

# Laboratorio II – 1° modulo

## Lezione 3

### Metodi Monte Carlo

# Laboratorio II – 1° modulo

## Lezione 3

### Metodi Monte Carlo

# Indice

---

- Generazione di numeri casuali:
  - La funzione `rand()` della libreria `<cstdlib>` per generare numeri casuali secondo una funzione densità di probabilità uniforme
  - I metodi del Try&Catch e della funzione cumulativa inversa per la generazione di numeri casuali secondo una generica funzione densità di probabilità
  - Generazione di numeri casuali distribuiti secondo una Normale con il metodo basato sul Teorema Centrale del Limite
- Rappresentazione grafica delle distribuzioni di probabilità mediante istogrammi in ROOT
- Esercizi

# Generazione di numeri casuali

Data una macchina deterministica, e.g. un computer, è difficile fare in modo che generi **numeri puramente casuali**. Si devono quindi sfruttare processi “esterni” casuali, come ad esempio il decadimento radioattivo, la corrente misurata in un conduttore non riferito a nessun potenziale (antenna), ecc... In genere però questi metodi sono di difficile implementazione e sono lenti, è invece più conveniente generare numeri **pseudo-casuali**: numeri generati mediante un algoritmo deterministico che produce una sequenza con, approssimativamente, le stesse proprietà statistiche di un processo puramente casuale

Esempio: **generatore lineare congruenziale**

$$X_{n+1} = (A \cdot X_n + C) \bmod M$$

- $0 < M$
- $0 < A < M$
- $0 \leq C < M$
- $0 \leq X_0 < M$  ( $X_0$  è detto “**seed**”)
- $M \sim 2^{32}$

Tutti i generatori di numeri pseudo-casuali hanno bisogno di un “**seed**” da cui partire. Questo ha il notevole vantaggio che la sequenza è riproducibile (a patto di usare lo stesso “**seed**”) → utile per riprodurre gli stessi risultati (e.g. in fase di *debug* del programma, o per il confronto dei risultati tra diversi colleghi, ecc...)

Inoltre è importante osservare che il risultato che si avrebbe lanciando **1000** volte il generatore di numeri pseudo-casuali da parte dello stesso programma, lo si potrebbe ottenere lanciando **100** volte lo stesso programma in parallelo e facendo generare **10** numeri pseudo-casuali a ciascuno programma → attenzione però che ad ogni programma va assegnato un “**seed**” diverso!

# Generazione di numeri casuali

La funzione `rand` (della libreria `<cstdlib>`) genera numeri **interi** pseudo-casuali nell'intervallo compreso tra 0 e `RAND_MAX` (=2147483647)

```
#include <cstdlib>
#include <ctime>

int main()
{
    srand(time(NULL));
    int count = 10;
    // Generazione di numeri casuali
    // interi compresi tra 0 e RAND_MAX
    for (int i = 0; i < count; i++)
    {
        std::cout << "Numero casuale intero: "
                  << rand() << std::endl;
    }
    // Generazione di numeri casuali
    // razionali tra min e max
    double min = -1;
    double max = 1;
    for (int i = 0; i < count; i++)
    {
        std::cout << "Numero casuale razionale: "
                  << min + (max - min) * rand() / RAND_MAX << std::endl;
    }
    return 0;
}
```

Tutti i numeri generati dalla funzione `rand()` hanno la stessa probabilità di essere estratti (seguono una funzione densità di probabilità, *pdf*, di tipo uniforme)

Possiamo sfruttare la funzione `rand` per generare numeri casuali con **pdf uniforme** in un generico intervallo (`min, max`) con le seguenti operazioni:

- **traslazione:**  $\min +$
- **dilatazione/contrazione:**  $(\max - \min) / \text{RAND\_MAX}$

$$x \in [0, \text{RAND\_MAX}] \rightarrow x' \in [\min, \max]$$

# Generazione di numeri casuali

La funzione `rand` (della libreria `<cstdlib>`) genera numeri **interi** pseudo-casuali nell'intervallo compreso tra 0 e `RAND_MAX` (=2147483647)

```
#include <cstdlib>
#include <ctime>

int main()
{
    srand(time(NULL)); ←
    int count = 10;
    // Generazione di numeri casuali
    // interi compresi tra 0 e RAND_MAX
    for (int i = 0; i < count; i++)
    {
        std::cout << "Numero casuale intero: "
                  << rand() << std::endl;
    }
    // Generazione di numeri casuali
    // razionali tra min e max
    double min = -1;
    double max = 1;
    for (int i = 0; i < count; i++)
    {
        std::cout << "Numero casuale razionale: "
                  << min + (max - min) * rand() / RAND_MAX << std::endl;
    }
    return 0;
}
```

La funzione `srand` inizializza il **seed** per la generazione dei numeri pseudo-casuali

