Guida ROOT

Fabio Prestipino

6 giugno 2022

INDICE

1 INTRODUZIONE ED INFORMAZIONI GENERALI UTILI

ROOT è un linguaggio ad oggetti sviluppato dal CERN basato su C++ utile per l'analisi ed elaborazione di grandi moli di dati. Ogni istogramma, grafico, funzione (...) che creiamo è un oggetto con delle sue opzioni e funzioni (funzioni membro). Una proprietà fondamentale è l'ereditarietà: tutte le classi sono figlie di una classe di base (TObject) da cui discendono le altre; ogni classe figlia eredita le funzioni membro di quelle precedenti. Il tipo degli oggetti comincia sempre con la lettera T, seguita da altre lettere che specificano le caratteristiche dell'oggetto. Di seguito riporto alcune informazioni di base in ordine casuale:

- I tipi base di ROOT hanno nomi leggermente diversi da quelli di c++: `Int_t`, `Float_t`, `Double_t`, TString...

- Per stampare a schermo basta usare i comandi di C++. Ad esempio:

  ```
  Double_t a;
  cout« "Scritta" « a;
  ```

- Per accedere ai file di ROOT il codice da scrivere sulla barra di comando è

  ```
  TBrowser b;
  ```

- Per usare funzioni matematiche è possibile utilizzare la libreria di Root "TMath", ad esempio:

  ```
  TMath::Sin(TMath::Pi()/2.) = 0
  ```

- Per salvare un oggetto (grafico, istogramma,...) come file ROOT bisogna aprire un file ROOT scriverci l'oggetto dentro e chiuderlo:

  ```
  TFile *nome = new TFile("nome_file.root","RECREATE");
  nome_oggetto -> Write();
  nome -> Close();
  ```

- Per accedere a file di dati (che dovrà essere nella stessa cartella della macro) si usa la funzione di C++ fstream(); in ordine bisogna creare un oggetto file, aprire quello desiderato, accedervi e chiuderlo. In ordine:

  ```
  fstream nome;
  nome.open("nome_file.txt");
  nome » x;
  nome.close();
  ```

1.1 *Come interfacciarsi*

Esistono 3 modi di interfacciarsi a ROOT:

- **Command line**
  Utile per comandi di base, è più flessibile delle macro (ad esempio non serve il ;). Per accedere al terminale direttamente da ROOT basta scrivere i normali comandi del terminale preceduti

da un ".! ". Per uscire da ROOT basta scrivere sulla barra di comando .q oppure se non funziona .qqqqq...

- **Macros**
  Un file scritto in un editor di testo che poi viene eseguito su ROOT. Una macro si presenta come una normale funzione di c++, può prendere parametri che poi vengono usati al suo interno e si possono creare altre funzioni all'interno. Di seguito un esempio di macro vuota

  void nome(parametro1, parametro 2, ...){}

  É inoltre possibile far girare una macro in due modi diversi: o facciamo partire tutta la macro, eseguendo così tutte le funzioni al suo interno, con il comando

  .c name.C

  oppure carichiamo le funzioni della macro in memoria con il comando

  .L name.C

  per poi chiamare le funzioni singolarmente, ad esempio:

  funz1();

  IMPORTANTE: LE MACROS DEVONO STARE NELLA CARTELLA "HOME$/root/macros"

- **Finestra grafica (GUI)**
  Per curare l'estetica dei grafici è possibile sfruttare l'interfaccia grafica, più intuitiva. Per far comparire le opzioni della finestra bisogna cliccare su "view" e spuntare "editor", "toolbar" ed "event statusbar".

2 ISTOGRAMMI

2.1 *Inizializzare e riempire un'istogramma*

Il tipo degli istogrammi è formato da 4 lettere: la T(come per tutti gli oggetti), una H di "histogram", un numero che indica il numero di dimensioni dell'istogramma e un'ultima lettera che può essere I (int), F (float), D (double) che indica il modo in cui vengono rappresentati i dati nell'istogramma. Un'istogramma di base sarà, ad esempio, del tipo "TH1F".
É raccomandato dichiarare (costruire) un'istogramma usando i puntatori, di seguito un format di dichiarazione di un'istogramma unidimensionale:

TH1F *nome = new TH1F("nome", "titolo", n-bin, x_{min}, x_{max});

