Московский авиационный институт

(национальный исследовательский университет)

**Институт информационных технологий и прикладной математики**

**Отчет по курсовому проекту**

**по объектно-ориентированному программированию**

|  |  |
| --- | --- |
| Работу выполнил: | Шиляева Н. С. |
| Группа: | М8O-204Б-18 |
| Руководитель: | Кузнецова С. В. |
| Подпись: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Москва, 2019**

**Оглавление**

[**Постановка задачи** 2](#_Toc27848057)

[Оригинальное описание алгоритма: 3](#_Toc27848058)

[Перевод: 3](#_Toc27848059)

[**Алгоритм** 4](#_Toc27848060)

[*Создание начальной популяции* 5](#_Toc27848061)

[*Селекция (отбор)* 6](#_Toc27848062)

[*Скрещивание* 6](#_Toc27848063)

[*Мутация* 7](#_Toc27848064)

[**Диаграмма классов** 7](#_Toc27848065)

[**Реализация** 7](#_Toc27848066)

[**Классы алгоритма** 9](#_Toc27848067)

[*Класс сравнения хромосом* 9](#_Toc27848068)

[*Класс хромосом* 9](#_Toc27848069)

[*Класс алгоритма* 10](#_Toc27848070)

[**Результат работы** 13](#_Toc27848071)

[**Вывод** 16](#_Toc27848072)

[**Литература** 17](#_Toc27848073)

# **Постановка задачи**

Алгоритм 33. Генетический алгоритм с элитарностью.

Необходимо реализовать с применением паттернов проектирования и визуализацией один из метаэвристических алгоритмов в терминах объектно-ориентированного программирования на языке C#.

## Оригинальное описание алгоритма:

Elitism is simple: we augment the Genetic Algorithm to directly inject into the next population the ﬁttest individual or individuals from the previous population. These individuals are called the elites. By keeping the best individual (or individuals) around in future populations, this algorithm begins to resemble (µ + l), and has similar exploitation properties. This exploitation can cause premature convergence if not kept in check: perhaps by increasing the mutation and crossover noise, or weakening the selection pressure, or reducing how many elites are being stored. A minor catch. If you want to maintain a population size of popsize, and you’re doing crossover, you’ll need to have popsize, minus the number of elites, be divisible by two, as in this algorithm:

1: popsize <- desired population size

2: n <- desired number of elite individuals (popsizen should be even)

3: P <- {}

4: for popsize times do

5: P <- P[{new random individual}]

6: Best <- res

7: repeat

8: for each individual Pi 2 P do

9: AssessFitness(Pi)

10: if Best = res or Fitness(Pi) > Fitness(Best) then

11: Best <- Pi

12: Q <- {the n ﬁttest individuals in P, breaking ties at random}

13: for (popsize - n)/2 times do

14: Parent Pa <- SelectWithReplacement(P)

15: Parent Pb <- SelectWithReplacement(P)

16: Children Ca,Cb <- Crossover(Copy(Pa), Copy(Pb))

17: Q <- Q[{Mutate(Ca), Mutate(Cb)}]

18: P <- Q

19: until Best is the ideal solution or we have run out of time

20: return Best

## Перевод:

Элитарность проста: мы используем Генетический алгоритм, чтобы напрямую ввести в численность следующей популяции наиболее устойчивую(-ых) особей из предыдущей популяции. Эти особи называются элитами. Сохраняя лучших особей (или индивидов) для будущих популяций, этот алгоритм начинает напоминать (µ + l) и обладает аналогичными эксплуатационными свойствами. Эта эксплуатация может вызвать преждевременную конвергенцию, если ее не сдерживать: возможно, путем увеличения мутации и кроссовера, или ослабления селекции, или уменьшения того, сколько элит сохранится. Напоследок. Если вы хотите сохранить размер численности популяции, и вы совершаете кроссовер, вы должны иметь размер популяции минус количество элит, например, две, как в данном алгоритме.

На базе данного алгоритма я решила реализовать задачу оптимизации функции путем поиска ее глобального максимума.

# **Алгоритм**

*Генети́ческий алгори́тм* — это [эвристический алгоритм](https://ru.wikipedia.org/wiki/%D0%AD%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных [естественному отбору](https://ru.wikipedia.org/wiki/%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BE%D1%82%D0%B1%D0%BE%D1%80) в природе. Является разновидностью [эволюционных вычислений](https://ru.wikipedia.org/wiki/%D0%AD%D0%B2%D0%BE%D0%BB%D1%8E%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких как [наследование](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%B1%D0%B8%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F)), [мутации](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D1%82%D0%B0%D1%86%D0%B8%D1%8F), [отбор](https://ru.wikipedia.org/wiki/%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BE%D1%82%D0%B1%D0%BE%D1%80) и [кроссинговер](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%BE%D1%81%D1%81%D0%B8%D0%BD%D0%B3%D0%BE%D0%B2%D0%B5%D1%80). Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

Задача формализуется таким образом, чтобы её решение могло быть закодировано в виде [вектора](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) («[генотипа](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%BE%D1%82%D0%B8%D0%BF)») генов, где каждый ген может быть [битом](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), числом или неким другим объектом. В классических реализациях генетического алгоритма (ГА) предполагается, что генотип имеет фиксированную длину. Однако существуют вариации ГА, свободные от этого ограничения.

Некоторым, обычно случайным, образом создаётся множество генотипов начальной популяции. Они оцениваются с использованием «[функции приспособленности](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8)», в результате чего с каждым генотипом ассоциируется определённое значение («приспособленность»), которое определяет насколько хорошо [фенотип](https://ru.wikipedia.org/wiki/%D0%A4%D0%B5%D0%BD%D0%BE%D1%82%D0%B8%D0%BF), им описываемый, решает поставленную задачу.

Из полученного множества решений («поколения») с учётом значения «приспособленности» выбираются решения (обычно лучшие особи имеют большую вероятность быть выбранными), к которым применяются «генетические операторы» (в большинстве случаев «[скрещивание](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D1%80%D0%B5%D1%89%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)» — crossover и «[мутация](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D1%82%D0%B0%D1%86%D0%B8%D1%8F)» — mutation), результатом чего является получение новых решений. Для них также вычисляется значение приспособленности, и затем производится отбор («селекция») лучших решений в следующее поколение.

