Guida ROOT

Fabio Prestipino

16 gennaio 2023

INDICE

Per istogrammi saranno convenzionalemnte usati i nomi h, ho, h1,...; per le funzioni f, fo, f1, f2,...; per le funzioni di fit fit, fit1,...; per i grafici g, g1, g2, ...; l per le legende e c per le canvas

1 INTRODUZIONE ED INFORMAZIONI GENERALI UTILI

ROOT è un pacchetto software orientato 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**
  Esistono due metodi: uno è quello **interpretato**. 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

  Se la macro è formata da una singola funzione si può usare direttamente il comando

  .x myMacro.C

  per far partire la funzione. É anche possibile caricare le funzioni della macro in memoria con il comando

  .L name.C

  per poi chiamare le funzioni singolarmente, ad esempio:

  funz1();

  IN MODALITA INTERPRETATA NON SI DEVONO INCLUDERE I PACCHETTI ROOT.
  Un metodo alternativo è quello **compilato**: si includono tutti gli oggetti di ROOT necessari e si scrive una macro (che si presenta sempre come una o più funzioni C++) che dovrà essere compilata esternamente (ad esempio con un classico g++) o direttamente in ROOT con il comando

  .L myMacro.C+

  (si faccia attenzione all'aggiunta del "+" a fine comando) per poi eseguirla come una normale funzione

  myMacro()

  IMPORTANTE: LE MACROS DEVONO STARE NELLA CARTELLA IN CUI VIENE APERTO ROOT

- **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 ROOT FILES

Esistono file di tipo .root all'interno dei quali si possono scrivere (e quindi salvare in memoria per poi poterli leggere in altri files) oggetti ROOT come funzioni e istogrammi o oggetti user-defined (trattati in seguito). I file .root sono gestiti con la classe TFile, l'idea è che si crea un oggetto di tipo TFile che si inizializza o assegnandogli un file .root preesistente o generandone uno ex-novo, in questo modo potremo aprire, leggere e scrivere quel file per poi salvarlo e chiuderlo.
Possiamo inizializzare un file in vari modi, a seconda di ciò che vogliamo fare con questo: possiamo aprirlo "in lettura" (se non specifichiamo opzioni), "in scrittura" (con opzione "RECREATE" crea un nuovo file e se già esisteva lo sovrascrive, "NEW" e "CREATE" creano un nuovo file e se già esiste non fanno niente), in "modifica" (l'opzione "UPDATE" apre un file che, se già esistente, può essere modificato e se non esistente viene creato). La sintassi è:

TFile *file = new TFile("example.root","OPZIONE");

Per listare il contenuto del file esiste il metodo ls:

file->ls();

Una volta aperto il file è possibile accedere ai suoi elementi con il metodo Get

TH1F *histo=(TH1F*)file->Get("histo");

Per scrivere un singolo oggetto sul file si usa il metodo:

object -> Write();

Se invece si vogliono scrivere sul file tutti gli oggetti nella memoria corrente (ovvero inizializzati nel file fino ad ora):

file->Write();

Per chiudereilfile root siusailmetodo Close:

file->Close();

2.1 *Decoupling objects*

Normalmente gli oggetti di un file si dicono "coupled" con il file, cioè spariscono dalla canvas una volta chiuso il file. Per evitare che ciò avvenga bisogna effettuare il "decouple" con uno dei tre seguenti metodi :

object->AddDirectory(kFALSE); (statico)
object->SetDirectory(kFALSE); (non statico)
object->DrawCopy() (non statico)

3 ISTOGRAMMI

3.1 *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 *h = new TH1F("h", "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. Possiamo inizializzare anche un'istogramma senza passare alcun parametro

TH1F *h = new TH1F();

Oppure con il copy constructor, passando by pointer un altro istogramma

TH1F *h1 = new TH1F("h1", "histo1", n-bin, x_{min}, x_{max});

TH1F *h2 = new TH1F(*h1);

Per aggiungere punti ad un'istogramma unidimensionale:

h -> Fill(x);

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

h -> Fill(x, 8);

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

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

Per aggiungere punti in un'istogramma bidimenzionale:

h -> Fill(x,y);

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

h1 -> ProjectionX();
h2 -> ProjectionY();

che crea un'istogramma già riempito.

