

ROOT Summer Student Course

ROOT

Data Analysis Framework

<https://root.cern>





Sharing the knowledge!

This course is recorded (slides and audio)

(For the benefit of everybody)

Jolly Chen



Computer Science PhD Student

My research:
Portable data structures

Marta Czurylo



*Physics PhD,
Fellow at ROOT team*

My research:
ROOT RDataFrame R&D

Vincenzo Eduardo Padulano



*Computer Science PhD,
Staff Computing Engineer*

My research:
ROOT R&D
(RDataFrame, RNTuple, Python)



Make sure you can login to SWAN: <https://swan.cern.ch>

- The Jupyter Notebook service of CERN
- IMPORTANT: first visit <https://cernbox.cern.ch>

Use the SWAN default settings



Configure Environment ✕

Specify the parameters that will be used to contextualise the container which is created for you. See [SWAN service website](#) for more details and contact to administrators.

Select **AlmaLinux 9** as Platform to try out our new experimental Alma9 image with **JupyterLab** as default interface! [More information here.](#)

Software stack more...

105a

Platform more...

CentOS 7 (gcc11)

Environment script more...

e.g. `$CERNBOX_HOME/MySWAN/myscript.sh`

Number of cores more...

2

Memory more...

8 GB

Start my Session

- Go to the [github repository](#)
- Click on the SWAN badge

ROOT course for students

Open in



SWAN



launch binder



open GH Codespaces

Introduction



A Quick Tour of ROOT



ROOT is an **international collaboration**:

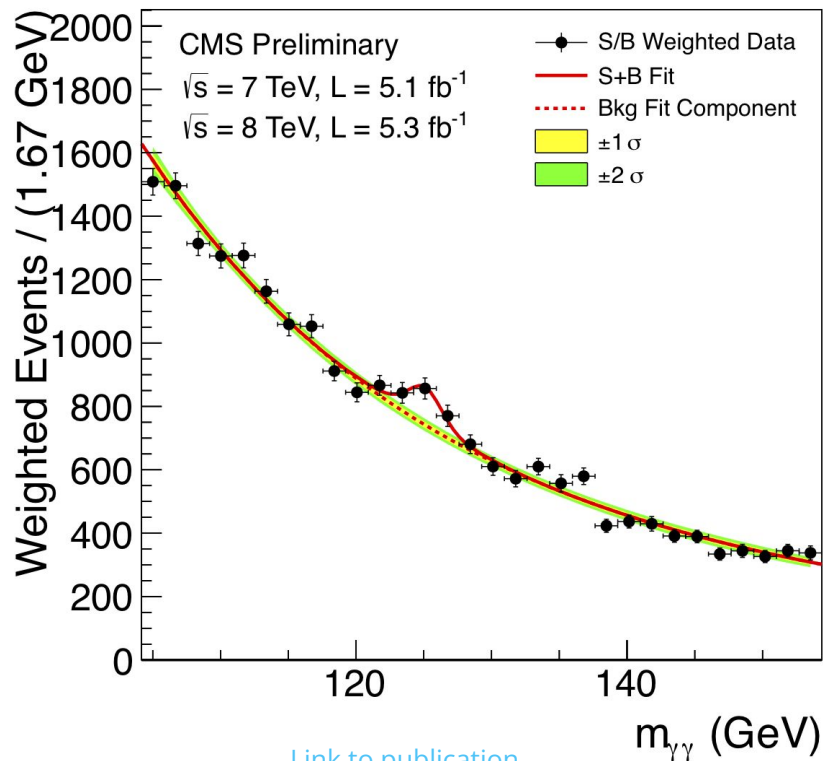
- Large effort contributed by **CERN**
- And also **FNAL, GSI, Princeton**

ROOT is its user **community, contributors** and **developers**

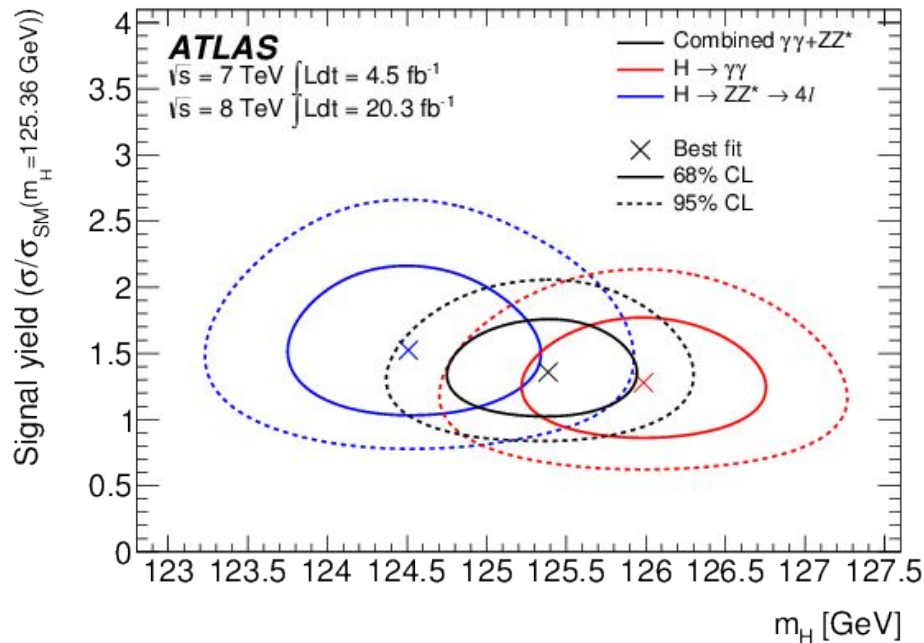
ROOT is **open source software**: contributions are welcome!



What can you do with ROOT?



[Link to publication](#)



[Link to publication](#)



ROOT in a Nutshell

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 serving, JavaScript visualisation



An Open Source Project



Fork 1.2k Starred 2.3k

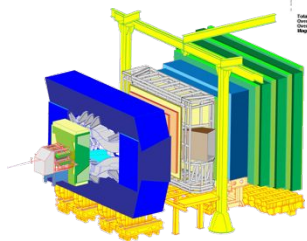
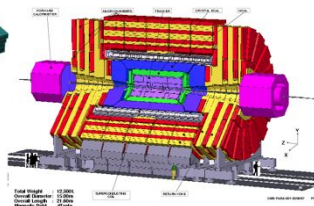
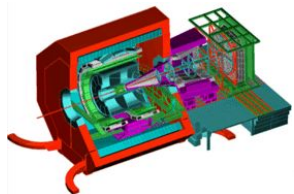
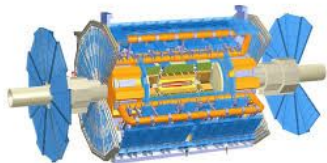
Contributors 370

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



ROOT Application Domains

A selection of the experiments adopting ROOT



Event Filtering

Offline Processing

Analysis

Reconstruction

Further processing, skimming

Event Selection, statistical treatment ...

Raw

Reco

...

Analysis Formats

Plots

Data Storage: Local, Network



~ 2EB

(exa = 10^{18})

as of 2024



- ▶ ROOT web site: **the** source of information for ROOT users
 - For beginners and experts
 - Installation instructions
 - Class documentation
 - Manuals, presentations
 - Forum

The screenshot shows the ROOT Data Analysis Framework website. At the top, there is a navigation bar with links for 'About', 'Install', 'Get Started', 'Forum & Help', 'Manual', 'Blog Posts', 'Contribute', and 'For Developers'. The main heading reads 'ROOT: analyzing petabytes of data, scientifically.' Below this, a sub-heading states 'An open-source data analysis framework used by high energy physics and others.' There are two buttons: 'Learn more' and 'Install v6.26/04'. A large background image depicts a particle detector with data points. Below the main content, there are four icons representing 'Start', 'Reference', 'Forum', and 'Gallery'. At the bottom, there are three columns of text: the first column is titled '√-1' and describes ROOT's use in scientific analysis; the second column is titled with a globe icon and describes ROOT as high-performance software; the third column is titled '\$ _' and discusses the C++ interpreter and Python integration.

ROOT
Data Analysis Framework

About Install Get Started Forum & Help Manual Blog Posts Contribute For Developers

ROOT: analyzing petabytes of data, scientifically.

An open-source data analysis framework used by high energy physics and others.

[Learn more](#) [Install v6.26/04](#)

Start Reference Forum Gallery

√-1

ROOT enables *statistically sound* scientific analyses and visualization of large amounts of data: today, more than 1 exabyte (1,000,000,000 gigabyte) are stored in ROOT files. The Higgs was found with ROOT!

🌐

As *high-performance* software, ROOT is written mainly in C++. You can use it on Linux, macOS, or Windows; it works out of the box. ROOT is open source: use it freely, modify it, [contribute to it!](#)

\$ _

ROOT comes with an incredible C++ interpreter, ideal for *fast prototyping*. Don't like C++? ROOT integrates super-smoothly with Python thanks to its *unique dynamic and powerful Python ⇄ C++ binding*. Or what about using ROOT in a Jupyter notebook?

- ▶ ROOT Forum: <https://root-forum.cern.ch>
- ▶ ROOT Website: <https://root.cern>
- ▶ Further reading: https://root.cern/get_started
 - ❖ (includes booklet for beginners: "The ROOT Primer")
- ▶ Documentation: <https://root.cern/doc/master/>





Scope of this Course

- ▶ We have 3 hours:
 - Not enough to teach you a HEP analysis
 - Instead, introduction to key elements physicists use from ROOT:
 - **Histogramming**
 - **Fitting**
 - **Reading data**
 - **Data analysis**



Course begins

We now move to the course material on Jupyter Notebooks

Wrap up



- Covered a number of topics today:
 - What is ROOT and how to use it?
 - How to draw histograms, functions and graphs
 - How to fit histograms
 - How to read and write files
 - What is RDataFrame and how to use it

- ▶ ROOT Forum: <https://root-forum.cern.ch>
- ▶ ROOT Website: <https://root.cern>
- ▶ Further reading: https://root.cern/get_started
 - ❖ (includes booklet for beginners: "The ROOT Primer")
- ▶ Documentation: <https://root.cern/doc/master/>





Post-workshop survey

- Thank you for attending the course today!
- At last - we would like to ask you to fill in a **short post-workshop survey** on indico - your opinion matters and we want to make the course even better in the future



Extra material for self study



- Most topics (but not all) were already covered in the main part of the course
- Treat the following slides as a good summary of what you've already learned plus some extra information
- Additionally, after every sub-module you are pointed to some extra exercises (in the) where you will practice both using notebooks (as during the course), but you will also attempt writing and executing C++ ROOT macros

ENJOY!