La funzione `time` restituisce il tempo in secondi dal 1 gennaio 1970  
Con questa istruzione, ogni volta che eseguo il programma, ci sarà una diversa sequenza di estrazione dei numeri casuali

# Statistiche di un campione di numeri casuali

Dato un campione di numeri casuali estratti secondo una certa distribuzione di probabilità, è possibile calcolarne le grandezze statistiche che lo caratterizzano (es: media e varianza)

```
...
double sum=0., sumSq=0.;
int N = 100000;
for (int i = 0; i < N; i++)
{
    double numero_casuale =
        min + (max - min) *
        rand() / RAND_MAX;
    sum += numero_casuale;
    sumSq += numero_casuale*
            numero_casuale;
}
double mean = sum/N;
double var = sumSq/N - mean*mean;
...
```

La **media** e la **varianza** campionarie si **stimano** a partire dalla somma degli elementi e dalla somma dei quadrati degli elementi di un campione (è utile ricordare che  $\text{Var}[x] := E[(x - \bar{x})^2] = E[x^2] - E[x]^2$ )

$$\sigma^2 = \frac{\sum x_i^2}{N} - \left( \frac{\sum x_i}{N} \right)^2$$

Varianza campionaria  
*affetta da bias*

$$\sigma^2 = \left( \frac{\sum x_i^2}{N} - \left( \frac{\sum x_i}{N} \right)^2 \right) \frac{N}{N-1}$$

Varianza campionaria  
**NON affetta da bias**

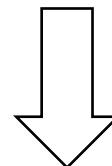
Quali valori vi aspettate per media e varianza di un campione di numeri casuali distribuiti uniformemente nell'intervallo  $[min, max]$ ?

# Il metodo Try&Catch

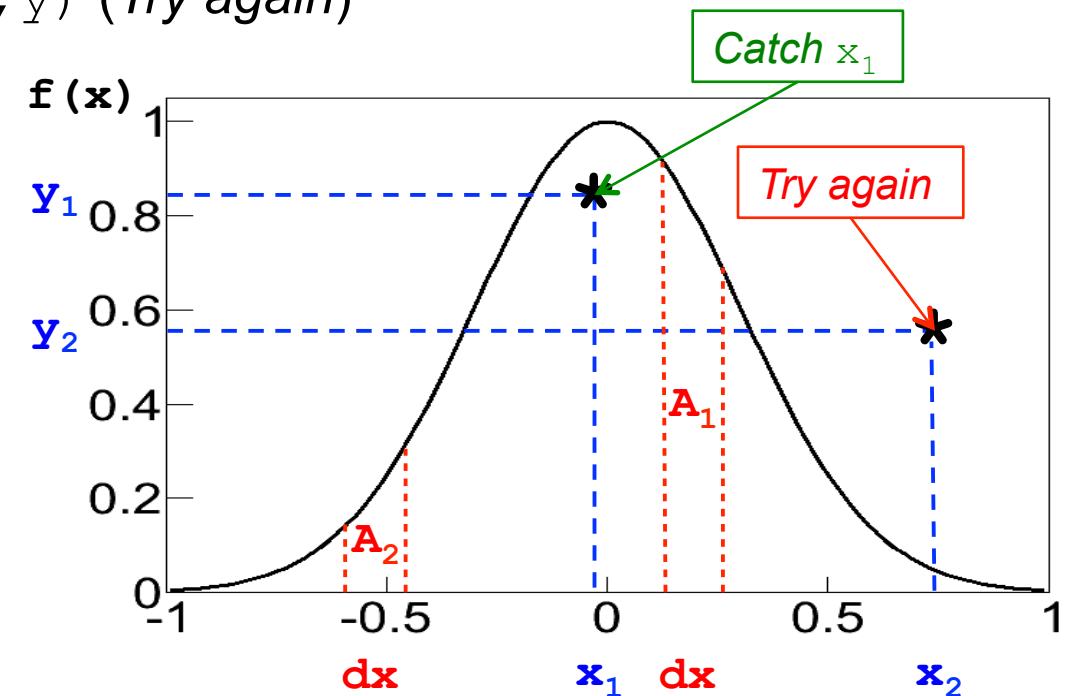
**Obiettivo:** generare numeri casuali secondo una distribuzione di probabilità descritta dalla funzione  $f(x)$

Il metodo del **Try&Catch** consiste nel:

- generare coppie di numeri casuali sull'asse  $x$  ed  $y$  con distribuzione uniforme
- se il punto identificato dalle coordinate  $(x, y)$  si trova sotto la funzione  $f(x)$ , il numero  $x$  viene “tenuto” (Catch), altrimenti viene generata una nuova coppia di coordinate  $(x, y)$  (Try again)



