БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет КСИС

Специальность ИиТП

Контрольная работа №1

по дисциплине «Математическое моделирование»

Выполнил студент: Драгун О.В.

группа 893551

Зачетная книжка № 2520050

Минск 2022

# Введение

Эту и последующие работы будем писать на языке программирования Kotlin, в связке с библиотекой lets-plot-kotlin для построения гистограмм и графиков.

<https://blog.jetbrains.com/kotlin/2020/12/lets-plot-in-kotlin/>

Для запуска используется IntelliJ IDEA Community Edition 2022.2.3

# Имитация случайного события

1) Напишем датчик случайных чисел такой, чтобы результат содержал СЧ X, заданную датчику вероятность Pa и результат вычислений true (событие A произошло)/false (произошло !A). Хорошо протестировать этот датчик важно, так как на нем строятся дальнейшие имитации.  
Результат однократного запуска выведем в консоль  
Изображение выглядит как текст

Автоматически созданное описание

2) Теперь покроем датчик тестами(можно найти в листинге), в которых проверим  
а) При 100000 запусков кол-во значений для 10 интервалов с шагом 0.1 находятся между 9001 и 10999

б) Ни один результат не равен 0.0, ни один результат не равен 1.0

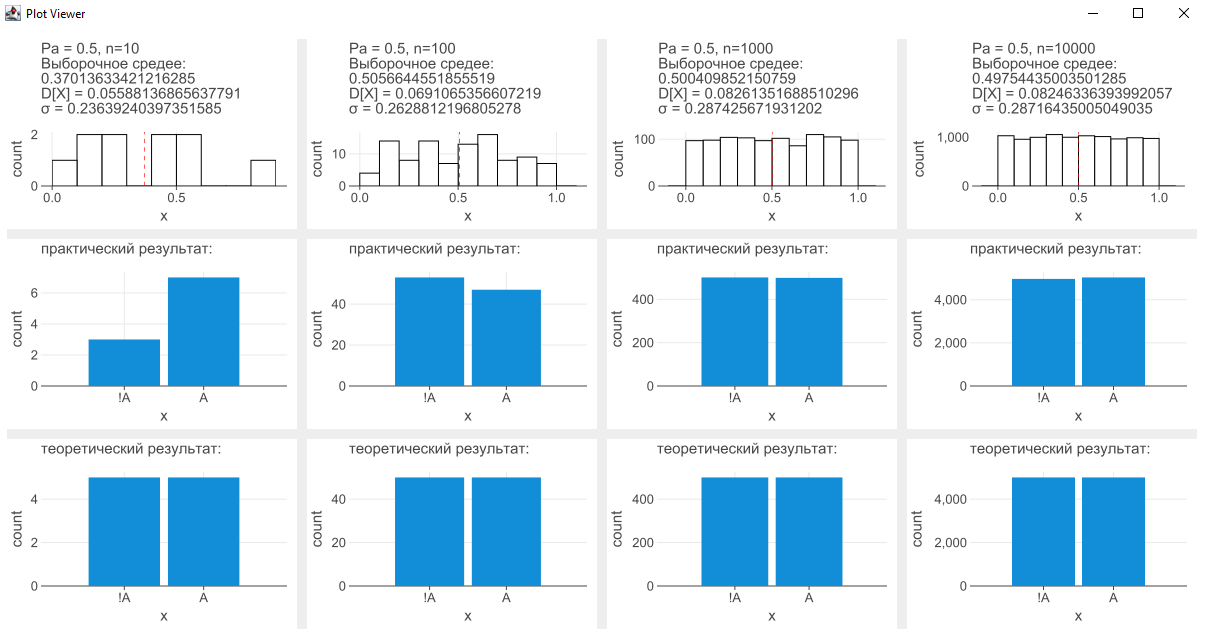
в) А также другие тесты

Изображение выглядит как текст

Автоматически созданное описание

3) Теперь проведем серию экспериментов с разным кол-вом случайных величин. Для каждой серии посчитаем оценки и предположим три гипотезы:  
а) Выборочное среднее при увеличении числа экспериментов будет приближаться к 0.5

б) Высота столбцов для интервалов с шагом 0.1 будет сравниваться с увеличением выборки  
в) Фактическая частота событий A и !A будет приближаться к теоретической  
Изображение выглядит как текст

Автоматически созданное описание  


Гипотезы подтверждаются.

# Имитация сложного события

На основе написанного ранее датчика случайных чисел напишем генератор сложных событий, который результатом будет отдавать, какому классу принадлежит

class AB(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
class A\_notB(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
class notA\_B(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
class notA\_notB(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)

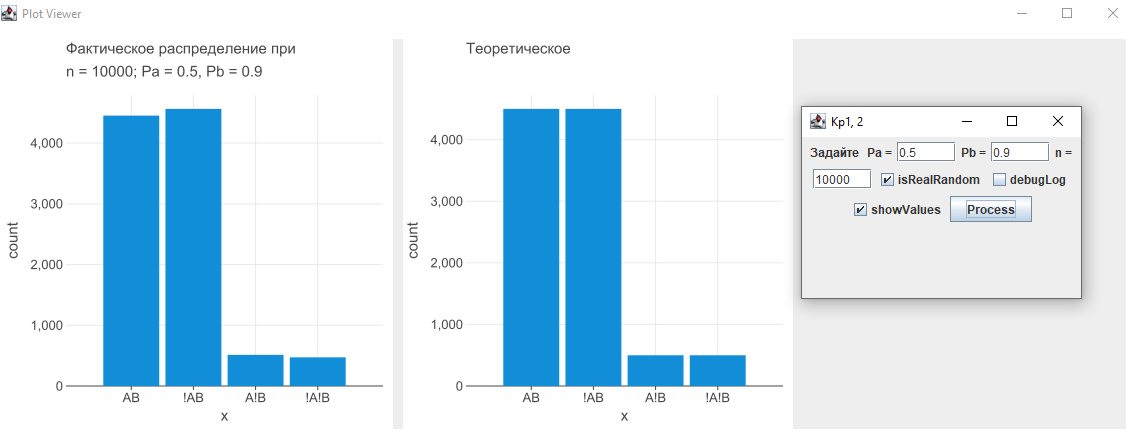
получившееся событие.

Проверим единичный запуск с Pa 0.5 и Pb 0.9:  
notA\_B :Pa = 0.45, x1 = 0.6089756356961242, Pb = 0.9, x2 = 0.5636461626915652

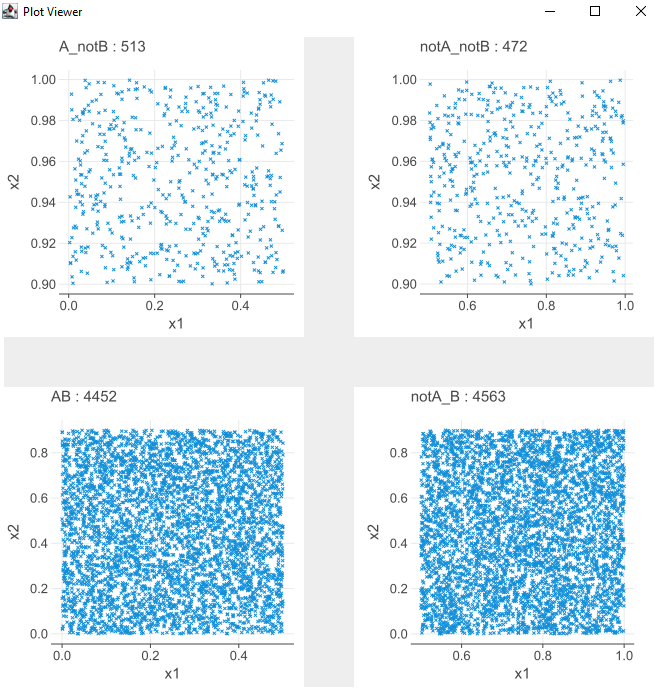
Теперь проверим следующую гипотезу: количество событий для каждой группы должно примерно соответствовать перемноженным вероятностям для каждой группы. Для Pa 0.5 и Pb 0.9:  
AB = 0.5\*0.9 = 0.45

notA\_B = 0.5\*0.9 = 0.45

A\_notB = 0.5\*0.1 = 0.05

notA\_notB = 0.5\*0.1 = 0.05  
Построим гистограмму

А также для наглядности будем выводить конкретные полученные x1 на ось X, x2 на ось Y. Тогда наш квадрат должен делиться строго по значения Pa и Pb

  
AB size = 4452

notA\_B size = 4563

A\_notB size = 513

notA\_notB size = 472  
Гипотеза подтверждается.

