Санкт-Петербургский государственный университет

Факультет прикладной математики – процессов управления

Лабораторная работа №4

**Отчет**

по дисциплине «Алгоритмы и структуры данных»

Генетический алгоритм

Вариант 4

Автор работы: Добренкова Л.С.

Группа: 22.Б15-пу

Преподаватель: Дик А.Г.

Санкт-Петербург, 2023

Оглавление

[1. Введение 3](#__RefHeading___Toc1536_845584960)

[2. Цель работы 3](#__RefHeading___Toc1538_845584960)

[3. Теоретическая часть 3](#__RefHeading___Toc1540_845584960)

[4. Модификация алгоритма 4](#__RefHeading___Toc1542_845584960)

[5. Представление и спецификация программной части 4](#__RefHeading___Toc1544_845584960)

[6. Контрольный пример 7](#__RefHeading___Toc1546_845584960)

[7. Тестирование и анализ результатов работы алгоритма 8](#__RefHeading___Toc1548_845584960)

[8. Сравнение кодировок 10](#__RefHeading___Toc1550_845584960)

[9. Выводы 11](#__RefHeading___Toc1552_845584960)

[10. Список литературы 12](#__RefHeading___Toc1554_845584960)

[11. Листинг 12](#__RefHeading___Toc1556_845584960)

# Введение

Генетический алгоритм (ГА) представляет собой эволюционный метод оптимизации, вдохновленный процессами естественного отбора и генетики. Разработанный для моделирования механизмов эволюции, ГА оперирует с популяцией индивидов, представляющих решения задачи оптимизации. В процессе эволюции, эти индивиды подвергаются генетическим операторам, таким как мутация и кроссовер, позволяя новым поколениям принимать черты успешных предыдущих поколений. Генетические алгоритмы успешно применяются в различных областях, включая оптимизацию параметров, обучение машин и другие задачи поиска глобальных экстремумов.

# Цель работы

Исследование особенностей генетических алгоритмов для решения задач глобальной оптимизации.

# Теоретическая часть

Генетический алгоритм (GA) представляет собой оптимизационный метод. В рамках GA, каждый "ген" в пространстве поиска представляет собой потенциальное решение задачи оптимизации. Применяется для нахождения приближенных решений задачи оптимизации. В контексте генетического алгоритма используются следующие ключевые термины:

* **Ген (Gene):** Отдельный элемент в генетическом коде, представляющий параметр или характеристику потенциального решения задачи оптимизации.
* **Функция приспособленности (Fitness Function):** Оценочная функция, измеряющая качество решения в данной точке пространства параметров.
* **Мутация (Mutation):** Процесс изменения генетического кода для внесения случайных изменений и разнообразия в популяцию.
* **Скрещивание (Crossover):** Процесс комбинирования генетического материала от двух родительских хромосом для создания потомства.
* **Селекция (Selection):** Процесс выбора индивидов для участия в процессе мутации и скрещивания на основе их функции приспособленности.
* **Итерации (Generations):** Шаги оптимизационного процесса, на каждом из которых происходит обновление популяции.

Основные шаги алгоритма:

1. **Инициализация:** 
   * Генерация генов с случайными положениям.
2. **Основной цикл оптимизации:**
   * Мутация некоторого количества генов
   * Скрещивание генов исходной популяции.
   * Селекция генов
3. **Повторение:**
   * Повторение шага 2 заданное количество раз (генераций).
4. **Вывод результатов:**
   * Вывод результатов

# Модификация алгоритма

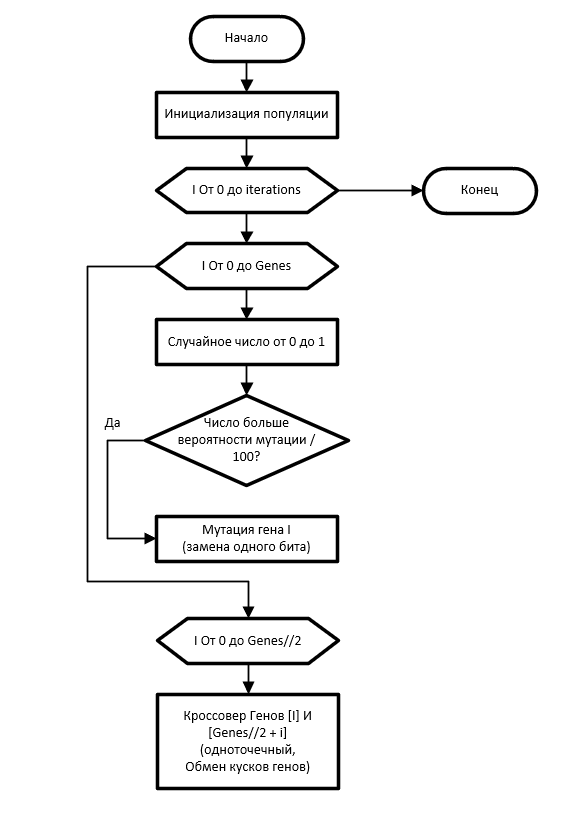
В данной работе реализована модификация алгоритма селекции, а именно – стратегия элитарной селекции в сочетании с частичной заменой популяции. Суть данной стратегии заключается в том, что 10% лучших особей предыдущего поколения переходят в следующее поколение, причем переходят без изменений и вне зависимости от того, насколько они лучше (или хуже) удовлетворяют условию оптимизации. То есть, может возникнуть ситуация, при которой эти 10% генов будут хуже всех генов из нового поколения, но все равно окажутся в нем. Эта модификация способствует разнообразию популяции.