La frazione di  $x$  “tenuti” in un certo intervallo è direttamente proporzionale all'area sottesa da  $f(x)$  in quell'intervallo ( $A_1 > A_2$ )

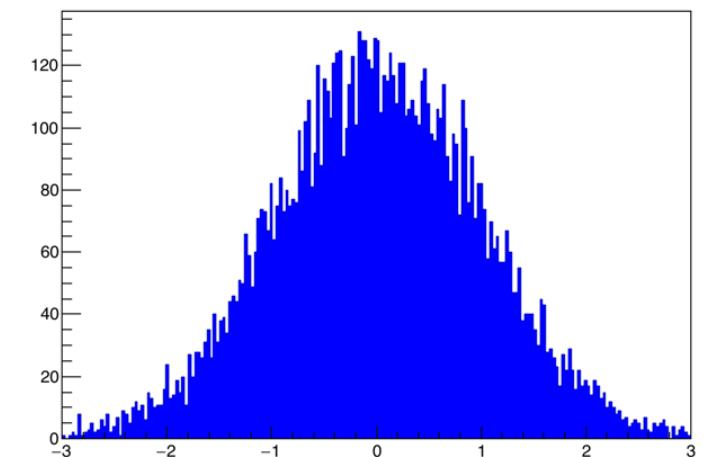


# Il metodo Try&Catch

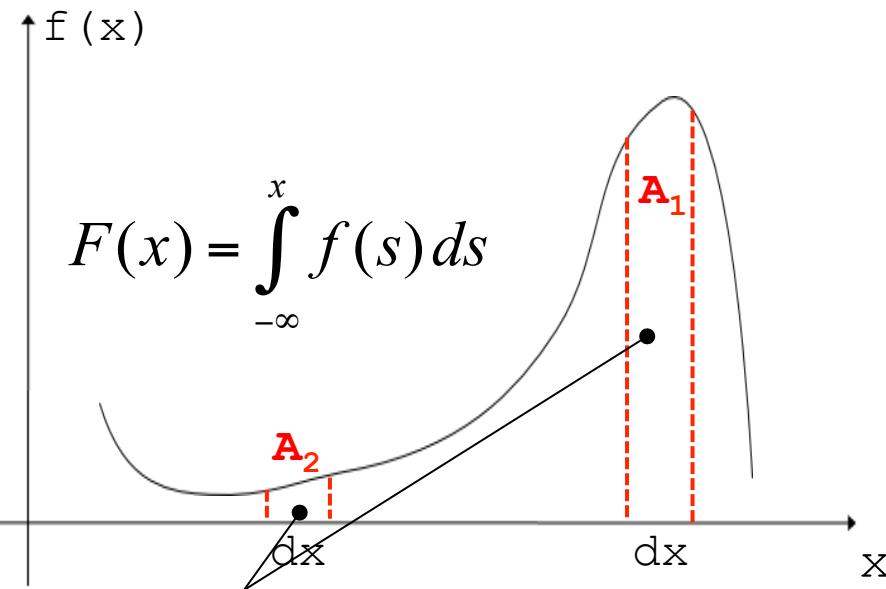
```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#define MAX_NUM 100000
double fgaus(double x)
{
    return exp(-0.5*x*x);
}
double rand_range(double min, double max)
{
    return min + (max - min) * rand() / RAND_MAX;
}
double rand_TAC(double xMin, double xMax, double yMin, double yMax)
{
    double x = 0., y = 0.;
    do {
        x = rand_range(xMin, xMax);
        y = rand_range(yMin, yMax);
    } while (y > fgaus(x));
    return x;
}
int main()
{
    srand(time(NULL));
    int N = MAX_NUM;
    double numero_casuale = 0.;
    for (int i = 0; i < N; ++i)
    {
        numero_casuale = rand_TAC(-1., 1., 0., 1.);
        std::cout << numero_casuale << std::endl;
    } ...
```

- L'algoritmo sfrutta una funzione (`rand_range`) che genera numeri casuali distribuiti uniformemente su un intervallo
- La funzione `rand_TAC` genera numeri casuali distribuiti secondo una certa  $f(x)$ , che in questo esempio è una Normale, `fgaus`

Ciclo `do...while`: funziona come il `while` con l'unica differenza che la condizione di uscita viene valutata dopo aver eseguito il codice contenuto nello scope

