



МИНИСТЕРСТВО НАУКИ  
И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное  
образовательное учреждение высшего образования  
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



**НГТУ  
НЭТИ** | **Факультет прикладной  
математики и информатики**

Кафедра прикладной математики  
Практическая работа №3  
по дисциплине «Цифровые модели и оценивание параметров»

## ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ

Группа ПМ-92      АРТЮХОВ РОМАН  
ВАСЬКИН ЛЕОНИД

Преподаватели      ВАГИН ДЕНИС ВЛАДИМИРОВИЧ

Новосибирск, 2022

## Цель работы:

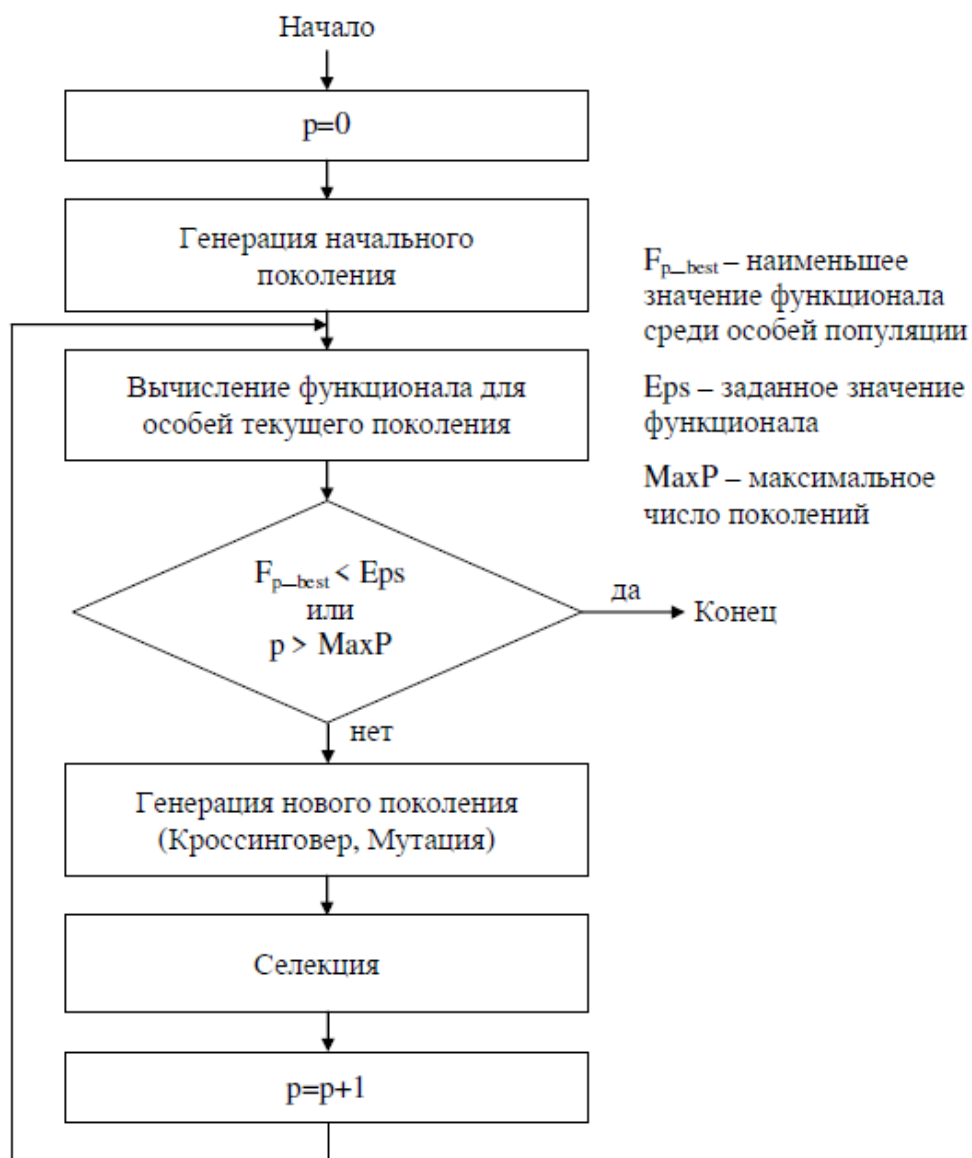
Реализовать ПГА для решения обратных задач.