# Имитация сложного события, состоящего из зависимых событий.

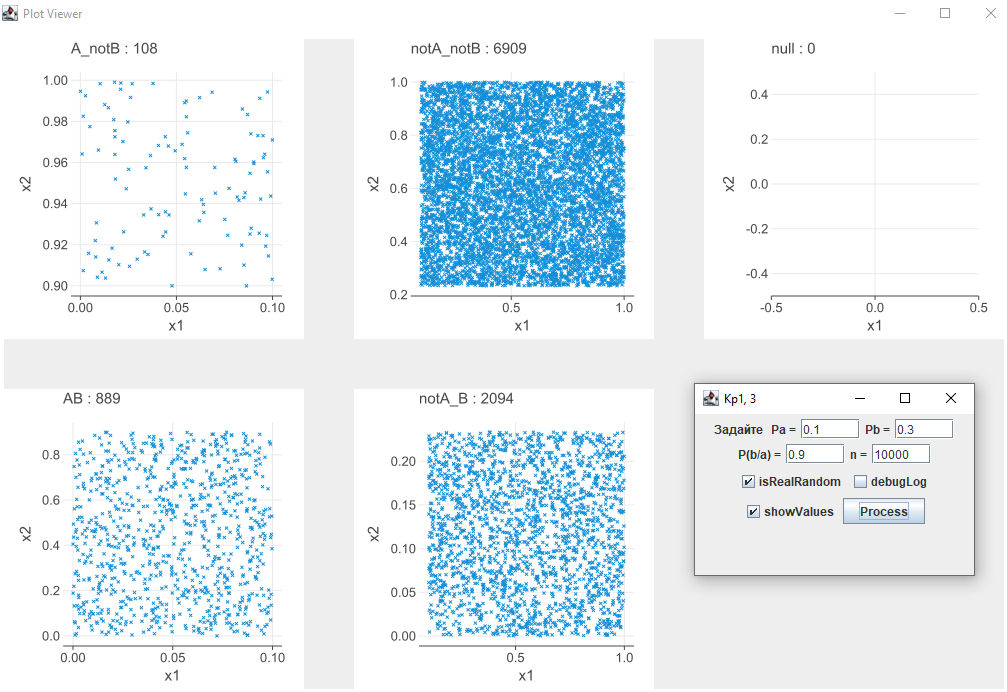
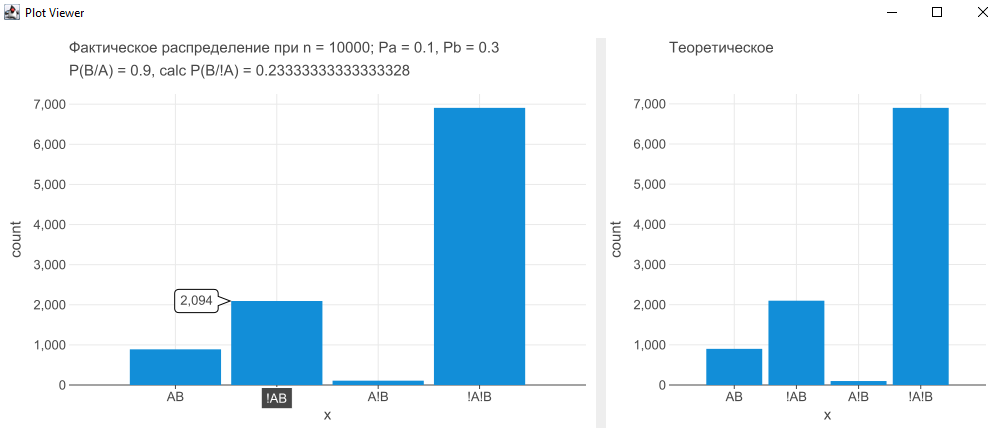
Используем похожий квадрат и гистограмму, только принадлежность к одной из четырех групп будет считаться иначе.

*\* формула полной ветоятности:  
\* P(B)=P(B/A)\*P(A) +P(B/!A)\*P(!A), отсюда выведем P(B/!A):  
\*  
\* P(B)-P(B/A)\*P(A)  
\* -------------- = P(B/!A)  
\* P(!A)*

Проведем эксперимент в 10000 событий с изначальными параметрами   
Pa = 0.1, Pb = 0.3, P(B/A) = 0.9. Выведем посчитанное P(B/!A) и проверим следующую гипотезу:  
Ожидаемые значения  
AB + A\_notB ~=1000

notA\_B + notA\_notB ~=9000

А группа notA\_B должна быть ~= 9000\* 0.23 ~=2100

Гипотеза подтверждается

# Имитация событий, составляющих полную группу

Разделим группы на заданные интервалы, кратные 16. Запустим генерацию 16000 событий, выведем все значения на гистограмму, разделенную по цветам в зависимости от принадлежности к группе. Проверим следующую гипотезу: кол-во значений в каждой порядковой группе должно соответствовать величине интервала.



|  |  |
| --- | --- |
| Размеры групп: | Величина интервала |
| A0: size = 0 | 0/16 |
| A1: size = 954 | 1/16 |
| A2: size = 2008 | 2/16 |
| A3: size = 5070 | 5/16 |
| A4: size = 2021 | 2/16 |
| A5: size = 1985 | 2/16 |
| A6: size = 3962 | 4/16 |

Гипотеза выполняется

# Листинг кода

## Cохранение параметров ввода

package kr\_one  
  
import java.util.prefs.Preferences  
  
object KrOneParamsSaver {  
 private fun prefNodeOne(): Preferences = Preferences.userRoot().node("KR\_ONE\_ONE")  
 private fun prefNodeTwo(): Preferences = Preferences.userRoot().node("KR\_ONE\_TWO")  
 private fun prefNodeThree(): Preferences = Preferences.userRoot().node("KR\_ONE\_THREE")  
  
 fun loadKrOneOneParams(): KrOneOneParams {  
 return prefNodeOne().*run* **{** KrOneOneParams(  
 Pa = getDouble(KrOneOneParams.PA\_KEY, 0.5),  
 n = get(KrOneOneParams.N\_KEY, "5 100 1000 100000 1000000"),  
 realRandom = getBoolean(KrOneOneParams.RANDOM\_KEY, true),  
 debug = getBoolean(KrOneOneParams.DEBUG\_KEY, false)  
 )  
 **}** }  
  
 fun saveKrOneOneParams(params: KrOneOneParams){  
 prefNodeOne().*apply* **{** putDouble(KrOneOneParams.PA\_KEY, params.Pa)  
 put(KrOneOneParams.N\_KEY, params.n)  
 putBoolean(KrOneOneParams.RANDOM\_KEY, params.realRandom)  
 putBoolean(KrOneOneParams.DEBUG\_KEY, params.debug)  
 **}** }  
  
 fun loadKrOneTwoParams(): KrOneTwoParams {  
 return prefNodeTwo().*run* **{** KrOneTwoParams(  
 Pa = getDouble(KrOneTwoParams.PA\_KEY, 0.5),  
 Pb = getDouble(KrOneTwoParams.PB\_KEY, 0.5),  
 n = getInt(KrOneTwoParams.N\_KEY, 10),  
 realRandom = getBoolean(KrOneTwoParams.RANDOM\_KEY, true),  
 debug = getBoolean(KrOneTwoParams.DEBUG\_KEY, false),  
 valuesDraw = getBoolean(KrOneTwoParams.VALUES\_DRAW\_KEY, true)  
 )  
 **}** }  
  
 fun saveKrOneTwoParams(params: KrOneTwoParams) {  
 prefNodeTwo().*apply* **{** putDouble(KrOneTwoParams.PA\_KEY, params.Pa)  
 putDouble(KrOneTwoParams.PB\_KEY, params.Pb)  
 putInt(KrOneTwoParams.N\_KEY, params.n)  
 putBoolean(KrOneTwoParams.RANDOM\_KEY, params.realRandom)  
 putBoolean(KrOneTwoParams.DEBUG\_KEY, params.debug)  
 putBoolean(KrOneTwoParams.VALUES\_DRAW\_KEY, params.valuesDraw)  
 **}** }  
  
