a * b)
                elif token == '/' and b != 0: stack.append(a / b)
            else:
                return float('inf')  # Malformed gene
    return stack[0] if len(stack) == 1 else float('inf')

def fitness(self, x):
    """ Evaluate fitness: minimize cost_function(output). """
    if self.cached_fitness is None:
        try:
            result = self.decode_gene(x)
            self.cached_fitness = abs(cost_function(result))
        except:
            self.cached_fitness = float('inf')
    return self.cached_fitness
```

```python
# --- GENETIC OPERATIONS ---
def selection(population, fitnesses):
    """ Tournament selection: Select the best from random candidates. """
    tournament_size = 3
    candidates = random.sample(list(zip(population, fitnesses)), tournament_size)
    return min(candidates, key=lambda c: c[1])[0]

def crossover(parent1, parent2):
    """ Perform single-point crossover between two parents. """
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENE_LENGTH - 1)
        child1 = GeneExpression()
        child2 = GeneExpression()
        child1.gene = parent1.gene[:point] + parent2.gene[point:]
        child2.gene = parent2.gene[:point] + parent1.gene[point:]
        return child1, child2
    return parent1, parent2

def mutate(individual):
    """ Apply mutation by altering random parts of the gene. """
    for i in range(GENE_LENGTH):
        if random.random() < MUTATION_RATE:
            individual.gene[i] = random.choice(TERMINALS + FUNCTIONS)

# --- MAIN EVOLUTION FUNCTION ---
def geneExpression():
    # Initialization
    population = [GeneExpression() for _ in range(POPULATION_SIZE)]
    x_value = random.uniform(-10, 10)  # Random input to test optimization
```

```
    # Evolutionary loop
    for generation in range(GENERATIONS):
        fitnesses = [ind.fitness(x_value) for ind in population]
        best_idx = fitnesses.index(min(fitnesses))
        print(f"Generation {generation}: Best Fitness = {fitnesses[best_idx]:.5f}")

        # Elitism: Preserve the best individual
        new_population = [population[best_idx]]

        # Create next generation
        while len(new_population) < POPULATION_SIZE:
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            mutate(child1)
            mutate(child2)
            new_population.extend([child1, child2])

        population = new_population

    # Final Solution
    final_fitnesses = [ind.fitness(x_value) for ind in population]
    best_idx = final_fitnesses.index(min(final_fitnesses))
    print("\nOptimized Solution:")
    print(f"Best Gene: {population[best_idx].gene}")
    print(f"Best Fitness: {final_fitnesses[best_idx]:.5f}")

if __name__ == "__main__":
    geneExpression()
```

Output:

```
Generation 0: Best Fitness = inf
Generation 1: Best Fitness = inf
Generation 2: Best Fitness = inf
Generation 3: Best Fitness = inf
Generation 4: Best Fitness = 37.56666
Generation 5: Best Fitness = 37.56666
Generation 6: Best Fitness = 37.56666
Generation 7: Best Fitness = 37.56666
Generation 8: Best Fitness = 37.56666
Generation 9: Best Fitness = 37.56666
Generation 10: Best Fitness = 37.56666
Generation 11: Best Fitness = 37.56666
Generation 12: Best Fitness = 37.56666
Generation 13: Best Fitness = 26.06944
Generation 14: Best Fitness = 26.06944
Generation 15: Best Fitness = 26.06944
Generation 16: Best Fitness = 26.06944
Generation 17: Best Fitness = 26.06944
Generation 18: Best Fitness = 26.06944
```