Numeri (pseudo)random

Matteo Duranti

matteo.duranti@pg.infn.it

(cfr. Bertucci – Metodi Statistici Per L' Analisi Dati, Lez.6
Bertucci – Metodi Statistici Per L' Analisi Dati, Lez.7

https://it.wikipedia.org/wiki/Numeri_pseudo-casuali#Distribuzioni_non_uniformi

http://www.cplusplus.com/reference/cstdlib/srand/

https://www.gnu.org/software/gsl/manual/html_node/Random-Number-Generation.html

https://it.wikipedia.org/wiki/Trasformazione_di_Box-Muller
https://en.wikipedia.org/wiki/Linear_congruential_generator)

Numeri random

Un vero generatore di numeri random può essere realizzato solamente basandosi su un processo fisico random:

- conteggi da una sorgente radiottiva in un intervallo fisso di tempo;
- tempo fra l’arrivo di due raggi cosmici;
- fluttuazione nella differenza di potenziale ai capi di un resistore a causa del rumore termico;
- il numero che esce da un lancio di dadi;
- …

Questo è fattibile, e viene fatto, ma ovviamente non è pratico nè tantomeno veloce

generatore hardware di numeri casuali

(TRNG, true random number generator)

Numeri pseudo-random

Si utilizzano, allora, delle particolari <u>sequenze</u> di numeri che:

- sono generate da un algoritmo deterministico;
- hanno approssimativamente le stesse proprietà statistiche di una sequenza generata da un processo reale

generatore di numeri pseudo-random

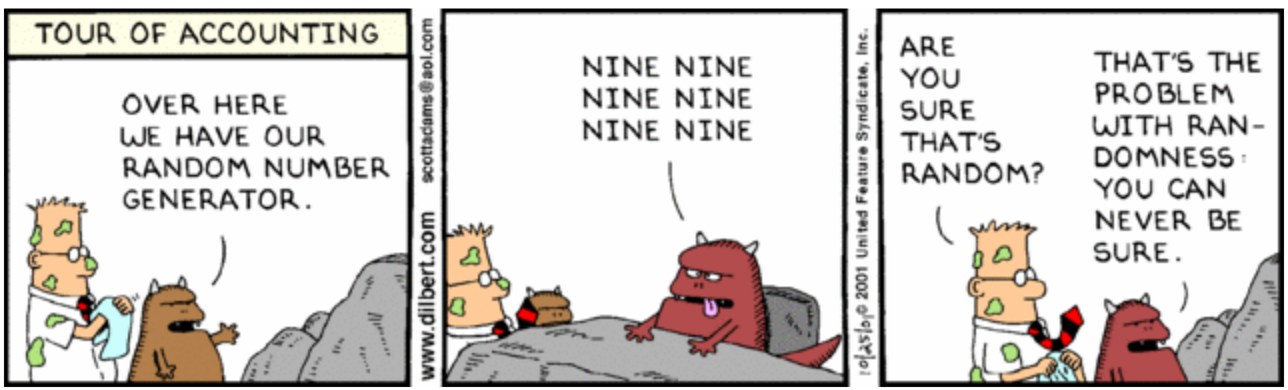
(PRNG, pseudo-random number generator)

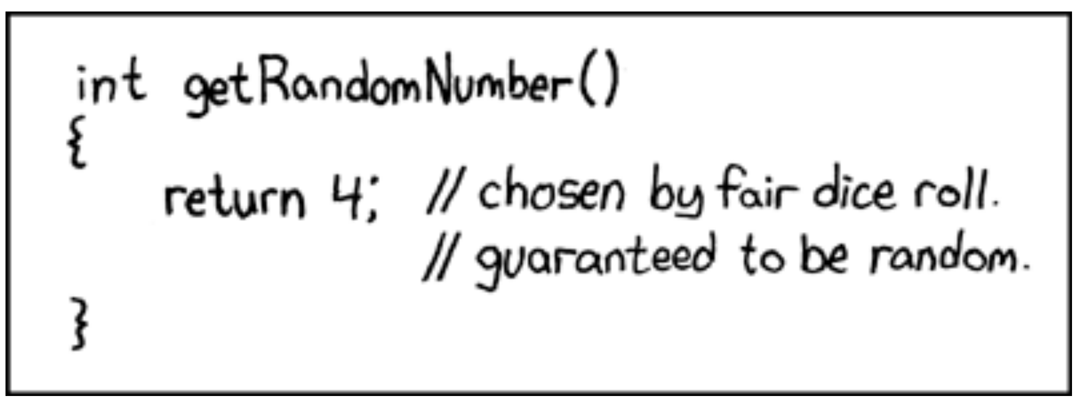
Prima di essere usato, un generatore deve essere inizializzato assegnando un opportuno valore a un parametro (o gruppo di parametri) numerico, che viene chiamato seme, *seed*. Ogni volta che si usa lo <u>stesso</u> seme, si otterrà sempre la <u>stessa</u> identica sequenza.

Numeri pseudo-random

Requisiti:

- sequenza “uniforme” e non biasata
  - eventi distribuiti secondo una funzione di distribuzione di probabilità (probability density function, p.d.f) predefinita (di solito uniforme in [0,1]);
  - indipendenza tra elementi successivi (i.e. non posso “prevedere” x_n da x_{n-1})





Numeri pseudo-random

Requisiti:

- sequenza “uniforme” e non biasata
  - eventi distribuiti secondo una funzione di distribuzione di probabilità (probability density function, p.d.f) predefinita (di solito uniforme in [0,1]);
  - indipendenza tra elementi successivi (i.e. non posso “prevedere” x_n da x_{n-1})
- sequenza lunga per evitare ripetizioni
  - “periodo”;
- computazionalmente efficiente e veloce;

Ad esempio in C:

```
void srand(unsigned seed);
int rand(void)
```

(è sufficiente #includere *stdlib.h*)

La prima funzione inizializza il seme della sequenza, la seconda estrae un numero intero equidistribuito tra *0* e *RAND_MAX*. Il valore di *RAND_MAX* dipende dall'implementazione; solitamente è 32767 (2^{15}-1} oppure 2147483647 (2^{31}-1)

Numeri pseudo-random

Ad esempio in C:

```
void srand(unsigned seed);
int rand(void)
```

```
/* srand example */
#include <stdio.h>        /* printf, NULL */
#include <stdlib.h>       /* srand, rand */
#include <time.h>         /* time */

int main ()
{
  srand (time(NULL));
  printf ("Random number: %d\n", rand()%100);

  return 0;
}
```

il *seed* è stato scelto con *time(0)*, cioè esso stesso ~ casuale. E' pratica comune ma rende il risultato non riproducibile.

