

# Numeri (pseudo)random

Matteo Duranti

[matteo.duranti@inf.na.cnr.it](mailto:matteo.duranti@inf.na.cnr.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](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://www.gnu.org/software/gsl/manual/html_node/Random-Number-Generation.html)

[https://it.wikipedia.org/wiki/Trasformazione\\_di\\_Box-Muller](https://it.wikipedia.org/wiki/Trasformazione_di_Box-Muller)

[https://en.wikipedia.org/wiki/Linear\\_congruential\\_generator](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 sequenze di numeri che:

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

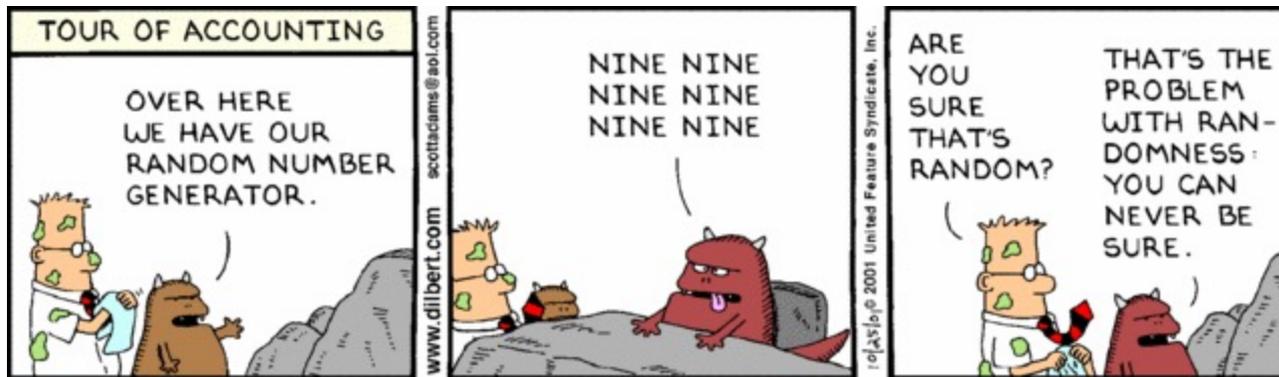
**generatore di numeri pseudo-random  
(PRNG, pseudo-random number generator)**

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 stesso seme, si otterrà sempre la stessa 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  $x_{n-1}$ )



```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

# 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 *gs/* 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.

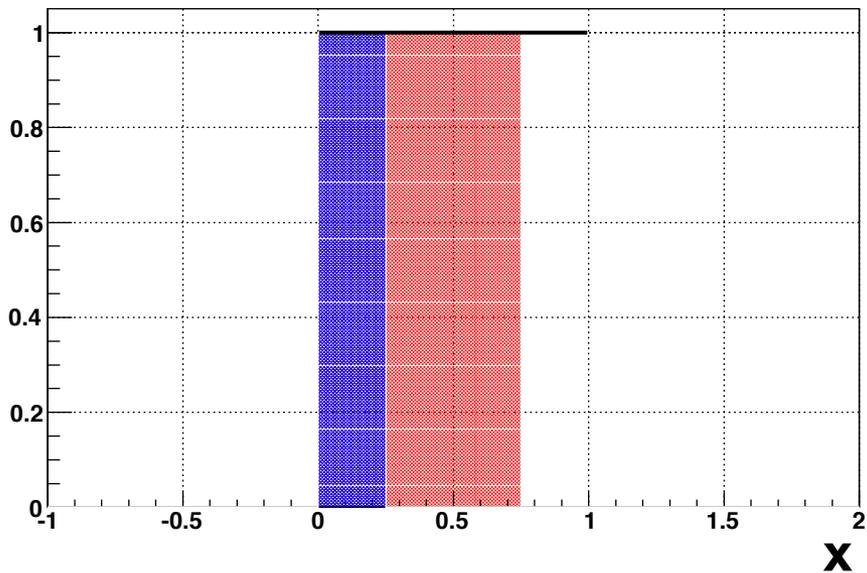
Ma se volessimo numeri random generati secondo una p.d.f. arbitraria?

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

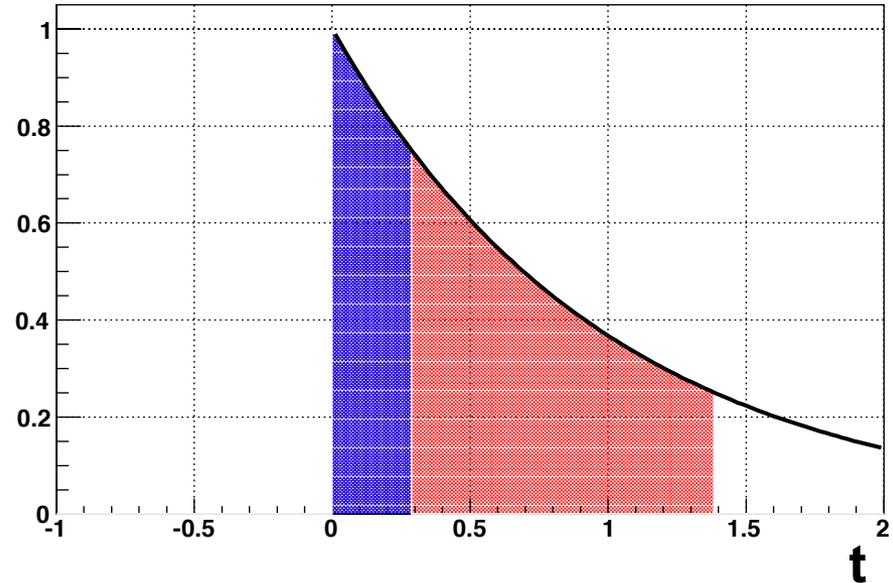
# Metodo di inversione

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

Uniform Distribution



Exponential Distribution



- 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:

$$\int_0^1 f(x)dx = \int_a^b g(t)dt$$

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

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,  $f(x)=1$ :

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

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  $t(R)$ :

$$\int_0^R f(x)dx = R = \int_0^{t(R)} g(t)dt$$

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

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

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

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 sola 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 ( $x = \rho \cos\phi, y = \rho \sin\phi$ ):

$$g(\rho, \phi) = \frac{1}{2\pi} e^{-\frac{\rho^2 \cos^2 \phi + \rho^2 \sin^2 \phi}{2}} |J(x, y)| = \frac{1}{2\pi} e^{-\frac{\rho^2}{2}} \begin{vmatrix} \frac{dx}{d\rho} & \frac{dx}{d\phi} \\ \frac{dy}{d\rho} & \frac{dy}{d\phi} \end{vmatrix}$$

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

# Metodo di inversione (gaussiana, Box-Muller)

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

che è integrabile!

Ripetendo quindi il “gioco”:

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

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

Quindi generando *due* numeri casuali:

- $\rho$  a partire da  $R$  (uniforme fra 0 e 1)
- $\Phi$  uniforme fra 0 e  $2\pi$

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

# Metodo di inversione (gaussiana, Box-Muller)

Quindi generando *due* numeri casuali:

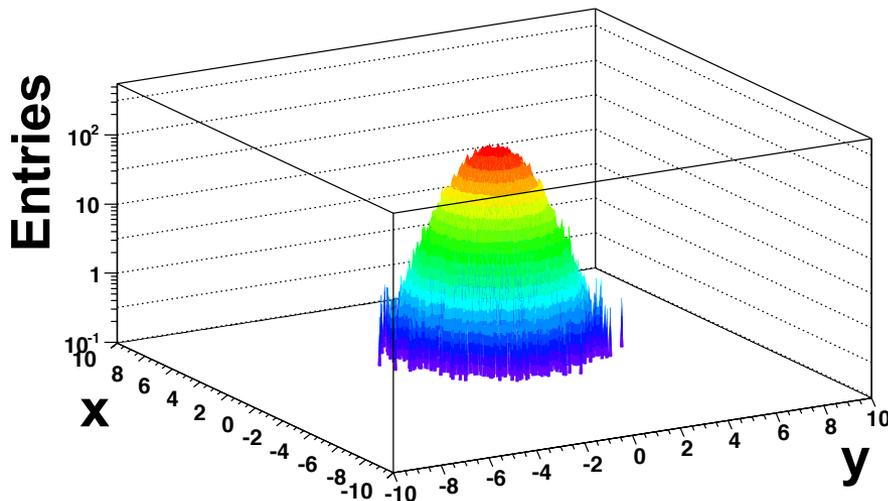
- $\rho$  a partire da  $R$  (uniforme fra 0 e 1)

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

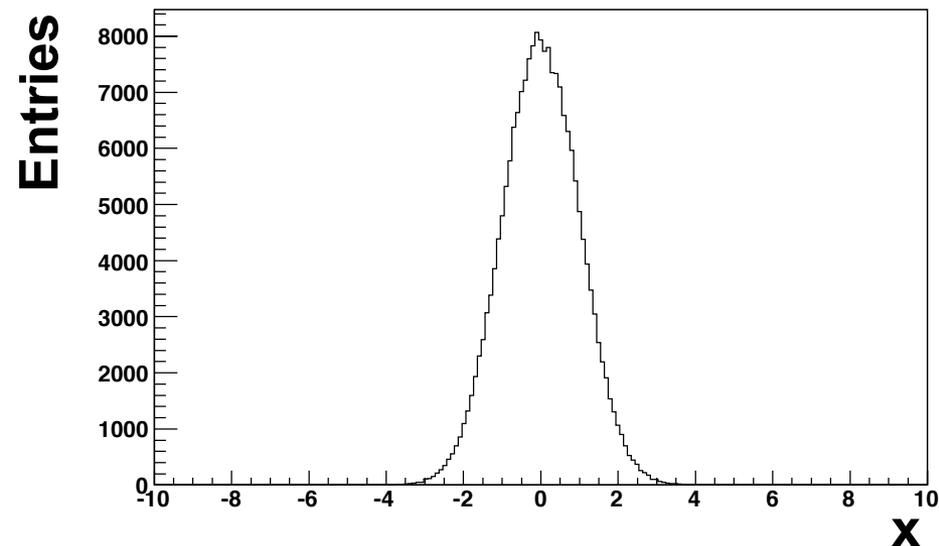
- $\Phi$  uniforme fra 0 e  $2\pi$

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

2D Gaussian Distribution



1D Gaussian Distribution



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

# Metodo di inversione (gaussiana, Box-Muller)

Quindi generando *due* numeri casuali:

- $\rho$  a partire da  $R$  (uniforme fra 0 e 1)

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

- $\Phi$  uniforme fra 0 e  $2\pi$

si ottengono  $x$  e  $y$ , ( $x = \rho\cos\phi$ ,  $y = \rho\sin\phi$ ), 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;  
}
```

