

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по учебной практике
Тема: Генетические алгоритмы

Студенты гр. 3384

Руководитель

Пьянков М.Ф.
Поляков Г.А.
Жангиров Т.Р.

Санкт-Петербург

2025

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студенты Пьянков М.Ф., Поляков Г.А.

Группа 3384

Тема практики: Генетические алгоритмы

Задание на практику:

Для заданного полинома $f(x)$ (степень не больше 8) необходимо найти все точки максимума (локальные и глобальные) на заданном интервале $[l, r]$.

Сроки прохождения практики: 25.06.2025 — 08.07.2025

Дата сдачи отчета:

Дата защиты отчета:

Студент

Руководитель

Пьянков М.Ф.
Поляков Г.А.
Жангиров Т.Р.

АННОТАЦИЯ

Цель учебной практики — освоение генетических алгоритмов на примере задачи поиска максимумов полинома. В ходе работы необходимо реализовать генетический алгоритм для нахождения всех локальных максимумов функции $f(x)$ (степени ≤ 8) на интервале $[l, r]$. Основные этапы: выбор подходов к реализации функций в генетическом алгоритме, его реализация, визуализация работы алгоритма с GUI интерфейсом. Практика закрепляет навыки работы с эволюционными алгоритмами и оптимизационными задачами.

SUMMARY

The goal of the training practice is to master genetic algorithms using the example of the problem of finding the maximums of a polynomial. During the work, it is necessary to implement a genetic algorithm to find all local maxima of the function $f(x)$ (degree ≤ 8) on the interval $[l, r]$. The main stages: choosing approaches to implementing functions in a genetic algorithm, its implementation, visualizing the algorithm's operation with a GUI interface. The practice consolidates the skills of working with evolutionary algorithms and optimization problems.

СОДЕРЖАНИЕ

	Введение	5
1.	Генетический алгоритм	6-9
1.1.	Архитектура	6-8
1.2.	Реализация	8-10
2.	GUI	11
2.1.	Реализация	11
	Заключение	0
	Список использованных источников	0
	Приложения	0

ВВЕДЕНИЕ

Генетический алгоритм — это эвристический метод оптимизации, основанный на принципах естественного отбора и генетики, данный метод использует операции скрещивания, мутации и отбора для поиска решений. Он эффективен для задач, где традиционные методы неприменимы или требуют больших вычислительных затрат.

Главные составляющие генетического алгоритма:

- 1) Отбор лучших особей в эпохе
- 2) Скрещивание особей
- 3) Мутация

Примечания по условию: так как степень полинома не больше 8, значит количество точек экстремума максимум 7 (степень производной полинома) из них максимумов может быть 3 или 4.

Основная проблема: особи могут группироваться вокруг одной точки максимума, после нескольких запусков алгоритма могут быть найдены не все максимумы.

1. ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

1.1. Архитектура

Работа генетического алгоритма в виде блок-схемы представлена на рисунке 1:

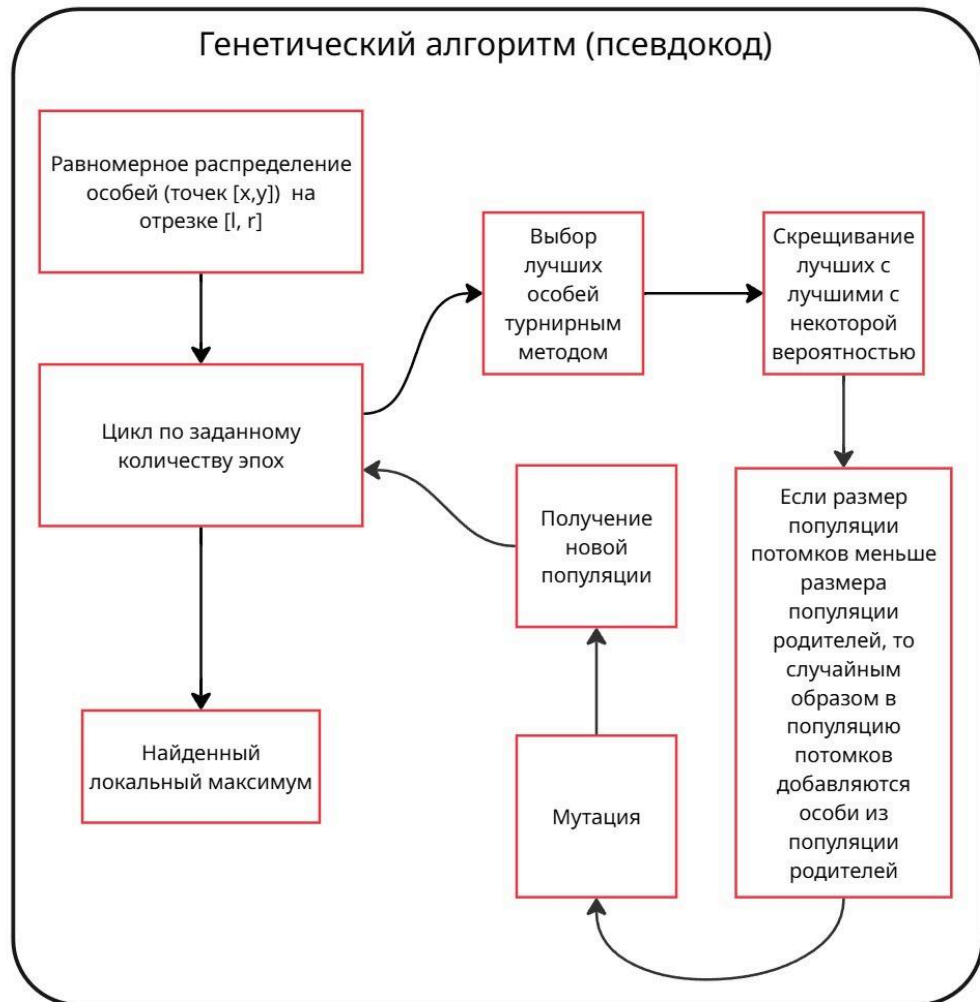


Рисунок 1 - Блок-схема

- 1) Каждая особь в популяции - это точка с одним вещественным параметром: x .
- 2) Особи первой популяции равномерно распределены на отрезке $[l, r]$.
- 3) Далее для каждой особи считается функция приспособленности и выявляется топ лучших особей турнирным методом.
- 4) Лучшие особи скрещиваются по методу скрещивания со смещением с лучшими с некоторой вероятностью, получается популяция потомков, при

необходимости случайным образом в популяцию потомков добавляются особи из популяции родителей.

5) Далее применяется вещественная мутация.

6) Шаги 3, 4 и 5 производятся до тех пор, пока количество эпох не превысит максимальное число.

Решение проблемы концентрации точек вокруг одного максимума: ввести штраф за близость к уже найденным максимумам. Также для случая, когда все максимумы уже найдены, а итерации запуска алгоритма продолжаются, предусмотрена специальная проверка на то, что найденный ответ (максимум в популяции) является локальным максимумом для этой популяции.

Алгоритм запускается 5 раз, за такое количество итераций возможно найти все локальные максимумы для заданного полинома степени не большей 8.

Особь и алгоритм будут представлены в виде классов с полями и методами.

Параметры, задаваемые для работы генетического алгоритма:

DEFAULT_POLINOM - полином, для которого производится поиск всех локальных максимумов.

DEFAULT_LEFT_BORDER - левая граница для поиска локального максимума.

DEFAULT_RIGHT_BORDER - правая граница для поиска локального максимума.

POPULATION_SIZE - размер популяции (сохраняется на каждом шаге работы алгоритма).

MAX_EPOCHS - число эпох.

ITERATIONS - число запусков генетического алгоритма (за каждый запуск предполагается найти локальный максимум).

P_CROSSOVER - вероятность скрещивания особей в 4-ом шаге.

P_MUTATION - вероятность мутации в 5-ом шаге.

DEFAULT_TOURNMENT_OPPONENTS - количество соперников в турнирном методе (3-ий шаг).

DEFAULT_ALPHA - параметр, задающий смещение в методе скрещивания смещением.

DEFAULT_SIGMA_SHARE - предполагаемая “ширина” ниши (длина отрезка между двумя последовательными минимумами). Формула расчёта, если (предполагается равномерное распределение 4 максимумов на отрезке поиска $(\text{DEFAULT_RIGHT_BORDER} - \text{DEFAULT_LEFT_BORDER})/12 + 1$).

1.2. Реализация

Функция *def getFunctionDots(n: int, l: float, r: float, func)* - получение точек функции в заданном интервале для её отображения.

Функция *def save_plots(i, j, max_epochs, l, r, x_func, y_func, history_x, history_y, history_max, population_size, ans, max_iterations)* - специальная функция используемая в параллельной отрисовки графиков.

