# Lect2

December 13, 2019

## 1 Istogrammi in ROOT, gestione file e macro

In questo notebook andremo a vedere come è possibile gestire i file con estensione .root, comunemente infatti il framework ROOT viene usato per analizzare gli elementi derivate dall'acquisizione di dati, salvati in formato .root appunto; andremo a vedere l'ultima classe fondamentale dell'analisi dati con ROOT: TH1, che gestisce gli istrogrammi.

Parleremo infine brevemente di come si possono creare e usare le macro in ROOT e dei tipi di file correlati al framework.

**!!! importante**

Nel file seguente alcune istruzioni (import ROOT e %%cpp) sono usate solo e soltanto per far funzionare il codice sul notebook, non vanno usate nel framework!

[1]:
```
import ROOT
#al solito questo serve solo per importare root nel notebook non serve␣
 ↪scriverlo nel framework
```

```
Welcome to JupyROOT 6.14/06
```

### 1.1 TFile

I TFile sono una classe Root che si occupa di aprire, modificare e spostarsi nei file .root all'interno del framework.

Una volta in possesso di un file .root che si vuole aprire per farvi analisi dati (ad esempio un file contenente delle misurazioni salvate come istrogramma) vi sono due modi per aprirlo nel framework:

1. Il modo più semplice per aprire un file all'interno del framework è direttamente dal terminale al momento dell'apertura di root dare il nome del file, ad esempio l'istruzione
    - root -l 100mSimK12_trigger.root

Aprirà il file 100mSimK12_trigger.root all'interno del framework e il suo contenuto (molti istogrammi) diventerà direttamente utilizzabile, (nel fare ciò viene aperto automaticamente un TFile a cui viene assegnato il file)

2. E' possibile aprire il file "manualmente" quando si è già all'interno di root tramite appunto l'uso dei TFile.

Il costruttore di un TFile (al parte quello di default) è:

TFile::TFile ( const char * fname1, Option_t * option = "", const char * ftitle ="", Int_t compress = ROOT::RCompressionSetting::EDefaults::kUseGeneralPurpose )

Come potete vedere l'unico paramentro "obbligatorio" è il nome del file, gli altri paramentri sono opzionali. E' importante ricordare che quando si apre un file spesso è importante dare al costruttore oltre al nome un'opzione che identifica l'uso che si farà del file: - CREATE crea un nuovo file se non ne esistono con lo stesso nome - RECREATE crea un nuovo file e sovrascrive quello vecchio se hanno lo stesso nome - UPDATE apre un file con possibilità di modificarlo - READ apre un file in sola lettura

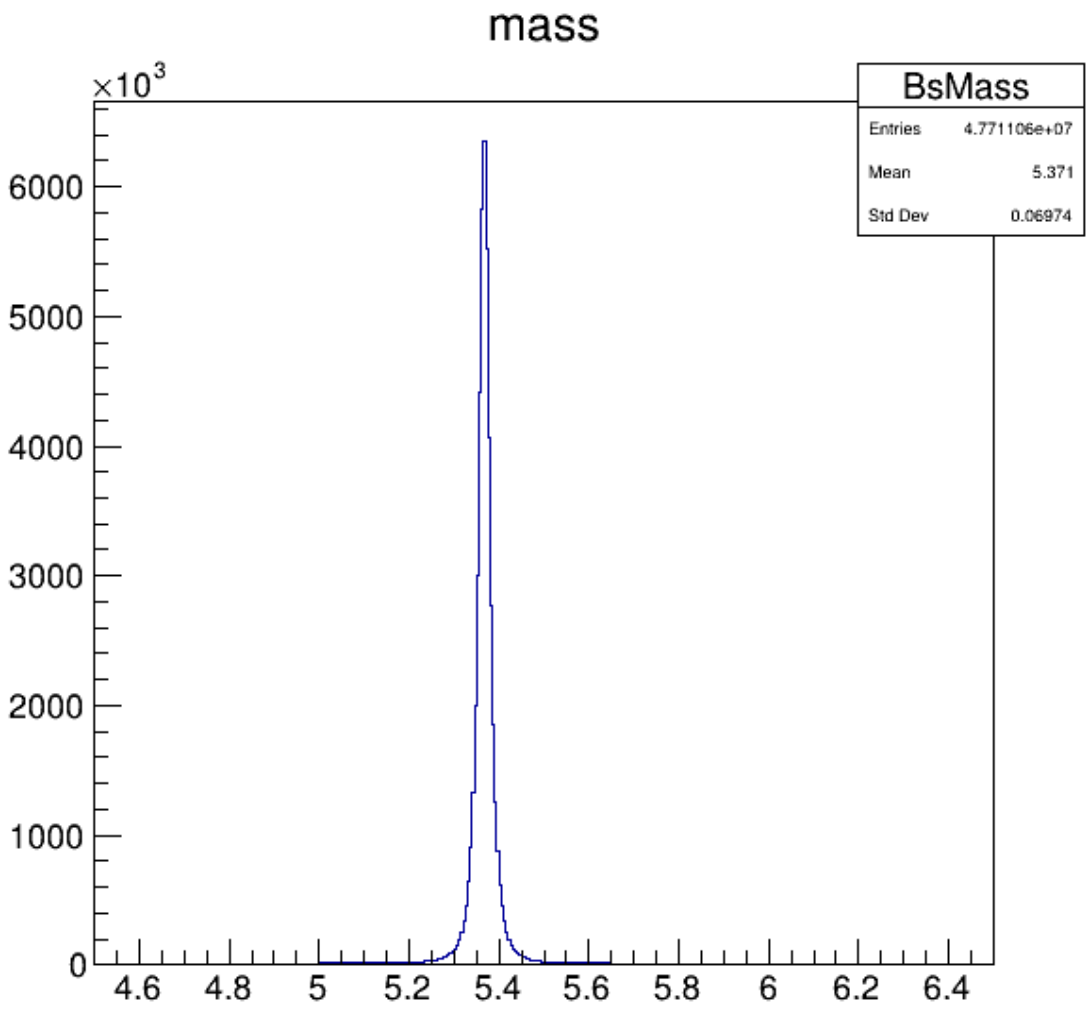
Nel notebook è possibile usare solo questa seconda opzione, ad esempio proviamo ad aprire il file 100mSimK12_trigger:

[2]:
```
%%cpp
TFile *file = new TFile ("100mSimK12_trigger.root"); //creo un Tfile con cui␣
 ↪gestire 100mSimK12_trigger
file->cd(); // mi sposto nel file tramite il TFile
```

Quando siete nel framework per vedere il contenuto di un file è sufficiente mandare l'istruzione .ls e a scehrmo verranno scritti tutti gli oggetti contenuti (sul notebook non funziona perchè non è cpp nativo).

Una volta aperto è possibile usare tutti gli elementi all'interno, per esempio nel file vi è un istogramma chiamato BsMass (vedremo poi gli istogrammi nel dettaglio) e quindi possiamo direttamente stamparlo

[3]:
```
%%cpp
TCanvas *c= new TCanvas("c", "c", 600, 600);
c->cd();
BsMass->Draw();
c->Draw()
```




E' estremamente importante sapere che una volta che un file viene aperto e modificato (per esempio quando create funzioni, fate fit eccetera) le modifiche NON vengono mai salvate automaticamente, questo è utile visto che permette di modificare "in modo sicuro" gli oggetti senza modificarli in modo irreversibile.

Per salvare ciò che avete prodotto e/o modificato è necessario chiamare il metodo Write che scrive nel file corrente gli elementi che avete creato.