Этот набор действий повторяется итеративно, так моделируется «эволюционный процесс», продолжающийся несколько жизненных циклов (*поколений*), пока не будет выполнен критерий остановки алгоритма. Таким критерием может быть:

* нахождение глобального, либо субоптимального решения;
* исчерпание числа поколений, отпущенных на эволюцию;
* исчерпание времени, отпущенного на эволюцию.

Генетические алгоритмы служат, главным образом, для поиска решений в многомерных пространствах поиска.

Таким образом, можно выделить следующие этапы генетического алгоритма:

1. Задать целевую функцию (приспособленности) для особей популяции
2. Создать начальную популяцию

* (Начало цикла)

1. Размножение (скрещивание)
2. Мутирование
3. Вычислить значение целевой функции для всех особей
4. Формирование нового поколения (селекция)
5. Если выполняются условия остановки, то (конец цикла), иначе (начало цикла).

## *Создание начальной популяции*

Перед первым шагом нужно случайным образом создать начальную популяцию; даже если она окажется совершенно неконкурентоспособной, вероятно, что генетический алгоритм всё равно достаточно быстро переведёт её в жизнеспособную популяцию. Таким образом, на первом шаге можно особенно не стараться сделать слишком уж приспособленных особей, достаточно, чтобы они соответствовали формату особей популяции, и на них можно было подсчитать функцию приспособленности (Fitness). Итогом первого шага является популяция H, состоящая из N особей.

## *Селекция (отбор)*

На этапе отбора нужно из всей популяции выбрать определённую её долю, которая останется «в живых» на этом этапе эволюции. Есть разные способы проводить отбор. Вероятность выживания особи h должна зависеть от значения функции приспособленности Fitness(h). Сама доля выживших s обычно является параметром генетического алгоритма, и её просто задают заранее. По итогам отбора из N особей популяции H должны остаться sN особей, которые войдут в итоговую популяцию H'. Остальные особи погибают.

## *Скрещивание*

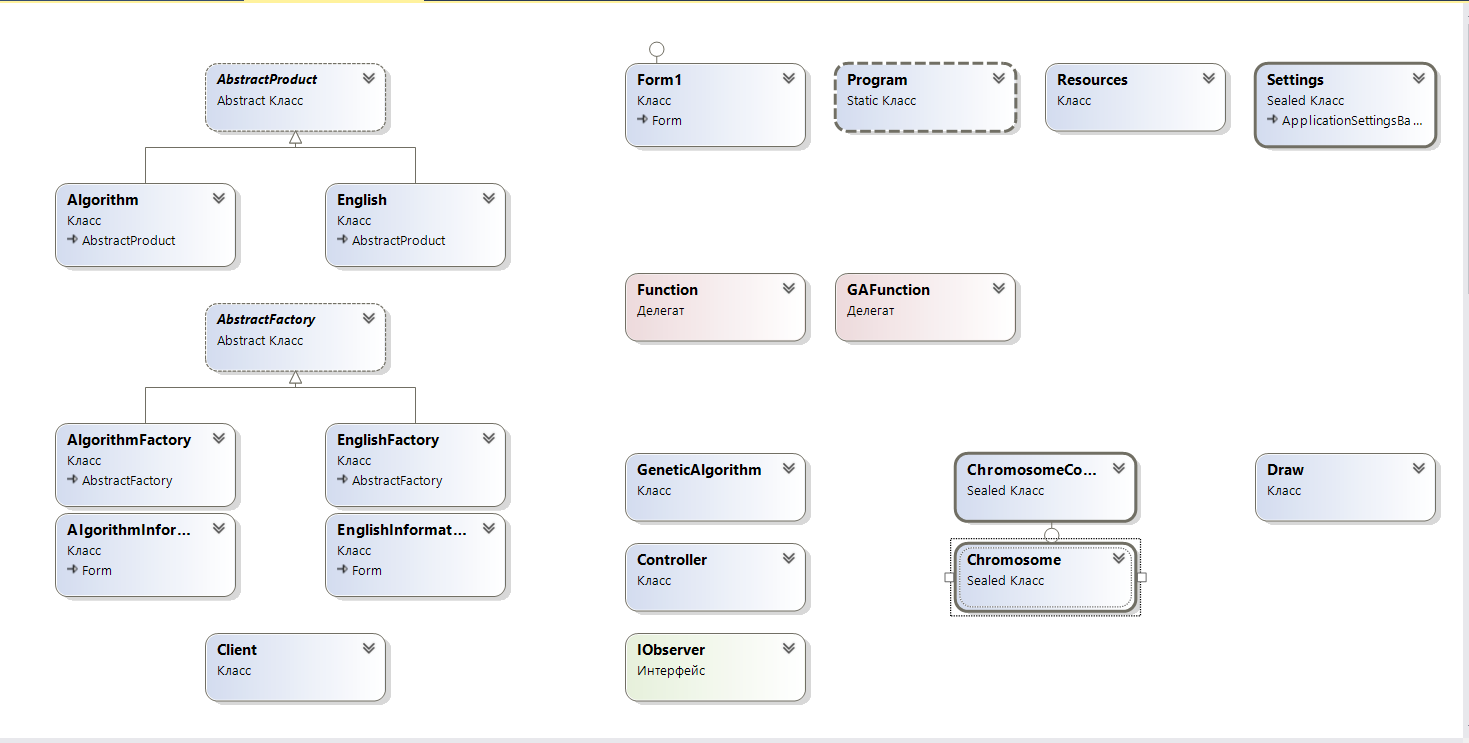
Размножение в разных алгоритмах определяется по-разному — оно, конечно, зависит от представления данных. Главное требование к размножению — чтобы потомок или потомки имели возможность унаследовать черты обоих родителей, «смешав» их каким-либо способом.

