Numeri casuali o pseudocasuali

I generatori di numeri casuali (RNGs) risultano componente fondamentale per diverse applicazioni:

- Esperimenti statistici
–analisi di algoritmi
- Simulazione di sistemi stocastici
- Analisi numerica basata su metodi Monte-Carlo
- Algoritmi probabilistici
- Computer games
- Crittografia
- Protocolli di comunicazione sicuri
- Gambling machines
- Virtual Casino

Librerie software contengono Generatori di Numeri Casuali (RNGs)
- Microsoft Excel
- Visual Basic
- SUN Java
- .....

Generatori random non basati su algoritmi

Produzione di sequenze di numeri con apparenza di casualità.

RNG sono programmi basati - dovrebbero esserlo - su solide basi matematiche;

Disponibilità di una sorgente (RNG) capace di generare variabili casuali indipendenti
nell'intervallo [0; 1]
(i.i.d. U[0; 1]).

Altre distribuzioni (Normale, Chi-quadro, esponenziale, Poisson, logonormale, etc.) si ottengono mediante trasformazioni della sequenza uniforme.

Casualità e indipendenza

Un generatore di numeri casuali è un programma la cui sequenza di output è la simulazione (imitazione) del comportamento di una sequenza di variabili casuali indipendenti.

Il comportamento del programma è deterministico; a parità di seme iniziale la sequenza di output è la stessa.

Lancio di un dato 8 volte:

- 55555555 prima sequenza
- 23613564 seconda sequenza

Il risultato di diversi lanci di un dado può essere modellato come una sequenza di variabili casuali uniformemente distribuite su un insieme di interi da 1 a 6.
Ogni intero compreso tra 1 e 6 ha probabilità 1/6 di essere osservato in ciascun lancio indipendentemente dai risultati degli altri lanci.

Concetto astratto: nessun dado perfetto esiste in natura!

Cosa ci aspettiamo da un RNG progettato e realizzato per simulare i risultati del lancio di un dado:

Per un numero di generazioni elevato (long run):

- ciascun intero compreso tra 1 e 6 dovrebbe apparire con una frequenza ‘vicina’ al valore 1/6;

- ciascuna coppia dovrebbe apparire con una frequenza ‘vicina’ al valore 1/36;

- ciascuna tripla di numeri dovrebbe apparire con una frequenza ‘vicina’ al valore 1/216;

- .....

Queste proprietà non solo garantiscono che il dado virtuale è non distorto, ma anche che le generazioni consecutive di numeri non sono correlate.

Definizione
Un RNG può essere definito come una struttura (S;μ; f; U; g),
S è un insieme finito di stati;

μ è una distribuzione di probabilità su S utilizzata per selezionare uno stato iniziale (seme) s_0;

f: S →S è una funzione di transizione;

U è un insieme finito simboli di output;

g: S →U è la funzione di output.

I cambiamenti di stato sono determinati dalla ricorrenza $s_n = f(s_{n-1})$, per n > 0.

L'output al passo n è: $u_n = g(s_n)$ εU,
i valori u_n sono i nostri "numeri random" generati dal RNG.

S è un insieme finito :

- $s_{i+j} = s_i$,per qualche i≥0 e j>0
- $s_{n+j} = s_n$, $u_{n+j} = u_n$ per tutti gli n ≥ i

il più piccolo valore di j > 0 per cui si verifica la precedente condizione definisce il periodo ρ del generatore.

ρ deve essere inferiore alla cardinalità di S, inoltre se b bit sono usati per rappresentare lo stato, $\rho \leq 2^b$

RNG dovrebbero avere il periodo molto vicino all'upper bound.

Progetto di generatori e misure di uniformità

- periodo di lunghezza arbitraria (molto grande)
- Generazione efficiente dei numeri: veloce e uso di poche risorse;
- Sequenza riproducibile
- Portabilità del codice;
- Jumping nella sequenza efficiente

Esempio: generatore uniforme U[0,1]

$S_{i+1} = s_i + 1$, if $s_i \leq 2^{500} - 1$, $s_i = 0$ altrimenti;

$u_i = s_i / 2^{500}$

Generatore poco raccomandabile!

I valori successivi di u_i dovrebbero apparire uniformi e indipendenti, si dovrebbero 'comportare' come se il test dell'ipotesi (ipotesi nulla): "i valori u_i sono i.i.d. U[0,1]", fosse vero.