Il modo con cui viene gestita la memoria è estremamente complicato quindi non vedremo nel dettaglio come funziona, in generale è sufficiente una volta che si è nel file chiamare il metodo write reimplementato per un particolare oggetto; ad esempio:

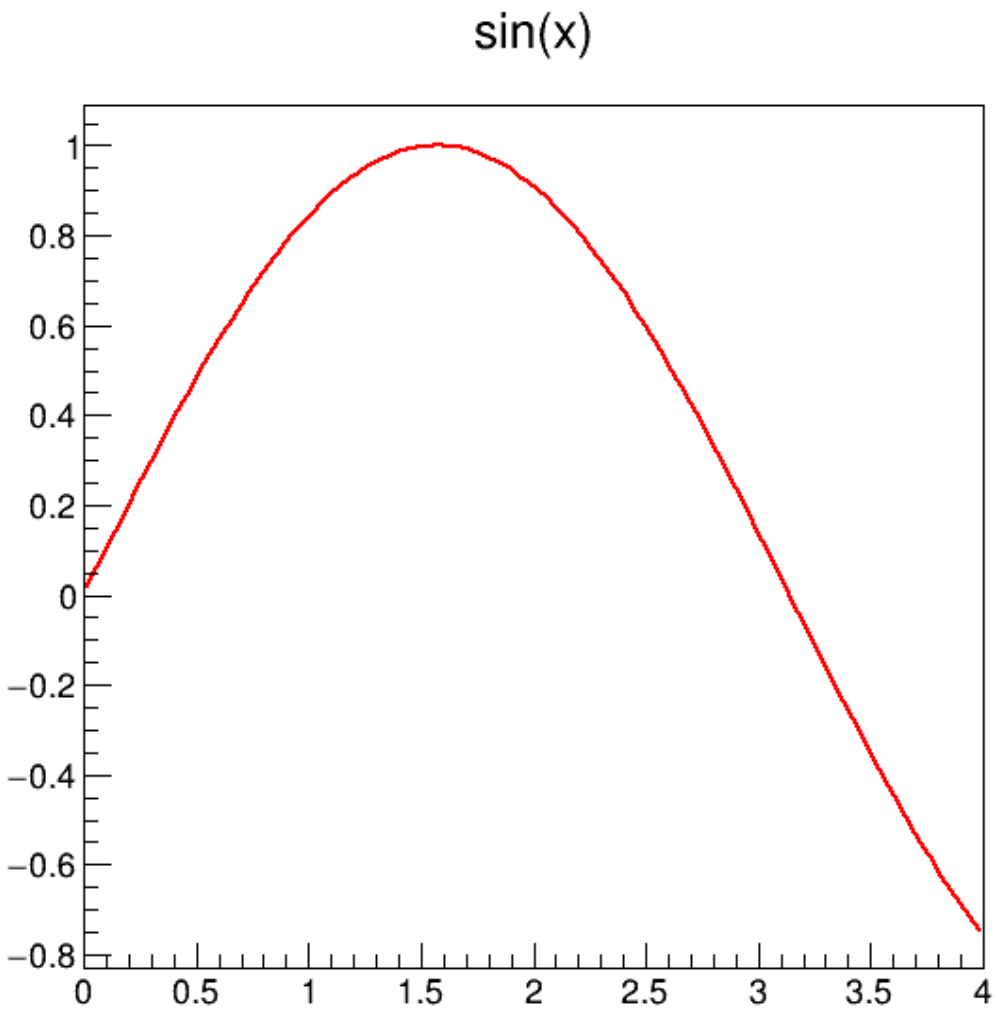
[4]:
```
%%cpp
TFile *file2= new TFile("file.root", "RECREATE"); //creo un nuovo file ( lo␣
↪sovrascrivo se gia esiste)
```

```
//state molto attenti a CREATE/RECREATE perche la sovrascrittura e irreversibile
file2->cd(); //mi sposto nel file
TF1 *f1= new TF1("f1", "sin(x)", 0,4); //creo una funzione
```

Se a questo punto scrivete .ls per vedere gli elementi nel file non vedrete nulla (non la TF1 almeno) questo perchè non l'avete ancora salvata:

[5]:
```
%%cpp
f1->Write(); //scrivo la funzione nel file
file2->Close(); //chiudo il file cosi da vedere se si e salvato davvero
```

[6]:
```
%%cpp
TFile *file3= new TFile("file.root", "READ"); //apro il file di prima in␣
 ↪modalita lettura
file3->cd(); //mi sposto nel file
TCanvas *c2=new TCanvas("c2", "c2", 600, 600);
c2->cd();
f1->Draw(); //disegno la funzione che prima ho salvato
c2->Draw();
```




Se poi volessi riaprire il file per modificarlo lo dovrò aprire in modalità “UPDATE”

### 1.2 TH1

I TH1 sono una delle classi più importanti di ROOT. I TH1 si occupano della gestione degli istogrammi 1-dimensionali e vengono reimplementati in sostanzialmente tre opzioni, che sono le classi concrete che useremo: - TH1F istogramma dove i dati ricevuti sono float - TH1D istogramma dove i dati ricevuti sono double - TH1I istogramma dove i dati ricevuti sono interi

Nei seguenti esempi useremo sostanzialmente solo i TH1F ma tutto ciò che diremo vale anche per gli altri.

**Nota** Esistono le rispettive classi TH2F, TH2D, TH2I che gestiscono gli istogrammi bidimensionali (poco usati)

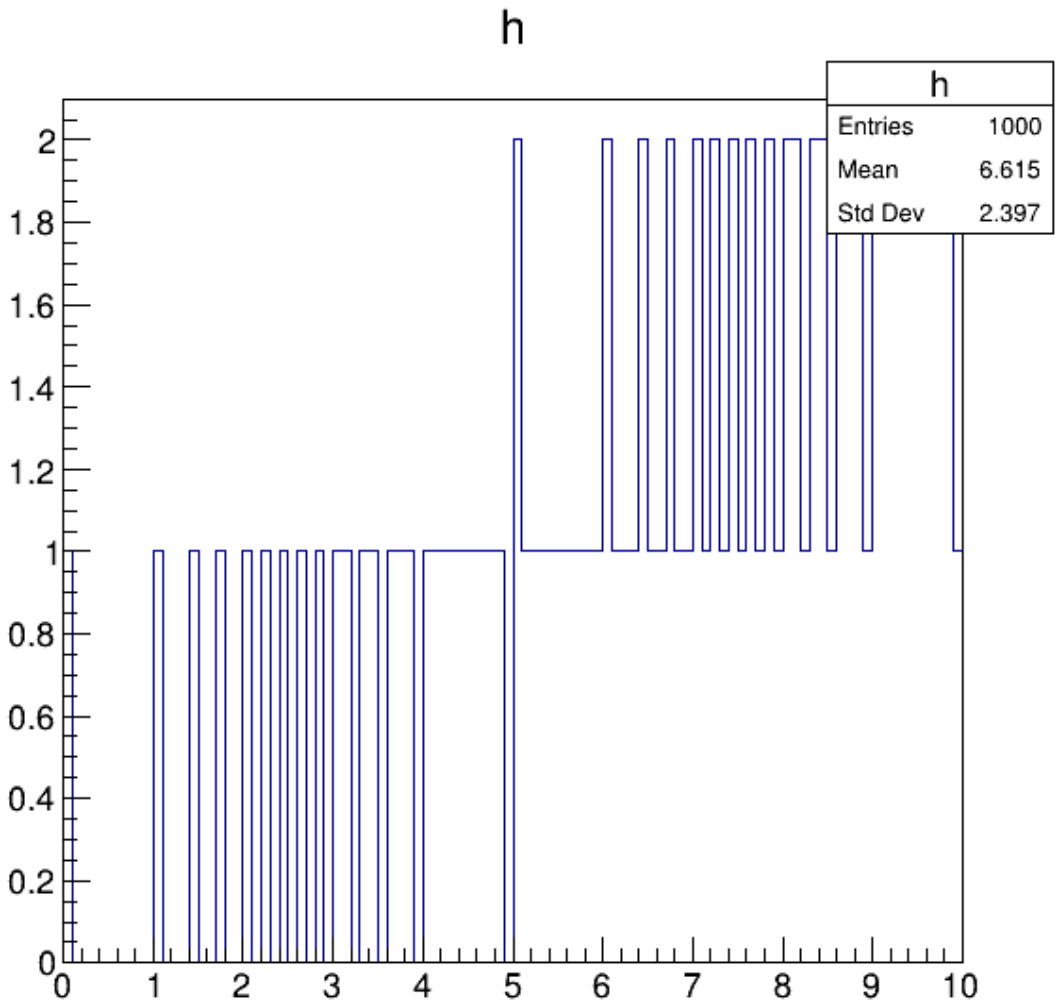
Come potete vedere dalla documentazione i TH1 hanno una quantità di metodi enorme, in ogni caso il costruttore più usato (si usa sempre questo a meno che non vogliate fare istogrammi a binning variabile) è:

```
TH1 (const char *name, const char *title, Int_t nbinsx, Double_t xlow, Double_t xup)
```

Che chiede nome, titolo, numero di bin, e range (min e max). Per riempire l'istrogramma esiste poi un metodo chiamato Fill che chiede in input un dato oppure un dato e il suo peso, mentre per disegnarlo esiste una reimplementazione del metodo Draw.

