REST-for-Physics Framework

1.1 Basic ROOT Concepts

24.01.2023 – Konrad Altenmüller – konrad.altenmueller@unizar.es

Centro de Astropartículas y Física de Altas Energías
Universidad Zaragoza

Session motivation

Our main objective in this session is to provide enough information to understand the basics and to break the walls that may impede others to use ROOT in a first instance.

There are many ROOT components, concepts and features that we are not covering in our lectures.

Those concepts are already properly described inside the ROOT courses, where you will also find working tutorials.

We hope that these lectures will help you as an introduction to ROOT, and as contextualization of the REST-for-Physics framework lectures.

ROOT Basics

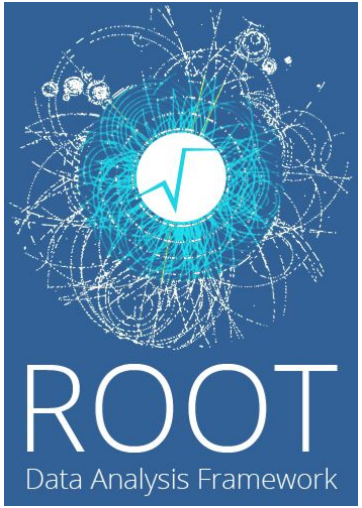
What is ROOT?

- A software framework for data analysis coordinated by CERN
    - Visualization of data (e.g. histograms, graphs, ...)
    - Mathematical transformation of data (e.g. Fourier transforms, peak search,...)
    - Efficient tools to store and process (large amounts of) data
    - Powerful libraries for complex analyses

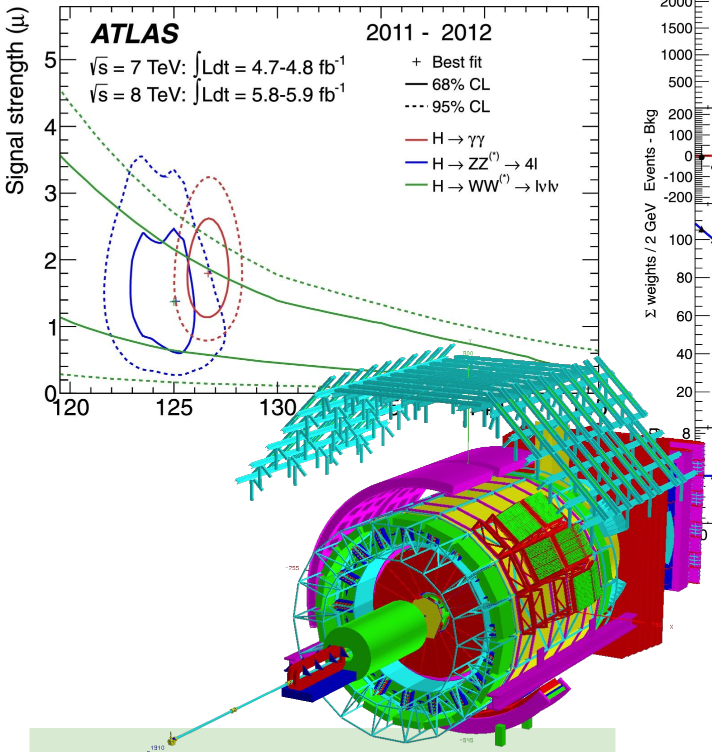
    - Programming interface to be used in custom projects
    - GUI

