Slajd 1

Przedstawienie tematu; prowadzący

Slajd 2

Spis treści;

Slajd 3

tekst na slajdzie;

Metoda została opracowana i pierwszy raz zastosowana przez Stanisława Ulama.

Metodą Monte Carlo można obliczyć pole figury zdefiniowanej nierównością: $x^2 + y^2 \leqslant R^2,$

czyli koła o promieniu R i środku w punkcie (0,0).

1. Losuje się n punktów z opisanego na tym kole kwadratu – dla koła o R=1 współrzędne wierzchołków (−1,−1), (−1,1), (1,1), (1,−1).
2. Po wylosowaniu każdego z tych punktów trzeba sprawdzić czy jego współrzędne spełniają powyższą nierówność (tj. czy punkt należy do koła).

Wynikiem losowania jest informacja, że z n wszystkich prób k było trafionych, zatem pole koła wynosi:

$$P_k = P\,\frac{k}{n},$$

gdzie P je tu opisanego na tym kole (dla R=1 : P=4).

Metoda bywa stosowana również w biznesie, a szczególnie w zarządzaniu projektami do zarządzania ryzykiem. Pozwala ocenić przy jakim czasie trwania projektu lub wysokości budżetu, osiągnie się określony poziom ryzykowności.

<u>Obrazki od lewej:</u>

Błędy całkowania maleją odwrotnie proporcjonalnie do pierwiastka z liczby próbek, czyli 1/sqrN
Całkowanie metodą Monte-Carlo działa na zasadzie porównywania losowych próbek z wartością funkcji
Aproksymacja liczby pi

Slajd 4

<u>Po 2 kropce:</u>
Co więcej, ponieważ rozmiar zmiennych reprezentujących wewnętrzny stan generatora jest ograniczony (zwykle decyzją programisty, do kilkudziesięciu lub kilkuset bitów; a rzadziej, po prostu rozmiarem pamięci komputera), i ponieważ w związku z tym może on znajdować się tylko w ograniczonej liczbie stanów, bez dostarczania nowych danych z zewnątrz musi po jakimś czasie dokonać pełnego cyklu i zacząć generować te same wartości. Teoretyczny limit długości cyklu wyrażony jest przez 2^{n}, gdzie n to liczba bitów przeznaczonych na przechowywanie stanu wewnętrznego. W praktyce, większość generatorów ma znacznie krótsze okresy

Szczególną klasę PRNG stanowią generatory uznane za bezpieczne do zastosowań kryptograficznych. Kryptografia opiera się na generatorach liczb pseudolosowych przede wszystkim w celu tworzenia unikalnych kluczy stałych oraz sesyjnych. Ze względu na fakt, że bezpieczeństwo komunikacji zależy od jakości klucza, od implementacji PRNG stosowanych w takich celach oczekuje się między innymi, że:

- Generowane wartości będą każdorazowo praktycznie nieprzewidywalne dla osób postronnych (np. przez wykorzystanie odpowiednich źródeł danych przy tworzeniu ziarna).

- Nie będzie możliwe ustalenie ziarna lub stanu wewnętrznego generatora na podstawie obserwacji dowolnie długiego ciągu wygenerowanych bitów.
- Znajomość dowolnej liczby wcześniej wygenerowanych bitów nie będzie wystarczała, by

  odgadnąć dowolny przyszły bit z prawdopodobieństwem istotnie wyższym od
- Dla wszystkich możliwych wartości ziarna, zachowana będzie pewna minimalna, dopasowana do zastosowania długość cyklu PRNG (aby uniknąć ponownego wykorzystania takiego samego klucza).

Uproszczony liniowy generator kongruencyjny (*Linear Congruential Generator*) określony jest algorytmem (a,b i m) to odpowiednio dobrane znane stałe):

stan początkowy to wartość ziarna

żeby wygenerować bit:

$$\text{nowy stan} = a \times \text{stary stan} + b \mod m$$
$$\text{wygenerowany bit} = \text{nowy stan} \mod 2$$

Slajd 5

przez Makoto Matsumoto i Takuji Nishimura[1].

Inną kwestią jest długi czas, który może zabrać przestawienie nielosowego stanu początkowego w stan wyjściowy, który spełnia testy losowości. Prosty generator Fibonacciego lub liniowy generator kongruencyjny startują dużo szybciej i mogą być[potrzebny przypis] używane do wyznaczania ziarna dla Mersenne Twister.

