DEPARTAMENTO DE FÍSICA NUCLEAR E ALTAS ENERGIAS

ROOT

Part 1

Content

ROOT part1

- Introduction
- ROOT prompt and Macros
- Some C++
- Histograms, Graphs and Functions
- ROOT TFile and TTree
- Fit and TMVA
- Extras
  - Get ROOT
  - A little about C++

ROOT part2

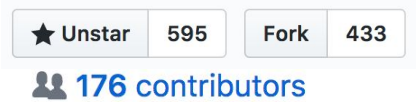
- Python Interface
- ROOTBooks
- Working with files
- Working with Columnar data

Introduction

ROOT

ROOT can be seen as a collection of building blocks for various activities, like:

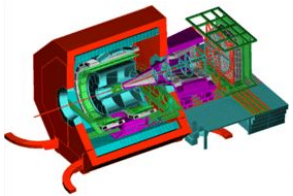
- ★ **Data analysis: histograms, graphs, functions**
- ★ **I/O: row-wise, column-wise** storage of any C++ object
- ★ **Statistical tools** (RooFit/RooStats): rich modeling and statistical inference
- ★ Math: **non-trivial functions** (e.g. Erf, Bessel), optimised math functions
- ★ **C++ interpretation**: full language compliance
- ★ **Multivariate Analysis** (TMVA): e.g. Boosted decision trees, Neural Nets
- ★ **Advanced graphics** (2D, 3D, event display)
- ★ **Declarative Analysis**: RDataFrame
- ★ And more: HTTP servering, JavaScript visualisation

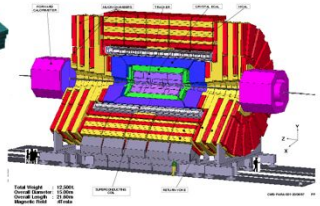


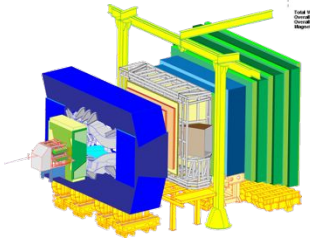
https://github.com/root-project/root

ROOT applications

A selection of the experiments adopting ROOT







Event Filtering

Data

Analysis

Offline Processing

Event Selection, statistical treatment …

Further processing, skimming

Reconstruction

Raw

Reco

…

Analysis Formats

Plots

Data Storage: Local, Network

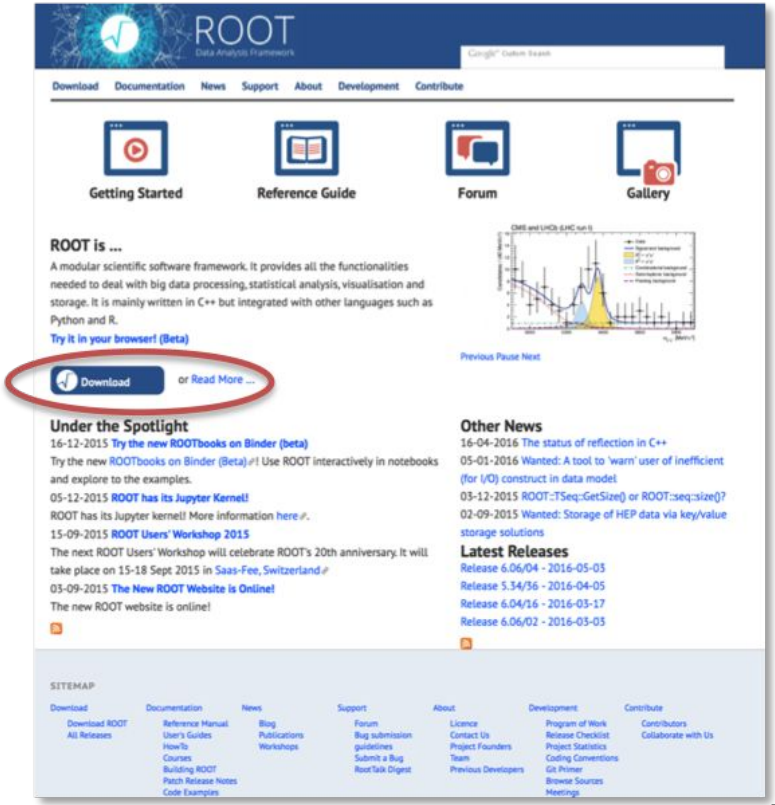
LHC data in ROOT format

~1 EB

as of 2019

https://root.cern

- ROOT web site: **the** source of information and help for ROOT users
  - For beginners and experts
  - Downloads, installation instructions
  - Documentation of all ROOT classes
  - Manuals, tutorials, presentations
  - Forum
  - …

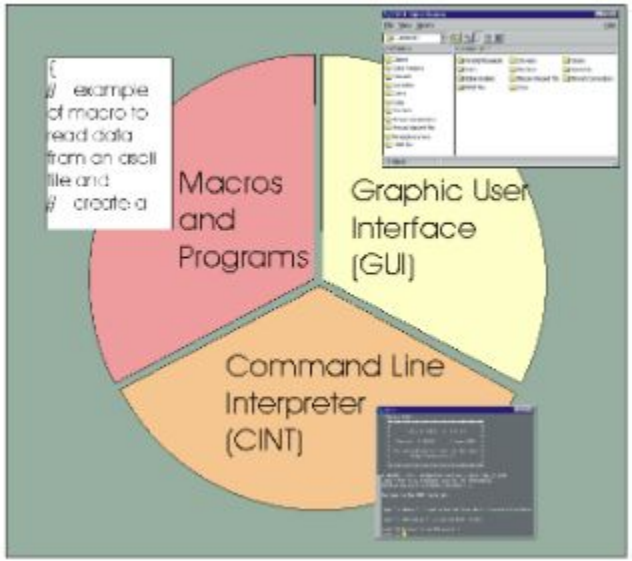


Resources

- ROOT Website: https://root.cern
- Training: https://github.com/root-project/training
- More material: https://root.cern/getting-started
  - Includes a booklet for beginners: **the “ROOT Primer”**
- Reference Guide: https://root.cern/doc/master/index.html
- Forum: https://root-forum.cern.ch

ROOT prompt and Macros

User Interfaces




```
sheilamarass — root.exe — 80×24
sheilamarass@amaral:~ $ export ROOTSYS=root/
sheilamarass@amaral:~ $ export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
sheilamarass@amaral:~ $ export PATH=$ROOTSYS/bin:$PATH
sheilamarass@amaral:~ $ root
   ****************************************
   *                                      *
   *       W E L C O M E  to  R O O T      *
   *                                      *
   *   Version   5.34/36      5 April 2016  *
   *                                      *
   *  You are welcome to visit our Web site *
   *          http://root.cern.ch         *
   *                                      *
   ****************************************

ROOT 5.34/36 (v5-34-36@v5-34-36, Apr 05 2016, 10:25:45 on macosx64)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
```

.q	Quit
.L macro.C	Load a macro file
.x macro.C	Load and execute macro file
.x macro.C++	Compile and execute

ROOT interfaces on Jupyter notebook

- ★ ROOT is well integrated with Jupyter Notebook, both for what concerns its Python and C++ interface
- ★ What is Jupyter Notebook? https://jupyter.org/
  - Language of choice, share notebooks, interactive output, big data integration
- ★ **How to integrate Jupyter notebook and ROOT:**
  - Install ROOT6 (> 6.05)
  - Install dependencies: pip install jupyter metakernel
  - Set up the ROOT environment (`. $ROOTSYS/bin/thisroot.[c]sh`) and then type in your shell:

```
root --notebook
```

The ROOT prompt

- C++ is a compiled language
  - A compiler is used to translate source code into machine instructions
- ROOT provides a C++ **interpreter**
  - Interactive C++, without the need of a compiler, like Python, Ruby, Haskell …
    - Code is **Just-in-Time compiled!**
  - Allows reflection (inspect layout of classes at runtime)
  - Is started with the command:

```
root
```

  - The interactive shell is also called “ROOT prompt” or “ROOT interactive prompt”

ROOT prompt

ROOT options:

```
(my-root) sheilamarass@amaral:~ $ root -?

usage: root [-b B] [-x X] [-e E] [-n N] [-t T] [-q Q] [-l L] [-a A]
            [-config CONFIG] [-memstat MEMSTAT] [-h HELP] [--version VERSION]
            [--notebook NOTEBOOK] [--web WEB] [--web=<browser> WEB=<BROWSER>]
            [dir] [file:data.root] [file1.C...fileN.C]

OPTIONS:
  -b                        Run in batch mode without graphics
  -x                        Exit on exceptions
  -e                        Execute the command passed between single quotes
  -n                        Do not execute logon and logoff macros as specified in .rootrc
  -t                        Enable thread-safety and implicit multi-threading (IMT)
  -q                        Exit after processing command line macro files
  -l                        Do not show the ROOT banner
  -a                        Show the ROOT splash screen
  -config                   print ./configure options
  -memstat                  run with memory usage monitoring
  -h, -?, --help            Show summary of options
  --version                 Show the ROOT version
  --notebook                Execute ROOT notebook
  --web                     Display graphics in a default web browser
  --web=<browser>           Display graphics in specified web browser
  [dir]                     if dir is a valid directory cd to it before executing
  [file:data.root]          Open the ROOT file data.root
  [file1.C...fileN.C]       Execute the the ROOT macro file1.C ... fileN.C
```

Controlling ROOT

- Special commands which are not C++ can be typed at the prompt, they start with a “.”

```
root [1] .<command>
```

- For example:
  - To quit root use **.q**
  - To issue a shell command use **.! <OS_command>**
  - To load a macro use **.L <file_name>** (see following slides about macros)
  - **.help** or **.?** gives the full list

ROOT as a calculator

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \ldots = \sum_{n=0}^{\infty} x^n$$

Here we make a step forward.
We declare **variables** and use a ***for*** control structure.

```
root [0] double x=.5
(double) 0.5
root [1] int N=30
(int) 30
root [2] double gs=0;
```

```
root [3] for (int i=0;i<N;++i) gs += pow(x,i)
root [4] std::abs(gs - (1/(1-x)))
(Double_t) 1.86265e-09
```

Interactivity

```
root [0] #include "a.h"
root [1] A o("ThisName"); o.printName()
ThisName
root [1] dummy()
(int) 42
```

a.h

```
# include <iostream>
class A {
public:
  A(const char* n) : m_name(n) {}
  void printName() { std::cout << m_name << std::endl; }
private:
  const std::string m_name;
};

int dummy() { return 42; }
```

ROOT macros

- ★ We have seen how to interactively type lines at the prompt
- ★ The next step is to write “ROOT Macros” – lightweight programs
- ★ The general structure for a macro stored in file *MacroName.C* is:

Function, no main, same name as the file

```
void MacroName() {
         <           ...
           your lines of C++ code
                     ...              >
}
```

Unnamed ROOT macros

- ★ Macros can also be defined with no name
- ★ Cannot be called as functions!
  - See next slide :)

```
{
        <           ...
          your lines of C++ code
                    ...             >
}
```

Running a macro

★ A macro is executed at the system prompt by typing:

```
> root MacroName.C
```

★ or executed at the ROOT prompt using .x:

```
> root
root [0] .x MacroName.C
```

★ or it can be loaded into a ROOT session and then be run by typing:

```
root [0] .L MacroName.C
root [1] MacroName();
```

Interpretation and Compilation

We have seen how ROOT interprets and “just in time compiles” code. ROOT also allows to compile code “traditionally”. At the ROOT prompt:

```
root [1] .L macro1.C+
root [2] macro1()
```

Generate shared library and execute function

Advanced Users

```
int main() {
  ExampleMacro();
  return 0;
}
```

```
> g++ -o ExampleMacro ExampleMacro.C `root-config --cflags --libs`
> ./ExampleMacro
```

Conventions

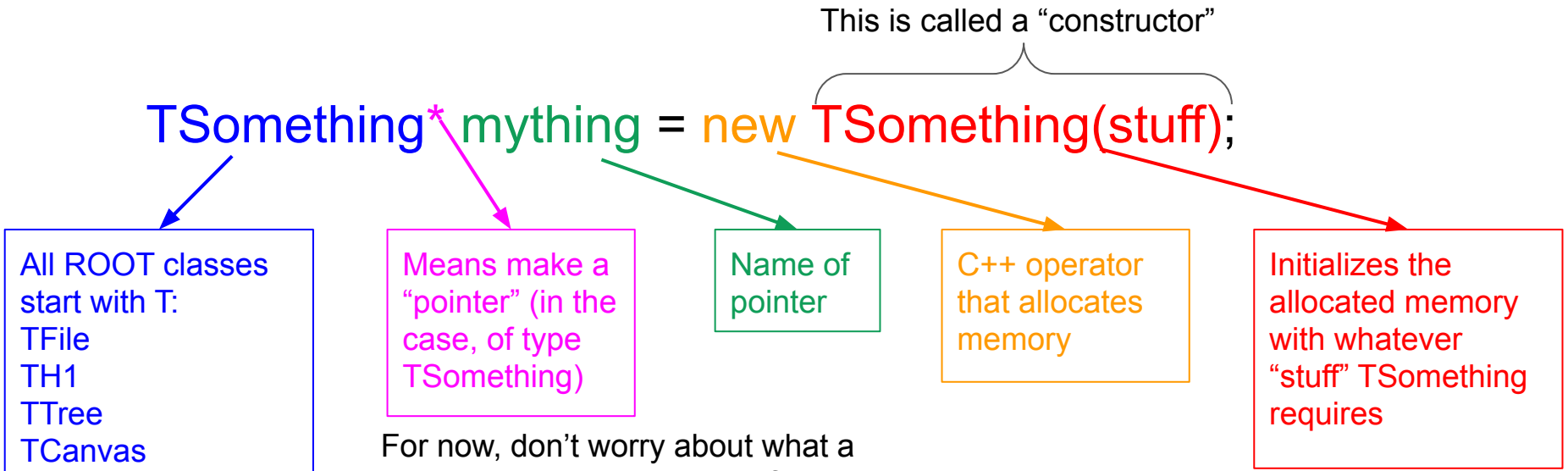
ROOT uses a set of coding conventions:

- ★ Classes begin with T
- ★ Non-class types end with _t
- ★ Member functions begin with a capital
- ★ Constants bekin with k
- ★ Global variables begin with g
- ★ Getters and setters begin with Get and Set
- ★ Predefined types in ROOT:
  - Int_t, Float_t, Double_t, Bool_t, etc
  - You can, however, use also the C++ types: int, double, etc…
- ★ ROOT has a set of global variables that apply to the session
  - For example the single instance of TROOT is accessible via the global gROOT and hold information relative to the current session:

```
gROOT->Reset();
gROOT->LoadMacro(“ftions.cxx”);
gSystem->Load(“libMyEvent.so”)
```

Sintax

Many of the commands we will use will have this general form:



For now, don’t worry about what a pointer is. It’s not important for this tutorial.