# Metodo di inversione (gaussiana, Box-Muller)

Quindi generando *due* numeri casuali:

- $\rho$  a partire da  $R$  (uniforme fra 0 e 1)

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

- $\Phi$  uniforme fra 0 e  $2\pi$

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

◆ Gaus() (*TRandom* "moderno")

```
Double_t TRandom::Gaus ( Double_t mean = 0,  
                        Double_t sigma = 1  
                        )
```

virtual

Samples a random number from the standard Normal (Gaussian) Distribution with the given mean and sigma.

Uses the Acceptance-complement ratio from W. Hoermann and G. Derflinger This is one of the fastest existing method for generating normal random variables. It is a factor 2/3 faster than the polar (Box-Muller) method used in the previous version of **TRandom::Gaus**. The speed is comparable to the Ziggurat method (from Marsaglia) implemented for example in GSL and available in the MathMore library.

REFERENCE: - W. Hoermann and G. Derflinger (1990): The ACR Method for generating normal random variables, OR Spektrum 12 (1990), 181-185.

Implementation taken from UNURAN (c) 2000 W. Hoermann & J. Leydold, Institut f. Statistik, WU Wien

Definition at line 263 of file **TRandom.cxx**.

# Distribuzione generica

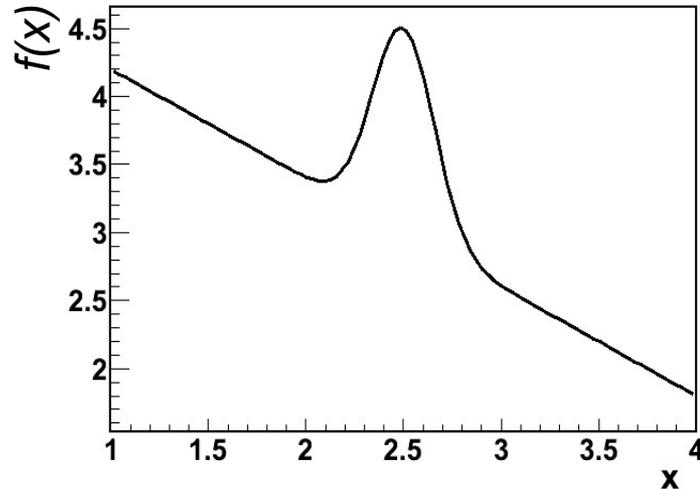
- generalmente un algoritmo di generazione fornisce un numero *decimale* random fra 0 ed 1.
- alternativamente fornisce un numero *intero* fra 0 e *MAX* (come nel caso di `rand`).