The ROOT Prompt and Macros



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!**
 - Is started with the command:

`root`

- The interactive shell is also called "ROOT prompt" or "ROOT interactive prompt"



ROOT As a Calculator

$$\begin{aligned}\frac{1}{1-x} &= 1 + x + x^2 + x^3 + x^4 + \dots \\ &= \sum_{n=0}^{\infty} x^n\end{aligned}$$

ROOT can be used as a simple calculator, but we let's make a step forward: 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
```



- ▶ 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>**
 - **.help** or **.?** gives the full list



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  
    >  
    ...  
}
```



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

- ▶ ROOT libraries can also be used to produce standalone, compiled applications:

Advanced Users

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

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

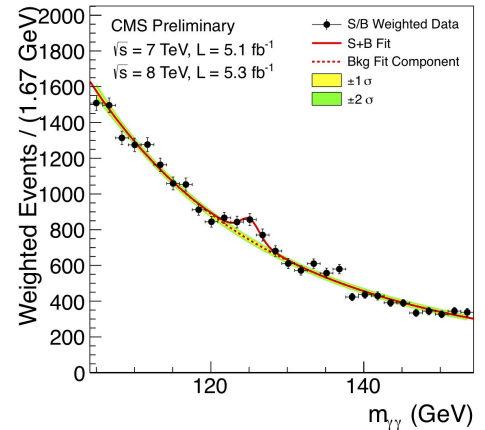


Time For Exercises

- ▶ Go to folder: [student-course/exercises/extra/00_C++_Interpreter](https://github.com/your-repo/student-course/exercises/extra/00_C++_Interpreter)

Histograms, Graphs and Functions

- ▶ A simple form of data reduction
 - Can have billions of collisions, the Physics displayed in a few histograms
 - Possible to calculate statistical quantities: mean, rms, skewness, ...
- ▶ Collect quantities in bins (discrete categories)
- ▶ ROOT provides a rich set of histograms
 - Focus on the class **TH1D today**: one dimensional histogram filled with doubles
 - but also available:
 - multiple dimensions histogram **TH{1,2,3}** classes
 - histograms holding different precision types: floats F, integers I, strings S





My First Histogram

```
root [0] TH1D h("myHist", "myTitle", 64, -4, 4)
root [1] h.Draw()
```

C++

Note that in **the SWAN notebooks**: the figure is not shown directly.

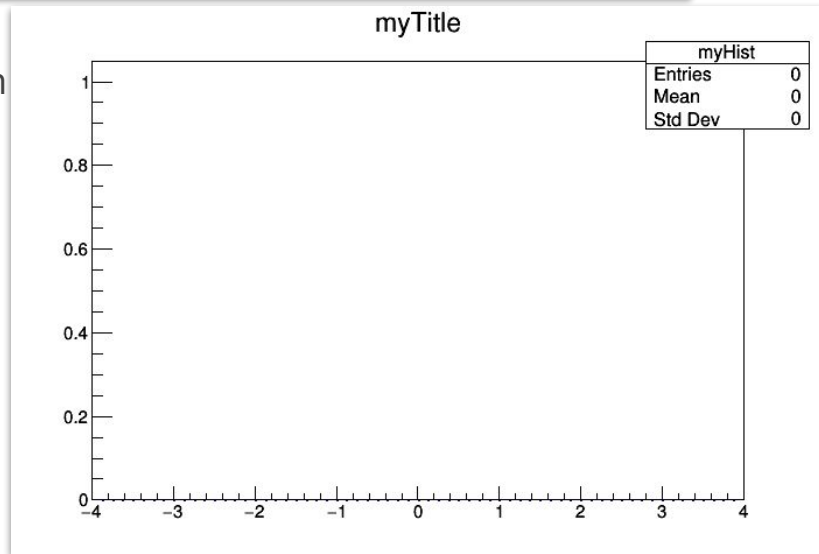
You have to:

1. Either call `gPad->Draw()` at the end:

```
In [1]: TH1D h("myHist", "myTitle", 64, -4, 4);
        h.Draw();
        gPad->Draw();
```

2. Or you can create a `TCanvas` and draw it:

```
In [2]: TCanvas c1;
        TH1D h("myHist", "myTitle", 64, -4, 4);
        h.Draw();
        c1.Draw();
```

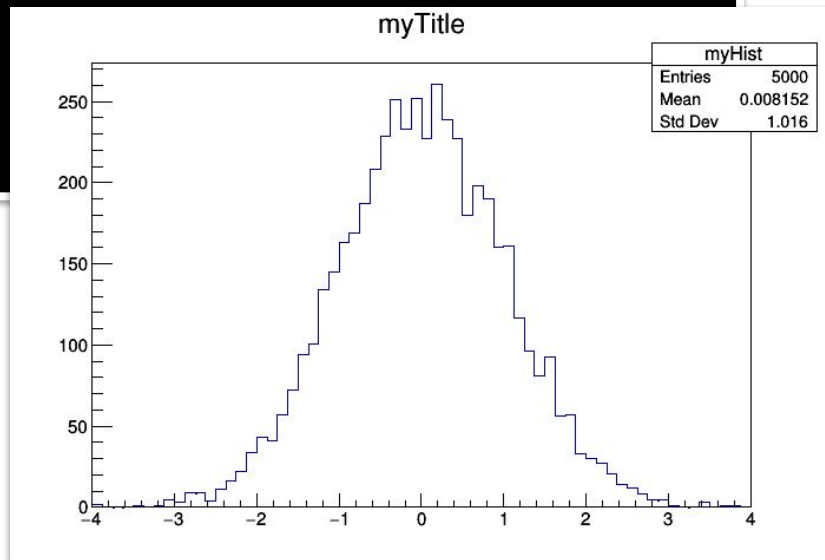




My First Histogram

```
root [0] TH1D h("myHist", "myTitle", 64, -4, 4)  
root [1] h.FillRandom("gaus")  
root [2] h.Draw()
```

C++





And now in Python!

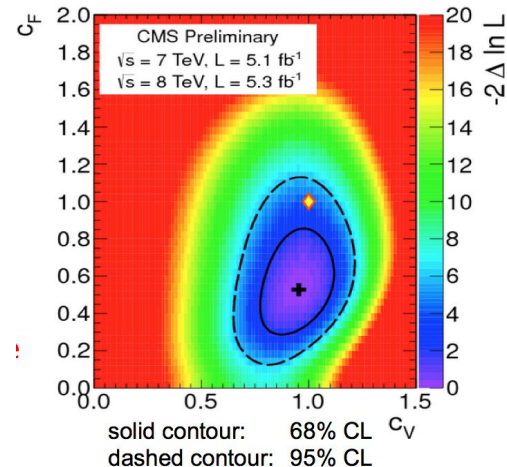
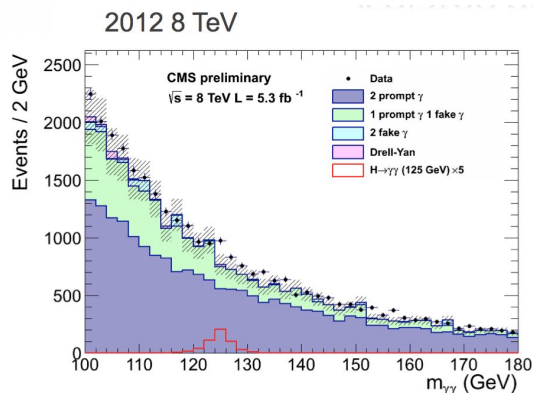
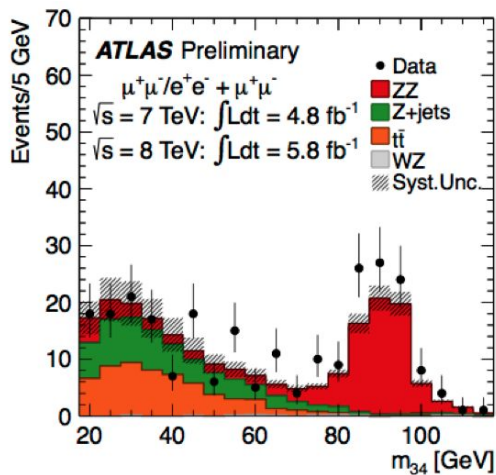
```
> python  
>>> import ROOT  
>>> h = ROOT.TH1F("myHist", "myTitle", 64,  
-4, 4)  
>>> h.FillRandom("gaus")  
>>> h.Draw()
```

Python



Drawing Options

- ▶ See the documentation of the [THistPainter](#) class for all possible options on drawing histograms