Note: In C++, if you you allocate memory using the “new” operator, you must later use “delete mything” to release the memory… otherwise your code will have a memory leak.
We will not worry about that today, but keep it in mind for your future code-writing

Some tips

★ You can use “TAB” key to complete names in ROOT or to get help about the argument of a function

```
root [0] TH1F histo(
TH1F TH1F()
TH1F TH1F(const TH1F& h1f)
TH1F TH1F(const TVectorF& v)
TH1F TH1F(const char* name, const char* title, Int_t nbinsx, Double_t xlow, Double_t xup)
TH1F TH1F(const char* name, const char* title, Int_t nbinsx, const Double_t* xbins)
TH1F TH1F(const char* name, const char* title, Int_t nbinsx, const Float_t* xbins)
root [0] TH1F histo(
```

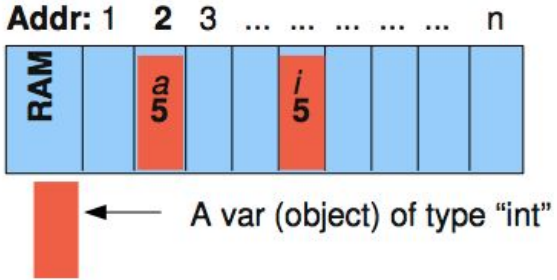
★ The history of your recent commands is kept in a file ~/.root_hist

- sh cat ~/.root_hist

Some C++

Some C++ (pointers, references)

A pointer is a variable that knows where another variable is stored in RAM




```
int a = 5;
int i = a;   // create another object and copy "a"
int * b = &a; //give to b the address of a
int & r = a;  // r is a reference to a
const & c = a;  // c is a const reference to a
```

```
cout << "a value is : " << a << endl;      //  b value is : 5
cout << "b value is : " << b << endl;      //  b  value is : 2
cout << "value pointed by b : " << *b << endl; // value pointed by b : 5

//references are not copies, they refer to the same address
r=6;
cout << a << endl;    // 6
cout << i << endl;    // 5
cout << &r << endl;   // 2

//constant reference cannot be modified (are often used to pass objects
// to functions that should not modify the passed object
cout << c << endl;    // 6
c=7; //this doesn't compile!
```

- the operator * return the value pointed (of a ptr)
- the operator & return the pointer of a variable

Some C++ (allocation, scope of objects)

New object can be created in two ways:

★ Object created by the users with “new” should be deleted by the users with “delete”

```
Object * myobj = new Object;
...
delete myobj;
```

★ Object declared in a block are deleted automatically when they go out of scope

```
{     My2DPoint P_a(2.1,5.4);
      My2DPoint * P_b = new My2DPoint(2.1,5.4);
} // here P_a is deleted, while P_b is not!
```

★ Two common problems

- Memory leaks when P_b is no deleted
- Invalid pointers when the address of a P_a is taken

```
My2DPoint *P_c = &P_a;//cannot be used after P_a is deleted
```

Histograms, Graphs and Functions

TCanvas

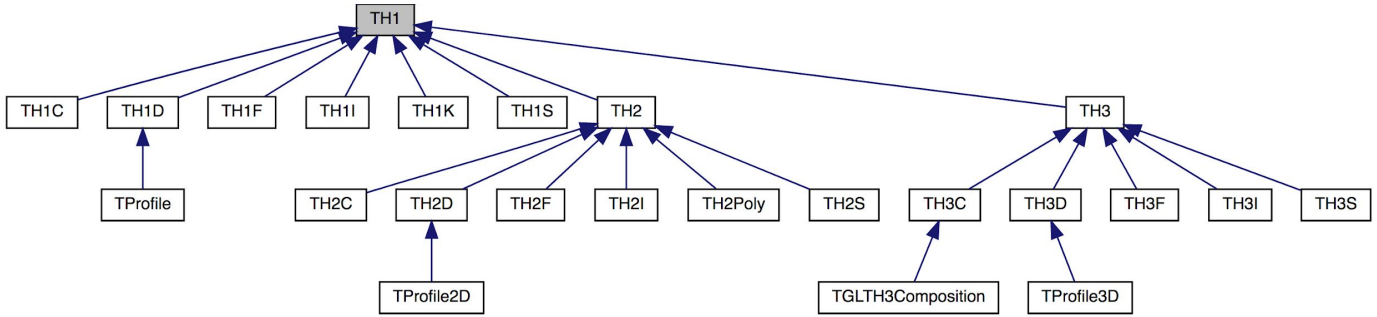
- ★ Canvases may be seen as windows.
- ★ In ROOT a graphical entity that contains graphical objects is called a Pad.

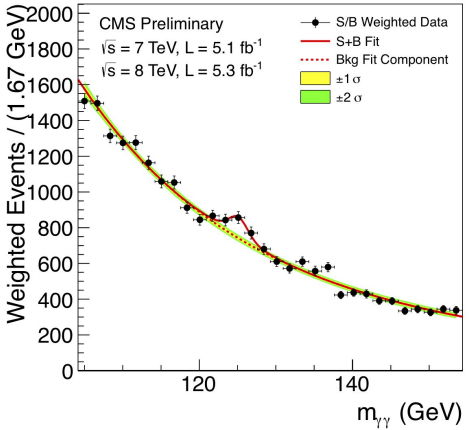
Command	Action
c1 = new TCanvas("c1","Title, w, h)	Creates a new canvas with width equal to w number of pixels and height equal to h number of pixels.
c1->Divide(2,2);	Divides the canvas to 4 pads.
c1->cd(3)	Select the 3rd Pad

c1->SetGridx();
c1->SetGridy();
c1->SetLogy();
	You can set grid along x and y axis.
	
	You can also set log scale plots.

Histograms

- ★ Simplest form of data reduction
  - Can have billions of collisions, the Physics displayed in a few histograms
  - Possible to calculate momenta: mean, rms, skewness, kurtosis …
- ★ Collect quantities in discrete categories, the bins
- ★ ROOT Provides a rich set of histogram types
  - We'll focus on histogram holding a *float* per bin

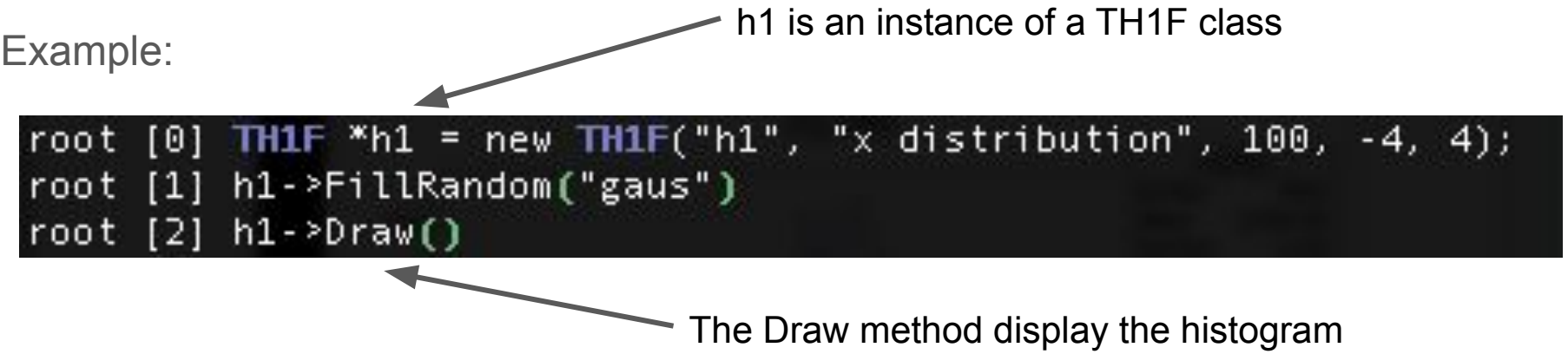




1D Histograms: ROOT

★ 1D histogram: TH1F *name = new TH1F(“name”, “title”, bins, lowest bin, highest bin);

Example:

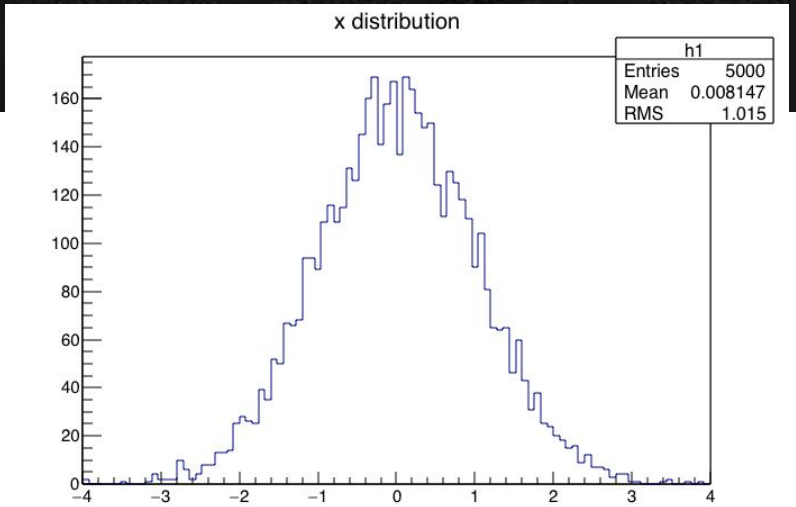


1D Histograms: ROOT

★ 1D histogram: TH1F *name = new TH1F(“name”, “title”, bins, lowest bin, highest bin);

Example:

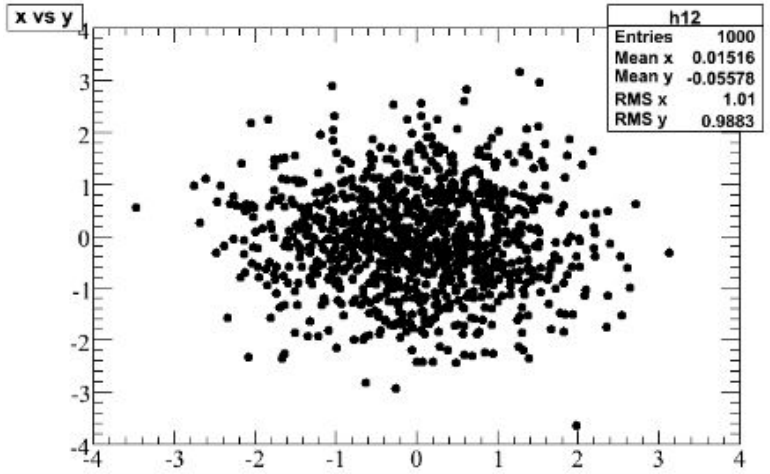
```
root [0] TH1F *h1 = new TH1F("h1", "x distribution", 100, -4, 4);
root [1] h1->FillRandom("gaus")
root [2] h1->Draw()
```



2D Histograms: ROOT

- ★ 2D histogram: TH2F *name = new TH2F(“name”, “title”, xbins, low xbin, up xbin, ybins, low ybin, up2 ybin);
- ★ Example:

```
TH2F *h12 = new TH2F(“h12”, “x vs y”, 100, -4, 4, 100, -4, 4);
h12->Fill(x,y);
h12->Draw();
```

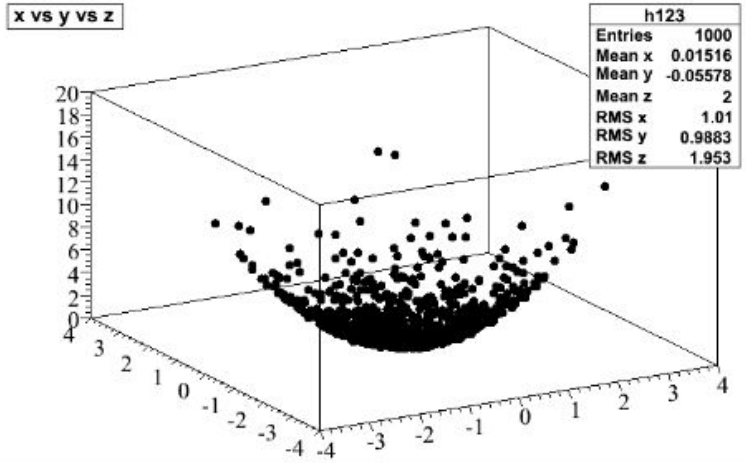


3D Histograms: ROOT

★ 3D histogram: TH3F *name = new TH3F(“name”, “title”, xbins, low xbin, up xbin, ybins, low ybin, up ybin, zbins, low zbin, up zbin);

★ Example:

```
TH3F *h123 = new TH3F(“h123”, “x vs y vs z”, 100, -4, 4, 100, -4, 4, 100, -4, 4);
h123>Fill(x,y,z);
h123->Draw();
```



Creating Histograms

- ★ There are several ways in which you can create a histogram object in ROOT.
- ★ The straightforward method is to use one of the several constructors provided for each concrete class in the histogram hierarchy.
- ★ Histograms may also be created by:
  - Calling the Clone() method of an existing histogram
  - Making a projection from a 2-D or 3-D histogram
  - Reading a histogram from a file

```
// using various constructors
   TH1* h1 = new TH1I("h1", "h1 title", 100, 0.0, 4.0);
   TH2* h2 = new TH2F("h2", "h2 title", 40, 0.0, 2.0, 30, -1.5, 3.5);
   TH3* h3 = new TH3D("h3", "h3 title", 80, 0.0, 1.0, 100, -2.0, 2.0,
                      50, 0.0, 3.0);

   // cloning a histogram
   TH1* hc = (TH1*)h1->Clone();

   // projecting histograms
   // the projections always contain double values !
   TH1* hx = h2->ProjectionX(); // ! TH1D, not TH1F
  TH1* hy = h2->ProjectionY(); // ! TH1D, not TH1F
```

Filling and Drawing Histograms

- The `Fill` method computes the bin number corresponding to the given x, y or z argument and increments this bin by the given weight.

```
h1->Fill(x);
h1->Fill(x,w); // with weight
h2->Fill(x,y);
h2->Fill(x,y,w);
h3->Fill(x,y,z);
h3->Fill(x,y,z,w);
```

- `TH1::FillRandom()` can be used to randomly fill a histogram using the contents of an existing TF1 function or another TH1 histogram (for all dimensions).

```
root[] TH1F h1("h1","Histo from a Gaussian",100,-3,3);
root[] h1.FillRandom("gaus",10000);
```

- When you call the Draw method for the first time, it creates a `THistPainter` object and saves a pointer to painter as a data member of the histogram.
  - You can use the "`SAME`" option to leave the previous display intact and superimpose the new histogram.
  - To create a copy of the histogram when drawing it, you can use `TH1::DrawClone()`.
  - You can use `TH1::DrawNormalized()` to draw a normalized copy of a histogram.

Adding, Dividing and Multiplying Histograms

★ Many types of operations are supported on histograms or between histograms:

- Addition of a histogram to the current histogram
- Additions of two histograms with coefficients and storage into the current histogram
- Multiplications and divisions are supported in the same way as additions.
- The Add, Divide and Multiply methods also exist to add, divide or multiply a histogram by a function.

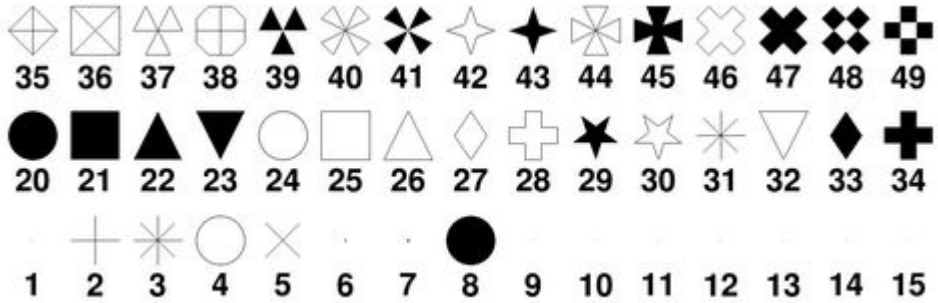
```
h1.Scale(const)
TH1F h3 = 8*h1
TH1F h3 = h1*h2
```

Histograms properties

Command	Parameters
GetMean()	Mean
GetRMS()	Root of Variance
GetMaximum()	Maximum bin content
GetMaximumBin(int bin_number);	Location of maximum
GetBinCenter(int bin_number);	Center of bin
GetBinContent(int bin_number);	Content of bin

Histogram cosmetics

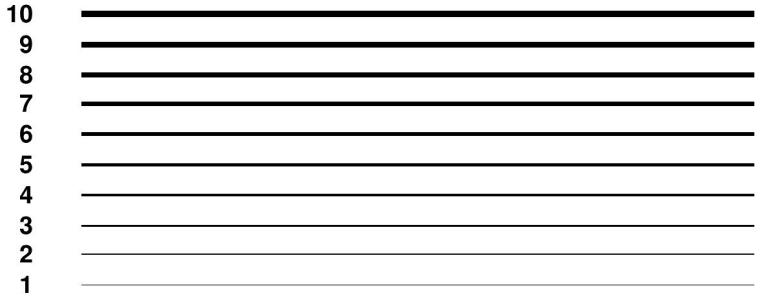
h1.SetMarkerStyle();



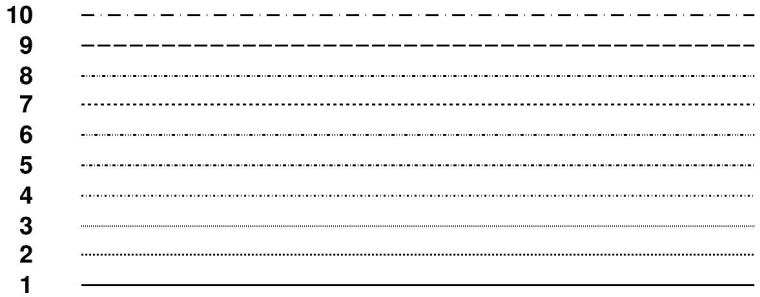
h1.SetFillColor();