 fun loadKrOneThreeParams(): KrOneThreeParams {  
 return prefNodeThree().*run* **{** KrOneThreeParams(  
 Pa = getDouble(KrOneThreeParams.PA\_KEY, 0.5),  
 Pb = getDouble(KrOneThreeParams.PB\_KEY, 0.3),  
 PBdependantA = getDouble(KrOneThreeParams.PB\_DEP\_A\_KEY, 0.5),  
 n = getInt(KrOneThreeParams.N\_KEY, 10),  
 realRandom = getBoolean(KrOneThreeParams.RANDOM\_KEY, true),  
 debug = getBoolean(KrOneThreeParams.DEBUG\_KEY, false),  
 valuesDraw = getBoolean(KrOneThreeParams.VALUES\_DRAW\_KEY, true)  
 )  
 **}** }  
  
 fun saveKrOneThreeParams(params: KrOneThreeParams) {  
 prefNodeThree().*apply* **{** putDouble(KrOneThreeParams.PA\_KEY, params.Pa)  
 putDouble(KrOneThreeParams.PB\_KEY, params.Pb)  
 putDouble(KrOneThreeParams.PB\_DEP\_A\_KEY, params.PBdependantA)  
 putInt(KrOneThreeParams.N\_KEY, params.n)  
 putBoolean(KrOneThreeParams.RANDOM\_KEY, params.realRandom)  
 putBoolean(KrOneThreeParams.DEBUG\_KEY, params.debug)  
 putBoolean(KrOneThreeParams.VALUES\_DRAW\_KEY, params.valuesDraw)  
 **}** }  
  
}  
  
data class KrOneOneParams(  
 val Pa: Double,  
 val n: String,  
 val realRandom: Boolean,  
 val debug: Boolean  
) {  
 companion object {  
 const val PA\_KEY = "Pa"  
 const val N\_KEY = "n"  
 const val RANDOM\_KEY = "realRandom"  
 const val DEBUG\_KEY = "debug"  
 }  
}  
  
data class KrOneTwoParams(  
 val Pa: Double,  
 val Pb: Double,  
 val n: Int,  
 val realRandom: Boolean,  
 val debug: Boolean,  
 val valuesDraw: Boolean  
) {  
 companion object {  
 const val PA\_KEY = "Pa"  
 const val PB\_KEY = "Pb"  
 const val N\_KEY = "n"  
 const val RANDOM\_KEY = "realRandom"  
 const val DEBUG\_KEY = "debug"  
 const val VALUES\_DRAW\_KEY = "drawValues"  
 }  
}  
  
data class KrOneThreeParams(  
 val Pa: Double,  
 val Pb: Double,  
 val PBdependantA: Double,  
 val n: Int,  
 val realRandom: Boolean,  
 val debug: Boolean,  
 val valuesDraw: Boolean  
) {  
 companion object {  
 const val PA\_KEY = "Pa"  
 const val PB\_KEY = "Pb"  
 const val PB\_DEP\_A\_KEY = "Pb/a"  
 const val N\_KEY = "n"  
 const val RANDOM\_KEY = "realRandom"  
 const val DEBUG\_KEY = "debug"  
 const val VALUES\_DRAW\_KEY = "drawValues"  
 }  
}

## Имитация случайного события

### Основной код

package kr\_one  
  
import org.jetbrains.letsPlot.GGBunch  
import org.jetbrains.letsPlot.geom.geomBar  
import org.jetbrains.letsPlot.geom.geomHistogram  
import org.jetbrains.letsPlot.geom.geomVLine  
import org.jetbrains.letsPlot.label.ggtitle  
import org.jetbrains.letsPlot.letsPlot  
import java.awt.FlowLayout  
import javax.swing.\*  
import kotlin.math.pow  
import kotlin.math.sqrt  
  
*/\*\*  
 \** ***@param*** *isRealRandom: true if you want real random. False if you want repeatable pseudoRandom  
 \** ***@param*** *isDebug: should println results or not  
 \*/*class RandomEventGenerator(  
 private val isRealRandom: Boolean,  
 private val isDebug: Boolean  
) {  
 private val realRandom = java.util.Random()  
 private val pseudoRandom = kotlin.random.*Random*(10)  
  
 private fun realRandomNext() = realRandom.nextDouble()  
 private fun pseudoRandomNext() = pseudoRandom.nextDouble()  
  
 */\*\*  
 \* Returning true/false if event happened, gained probability and generated x  
 \** ***@see*** *Result  
 \*/* operator fun invoke(Pa: Double): Result {  
 val randomNumberX = if (isRealRandom) realRandomNext() else pseudoRandomNext()  
 val result = randomNumberX <= Pa  
  
 if (isDebug) {  
 *println*(  
 "x = $randomNumberX \n" +  
 "Pa = $Pa"  
 )  
 if (result) {  
 *println*("Event happened")  
 } else {  
 *println*("Event didn't happened")  
 }  
 }  
  
 return Result(result = result, Pa = Pa, randomNumber = randomNumberX)  
 }  
  
 */\*\*  
 \** ***@param*** *result is true, if event happened  
 \** ***@param*** *Pa given probability  
 \** ***@param*** *randomNumber number from >0 & <1  
 \*/* data class Result(  
 val result: Boolean,  
 val Pa: Double,  
 val randomNumber: Double  
 )  
}  
  
  
class KrOne1InputGetter : JFrame("Кр 1, 1") {  
 companion object {  
 @JvmStatic  
 fun main(args: Array<String>) {  
 //remove annoying warning "Graphics2D from BufferedImage lacks BUFFERED\_IMAGE hint", was actual for 1/2 PC  
 System.setProperty("org.apache.batik.warn\_destination", "false")  
  
 SwingUtilities.invokeLater **{** KrOne1InputGetter() **}** }  
  
 }  
  
 private val button = JButton("Process")  
 private val label = JLabel("Задайте Pa, и несколько n через пробел ")  
  
 private val labelN = JLabel("n =")  
 private val textFieldN = JTextField(KrOneParamsSaver.loadKrOneOneParams().n,21)  
  
 private val labelPa = JLabel("Pa =")  
 private val textFieldPa = JTextField(KrOneParamsSaver.loadKrOneOneParams().Pa.toString(),20)  
 private val cbRandom = JCheckBox("isRealRandom").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneOneParams().realRandom  
 **}** private val cbDebug = JCheckBox("debugLog").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneOneParams().debug  
 **}** init {  
 *layout* = FlowLayout()  
 *defaultCloseOperation* = *EXIT\_ON\_CLOSE* setLocationRelativeTo(null)  
 *isVisible* = true  
 setSize(295, 200)  
  
 button.addActionListener **{** processButtonClick()  
 **}** add(label)  
 add(labelPa)  
 add(textFieldPa)  
 add(labelN)  
 add(textFieldN)  
 add(cbRandom)  
 add(cbDebug)  
 add(button)  
 }  
  
 private fun processButtonClick() {  
 try {  
 val n = textFieldN.*text* val sizes = n.*trim*().*split*(" ").*map* **{ it**.*toInt*() **}** val Pa = textFieldPa.*text*.*toDouble*()  
 val isRealRandom = cbRandom.*isSelected* val isDebug = cbDebug.*isSelected* val randomEventGenerator = RandomEventGenerator(  
 isRealRandom = isRealRandom,  
 isDebug = isDebug  
 )  
 drawGraphs(Pa, sizes, randomEventGenerator)  
 KrOneParamsSaver.saveKrOneOneParams(  
 KrOneOneParams(  
 Pa = Pa,  
 n = n,  
 realRandom = isRealRandom,  
 debug = isDebug  
 )  
 )  
 } catch (e: Exception) {  
 JOptionPane.showMessageDialog(  
 this@KrOne1InputGetter,  
 """  
 Ошибка во время процессинга:  
 ${e.message}  
 """.*trimIndent*()  
 )  
 }  
 }  
  
 private fun drawGraphs(Pa: Double, sizes: List<Int>, randomEventGenerator: RandomEventGenerator) {  
 val bunch = GGBunch()  
 sizes.*forEachIndexed* **{** index, n **->** val list = *List*(n) **{** randomEventGenerator.invoke(Pa) **}** val data = *mapOf*<String, List<Double>>(  
 "x" *to* list.*map* **{ it**.randomNumber **}** )  
 val mx = data["x"]!!.*average*()  
 var dxSum = 0.0  
 data["x"]!!.*forEach* **{** dxSum += (**it** - mx).*pow*(2)  
 **}** //Несмещенная состоятельная оценка дисперсии  
 val dx = dxSum / (n - 1)  
  