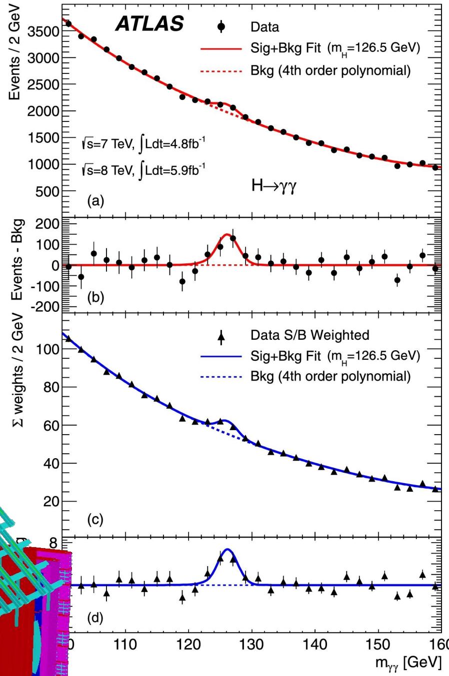
    - Based on C++

- REST is an expansion of the ROOT framework



LHC: Discovery of the Higgs boson

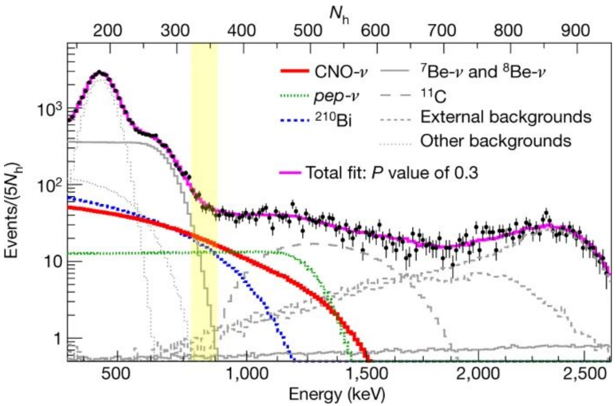




Physics Letters B, Volume 716, Issue 1, 17 September 2012

Alice geometry, produced with ROOT

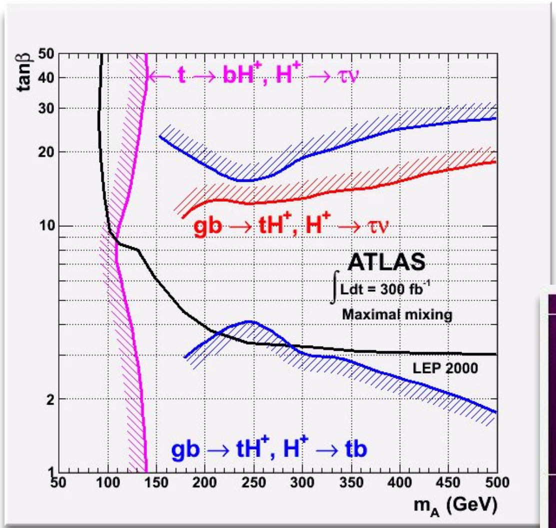
Borexino: multivariate analysis of solar neutrino spectrum, measurement of CNO neutrino flux

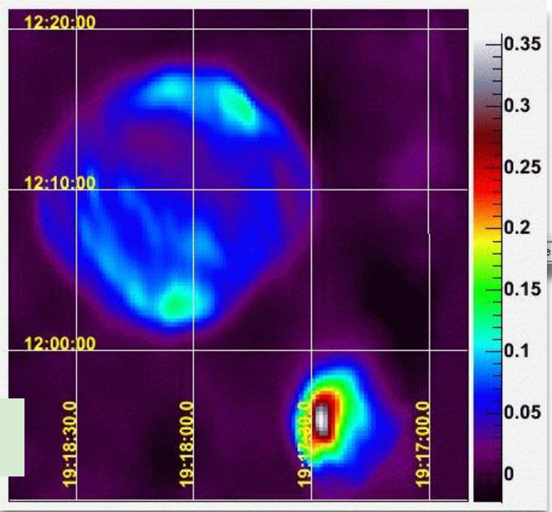


Nature volume 587, pages 577–582 (2020)

