МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Кафедра вычислительных технологий**

**ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3**

**по дисциплине Методы поисковой оптимизации**

Работу выполнила\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ромашкина А.А.

Факультет Компьютерных технологий и прикладной математики

Направление подготовки 02.03.02 Фундаментальная информатика и

информационные технологии курс 4

Краснодар

2022

СОДЕРЖАНИЕ

[Задание 3](#_Toc121175121)

[Алгоритм 3](#_Toc121175122)

[Результат работы программы 4](#_Toc121175123)

[Листинг 5](#_Toc121175124)

Тема работы: Простой генетический алгоритм.

# Задание

Необходимо разработать генетический алгоритм оптимизации функции Розенброкка.

*.*

Она имеет глобальный минимум в точке , где .

# Алгоритм

Генетический алгоритм – это стохастический эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём последовательного подбора, комбинирования и вариации искомых параметров с использованием механизмов, напоминающих биологическую эволюцию. Является разновидностью эволюционных вычислений. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

Предполагает имитацию процесса эволюции (различные реализации могут использовать различные приемы имитирующие этапы эволюционного процесса). Например:

Значения переменных (точка) – особь.

Значение функции в точке характеризует приспособленность особи (выживаемость).

Пусть задана выпуклая функция переменных .

Данный метод предполагает следующие *шаги*:

1. **Создание начальной популяции** – множества особей.
2. **Выбор родителей**. Полагаем, что для появления новой особи требуются наследственные признаки двух других особей. Выбор может производится различными способами, например, случайно (панмиксия).
3. **Рекомбинация**. Передача наследственных признаков потомку. Существуют различные способы реализации данного шага. Например, вектор переменных можно рассматривать как набор генов, при этом одноименные гены родителей объединяются со случайными коэффициентами.
4. **Мутация**. Случайное изменение наследственных признаков. Например, умножение гена на случайное число.
5. **Селекция**. Новая особь помещается в популяцию на место наименее приспособленной особи, которая, освобождая позицию, погибает.

**Сходимость**. Процесс смены поколений повторяется пока на протяжении некоторого (заданного) числа поколений не будет происходить уменьшение (увеличение) значения целевой функции.

# Результат работы программы

Вся работа программы представлена на рисунках 1-2 ниже.

