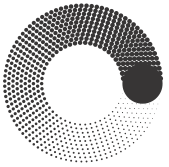
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**



**МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

***Факультет информационных технологий***

***Кафедра Информатики и информационных технологий***

**направление подготовки**

**09.04.02 «Информационные системы и технологии»,**

**ПРАКТИЧЕСКАЯ РАБОТА № 2**

**ВАРИАНТ: 8**

**Дисциплина:** **Искусственный интеллект в мобильных системах**

**Тема: Разработка генетического алгоритма поиска экстремума функции**

**Выполнила: студентка группы 224-371** Лейн Ф. Е.

(Фамилия И.О.)

**Проверил:** Попов Д.И.

(Фамилия И.О. )

**Дата, подпись** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  ***\_\_\_\_\_\_\_\_\_\_\_***

(Дата) (Подпись)

**ЗАДАНИЕ**

Реализовать на языке программирования (по Вашему выбору) программу поиска оптимума (максимума/минимума) функции F(X) на интервале [A,B] с использованием генетического алгоритма.

**ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ**

1. F(x) = sin(x)-x/3

3. F(x) = cos(x)-x/3

8. F(x) = 2\*sin(x)+x/5

**ХОД РАБОТЫ**

Для разработки приложения был выбран язык Python с использованием пакета PyQT для создания графического интерфейса и пакета matplotlib для отображения графиков.

В начале была выполнена разработка макета (формы) приложения, используя QtDesigner, согласно заданию. Результат показан на рисунке 1.

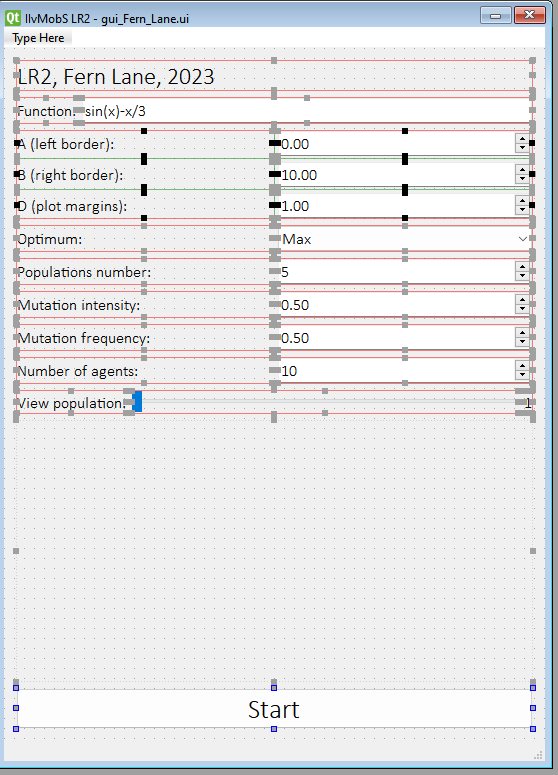


Рисунок 1 – Разработка формы приложения

Далее, по заданию, был создан класс Agent, хранящий данные особи и реализующий функцию мутации, что показано в листинге 1.

Листинг 1 – Класс Agent

class Agent:  
 def \_\_init\_\_(self, x) -> None:  
 *# Initialize private variables* self.\_x = x  
 self.\_y = None  
  
 def mutate(self, intensity: float, frequency: float) -> None:  
 *"""  
 Calculates one mutation step  
 :param intensity: Intensity of mutation  
 :param frequency: Frequency of mutation  
 :return:  
 """* if np.random.rand() <= frequency:  
 if np.random.rand() > .5:  
 self.\_x += intensity  
 else:  
 self.\_x -= intensity  
  
 def calculate(self, function) -> None:  
 *"""  
 Applies function to calculate y from x  
 :param function: parsed function  
 :return:  
 """* self.\_y = function(self.\_x)  
  
 def r\_x(self) -> float:  
 *"""  
 Returns private x variable value  
 :return:  
 """* return self.\_x  
  
 def r\_y(self) -> float | None:  
 *"""  
 Returns private y variable value  
 :return:  
 """* return self.\_y  
  
 def as\_point(self) -> np.ndarray:  
 *"""  
 Returns x and y  
 :return: x and y as numpy array  
 """* return np.asarray([self.\_x, self.\_y], dtype=np.float32)

После чего, был создан класс Window, реализующий графический интерфейс и основной код приложения, согласно заданию. Исходный код находится в приложении А.