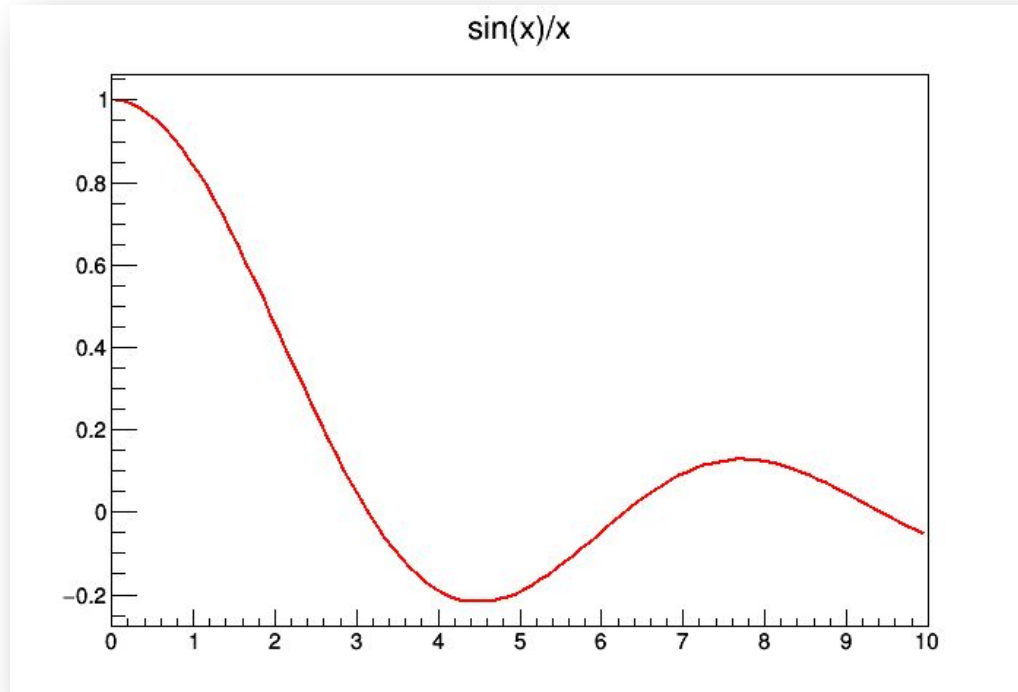




- ▶ Mathematical functions are represented by the **TF1** class
- ▶ Functions have **names**, **formulas** and **line properties**
- ▶ The **formulas** can be:
 - Mathematical formulas (written as strings)
 - C++ functions/functors/lambda (highly performant custom functions)
 - Python functions
- ▶ Functions can be written with and without **parameters**
 - Crucial for fits and parameter estimation
- ▶ Functions (as well as integrals and derivatives of functions) **can be evaluated**



ROOT as a Function Plotter





ROOT as a Function Plotter

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

C++

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

- ▶ An extended version of this example is the definition of a function **with parameters**:

Python

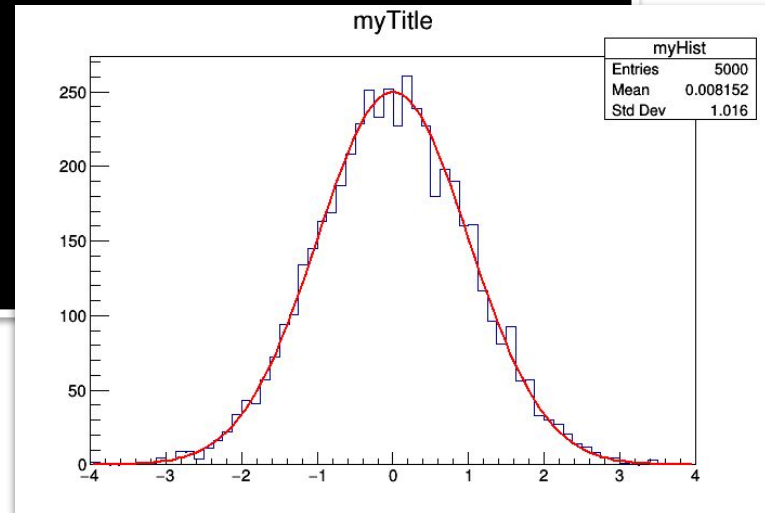
```
>>> f2 = ROOT.TF1("f2", "[0]*sin([1]*x)/x", 0., 10.)
>>> f2.SetParameters(1,1)
>>> f2.Draw()
```



Another Example: Histogram and function drawn together

C++

```
root [0] TH1D h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
root [3] TF1 f("g", "gaus", -8, 8)
root [4] f.SetParameters(250, 0, 1)
root [5] f.Draw("Same")
```

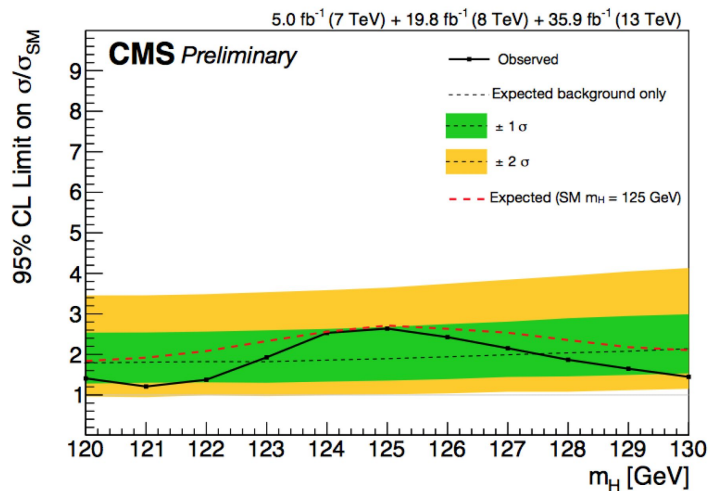
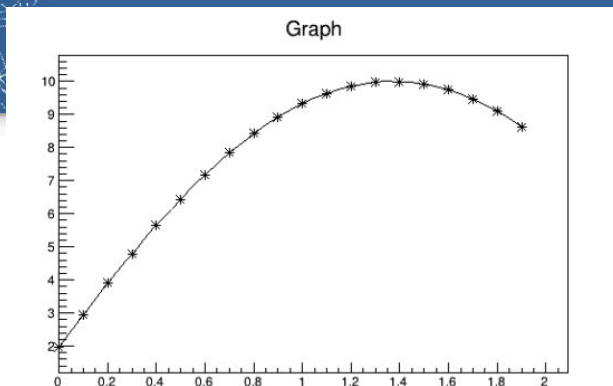




Drawing - important options

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

[TGraph Painter
documentation](#)



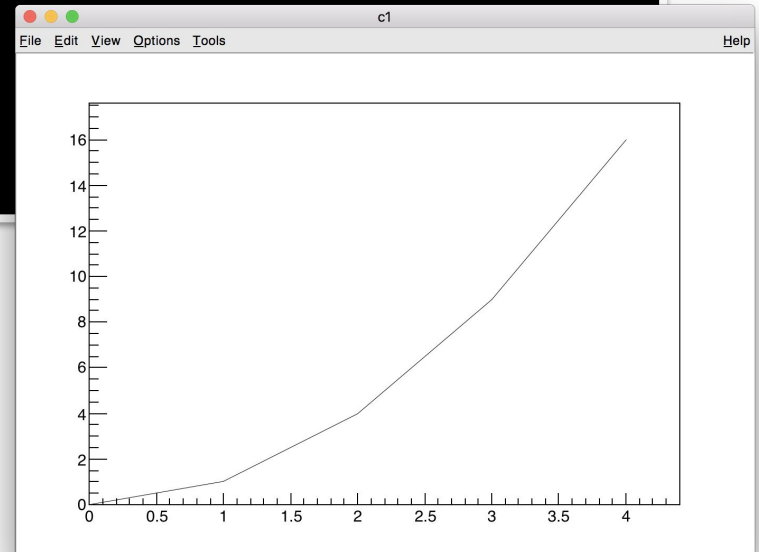
- ▶ Display points and associated errors
- ▶ **Fundamental to display trends**

[See 132nd LHCC Meeting](#)

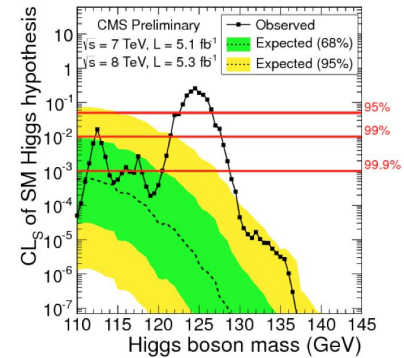
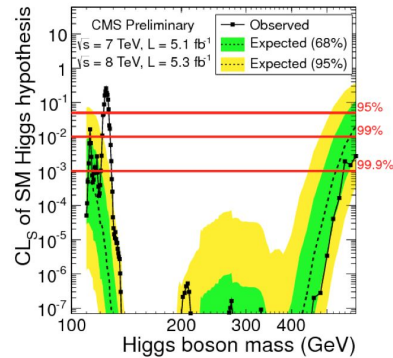
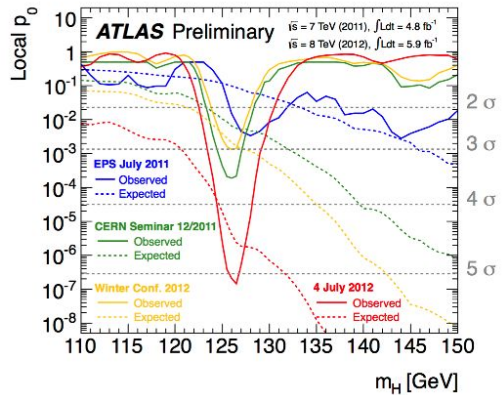
My First Graph

```
>>> g = ROOT.TGraph()  
>>> for i in range(5): g.SetPoint(i,i,i*i)  
>>> g.Draw("APL")
```

Python



- ▶ See the documentation of [TGraphPainter](#) for the Graph drawing options





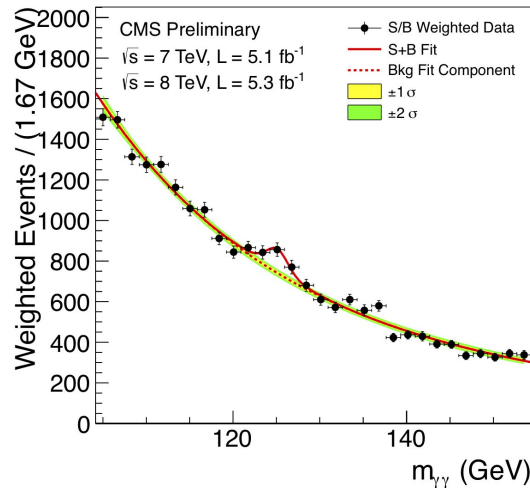
Time For Exercises

- ▶ Go to folder:
student-course/exercises/extra/01_Histograms_Graphs_Functions

Parameter Estimation and Fitting

What is Fitting ?