Facciamo un esempio:

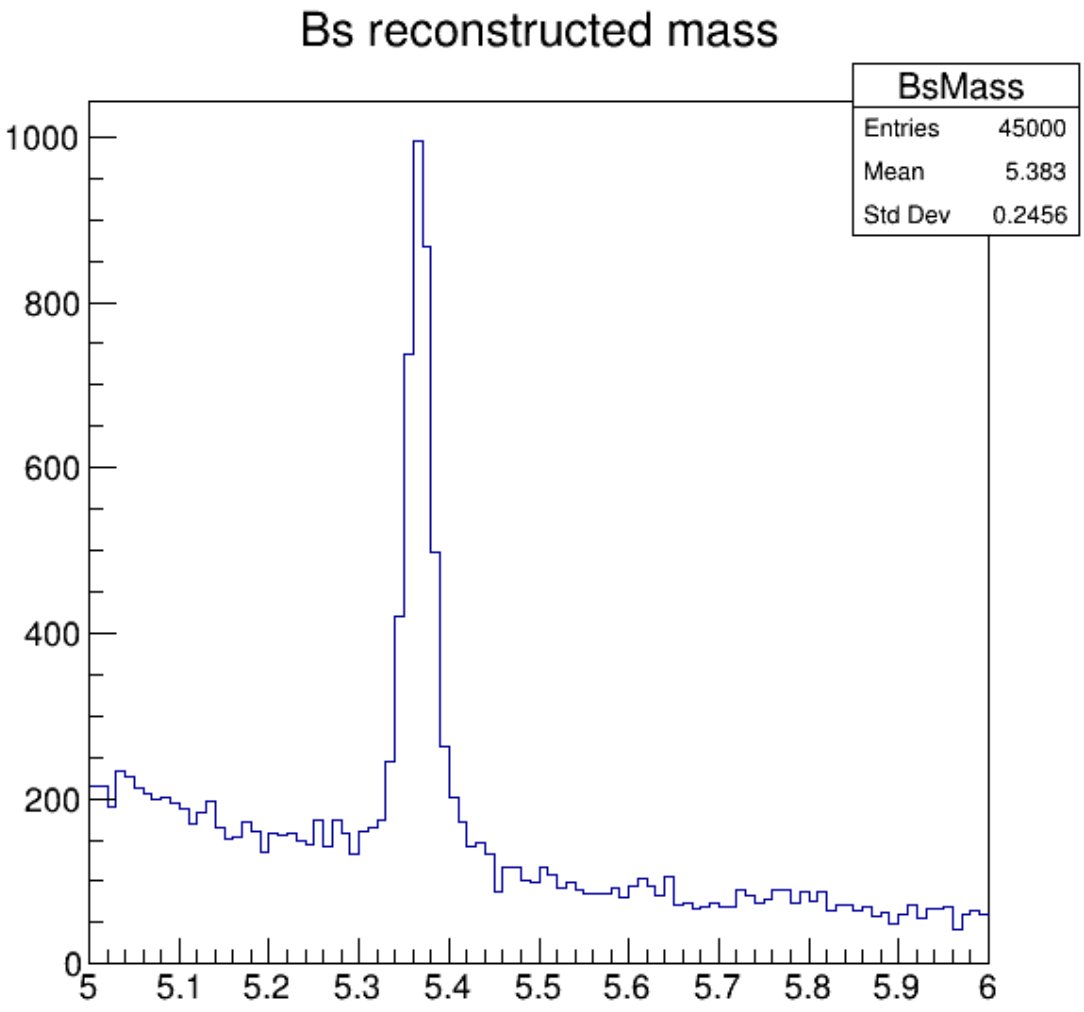
[7]:
```
%%cpp

TH1F *h=new TH1F("h","h", 100, 0,10); //creo un istrogramma con dati da 0 a 10␣
 ↪e 100 bin
//faccio un ciclo con cui creare un po di dati da inserire nell istrogramma
for (int i=0; i<1000; i++){
    double t=sqrt(i);
    h->Fill(t);
}
TCanvas *c3=new TCanvas("c3", "c3", 600, 600);
c3->cd();
h->Draw(); //disegno l istogramma
c3->Draw();
```




Questo istogramma è particolarmente brutto (i dati sono generati un po' a caso) quindi proviamo a fare qualcosa di più interessante: nel file LD30kAngle.txt ci sono tre colonne che rappresentano massa ricostruita (con valore tra 5 e 6), distanza e suo errore nel decadimento del $B_s^0$, proviamo a creare l'istogramma della massa ricostruita:

[8]:
```
%%cpp
//creo un tfile in cui salvare i dati poi
TFile *Bs=new TFile("BsMass.root", "RECREATE");
//ora devo caricare i dati per farlo uso un ifstream
ifstream in ("LD30kAngle.txt");
//creo un TH1F in cui caricare i dati
TH1F *BsMass = new TH1F ("BsMass", "Bs reconstructed mass", 100, 5, 6); // 100␣
 ↪bin tra 5 e 6
//ora faccio un ciclo con cui caricare i dati
```

```
double m,d,e;
while (in>>m){
    in>>d,e; //bisogna ricevere una riga intera anche se poi d ed e non li uso
    BsMass->Fill(m); //carico la massa nell istogramma
}
BsMass->Write();// salvo il Th1f nel file

//ora proviamo a disegnarlo
TCanvas *cbs= new TCanvas("cbs", "Bs Mass", 600, 600);
cbs->cd();
BsMass->Draw();
cbs->Draw();
```




In questo modo avendo un qualsiasi file txt contenente dei dati su cui si vuole costruire un istro-

gramma possiamo caricarlo in un TH1F.

### 1.2.1 Lavorare sui TH1

La classe TH1 permette di fare moltissime cose con gli istogrammi dal semplice fit alla pulizia del fondo (se si hanno le misure del fondo ovviamente).

I metodi implementati gestiscono tutte queste operazioni: nel file 100mSimK12_trigger sono presenti molti TH1F con cui possiamo provare a lavorare.

**Nota** Nei codici qui sono non andrò a salvare gli istogrammi modificati o creati per non sovrascrivere/modificare il file, in ogni caso ricordate che qualsiasi cosa creata/modificata viene salvata con Write()

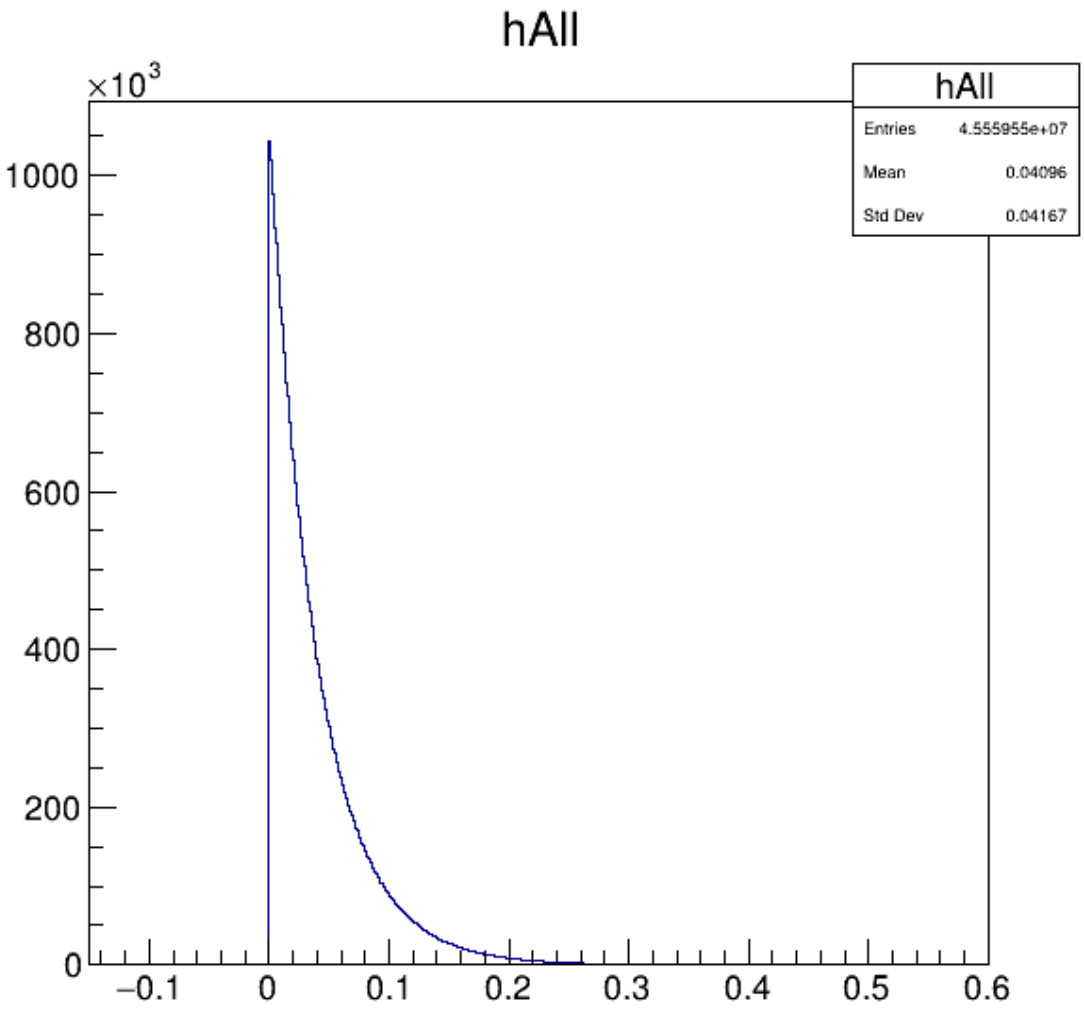
**Nota 2** Tutti quello che c'è nel file si può già subito usare senza dover usare costruttori o fare operazioni