 //Состоятельная оценка среднеквадратичного отклонения  
 val sigma = *sqrt*(dx)  
 val p = *letsPlot*(data) **{** x = "x" **}** + *ggtitle*(  
 "Pa = $Pa, n=$n\n" +  
 "Выборочное средее:\n" +  
 "$mx\n" +  
 "D[X] = $dx\n" +  
 "σ = $sigma"  
 )  
 *println*(  
 "Выборочное средее,n=$n:\n" +  
 "$mx\n" +  
 "D[X] = $dx\n" +  
 "σ = $sigma"  
 )  
  
 bunch.addPlot(  
 p + geomHistogram(  
 boundary = 0.0,  
 binWidth = 0.1,  
 color = "black",  
 fill = "white"  
 ) + geomVLine(  
 xintercept = (data["x"] as List<Double>).*average*(),  
 color = "red",  
 linetype = "dashed"  
 ), index \* 300, 0, 290, 190  
 )  
 val data2 = *mapOf*<String, List<String>>(  
 "x" *to* list.*map* **{** if (**it**.result) {  
 "A"  
 } else {  
 "!A"  
 }  
 **}**.*sortedBy* **{ it }** )  
 val p2 = *letsPlot*(data2) **{** x = "x";**}** + *ggtitle*("практический результат:")  
 bunch.addPlot(  
  
 p2 + geomBar(),  
 index \* 300, 200, 290, 190  
 )  
 val PaNInt = (Pa \* n).toInt()  
 val data3 = *mapOf*<String, List<String>>(  
 "x" *to* (*List*(PaNInt) **{** "A" **}** + *List*(n-PaNInt) **{** "!A" **}**).*sortedBy* **{ it }** )  
 val p3 = *letsPlot*(data3) **{** x = "x" **}** + *ggtitle*("теоретический результат:")  
 bunch.addPlot(  
  
 p3 + geomBar(),  
 index \* 300, 400, 290, 190  
 )  
 **}** bunch.show()  
 }  
}

### Тесты

package kr\_one  
  
import kotlin.test.Test  
import kotlin.test.assertEquals  
  
internal class RandomEventGeneratorTest {  
  
 private val randomEventGenerator = RandomEventGenerator(isRealRandom = true, isDebug = false)  
  
 @Test  
 fun testBounds() {  
 val resultList = *List*<RandomEventGenerator.Result>(100000) **{** randomEventGenerator.invoke(0.5) **}** *assertEquals*(false, resultList.*any* **{ it**.randomNumber == 0.0 **}**)  
 *assertEquals*(false, resultList.*any* **{ it**.randomNumber == 1.0 **}**)  
 val bounds1 = resultList.*filter* **{ it**.randomNumber < 0.1 **}**.*apply* **{** *println*("bounds1 size = ${this.size}") **}** val bounds2 = resultList.*filter* **{ it**.randomNumber >= 0.1 && **it**.randomNumber < 0.2 **}**.*apply* **{** *println*("bounds2 size = ${this.size}") **}** val bounds3 = resultList.*filter* **{ it**.randomNumber >= 0.2 && **it**.randomNumber < 0.3 **}**.*apply* **{** *println*("bounds3 size = ${this.size}") **}** val bounds4 = resultList.*filter* **{ it**.randomNumber >= 0.3 && **it**.randomNumber < 0.4 **}**.*apply* **{** *println*("bounds4 size = ${this.size}") **}** val bounds5 = resultList.*filter* **{ it**.randomNumber >= 0.4 && **it**.randomNumber < 0.5 **}**.*apply* **{** *println*("bounds5 size = ${this.size}") **}** val bounds6 = resultList.*filter* **{ it**.randomNumber >= 0.5 && **it**.randomNumber < 0.6 **}**.*apply* **{** *println*("bounds6 size = ${this.size}") **}** val bounds7 = resultList.*filter* **{ it**.randomNumber >= 0.6 && **it**.randomNumber < 0.7 **}**.*apply* **{** *println*("bounds7 size = ${this.size}") **}** val bounds8 = resultList.*filter* **{ it**.randomNumber >= 0.7 && **it**.randomNumber < 0.8 **}**.*apply* **{** *println*("bounds8 size = ${this.size}") **}** val bounds9 = resultList.*filter* **{ it**.randomNumber >= 0.8 && **it**.randomNumber < 0.9 **}**.*apply* **{** *println*("bounds9 size = ${this.size}") **}** val bounds10 = resultList.*filter* **{ it**.randomNumber >= 0.9 && **it**.randomNumber < 1.0 **}**.*apply* **{** *println*("bounds10 size = ${this.size}") **}** *assertEquals*(true, bounds1.size in 9001..10999)  
 *assertEquals*(true, bounds2.size in 9001..10999)  
 *assertEquals*(true, bounds3.size in 9001..10999)  
 *assertEquals*(true, bounds4.size in 9001..10999)  
 *assertEquals*(true, bounds5.size in 9001..10999)  
 *assertEquals*(true, bounds6.size in 9001..10999)  
 *assertEquals*(true, bounds7.size in 9001..10999)  
 *assertEquals*(true, bounds8.size in 9001..10999)  
 *assertEquals*(true, bounds9.size in 9001..10999)  
 *assertEquals*(true, bounds10.size in 9001..10999)  
 }  
  
 @Test  
 fun test1\_0probability() {  
 val resultList = *List*<RandomEventGenerator.Result>(100000) **{** randomEventGenerator.invoke(1.0) **}** *assertEquals*(true, resultList.*all* **{ it**.result **}**)  
 *assertEquals*(true, resultList.*all* **{ it**.Pa == 1.0 **}**)  
 }  
  
 @Test  
 fun test0\_0probability() {  
 val resultList = *List*<RandomEventGenerator.Result>(100000) **{** randomEventGenerator.invoke(0.0) **}** *assertEquals*(true, resultList.*all* **{** !**it**.result **}**)  
 *assertEquals*(true, resultList.*all* **{ it**.Pa == 0.0 **}**)  
 }  
  
 @Test  
 fun testPseudoRandom() {  
 val pseudoRandom1 = RandomEventGenerator(isRealRandom = false, isDebug = false)  
 val pseudoRandom2 = RandomEventGenerator(isRealRandom = false, isDebug = false)  
  
 val result1List = *List*<RandomEventGenerator.Result>(100000) **{** pseudoRandom1.invoke(0.5) **}** val result2List = *List*<RandomEventGenerator.Result>(100000) **{** pseudoRandom2.invoke(0.5) **}** *assertEquals*(result1List, result2List)  
 *assertEquals*(true, result1List.subList(0, 1000) == result2List.subList(0, 1000))  
 *assertEquals*(true, result1List[10000] == result2List[10000])  
 }  
}

## Имитация сложного события

### Основной код

package kr\_one  
  
import org.jetbrains.letsPlot.GGBunch  
import org.jetbrains.letsPlot.geom.geomBar  
import org.jetbrains.letsPlot.geom.geomPoint  
import org.jetbrains.letsPlot.intern.Plot  
import org.jetbrains.letsPlot.label.ggtitle  
import org.jetbrains.letsPlot.letsPlot  
import java.awt.FlowLayout  
import javax.swing.\*  
  
class ComplexEventGenerator(  
 val isRealRandom: Boolean, val isDebug: Boolean  
) {  
 private val randomEventGenerator = RandomEventGenerator(isRealRandom = isRealRandom, isDebug = false)  
  
 operator fun invoke(Pa: Double, Pb: Double): Result {  
 val resultA = randomEventGenerator.invoke(Pa)  
 val resultB = randomEventGenerator.invoke(Pb)  
  
 val complexEventResult = if (resultA.result) {  
 if (resultB.result) {  
 Result.AB(Pa = Pa, x1 = resultA.randomNumber, Pb = Pb, x2 = resultB.randomNumber)  
 } else {  
 Result.A\_notB(Pa = Pa, x1 = resultA.randomNumber, Pb = Pb, x2 = resultB.randomNumber)  
 }  
 } else {  
 if (resultB.result) {  
 Result.notA\_B(Pa = Pa, x1 = resultA.randomNumber, Pb = Pb, x2 = resultB.randomNumber)  
 } else {  
 Result.notA\_notB(Pa = Pa, x1 = resultA.randomNumber, Pb = Pb, x2 = resultB.randomNumber)  
 }  
 }  
 if (isDebug) *println*(complexEventResult.debugString())  
 return complexEventResult  
 }  
  
 sealed class Result(  
 val Pa: Double, val x1: Double, val Pb: Double, val x2: Double  
 ) {  
 class AB(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
 class A\_notB(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
 class notA\_B(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
 class notA\_notB(Pa: Double, x1: Double, Pb: Double, x2: Double) : Result(Pa, x1, Pb, x2)  
  
 fun debugString(): String {  
 return ("${this.*javaClass*.*simpleName*} :" + "Pa = $Pa, x1 = $x1, Pb = $Pb, x2 = $x2")  
 }  
  
 fun graphTitle(): String = this.*javaClass*.*simpleName* }  
}  
  
class KrOne2InputGetter : JFrame("Кр1, 2") {  
 companion object {  
 @JvmStatic  
 fun main(args: Array<String>) {  
 //remove annoying warning "Graphics2D from BufferedImage lacks BUFFERED\_IMAGE hint", was actual for 1/2 PC  
 System.setProperty("org.apache.batik.warn\_destination", "false")  
  
