

**Министерство образования Российской Федерации**  
**МОСКВОСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**  
**УНИВЕРСИТЕТ им. Н.Э.БАУМАНА**

Факультет: Информатика и системы управления (ИУ)  
Кафедра: Информационная безопасность (ИУ8)

**МЕТОДЫ ОПТИМИЗАЦИИ**

**Домашнее задание №2 на тему:**  
**«Исследование генетических алгоритмов в задачах поиска**  
**экстремумов»**

Вариант – 19 (3)

**Преподаватель:**  
Коннова Н.С.

**Студент:**  
Михалева С.И

**Группа:**  
ИУ8-34

Москва, 2023

## Цель работы:

изучить основные принципы действия генетических алгоритмов на примере решения задач оптимизации функций двух переменных.

## Постановка задачи:

Найти максимум функции в области с помощью простого (классического) генетического алгоритма.

За исходную популяцию принять 4 случайных точки. Хромосома каждой особи состоит из двух генов: значений координат  $x$ ,  $y$ . В качестве потомков следует выбирать результат скрещивания лучшего решения со вторым и третьим в порядке убывания значений функции приспособленности с последующей случайной мутацией обоих генов.

В качестве критерия остановки эволюционного процесса задаться номером конечной популяции (). Визуализировать результаты расчетов.

## Ход решения:

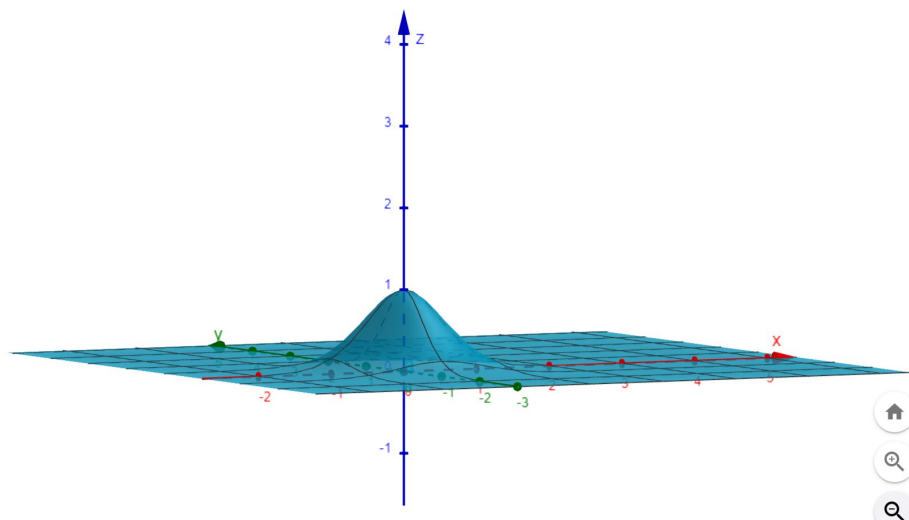
Исходная функция имеет вид:

$$\frac{e^{(-x^2-y^2)}}{(1+x^2+y^2)}$$

Область допустимых значений

$$(-2,2) \times (-2,2)$$

1) Построим график исследуемой поверхности



Для решения задачи была написана программа на языке C# (см.

Приложение А):

### 1. genetic\_algorithm.cs

Этот код реализует генетический алгоритм для оптимизации функции вещественных переменных. Вот пошаговое объяснение:

#### Класс Individual:

Определяет структуру для представления индивида с вещественными переменными x, y и их фитнес-функцией.

#### CreateInitialPopulation:

Создает начальную популяцию индивидов с случайными значениями переменных x и y в заданных пределах (MIN X, MAX X, MIN Y, MAX Y).

#### FitnessFunction:

Определяет функцию приспособленности (fitness) для индивида на основе его переменных x и y.

#### EvaluatePopulation:

Оценивает приспособленность каждого индивида в популяции, используя функцию приспособленности.

#### SelectParents:

Использует турнирный отбор для выбора родителей для создания следующего поколения. Родители выбираются на основе их взвешенной приспособленности, учитывая расстояние от определенной точки.

#### Mutate:

Производит мутацию индивида, изменяя его переменные x и y с определенной вероятностью (MUTATION\_RATE).

#### Crossover:

Создает потомка (child) путем смешивания переменных родителей (parent1 и parent2) с использованием случайного кроссовера.

CheckConvergence:

Проверяет сходимость популяции, основываясь на средней приспособленности индивидов. Если большинство индивидов близки к средней приспособленности, считается, что популяция сошлась.