[9]:
```
%%cpp
//apriamo il file
TFile *hfile = new TFile ("100mSimK12_trigger.root"); //creo un Tfile con cui␣
 ↪gestire 100mSimK12_trigger
hfile->cd()
```

```
(bool) true
```

Proviamo per esempio a plottare l'istrogramma hAll e a fittarlo con una funzione:

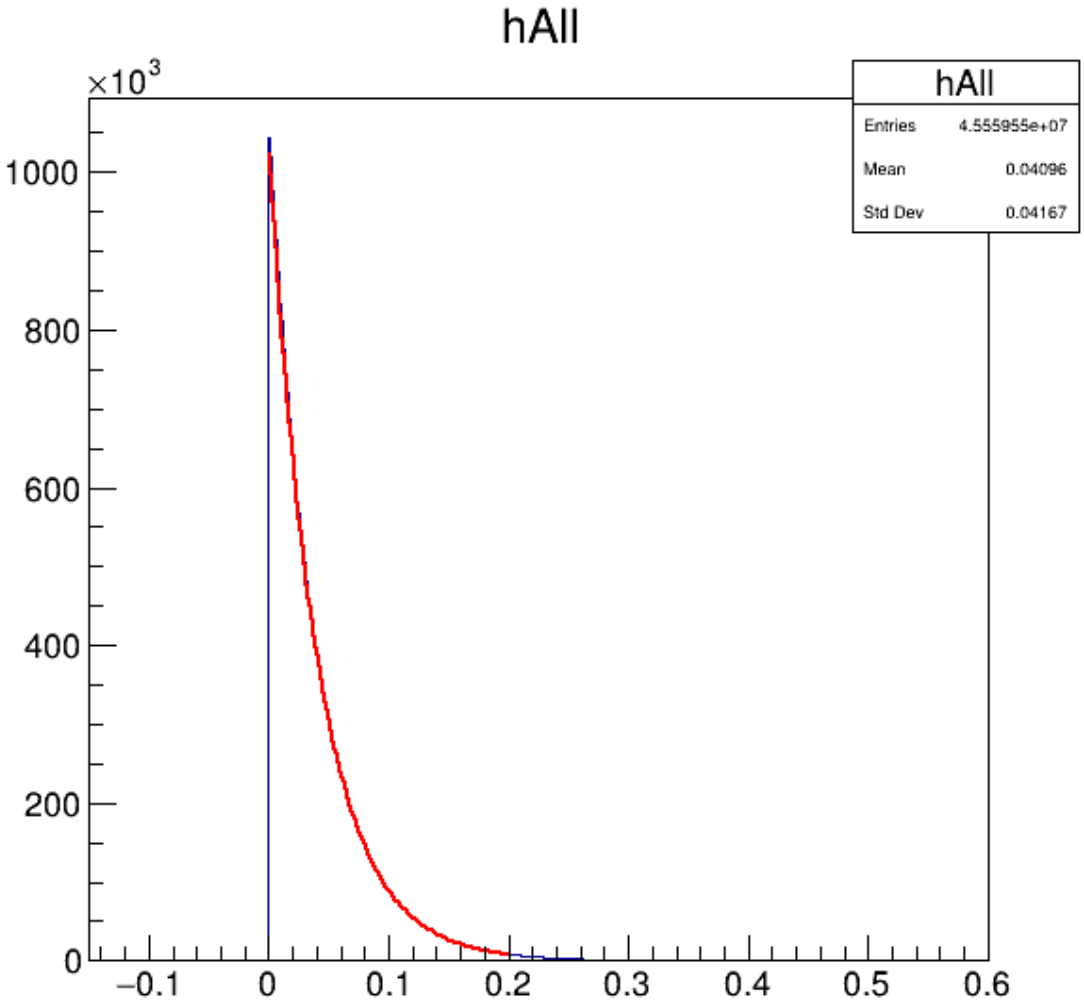
[10]:
```
%%cpp
//creo un canvas in cui plottare
TCanvas *call= new TCanvas("call", "call", 600, 600);
call->cd();
hAll->Draw(); //disegno l'istogramma
call->Draw();
```




[11]:
```
%%cpp
//proviamo a fittarlo con un esponenziale
//creo la funzione esponenziale
TF1 *fexp= new TF1("fexp", "expo", 0, 0.2);
//fitto la funzione al Th1
hAll->Fit("fexp", "R");
call->Draw(); //riplotto il canvas

//e ora facciamoci dire chi quadro e NDF
cout<<"chi2 :"<<fexp->GetChisquare()<<endl;
cout<<"NDF  :"<<fexp->GetNDF()<<endl;
```

```
 FCN=7795.02 FROM MIGRAD    STATUS=CONVERGED      49 CALLS          50 TOTAL
                     EDM=8.38171e-10    STRATEGY= 1      ERROR MATRIX ACCURATE
```

```
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  Constant     1.38627e+01   2.19426e-04   6.61024e-06   2.75144e-01
   2  Slope       -2.45760e+01   4.11317e-03   1.20256e-04   1.08292e-02
chi2 :7795.02
NDF  :211
```




Supponiamo ora per esempio di avere un istogramma (ad esempio quello del Bs) dove è presente un fondo e di averne poi uno rappresentante il suo fondo.

E' possibile ottenere quindi l'istogramma "ripulito" sottraendo (o meglio sommando negativamente) i due istogrammi. Per fare questo esiste il metodo **Add(TH1, double)** (in tre implementazioni ma a noi interessa una solo) che richiede in input un secondo istogramma e il peso con cui eseguire la somma.

Altri metodi utili per l'analisi (o per estrarre dati) sono i metodi - GetBinContent(int) - SetBin-

Content(int, double) - GetNbinsX(int)

Che rispettivamente settano e restituiscono il valore dell'i-esimo bin dell'istrogramma in analisi e restituiscono il numero totale di Bins

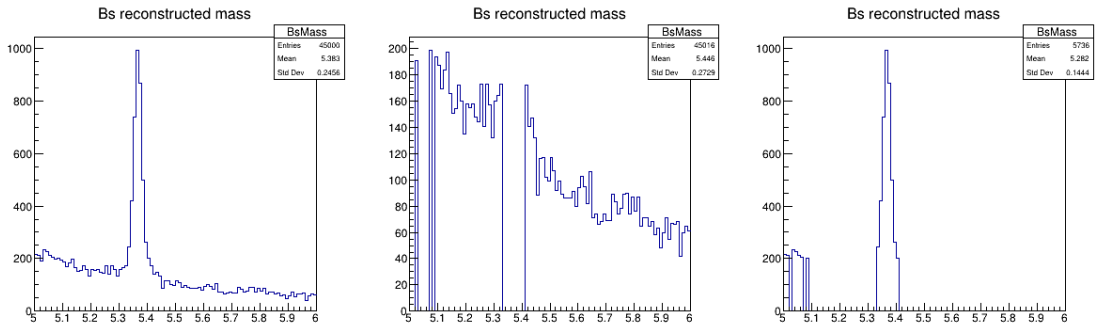
Proviamo a fare un esempio: nel grafico di prima (il Bs) è chiaramente visibile che il fondo è sempre inferiore a 200 (intendendo il valore dei bin di fondo), proviamo allora a fare questa cosa: in un canvas diviso in tre plotteremo l'istogramma, una sua copia in cui i bin con contenuto maggiore di 200 vengono azzerati (con i metodi di prima) e la differenza dei due.

Per copiare un istogramma, in particolare se non si conosce bene il binning o i dati da cui si è partiti è possibile usare i metodi sopra ma risulta molto più semplice usare il metodo **Clone** il quale appunto clona un TH1.