Le librerie *gsl* includono un generatore di numeri pseudo-random con diversi algoritmi implementati

Numeri pseudo-random

Ad esempio in Bash, *$RANDOM* è una funzione (anche se sembra una costante!) che restituisce un intero in *[0,32767]*:

```
$> echo $RANDOM
19842
$> echo $RANDOM
22904
$> echo $RANDOM
10531
$>
$>
$> RANDOM=1
$> echo $RANDOM
16838
$> echo $RANDOM
5758
$> RANDOM=1
$> echo $RANDOM
16838
$> echo $RANDOM
5758
$>
$>
$> echo $(( $RANDOM % 10 + 1 ))
8
$> echo $(( $RANDOM % 10 + 1 ))
5
$> echo $(( $RANDOM % 10 + 1 ))
1
```

Numeri pseudo-random

Ad esempio in ROOT:

TRandom class: four different types of generators

TRandom: Linear Congruential Random Generator (BAD but FAST)
TRandom1: RANLUX (Luscher generator) period of 10^{171} (GOOD but SLOW)
TRandom2: Tausworthe generator, period of 10^{26} (GOOD for SMALL samples, FAST)
TRandom3: Mersenne and Twister, period of 10^{6000} (GOOD for large samples, not so FAST)

che ad esempio potete utilizzare:

```
#include "TRandom.h"

int main() {
  TRandom* trand = new TRandom(356); // 356 è la seed
  double rnd = trand->Uniform();
  return 0;
}
```

Numeri pseudo-random

Generatore di una sequenza pseudo-random a 16 bit:

```
unsigned int random;  // Variabile globale in cui è memorizzato il numero casuale
(16bit)

void randomNext(void) {
   // Aggiorna sequenza random
   // Algoritmo Polinomiale:
   // +> b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15
   // |     |  |     |                                     |
   // ------+--+-----+-------------------------------------+
   // carry = b1^b2^b4^b15
   // Pn+1=(Pn<<1)|carry
   unsigned short int randomtmp;  // Accumulo le operazioni ex-OR
   if (random==0) random++;  // N.B. : il seed dovrebbe essere != 0
   randomtmp=0;
   if ((unsigned short int)random&0x02) randomtmp=1;
   if ((unsigned short int)random&0x04) randomtmp^=1;
   if ((unsigned short int)random&0x10) randomtmp^=1;
   if ((unsigned short int)random&0x8000) randomtmp^=1;
   random = (unsigned short int)((random<<1)|randomtmp);
}
```

Distribuzione generica

- generalmente un algoritmo di generazione fornisce un numero *decimale* random fra 0 ed 1.
- alternativamente fornisce un numero *intero* fra *0* e *MAX* (come nel caso di `rand`).

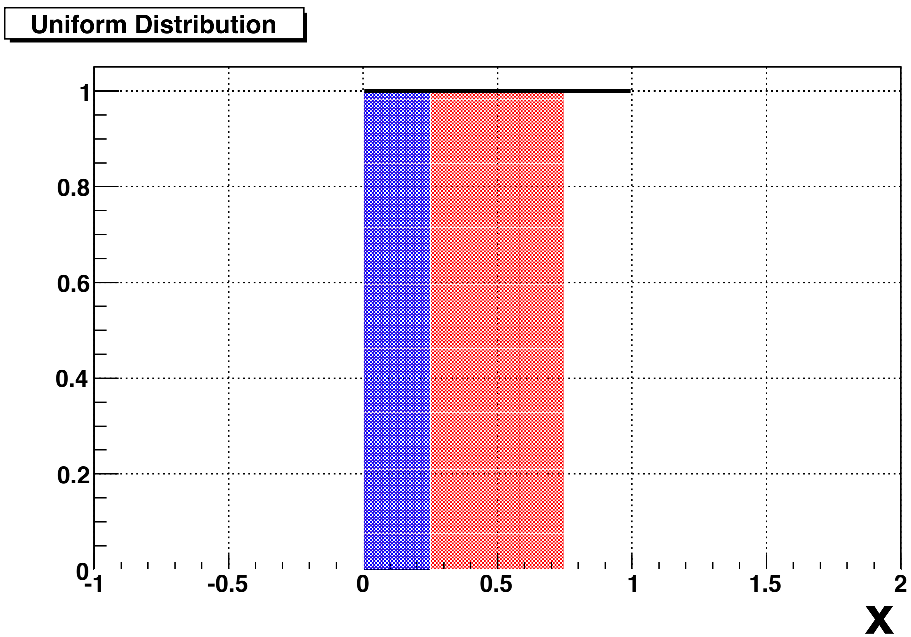
si può passare dal secondo caso al primo semplicemente dividendo l'intero ottenuto per *MAX*. Ovviamente, conoscendo *MAX* (ovvero la *precisione numerica* del numero decimale, si può anche fare l'inverso.