## Задание

Реализовать ПГА. Генотип состоит из  $N$  чисел. Фенотип состоит из  $M$  чисел. Числа фенотипа вычисляются регулярным образом из чисел генотипа.

С помощью ПГА восстановить коэффициенты полинома десятой степени по его значениям в заданном наборе точек. Значения зашумить на 1, 2, 5%.

## ПГА:



- Входные данные

```

1  /** Данные для задачи
2  public double[] Coefficients    { get; set; }    /// Коэффициенты полинома (генотип)
3  public double[] Points         { get; set; }    /// Заданный набор точек (фенотип)
4  public double MinFunctional     { get; set; }    /// Значение функционала для выхода
5  public double MutationProbability { get; set; } /// Вероятность мутации
6  public uint CountGen           { get; set; }    /// Число генов
7  public uint CountPopulation     { get; set; }    /// Число особей в популяции
8  public uint CountGeneration     { get; set; }    /// Число генераций
9  public uint MaxParent          { get; set; }    /// Максимальное число родителей

```

### • Генерация начального поколения

Создание особей. Для каждой особи случайно выбираются гены, высчитывается фенотип особи по созданному генотипу. По созданному фенотипу высчитывается функционал особи. Далее особь добавляется в поколение.

```

1  // Генерация начального поколения
2  Population = new List<Being>();
3  for (int i = 0; i < CountPopulation; i++) {
4      Vector<double> Gens = new Vector<double>((int)CountGen);
5      for (int j = 0; j < CountGen; j++)
6          Gens[j] = (GetRandomDouble(MinValue, MaxValue));
7
8      var being = new Being(Gens);
9      being.SetPhenotip(Points);
10     being.Functional = Functional(being, TrueBeing);
11     Population.Add(being);
12 }
13 Population = Population.OrderBy(item => item.Functional).ToList();

```

### • Высчитывание функционала особи

Значение функционала – сумма модулей разности чисел фенотипа текущей особи и фенотипа истинной особи.

```

1  /** Расчет функционала
2  public static double Functional(Being CurBeing, Being TrueBeing) {
3      double f = 0;
4      for (int i = 0; i < CurBeing.Phenotype.Length; i++)
5          f += Abs(TrueBeing.Phenotype[i] - CurBeing.Phenotype[i]);
6      return f;
7  }

```

### • Кроссинговер

На вход приходят 2 родителя. Генерируются две случайные точки. До первой точки, гены берутся от отца. От первой точки до второй гены матери. От второй точки до конца, снова берутся гены отца.

```
1  /** Кроссинговер (скрещивание отца и матери)
2  public Being GetChild(Being mother, Being father) {
3      Being child = new Being((int)CountGen, Points.Length);
4      int index1 = (new Random()).Next(0, (int)CountGen); // Первая точка кроссинговера
5      int index2 = (new Random()).Next(index1, (int)CountGen); // Вторая точка Кроссинговера
6
7      // Первая часть генотипа от отца
8      for (int i = 0; i < index1; i++)
9          child.Genotype[i] = father.Genotype[i];
10
11     // Вторая часть генотипа от матери
12     for (int i = index1; i < index2; i++)
13         child.Genotype[i] = mother.Genotype[i];
14
15     // Третья часть генотипа от отца
16     for (int i = index2; i < CountGen; i++)
17         child.Genotype[i] = father.Genotype[i];
18
19     return child;
20 }
```