3.2 *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;
h -> Fill(x);
if(!nome.good()) break;
}
nome.close();
```

Dove "nome" è il nome dell'oggetto-file che inizializziamo e a cui assegniamo il file di testo che si desidera aprire, che si chiama "nome_file.txt". Il metodo nome.good() dice se il file è concluso o meno. Nell'ultima riga è stato chiuso il file.

3.3 *Metodi utili istogrammi*

Di seguito sono riportati i metodi più importanti per gli istogrammi

- h ->GetXaxis()->SetTitle("titolo"); setta il titolo dell'asse
- h -> SetMarkerStyle(numero);
  assegna uno stile al marker
- h -> SetMarkerSize(numero);
  assegna una grandezza al marker
- h -> GetBinError(nbin);
  prende l'errore di un bin
- h -> GetBinContent(nbin); prende il numero di entrate nel bin
- h -> GetBinContent(0); prende il numero di entrate in underflow
- h -> GetBinContent(nbin+1); prende il numero di entrate in overflow
- h -> GetEntries();
  prende il numero di entrate totali
- h -> SetBinContent(ibin,val);
  assegna al contenuto del bin un valore val
- h -> GetMean();
  calcola la media
- h -> GetRMSError();
  calcola l'errore sulla media
- h -> GetRMS();
  calcola la deviazione standard
- h -> GetMeanError();
  calcola l'errore sulla deviazione standard
- h -> GetMaximum();
  prende il bin con numero massimo di entrate
- h -> GetMaximumbin();
  prende il numero del bin con numero massimo di entrate

- h -> GetBinCenter();
  prende il centro del bin
- h -> GetIntegral();
  restituisce un'array dei conteggi cumulativi
- h -> Integral(bini, binf);
  calcola l'integrale nel range

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

   Per la somma:

   TH1F *hsum =new TH1F(*h1);
   hsum -> 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() prima di effettuare la somma prima di riempire l'istogramma:

   h1 -> Sumw2();
   h2-> Sumw2();

4 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.

4.1 *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 *g = new TGraph (n_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 *g = 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 *g = new TGraph (*filename)

4.2 *Costruttore di TGraphErrors*

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

TGraphErrors *g = new TGraphErrors (n_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 *g =new TGraphErrors(n_punti, *nome_arrayx, *nome_arrayy, 0, *nome_erry);

Il secondo metodo invece diventa:

TGraphErrors *g = 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.

4.3 *Metodi per grafici*

Di seguito i principali metodi dei grafici

- g -> SetTitle("Titolo; Titolo_asseX; Titolo_asseY");
- g -> SetMarkerStyle(kOpenCircle);
- g -> SetMarkerStyle(numero);
- g -> SetMarkerColor(kBlue);
- g -> SetLineWidth(2);
- g -> SetLineColor(kBlue);
- g -> GetPoint(i, x, y); prende l'iesimo punto del grafico
- g -> GetX ()/GetY (); ritorna un puntatore ad un'array con tutte le coordinate x (o y).
- g -> GetN(); ritorna il numero di punti nel grafico

- g -> Integral(); calcola l'integrale da inizio a fine grafico
- g -> AddPoint(x,y); aggiunge un punto di coordinate x, y al grafico
- g -> SetPoint (i,x,y); sovrascrive il valore dell'iesimo punto del grafico
- g -> GetCorrelationFactor(); calcola il coefficiente di correlazione lineare fra i punti
- g -> GetCovariance(); calcola la covarianza fra i punti
- g -> Fit("formula"); effettua un fit a partire da una formula scritta come una stringa; deve essere scritta nella forma [0]+x*[1]
- g -> Fit(nome_funzione); effettua un fit a partire da una funzione precedentemente definita

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

5 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 10.

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); }

6 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.

6.1 *Inizializzazione e funzioni utili*

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

- Stringa di caratteri nel costruttore (riconosce le funzioni della libreria cmath)

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

  É inoltre possibile usare e modificare funzioni già esistenti

  TF1 *f1 = new TF1("f1", "f*x", x_iniziale, x_finale);

- Stringa di caratteri con parametri da inizializzare

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

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

  f -> 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 *f = new TF1 ("f","funzione",x_iniziale, x_finale, numero_parametri);

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

  f -> 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; };
  e poi si inizializza

  TF1* f = new TF1("f", "MyFunction(x, par)", 0, 10);

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

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

dove "funzione" indica il nome della funzione di ROOT, alcuni esempi sono:

- poln: [0] + [1]*x + [2]*x2 +...
- gaus: [0]*exp(-0.5 *(x-[1])/[2])2̂

  dove i parametri rappresentano:

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

- expo: exp([0]+[1]*x)

  dove i parametri rappresentano:

  [0] = ampiezza
  [1] = media

- gausn: [0]*exp(-0.5*((x-[1])/[2])2̂)/([2] *sqrt(2PI))

Alcuni metodi delle funzioni sono:

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

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

2. 'E altamente consigliato (talvolta necessario se no non fitta) settare i parametri che ci aspettiamo di ottenere approssimativamente dal fit con l'apposita funzione

   fit->SetParameter(n_parameter, value);

3. Effettuare il fit, che non fa altro che assegnare parametri alla funzione del fit

   h -> Fit("fit");

4. Possiamo prendere i parametri con la funzione:

   fit -> GetParameter(n_parameter);
   fit -> 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, gausn...). Per evitare di stampare informazioni indesiderate si può usare l'opzione "Q".

nome_isto -> Fit(funzione, "Q");

Di seguito altre utili funzioni membro delle funzioni fit

- fit -> GetChisquare(); prende il Chiquadro
- fit -> GetNDF(); prende i gradi di libertà
- fit -> GetParError(i); prende l'errore dell'iesimo parametro
- fit -> 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:

g -> Fit("nome_funzione");

/.../lab-root-2/ex7.C http://localhost:37941/d655b76c-c5a9-4a07-b61b-f0c7cfc5841f/