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 diverse tecniche:

- 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:

$$w = f(x)$$

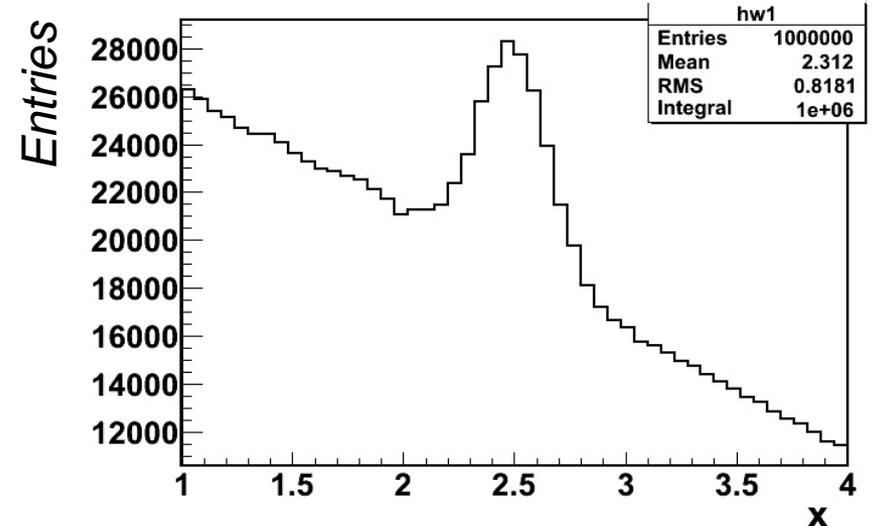
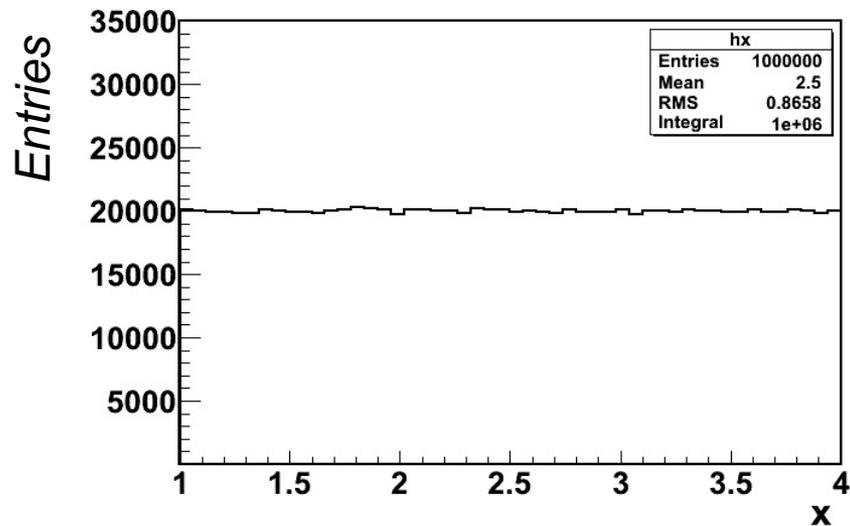
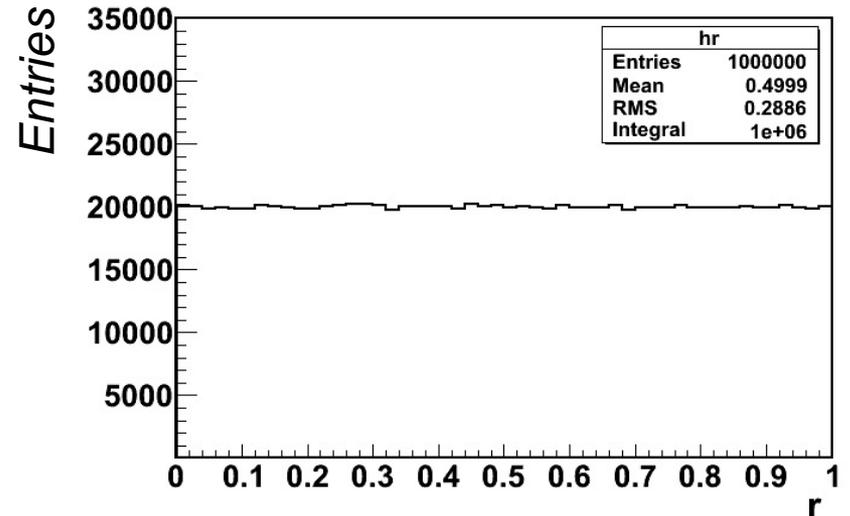
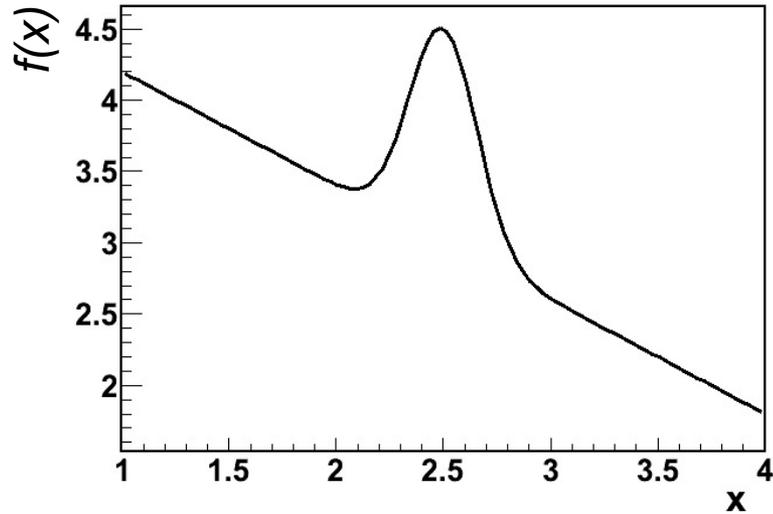
(tralasciando per ora considerazioni sulle costanti di normalizzazione).

Praticamente “ripesiamo” (i.e. lo “consideriamo” più o meno,  $h \rightarrow \text{Fill}(x, \omega)$  in ROOT) ogni evento in base al valore della variabile generata.

Le fluttuazioni statistiche, però, sono quelle che si hanno per una distribuzione uniforme