- ▶ Estimate parameters of a hypothetical distribution from the observed data distribution
 - $y = f(x | \theta)$ is the fit model function
- ▶ Find the best estimate of the parameters θ assuming $f(x | \theta)$
- ▶ Both Likelihood and Chi2 fitting are supported in ROOT



Example

Higgs $\rightarrow \gamma\gamma$ spectrum

We can fit for:

- the expected number of Higgs events
- the Higgs mass



- ▶ Create first a **parametric function object**, **TF1**, which represents our model
 - need to set the initial values of the function parameters
 - or use a pre-defined function
- ▶ **Fit the data object** (Histogram or Graph):
 - Call the **Fit** method passing the function object
 - various options are possible (see the [TH1::Fit](#) documentation)
- ▶ **Examine the result**:
 - get parameter values, uncertainties, correlation
 - get fit quality estimation
- ▶ **Draw the fit function**:
 - automatically, on top of the Histogram or the Graph when calling **TH1::Fit** or **TGraph::Fit**



Creating the Fit Function

▶ Parametric function object (**TF1**) :

- write formula expressions using functions:

```
TF1 f1 ("f1", "[0]*TMath::Gaus(x, [1], [2])") ;
```

- **[0],[1],[2] indicate the parameters.**
- We could also use meaningful names, like [a],[mean],[sigma]

▶ Use the available functions in ROOT library

- Pre-defined functions e.g.: *gaus*, *expo*, *landau*...

```
TF1 ("f1", "gaus") ;
```

- ▶ for more complex examples and fitting options see [backup slides](#)
- ▶ for full list of functions see the documentation of [TH1::Fit\(\)](#), and the [TFormula reference doc](#)

Fitting Histograms example

- ▶ We have a histogram, h1, and we want to fit a function to it:

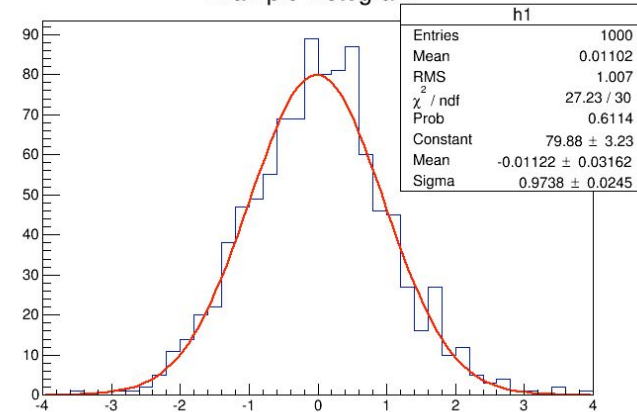
```
root [0] TF1 f1("f1","gaus");  
root [1] h1.Fit(&f1);
```

```
FCN=27.2252 FROM MIGRAD      STATUS=CONVERGED      60 CALLS      61 TOTAL  
EDM=1.12393e-07      STRATEGY= 1      ERROR MATRIX ACCURATE
```

EXT PARAMETER

NO.	NAME	VALUE	ERROR	STEP	FIRST
1	Constant	7.98760e+01	3.22882e+00	6.64363e-03	-1.55477e-05
2	Mean	-1.12183e-02	3.16223e-02	8.18642e-05	-1.49026e-02
3	Sigma	9.73840e-01	2.44738e-02	1.69250e-05	-5.41154e-03

Example histogram



For displaying the fit parameters:

```
gStyle->SetOptFit(1111);
```



Minimization

- ▶ The fit is done by minimizing the least-square or maximizing the likelihood function.
- ▶ A direct solution exists only in case of linear fitting
 - it is done automatically in such cases (e.g fitting polynomials).
- ▶ Otherwise an **iterative algorithm** is used:
 - Minuit is the minimization algorithm used by default
 - ROOT provides two implementations*: Minuit and Minuit2
 - To change the minimizer:

```
ROOT::Math::MinimizerOptions::SetDefaultMinimizer("Minuit2");
```
 - Other commands are also available to control the minimization, see [documentation](#)

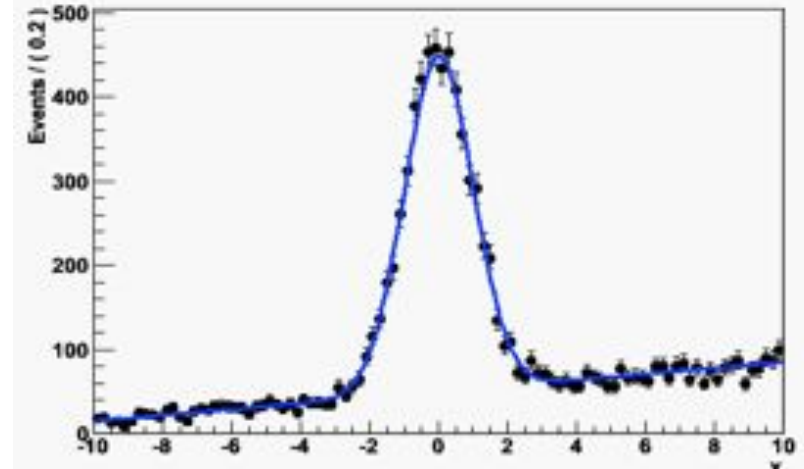
*other algorithms exists, for example, Fumili, or minimizers based on GSL (genetic and simulated annealing algorithms)

Fitting - references for the future



RooFit: ROOT toolkit for complex fitting

- ▶ ROOT fitting can handle complicated functions but complex models require many lines of code
- ▶ **RooFit** provides functionality for building complex fitting models
- ▶ Fitting often requires **Normalization** of pdfs
 - not always trivial to perform → RooFit does it automatically
- ▶ RooFit also provides:
 - **MC data generation** from model
 - advance **visualization** of fitting results
 - **simultaneous fit** to different data samples
 - full model description for **reusability**
 - **built-in optimization** for optimal computational performances



For more info see the [manual](#) or the RooFit [courses](#)



TMVA: Machine Learning in ROOT

- ▶ ROOT ML tools are provided in **TMVA** (*Toolkit for MultiVariate Analysis*)
- ▶ TMVA provides a set of algorithms for standard HEP usage
 - Common interface to different algorithms with consistent evaluation and comparison
 - Capability for classification and regression
 - Embedded in ROOT: **direct connection to input data** (ROOT I/O)
 - Most popular algorithms are BDT and ANN (also supporting some DL tools)
- ▶ Interfaces to external ML library :
 - e.g. to Python tools: scikit-learn, Tensorflow/Keras, PyTorch
- ▶ Fast inference system for Deep Learning models (SOFIE) and BDT
 - new tool to generate code and easily evaluate ML models in ROOT that can be trained with other tools (e.g Keras, PyTorch) or xgboost
- ▶ For more info see the [manual](#)



Time For Exercises

- ▶ Go to folder: [student-course/exercises/extra/02_Fitting](#)
 - plenty of examples – start from the easier ones, continue with more complex

- ▶ Note on extras – how to make nice plots:
 - see the [backup](#) slides
 - see extra tutorial module, go to folder: [student-course/exercises/extra/05_Graphics](#)

Reading and Writing Data



- ▶ In ROOT, objects are written in files*, represented by **TFile** instances
- ▶ TFiles are *binary* and can be compressed (transparently for the user)
- ▶ **TFiles are self-descriptive:**
 - The information how to retrieve objects from a file is stored with the objects

* this is an understatement - we'll not go into the details in this course!



```
TFile f("myfile.root", "RECREATE");
```

Option	Description
NEW or CREATE	Create a new file and open it for writing, if the file already exists the file is not opened.
RECREATE	Create a new file, if the file already exists it will be overwritten.
UPDATE	Open an existing file for writing. If no file exists, it is created.
READ	Open an existing file for reading (default).



TFile in Action: Writing

```
TFile f("file.root", "RECREATE");  
TH1F h("h", "h", 64, 0, 8);  
h.Write("h");  
f.Close();
```

C++

- ▶ Write to a file
- ▶ Close the file and make sure the operation succeeded

```
> rootls -l file.root  
TH1F Jun 24 15:02 2022 h "h"
```



TFile in Action: Reading

```
TFile f("file.root");  
TH1F* h = f.Get<TH1F>("h");  
h->Draw();
```