 SwingUtilities.invokeLater **{** KrOne2InputGetter() **}** }  
 }  
  
 private val button = JButton("Process")  
 private val label = JLabel("Задайте ")  
  
 private val labelN = JLabel("n =")  
 private val textFieldN = JTextField(KrOneParamsSaver.loadKrOneTwoParams().n.toString(), 5)  
  
 private val labelPa = JLabel("Pa =")  
 private val textFieldPa = JTextField(KrOneParamsSaver.loadKrOneTwoParams().Pa.toString(), 5)  
 private val labelPb = JLabel("Pb =")  
 private val textFieldPb = JTextField(KrOneParamsSaver.loadKrOneTwoParams().Pb.toString(), 5)  
 private val cbRandom = JCheckBox("isRealRandom").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneTwoParams().realRandom  
 **}** private val cbDebug = JCheckBox("debugLog").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneTwoParams().debug  
 **}** private val cbValues = JCheckBox("showValues").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneTwoParams().valuesDraw  
 **}** init {  
 *layout* = FlowLayout()  
 *defaultCloseOperation* = *EXIT\_ON\_CLOSE* setLocationRelativeTo(null)  
 *isVisible* = true  
 setSize(295, 200)  
  
 button.addActionListener **{** processButtonClick()  
 **}** add(label)  
 add(labelPa)  
 add(textFieldPa)  
 add(labelPb)  
 add(textFieldPb)  
 add(labelN)  
 add(textFieldN)  
 add(cbRandom)  
 add(cbDebug)  
 add(cbValues)  
 add(button)  
 }  
  
  
 private fun processButtonClick() {  
 try {  
 val n = textFieldN.*text*.*toInt*()  
 val Pa = textFieldPa.*text*.*toDouble*()  
 val Pb = textFieldPb.*text*.*toDouble*()  
 val isRealRandom = cbRandom.*isSelected* val isDebug = cbDebug.*isSelected* val isValuesDraw = cbValues.*isSelected* val complexEventGenerator = ComplexEventGenerator(  
 isRealRandom = isRealRandom, isDebug = isDebug  
 )  
 drawGraphs(n, Pa, Pb, complexEventGenerator, isValuesDraw)  
  
 KrOneParamsSaver.saveKrOneTwoParams(  
 KrOneTwoParams(  
 Pa = Pa, Pb = Pb, n = n, realRandom = isRealRandom, debug = isDebug, valuesDraw = isValuesDraw  
 )  
 )  
  
  
 } catch (e: Exception) {  
 JOptionPane.showMessageDialog(  
 this@KrOne2InputGetter, """  
 Ошибка во время процессинга:  
 ${e.message}  
 """.*trimIndent*()  
 )  
 }  
 }  
  
 private fun drawGraphs(  
 n: Int, Pa: Double, Pb: Double, complexEventGenerator: ComplexEventGenerator, valuesDraw: Boolean  
 ) {  
 val resultList = *List*<ComplexEventGenerator.Result>(n) **{** complexEventGenerator.invoke(Pa, Pb) **}** val ABList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** val notA\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** val bunch = GGBunch()  
  
 val data =  
 *mapOf*<String, List<\*>>("x" *to* ABList.*map* **{** "AB" **}** + notA\_BList.*map* **{** "!AB" **}** + A\_notBList.*map* **{** "A!B" **}** + notA\_notBList.*map* **{** "!A!B" **}**)  
  
 val p = *letsPlot*(data) **{** x = "x" **}** + *ggtitle*("Фактическое распределение при", "n = $n; Pa = $Pa, Pb = $Pb")  
  
 val ABt = *List*((n \* Pa \* Pb).toInt()) **{** "AB"**}** val notA\_Bt = *List*((n \* (1.0-Pa) \* Pb).toInt()) **{** "!AB"**}** val A\_notBt = *List*((n \* Pa \* (1.0-Pb)).toInt()) **{** "A!B"**}** val notA\_notBt = *List*((n \* (1.0-Pa) \* (1.0-Pb)).toInt()) **{** "!A!B"**}** val data2 =  
 *mapOf*<String, List<\*>>("x" *to* ABt + notA\_Bt + A\_notBt + notA\_notBt)  
 val p2 = *letsPlot*(data2) **{**x = "x"**}** + *ggtitle* ("Теоретическое","")  
  
 bunch.addPlot(p+geomBar(), 0, 0, 390, 390)  
 bunch.addPlot(p2+geomBar(), 400, 0, 390, 390)  
  
 bunch.show()  
 if (valuesDraw) {  
 *drawValues*(n, Pa, Pb, resultList)  
 }  
  
 }  
}  
  
  
private fun drawValues(n: Int, Pa: Double, Pb: Double, resultList: List<ComplexEventGenerator.Result>) {  
 *println*("Testing resultList with Pa = $Pa, Pb = $Pb, n = $n")  
 val ABList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.AB>()  
 val notA\_BList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_B>()  
 val A\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.A\_notB>()  
 val notA\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_notB>()  
 GGBunch().addPlot(  
 *plotList*(ABList), 0, 350, 300, 300  
 ).addPlot(  
 *plotList*(notA\_BList), 350, 350, 300, 300  
 ).addPlot(  
 *plotList*(A\_notBList), 0, 0, 300, 300  
 ).addPlot(  
 *plotList*(notA\_notBList), 350, 0, 300, 300  
 ).show()  
}  
  
private fun plotList(list: List<ComplexEventGenerator.Result>): Plot {  
 val data = *mapOf*<String, List<\*>>("x1" *to* list.*map* **{ it**.x1 **}**, "x2" *to* list.*map* **{ it**.x2 **}**)  
  
 val p = *letsPlot*(data) **{** x = "x1"; y = "x2" **}** + *ggtitle*(  
 list.*firstOrNull*()?.graphTitle() + " : ${list.size}"  
 )  
 return (p + geomPoint(shape = 4))  
}

### Тесты

package kr\_one  
  
import kotlin.test.Test  
import kotlin.test.assertEquals  
  
internal class ComplexEventGeneratorTest {  
  
 private val complexEventGenerator = ComplexEventGenerator(isRealRandom = true, isDebug = false)  
  
 @Test  
 fun testAtleastOne() {  
 val resultList = *List*<ComplexEventGenerator.Result>(100000) **{**complexEventGenerator.invoke(0.5, 0.5)**}** *assertEquals*(true, resultList.*any* **{ it** is ComplexEventGenerator.Result.AB **}**)  
 *assertEquals*(true, resultList.*any* **{ it** is ComplexEventGenerator.Result.notA\_B **}**)  
 *assertEquals*(true, resultList.*any* **{ it** is ComplexEventGenerator.Result.A\_notB **}**)  
 *assertEquals*(true, resultList.*any* **{ it** is ComplexEventGenerator.Result.notA\_notB **}**)  
 }  
  
 @Test  
 fun test0\_5Distribution() {  
 val resultList = *List*<ComplexEventGenerator.Result>(100000) **{**complexEventGenerator.invoke(0.5, 0.5)**}** *println*("Testing resultList with Pa = 0.5, Pb = 0.5")  
 val ABList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** val notA\_notBLIst = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** *assertEquals*(true, ABList.size in 24000..26000)  
 *assertEquals*(true, notA\_BList.size in 24000..26000)  
 *assertEquals*(true, A\_notBList.size in 24000..26000)  
 *assertEquals*(true, notA\_notBLIst.size in 24000..26000)  
 *assertEquals*(true, resultList.size == ABList.size + notA\_BList.size + A\_notBList.size + notA\_notBLIst.size)  
  