# Eventi ripesati

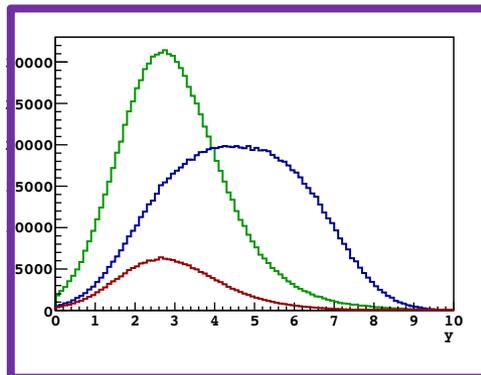
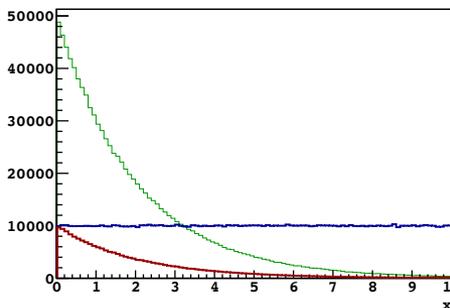
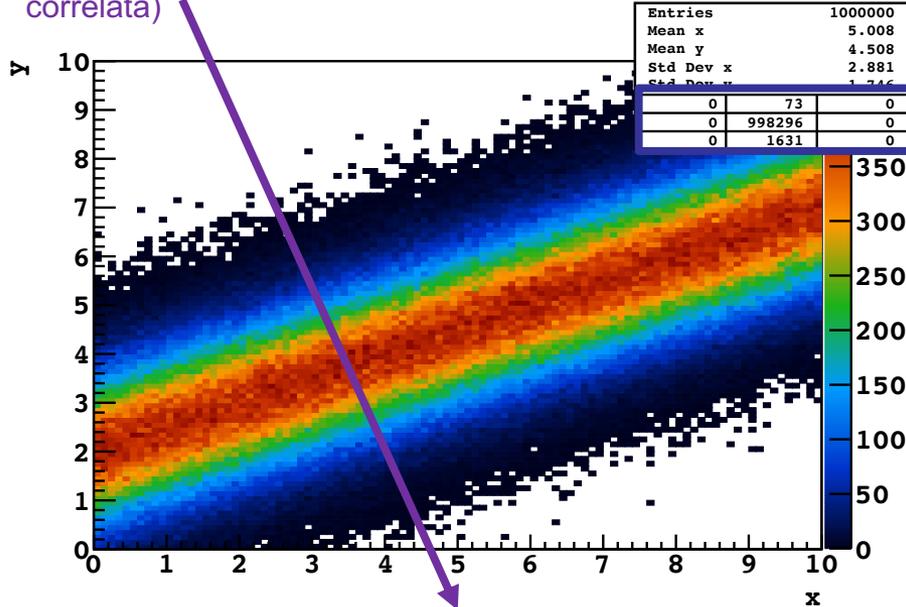
I pesi li calcoliamo come:  $w = f(x)$



# Eventi ripesati

- la variabile  $x$  è presa da una Uniforme
- la variabile  $y$  è presa da una Gaussiana attorno ad una retta  $y = x/2 + 2$

la distribuzione  $y$  (sotto studio) cambia in base a come sono distribuiti gli eventi nella variabile  $x$  (indipendente, ma correlata)



```
macro_2D.C

TH2F* h = new TH2F("h", "h", 100, 0, 10, 100, 0, 10);
h->GetXaxis()->SetTitle("x");
h->GetYaxis()->SetTitle("y");

TH2F* h2 = new TH2F("h2", "h2", 100, 0, 10, 100, 0, 10);
h2->GetXaxis()->SetTitle("x");
h2->GetYaxis()->SetTitle("y");

TH2F* h3 = new TH2F("h3", "h3", 100, 0, 10, 100, 0, 10);
h3->GetXaxis()->SetTitle("x");
h3->GetYaxis()->SetTitle("y");

for (int ii=0; ii<1000000; ii++) {
    double x = gRandom->Uniform(0, 10);
    double y = gRandom->Gaus(0.5*x+2, 1);
    h->Fill(x, y);
    h2->Fill(x, y, TMath::Exp(-x/2.0));
    double x3 = gRandom->Exp(2);
    double y3 = gRandom->Gaus(0.5*x3+2, 1);
    h3->Fill(x3, y3);
}

TCanvas* a = new TCanvas();
h->Draw("colz");

TCanvas* a2 = new TCanvas();
h2->Draw("colz");

TCanvas* a3 = new TCanvas();
h3->Draw("colz");

TCanvas* b = new TCanvas();
TH1* h_px = h->ProjectionX();
TH1* h2_px = h2->ProjectionX();
TH1* h3_px = h3->ProjectionX();
h_px->SetLineColor(kBlue+2);
h2_px->SetLineColor(kRed+2);
h3_px->SetLineColor(kGreen+2);
h_px->SetLineWidth(3);
h2_px->SetLineWidth(3);
h3_px->Draw("hist");
h_px->Draw("samehist");
h2_px->Draw("samehist");

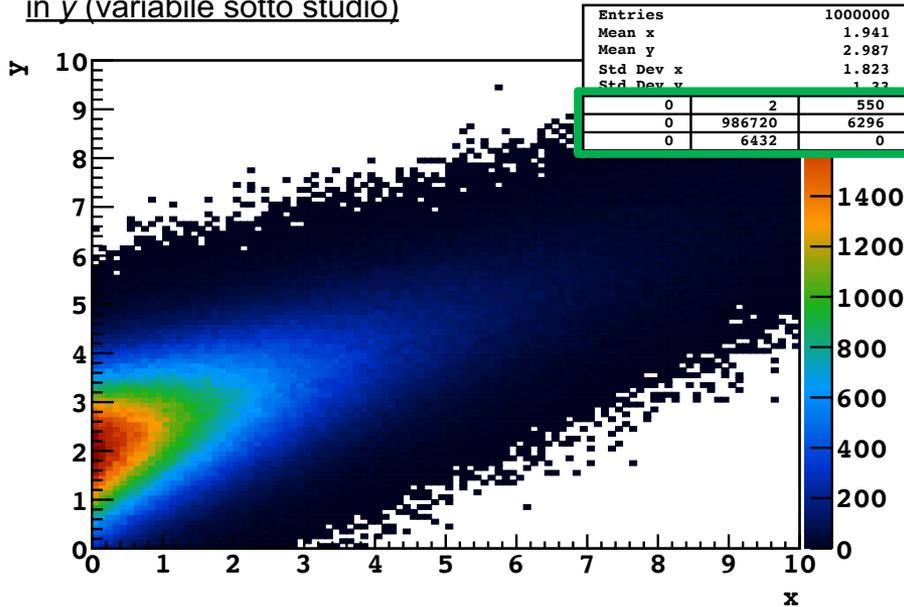
TCanvas* c = new TCanvas();
TH1* h_py = h->ProjectionY();
TH1* h2_py = h2->ProjectionY();
TH1* h3_py = h3->ProjectionY();
h_py->SetLineColor(kBlue+2);
h2_py->SetLineColor(kRed+2);
h3_py->SetLineColor(kGreen+2);
h_py->SetLineWidth(3);
h2_py->SetLineWidth(3);
h3_py->SetLineWidth(3);
h3_py->Draw("hist");
h_py->Draw("samehist");
h2_py->Draw("samehist");
}
```

U:--- macro\_2D.C All L1 (C/\*l Abbrev)