C++

Python

```
import ROOT  
f = ROOT.TFile("file.root")  
f.h.Draw()
```

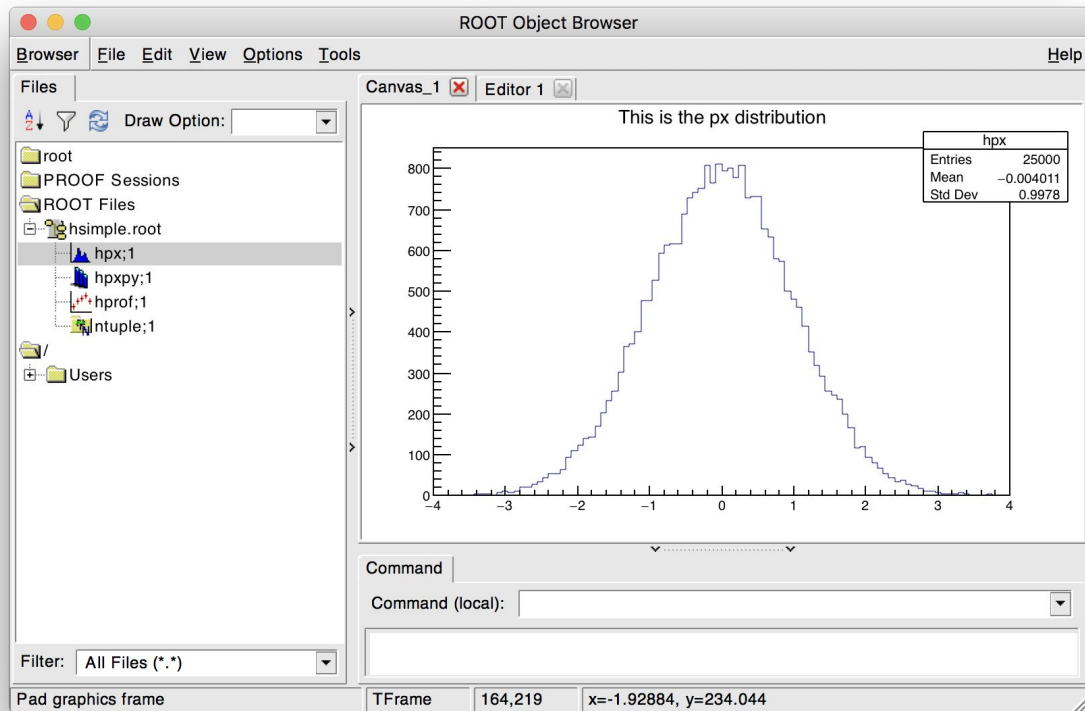
Get the histogram by name!
Possible only in Python





Listing TFile Content

- ▶ **TBrowser** interactive tool
`> root [0] TBrowser t`
- ▶ **rootls** tool: list content
- ▶ **TFile::ls()**: prints content
 - Great for interactive usage





Time For Exercises

- ▶ Go to folder: student-course/exercises/extra/03_Working_With_Files

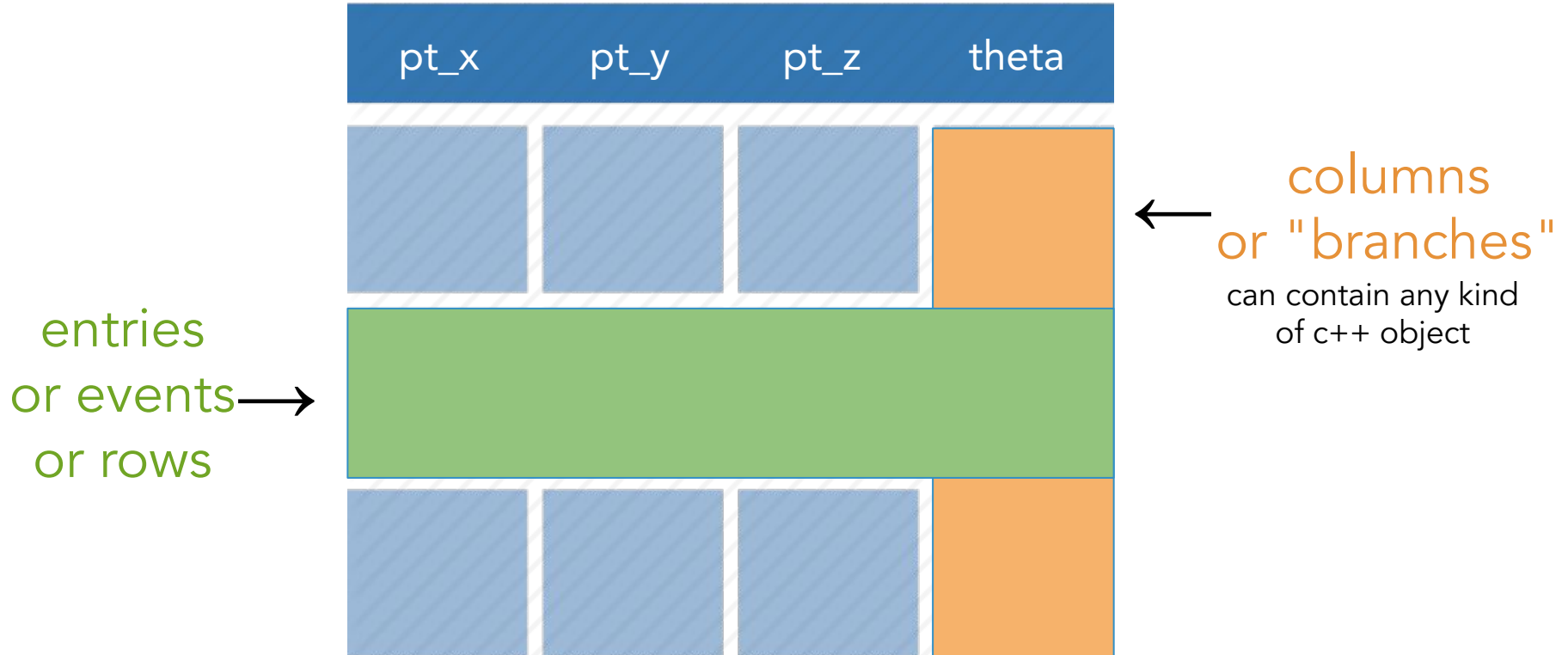
The ROOT Columnar Format



- ▶ High Energy Physics: many statistically independent *collision events*
- ▶ Create an event class, serialise and write out N instances into a file?
→ No. Very inefficient!
- ▶ Organise the dataset in **columns**



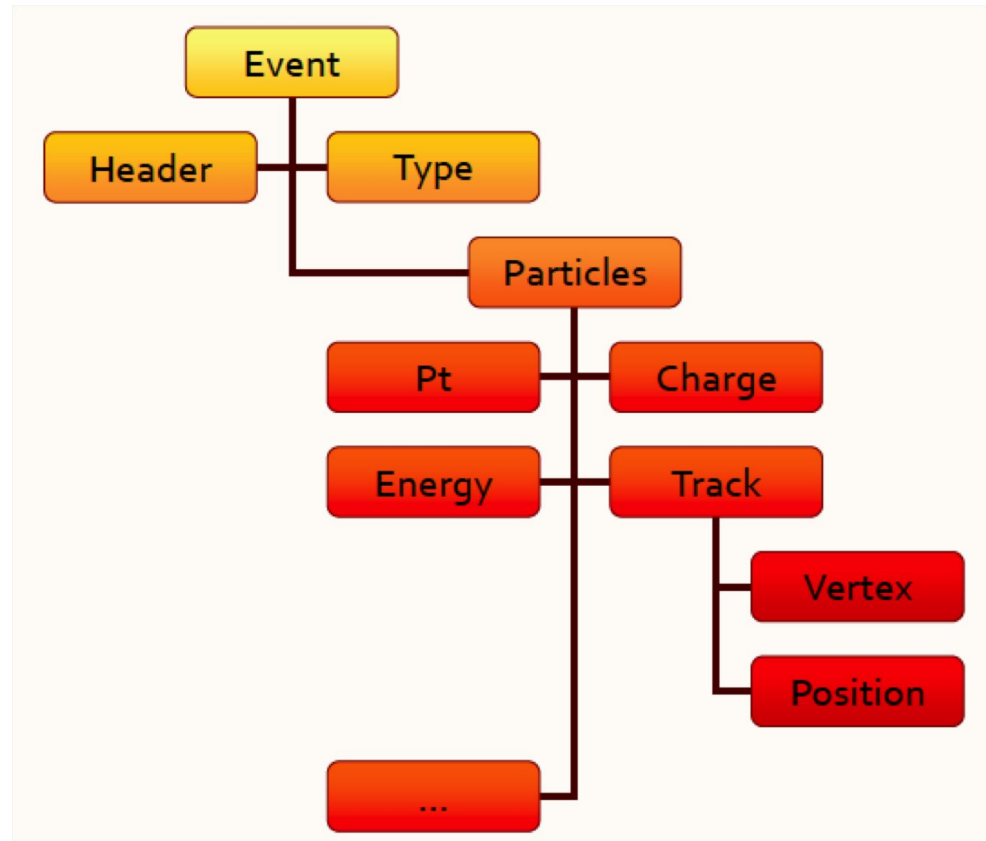
Columnar Representation





Relations Among Columns

x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
0.55254	-0.21231	1.50281
-0.184	1.187305	1.443902
0.20564	-0.7701	0.635417
1.079222	-0.32	1.271904
-0.27492	-0.43	3.038899
2.047779	-0.268	4.197329
-0.45868	-0.4	2.293266
0.304731	-0.884	0.875442
-0.7125	-0.2223	0.556881
-0.27	1.181767	1.470484
0.85	-0.65411	1.13209
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
		3.562347