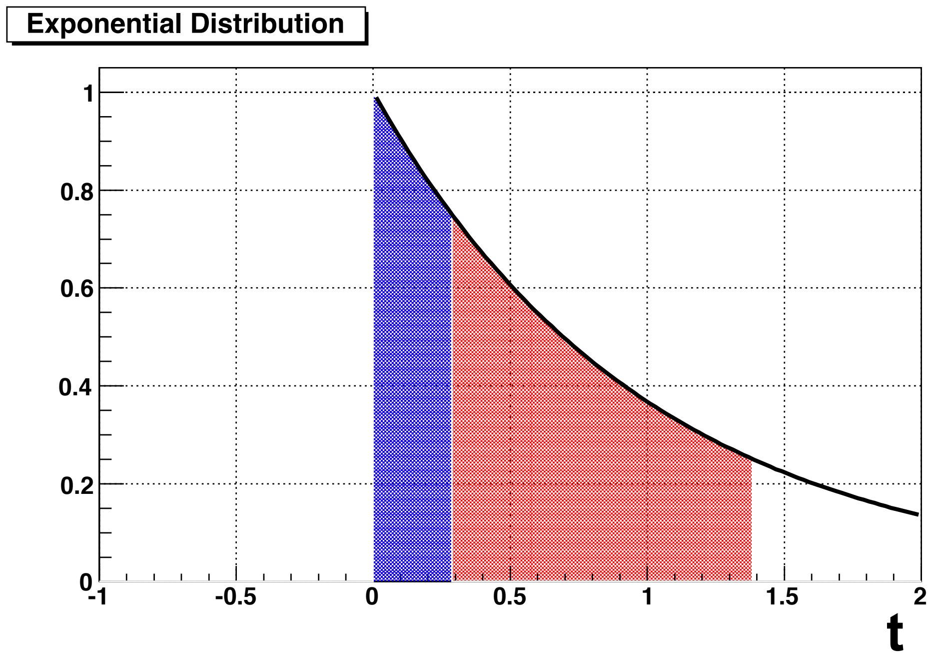
Ma se volessimo numeri random generati secondo una p.d.f. arbitraria?

La tecnica "standard" (ma non sempre percorribile) è quella della trasformazione di variabile (metodo di inversione, *I.T.M, inverse transformation method*)

Metodo di inversione

L'idea è quella di "rimappare" la distribuzione uniforme generata in quella desiderata:





- la zona blu contiene il 25% dell'integrale delle due funzioni;
- la zona rossa il 75%

Metodo di inversione

Il problema è quindi quello di trovare *b(R)* tale che:

$$\int_0^1 f(x)\mathrm{d}x = \int_a^b g(t)\mathrm{d}t$$

$$\int_0^R f(x)\mathrm{d}x = \int_a^{b(R)} g(t)\mathrm{d}t$$

con $0 \leq R \leq 1.$

Cioè se la distribuzione uniforme, fra *0* e *R*, contiene *N* eventi, allora anche la distribuzione arbitraria deve contenerne *N*, fra *a* e *b(R)*.

Cioè per la nostra uniforme, *f(x)=1*:

$$\int_0^R f(x)\mathrm{d}x = R = [G(t)]_a^{b(R)} = G(b(R)) - G(a)$$

Questo è possibile solamente se esiste la primitiva, *G(t)*, della nostra funzione arbitraria *g(t)* (e se sappiamo calcolarla…)

Metodo di inversione (esponenziale)

Fra 0 e *t(R)*:

$$\int_0^R f(x)\mathrm{d}x = R = \int_0^{t(R)} g(t)\mathrm{d}t$$

$$R = \int_0^{t(R)} \frac{1}{\tau} e^{-t/\tau}\mathrm{d}t = \left[-\frac{1}{\tau}\tau e^{-t/\tau}\right]_0^{t(R)} = -e^{-t(R)/\tau} + e^0 = -e^{-t(R)/\tau} + 1;$$

$$R = -e^{-t(R)/\tau} + 1 \quad \Rightarrow \quad e^{-t(R)/\tau} = 1 - R \quad \Rightarrow \quad -\frac{t(R)}{\tau} = log(1 - R);$$

$$\Rightarrow \quad t(R) = -\tau log(1 - R)$$

Quindi, generando una *R* uniforme fra *0* e *1*, avremo che *t(R)*, sarà distribuita esponenzialmente.

In realtà se R è fra 0 e 1, lo è anche 1-R, e quindi “basta”:

$$t(R) = -\tau log(R)$$

Metodo di inversione (gaussiana, Box-Muller)

Purtroppo la primitiva della gaussiana non esiste…

Non è quindi possibile generare, con il metodo dell’inversione, una <u>sola</u> variabile distribuita gaussianamente, a partire da una distribuita uniformemente (o in qualsiasi altro modo *invertibile* a uniforme).

Però possiamo generarne una *coppia*:

$$g(x,y) = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}}$$

che trasformata in coordinate polari $(x = \rho cos\phi, y = \rho sen\phi)$:

$$g(\rho,\phi) = \frac{1}{2\pi} e^{-\frac{\rho^2 cos^2\phi + \rho^2 sen^2\phi}{2}} \left|J(x,y)\right| = \frac{1}{2\pi} e^{-\frac{\rho^2}{2}} \left| \begin{array}{cc} \frac{\mathrm{d}x}{\mathrm{d}\rho} & \frac{\mathrm{d}x}{\mathrm{d}\phi} \\ \frac{\mathrm{d}y}{\mathrm{d}\rho} & \frac{\mathrm{d}y}{\mathrm{d}\phi} \end{array} \right|$$

$$g(\rho,\phi) = \frac{1}{2\pi} e^{-\frac{\rho^2}{2}} \left| \begin{array}{cc} cos\phi & -\rho sen\phi \\ sen\phi & \rho cos\phi \end{array} \right| = \frac{1}{2\pi} \rho e^{-\frac{\rho^2}{2}}$$

Metodo di inversione (gaussiana, Box-Muller)

$$g(\rho,\phi) = \frac{1}{2\pi}e^{-\frac{\rho^2}{2}} \begin{vmatrix} cos\phi & -\rho sen\phi \\ sen\phi & \rho cos\phi \end{vmatrix} = \frac{1}{2\pi}\rho e^{-\frac{\rho^2}{2}}$$

che è integrabile!