# Eventi ripesati

- la variabile  $x$  è presa da una Esponenziale
- la variabile  $y$  è presa da una Gaussiana attorno ad una retta  $y = x/2 + 2$

le distribuzioni verdi hanno lo stesso numero di eventi (i.e. stesso integrale) delle blu, ma differente forma, in particolare in  $y$  (variabile sotto studio)



```

macro_2D.C

TH2F* h = new TH2F("h", "h", 100, 0, 10, 100, 0, 10);
h->GetXaxis()->SetTitle("x");
h->GetYaxis()->SetTitle("y");

TH2F* h2 = new TH2F("h2", "h2", 100, 0, 10, 100, 0, 10);
h2->GetXaxis()->SetTitle("x");
h2->GetYaxis()->SetTitle("y");

TH2F* h3 = new TH2F("h3", "h3", 100, 0, 10, 100, 0, 10);
h3->GetXaxis()->SetTitle("x");
h3->GetYaxis()->SetTitle("y");

for (int ii=0; ii<1000000; ii++) {
    double x = gRandom->Uniform(0, 10);
    double y = gRandom->Gaus(0.5*x+2, 1);
    h->Fill(x, y);
    h2->Fill(x, y, TMath::Exp(-x/2.0));
    double x3 = gRandom->Exp(2);
    double y3 = gRandom->Gaus(0.5*x3+2, 1);
    h3->Fill(x3, y3);
}

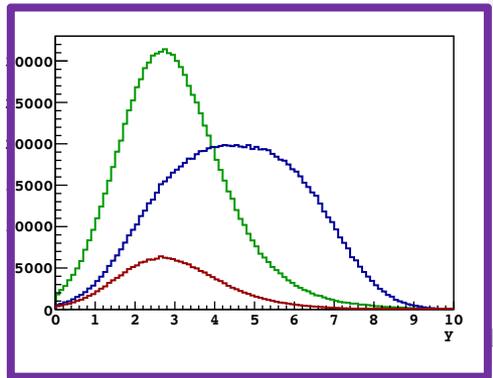
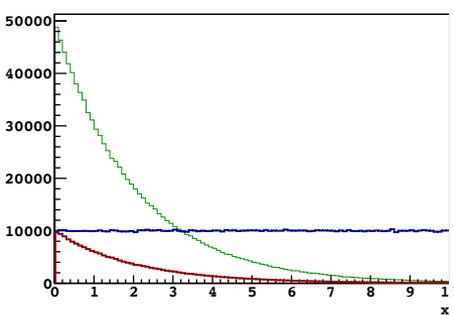
TCanvas* a = new TCanvas();
h->Draw("colz");

TCanvas* a2 = new TCanvas();
h2->Draw("colz");

TCanvas* a3 = new TCanvas();
h3->Draw("colz");

TCanvas* b = new TCanvas();
TH1* h_px = h->ProjectionX();
TH1* h2_px = h2->ProjectionX();
TH1* h3_px = h3->ProjectionX();
h_px->SetLineColor(kBlue+2);
h2_px->SetLineColor(kRed+2);
h3_px->SetLineColor(kGreen+2);
h_px->SetLineWidth(3);
h2_px->SetLineWidth(3);
h3_px->Draw("hist");
h_px->Draw("samehist");
h2_px->Draw("samehist");

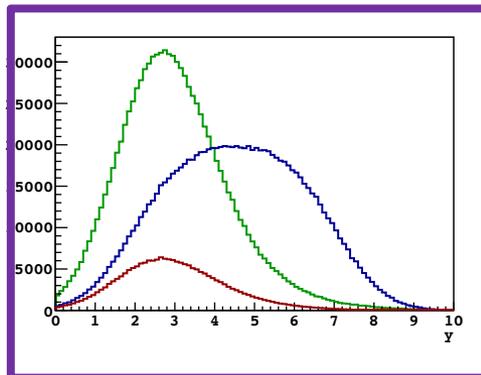
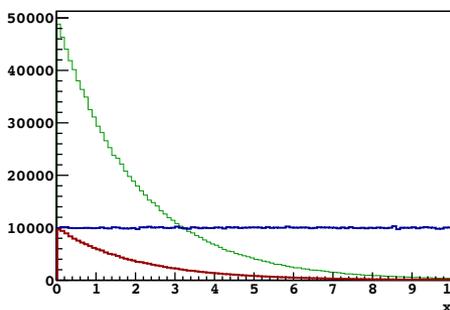
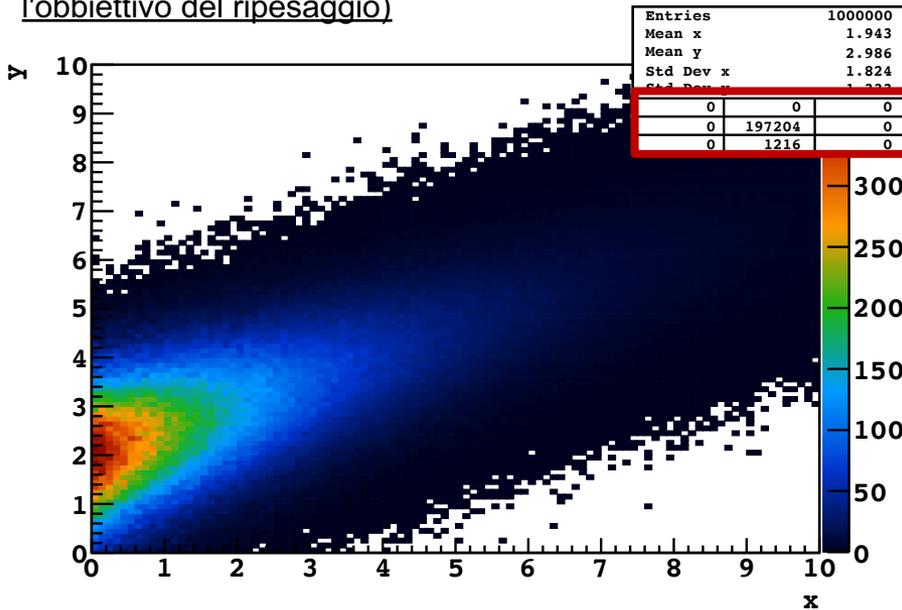
TCanvas* c = new TCanvas();
TH1* h_py = h->ProjectionY();
TH1* h2_py = h2->ProjectionY();
TH1* h3_py = h3->ProjectionY();
h_py->SetLineColor(kBlue+2);
h2_py->SetLineColor(kRed+2);
h3_py->SetLineColor(kGreen+2);
h_py->SetLineWidth(3);
h2_py->SetLineWidth(3);
h3_py->SetLineWidth(3);
h3_py->Draw("hist");
h_py->Draw("samehist");
h2_py->Draw("samehist");
}
    
```



