МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| профессор |  |  |  | Скобцов Ю. А. |
| должность, звание |  | подпись, дата |  | фамилия, инициалы |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1 |
| «Простой генетический алгоритм» |
| по дисциплине: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4936 |  |  |  | Нестеренко М.Ю. |
|  |  |  | подпись, дата |  | фамилия, инициалы |

Санкт-Петербург 2022

**Цель работы**

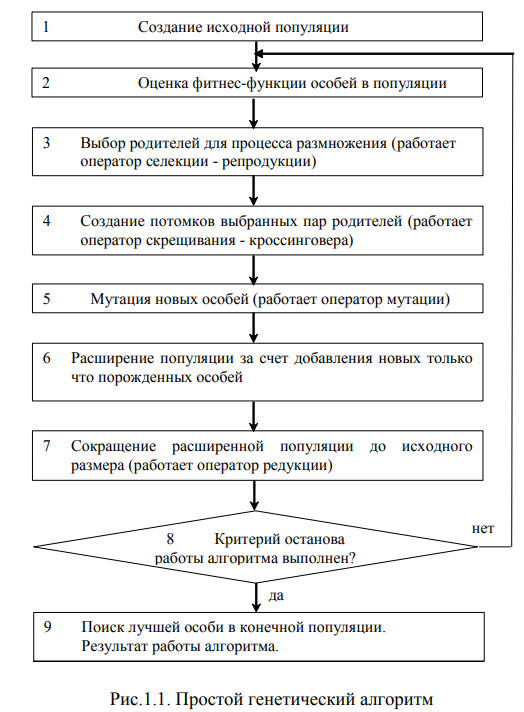
Разработать простой генетический алгоритм на основе теоретических данных методического пособия.

**Вариант задания 14**

x ∈ [-10,0),(0,10] , min

**Теоретические сведения**

Генетический алгоритм представляет собой следующую схему:



Далее, в зависимости от потребностей и сложности алгоритма, схема может меняться

На основе схемы можно выделить следующие основные этапы:

1. Создание начальной популяции;
2. Отбор;
3. Скрещивание;
4. Мутация;
5. Проверка достижения целей.

Пункт 2-5 повторяются до тех пор, пока не будет выполнена условие остановки эволюционного алгоритма.

Отбор производится на основе турнира с выборкой из трех особей текущей популяции.

Скрещивания и впоследствии мутация особей случается с вероятность Pc и Pm соответственно, вероятности задаются в начале эксперимента. Как правило, Pc = 0.5,Pm = 0.001.

После завершения эволюционного алгоритма из производной популяции выбирается наиболее приспособленная особь (решение) на основе фитнесс (целевой) функции.

**Программа и результат ее выполнения**

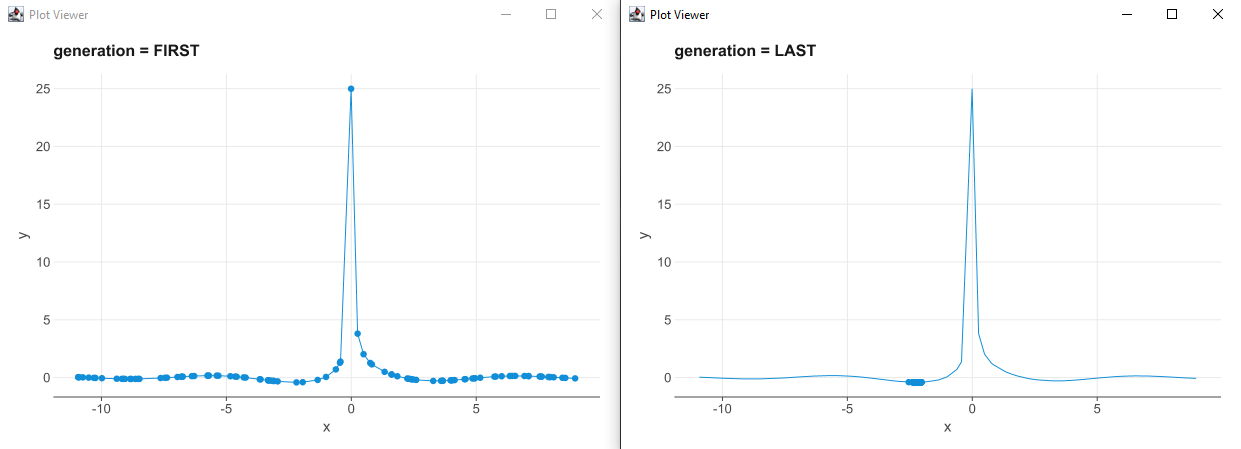
Код программы написан на языке программирования Kotlin с применением библиотеки lets-plot-kotlin для построения графиков.