Algorytm Mersenne Twister otrzymał pewne krytyczne uwagi ze strony informatyków, szczególnie od George'a Marsaglia. Krytycy twierdzą, że choć jest dobry w generowaniu liczb pseudolosowych, to nie jest zbytnio elegancki i jest nazbyt skomplikowany w implementacji. Marsaglia podał kilka przykładów generatorów, które są mniej złożone i jak twierdzi mają znacząco większe okresy. Na przykład generator dopełniający mnożenie z przeniesieniem może mieć dłuższy okres – 1033000 – jest znacząco szybszy i zachowuje lepszą lub równie dobrą losowość[3][4].

The Mersenne Twister is used as default PRNG by the following software:

- Programming languages: Dyalog APL,[4] IDL,[5] R,[6] Ruby,[7] Free Pascal,[8] PHP,[9] Python (also available in NumPy, however the default was changed to PCG64 instead as of version 1.17[10]),[11][12][13], CMU Common Lisp,[14] Embeddable Common Lisp,[15] Steel Bank Common Lisp,[16]
- Linux libraries and software: GLib,[17] GNU Multiple Precision Arithmetic Library,[18] GNU Octave,[19] GNU Scientific Library,[20]
- Other: Microsoft Excel,[21] GAUSS,[22] gretl,[23] Stata.[24] SageMath,[25] Scilab,[26] Maple,[27] MATLAB ,[28]

It is also available in Apache Commons,[29] in the standard C++ library (since C++11),[30][31] and in Mathematica.[32] Add-on implementations are provided in many program libraries, including the Boost C++ Libraries,[33] the CUDA Library,[34] and the NAG Numerical Library.[35]

- Permissively-licensed and patent-free for all variants except CryptMT.

Slajd 6

Sekwencja Weyla – sekwencja wielokrotności irracjonalnej alfa: 0 alfa 2alfa 3 alfa która jest równoważna (equidistributed – równomierny rozkład) modulo 1. Innymi słowy, ciąg części ułamkowych każdego wyrazu będzie równomiernie rozłożony w przedziale **[0, 1].**

W komputerologii uzywana do generowania dyskretnego jednostajnego rozkładu (discrete uniform distribution). Zamiast używania liczby irracjonalnej, która nie może być obliczona na komputerze, używany jest stosunek 2 liczb całkowitych. Wybrana zostaje zmienna k, względnie pierwsza do liczby modulos m. Najcześciej liczba m jest potęgą 2, co sprawia że k jest nieparzysta. TAK SAMO JAK WYŻEJ obliczenie, ciąg rozłożony **[0,m).**

Termin wydaje się pochodzić z artykułu George'a Marsaglii „Xorshift RNGs". Następujący kod generuje co Marsaglia nazywa „Weyl sequence" d+= 362437, modulo m=2^32, ponieważ d ma 32bity.

Slajd 7

-

Slajd 8

-

Slajd 9

TestU01 – biblioteka zaimplementowana w C, która oferuje wiele możliwości testowania empirycznych losowości oraz generatorów liczb (pseudo)losowych (RNG). Stworzona w 2007. W bibliotece zaimplementowano kilka typów generatorów liczb losowych, w tym niektóre proponowane w literaturze, a niektóre spotykane w powszechnie używanym oprogramowaniu. Przedstawia ogólne implementacje klasycznych testów statystycznych dla generatorów liczb losowych, a także kilka innych proponowanych w literaturze oraz kilka oryginalnych.

Testy te można zastosować do generatorów predefiniowanych w bibliotece, generatorów zdefiniowanych przez użytkownika oraz strumieni liczb losowych przechowywanych w plikach.

Dostępne są również specjalne zestawy testów dla sekwencji jednolitych liczb losowych w [0,1] lub sekwencji bitowych. Dostępne są również podstawowe narzędzia do wykreślania wektorów punktów generowanych przez generatory.

TESTU01 oferuje kilka baterii testów, w tym „Small Crush" (na który składa się 10 testów), „Crush" (96 testów) i „Big Crush" (160 testów).

