



# Introducción a ROOT

*Sonja Orrigo*

*Curso de Técnicas Experimentales Avanzadas en Física Nuclear*

*Master Inter-universitario de Física Nuclear, Curso 2023-2024*





ROOT  
Data Analysis Framework

- ROOT es un proyecto Open Source que se empezó en 1995
- El proyecto es desarrollado por una colaboración entre CERN y Fermilab/USA, más muchos otros colaboradores part-time
- Además muchos usuarios registrados (6000 en el fórum RootTalk) contribuyen con feedback, comentarios y en solucionar bugs

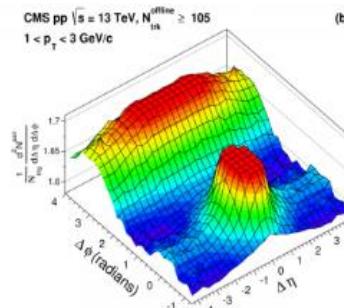
- Main ROOT page: <http://root.cern.ch>
- Class Reference Guide: <http://root.cern.ch/root/html>
- User Guides and Manuals: <https://root.cern.ch/root-user-guides-and-manuals>
- Tutorials: <https://root.cern.ch/courses>

[Getting Started](#)[Reference Guide](#)[Forum](#)[Gallery](#)

## ROOT is ...

A modular scientific software framework. It provides all the functionalities needed to deal with big data processing, statistical analysis, visualisation and storage. It is mainly written in C++ but integrated with other languages such as Python and R.

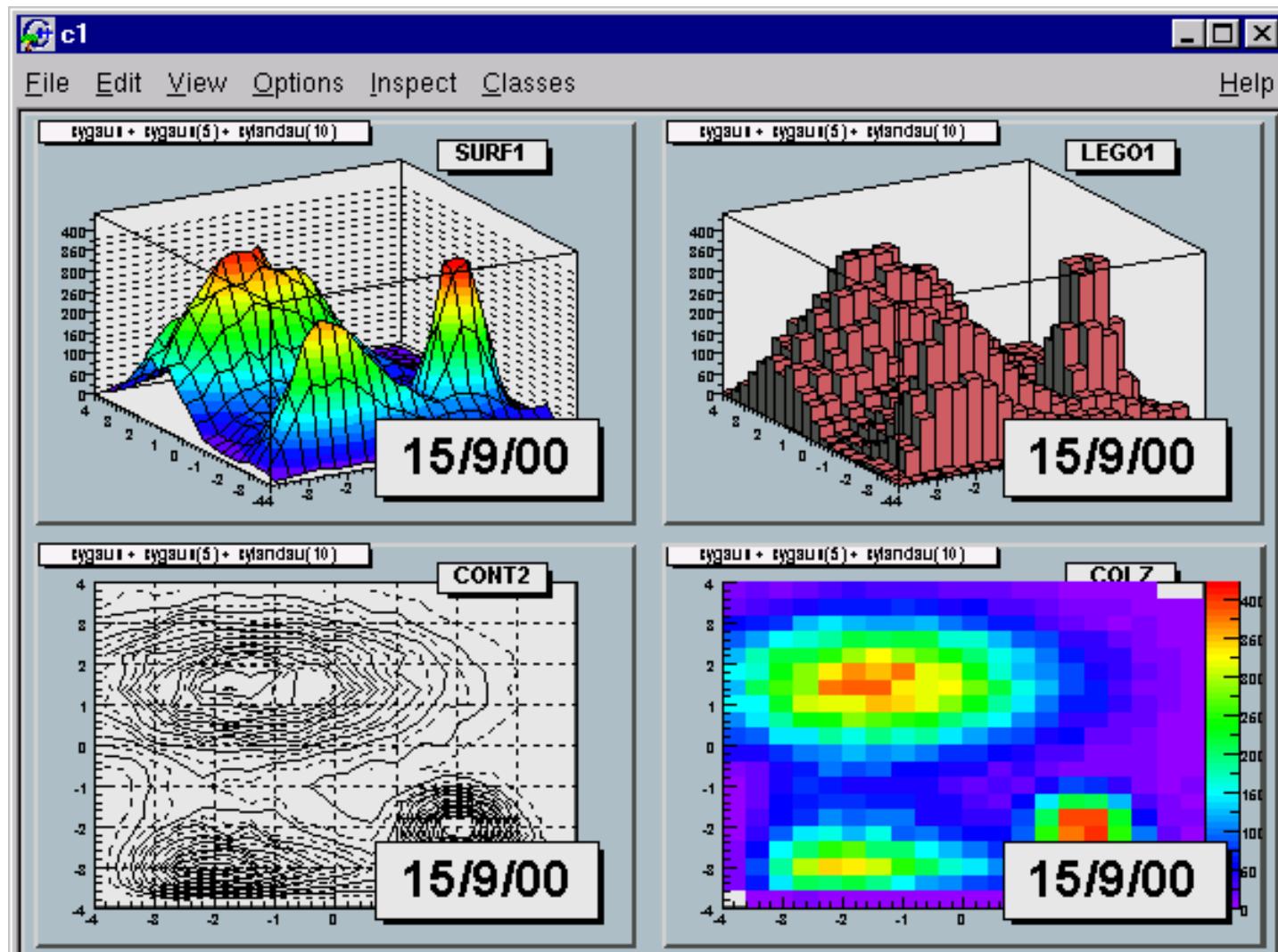
[Start from examples](#) or [try it in your browser!](#)

[Download](#)[or Read More ...](#)[Previous](#) [Resume](#) [Next](#)

# ROOT

- Es un framework modular de software científico que se utiliza comúnmente para análisis de datos en física nuclear y de altas energías
- Utiliza mayormente el lenguaje de programación C++: desarrollado en C++; interprete nativo de C++; pero se puede usar también python

- Es una herramienta muy poderosa y versátil que permite de manipular, procesar y analizar largas cantidad de datos
- Permite una amplia variedad de métodos de análisis, representaciones de datos, fits, ...
- Extensas capacidad de visualización de datos científicos en 2D y 3D



# Varias maneras de utilizar ROOT

- El lenguaje de los comandos y programas es C++
- ROOT contiene un interprete de C++: CINT en ROOT5, CLING en ROOT6
- Dependiendo de las exigencias de los usuarios, ROOT se puede utilizar en distintas maneras:
  - **ROOT command line**
    - ✓ Empezando una sesión de ROOT en un terminal, el interprete de ROOT ejecuta cada línea de comando que se escribe: ejecuta una línea a la vez
    - ✓ Ventaja en comparación con la compilación: se ve el resultado sin ninguna espera
    - ✓ Desventaja: interpretar los comandos es mucho más lento que ejecutar código compilado  
⇒ La línea de comandos es útil en fase de escritura del código, para testear los comandos y hacer rápidos checks de los datos, mientras que los programas más largos van compilados

\$ root	\$ root -l	Starting ROOT
root [0] Int_t number = 2;		The ROOT prompt
root [1] cout<<"number = "<<number<<endl;		
root [#] .q		Ending ROOT

# Varias maneras de utilizar ROOT

## ○ ROOT macros

- ✓ Si tienes varias líneas de comandos para ejecutar es más cómodo ponerlas en un fichero que se pueda editar y ejecutar varias veces en ROOT: una macro
- ✓ Una macro es un **fichero .C o .cc** donde escribir secuencialmente los comandos
- ✓ La macro es **interpretada línea a línea** para el interprete CINT/CLING  
⇒ no es compilada, equivale a utilizar las líneas de comandos
- ✓ Importante: en la macro cada línea de código se tiene que terminar con ;

### Unnamed macros (no parameters) ←(ONLY UP TO ROOT 5)

**\$ root -l macro.C** Execute the macro

**root [0] .x macro.C** Execute the macro

```
{  
  commands...  
}
```

### Named macros (you can pass parameters to the function)

**root [0] .L analysis.C** Load the macro in the memory

**root [1] analysis(20)** Call the macro to execute it

**root [0] .x analysis.C(20)** Load and execute the macro

```
void analysis(int N)  
{  
  commands...  
  return;  
}
```

# Varias maneras de utilizar ROOT

- **Compiled ROOT macros *on the fly***

- ✓ Ejecutar código compilado es muchos más rápido que interpretar línea a línea
- ✓ Además mediante la compilación se pueden detectar errores que no se perciben interpretando línea a línea
- ✓ Compilar es importante para los programas más largos y/o lentos
- ✓ El mismo **fichero .C o .cc** se puede compilar, pero la macro tiene que tener un nombre
- ✓ La compilación esta hecha por

**ACLiC (The Automatic Compiler of Libraries for CINT)**

```
void analysis(int N)
{
    commands...
    return;
}
```

## Named macros ONLY

**root [0] .L analysis.C++**

Load the macro in the memory and compile it

**root [1] analysis(20)**

Call the macro to execute it

**root [0] .x analysis.C(20)++**

Load, compile and execute the macro

# Varias maneras de utilizar ROOT

## ○ Stand-alone application

- ✓ La manera más sofisticada es construir una aplicación *stand-alone*
- ✓ ROOT es básicamente una colección de clases de C++
- ✓ Estas se pueden utilizar para construir nuevas aplicaciones que se pueden compilar y ejecutar del todo independientemente desde el ROOT originario
- ✓ Para cumplir con los estándares de C++, se incluye en el código una función **main()** que solo es vista por el compilador g++ y no para CINT