import org.jetbrains.letsPlot.geom.\*  
import org.jetbrains.letsPlot.label.ggtitle  
import org.jetbrains.letsPlot.letsPlot  
import kotlin.math.\*  
import kotlin.random.Random  
  
  
const val *VARIANT* = "cos(x - 0.5) / |x|"  
const val *IS\_MAX\_SEARCHING* = false  
const val *FROM* = -10  
const val *TO* = 10 *// [-10,0),(0,10], min*const val *P\_M* = 0.001  
const val *P\_C* = 0.5  
const val *TOURNAMENT\_COUNT* = 3  
const val *INIT\_POPULATION\_COUNT* = 100  
const val *MAX\_GENERATION\_COUNT* = 100  
const val *BIT\_COUNT* = 15  
const val *MIN\_FITNESS* = -0.411*//009*const val *MAX\_FITNESS* = 1000.1 *// в этом варианте максимума нет \_/\\_ улетает в небо*val *EXTREME\_FITNESS* = if (*IS\_MAX\_SEARCHING*) *MAX\_FITNESS* else *MIN\_FITNESS*val *population* = *mutableListOf*<String>()  
val *clone* = *mutableListOf*<String>()  
  
fun main() {  
 val begin: Long = System.currentTimeMillis()  
  
 *println*("VARIANT #14:\n$*VARIANT*\n")  
 *population*.*apply* **{** *initPopulation*()  
 *draw*(0)  
 *println*("max fitness before = $*maxFitness*")  
 *println*("min fitness before = $*minFitness*")  
 *evolution*()  
 *println*("max fitness after = $*maxFitness*")  
 *println*("min fitness after = $*minFitness*")  
 *draw*(*MAX\_GENERATION\_COUNT*)  
 **}** val end: Long = System.currentTimeMillis()  
 *println*("Time spent: ${ end - begin } ms")  
}  
  
fun MutableList<String>.draw(generation: Int) {  
  
 val xs = *List*(this.size) **{** this[**it**].*real* **}** val xs2 = *List*(this.size) **{** *clone*[**it**].*real* **}** val ys = *List*(this.size) **{** if (this[**it**].*fitness* <= 25) this[**it**].*fitness* else 25 **}** val ys2 = *List*(this.size) **{** if (*clone*[**it**].*fitness* <= 25) *clone*[**it**].*fitness* else 25 **}** val data = *mapOf* ("x" *to* xs, "y" *to* ys, "fx" *to* xs2, "fy" *to* ys2)  
  
 val p = *letsPlot*(data) +  
 geomPoint(size = 3) **{** x = "x"; y = "y" **}** +  
 geomLine **{** x = "fx"; y = "fy" **}** +  
 *ggtitle*("generation = ${ if (generation == *MAX\_GENERATION\_COUNT*) "LAST" else if (generation == 0) "FIRST" else generation}")  
 p.show()  
}  
  
fun MutableList<String>.initPopulation() {  
 *repeat*(*INIT\_POPULATION\_COUNT*) **{** var chromosome = ""  
 *repeat*(*BIT\_COUNT*) **{** chromosome += (0..1).*random*() **}** this.add(chromosome)  
 *clone*.add(chromosome)  
 **}**}  
  
fun MutableList<String>.evolution() {  
 var generationCounter = 0  
 while (generationCounter <= *MAX\_GENERATION\_COUNT* && *endCondition*) {  
 val offspring = *selection*().*apply* **{** *crossing*()  
 *mutation*()  
 **}** *population*.clear()  
 *population*.addAll(offspring)  
 if (generationCounter == *MAX\_GENERATION\_COUNT* / 2) *draw*(generationCounter)  
 generationCounter++  
 }  
 *println*("GENERATION COUNT = $generationCounter")  
}  
  