Далее, были выбраны формулы, в соответствии с индивидуальным заданием. После чего был произведен подбор различных настроек генерации популяций. На рисунках 2-3 показан результат выполнения программы для формулы F(x) = sin(x)-x/3.

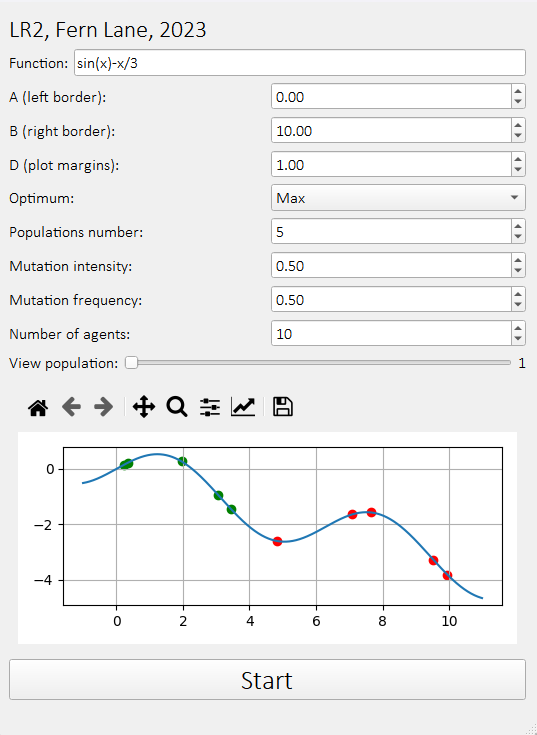
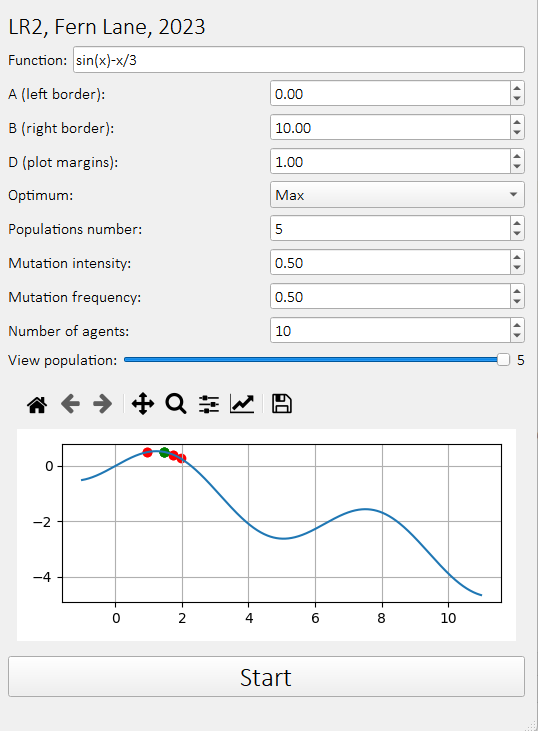
 

Рисунок 2 – 1-ая популяция Рисунок 3 – 5-ая популяция

Как можно заметить, при выбранном максимуме, в качестве оптимума, после 5 поколений, особи оказались в точке максимума функции на указанном промежутке.

Далее, На рисунках 4 и 5 показан результат выполнения для формулы F(x) = cos(x)-x/3 с использованием минимума в качестве оптимума. Как можно заметить, особи оказались в минимуме функции после 5 популяции.

Далее, на рисунках 6 и 7 показан результат выполнения для формулы F(x) = 2\*sin(x)+x/5.

