**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**Факультет прикладной математики – процессов управления**

**отчет**

**по лабораторной работе**

**по дисциплине «Алгоритмы и структуры данных»**

**на тему «Исследование генетического алгоритма. Изучение различных кодировок генотипа»**

**Вариант 3 (модификация в части «новая популяция»)**

|  |  |  |
| --- | --- | --- |
| Студент гр. 23.Б16 |  | Барафанов А.А. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2024 г.**

**Оглавление**

[**1.** **Цель работы** 3](#_Toc182514431)

[**2.** **Задача (формализация)** 3](#_Toc182514432)

[**3.** **Теоретическая часть** 4](#_Toc182514433)

[**4.** **Описание алгоритма** 6](#_Toc182514435)

[**5.** **Описание программы** 9](#_Toc182514436)

[**5.1** **Описание классов** 9](#_Toc182514437)

[**5.2 Описание функций** 10](#_Toc182514438)

[**6.** **Рекомендации пользователю** 11](#_Toc182514439)

[**7.** **Рекомендации программисту** 12](#_Toc182514450)

[**8.** **Контрольный пример** 13](#_Toc182514457)

[**10.** **Вывод** 23](#_Toc182514458)

[**11.** **Листинг программы с детальными комментариями** 23](#_Toc182514460)

# **Цель работы**

Исследование основных способов кодирования генотипа хромосом в генетическом алгоритме и проверка эффективности различных модификаций ГА.

# **Задача (формализация)**

Создание генетического алгоритма, использующего вещественное кодирование генов хромосом и поддерживающего различные модификации попадания особей в новую популяцию, а также создание интерфейса для ввода необходимых параметров пользователем и сравнение эффективности скорости схождения и точности результатов всех модификаций на примере исследования функции двух переменных на глобальный минимум.

# **Теоретическая часть**

# *Генетический алгоритм (ГА)* — это эволюционный метод оптимизации и поиска, который основывается на принципах естественного отбора и генетической репродукции. Он используется для решения сложных задач, где традиционные методы оптимизации могут оказаться неэффективными. ГА моделирует процесс естественного отбора, в ходе которого лучшие особи (или решения) имеют больше шансов на выживание и размножение, передавая свои характеристики следующему поколению.

*Основные компоненты* генетического алгоритма включают:

1. *Популяция:* Это множество возможных решений, представленных в виде хромосом (обычно в виде строк битов или других кодировок).
2. *Функция приспособленности:* Каждой хромосоме присваивается оценка, отражающая, насколько хорошо она решает поставленную задачу. Эта функция позволяет сравнивать различные решения и определять, какие из них "лучшие".
3. *Операторы селекции:* На основе значений функции приспособленности выбираются хромосомы для создания нового поколения. Существует несколько методов селекции, таких как рулетка, турнирная селекция и другие.
4. *Операторы скрещивания (кроссовер):* Эти операторы комбинируют две или более хромосом для создания новых особей, которые наследуют характеристики родителей. Это позволяет генерировать разнообразие в популяции.
5. *Операторы мутации:* Они вносят случайные изменения в хромосомы, что помогает поддерживать генетическое разнообразие и предотвращает преждевременное сходимость к локальным минимумам.

*Основные типы кодирования генов в ГА:*

*Вещественное кодирование* (или непрерывное кодирование) представляет гены в виде вещественных чисел. Каждый ген в хромосоме соответствует определенному параметру решения и может принимать значения в заданном диапазоне. Это кодирование удобно использовать для задач, где параметры решений могут быть представлены непрерывными величинами, такими как оптимизация функций, где значения могут варьироваться в диапазоне. Вещественное кодирование позволяет более точно представлять решения и может привести к более эффективному поиску в пространстве решений, особенно в задачах с непрерывными переменными.

*Бинарное кодирование* представляет гены в виде последовательностей битов (0 и 1). Каждая хромосома состоит из ряда битовых строк, где каждая строка соответствует определенному параметру решения. Бинарное кодирование часто используется в задачах, где решения могут быть выражены в дискретной форме, например, в задачах комбинаторной оптимизации. Этот метод прост в реализации и позволяет легко применять операторы скрещивания и мутации, но может требовать дополнительных усилий для преобразования бинарных значений в действительные параметры, особенно в случаях, когда необходимо работать с непрерывными величинами.

*Для текущей задачи было выбрано вещественное кодирование*, как более простой и точный метод в реализации (задача поиска минимума действительной функции предполагает непрерывность изменения аргументов, а потому естественно выбрать в качестве генов координаты точек, их перевод в двоичную систему счисления для бинарного кодирования, во-первых – избыточен, а во-вторых, требует дополнительной оптимизации для большей точности схождения.

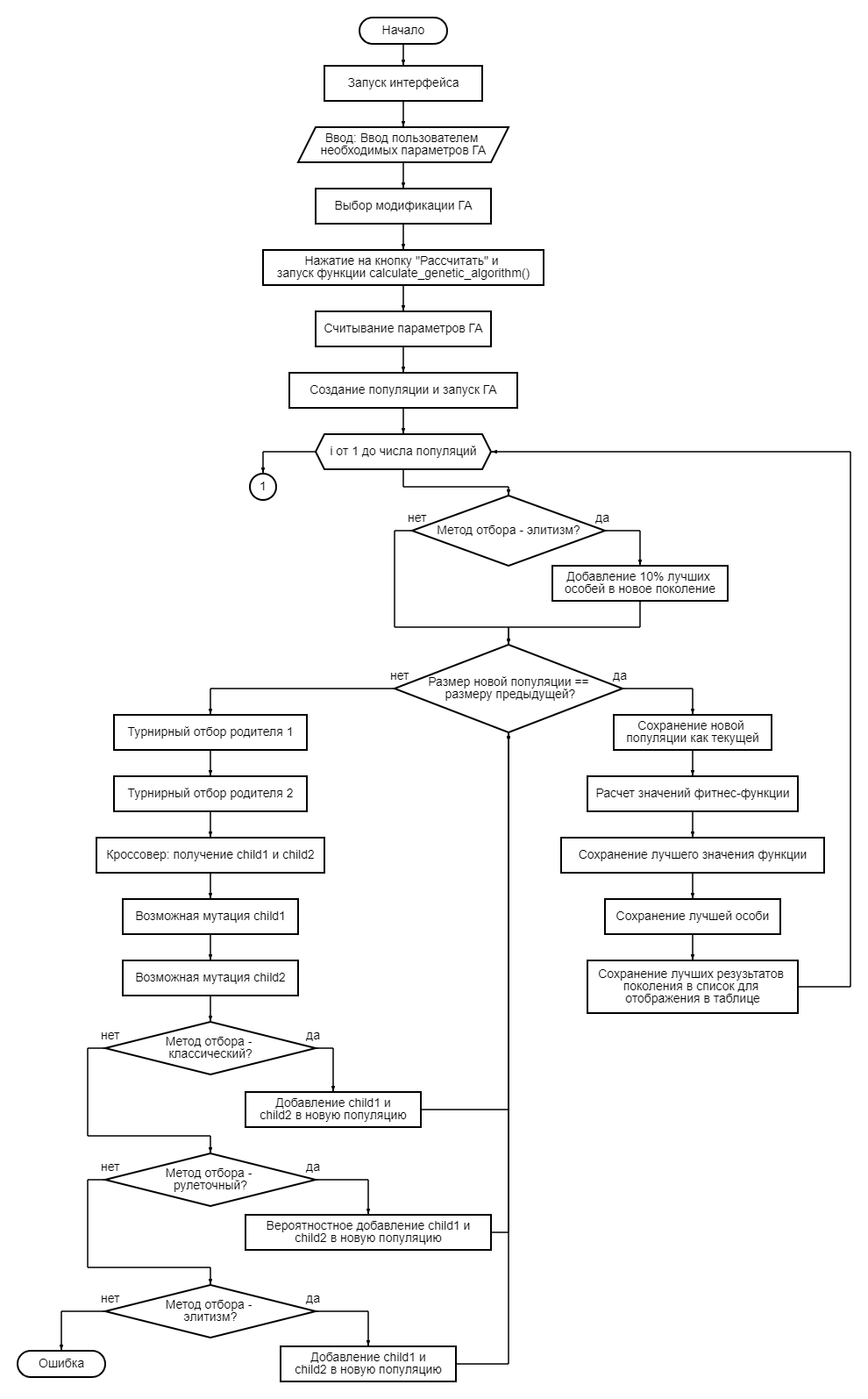
*Элитизм* — это стратегия в генетических алгоритмах, при которой несколько лучших особей из текущего поколения сохраняются и передаются в следующее поколение без изменений. Эта модификация гарантирует, что лучшие решения (или хромосомы) не потеряются в процессе селекции и размножения, что может существенно повысить эффективность алгоритма и ускорить сходимость к оптимальному решению. Количество элитных особей, которые сохраняются, может варьироваться в зависимости от конкретной реализации алгоритма.