Почему особи для размножения обычно выбираются из всей популяции H, а не из выживших на первом шаге элементов H' (хотя последний вариант тоже имеет право на существование)? Дело в том, что главный недостаток многих генетических алгоритмов — отсутствие разнообразия (diversity) в особях. Достаточно быстро выделяется один-единственный генотип, который представляет собой локальный максимум, а затем все элементы популяции проигрывают ему отбор, и вся популяция «забивается» копиями этой особи. Есть разные способы борьбы с таким нежелательным эффектом; один из них — выбор для размножения не самых приспособленных, но вообще всех особей. Однако такой подход вынуждает хранить всех существовавших ранее особей, что увеличивает вычислительную сложность задачи. Поэтому часто применяют методы отбора особей для скрещивания таким образом, чтобы «размножались» не только самые приспособленные, но и другие особи, обладающие плохой приспособленностью. При таком подходе для разнообразия генотипа возрастает роль мутаций.

## *Мутация*

К мутациям относится все то же самое, что и к размножению: есть некоторая доля мутантов m, являющаяся параметром генетического алгоритма, и на шаге мутаций нужно выбрать mN особей, а затем изменить их в соответствии с заранее определёнными операциями мутации.

# **Диаграмма классов**



# **Реализация**

Программа разработана при помощи интегрированной среды разработки Microsoft Visual Studio 2019 на языке программирования C# с использованием Windows Forms. В качестве оборудования для разработки курсового проекта использовался Ноутбук HP Pavilion Power 15-cb011ur.

Основу структуры программы составляют паттерны MVC, Abstract Factory и Singleton.

Код алгоритма содержит два простых класса, GeneticAlgorithm и Chromosome, плюс вспомогательный класс ChromosomeComparer.  
Класс Chromosome можно считать простым контейнером. Нижележащая структура является массивом вещественных чисел двойной точности в диапазоне от 0 до 1. Ожидается, что пользователь берет эти значения и масштабирует их до любых нужных значений. (Я выставила значения в пределах от 0 до 10, так как график функции прорисовывается именно на таком ограничении). Так как мутация происходит в хромосоме, метод Mutate находится в этом классе. Оператор кроссовера требует доступа к закрытым данным хромосомы, поэтому он является функцией-членом, принимающей вторую хромосому и выдающей две дочерних хромосомы. Пригодность конкретной хромосомы также хранится в объекте генома. В коде есть еще дополнительные вспомогательные функции.

Класс GeneticAlgorithm делает всю работу. Генетический алгоритм состоит из следующих базовых шагов:  
1. Создать новую популяцию  
2. Выбрать двух индивидуумов из популяции, взвешивая относительно индивидуума, представляющего лучшее на настоящий момент решение.  
3. 'Размножить' их, чтобы произвести потомство.  
4. Если нет достаточного потомства для новой популяции, вернуться к шагу 2.  
5. Заменить старую популяцию новой.  
6. Если не было произведено достаточно поколений, вернуться к шагу 2.  
7. Конец.

Кроссовер берет две хромосомы, разделяет их в некоторой точке и порождает две новых путем обмена местами конечных частей, например:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 10 20 30 40 50 60 70 | |  | | --- | | 80 90 00 | |  | 10 20 30 40 50 60 70 | |  | | --- | | 30 20 10 | |
|  |  | ===> |  |  |
| 00 90 80 70 60 50 40 | |  | | --- | | 30 20 10 | |  | 00 90 80 70 60 50 40 | |  | | --- | | 80 90 00 | |

Разделение происходит в случайно выбранной точке вдоль длины генома, только если успешно пройден тест вероятности. Она обычно устанавливается весьма высоко, что отражает происходящее в природе.  
Мутация, в сравнении, происходит редко, поэтому вероятность ее возникновения устанавливается низкой, как правило, менее 5%. По очереди тестируется каждый ген в хромосоме, чтобы узнать, разрешено ли ему мутировать, и если да, то он меняется на случайное число, например:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 | |  | | --- | | 70 | | 80 | 90 | ===> | 10 | 20 | 30 | 40 | 50 | 60 | |  | | --- | | 22 | | 80 | 90 |

# **Классы алгоритма**

## *Класс сравнения хромосом*

//класс сравнения хромосом

public sealed class ChromosomeComparer : IComparer

{

//сравнение

public int Compare(object x, object y)

{

if (!(x is Chromosome) || !(y is Chromosome))

throw new ArgumentException("Not of type Chromosome");

//сравнение коэффициентов приспособленности

if (((Chromosome)x).ChromosomeFitness > ((Chromosome)y).ChromosomeFitness)

return 1;

else if (((Chromosome)x).ChromosomeFitness == ((Chromosome)y).ChromosomeFitness)

return 0;

else

return -1;

}

}

## *Класс хромосом*

public sealed class Chromosome

{

private static Random rand = new Random(); //случайное число

public double[] ChromosomeGenes { set; get; } //массив генов

public int ChromosomeLength { set; get; } //кол-во генов

public double ChromosomeFitness { set; get; } //коэффициент приспособленности хромосомы

public static double ChromosomeMutationRate; //процент мутации

//конструктор

public Chromosome(int length, bool createGenes)

{

ChromosomeLength = length;

ChromosomeGenes = new double[length];

//создание случайных генов

if (createGenes)

{

for (int i = 0; i < ChromosomeLength; i++)

ChromosomeGenes[i] = rand.NextDouble() \* 10;

}

}

//кроссовер

public void Crossover(ref Chromosome Chromosome2, out Chromosome child1, out Chromosome child2)

{

int position = (int)(rand.NextDouble() \* 10 \* (double)ChromosomeLength);

child1 = new Chromosome(ChromosomeLength, false);

child2 = new Chromosome(ChromosomeLength, false);

//перекрещивание генов и создание потомков с новым геномом

for (int i = 0; i < ChromosomeLength; i++)

{

if (i < position)

{

child1.ChromosomeGenes[i] = ChromosomeGenes[i];

child2.ChromosomeGenes[i] = Chromosome2.ChromosomeGenes[i];

}

else

{

child1.ChromosomeGenes[i] = Chromosome2.ChromosomeGenes[i];

child2.ChromosomeGenes[i] = ChromosomeGenes[i];