- **Мутация**

Генерируется вероятность мутации. Если она меньше входной мутации, мутируем особь. Выбираем ген для мутации и задаем для этого гена случайное число.

```
1  /** Проявление мутации у существа
2  public void Mutation(Being being) {
3      double probability = GetRandomDouble(0, 1); // Вероятность мутации
4
5      // Если меньше заданной мутации (мутируем существо)
6      if (probability < MutationProbability) {
7          int index = (new Random()).Next(0, (int)CountGen); // Рандомное место для мутации
8          being.Genotype[index] = GetRandomDouble(MinValue, MaxValue);
9      }
10 }
```

- **Селекция**

Сортируем новое поколение по значению функционала. Первую половину поколения отбираем, вторую уничтожаем.

```
1  /* Селекция (отбор лучших существ)
2  public void Selection(List<Being> population, double bestFunctional) {
3      // Сортируем новое поколение
4      Population = population.OrderBy(item => item.Functional).ToList();
5
6      // Оставляем лучших существ
7      Population.RemoveRange((int)CountPopulation, (int)CountPopulation);
8  }
```

## Исследования:

- **Размер фенотипа (5, 10, 20 точек)**

*Coefficients* : [1, 2, 3]  
*Points* : [0.1, 0.2, 0.3, 0.4, 0.5]  
*MinFunctional* : 0.01  
*MutationProbability* : 0.8  
*CountGen* : 3  
*CountPopulation* : 40  
*CountGeneration* : 1000  
*MaxParent* : 4

**Вывод:** Gens: [0,568, 2,311, 2,953]; Functional = 0,03791658068773085

*Coefficients* : [1, 2, 3]  
*Points* : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]  
*MinFunctional* : 0.01  
*MutationProbability* : 0.8  
*CountGen* : 3  
*CountPopulation* : 40  
*CountGeneration* : 1000  
*MaxParent* : 4

**Вывод:** Gens: [1,407, 1,493, 3,132]; Functional = 0,273717064297085

*Coefficients* : [1, 2, 3]  
*Points* : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2]  
*MinFunctional* : 0.01  
*MutationProbability* : 0.8  
*CountGen* : 3  
*CountPopulation* : 40  
*CountGeneration* : 1000  
*MaxParent* : 4

**Вывод:** Gens: [1,288, 1,354, 3,294]; Functional = 1,4645878006383968

Чем меньше размер фенотипа, функционал ниже, но плохая точность решения. Чем больше размер фенотипа, функционал выше, но получше точность решения.

- **Размер генотипа (3, 6, 10)**

*Coefficients* : [1, 2, 3]  
*Points* : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]  
*MinFunctional* : 0.01  
*MutationProbability* : 0.8  
*CountGen* : 3  
*CountPopulation* : 40  
*CountGeneration* : 1000  
*MaxParent* : 4

**Вывод:** Gens: [0,800, 2,222, 2,951]; Functional = 0,12100022993219817

*Coefficients* : [1, 2, 3, 4, 5, 6]  
*Points* : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]  
*MinFunctional* : 0.01  
*MutationProbability* : 0.8  
*CountGen* : 6  
*CountPopulation* : 40  
*CountGeneration* : 1000  
*MaxParent* : 4

**Вывод:**

Gens: [0,863, 2,210, 4,573, 0,902, 6,671, 5,780]; Functional = 0,3008514004074039

$\left\{ \begin{array}{l} \text{Coefficients} : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \\ \text{Points} : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1] \\ \text{MinFunctional} : 0.01 \\ \text{MutationProbability} : 0.8 \\ \text{CountGen} : 10 \\ \text{CountPopulation} : 40 \\ \text{CountGeneration} : 1000 \\ \text{MaxParent} : 4 \end{array} \right.$

**Вывод:**

Gens: [2,288, 1,311, 3,105, 1,389, 5,835, 6,641, 8,810, 6,343, 9,276, 9,996];

Functional = 0,1219316917995652

• **Вероятность мутации (0.2)**

$\left\{ \begin{array}{l} \text{Coefficients} : [1, 2, 3] \\ \text{Points} : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1] \\ \text{MinFunctional} : 0.01 \\ \text{MutationProbability} : 0.2 \\ \text{CountGen} : 3 \\ \text{CountPopulation} : 40 \\ \text{CountGeneration} : 1000 \\ \text{MaxParent} : 4 \end{array} \right.$