PrintToTxt:

Записывает информацию о популяции в текстовый файл для некоторых выбранных итераций.

Main:

Запускает основной цикл генетического алгоритма.

Оценивает приспособленность популяции, выбирает родителей, мутирует и кроссоверит индивидов, проверяет сходимость и записывает результаты.

При достижении времени выполнения или сходимости, выводит соответствующее сообщение и завершает выполнение.

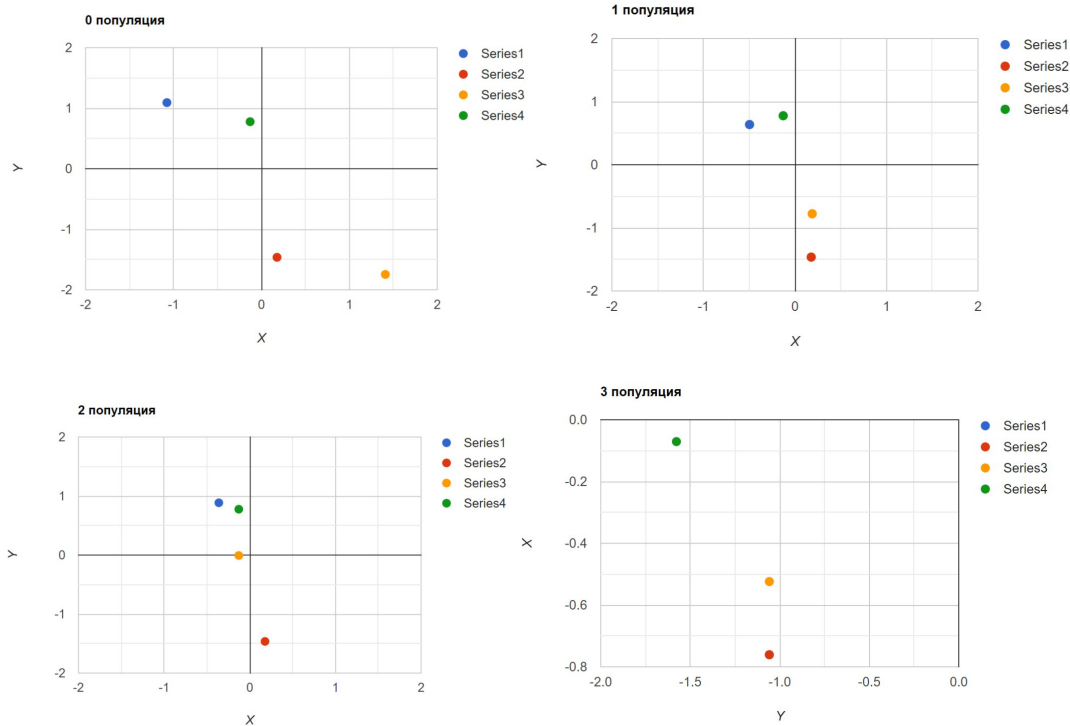
По условию задачи требуется написать программу, реализующую создание 4-х особей, имеющих по 2 хромосомы каждая (координаты  $x$  и  $y$ ). При этом создание особей происходит случайным способом (невоспроизводимый ГПСЧ). После создания особей программа должна выполнить расчет среднего и максимального значения FIT-функции для популяции. Далее программа производит селекцию (отбор) и последующий кроссовер особей в соответствии с условием задания. При этом следует учесть мутацию (положим вероятность мутации равной 10%). Данные действия повторяются для каждого поколения (итерации алгоритма) для достижения критерия останова алгоритма, в данном случае – номера поколения  $N$ , схождения фитнес функции или превышения времени работы алгоритма (15с). Сгенерированные числа, а также среднее и максимальное значения FIT-функции популяции для поколений:

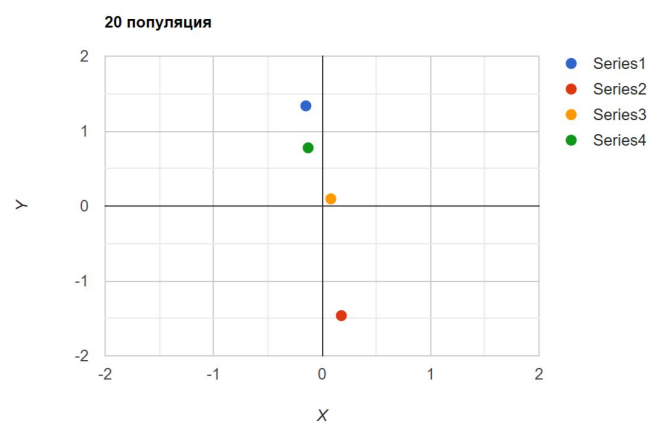
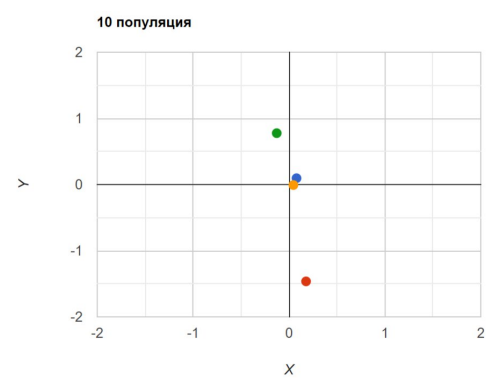
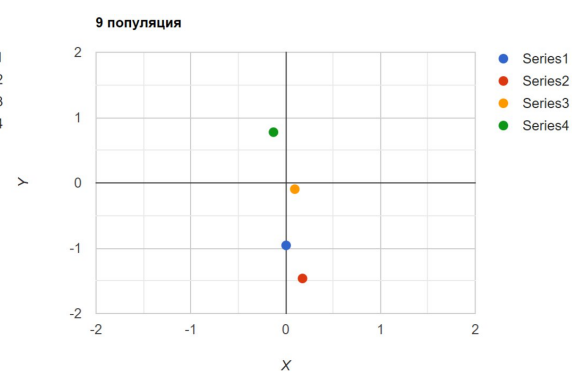
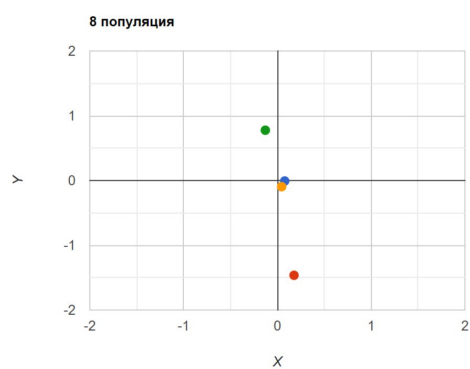
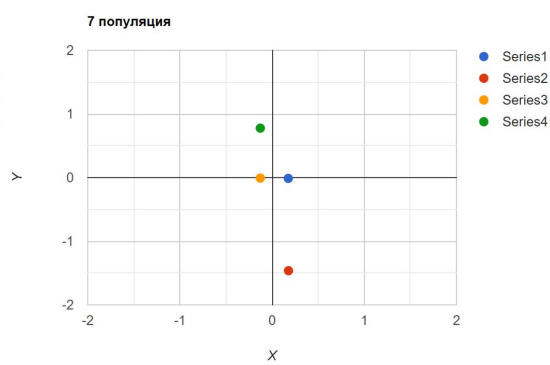
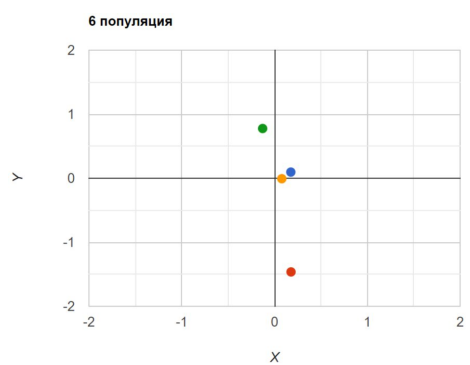
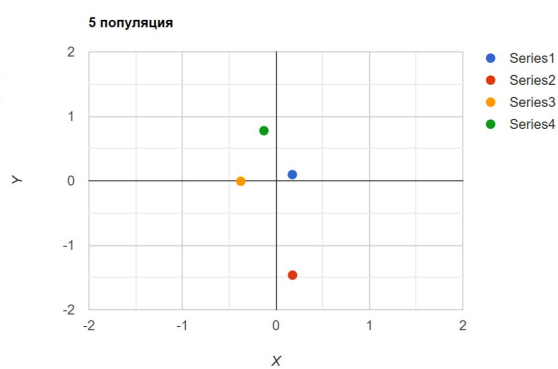
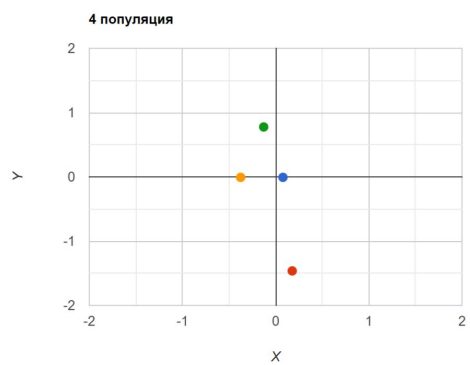
№ поколения	X	Y	FIT	Максимум	Среднее значение
0 (исходное)	-1.0747 0.1778 1.4099 -0.1282	1.0918 -1.4635 -1.7465 0.7766	0.0285 0.0358 0.0010 0.3322	0.9876	0.1098