}

}

}

//мутация

public void Mutate()

{

for (int position = 0; position < ChromosomeLength; position++)

{

if (rand.NextDouble() < ChromosomeMutationRate)

ChromosomeGenes[position] = (ChromosomeGenes[position] + rand.NextDouble() \* 10) / 2.0;

}

}

//извлечение генов

public void ExtractChromosomeValues(ref double[] values)

{

for (int i = 0; i < ChromosomeLength; i++)

values[i] = ChromosomeGenes[i];

}

}

## *Класс алгоритма*

public class GeneticAlgorithm

{

public bool SwitchedOn { set; get; } //включение-выключение

private static Random rand = new Random(); //случайное число

public double MutationRate; //процент мутации

public double CrossoverRate; //процент кроссовера

public int ChromosomeLength; //кол-во генов

public int PopulationSize; //размер популяции

public int GenerationSize; //кол-во поколений

public double TotalFitness; //текущая приспособленность

public bool Elitism; //элитизм

private ArrayList CurrentGenerationList; //массив текущего поколения

private ArrayList NextGenerationList; //массив следующего поколения

private ArrayList FitnessList; //массив приспособленности

static private GAFunction getFitness; //делегат функции

public GAFunction FitnessFunction //атрибут доступа к функции

{

get { return getFitness; }

set { getFitness = value; }

}

//список зарегистрированных видов

private ArrayList listeners = new ArrayList();

//конструктор

public GeneticAlgorithm()

{

Elitism = false;

SwitchedOn = false;

MutationRate = 0.05;

CrossoverRate = 0.8;

PopulationSize = 100;

GenerationSize = 2000;

ChromosomeLength = 2;

}

//работа алгоритма

public void LaunchGA()

{

FitnessList = new ArrayList();

CurrentGenerationList = new ArrayList(GenerationSize);

NextGenerationList = new ArrayList(GenerationSize);

Chromosome.ChromosomeMutationRate = MutationRate;

//создание популяции

for (int i = 0; i < PopulationSize; i++)

{

Chromosome g = new Chromosome(ChromosomeLength, true);

CurrentGenerationList.Add(g);

}

RankPopulation();

//проход по всем поколениям

for (int i = 0; i < GenerationSize; i++)

{

if (i % 100 == 0 || (i + 1) == GenerationSize)

{

UpdateObservers();

}

CreateNextGeneration();

RankPopulation();

}

}

//выборка особей

private int RouletteSelection()

{

//случайный коэффициент приспособленности

double randomFitness = rand.NextDouble() \* TotalFitness;

//двоичный поиск для выборки

int idx = -1;

int mid;

int first = 0;

int last = PopulationSize - 1;

mid = (last - first) / 2;

while (idx == -1 && first <= last)

{

if (randomFitness < (double)FitnessList[mid])

{ last = mid; }

else if (randomFitness > (double)FitnessList[mid])

{ first = mid; }

mid = (first + last) / 2;

if ((last - first) == 1) idx = last;

}

return idx;

}

//классификация популяции

private void RankPopulation()

{

TotalFitness = 0;

//текущая приспособленность

for (int i = 0; i < PopulationSize; i++)

{

Chromosome g = ((Chromosome)CurrentGenerationList[i]);

g.ChromosomeFitness = FitnessFunction(g.ChromosomeGenes);

TotalFitness += g.ChromosomeFitness;

}

//сортировка текущего поколения

CurrentGenerationList.Sort(new ChromosomeComparer());

double fitness = 0.0;

FitnessList.Clear();

//вычисление приспособленности для каждой особи

for (int i = 0; i < PopulationSize; i++)

{

fitness += ((Chromosome)CurrentGenerationList[i]).ChromosomeFitness;

FitnessList.Add((double)fitness);

}

}

//создание следующего поколения

private void CreateNextGeneration()

{

NextGenerationList.Clear();

Chromosome g = null;

if (Elitism)

g = (Chromosome)CurrentGenerationList[PopulationSize - 1];

//выборка, кроссовер или мутация, добавление потомков в следующее поколение

for (int i = 0; i < PopulationSize; i += 2)

{

int pidx1 = RouletteSelection();

int pidx2 = RouletteSelection();

Chromosome parent1, parent2, child1, child2;

parent1 = ((Chromosome)CurrentGenerationList[pidx1]);

parent2 = ((Chromosome)CurrentGenerationList[pidx2]);

if (rand.NextDouble() < CrossoverRate)

{ parent1.Crossover(ref parent2, out child1, out child2); }

else

{

child1 = parent1;

child2 = parent2;

}

child1.Mutate();

child2.Mutate();

NextGenerationList.Add(child1);

NextGenerationList.Add(child2);

}

if (Elitism && g != null) NextGenerationList[0] = g;

CurrentGenerationList.Clear();

for (int i = 0; i < PopulationSize; i++)

CurrentGenerationList.Add(NextGenerationList[i]);

}

//получение лучших генов

public void GetBestValues(out double[] values, out double fitness)

{

Chromosome g = ((Chromosome)CurrentGenerationList[PopulationSize - 1]);

values = new double[g.ChromosomeLength];

g.ExtractChromosomeValues(ref values);

fitness = (double)g.ChromosomeFitness;

}

//обновление видов

public void UpdateObservers()

{

foreach (IObserver ob in listeners)

{

ob.UpdateState();

}

}

//регистрация видов

public void Register(IObserver o)

{

listeners.Add(o);

}

//дерегистрация видов

public void Deregister(IObserver o)