Dove n-bin è il numero di bin che vogliamo nell'istogramma ed x_{min}, x_{max} il range entro cui vogliamo vedere l'istogramma. Per aggiungere punti ad un'istogramma unidimensionale:

nome -> Fill(x);

la funzione Fill può accettare anche un secondo paramentro che permette di aggiungere in una volta tante entrate uguali:

nome -> Fill(x, 8);

aggiunge 8 entrate x all'istogramma.
Se invece volessimo più dimensioni:

TH2F *nome = new TH2F("nome", "titolo", $n_x - bin$, x_{min}, x_{max}, $n_y - bin$, y_{min}, y_{max});

Per aggiungere punti in un'istogramma bidimenzionale:

nome -> Fill(x,y);

É inoltre possibile fare le proiezioni degli istogrammi multidimenzionali sui singoli assi:

nome1 -> ProjectionX();
nome2 -> ProjectionY();

che crea un'istogramma già riempito.

2.2 *Inserire dati da file*

É possibile riempire un'istogramma sfruttando i dati di un file di testo (che dovrà essere nella stessa cartella della macro). Innanzitutto bisogna costruire l'oggetto file

fstream nome;

poi bisogna aprire il file desiderato

nome.open("nome_file.txt")

infine con un ciclo aggiungiamo i dati del file fin quando questo non finisce:

```
Float_t x;
while(1){
nome » x;
nome_isto -> Fill(x);
if(!nome.good()) break;
}
nome.close();
```

Dove la funzione nome.good() dice se il file è concluso o meno. Nell'ultima riga è stato chiuso il file.

2.3 *Metodi utili istogrammi*

Di seguito sono riportati i metodi più importanti per gli istogrammi

- nome ->GetXaxis()->SetTitle("nome"); setta il titolo dell'asse
- nome -> SetMarkerStyle(numero);
  assegna uno stile al marker
- nome -> SetMarkerSize(numero);
  assegna una grandezza al marker
- nome -> GetBinError(nbin);
  prende l'errore di un bin
- nome -> GetBinContent(nbin); prende il numero di entrate nel bin

- nome -> GetBinContent(0); prende il numero di entrate in underflow
- nome -> GetBinContent(nbin+1); prende il numero di entrate in overflow
- nome -> GetEntries();
  prende il numero di entrate totali
- nome -> SetBinContent(ibin,val);
  assegna al contenuto del bin un valore val
- nome -> GetMean();
  calcola la media
- nome -> GetRMSError();
  calcola l'errore sulla media
- nome -> GetRMS();
  calcola la deviazione standard
- nome -> GetMeanError();
  calcola l'errore sulla deviazione standard
- nome -> GetMaximum();
  prende il bin con numero massimo di entrate
- nome -> GetMaximumbin();
  prende il numero del bin con numero massimo di entrate
- nome -> GetBinCenter();
  prende il centro del bin
- nome -> GetIntegral();
  restituisce un'array dei conteggi cumulativi
- nome -> Integral();
  calcola l'integrale nel range

2.4 *Operazioni fra istogrammi*

Si possono usare metodi dedicati della classe per effettuare operazioni tra oggetti di ROOT; le operazioni utili sono somma (talvolta sottrazione) e divisione. Seguiamo gli step necessari:

1. Creare e riempire gli istogrammi con cui operare
2. Creare l'istogramma risultato, inizialmente vuoto, passando come parametro il primo istogramma con cui si vole operare. Per la divisione:

   TH1F *nome_divisione = new TH1F(*h1);
   nome_divisione -> Divide(h1,h2,1,1);

   Per la somma:

   TH1F *nome_somma =new TH1F(*h1);
   nome_somma -> Add(h1,h2,1,1);

   Per la sottrazione si usa la stessa sintassi dell'addizione ma con un "-1" al posto del secondo "1". Infine, per avere incertezze corrette sugli istogrammi risultato, è necessario usare il metodo Sumw2():

   nome_risultato -> Sumw2();

3 GRAFICI

Un grafico rappresenta una serie di N coppie di due variabili (x, y). Esistono due tipi di grafici: quello semplice TGraph e quello in cui è possibile includere anche gli errori nel grafico TGraphErrors.

3.1 *Costruttore di TGraph*

Il costruttore di TGraph di base genera semplicemente un grafico di n punti presi da due array (una che contiene tutte le x ed una le y).