 }  
  
 @Test  
 fun testPa0Distribution() {  
 val resultList = *List*<ComplexEventGenerator.Result>(100000) **{**complexEventGenerator.invoke(0.0, 0.5)**}** *println*("Testing resultList with Pa = 0.0, Pb = 0.5")  
 val ABList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** val notA\_notBLIst = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** *assertEquals*(0, ABList.size)  
 *assertEquals*(0, A\_notBList.size)  
 *assertEquals*(true, notA\_BList.size in 49000..51000)  
 *assertEquals*(true, notA\_notBLIst.size in 49000..51000)  
 *assertEquals*(true, resultList.size == ABList.size + notA\_BList.size + A\_notBList.size + notA\_notBLIst.size)  
 }  
  
 @Test  
 fun testPb0Distribution() {  
 val resultList = *List*<ComplexEventGenerator.Result>(100000) **{**complexEventGenerator.invoke(0.5, 0.0)**}** *println*("Testing resultList with Pa = 0.5, Pb = 0.0")  
 val ABList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** val notA\_notBLIst = resultList.*filterIsInstance*<ComplexEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** *assertEquals*(0, ABList.size)  
 *assertEquals*(0, notA\_BList.size)  
 *assertEquals*(true, A\_notBList.size in 49000..51000)  
 *assertEquals*(true, notA\_notBLIst.size in 49000..51000)  
 *assertEquals*(true, resultList.size == ABList.size + notA\_BList.size + A\_notBList.size + notA\_notBLIst.size)  
 }  
}

## Имитация сложного события, состоящего из зависимых событий.

### Основной код

package kr\_one  
  
import org.jetbrains.letsPlot.GGBunch  
import org.jetbrains.letsPlot.geom.geomBar  
import org.jetbrains.letsPlot.geom.geomPoint  
import org.jetbrains.letsPlot.intern.Plot  
import org.jetbrains.letsPlot.label.ggtitle  
import org.jetbrains.letsPlot.letsPlot  
import java.awt.FlowLayout  
import javax.swing.\*  
  
class ComplexDependantEventGenerator(  
 val isRealRandom: Boolean,  
 val isDebug: Boolean,  
 val randomEventGenerator: RandomEventGenerator  
) {  
  
 operator fun invoke(Pa: Double, Pb: Double, PBdependantA: Double): Result {  
 val resultA = randomEventGenerator(Pa)  
 val resultB = randomEventGenerator(Pb)  
  
 val PBdependant\_notA = try {  
 calculate\_PBdepentant\_notA(Pa, Pb, PBdependantA)  
 } catch (e: Exception) {  
 val exceptionResult =  
 Result.InvalidData(  
 Pa,  
 Pb,  
 PBdependantA,  
 INVALID\_PBDEPENDANT\_NOTA,  
 resultA.randomNumber,  
 resultB.randomNumber  
 )  
 if (isDebug) *println*(exceptionResult.debugString())  
 return exceptionResult  
 }  
  
 val complexDependantEventResult =  
 if (resultA.result) { //x1<=Pa  
 if (resultB.randomNumber <= PBdependantA) { //x2<=P(B/A)  
 Result.AB(Pa, Pb, PBdependantA, PBdependant\_notA, resultA.randomNumber, resultB.randomNumber)  
 } else { //x2>P(B/A)  
 Result.A\_notB(Pa, Pb, PBdependantA, PBdependant\_notA, resultA.randomNumber, resultB.randomNumber)  
 }  
 } else { //x1>Pa  
 if (resultB.randomNumber <= PBdependant\_notA) { //x2<=P(B/!A)  
 Result.notA\_B(Pa, Pb, PBdependantA, PBdependant\_notA, resultA.randomNumber, resultB.randomNumber)  
 } else { //x2>P(B/!A)  
 Result.notA\_notB(Pa, Pb, PBdependantA, PBdependant\_notA, resultA.randomNumber, resultB.randomNumber)  
 }  
 }  
 if (isDebug) *println*(complexDependantEventResult.debugString())  
 return complexDependantEventResult  
  
 }  
  
 */\*\*  
 \* Возьмем за А - событие из RandomEventGenerator.  
 \* формула полной ветоятности:  
 \* P(B)=P(B/A)\*P(A) +P(B/!A)\*P(!A), отсюда выведем P(B/!A):  
 \*  
 \* P(B)-P(B/A)\*P(A)  
 \* -------------- = P(B/!A)  
 \* P(!A)  
 \*  
 \*/* @Throws(Exception::class)  
 private fun calculate\_PBdepentant\_notA(Pa: Double, Pb: Double, PBdependantA: Double): Double {  
 //если невозможно событие !A, то и P(B/!A) невозможно (+ деление на ноль)  
 if (Pa == 1.0) {  
 throw Exception("P(A) = 1.0")  
 }  
 //если невозможно событие B, то и зависимые события невозможны и задача не имеет смысла  
 if (Pb == 0.0) {  
 throw Exception("P(B) = 0.0")  
 }  
 //вероятность одной части уравнения не может быть больше P(B)  
 if (PBdependantA \* Pa > Pb) {  
 throw Exception("P(B/A)\*P(A) > P(B)")  
 }  
  
 val PBdependant\_notA = (Pb - PBdependantA \* Pa) / (1.0 - Pa)  
  
 if (PBdependant\_notA > 1.0 || PBdependant\_notA < 0) {  
 throw Exception("P(B/A) не валидно, данные не совместны")  
 }  
  
 return PBdependant\_notA  
 }  
  
 sealed class Result(  
 val Pa: Double,  
 val Pb: Double,  
 val PBdependant\_A: Double,  
 val PBdependant\_notA: Double,  
 val x1: Double,  
 val x2: Double  
 ) {  
 */\*\*  
 \* Деталь вышла с завода 1 и она хорошая  
 \*/* class AB(Pa: Double, Pb: Double, PBdependantA: Double, PBdependant\_notA: Double, x1: Double, x2: Double) :  
 Result(Pa, Pb, PBdependantA, PBdependant\_notA, x1, x2)  
  
 */\*\*  
 \* Деталь вышла с завода 1 и она плохая  
 \*/* class A\_notB(Pa: Double, Pb: Double, PBdependantA: Double, PBdependant\_notA: Double, x1: Double, x2: Double) :  
 Result(Pa, Pb, PBdependantA, PBdependant\_notA, x1, x2)  
  
 */\*\*  
 \* Деталь вышла с завода 2 и она хорошая  
 \*/* class notA\_B(Pa: Double, Pb: Double, PBdependantA: Double, PBdependant\_notA: Double, x1: Double, x2: Double) :  
 Result(Pa, Pb, PBdependantA, PBdependant\_notA, x1, x2)  
  
 */\*\*  
 \* Деталь вышла с завода 2 и она плохая  
 \*/* class notA\_notB(  
 Pa: Double,  
 Pb: Double,  
 PBdependantA: Double,  
 PBdependant\_notA: Double,  
 x1: Double,  
 x2: Double  
 ) :  
 Result(Pa, Pb, PBdependantA, PBdependant\_notA, x1, x2)  
  
 class InvalidData(  
 Pa: Double,  
 Pb: Double,  
 PBdependantA: Double,  
 PBdependant\_notA: Double = 0.0,  
 x1: Double,  
 x2: Double  
 ) : Result(Pa, Pb, PBdependantA, PBdependant\_notA, x1, x2)  
  
 fun debugString(): String {  
 return ("Pa = $Pa, Pb = $Pb, PBdependant\_A = $PBdependant\_A, PBdependant\_notA = $PBdependant\_notA"  
 //+ "\n ${this.javaClass.simpleName} : x1 = $x1, x2 = $x2"  
 )  
 }  
  
 fun graphTitle(): String = this.*javaClass*.*simpleName* }  
  
 companion object {  
 const val INVALID\_PBDEPENDANT\_NOTA = 10000.0  
 }  
}  
  
class KrOne3InputGetter : JFrame("Кр1, 3") {  
 companion object {  
 @JvmStatic  
 fun main(args: Array<String>) {  
 //remove annoying warning "Graphics2D from BufferedImage lacks BUFFERED\_IMAGE hint", was actual for 1/2 PC  
 System.setProperty("org.apache.batik.warn\_destination", "false")  
  