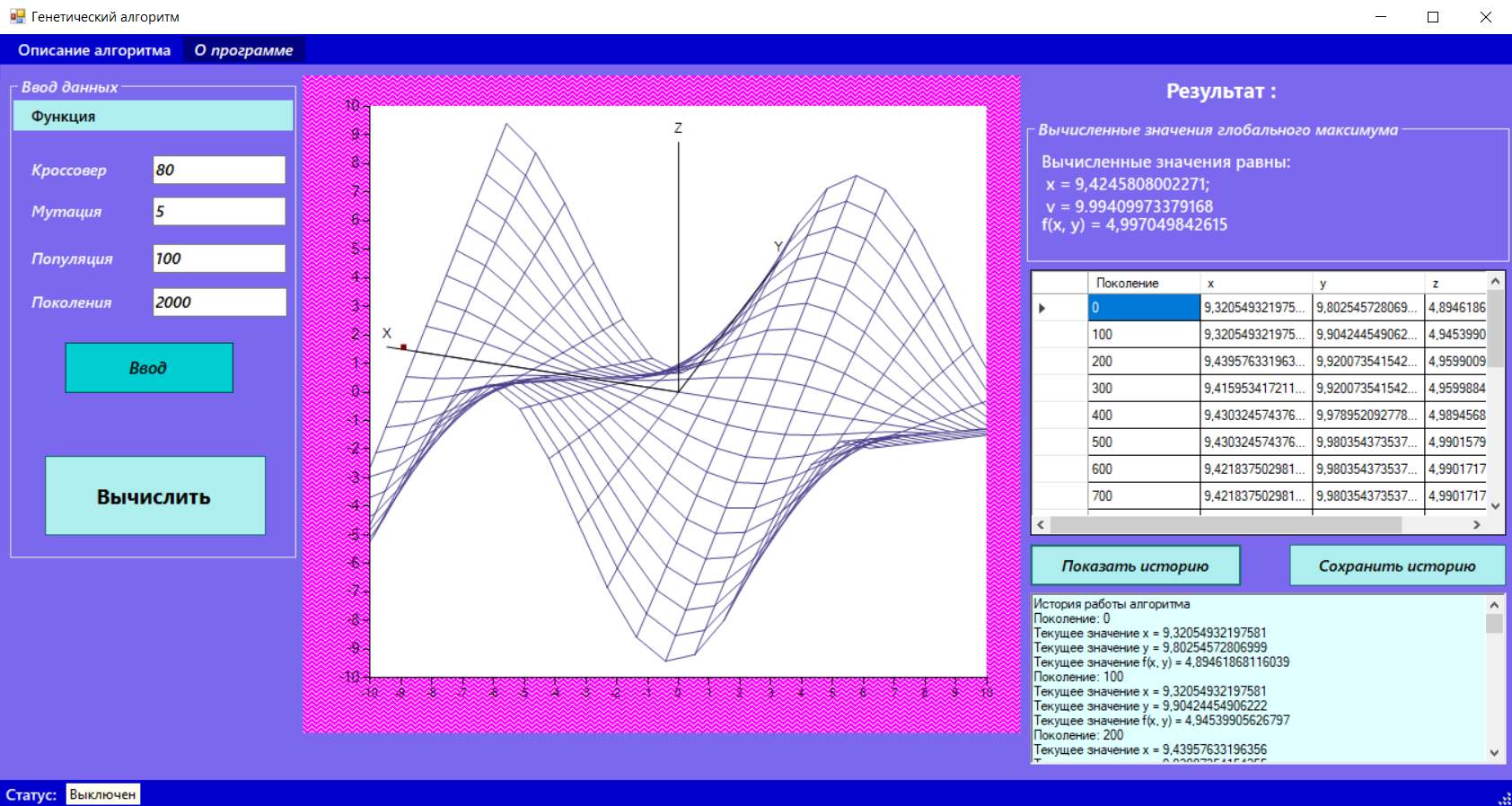
{

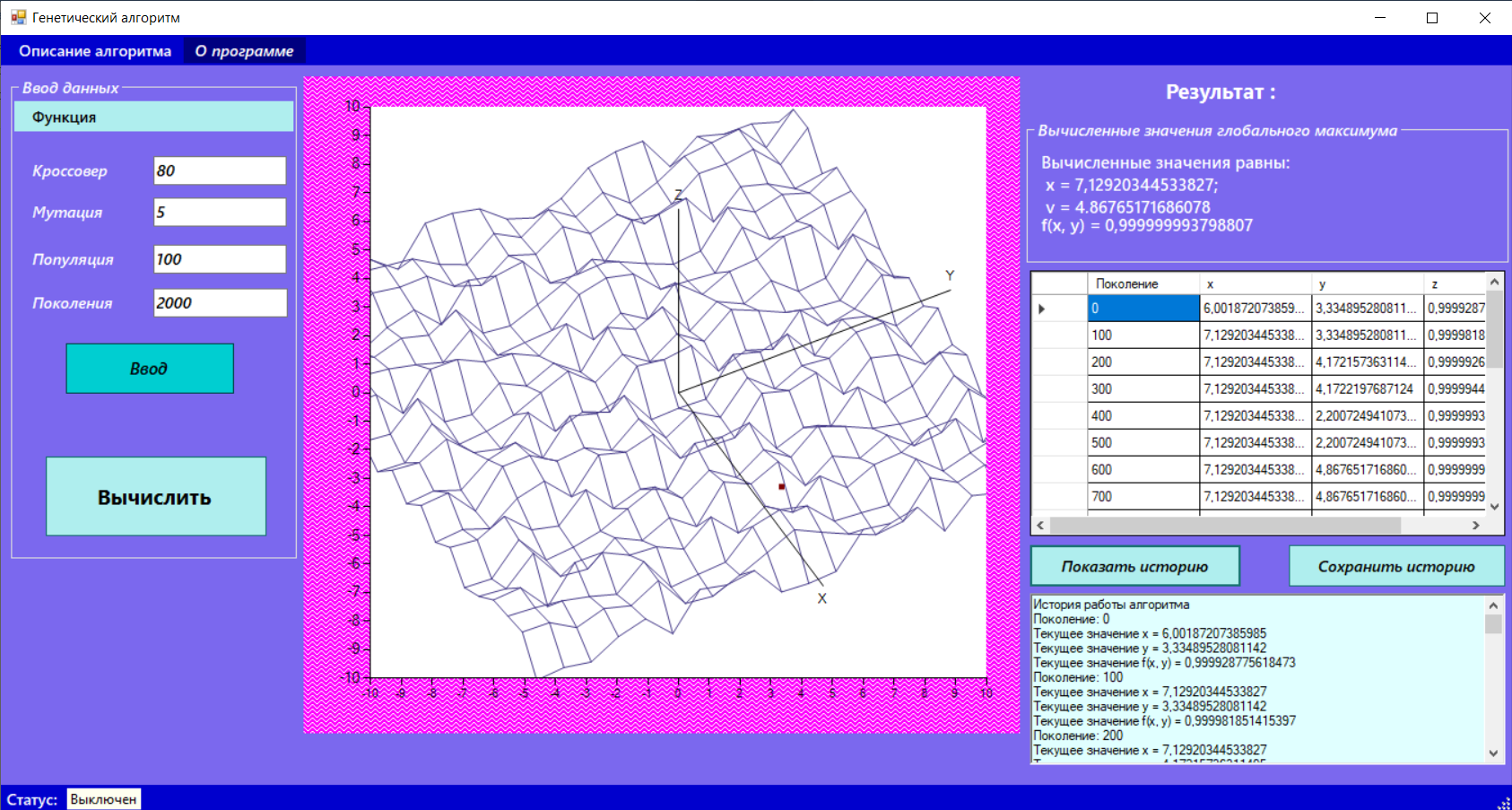
listeners.Remove(o);