Compile the program

```
$ g++ -o analysis analysis.cc `root-config --cflags --glbs`
```

Execute it outside ROOT

```
$ ./analysis
```

```
void analysis()
{
    commands...
    return;
}

#ifndef __CINT__
int main()
{
    analysis();
    return 0;
}
#endif
```

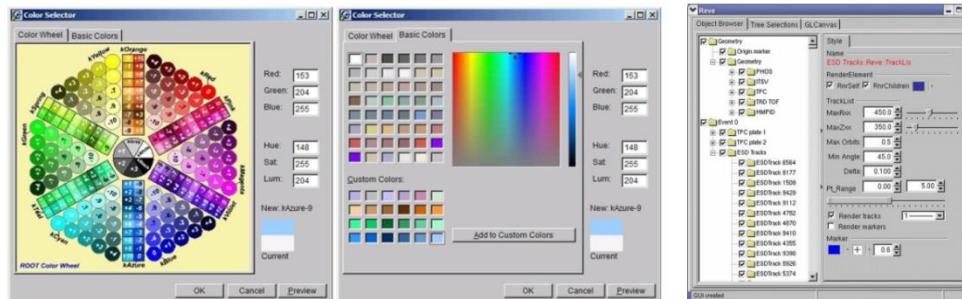
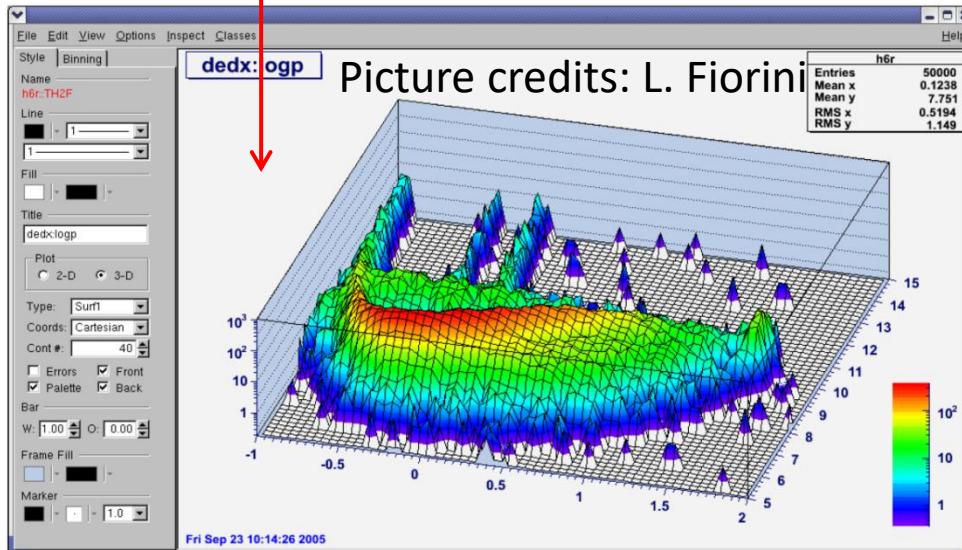
# Varias maneras de utilizar ROOT

## ○ Graphical interfaces

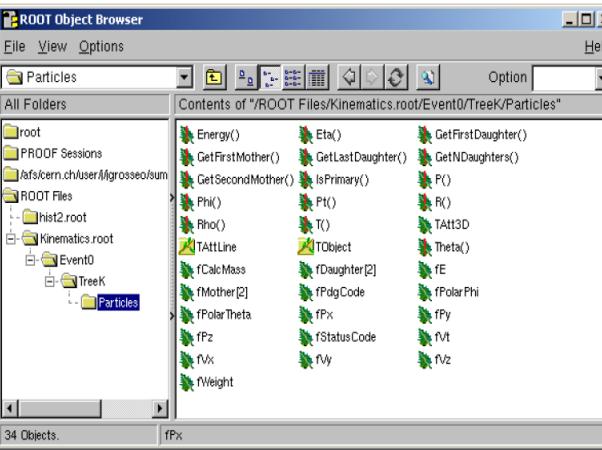
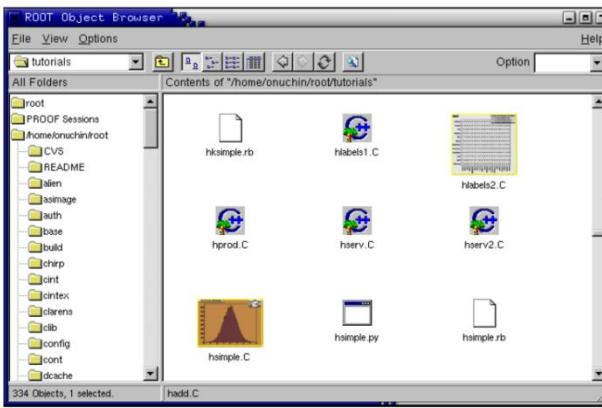
ROOT Object browser (TBrowser), Graphical user-interface (GUI), Editor, Fit Panel...

- ✓ Permiten un uso más interactivo: abrir ficheros, explorarlos, mirar los trees, plotear objetos, editarlos, cambiar el estilo, fitear...

La ventana gráfica se llama **TCanvas**



**root[] new TBrowser**



# Varias maneras de utilizar ROOT

## ○ ROOTBook

- ✓ ROOT interactivo en el navegador web (parecido a un notebook de *Mathematica*):

[https://root.cern.ch/notebooks/HowTos/HowTo\\_ROOT-Notebooks.html](https://root.cern.ch/notebooks/HowTos/HowTo_ROOT-Notebooks.html)

### HowTo ROOT-Notebooks: pure C++ notebook

The manual switch to C++ mentioned above can be done by typing:

```
In [16]: ROOT.toCpp()
```

Notebook is in Cpp mode

Now our notebook behaves as the ROOT prompt, with no need of any magic.

```
In [17]: cout << "From this point on..." << endl;
```