```
void macroo(){
    TH1F *h1= new TH1F("h1", "H1", 500, 0., 5.);
    TH1F *h2= new TH1F("h2", "H2", 500, 0., 5.);

    gRandom -> SetSeed();

    for(int i=0; i<10e6; i++){ //generazione esplicita
        double x = gRandom -> Gaus(2.5, 0.25);
        h1->Fill(x);
        if (i<10e4){
        double y = gRandom -> Uniform(0., 5.);
        h2->Fill(y);}
    }

    h1 -> Sumw2();
    h2 -> Sumw2();

    TH1F *h3 = new TH1F("h3", "H3", 500, 0., 5.);
    h3 -> Add(h1, h2, 1, 1); //somma

    //TH1F *h3 = new TH1F(*h1);
    //h3 -> Add(h2, 1);
    TF1 *fit = new TF1("fit", "gaus + [3]", 0., 5.);
    fit->SetParameter(1, 2.5);
    fit->SetParameter(2, 0.25);

    h3 -> Fit("fit");

    std::cout << "Ampiezza= " << fit -> GetParameter(0) << "+/-" << fit -> GetParError(0) <<
'\n';
    std::cout << "Media= " << fit -> GetParameter(1) << "+/-" << fit -> GetParError(1) <<
'\n';
    std::cout << "Deviazione standard= " << fit -> GetParameter(2) << "+/-" << fit ->
GetParError(2) << '\n';
    std::cout << "Deviazione standard= " << fit -> GetParameter(3) << "+/-" << fit ->
GetParError(3) << '\n';
    std::cout << "Chisquare/NDF= " << fit -> GetChisquare() / fit -> GetNDF() << '\n';

}
```

1 di 1 16/01/2023, 11:55

Figura 1: Esempio di fit

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 *f = new TF1("f", "funzione", range_min, range_max);
  h -> Fit("f", "R");
  ```

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

  ```
  h -> Fit("f", " ", " ", 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 *f1 = new TF1("f1", f1, range1, range2);
   TF1 *f2 = new TF1("f2", f2, range3, range4);
   TF1 *fn = new TF1("fn", fn, rangen-1, rangen);
   TF1 *ftot = new TF1("ftot","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

   ```
   h -> Fit(f1, "R");
   h -> Fit(f2, "R");
   h -> 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)

   ```
   ftot -> SetParameters(nome_array);
   ```

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

   ```
   h -> Fit(total, "R")
   ```

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

7 CLASSI USER-DEFINED IN ROOT

In ROOT vi sono classi predefinite come quella delle funzioni o degli istogrammi, e se volessimo usare dentro ROOT una classe user-defined, che eventualmente faccia uso di altre classi ROOT? Possiamo semplicemente implementaere una classe esternamente a ROOT (con la convenzione classica di creare un file di intestazione .h ed uno di implementazione .cpp), in questo caso dovremo includere tutti gli oggetti di ROOT necessari (per includere una classe user-defined si deve usare una macro compilata). Per compilare la classe si usa la sintassi:

gROOT->LoadMacro("MyClass.cxx+")

Per usare oggetti della nostra classe in un file ROOT (e non solo in macros compilate) bisogna modificare il file .h della classe user-defined rendendola classe figlia (ereditarietà pubblica) di TObject, bisogna inoltre aggiungere la chiamata alla macro ClassDef(ClassName,N); bisogna modificare il file .cpp la chiamata a ClassImp(ClassName).

includere statement in rosso!

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include "TF1.h"
#include "TH1F.h"
class MyClass: public TObject{ //public inheritance from TObject
public:
MyClass(); //def ctor
MyClass(TH1F *hIn); //parametric ctor
//public methods
void Generate(TF1*f, int nGen); //generate according to a given function
void ShowHisto() const; //display the histogram
TH1F* GetHisto() const; //retrieve the histogram
~MyClass(); //dtor
private:
TH1F * h_; //pointer to a histogram
//for persistency, to make MyClass Objects writable on root files
ClassDef(MyClass,1)
};
#endif
```

Figura 2: Ciò che bisogna modificare alla macro (nel file .h) che definisce una classe user-defined per poter usare quella classe in un file .root

includere statement in rosso!