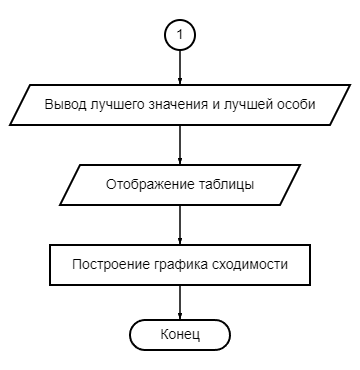
*Рулеточная модификация* (или рулеточная селекция) — это метод селекции, который использует вероятностный подход для выбора особей для участия в создании нового поколения. Каждой хромосоме присваивается вероятность выбора, пропорциональная ее значению функции приспособленности. В процессе "рулетки" создается круг, на котором размер сектора каждой хромосомы соответствует ее приспособленности. Затем случайным образом выбирается точка на круге, и особь, соответствующая этому сектору, выбирается для размножения. Этот метод позволяет сохранить разнообразие в популяции и избегать преждевременной сходимости к локальным минимумам, обеспечивая при этом возможность выбора более приспособленных особей.

# **Описание алгоритма**

1. *Импорт библиотек:* Подключаются необходимые библиотеки для работы с массивами, создания графического интерфейса, генерации случайных чисел и построения графиков.
2. *Инициализация класса Genetic:* Создается класс Genetic, который будет содержать все методы и данные, необходимые для работы генетического алгоритма.
3. *Инициализация популяции:* Создается начальная популяция особей.
4. *Расчет пригодности:* Для каждой особи в популяции вычисляется значение функции пригодности, которую мы хотим минимизировать.
5. *Отбор родителей:* Используется турнирный отбор для выбора родителей, которые будут участвовать в создании потомков.
6. *Кроссовер:* Создаются потомки путем кроссовера (скрещивания) генов родителей.
7. *Мутация:* Применяется мутация к потомкам, чтобы ввести разнообразие в популяцию.
8. *Формирование новой популяции:* Потомки добавляются в новую популяцию. В этом разделе как раз и проводятся исследования эффективности различных модификаций.
9. *Вывод результатов:* После завершения всех поколений, выводится лучшая особь, её пригодность и данные о поколениях.
10. *Создание графического интерфейса:* Создается графический интерфейс с настройками параметров алгоритма, кнопками для запуска расчета и отображением результатов, включая график сходимости алгоритма.

*Блок-схема, иллюстрирующая шаги алгоритма показана на рисунке 4.1.*



**

*Рисунок 4.1. Блок-схема программы.*

# **Описание программы**

## **Описание классов**

В программе используется 1 класс. В таблице 5.1.1 представлено его описание.

*Таблица 5.1.1. Описание классов*

|  |  |  |
| --- | --- | --- |
| Имя класса | Наследование | Описание класса |
| Genetic | \_\_ | Класс, содержащий методы для расчета генетического алгоритма (одного поколения) |

Описание переменных класса Genetic представлено в таблице 5.1.2.

*Таблица 5.1.2. Описание переменных класса Genetic*

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Права доступа | Имя | Описание |
| Ссылка на функцию | public | fitness\_function | Функция пригодности |
| int | public | num\_generations | Количество поколений |
| int | public | population\_size | Размер популяции |
| float | public | mut\_probab | Вероятность мутации |
| float | public | low | Минимальное значение гена |
| float | public | hight | Максимальное значение гена |
| float | public | sigma | Стандартное отклонение мутации |
| str | public | selection\_method | Метод отбора |
| list | public | population | Популяция |
| list | public | fitness\_values | Значения фитнес-функции текущего поколения |

В таблице 5.1.3 представлено описание методов класса Genetic.

*Таблица 5.1.3. Описание методов класса Person*

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Модификатор  доступа/ Возвращаемый тип | Входные данные | Описание |
| \_\_init\_\_ | public | self, фитнес-функция, число поколений, размер поколения, вероятность мутации, min ген, max ген, метод отбора | Конструктор класса |
| init\_population | public/list | self | Инициализация популяции |
| calculate\_fitness | public/list | self | Расчет значений пригодности для всех хромосом в популяции |
| tournament\_selection | public/Genetic | self, размерность турнира | Турнирный отбор |
| crossover | public/Genetic | self, родитель1, родитель2 | Скрещивание (кроссовер) |
| gaussian\_mutation | public/int | self, хромосома | Мутация с использованием гауссовского распределения |
| run | public/Genetic, list, int | self | Запуск цикла генетического алгоритма |

## **5.2 Описание функций**

В программе используется 4 функции, выполняющие вспомогательные процедуры программы. В таблице 5.2.1 представлено их описание.

*Таблица 5.2.1. Описание функций*

|  |  |  |
| --- | --- | --- |
| Имя функции | Входные данные | Описание функции |
| fitness\_function | Координаты x и y | Расчет функции пригодности |
| inserted | Поле ввода, вводимое значение | Функция для вставки значения в поле ввода |
| display\_table | Список из лучшего значения функции и лучшей особи каждого поколения | Функция для отображения данных в таблице |
| plot\_convergence | Словарь с данными поколений, метод отбора | Функция для построения графика сходимости |
| calculate\_genetic\_algorithm | \_\_ | Функция для запуска генетического алгоритма (с заданным количеством поколений) |

# **Рекомендации пользователю**

* Функция: Программа оптимизирует функцию -12y + 4x^2 + 4y^2 - 4xy.
* Программа запускается с помощью интерпретатора Python. Убедитесь, что у вас установлены все необходимые библиотеки: numpy, tkinter, random, matplotlib. Запустите программу, выполнив команду python имя\_файла.py.
* Установите все необходимые параметры ГА, введя значения в соответствующие поля ввода. Также выберите метод отбора особей для скрещивания:
* Classic: Классический метод, где все потомки добавляются в популяцию.
* Roulette: Метод рулетки, где вероятность выбора потомка зависит от его приспособленности.
* Elitism: Элитизм, где часть лучших особей переходят в следующее поколение без изменений.
* После настройки параметров нажмите кнопку "Рассчитать". На графике сходимости можно будет оценить, насколько быстро алгоритм находит решение.
* Для получения более точного решения увеличьте количество поколений.
* Для ускорения работы алгоритма уменьшите количество хромосом или количество поколений.
* Попробуйте разные методы отбора, чтобы найти наиболее эффективный для вашей задачи.

1. **Рекомендации программисту**

Для работы с программой должен быть установлен интерпретатор язык Python, а также же интегрированная среда разработки (Visual Studio). Поддерживайте актуальные версии интерпретатора и сред разработки. Установите необходимые библиотеки языка python в случае их отсутствия: numpy, tkinter, random, matplotlib.

Вы также можете настроить параметры генетического алгоритма для оптимизации конкретно под вашу задачу, а именно:

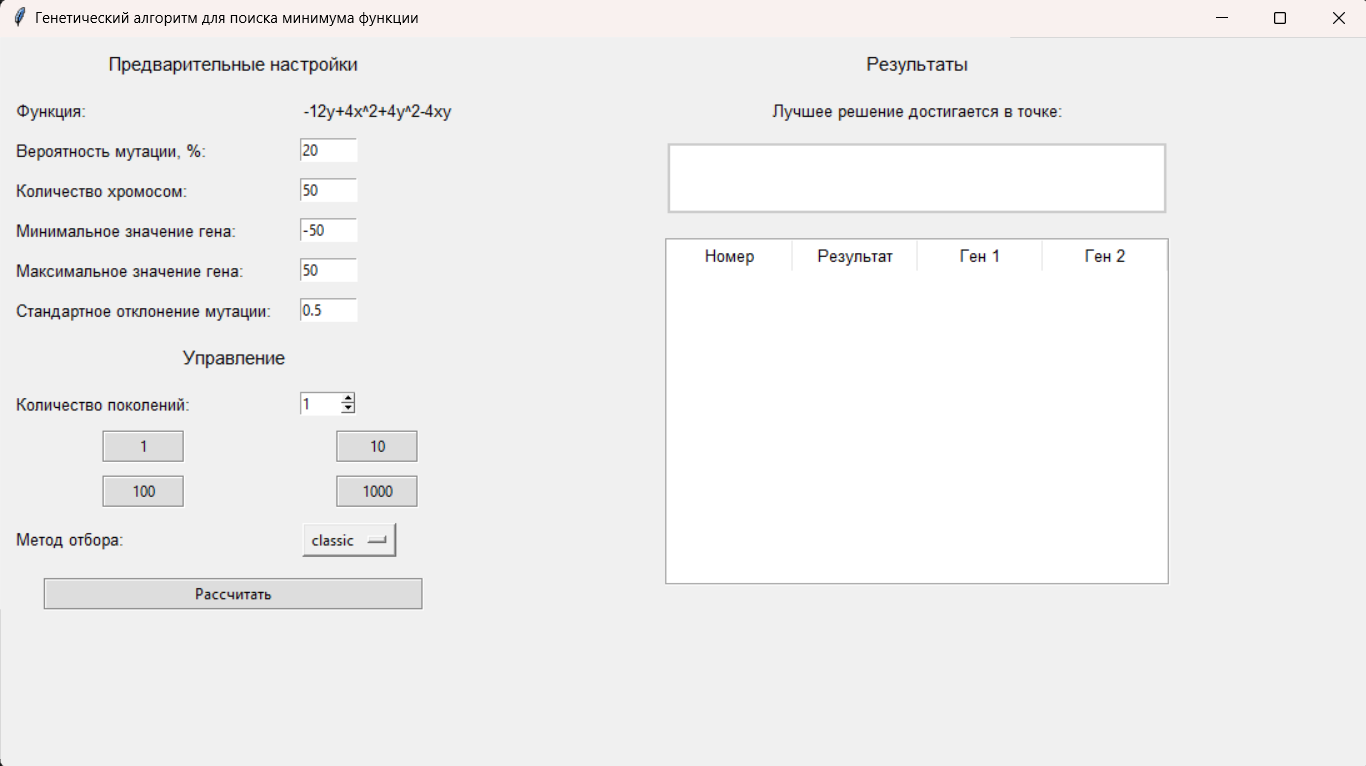
* Вероятность мутации (ProbMut): Рекомендуется выбирать значение в диапазоне от 1% до 10% для стабильной работы алгоритма. Слишком высокая вероятность мутации может привести к потере хороших решений, а слишком низкая — к замедлению сходимости.
* Количество хромосом (numChrom): Рекомендуется выбирать значение в диапазоне от 20 до 100. Чем больше популяция, тем больше разнообразия, но и тем медленнее работает алгоритм.
* Стандартное отклонение мутации (sigmaEntry): Рекомендуется выбирать значение в диапазоне от 0.1 до 1.0. Слишком большое значение может привести к сильным мутациям, а слишком маленькое — к незначительным изменениям.
* Количество поколений (cntIt): Рекомендуется выбирать значение в диапазоне от 100 до 1000. Чем больше поколений, тем больше вероятность нахождения оптимального решения, но и тем дольше работает алгоритм.