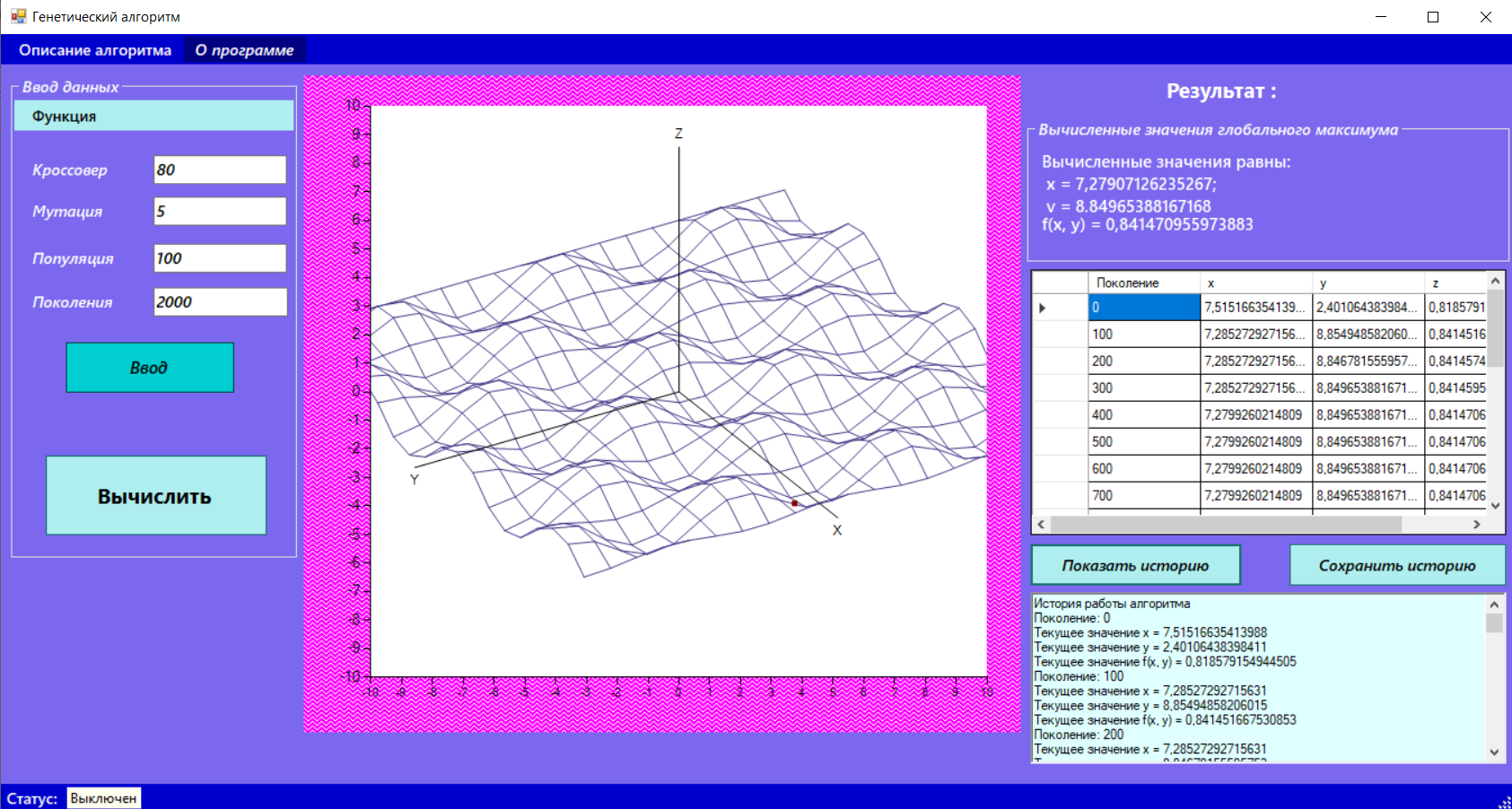
}

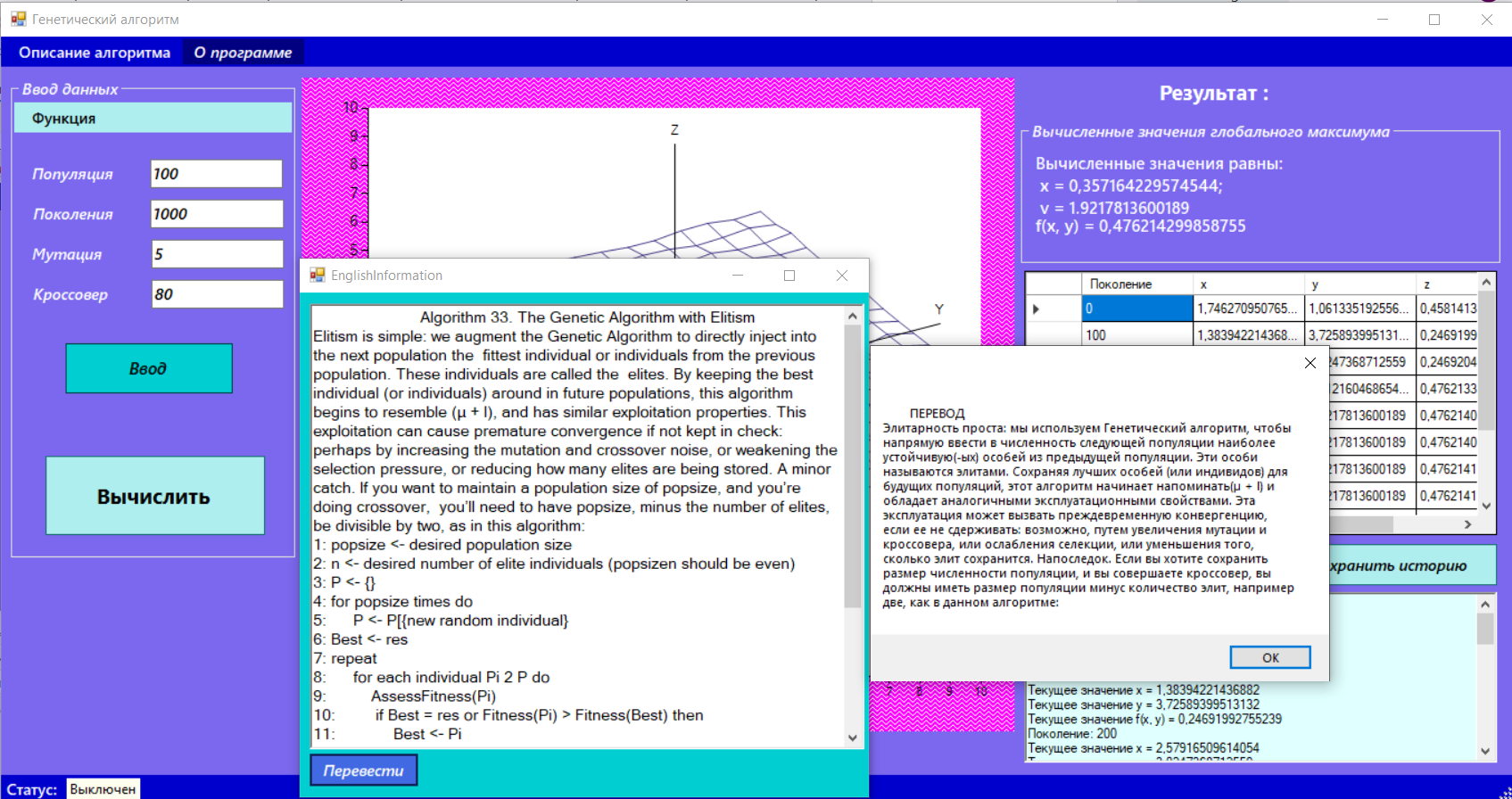