fun MutableList<String>.selection(): MutableList<String> {  
 val newPopulation = *mutableListOf*<String>()  
 *repeat*(*INIT\_POPULATION\_COUNT*) **{** var ies = *List*(*TOURNAMENT\_COUNT*) **{** (0 *until INIT\_POPULATION\_COUNT*).*random*() **}** while (ies.size != ies.*distinct*().size) {  
 ies = *List*(*TOURNAMENT\_COUNT*) **{** (0 *until INIT\_POPULATION\_COUNT*).*random*() **}** }  
 val tournamentIndividuals = *MutableList*(*TOURNAMENT\_COUNT*) **{** this[ies[**it**]] **}** val index = *indexOfFirst* **{ it**.*fitness* == tournamentIndividuals.*extremeFitness* **}** newPopulation.add(this[index])  
 **}** return newPopulation  
}  
  
fun MutableList<String>.crossing() {  
 for (i in 0 .. *INIT\_POPULATION\_COUNT step* 2) {  
 if (Random.nextDouble() <= *P\_C* && (i + 1) < *INIT\_POPULATION\_COUNT*) {  
 val k = (0 *until BIT\_COUNT*).*random*()  
 val crossedA = this[i].*substring*(0, k) + this[i + 1].*substring*(k, *BIT\_COUNT*)  
 val crossedB = this[i + 1].*substring*(0, k) + this[i].*substring*(k, *BIT\_COUNT*)  
  
 this[i] = crossedA  
 this[i + 1] = crossedB  
 }  
 }  
}  
  
fun MutableList<String>.mutation() {  
 *repeat*(*INIT\_POPULATION\_COUNT*) **{** if (Random.nextDouble() <= *P\_M*) {  
 val k = (0 *until BIT\_COUNT*).*random*()  
 val chromosome = this[**it**]  
 val mutatedGen = if (this[**it**][k] == '0') '1' else '0'  
 this[**it**] = chromosome.*substring*(0, k) + mutatedGen + chromosome.*substring*(k + 1, chromosome.length)  
 }  
 **}**}  
  
fun MutableList<String>.print() = *println*(this)  
val MutableList<String>.*extremeFitness* get () = if (*IS\_MAX\_SEARCHING*) *maxFitness* else *minFitness*val MutableList<String>.*maxFitness* get () = this.*maxOf* **{ it**.*fitness* **}**val MutableList<String>.*minFitness* get () = this.*minOf* **{ it**.*fitness* **}**val MutableList<String>.*endCondition* get () = if (*IS\_MAX\_SEARCHING*) *extremeFitness* <= *EXTREME\_FITNESS* else *extremeFitness* >= *EXTREME\_FITNESS*fun Int.toBinary() = *toString*(radix = 2)  
fun String.fromBinary() = *toInt*(radix = 2)  
val String.*real*: Double  
 get() = *FROM* + *fromBinary*() \* (*TO* - *FROM*) / (2.0.*pow*(*BIT\_COUNT*.toDouble()) - 1.0)  
  
val String.*fitness*: Double  
 get() =  
 if (*abs*(*real*) == 0.0)  
 if (*IS\_MAX\_SEARCHING*)  
 Int.MIN\_VALUE.toDouble()  
 else  
 Int.MAX\_VALUE.toDouble()  
 else  
 *cos*(*real* - 0.5) / *abs*(*real*)  
  
fun String.printData() = *println*("real = ${*real*}\nfitness = ${*fitness*}\n")

Левый график – изначально сформированная популяция

Правый – популяция, содержащая особь с максимальным значением целевой функции варианта.