Dla prostego RNG, Small Crush zajmuje około 2 minut. Crush zajmuje około 1,7 godziny. Big Crush zajmuje około 4 godzin.

W przypadku bardziej złożonego RNG wszystkie te czasy zwiększają się dwukrotnie lub więcej. Dla porównania testy Dieharda trwają około 15 sekund.

TYLKO DLA 32 BITÓW i liczby w zakresie [0,1]

Diehard tests – 1995, 15 testów, George Marsaglia;

<u>Odległości urodzinowe</u>

Wybieramy losowe punkty na dużym przedziale. Odstępy między tymi punktami powinny mieć rozkład asymptotycznie wykładniczy. Nazwa pochodzi od paradoksu urodzin.

<u>Nakładające się permutacje</u>

Analizujemy ciągi pięciu kolejnych liczb losowych. 120 możliwych kolejności powinno wystąpić ze statystycznie równym prawdopodobieństwem.

<u>Szeregi macierzy</u>

Z pewnej liczby liczb losowych wybrać pewną liczbę bitów, które utworzą macierz nad {0,1}, a następnie wyznaczyć rangę tej macierzy. Policz rangi.

Małpie testy

Traktuj ciągi pewnej liczby bitów jako "słowa". Policzyć nakładające się na siebie słowa w strumieniu. Liczba "słów", które się nie pojawiają, powinna mieć znany rozkład. Nazwa wywodzi się z twierdzenia o nieskończonej małpie.

Liczenie jedynek

Policz bity 1 w każdym z kolejnych lub wybranych bajtów. Przekształć zliczenia na "litery" i policz wystąpienia pięcioliterowych "słów".

Test parkingu

W kwadracie o wymiarach 100×100 losowo rozmieść kółka jednostkowe. Kółko zostaje pomyślnie zaparkowane, jeśli nie zachodzi na istniejące już pomyślnie zaparkowane kółko. Po 12 000 próbach liczba pomyślnie zaparkowanych kół powinna mieć rozkład normalny.

Test minimalnej odległości

Umieść losowo 8000 punktów w kwadracie 10000×10000, a następnie znajdź minimalną odległość między tymi parami. Kwadrat tej odległości powinien mieć rozkład wykładniczy z pewną średnią.

Test losowych kul

Wybierz losowo 4000 punktów w sześcianie o krawędzi 1000. Na każdym punkcie wyśrodkuj kulę, której promień jest minimalną odległością od innego punktu. Najmniejsza objętość kuli powinna mieć rozkład wykładniczy z pewną średnią.

Test ściskania

Pomnóż 231 przez losowe zmienne na (0,1), aż dojdziesz do 1. Powtórz tę czynność 100000 razy. Liczba zmiennych potrzebna do osiągnięcia 1 powinna mieć pewien rozkład.

Test sum nakładających się

Wygeneruj długą sekwencję losowych liczb zmiennoprzecinkowych na (0,1). Dodaj sekwencje 100 kolejnych zmiennoprzecinkowych. Sumy powinny mieć rozkład normalny z charakterystyczną średnią i wariancją.

Test przebiegów

Wygenerować długą sekwencję losowych zmiennoprzecinkowych na (0,1). Policzyć przebiegi rosnące i malejące. Liczby powinny mieć określony rozkład.

Test kości

Rozegraj 200000 gier w kości, licząc wygrane i liczbę rzutów w każdej grze. Każde zliczenie powinno mieć pewien rozkład.

Slajd 10

Z pierwszych wersji języka pochodzi zasada braku rozróżniania małych i wielkich liter w słowach kluczowych języka oraz używanych zmiennych, a także bogate zasady tworzenia formatów zapisywanych i drukowanych danych.

Fortran dysponuje wielką liczbą bibliotek, które pozwalają rozwiązać praktycznie każde zadanie numeryczne. Najważniejsze przyczyny, z powodu których Fortran jest wykorzystywany i rozwijany do dziś, to szybkość obliczeń oraz wysoka wydajność kodu generowanego przez kompilatory Fortranu, wynikająca m.in. z jego długiej obecności na rynku programistycznym, znakomita skalowalność i przenośność oprogramowania (pomiędzy różnymi platformami sprzętowymi i systemami operacyjnymi), a także dostępność bibliotek dla programowania wieloprocesorowego i równoległego oraz bibliotek graficznych.