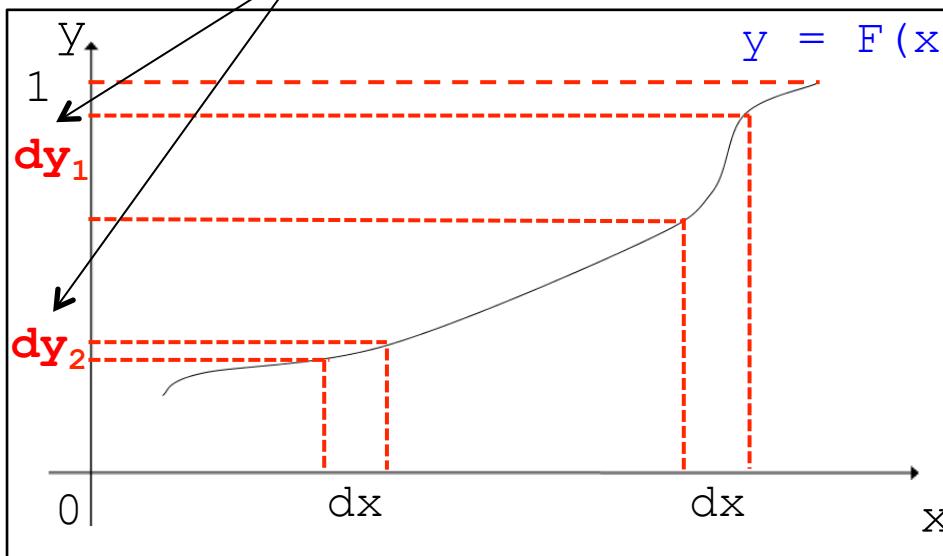


# Il metodo della funzione cumulativa inversa



Metodo alternativo (più efficiente del Try&Catch) per generare numeri casuali secondo un'arbitraria distribuzione  $f(x)$

- Si calcola la funzione integrale della  $f(x)$ ,  $F(x)$ , detta **funzione cumulativa** o **funzione di ripartizione**
- Poiché  $f(x)$  è una pdf,  $F(x)$  ha le seguenti proprietà:
  - definita su  $\mathbb{R}$
  - monotonamente crescente
  - assume valori tra 0 e 1



Per definizione di funzione integrale:

$$dy = dF(x) = (dF/dx) \cdot dx = f(x) \cdot dx$$

- Campiono uniformemente valori di  $y$  in  $[0, 1]$
- Per ogni  $y$ , calcolo  $x = F^{-1}(y)$
- Campionando uniformemente  $y$  otterremo più valori nell'intervallo  $dy_1$  rispetto all'intervallo  $dy_2$  poiché  $A_1 > A_2$ , generando così più numeri in  $dx$  di  $A_1$  che in  $dx$  di  $A_2$

# Esempio: distribuzione esponenziale

Voglio generare numeri casuali distribuiti secondo la distribuzione esponenziale:

$$f(x) = e^{-x/\mu} / \mu \quad x \in [0, +\infty)$$

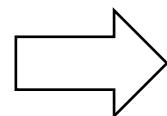
Calcolo la funzione cumulativa:

$$F(x) = \int_0^x f(s) ds = -e^{-x/\mu} + 1$$

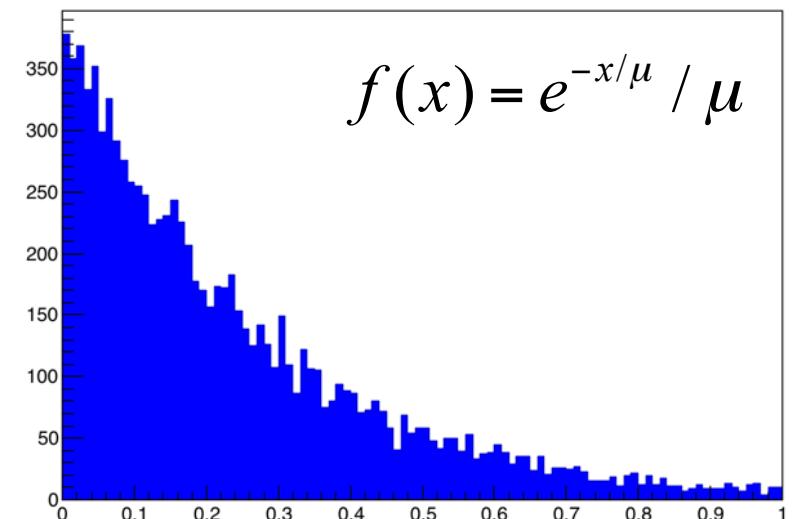
Invertendo la  $F(x)$  si ottiene:

$$y = F(x) \rightarrow x = F^{-1}(y) = -\mu \log(1 - y)$$

**Riassumendo:** si generano numeri casuali uniformi in  $y$  tra  $[0, 1)$  e si calcola  $x$  dalla formula precedente



$x$  risulta avere distribuzione esponenziale, con media  $\mu$



# Teorema Centrale del Limite

Per generare numeri secondo una Normale si può anche sfruttare il **Teorema Centrale del Limite**: “La *media di  $N$  variabili aleatorie indipendenti*, qualunque sia la loro distribuzione di probabilità, è a sua volta una variabile aleatoria che segue la distribuzione **Normale** nel limite in cui  $N \rightarrow \infty$ ”

```
double rand_CLT(double xMin, double xMax, int tries)
{
    double x = 0.;
    for (int i = 0; i < tries; i++)
        x += rand_range(xMin, xMax);
    return x / tries;
}

int main()
{
    srand(time(NULL));
    int N = MAX_NUM;
    for (int i = 0; i < N; ++i)
    {
        double numero_casuale =
            rand_CLT(-1, 1, 10);

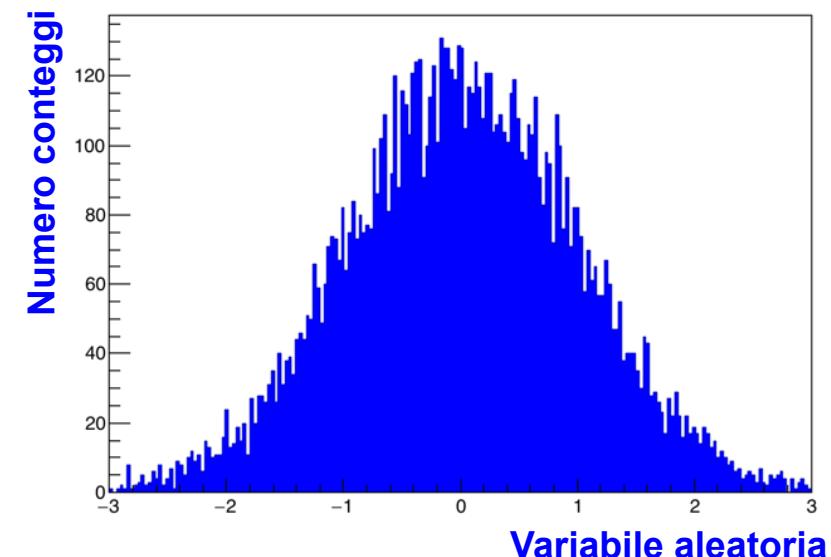
        std::cout << "Numero casuale: "
            << numero_casuale << std::endl;
    }
    return 0;
}
```

- Nel main viene chiamata rand\_CLT, che a sua volta chiama rand\_range per la generazione di numeri casuali secondo una distribuzione uniforme
- In rand\_CLT si calcola la media aritmetica di “tries” numeri casuali estratti con rand\_range
- I numeri casuali prodotti da rand\_CLT seguono una distribuzione Normale nel limite in cui “tries” è sufficientemente grande

**Come facciamo a visualizzare  
come si distribuiscono i numeri  
casuali che abbiamo generato?**

# Gli histogrammi

- Gli histogrammi si usano per rappresentare i risultati di esperimenti di conteggio
- Sono lo strumento giusto per visualizzare le distribuzioni dei numeri casuali
- Sull'asse delle ascisse c'è il *range* dei possibili valori che può assumere la variabile aleatoria che sto misurando: ad esempio, se lancio un dado a sei facce dovrò avere i numeri interi da 1 a 6
- Sull'asse delle ordinate c'è il numero di volte che ho misurato ciascun valore durante l'esperimento: ad esempio, ho lanciato il dado 60 volte e ho fatto "1" per 13 volte, "2" per 9 volte, etc...
- Si definisce un *binning*, cioè si suddivide l'asse delle ascisse in intervalli, detti **bin**
- Durante l'esperimento, ogni volta che viene misurata la variabile aleatoria, si determina in quale bin è contenuto il risultato e si incrementa di un'unità il numero di conteggi in quel bin