 SwingUtilities.invokeLater **{** KrOne3InputGetter() **}** }  
 }  
  
 private val button = JButton("Process")  
 private val label = JLabel("Задайте ")  
  
 private val labelN = JLabel("n =")  
 private val textFieldN = JTextField(KrOneParamsSaver.loadKrOneThreeParams().n.toString(), 5)  
  
 private val labelPa = JLabel("Pa =")  
 private val textFieldPa = JTextField(KrOneParamsSaver.loadKrOneThreeParams().Pa.toString(), 5)  
 private val labelPb = JLabel("Pb =")  
 private val textFieldPb = JTextField(KrOneParamsSaver.loadKrOneThreeParams().Pb.toString(), 5)  
 private val labelPBdependantA = JLabel("P(b/a) =")  
 private val textFieldPBdependantA = JTextField(KrOneParamsSaver.loadKrOneThreeParams().PBdependantA.toString(), 5)  
 private val cbRandom = JCheckBox("isRealRandom").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneThreeParams().realRandom  
 **}** private val cbDebug = JCheckBox("debugLog").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneThreeParams().debug  
 **}** private val cbValues = JCheckBox("showValues").*apply* **{** *isSelected* = KrOneParamsSaver.loadKrOneThreeParams().valuesDraw  
 **}** init {  
 *layout* = FlowLayout()  
 *defaultCloseOperation* = *EXIT\_ON\_CLOSE* setLocationRelativeTo(null)  
 *isVisible* = true  
 setSize(295, 200)  
  
 button.addActionListener **{** processButtonClick()  
 **}** add(label)  
 add(labelPa)  
 add(textFieldPa)  
 add(labelPb)  
 add(textFieldPb)  
 add(labelPBdependantA)  
 add(textFieldPBdependantA)  
 add(labelN)  
 add(textFieldN)  
 add(cbRandom)  
 add(cbDebug)  
 add(cbValues)  
 add(button)  
 }  
  
 private fun processButtonClick() {  
 try {  
 val n = textFieldN.*text*.*toInt*()  
 val Pa = textFieldPa.*text*.*toDouble*()  
 val Pb = textFieldPb.*text*.*toDouble*()  
 val PBdependant\_A = textFieldPBdependantA.*text*.*toDouble*()  
 val isRealRandom = cbRandom.*isSelected* val isDebug = cbDebug.*isSelected* val isValuesDraw = cbValues.*isSelected* val randomEventGenerator = RandomEventGenerator(  
 isRealRandom = isRealRandom, isDebug = isDebug  
 )  
 val complexDependantEventGenerator = ComplexDependantEventGenerator(  
 isRealRandom = isRealRandom, isDebug = isDebug, randomEventGenerator = randomEventGenerator  
 )  
  
 drawGraphs(n, Pa, Pb, PBdependant\_A, complexDependantEventGenerator, isValuesDraw)  
  
 KrOneParamsSaver.saveKrOneThreeParams(  
 KrOneThreeParams(  
 Pa = Pa,  
 Pb = Pb,  
 PBdependantA = PBdependant\_A,  
 n = n,  
 realRandom = isRealRandom,  
 debug = isDebug,  
 valuesDraw = isValuesDraw  
 )  
 )  
 } catch (e: Exception) {  
 JOptionPane.showMessageDialog(  
 this@KrOne3InputGetter, """  
 Ошибка во время процессинга:  
 ${e.message}  
 """.*trimIndent*()  
 )  
 }  
 }  
  
 private fun drawGraphs(  
 n: Int,  
 Pa: Double,  
 Pb: Double,  
 PBdependant\_A: Double,  
 complexDependantEventGenerator: ComplexDependantEventGenerator,  
 valuesDraw: Boolean  
 ) {  
 //хороший пример для отчета  
 // val Pa = 0.5  
 // val Pb = 0.3  
 // val PBdependant\_A = 0.5  
 //тогда P(B) = P(A)\*P(B/A) + P(!A)\*P(B/!A)  
 //0.3 = 0.5\*0.5 + (1-0.5)\*x  
 //искомое = 0.1, тогда !A!B должно быть 0.9 от всех !A, или для десяти тысяч тысяч  
 //4500 и 500 для искомого  
 val resultList = *List*<ComplexDependantEventGenerator.Result>(n) **{** complexDependantEventGenerator.invoke(Pa, Pb, PBdependant\_A)  
 **}** *println*("Testing resultList with n = $n, Pa = $Pa, Pb = $Pb, P(B/A) = $PBdependant\_A, calc P(B/!A) = ${resultList.*first*().PBdependant\_notA}")  
  
 val ABList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.AB>().*apply* **{** *println*("AB size: $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size: $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size: $size")  
 **}** val notA\_notBList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size: $size")  
 **}** val invalidDataList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.InvalidData>().*apply* **{** *println*("Invalid size: $size")  
 **}** val bunch = GGBunch()  
  
  
 val data =  
 *mapOf*<String, List<\*>>("x" *to* ABList.*map* **{** "AB" **}** + notA\_BList.*map* **{** "!AB" **}** + A\_notBList.*map* **{** "A!B" **}** + notA\_notBList.*map* **{** "!A!B" **}** + invalidDataList.*map* **{** "invalid" **}**)  
 val p = *letsPlot*(data) **{** x = "x" **}** + *ggtitle*(  
 "Фактическое распределение при n = $n; Pa = $Pa, Pb = $Pb",  
 "P(B/A) = $PBdependant\_A, calc P(B/!A) = ${resultList.*first*().PBdependant\_notA}"  
 )  
  
 val ABt = *List*((n\*Pa\*PBdependant\_A).toInt()) **{**"AB"**}** val A\_notBt = *List*((n\*Pa\*(1.0-PBdependant\_A)).toInt()) **{** "A!B"**}** val notA\_Bt = *List*((n \* (1.0-Pa) \* resultList.*first*().PBdependant\_notA).toInt()) **{** "!AB"**}** val notA\_notBt = *List*((n \* (1.0-Pa) \* (1.0-resultList.*first*().PBdependant\_notA)).toInt()) **{** "!A!B"**}** val data2 =  
 *mapOf*<String, List<\*>>("x" *to* ABt + notA\_Bt + A\_notBt + notA\_notBt)  
 val p2 = *letsPlot*(data2) **{**x = "x"**}** + *ggtitle* ("Теоретическое", "")  
  
 bunch.addPlot(p+geomBar(), 0, 0, 590, 390)  
 bunch.addPlot(p2+geomBar(), 600, 0, 390, 390)  
 bunch.show()  
 if (valuesDraw) {  
 *drawValues*(resultList)  
 }  
 }  
}  
  
private fun drawValues(  
 resultList: List<ComplexDependantEventGenerator.Result>  
) {  
 val ABList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.AB>()  
 val notA\_BList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_B>()  
 val A\_notBList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.A\_notB>()  
 val notA\_notBList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_notB>()  
 val invalidDataList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.InvalidData>()  
  
 GGBunch()  
 .addPlot(  
 *plotList*(ABList), 0, 350, 300, 300  
 )  
 .addPlot(  
 *plotList*(notA\_BList), 350, 350, 300, 300  
 )  
 .addPlot(  
 *plotList*(A\_notBList), 0, 0, 300, 300  
 )  
 .addPlot(  
 *plotList*(notA\_notBList), 350, 0, 300, 300  
 )  
 .addPlot(  
 *plotList*(invalidDataList), 700, 0, 300, 300  
 )  
 .show()  
}  
  
private fun plotList(list: List<ComplexDependantEventGenerator.Result>): Plot {  
 val data = *mapOf*<String, List<\*>>(  
 "x1" *to* list.*map* **{ it**.x1 **}**,  
 "x2" *to* list.*map* **{ it**.x2 **}** )  
  
 val p = *letsPlot*(data) **{** x = "x1"; y = "x2" **}** + *ggtitle*(list.*firstOrNull*()?.graphTitle() + " : ${list.size}")  
 return (p + geomPoint(shape = 4))  
}

### Тесты

package kr\_one  
  
import kotlin.test.Test  
import kotlin.test.assertEquals  
  
internal class ComplexDependantEventGeneratorTest {  
  
 private val complexDependantEventGenerator = ComplexDependantEventGenerator(  
 isRealRandom = true,  
 isDebug = false,  
 randomEventGenerator = RandomEventGenerator(isRealRandom = true, isDebug = false))  
  
 @Test  
 fun test0\_5Distributions() {  
 val Pa = 0.5  
 val Pb = 0.5  
 val PBdependantA = 0.5  
 val resultList = *List*<ComplexDependantEventGenerator.Result>(100000) **{** complexDependantEventGenerator.invoke(  
 Pa,  
 Pb,  
 PBdependantA  
 )  
 **}** *assertEquals*(true, resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.InvalidData>().isEmpty())  
 *println*("Testing resultList with Pa = $Pa, Pb = $Pb, P(B/A) = $PBdependantA")  
 val ABList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** val notA\_notBLIst = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** *assertEquals*(true, ABList.size in 24000..26000)  
 *assertEquals*(true, notA\_BList.size in 24000..26000)  
 *assertEquals*(true, A\_notBList.size in 24000..26000)  
 *assertEquals*(true, notA\_notBLIst.size in 24000..26000)  
 *assertEquals*(true, resultList.size == ABList.size + notA\_BList.size + A\_notBList.size + notA\_notBLIst.size)  
 }  
  