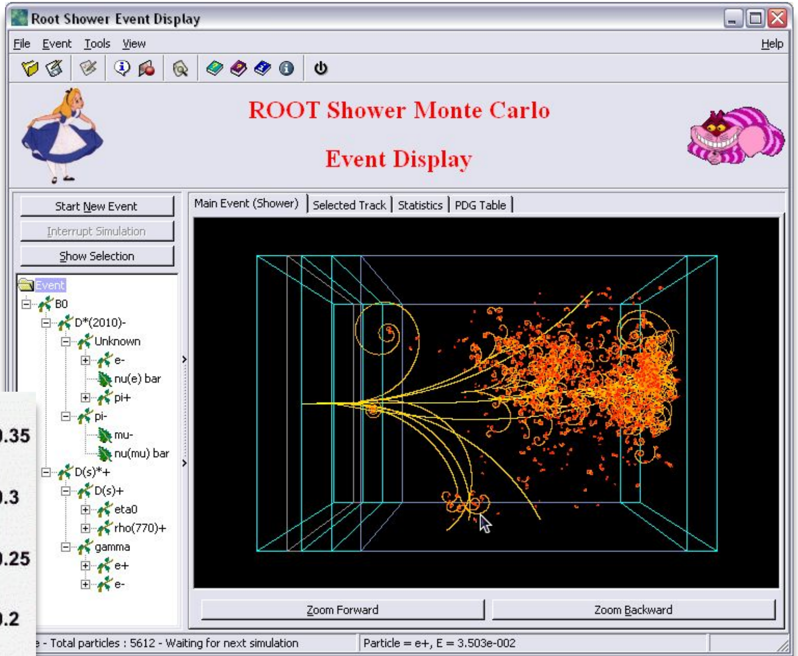
ROOT in physics research

Statistical analysis, exclusion curves





Astrophysics



ROOT GUI to display events

Application of ROOT

For what should I use root?

- Analysis or plots that are often repeated (running macros)
- Established data analysis that runs very fast
- Analysis of large amounts of data
- Customized analysis framework for a large experiment

When it *could* be better to use other software

- Quick and pretty plots (Gnuplot, ...)
- Analysis prototyping, or some quick one-time analysis (more convenient to use some interpreted programming language like Python, MATLAB, or pyROOT)

You should know

- Effective use requires knowledge of basic programming concepts.
- There are a good manual, user guides, tutorials and a helpful and active forum (see links at end of lecture!)

```
konrad@sultan:~$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.26/06                        https://root.cern |
  | (c) 1995-2021, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for linuxx8664gcc on Oct 11 2022, 13:18:00                 |
  | From heads/latest-stable@274b476a                                |
  | With c++ (Debian 8.3.0-6) 8.3.0                                  |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'       |
   ------------------------------------------------------------------

root [0]
```

- Command line based interface:

```
>root
root [0]
```

- You can use it like a calculator:

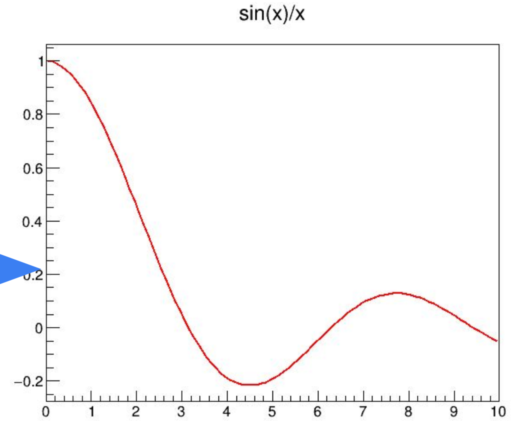
```
root [0] 1 + 1
(int) 2
```

- You can write C++ code

```
root [2] cout << "Hello world" << endl;
Hello world
```

- You can plot functions:

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

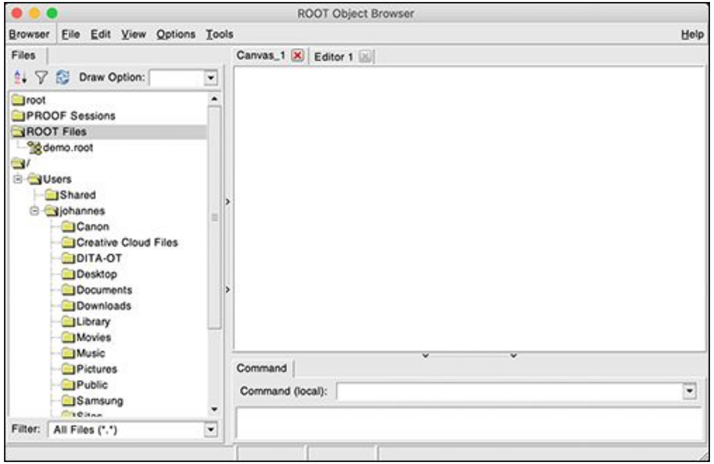


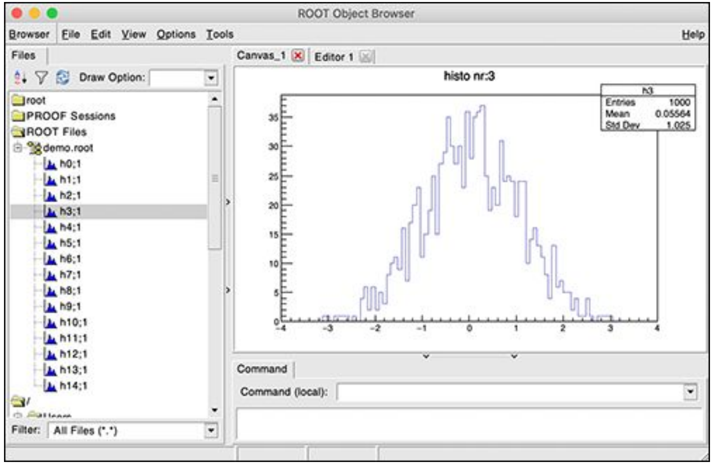
Controlling Root:

- Obtain full list of commands `.?` or `.help`
- Execute a macro `.X <filename.C>` (macro needs to contain a function similar to filename)
- Load a macro `.L <filename.C>` and then call the function within, e.g. `main()`
- Quit root `.q`

- ROOT files contain ROOT objects, e.g. data in form of a **`TTree`** object, or a histogram as a **`TH1D`**
- Can be browsed with a GUI
- Open a file with ROOT, open the object browser:

```
>root file.root
root [0] new TBrowser()
```





Writing ROOT Macros (1)

ROOT Macro: a little program that can be launched within ROOT to do something useful

- Write a text file, e.g `myFunction.C`
- Enter some C++ code
  - Include libraries / classes so that you can use their functions (not necessary if you run the macro within ROOT):

```
#include <TH1D.h>   // ROOT class to create histograms
#include <TMath.h>  // ROOT class that provides math functions
#include <string.h> // C++ class for strings
```

  - Write a function:

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

- Run your Macro with ROOT:

```
> root
root [0] .L myFunction.C
root [1] myFunction()
```

Start root
Load macro / compile it
Execute your function from the macro

The most important classes: `TTree`

Trees

- Trees are optimized for reduced disk space and selecting, high-throughput columnar access with reduced memory usage.
- E.g. used by LHC experiments

A tree consists of a list of independent columns – called **branches**

- A branch can contain values of any type (data types, C++ objects, ROOT objects)
  - For example every entry in the column could contain a digitized signal (e.g. as a vector)
  - When reading the tree, you can select to process only the branches you want → increased efficiency

Branches provide the structure, the data is accessible through objects inheriting from the the **`TLeaf`** class

The most important functions to read trees:

- Launch root with file loaded: `> root myFile.root`
  - Open the GUI: `root [0] new TBrowser()`

    or
  - Examine the tree in the terminal:

```
root [0] .ls                                   # list content of the file  (e.g. trees)
root [1] myTree->Print()                       # get info on branches in the tree

root [2] myTree->Scan()                        # show the data in the tree as a table
root [3] myTree->Scan("myColumn1:myColumn2")   # get a table of the selected branches

root [4] myTree->Show(42)                      # get the values for entry 42

root [5] myTree->Draw("myColumn1")             # draw histogram of column content
```

Add a new branch to a tree and fill it with data:

```
vector<double> myData;                          # Your vector with data
double value;                                   # Initialize the variable to be filled

auto newBranch = myTree->Branch("new branch", &value);   # Create a new branch in the tree


for (auto i : myData){
      value = i;
      newBranch->Fill();                        # Loop over data and fill the branch
}


myTree->Write();                                # Write tree to current directory
```

In modern ROOT, reading / writing trees and high level processing of the data is done with **`RDataFrame`** (see later)

The most important classes: Creating and drawing histograms

There are several classes to deal with histograms, all deriving from **TH1**:

- Most commonly used: **TH1D** for the 1-D case, **TH2D** for the 2-D and **TH3D** for 3-D
- Initialize a histogram:

Number of bins
Lower limit
Upper limit

```
TH1* h1 = new TH1D("h1", "h1 title", 100, 0.0, 4.0);
```

The * means that the variable is assigned to a pointer

- Fill data from your vector `vector<double> data` into the histogram:

```
for (unsigned int i; i < data.size(); i++) h1->Fill(data[i]);
```

- Draw the histogram on a canvas (the canvas is mandatory!), and save it:

```
TCanvas* c = new TCanvas();
h1->Draw();
c->Print("c.pdf");
```

Set up your plot

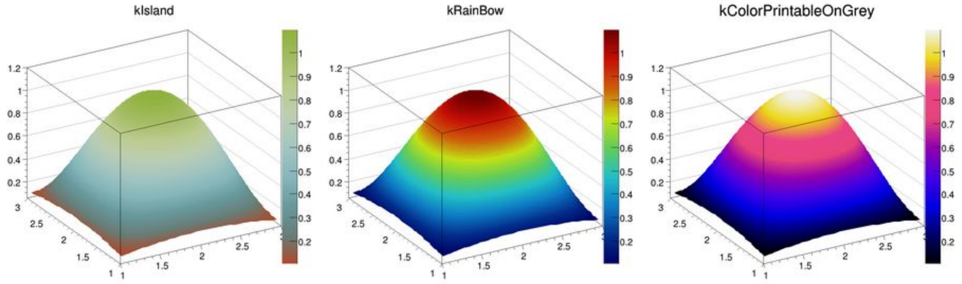
The appearance of your plot is controlled by methods of **`TH1`**, **`TCanvas`**, **`TAxis`**, **`TLegend`**, **`TStyle`**, **`TColor`** ... (very confusing)

- Draw options: `h1->Draw("OPTION");`
  - "SAME" Superimpose on previous histogram on the same canvas / pad
  - "E" Draw error bars

  For 2D histograms:
  - "COLZ" Color depending on count in bin
  - "CONT" Contour plot
  - "SURF" Surface plot

- Set color palette:

```
gStyle->SetPalette(kRainBow);
```



Set up your plot

- Setting log scale (e.g. on Y axis): `c->SetLogY()` (yes, it is a property of the canvas)
- Setting axis labels:

```
h1->GetXaxis()->SetTitle("Energy [keV]");
h1->GetYaxis()->SetTitle("Counts");
```

- Creating a legend:

```
auto legend = new TLegend(0.7,0.5,0.9,0.9);
legend->AddEntry(“h1”, “raw data”);
legend->AddEntry(“h2”, “processed data”);
legend->Draw(“Same”);
```