Конечно же, вы можете изменить функцию, для которой вы хотите найти минимум, однако помните, что генетический алгоритм не гарантирует нахождение глобального минимума, а только приближенное решение.

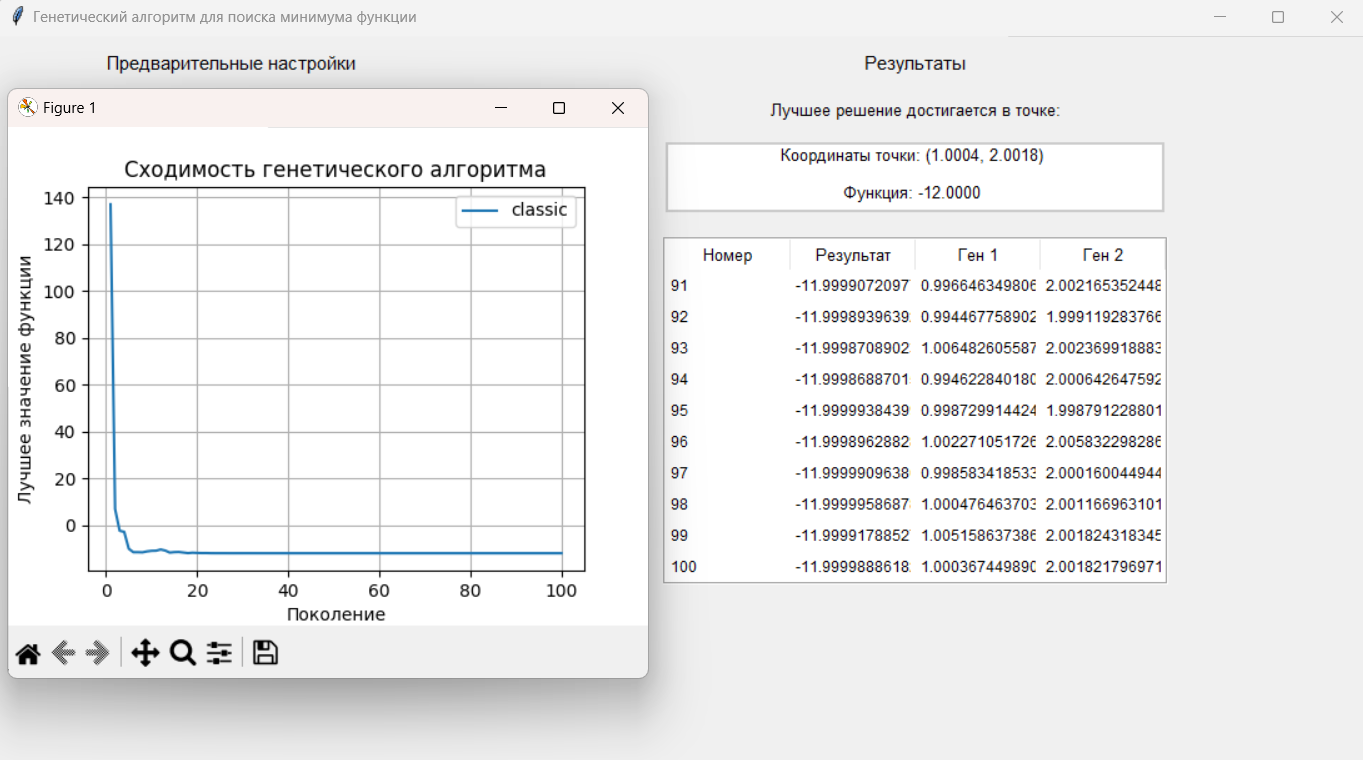
1. **Контрольный пример**

В этом разделе продемонстрированы три примера запуска генетического алгоритма с начальными параметрами, но разными модификациями в отборе особей в новое поколение.

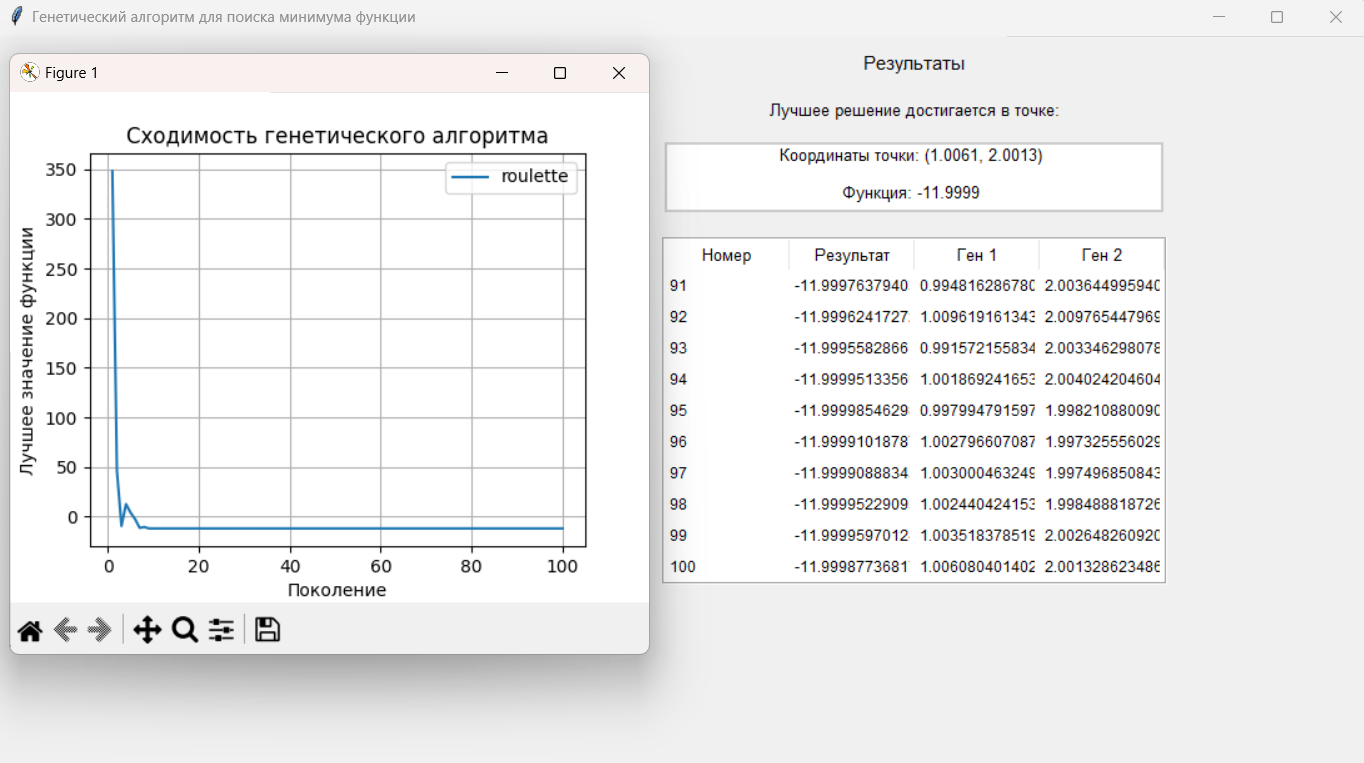
Тонкости взаимодействия с программой указаны в рекомендациях пользователю, здесь же приведены результаты работы программы.



