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 рандомного числа в диапозоне
    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 фукнционала 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;    /// перем. хранит степени X
        double sum = 0;        /// рещультат 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;

//* Генетека
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 фукнционала 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 рандомного 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 рандомного 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); // Рандомное 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);
}
```