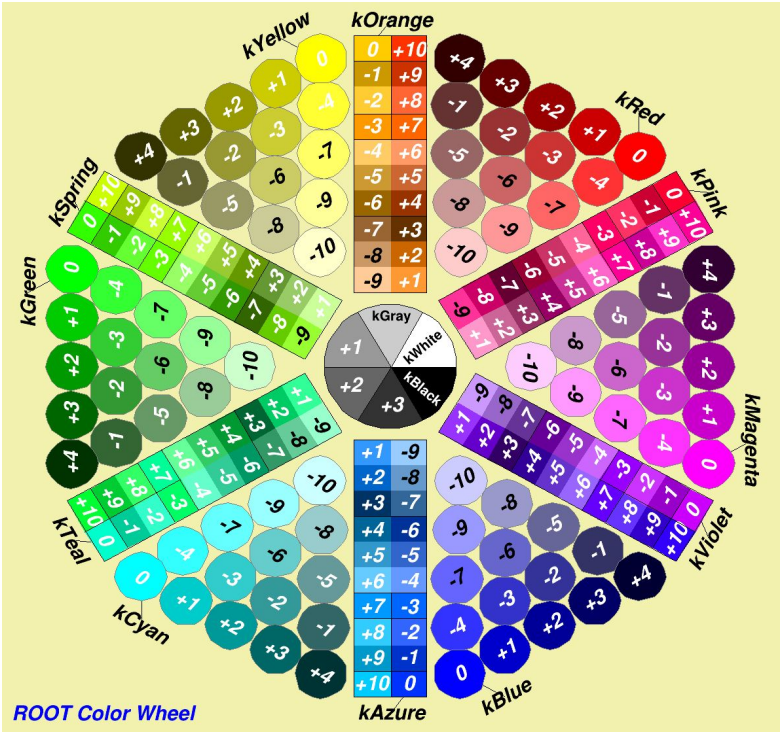
Histogram cosmetics: lines

h1.SetLineWidth();



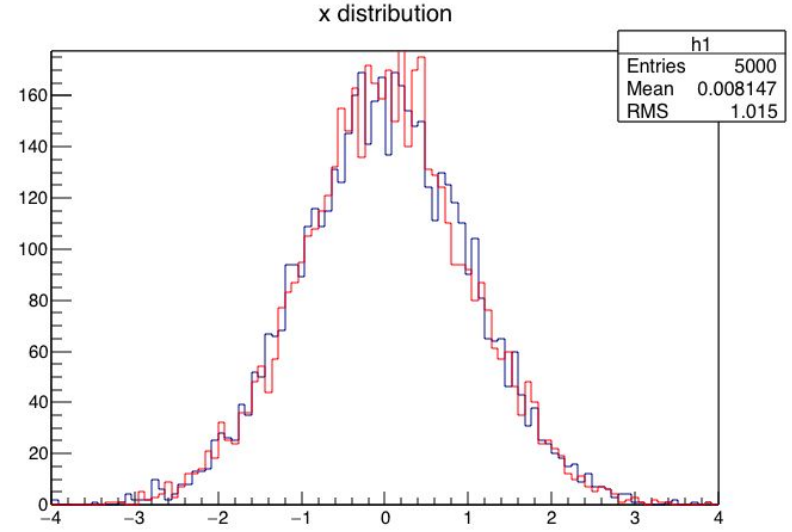
h1.SetLineStyle();





Histogram Drawing Options

```
root [0] TH1F *h1 = new TH1F("h1", "x distribution", 100, -4, 4);
root [1] TH1F *h1same = new TH1F("h1same", "x distribution", 100, -4, 4);
root [2] h1->FillRandom("gaus")
root [3] h1same->FillRandom("gaus")
root [4] h1->Draw()
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [5] h1same->Draw("same")
```

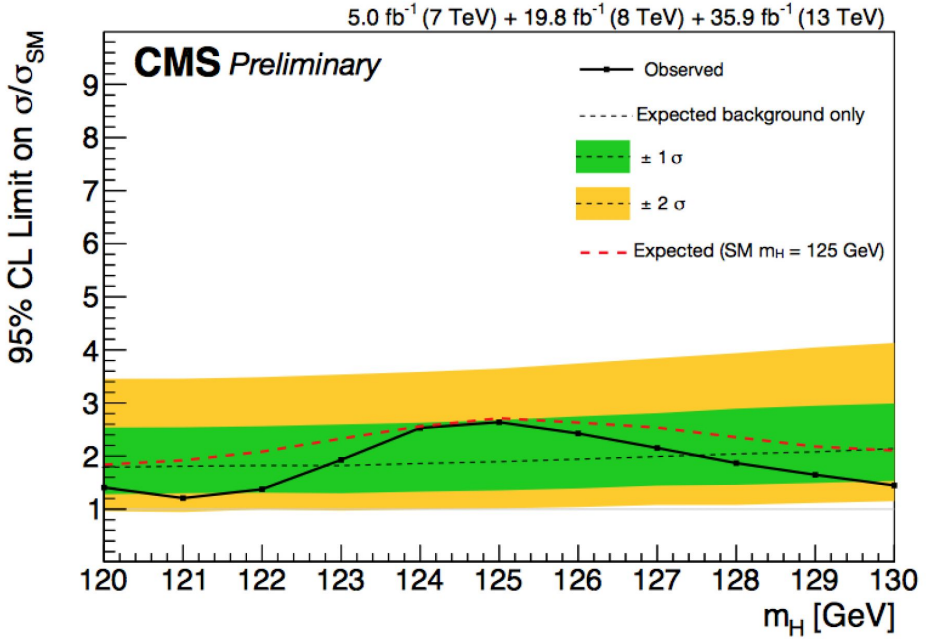


" SAME": Superimpose on previous picture in the same pad.
" CYL": Use cylindrical coordinates.
" POL": Use polar coordinates.
" SPH": Use spherical coordinates.
" PSR": Use pseudo-rapidity/phi coordinates.
" LEGO": Draw a lego plot with hidden line removal.
" LEGO1": Draw a lego plot with hidden surface removal.
" LEGO2": Draw a lego plot using colors to show the cell contents.
" SURF": Draw a surface plot with hidden line removal.
" SURF1": Draw a surface plot with hidden surface removal.
" SURF2": Draw a surface plot using colors to show the cell contents.
" SURF3": Same as SURF with a contour view on the top.
" SURF4": Draw a surface plot using Gouraud shading.
“ SURF5”: Same as SURF3 but only the colored contour is drawn.

Note: Please check chapter 3 in user’s guide to learn more about options.

Graphs

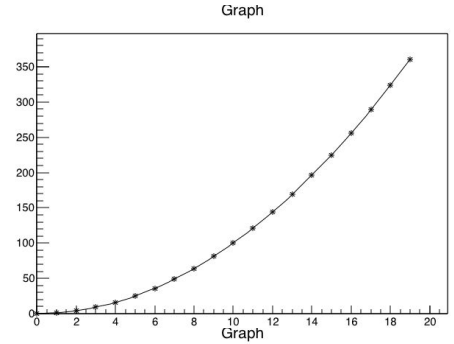
- ★ Graphics object made of two arrays X and Y, holding the x, y coordinates of n points
- ★ Display points and errors
- ★ Not possible to calculate momenta
- ★ Not a data reduction mechanism
- ★ Fundamental to display trends
- ★ Focus on TGraph and TGraphErrors classes in this course



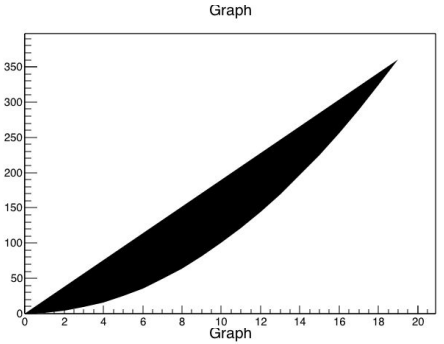
Graph: example

```
root [0] Int_t n=20;
root [1] Double_t x[n], y[n];
root [2] for(Int_t i=0; i<n; i++){ x[i]=i; y[i]=i*i; }
root [3] TGraph *gr1=new TGraph(n,x,y);
root [4] gr1->Draw("AC*");
```

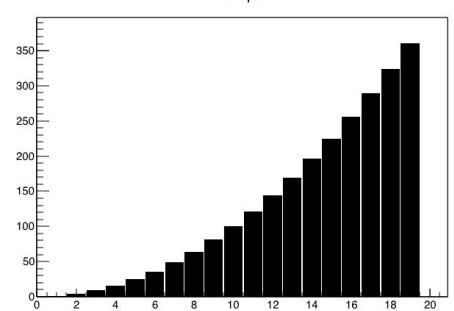
AC*



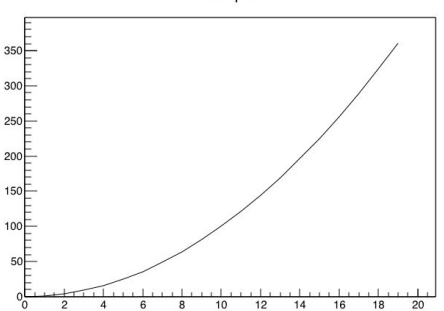
AF



AB

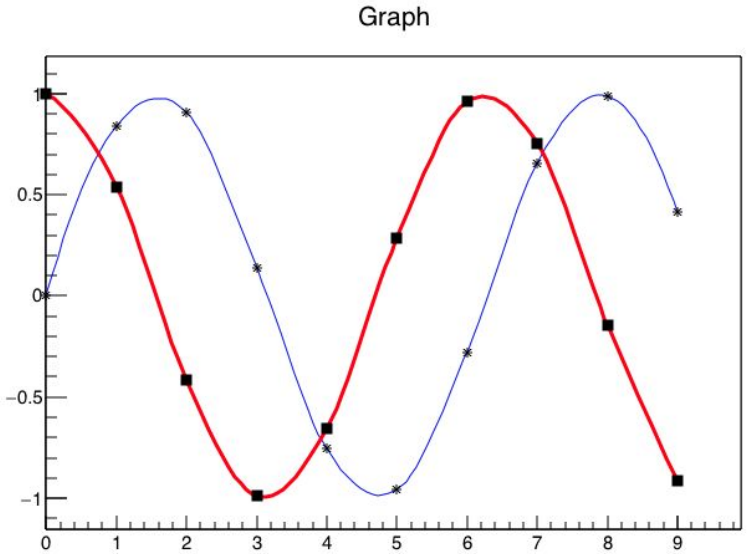


AL



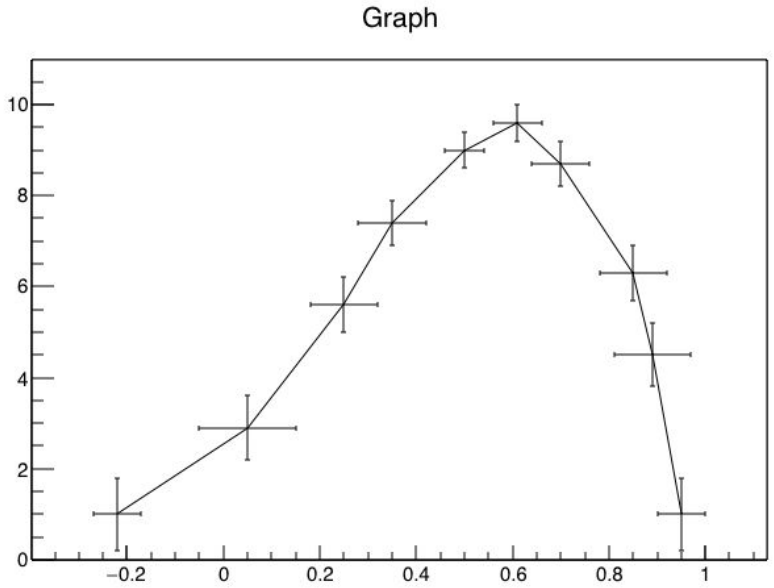
Superimpose two graphs

```
root [0] Int_t n=10;
root [1] Double_t x[n], y[n], x1[n], y1[n];
root [2] for(Int_t i=0; i<n; i++){ x[i]=i; y[i]=sin(i); x1[i]=i; y1[i]=cos(i); }
root [3] TGraph *gr1=new TGraph(n,x,y);
root [4] TGraph *gr2=new TGraph(n,x1,y1);
root [5] gr1->SetLineColor(4);
root [6] gr2->SetLineWidth(3);
root [7] gr2->SetMarkerStyle(21);
root [8] gr2->SetLineColor(2);
root [9] gr1->Draw("AC*");
Info in <TCanvas::MakeDefCanvas>:  created defau
root [10] gr2->Draw("CP");
```



Graph with error bars

```
root [0] Int_t n=10;
root [1] Float_t x[n]={-.22,.05,.25,.35,.5,.61,.7,.85,.89,.95};
root [2] Float_t y[n]={1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};
root [3] Float_t ex[n]={.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
root [4] Float_t ey[n]={.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
root [5] TGraph *gr1=new TGraphErrors(n,x,y,ex,ey);
root [6] gr1->Draw();
```

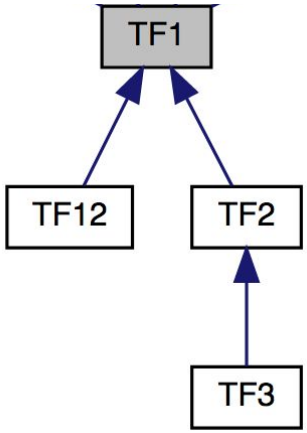


Functions

- ★ Mathematical functions are represented by the **TF1** class
- ★ They have names, formulas, line properties, can be evaluated as well as their integrals and derivatives
  - Numerical techniques for the time being

option	description
"SAME"	superimpose on top of existing picture
"L"	connect all computed points with a straight line
"C"	connect all computed points with a smooth curve
"FC"	draw a fill area below a smooth curve

From the TGraphPainter documentation:
https://root.cern.ch/doc/master/classTGraphPainter.html



Functions

Can describe functions as:

- ★ Formulas (strings)
- ★ C++ functions/functors/lambdas
  - Implement your highly performant custom function
- ★ With and without parameters
  - Crucial for fits and parameter estimation

ROOT as a function plotter

★ The class TF1 represents one-dimensional functions (e.g. f(x)):

Declare a pointer to an object of type TF1. The pointer's name is f1

Name of the function, Can be nearly anything

Define the function using C-style math x - is the evaluation variable

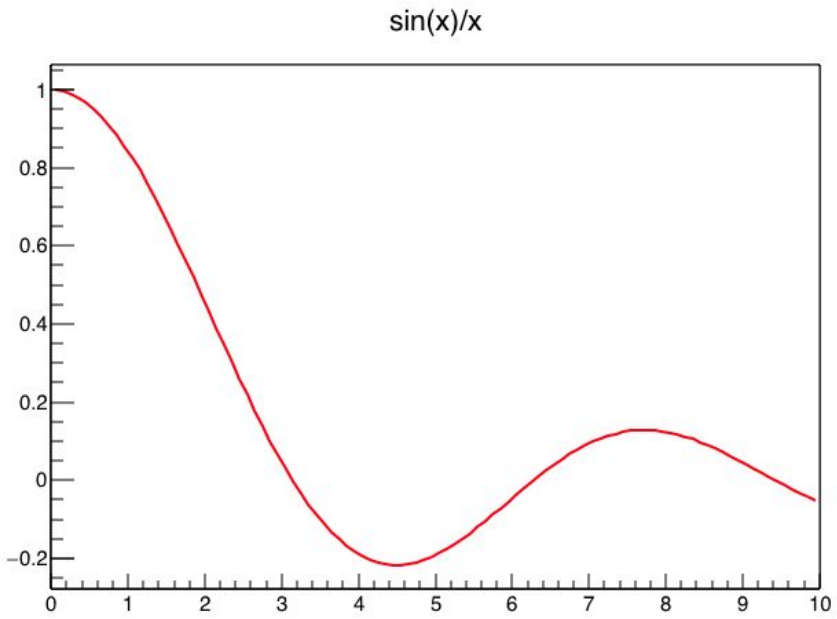
```
root [0] TF1 f1("f1","sin(x)/x",0.,10.); //name, formula, min, max
root [1] f1.Draw();
```

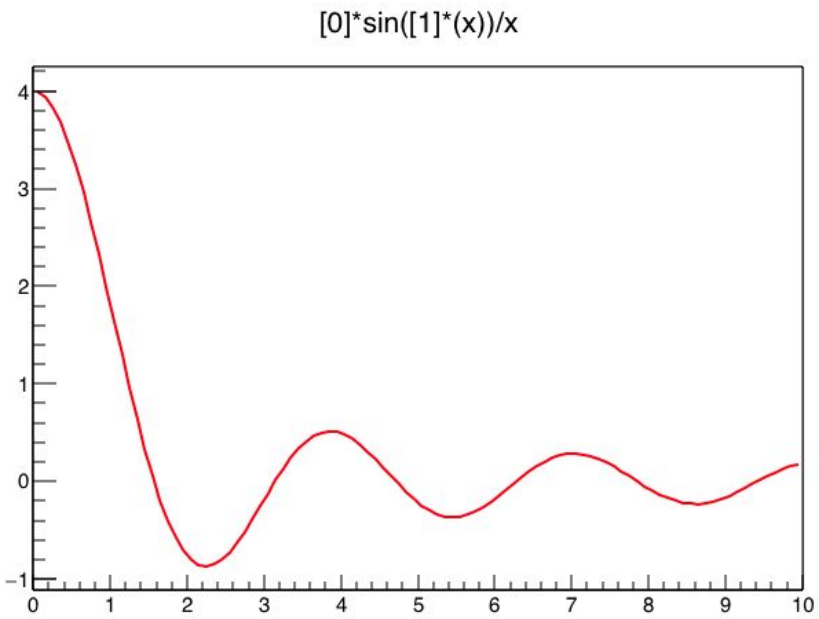
★ An extended version of this example is the definition of a function with parameters:

[0] and [1] - numbers in "[..]" are parameters, and can be set externally.

```
root [1] TF1 f2("f2","[0]*sin([1]*(x))/x",0.,10.);
root [2] f2.SetParameters(2,2);
root [3] f2.Draw();
```

ROOT as a function plotter





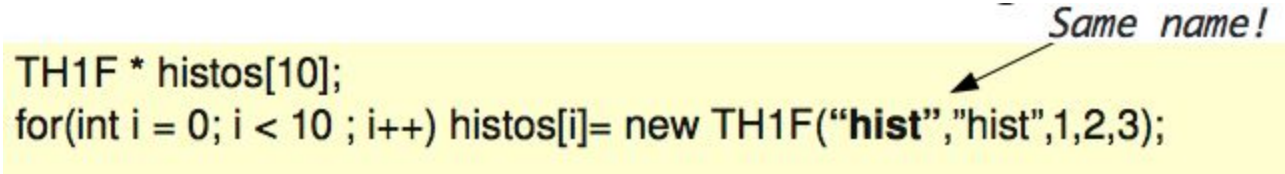
What can we do with functions?