Класс *class Individual*: поля: *self.value* - вещественное значение (x - на оси абсцисс), методы: *def getValue(self) -> float* - получить хромосому, *def __repr__(self) -> str* - вывести в виде строки особь (для удобства).

Функция *def createIndividual(x: float) -> Individual*: создаёт особь с хромосомой x.

Функция *def createPopulation(n: int, l: float, r: float)*: создаёт равномерно распределённую на интервале [l, r] выборку особей при помощи функции *createIndividual(x: float)*.

Функция *def mutation(individual: Individual) -> None*: производит вещественную мутацию особи.

Функция *def crossFunc(first: Individual, second: Individual, alpha=DEFAULT_ALPHA) -> Individual*: реализация скрещивания смещением.

Класс *class GenAlgorithm*: поля: все параметры, указанные в разделе 1.1 Архитектура, *self.history_x* - значения хромосом всех особей на каждой эпохе, *self.history_y* - значения полинома в точках - значениях хромосом всех особей на каждой эпохе, *self.history_max* - хранение всех найденных максимумов.

Функции класса *class GenAlgorithm* будут рассмотрены отдельно ниже.

Функция *def fitnessFunc(self, individual: Individual, sigma_share=DEFAULT_SIGMA_SHARE) -> float*: - функция приспособленности (фитнес-функция), производит расчёт значения, характеризующего близость особи к максимуму. Штраф считается отдельно по формуле

$$\sum_{X_i \in \text{history_max}} \frac{(RIGHT-LEFT)*2}{|X_i - x| + 0.001} * \text{indicator}(|X_i - x| < \text{sigma_share}), \text{ где}$$

RIGHT, LEFT- левая и правая границы поиска локальный максимумов

полинома соответственно, X_i - максимум из *self.history_max*, x - значение

хромосомы рассматриваемой особи, *indicator*($|X_i - x| < \text{sigma_share}$) -

возвращает 1, если условие в скобках выполнено, 0 - в противном случае.

Функция *def tournamentSelection(self, population, n=DEFAULT_TOURNAMENT_OPPONENTS)* - реализует турнирный отбор: *population_size* раз выбирается n особей из популяции, лучший добавляется в результат. Таким образом получается *population_size* победителей в турнире (лучших особей) (они могут повторяться).

Функция *def findLocalMax(self, ans)* - ищет локальный максимум для последовательности особей *ans*, предварительно расположив их в порядке возрастания по значениям на оси абсцисс. Необходима для определения ложного нахождения локального максимума.

Функция *def fit(self)* - основная функция, в которой выполняются шаги 3, 4 и 5 (реализуется полный цикл работы алгоритма по всем эпохам). Производится вызов функций *tournamentSelection()*, *crossFunc()*, *mutation()*, *findLocalMax()*, накопление статистики положения особей для каждой эпохи на координатной плоскости, сохранение найденных максимумов.

Функция *def run(iterations=ITERATIONS, max_epochs=MAX_EPOCHS, l=DEFAULT_LEFT_BORDER, r=DEFAULT_RIGHT_BORDER, polinom=DEFAULT_POLINOM, population_size=POPULATION_SIZE, p_crossover=P_CROSSOVER, p_mutation=P_MUTATION,*

tournment_opponents=DEFAULT_TOURNAMENT_OPPONENTS,
alpha=DEFAULT_ALPHA) - запуск алгоритма с переданными параметрами,
применение параллельной отрисовки графиков при помощи *Pool()* из
библиотеки *multiprocessing*.

В точке входа запускается функция *run()* Весь код см. в ПРИЛОЖЕНИЕ Б.

2. GUI

2.1. Реализация

Для создания графической оболочки программы было принято решения использовать библиотеку PyQ6, которая является набором расширений графического фреймворка Qt для Python.

PyQt6 обеспечивает полную кроссплатформенность: приложения, разработанные с его помощью, работают на Windows, macOS, Linux без изменения кода. Библиотека предоставляет более 600 классов и 6000 функций, что позволяет создавать современный пользовательский интерфейс.

При помощи данной библиотеки были созданы классы MainWindow, MainMenu, Visualisation и Results.

MainView, WelcomeView, SettingsDialog, IterationView и ResultView.

Класс MainWindow реализует функционал главного окна приложения, данный класс управляет жизненным циклом приложения и координирует взаимодействие между другими модулями.