From this point on...

```
In [18]: cout << "... it's only C++ ..." << endl;
```

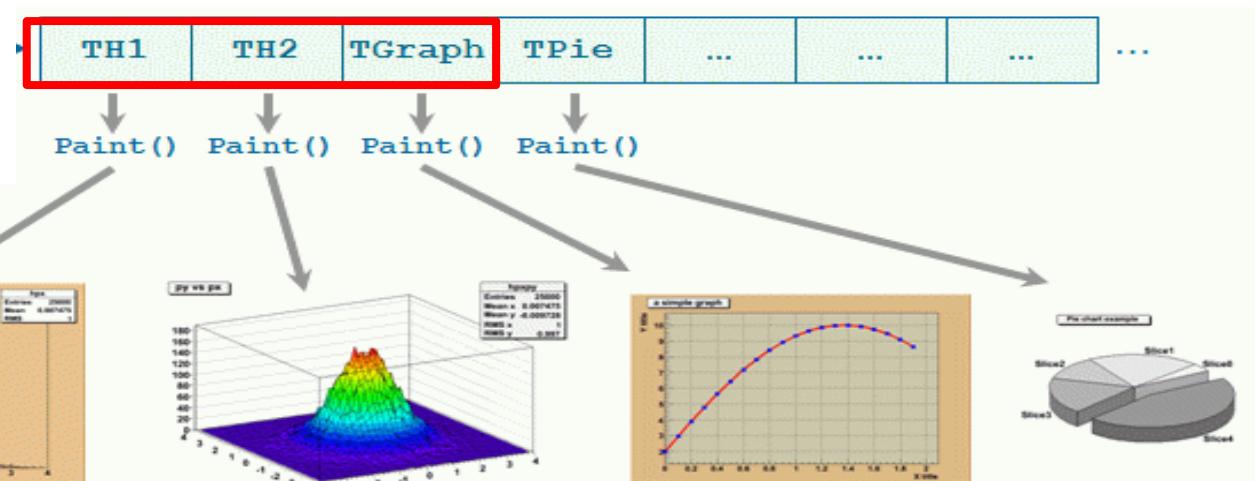
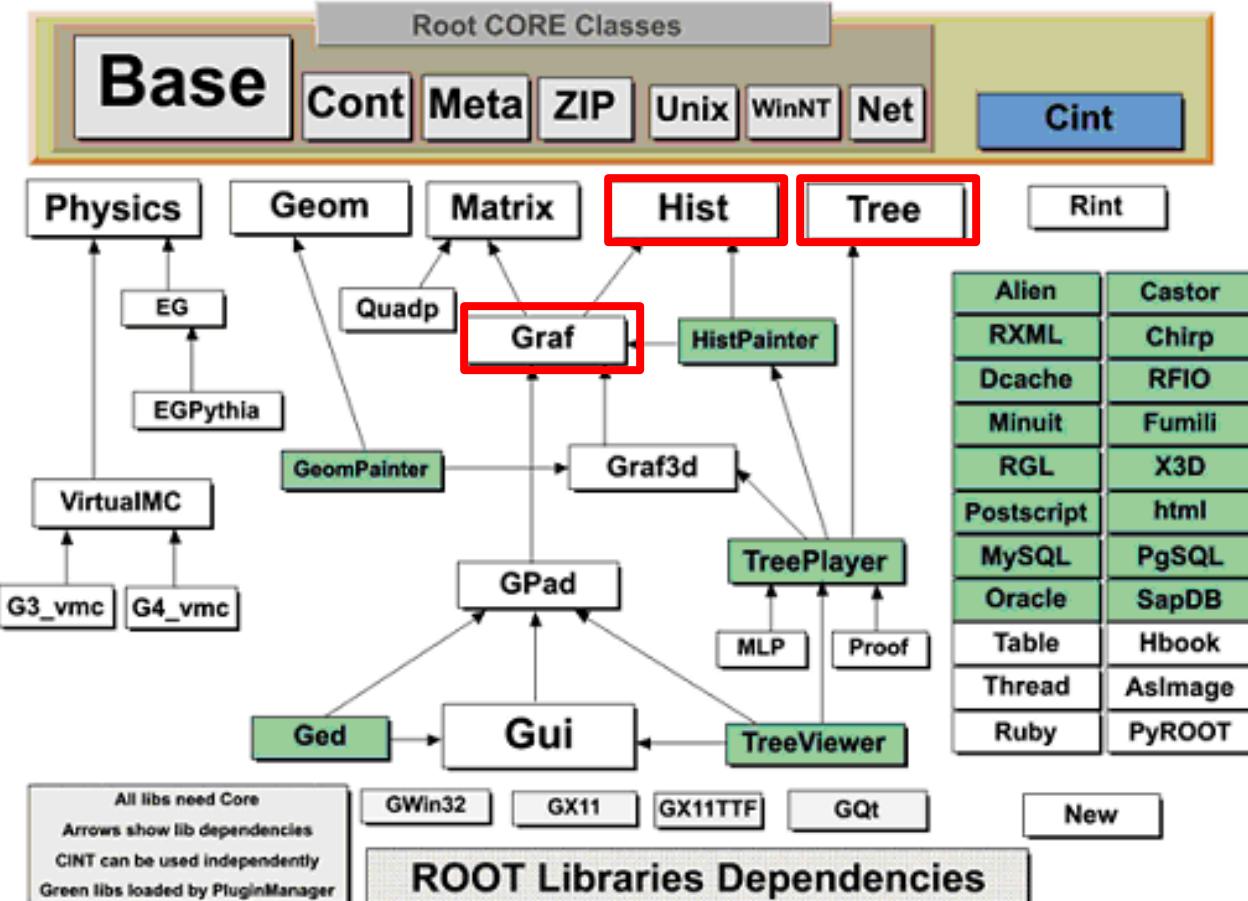
... it's only C++ ...

```
In [19]: cout << "... With the usual goodies!" << endl;
std::unordered_map<int, string> m = {{1, "one"}, {2, "two"}, {3, "three"}}
```

... With the usual goodies!  
(std::unordered\_map<int, std::string> &) { 3 => "three", 1  
=> "one", 2 => "two" }

- ROOT contiene muchísimas librerías y clases de objetos
- En este curso nos enfocaremos en los objetos típicamente utilizados en análisis de datos en física:

- ✓ Funciones
- ✓ Histogramas
- ✓ Graphs
- ✓ Trees



- **Variables en ROOT**

- **Variables en ROOT**

int	→	Int_t
float	→	Float_t
double	→	Double_t
char	→	Char_t
bool	→	Bool_t
...		

- **Colors**



- **Markers**



# TCanvas

- **Canvas:** La Canvas es una ventana grafica donde se dibujan todos los objetos de ROOT:  
*“area mapped to a window directly under the control of the display manager”*
- La clase correspondiente de ROOT se llama **TCanvas**
- Si la Canvas no existe, ROOT la crea automáticamente al dibujar un objeto
- Las Canvas se pueden dividir en **Pads**

TCanvas (const char \*name, const char \*title, Int\_t wtopx, Int\_t wtopy, Int\_t ww, Int\_t wh)

**root [ ] TCanvas \*c1 = new TCanvas("c1", "Ventana grafica",200,10,700,900)**

(name,        title,         $x_0, y_0, x_1, y_1$ )

**root [ ] c1->Divide(1,2)**

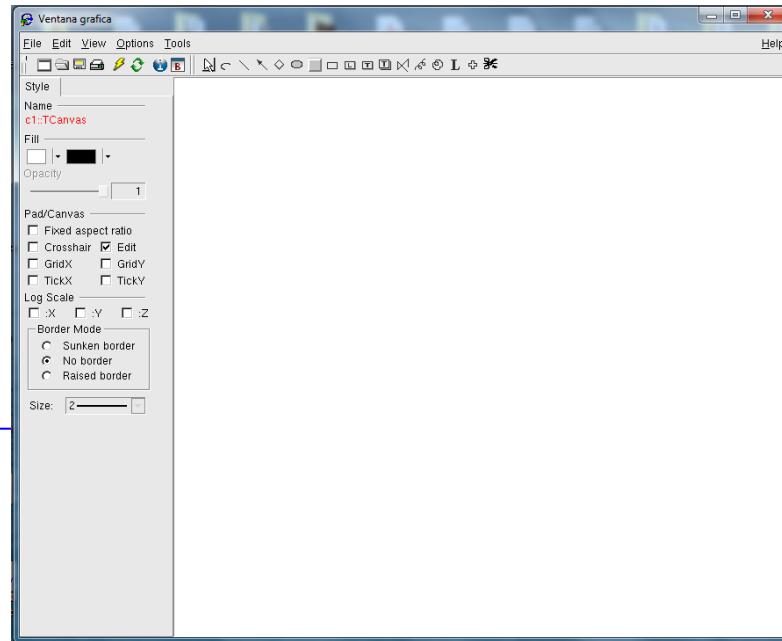
**root [ ] c1->cd(1)**

**root [ ] c1->cd(2)**

**root [ ] c1->SaveAs("file.png")**

**root [ ] c1->SaveAs("file.pdf")**

**root [ ] c1->SaveAs("file.C")**



# Tutorials

# Funciones

- **Función:** Correspondencia que asocia a cada elemento de un primero conjunto uno y un solo elemento de un segundo conjunto. Es. una línea recta:  $y = a x + b$
- Utilizando la **clase TF1** se pueden definir funciones 1D
- Las **clases TF2** y **TF3** permiten crear funciones 2D y 3D

- Función creada como ***value-type*** (instancia de un objeto)

```
root [ ] TF1 f2("func", " sin(x) ", 0, 10)
```

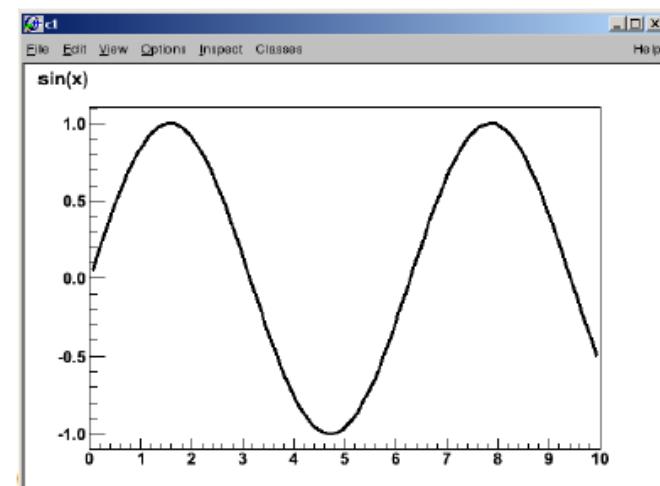
- Función creada como ***pointer*** (punta a la instancia del objeto)

```
root [ ] TF1* f1 = new TF1("func", "sin(x)", 0, 10)
```

- "func" is a (unique) name
- "sin(x)" is the formula
- 0, 10 is the x-range for the function

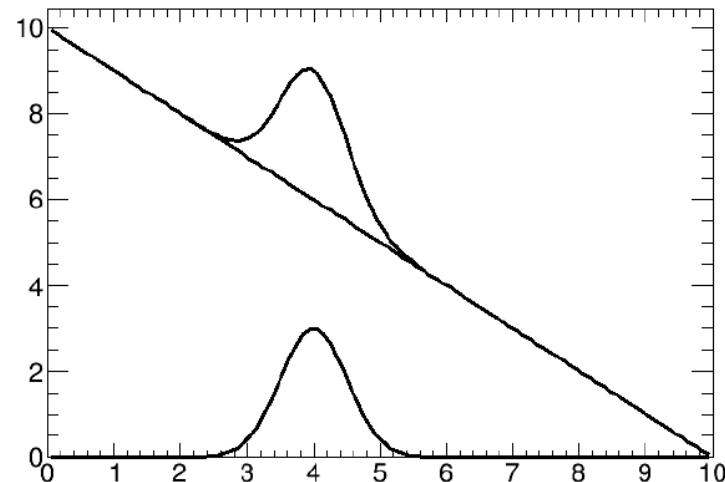
```
root [ ] f1->Draw()
```

```
root [ ] f1->SetLineColor(kRed)
```