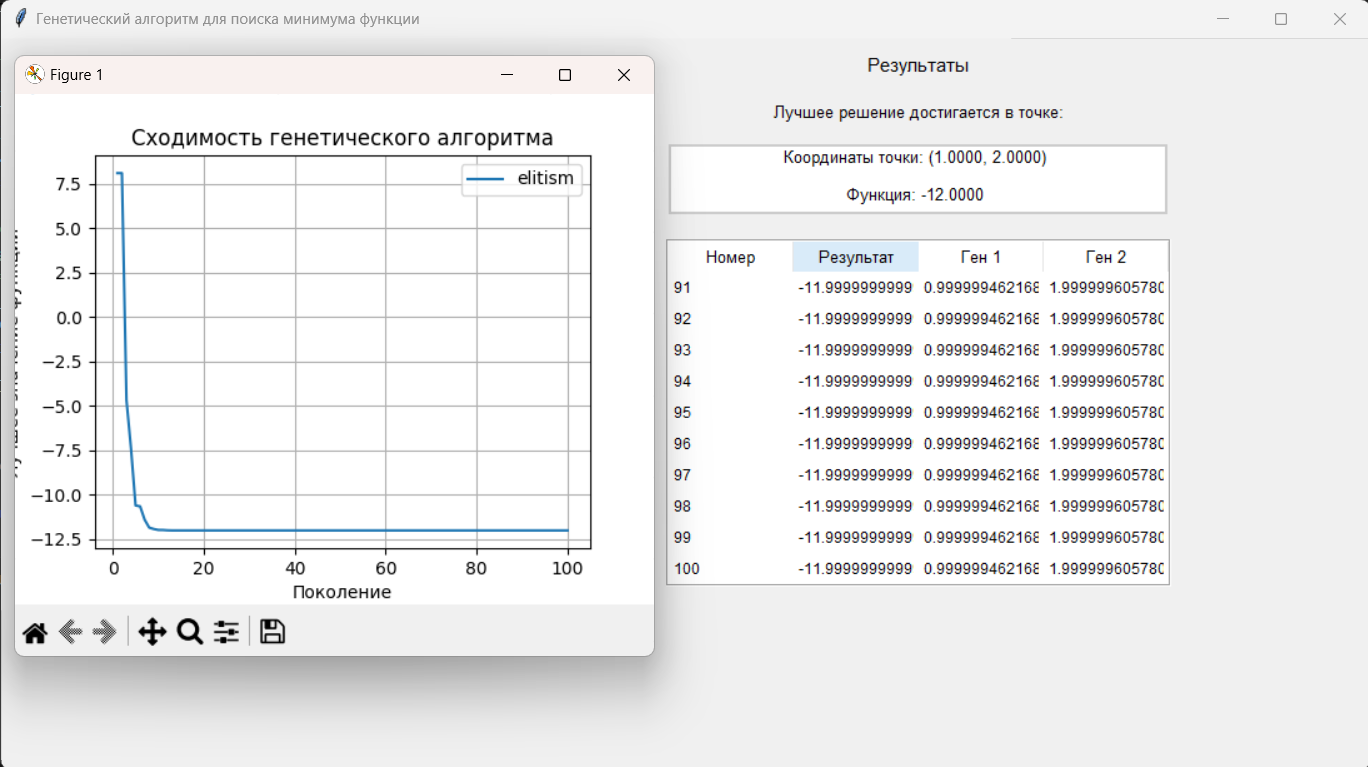
*Рисунок 8.1 Интерфейс программы*



*Рисунок 8.2 Результат тестового запуска модификации classic*

**

*Рисунок 8.3 Результат тестового запуска модификации roulette*

**

*Рисунок 8.4 Результат тестового запуска модификации elitism*

1. **Анализ результатов**

Для оценки эффективности алгоритма будем использовать два параметра:

* Скорость схождения алгоритма
* Точность результатов поиска

Для подбора оптимальных параметров алгоритма для заданной функции проведем три серии экспериментов с изменениями в:

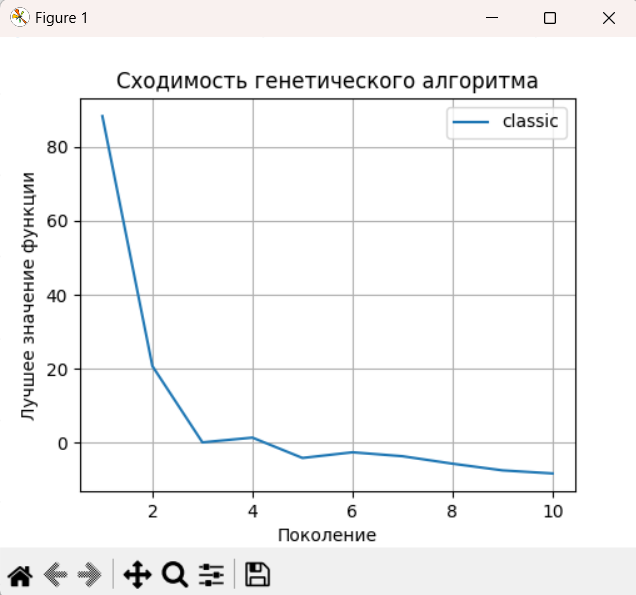
* Модификации отбора в новое поколение
* Вероятности мутации
* Количестве хромосом

Для чистоты эксперимента проведем 5 запусков в каждой серии, чтобы избавиться от фактора неудачной начальной популяции.

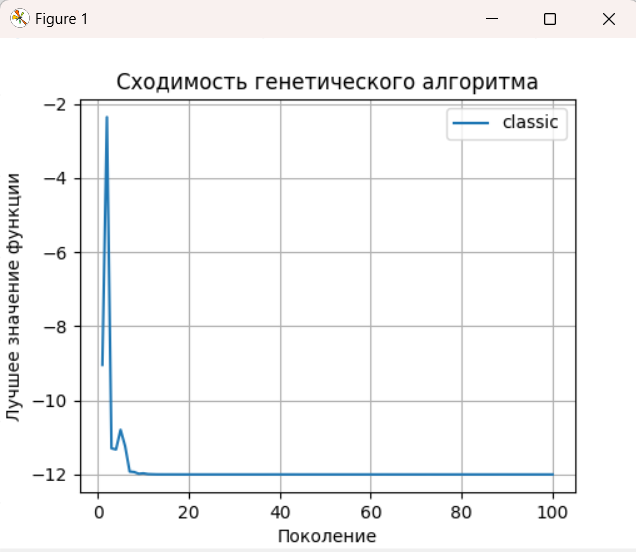
1. Модификации в части отбора особей в новую популяцию.

Начальные параметры: вероятность мутации – 20%, кол-во хромосом – 50, кол-во поколений – 10/100 (Скорость схождения исследуем на первых 10-ти поколениях, точность результата на 100)

*Классический метод:*

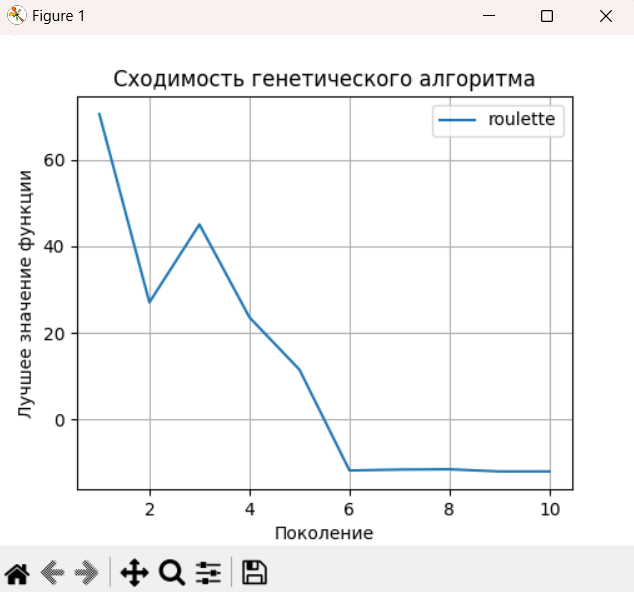


*Рисунок 9.1. «Динамика схождения classic»*

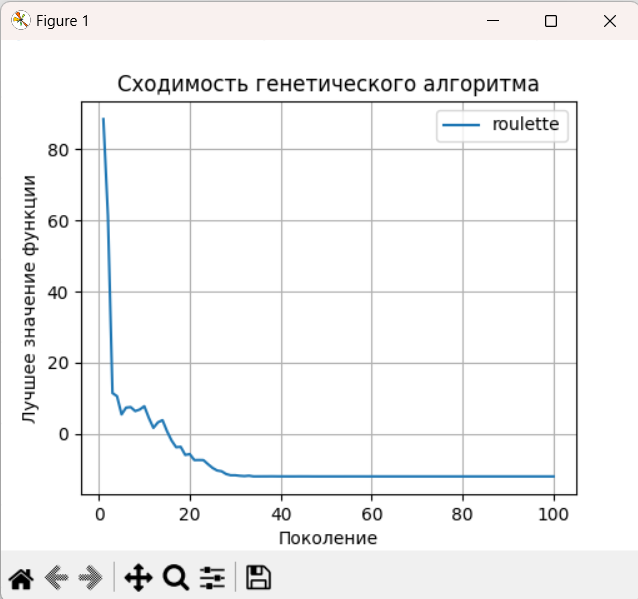


*Рисунок 9.2. «Точность результата classic»*

*Рулеточная модификация:*



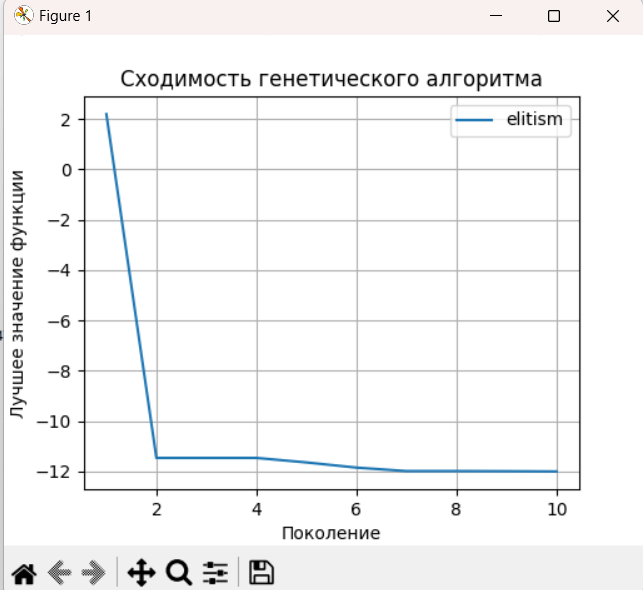
*Рисунок 9.3. «Динамика схождения roulette»*