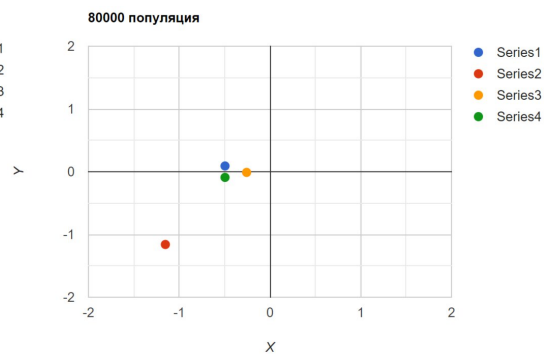
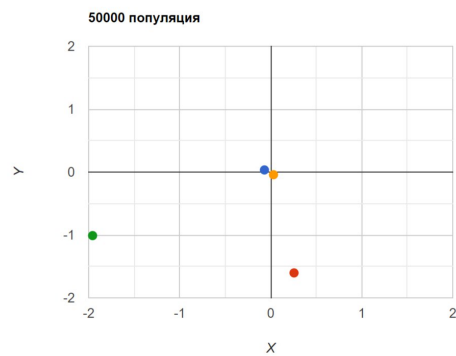
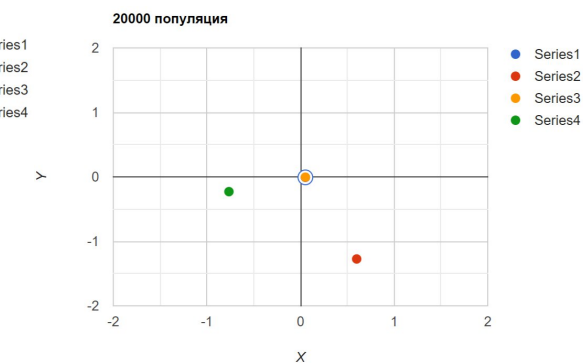
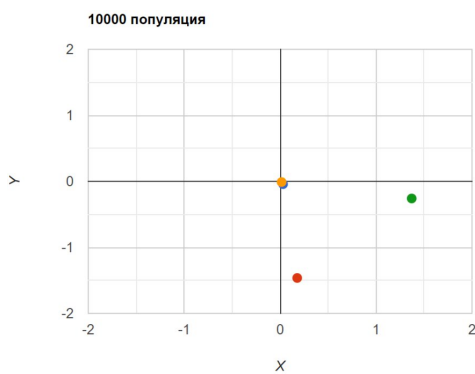
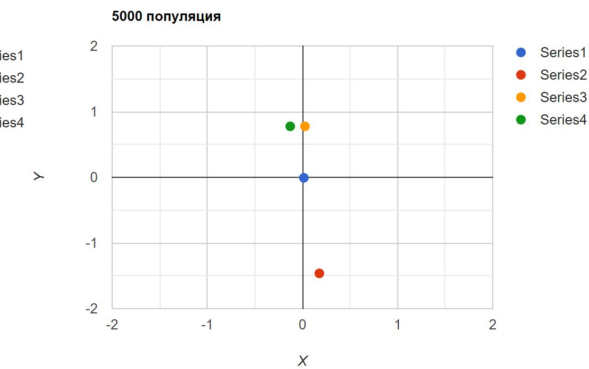
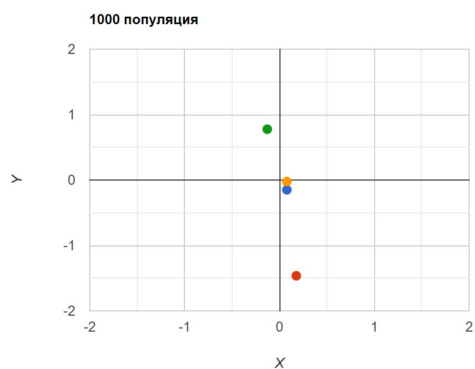
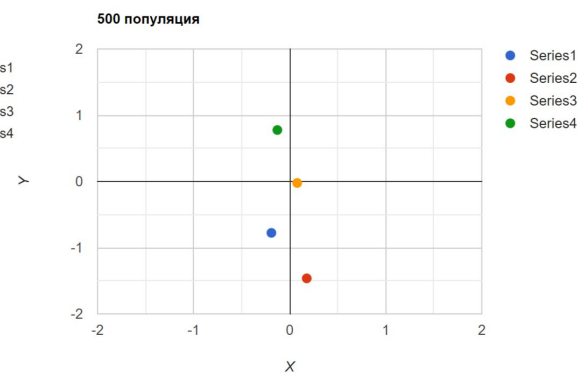
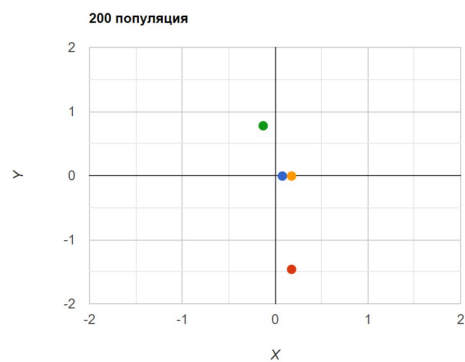
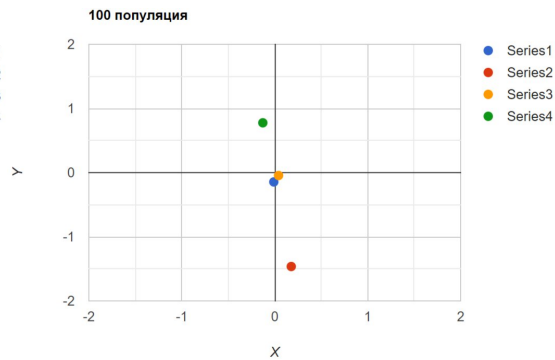
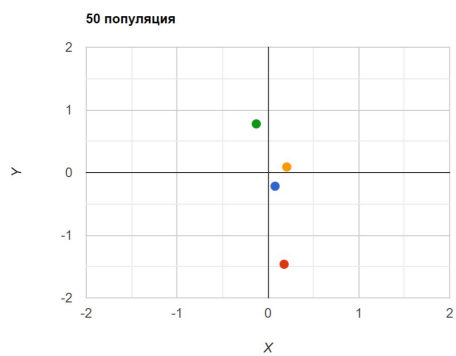
1	-0.4916 0.1778 0.1885 -0.1282	0.6367 -1.4635 0.7766 0.7766	0.3178 0.0358 0.3222 0.3322	0.9876	0.1593
2	-0.3611 0.1778 -0.3611 -0.1282	0.8856 -1.4635 0.8154 0.7766	0.2092 0.0358 0.2514 0.3322	0.9876	0.2229
3	0.0784 0.1778 -0.1282 -0.1282	-0.0076 -1.4635 -0.00767 0.7766	0.9876 0.0358 0.9676 0.3322	0.9876	0.3021
4	0.0784 0.1778 0.0784 -0.1282	-0.0076 -1.4635 -0.0751 0.7766	0.9876 0.0358 0.9767 0.3322	0.9876	0.3799
5	0.1756 0.1778 -0.3754 -0.1282	0.0958 -1.4635 -0.0076 0.7766	0.9237 0.0358 0.7611 0.3322	0.9991	0.4085
6	0.1756 0.1778 0.0784 -0.1282	0.0206 -1.4635 -0.0751 0.7766	0.9398 0.0358 0.9767 0.3322	0.9997	0.4198
7	0.1756 0.1778 -0.1282 -0.1282	-0.0127 -1.4635 -0.0076 0.7766	0.9403 0.0358 0.9676 0.3322	0.9997	0.4896
8	0.0784 0.1778 0.0457 -0.1282	-0.0076 -1.4635 0.0958 0.7766	0.9876 0.0358 0.9777 0.3322	0.9998	0.5251
9	0.0048 0.1778 0.0965 -0.1282	0.9569 -1.4635 -0.0076 0.7766	0.2089 0.0358 0.9814 0.3322	0.9996	0.4974
10	0.0784 0.1778 0.0784 -0.1282	0.0958 -1.4635 -0.0076 0.7766	0.9698 0.0358 0.9876 0.3322	0.9876	0.4712
20	-0.14945 0.1778 0.0807 -0.1282	1.3365 -1.4635 0.09536 0.7766	0.0583 0.0358 0.9693 0.3322	0.9992	0.4500
50	0.0784 0.1778 0.2072 -0.1282	-0.2181 -1.4635 0.0890 0.7766	0.8993 0.0358 0.9043 0.3322	0.9876	0.4845
100	-0.0102 0.1778 0.0410 -0.1282	-0.1450 -1.4635 -0.0447 0.7766	0.9587 0.0358 0.9926 0.3322	0.9987	0.5280
200	0.0784 0.1778 -0.1748	-0.0076 -1.4635 -0.0076	0.9876 0.0358 0.9409	0.9991	0.4930