TGraph *nome = new TGraph (numero_punti, *nome_arrayx, *nome_arrayy)

dove le array sono passate con il puntatore (nome preceduto da asterisco).
Un altro importante costruttore di TGraph è quello che prende i punti da un file esterno, la sua inizializzazione è

TGraph *nome = new TGraph (*filename, *format="%lg %lg")

dove il format fornisce informazioni al programma su come prendere i dati dal file di testo, in questo caso "%lg %lg" dice di prendere le prime due colonne di numeri double separate da un blank delimiter (spazio bianco). Se si volesse saltare una colonna basta aggiungere un'asterisco: "%lg %*lg %lg". Se si inseriscono più di due colonne (al minimo devono essere due), si crea un grafico a più dimensioni. Se il file di dati è formato solamente dal numero di colonne necessario e non bisogna escludere nessuna colonna è possibile inizializzare il grafico semplicemente come

TGraph *nome = new TGraph (*filename)

3.2 *Costruttore di TGraphErrors*

TGraphErrors si inizializza alo stesso modo di tGraph ma bisogna aggiungere l'informazione sugli errori. Il primo metodo diventa:

TGraphErrors *nome = new TGraphErrors (numero_punti, *nome_arrayx, *nome_arrayy, *nome_errx, *nome_erry)

dove err_x ed err_y sono array conteneti gli errori per ogni coordinata di ogni punto del grafico. Per specificare che non si vogliono inserire errori in una variabile, ad esempio la x, basta sostituire uno 0 a *nome_errx

TGraphErrors *nome =new TGraphErrors(numero_punti, *nome_arrayx, *nome_arrayy, 0, *nome_erry);

Il secondo metodo invece diventa:

TGraphErrors *nome = new TGraphErrors (*filename, *format="%lg %lg %lg"))

dove questa volta il numero minimo di colonne è 3. A rigore dovrebbe essere 4 (le prime due sono le misure di x ed y e le seconde i relativi errori) ma se sono solo 3 si interpreta l'ultima come gli errori sia sulla x che sulla y. Anche in questo caso se il file dei dati è formato solamente dalle colonne necessarie e non serve escludere colonne la sintassi si riduce a

TGraphErrors *nome = new TGraph (*filename)

ATTENZIONE: LA VIRGOLA NON É LETTA COME SEPARATORE DECIMALE, SERVE IL PUNTO.

3.3 *Metodi per grafici*

Di seguito i principali metodi dei grafici

- nome -> SetTitle(“Titolo; Titolo_asseX; Titolo_asseY");
- nome -> SetMarkerStyle(kOpenCircle);
- nome -> SetMarkerStyle(numero);
- nome -> SetMarkerColor(kBlue);
- nome -> SetLineWidth(2);
- nome -> SetLineColor(kBlue);
- nome -> GetPoint(i, x, y); prende l’iesimo punto del grafico
- nome -> GetX ()/GetY (); ritorna un puntatore ad un’array con tutte le coordinate x (o y).
- nome -> GetN(); ritorna il numero di punti nel grafico
- nome -> Integral(); calcola l’integrale da inizio a fine grafico
- nome -> AddPoint(x,y); aggiunge un punto di coordinate x, y al grafico
- nome -> SetPoint (i,x,y); sovrascrive il valore dell’iesimo punto del grafico
- nome -> GetCorrelationFactor(); calcola il coefficiente di correlazione lineare fra i punti
- nome -> GetCovariance(); calcola la covarianza fra i punti
- nome -> Fit("formula"); effettua un fit a partire da una formula scritta come una stringa; deve essere scritta nella forma [0]+x*[1]
- nome -> Fit(nome_funzione); effettua un fit a partire da una funzione precedentemente definita

(kOpenCirclee kBlue sono codici predefiniti in ROOT, li ho messi per esempio)

4 VARIABILI GLOBALI

I puntatori globali puntano ad informazioni generali sul file in uso sono principalmente 4:

- gROOT
  informazione globale relativa alla sessione corrente con cui si può accedere a qualunque oggetto in essa
- gFile
  puntatore al root file corrente
- gStyle
  puntatore alle funzionalità di stile
- gRandom
  puntatore al generatore di numeri casuale, che verrà approfondito nella sezione 8.

Di solito si crea una funzione prima di cominciare la funzione principale della macro in cui si settano le variabili globali, ad esempio void SetStyle() { gROOT->SetStyle("Plain"); gStyle->SetPalette(57); gStyle->SetOptTitle(0); }

5 FUNZIONI DI ROOT

Le funzioni sono oggetti che possono essere definite dall'utente o prese dagli oggetti di ROOT (ad esempio gaussiana, esponenziale, polinomiale,...). Le funzioni in generale sono di tipo TF seguito da un numero che ne indica la dimensione; ad esempio quelle ad una variabile sono contraddistinte dall'"1" alla fine del nome del tipo.