# Eventi ripesati

- la variabile  $x$  è presa da una Uniforme
- la variabile  $y$  è presa da una Gaussiana attorno ad una retta  $y = x/2 + 2$
- gli eventi sono ripesati secondo un'Esponenziale funzione di  $x$

le distribuzioni rosse sono sempre sotto le blu (eventi ripesati con  $\text{Exp}(-x/2)$ , i.e.  $< 1$ ) ma identiche alle verdi (che è proprio l'obbiettivo del ripesaggio)



```
macro_2D.C

TH2F* h = new TH2F("h", "h", 100, 0, 10, 100, 0, 10);
h->GetXaxis()->SetTitle("x");
h->GetYaxis()->SetTitle("y");

TH2F* h2 = new TH2F("h2", "h2", 100, 0, 10, 100, 0, 10);
h2->GetXaxis()->SetTitle("x");
h2->GetYaxis()->SetTitle("y");

TH2F* h3 = new TH2F("h3", "h3", 100, 0, 10, 100, 0, 10);
h3->GetXaxis()->SetTitle("x");
h3->GetYaxis()->SetTitle("y");

for (int ii=0; ii<1000000; ii++) {
    double x = gRandom->Uniform(0, 10);
    double y = gRandom->Gaus(0.5*x+2, 1);
    h->Fill(x, y);
    h2->Fill(x, y, TMath::Exp(-x/2.0));
    double x3 = gRandom->Exp(2);
    double y3 = gRandom->Gaus(0.5*x3+2, 1);
    h3->Fill(x3, y3);
}

TCanvas* a = new TCanvas();
h->Draw("colz");

TCanvas* a2 = new TCanvas();
h2->Draw("colz");

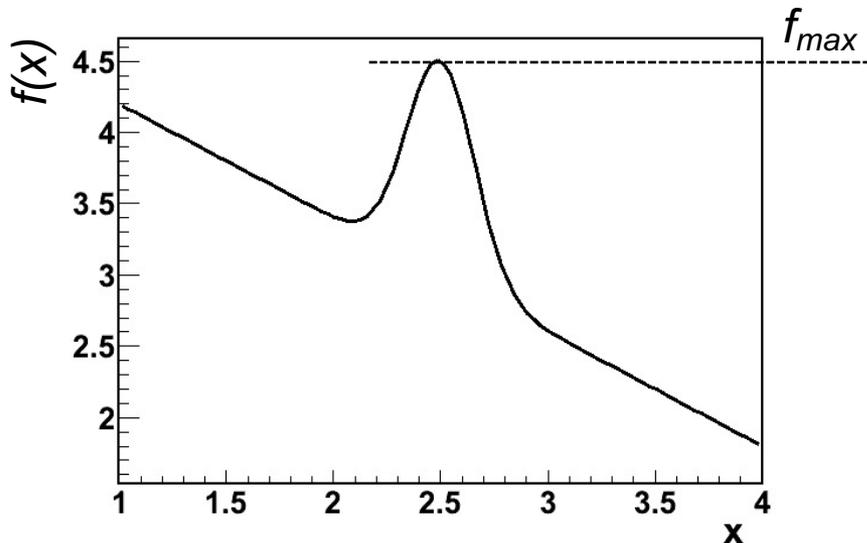
TCanvas* a3 = new TCanvas();
h3->Draw("colz");

TCanvas* b = new TCanvas();
TH1* h_px = h->ProjectionX();
TH1* h2_px = h2->ProjectionX();
TH1* h3_px = h3->ProjectionX();
h_px->SetLineColor(kBlue+2);
h2_px->SetLineColor(kRed+2);
h3_px->SetLineColor(kGreen+2);
h_px->SetLineWidth(3);
h2_px->SetLineWidth(3);
h3_px->SetLineWidth(3);
h_px->Draw("hist");
h_px->Draw("samehist");
h2_px->Draw("samehist");

TCanvas* c = new TCanvas();
TH1* h_py = h->ProjectionY();
TH1* h2_py = h2->ProjectionY();
TH1* h3_py = h3->ProjectionY();
h_py->SetLineColor(kBlue+2);
h2_py->SetLineColor(kRed+2);
h3_py->SetLineColor(kGreen+2);
h_py->SetLineWidth(3);
h2_py->SetLineWidth(3);
h3_py->SetLineWidth(3);
h_py->Draw("hist");
h_py->Draw("samehist");
h2_py->Draw("samehist");
}
```

U:--- macro\_2D.C All L1 (C/\*l Abbrev)

# Metodo di reiezione (hit and miss)



Si generano due variabili random:

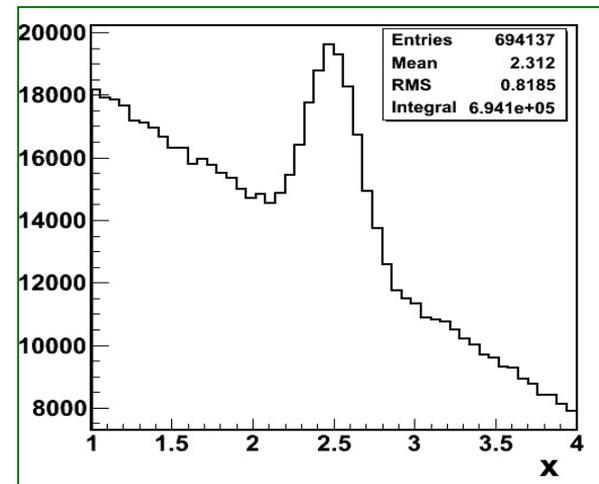
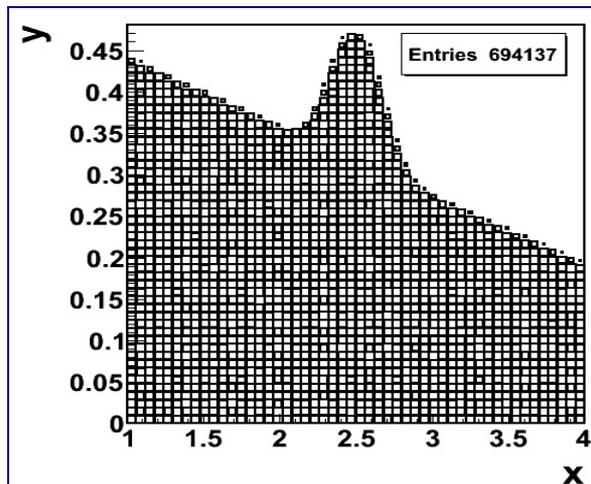
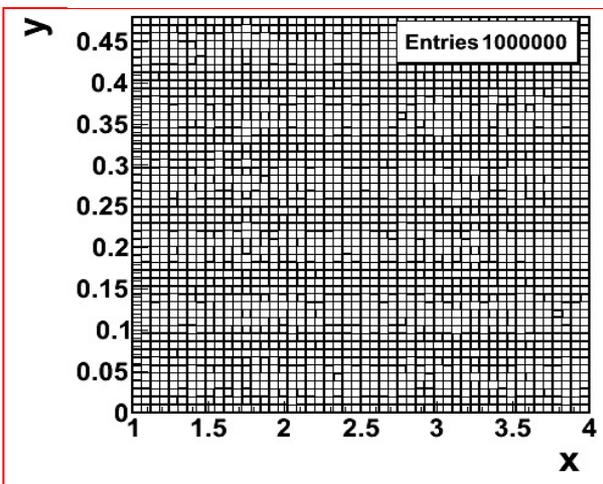
- $x$  in  $[a, b]$
- $p$  in  $[0, f_{max}]$