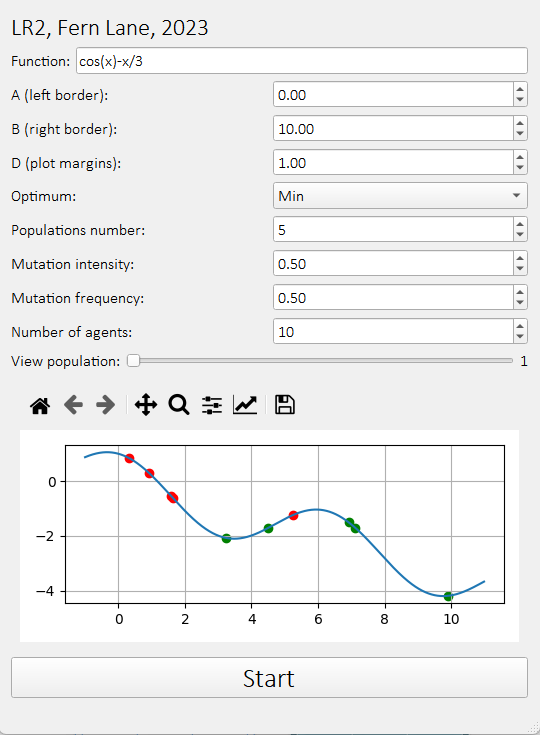
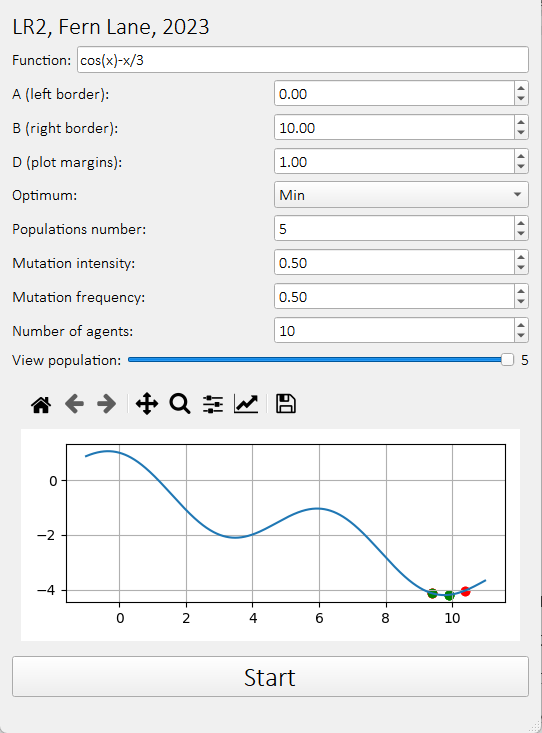
 

Рисунок 4 – 1-ая популяция Рисунок 5 – 5-ая популяция

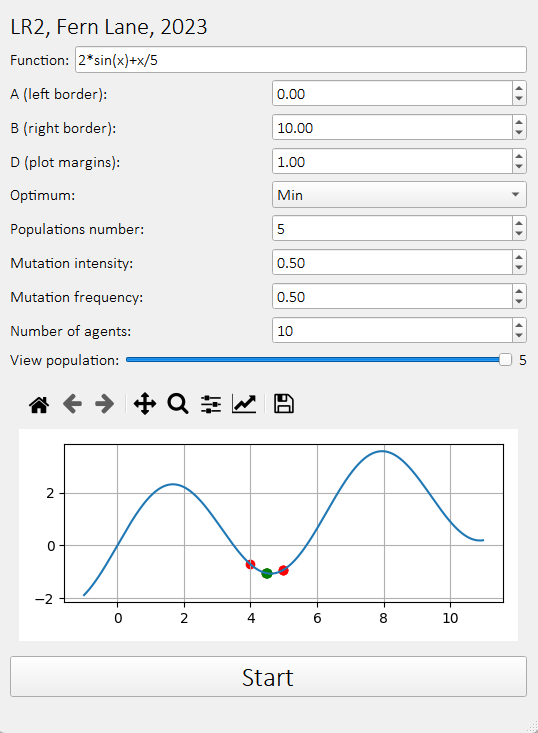
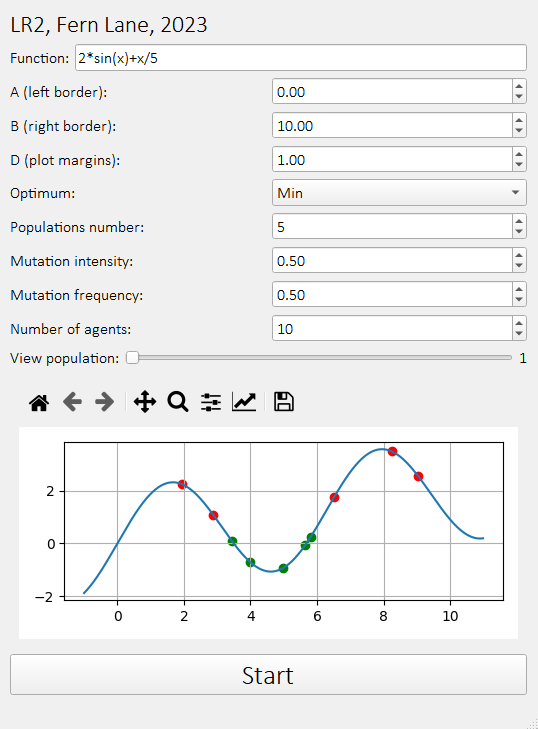


Рисунок 6 – 1-ая популяция Рисунок 7 – 5-ая популяция

**ВЫВОД**

В результате выполнения данной практической работы были изучены навыки по созданию приложения для разработки генетического алгоритма поиска экстремума функции. Разработанное приложение было протестировано с использованием 3-ёх функций из индивидуального задания.