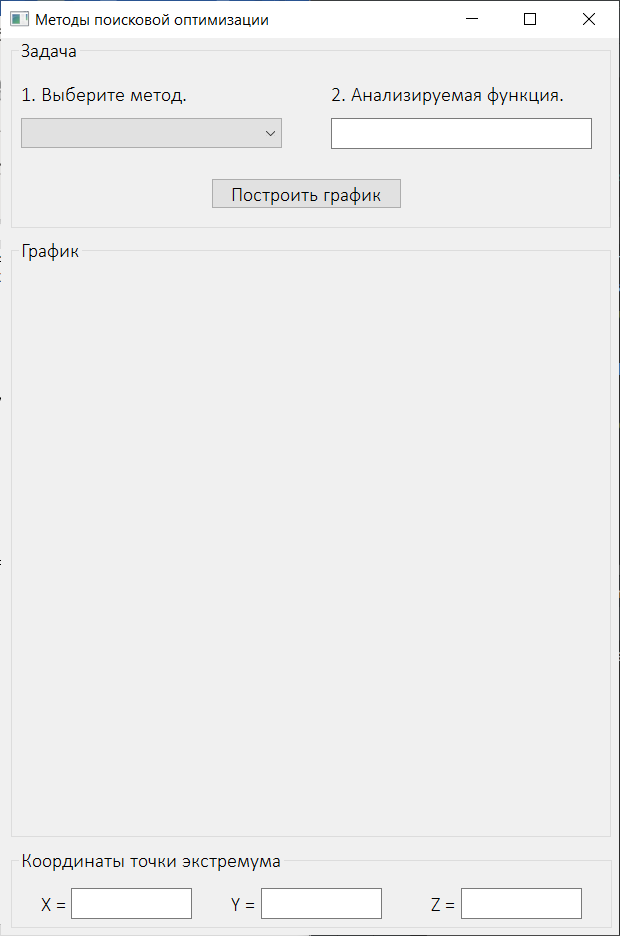


Рисунок 1 – Начальный вид программы при запуске

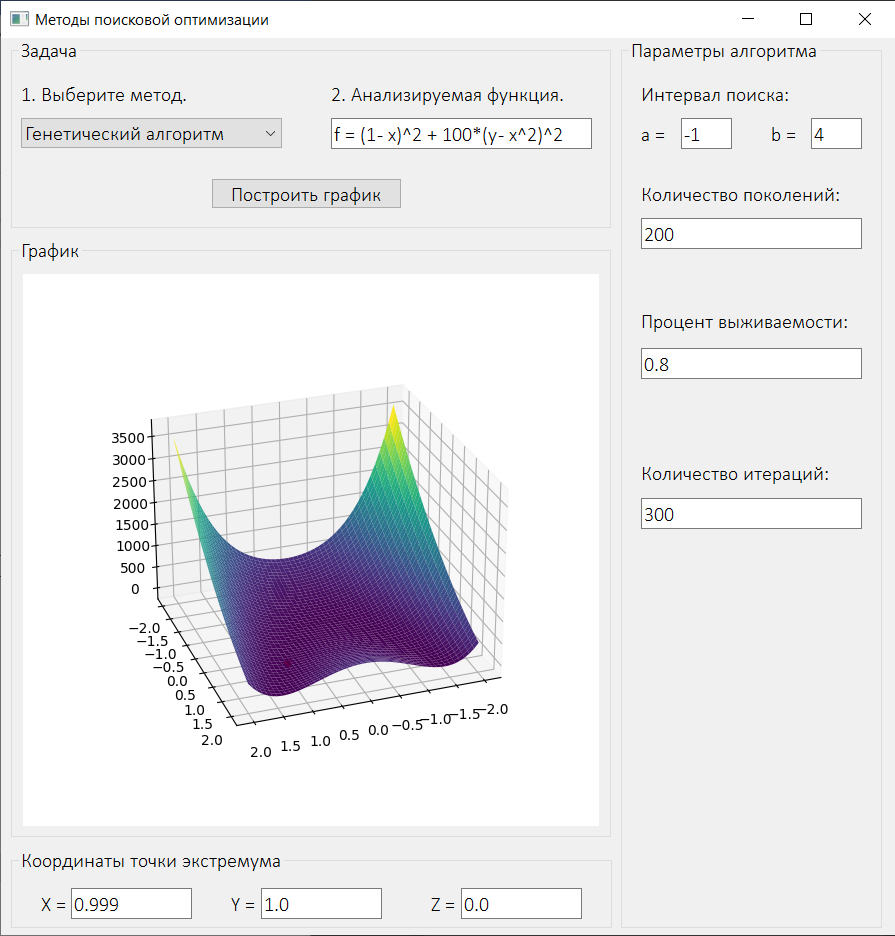


Рисунок 2 – Вид программы при выборе метода градиентного спуска с постоянным шагом

# Листинг

import math

import numpy as np

import random

class Chromosome:

def \_\_init\_\_(self, value):

self.value = value

class Population:

def \_\_init\_\_(self):

self.chromosomes = None

def to\_list(self):

list\_population = []

for i in self.chromosomes:

list\_population.append(i.value)

return list\_population

def min(self):

min = self.chromosomes[0].value

for i in self.chromosomes:

if min > i.value:

min = i.value

return min

def print(self):

s = ""

for i in self.chromosomes:

s += str(i.value) + " "

print(s)

class GeneticAlgorithm:

def \_\_init\_\_(self, f, a, b, number\_generations, percent\_of\_best\_ones\_to\_live):

self.a = a

self.b = b

self.population\_x = None

self.population\_y = None

self.number\_generations = number\_generations

self.probability = 0.00001

self.percent\_of\_best\_ones\_to\_live = percent\_of\_best\_ones\_to\_live

self.f = f

def generation\_initial\_population(self):

self.population\_x = Population()

self.population\_x.chromosomes = self.create\_new\_population()

self.population\_y = Population()

self.population\_y.chromosomes = self.create\_new\_population()

def create\_new\_population(self):

return [Chromosome(random.uniform(self.a, self.b)) for i in range(self.number\_generations)]

def get\_best\_members(self):

function\_values = []