Un evento viene:

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

Quindi, ad esempio, in ROOT:

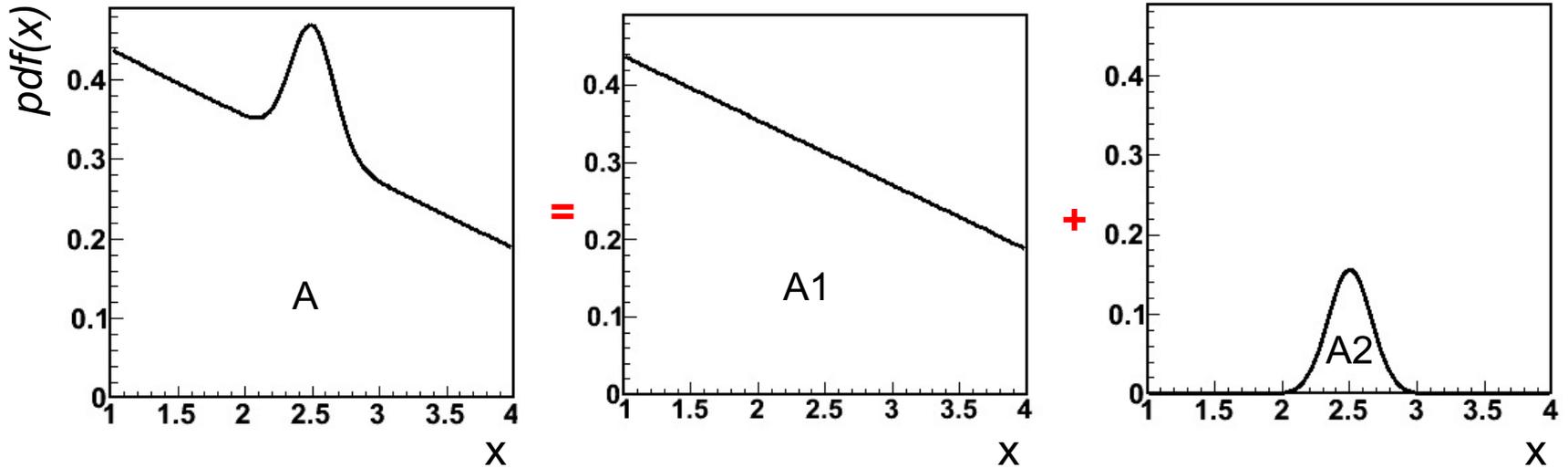
```
if (p <= f(x)) { h → Fill(x); }  
else { }
```



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

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

# Metodo composito



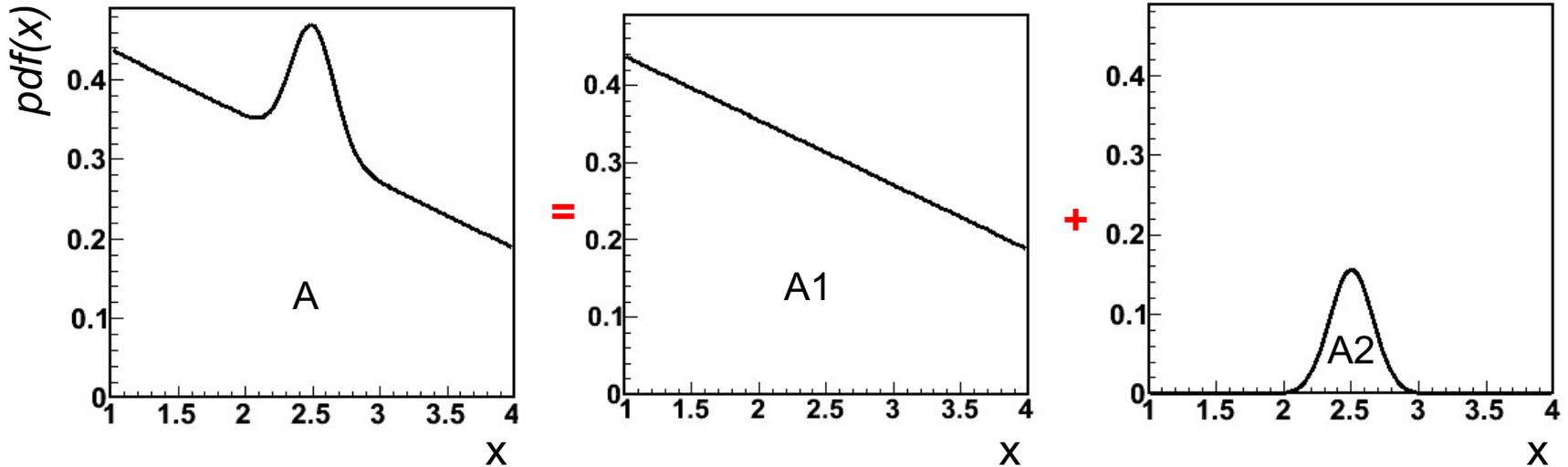
Si “decompono” la p.d.f (di area  $A$ ) in somma di p.d.f. e per ciascuna si utilizza il metodo più conveniente/adatto.

Nell'esempio sopra (retta + gaussiana) potremmo:

- utilizzare il metodo di inversione per generare gli eventi della p.d.f. di area  $A1$ ;
- 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 – accortezza I



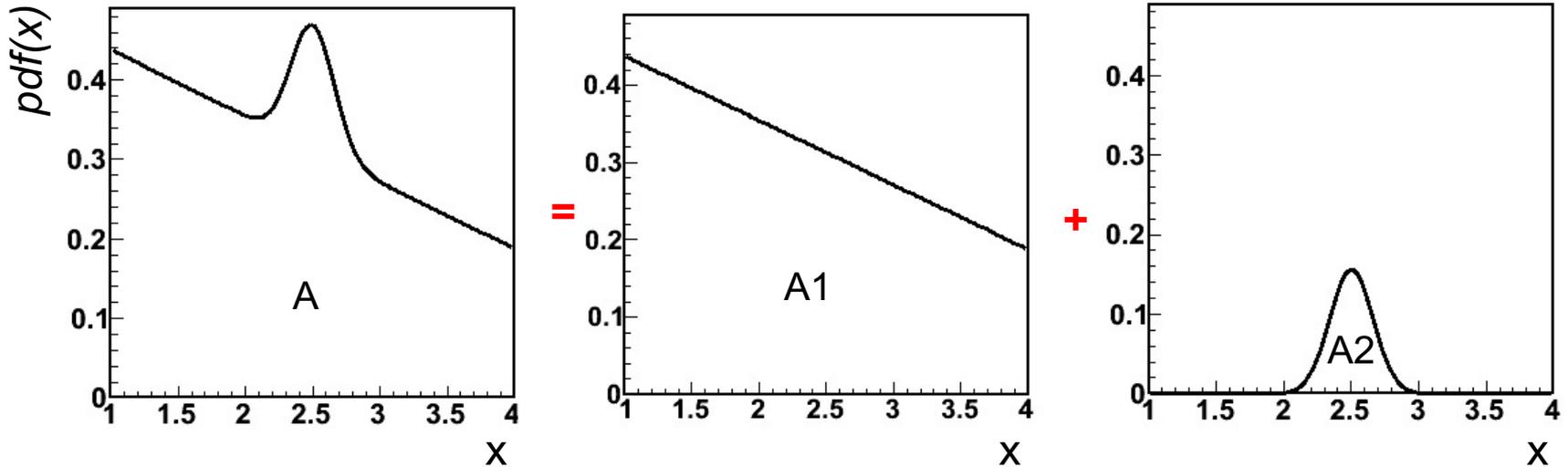
Gli eventi generati in a) e quelli generati in b) dovranno essere nelle giuste proporzioni: per ogni  $A$  eventi se ne genereranno  $A1$  secondo a) e  $A2$  secondo b), con, ovviamente,  $A=A1+A2$ .