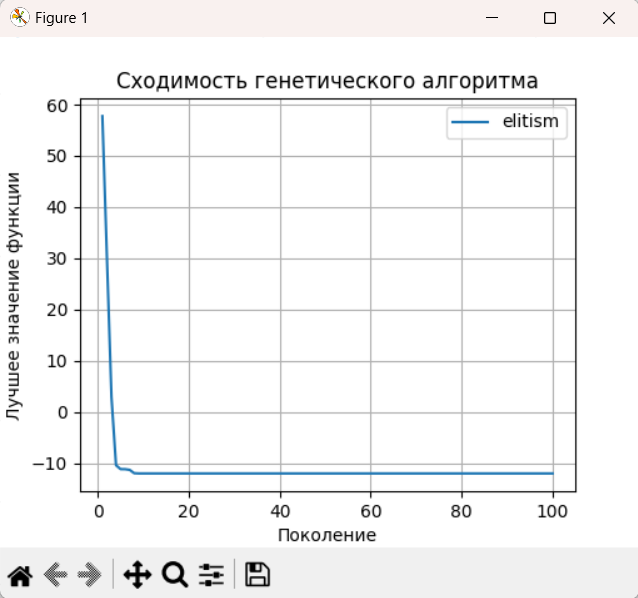


*Рисунок 9.4. «Точность результата roulette»*

*Элитизм:*

**

*Рисунок 9.5. «Динамика схождения elitism»*

**

**

*Рисунок 9.6. «Точность результата elitism»*

Итоги экспериментов:

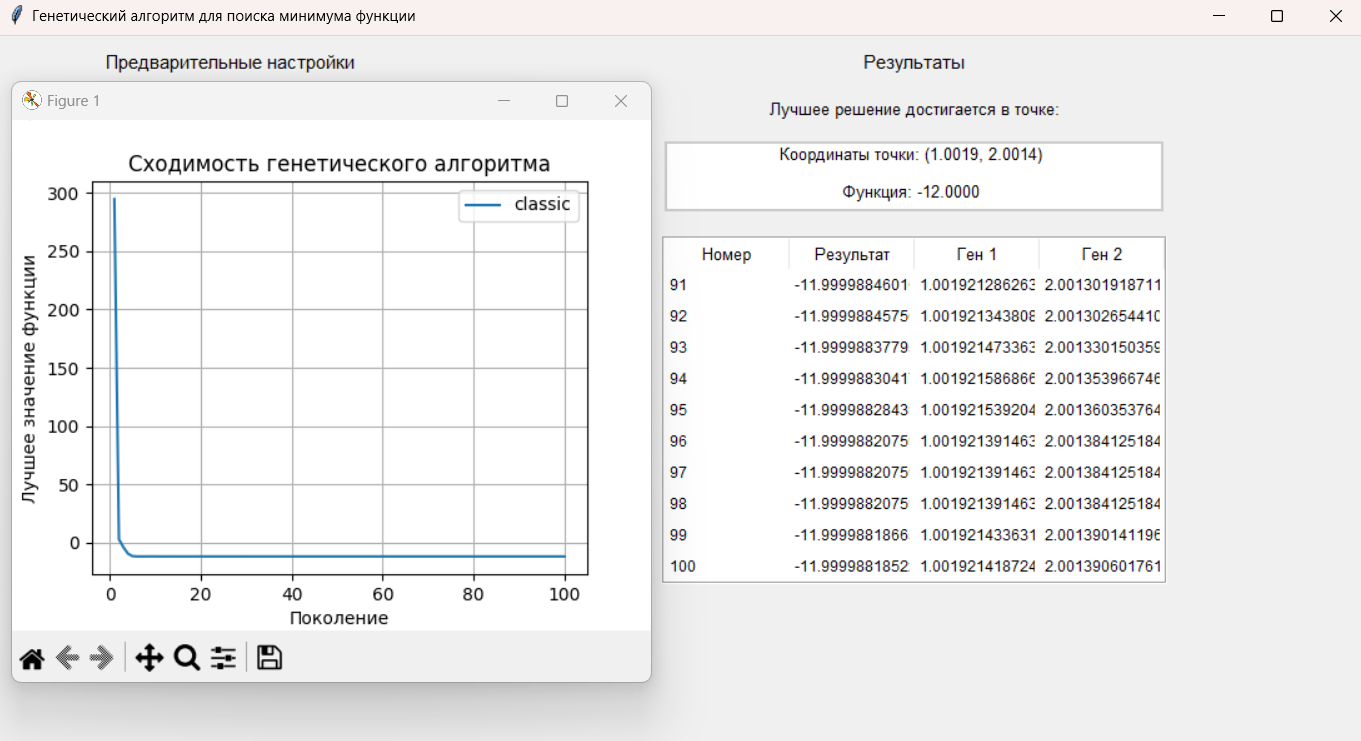
Самую быструю скорость схождения показал элитизм. Самый точный результат также получился в этой модификации.

Как мы видим, мы получили результаты, которые подтверждаются теорией: 1) из-за введения вероятностей в рулеточной модификации метод сходится медленнее, однако позволяет увеличить пространство поиска. Сохранение же лучших особей в элитизме наоборот сужает поиск, но позволяет наиболее быстро прийти к максимально приближенному к истине ответу.

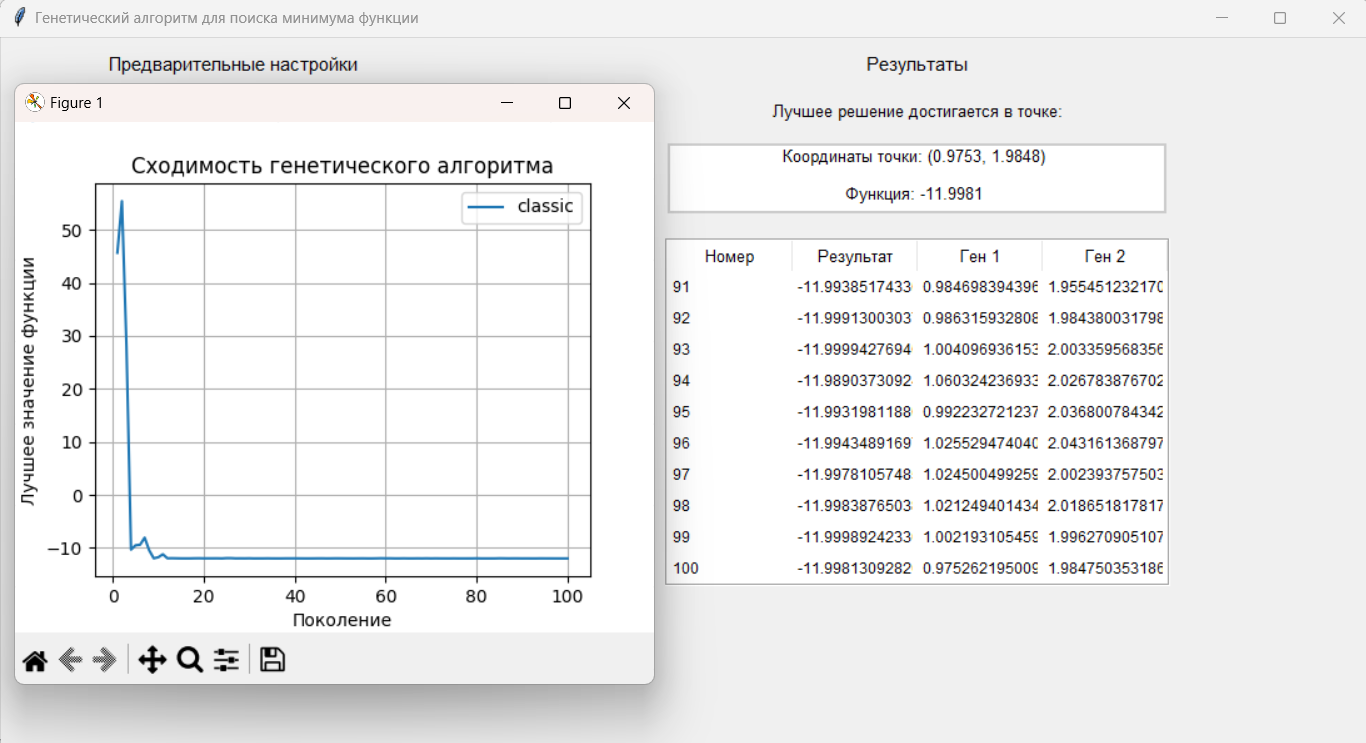
1. Вероятность мутации:

Оставим начальные параметры теми же, а в качестве модификации выберем «Классический метод». Будем экспериментировать со значениями 10%, 50%, 100%.