# Представление и спецификация программной части

Программная реализация выполнена на языке Python3.11 с использованием библиотеки PyQT для визуализации алгоритма и некоторых встроенных библиотек. Нельзя не отметить, что программа выполнена в функциональном стиле и не содержит классов (за исключением GUI); вся логика программы заключена в следующих функциях:

1. **Encode** :Кодирует кортеж из двух вещественных чисел в бинарное представление
2. **decode** :Декодирует бинарное представление вещественных чисел в кортеж из двух вещественных чисел.
3. **initialize\_genes** Инициализирует список генетических индивидуумов случайными значениями в пределах заданных границ.
4. **Func** Вычисляет значение математической функции, заданной строкой f, для входных значений x1 и x2.
5. **Mutate** Производит мутацию генетического индивида.
6. **Crossover** Производит кроссовер между двумя генами. Отбор генов идет в функции algorithm.
7. **Algorithm** Выполняет генетический алгоритм для оптимизации функции.

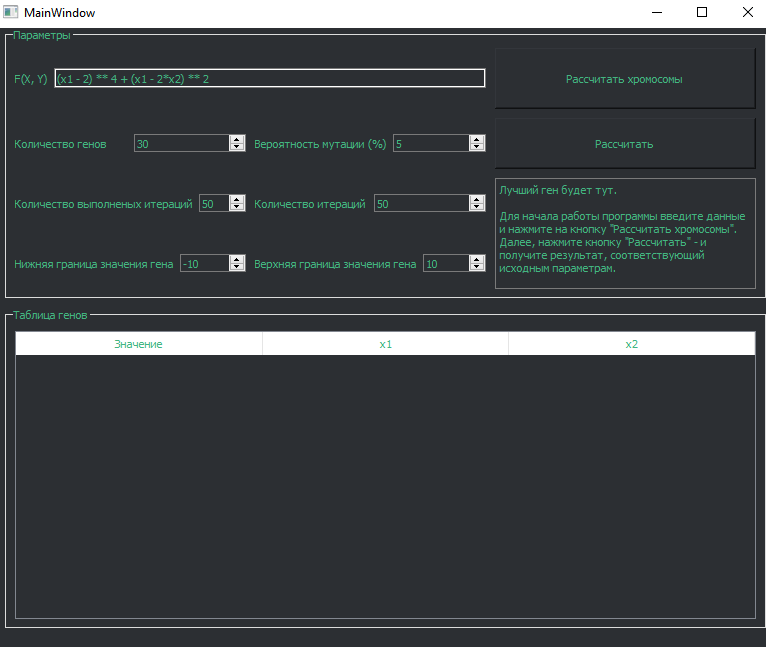
На рисунке 5.1 представлена блок-схема алгоритма:



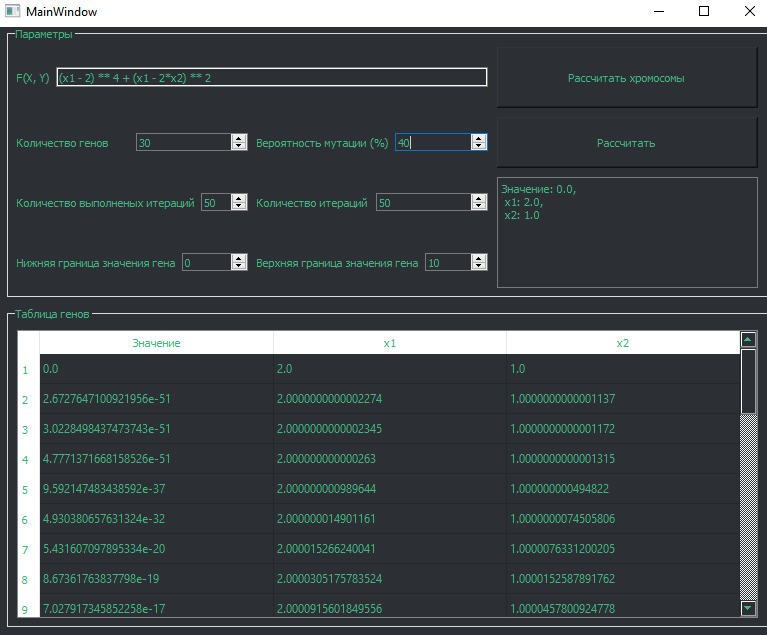
*Блок-схема 5.1 Блок-схема алгоритма*

# Контрольный пример

При запуске программы пользователь увидит MainWindow с предустановленными параметрами. В одном из окон он может наблюдать инструкцию по использованию программы.



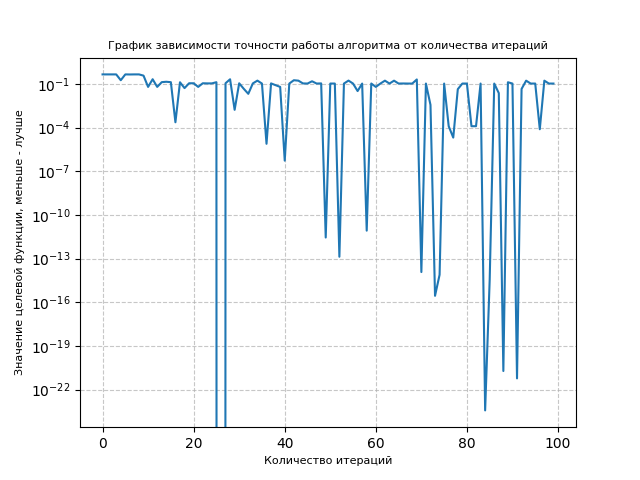
*Рис. 6.1 Main Window при запуске программы*



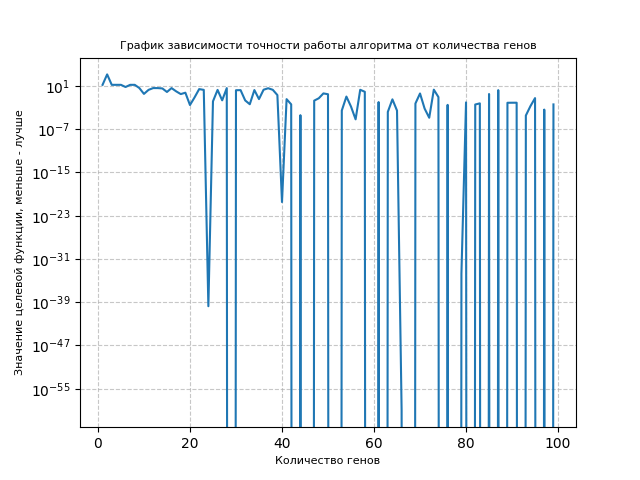
*Рис. 6.2 Результат работы программы*

# Тестирование и анализ результатов работы алгоритма

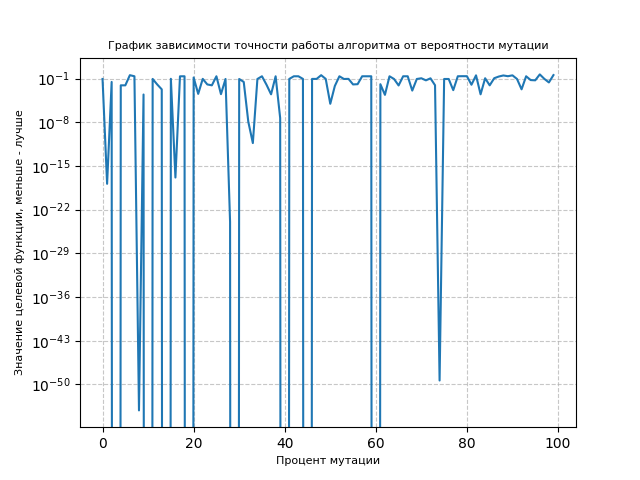
На рисунках 7.1-7.2 представлены результаты тестирования программы. Начальные данные: область [0, 10], функция с минимумом, равным 0, в точке (2, 1). Количество генов в графике 7.1 – 30, количество итераций в графике 7.2– 50. Количество генов и итераций в графике 7.3 – 30 и 50 соответсвтенно.