[12]:
```
%%cpp
TCanvas *can5= new TCanvas("can5", "can5", 1200, 400); //creo il canvas e lo␣
 ↪divido
can5->Divide(3,1); //divido il canvas

//facciamo le copie degli istogrammi, bisogna essere un po accorti mentre si␣
 ↪chiama clone
//chiamare clone non basta, serve anche eseguire un cast durant l assegnazione
TH1F *BsMassBkg= (TH1F*)BsMass->Clone();
TH1F *BsMassSub= (TH1F*)BsMass->Clone();

int n; //la dichiaro qua ma mi serve dopo
```

[13]:
```
%%cpp
can5->cd(1);
//plottiamo l originale
BsMass->Draw();

can5->cd(2);
//ora modifichiamo la copia, possiamo
//ininnanzitutto mi serve sapere quanti bin ci sono
n=BsMass->GetNbinsX();
//eseguo un ciclo su tutti i bin
for (int i =0; i<n; i++){
    //faccio un if ricevendo il numero di dati in ogni bin
    if (BsMassBkg->GetBinContent(i)>200){
        //se il numero e maggiore di 200 lo azzero cosi da avere solo il fondo
        BsMassBkg->SetBinContent(i, 0);
    }
}
//disegno l istogramma di fondo
BsMassBkg->Draw();

can5->cd(3);
```

```
//ora sottraggo l istogramma di fondo a quello originale, per farlo basta␣
 ↪sommarli am dando un peso negativo
BsMassSub->Add(BsMassBkg, -1);
BsMassSub->Draw();

can5->Draw();
```




Una richiesta molto comune nell'analisi di un istogramma è il calcolo del suo integrale in un dato range, per fare questo esiste il metodo Integral(min, max).

E' necessario stare attenti poichè Integral richiede in input il min e max intesi come numero di bin non come range nelle unità del grafico.

Per conoscere il particolare numero di un bin esiste il metodo FindBin() il quale riceve una coordinata in "UserUnit" e restituisce il bin corriscondente.

Supponiamo per esempio di voler conoscere il valor dell'integrale (che corrisponde al numero di eventi) tra 5.3 e 5.4 dell'istogramma precedente, basterà allora fare:

[14]:
```
%%cpp
//qui serve mettere cout nel framework non e necessario
cout<<BsMass->FindBin(5.3)<<endl;
cout<<BsMass->FindBin(5.4)<<endl;
```

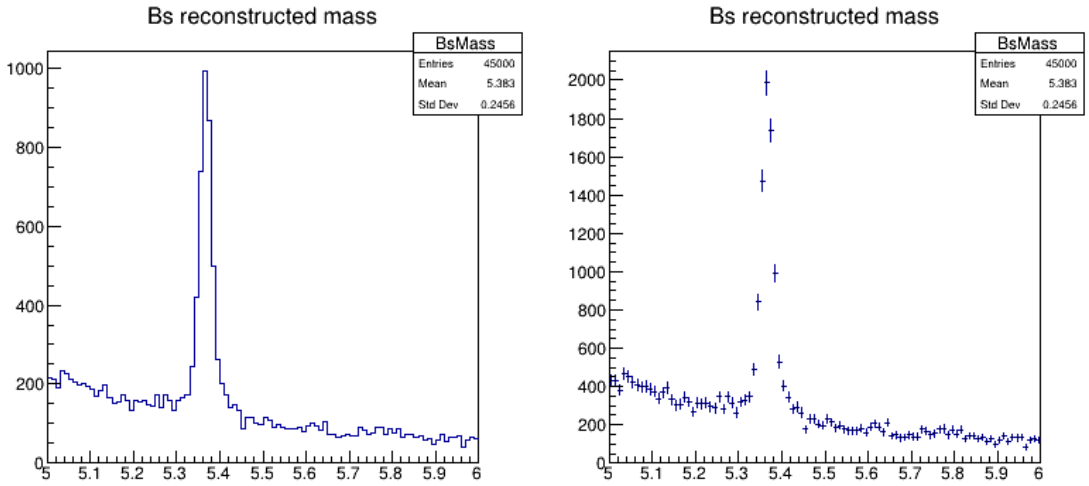
```
30
41
```

[15]:
```
%%cpp
//ora posso integrare o mettendo i numeri a schermo oppure chiamando di nuovo␣
 ↪findbin
cout<<BsMass->Integral(BsMass->FindBin(5.3),BsMass->FindBin(5.4));
```

```
4856
```

E' inoltre possibile riscalare un istogramma di un fattore arbitrario (positivo), supponendo per esempio di voler normalizzare due istogrammi a un certo valore basterà riscalarli in modo tale che lo raggiungiano, Il metodo per far ciò è Scale(double):

[16]:
```
%%cpp
TCanvas *cans= new TCanvas("cans", "cans", 800, 400); //creo il canvas e lo␣
 ↪divido
cans->Divide(2,1); //divido il canvas

TH1F *BsMassScale= (TH1F*)BsMass->Clone(); //faccio una copia
BsMassScale->Scale(2); //raddoppio il contenuto di ogni bin
cans->cd(1);
BsMass->Draw(); //disegno l originale
cans->cd(2);
BsMassScale->Draw(); //disegno la copia rebinnat

cans->Draw();
```




Infine un ultimo metodo che risulta particolarmente utile in caso si stia facendo analisi statistica è il metodo Rebin. Esso permetto di modificare il binning di un istogramma unendo due o più bin

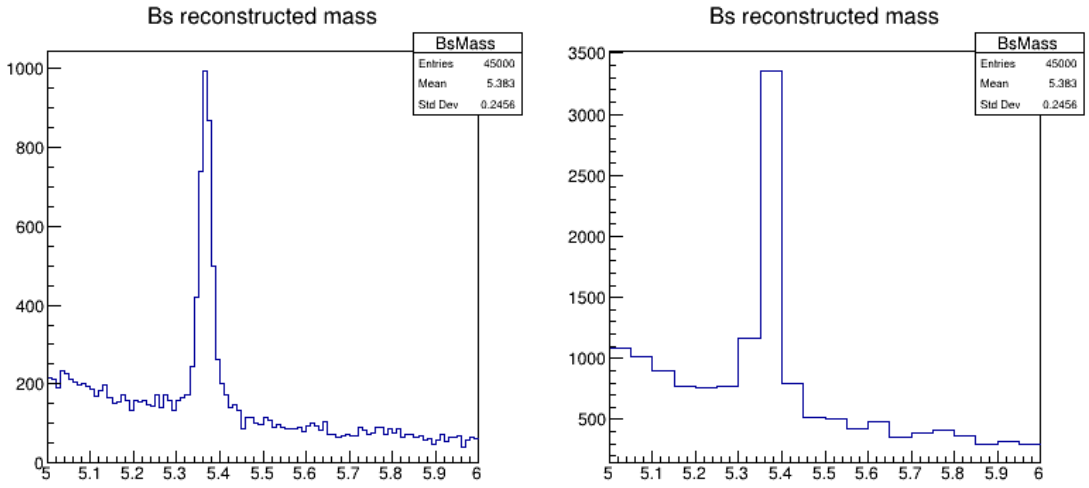
**Nota** Il metodo riceve in input un qualsiasi intero ma è necessario che questo sia un sottomultiplo del numero di bin originariamente presenti perchè l'operazione funzioni correttamente

**Consiglio** Quando definite un TH1 usate un binning adeguato ad esempio 100, 500, 2000 o comqunue numeri facilmente divisibili, nel dubbio abbondate con i bin tanto si può sempre rebinnnare ma non il contrario.

**Consiglio 2** Il Rebinning è irreversibile quindi prima di salvare un istogramma rebinnato (mai farlo piuttosto copiatelo e salvate la copia rebinnata) assicuratevi di avere i dati salvati in un altro posto o di essere sicuri di ciò che fate!

[17]:
```
%%cpp
TCanvas *can6= new TCanvas("can6", "can6", 800, 400); //creo il canvas e lo␣
 ↪divido
```

```
can6->Divide(2,1); //divido il canvas

//facciamo le copie degli istogrammi, bisogna essere un po accorti mentre si␣
 ↪chiama clone
//chiamare clone non basta, serve anche eseguire un cast durant l assegnazione
TH1F *BsMassClone= (TH1F*)BsMass->Clone();

BsMassClone->Rebin(5); //riduco a un quinto il numero dei bin
can6->cd(1);
BsMass->Draw(); //disegno l originale
can6->cd(2);
BsMassClone->Draw(); //disegno la copia rebinnata

can6->Draw();
```




### 1.3 TAxis e THistPainter

Queste due classi gestiscono la parte grafica dell'istogramma, in particolare (come si evince dal nome) il TAxis gestisce la grafica (ma non solo degli assi) mentre il THistPainter ha un ruolo analogo a quello dei painters visti in precedenza.