**Приложение А. Листинг кода main.py**

import os  
import sys  
import traceback  
  
import numpy as np  
import sympy  
from PyQt5 import uic  
from PyQt5.QtWidgets import QMainWindow, QApplication, QVBoxLayout, QMessageBox  
from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg  
from matplotlib.backends.backend\_qt5agg import NavigationToolbar2QT  
from matplotlib.figure import Figure  
  
*# GUI file*GUI\_FILE = "gui\_Fern\_Lane.ui"  
GUI\_FILE = os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), GUI\_FILE))  
  
*# "fusion" or "windows"*GUI\_STYLE = "fusion"  
  
  
class Window(QMainWindow):  
 def \_\_init\_\_(self):  
 super(Window, self).\_\_init\_\_()  
  
 *# 3D lists for storing calculated population's points instead af pictures of the plots as files =)  
 # (x, y) \* points \* populations  
 # ex.: point\_x = self.population\_points\_bad[population\_index][point\_n][0]* self.points\_bad = []  
 self.points\_good = []  
  
 *# Numpy arrays for storing function (for plotting)* self.x\_values = np.empty((0,), dtype=np.float32)  
 self.y\_values = np.empty((0,), dtype=np.float32)  
  
 *# Load GUI from file* uic.loadUi(GUI\_FILE, self)  
  
 *# Initialize matplotlib* self.plt\_view = FigureCanvasQTAgg(Figure(tight\_layout=True))  
 self.plt\_axes = self.plt\_view.figure.subplots()  
 self.plt\_axes.grid(True, which="both")  
 plt\_toolbar = NavigationToolbar2QT(self.plt\_view, self)  
 v\_box\_layout = QVBoxLayout()  
 v\_box\_layout.addWidget(plt\_toolbar)  
 v\_box\_layout.addWidget(self.plt\_view)  
 self.widget.setLayout(v\_box\_layout)  
  
 *# Connect button* self.pushButton.clicked.connect(self.start)  
  
 *# Connect slider* self.horizontalSlider.valueChanged.connect(self.change\_view)  
  
 *# Show GUI* self.show()  
  
 def start(self) -> None:  
 *"""  
 Start button callback. Main program entry  
 :return:  
 """* try:  
 *# Parse function from GUI* function = sympy.lambdify(sympy.Symbol("x"), self.lineEdit.text())  
  
 *# Retrieve other data from elements* border\_left = self.doubleSpinBox.value()  
 border\_right = self.doubleSpinBox\_2.value()  
 plot\_margins = self.doubleSpinBox\_3.value()  
 optimum = self.comboBox.currentIndex()  
 population\_count = self.spinBox.value()  
 mutation\_intensity = self.doubleSpinBox\_4.value()  
 mutation\_frequency = self.doubleSpinBox\_5.value()  
 agents\_number = self.spinBox\_2.value()  
  
 *# Set population count as maximum for the slider and reset it* self.horizontalSlider.setMaximum(population\_count - 1)  
 self.horizontalSlider.setValue(0)  
  
 *# Clear population points* self.points\_bad.clear()  
 self.points\_good.clear()  
  
 *# Calculate function (for plotting)* self.x\_values = np.arange(border\_left - plot\_margins, border\_right + plot\_margins, 0.01, dtype=np.float32)  
 self.y\_values = function(self.x\_values)  
  
 *# Initialize agents and append them into list* agents = []  
 for i in range(agents\_number):  
 agent\_x = (border\_right - border\_left) \* np.random.rand() + border\_left  
 agent = Agent(agent\_x)  
 agent.calculate(function)  
 agents.append(agent)  
  
 for population\_index in range(population\_count):  
 *# Initialize lists for storing current population's points* population\_points\_bad = []  
 population\_points\_good = []  
  
 *# Find bad points (maximums or minimums) using half of agents  
 # (delete half of the agents, the worst ones)* for \_ in range(agents\_number // 2):  
 *# Index of rejected agent* index = 0  
  