Ripetendo quindi il "gioco":

$$R = \int_0^{\rho(R)} \rho e^{-\frac{\rho^2}{2}} \mathrm{d}\rho = \left[-e^{-\frac{\rho^2}{2}}\right]_0^{\rho(R)} = \left(1 - e^{-\frac{\rho(R)^2}{2}}\right)$$

$$1 - R = e^{-\frac{\rho(R)^2}{2}} \quad \Rightarrow \quad \rho(R) = \sqrt{-2ln(1-R)}$$

Quindi generando *due* numeri casuali:

- *ρ* a partire da *R* (uniforme fra *0* e *1*)
- *Φ* uniforme fra *0* e *2π*

si ottengono *x* e *y*, $(x = \rho cos\phi, y = \rho sen\phi)$, distribuite gaussianamente

Metodo di inversione (gaussiana, Box-Muller)

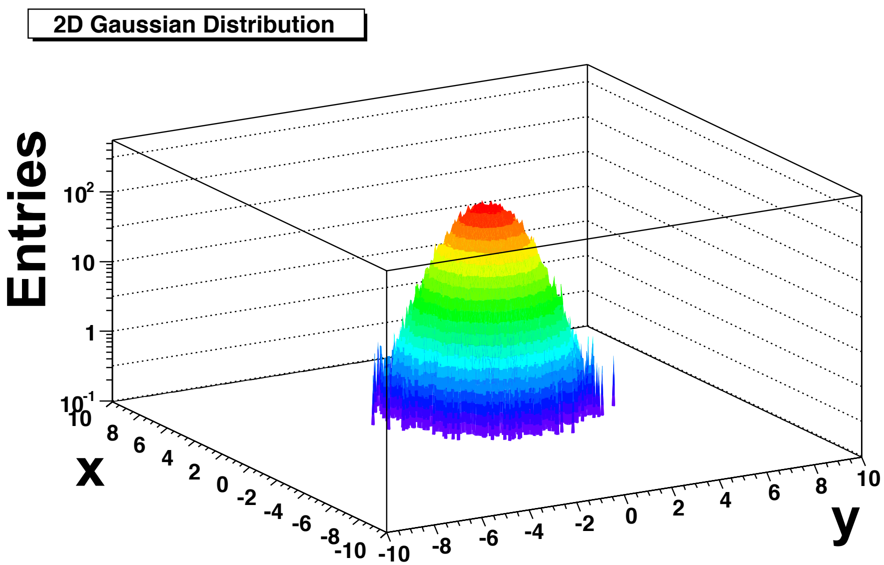
Quindi generando *due* numeri casuali:

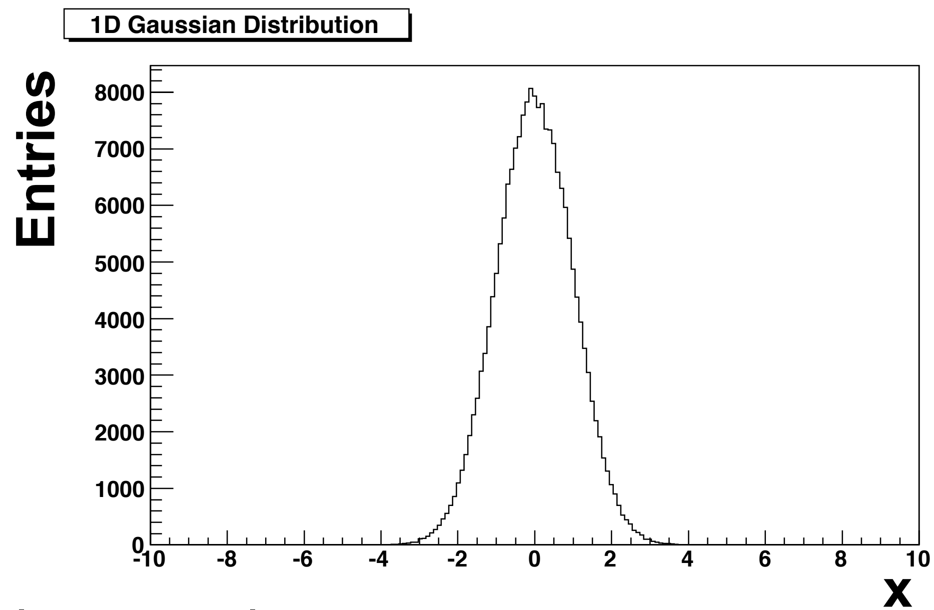
- ρ a partire da R (uniforme fra *0* e *1*)

$$\rho(R) = \sqrt{-2ln(1-R)}$$

- Φ uniforme fra *0* e 2π

si ottengono *x* e *y*, $(x = \rho cos\phi, y = \rho sen\phi)$, distribuite gaussianamente





… che però è efficiente computazionalmente solamente se ce ne servivano due…

Metodo di inversione (gaussiana, Box-Muller)

Quindi generando *due* numeri casuali:

- ρ a partire da *R* (uniforme fra *0* e *1*)

$$\rho(R) = \sqrt{-2ln(1-R)}$$

- Φ uniforme fra *0* e *2π*

si ottengono *x* e *y*, $(x = \rho cos\phi, y = \rho sen\phi)$, distribuite gaussianamente

TRandom di qualche anno fa:

```
https://root.cern.ch/root/html304/src/TRandom.cxx.html

Double_t TRandom::Gaus(Float_t mean, Float_t sigma) {
// Return a number distributed following a gaussian with mean and sigma
// Local variables
 Float_t x, y, z, result;
 y = Rndm();
 z = Rndm();
 x = z * 6.283185;
 result = mean + sigma*TMath::Sin(x)*TMath::Sqrt(-2*TMath::Log(y));
 return result;
}
```