*Рис. 7.1 График зависимости  
 точности результатов алгоритма от количества итераций.*



*Рис. 7.2 График зависимости  
 точности результатов алгоритма от количества частиц.*



*Рис. 7.3 График зависимости  
 точности результатов алгоритма от вероятности мутации.*

Как видно из рисунков 7.1-7.2, увеличение количества частиц и итераций не гарантирует повышения точности из-за стохастической природы GA; при определенном везении даже при относительно малых значениях параметров возможно получение результата с абсолютной точностью (см. впадины на графиках).

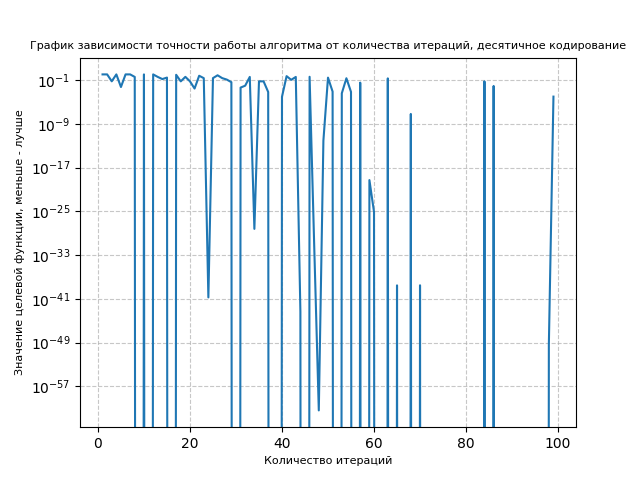
Как видно из рисунка 7.3, вероятность мутации не оказывает существенного влияния на сходимость и/или на её точность. Важно скорее то, что мутация ввиду потенциально малого изменения гена (1 бит, может отвечать за значения >10^10) дает алгоритму сходится практически без погрешности.

К сожалению, при увеличении этих параметров в одном конкретном запуске не гарантируется нахождение верного решения – это видно из того, что значения целевой функции все же продолжают колебаться около 10^-1 во всех трех графиках.

Выбор оптимальных значений исходя из полученных данных не может быть единственно верным даже для одной конкретной ситуации, но для определенности – процент мутации 30, количество генов – 30, количество итераций – 30.

# Сравнение кодировок

До этого в отчете рассматривалась бинарная кодировка гена, в этом пункте покажем её сравнение с десятичной. На графике 8.1 показана зависимость точности алгоритма от количества итераций при десятичной кодировке гена.



*Рис. 8.1 График зависимости  
 точности результатов алгоритма от итерации, десятичная кодировка.*

Сравнивая графики 8.1 и 7.3, видно, что десятичная кодировка сходится чаще, нежели двоичная. Впрочем, это верно только для конкретной популяции, и прочих входных данных.

Подведя итог, можно сделать вывод, что для данной имплементации алгоритма кодировка особой роли не играет. Но отмечу, что это верно только при схожих входных данных (область, функция и т. д.)

# Выводы

Были получены все нужные навыки для реализации генетических алгоритмов, а также реализован оный, с интерфейсом к нему. Проанализированы результаты работы алгоритма, выполнено сравнение кодировок.

# Список литературы

[1] Статья про генетические алгоритмы <https://habr.com/ru/articles/128704/>

# Листинг

import random  
import struct  
from copy import deepcopy  
from typing import List, Tuple  
  
def encode(x: Tuple[float, float]) -> Tuple[str, str]:  
 *"""  
 Encodes a tuple of float values into a binary representation.  
  
 Parameters:  
 - x: Tuple of two float values to be encoded.  
  
 Returns:  
 Tuple of two binary strings.  
 """* binary\_representation = (  
 format(struct.unpack('>Q', struct.pack('>d', x[0]))[0], '064b'),  
 format(struct.unpack('>Q', struct.pack('>d', x[1]))[0], '064b')  
 )  
 return binary\_representation  
  
def decode(binary\_str: Tuple[str, str]) -> Tuple[float, float]:  
 *"""  
 Decodes a tuple of binary strings into a tuple of float values.  
  
 Parameters:  
 - binary\_str: Tuple of two binary strings.  
  
 Returns:  
 Tuple of two float values.  
 """* float\_value1 = struct.unpack('>d', int(binary\_str[0], 2).to\_bytes(8, 'big'))[0]  
 float\_value2 = struct.unpack('>d', int(binary\_str[1], 2).to\_bytes(8, 'big'))[0]  
 return float\_value1, float\_value2  
  
def initialize\_genes(bounds: Tuple[float, float], amount: int) -> List[Tuple[float, float]]:  
 *"""  
 Initializes a list of genetic individuals.  
  