Draw	`f->Draw()`
Print	`f->Print()`
Evaluate	`f->Eval(2.3)`
Integrate	`f->Integral(0.3,1.5)`
Differentiate	`f->Derivative(4.8)`

and many more things… https://root.cern.ch/doc/master/classTF1.html

ROOT memory management

- ★ ROOT objects (Histograms, Canvas, etc) are managed in memory (and disk) by root using “names”
- ★ In the same directory you cannot have two objects with the same name (ROOT will complain about memory leaks)
  - ROOT does not like the following



- ★ Objects member functions can be accessed with “.” (for instance and references) or “->” (for pointers)
  - Interactive ROOT fixes for you wrong usage of pointers vs reference, but when you compile you MUST use the correct syntax

ROOT TFile and TTree

TFile

- ★ Creating ROOT files and Writing objects.
- ★ ROOT stores objects in ROOT files described by TFile

```
TFile* f = new TFile("afile.root",
"NEW")
```

- ★ TFile behaves like file system:

```
f->mkdir("dir");
```

- ★ TFile has a current directory:

```
f->cd("dir");
```

- ★ Opening ROOT files and reading objects

```
TFile* f = new TFile("myfile.root");
TH1F* h = 0;
f->GetObject("h", h);
h->Draw();
delete f;
```

Remember: TFile owns histograms!
File gone, histogram gone!

Options:
“READ” (default): open the file in read mode
“NEW” or “CREATE”: create a new file
“RECREATE”: create a new file, overwrite existing one
“UPDATE”: open a file in update mode

Saving objects in TFile

★ Once the dictionary is available, an object deriving from TObject can be written to the file, with default name

```
root [] f->cd()
root [] object->Write()
```

★ or changing the name to "newName": `root [] object->Write("newName")`

★ Alternative way: `f->WriteObject(object, "name");`

ROOT Command Line: Some Objects

Let's open a file (histograms.root) and see what is inside

Declare a pointer to a TFile object

Create the object, it will point to (i.e. open the file histograms.root)

```
root [0] .ls()
root [1] TFile * input = new TFile("histograms.root");
root [2] input->ls();
TFile**         histograms.root
 TFile*         histograms.root
  KEY: TH1F     GausHist1d;1    One dimensional Gaussian Distribution
  KEY: TH2F     GausHist2d;1    Two dimensional Gaussian Distribution
```

Use the pointer "input" to list its file's contents with the member operator "->" and function "ls()"

There are two histograms in this file. Both are object with properties and functions we can use to display our data

Declare a pointer to a TH1F object

Tell it to point to the object "GausHist1d" stored in our file

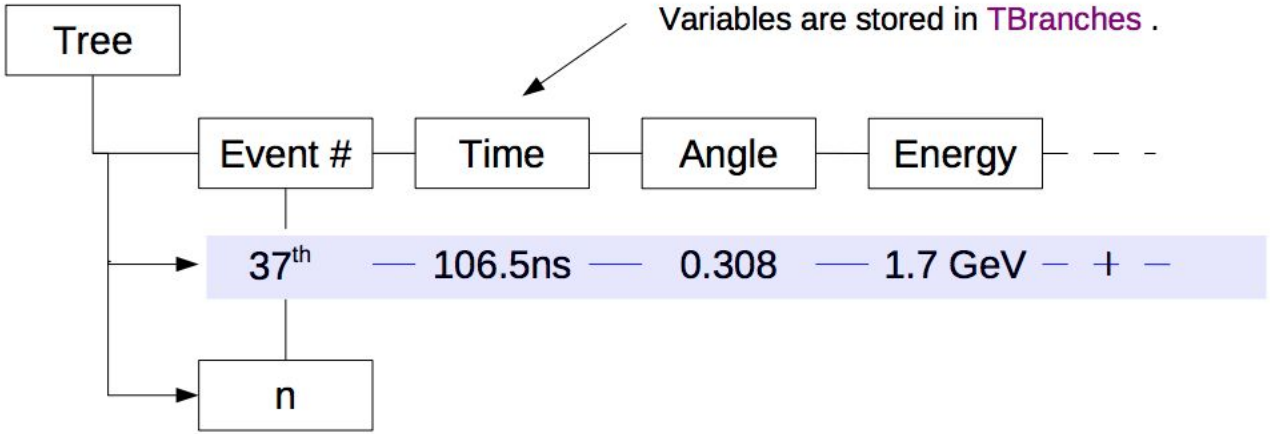
```
root [3] TH1F * my1Dhist = (TH1F*)input->Get("GausHist1d");
root [4] my1Dhist->Draw();
```

Reading data

- ROOT can read data from different sources such as file, network, databases
- In case you want to store large quantities of same-class objects, ROOT designed the `TTree` and `TNtuple`
  - Trees/Ntuples are like “tables”
    - Each row represent usually one “event”
    - Each column is a given quantity (energy, mass, angle, etc…)
  - The TTree class is optimized to reduce disk space and enhance access speed
- A `TNtuple` is a `TTree` that is limited to only hold floating-point numbers, a `TTree` on the other hand can hold all kind of data.
- `TNtuple` and `TTree`
  - Can be read from “ROOT files” in which they are stored
  - Can be created and filled from ASCII file
  - Can be saved by the user

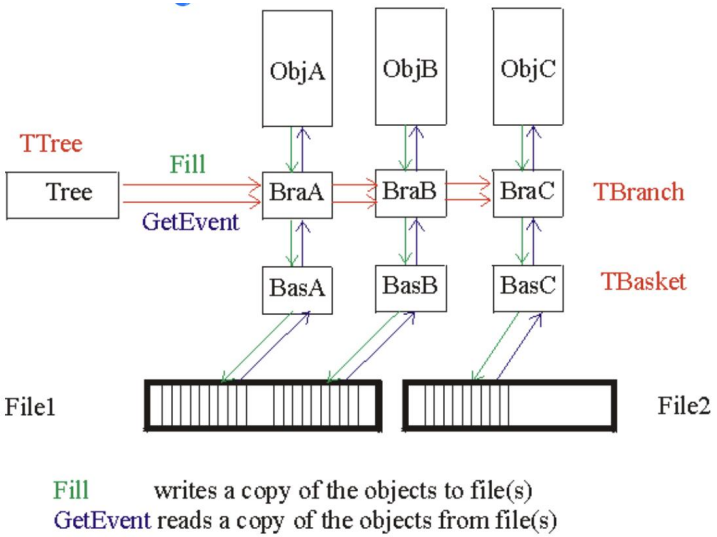
ROOT TTree

- ★ A TTree is a data structure for organizing and manipulating several data variables at once
- ★ Capable of drawing histograms on the fly including making selection cuts on the data
- ★ Uses ROOT's internal compression algorithms to reduce the data size
  - Very useful for data storage



ROOT TBranch

- ★ A TTree is a list of TBranches.
- ★ TTree is the top level bank. It has a set of TBranch objects.
- ★ Each TBranch looks after a single object.
- ★ The user sets up a TTree, creating a TBranch for each object in the event.
- ★ Then I/O commands sent to TTree results in all TBranch objects reading or writing the objects they manage.
- ★ The following diagram shows the basic scheme:



ROOT Command Line: TTree Example

```
Troot [0] TFile *f1 = new TFile("tree.root");
root [1] f1->ls();
TFile**         tree.root
 TFile*         tree.root
  KEY: TTree    tree1;1 Reconstruction ntuple
root [2] TTree *mytree = (TTree *)f1->Get("tree1");
root [3] mytree->Print();
******************************************************************************
*Tree    :tree1     : Reconstruction ntuple                                  *
*Entries :   100000 : Total =         2810073 bytes  File  Size =    2171135 *
*        :          : Tree compression factor =   1.30                       *
******************************************************************************
*Br    0 :event     : event/I                                                *
*Entries :   100000 : Total  Size=     421248 bytes  File Size  =     134514 *
*Baskets :       12 : Basket Size=      32000 bytes  Compression=   2.85     *
*.............................................................................*
*Br    1 :ebeam     : ebeam/F                                                *
*Entries :   100000 : Total  Size=     421248 bytes  File Size  =     260330 *
*Baskets :       12 : Basket Size=      32000 bytes  Compression=   1.47     *
*.............................................................................*
*Br    2 :px        : px/F                                                   *
*Entries :   100000 : Total  Size=     421194 bytes  File Size  =     359238 *
*Baskets :       12 : Basket Size=      32000 bytes  Compression=   1.07     *
*.............................................................................*
*Br    3 :py        : py/F                                                   *
*Entries :   100000 : Total  Size=     421194 bytes  File Size  =     359138 *
*Baskets :       12 : Basket Size=      32000 bytes  Compression=   1.07     *
*.............................................................................*
```

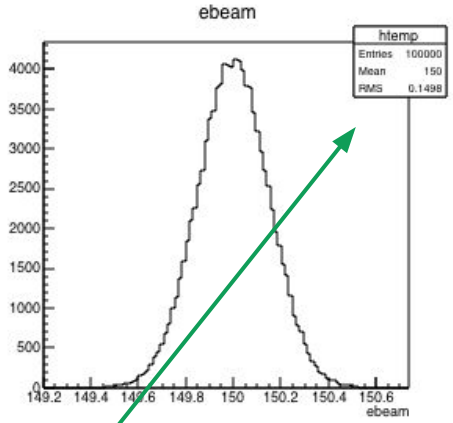
Create pointer to “tree1”

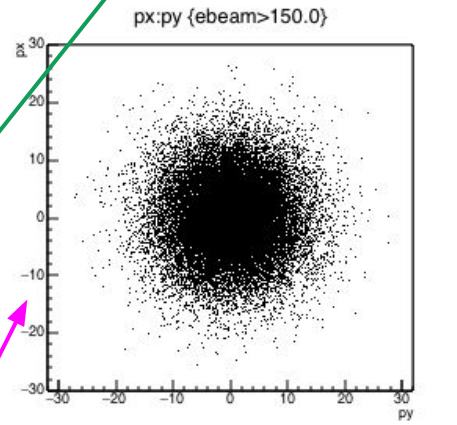
Print structure of tree to screen
This tree contains 7 variables:
`event, ebeam, px, py, pz, zv, chi2`

Turn on statistics box

```
root [4] TCanvas *c2 = new TCanvas("c2", "Tree canvas", 300, 600);
root [5] c2->Divide(1,2);
root [6] c2->cd(1);
root [7] gStyle->SetOptStat(1);
root [8] mytree->Draw("ebeam");
root [9] c2->cd(2);
root [10] mytree->Draw("px:py", "ebeam>150.0");
```

Draw scatter plot (`py` vs `px`) for events with `ebeam` > 150.





Creating and Saving Trees

★ To create a TTree we use its constructor. Then we design our data layout and add the branches. A tree can be created by giving a name and title:

```
TTree t("MyTree","Example Tree");
```

★ The organization of branches allows the designer to optimize the data for the anticipated use.

- If two variables are independent, and the designer knows the variables will not be used together, they should be placed on separate branches.
- A variable on a `TBranch` is called a leaf (`TLeaf`).
- To add a TBranch to a TTree we call the method `TTree::Branch()`

★ Adding a Branch to hold a list of variables: as in the example 2, we saved a list of simple variables, such as integers or floats:

```
tree->Branch("Ev_Branch",&event,"temp/F:ntrack/I:nseg:nvtex:flag/i");
```

Branch name

Address from which the first variable is to be read

Leaf list
<variable>/<type>:<variable>/<type>

Example: A Tree with Simple Variables

```
void tree1w(){

   // create a tree file tree1.root
   //create the file, the Tree and
   // a few branches
   TFile f("tree1.root","recreate");
   TTree t1("t1","a simple Tree with simple variables");
   Float_t px, py, pz;
   Double_t random;
   Int_t ev;
   t1.Branch("px",&px,"px/F");
   t1.Branch("py",&py,"py/F");
   t1.Branch("pz",&pz,"pz/F");
   t1.Branch("ev",&ev,"ev/I");

   // fill the tree
   for (Int_t i=0; i<10000; i++) {
      gRandom->Rannor(px,py);
      pz = px*px + py*py;
      random = gRandom->Rndm();
      ev = i;
      t1.Fill();
   }
   // save the Tree heade;
   // the file will be automatically closed
   // when going out of the function scope
   t1.Write();
}
```

The script creates a ROOT file called tree1.root, the TTree t1

It creates branches with the TTree::Branch method.

```
TBranch* TTree::Branch(const char* name,
                       void* address,
                       const char* leaflist,
                       Int_t bufsize =
32000)
```

The scripts fill the tree. The `px` and `py` is assigned a Gaussian with mean=0 and sigma=1 by calling `gRandom->Rannor(px,py)`, and calculate `pz`

ROOT file is written to disk saving the tree

Example: Inspecting the Tree

- ★ An easy way to access one entry of a tree is the use the TTree::Show method
- ★ A helpful command to see the tree structure meaning the number of entries, the branches and the leaves, is TTree::Print
- ★ The TTree::Scan method shows all values of the list of leaves separated by a colon.