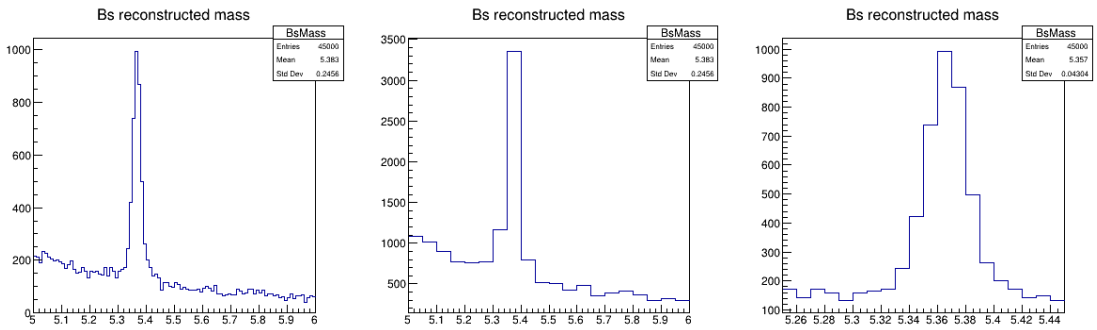
Ogni TH1 ha una coppia (o tripletta di) assi i quali possono essere assegnati a dei puntatori o più semplicemente modificati direttamente (sempre tramite locazione dinamica) attraverso i metodi - GetXaxis - GetYaxis - GetZaxis

I quali ritornano un puntatore al rispettivo asse dell'istogramma. Una volta ottenuto l'asse si possono eseguire le istruzioni a esso inerenti, ad esempio si può settare il Range che si vuole disegnare tramite - SetRange - SetRangeUser

La differenza è che User usa le unità disegnate mentre il metodo "classico" richiede il numero dei

bin, proviamo ad esempio a zoomare l'istogramma di prima sul picco:

[18]:
```
%%cpp
TCanvas *can7= new TCanvas("can7", "can7", 1200, 400); //creo il canvas e lo␣
 ↪divido
can7->Divide(3,1); //divido il canvas

//facciamo le copie degli istogrammi, bisogna essere un po accorti mentre si␣
 ↪chiama clone
//chiamare clone non basta, serve anche eseguire un cast durant l assegnazione
TH1F *BsMassClone1= (TH1F*)BsMass->Clone();
TH1F *BsMassClone2= (TH1F*)BsMass->Clone();

can7->cd(1);
BsMass->Draw(); //disegno l originale
can7->cd(2);
//la prima chiamata mi da il puntatore a TAxis mentre la seconda setta il range
BsMassClone->GetXaxis()->SetRange(25, 45);
BsMassClone->Draw(); //disegno la copia

can7->cd(3);
//alternativamente si puo prima ricevere l asse e poi modficarlo
//assegno il taxis
TAxis *Xax=BsMassClone2->GetXaxis();
//modifico il range in user units
Xax->SetRangeUser(5.25, 5.45);
BsMassClone2->Draw(); //disegno la copia

can7->Draw();
```

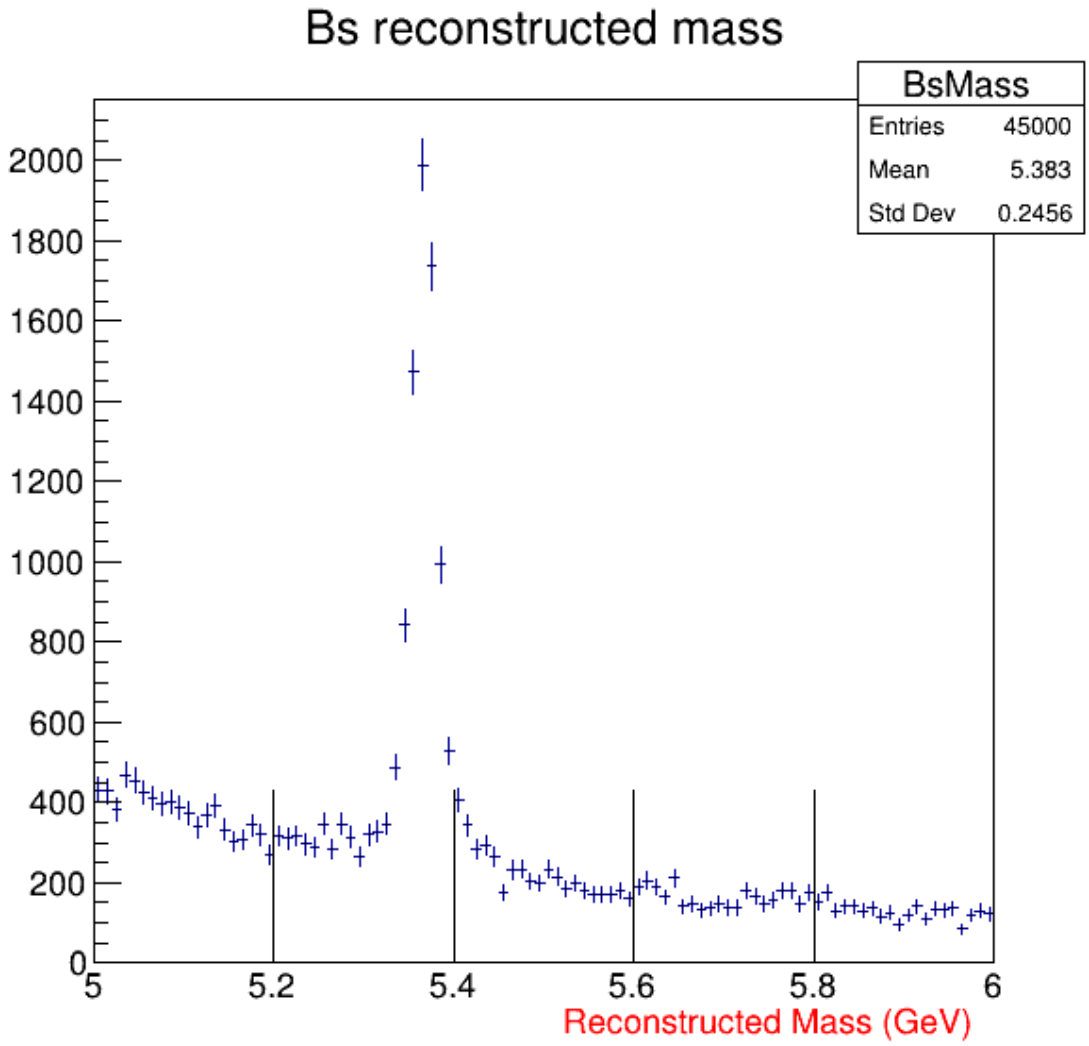



**IMPORTANTE** Come potete notare chiamare SetRange e SetRangeUser ha avuto due risultati ben diversi, questo per Set Range ha eseguito un rebinning ma ha tenuto costante il range che viene disegnato mentre set range user ha modificato il range mostrando meno bin ma tenendo binning costante!!!

In caso stiate scivendo un codice usate SetRangeUser se volete zommare su zone dei grafici, se avete disponibile però il bin e non il valore ricordate che esiste GetBinContent!