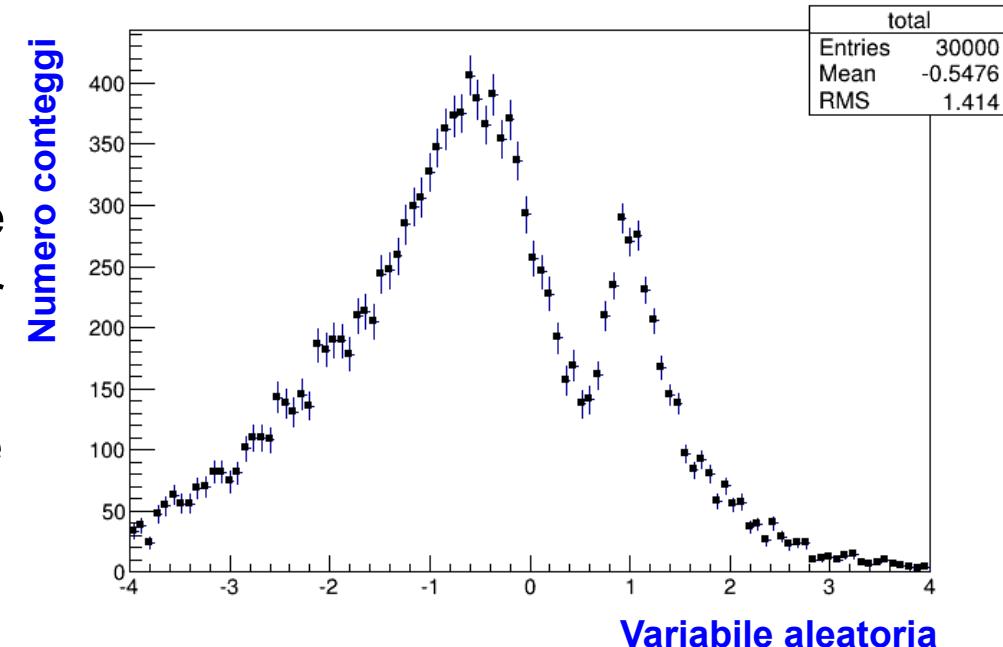


# Disegnare istogrammi con ROOT

- ROOT è un pacchetto per l'analisi dati scritto in C++
- È nato al CERN
- Home page: [root.cern.ch](http://root.cern.ch)



- Oggi vediamo come è possibile sfruttare le librerie di ROOT per costruire e disegnare histogrammi con cui rappresentare le distribuzioni dei numeri casuali che abbiamo generato



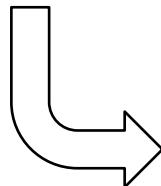
# Definire un istogramma

- Per creare gli histogrammi in ROOT usiamo la libreria TH1F.h (o TH1D.h)

#include "TH1F.h" oppure #include "TH1D.h" a seconda che vogliate usare bin di tipo float oppure double

- La definizione di un nuovo istogramma è simile alla definizione di una qualsiasi variabile:

```
double myNum (*inizializzazione);
```



```
TH1F myHisto (*inizializzazione);
```

- Per inizializzare una variabile numerica è sufficiente scrivere un numero:

```
double myNum (1.5);
```

- Per creare un istogramma di ROOT è necessario inizializzare alcuni parametri che ne descrivono la struttura:

```
TH1F myHisto ("name", "title", Nbins, xmin, xmax);
```

# Riempire un istogramma

```

#include "TCanvas.h"
#include "TH1F.h"

... funzioni rand_range, rand_CLT

int main()
{
    srand(time(NULL));
    double min      = -1;
    double max      = 1;
    int nEstrazioni = 10;
    int N           = 10000;

    TH1F pdf ("name", "title", 100, min, max);

    for (int i = 0; i < N; i++)
    {
        double numero_casuale =
            rand_CLT(min, max, nEstrazioni);
        std::cout << "Numero casuale: "
            << numero_casuale << std::endl;
        pdf.Fill(numero_casuale);
    }
    ... il codice prosegue nella prossima slide ...

```

- Librerie di ROOT da includere
- Definizione di un istogramma con 100 bin tra min e max
- Ad ogni iterazione del ciclo **for**:
  - viene calcolata la media di nEstrazioni=10 numeri casuali generati tra min e max, mediante la funzione rand\_CLT
  - si usa il metodo **Fill** per incrementare di un conteggio il bin che include il numero\_casuale estratto

# Disegnare un istogramma

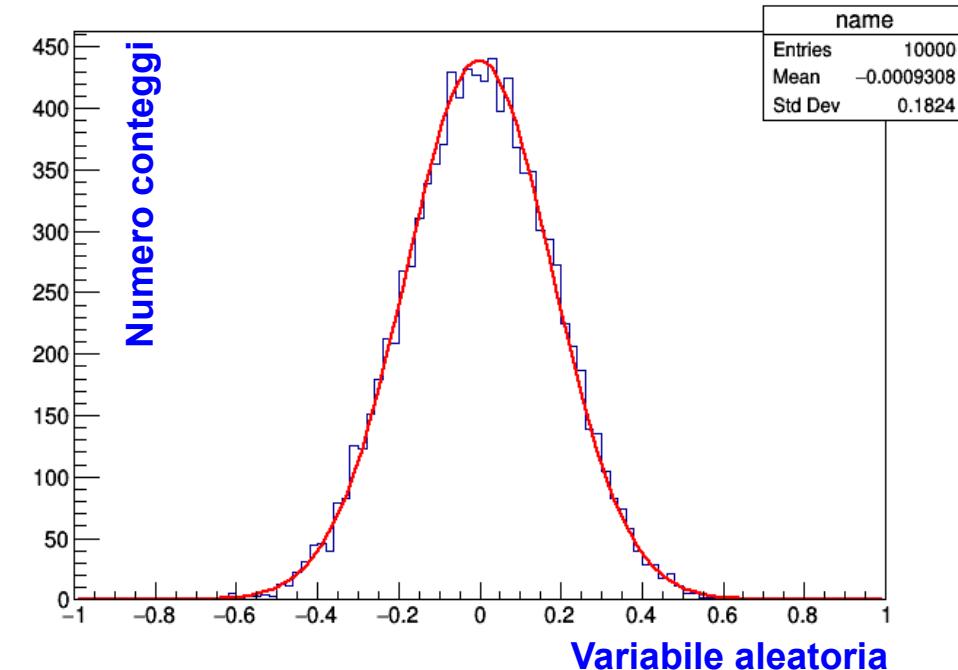
*... il codice prosegue dalla slide precedente*

```
TCanvas cnv;
pdf.Draw();
pdf.Fit("gaus");
cnv.Print("istogramma.png","png");
return 0;
}
```