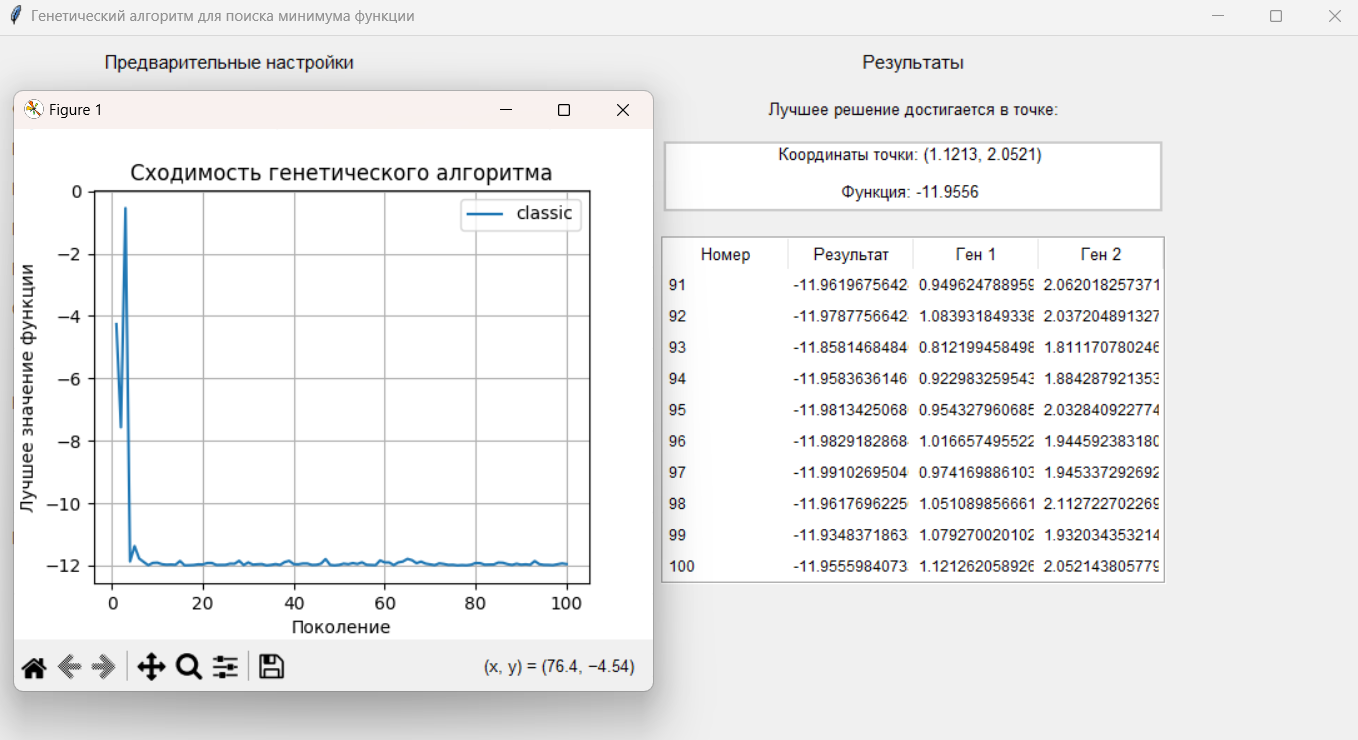
Все результаты будем смотреть на 100 поколениях.



*Рисунок 9.7. «Результаты для мутации 10%*

**

*Рисунок 9.8. «Результаты для мутации 50%*

**

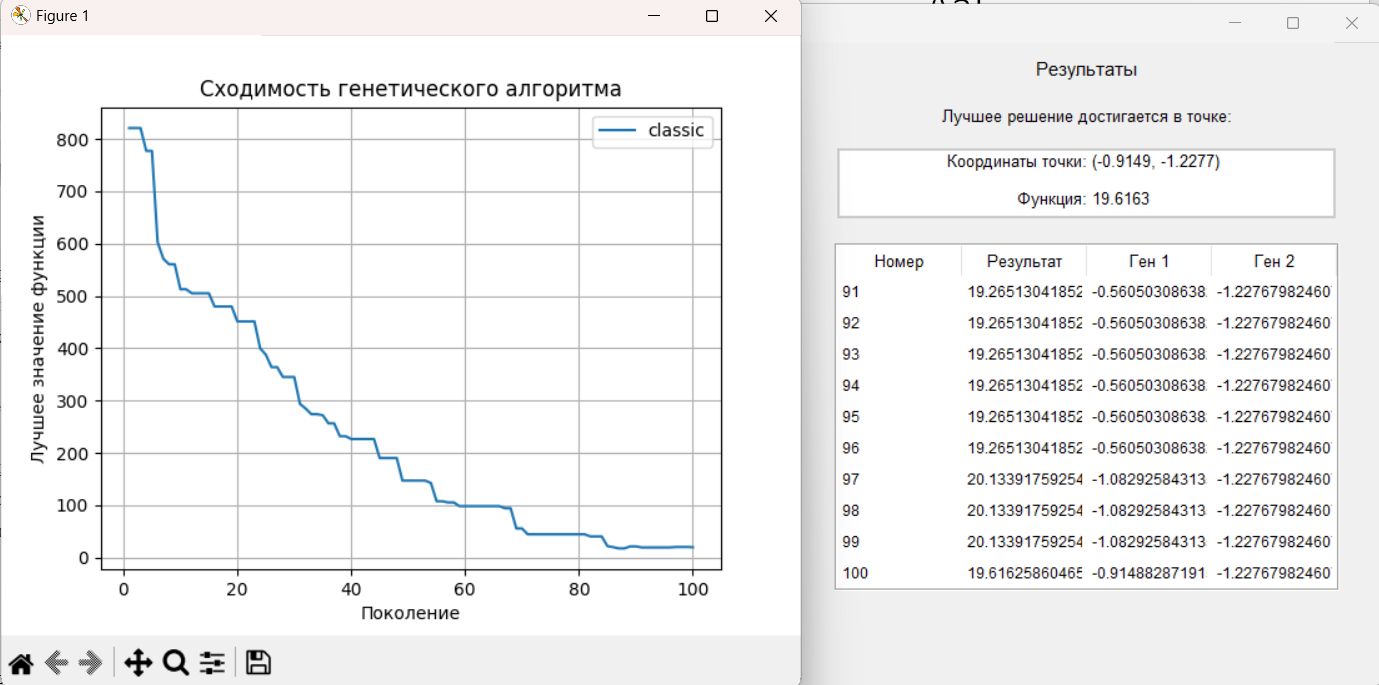
*Рисунок 9.9 «Результаты для мутации 100%*

Итоги экспериментов:

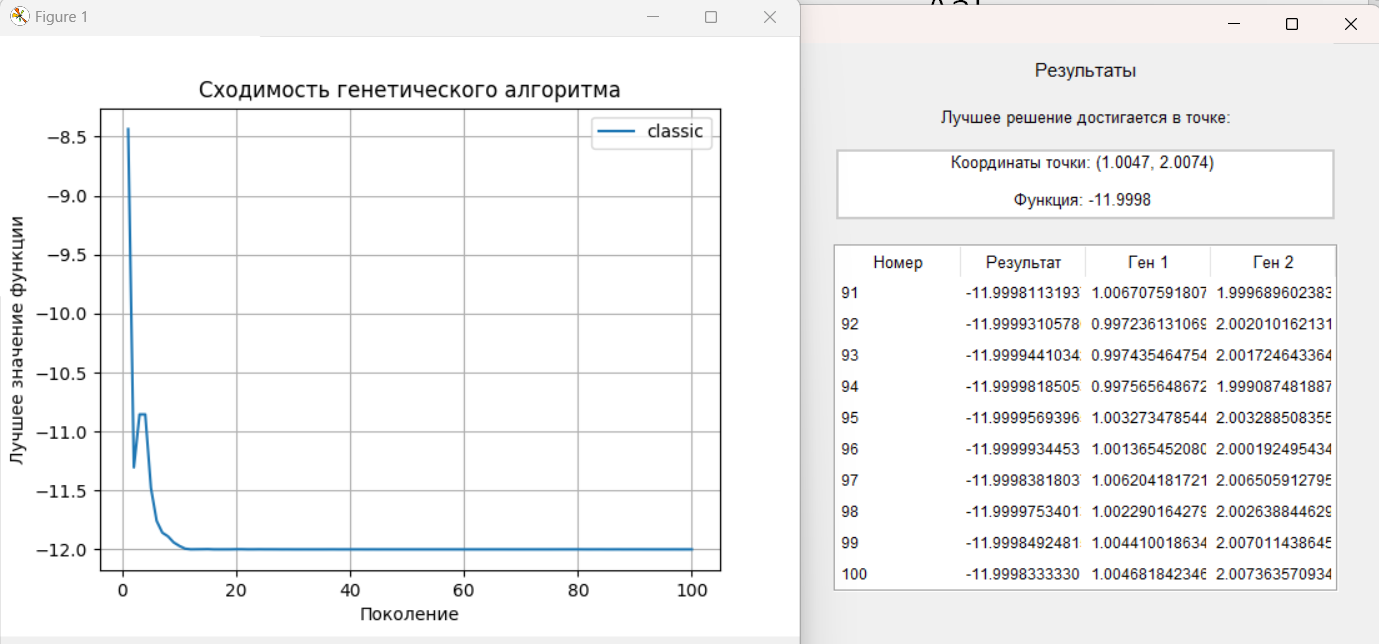
Видны колебания на графике мутации 100%: это говорит о том, что в значениях присутствует слишком большая доля случайности, которая не позволяет наиболее точно рассчитать результат. Самые точные данные получены, когда вероятность мутации – 10%, что также подтверждает теорию.