Altre funzioni importanti sono la possibilità di cambiare il ticking e il titolo agli assi con gli appositi metodi, possiamo fare direttamente un esempio:

[19]:
```
%%cpp
TCanvas *can8= new TCanvas("can8", "can8",600, 600); //creo il canvas e lo␣
 ↪divido
can8->cd();

//prendo asse X
TAxis *Xax2=BsMassScale->GetXaxis();
//gli do un titolo
Xax2->SetTitle("Reconstructed Mass (GeV)");
//metto un tick a mia scelta
Xax2->SetTickLength(0.1);
Xax2->SetTickSize(0.2);

//scelgo il numero di divisioni dell asse
Xax2->SetNdivisions(5);

//cambio colore al titolo
Xax2->SetTitleColor(kRed);

BsMassScale->Draw();
can8->Draw();
```

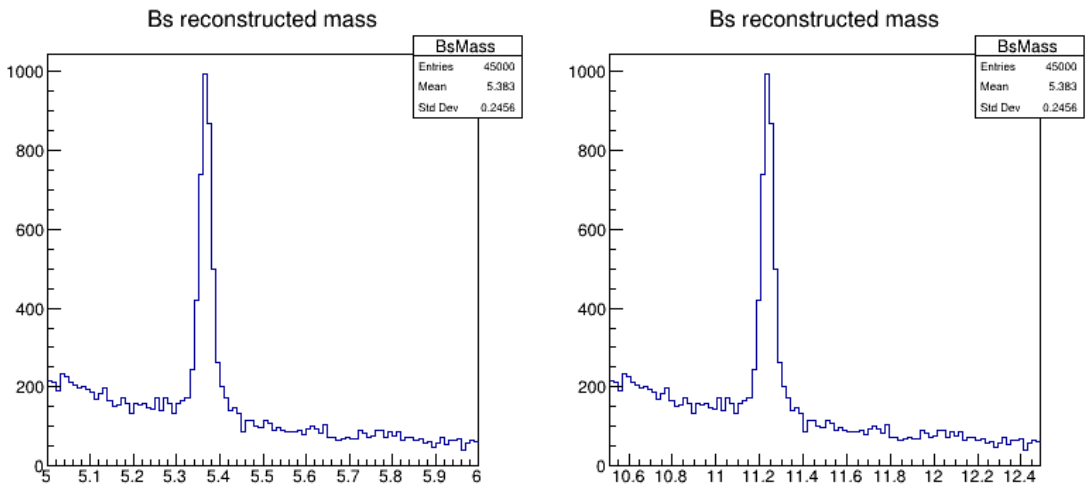



Un'ultima funzione estremamente utile dei TAxis è il metodo SetLimits(min, max) che modifica il range in cui è definito l'istogramma "mappando" i suoi estremi in una nuova coppia di punti, supponiamo per esempio di sapere che l'istrogramma di prima ricostruisce la massa non nell'unità corretta ma con un fattore di scala lineare, sarà allora sufficiente mappare gli estremi usando la funzione di scala.

In genere questo metodo viene usato insieme a GetLast e GetFirst con cui si ottengono i bin iniziali e finali e GetBinCenter con cui si ottinene il centroide di tali bin, questo poichè SetLimits lavora in user units

[20]:
```
%%cpp
//definiamo i coefficienti di una retta
double mm=2, q=0.5;
```

```
TCanvas *can9= new TCanvas("can9", "can9", 800, 400); //creo il canvas e lo␣
 ↪divido
can9->Divide(2,1); //divido il canvas

//clono il th1f
TH1F *BsMassCl= (TH1F*)BsMass->Clone();

can9->cd(1);
BsMass->Draw(); //disegno l originale
can9->cd(2);
//per settare i limits convinee prendersi il taxis ma si puo fare anche senza␣
 ↪(solo piu lunga la riga di codice)
TAxis *assex=BsMassCl->GetXaxis();
//chiamo setlimits dandogli primo e ultimo bin (presi tramite taxis)
//e moltiplicandoli per i coefficienti della retta


assex->SetLimits(BsMassCl->GetBinCenter(assex->GetFirst())*mm+q,␣
 ↪BsMassCl->GetBinCenter(assex->GetLast())*mm+q);
BsMassCl->Draw( ); //disegno la copia rimappata

can9->Draw();
```




Il THistPainter analogamente a quanto visto per il painter dei TGraph permette di personalizzare la grafica di un TH1 con varie opzioni ( la lista completa per tutti i tipi di TH* è in referenza).

Principalmente le modifiche che si apportano riguardano il plot in stile istogramma o in stile punti con barre d'errore, per fare ciò è sufficiente usare la rispettiva opzione nella chiamata del metodo Draw().

Ci sono però altre varie opzioni, in particolare l'opzione "lego" utile in caso di istogrammi multidimensionali.

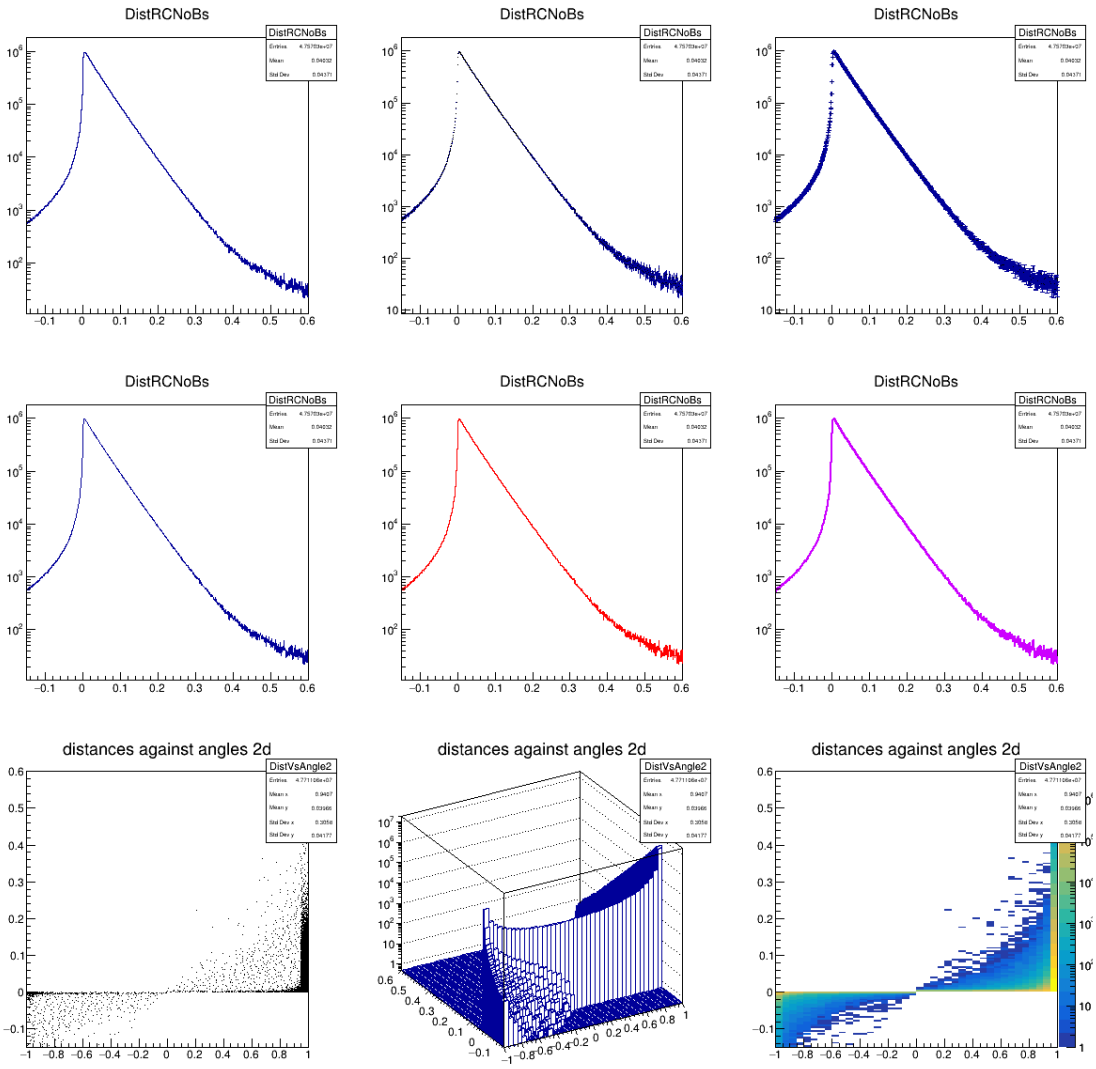
Per modificare colori e spessore e stile delle linee e dei punti la metodologia è la stessa che per i TF1 e i TGraph visto che i TH1 ereditano anch'esse da TAttLine e TAttMarker.