 Parameters:  
 - bounds: Tuple representing the lower and upper bounds for genetic values.  
 - amount: Number of genetic individuals to generate.  
  
 Returns:  
 List of tuples representing genetic individuals.  
 """* genes = []  
 for \_ in range(amount):  
 genes.append((random.uniform(bounds[0], bounds[1]), random.uniform(bounds[0], bounds[1])))  
 return genes  
  
def func(f: str, x1: float, x2: float) -> float:  
 *"""  
 Evaluates a mathematical function.  
  
 Parameters:  
 - f: Mathematical expression as a string.  
 - x1: First input value.  
 - x2: Second input value.  
  
 Returns:  
 Result of the mathematical expression.  
 """* return eval(f)  
  
def mutate(gene: Tuple[float, float], bounds: Tuple[float, float]) -> Tuple[float, float]:  
 *"""  
 Applies mutation to a genetic individual.  
  
 Parameters:  
 - gene: Tuple representing a genetic individual.  
 - bounds: Tuple representing the lower and upper bounds for genetic values.  
  
 Returns:  
 Mutated genetic individual.  
 """* mutated\_gene = list(encode(gene))  
 bit\_position = random.randint(0, 63)  
  
 for i in range(len(mutated\_gene)):  
 binary\_list = list(mutated\_gene[i])  
 binary\_list[bit\_position] = '1' if binary\_list[bit\_position] == '0' else '0'  
 mutated\_gene[i] = ''.join(binary\_list)  
  
 mutated\_gene = (  
 max(bounds[0], min(bounds[1], decode(mutated\_gene)[0])),  
 max(bounds[0], min(bounds[1], decode(mutated\_gene)[1]))  
 )  
 return mutated\_gene  
  
def crossover(gene1: Tuple[float, float], gene2: Tuple[float, float], bounds: Tuple[float, float]) -> Tuple[Tuple[float, float], Tuple[float, float]]:  
 *"""  
 Applies crossover between two genetic individuals.  
  
 Parameters:  
 - gene1: First genetic individual.  
 - gene2: Second genetic individual.  
 - bounds: Tuple representing the lower and upper bounds for genetic values.  
  
 Returns:  
 Tuple of two new genetic individuals after crossover.  
 """* gene1 = encode(gene1)  
 gene2 = encode(gene2)  
  
 crossover\_point = random.randint(2, 12)  
 new\_gene1 = (  
 gene1[0][:crossover\_point] + gene2[0][crossover\_point:],  
 gene1[1][:crossover\_point] + gene2[1][crossover\_point:]  
 )  
 new\_gene2 = (  
 gene2[0][:crossover\_point] + gene1[0][crossover\_point:],  
 gene2[1][:crossover\_point] + gene1[1][crossover\_point:]  
 )  
 gene1 = (  
 max(bounds[0], min(bounds[1], decode(new\_gene1)[0])),  
 max(bounds[0], min(bounds[1], decode(new\_gene1)[1]))  
 )  
 gene2 = (  
 max(bounds[0], min(bounds[1], decode(new\_gene2)[0])),  
 max(bounds[0], min(bounds[1], decode(new\_gene2)[1]))  
 )  
 return gene1, gene2  
  
def algorithm(genes: List[Tuple[float, float]], generations: int, f: str, mutation: float, bounds: Tuple[float, float]) -> List[List[Tuple[float, float]]]:  
 *"""  
 Executes a genetic algorithm.  
  
 Parameters:  
 - genes: List of genetic individuals.  
 - generations: Number of generations for the algorithm.  
 - f: Mathematical expression as a string.  
 - mutation: Mutation rate as a percentage.  
 - bounds: Tuple representing the lower and upper bounds for genetic values.  
  
 Returns:  
 List containing the history of genetic individuals over generations.  
 """* history = [deepcopy(sorted(genes, key=lambda x: func(f, \*x)))]  
 for \_ in range(generations):  
 for i in range(len(genes)):  
 if random.uniform(0, 1) > mutation / 100:  
 genes[i] = mutate(genes[i], bounds)  
  
 for i in range(0, len(genes) - len(genes) % 2, 2):  
 genes[i], genes[i + 1] = crossover(genes[i], genes[i + 1], bounds)  
  
 genes = sorted(genes, key=lambda x: func(f, \*x))[:len(genes) // 10 \* 9 + len(genes) % 10] + history[-1][:len(genes) // 10]  
  
 genes = sorted(genes, key=lambda x: func(f, \*x))  
 history.append(deepcopy(genes))  
 return history