- Con il metodo **Fit("gaus")** è possibile fissare l'istogramma con una Normale (funzione rossa nel grafico)
- L'algoritmo di fit calcola i parametri della Normale (normalizzazione, media e sigma) che la fanno adattare meglio ai dati “sperimentali” contenuti nell'istogramma

$$p_0 \times \exp\left(-\frac{(x - p_1)^2}{2p_2^2}\right)$$

- **TCanvas** è la tavolozza su cui si disegnerà l'istogramma
- Metodo **Draw** serve per disegnare l'istogramma
- Metodo **Print** serve per salvare il Canvas in formato grafico (.png)



# Compilare un programma che utilizza le librerie di ROOT

Le librerie di ROOT non sono librerie di sistema, pertanto è necessario *linkarle* esplicitamente in fase di compilazione

```
c++ -o testROOT testROOT.cpp `root-config --cflags --glibs`
```

- **c++**: il comando di compilazione è sempre lo stesso
- **-o testROOT**: nome del file eseguibile
- **testROOT.cpp**: nome del file da compilare
- **`root-config --cflags --glibs`**: comando per *linkare* le librerie di ROOT

**Attenzione:** gli apici di `root-config `...`` sono **backtick** (diversi dall'apostrofo): li ottenete digitando la combinazione AltGr + apostrofo

# Esercizi

---

Svolgere gli esercizi seguenti organizzando le funzioni esterne al main in un'unica libreria, composta da un file con l'implementazione delle funzioni (.cc) e da un header file (.h) contenente i prototipi delle funzioni

**Esercizio 1:** Generare  $N=10^4$  numeri casuali distribuiti uniformemente nell'intervallo  $[a, b]$  mediante una funzione (rand\_range)

Stimare la media e la varianza del campione di numeri casuali generati e verificare che:

$$\mu = \frac{(a+b)}{2}$$

$$\sigma^2 = \frac{(b-a)^2}{12}$$

Definire e riempire un istogramma TH1F con 100 bin per rappresentare la distribuzione di probabilità dei dati generati

# Esercizi

---

**Esercizio 2:** Generare  $N=10^4$  numeri casuali secondo la distribuzione di Cauchy nell'intervallo  $[-5, 5]$  utilizzando il metodo Try&Catch(\*):

$$f_{Cauchy} = \frac{1}{\pi} \frac{1}{1+x^2}$$

Definire e riempire un istogramma `TH1F` con 100 bin per rappresentare la distribuzione di probabilità dei dati generati

(\*) Implementare le funzioni `rand_TAC` e `f_Cauchy` nella stessa libreria utilizzata nell'esercizio 1

# Esercizi

**Esercizio 3:** Utilizzare il metodo della funzione cumulativa inversa(\*) per generare  $N=10^4$  numeri casuali distribuiti secondo una pdf esponenziale con  $\mu=0.25$  nell'intervallo  $[0, 1]$ :

$$f_{Exp} = \frac{1}{\mu} \exp(-x / \mu)$$

Definire e riempire un istogramma `TH1F` con 100 bin per rappresentare la distribuzione di probabilità dei dati generati

(\*) Implementare la funzione `rand_FCI_Exp` nella stessa libreria utilizzata nell'esercizio 1