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) prima e  $A2$  eventi b), dopo (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

# Metodo composito – accortezza 2



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,  $A=A1+A2$ .

In generale se abbiamo una distribuzione di eventi che è la "somma" di due popolazioni è più "realistico" (statisticamente) utilizzare  $A1/A$  e  $A2/A$  come "occupazioni medie" (i.e. "probabilità") delle varie sotto-popolazioni, non come numeri esatti. Quindi abbiamo due alternative per generare i numeri di eventi di ogni sotto-popolazione,  $\alpha1$  e  $\alpha2$ :

- come "estrazioni" poissoniane  $\rightarrow A \sim A1 + A2 \sim \alpha1 + \alpha2$ ;
- come "estrazioni" binomiali  $\rightarrow A = A1 + A2 = \alpha1 + \alpha2$  ma  $A1 \sim \alpha1$  e  $A2 \sim \alpha2$

# Generatore Lineare Congruente (Linear Congruent Generators, LCG)

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

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

$x_0$  ( $0 \leq x_0 < m$ ) è il valore di partenza o **seed**

$\lambda$  ( $0 < \lambda < m$ ) è il **moltiplicatore**

$\mu$  ( $0 \leq \mu < m$ ) 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 periodicità) è  $m$ , per tutte le seed, solo se le  $\lambda$  e la  $\mu$  sono scelti accuratamente:

- $\mu$  e  $m$  sono co-primi (massimo comun divisore è 1)
- $\lambda-1$  è divisibile per tutti i fattori primi di  $m$
- $\lambda-1$  è divisibile per 4 se  $m$  è divisibile per 4

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

~~$$x_0 = 1; \lambda = 7; \mu = 3; m = 32$$~~

# Generatore Lineare Congruente (Linear Congruent Generators, LCG)

La sequenza è definita da:  $x_{n+1} = (\lambda x_n + \mu) \bmod m$   
cioè  $x_{n+1}$  è il resto della divisione per  $m$  di:  $(\lambda x_n + \mu)$

$x_0$  ( $0 \leq x_0 < m$ ) è il valore di partenza o **seed**

$\lambda$  ( $0 < \lambda < m$ ) è il **moltiplicatore**

$\mu$  ( $0 \leq \mu < m$ ) è l'**incremento**

$m$  ( $m > 0$ ) è il **modulo**

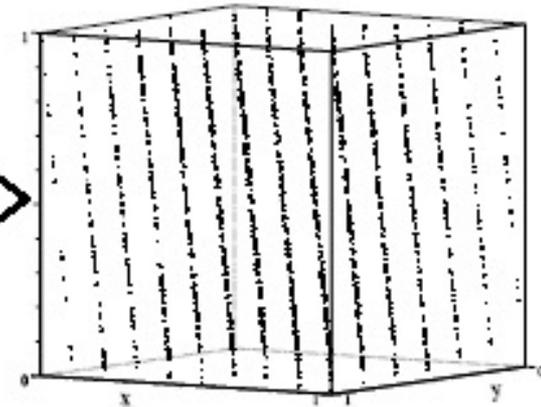
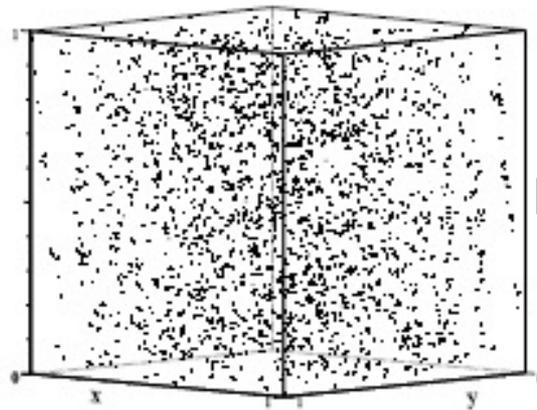
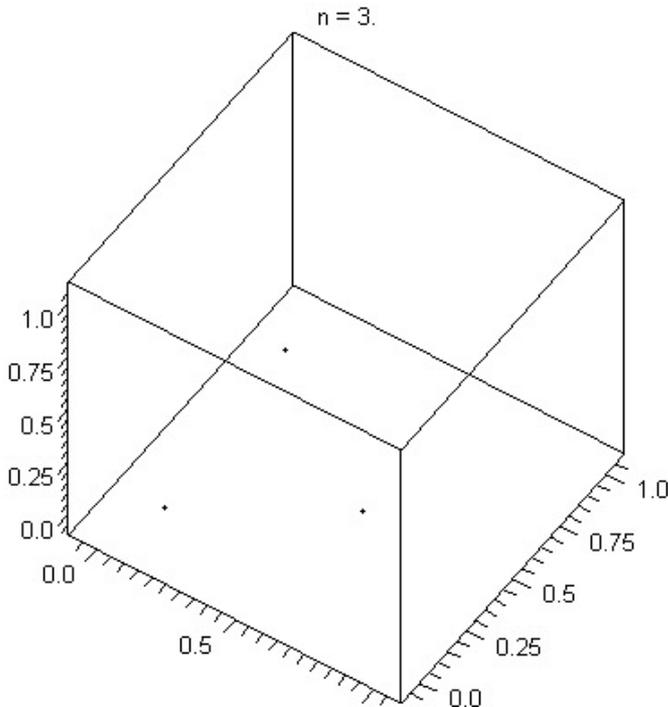
→ la lunghezza della sequenza (i.e. la periodicità) è  $m$ , per tutte le seed, solo se le  $\lambda$  e la  $\mu$  sono scelti accuratamente:

- $\mu$  e  $m$  sono co-primi (massimo comun divisore è 1)
- $\lambda-1$  è divisibile per tutti i fattori primi di  $m$
- $\lambda-1$  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;