Distribuzione generica

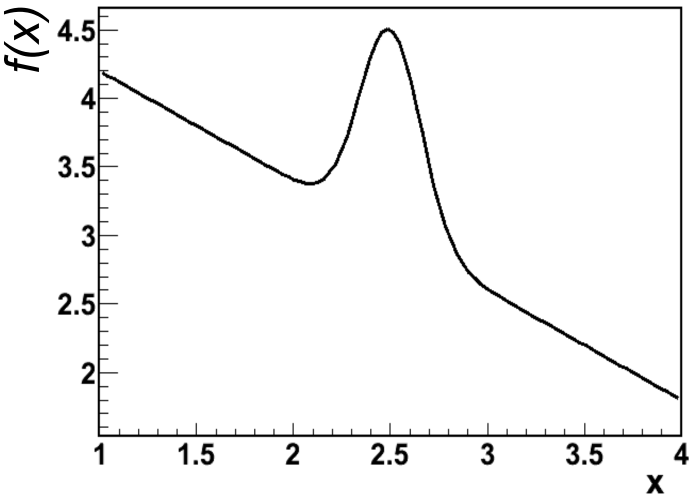
- generalmente un algoritmo di generazione fornisce un numero *decimale* random fra 0 ed 1.
- alternativamente fornisce un numero *intero* fra *0* e *MAX* (come nel caso di `rand`).

si può passare dal secondo caso al primo semplicemente dividendo l'intero ottenuto per *MAX*. Ovviamente, conoscendo *MAX* (ovvero la *precisione numerica* del numero decimale, si può anche fare l'inverso.

Se vogliamo numeri random generati secondo una p.d.f. arbitraria, si possono utilizzare <u>diverse tecniche</u>:

- metodo di inversione (*inverse transformation method, I.T.M.*);
- eventi ripesati;
- metodo di reiezione di Von Neuman (*hit and miss*);
- metodo composito

Eventi ripesati



Per riprodurre una *f(x)* nel range *[a,b]* a partire da una variabile *r*, generata uniforme in *[0,1]*, possiamo prima passare da r_i a x_i, che sarà uniformemente distribuito in *[a,b]*:

$$x_i = (b-a)r_i + a$$

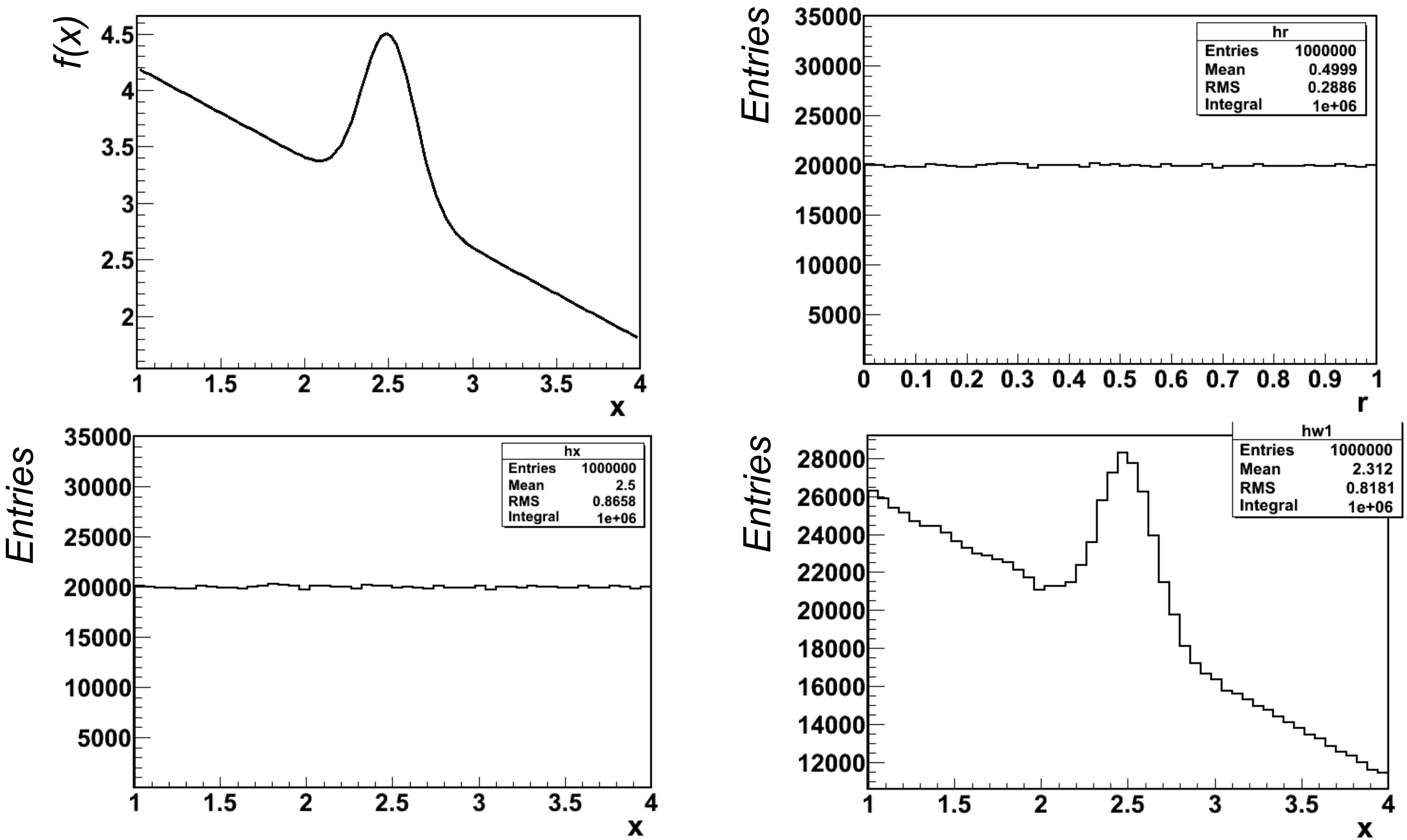
a questo punto calcoliamo i pesi come:

$$w = f(x)$$

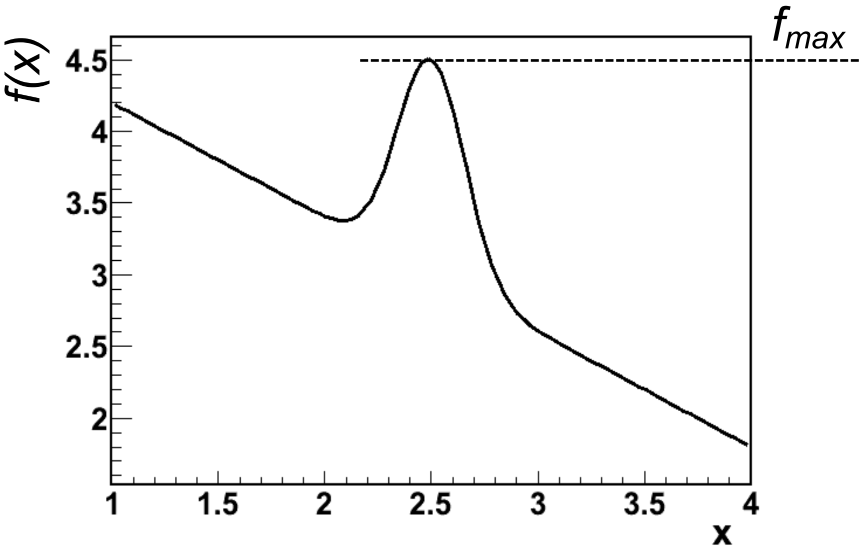
(tralasciando per ora considerazioni sulle costanti di normalizzazione). Praticamente "ripesiamo" (i.e. lo "consideriamo" più o meno, *h→Fill(x, ω)* in ROOT) ogni evento in base al valore della variabile generata. Le fluttuazioni <u>statistiche</u>, però, sono quelle che si hanno per una distribuzione <u>uniforme</u>

Eventi ripesati

I pesi li calcoliamo come: $w = f(x)$



Metodo di reiezione (hit and miss)



Si generano <u>due</u> variabili random:

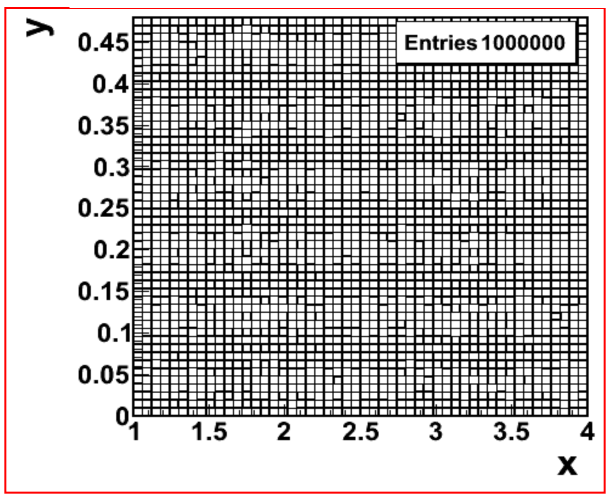
- x in $[a,b]$
- p in $[0, f_{max}]$

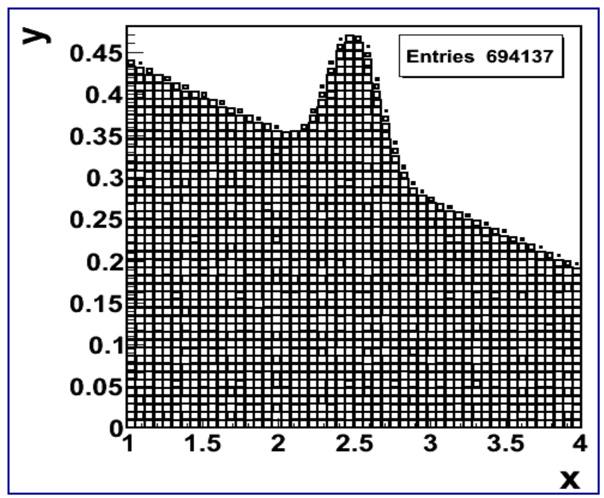
Un evento viene:

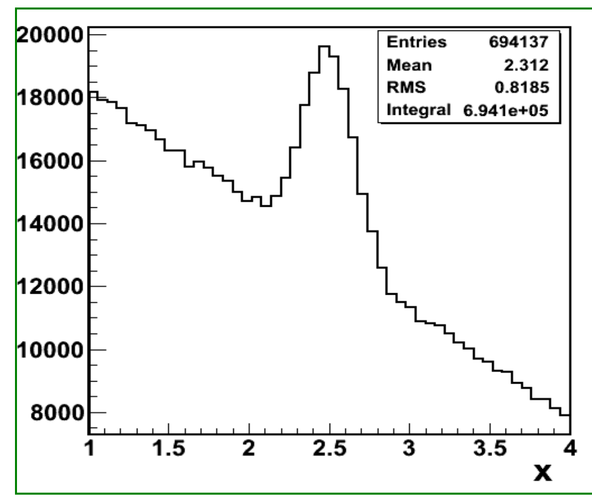
- accettato se $p \leq f(x)$
  scartato se $p > f(x)$