```
#include "MyClass.h"

MyClass::MyClass(){h_=new TH1F("h","titolo",100,-5,5);}  //def Ctor
MyClass::MyClass(TH1F *hIn){h_=new TH1F(*hIn);} //parametric ctor

void MyClass::Generate(TF1*f, int nGen){
  h_->GetXaxis()->SetRangeUser(f->GetXmin(),f->GetXmax()); //set the range to f range
  h_->FillRandom(f->GetName(),nGen); //accepts char* as arg, must use GetName()
}

void MyClass::ShowHisto() const { h_->Draw();}
TH1F* MyClass::GetHisto() const {return h_;}
MyClass::~MyClass(){delete h_;} //dtor
//for persistency, to make MyClass Objects writable on root files
ClassImp(MyClass)
```

Figura 3: Ciò che bisogna aggiungere al file .cpp che definisce la classe user-defined.

Ora possiamo scrivere l'oggetto della classe user-defined. Per leggere l'oggetto user defined scritto in un file, su un altro file, si segue il normale iter già esposto nella sezione "ROOT files. Infine, per evitare di includere più volte le stesse classi user-defined si usano le inlude guards, da includere in ogni macro che usa un'oggetto user-defined:

R__LOAD_LIBRARY(MyClass_cxx.so)

8 LEGENDA

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

TLegend *l = new TLegend(.1,.7,.3,.9,"Titolo");
l -> AddEntry(oggetto1, "label1");
l -> AddEntry(oggetto2, "label2");
l -> 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).

9 DISEGNARE OGGETTI

9.1 *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 *c = 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 *c = new TCanvas("c", "titolo", larghezza, altezza);

Per disegnare su una canvas basta chiamare il metodo Draw() su un oggetto, che sarà disegnato automaticamente sull'ultima canvas. Se però si chiama nuovamente il Draw() su un altro oggetto questo verrà sovrascritto su quello precedente, eliminandolo. Per disegnare più oggetti su una stessa canvas bisogna dividerla in pads:

c -> Divide(larghezza1, altezza1,larghezza2, altezza2);

se si vuole dividere specificando le misure. Se invece si volesse dividere in parti uguali bisogna specificare il numero di righe e colonne:

c -> Divide(n_colonne, n_righe);

La numerazione dei pads parte da in alto a sinistra e finisce in basso a destra. Per disegnare su uno specifico pad bisogna accedervi con la funzione

c->cd(n_pad);

h->Draw();

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

c -> Print("nomefile.C/.png/.gif/.pdf/.jpg...");

9.2 *Draw()*

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

nome_oggetto -> Draw();

che disegna l'oggetto sull'ultima canvas istanziata.
Esistono molte opzioni di questa funzione, riportate in appendice **??**. Le opzioni si inseriscono con

nome_oggetto -> Draw("lettere");

Le opzioni si possono unire con virgole o semplicemente scrivendo le lettere una attaccata all'altra.
Di seguito sono rimportate le principali 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().

10 GENERAZIONE DI MONTECARLO

I generatori di numeri casuali (PRNG) sono di fondamentale importanza in fisica per studiare eventi complessi formati da numerose acquisizioni dati. Innanzitutto per usare la generazione casuale abbiamo bisogno di un "seed": all'inizio del nostro codice lo settiamo (una volta sola!) per poi usarlo anche ripetutamente:

gRandom->SetSeed();

10.1 *Generazione secondo pdf*

Un PRNG può seguire una qualsiasi p.d.f., che possiamo definire con una funzione, come visto precedentemente. Un esempio di generazione casuale apartire da una pdf built-in, utilizzando il metodo GetRandom() (esplicito) è:

TF1 *f = new TF1("f","funzione_userdef",x_min, x_max);
double x = f -> GetRandom();

É anche possibile generare numeri casuali che seguono pdf built-in ROOT o user-defined. Alcuni esempi di p.d.f. built-in ROOT sono:

- Rndm()
  genera numeri casuali distribuiti uniformemente tra 0 ed 1 (utile per generare secondo definite proporzioni)
- Uniform(n_{min}, n_{max})
- Gaus(media, sigma)
- Poisson(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 al denominatore dell'esponente
- Integr(imax)

- Landau(moda, sigma)

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

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

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

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

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

```
void ex2(){
    gRandom->SetSeed();