В конструкторе класса происходит инициализация главного окна и модулей приложения, создание контейнера для экранов и настройку связей между компонентами. Рассмотрим другие методы класса:

def closeEvent(self, event) - обрабатывает закрытие окна.

def switch_to_main_menu(self) - переключает на стартовый экран приложения.

def switch_to_visualisation(self) - переключает на экран просмотра итераций алгоритма.

def switch_to_results(self) - переключает на экран просмотра результатов работы алгоритма (просмотр найденных максимумов).

Класс MainView является стартовым экраном приложения, он реализует интерфейс для ввода функции и настройки границ интервала для работы алгоритма, отображает график введенного полинома на экране, также предоставляет возможность пользователю изменить гиперпараметры алгоритма. Конструктор класса выполняет инициализацию словаря для хранения

параметров и запускает метод для построения интерфейса. Рассмотрим другие методы класса:

`def initUI(self)` - создаёт и настраивает пользовательский интерфейс, который состоит из навигационной панели, области для ввода полинома и визуализации его графика, а также области для изменения гиперпараметров и интервала работы алгоритма.

`def launch_algorithm(self)` - запускает генетический алгоритм с заданными параметрами, экран переключается на визуализацию работы алгоритма.

`def update_function_plot(self)` - выполняет отрисовку графика введенной функции при изменении параметров.

`def create_lambda(self, func_str)` - выполняет преобразование введенной пользователем функции в лямбда-функцию.

Стартовый экран приложения представлен на рисунке 2:

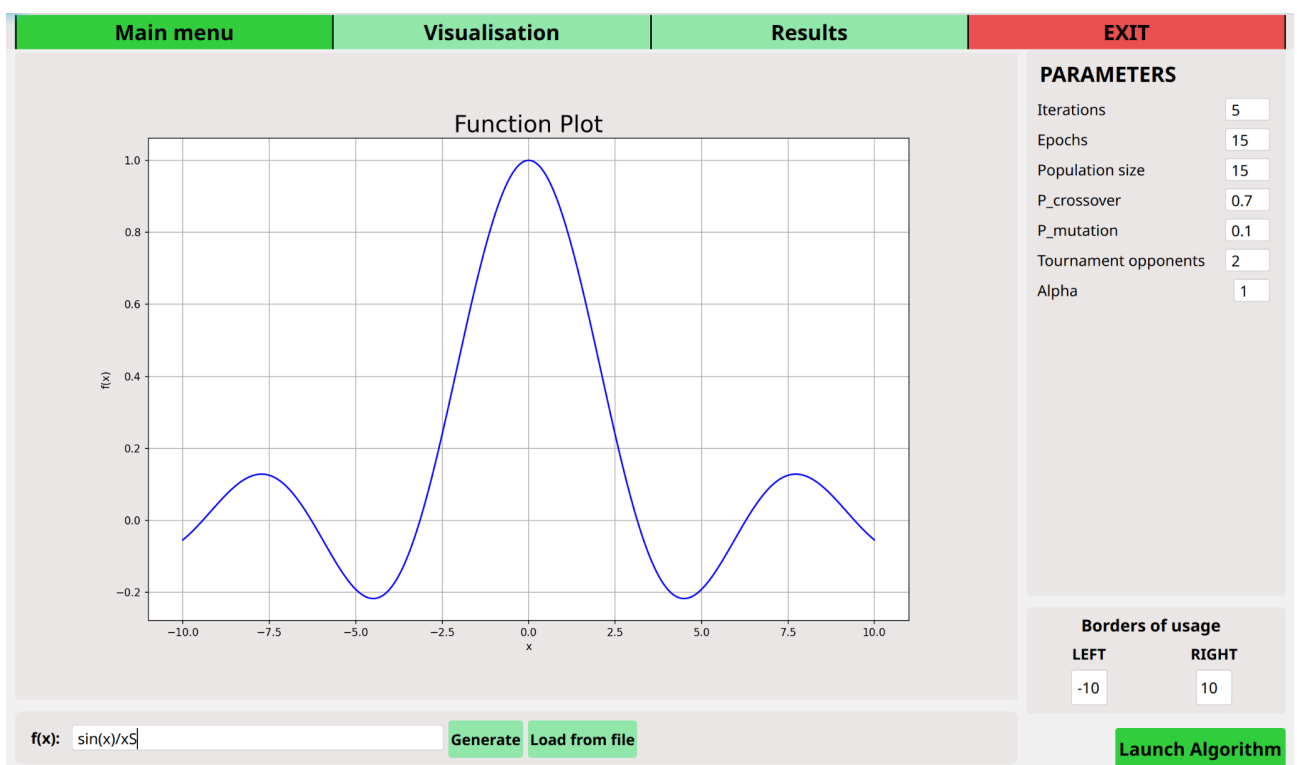


Рисунок 2 - стартовый экран приложения

Класс Visualisation выполняет визуализацию процесса выполнения генетического алгоритма: пошаговый просмотр итераций оптимизации и графика средней функции приспособленности. Рассмотрим методы класса:

`def initUI(self)` - создаёт и настраивает пользовательский интерфейс: навигационную панель, область визуализации графика работы алгоритма и графика средней функции приспособленности, слайдер для перехода между шагами алгоритма, а также кнопки для управления кадрами.

`def create_visualisation_frame(self, title)` - создаёт области для отображения графика.

Методы для навигации по шагам алгоритма:

`def go_to_first(self)` - выполняет переход к началу выполнения алгоритма.

`def go_to_last(self)` - выполняет переход в конец выполнения алгоритма.

`def go_to_previous(self)` - переходит к предыдущему шагу алгоритма.

`def go_to_next(self)` - переходит к следующему шагу алгоритма

`def go_to_specific_iteration(self)` - переход к определённой итерации алгоритма.

`def start_visualisation(self)` - запускает визуализацию алгоритма.

`def pause_visualisation(self)` - останавливает визуализацию алгоритма.

`def update_visualisation(self)` - выполняет переход к следующему кадру.

`def slider_changed(self, value)` - обрабатывает изменения положения слайдера.

`def load_frame(self, frame_num)` - выполняет отображение конкретного кадра.

Экран приложения при просмотре итерации алгоритма представлен на рисунке 3:

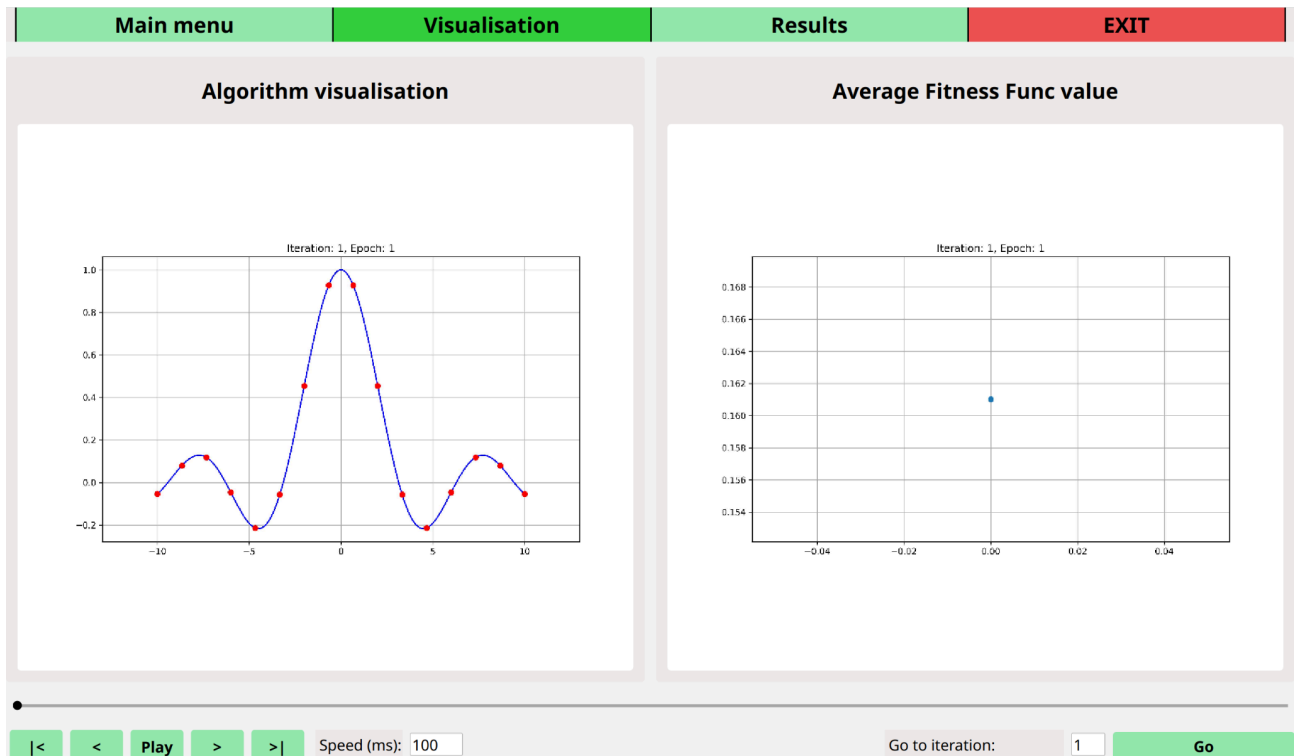


Рисунок 3 - экран приложения при просмотре шагов алгоритма
Класс Results отображает итоговые результаты работы алгоритма в виде таблицы. Рассмотрим методы класса:

`def initUI(self)` - создаёт и настраивает пользовательский интерфейс: навигационную панель и таблицу для отображения результатов.

`def update_results(self, results)` - обновляет отображаемые результаты.

Экран приложения при просмотре результатов работы алгоритма представлен на рисунке 4:

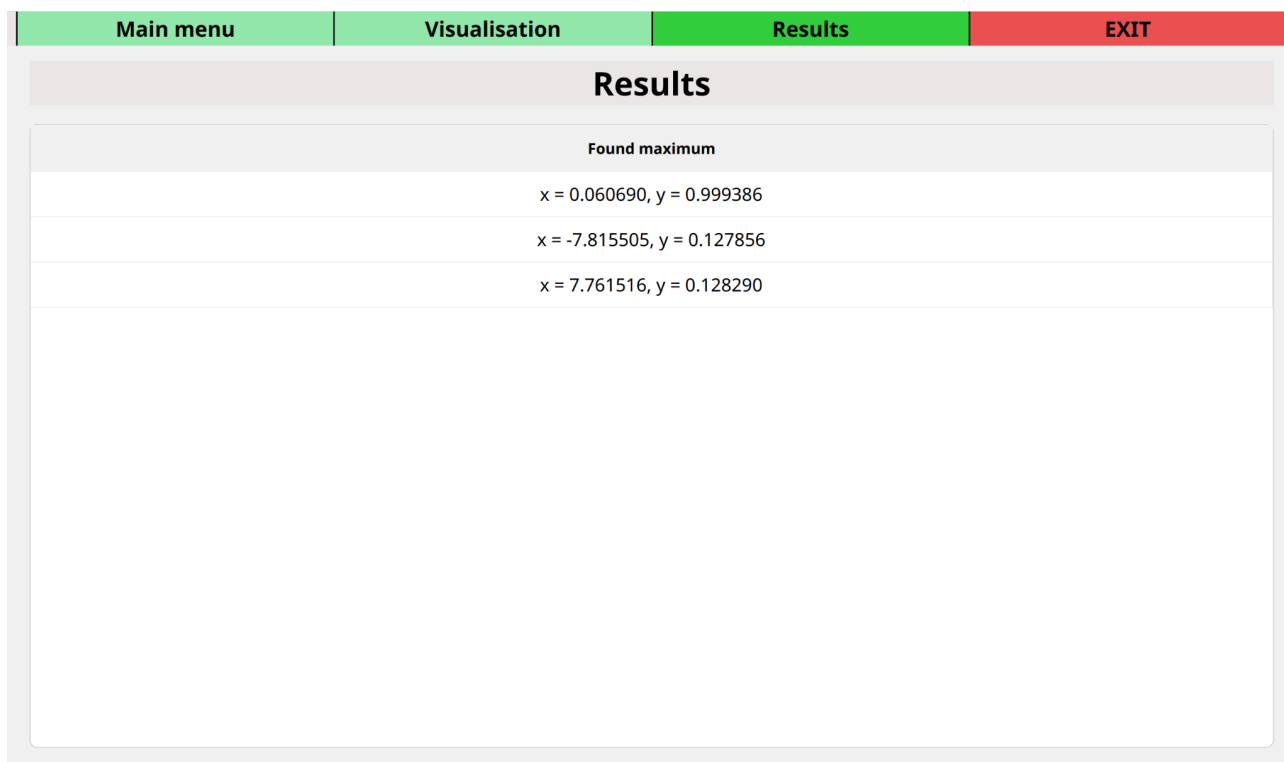


Рисунок 4 - экран приложения при просмотре результатов работы алгоритма

ЗАКЛЮЧЕНИЕ

Кратко подвести итоги, проанализировать соответствие поставленной цели и полученного результата.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