Quindi, ad esempio, in ROOT:

if (p<=f(x)) { h→Fill(x); }
else {}



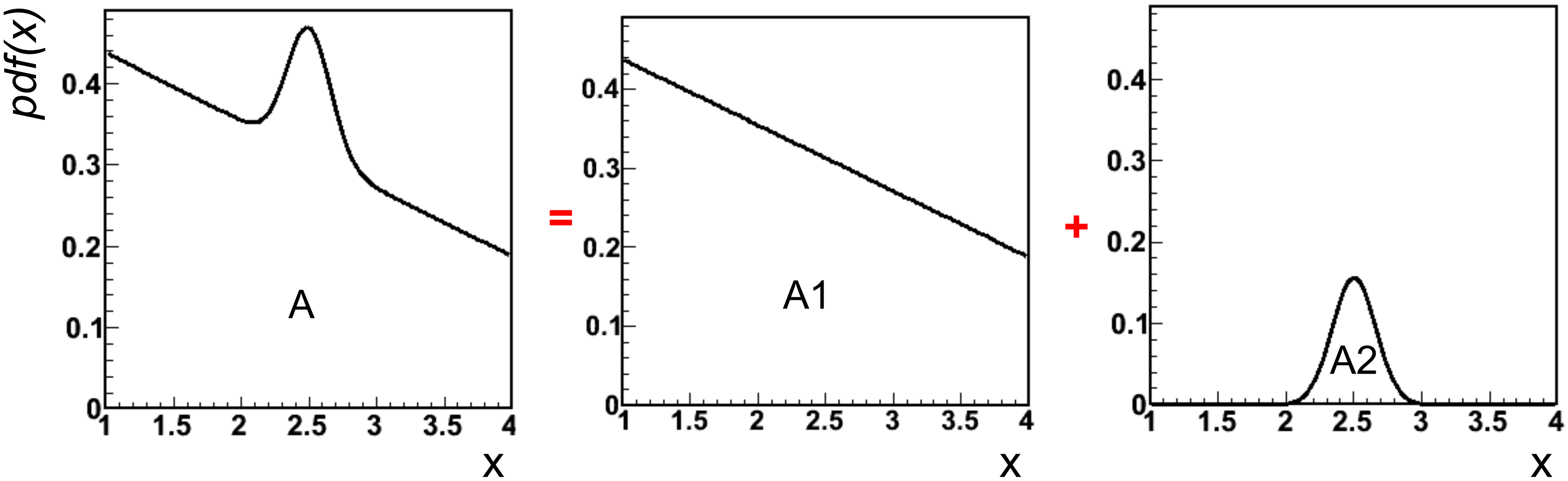




Le fluttuazioni statistiche sono "realistiche" ma il metodo è assolutamente non efficiente:

- è necessario generare *due* variabili random;
- molti eventi vengono "sprecati";

Metodo composito



Si “decompone” la p.d.f (di area *A*) in somma di p.d.f. e per ciascuna si utilizza il metodo più conveniente/adatto.

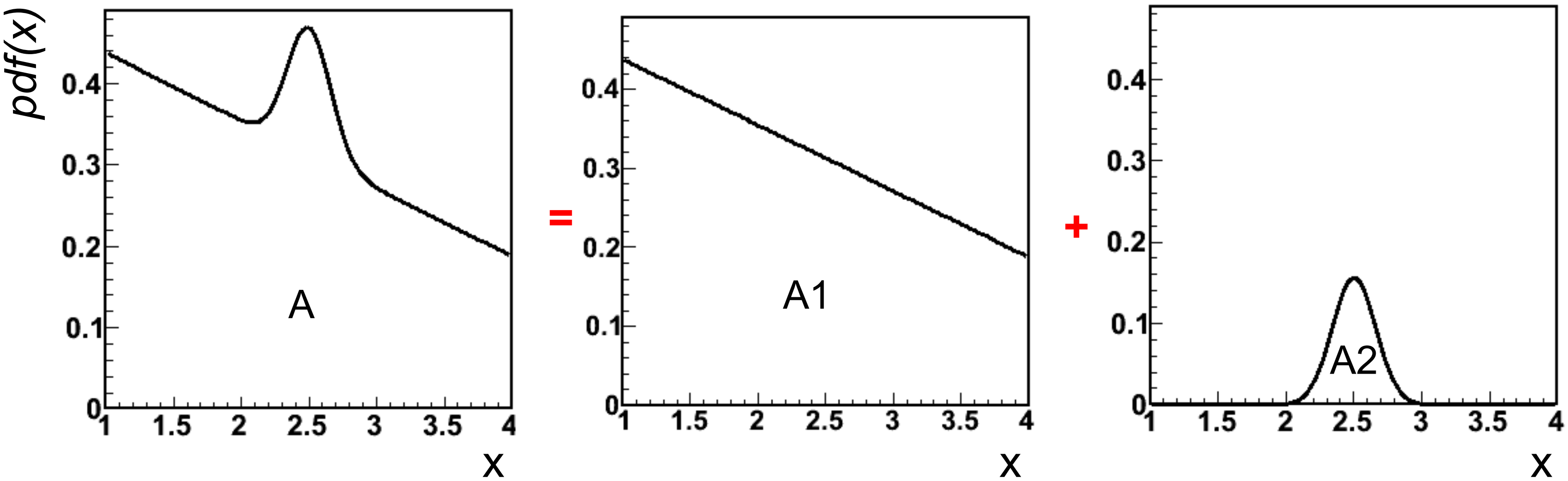
Nell’esempio sopra (retta + gaussiana) potremmo:

a) utilizzare il metodo di inversione per generare gli eventi della p.d.f. di area *A1*;

b) utilizzare un altro metodo (ad esempio Box-Muller, usando entrambe gli eventi, oppure uno degli altri) per generare gli eventi della p.d.f. di area *A2*;

Gli eventi generati in a) e quelli generati in b) dovranno essere nelle giuste proporzioni: per ogni *A* eventi se ne genereranno *A1* secondo a) e *A2* secondo b), con, ovviamente, *A=A1+A2.*

Metodo composito



Gli eventi generati in a) e quelli generati in b) dovranno essere nelle giuste proporzioni: per ogni *A* eventi se ne genereranno *A1* secondo a) e *A2* secondo b), con, ovviamente, *A=A1+A2.*

Randomizzare anche la sequenza con cui si generano gli eventi secondo a) e secondo b):

- evitare di generare *A* eventi costituiti da *A1* eventi a) <u>prima</u> e *A2* eventi b), <u>dopo</u> (i.e. un pattern assolutamente non casuale!)
- *A1* e *A2,* di ogni "bunch" di *A* eventi, andrebbero fatti fluttuare statisticamente

tipicamente mi "costa" un ulteriore numero random da generare

Generatore Lineare Congruente (Linear Congruent Generators, LCG)

La sequenza è definita da: $x_{n+1} = (\lambda x_n + \mu) \mod m$

cioè x_{n+1} è il resto della divisione per *m* di $(\lambda x_n + \mu)$

x_0 ($0 \le x_0 < m$) è il valore di partenza o ***seed***
λ ($0 < \lambda < m$) è il **moltiplicatore**
μ ($0 \le \mu < m$) è l'**incremento**
m ($m > 0$) è il **modulo**

tutti interi

se $\mu=0$: Multiplicative LCG

Questo genera numeri pseudo-random fra *0* and *m-1*.
Per convertirli in *[0-1]*, al solito, è sufficiente dividere per *m-1*.