5.1 *Inizializzazione e funzioni utili*

Le funzioni definite dal'utente possono essere dichiarate in tre modi:

- Stringa di caratteri nel costruttore

  TF1 *nome1 = new TF1("nome1", "sin(x)/x“, x_iniziale, x_finale);

  É inoltre possibile usare funzioni già esistenti

  TF1 *nome2 = new TF1("nome2", "nome1*x“, x_iniziale, x_finale);

- Stringa di caratteri con parametri da inizializzare

  TF1 *nome = new TF1 ("nome",“ [0]*x*sin([1]*x)",x_iniziale, x_finale);

  dove i numeri fra parentesi quadre sono parametri liberi che possono essere definiti con

  nome -> SetParameter(numero_parametro, valore);

  dove "numero_parametro" è il numero del parametro che vogliamo definire (nel nostro esempio 0 o 0 1) e "valore" è il valore che vogliamo dare a quel parametro.
  É del tutto equivalente inizializzare la funzione specificando nel costruttore il numero di parametri:

  TF1 *nome = new TF1 ("nome","funzione",x_iniziale, x_finale, numero_parametri);

  In questo caso sarà possibile definire tutti i parametri in una volta con

  nome -> SetParameters(valore1,valore2, ...);

- Definita in una macro
  Si usa la sintassi di C++, ad esempio:

  MyFunction(Double_t *x, Double_t *par){ Float_txx =x[0];
  Double_t val= TMath::Abs(par[0]*sin(par[1]*xx)/xx);
  return val; };

Esistono anche funzioni built-in ROOT che si dichiarano così:

TF1 *nome_funz = new TF1("nome_funz", "funzione");

dove "funzione" indica il nome della funzione di ROOT, alcuni esempi sono: linear, poln, gaus, expo... Ad esempio, la funzione gaus ha 3 parametri ed è definita come:

[0]*exp(-0.5 *(x-[1])/[2]) *(x-[1])/[2]))

dove i parametri rappresentano:

[0] = ampiezza massima in corrispondenza della media
[1] = media
[2] = sigma

Alcune funzioni membro delle funzioni sono:

- nome -> Eval(x);
  valuta la funzione in quel punto
- nome -> Integral(a, b)
  calcola l'integrale fra i punti a e b

5.2 *Fitting*

Per effettuare un fit di dati in un istogramma rispetto ad una funzione definita dall'utente bisogna:

1. Creare una funzione con parametri liberi che pensiamo sia adeguata a fittare i dati

   TF1 *nome_funz = new TF1("nome_funz","funzione con parametri", x_min, x_max);

2. Effettuare il fit, che non fa altro che assegnare parametri

   nome_isto -> Fit("nome_funz");

3. Per accedere alla funzione di fit usiamo una funzione dedicata per prenderla:

   TF1 *fit_funz = h->GetFunction("nome_funz");

   ora è una normale funzione; possiamo prendere i parametri con la funzione

   fit_funz -> GetParameter(i);
   fit_funz -> GetParameters(nome_array);

   il primo metodo prende solamente l'iesimo parametro mentre il secondo crea una array con tutti i parametri.

Per effettuare un fit a partire da una funzione built-in ROOT basta scrivere:

nome_isto -> Fit(funzione);

dove al posto di funzione si può mettere il nome di una funzione built-in ROOT (linear, poln, gaus, expo...) Di seguito altre utili funzioni membro delle funzioni fit

- fit_funz -> GetChisquare(); prende il Chiquadro
- fit_funz -> GetNDF(); prende i gradi di libertà
- fit_funz -> GetParError(i); prende l'errore dell'iesimo parametro
- fit_funz -> GetParErrors(nome_array); crea un'array con tutti gli errori sui parametri. Si potrebbe anche creare un'oggetto array ed assegnargli la funzione con le parentesi vuote.