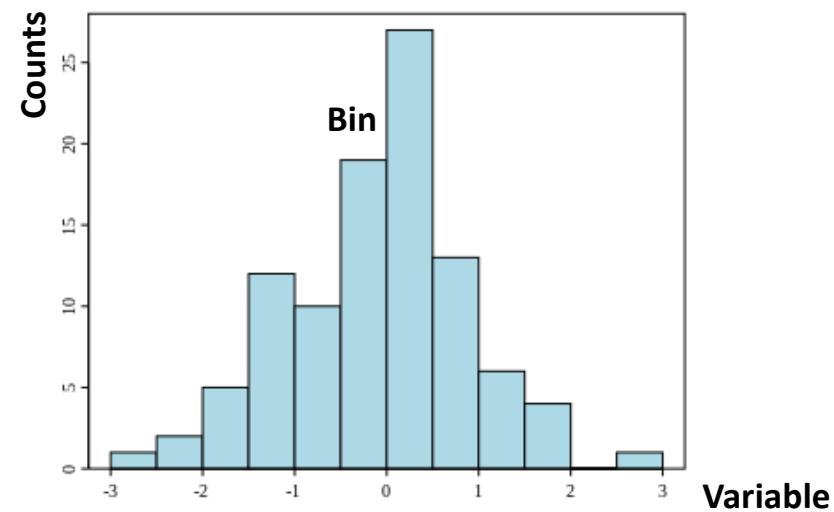
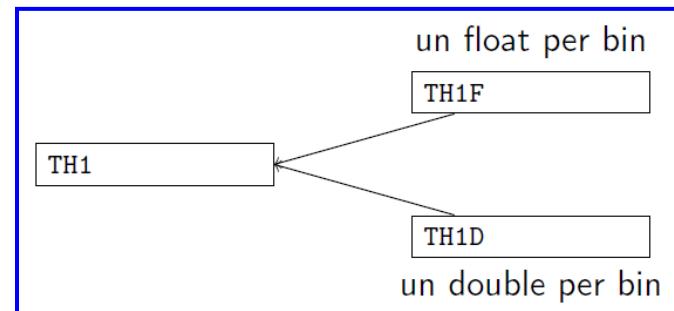
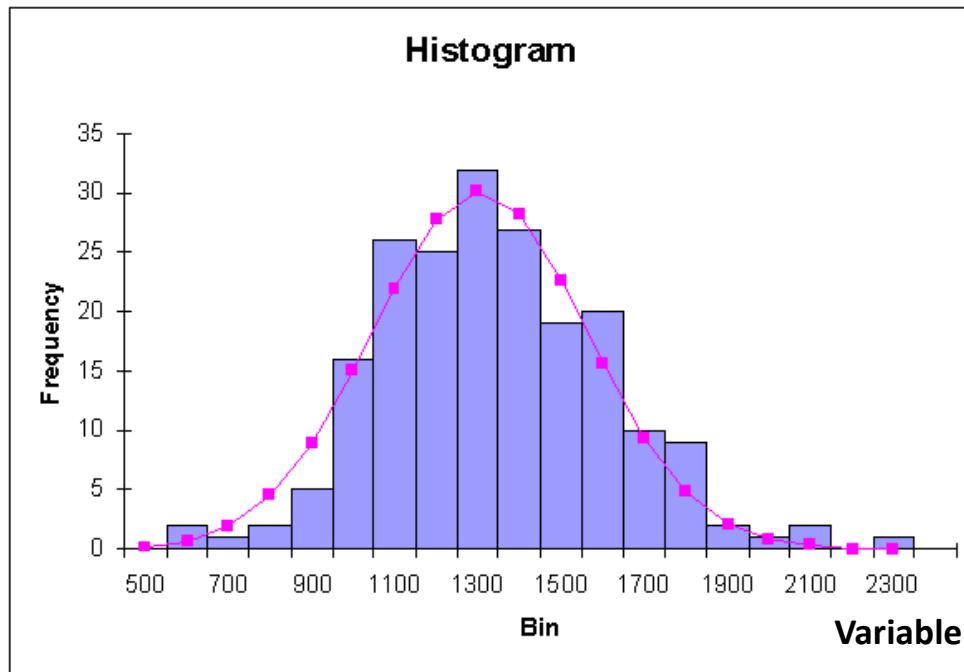
# Funciones: ejemplos

```
root [] TF1 *f1 = new TF1("f1","gaus ",0,10)
root [] TF1 *f2 = new TF1("f2","10.-x",0,10)
root [] f1->Draw()
root [] f2->Draw()
root [] f1->SetParameter(0,2)
root [] f1->SetParameter(1,4)
root [] f1->SetParameter(2,2.5)
root [] f1->Draw()
root [] TF1 *f3 = new TF1("f3","f1+f2",0,10)
root [] f3->Draw()
root [] f3->SetParameter(0,3)
root [] f3->SetParameter(2,0.5)
root [] f3->Draw()
root [] f2->Draw("same")
root [] f1->SetParameter(0,3)
root [] f1->SetParameter(2,0.5)
root [] f1->Draw("same")
```



# Histogramas

- **Histograma:** Un histograma es un grafico que representa el numero de sucesos que pertenecen a una categoría (intervalo de una variable)
- Los intervalos en que se divide el eje de la variable se llaman **bines**
- Tener cuidado en la elección del numero de bines (**binning**) del histograma
- Las clases de histogramas en ROOT heredan desde la **clase TH1** y son de varios tipos, se utilizan más **TH1F** y **TH1D**
- Hay también histogramas en 2D y 3D



# Histogramas

- Los histogramas son una de las clases de ROOT más importantes para los físicos

- Histograma creado como ***value-type*** (instancia de un objeto)

```
root [ ] TH1F h2("hist2","A new histogram",50,-25,25)
```

- Histograma creado como ***pointer*** (punta a la instancia del objeto)

```
root [ ] TH1F *h = new TH1F("hist","Histogram",10,0,10)
```

```
root [ ] TH1F *h = new TH1F("hist", "histogram title", 10, 0, 10)
```

- "hist" is a (unique) name
- "histogram title" is the title of the histogram
- 10 is the number of bins
- 0, 10 are the limits on the x axis.  
Thus the first bin is from 0 to 1,  
the second from 1 to 2, etc.

```
root [ ] h->Fill(3.5)
```

```
root [ ] h->Fill(5.5)
```

```
root [ ] h->Draw()
```

```
root [ ] h->Rebin(2)
```

Example credits: ROOT Tutorial, L. Fiorini,

[http://ifac.uv.es/~fiorini/ROOTTutorial/root\\_tutorial.pdf](http://ifac.uv.es/~fiorini/ROOTTutorial/root_tutorial.pdf)

# Histogramas 2D

- Se pueden definir histogramas en 2D utilizando las clases **TH2F** y **TH2D**

```
root [ ] TH2D h2d("h2d","A 2-dimensional histogram",binX,x1,x2,binY,y1,y2)
```

```
root [ ] gStyle->SetPalette(1)
```