Per ciascun t ≥ 0, il vettore $\{(u_0, u_1, u_2, ..., u_{t-1})\}$,è uniformemente distribuito in

$[0,1]^t$ (test di discrepanza) si considerano gli insiemi:

$\Psi_t = \{(u_0, u_1, u_2, ..., u_{t-1})\}$

Molto più pratici e semplici i test empirici –essenzialmente test Goodness of fit.

Famiglie di RNG

- generatori basati su ricorrenze lineari

$x_i = (a_1 x_{i-1} + ...a_k x_{i-k}) \bmod m$

m, k > 0,

m è detto modulo e k rappresenta l'ordine della ricorrenza.

I coefficienti a_i appartengono a:

$Z_m = (0,1,2,.....m-1)$

Stato del generatore al passo i : $s_i = (x_{i-k+1}, ...x_i)$

Periodo massimo -m primo e a_i soddisfacenti certe relazioni - $\rho = m^k - 1$

Funzione di output: $u_i = x_i / m$

Caso particolare: k = 1
Generatore congruente moltiplicativo:

http://www.firstpr.com.au/dsp/rand31/p1192-park.pdf

$x_i = (a\ x_{i-1})$ mod m

condizioni di periodo massimo
m primo: ($m = 2^{31} - 1$)

a elemento primitivo modulo m

(a è un elemento primitivo modulo m se il più piccolo intero s per il quale $a_s - 1$ è divisibile per m è s = m-1):

alcuni valori:
a= 16807
a= 630360016

LCG16807 (a=16807 e $m = 2^{31} - 1$)

Codice C per l'algoritmo proposto nell'articolo di Park e Miller:

```
static int rnd_seed;

void set_rnd_seed (int new_seed)
{
    rnd_seed = new_seed;
}

int rand_int (void)
{
    int k1;
    int ix = rnd_seed;
     int a = 16807;
     int m = 2147483647;
     int q = 127773;
     int r = 2836;
     k1 = ix/q;
     ix = a*(ix –k1*q) – k1*r;
    if (ix < 0)
        ix += 2147483647;
    rnd_seed = ix;
    return rnd_seed;
}
```

Altri approcci:

Combinare generatori congruenti moltiplicativi :

http://www.iro.umontreal.ca/~lecuyer/myftp/papers/combmrg2.ps

$$x_{1,n} = (a_{1,1}\, x_{1,n-1} + a_{1,2}\, x_{1,n-2} \ldots\ldots a_{1,k}\, x_{1,n-k}) \bmod m_1$$

$$x_{2,n} = (a_{2,1}\, x_{2,n-1} + a_{2,2}\, x_{2,n-2} \ldots\ldots a_{2,k}\, x_{2,n-k}) \bmod m_2$$

.....

$$x_{j,n} = (a_{j,1}\, x_{j,n-1} + a_{j,2}\, x_{j,n-2} \ldots\ldots a_{2,k}\, x_{j,n-k}) \bmod m_j$$

Una particolare implementazione: (j=2, k=3)
Generatore MRG32k3a:

http://www.iro.umontreal.ca/~simardr/rng/MRG3 2k3a.c

$$x_{1,n} = (a_{1,1}\, x_{1,n-1} + a_{1,2}\, x_{1,n-2} + a_{1,3}\, x_{1,n-3}) \bmod m_1$$

$$x_{2,n} = (a_{2,1}\, x_{2,n-1} + a_{2,2}\, x_{2,n-2} + a_{2,3}\, x_{2,n-3}) \bmod m_2$$

$m_1 = 2^{32} - 209 = 4294967087$

$(a_{11}, a_{12}, a_{1,3}) = (0, 1403580, -810728)$

$m_2 = 2^{32} - 22853 = 4294944443$

$(a_{21}, a_{22}, a_{2,3}) = (527612, 0, -1370589)$

$z_n = (x_{1,n} - x_{2,n}) \bmod m_1$

$u_n = z_n / 4294967088$ se $z_n > 0$

$u_n = 4294967087 / 4294967088$ se $z_n = 0$

Periodo = $(m_1^3 - 1)(m_2^3 - 1)/2 = 3.1 \times 10^{57}$

Implementazione proposta da Pierre L'Ecuyer:

32-bits Random number generator U(0,1): MRG32k3a
Author: Pierre L'Ecuyer,
Source: Good Parameter Sets for Combined Multiple Recursive Random
Number Generators,
Shorter version in Operations Research,
47, 1 (1999), 159--164.

```
   ---------------------------------------------------
*/
#include "MRG32k3a.h"

#define norm 2.328306549295728e-10
#define m1   4294967087.0
#define m2   4294944443.0
#define a12     1403580.0
#define a13n     810728.0
#define a21      527612.0
#define a23n    1370589.0

/***
The seeds for s10, s11, s12 must be integers in [0, m1 - 1] and not all
0.
The seeds for s20, s21, s22 must be integers in [0, m2 - 1] and not all
0.
***/

#define SEED 12345

static double s10 = SEED, s11 = SEED, s12 = SEED,
              s20 = SEED, s21 = SEED, s22 = SEED;


double MRG32k3a (void)
{
   long k;
   double p1, p2;
   /* Component 1 */
   p1 = a12 * s11 - a13n * s10;
   k = p1 / m1;
```

```
    p1 -= k * m1;
    if (p1 < 0.0)
      p1 += m1;
    s10 = s11;
    s11 = s12;
    s12 = p1;

    /* Component 2 */
    p2 = a21 * s22 - a23n * s20;
    k = p2 / m2;
    p2 -= k * m2;
    if (p2 < 0.0)
      p2 += m2;
    s20 = s21;
    s21 = s22;
    s22 = p2;

    /* Combination */
    if (p1 <= p2)
      return ((p1 - p2 + m1) * norm);
    else
      return ((p1 - p2) * norm);
}
```

Generatore MarsenneTwister:

http://www.math.sci.hiroshima- u.ac.jp/~m-mat/MT/emt.html

Alcuni generatori: Java

$$x_{i+1} = (25214903917 x_i + 11) \bmod 2^{48}$$

$$u_i = (2^{27} \lfloor x_{2i} / 2^{22} \rfloor + \lfloor x_{2i+1} / 2^{21} \rfloor) / 2^{53}$$

Visual Basic

$$x_{i+1} = (1140671485 x_i + 12820163) \bmod 2^{24}$$

$$u_i = x_i / 2_{24}$$

http://random.mat.sbg.ac.at
http://statistik.wu-wien.ac.at/src/unuran/

Bibliography

Bratley, P., B. L. Fox, and L. E. Schrage. 1987. A Guide to Simulation. Second ed. New York: Springer-V erlag..8
Devroye, L. 1986. Non-Uniform Random Variate Generation. New York: Springer-Verlag.
Gentle, J. E. 1998. Random Number Generation and Monte Carlo Methods. New York: Springer. ed. P. Hellekalek and G. Larcher. 1998. Random and Quasi-Random Point Sets. volume 138 of Lecture Notes in Statistics. New York: Springer.
H.ormann, W. and J. Leydold. 2000. Automatic random variate generation for simulation input. In Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 675{682, Pistacaway, NJ. IEEE Press.
Knuth, D. E. 1998. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Third ed. Reading, Mass.: Addison-Wesley
Lagarias, J. C. 1993. Pseudorandom numbers. Statistical Science, 8(1), 31
.Law, A. M and W. D. Kelton. 2000. Simulation Modeling and Analysis. Third ed. New York: McGraw-Hill.
L'Ecuyer, P. 1994. Uniform random number generation. Annals of Operations Research, 53, 77.
L'Ecuyer, P. 1998. Random number generation. In Handbook of Simulation, ed. J. Banks, Wiley. chapter 4.
L'Ecuyer, P. 1999a. Good parameters and implementations for combined multiple recursive random number generators. Operations Research, 47(1).
L'Ecuyer, P. 1999b. Tables of maximally equidistributed combined LFSR generators. Mathematics of Computation, 68(225), 261.
L'Ecuyer, P and T. H. Andres. 1997. A random number generator based on the combination of four LCGs. Mathematics and Computers in Simulation, 44, 99{107.
L'Ecuyer, P. and R. Simard. 2000. On the performance of birthday spacings tests for certain families of random number generators. Mathematics and Computers in Simulation, 55(1{3), 131.

Marsaglia, G. 1985. A current view of random number generators. In Computer Science and Statistics, Sixteenth Symposium on the Interface, 3, North-Holland, Amsterdam. Elsevier Science Publishers. Matsumoto, M and T. Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation, 8(1).
Niederreiter, H. 1992. Random Number Generation and Quasi-Monte Carlo Methods. volume 63 of SIAM CBMS-NSF Regional Conference Series in Applied Mathematics. Philadelphia: SIAM. Tezuka, S. 1995. Uniform Random Numbers: Theory and Practice. Norwell, Mass.: Kluwer Academic Publishers.