ROOT

- I tools di analisi
- Cos'è ROOT
- Istogrammi
- N-tuples in ROOT: Trees
- Estensioni e librerie dinamiche
- Esempi

I “tools di analisi”

I “tools di analisi” sono strumenti che permettono di svolgere in modo semplice ed efficace le principali operazioni necessarie nell’analisi dei dati:

- Costruzione e rappresentazione grafica di istogrammi
- Fit degli istogrammi
- Costruzione di N-tuple
- Scrittura e lettura di istogrammi e N-tuple
- ...

Il primo strumento di questo tipo, ormai obsoleto ma ancora molto usato, si chiama PAW ed è basato sul linguaggio FORTRAN; viene solitamente utilizzato insieme ad una libreria chiamata HBOOK.

Libreria HBOOK

La libreria HBOOK permette di creare, riempire e maneggiare istogrammi e N-tuple, che possono essere poi analizzati interattivamente.

http://wwwasdoc.web.cern.ch/wwwasdoc/hbook_html3/hboomain.html

http://wwwasdoc.web.cern.ch/wwwasdoc/pdfdir/hbook.pdf.gz

I dati relativi ad istogrammi e N-tuple vengono conservati in un unico array di dimensioni sufficienti; l'accesso ad essi avviene attraverso la chiamata di opportune funzioni e subroutines.

All'inizio dell'esecuzione deve quindi venire definito lo spazio disponibile, posto in un blocco COMMON di nome PAWC .

```
      parameter(nhbook=50000)
      common/pawc/hhh(nhbook)
C
      call hlimit(nhbook)
```

Istogrammi

La prima operazione da eseguire con un istogramma è, ovviamente, la sua creazione. Esistono istogrammi ad una dimensione e a due.

```
      call hbook1(id1,cht,nbc,xmin,xmax,vmax)
      call hbook2(id2,cht,nbx,xmin,xmax,
     1                    nby,ymin,ymax,vmax)
```

Gli istogrammi sono identificati da un numero intero ed hanno un titolo. Devono poi essere specificati il numero di bin e gli estremi; il valore massimo è invece opzionale.

Un istogramma viene riempito mediante una chiamata alla subroutine HFILL:

```
      call hfill(i,x,y,w)
```

Al contenuto del canale corrispondente a x,y (y viene ovviamente ignorato per gli istogrammi ad una dimensione) viene sommato il "peso" w . Istogrammi con bin non uniformi possono venire creati definendo un array XBIN con gli estremi di tutti i canali:

```
      call hbookb(id1,cht,nbc,xbin,vmax)
```

PAW

Con la libreria HBOOK istogrammi e N-tuple vengono creati, riempiti, scritti su file e letti; ciò avviene però in modo poco flessibile. È quindi molto utile uno strumento che permette di svolgere le stesse operazioni in modo interattivo: tale strumento si chiama "PAW", acronimo per Physics Analysis Workstation.

Informazioni disponibili su web: http://paw.web.cern.ch/paw/

L'operazione piú semplice con PAW è la visualizzazione grafica di un istogramma letto da un file:

```
> paw
...
PAW > hi/file 1 file
PAW > hi/plot 11
PAW > ...
PAW > quit
```

Cos'è ROOT

ROOT è un tool di analisi analogo a PAW, ma basato sul linguaggio C++ anziché sul FORTRAN.

Informazioni disponibili su web: http://root.cern.ch

La differenza piú significativa tra i due strumenti è data dall'interprete dei comandi:

- PAW: linguaggio "privato"
- ROOT: puro C++ (CINT)
  - ⇝ un solo linguaggio
  - ⇝ grande flessibilità tra programmi compilati e analisi interattive

Altre differenze si possono trovare nella grafica, di altissima qualità in ROOT, e nella possibilità di immagazzinare su file oggetti di varia natura e non solo di classi predefinite (istogrammi e trees).

Per utilizzare ROOT è usualmente necessario definire una variabile d'ambiente ROOTSYS con la top-directory di ROOT; l'eseguibile si trova in $ROOTSYS/bin/root .