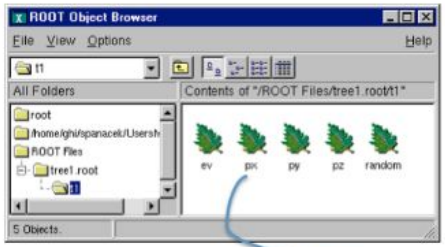
```
root [4] t1->Print()
******************************************************************************
*Tree    :t1        : a simple Tree with simple variables                     *
*Entries :    10000 : Total =          162861 bytes  File  Size =     126098 *
*        :          : Tree compression factor =   1.28                       *
******************************************************************************
*Br    0 :px        : px/F                                                    *
*Entries :    10000 : Total  Size=      40611 bytes  File Size  =      37333 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.08     *
*.............................................................................*
*Br    1 :py        : py/F                                                    *
*Entries :    10000 : Total  Size=      40611 bytes  File Size  =      37310 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.08     *
*.............................................................................*
*Br    2 :pz        : pz/F                                                    *
*Entries :    10000 : Total  Size=      40611 bytes  File Size  =      36663 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.09     *
*.............................................................................*
*Br    3 :ev        : ev/I                                                    *
*Entries :    10000 : Total  Size=      40611 bytes  File Size  =      14155 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   2.84     *
*.............................................................................*
```

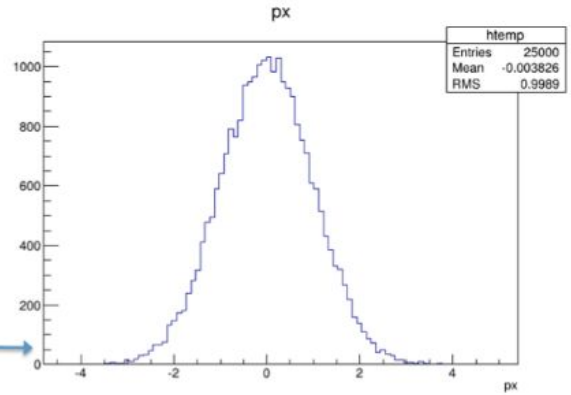
```
root [0] TFile f("tree1.root")
(TFile &) Name: tree1.root Title:
root [1] f.ls()
TFile**         tree1.root
 TFile*         tree1.root
  KEY: TTree    t1;1    a simple Tree with simple variables
root [2] t1->Show(5)
======> EVENT:5
 px             = 0.288987
 py             = -1.86474
 pz             = 3.56078
 ev             = 5
root [3] t1->Scan("px:py:pz:ev")
************************************************************
*    Row   *        px *        py *        pz *        ev *
************************************************************
*        0 * 0.0194094 * 0.0118254 * 0.0005165 *         0 *
*        1 * 0.3271895 * 0.0378782 * 0.1084877 *         1 *
*        2 * -0.294611 * -0.010545 * 0.0869072 *         2 *
*        3 * -0.774589 * 0.0484584 * 0.6023373 *         3 *
*        4 * -1.430795 * 0.5090636 * 2.3063209 *         4 *
*        5 * 0.2889872 * -1.864742 * 3.5607798 *         5 *
*        6 * 1.6896238 * 0.3978982 * 3.0131516 *         6 *
*        7 * 1.3915746 * 0.5094206 * 2.1959893 *         7 *
*        8 * -0.622538 * 0.8379354 * 1.0896899 *         8 *
*        9 * -0.438629 * -1.095200 * 1.3918609 *         9 *
*       10 * 1.0487310 * 1.1538660 * 2.4312439 *        10 *
```

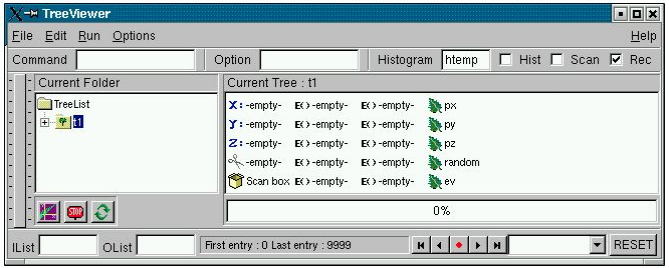
Example: Viewing the Tree

★ The tree viewer is a quick and easy way to examine a tree. To start the tree viewer, open a file and object browser. Right click on a **TTree** and select StartViewer. You can also start the tree viewer from the command line.

```
root[] TFile f("tree1.root")
root[] T->StartViewer()
 Or
root[] gSystem->Load("libTreeViewer.so")
root[] new TTreeViewer()
```







Example: Reading a Tree

```
void tree1r(){
   TFile *f = new TFile("tree1.root");
   TTree *t1 = (TTree*)f->Get("t1");
   Float_t px, py, pz;
   Double_t random;
   Int_t ev;
   t1->SetBranchAddress("px",&px);
   t1->SetBranchAddress("py",&py);
   t1->SetBranchAddress("pz",&pz);
   t1->SetBranchAddress("random",&random);
   t1->SetBranchAddress("ev",&ev);
   // create two histograms
   TH1F *hpx   = new TH1F("hpx","px distribution",100,-3,3);
   TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);
   //read all entries and fill the histograms
   Int_t nentries = (Int_t)t1->GetEntries();
   for (Int_t i=0; i<nentries; i++) {
      t1->GetEntry(i);
      hpx->Fill(px);
      hpxpy->Fill(px,py);
   }
   if (gROOT->IsBatch()) return;
   new TBrowser ();
   t1->StartViewer();
}
```

Define variables to hold the read values

We tell the tree to populate these variables when reading an entry, using the `TTree::SetBranchAddress` method. The first parameter is the branch name, and the second, is the address of the variable where the branch data is to be places

A specific entry can be read into the variables with the method `TTree::GetEntry(n)`. It reads all the branches for entry (n) and populates the given address accordingly

We do not close the file, since we want to keep generated histograms,
Then we open a browser and the TreeViewer

Simple Analysis using TTree::Draw

- ★ Arguments to many functions in ROOT objects are passed by character strings
- ★ Strings are parsed for both logic and mathematics
- ★ For trees:
  - Any variable in the tree can be manipulated as part of an argument

```
root [10] mytree->Draw("px:py", "ebeam>150.0", "lego");
```

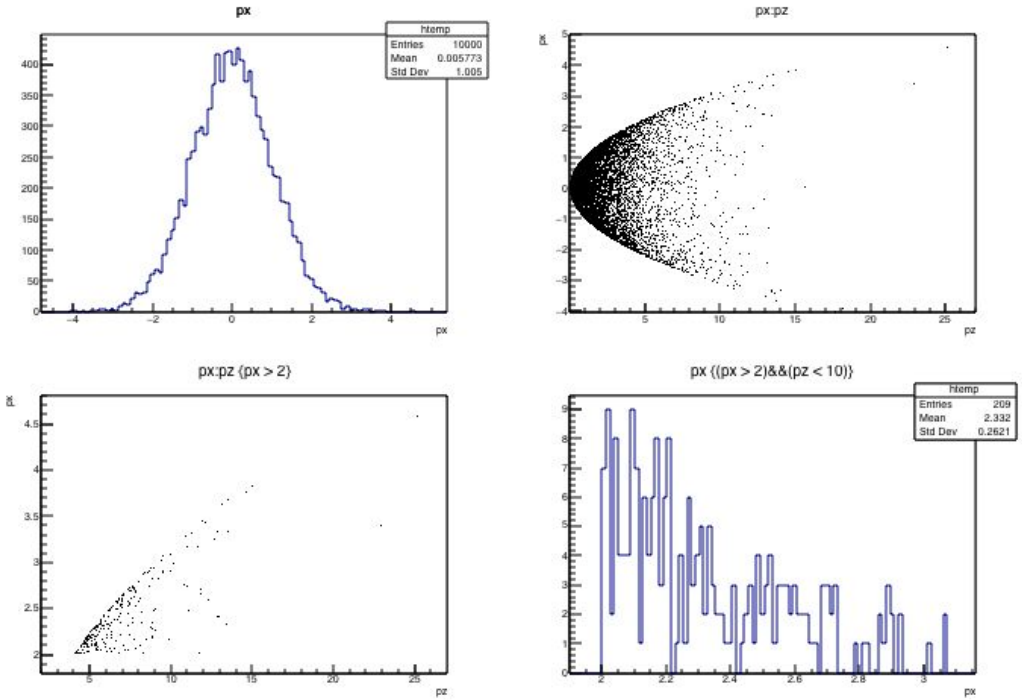
What to draw for each event
- Semicolon ":" indicates adding a new dimension
- Can be functions of variables: e.g. "sqrt(py)"
- Can be combinations of variables: e.g. "ebeam/px : py**2"

Drawing options
Options for n-dimensional histograms go here as in previous example

Selection cuts: i.e. which events or entries to draw
- Multiple cuts are allowed, combined with C-style logic operators
- Can be functions of variables
- Can be combinations of variables

Example: Simple Analysis using TTree::Draw

```
root [0] TFile f("tree1.root")
f.(TFile &) Name: tree1.root Title:
root [1] f.ls()
TFile**         tree1.root
 TFile*         tree1.root
  KEY: TTree    t1;1    a simple Tree with simple variables
root [2] TTree *MyTree = t1
(TTree *) 0x7fe3ad17b780
root [3] TCanvas *myCanvas = new TCanvas()
(TCanvas *) 0x7fe3ad0edbc0
root [4] myCanvas->Divide(2,2)
root [5] myCanvas->cd(1)
(TVirtualPad *) 0x7fe3ad424cf0
root [6] MyTree->Draw("px")
root [7] myCanvas->cd(2)
(TVirtualPad *) 0x7fe3ad4251b0
root [8] MyTree->Draw("px:pz")
root [9] myCanvas->cd(3)
(TVirtualPad *) 0x7fe3ad425fb0
root [10] MyTree->Draw("px:pz","px > 2")
(long long) 235
root [11] myCanvas->cd(4)
(TVirtualPad *) 0x7fe3ad426e60
root [12] TCut c1 = "px > 2"
(TCut &) Name: CUT Title: px > 2
root [13] TCut c2 = "pz < 10"
(TCut &) Name: CUT Title: pz < 10
root [14] TCut c3 = c1 && c2
(TCut &) Name: CUT Title: (px > 2)&&(pz < 10)
root [15] MyTree->Draw("px", c3)
(long long) 209
```



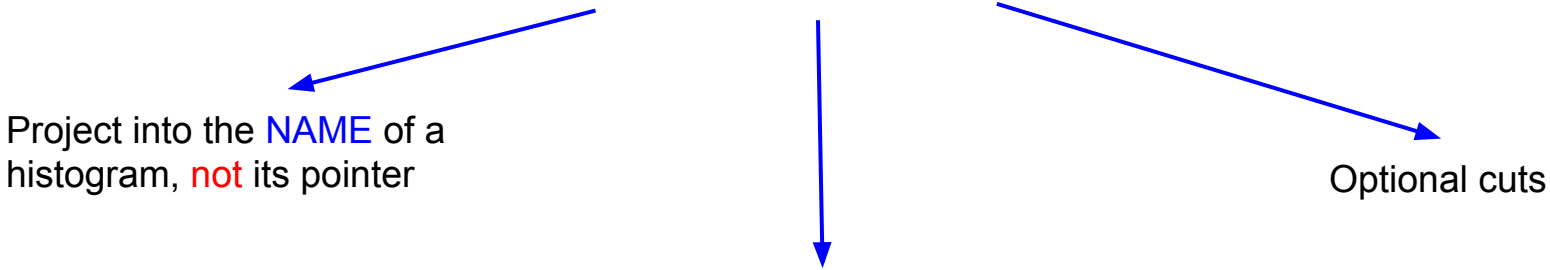
Using TTree to fill a histogram

★ Step 1: Define a histogram with a suitable range

```
root [2] TH1F * h = new TH1F("hBeamEnergy", "Beam Energy", 200, 148.0, 152.0);
```

★ Step 2: Project the TTree contents into the histogram

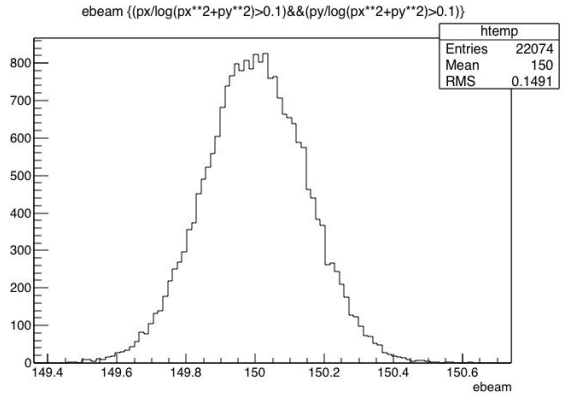
```
root [3] mytree->Project("hBeamEnergy", "ebeam", "px>10.0");
```



Complicated cuts using TCut

- ★ Consider encapsulating your cuts as TCut objects
- ★ TCut objects can be combined using C-style operators as usual
- ★ They can be combined with other string cuts

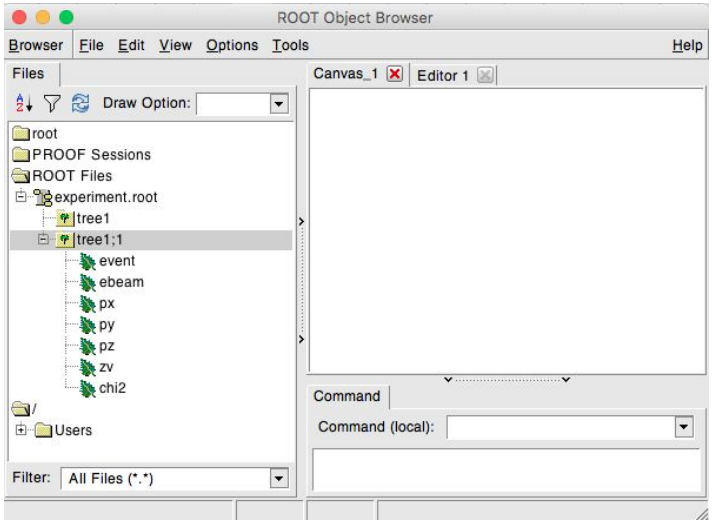
```
root [14] TCut * px_plane = new TCut(“px / log(px*2 + py**2) > 0.10”);
root [15] TCut * py_plane = new TCut(“py / log(px*2 + py**2) > 0.10”);
root [16] mytree->Draw(“ebeam”, *px_plane && *py_plane);
```

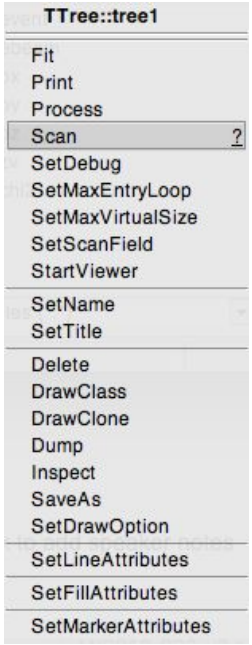


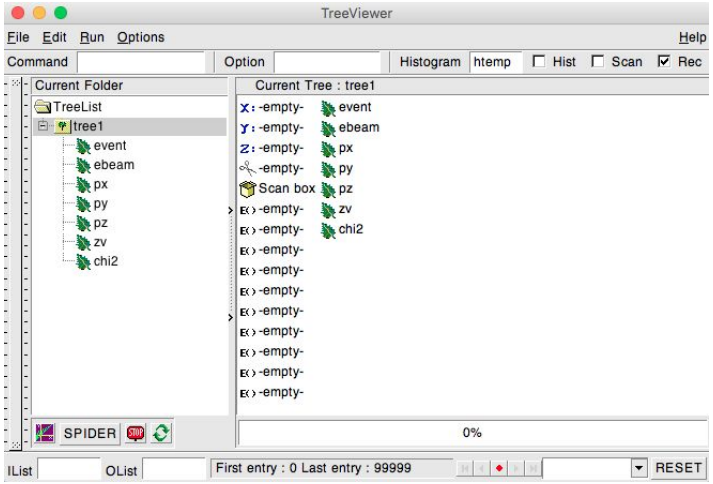
Variables in ROOT NTuples/Trees

```
[] TFile *f = new TFile("experiment.root")
[] TBrowser b
```

Right-click over the Tree:







Using TTree::MakeSelector

- With a TTree we can make a selector and use it to process a limited set of entries

```
root[0] TFile *f = new TFile("experiment.root")
root[1] f->ls()
root[2] tree1->MakeSelector("Analyze")
```

- We can call the `TTree::MakeSelector` method to create a code skeleton
- It implements the following methods:
  - `TSelector::Begin()` - this method is called every time a loop over the tree starts. This is a convenient place to create your histograms
  - `TSelector::Notify()` - it is called at the first entry of a new tree in a chain
  - `TSelector::Process()` - it is called to process an event
  - `TSelector::Terminate()` - it is called at the end of a loop on a TTree. This a convenient place to draw and fit your histograms

Using TTree::MakeSelector

- ROOT provides a utility that generates a skeleton class designed to loop over the entries of the tree (example: from file experiment.root)

```
root[0] TFile *f = new TFile("experiment.root")
root[1] f->ls()
root[2] tree1->MakeSelector("Analyze")
```

- It creates two files: Analyze.h and Analyze.C
  - Definition: define the variables we're going to use
  - Set-up: open file, create histograms, etc.
  - Loop: for each event in the n-tuple or Tree, perform some tasks: calculate values, apply cuts, fill histograms, etc.
  - Wrap-ip: display results, save histograms, etc.

Analyze.h

- The generated code in Analyze.h includes the following:
  - Identification of the original Tree and Input file name
  - Definition of selector class (data and functions)
  - The following class functions:
    - constructor and destructor
    - `void Begin(TTree *tree)` : called every time a loop on the tree starts, a convenient place to create your histograms.
    - `void Init(TTree *tree)` : it is called by the constructor to initialize the tree for reading. It associates each branch with the corresponding leaf data member
    - `Bool_t Notify()` : function called when loading a new class library.
    - `Bool_t Process(Long64_t entry)` : called for each event, in this function you decide what to read and fill your histograms.
    - `void Terminate()` : called at the end of the loop on the tree, a convenient place to draw/fit your histograms.