population\_x\_length = len(self.population\_x.chromosomes)

for i in range(population\_x\_length):

function\_values.append(self.f([self.population\_x.chromosomes[i].value, self.population\_y.chromosomes[i].value]))

self.Hoare\_sorting(function\_values, self.population\_x, self.population\_y)

amount\_of\_best\_values = round(population\_x\_length \* self.percent\_of\_best\_ones\_to\_live)

return self.population\_x.chromosomes[0:amount\_of\_best\_values], self.population\_y.chromosomes[0:amount\_of\_best\_values]

def crossing\_over(self):

population\_x\_length = len(self.population\_x.chromosomes)

for i in range(population\_x\_length, self.number\_generations):

self.population\_x.chromosomes.append(Chromosome((self.population\_x.chromosomes[random.randint(0, population\_x\_length - 1)].value + self.population\_x.chromosomes[random.randint(0, population\_x\_length - 1)].value) // 2))

self.population\_y.chromosomes.append(Chromosome((self.population\_y.chromosomes[random.randint(0, population\_x\_length - 1)].value + self.population\_y.chromosomes[random.randint(0, population\_x\_length - 1)].value) // 2))

def mutate(self):

population\_x\_length = len(self.population\_x.chromosomes)

min\_population\_x = self.population\_x.min()

min\_population\_y = self.population\_y.min()

for i in range(population\_x\_length):

self.population\_x.chromosomes[i].value += min\_population\_x \* random.uniform(0, self.probability)

for i in range(population\_x\_length):

self.population\_y.chromosomes[i].value += min\_population\_y \* random.uniform(0, self.probability)

def get\_min\_value\_index(self):

function\_values = []

population\_x\_length = len(self.population\_x.chromosomes)

for i in range(population\_x\_length):

function\_values.append(self.f([self.population\_x.chromosomes[i].value, self.population\_y.chromosomes[i].value]))

return function\_values.index(np.min(function\_values))

def get\_arguments\_of\_min\_value(self):

minimum\_value\_index = self.get\_min\_value\_index()

return self.population\_x.chromosomes[minimum\_value\_index].value, self.population\_y.chromosomes[minimum\_value\_index].value

def search\_minimum(self, iterations):

# population = []

self.generation\_initial\_population()

for i in range(iterations):

self.population\_x.chromosomes, self.population\_y.chromosomes = self.get\_best\_members()

self.crossing\_over()

self.mutate()

min\_point\_x, min\_point\_y = self.get\_arguments\_of\_min\_value()

return min\_point\_x, min\_point\_y, self.f([min\_point\_x, min\_point\_y])

@staticmethod

def Hoare\_sorting(function\_values, population\_x, population\_y):

left = 0

right = len(function\_values) - 1

l2 = None

r2 = None

pivot\_value = None

lows = []

highs = []

lows.append(left)

highs.append(right)

while lows:

left = lows.pop()

right = highs.pop()

l2 = left

r2 = right

pivot\_value = function\_values[(left + right) // 2]

while True:

while function\_values[l2] < pivot\_value:

l2 += 1

while function\_values[r2] > pivot\_value:

r2 -= 1

if l2 <= r2:

if function\_values[l2] > function\_values[r2]:

function\_values[l2], function\_values[r2] = function\_values[r2], function\_values[l2]

population\_x.chromosomes[l2].value, population\_x.chromosomes[r2].value = population\_x.chromosomes[r2].value, population\_x.chromosomes[l2].value

population\_y.chromosomes[l2].value, population\_y.chromosomes[r2].value = population\_y.chromosomes[r2].value, population\_y.chromosomes[l2].value

l2 += 1

if r2 > 0:

r2 -= 1

if l2 > r2:

break

if l2 < right:

lows.append(l2)

highs.append(right)

if r2 > left:

lows.append(left)

highs.append(r2)

def from\_range\_to\_range(x, low1, hight1, low2, hight2):

return low2 + (hight2 - low2) \* (x - low1) / (hight1 - low1)

# f = Function()

# ga = GeneticAlgorithm(f, -1, 4, 100, 0.8)

# print("Rosenbrok Function: ")

# min = ga.search\_minimum(100)

# print("Min: " + str(min[0]) + " " + str(min[1]) + " " + str(min[2]))