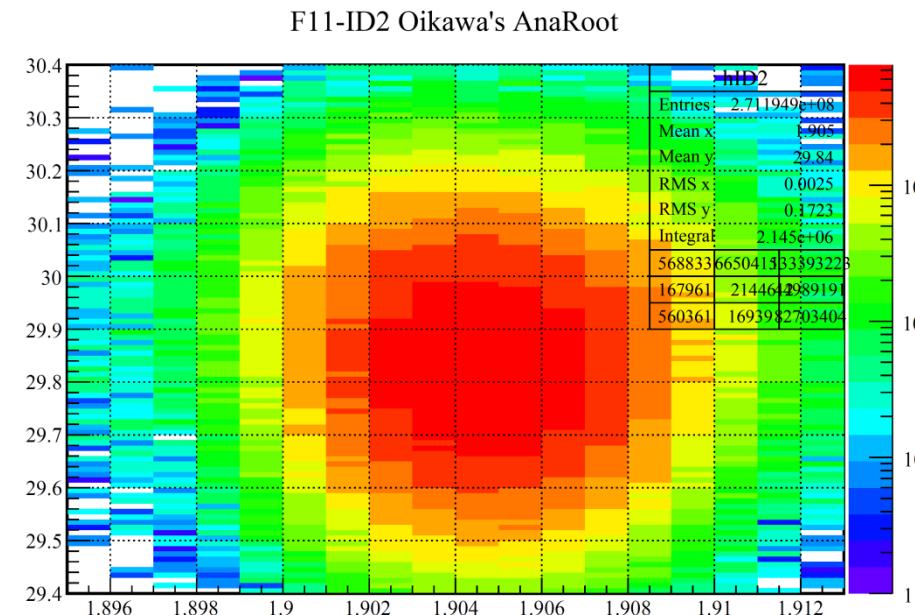
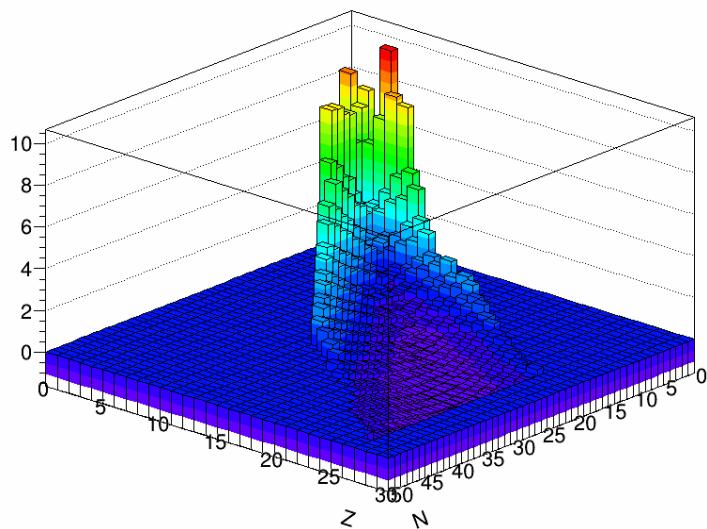
```
root [ ] h2d->Draw("colz")
```

```
root [ ] h2d->Draw("lego")
```

```
root [ ] TH2F *hID2 = new TH2F("hID2","F11-ID2 Oikawa's AnaRoot",800,1.8,2.6,4000,10,50)
```

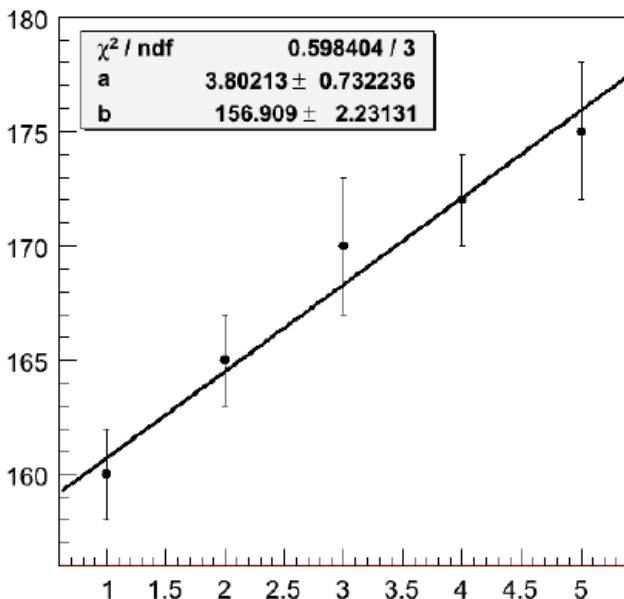
```
root [ ] hID2->GetXaxis()->SetRangeUser(1.895,1.913)
```

```
root [ ] hID2->GetYaxis()->SetRangeUser(29.4,30.4)
```



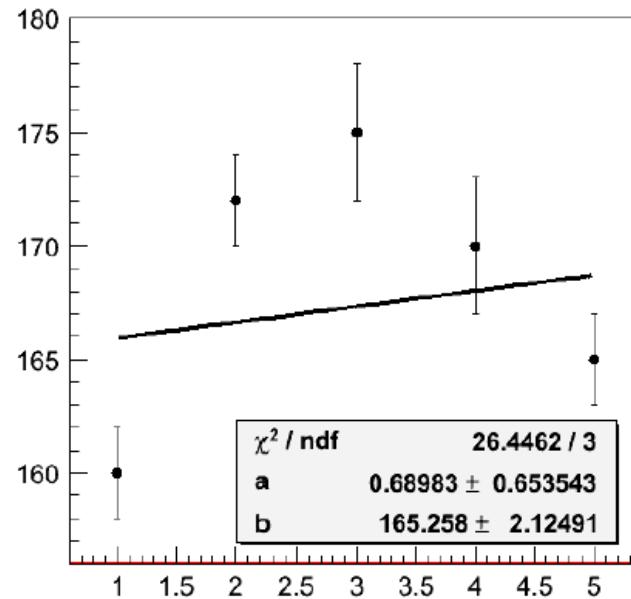
# Fits

- **Fit:** Un fit (ajuste) es una procedimiento que permite de comparar un conjunto de datos con una función y determinar los parámetros que mejor se adaptan a los datos
- El valor de  $\chi^2/\text{ndf}$  nos da una medida de la calidad del fit
- Numero de grados de libertad **ndf** = numero de los puntos meno el numero de parámetros de la función de fit



$$\chi^2 = \sum_i \frac{(y_i - f(x_i))^2}{(\Delta y_i)^2}$$

Picture credits: M. Floris



# Fits de histogramas

Fit of a histogram

```
root [ ] TF1 *fitfunction = new TF1("fitfunction","gaus",x1,x2)
```

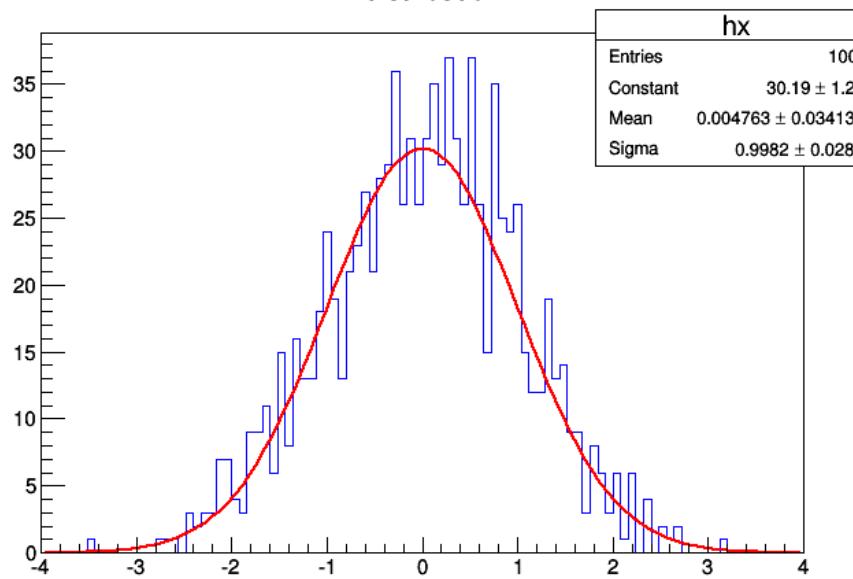
```
root [ ] h->Fit("fitfunction","R")
```

Getting  $\chi^2$  and ndf

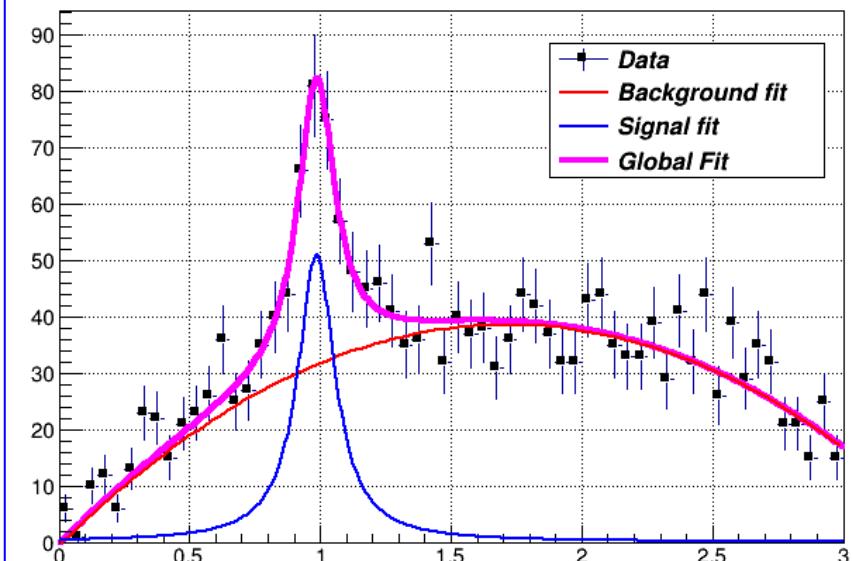
```
root [ ] fitfunction->GetChisquare()
```

```
root [ ] fitfunction->GetNDF()
```