	-0.1282	0.7766	0.3322		
500	-0.1903 0.1778 0.0784 -0.1282	0.7766 -1.4635 0.1078 0.7766	0.3218 0.0358 0.9652 0.3322	0.9989	0.4837
1000	0.0784 0.1778 0.0784 -0.1282	-0.1497 -1.4635 -0.0231 0.7766	0.9448 0.0358 0.9867 0.3322	0.9950	0.5254
5000	0.0150 0.1778 0.0268 -0.1282	-0.0076 -1.463 0.7766 0.7766	0.9994 0.0358 0.3408 0.3322	0.9995	0.4045
10000	0.0304 -0.4852 0.0152 1.3714	-0.0421 1.1670 -0.0107 -0.2593	0.9946 0.0779 0.9993 0.0483	0.9994	0.5503
20000	0.0518 0.5999 0.0516 -0.7646	-0.0555 -1.2734 -0.0052 -0.2291	0.9885 0.0462 0.9946 0.3230	0.9665	0.4504
50000	-0.0689 0.2563 0.0304 -1.957	0.0339 -1.6033 -0.0421 -1.0102	0.9882 0.0196 0.9946 0.0013	0.9982	0.4420
80000	-0.4952 -1.1524 -0.2572 -0.4952	-0.0914 -1.1595 -0.0107 -0.0914	0.6189 0.0188 0.8777 0.6189	0.9931	0.4330

