```
import numpy as np
import matplotlib.pyplot as plt

# Datele de intrare
population_size = 20              # dimensiunea populatiei
#lowerBound = None                # limita de jos a domeniului
#upperBound = None                # limita de sus
domain = (-1, 2)                  # domeniul functiei
coefficients = (-1, 1, 2)         # parametrii functiei
precision = 6                     # precizia de discretizare a intervalului
crossover_probability = 0.25      # probabilitatea de recombinare
mutatioin_probability = 0.01      # probabilitatea de mutatie
generations = 50                  # numarul de generatii
leap = 25                         # numarul de generatii peste care se sare la plotare

f = open("Evolutie.txt", "w")

# Calculam numarul de biti necesari pentru precizia si domeniul date
domain_range = domain[1] - domain[0]
total_values = domain_range * (10 ** precision)
bits_needed = int(np.ceil(np.log2(total_values)))

#print(bits_needed)

# Functia de fitness
def fitness(x):
    return coefficients[0] * x ** 2 + coefficients[1] * x + coefficients[2]


def binary_to_double(bit_string):
    max_val = 2 ** len(bit_string) - 1
    int_val = int(bit_string, 2)
    real_val = domain[0] + (domain_range * int_val) / max_val
    return real_val


def binary_search(array, target):
    left = 0
    right = len(array) - 1

    while left <= right:
        mid = (left + right) // 2
        if array[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return left


def selection_binary_search(population, fitness_vals):
    fitness_sum = sum(fitness_vals)
    probabilities = [f / fitness_sum for f in fitness_vals]
    cumulative_prob = np.cumsum(probabilities)
```

```
    selected_indices = []
    indices_prob = []
    for _ in range(population_size - 1):
        r = np.random.rand()
        index = binary_search(cumulative_prob, r)
        selected_indices.append(index)
        indices_prob.append(r)

    selected = [population[i] for i in selected_indices]
    selected_indices.append(np.argmax(fitness_vals))
    best_individual = population[np.argmax(fitness_vals)]
    selected.append(best_individual)

    return (selected, selected_indices, indices_prob)


# Functii pentru afisare

def print_best_mean_fitness(real_vals, fitness_vals):
    best_fitness = fitness_vals[np.argmax(fitness_vals)]
    best_val = real_vals[np.argmax(fitness_vals)]
    mean_fitness = sum(fitness_vals) / population_size
    f.write(f"x = {(best_val + 1):.6f}  Best fitness: {best_fitness}  Mean fitness: {mean_fitness}\n")


def print_initial_population(population, real_vals, fitness_vals):
    f.write("Populatia Initiala: \n")
    for i, (bit_string, real_val, fitness) in enumerate(zip(population, real_vals,fitness_vals)):
        f.write(f"  {i + 1}: {bit_string} x = {real_val:.6f} f = {fitness}")
        f.write("\n")


def print_selection_prob(fitness_vals):
    fitness_sum = sum(fitness_vals)
    probabilities = [f / fitness_sum for f in fitness_vals]
    cumulative_prob = np.cumsum(probabilities)

    f.write("Probabilitati Selectie: \n")

    for cromozome in range(population_size):
        f.write(f"Cromozom {cromozome + 1}:   probabilitate: {probabilities[cromozome]}\n")

    f.write("Intervale Probabilitati Selectie: \n")
    for item in cumulative_prob:
        f.write(str(item))
        f.write("\n")


def print_selections(selected, selected_prob):
    for i in range(len(selected) - 1):
        f.write(f"u = {selected_prob[i]}    selectam cromozomul {selected[i] + 1}\n")

    f.write(f"u = Elitism       selectam cromozomul {selected[i + 1] + 1}\n")
```

```
def print_selected_cromozomes(selected, selected_ind, real_vals, fitness):
    f.write("Dupa selectie: \n")
    for i in range(len(selected)):
        f.write(f"{i + 1}: {selected[i]}  x = {real_vals[selected_ind[i]]:.6f}  f = {fitness[selected_ind[i]]}\n")


def print_pair_crossover_prob(index1, chromosome1, index2, chromosome2, probability):
    if(index1 == 0):
        f.write(f"Probabilitatea de incrucisare este: {crossover_probability}\n")

    f.write(f"{index1 + 1}: {chromosome1}   ")
    f.write(f"{index2 + 1}: {chromosome2}\n")
    if(probability < crossover_probability):
        f.write(f"{probability} < {crossover_probability} -> participa\n")


def print_crossover_result(index1, chromosome1, index2, chromosome2, crossover_point, offspring1,
offspring2):
    f.write(f"Recombinare dintre cromozomul {index1 + 1} cu cromozomul {index2 + 1}:\n")
    f.write(f"{chromosome1} {chromosome2} punct {crossover_point}\n")
    f.write(f"Rezultat  {offspring1}    {offspring2}\n")


def print_crossover_offsprings(offsprings):
    for i in range(len(offsprings)):
        real_val = binary_to_double(offsprings[i])
        f.write(f"{i + 1}: {offsprings[i]}  ")
        f.write(f"x = {real_val:.6f}   ")
        f.write(f"f = {fitness(real_val)}\n")


def print_mutated_indices(mutated_indices):
    f.write(f"Probabilitatea de mutatie pentru fiecare gena {mutatioin_probability}\n")
    f.write("Au fost modificati cromozomii: \n")
    for index in mutated_indices:
        f.write(f"{index + 1}\n")


def print_mutation_offsprings(offsprings):
    f.write("Dupa mutatie: \n")
    for i in range(len(offsprings)):
        real_val = binary_to_double(offsprings[i])
        f.write(f"{i + 1}: {offsprings[i]}  x = {real_val:.6f}  f = {fitness(real_val)}\n")


def print_generation_visual(real_vals, fitness_vals, generation):
    real_vals_distrib = np.linspace(domain[0], domain[1], 150)
    fitness_vals_distrib = fitness(real_vals_distrib)

    plt.scatter(real_vals, fitness_vals, color = "blue")