É possibile fittare anche grafici grazie all'overloading della funzione Fit:

nome_grafico -> Fit("nome_funzione");

Fit in sotto-range

Spesso capita che si voglia fittare solo una parte di grafico o istogramma. Esistono vari modi per farlo:

- Si dichiara un range di validità (dominio) nel costruttore della funzione e poi si fitta usando l'opzione "R":

  TF1 *nome_funz = new TF1("nome_funz", "funzione", range_min, range_max);
  nome_isto -> Fit("nome_funz", "R");

- Specificare come opzione di Fit il range. In questo caso la funzione si dichiara nel modo classico e per fittare si scrive:

  nome_isto -> Fit("nome_funz", " ", " ", range_min, range_max);

Se si specifica il range è possibile fittare utilizzando più funzioni per i diversi range. Gli step da seguire sono:

1. Definire le n funzioni con cui si vuole fittare (di solito si usano le built-in come "gaus" da mettere al posto di "fi") e fare la somma:

   TF1 *nome_funz1 = new TF1(“nome_funz1", f1, range1, range2);
   TF1 *nome_funz2 = new TF1(“nome_funz2", f2, range3, range4);
   TF1 *nome_funzn = new TF1(“nome_funz3", fn, rangen-1, rangen);
   //funzionesomma
   TF1 *total = new TF1("total","f1+f2+...+fn", range1, rangen);

   dove f1,...,fn sono le funzioni pre definite con cui si vuole fittare.

2. Fittare ogni sotto-range con le funzioni

   nome_isto -> Fit(f1, "R"); nome_isto -> Fit(f2, "R"); nome_isto -> Fit(fn, "R");

3. Recuperare i parametri dopo il fit ad esempio in una array

   Double_t nome_array[numero_parametritot];
   g1->GetParameters(&nome_array[i]);
   g2->GetParameters(&nome_array[j]);
   g3->GetParameters(&nome_array[k]);

   dove la "&" serve perchè si passa l'array by-reference e i numeri i,, j, k indicano i parametri che si devono prendere dalle singole funzioni.

4. Inserire i parametri risultanti in “total” (funzione somma)

   total -> SetParameters(nome_array);

5. Fittare nuovamente con la funzione somma per una stima definitiva dei parametri:

   nome_isto -> Fit(total, "R")

6. Estrarre risultati finali dai metodi delle funzioni fit già visti.

6 LEGENDA

La legenda è un oggetto a se stante (TLegend) che deve essere disegnato sulla canvas. Un esempio è:

```
TLegend *nome = new TLegend(.1,.7,.3,.9,“Titolo");
nome -> AddEntry(oggetto1, "label1");
nome -> AddEntry(oggetto2, “label2");
nome -> Draw("SAME");
```

dove i numeri nella dichiarazione di TLegend sono parametri per le dimensioni del rettangolo della legenda; AddEntry aggiunge alla legenda oggetti (quindi non solo istogrammi o grafici ma anche numeri, ad esempio il Chiquadro).

7 DISEGNARE OGGETTI

7.1 *Canvas*

Per prima cosa bisogna creare una "canvas" che è l’ambiente in cui è possibile disegnare gli oggetti. Un modo base per inizializzarla è

```
TCanvas *nome = new TCanvas();
```

In questo modo si crea una canvas vuota di dimensioni automatiche. É inoltre possibile specificare il nome, il titolo e le dimensioni:

```
TCanvas *nome = new TCanvas("nome", "titolo", larghezza, altezza);
```

É infine possibile dividere una canvas il più "pads" per accostare diversi grafici.

```
nome -> Divide(larghezza1, altezza1,larghezza2, altezza2);
```

oppure, se si volesse dividere in 4 parti uguali:

```
nome -> Divide(2, 2);
```

Infine, è possibie salvare il la canvas come immagine con il seguente comando

```
nomecanvas -> Print("nomefile.C/.png/.gif/.pdf/.jpg...")
```

7.2 *Draw()*

Una volta creata la canvas, per disegnare gli istogrammi si usa il metodo Draw(). Di base è possibile disegnare un oggetto semplicemente con

```
nome -> Draw();
```

Esistono però molte opzioni di questa funzione, riportate in appendice A. Le opzioni si inseriscono con

```
nome -> Draw("lettere")
```

Le opzioni si possono unire con virgole o semplicemente scrivendo le lettere di seguito.