№1



Pc = 0.5

Pm = 0.001

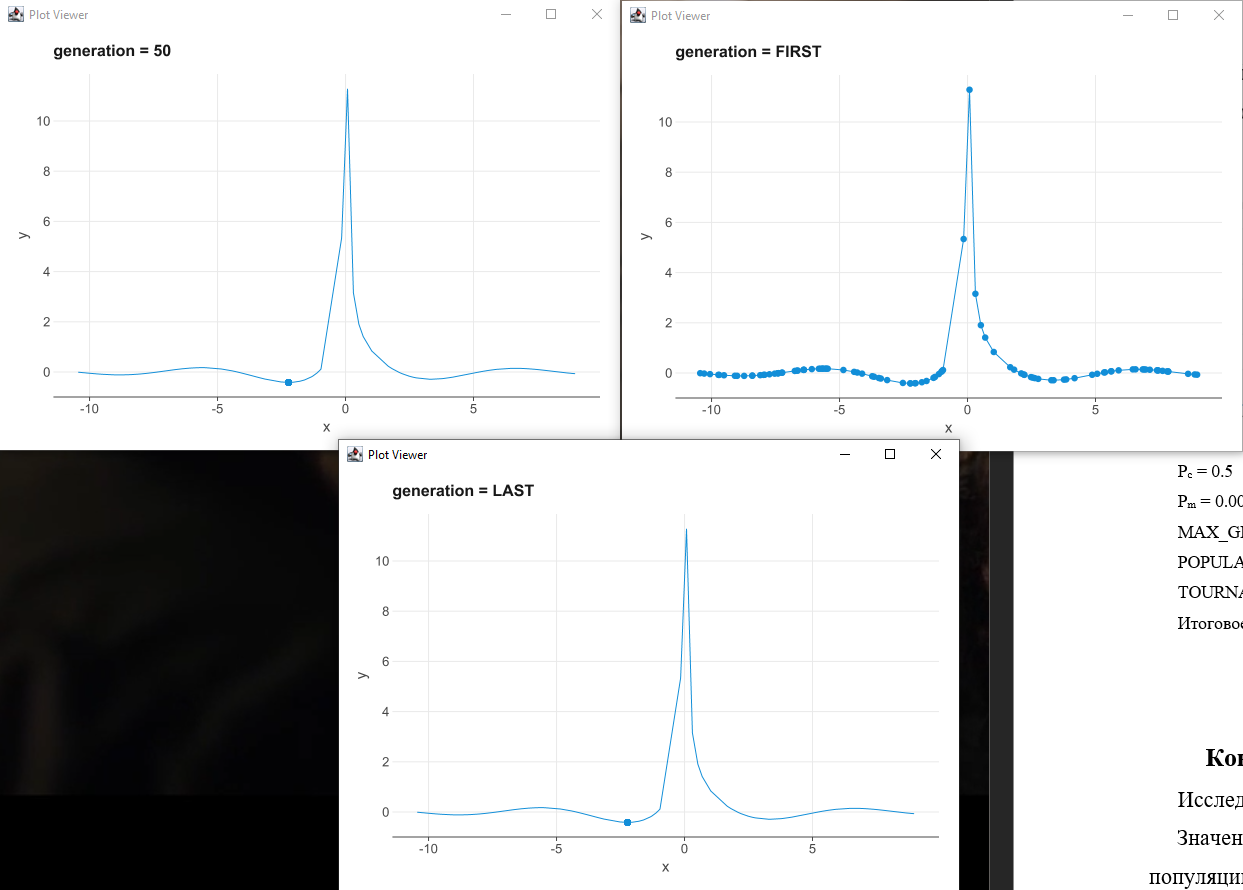
MAX\_GENERATIONS = 100

POPULATION = 100

TOURNAMENT = 3

Итоговое количество поколений: 7

№2



Pc = 0.5

Pm = 0.001

MAX\_GENERATIONS = 100

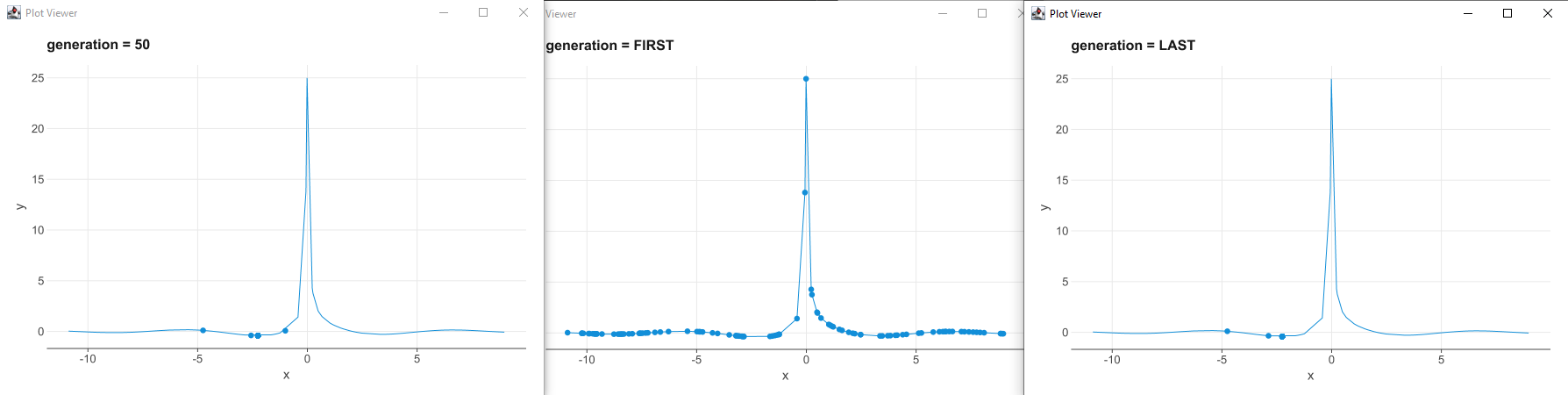
POPULATION = 100

TOURNAMENT = 3

Итоговое количество поколений: 100

Генетический алгоритм не смог найти минимальное значение, но максимально приблизился к нему.

