**Лабораторна робота №1**

**Розробка програмного генератора псевдовипадкових чисел з рівномірним розподілом**

**Мета роботи**: навчитися розробляти програмний генератори псевдовипадкових чисел з рівномірним розподілом та перевіряти їх якість.

**Завдання**

Розробити генератор псевдовипадкових чисел з рівномірним розподілом  
(=Nv\*100, де Nv – номер варіанту) та згенерувати 10 000 чисел; перевірити період генератора та закон розподілу (рівномірний) псевдовипадкових чисел (тест Пірсона), побудувати гістограму розподілу. Зробити висновки згідно виконаного моделювання.

Короткі теоретичні відомості

Існують 3 методу генерування на ЕОМ випадкових чисел:

1. Апаратний;
2. Табличний;
3. Програмний.

Апаратний метод - заснований на якому-небудь фізичному способі, цей метод дозволяє одержувати випадкові числа. На практиці не застосовується.

Табличний - заздалегідь будуються таблиці випадкових чисел. Потрібна додаткова пам'ять, обмеження одержуваних чисел, утворюються псевдовипадкові числа. Для більшості задач не застосовується.

Програмний спосіб - одержання випадкових чисел за допомогою якої небудь функції:

Розглянемо метод, призначений для одержання чисел з рівномірним розподілом. Таким методом є метод середніх квадратів.

,

де коефіцієнти a, b, c можуть бути довільними.

Повний період циклу цього генератора буде отриманий у тому випадку, якщо це буде вибиратися в такий спосіб:

C=2B, B – розрядність використовуваного комп'ютера. Повний період тоді буде 2B. Коефіцієнт *a* - вибираємо відповідно до формули *a*=1+4\**k*, де *k* це ціле число. *b* - просте число відносно *с* (найбільший спільний дільник буде 1).

Період генератору – це кількість згенерованих значень, після якої наступні значення будуть повторюватися. Для перевірки періоду *m* треба зафіксувати кількість значень генератора, що не повторюються, і порівняти її зі значенням періоду. Якщо вони рівні то період генератору є дійсним.

Випадкова величина X називається рівномірно розподіленою на відрізку [*a*,*b*], якщо щільність розподілу її стала на цьому відрізку і має вигляд

. (1)

ЇЇ функція розподілу *F*(*x*):

. (2)

Висунемо гіпотезу *H0*, що генеральна сукупність розподілена за рівномірним законом. Як критерій перевірки гіпотези береться випадкова величина

 (3)

де  - емпіричні частоти;  - теоретичні частоти рівномірного розподілу; *k* – кількість інтервалів варіаційного ряду, побудованого за даними вибірки.

Теоретичні частоти знаходяться за формулою:



де  - ймовірність того, що випадкова величина *X* набуде значення з часткового інтервалу *Ii* інтервального варіаційного ряду. Вона дорівнює:



Для рівномірного розподілу дана ймовірність ,

де *k* – кількість інтервалів варіаційного ряду, побудованого за даними вибірки.

 – вибіркова дисперсія;

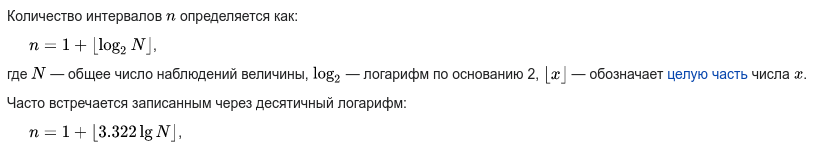
 – вибіркове середнє.

Для інтервального варіаційного ряду  - середина інтервалу.

Критерій (3) характеризує близькість теоретичного і емпіричного розподілів.

Критичне значення (межу правобічної критичної області)  знаходять за таблицею критичних точок розподілу  з використанням двох параметрів: заданого рівня значимості α і числа ступенів вільності *r* = *k*-1 (*k* – кількість інтервалів варіаційного ряду).

У випадку, якщо спостережене значення критерію , визначене за співвідношенням (3), виявиться меншим, ніж  то гіпотеза *H*0 є дійсною.



k=1+3.322lg(N)=14

α=0.05, r=k-1=13

Хід роботи

Запрограмуємо формулу методу середніх квадратів та згенеруємо 10000 псевдовипадкових рівномірно розподілених чисел.

За результатами генерації розділимо згенеровані числа на 15 інтервалів. Для цього використаємо знайдемо розмах вибірки та довжину інтервалу за наступними формулами:

Розмах вибірки:

, де – кількість згенерованих чисел.

Довжина інтервалу

Після цього визначимо абсолютні частоти на інтервалах та виконаємо перевірку критерія Пірсона ().

За результатами роботи отримали значення критерію = 7.995. Відповідне значення .