**Osservazione** Nel seguente codice faremo un esempio usando un TH2F, del quale abbiamo solo accennato, tuttavia sostanzialmente tutto quando detto per i TH1 si applica ugualmente al TH2 con le dovute generalizzazioni al caso 2-dim

[21]:
```
%%cpp
TCanvas * mcan = new TCanvas ("mcan", "mcan", 1200,1200);
mcan->Divide (3,3);
TVirtualPad *cc1=mcan->cd(1);
TVirtualPad *cc2=mcan->cd(2);
TVirtualPad *cc3=mcan->cd(3);
TVirtualPad *cc4=mcan->cd(4);
TVirtualPad *cc5=mcan->cd(5);
TVirtualPad *cc6=mcan->cd(6);
TVirtualPad *cc7=mcan->cd(7);
TVirtualPad *cc8=mcan->cd(8);
TVirtualPad *cc9=mcan->cd(9);

//faccio un clone senno alcune modifiche hanno impatto su tutti gli histo
TH1F *RCNoBs;
```

[22]:
```
%%cpp

//disegnamo questo istogramma in scala logaritmica
//in generale HISTO dovrebbe essere il modo standard di essere plottato ma␣
 ↪alcuni metodi lo modifcano
mcan->cd(1);
cc1->SetLogy();
RCNoBs = (TH1F*)DistRCNoBs->Clone();
RCNoBs->Draw("HISTO");

//ora disegnamolo con le barre di errore
mcan->cd(2);
cc2->SetLogy();
RCNoBs = (TH1F*)DistRCNoBs->Clone();
RCNoBs->Draw("E");

//le barre fanno varie opzioni, E1 fa le barre con linea ai capi
mcan->cd(3);
cc3->SetLogy();
RCNoBs = (TH1F*)DistRCNoBs->Clone();
RCNoBs->Draw("E1");
```

```
//C unisce con una linea "smooth"
mcan->cd(4);
cc4->SetLogy();
RCNoBs = (TH1F*)DistRCNoBs->Clone();
RCNoBs->Draw("C");

//cambiamo colore alla linea
mcan->cd(5);
cc5->SetLogy();
RCNoBs= (TH1F*)DistRCNoBs->Clone();
RCNoBs->SetLineColor(kRed);
RCNoBs->Draw();

//cambiamo colore 3 spessore alla linea
mcan->cd(6);
cc6->SetLogy();
RCNoBs= (TH1F*)DistRCNoBs->Clone();
RCNoBs->SetLineColor(kViolet);
RCNoBs->SetLineWidth(2);
RCNoBs->Draw();

//disegnamo il TH2
//il draw normale mostra tutti i punti in un piano 2D
mcan->cd(7);
DistVsAngle2->Draw();

//per avere un istogramma vero e proprio serve l opzione lego
mcan->cd(8);
//prima di disegnarlo serve rebinnare e settare in logaritmico
DistVsAngle2->RebinX(25);
DistVsAngle2->RebinY(5);
//ora che siamo in 2d l asse su cui ci sono il numero di aventi per bin non e y␣
 ↪ma z
cc8->SetLogz();
DistVsAngle2->Draw("lego");

//oppure lo si puo plottare in una scala cromatica
mcan->cd(9);
cc9->SetLogz();
DistVsAngle2->Draw("COLZ");

mcan->Draw();
```




## 1.4 Macro e file .mac e .sh

Il framework di ROOT permette di caricare ed eseguire macro, tuttavia cosa è una macro?

Le librerie di ROOT possono essere sostanzialmente usate in due modi: si possono integrare in un programma compilabile in C++ (complicato da integrare e necessita di alcuni flag in compilazione) di cui parleremo brevemente oppure si possono scrivere dei codici in formalismo C++ che vengono poi eseguiti direttamente all'interno del framework, questa seconda opzione è detta macro.

Facciamo un esempio: il file TH1toTxt.cc è un file c++ in cui è presente solamente un omonima funzione, esso non è eseguibile e compilabile con gcc (il classico compilatore C++) ma può essere eseguito comodamente all'interno del framework, analizziamolo meglio

[23]:
```
#per prima cosa vediamo cosa si trova nel file
#!!!!questo non è root sto eseguendo nella shell
!cat TH1toTxt.cc
```

```
void TH1toTxt (TH1F *h, string outfile) {
        ofstream out (outfile);
        int n=h->GetNbinsX();
        for (int i=0; i<n; i++)
out<<h->GetBinCenter(i)<<"\t"<<h->GetBinContent(i)<<endl;
}
```

Sostanzialmente questo codice chiede un istogramma e il nome di un file di output in cui creare le coppie di punti corrispondenti ai suoi bin (così da poterne fare poi un TGraph).

Il codice può essere comodamente eseguito all'interno del framework in due modi: 1. con l'instruzione .x si esegue direttamente la macro dando i paramentri di input a patto che il nome del file e quello della funzione principale (quella che viene eseguita, simile al main in cpp) coincidano 2. con l'istruzione .L si caricano tutte le funzioni presenti nel file e queste diventano poi eseguibili liberamente all'interno del framework

Nel notebook non è possibile eseguire questi comandi (non sono cp nativi) quindi faremo un esempio direttamente da terminale, in ogni caso il codice che bisogna usare sarà:

- root -l 100mSimK12_trigger.root -> apro il file root
- .x TH1toTxt.cc (hAll, "hAllout.txt") ->eseguo la macro su hAll

oppure nel secondo caso - root -l 100mSimK12_trigger.root -> apro il file root - .L TH1toTxt.cc ->carico la macro - TH1toTxt(hAll, "hAllout.txt") ->eseguo la macro su hAll

Il secondo approccio risulta utile nel caso in un unico file vi siano molte macro che si vogliono eseguire indipendentemente.

Infine analizziamo un ultimo caso: in caso si voglia usare la stessa macro ripetutamente è possibile scrivere un file che contiene un'esecuzione "ripetuta" della stessa, questo si può fare con i file .mac se si vuole agire all'interno del framework o con i file .sh se si vuole agire da terminale.

Supponiamo per esempio di voler eseguire ripetutamente la macro di prima sui TH1F hSim, hAll, hRec ed hSel allora è possibile "meccanizzare" la cosa in due modi come vedremo nei due file TH1conv.mac e TH1conv.sh

[24]:
```
!cat TH1conv.mac
```

```
.x TH1toTxt.cc (hSim, "hSimout.txt")
.x TH1toTxt.cc (hAll, "hAllout.txt")
.x TH1toTxt.cc (hSel, "hSelout.txt")
.x TH1toTxt.cc (hRec, "hRecout.txt")
```

In questo file vi è semplicemente l'istruzione di prima ripetuta 4 volte questo file va eseguito sempre con .x all'interno del framework di root quindi si può eseguire anche all'interno di un file già aperto.

In alternativa se all'interno della macro vi sono le istruzioni necessarie per ricevere autonomamente i dati (ovvero se per eseguirla non è necessario essere già all'interno di uno specifico TFile allora è possibile eseguire la macro direttamente nella shell del terminale e la sua "meccanizzazione" può

essere eseguita anche con un file .sh nel seguente modo: creiamo una macro che disegna e salva i file txt creati in precedenza:

[25]:
```
!cat TxttoTGraph.cc
```

```
void TxttoTGraph (string name) {
        TGraph * gr= new TGraph((name+".txt").c_str());
        TCanvas *c= new TCanvas("c", "c", 600, 600);
        c->cd();
        gr->Draw();
        c->Draw();
        c->Print((name+".png").c_str());
}

//to execute in batch
//root -l -b 'TxttoTGraph.cc ("hAllout"); exit (0)'
```

Il file semplicemente crea un TGraph e poi salva il canvas in png, piccola nota è il fatto che il metodo print chiede un array di char quindi è necessario convertire con .c_str().

La macro in questo caso può essere eseguita dentro al framework come già visto semplicemente chiamando - .x TxttoTGraph.cc ("hAllout")

Oppure eseguita nella shell con l'istruzione - root -l -b 'TxttoTGraph.cc ("hAllout"); exit (0)' Dove exit zero serve per tornare poi al terminale e -b serve per eseguire in background (ovvero non mostrerà i canvas)

Per meccanizzare questa operazione ai quattro file creati prima basta scrivere il codice di esecuzione in un file .mac da eseguire nel framework (quindi con quattro copie della prima istruzione) oppure in modo da eseguirlo all'esterno in un file sh eseguibile come un normale file bash

[26]:
```
!cat Txtconv.sh
```

```
root -l -b 'TxttoTGraph.cc ("hAllout"); exit (0)'
root -l -b 'TxttoTGraph.cc ("hSimout"); exit (0)'
root -l -b 'TxttoTGraph.cc ("hRecout"); exit (0)'
root -l -b 'TxttoTGraph.cc ("hSelout"); exit (0)'
```

Questo file semplicemente esegue 4 volte l'struzione ed esce ogni volta in modo che sia nuovamente chiamabile da shell, per eseguirlo è sufficiente digitare - source Txtconv.sh

La cosa comoda è che volendo in questo secondo modo si possono aggiungere comandi nativi della shell, per esempio si può chiedere alla funzione di aprire i file .png creati, come in questo secondo file (analogo al primo ma con un'istruzione in più):

[27]:
```
!cat Txtconv2.sh
```

```
root -l -b 'TxttoTGraph.cc ("hAllout"); exit (0)'
root -l -b 'TxttoTGraph.cc ("hSimout"); exit (0)'
root -l -b 'TxttoTGraph.cc ("hRecout"); exit (0)'
root -l -b 'TxttoTGraph.cc ("hSelout"); exit (0)'
```

```
eog *.png &
```