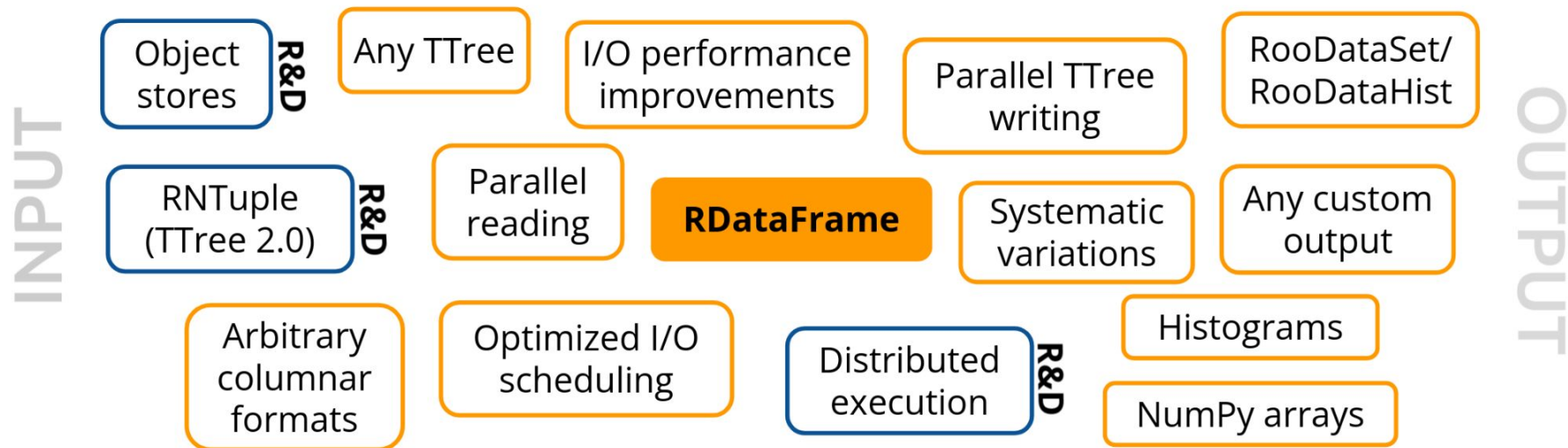
The TTree data format

A columnar dataset in ROOT is represented by the class **TTree**:

- ▶ Also called ***tree***, columns also called ***branches***
- ▶ Columns can contain **any arbitrary** C++ type



An entry point to modern ROOT





RDataFrame: quick how-to

1. build a data-frame object by specifying your data-set
2. apply a series of **transformations** to your data
 - filter (e.g. apply some cuts) or
 - define new columns
3. apply **actions** to the transformed data to produce results (e.g. fill a histogram)



Simple Code Example

1. Build RDataFrame

```
import ROOT
```

```
df = ROOT.RDataFrame("t", "f.root")  
h = df.Filter("theta > 0").Histo1D("pt")
```

```
h.Draw()
```

2. Cut on
theta

3. Fill histogram
with pt

Lazily build
computation
graph

Trigger
execution



Filling multiple histograms

```
h1 = df.Filter("theta > 0").Histo1D("pt")  
h2 = df.Filter("theta < 0").Histo1D("pt")  
h1.Draw()           // event loop is run lazily once here  
h2.Draw("SAME")    // no need to run loop again here
```

Book all your actions upfront. The first time a result is accessed, RDataFrame will fill all booked results.



More on histograms

```
h = df.Histo1D(("myName", "Title;x", 10, 0., 1.), "x")
```

You can specify a model histogram with

- a name and a title
- a predefined axis range

Here, the histogram is created with 10 bins ranging from 0 to 1, and the axis is labelled "x".



Define a new column

```
m = (  
  df.Filter("x > y")  
    .Define("z", "sqrt(x*x + y*y)")  
    .Mean("z")  
)
```

‘Define’ takes the name of the new column and its expression. Later you can use the new column as if it was present in your data.



Working with collections

```
h = df.Define(  
    "good_pt",  
    "sqrt(px*px + py*py)[E>100]"  
).Histo1D("good_pt")
```

`sqrt(px*px + py*py)[E>100]`:

- `px`, `py` and `E` are columns the elements of which are `RVecs`
- Operations on `RVecs` like `sum`, `product`, `sqrt` preserve the dimensionality of the array
- `[E>100]` selects the elements of the array that satisfy the condition
- `E > 100`: boolean expressions on `RVecs` such as `E > 100` return a mask, that is an array with information on which values pass the selection (e.g. `[0, 1, 0, 0]` if only the second element satisfies the condition)



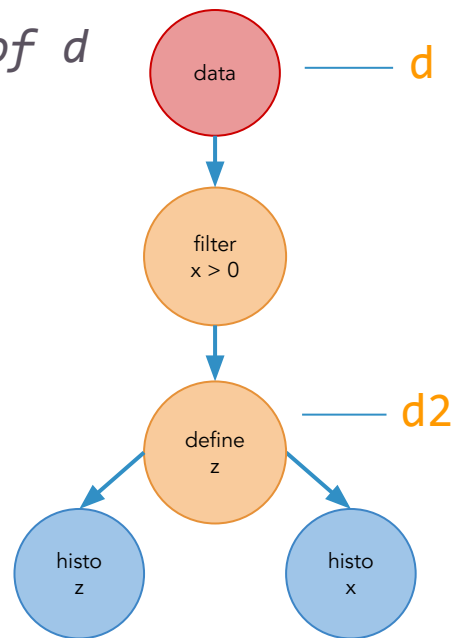
Think of your analysis as data-flow

```
// d2 is a new data-frame, a transformed version of d
```

```
auto d2 = d.Filter("x > 0")  
          .Define("z", "x*x + y*y");
```

```
// make multiple histograms out of it
```

```
auto hz = d2.Histo1D("z");  
auto hx = d2.Histo1D("x");
```



You can store transformed data-frames in variables, then use them as you would use an RDataFrame.



Cutflow reports

```
df = (df.Filter("x > 0", "xcut")  
      .Filter("y < 2", "ycut"))  
df.Report().Print()
```

```
// output
```

```
xcut      : pass=49          all=100          --   49.000 %  
ycut      : pass=22          all=49          --   44.898 %
```

When called on the main RDF object, `Report` prints statistics for all filters *with a name*



Saving data to file

```
new_df = (  
    df.Filter("x > 0")  
        .Define("z", "sqrt(x*x + y*y)")  
        .Snapshot("tree", "newfile.root")  
)
```

We filter the data, add a new column, and then save everything to file. No boilerplate code at all.



Using callables instead of strings

Expert Feature

```
// define a c++11 lambda - an inline function - that checks "x>0"  
auto IsPos = [](double x) { return x > 0.; };  
// pass it to the filter together with a list of branch names  
auto h = df.Filter(IsPos, {"theta"}).Histo1D("pt");  
h->Draw();
```

any callable (function, lambda, functor class) can be used as a filter, as long as it returns a boolean



RDataFrame: declarative analyses

```
df = ROOT.RDataFrame("t", "f.root")
df = df.Define("dphi",
              "MyDeltaPhi(phi1, phi2)")
h = df.Histo1D("dphi")
h.Draw()
```

- full control over *the analysis*
- no boilerplate
- common tasks are not already implemented?
- parallelization is not trivial?



RDataFrame: parallelism

```
ROOT.EnableImplicitMT()  
df = ROOT.RDataFrame("t", "f.root")  
df = df.Define("dphi",  
              "ROOT::VecOps::DeltaPhi(phi1, phi2)")  
h = df.Histo1D("dphi")  
h.Draw()
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented
- parallelization is trivial



C++ and just-in-time compiled code

```
d.Filter("th > 0").Snapshot("t", "f.root", "pt*");
```



PyROOT -- just leave out the ``

```
d.Filter("th > 0").Snapshot("t", "f.root", "pt*")
```



Time For Exercises

- ▶ Go to folder: student-course/exercises/extra/04_RDataFrame

Backup - fitting details



Building More Complex Functions

- ▶ Any C++ object (functor) implementing
`double operator() (double *x, double *p)`

```
struct Function {  
    double operator() (double *x, double *p){  
        return p[0]*TMath::Gaus(x[0],p[1],p[2]);  
    }  
};  
  
Function f;  
TF1 f1("f1",f,xmin,xmax,npar);
```

- also a lambda function (with Cling and C++-11)

```
TF1 f1("f1",[] (double *x, double *p){return p[0]*x[0];},0,10,1);
```

a lambda can be used also as a string expression, which will be JIT'ed by CLING

```
TF1 f1("f1","[] (double *x, double *p){return p[0]*x[0];}",0,10,1);
```



Functionality provided by TFormula

TFormula is based on [Cling](#). Additional functionality provided:

- ▶ better parameter definition
 - `TF1("f1", "gaus(x, [Constant], [Mean], [Sigma])");`
- ▶ function composition by concatenating expressions
 - `TF1 fs("sigma", "[0]*x+[1]");`
 - `TF1 f1("f1", "gaus(x, [C], [Mean], sigma(x, [A], [B]))");`
- ▶ normalized sum for component fitting
 - `TF1 model("model", "NSUM(expo, gaus)");`
- ▶ convolutions
 - `TF1 voigt("voigt", "CONV(breitwiegner, gaus)", xmin, xmax);`
- ▶ can define vectorized functions for faster fitting and evaluation
 - see [vectorizedFit](#) tutorial
- ▶ support for auto-differentiation (automatic generation of gradient and Hessian)



Fitting Options

▶ Likelihood fit for histograms

- option "L" for count histograms;
- option "WL" in case of weighted counts.