Текст програми:

***main.go***

package main

import (

"fmt"

"log"

"random-module/chi"

"random-module/random"

)

func main() {

random.SetSeed(6 \* 100)

size := 10000

stdDev := 6.0

mean := 6.0

fmt.Println("- - - Linear Distribution Generator - - -")

calcLcgPeriod()

testLcg(size)

fmt.Println("- - - Gaussian Distribution Generator - - -")

testGaussian(stdDev, mean, size)

}

func testLcg(size int) {

sample := random.GenRandomSlice(size)

intervals, \_, \_, \_ := chi.MakeIntervals(sample)

frequencies := chi.GetAbsoluteFrequencies(intervals)

critical, \_ := chi.GetChi2Critical(len(intervals)-1, 0.05)

actual := chi.CalcChi2Uni(frequencies, len(sample))

ok := actual < critical

strOk := "PASSED"

if !ok {

strOk = "FAILED"

}

printFrequencies(frequencies)

fmt.Printf("Chi-square distribution (actual): %.3f\n", actual)

fmt.Printf("Chi-square distribution (critical): %.3f\n", critical)

fmt.Printf("\n%s\n\n", strOk)

}

func testGaussian(stdDev, mean float64, size int) {

sample := random.GenNormallyRandomSlice(stdDev, mean, size)

intervals, intervalLen, min, \_ := chi.MakeIntervals(sample)

borders := chi.GetIntervalBorders(min, intervalLen, len(intervals))

frequencies := chi.GetAbsoluteFrequencies(intervals)

critical, \_ := chi.GetChi2Critical(len(intervals)-3, 0.05)

actual := chi.CalcChi2Gaussian(frequencies, borders, len(sample), intervalLen, stdDev, mean)

ok := actual < critical

strOk := "PASSED"

if !ok {

strOk = "FAILED"

}

printFrequencies(frequencies)

fmt.Printf("Chi-square distribution (actual): %.3f\n", actual)

fmt.Printf("Chi-square distribution (critical): %.3f\n", critical)

fmt.Printf("\n%s\n\n", strOk)

}

func printFrequencies(frequencies []float64) {

fmt.Println("Frequencies:")

for \_, item := range frequencies {

fmt.Printf("%.0f ", item)

}

fmt.Println()

}

func calcLcgPeriod() int {

fmt.Println("Calculating the generator period:")

first := random.LcgAllBits()

i := 1

for ; random.LcgAllBits() != first; i++ {

if i%1e8 == 0 {

fmt.Print(".")

}

}

fmt.Printf("\nPeriod: %d\n", i)

return i

}

func checkError(message string, err error) {

if err != nil {

log.Fatal(message, err)

}

}

***chi.go***

package chi

import (

"errors"

"math"

)

var critical5 = []float64{3.841, 5.991, 7.815, 9.488, 11.070, 12.592, 14.067, 15.507, 16.919, 18.307, 19.675, 21.026, 22.362, 23.685, 24.996, 26.296, 27.587, 28.869, 30.144, 31.410, 32.671, 33.924, 35.172, 36.415, 37.652, 38.885, 40.113, 41.337, 42.557, 43.773}

func CalcChi2Uni(frequencies []float64, total int) float64 {

res := 0.0

expected := float64(total) / float64(len(frequencies))

for \_, f := range frequencies {

res += (f - expected) \* (f - expected) / expected

}

return res

}

func CalcChi2Gaussian(frequencies, borders []float64, total int, intervalLen, stdDev, mean float64) float64 {

sum := 0.0

for i, f := range frequencies {

it := 200

x := make([]float64, it+1)

h := intervalLen / float64(it)

for j := 0; j < it+1; j++ {

x[int(j)] = borders[i] + h\*float64(j)

}

integral := calcIntegral(x, stdDev, mean) \* h

sum += math.Pow(f-(integral\*float64(total)), 2) / (integral \* float64(total))

}

return sum

}

func GetChi2Critical(df int, p float64) (float64, error) {

var critical \*[]float64 = nil

switch p {

case 0.05:

critical = &critical5

}

if critical == nil || df <= 0 || len(\*critical) < df {

return 0, errors.New("Unsupported parameters")

}

return (\*critical)[df-1], nil

}

func calcIntegral(interval []float64, stdDev, mean float64) float64 {

sum := 0.0

for i := 1; i < len(interval); i++ {

f := calcLaplasFunction(interval[i], stdDev, mean)

if !math.IsNaN(f) {

sum += f

}

}

a := calcLaplasFunction(interval[0], stdDev, mean)

b := calcLaplasFunction(interval[len(interval)-1], stdDev, mean)

sum += (a + b) / 2

return sum

}

func calcLaplasFunction(x, stdDev, mean float64) float64 {

return 1.0 / (stdDev \* math.Sqrt(2\*math.Pi)) \*

math.Exp(-math.Pow(x-mean, 2)/(2\*stdDev\*stdDev))