Графики поколений:









Оптимальным решением, найденным алгоритмом, являются синяя точка Series1 на графике «5000 популяция»

Решение, найденное алгоритмом:  $x = -0.02354897$ ,  $y = 0.0247039713835315$ ,  $f(x, y) = 0.99767$

Ожидаемое решение:  $f(x, y) = 1$ ,  $x = 0,0$ ,  $y = 0,0$

Оптимальные значения:  $x = 0.0150$ ,  $y = -0.076$

Следовательно, алгоритм не нашел реальный максимум, но достаточно сильно приблизился к нему.

### **Вывод:**

Проделав работу, я изучила основные принципы действия генетических алгоритмов на примере поиска экстремума функции двух переменных; изучила основные шаги алгоритма, возможные критерии останова, виды алгоритмов селекции, операторов кроссовера и мутации.

По результатам численного эксперимента мы видим, что на 10 поколении результат ещё далековат от необходимого экстремума, но на 100 - совсем близок к идеальному.

Идея эволюционных алгоритмов в том, чтобы найти некое приближенное к лучшему, к оптимальному решению, которое скорее всего будет нас удовлетворять. Им можно найти применение для решения различных инженерных и практических задач.

К недостаткам генетических алгоритмов можно отнести ограниченность в способности нахождения верного решения. Они не могут регулировать логику развития поиска, существует вероятность, что в некоторых случаях алгоритм будет существенно ошибаться.

## **Приложение А**

*Код программы*

**Файл “program.cs”**

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.IO;
```

```
using System.Linq;
```

```
class Program
```

```
{  
  
    const int POPULATION_SIZE = 4;  
    const int NUM_GENERATIONS = 80000;  
    static double MUTATION_RATE = 0.25;  
    const double MIN_X = -2.0;  
    const double MAX_X = 2.0;  
    const double MIN_Y = -2.0;  
    const double MAX_Y = 2.0;  
    const int MAX_EXECUTION_TIME_SECONDS = 600;  
    static DateTime startTime = DateTime.Now;
```

```
class Individual
```

```
{  
  
    public double x;  
    public double y;  
    public double fitness;  
  
    public Individual(double x, double y)  
    {  
        this.x = x;  
        this.y = y;  
        this.fitness = 0.0;  
    }  
  
    public Individual() {}  
}
```

```
static List<Individual> CreateInitialPopulation()
```

```
{  
    List<Individual> population = new List<Individual>(POPULATION_SIZE);  
  
    Random random = new Random();  
    for (int i = 0; i < POPULATION_SIZE; ++i)  
    {  
        double x = random.NextDouble() * (MAX_X - MIN_X) + MIN_X;  
        double y = random.NextDouble() * (MAX_Y - MIN_Y) + MIN_Y;
```