 @Test  
 fun test0\_1and0\_9params() {  
 val Pa = 0.5  
 val Pb = 0.3  
 val PBdependantA = 0.5  
 //тогда P(B) = P(A)\*P(B/A) + P(!A)\*P(B/!A)  
 //0.3 = 0.5\*0.5 + (1-0.5)\*x  
 //искомое = 0.1, тогда !A!B должно быть 0.9 от всех !A, или для ста тысяч  
 //45 тысяч и 5 тысяч  
 val resultList = *List*<ComplexDependantEventGenerator.Result>(100000) **{** complexDependantEventGenerator.invoke(  
 Pa,  
 Pb,  
 PBdependantA  
 )  
 **}** *println*("Testing resultList with Pa = $Pa, Pb = $Pb, P(B/A) = $PBdependantA")  
 val ABList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** val notA\_BList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** val A\_notBList = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** val notA\_notBLIst = resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** *assertEquals*(true, ABList.size in 24000..26000)  
 *assertEquals*(true, notA\_BList.size in 4000..6000)  
 *assertEquals*(true, A\_notBList.size in 24000..26000)  
 *assertEquals*(true, notA\_notBLIst.size in 44000..46000)  
 *assertEquals*(true, resultList.size == ABList.size + notA\_BList.size + A\_notBList.size + notA\_notBLIst.size)  
  
 }  
  
 @Test  
 fun testOpenParams() {  
 val Pa = 0.5  
 val Pb = 0.5  
 val PBdependantA = 0.5  
 val resultList = *List*<ComplexDependantEventGenerator.Result>(100000) **{** complexDependantEventGenerator.invoke(  
 Pa,  
 Pb,  
 PBdependantA  
 )  
 **}** *println*("Testing resultList with Pa = $Pa, Pb = $Pb, P(B/A) = $PBdependantA")  
 resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.AB>().*apply* **{** *println*("AB size = $size")  
 **}** resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_B>().*apply* **{** *println*("notA\_B size = $size")  
 **}** resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.A\_notB>().*apply* **{** *println*("A\_notB size = $size")  
 **}** resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.notA\_notB>().*apply* **{** *println*("notA\_notB size = $size")  
 **}** resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.InvalidData>().*apply* **{** *println*("invalid size = $size")  
 **}** *assertEquals*(true, resultList.*filterIsInstance*<ComplexDependantEventGenerator.Result.InvalidData>().isEmpty())  
  
 }  
}

## Имитация событий, составляющих полную группу

### Основной код

package kr\_one  
  
import org.jetbrains.letsPlot.geom.geomHistogram  
import org.jetbrains.letsPlot.ggsize  
import org.jetbrains.letsPlot.letsPlot  
  
fun main() {  
 //remove annoying warning "Graphics2D from BufferedImage lacks BUFFERED\_IMAGE hint", was actual for 1/2 PC  
 System.setProperty("org.apache.batik.warn\_destination", "false")  
  
 val wholeGroupEventGenerator = WholeGroupEventGenerator(isReadRandom = true, isDebug = true)  
 val group = *listOf*<Double>(  
 0.0,  
 (1.0 / 8),  
 (2.0 / 8),  
 (5.0 / 8)  
 )  
 wholeGroupEventGenerator(group, 0.1)  
 wholeGroupEventGenerator(group, 0.35)  
 wholeGroupEventGenerator(group, 0.799999)  
  
 wholeGroupEventGenerator(group)  
 *drawValues*()  
}  
  
class WholeGroupEventGenerator(  
 isReadRandom: Boolean,  
 private val isDebug: Boolean  
) {  
 companion object {  
 private const val PA\_CONSTANT = 1.0  
 }  
  
 private val randomEventGenerator = RandomEventGenerator(isRealRandom = isReadRandom, isDebug = false)  
  
 */\*\*  
 \** ***@return*** *index of pGroupElement  
 \* pGroup[0] SHOULD be 0.0  
 \* PGroup element should be power of 2.  
 \* https://stackoverflow.com/questions/15625556/adding-and-subtracting-doubles-are-giving-strange-results  
 \*/* operator fun invoke(  
 pGroup: List<Double>,  
 predefinedX: Double? = null  
 ): Result {  
 validatePGroup(pGroup)?.*apply* **{** throw this **}** val resultX = predefinedX ?: randomEventGenerator(PA\_CONSTANT).randomNumber  
 var pSum = 0.0  
 pGroup.*forEachIndexed* **{** index, d **->** pSum += d  
 if (resultX < pSum) {  
 if (isDebug) *println*(  
 "resultX = $resultX \n" +  
 "pGroup = $pGroup \n" +  
 "index to return = $index \n" +  
 "pGroup element = $d \n"  
 )  
 return Result(  
 pGroup,  
 index,  
 resultX  
 )  
 }  
 **}** //shouldn't be the case  
 return Result(*listOf*(), 0, 0.0)  
 }  
  
 data class Result(  
 val group: List<Double>,  
 val indexBelonging: Int,  
 val randomNumber: Double  
 )  
  
 */\*\*  
 \* Validate input  
 \*/* private fun validatePGroup(list: List<Double>): Exception? {  
 if (list.*sum*() != 1.0)  
 return Exception("Group is not full, rebalance it to have 1.0 sum")  
 if (list.size < 2)  
 return Exception("List is empty, remake!")  
 if (list[0] != 0.0)  
 return Exception("P0 is not 0.0")  
 return null  
  
 }  
}  
  
private fun drawValues() {  
 val wholeGroupEventGenerator = WholeGroupEventGenerator(isReadRandom = true, isDebug = false)  
 val group = *listOf*<Double>(  
 0.0,  
 (1.0 / 16),  
 (2.0 / 16),  
 (5.0 / 16),  
 (2.0 / 16),  
 (2.0 / 16),  
 (4.0 / 16)  
 )  
 val possibleEventSublist = group  
 .*mapIndexed* **{** index, \_ **->** index *to* "A$index"  
 **}** val list = *List*(16000) **{** wholeGroupEventGenerator.invoke(group) **}** val resultCondList: MutableList<String> = *mutableListOf*()  
 val resultXList: MutableList<Double> = *mutableListOf*()  
  
 possibleEventSublist.*forEach* **{** possibleEvent **->** val resultXListIteration = list  
 .*filter* **{ it**.indexBelonging == possibleEvent.first **}** .*map* **{ it**.randomNumber **}** resultCondList += list  
 .*filter* **{it**.indexBelonging == possibleEvent.first**}** .*map* **{** possibleEvent.second **}** resultXList += resultXListIteration  
 *println*("event ${possibleEvent.second}: size = ${resultXListIteration.size}")  
 **}** val data = *mapOf*<String, Any>(  
 "cond" *to* resultCondList,  
 "x" *to* resultXList  
 )  
 val p = *letsPlot*(data) **{**x = "x"; fill = "cond"**}** + *ggsize*(800, 500)  
  
 (p+geomHistogram()).show()  
}

### Тесты

package kr\_one  
  
import kotlin.test.Test  
import kotlin.test.assertEquals  
  
internal class WholeGroupEventGeneratorTest {  
  
 private val wholeGroupEventGenerator = WholeGroupEventGenerator(isReadRandom = true, isDebug = false)  
  
 private val groupToTest = *listOf*<Double>(  
 0.0,  
 (2.0 / 16),  
 (4.0 / 16),  
 (10.0 / 16)  
 )  
  
 @Test  
 fun testBounds() {  
 val resultList = *List*<WholeGroupEventGenerator.Result>(100000) **{** wholeGroupEventGenerator.invoke(groupToTest)  
 **}** *assertEquals*(false, resultList.*any* **{ it**.indexBelonging == 0 **}**)  
 *assertEquals*(false, resultList.*any* **{ it**.indexBelonging > groupToTest.*lastIndex* **}**)  
 *assertEquals*(false, resultList.*any* **{ it**.randomNumber <= 0.0 **}**)  
 *assertEquals*(false, resultList.*any* **{ it**.randomNumber >= 1.0 **}**)  
 }  
}