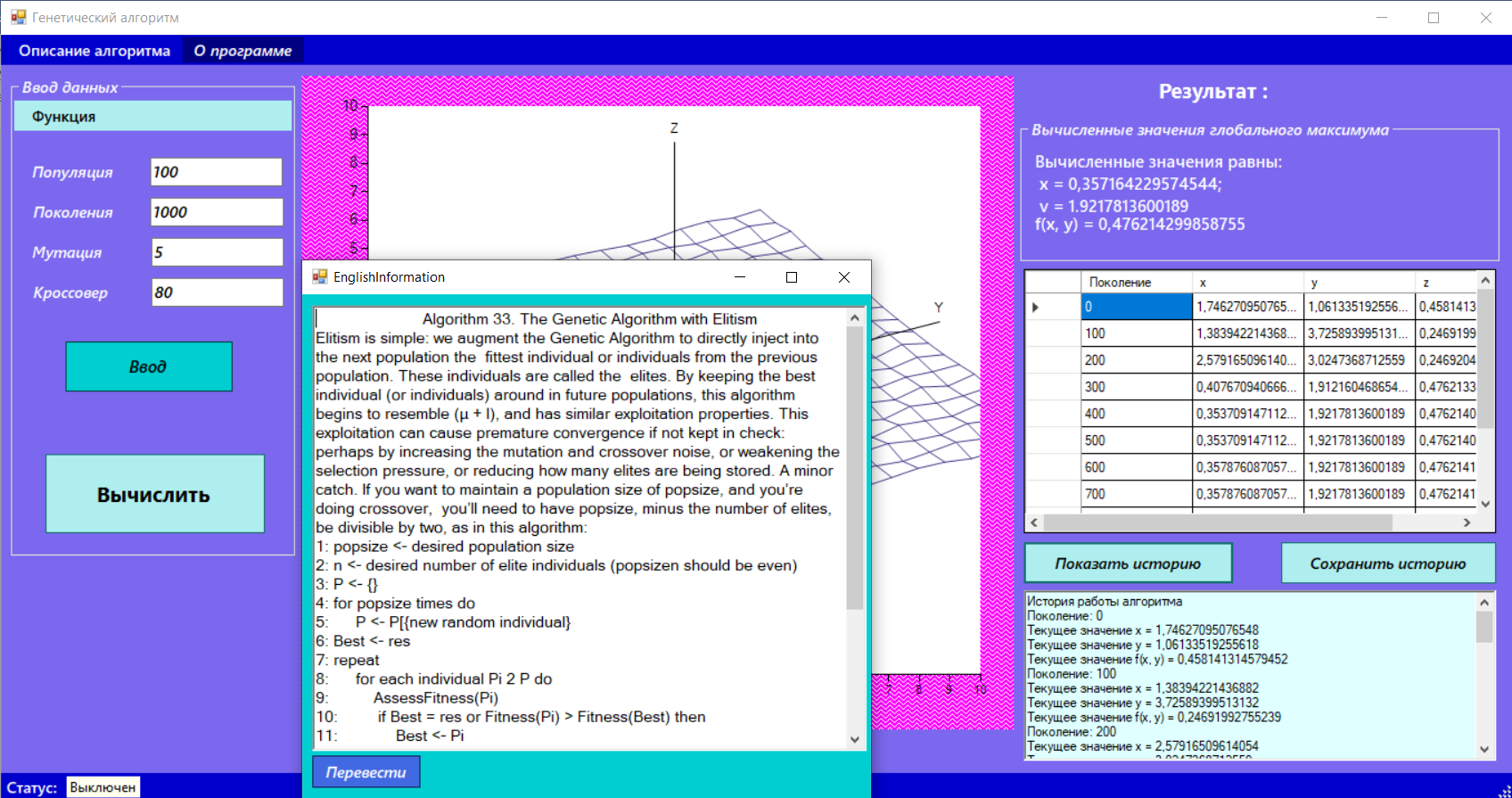
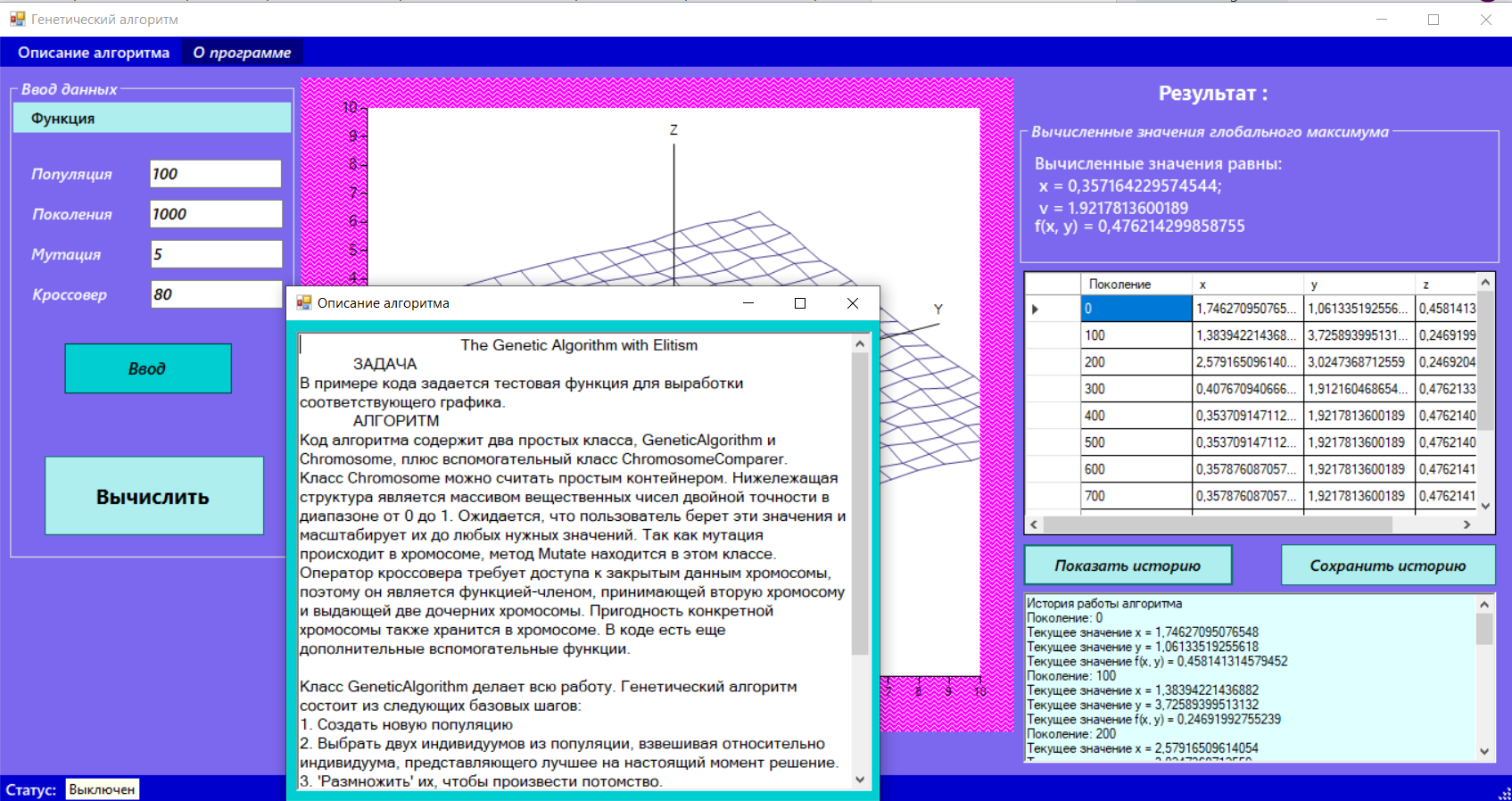
}