1. Количество хромосом:

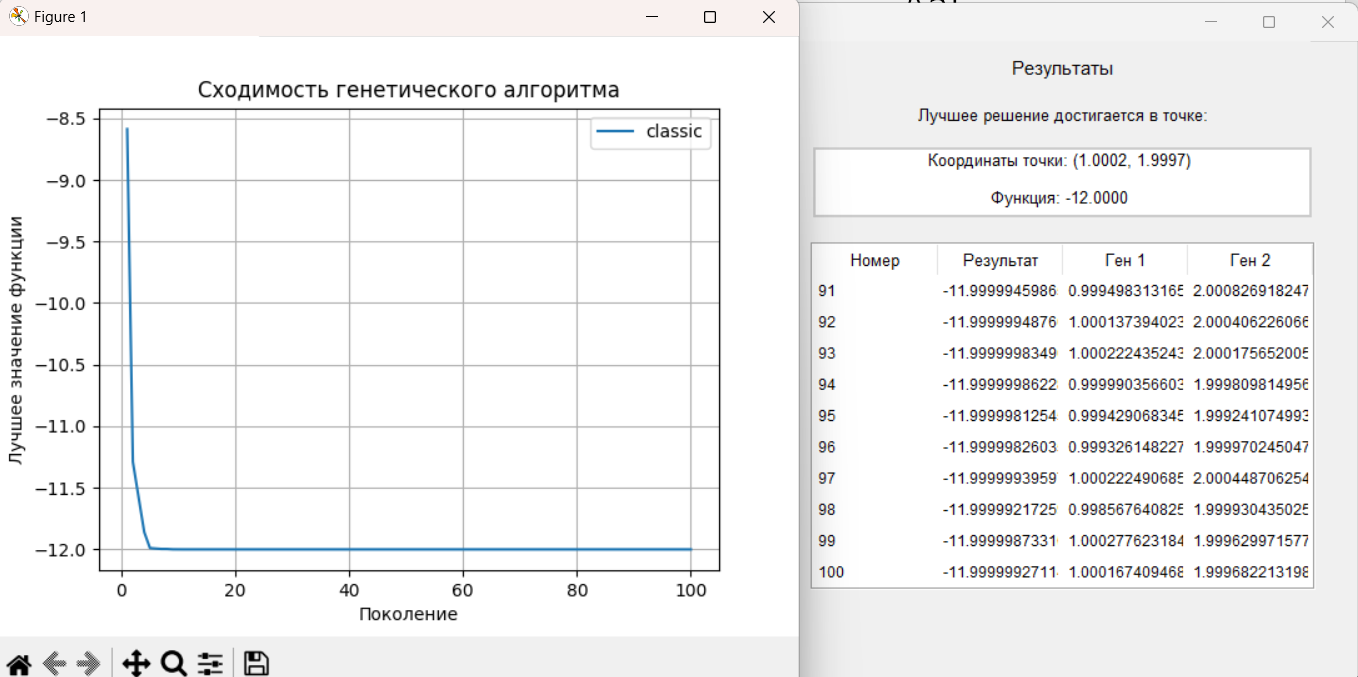
Будем проводить эксперименты с тремя группами ГА: 2 хромосомы в популяции, 100 хромосом и 1000 хромосом.



*Рисунок 9.10. «Результаты для 2-х хромосом*

**

*Рисунок 9.11. «Результаты для 100 хромосом*

**

*Рисунок 9.12. «Результаты для 1000-чи хромосом*

Итоги экспериментов:

Скорость схождения для 2-х хромосом явно недостаточна, так как результат за сто поколений результат был не достигнут. Эксперимент с 1000 хромосом показал лучшую стабильность, точность результата и скорость схождения, но занял больше времени. Однако эта издержка оказалась не критичной для заданной тестовой функции.

1. **Вывод**

В ходе работы был создан генетический алгоритм с тремя модификациями в части нового поколения: классической, рулеточной и элитичной, а также получены базовые знания в создании пользовательского интерфейса. Были проанализированы результаты исследований, которые подтвердили теорию. В результате были определены некоторые оптимальные параметры ГА для заданной тестовой функции.

# **Листинг программы с детальными комментариями**

**import** numpy **as** np # Библиотека для работы с массивами и математическими функциями

**import** tkinter **as** tk # Библиотека для создания графического интерфейса

**from** tkinter **import** ttk # Расширение tkinter для создания более современных виджетов

**import** random # Библиотека для генерации случайных чисел

**import** matplotlib.pyplot **as** plt # Библиотека для построения графиков

# Класс Genetic реализует генетический алгоритм

**class** Genetic:

**def** \_\_init\_\_(self, fitness\_function, num\_generations, population\_size, mut\_probab, low, hight, sigma, selection\_method):

self.fitness\_function = fitness\_function # Функция пригодности

self.num\_generations = num\_generations # Количество поколений

self.population\_size = population\_size # Размер популяции

self.mut\_probab = mut\_probab # Вероятность мутации

self.low = low # Минимальное значение гена

self.hight = hight # Максимальное значение гена

self.sigma = sigma # Стандартное отклонение мутации

self.selection\_method = selection\_method # Метод отбора

self.population = self.init\_population() # Инициализация популяции

self.fitness\_values = self.calculate\_fitness() # Расчет значений пригодности

# Инициализация популяции

**def** init\_population(self):

population = []

**for** \_ **in** range(self.population\_size):

x = random.uniform(self.low, self.hight)

y = random.uniform(self.low, self.hight)

population.append([x, y])

**return** population

# Расчет значений пригодности для всех хромосом в популяции

**def** calculate\_fitness(self):

fitness\_values = []

**for** chromo **in** self.population:

x, y = chromo

fitness = self.fitness\_function(x, y)

fitness\_values.append(fitness)

**return** fitness\_values

# Турнирный отбор

**def** tournament\_selection(self, tournament\_size):

tournament = random.sample(range(len(self.population)), tournament\_size)

winner\_index = min(tournament, key=**lambda** i: self.fitness\_values[i])

**return** self.population[winner\_index]

# Скрещивание (кроссовер)

**def** crossover(self, parent1, parent2):

alpha = random.random()

child1 = [alpha \* x1 + (1 - alpha) \* x2 **for** x1, x2 **in** zip(parent1, parent2)]

child2 = [(1 - alpha) \* x1 + alpha \* x2 **for** x1, x2 **in** zip(parent1, parent2)]

**return** child1, child2

# Мутация с использованием гауссовского распределения

**def** gaussian\_mutation(self, chromo):

mutated\_chromo = []

**for** gene **in** chromo:

**if** random.random() < self.mut\_probab:

mutation = np.random.normal(0, self.sigma)

mutated\_gene = gene + mutation

**else**:

mutated\_gene = gene

mutated\_chromo.append(mutated\_gene)

**return** mutated\_chromo

# Основной цикл генетического алгоритма

**def** run(self):

generations\_data = []

**for** generation **in** range(self.num\_generations):

new\_population = []

**if** self.selection\_method == "elitism":

elite\_size = int(self.population\_size \* 0.1)

elite\_indices = np.argsort(self.fitness\_values)[:elite\_size]

**for** index **in** elite\_indices:

new\_population.append(self.population[index])

**while** len(new\_population) < self.population\_size:

parent1 = self.tournament\_selection(tournament\_size=2)

parent2 = self.tournament\_selection(tournament\_size=2)

child1, child2 = self.crossover(parent1, parent2)

child1 = self.gaussian\_mutation(child1)

child2 = self.gaussian\_mutation(child2)

**if** self.selection\_method == "classic":

new\_population.append(child1)

new\_population.append(child2)

**elif** self.selection\_method == "roulette":

fitness1 = self.fitness\_function(\*child1)

fitness2 = self.fitness\_function(\*child2)

total\_fitness = fitness1 + fitness2

**if** total\_fitness > 0:

prob1 = fitness1 / total\_fitness

prob2 = fitness2 / total\_fitness

**else**:

prob1 = prob2 = 0.5

**if** random.random() < prob1:

new\_population.append(child1)