**Вывод:** Gens: [1,691, 1,096, 3,253]; Functional = 0,4932298154722412

• **Вероятность мутации (0.99)**

$\left\{ \begin{array}{l} \text{Coefficients} : [1, 2, 3] \\ \text{Points} : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1] \\ \text{MinFunctional} : 0.01 \\ \text{MutationProbability} : 0.99 \\ \text{CountGen} : 3 \\ \text{CountPopulation} : 40 \\ \text{CountGeneration} : 1000 \\ \text{MaxParent} : 4 \end{array} \right.$

**Вывод:** Gens: [1,161, 1,813, 3,044]; Functional = 0,1017336436644447

Мутация очень хорошо влияет как на функционал, так и на получаемое решение.

## Helper.cs

```
namespace PGA.other;
public static class Helper
{
    public static readonly double MinValue = 0;    /// Минимальное значение для генерации
    public static readonly double MaxValue = 10;   /// Максимальное значение для генерации
    public static readonly double Noise = 0.5;     /// Значение шума

    /// Расчет функционала
    public static double Functional(Being CurBeing, Being TrueBeing) {
        double f = 0;
        for (int i = 0; i < CurBeing.Phenotype.Length; i++)
            f += Abs(TrueBeing.Phenotype[i] - CurBeing.Phenotype[i]);
        return f;
    }

    /// Генерация случайного числа в диапазоне
    public static double GetRandomDouble(double Min, double Max) => (new Random()).NextDouble() * (Max - Min) + Min;

    /// Окно помощи при запуске (если нет аргументов или по команде)
    public static void ShowHelp() {
        WriteLine("----Команды----\n" +
            "-help          - показать справку\n" +
            "-i              - входной файл\n");
    }
}
```

## Data.cs

```
namespace PGA;
public class Data
{
    /// Данные для задачи
    public double[] Coefficients { get; set; }    /// Коэффициенты полинома (генотип)
    public double[] Points { get; set; }         /// Заданный набор точек (фенотип)
    public double MinFunctional { get; set; }     /// Значение функционала для выхода
    public double MutationProbability { get; set; } /// Вероятность мутации
    public uint CountGen { get; set; }           /// Число генов
    public uint CountPopulation { get; set; }     /// Число особей в популяции
    public uint CountGeneration { get; set; }     /// Число генераций
    public uint MaxParent { get; set; }          /// Максимальное число родителей

    /// Деконструктор
    public void Deconstruct(out Vector<double> coefs,
        out Vector<double> points,
        out double minFunctional,
        out double mutationProbability,
        out uint countGen,
        out uint countPopulation,
        out uint countGeneration,
        out uint maxParent)
    {
        coefs = new Vector<double>(Coefficients);
        points = new Vector<double>(Points);
        minFunctional = this.MinFunctional;
        mutationProbability = this.MutationProbability;
        countGen = this.CountGen;
        countPopulation = this.CountPopulation;
        countGeneration = this.CountGeneration;
        maxParent = this.MaxParent;
    }
}
```



```

    /** Проверка входных данных
public bool Incorrect(out string mes) {
    StringBuilder errorStr = new StringBuilder("");

    if (CountGen != Coefficients.Count())
        errorStr.Append($"Incorrect data (CountGen = {Coefficients.Count()}: {CountGen} = {Coefficients.Count()}\n");

    if (MutationProbability < 0.0 && MutationProbability > 1.0)
        errorStr.Append($"Incorrect data (MutationProbability out of range): 0 < {MutationProbability} < 1\n");

    if (!errorStr.ToString().Equals("")) {
        mes = errorStr.ToString();
        return false;
    }

    mes = errorStr.ToString();
    return true;
}
}