}

func GetAbsoluteFrequencies(intervals [][]float64) []float64 {

frequencies := make([]float64, len(intervals))

for i, interval := range intervals {

frequencies[i] = float64(len(interval))

}

return frequencies

}

func MakeIntervals(sample []float64) ([][]float64, float64, float64, float64) {

intervals := make([][]float64, GetNumberOfIntervals(len(sample)))

min, max := minMax(sample)

dataRange := max - min

intervalLen := dataRange / float64(len(intervals))

for \_, n := range sample {

index := int(math.Min((n-min)/intervalLen, float64(len(intervals)-1)))

intervals[index] = append(intervals[index], n)

}

return intervals, intervalLen, min, max

}

func GetIntervalBorders(min, intevalLen float64, intervals int) []float64 {

borders := make([]float64, intervals+1)

for i := 0; i < intervals+1; i++ {

borders[i] = intevalLen\*float64(i) + min

}

return borders

}

func GetNumberOfIntervals(n int) int {

return 1 + int(math.Log2(float64(n)))

}

func minMax(arr []float64) (float64, float64) {

max := arr[0]

min := arr[0]

for \_, n := range arr {

if n > max {

max = n

}

if n < min {

min = n

}

}

return min, max

}

***random.go***

package random

import "math"

const RAND\_MAX = 32767

const a = 1103515245

const c = 12345

const m = 2147483648

var next int64 = 1

var gaussianHasCalculated = false

var gaussianCalculated = 0.0

func Lcg() int {

next = (next\*a + c) % m

return int(next>>16) % (RAND\_MAX + 1)

}

func LcgAllBits() int {

next = (next\*a + c) % m

return int(next)

}

func Gaussian(stdDev, mean float64) float64 {

if gaussianHasCalculated {

gaussianHasCalculated = false

return gaussianCalculated\*stdDev + mean

}

x := float64(Lcg())/RAND\_MAX\*2 - 1

y := float64(Lcg())/RAND\_MAX\*2 - 1

s := x\*x + y\*y

if s == 0 || s > 1 {

return Gaussian(mean, stdDev)

}

c := math.Sqrt(-2 \* math.Log(s) / s)

z0 := x \* c

z1 := y \* c

gaussianHasCalculated = true

gaussianCalculated = z1

return z0\*stdDev + mean

}

func SetSeed(seed int64) {

next = seed

}

func GenRandomSlice(len int) []float64 {

res := make([]float64, len)

for i := 0; i < len; i++ {

res[i] = float64(Lcg())

}

return res

}

func GenNormallyRandomSlice(stdDev, mean float64, len int) []float64 {

res := make([]float64, len)

for i := 0; i < len; i++ {

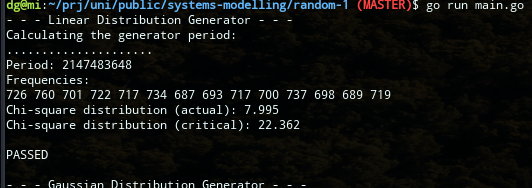
res[i] = Gaussian(stdDev, mean)

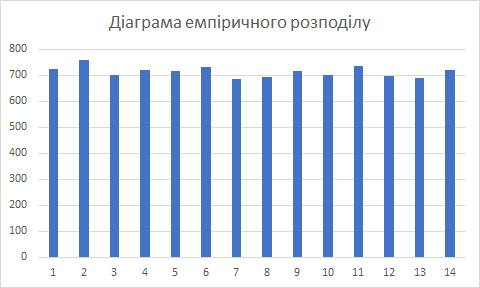
}

return res

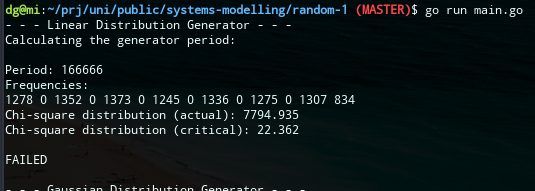
}

Результат виконання програми:





Також спробуємо «зіпсувати» функцію генерації випадкових чисел, встановивши не оптимальні параметри:



**Висновок:** виконавши лабораторну роботу, я набув теоретичних знань та практичних навичок розробки генераторів псевдовипадкових чисел з рівномірним розподілом; навчився перевіряти адекватність розподілу таких генераторів.

За результатами перевірки згенерованих чисел можна сказати, що їх розподіл дійсно є рівномірним, адже (7.995, 22.362), що в свою чергу дозволяє прийняти нульову гіпотезу про рівномірний розподіл псевдовипадкових чисел з достовірчою ймовірністю 0,95.

А у випадку не оптимальних параметрів видно, що , та нульову гіпотезу необхідно відхилити.