Analyze.C

★ `Analyze::Process` process this tree executing the TSelector code in the specified filename. The return value is -1 in case of error and TSelector::GetStatus() in case of success.

```
root[0] TFile *f = new
TFile("experiment.root")
root[1] tree1->Process("Analyze.C")
```

Load Analyze.C and run its analysis code on the contents of the tree. This means:

- Load your definitions
- Execute your set-up
- Execute the loop code for each entry in the tree
- Execute your wrap-up code

```
#define Analyze_cxx
#include "Analyze.h"
#include <TH2.h>
#include <TStyle.h>

//******** Definition section *********

void Analyze::Begin(TTree * /*tree*/)
{
  TString option = GetOption();

  //******** Initialization section *********
}

void Analyze::SlaveBegin(TTree* tree) {}

Bool_t Analyze::Process(Long64_t entry)
{
  // Don't delete this line! Without it the program will crash.
  fReader.SetEntry(entry);

  //******** Loop section *********
  // You probably want GetEntry(entry) here.
  return kTRUE;
}

void Analyze::SlaveTerminate() {}

void Analyze::Terminate()
{
  //******** Wrap-up section *********
}
```

Loading Analyze.C

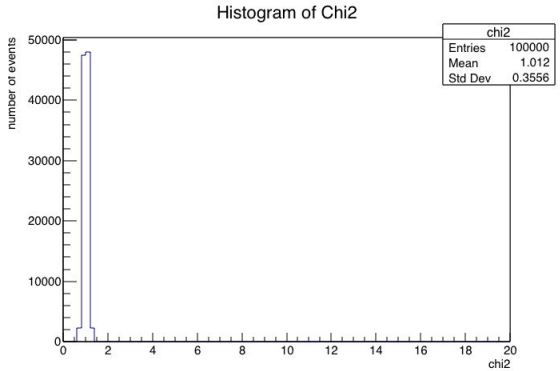
```
#define Analyze_cxx
#include "Analyze.h"
#include <TH2.h>
#include <TStyle.h>

//**********Definition section**********
TH1* chi2Hist = NULL;

void Analyze::Begin(TTree * /*tree*/)
{
   TString option = GetOption();

   //**********Initialization section**********
   chi2Hist = new TH1D("chi2", "Histogram of Chi2", 100, 0, 20);
   chi2Hist->GetXaxis()->SetTitle("chi2");
   chi2Hist->GetYaxis()->SetTitle("number of events");
}

void Analyze::SlaveBegin(TTree * /*tree*/){}

Bool_t Analyze::Process(Long64_t entry)
{
   // Don't delete this line! Without it the program will crash
   fReader.SetLocalEntry(entry);

   //**********Loop section**********
   GetEntry(entry);
   chi2Hist->Fill(*chi2);

   return kTRUE;
}

void Analyze::SlaveTerminate(){}

void Analyze::Terminate()
{
    //**********Wrap-up section**********
    chi2Hist->Draw();
}
```

Define a new histogram pointer and call it `chi2Hist`

Set this pointer to a new histogram object

Get an entry from the TTree and add 1 to a bin that correspond to the value of `*chi2` in the histogram `chi2Hist`

```
root[0] TFile *f = new TFile("experiment.root")
root[1] tree1->Process("Analyze.C")
```



Chains

★ A `TChain` object is a list of ROOT files containing the same tree. As an example, assume we have three files called `file1.root, file2.root, file3.root`. Each file contains one tree called "`T`". We can create a chain with the following statements:

```
TChain chain("T");   // name of the tree is the argument
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

★ We can use the `TChain::Draw` method `chain.Draw("x");`

★ When using a TChain, the branch address(es) must be set with:

```
chain.SetBranchAddress(branchname,...)  // use this for TChain
```

Fit with ROOT

A few words about fitting...

Jennifer Raaf

- ★ A mathematical procedure to find parameter values of a function, **f**, that best describe your data distribution
- ★ One common method for finding an optimum fit is to minimize a χ^2 function

Adjustable parameters a_0, a_1, …, a_k

Measured value in bin i

Value predicted by function **f** in bin i for parameters a_0, a_1, etc.

$$\chi^2(a_0 \ldots a_k) = \sum_{i=1}^{nbins} \left(\frac{y_i - f(x_i; a_0 \ldots a_k)}{\sigma_i} \right)^2$$

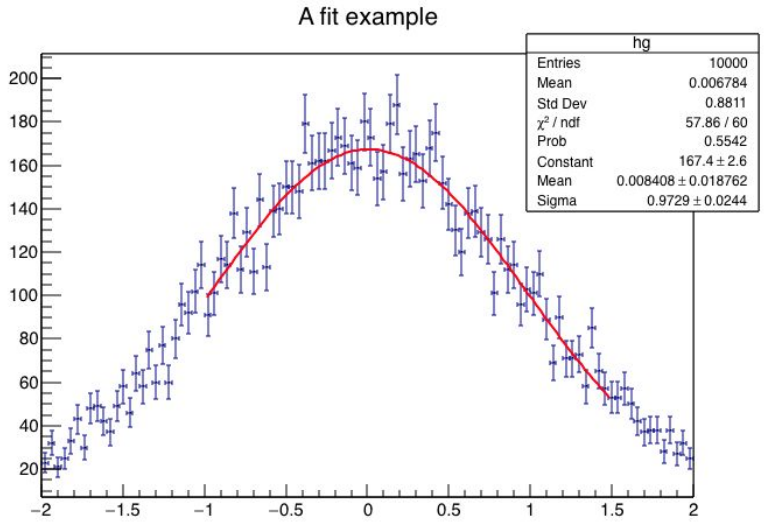
- ★ A rule of thumb for a "reasonable" fit: χ^2/NDF ~ 1
- ★ When you use ROOT to fit a distribution, its task is to find the set of parameters that give the lowest value of χ^2
- ★ It reports the parameter values for the lowest χ^2 and its best estimate of the uncertainties on those parameters. You can tell a lot about the reliability of your fir results by looking the fit itself and the parameter uncertainties.

Fitting histograms

- ★ ROOT provides predefined fittable functions for polynomials, exponentials, Gaussian, landau
- ★ User defined functions can be defined
- ★ Histograms can be fitted with `TH1F::Fit(name of TF1)`
- ★ ROOT uses Minuit package to fit

```
root [0] TH1F *h = new TH1F("hg","A fit example", 100, -2, 2
 (TH1F *) 0x7fc01af9c600
root [1] h->FillRandom("gaus",10000)
root [2] h->Fit("gaus","V","E1",-1,1.5)
```

```
FCN=57.8637 FROM MIGRAD    STATUS=CONVERGED      67 CALLS          68 TOTAL
                     EDM=5.49103e-09    STRATEGY= 1  ERROR MATRIX UNCERTAINTY   3.0 per cent
 EXT PARAMETER                                   STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  Constant     1.67423e+02   2.55377e+00   3.98809e-03  -5.90421e-05
  2  Mean         8.40797e-03   1.87621e-02   3.14518e-05  -6.77720e-03
  3  Sigma        9.72947e-01   2.43965e-02  -2.81597e-05  -1.60699e-02
```

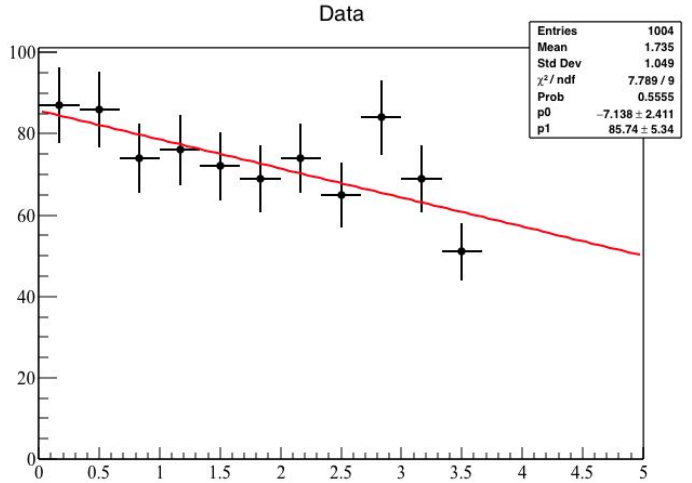


Example: straight line fit

- ★ Download the file:
- ★ Define a 1-dimension function to fit a straight line to histogram “h0” and perform the fit
  - Hint: fitting can be done using the Fit method of TH1, fo the form histogram->Fit(“myfunction”)
- ★ What are χ^2 the and number of degrees of freedom? Is it a good fit?
- ★ Do you think the uncertainties on the fit parameters are reasonable?
  - Draw two extra TF1’s on the plot:
    - First, a new line with its parameters set to (best fi values + uncertainties) from part 1
    - Second, a new line with parameters set to (best fi values - uncertainties) from part 1
  - Why don’t the lines look like they describe the data well?

Example: straight line fit

```
root [0] TFile *infile = new TFile("lines2.root")
(TFile *) 0x7fd9c8ff6d40
root [1] TF1 *line0 = new TF1("line0","[0]*x+[1]",0,5)
(TF1 *) 0x7fd9ca0a7d80
root [2] line0->SetParameters(-1.0,10.0)
root [3] h0->Fit("line0")
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
 FCN=7.78919 FROM MIGRAD    STATUS=CONVERGED      29 CALLS          30 TOTAL
                     EDM=1.56469e-22    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           -7.13820e+00   2.41071e+00   1.67023e-03   1.46237e-11
   2  p1            8.57422e+01   5.34020e+00   3.69987e-03   4.80113e-12
(TFitResultPtr) <nullptr TFitResult>
```



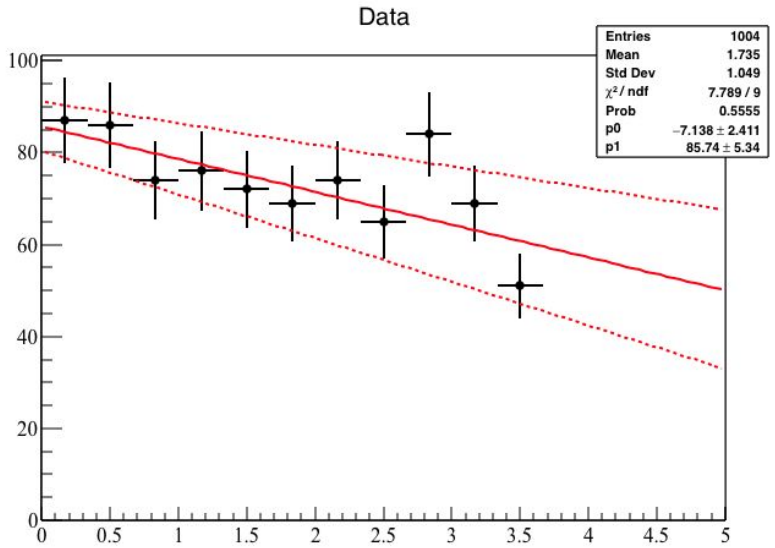
★ What are the χ^2 and number of degrees of freedom? Is it a good fit?

```
root [4] TF1 *fitresult = h0->GetFunction("line0")
(TF1 *) 0x7f85bd1b46f0
root [5] fitresult->GetChisquare()
(double) 7.7891902
root [6] fitresult->GetNDF()
(int) 9
root [7] fitresult->GetProb()
(double) 0.55552258
```

This probability is not the “probability that your fit is good”.
If you did many fake experiments (draw many random samples of data points from the assumed distribution (your fit function)), this is the percentage of experiments that would give χ^2 values >= to the one you got in this experiment.

Example: straight line fit

- ★ Do you think the uncertainties on the fit parameters are reasonable?
  - Draw two extra TF1's on the plot:
    - First, a new line with its parameters set to (best fi values + uncertainties) from part 1
    - Second, a new line with parameters set to (best fi values - uncertainties) from part 1
  - Why don't the lines look like they describe the data well?

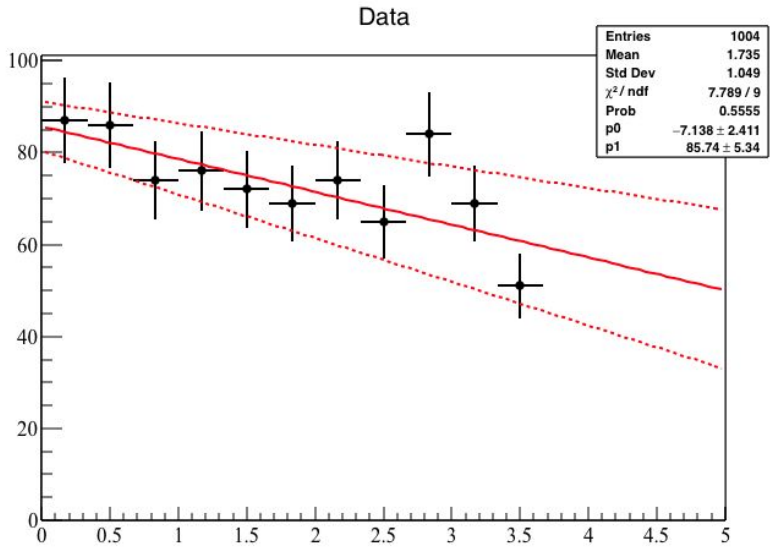


Blindly adjusting the fit parameters to the bounds of their 1-sigma errors gives unreasonable looking results. Why?

```
root [4] line0->GetParameter(0)
(double) -7.1382035
root [5] line0->GetParameter(1)
(double) 85.742234
root [6] line0->GetParError(0)
(double) 2.4107142
root [7] line0->GetParError(1)
(double) 5.3401957
root [8] TF1 *line0_m = new TF1("line0_m","-9.5*x+80.4",0,5)
(TF1 *) 0x7f90e4523260
root [9] TF1 *line0_p = new TF1("line0_p","-4.7*x+91.1",0,5)
(TF1 *) 0x7f90e452e260
root [10] line0_m->Draw("same")
root [11] line0_p->Draw("same")
```

Example: straight line fit

- ★ Do you think the uncertainties on the fit parameters are reasonable?
  - Draw two extra TF1's on the plot:
    - First, a new line with its parameters set to (best fi values + uncertainties) from part 1
    - Second, a new line with parameters set to (best fi values - uncertainties) from part 1
  - Why don't the lines look like they describe the data well?



Blindly adjusting the fit parameters to the bounds of their 1-sigma errors gives unreasonable looking results. Why?

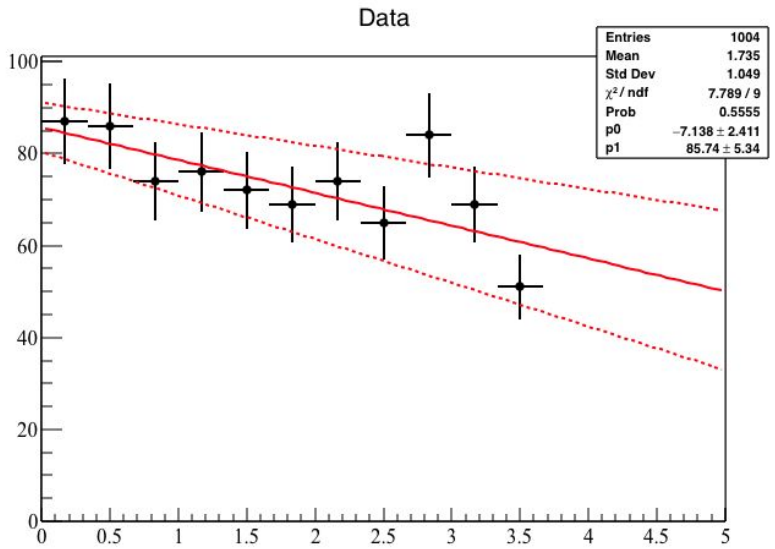
The parameters are not independent - they are correlated! ROOT (by way of its underlying minimizer, called TMinuit) can print the covariance (error) matrix for you

```
root [15] TMatrixD matrix0(2,2)
(TMatrixD &) Name: TMatrixT<double> Title:
root [16] gMinuit->mnemat(matrix0.GetMatrixArray(),2)
root [17] matrix0.Print()