```

## Being.cs

```

namespace PGA;

/** Класс особи
public class Being
{
    /** Фенотипы и генотипы особи
    public Vector<double> Phenotype { get; set; }    /// Фенотип (значение в точках)
    public Vector<double> Genotype { get; set; }    /// Генотип (значения коэффициентов)

    public double Functional { get; set; } = Double.MaxValue;    /// Значение функционала

    // ***** Конструктор ***** //
    public Being(Vector<double> gen) {
        Genotype = new Vector<double>(gen.Length);
        Vector<double>.Copy(gen, Genotype);
    }

    public Being(int CountGens, int CountPoints) {
        Genotype = new Vector<double>(CountGens);
    }

    /** Расчет полинома от точки с генами особи
    public double Polynom(double x) {
        double degreeX = 1;    /// перемен. хранит степени X
        double sum = 0;    /// результат полинома
        for (int i = Genotype.Length - 1; i >= 0; i--) {
            sum += Genotype[i] * degreeX;
            degreeX *= x;
        }
        return sum;
    }

    /** Изменение фенотипа особи
    public void SetPhenotip(Vector<double> points) {
        Phenotype = new Vector<double>(points.Length);
        for (int i = 0; i < Phenotype.Length; i++)
            Phenotype[i] = Polynom(points[i]);
    }

    /** Строковое представление особи
    public override string ToString() {

```

```

        StringBuilder str = new StringBuilder();
        str.Append($"Gens: [");
        for (int i = 0; i < Genotype.Length - 1; i++)
            str.Append(Genotype[i].ToString("F3") + ", ");
        str.Append(Genotype[Genotype.Length - 1].ToString("F3"));
        str.Append($"]; Functional = {Functional}");
        return str.ToString();
    }
}