№3



Pc = 0.5

Pm = 0.1

MAX\_GENERATIONS = 100

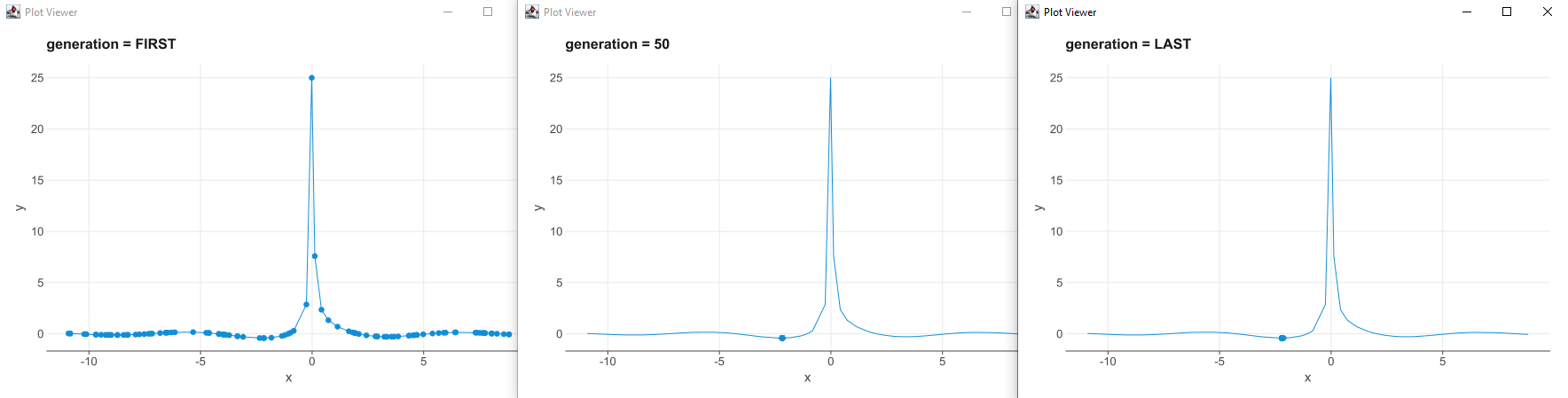
POPULATION = 100

TOURNAMENT = 3

Итоговое количество поколений: 100

Генетический алгоритм не смог найти минимальное значение, но максимально приблизился к нему. Слишком частые мутации не дали устояться геному и соответственно найти экстремум.

№4



Pc = 0.05

Pm = 0.001

MAX\_GENERATIONS = 100

POPULATION = 100

TOURNAMENT = 3

Итоговое количество поколений: 100

Генетический алгоритм не смог найти минимальное значение, но максимально приблизился к нему. Недостаточный обмен генами не дал достигнуть экстремума.

**Контрольный вопрос**

Исследуйте зависимость работы ПГА от значения вероятности ОК Pc.

Значение вероятности скрещивания Pc влияет на то, как часто особи в популяции будут обмениваться генами, т.е. как быстро отобранные особи передадут свои полезные гены следующему поколению

Поскольку геном застаивается, эволюция происходит медленнее и оптимальное решение находится дольше или прерывается из-за превышения количества поколений.

|  |  |  |  |
| --- | --- | --- | --- |
| Количество поколений | | | |
| Pc = 1 | Pc = 0,5 | Pc = 0,1 | Pc = 0,001 |
| 4 | 3 | 100 | 100 |
| 5 | 2 | 100 | 100 |
| 1 | 100 | 100 | 100 |
| 2 | 6 | 100 | 100 |
| 7 | 100 | 100 | 100 |
| 3 | 100 | 100 | 100 |
| 3 | 7 | 4 | 15 |

KПср(1) = 3,57

KПср(0,5) = 45,42

KПср(0,1) = 86,29

KПср(0,001) = 87,86

**Вывод**

В ходе выполнения лабораторной работы были получены основные навыки создания простого эволюционного алгоритма и его применения для вычисления точки приближенной к экстремуму математической функции.