    TH1F *h0 = new TH1F("h0", "histo0", 500, 0, 5);
    TH1F *h1 = new TH1F("h1", "histo0", 500, 0, 5);
    TH1F *sum = new TH1F("sum", "sum0", 500, 0, 5);

    for(int i = 0; i < 10e6; i++){
        float x = gRandom->Gaus(2.5, 0.25);
        h0->Fill(x);
    }
    for(int i = 0; i < 10e5; i++){
        float x = gRandom->Exp(1);
        h1->Fill(x);
    }

    h0->Sumw2();
    h1->Sumw2();
    sum -> Add(h0,h1,1,1);

    TF1 *fit = new TF1("fit", "[0]*exp(-0.5 *(x-[1])/[2] *(x-[1])/[2]) + [3] * (1/[4]) *
exp(-x/[4])");

    fit->SetParameter(1, 2.5);
    fit->SetParameter(2, 0.25);
    fit->SetParameter(4, 1.);

    sum->Fit(fit, "Q");
    sum->Draw();

  std::cout << "Par[0]: " << fit->GetParameter(0) << " +/- "<< fit->GetParError(0) << '\n';
  std::cout << "Par[1]: " << fit->GetParameter(1) << " +/- "<< fit->GetParError(1) << '\n';
  std::cout << "Par[2]: " << fit->GetParameter(2) << " +/- "<< fit->GetParError(2) << '\n';
  std::cout << "Par[3]: " << fit->GetParameter(3) << " +/- "<< fit->GetParError(3) << '\n';
  std::cout << "Par[4]: " << fit->GetParameter(4) << " +/- "<< fit->GetParError(4) << '\n';
  std::cout << "Chisquare/DOF: " << (fit->GetChisquare()) / (fit->GetNDF())<< '\n';
}
```


Figura 4: Fit di variabile generata casualmente secondo pdf con metodo di montecarlo.

10.2 *Simulare effetto smearing*

Nelle misure reali, teoricamente distribuite gaussianamente, si osserva talvolta che la gaussiana non presenta un picco preciso ma un plateau di massimo non puntiforme, questo effetto si simula con lo "smearing" di una curva gaussiana generata mediante PRNG. In pratica, al posto di riempire l'istogramma con una distribuzione di media con un valore fissato, si usa una media a sua volta distribuita secondo una pdf uniforme. Di seguito è riportato un esempio di generazione gaussiana senza e con smearing

```
double media = 1.5;
for(int i = 0; i< 10e5; i++)
h -> Fill(gRandom->Gaus(media, sigma))
```

```
double media = gRandom->Uniform(1, 2);
for(int i = 0; i< 10e5; i++)
h -> Fill(gRandom->Gaus(media, sigma))
```

```
void resolution(Double_t res=0.1, Int_t nGen=1E6){


  TString histName="h";
  TH1F *h[2];
  for(Int_t i=0;i<2;i++){
    h[i] =new TH1F(histName+i,"test histogram",90,0,3);
//cosmetics
    h[i]->SetLineColor(1);
    h[i]->GetYaxis()->SetTitleOffset(1.2);
    h[i]->GetXaxis()->SetTitleSize(0.04);
    h[i]->GetYaxis()->SetTitleSize(0.04);
    h[i]->GetXaxis()->SetTitle("x after Resolution Effect");
    h[i]->GetYaxis()->SetTitle("Entries");
  }

  h[0]->SetFillColor(4);
  h[1]->SetFillColor(2);

//first case: fixed value smeared
  Double_t fixedValue= 1.5;
  for(Int_t i=0;i<nGen;i++)h[0]->Fill(gRandom->Gaus(fixedValue,res));
//second case: Uniform distribution smeared
  for(Int_t i=0;i<nGen;i++)h[1]->Fill(gRandom->Gaus(gRandom->Uniform(1,2),res));

  TCanvas *c1 = new TCanvas("c1","Resolution Effects, Examples",200,10,600,400);
  c1->Divide(1,2);
  for(Int_t i=0;i<2;i++){
    c1->cd(i+1);
    h[i]->Draw("H");
    h[i]->Draw("E,SAME");
  }
}
```


Figura 5: Esempio di simulazione di effetto smearing

10.3 *Simulare efficienza*

Gli strumenti di rilevazione sono sempre caratterizzati da un'efficienza, espressa in percentuale e solitamente indicata con la lettera ε, che esprime l'efficienza con la quale vengono rilevati gli eventi (efficienza del 70% equivale a dire che su 10 eventi reali ne rivela 7 in media). L'efficienza può essere costante, come nell'esempio appena fatto, o può seguire l'andamento di una funzione.
Per simulare un'acquisizione dati di una variabile distribuita secondo una pdf, tenedo conto dell'efficienza dello strumento (che ora consideriamo costante) bisogna

- Generare la variabile x distribuita casualmente secondo una pdf con una PRNG ed inserira in un istogramma (contenente gli eventi reali)
- Generare una variabile y distribuita uniformemente tra 0 ed 1
- Riempire un istogramma con i valori x solo se y è minore o uguale dell'efficienza richiesta

  if(y<ε) h->Fill(x);

- Infine, per ottenere l'istogramma dell'efficienza, dividere l'istogramma delle acquisizioni tenendo conto dell'efficienza con quello dei valori reali.

Se l'efficienza è funzione della variabile x ($\varepsilon(x)$), bisogna definire la funzione che segue l'efficienza ed inserire la x nell'istogramma solo se la y generata casualmente con distribuzione uniforme tra 0 e 1 è minore della funzione $\varepsilon(x)$, valutata nello specifico x. Per valutare la funzione si usa il metodo Eval.