x distribution

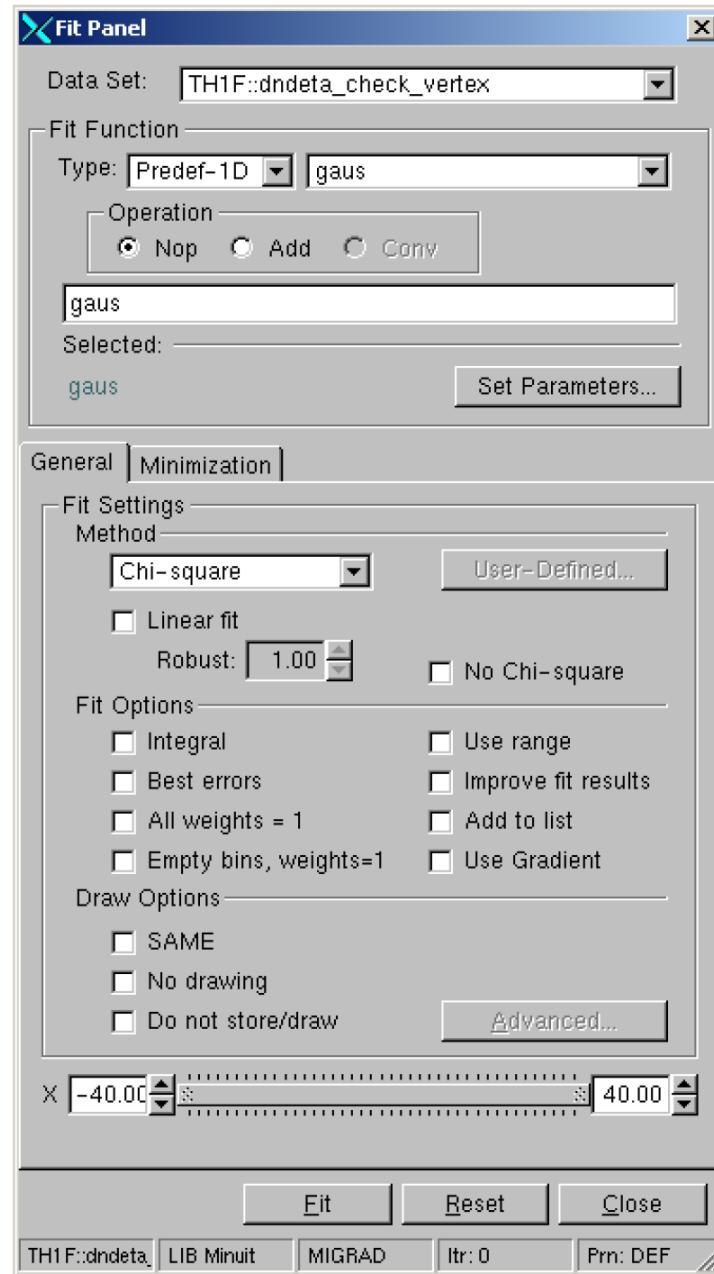
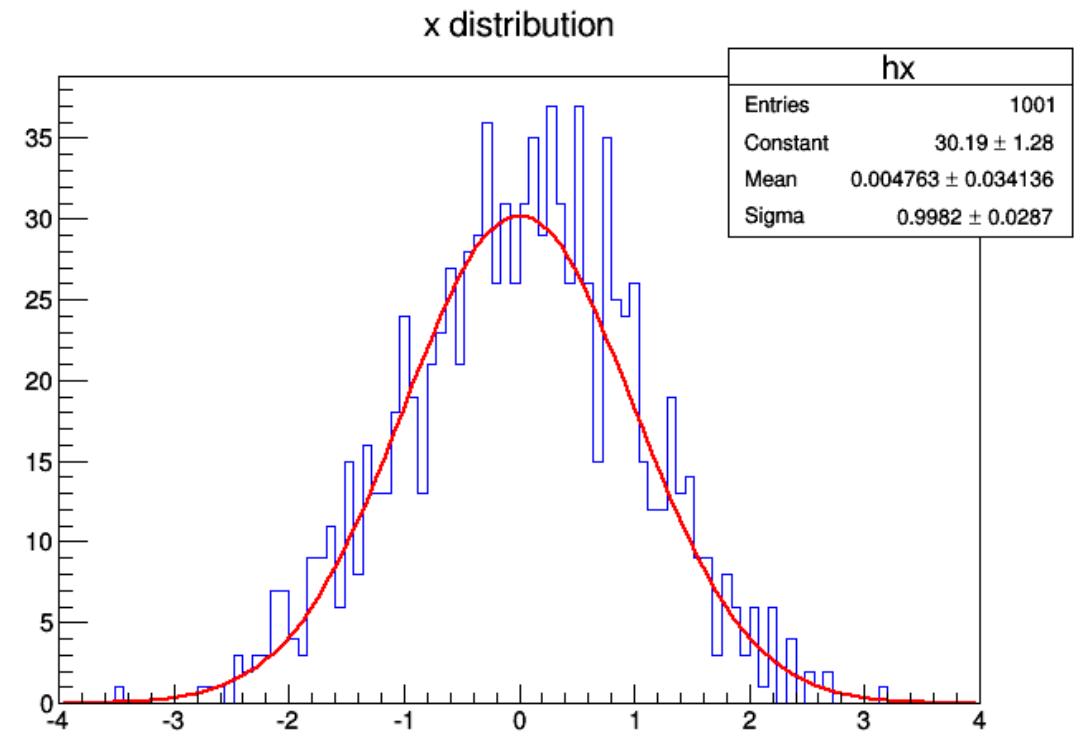


Lorentzian Peak on Quadratic Background



# Fit Panel

- Además de las líneas de comandos y macros, se puede utilizar el **FitPanel** en manera interactiva
- En ROOT se pueden utilizar funciones predefinidas y también definidas por el usuario



# Graphs

- **Graph:** Un graph es un grafico donde se representan los valores de una variable en función de los valores de otra variable. Es. ***scatter plot (x,y)***
- Se pueden crear graphs con o sin errores y también con errores asimétricos, utilizando las clases **TGraph**, **TGraphErrors** y **TGraphAsymmErrors**

**TGraph(n,x,y)**

```
root [ ] TGraph *graph = new TGraph()
```

```
root [ ] graph->SetPoint(i,x,y)
```

**TGraphErrors(n,x,y,dx,dy)**

```
root [ ] TGraphErrors *grapherr = new TGraphErrors()
```

```
root [ ] grapherr->SetPoint(i,x,y)
```

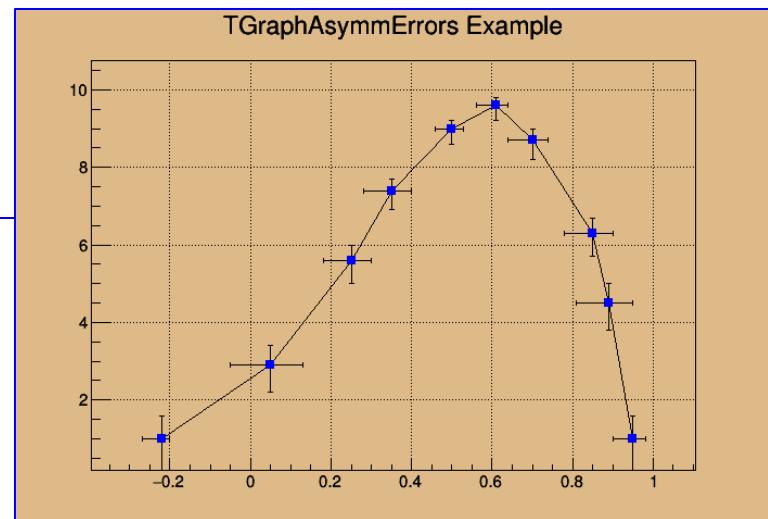
```
root [ ] grapherr->SetPointError(i,dx,dy)
```

**TGraphAsymmErrors(n,x,y,dxL,dxR,dyL,dyR)**

```
root [ ] TGraphAsymmErrors *graphaserr = new TGraphAsymmErrors()
```

```
root [ ] graphaserr->SetPoint(i,x,y)
```

```
root [ ] graphaserr->SetPointError(i,dxL,dxR,dyL,dyR)
```