Istogrammi

In ROOT gli istogrammi vengono gestiti mediante oggetti TH1F o TH1D per variabili in singola o doppia precisione, rispettivamente. Essi sono identificati da un nome (anziché un numero):

```
root [0] TH1F h1("name","title",10,0.0,10.0);
root [1] h1.Fill(3.4);
root [2] h1.Draw();
```

La chiamata a Draw provoca l'apertura automatica di una finestra grafica, corrispondente ad un oggetto TCanvas; ovviamente è possibile creare una o piú finestre di dimensioni specificate. La finestra grafica non contiene solo un disegno, ma un riferimento all'oggetto TH1F, e viene quindi aggiornata quando si presentano nuovi eventi.

Piú comandi possono essere raggruppati in un file, eseguito con il comando .x . Lo stesso codice può essere, se opportuno, inserito in un programma compilato: è necessario allora caricare le librerie in $ROOTSYS/lib e definire corrispondentemente la variabile d'ambiente LD_LIBRARY_PATH

File Macro

```
{
TH1F h1("name","title",10,0.0,10.0);
h1.Fill(3.4);
TCanvas cc("cname","ctitle",20,50,600,400);
TPad p1("p1","p1t",0.02,0.02,0.49,0.98);
TPad p2("p2","p2t",0.51,0.02,0.98,0.98);
p1.Draw();
p2.Draw();
p1.cd();
h1.Draw();
p2.cd();
h1.DrawCopy();
h1.Fill(5.6);
}
```

Esecuzione interattiva o compilata

Il codice scritto in un file .mac viene eseguito interattivamente con il comando .x seguito dal nome del file:

```
> root
root [] .x histotest.mac;
```

Per eseguire le stesse istruzioni in un programma “standalone” è necessario includere i file .h necessari, disponibili in $ROOTSYS/include :

```
#include "TH1.h"
#include "TCanvas.h"
#include "TPad.h"
int main() {
   ...
}
```

```
> c++ -I $ROOTSYS/include -L $ROOTSYS/lib -o prog \
      prog.cc -l...  -l...
```

Programmi con funzioni grafiche

Se nel codice sono presenti chiamate a funzioni grafiche è necessario infine creare, all'inizio del programma, un oggetto TApplication ed eseguirne, al termine, la funzione Run:

```
#include "TApplication.h"
int main( int argc, char* argv[] ) {
  new TROOT("troot","Troot");
  TApplication app("App",&argc,argv);
  ...
  app.Run(kTRUE);
  ...
}
```

L'esecuzione si arresta all'interno della funzione Run, e ne esce con il comando "Quit ROOT" del menu "File" di un Canvas aperto. Tale comando provoca il termine immediato del programma se la funzione viene chiamata con argomento kFALSE .

I/O su file

Gli istogrammi, creati e riempiti, possono venire scritti su file e riletti successivamente:

```
> root
root [] TH1F* his = new TH1F("name","title",10,0.0,10.0);
root [] ...
root [] TFile file("filename","CREATE");
root [] his->Write();
root [] .q
```

```
> root
root [] TFile file("filename");
root [] TH1F* his = dynamic_cast<TH1F*>(file.Get("name"));
root [] his->Draw();
```

È possibile scrivere su file tutti gli oggetti con una sola chiamata file.Write() , in tal caso è necessario aprire il file prima di aver creato gli oggetti.