8 GENERAZIONE DI MONTECARLO

I generatori di numeri casuali (PRNG) sono di fondamentale importanza in fisica per studiare eventi complessi formati da numerose acquisizioni dati. Un PRNG può seguire una qualsiasi p.d.f.; in ROOT è possibile generare numeri casuali che seguono p.d.f. built-in ROOT o user-defined. Alcuni esempi di p.d.f. built-in ROOT sono:

- Uniform(x1, x2) vuole la x iniziale e finale fra cui generare
- Gaus(media, sigma) vuole media e deviazione standard
- Poisson(media) vuole la media
- Binomial(ntot, prob) vuole il numero totale di elementi e la probabilità che avvenga l'evento binario.
- Exp(tau) vuole il coefficiente che sta all'esponente
- Integr(imax)
- Landau(moda, sigma) vuole moda e deviazione standard

Un esempio di generazione casuale apartire da una built-in, utilizzando il metodo gRandom è:

Double_t x = gRandom -> Uniform(xmin, xmax);

É inoltre possibile generare numeri casuali partendo da una funzione user-defined utilizzando il metodo GetRandom():

TF1 *nome_funz = new TF1("nome_funz","funzione_userdef",x_min, x_max);
doubler = nome_funz -> GetRandom();

É anche possibile riempire direttamente un'istogramma di entrate casuali con il metodo FillRandom():

nome_isto -> FillRandom("funzione", num_entrate);

dove "funzione può essere sa user-defined che built-in.

A opzioni di draw

Option	Description
"A"	Axis are drawn around the graph
"I"	Combine with option 'A' it draws invisible axis
"L"	A simple polyline is drawn
"F"	A fill area is drawn ('CF' draw a smoothed fill area)
"C"	A smooth Curve is drawn
"*"	A Star is plotted at each point
"P"	The current marker is plotted at each point
"B"	A Bar chart is drawn
"1"	When a graph is drawn as a bar chart, this option makes the bars start from the bottom of the pad. By default they start at 0.
"X+"	The X-axis is drawn on the top side of the plot.
"Y+"	The Y-axis is drawn on the right side of the plot.
"PFC"	Palette Fill Color: graph's fill color is taken in the current palette.
"PLC"	Palette Line Color: graph's line color is taken in the current palette.
"PMC"	Palette Marker Color: graph's marker color is taken in the current palette.
"RX"	Reverse the X axis.
"RY"	Reverse the Y axis.

Option	Description
"E"	Draw error bars.
"AXIS"	Draw only axis.
"AXIG"	Draw only grid (if the grid is requested).
"HIST"	When an histogram has errors it is visualized by default with error bars. To visualize it without errors use the option "HIST" together with the required option (eg "hist same c"). The "HIST" option can also be used to plot only the histogram and not the associated function(s).
"FUNC"	When an histogram has a fitted function, this option allows to draw the fit result only.
"SAME"	Superimpose on previous picture in the same pad.
"SAMES"	Same as "SAME" and draw the statistics box
"PFC"	Palette Fill Color: histogram's fill color is taken in the current palette.
"PLC"	Palette Line Color: histogram's line color is taken in the current palette.
"PMC"	Palette Marker Color: histogram's marker color is taken in the current palette.
"LEGO"	Draw a lego plot with hidden line removal.
"LEGO1"	Draw a lego plot with hidden surface removal.
"LEGO2"	Draw a lego plot using colors to show the cell contents When the option "0" is used with any LEGO option, the empty bins are not drawn.
"LEGO3"	Draw a lego plot with hidden surface removal, like LEGO1 but the border lines of each lego-bar are not drawn.
"LEGO4"	Draw a lego plot with hidden surface removal, like LEGO1 but without the shadow effect on each lego-bar.
"TEXT"	Draw bin contents as text (format set via `gStyle->SetPaintTextFormat`).
"TEXTnn"	Draw bin contents as text at angle nn (0 < nn < 90).
"X+"	The X-axis is drawn on the top side of the plot.
"Y+"	The Y-axis is drawn on the right side of the plot.
"MIN0"	Set minimum value for the Y axis to 0, equivalent to gStyle->SetHistMinimumZero().

Option	Description
"E"	Draw error bars.
"AXIS"	Draw only axis.
"AXIG"	Draw only grid (if the grid is requested).
"HIST"	When an histogram has errors it is visualized by default with error bars. To visualize it without errors use the option "HIST" together with the required option (eg "hist same c"). The "HIST" option can also be used to plot only the histogram and not the associated function(s).
"FUNC"	When an histogram has a fitted function, this option allows to draw the fit result only.
"SAME"	Superimpose on previous picture in the same pad.
"SAMES"	Same as "SAME" and draw the statistics box
"PFC"	Palette Fill Color: histogram's fill color is taken in the current palette.
"PLC"	Palette Line Color: histogram's line color is taken in the current palette.
"PMC"	Palette Marker Color: histogram's marker color is taken in the current palette.
"LEGO"	Draw a lego plot with hidden line removal.
"LEGO1"	Draw a lego plot with hidden surface removal.
"LEGO2"	Draw a lego plot using colors to show the cell contents When the option "0" is used with any LEGO option, the empty bins are not drawn.
"LEGO3"	Draw a lego plot with hidden surface removal, like LEGO1 but the border lines of each lego-bar are not drawn.
"LEGO4"	Draw a lego plot with hidden surface removal, like LEGO1 but without the shadow effect on each lego-bar.
"TEXT"	Draw bin contents as text (format set via `gStyle->SetPaintTextFormat`).
"TEXTnn"	Draw bin contents as text at angle nn (0 < nn < 90).
"X+"	The X-axis is drawn on the top side of the plot.
"Y+"	The Y-axis is drawn on the right side of the plot.
"MIN0"	Set minimum value for the Y axis to 0, equivalent to gStyle->SetHistMinimumZero().