# Fits de graphs

Fit of a graph

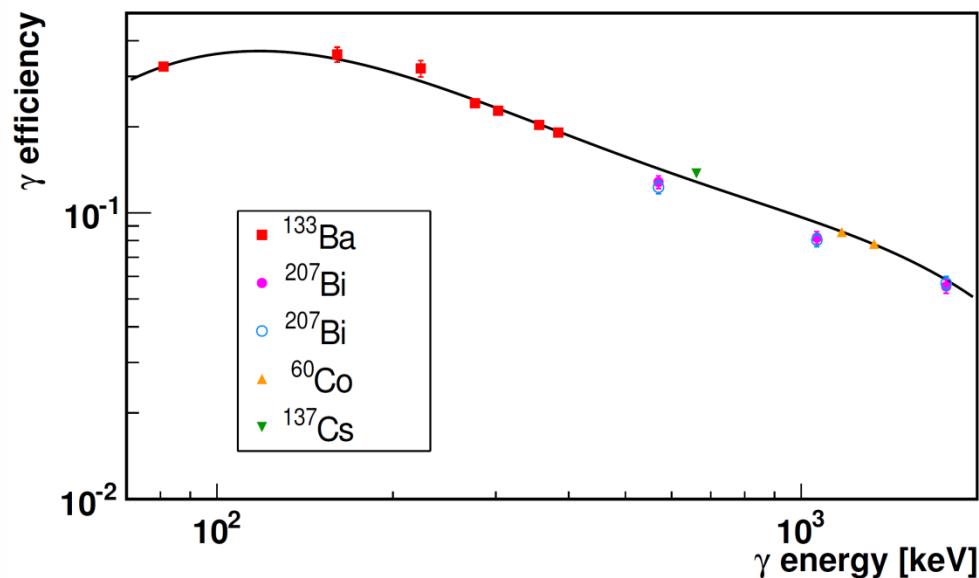
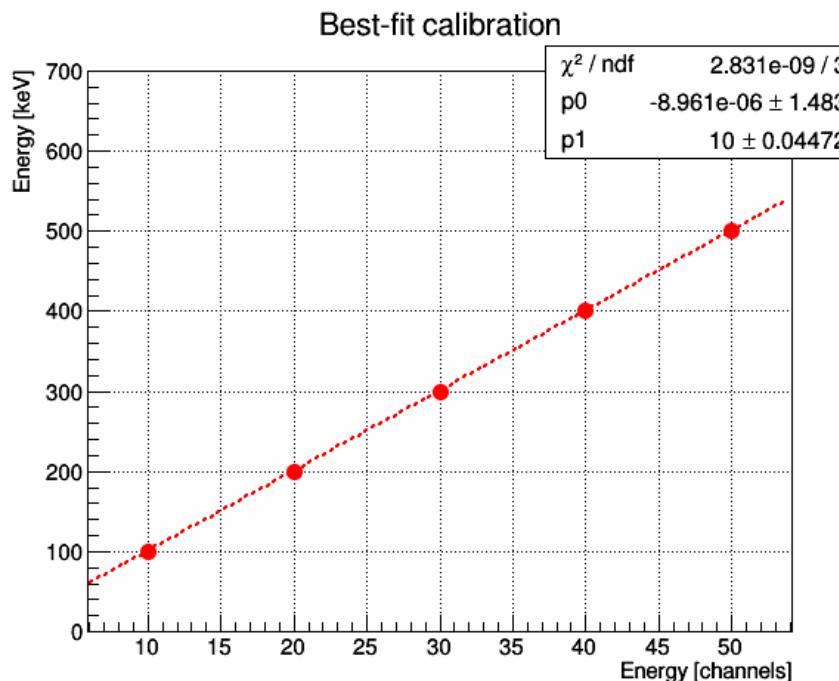
```
root [ ] TF1 *bestfit = new TF1("bestfit","pol1",x1,x2)
```

```
root [ ] GraphErrors->Fit("bestfit","R")
```

Getting  $\chi^2$  and ndf

```
root [ ] bestfit->GetChisquare()
```

```
root [ ] bestfit->GetNDF()
```

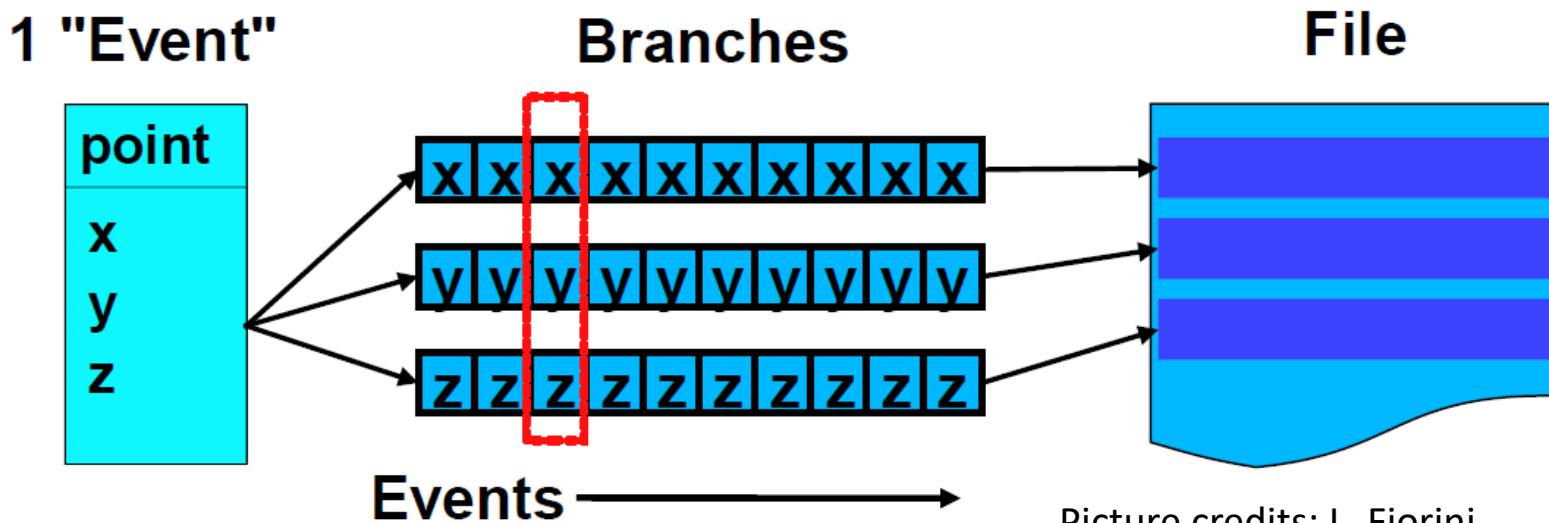


# Trees

- Los resultados de las medidas tomadas durante los experimentos usualmente tienen una estructura como de enormes tablas

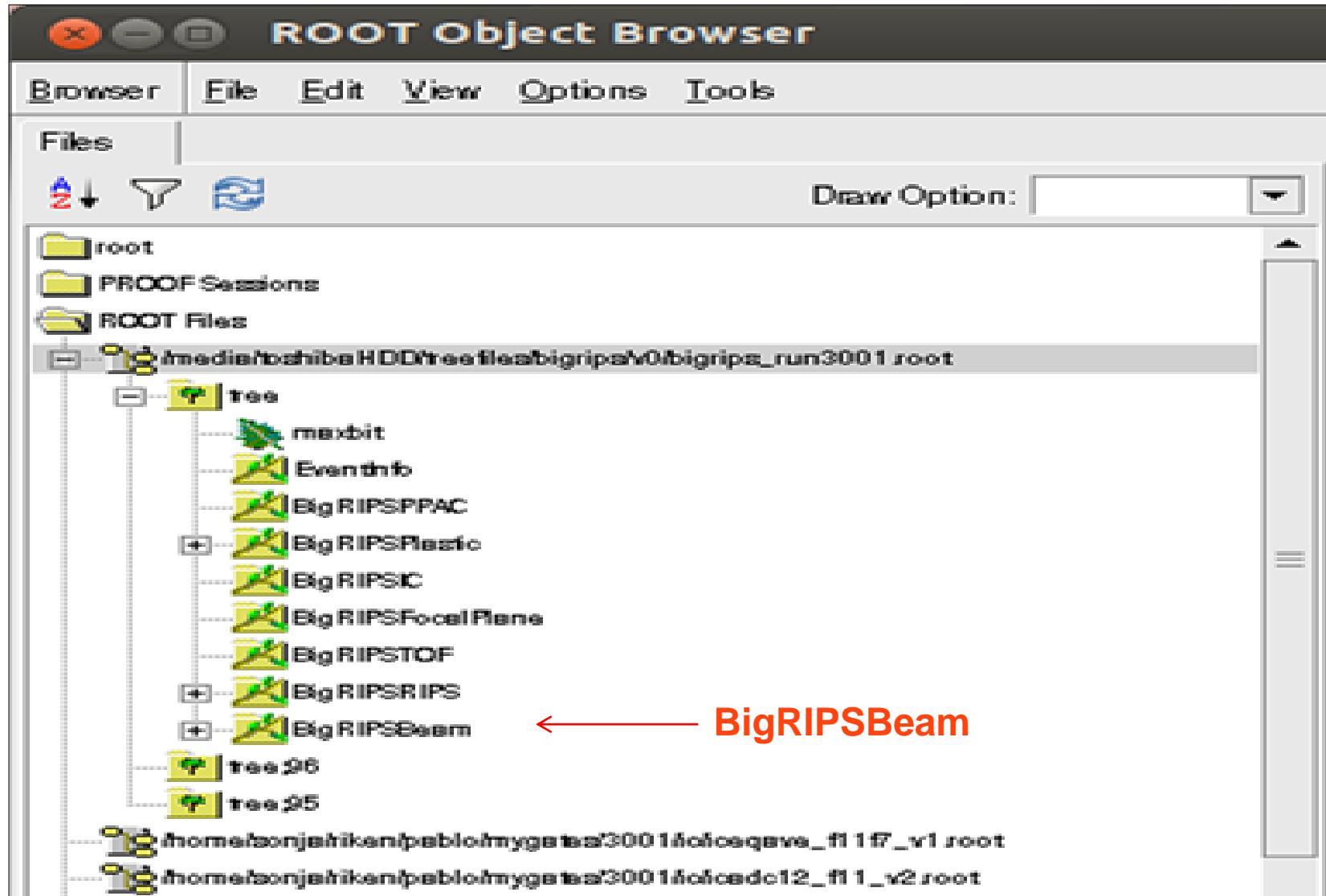
Evento	numero di hit	numero di tracce	proprietà delle tracce	.....
1	10	3	aaa	bbb
2	30	4	ccc	ddd
...	...	...	...	...