- Funzione cumulativa inversa

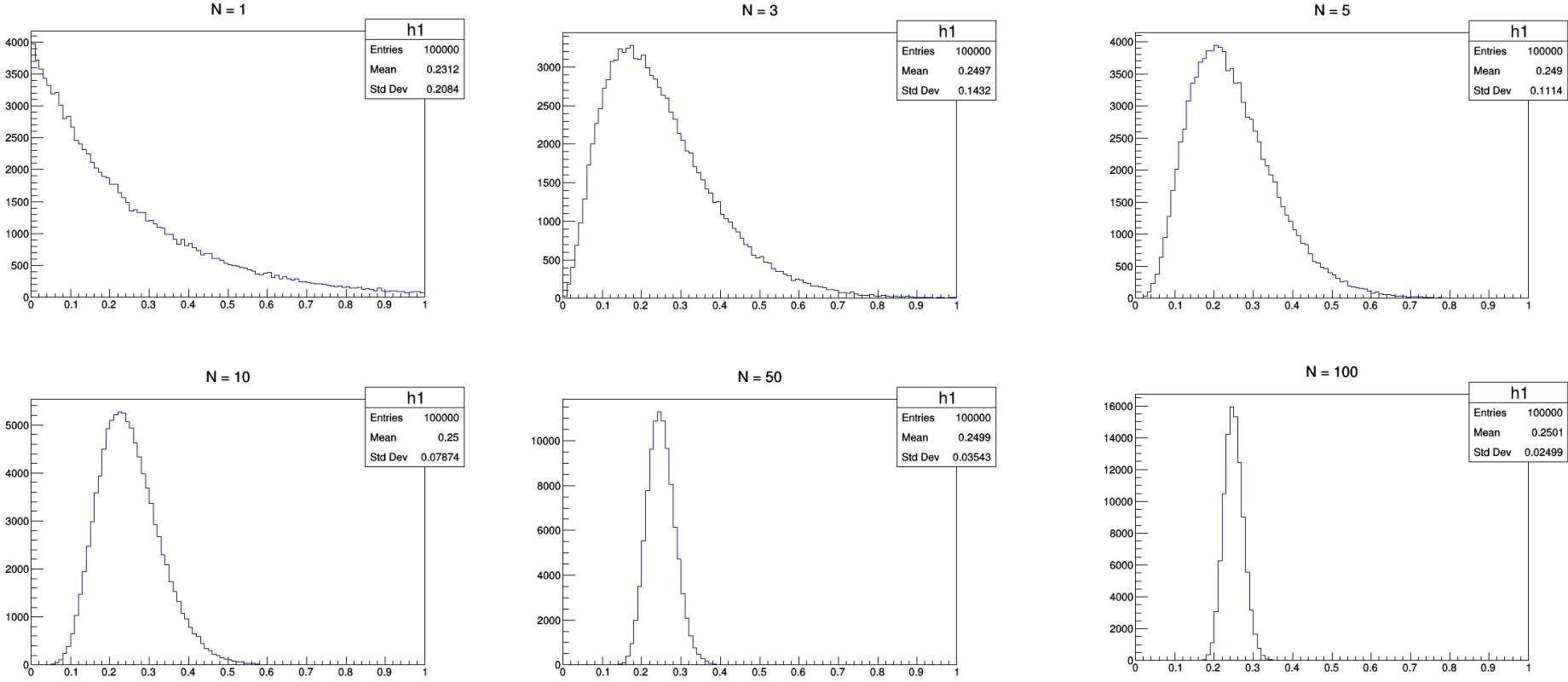
# Esercizi

---

## Esercizio 4:

- Utilizzare la funzione `rand_FCI_Exp` implementata nell'esercizio precedente per generare  $N=2$  numeri casuali secondo la pdf esponenziale
- Ripetere l'operazione  $10^4$  volte all'interno di un ciclo, calcolando, per ogni coppia di numeri estratti, il loro valor medio
- Riempire un istogramma `TH1F` con 100 bin per rappresentare la distribuzione di probabilità dei valori medi calcolati
- Modificare il codice precedente per chiedere all'utente di inserire da terminale il valore del numero  $N$  di numeri casuali da estrarre e mediare. Produrre i grafici per i casi  $N=3, 5, 10, 50, 100$

# Soluzione ... grafica



Per il Teorema Centrale del Limite, all'aumentare del numero  $N$  di variabili aleatorie che vengono sommate per calcolare i valori medi, la distribuzione di probabilità dei valori medi tende ad una Normale

# Esercizi

---

**Esercizio 5:** Implementare il generatore lineare congruenziale. Si consiglia di usare come parametri i seguenti valori:

- $M = 2147483647$
- $A = 214013$
- $C = 2531011$

Come tipo di variabile per contenere i numeri casuali si consiglia di usare il **long int**

- Generare  $10^6$  numeri casuali nell'intervallo  $[0, 1)$  e rappresentarne la distribuzione mediante un istogramma **TH1D** dotato di 1000 bin
- Verificare inoltre il Teorema Centrale del Limite
  1. Generare  $Q=10^4$  numeri casuali, farne la media ( $\bar{m}$ ) e riempire con il metodo **Fill ( $\bar{m}-\mu$ ) / \sigma**) un istogramma di 100 bin (dove  $\mu$  è la media di  $m$ , mentre  $\sigma$  è la deviazione standard di  $m$ )
  2. Ripetere il punto 1. per  $N=10^4$  volte