```
TF1 *eff = new TF1("eff", "ε(x)", x_min, x_max);
if(y< eff->Eval(x)) h->Fill(x);
```

Si ricordi di dividere infine l'istogramma che tiene conto dell'efficienza con quello dei valori reali generati per ottenere un'istogramma dell'efficienza.
É frequente che l'efficienza segua la distribuzione cumulativa della gaussiana normale standard, detta error function, che su ROOT è implementata come TMath::Erf(x). Erf prende come parametro una variabile distribuita normalmente, per normalizzare il valore di una distribuzione gaussiana di media μ e deviazione standard σ si usa

$$\frac{x-\mu}{\sigma}$$

Per definire la funzione Erf che normalizzi automaticamente la variabile ed usarla come efficienza bisogna definirla all'esterno della macro (usando la sintassi di C++) e poi inizializzarla all'interno di una funzione nella macro (come visto nella sezione sull'inizializzazione di funzioni). Le variabili di Erf vanno settate con il metodo SetParameter.

```
double effErf(double* x, double* p) {
//p[0]==media
//p[1]==sigma
//p[2]==valore di saturazione
//offset 1 e fattore 1/2 per riscalare Erf in [0,1]

    return (TMath::Erf( ( x[0] - p[0] ) / p[1] ) +1) / 2. * p[2];
}
```

```
void ex1(){
    gRandom -> SetSeed();//inizializza il seed

    TH1F *h0 = new TH1F("h0", "histo0", 100, 0, 10);//definisci due istogrammi
    TH1F *h1 = new TH1F("h1", "histo1", 100, 0, 10);

    for(int i = 0; i<10e5; i++){
        float x = gRandom->Gaus(5, 1);//Generata esplicitamente e singolarmente con variabile
gaussiana
        h0 -> Fill(x);//fill del primo istogramma

        float y = gRandom->Rndm();//genero un numero con distribuzione uniforme tra 0 e 1

        //VARIANTE 1
        // if(y <= x/10){
        //     h1 -> Fill(x);
        // }

        //VARIANTE 2
        if(x<=3 && x>= 0){
            if(y<=0.3){
                h1 -> Fill(x);
            }
        }
        else{
            if(y<=0.7){
                h1 -> Fill(x);
            }
        }
    }
    TH1F *div = new TH1F(*h1);
    div -> Divide(h1, h0, 1, 1);
    TCanvas *c = new TCanvas("c", "efficienza");
    c->Divide(2,2);
    c->cd(1);
    h0 -> Draw();
    c->cd(2);
    h1 -> Draw();
    c->cd(3);
    div -> Draw();
}
```


Figura 6

```
Double_t effErf(double* x, double* p) {
//p[0]==media
//p[1]==sigma
//p[2]==valore di saturazione
//offset 1 e fattore 1/2 per riscalare Erf in [0,1]

  return (TMath::Erf(  ( x[0] - p[0] ) / p[1] ) +1) / 2. * p[2];
}

void efficiency(Int_t nGen=1E6){
  gRandom->SetSeed();
  cout <<"random generation seed: " <<gRandom->GetSeed() <<endl;
  TH1F *h[2];
  TString histName="h";
  TString title[2]={"generated distribution","observed distribution"};
  for(Int_t i=0;i<2;i++){

    h[i] =new TH1F(histName+i,title[i],100,0,10);
//cosmetics
    h[i]->SetLineColor(1);
    h[i]->GetYaxis()->SetTitleOffset(1.2);
    h[i]->GetXaxis()->SetTitleSize(0.04);
    h[i]->GetYaxis()->SetTitleSize(0.04);
    h[i]->GetXaxis()->SetTitle("x");
    h[i]->GetYaxis()->SetTitle("Entries");
    h[i]->Sumw2();//Important
  }

  h[0]->SetFillColor(4);
  h[1]->SetFillColor(2);