ПРИЛОЖЕНИЕ А

Код алгоритма

```
import random
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use('agg')
import math

from multiprocessing import Pool
from functools import partial

import os

DEFAULT_LEFT_BORDER = 1
DEFAULT_RIGHT_BORDER = 30

DEFAULT_POLINOM = lambda x: x**3

POPULATION_SIZE = 15
P_CROSSOVER = 0.7
P_MUTATION = 0.1
MAX_EPOCHS = 15
ITERATIONS = 5

DEFAULT_TOURNAMENT_OPPONENTS = 3
DEFAULT_ALPHA = 1

class Individual:
    def __init__(self, x: float) -> None:
        self.value = x
    def getValue(self) -> float:
        return self.value
    def __repr__(self) -> str:
        return str(self.value)

def createIndividual(x: float) -> Individual:
    return Individual(x)

def createPopulation(n: int, l: float, r: float):
    population= []

    interval_length = r - l
    step = interval_length / n
    current = l
    while(current < r):
        population.append(createIndividual(current))
        current += step
    return population

def mutation(individual: Individual, l, r) -> None:
    temp = individual.getValue()
    value = temp + random.uniform(-2, 2)
    value = max(min(value, r), l)
    individual.value = value
```

```

def crossFunc(first: Individual, second: Individual, alpha, l, r) ->
Individual:
    x = first.getValue()
    if random.random() < 0.5:
        x = second.getValue()
    left_border = x - alpha
    right_border = x + alpha

    child_value = random.uniform(left_border, right_border)

    child_value = max(min(child_value, r), l)

    return Individual(child_value)

class GenAlgorithm:
    def __init__(self, max_epochs=MAX_EPOCHS,
population_size=POPULATION_SIZE,\
                left_border=DEFAULT_LEFT_BORDER,
right_border=DEFAULT_RIGHT_BORDER,\
                function=DEFAULT_POLINOM, p_crossover=P_CROSSOVER,\
                p_mutation=P_MUTATION,
tournament_opponents=DEFAULT_TOURNAMENT_OPPONENTS,\
                alpha=DEFAULT_ALPHA, sigma_share=1) -> None:
        self.population_size = population_size
        self.p_crossover = p_crossover
        self.p_mutation = p_mutation
        self.max_epochs = max_epochs
        self.sigma_share = sigma_share
        self.alpha = alpha
        self.tournament_opponents = tournament_opponents

        self.function = function
        self.left_border = left_border
        self.right_border = right_border

        self.history_x = []
        self.history_y = []
        self.history_max = []
        self.population = None

    def fitnessFunc(self, individual: Individual) -> float:
        value = self.function(individual.getValue())
        fine = 0.0
        for maximum in self.history_max:
            if math.fabs(maximum[0] - individual.getValue()) <
self.sigma_share:
                fine += (self.right_border - self.left_border)*2 /
(math.fabs(maximum[0] - individual.getValue()) + 0.001)

        return value - fine

    def tournamentSelection(self, population):
        selected = []
        for _ in range(self.population_size):
            participants = random.sample(population,
self.tournament_opponents)

```

```

        best_ind = max(participants, key=lambda ind:
self.fitnessFunc(ind))
        selected.append(best_ind)
        return selected

def findLocalMax(self, ans):
    result = None
    ans = sorted(ans, key=lambda x: x[0])
    for i in range(1, len(ans) - 1):
        if ans[i-1][1] < ans[i][1] > ans[i+1][1]:
            result = ans[i]
    return result

def fit(self):
    self.population = createPopulation(self.population_size,
self.left_border, self.right_border)

    self.history_x.append([ind.getValue() for ind in self.population])
    self.history_y.append([self.function(ind.getValue()) for ind in
self.population])

    state = []
    i = 0
    while(i < self.max_epochs):

        best_ind = self.tournamentSelection(self.population)
        best_ind_shuffled = self.tournamentSelection(self.population)
        random.shuffle(best_ind_shuffled)

        childs = []
        for j in range(self.population_size):
            if random.random() < self.p_crossover:
                childs.append(crossFunc(best_ind[j], best_ind_shuffled[j],
self.alpha, self.left_border, self.right_border))

        childs_length = len(childs)
        add = set()
        while(childs_length) < self.population_size:
            choise = random.choice(self.population)

            if not(choise in add):
                add.add(choise)
                childs_length += 1

        self.population = childs + list(add)

        for j in range(self.population_size):
            if random.random() < self.p_mutation:
                mutation(self.population[j], self.left_border,
self.right_border)

        self.history_x.append([ind.getValue() for ind in
self.population])
        self.history_y.append([self.function(ind.getValue()) for ind in
self.population])