→ la lunghezza della sequenza (i.e. la periodicita) è *m*, per tutte le seed, solo se le λ e la μ sono scelti accuratamente:

- *μ* e *m* sono co-primi (massimo comun divisore è *1*)
- *λ*-1 è divisibile per tutti i fattori primi di *m*
- λ-1 è divisibile per *4* se *m* è divisibile per *4*

$$x_0 = 1\,; \lambda = 9; \mu = 3; m = 32$$

$$\cancel{x_0 = 1\,; \lambda = 7; \mu = 3; m = 32}$$

Generatore Lineare Congruente (Linear Congruent Generators, LCG)

La sequenza è definita da: $x_{n+1} = (\lambda x_n + \mu) \mod m$

cioè x_{n+1} è il resto della divisione per *m* di $(\lambda x_n + \mu)$

$\mathbf{x_0}$ ($0 \leq \mathbf{x_0} < m$) è il valore di partenza o ***seed***
λ ($0 < \lambda < m$) è il **moltiplicatore**
μ ($0 \leq \mu < m$) è l'**incremento**
m ($m > 0$) è il **modulo**

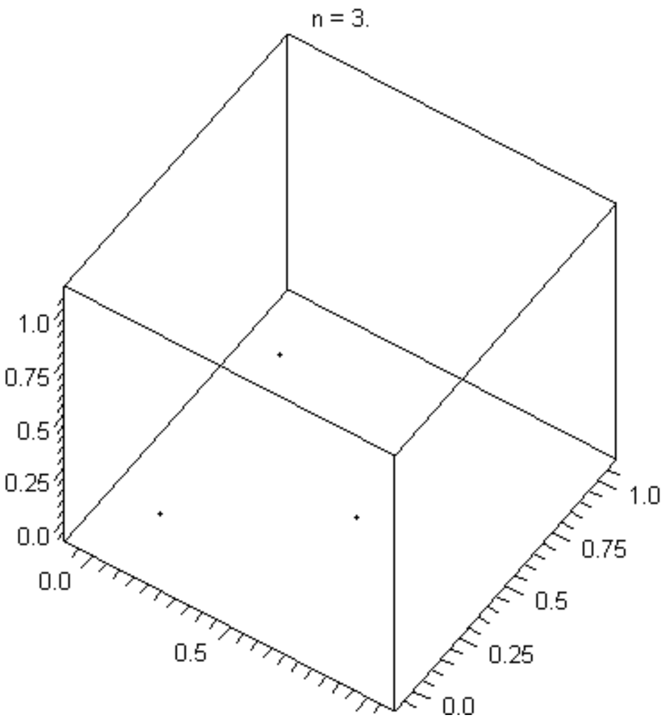
→ la lunghezza della sequenza (i.e. la periodicita) è *m*, per tutte le seed, solo se le λ e la μ sono scelti accuratamente:

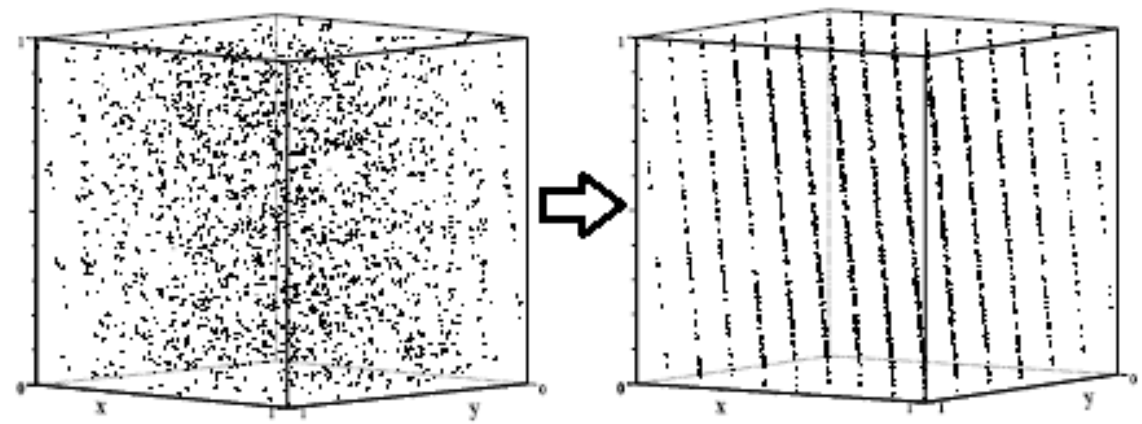
- μ e m sono co-primi (massimo comun divisore è *1*)
- λ-1 è divisibile per tutti i fattori primi di *m*
- λ-1 è divisibile per *4* se *m* è divisibile per *4*

Utilizzare come modulo una potenza del 2 produce un LCG computazionalmente molto efficiente: i bit più significativi non vengono nemmeno calcolati (scrivendo il codice nel giusto modo…)

Problemi dei generatori congruenti (Marsaglia effect)

Se un generatore è utilizzato per produrre numeri pseudo-random in uno spazio a *d* dimensioni, questi giaceranno su, al massimo, $(d!\,m)^{1/d}$ iperpiani (teorema di Marsaglia)





- di fatto i numeri sono generati con dei pattern specifici;
- l'effetto può essere mitigato usando un modulo, *m*, molto grande;