```
h1->Fit("gaus","L");
```

```
h1->Fit("gaus","LW");
```

▶ Default is chi-square with observed errors (and skipping empty bins)

- option "P" for Pearson chi-square

```
h1->Fit("gaus","P");
```

expected errors, and including empty bins

▶ Use integral function of the function in bin

```
h1->Fit("gaus","L I");
```

▶ Compute MINOS errors : option "E"

```
h1->Fit("gaus","L E");
```




Some More Fitting Options

▶ Fitting in a Range

- `h1->Fit("gaus","",",-1.5,1.5);`

▶ For doing several fits

- `h1->Fit("expo","+","",2.,4);`

▶ Quiet / Verbose: option "Q"/"V"

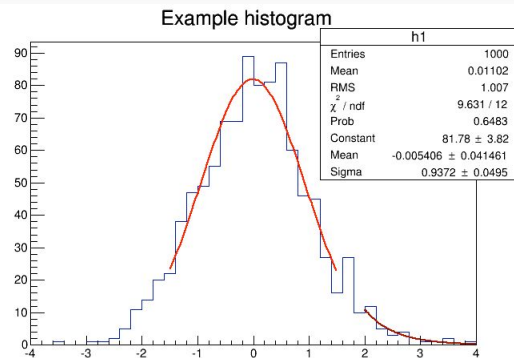
- `h1->Fit("gaus","V");`

▶ Avoid storing and drawing fit function (useful when fitting many times)

- `h1->Fit("gaus","L N 0");`

▶ Save result of the fit, option "S"

- `auto result = h1->Fit("gaus","L S");
result->Print("V");`



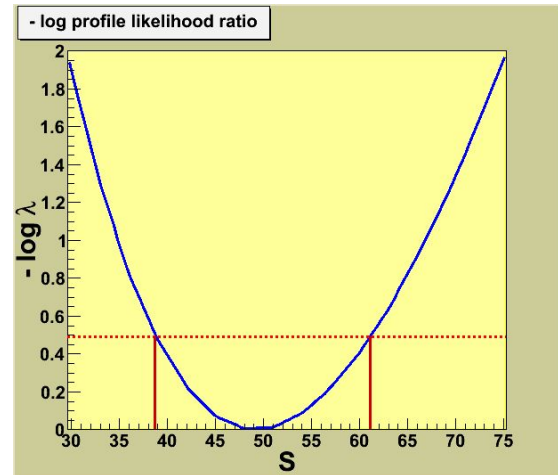


Parameter Errors

Errors returned by the fit are computed from the second derivatives of the log-likelihood function

- Assume the negative log-likelihood function is a parabola around minimum
- This is true asymptotically and in this case the parameter estimates are also normally distributed.
- The estimated correlation matrix is then:

$$\hat{\mathbf{V}}(\hat{\boldsymbol{\theta}}) = \left[\left(-\frac{\partial^2 \ln L(\mathbf{x}; \boldsymbol{\theta})}{\partial^2 \boldsymbol{\theta}} \right)_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} \right]^{-1} = \mathbf{H}^{-1}$$

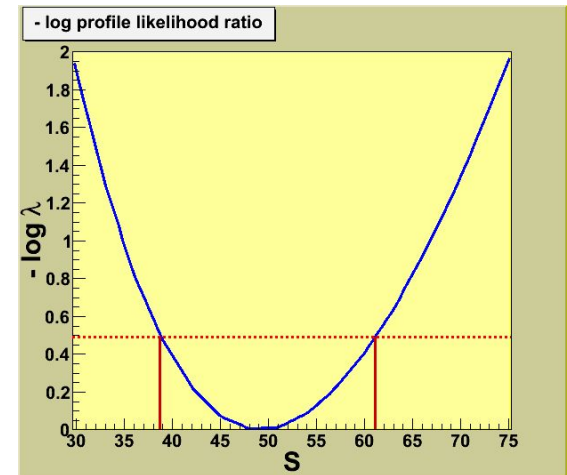




Parameter Errors

- ▶ A better approximation to estimate the confidence level of the parameter is to use directly the log-likelihood function and look at the difference from the minimum.
 - Method of Minuit/Minos (Fit option "E")
 - obtain a confidence interval which is in general not symmetric around the best parameter estimate

```
auto r = h1->Fit(f1, "E S");  
r->LowerError(par_number);  
r->UpperError(par_number);
```



Backup - Creating a Nice Plot Survival Kit

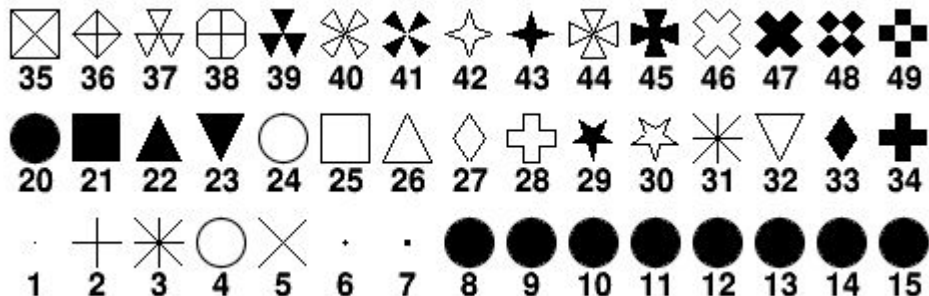




The Markers

From the TAttMarker documentation:

<https://root.cern/doc/master/classTAttMarker.html>

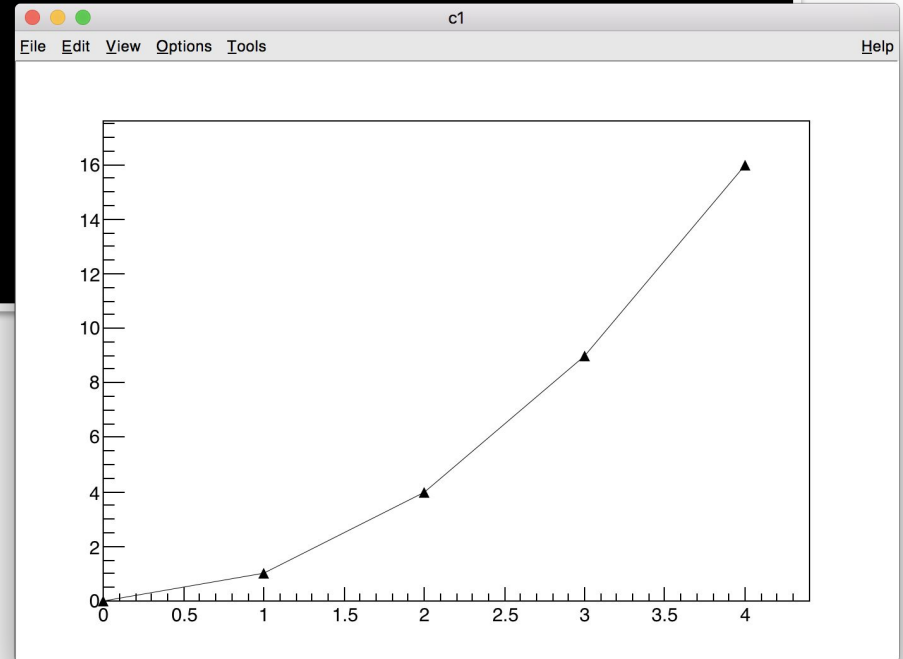


```
kDot=1, kPlus, kStar, kCircle=4, kMultiply=5,  
kFullDotSmall=6, kFullDotMedium=7, kFullDotLarge=8,  
kFullCircle=20, kFullSquare=21, kFullTriangleUp=22,  
kFullTriangleDown=23, kOpenCircle=24, kOpenSquare=25,  
kOpenTriangleUp=26, kOpenDiamond=27, kOpenCross=28,  
kFullStar=29, kOpenStar=30, kOpenTriangleDown=32,  
kFullDiamond=33, kFullCross=34 etc...
```

Also available
through more friendly
names 😊

My First Graph

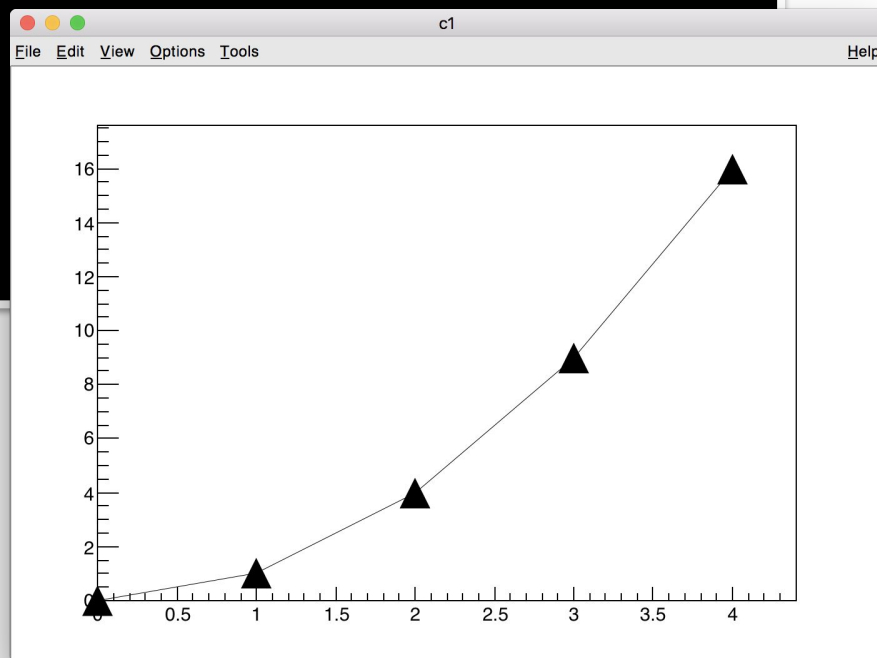
```
root [3] g.SetMarkerStyle(kFullTriangleUp)
```





My First Graph

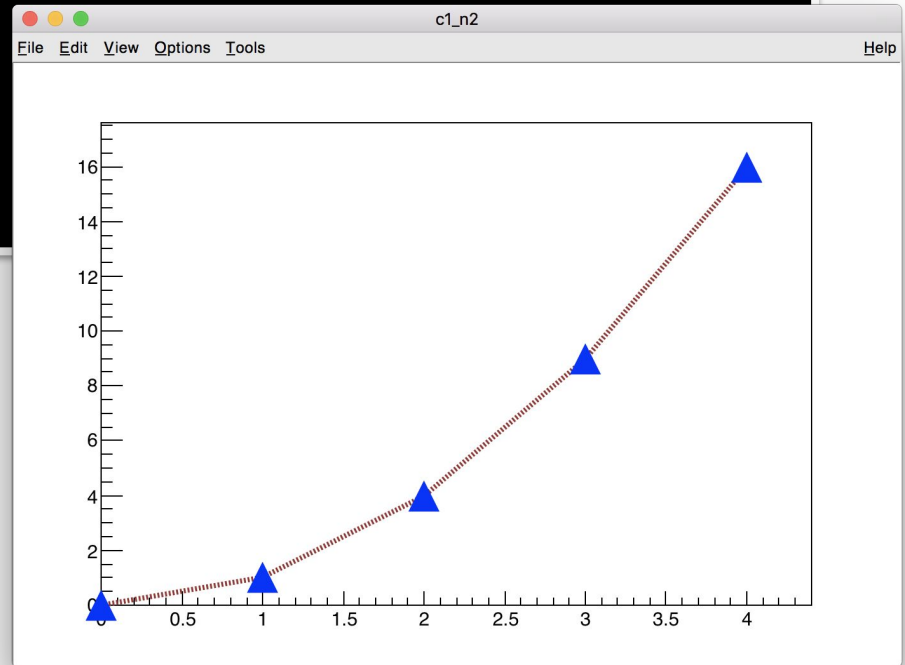
```
root [3] g.SetMarkerStyle(kFullTriangleUp)  
root [4] g.SetMarkerSize(3)
```





My First Graph