```

        population.Add(new Individual(x, y));
    }

    return population;
}

static double FitnessFunction(double x, double y)
{
    return Math.Exp(-x * x - y * y) / (1 + x*x + y*y);
}

static List<Individual> EvaluatePopulation(List<Individual> population)
{
    foreach (var individual in population)
    {
        double fitness = FitnessFunction(individual.x, individual.y);
        individual.fitness = fitness;
    }
    return population;
}

static List<Individual> SelectParents(List<Individual> population)
{
    List<Individual> parents = new List<Individual>();
    const int TOURNAMENT_SIZE = 3;

    Random random = new Random();
    for (int i = 0; i < population.Count; ++i)
    {
        Individual bestParent = new Individual();
        double bestFitness = double.NegativeInfinity;

        for (int j = 0; j < TOURNAMENT_SIZE; ++j)
        {
            int randomIndex = random.Next(0, population.Count);
            Individual candidate = population[randomIndex];

```

```

        double distance = Math.Sqrt(Math.Pow(candidate.x - 0.653297871, 2) + Math.Pow(candidate.y +
0.00000000564618584, 2));

        double distanceWeight = Math.Exp(-0.1 * distance);

        double weightedFitness = candidate.fitness * distanceWeight;

        if (weightedFitness > bestFitness)
        {
            bestParent = candidate;
            bestFitness = weightedFitness;
        }
    }

    parents.Add(bestParent);
}

return parents;
}

static Individual Mutate(Individual individual)
{
    Random random = new Random();

    double x = individual.x;
    double y = individual.y;

    if (random.NextDouble() < MUTATION_RATE)
    {
        x += (random.NextDouble() * 2 - 1) * MUTATION_RATE;
        x = Math.Max(MIN_X, Math.Min(x, MAX_X));
    }

    if (random.NextDouble() < MUTATION_RATE)
    {
        y += (random.NextDouble() * 2 - 1) * MUTATION_RATE;
        y = Math.Max(MIN_Y, Math.Min(y, MAX_Y));
    }
}

```

```
}
```

```
Individual mutatedIndividual = new Individual(x, y);  
mutatedIndividual.fitness = FitnessFunction(x, y);
```

```
return mutatedIndividual;  
}
```

```
static Individual Crossover(Individual parent1, Individual parent2)
```

```
{  
    Random random = new Random();  
  
    double x = (random.NextDouble() < 0.5) ? parent1.x : parent2.x;  
    double y = (random.NextDouble() < 0.5) ? parent1.y : parent2.y;
```

```
    Individual child = new Individual(x, y);  
    child.fitness = FitnessFunction(x, y);
```

```
    return child;  
}
```

```
static bool CheckConvergence(List<Individual> population)
```

```
{  
    double sum = 0;  
    foreach (var p in population)  
    {  
        sum += p.fitness;  
    }  
    double average = sum / population.Count;
```

```
    int numConverged = 0;  
    double tolerance = 0.0001;
```

```
    foreach (var p in population)  
    {  
        if (Math.Abs(p.fitness - average) < tolerance)  
        {
```

```

        numConverged++;
    }
}

double convergenceRatio = (double)numConverged / population.Count;
return (convergenceRatio >= 0.7);
}

static void PrintToTxt(List<Individual> population, int i)
{
    string filePath = @"C:\Users\mi\source\repos\mo-hw2\result.txt";
    using (StreamWriter writer = new StreamWriter(filePath, true))
    {
        double sum = 0;
        List<int> iterationsToPrint = new List<int>
        {
            0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100, 200, 500, 1000, 5000, 10000, 20000, 50000, 80000
        };

        if (iterationsToPrint.Contains(i))
        {
            writer.WriteLine(i);
            writer.WriteLine("X:");
            foreach (var ind in population)
            {
                writer.WriteLine(ind.x);
            }
            writer.WriteLine("Y:");
            foreach (var ind in population)
            {
                writer.WriteLine(ind.y);
            }
            writer.WriteLine("F:");
            foreach (var ind in population)
            {
                sum += ind.fitness;
                writer.WriteLine(ind.fitness);
            }
        }
    }
}

```

```

    }

    Individual bestIndividual = population.OrderByDescending(p => p.fitness).First();
    double maxF = bestIndividual.fitness;
    writer.WriteLine("MAX:" + maxF);
    writer.WriteLine("AVG:" + sum / population.Count);
    writer.WriteLine();
}
}
}

```