//The efficiency profile
  TF1* myErf = new TF1("myErf", effErf, 0, 10., 3);
  myErf->SetParameter(0, 5.);//flexus
  myErf->SetParameter(1, 0.5);//width
  myErf->SetParameter(2, 0.7);//saturation value
  TCanvas *cFunc = new TCanvas("cFunc","Efficiency Profile",200,10,600,400);
  myErf->SetLineColor(1);
  myErf->SetMaximum(1);
  myErf->Draw();


//case: uniform distribution
  Double_t x=0,xRNDM=0;

  for(Int_t i=0;i<nGen;i++){
   //x=gRandom->Uniform(0,10);
     x=gRandom->Exp(1); //negative exponential with mu=1
     h[0]->Fill(x); //generated
     xRNDM=gRandom->Rndm();
     if( xRNDM<myErf->Eval(x))h[1]->Fill(x); //observed
  }

  TCanvas *cHisto = new TCanvas("cHisto","Efficiency effects, Generated and Observed
",200,10,600,400);
```


Figura 7

```
double pdf(double x) {
    if (x < 5) {
        return x / 5;
    } else {
        return 1;
    }
}

void ex4(){
    gRandom -> SetSeed();
    TH1F *h0 = new TH1F("h0", "histo0", 100, 0, 10);
    //Versione 1
    // TF1 *f = new TF1("f", "x", 0, 10);
    // h0->FillRandom("f", 10e5);

    //Versione 2 (metodo 1)
    // TF1* f2 = new TF1("f2", "(x/5)*(x>=0 && x<5) + (x<=10 && x>=5)", 0, 10);
    //ROOT valuta espressioni booleane (vero -> 1; falso -> 0)

    //Versione 2 (metodo 2)
    TF1* f2 = new TF1("f2", "pdf(x)", 0, 10);
    //Definisco una funzione in linguaggio C++ standard e inizializzo con quella una funzione
ROOT

    h0->FillRandom("f2", 10e5);
    h0->Draw();
}
```


Figura 8: Simulazione di efficienza con funzione definita a tratti

11 BENCHMARK

Il benchmark è un metodo per valutare la performance di una macro/programma di ROOT; esiste l'oggetto nativo di ROOT TBenchmark adatto allo scopo.
Innanzitutto bisogna creare l'oggetto TBenchmark, che come per gRandom deve essere istanziato una volta sola.

```
TBenchmark *b = new TBenchmark();
```

In seguito possiamo usare i 4 seguenti metodi

- Start: comincia il conteggio del tempo, bisogna assegnare un nome

  ```
  b->Start("nome");
  ```

- Stop: ferma il conteggio del benchmark (già iniziato) con lo stesso nome

  ```
  b->Stop("nome");
  ```

- Print: Effettua automaticamente la stampa a schermo del tempo misurato fino a quel momento, senza fermare il conteggio.
- Show: è una combinazione di Stop e Print

  ```
  b->Show("nome");
  ```

```
void ex5(){
    //oggetto benchmark
    TBenchmark *b = new TBenchmark();

    gRandom->SetSeed();
    //Operazione 1
    TH1F *h1 = new TH1F("h1", "histo1", 100, -10, 10);
    b->Start("With cycle");
    for(int i = 0; i < 10e5; i++){
        h1-> Fill(gRandom->Gaus(0., 1.));
    }
    b->Show("With cycle");
    //Operazione 2
    TH1F *h2 = new TH1F("h2", "histo2", 100, -10, 10);
    b->Start("With FillRandom");
    h2->FillRandom("gaus", 10e5);
    b->Show("With FillRandom");


   TCanvas *c = new TCanvas("c", "efficienza");
    c->Divide(2,1);
    c->cd(1);
    h1 -> Draw();
    c->cd(2);
    h2 -> Draw();
}
```


Figura 9: Esempio di utilizzo dei benchmark

12 LISTE

In ROOT esiste un oggetto simile alle liste della standard library: le TList. Queste non è altro che un container che può contenere oggetti eterogenei, a patto che derivino tutti da TObject (classe madre di tutti gli oggetti di ROOT). Per inizializzare una TList la sintassi è:

TList * list= new TList();

Per aggiungervi un'oggetto:

list->Add(obj);

Per accedere all'elemento i-esimo:

list->At(i);

Per stampare a schermo (testo) il contenuto della TList:

list->Print();

Infine, il metodo InheritsFrom("Tipo dell'oggetto") serve a controllare se un'oggetto è di un certo tipo, ad esempio può essere usato come condizione di un if:

```
if(!testList->At(i)->InheritsFrom("TH2"))
testList->At(i)->Draw("H");
```

13 ALBERI

Un tree è una struttura dati organizzata in "branches" che possono essere visti come cartelle all'interno dei quali si salvano dati correlati fra loro (che ad esempio è probabile vengano usati insieme), per convenzione ogni branch contiene sottostrutture dette leaves, ciascune delle quali corrisponde ad un tipo di dato.

13.1 *Creare e riempire un Tree*

Per usare un Tree innanzitutto bisogna inizializzare un file

TFile *F = new TFile("nome_file.root", RECREATE);

in seguito, oggetto di tipo Tree si inizializza con la seguente sintassi

TTree *t = new TTree("t, "titolo");

Per creare i branches si usa il metodo Branch

double x; t -> ("x", &x, "x/F");
Per riempire il tree si usail metodo Fill che riempie automaticamente il branch corretto. F è il tipo della variabile (ad esempio F sta per Float, D per Double).

t->Fill();

Infine si scrive il tree sul file e si chiude il file.

```
t->Write()
F->Close()
```

```
void makeTree()
{

  TFile *file = new TFile("testTree.root","recreate"); //opening the ROOT file
  TTree *T = new TTree("T","test tree"); //creating the Tree

  Float_t x,y,z; //local variables

  //Defining the branches
  T->Branch("x",&x,"x/F");
  T->Branch("y",&y,"y/F");
  T->Branch("z",&z,"z/F");

  for(Int_t i=0;i<100000;i++){
    x=gRandom->Gaus(3,2); //gaussian
    y=gRandom->Gaus(10,3); //gaussian
    z=gRandom->Exp(3.); //expo
    T->Fill();//filling the Tree
  }

  T->Write(); //writing the Tree on the ROOT file
  file->Close(); //closing the ROOT file

}
```


Figura 10

13.2 *Analisi dati*

Una volta inizializzato e riempito l'albero possiamo passare alla fase di analisi dei dati salvati nell'albero. Innanzitutto per accedere al tree (creato in un altro file) dal file di analisi dati, bisogna aprire il file precedente e recuperare il Tree

TFile *file = new TFile("test.root");
TTree * Tout= (TTree*)file->Get("T");
Per stampare a schermo le variabili contenute nel Tree si usa il metodo Print

Tout->Print();

Per graficare una variabile, ad esempio x (con nome binnaggio e range automatico):

Tout->Draw(“x”);

Per riempire e disegnare un istogramma user-defined con una variabile di un tree si usa l'operatore » (dove x è il nome del branch):

TH1F *h 1= new TH1F("h1","hist from tree",50,-4, 4);
Tout->Draw("x»h1");

É inoltre possibile imporre condizioni sulle variabili con un'opzione di Draw (che accetta operatori logici di C++)

Tout->Draw("x”,”y>0 && x<10”);

Si possono disegnare anche grafici bi dimensionali e tridimensionali che correlano set di dati contenuti in diversi branches:

T->Draw(“y:x”);
T->Draw(“z:y:x”);

```
TCanvas * CreateC(Int_t i)
{
   TString cName="c";
   TString cTitle="Tree examples ";
   TCanvas *c;
   c=new TCanvas(cName+i,cTitle+i,200,10,600,400);
   return c;
}


void readTree()
{

 // TH1::AddDirectory(kFALSE);

  TCanvas *c[10];

  TFile *file = new TFile("testTree.root"); //opening the root file
  file->ls(); // listing the file content

  TTree * Tout= (TTree*)file->Get("T"); //getting the Tree

  Tout->Print(); //listing the Tree content

  Int_t nentries=Tout->GetEntries(); // number of entries in the Tree
  cout << " nentries in tree = " << nentries << endl;


   //Drawing the Tree variables (automatic loop on Tree entries done by ROOT)

  int i=0;

  CreateC(i)->cd();  i++;
  Tout->Draw("x");//Drawing x looping over all entries, on a temporary histo

  CreateC(i)->cd();  i++;
  Tout->Draw("x","","",1000,nentries-1000);//same as above, but for the last 1000 entries

  CreateC(i)->cd();  i++;
  Tout->Draw("y:x");//Drawing y vs x, looping over all entries, on a temporary histo

  TH1F *h1=new TH1F("h1","x distribution",200, -10., 10.);
  CreateC(i)->cd();  i++;
  Tout->Draw("x>>h1","y>10.");//Drawing x over a predefined histo h1 and apply a selection on
y.

  CreateC(i)->cd();  i++;
  Tout->Draw("sqrt(x**2+y**2)","log(y)>1.");//you may use functions

  CreateC(i)->cd();  i++;
  Tout->Draw("sqrt(x**2+y**2):log(z)","y>1 && x<0");//correlation plot with selection

  //in this mode, the user loops explicitly over the tree and can recover/select each of the
entries
```


Figura 11