**if** random.random() < prob2:

new\_population.append(child2)

**elif** self.selection\_method == "elitism":

new\_population.append(child1)

new\_population.append(child2)

self.population = new\_population[:self.population\_size]

self.fitness\_values = self.calculate\_fitness()

best\_fitness = min(self.fitness\_values)

best\_index = np.argmin(self.fitness\_values)

best\_chromo = self.population[best\_index]

generations\_data.append((generation + 1, best\_fitness, best\_chromo[0], best\_chromo[1]))

**return** best\_chromo, best\_fitness, generations\_data

# Функция пригодности

**def** fitness\_function(x, y):

**return** (-12\*y+4\*x\*\*2+4\*y\*\*2-4\*x\*y)

# Функция для вставки значения в поле ввода

**def** inserted(place, num):

place.delete(0, tk.END)

place.insert(0, str(num))

# Функция для отображения данных в таблице

**def** display\_table(data):

**for** row **in** tree.get\_children():

tree.delete(row)

**for** row\_data **in** data:

tree.insert("", "end", values=row\_data)

# Функция для построения графика сходимости

**def** plot\_convergence(data\_dict, selected\_method):

plt.clf() # Очистка текущего графика

**if** selected\_method **in** data\_dict:

data = data\_dict[selected\_method]

generations = [d[0] **for** d **in** data]

fitness\_values = [d[1] **for** d **in** data]

plt.plot(generations, fitness\_values, label=selected\_method)

plt.xlabel('Поколение')

plt.ylabel('Лучшее значение функции')

plt.title('Сходимость генетического алгоритма')

plt.legend()

plt.grid(True)

plt.draw()

plt.pause(0.01)

# Функция для запуска генетического алгоритма

**def** calculate\_genetic\_algorithm():

# Считывание данных с интерфейса

mut\_probab = float(ProbMut.get()) / 100.0

population\_size = int(numChrom.get())

num\_generations = int(cntIt.get())

low = float(minGen.get())

hight = float(maxGen.get())

sigma = float(sigmaEntry.get())

selection\_method = selectionVar.get()

# Создание популяции

genetic\_algorithm = Genetic(fitness\_function, num\_generations, population\_size, mut\_probab, low, hight, sigma, selection\_method)

best\_chromo, best\_fitness, generations\_data = genetic\_algorithm.run()

# Вывод результата

coord\_text = f"Координаты точки: ({best\_chromo[0]:.4f}, {best\_chromo[1]:.4f})"

fitness\_text = f"Функция: {best\_fitness:.4f}"

canvas2.delete("all")

canvas2.itemconfig(canvas2.create\_text(197, 10, text=coord\_text, font=("Arial", 10)), tags="result\_text\_coord")

canvas2.itemconfig(canvas2.create\_text(197, 40, text=fitness\_text, font=("Arial", 10)), tags="result\_text\_fitness")

display\_table(generations\_data)

# Сохранение данных для построения графика

**if** **not** hasattr(calculate\_genetic\_algorithm, "data\_dict"):

calculate\_genetic\_algorithm.data\_dict = {}

calculate\_genetic\_algorithm.data\_dict[selection\_method] = generations\_data

# Построение графика сходимости

plot\_convergence(calculate\_genetic\_algorithm.data\_dict, selection\_method)

# Создание основного окна

root = tk.Tk()

root.title("Генетический алгоритм для поиска минимума функции")

root.geometry('800x450')

# Создание основных фреймов

left\_frame = tk.Frame(root, width=400, height=450)

left\_frame.grid(row=0, column=0, sticky="nsew")

right\_frame = tk.Frame(root, width=400, height=450)

right\_frame.grid(row=0, column=1, sticky="nsew")

# Настройка сетки

root.grid\_rowconfigure(0, weight=1)

root.grid\_columnconfigure(0, weight=1)

root.grid\_columnconfigure(1, weight=1)

# Левая часть интерфейса

tk.Label(left\_frame, text="Предварительные настройки", font=("Arial", 11)).grid(row=0, column=0, columnspan=2, pady=10)

tk.Label(left\_frame, text="Функция:", font=("Arial", 10)).grid(row=1, column=0, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="-12y+4x^2+4y^2-4xy", font=("Arial", 10)).grid(row=1, column=1, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="Вероятность мутации, %:", font=("Arial", 10)).grid(row=2, column=0, sticky="w", padx=10, pady=5)

ProbMut = tk.Entry(left\_frame, width=7)

ProbMut.insert(0, "20")

ProbMut.grid(row=2, column=1, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="Количество хромосом:", font=("Arial", 10)).grid(row=3, column=0, sticky="w", padx=10, pady=5)

numChrom = tk.Entry(left\_frame, width=7)

numChrom.insert(0, "50")

numChrom.grid(row=3, column=1, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="Минимальное значение гена:", font=("Arial", 10)).grid(row=4, column=0, sticky="w", padx=10, pady=5)

minGen = tk.Entry(left\_frame, width=7)

minGen.insert(0, "-50")

minGen.grid(row=4, column=1, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="Максимальное значение гена:", font=("Arial", 10)).grid(row=5, column=0, sticky="w", padx=10, pady=5)

maxGen = tk.Entry(left\_frame, width=7)

maxGen.insert(0, "50")

maxGen.grid(row=5, column=1, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="Стандартное отклонение мутации:", font=("Arial", 10)).grid(row=6, column=0, sticky="w", padx=10, pady=5)

sigmaEntry = tk.Entry(left\_frame, width=7)

sigmaEntry.insert(0, "0.5")

sigmaEntry.grid(row=6, column=1, sticky="w", padx=10, pady=5)

tk.Label(left\_frame, text="Управление", font=("Arial", 11)).grid(row=7, column=0, columnspan=2, pady=10)

tk.Label(left\_frame, text="Количество поколений:", font=("Arial", 10)).grid(row=8, column=0, sticky="w", padx=10, pady=5)

cntIt = tk.Spinbox(left\_frame, from\_=1, to=5000, width=5)

cntIt.grid(row=8, column=1, sticky="w", padx=10, pady=5)

tk.Button(left\_frame, text="1", width=8, command=**lambda**: inserted(cntIt, 1), bg="#DDDDDD", activebackground="#CCCCCC", relief=tk.GROOVE).grid(row=9, column=0, padx=10, pady=5)

tk.Button(left\_frame, text="10", width=8, command=**lambda**: inserted(cntIt, 10), bg="#DDDDDD", activebackground="#CCCCCC", relief=tk.GROOVE).grid(row=9, column=1, padx=10, pady=5)

tk.Button(left\_frame, text="100", width=8, command=**lambda**: inserted(cntIt, 100), bg="#DDDDDD", activebackground="#CCCCCC", relief=tk.GROOVE).grid(row=10, column=0, padx=10, pady=5)

tk.Button(left\_frame, text="1000", width=8, command=**lambda**: inserted(cntIt, 1000), bg="#DDDDDD", activebackground="#CCCCCC", relief=tk.GROOVE).grid(row=10, column=1, padx=10, pady=5)

selectionVar = tk.StringVar(value="classic")

tk.Label(left\_frame, text="Метод отбора:", font=("Arial", 10)).grid(row=11, column=0, sticky="w", padx=10, pady=5)

selection\_method\_menu = tk.OptionMenu(left\_frame, selectionVar, "classic", "roulette", "elitism")

selection\_method\_menu.grid(row=11, column=1, sticky="w", padx=10, pady=5)

tk.Button(left\_frame, text="Рассчитать", width=42, command=calculate\_genetic\_algorithm, bg="#DDDDDD", activebackground="#CCCCCC", relief=tk.GROOVE).grid(row=12, column=0, columnspan=2, padx=10, pady=10)

# Правая часть интерфейса

tk.Label(right\_frame, text="Результаты", font=("Arial", 11)).grid(row=0, column=0, columnspan=2, pady=10)

tk.Label(right\_frame, text="Лучшее решение достигается в точке:", font=("Arial", 10)).grid(row=1, column=0, columnspan=2, padx=10, pady=5)

canvas2 = tk.Canvas(right\_frame, width=393, height=50, bg="white", borderwidth=1, highlightbackground="#CCCCCC", highlightthickness=2)

canvas2.grid(row=2, column=0, columnspan=2, padx=10, pady=10)

frame\_tree = ttk.Frame(right\_frame, borderwidth=2, relief="groove")

frame\_tree.grid(row=3, column=0, columnspan=2, padx=10, pady=10)

columns = ("Номер", "Результат", "Ген 1", "Ген 2")

tree = ttk.Treeview(frame\_tree, columns=columns, show="headings", height=10)

style = ttk.Style()

style.configure("Treeview.Heading", font=("Arial", 10))

style.configure("Treeview", font=("Arial", 9), rowheight=25)

style.layout("Treeview", [('Treeview.treearea', {'sticky': 'nswe'})])

**for** col **in** columns:

tree.heading(col, text=col)

tree.column(col, width=100)

tree.pack(expand=True, fill="both")

# Включение интерактивного режима для matplotlib

plt.ion()

# Запуск главного цикла обработки событий

root.mainloop()