    plt.plot(real_vals_distrib, fitness_vals_distrib)
    plt.xlabel("values")
    plt.ylabel("fitness")
    plt.title(f"Generation: {generation + 1}")
    plt.show()
```

```
def Crossover(population_size, chromosomes, generation):
    offspring = []
    for i in range(0, population_size, 2):
        parent1 = chromosomes[i]
        parent2 = chromosomes[min(i + 1, len(chromosomes) - 1)]

        u = np.random.rand()
        if generation == 0:
            print_pair_crossover_prob(i, parent1, min(i + 1, len(chromosomes) - 1), parent2, u)

        if u < crossover_probability:
            crossover_point = np.random.randint(1, bits_needed - 1)
            offspring1 = parent1[ : crossover_point] + parent2[crossover_point : ]
            offspring2 = parent2[ : crossover_point] + parent1[crossover_point : ]
            if generation == 0:
                print_crossover_result(i, parent1, min(i + 1, len(chromosomes) - 1), parent2, crossover_point,
                                       offspring1, offspring2)
        else:
            offspring1, offspring2 = parent1, parent2

        offspring.extend([offspring1, offspring2])

    return offspring


def print_init_crossover_prob(chromosomes, participant_indices, participant_prob):
    f.write(f"Probabilitatea de incrucisare este {crossover_probability}\n")
    counter = 0
    for i in range(len(chromosomes)):
        #f.write(f"{i + 1}: {chromosomes[i]}  u = {participant_prob[i]}")
        if i in participant_indices:
            f.write(f"{i + 1}: {chromosomes[i]}  u = {participant_prob[i]} < {crossover_probability} participa\n")
        else:
            f.write(f"{i + 1}: {chromosomes[i]}  u = {participant_prob[i]}\n")




def Crossover_2(population_size, chromosomes, generation):
    participant_indices = []
    participant_probabilities = []
    for i in range(population_size):
        u = None
        if(i == population_size - 1):
            u = 2
        else:
            u = np.random.rand()
        participant_probabilities.append(u)
        if u < crossover_probability:
            participant_indices.append(i)

    if generation == 0:
        print_init_crossover_prob(chromosomes, participant_indices, participant_probabilities)
```

```
    for i in range(population_size):
        if i in participant_indices:
            participant_indices.remove(i)
            for j in range(i, population_size):
                if j in participant_indices:
                    parent1 = chromosomes[i]
                    parent2 = chromosomes[j]
                    crossover_point = np.random.randint(1, bits_needed - 1)
                    offspring1 = parent1[ : crossover_point] + parent2[crossover_point : ]
                    offspring2 = parent2[ : crossover_point] + parent1[crossover_point : ]
                    chromosomes[i] = offspring1
                    chromosomes[j] = offspring2
                    participant_indices.remove(j)
                    if generation == 0:
                        print_crossover_result(i, parent1, j, parent2, crossover_point,
                                offspring1, offspring2)
                    break

    return chromosomes


def Mutate(offsprings):
    mutated_offspring = []
    mutated_indices = []
    for i in range(len(offsprings)):
        mutated_individual = None
        if(i == len(offsprings) - 1):
            mutated_individual = offsprings[i]
        else:
            mutated_individual = ''.join(
                bit if np.random.rand() > mutatioin_probability else '1' if bit == '0' else '0'
                for bit in offsprings[i]
            )
        mutated_offspring.append(mutated_individual)
        if mutated_individual != offsprings[i]:
            mutated_indices.append(i)

    return (mutated_offspring, mutated_indices)


# Generare populatie initiala
population = [''.join(np.random.choice(['0', '1']) for _ in range(bits_needed)) for _ in
range(population_size)]

real_vals = [binary_to_double(individual) for individual in population]
fitness_vals = [fitness(x) for x in real_vals]
print_initial_population(population, real_vals, fitness_vals)
print_selection_prob(fitness_vals)

print_generation_visual(real_vals, fitness_vals, 0)

for generation in range(generations):

    # Evaluarea fitness-ului
```

```
    real_vals = [binary_to_double(individual) for individual in population]
    fitness_vals = [fitness(x) for x in real_vals]

    if (generation + 1) % leap == 0:
        print_generation_visual(real_vals, fitness_vals, generation)

    if generation != 0:
        print_best_mean_fitness(real_vals, fitness_vals)

    # Selectia
    selected, selected_ind, ind_prob = selection_binary_search(population, fitness_vals)
    if generation == 0:
        print_selections(selected_ind, ind_prob)
        print_selected_cromozomes(selected, selected_ind, real_vals, fitness_vals)

    # Crossover
    offsprings = Crossover_2(population_size, selected, generation)
    if generation == 0:
        print_crossover_offsprings(offsprings)

    # Mutatii
    mutated_offsprings, mutated_indices = Mutate(offsprings)
    if generation == 0:
        print_mutated_indices(mutated_indices)
        print_mutation_offsprings(mutated_offsprings)

    # Updatam populatia pentru urmatoarea generatie
    population = mutated_offsprings

final_real_vals = [binary_to_double(individual) for individual in population]
final_fitness_vals = [fitness(x) for x in final_real_vals]
best_sol_index = np.argmax(final_fitness_vals)
best_sol_real_val = final_real_vals[best_sol_index]
best_sol_fitness = final_fitness_vals[best_sol_index]

print(best_sol_real_val, best_sol_fitness)
```