B esempi

```
void setStyle(){
  gROOT->SetStyle("Plain");
  gStyle->SetOptStat(1111);
  gStyle->SetOptFit(111);
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(0);
}



void MonteCarlo() {
// funzione principale della macro


   Int_t ngen=100000;

   Int_t ntot[4]={5,10,50,100};
   Int_t nbins[4]={5,10,50,100};
   Double_t prob=0.3;
//creazione istogramma
   TString names[4]={"test1","test2","test3","test4"};
   Double_t xmin[4]={0,0,0,0};
   Double_t xmax[4]={5,10,50,100};
TH1F *h[4];

   for (Int_t i=0;i<4;i++){
   h[i] = new TH1F("h_"+names[i],"Binomial "+names[i],nbins[i],xmin[i],xmax[i]);
//cosmetica: assi, colore, spessore linea,tipo di Marker...
      h[i]->GetXaxis()->SetTitle("x");
   h[i]->GetYaxis()->SetTitleOffset(1.3);
   h[i]->GetYaxis()->SetTitle("Occorrenze");
   h[i]->SetFillColor(kBlue);
   h[i]->SetLineWidth(2);
   h[i]->SetMarkerStyle(4);
      h[i]->SetMinimum(0);
    for(Int_t j=0;j<ngen;j++){
      Double_t x=gRandom->Binomial(ntot[i],prob);
         h[i]->Fill(x);
}
}


//Questa è una Canvas, la finestra grafica....
 TCanvas *cRandom = new TCanvas("cRandom","Test Distribuzione Uniforme");
 cRandom->Divide(2,2);

 for (Int_t i=0;i<4;i++){
  cRandom->cd(i+1);
  h[i]->Draw();
   cout << "*-------------------------------------------------------------*" <<endl;

 cout << "Occorrenze Totali: " <<h[i]->GetEntries() <<endl;
 cout << "Media dell'istogramma: " <<h[i]->GetMean() << " +/- " <<h[i]->GetMeanError()<<endl;
 cout << "RMS dell'istogramma: " <<h[i]->GetRMS() << " +/- " <<h[i]->GetRMSError()<<endl;
}
}
```

Figura 1: Programma che verifica che una binomiale tende ad una gaussiana usando la generazione di Montecarlo.

```
void grafico(){

  // The values and the errors on the X and Y axis
   const int n_points=10;
   double x_vals[n_points]={1,2,3,4,5,6,7,8,9,10};
   double y_vals[n_points]={6,12,14,20,22,24,35,45,44,53};
   double y_errs[n_points]={5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43};
   double x_errs[n_points]={1,1,1,1,1,1,1,1,1,1};


   // Instance of the graph
   TGraphErrors * graph =new TGraphErrors(n_points,x_vals,y_vals,0,y_errs);
   //TGraphErrors * graph =new TGraphErrors(n_points,x_vals,y_vals,x_errs,y_errs);
   //TGraphErrors * graph =new TGraphErrors("grafico.dat","%lg %lg %*lg %lg");
   //TGraphErrors * graph =new TGraphErrors("grafico.dat","%lg %lg %lg %lg");

   graph->SetTitle("Verifica della legge di Ohm;DDP (V); Corrente (A)");

   // Cosmetics
   graph->SetMarkerStyle(kOpenCircle);
   graph->SetMarkerColor(kBlue);
   graph->SetLineColor(kBlue);

   // The canvas on which we'll draw the graph
   TCanvas *myCanvas = new TCanvas();

   // Define a linear function
   TF1 *f = new TF1("linear","[0]+x*[1]",.5,10.5);
   // Let's make the function line nicer
   f->SetLineColor(kRed); f->SetLineStyle(2);
   // Fit it to the graph and draw it
   graph->Fit(f);

   // Draw the graph !
   gStyle->SetOptFit(111);
   graph->Draw("APE");

   cout << "x and y measurements correlation factor=" <<
   graph->GetCorrelationFactor()<<endl;
   cout << "The Chisquare of the fit = "<< graph->Chisquare(f)<<endl;
   cout << "From function  "<<f->GetChisquare() << endl ;
   cout << "From function  "<<f->GetNDF() << endl ;

  // Build and Draw a legend
   TLegend *leg=new TLegend(.1,.7,.3,.9,"Prova di Laboratorio");
   leg->AddEntry(graph,"Punti Sperimentali");
   leg->AddEntry(f,"Fit Lineare");
   leg->Draw("Same");

   myCanvas->Print("LeggeDiOhm.gif");
}
```