 *# 0 - Max (selected item's index from comboBox)  
 # Find minimum (because minimum is the worst)* if optimum == 0:  
 minimum = np.inf  
 for agent\_index in range(len(agents)):  
 agent = agents[agent\_index]  
 if agent.r\_y() < minimum:  
 minimum = agent.r\_y()  
 index = agent\_index  
  
 *# 1 - Min (selected item's index from comboBox)  
 # Find maximum (because maximum is the worst)* else:  
 maximum = -np.inf  
 for agent\_index in range(len(agents)):  
 agent = agents[agent\_index]  
 if agent.r\_y() > maximum:  
 maximum = agent.r\_y()  
 index = agent\_index  
  
 *# Save it as bad point* population\_points\_bad.append(agents[index].as\_point())  
  
 *# Delete this agent because it's bad* del agents[index]  
  
 *# Consider the remaining agents (half of them) as good ones* for agent in agents:  
 population\_points\_good.append(agent.as\_point())  
  
 *# Save current population points* self.points\_bad.append(population\_points\_bad)  
 self.points\_good.append(population\_points\_good)  
  
 *# Duplicate remaining agents (good ones)* for agent\_index\_half in range(len(agents)):  
 agents.append(Agent(agents[agent\_index\_half].r\_x()))  
  
 *# Mutate them and calculate Y (function) for mutated X* for agent in agents:  
 agent.mutate(mutation\_intensity, mutation\_frequency)  
 agent.calculate(function)  
  
 *# Show first plot* self.plot\_population(0)  
  
 *# Log error in main function if it occurs* except Exception as e:  
 msg = QMessageBox()  
 msg.setIcon(QMessageBox.Critical)  
 msg.setText("Error")  
 msg.setInformativeText(str(e))  
 msg.setWindowTitle("Error")  
 msg.exec\_()  
 traceback.print\_exc()  
  
 def change\_view(self) -> None:  
 *"""  
 Slider callback. Changes label text and plots selected population  
 :return:  
 """* population\_index\_to\_plot = self.horizontalSlider.value()  
 self.label\_11.setText(str(population\_index\_to\_plot + 1))  
 if self.x\_values.shape[0] > population\_index\_to\_plot and self.y\_values.shape[0] > population\_index\_to\_plot:  
 self.plot\_population(population\_index\_to\_plot)  
  
 def plot\_population(self, population\_index: int) -> None:  
 *"""  
 Plots current function and bad and good points for certain population index  
 :param population\_index: index of points to plot  
 :return:  
 """  
 # Clear current plot* self.plt\_axes.clear()  
  
 *# Enable grid* self.plt\_axes.grid(True, which="both")  
  
 *# Plot function* self.plt\_axes.plot(self.x\_values, self.y\_values)  
  
 *# Plot bad points using red color* for point in self.points\_bad[population\_index]:  
 self.plt\_axes.scatter(point[0], point[1], c="r")  
  
 *# Plot good points using green color* for point in self.points\_good[population\_index]:  
 self.plt\_axes.scatter(point[0], point[1], c="g")  
  
 *# Update plot* self.plt\_view.draw()  
  
  
class Agent:  
 def \_\_init\_\_(self, x) -> None:  
 *# Initialize private variables* self.\_x = x  
 self.\_y = None  
  
 def mutate(self, intensity: float, frequency: float) -> None:  
 *"""  
 Calculates one mutation step  
 :param intensity: Intensity of mutation  
 :param frequency: Frequency of mutation  
 :return:  
 """* if np.random.rand() <= frequency:  
 if np.random.rand() > .5:  
 self.\_x += intensity  
 else:  
 self.\_x -= intensity  
  
 def calculate(self, function) -> None:  
 *"""  
 Applies function to calculate y from x  
 :param function: parsed function  
 :return:  
 """* self.\_y = function(self.\_x)  
  
 def r\_x(self) -> float:  
 *"""  
 Returns private x variable value  
 :return:  
 """* return self.\_x  
  
 def r\_y(self) -> float | None:  
 *"""  
 Returns private y variable value  
 :return:  
 """* return self.\_y  
  
 def as\_point(self) -> np.ndarray:  
 *"""  
 Returns x and y  
 :return: x and y as numpy array  
 """* return np.asarray([self.\_x, self.\_y], dtype=np.float32)  
  
  
*# Press the green button in the gutter to run the script.*if \_\_name\_\_ == "\_\_main\_\_":  
 *# Start GUI* app = QApplication.instance() or QApplication(sys.argv)  
 app.setStyle(GUI\_STYLE)  
 win = Window()  
 sys.exit(app.exec\_())