Accesso globale

Da un istogramma è possibile estrarre gli estremi, il numero di bin e il loro contenuto; il conteggio dei bin parte da 1 mentre 0 estrae il numero di underflow. È possibile ottenere il pointer all'array (che non è una copia).

```
root [] int nbin = h1->GetNbinsX();
root [] int nsize = h1->GetSize();
root [] float* cont = h1->GetArray();
root [] float xb = h1->GetBinContent(12);
```

Viceversa è anche possibile impostare il contenuto dei bin:

```
root [] float* arr = new float[nsize];
root [] h1->Set(nsize,arr);
root [] h1->SetBinContent(12,xb);
```

Gli istogrammi possono essere sommati o moltiplicati tra loro, e moltiplicati per una costante:

```
root [] TH1F hsum = (a*h1)+(b*h2);
root [] TH1F hprod = h1*h2;
```

Grafici

Un grafico può essere disegnato per punti costruendo due arrays contenti le coordinate x e y , ed eventualmente altre coppie di arrays con i corrispondenti errori:

```
root [] int n = 10;
root [] float* x = new float[n];
root [] float* y = new float[n];
root [] TGraph g1(n,x,y);
root [] g1.Draw("AP");
...
root [] TGraphErrors ge(n,x,y,ex,ey);
root [] TGraphAsymmErrors ga(n,x,y,xl,xh,yl,yh);
```

Funzioni analitiche

È possibile disegnare funzioni utilizzando oggetti TF1; per le funzioni piú semplici l'espressione analitica può essere data come stringa di caratteri:

```
> root
root [] TF1 pf1("simple","sin(x)/exp(0.05*x)",0.0,30.0);
root [] pf1.Draw();
root [] pf1.SetNpx(10);
root [] pf1.Draw();
```

Per disegnare funzioni piú complesse e/o con parametri variabili queste possono essere codificate in una funzione C++ , associata all'oggetto TF1.

Funzioni codificate

Le funzioni da disegnare hanno due variabili in ingresso, un puntatore all'argomento ed un puntatore ad un array contenente i parametri:

```
double fun( double* arg, double *par ) {
  return sin( *arg * par[0] ) / exp( *arg * par[1] );
}
  ...
  TF1* plot = new TF1( "pfun", fun, 0.0, 20.0, 2 );
  double par[2];
  par[0] = 1.0;
  par[1] = 0.1;
  plot->SetParNames( "Frequency", "Damp" );
  plot->SetParameters( par );
  plot->Draw();
  ...
```

Fit di istogrammi

Un oggetto TF1, indentificato dal puntatore o dal nome, può essere usato per eseguire il fit di un istogramma; le funzioni ”polN”,”expo”,”gaus” sono predefinite.

```
root [] TFile file("his.root");
root [] TH1F* h = dynamic_cast<TH1F*>(file.Get("hs"));
root [] double par[6];
root [] TF1* g1 = new TF1("gaus1","gaus",-4,-1);
root [] TF1* g2 = new TF1("gaus2","gaus", 0, 4);
root [] TF1* gs = new TF1("total","gaus(0)+gaus(3)",-1,1);
root [] h->Fit(g1,"R");
root [] g1->GetParameters(par);
root [] h->Fit(g2,"R");
root [] g2->GetParameters(par+3);
root [] gs->SetParameters(par);
root [] h->Fit("total","");
```

Trees

In ROOT gli oggetti equivalenti alle N-tuple si chiamano “Tree” e, nel caso piú semplice, hanno la stessa struttura.

Come per le N-tuple i dati vengono letti da una locazione di memoria fissa, è quindi necessario costruire una struttura (o una classe) in cui disporli prima della scrittura sul Tree:

```
struct evData {
  int eventId;
  int nData;
  int   iData[10];
  float xData[10];
  float yData[10];
} event;
```

Esistono anche oggetti TNtuple, che di fatto sono Trees contenenti solo variabili float.

Branches

L'associazione tra le variabili in memoria e nel Tree viene eseguita mediante chiamate alla funzione Branch:

```
TFile file( "simpletree.root", "CREATE" );
TTree* tree = new TTree( "tsimple", "simple data" );
tree->Branch( "head", &event.eventId, "idEvt/I:nData/I" );
tree->Branch( "iData", event.iData, "iData[nData]/I" );
tree->Branch( "xData", event.xData, "xData[nData]/F" );
tree->Branch( "yData", event.yData, "yData[nData]/F" );
...
tree->Fill();
...
tree->Write();
```

È possibile scrivere, come nell'esempio, arrays di dimensione variabile, ma non insieme ad altre variabili nello stesso branch.