```

## Genetica.cs

```
namespace PGA;
```

```
/* Генетика
```

```
public class Genetica
```

```
{
```

```
    /* Данные для задачи
```

```
    protected Vector<double> Coefficients { get; set; }    /// Коэффициенты полинома
```

```
    protected Vector<double> Points          { get; set; }    /// Заданный набор точек (фенотип)
```

```
    protected double MinFunctional           { get; set; }    /// Значение функционала для выхода
```

```
    protected double MutationProbability    { get; set; }    /// Вероятность мутации
```

```
    protected uint CountGen                  { get; set; }    /// Число генов
```

```
    protected uint CountPopulation           { get; set; }    /// Число особей в популяции
```

```
    protected uint CountGeneration          { get; set; }    /// Число генераций
```

```
    protected uint MaxParent                 { get; set; }    /// Максимальное число родителей
```

```
    private Being TrueBeing;                /// Особь с правильным генотипом
```

```
    private List<Being> Population;         /// Поколение
```

```
    // ***** Конструктор ***** //
```

```
    public Genetica(Data data) {
```

```
        (Coefficients, Points, MinFunctional, MutationProbability, CountGen, CountPopulation, CountGeneration, MaxParent) =
```

```
        // Истинное существо и его функционал (с зашумлением)
```

```
        double noise = 1 + GetRandomDouble(-Noise, Noise);
```

```
        Vector<double> trueGens = Coefficients * noise;
```

```
        TrueBeing = new Being(trueGens);
```

```
        TrueBeing.SetPhenotip(Points);
```

```
        TrueBeing.Functional = Functional(TrueBeing, TrueBeing);
```

```
        // Генерация начального поколения
```

```
        Population = new List<Being>();
```

```
        for (int i = 0; i < CountPopulation; i++) {
```

```
            Vector<double> Gens = new Vector<double>((int)CountGen);
```

```
            for (int j = 0; j < CountGen; j++)
```

```
                Gens[j] = (GetRandomDouble(MinValue, MaxValue));
```

```
            var being = new Being(Gens);
```

```
            being.SetPhenotip(Points);
```

```
            being.Functional = Functional(being, TrueBeing);
```

```
            Population.Add(being);
```

```
        }
```

```
        Population = Population.OrderBy(item => item.Functional).ToList();
```

```
    }
```

```
    /* Реализация простого генетического алгоритма
```

```
    public void pga() {
```

```
        int Iter = 0;                                /// Текущее количество итераций
```

```
        double Functional = Population[0].Functional; /// Текущий лучший функционал
```

```
        do {
```

```
            // Создаем новое поколение
```

```
            List<Being> newPopulation = NewPopulation();
```

```

        // Производим селекцию
        Selection(newPopulation, Population[0].Functional);

    } while (++Iter < CountGeneration &&
        Population[0].Functional > MinFunctional);

    WriteLine(Population[0].ToString());
}

/** Генерация нового поколения
public List<Being> NewPopulation() {
    List<Being> NewPopul = new List<Being>(Population);
    // Для каждого мамы-родителя
    for (int i = 0; i < MaxParent; i++)
        // Генерируем по (CountPopulation / MaxParent) детей
        for (int j = 0; j < CountPopulation / MaxParent; j++) {
            Being father = GetBeing(); // Выбираем randomного отца
            Being child = GetChild(Population[i], father); // Скрещиваем отца и мать (Кроссинговер)
            Mutation(child); // Вероятность проявление мутации у созданного ребенка
            child.SetPhenotip(Points); // Изменение фенотипа
            child.Functional = Functional(child, TrueBeing);
            NewPopul.Add(child);
        }
    return NewPopul;
}

/** Выбор randomного существа из поколения
public Being GetBeing() {
    int index = (new Random()).Next(0, (int)CountPopulation);
    return Population[index];
}

/** Кроссинговер (скрещивание отца и матери)
public Being GetChild(Being mother, Being father) {
    Being child = new Being((int)CountGen, Points.Length);
    int index1 = (new Random()).Next(0, (int)CountGen); // Первая точка кроссинговера
    int index2 = (new Random()).Next(index1, (int)CountGen); // Вторая точка Кроссинговера

    // Первая часть генотипа от отца
    for (int i = 0; i < index1; i++)
        child.Genotype[i] = father.Genotype[i];

    // Вторая часть генотипа от матери
    for (int i = index1; i < index2; i++)
        child.Genotype[i] = mother.Genotype[i];

    // Третья часть генотипа от отца
    for (int i = index2; i < CountGen; i++)
        child.Genotype[i] = father.Genotype[i];

    return child;
}

/** Проявление мутации у существа
public void Mutation(Being being) {
    double probability = GetRandomDouble(0, 1); // Вероятность мутации

    // Если меньше заданной мутации (мутируем существо)
    if (probability < MutationProbability) {
        int index = (new Random()).Next(0, (int)CountGen); // Randomное место для мутации
        being.Genotype[index] = GetRandomDouble(MinValue, MaxValue);
    }
}

/** Селекция (отбор лучших существ)
public void Selection(List<Being> population, double bestFunctional) {
    // Сортируем новое поколение
    Population = population.OrderBy(item => item.Functional).ToList();
}

```

```
// Оставляем лучших особей
Population.RemoveRange((int)CountPopulation, (int)CountPopulation);
}

}
```

## Program.cs

```
try {
    // Проверка аргументов
    if (args.Length == 0) throw new ArgumentNullException("Not found arguments!");
    if (args[0] == "-help") {
        ShowHelp(); return;
    }

    // Входные данные
    string json = File.ReadAllText(args[1]);
    Data data = JsonConvert.DeserializeObject<Data>(json)!;
    if (data is null) throw new FileNotFoundException("File uncorrected!");

    // Проверка входных данных
    if (!data.Incorrect(out string mes)) throw new ArgumentException(mes);

    // Решение задача
    Genetica genetica = new Genetica(data);
    genetica.pga();
}
catch (FileNotFoundException ex) {
    WriteLine(ex.Message);
}
catch (ArgumentNullException ex) {
    ShowHelp();
    WriteLine(ex.Message);
}
catch (ArgumentException ex) {
    WriteLine(ex.Message);
}
```