```

static void Main()
{
    List<Individual> population = CreateInitialPopulation();
    double initialMutationRate = MUTATION_RATE;

    for (int generation = 0; generation < NUM_GENERATIONS; generation++)
    {
        population = EvaluatePopulation(population);
        List<Individual> parents = SelectParents(population);
        PrintToTxt(population, generation);

        double currentConvergence = (double)generation / NUM_GENERATIONS;
        MUTATION_RATE = initialMutationRate * (1.0 - currentConvergence);

        for (int j = 0; j < POPULATION_SIZE - 1; j += 2)
        {
            Individual child1 = Crossover(parents[j], parents[j + 1]);
            child1 = Mutate(child1);

            population[j] = child1;

            population[j].x = child1.x;
            population[j].y = child1.y;
            population[j].fitness = FitnessFunction(population[j].x, population[j].y);
        }

        var currentTime = DateTime.Now;
        var executionTime = (int)(currentTime - startTime).TotalSeconds;
    }
}

```

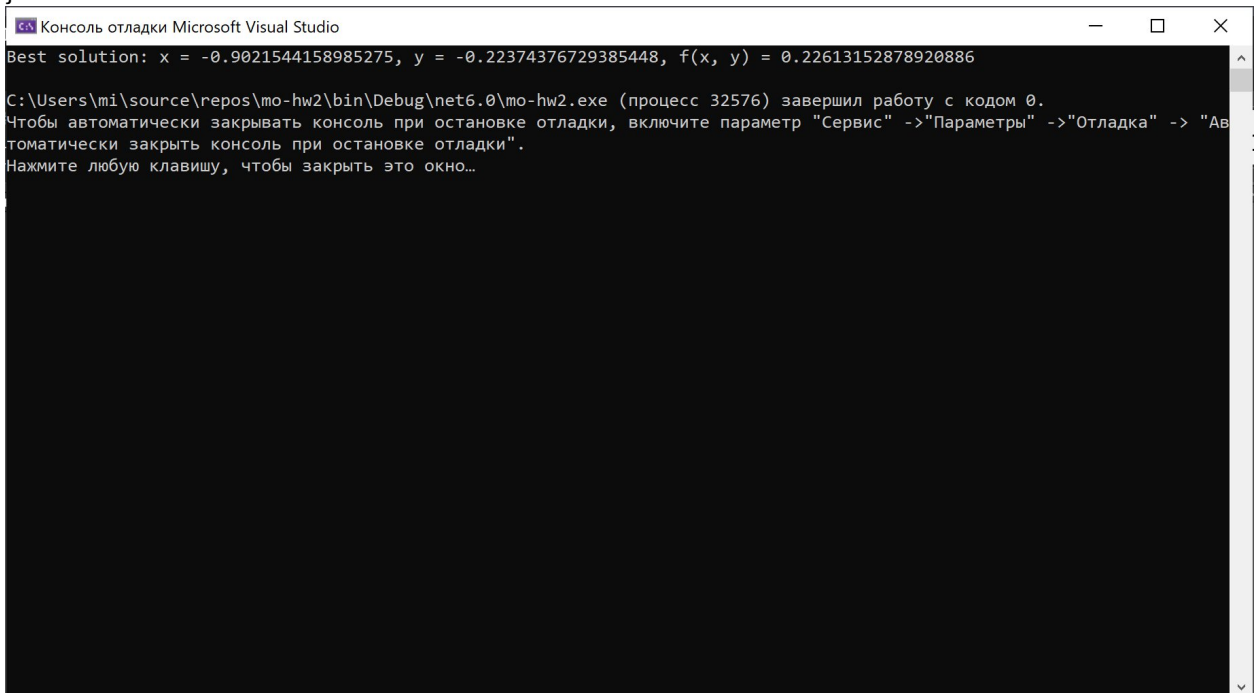
```

        if (executionTime >= MAX_EXECUTION_TIME_SECONDS)
        {
            Console.WriteLine("RunTime Limit");
            break;
        }

        if (CheckConvergence(population))
        {
            Console.WriteLine($"Fitness Convergence {generation}");
            break;
        }

        Individual bestIndividual = population.OrderByDescending(p => p.fitness).First();
        Console.WriteLine($"Best solution: x = {bestIndividual.x}, y = {bestIndividual.y}, f(x, y) = {bestIndividual.fitness}");
    }
}
}
}

```



Консоль отладки Microsoft Visual Studio

```

Best solution: x = -0.9021544158985275, y = -0.22374376729385448, f(x, y) = 0.22613152878920886

C:\Users\mi\source\repos\mo-hw2\bin\Debug\net6.0\mo-hw2.exe (процесс 32576) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" -> "Параметры" -> "Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно...

```