Analisi interattiva

Le variabili contenute in un Tree possono essere istogrammate, o scritte sullo schermo, in modo interattivo:

```
root [] TFile file("simpletree.root");
root [] TTree* tree = file.Get("tsimple");
root [] tree->Draw("xData[1]");
root [] tree->Draw("yData:xData");
root [] tree->Draw("iData","xData<3.4");
root [] tree->Draw("xData:yData","iData<6","lego");
root [] tree->Draw("xData:yData","","lego");
root [] tree->Scan("idEvt:iData[1]:xData[1]",
.......  > "yData[1]>2.0&&nData==4");
```

Per analisi piú complesse è necessario ricostruire la struttura completa usata in scrittura; il codice può essere generato automaticamente con MakeClass() .

Lettura di Trees

```
TFile file( "simpletree.root" );
TTree* tree = file.Get( "tsimple" );
evData event;
tree->SetBranchAddress( "head", &event.eventId );
tree->SetBranchAddress( "iData",  event.iData );
tree->SetBranchAddress( "xData",  event.xData );
tree->SetBranchAddress( "yData",  event.yData );
int entnum = tree->GetEntries();
int ientry = 10;
tree->GetEntry( ientry );
...
TBranch* b_head = tree->GetBranch( "head" );
b_head->GetEntry( ientry = 34 );
...
```

Oggetti complessi

Oggetti piú complessi possono essere creati senza particolari difficoltà: data una classe Point definita nei file Point.h e Point.cc per creare oggetti di tale classe è sufficiente caricare il file contenente la definizione:

```
> root
root [] .L Point.cc
root [] Point p1(2.,3.);
root [] Point p2(5.,7.);
root [] float d = p1.dist(p2);
root [] cout << d << " " << p1.dist(3.6,1.1) << endl;
```

Per scrivere oggetti complessi su file è necessario (nelle prime versioni) che tali oggetti ereditino da un oggetto base TObject.

Uso della base TObject

Affinché ROOT possa eseguire tutte le operazioni con gli oggetti complessi essi devono ereditare dalla base comune TObject; inoltre nella definizione della classe deve essere inclusa la macro ClassDef:

```
#include "TObject.h"
class RootPoint :  public TObject {
  ClassDef(RootPoint,1)
  ...
};
```

Nel file .cc deve essere poi inclusa la macro ClassImp:

```
#include "RootPoint.h"
ClassImp(RootPoint)
...
```

Dizionario

Il codice che permette di eseguire le operazioni di I/O sugli oggetti è creato automaticamente dal programma rootcint (in $ROTSYS/bin), e deve essere anch'esso compilato:

```
> rootcint DictPoint.cc -c RootPoint.h
> c++ -I $ROOTSYS/include -c RootPoint.cc
> c++ -I $ROOTSYS/include -c DictPoint.cc
> c++ -shared -o libRootPoint.so RootPoint.o DictPoint.o
> root
root [] .L libRootPoint.so
root [] RootPoint p1(2.,3.);
root [] RootPoint p2(5.,7.);
root [] TFile file("pts.root","CREATE");
root [] p1.Write("punto1");
root [] p2.Write("punto2");
```

Estensioni

La libreria condivisa può essere caricata anche con una istruzione C++ :

```
gSystem->Load("libRootPoint.so")
```

Infine, è possibile creare un eseguibile che contiene già le librerie richieste; è sufficiente creare un piccolo programma rootpt.cc :

```
#include "TRint.h"
int main( int argc, char* argv[] ) {
  TRint app("PointApp",&argc,argv);
  app.Run(kTRUE);
  return 0;
}
```

da compilare come qualsiasi altro, includendo in piú le librerie desiderate:

```
> c++ -I $ROOTSYS/include -L $ROOTSYS/lib -L. \
      -o exroot exroot.cc -lRootPoint -l...  -l...
```