2x2 matrix is as follows

     |      0    |      1    |
--------------------------------
   0 |      5.812       -11.3
   1 |      -11.3       28.52
```

Diagonal elements: (parameter error)^2
Off-diagonal elements: how parameters co-vary

Example: straight line fit

- ★ Do you think the uncertainties on the fit parameters are reasonable?
  - Draw two extra TF1's on the plot:
    - First, a new line with its parameters set to (best fi values + uncertainties) from part 1
    - Second, a new line with parameters set to (best fi values - uncertainties) from part 1
  - Why don't the lines look like they describe the data well?

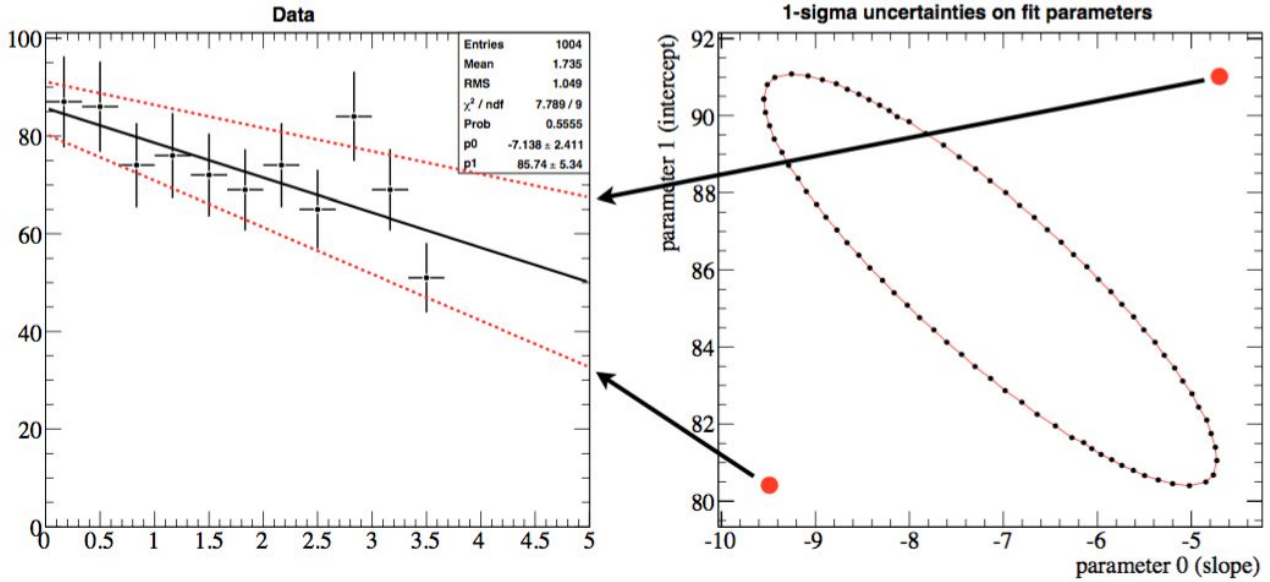


TMinuit can also calculate the correlation matrix from the covariance matrix...

```
[root [18] gMinuit->mnmatu(1)
  EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=1
   5.812e+00 -1.130e+01
  -1.130e+01  2.852e+01
  PARAMETER  CORRELATION COEFFICIENTS
        NO.  GLOBAL      1      2
         1  0.87803   1.000 -0.878
         2  0.87803  -0.878  1.000
```

-1 would indicate perfect anti-correlation
0 no correlation of these parameters
+1 perfect correlation

Parameter Correlations

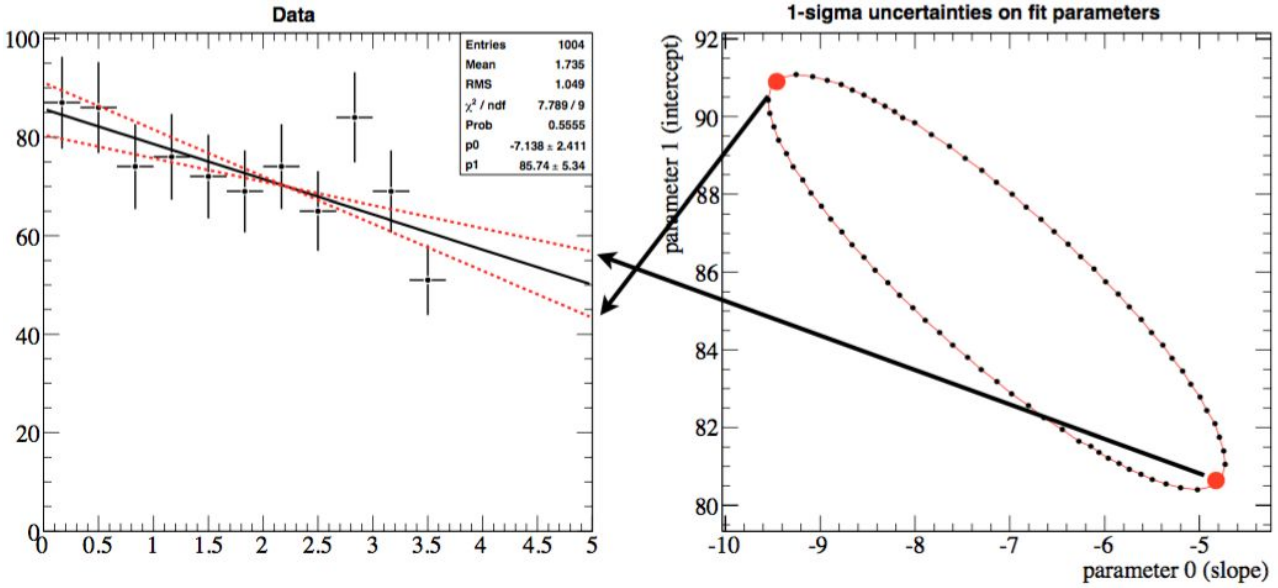



```
[root [8] gMinuit->SetErrorDef(1)
 (int) 0
 root [9] TGraph *gr0 = (TGraph *)gMinuit->Contour(80,0,1)
 (TGraph *) 0x7f87b24af860
 root [10] gr0->SetLineColor(kRed)
 root [11] gr0->Draw("alp")
 root [12] gr0->GetXaxis()->SetTitle("parameter 0 (slope)")
 root [13] gr0->GetYaxis()->SetTitle("parameter 1 (intercept)")
[root [14] gr0->SetTitle("1-sigma uncertainties on fit parameters")
```

SetErrorDef(N^2) for N-sigma error

Show 80 points on contour, show correlation of param 0 with param 1

Parameter Correlations

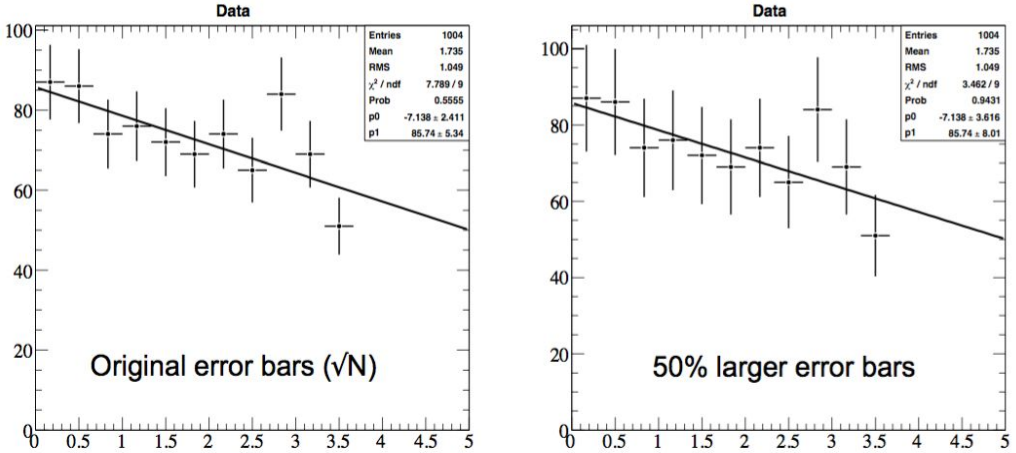



```
root [8] gMinuit->SetErrorDef(1)
(int) 0
root [9] TGraph *gr0 = (TGraph *)gMinuit->Contour(80,0,1)
(TGraph *) 0x7f87b24af860
root [10] gr0->SetLineColor(kRed)
root [11] gr0->Draw("alp")
root [12] gr0->GetXaxis()->SetTitle("parameter 0 (slope)")
root [13] gr0->GetYaxis()->SetTitle("parameter 1 (intercept)")
root [14] gr0->SetTitle("1-sigma uncertainties on fit parameters")
```

SetErrorDef(N^2) for N-sigma error

Show 80 points on contour, show correlation of param 0 with param 1

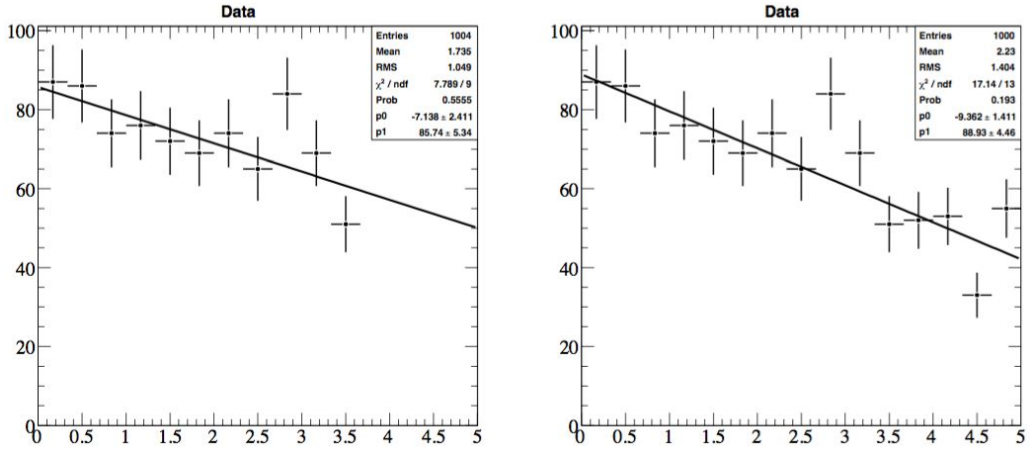
How do error bars affect the fit?



Entries	1004		Entries	1004
Mean	1.735		Mean	1.735
RMS	1.049		RMS	1.049
χ^2 / ndf	7.789 / 9	Lower χ^2 value, higher probability	χ^2 / ndf	3.462 / 9
Prob	0.5555		Prob	0.9431
p0	-7.138 ± 2.411	Best fit values remain the same, but larger uncertainty on parameter values.	p0	-7.138 ± 3.616
p1	85.74 ± 5.34		p1	85.74 ± 8.01

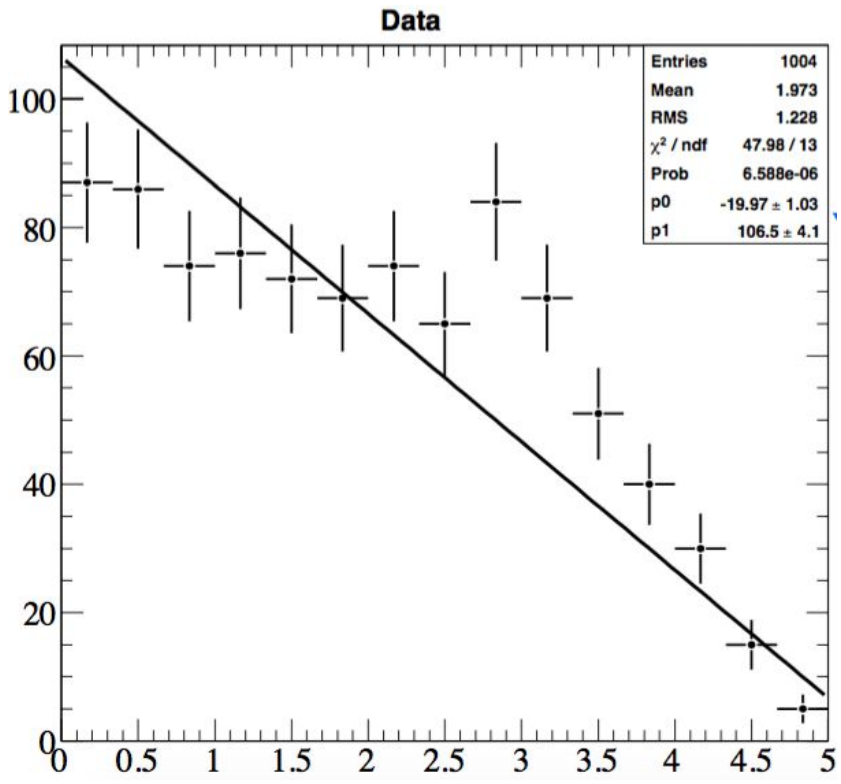
- ★ Very high probability, but very low χ^2 value: This does NOT necessarily mean that your fit is good!
- ★ A strong indication that something is not quite right…
- ★ Maybe you have overestimated your measurement errors

What about adding more data points?



- ★ Both have reasonable χ^2/NDF, but the fit with more data has smaller uncertainties on the parameters
- ★ Fit goodness (χ^2/NDF) and fit parameter uncertainties are affected by both number of data points and by the size of the data errors.

Terrible fit with tiny uncertainties



Two indications of a bad fit:

- ★ χ^2/NDF >> 1 “unreasonably small” probability”
- ★ But the parameter uncertainties are smaller than in original fit with same number of data points!
- ★ Low probability and large may indicate that the fit parameters and their uncertainties are meaningless

- ★ This is not a good model to describe the data!
- ★ Don’t just blindly accept the best fit parameters and their uncertainties from the fitter.
- ★ Think about whether or not they make sense!

Exercises

1. Revise the `Analyze::Terminate` method in Analyze.C to draw the histograms with error bars
2. Revise Analyze.C to create, fill and display an additional histogram of the variable `ebeam` (with error bars and axis label)
3. Fit the `ebeam` histogram to a gaussian distribution.
4. Add another plot: a scatterplot of `chi2` versus `ebeam`. Don't forget to label the axes!
5. Calculate `pt` in an analysis macro and make a histogram of the variable. (Remember that all n-tuple variables are pointers: `pt = TMath::sqrt((*px)*(*px) + (*py)*(py));`)
6. Include a histogram of `theta`, using the math function `TMath::ATan2(y,x)`.
7. Apply a cut in your analysis macro. Your goal is to count the number of events for which pz is less than 145 geV, and then display the value.
8. Revise your code to write the histograms to a file.

Extras

Get ROOT

Installation from source

Expert Level

- Get the ROOT sources:
  - *git clone http://github.com/root-project/root*
  - *Or visit https://root.cern.ch/content/release-61600*
- Create a build directory and configure ROOT:
  - *mkdir rootBuild; cd rootBuild*
  - *cmake ../root*
  - *https://root.cern.ch/building-root* for all the config options
- Start compilation
  - *make -j*
- Prepare environment:
  - *. bin/thisroot.sh*

Installing ROOT from source (using Git repository)

- ★ You can install the ROOT's sources from the download area or using directly the Git repository.
- ★ Install using Git repository:

Clone the repo

```
$ git clone https://github.com/root-project/root.git
```

Make a directory for building

```
$ mkdir build
$ cd build
```

Run cmake and make

```
$ cmake ../root
$ make -j8
```

Setup and run ROOT

```
$ source bin/thisroot.sh
$ root
```

Installing ROOT from source

- ★ You can install the ROOT's sources from the download area or using directly the Git repository.
- ★ Install from the download area: Download the source from https://root.cern/install/all_releases/

Unpack tar file

```
$ tar zxvf root_6.20.xx.source.tar.gz
```

Create a directory for containing the build

```
$ mkdir root-build
$ cd root-build
```

Execute the cmake command on the shell

```
$ cmake
/Users/sheilamarass/Downloads/root-6.20
.04
```

After CMake has finished running, start the build

```
$ cmake --build .
```

Setup the environment to run

```
$ source
/Users/sheilamarass/root-build/bin/thisr
oot.sh
```

Atart ROOT interactive application

```
$root
```

Installing ROOT from precompiled binary

The binaries are available for downloading from http://root.cern.ch/drupal/content/downloading-root. Once downloaded you need to unzip and de-tar the file. For example, if you have downloaded ROOT v5.30 for Linux-SLC5:

```
% gunzip root_v5.30.00.Linux-slc5-gcc4.3.tar.gz
% tar xvf root_v5.30.00.Linux-slc5-gcc4.3.tar
```

This will create the directory root. Before getting started read the file README/README. Also, read the Introduction chapter for an explanation of the directory structure.

https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html#installing-precompiled-binaries

Installing ROOT from Conda

- ★ Conda is a package management system and environment management system, which can install ROOT in a few minutes on Linux and MacOS.
- ★ The fastest way to get ROOT is installing Miniconda (a minimal Conda installation) and then run the following two commands:

```
bash
conda create -c conda-forge --name <my-environment> root
conda activate <my-environment>
```

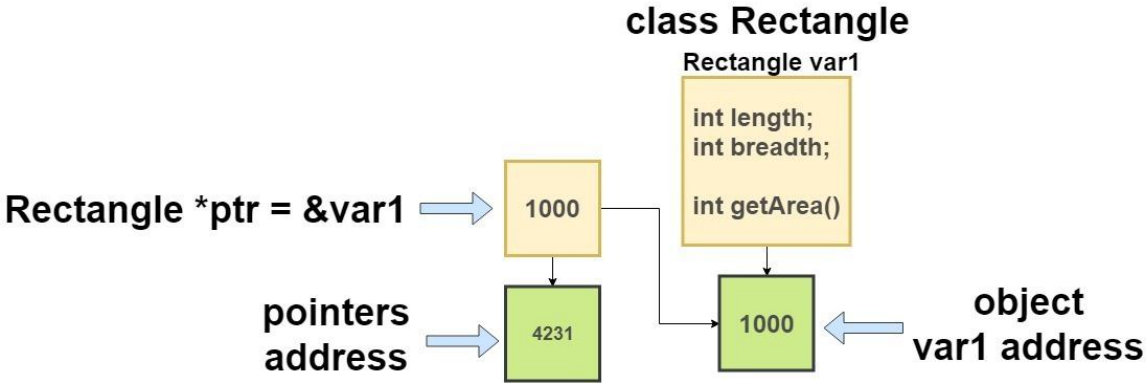
Verify ROOT version

```
# ROOT version and build tag

root --version

# Again the ROOT version (this also works with older ROOT versions)

root-config --version

# Check that ROOT was built with C++14 support

# The output must contain one of -std=c++{14,17,1z} so that all code examples of this
lesson run!

root-config --cflags

# List all the ROOT configuration options that can be checked

root-config --help
```

A Little About C++

Object-Oriented Programming Concepts

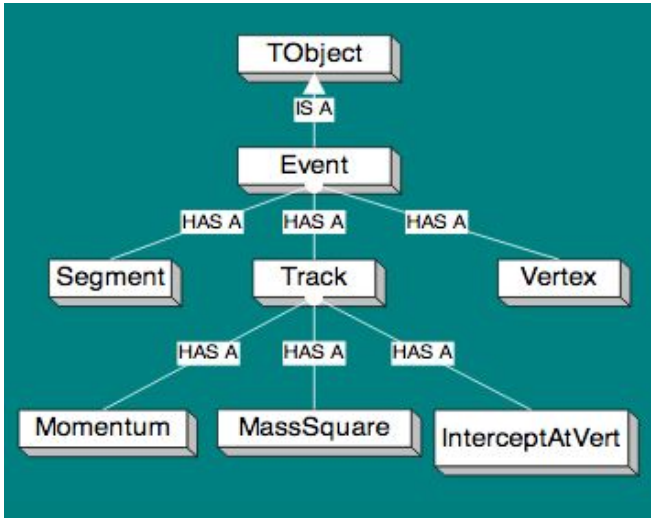
- ★ Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members
- ★ Object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.
- ★ Pointers is a variable that stores the memory address as its value.




```
class class_name{
   access_specifier_1: member1;
   access_specifier_2: member2;
   …
} object_names;
```

Object-Oriented Programming Concepts

- ★ Classes: the description of a “thing” in the system
- ★ Object : instance of a class
- ★ Methods: functions for a class
  - ○ Members: a “has a” relationship to the class
  - ○ Inheritance: an “is a” relationship to the class



The class constructor

- ★ A *constructor* constructs values of the class type. It is a member function whose name is the same as the class name.
- ★ This process involves initializing data members and, frequently, allocating free store using *new*.
- ★ A class constructor will have exact same name as the class and it does not have any return type at all, not even void.

For example: the Graph class (https://root.cern.ch/doc/master/TGraph_8h_source.html)

```
class TGraph : public TNamed, public TAttLine, public TAttFill, public TAttMarker {
...
   public:
         TGraph();
         TGraph(Int_t n);
         TGraph(Int_t n, const Int_t *x, const Int_t *y);
         TGraph(const TGraph &gr);

         virtual Double_t GetErrorX(Int_t bin) const;
         virtual Double_t GetErrorY(Int_t bin) const;
         ….
};
```

Loops: C++

for

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

Example:

```
int i=0;
for (i = 1; i <= 10; i++)
{
    printf( "Hello World\n");
}
```

Loops: C++

while

```
initialization expression;
while (test_expression)
{
   // statements
   update_expression;
}
```

Example:

```
int i = 1; // initialization expression

while (i < 6) // test expression
{
     printf( "Hello World\n");

     I++; // update expression
}
```

Loops: C++

do

```
initialization expression;
do
{
   // statements
   update_expression;
} while (test_expression);
```

Example:

```
int i = 2; // Initialization expression

do
{
    printf( "Hello World\n");

     I++; // update expression
}  while (i < 1);   // test expression
```

if … then … else: ROOT

Logical conditions:
A == B (A equal to B)
A != B (A not equal to B)
A && B (condition A and B)
A || B (condition A or B)
A >= B (A greater or equal than B)
A >=B (A greater than B)
A <= B (A less or equal than B)
A < B (A less than B)

```
if (testExpression1)
{
   // statements to be executed if testExpression1 is true
}
else if(testExpression2)
{
   // statements to be executed if testExpression1 is false and testExpression2 is true
}
else if (testExpression 3)
{
   // statements to be executed if testExpression1 and testExpression2 is false and
testExpression3 is true
}
.
.
else
{
   // statements to be executed if all test expressions are false
}
```

Function: C++

A function is a block of code which only runs when it is called

```
type name(parameter1, parameter2, ...)
{
        statements
}
```

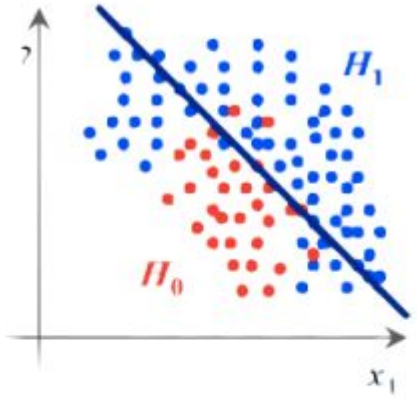
- type is the type of the value returned by the function
- name is the identifier by which the function can be called
- parameters (as many as needed): each parameter consists of a type followed by an identifier.
- Statements is the function's body

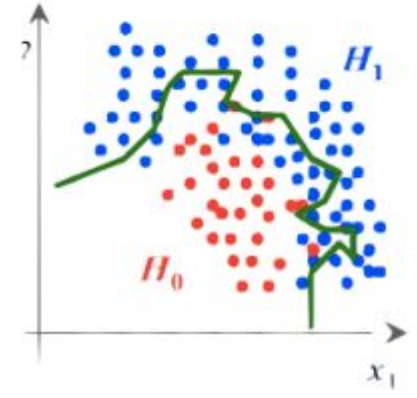
Void functions are created and used just like value-returning functions except they do not return a value after the functions executes.

& and *
http://www.cplusplus.com/doc/tutorial/functions/

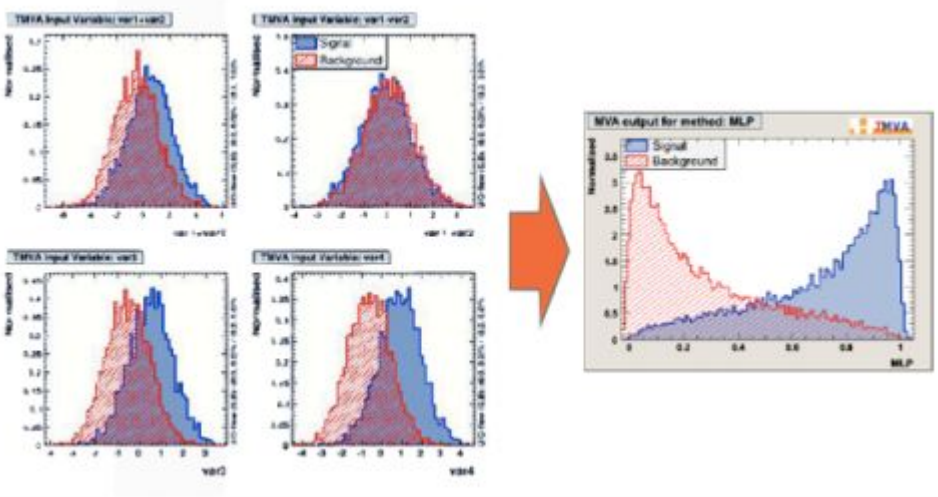
TMVA

TMVA (Toolkit for Multivariate Analysis)



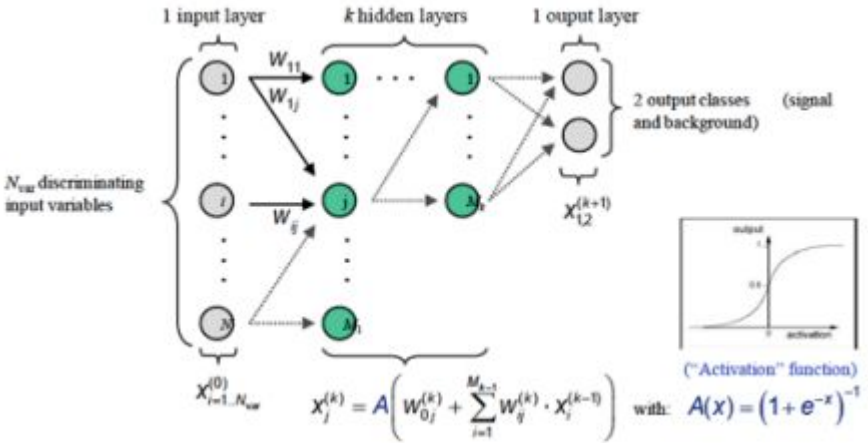


- ★ Mainly used for Classification of Signal (H1) vs Background (H0)
- ★ Based on supervised learning (usually on MC samples)
- ★ Providing a common interface to train and use different Classifiers

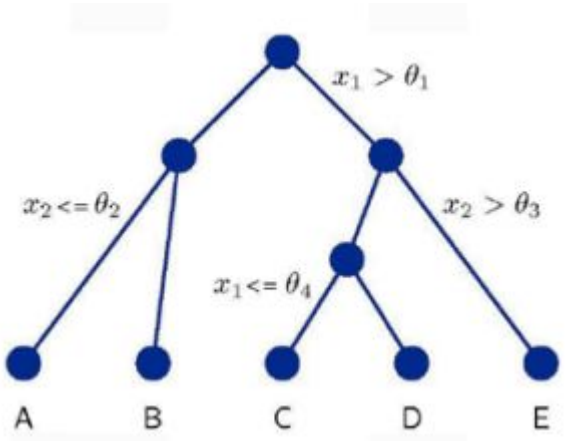


TMVA - Classifiers Examples

Artificial Neural Network



Decision Tree



… and many more (Cuts, Fisher, Likelihood, FunctionalDiscriminant, kNN, SupportVectorMachine, RuleFit…)

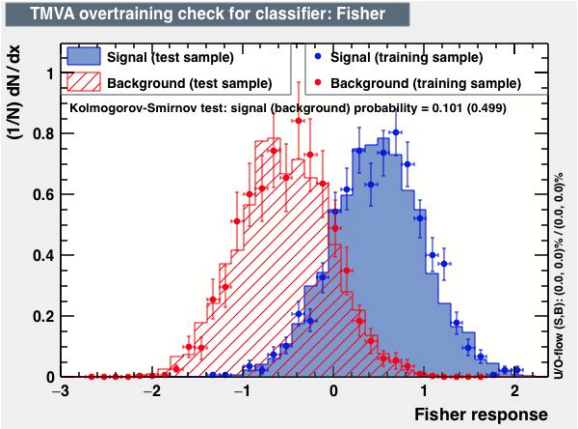
TMVA - Technicalities

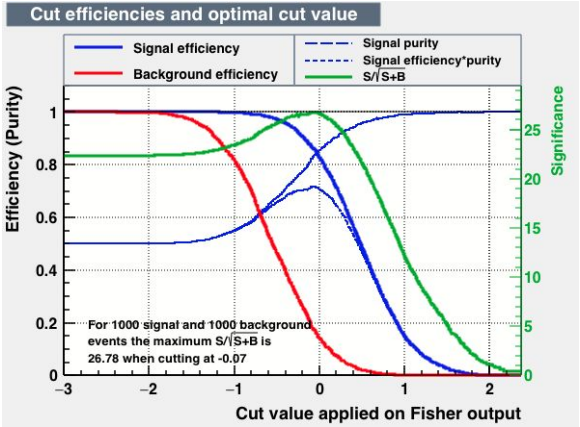
- ★ Installed with standard ROOT setup (or more versions and very good reference available under http://tmva.sourceforge.net/)
- ★ C and Python version available
  - In $ROOTSYS/tmva/test/ are some examples how to run TMVA
  - Try TMVAClassification.C (or TMVAClassification.py)
  - You can run an example by: **root -l TMVAClassification.C\(\"Fisher,Likelihood\"\)**
  - And a GUI is provided to look at many features of the trained Classifiers

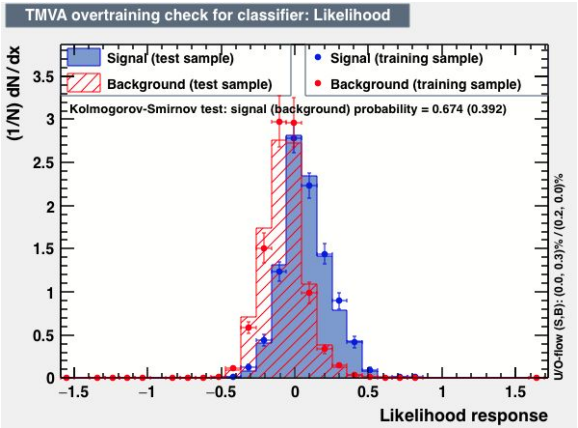
TMVA - Example

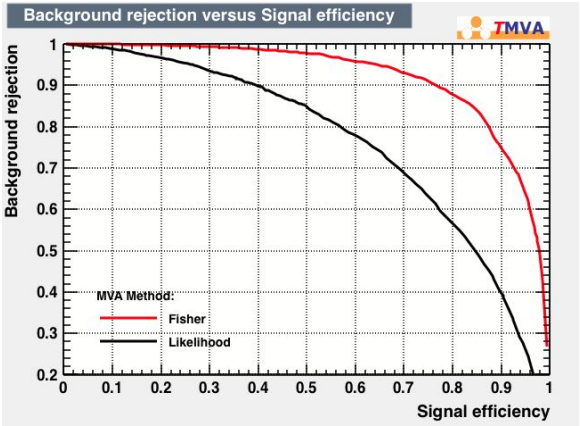
TMVA Plotting Macros for Classification

(1a) Input variables (training sample)
(1b) Input variables 'Deco'-transformed (training sample)
(1c) Input variables 'PCA'-transformed (training sample)
(1d) Input variables 'Gauss_Deco'-transformed (training sample)
(2a) Input variable correlations (scatter profiles)
(2b) Input variable correlations 'Deco'-transformed (scatter profiles)
(2c) Input variable correlations 'PCA'-transformed (scatter profiles)
(2d) Input variable correlations 'Gauss_Deco'-transformed (scatter profiles)
(3) Input Variable Linear Correlation Coefficients
(4a) Classifier Output Distributions (test sample)
(4b) Classifier Output Distributions (test and training samples superimposed)
(4c) Classifier Probability Distributions (test sample)
(4d) Classifier Rarity Distributions (test sample)
(5a) Classifier Cut Efficiencies
(5b) Classifier Background Rejection vs Signal Efficiency (ROC curve)
(5b) Classifier 1/(Backgr. Efficiency) vs Signal Efficiency (ROC curve)
(6) Parallel Coordinates (requires ROOT-version >= 5.17)
(7) PDFs of Classifiers (requires "CreateMVAPdfs" option set)
(8) Training History
(9) Likelihood Reference Distributiuons
(10a) Network Architecture (MLP)
(10b) Network Convergence Test (MLP)
(11) Decision Trees (BDT)
(12) Decision Tree Control Plots (BDT)
(13) Plot Foams (PDEFoam)
(14) General Boost Control Plots
(15) Quit









References

- http://webhome.phy.duke.edu/~raw22/public/root.tutorial/basic_root_20100701.pdf
- https://docs.google.com/presentation/d/1nNFRdh483KSYnoaA6q7x0nVeDPbhY7qjWyaGmr0ZdTA/edit#slide=id.g2a0483ea55_3_300