Figura 2: Programma che effettua il plot di dati con errore su una delle due misure usando TGraphErrors.

```
//General settings for graphics
void setStyle(){
  gROOT->SetStyle("Plain");
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(0);
}

void maxwell() {
// funzione principale della macro

//creazione istogramma
  TH1F *h1 = new TH1F("h1","Tempi di Caduta 1",8,-0.5,15.5);
  TH1F *h2 = new TH1F("h2","Tempi di Caduta 2 ",8,-0.5,15.5);

//lettura da file ascii e riempimento istogramma
  ifstream in;

  in.open("maxwell1.dat");
  Float_t x,y;
  while (1) {
    in >> x >> y;
    if (!in.good()) break;
    h1->Fill(y);
  }
  in.close();

  in.open("maxwell2.dat");
  while (1) {
    in >> x >> y;
    if (!in.good()) break;
     h2->Fill(y);
  }
  in.close();

   // Istogrammi rapporto e somma
  TH1F *hRatio=new TH1F(*h1);
  // hRatio->Sumw2(); //importante per avere gli errori corretti
   hRatio->Divide(h1,h2,1,1);

   TH1F *hSum=new TH1F(*h1);
   hSum->Sumw2(); //importante per avere gli errori corretti
   hSum->Add(h1,h2,1,1);

//Canvas, la finestra grafica (divisa in due)
 TCanvas *cMaxwell = new TCanvas("cMaxwell", "Tempi di caduta del pendolo di Maxwell",10,30,1000,600);
cMaxwell->Divide(2,1); //divisa in 2 pad
cMaxwell->cd(1);
//cosmetica: assi, colore, spessore linea,tipo di Marker...
 hRatio->GetXaxis()->SetTitle("Tempi di Caduta (s)");
 hRatio->GetYaxis()->SetTitleOffset(1.3);
 hRatio->GetYaxis()->SetTitle("h1/h2");
 hRatio->SetMarkerStyle(4); //Open Circle // altrimenti kOpenCircle
 hRatio->SetLineWidth(2); //spessore linea
 hRatio->Draw();
 hRatio->Draw("E,same");

cMaxwell->cd(2);
//cosmetica: assi, colore, spessore linea,tipo di Marker...
 hSum->GetXaxis()->SetTitle("Tempi di Caduta (s)");
 hSum->GetYaxis()->SetTitleOffset(1.3);
 hSum->GetYaxis()->SetTitle("Occorrenze");
 hSum->SetFillColor(kBlue); //(KBlue-2)
 hSum->SetMarkerStyle(4); //Open Circle // altrimenti kOpenCircle
 hSum->SetLineWidth(2); //spessore linea
 gStyle->SetOptStat(112210); //no nome, entries, media e rms con errori, under (over)flow
 hSum->Draw("histo");
 hSum->Draw("E,same");

 cout << "*-------------------------------------------------------*" <<endl;
 cout << "Occorrenze Totali: " <<hSum->GetEntries() <<endl;
 cout << "Media dell'istogramma: " <<hSum->GetMean() << " +/- " <<hSum->GetMeanError()<<endl;
 cout << "RMS dell'istogramma: " <<hSum->GetRMS() << " +/- " <<hSum->GetRMSError()<<endl;
 cMaxwell->Print("cMaxwell.pdf");
 cMaxwell->Print("cMaxwell.C");
 cMaxwell->Print("cMaxwell.root");
}
```

Figura 3: Programma che crea due istogrammi con i tempi di caduta del pendolo di Maxwell prendendo dati da un file esterno usando ifstream.