# **Результат работы**









# **Вывод**

С помощью изученного материала по объектно-ориентированному программированию и инструментальным средствам прикладных программных систем был реализован Алгоритм 33 - Генетический алгоритм с элитарностью с применением паттернов проектирования и визуализацией одиним из метаэвристических алгоритмов в терминах объектно-ориентированного программирования на языке C#.

В качестве интегрированной среды разработки использовалась Microsoft Visual Studio 2019 с использованием Windows Forms. В качестве оборудования для разработки курсового проекта использовался Ноутбук HP Pavilion Power 15-cb011ur.

Основу структуры программы составили паттерны MVC, Abstract Factory и Singleton.

Был изучен и реализован Генетический алгоритм. Реализованы следующие этапы генетического алгоритма:

1. Задана целевая функцию (приспособленности) для особей популяции
2. Создана начальная популяция

* (Начало цикла)

1. Размножение (скрещивание)
2. Мутирование
3. Вычислено значение целевой функции для всех особей
4. Сформировано новое поколение (селекция)
5. Если выполняются условия остановки, то (конец цикла), иначе (начало цикла).

# **Литература**

«Essentials of Metaheuristics» Sean Luke (Department of Computer Science George Mason University)

<https://ru.wikipedia.org/wiki/Генетический_алгоритм>

http://www.cyberguru.ru/algorithms/algorithms-theory/algorithms-simple-genetic-algorithm.html?showall=&start=1