```
root [5] g.SetMarkerColor(kAzure)
root [6] g.SetLineColor(kRed - 2)
root [7] g.SetLineWidth(2)
root [8] g.SetLineStyle(3)
```



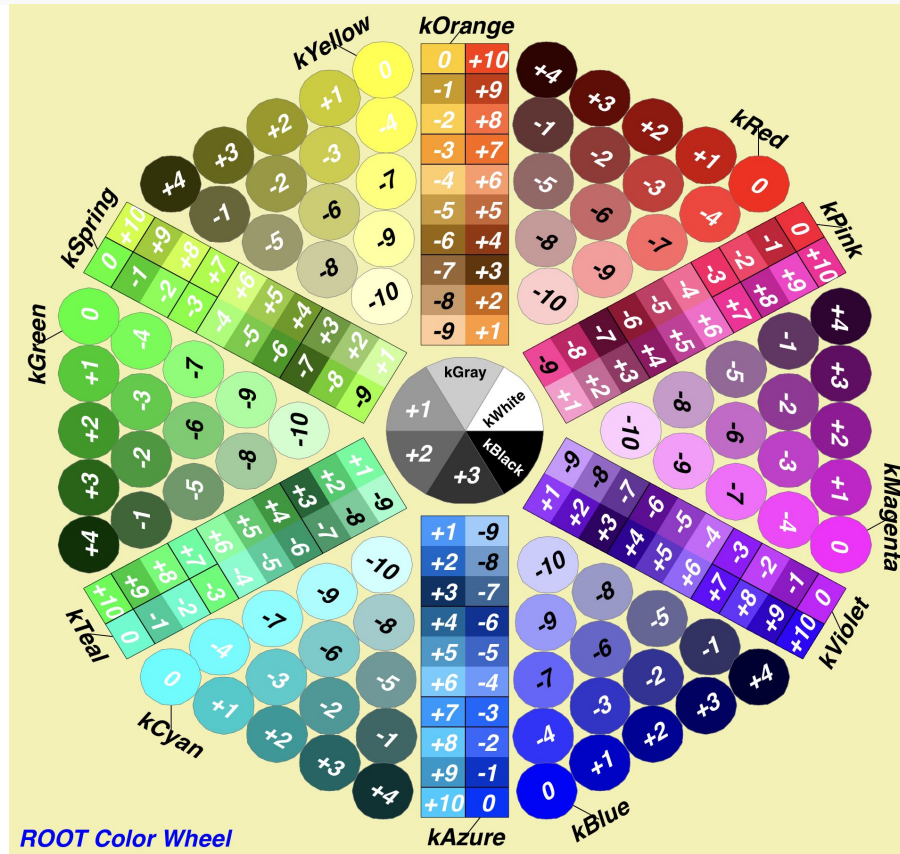
Question:

How do you find information on line styles?

See [TAttLine documentation](#)

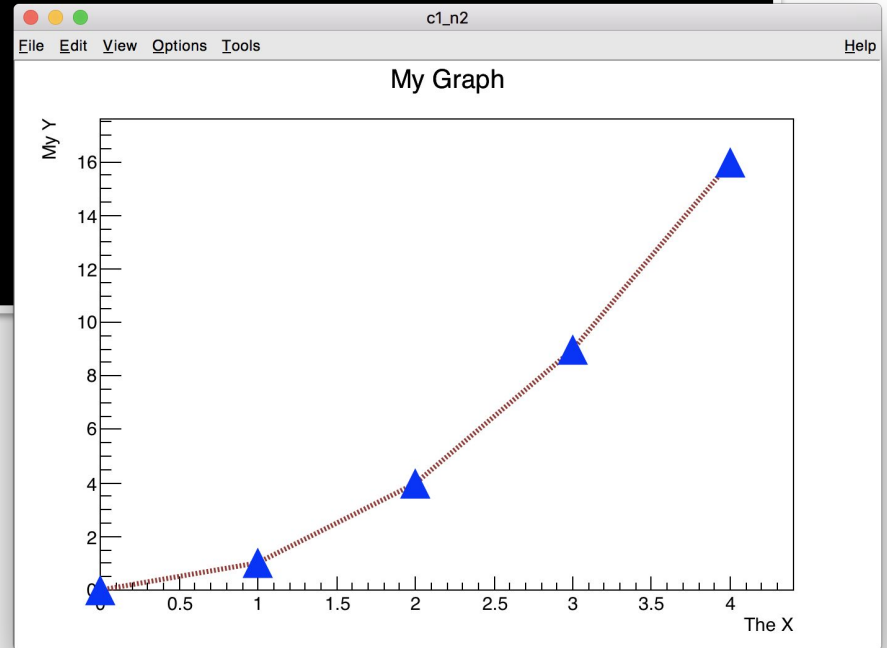


The Colors (TColorWheel)



My First Graph

```
root [9] g.SetTitle("My Graph;The X;My Y")
```



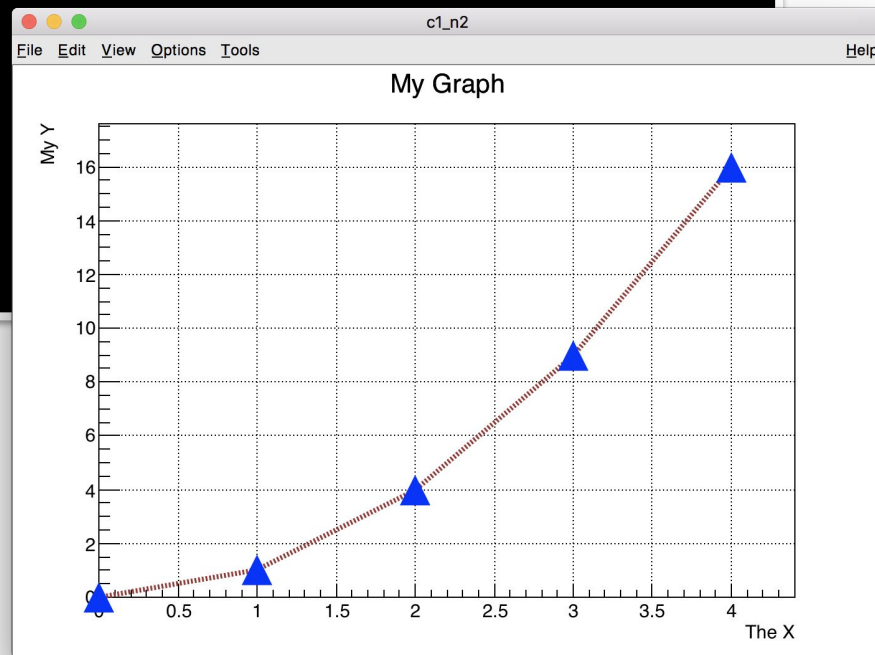
Shortcut:

Directly set titles for x
and y axis.



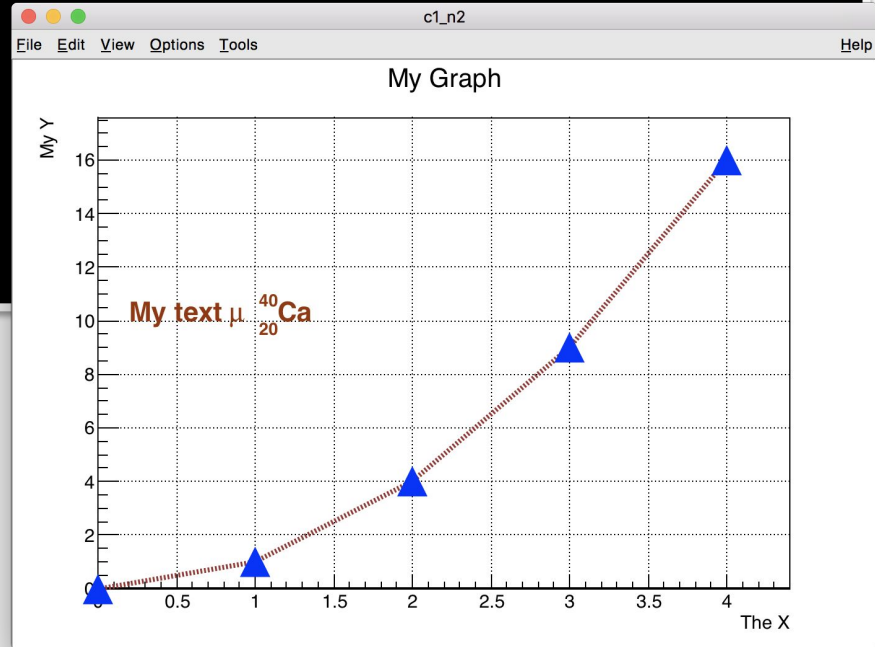
My First Graph

```
root [10] gPad->SetGrid()
```



My First Graph

```
root [10] auto txt = "#color[804]{My text #mu {}^{40}_{20}Ca}"  
root [11] TLatex l(.2, 10, txt)  
root [12] l.Draw()
```





My First Graph

```
root [13] gPad->SetLogy();
```