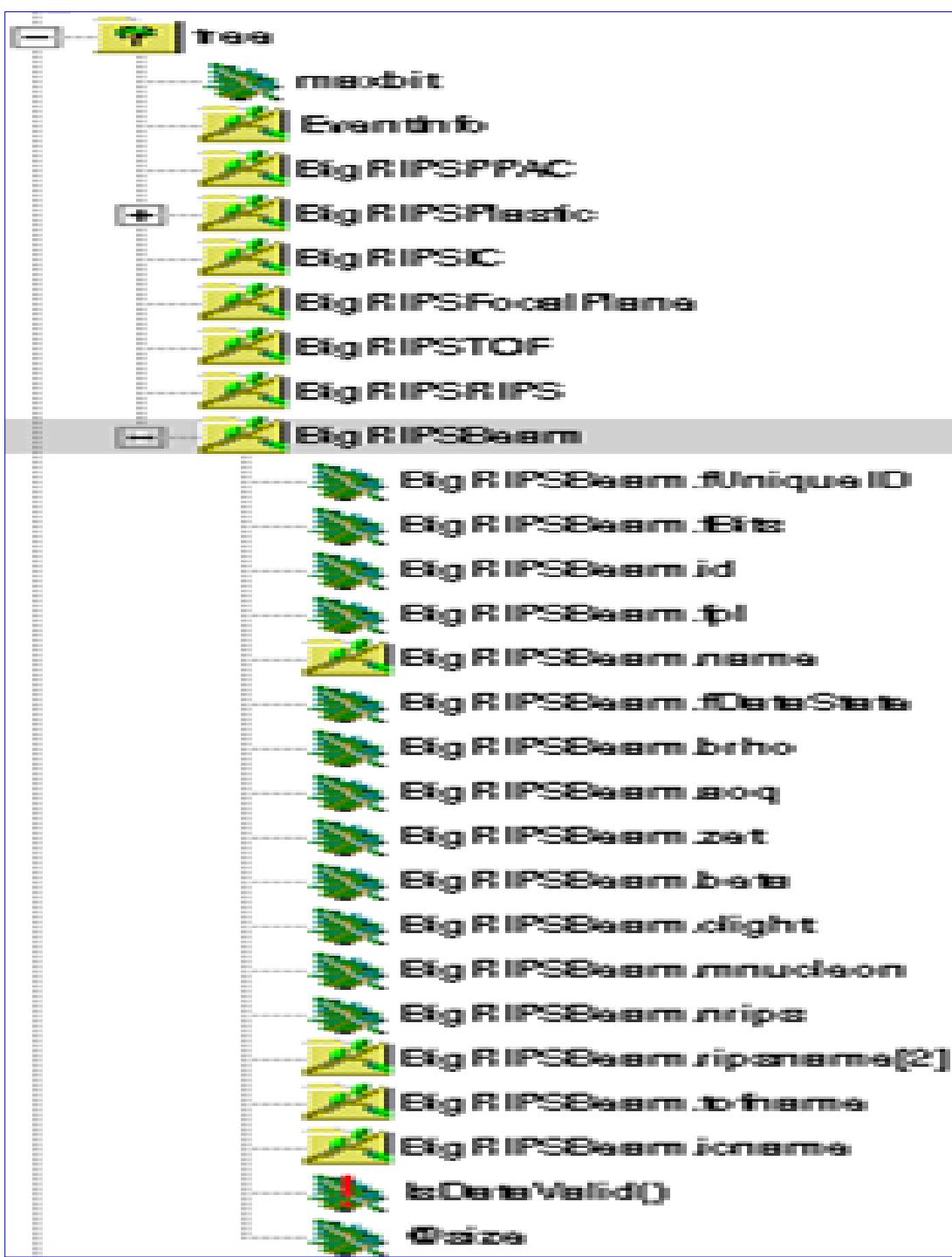
- Trees:** Los trees de ROOT implementan estas tablas “generalizadas”, donde las columnas pueden contener grandes y complejas estructuras de datos (objetos de cualquier clase)
- La clase de ROOT se llama **TTrees**



# Trees

- Los Trees están estructurado en **Branches** y **Leaves**





# Trees

```
root [ ] tree->Draw("variable1", "CONDITION")
```

```
root [ ] tree->Draw("variable1", " variable1>10") 1D-histogram binned automatically
```

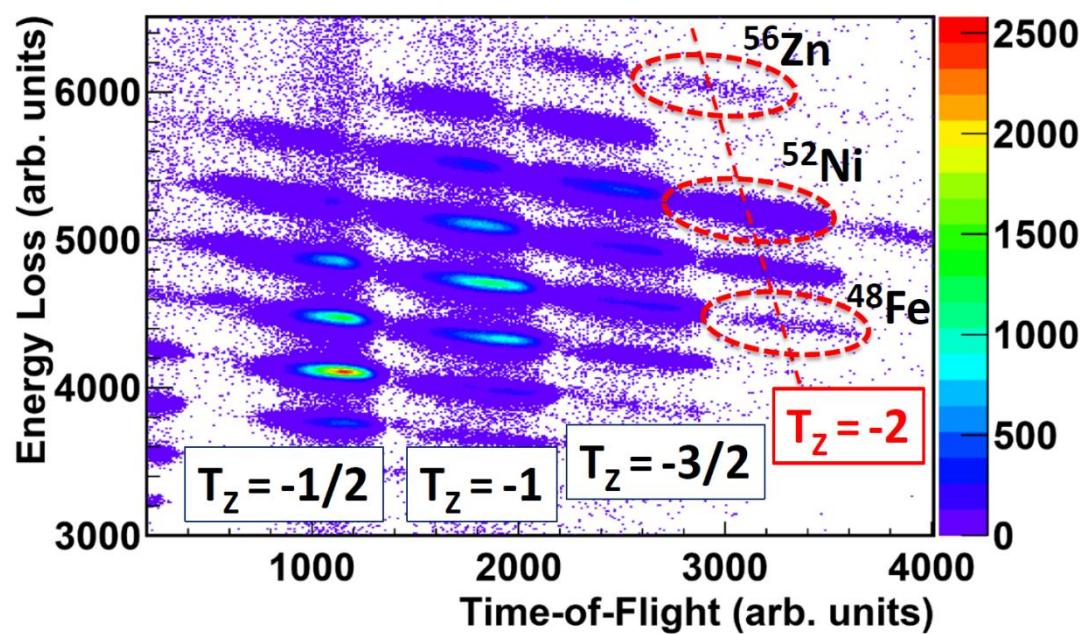
```
root [ ] tree->Draw("variable1>>histo1D(binX,x1,x2)", "")
```

```
root [ ] tree->Draw("variable1>>histo1D(binX,x1,x2)", "variable1>10")
```

```
root [ ] tree->Draw("variable2:variable1", "CONDITION", "OPTION") 2D-histogram
```

```
root [ ] tree->Draw("variable2:variable1", "variable2>0", "colz")
```

```
root [ ] tree->Draw("variable2:variable1>>histo2D(binX,x1,x2,binY,y1,y2)", "", "")
```



# Writing a Tree to file

```
root [ ] TFile *file = TFile::Open("mytree.root","RECREATE");
root [ ] TTree *mytree = new TTree("mytree","title");
root [ ] Float_t var1;
root [ ] tree->Branch("var1", &var1, "var1/F");
root [ ] var1 = 2.16;
root [ ] tree->Fill();
root [ ] tree->Print();
root [ ] tree->Write();
root [ ] file->Close();
```

La modalità di apertura può essere (maiuscole non importano):

- **CREATE**: Crea un nuovo file. Se esiste già un file con lo stesso nome non viene fatto niente.
  - **RECREATE**: Crea un nuovo file. Se esiste già viene un file con lo stesso nome viene cancellato.
  - **UPDATE**: Apre un file esistente per aggiungere dati. Se non esiste, viene creato un nuovo file.
  - **READ**: Apre un file per la lettura.
- 
- Per scrivere dati su un file bisogna usare CREATE, RECREATE o UPDATE.
  - Per leggere dati da file bisogna usare READ o UPDATE.

# Reading a Tree from file

```
root [ ] TFile *file = TFile::Open("mytree.root");
root [ ] TTree *tree = (Ttree*)file->Get("mytree");
root [ ] Float_t var1;
root [ ] tree->SetBranchAddress("var1", &var1);
root [ ] Int_t nentries = tree->GetEntries();
root [ ] for(Int_t i=0;i<nentries;i++) {
    tree->GetEntry(i);
    cout << var1 << endl;
}
```