```

```

        i += 1

        local_ans = [(self.population[i].getValue(),
self.function(self.population[i].getValue())) for i in
range(self.population_size)]
        found_max = max(local_ans, key=lambda x: x[1])
        found_local_max = self.findLocalMax(local_ans)

        total_ans = None
        if found_local_max != None and found_max[1] == found_local_max[1]
or (found_local_max == None and (found_max[0] - self.left_border < 1e-2
or self.right_border - found_max[0] < 1e-2)):
            flag = True
            for maximum in self.history_max:
                if math.fabs(maximum[0] - found_max[0]) < self.sigma_share:
                    flag = False
                    break
            if flag:
                total_ans = found_max
                self.history_max.append(found_max)

        return total_ans

def getFunctionDots(n: int, l: float, r: float, func):
    interval_length = r - l
    step = interval_length / n
    current = l
    x = []
    y = []
    while(current < r):
        x.append(current)
        y.append(func(current))
        current += step
    return x, y

def save_plots(i, j, max_epochs, l, r, x_func, y_func, history_x,
history_y, history_max, population_size, ans, max_iterations):
    plt.figure(figsize=(10, 6))
    plt.plot(x_func, y_func, 'b')
    plt.xlim(l - abs(0.3 * r), r + abs(0.3 * r))
    plt.plot(history_x[i], history_y[i], 'ro')
    plt.grid()

    x_max, y_max = [history_max[i][0] for i in range(len(history_max) -
1)], [history_max[i][1] for i in range(len(history_max) - 1)]
    if ans == None or (j+1)*(i+1) == max_iterations*max_epochs:
        x_max.append(history_max[len(history_max) - 1][0])
        y_max.append(history_max[len(history_max) - 1][1])
        plt.plot(x_max, y_max, 'go')

    plt.title(f"Iteration: {j}, Epoch: {i}")
    filename = f'./frames/algorithm_{j * max_epochs + i}.jpg'
    plt.savefig(filename, dpi=300)
    plt.close()

    plt.figure(figsize=(10, 6))

```

```

        average_fitness = [sum(history_y[k]) / population_size for k in
range(i + 1)]
        plt.plot(average_fitness, marker='o', linestyle='-')
        plt.grid()
        plt.title(f"Iteration: {j}, Epoch: {i}")
        plt.savefig(f'./frames/average_fitness_{j * max_epochs + i}.jpg',
dpi=300)
        plt.close()

def run(iterations=ITERATIONS, max_epochs=MAX_EPOCHS,
        l=DEFAULT_LEFT_BORDER, r=DEFAULT_RIGHT_BORDER,
        polinom=DEFAULT_POLINOM, population_size=POPULATION_SIZE,
        p_crossover=P_CROSSOVER, p_mutation=P_MUTATION,
        tournament_opponents=DEFAULT_TOURNAMENT_OPPONENTS,
alpha=DEFAULT_ALPHA):
    sigma_share = (r - l) / 12 + 1

    if not os.path.exists('frames'):
        os.makedirs('frames')
    else:
        for file in os.listdir('frames'):
            if file.endswith('.jpg'):
                os.remove(os.path.join('frames', file))

    x_func, y_func = getFunctionDots(1000, l, r, polinom)

    random.seed(42)
    A = GenAlgorithm(max_epochs, population_size, l, r, polinom,
p_crossover, p_mutation, tournament_opponents, alpha, sigma_share)

    with Pool() as pool:
        for j in range(iterations):
            ans = A.fit()

            worker = partial(save_plots,
                            j=j,
                            max_epochs=max_epochs,
                            l=l,
                            r=r,
                            x_func=x_func,
                            y_func=y_func,
                            history_x=A.history_x,
                            history_y=A.history_y,
                            history_max=A.history_max,
                            population_size=population_size,
                            ans=ans,
                            max_iterations=iterations)

            pool.map(worker, range(max_epochs))

    A.history_x = []
    A.history_y = []
    print(ans)

    print(A.history_max)

    return A.history_max

```

```
if __name__ == "__main__":  
    run()
```

ПРИЛОЖЕНИЕ Б

Код GUI

ПРИЛОЖЕНИЕ В

Примеры