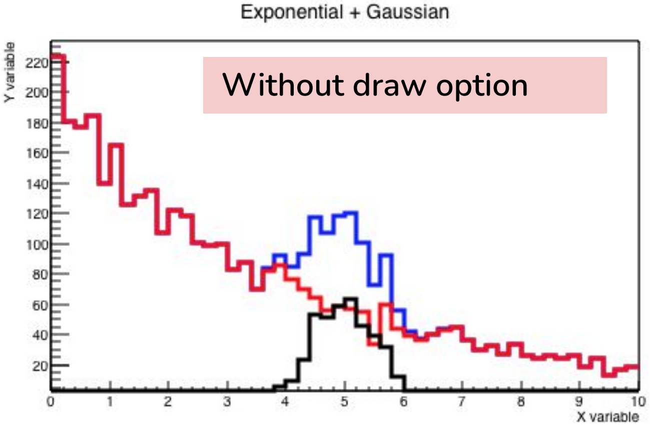
the numbers define the position of the legend box (x1,y1,x2,y2)
→ trial and error

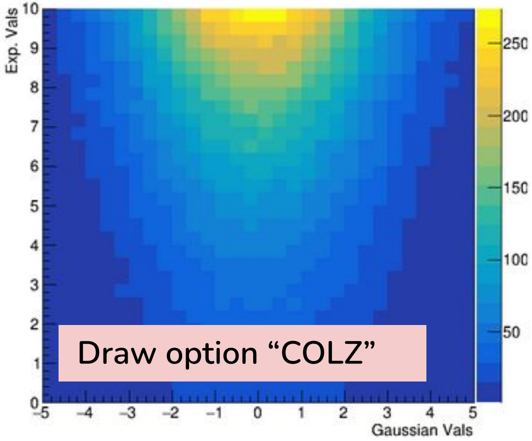
- Choosing line color and style:

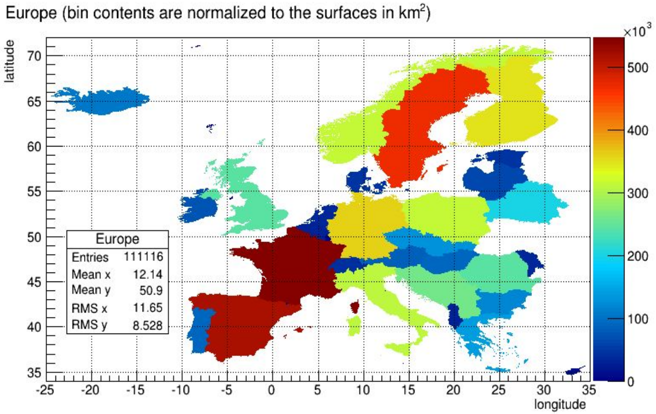
```
h1->SetLineWidth(3);
g1->SetLineStyle(2);        # dashed graph

h1->SetLineColor(2);
h1->SetFillColor(kBlue, 0.35);
h1->SetFillColorAlpha(kBlue, 0.35);
```

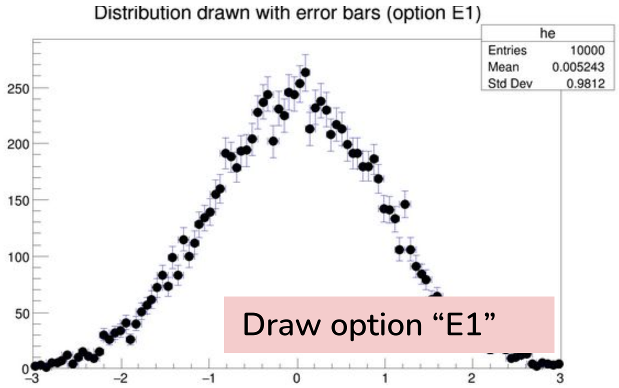
Less ugly colors possible by using `TColor`

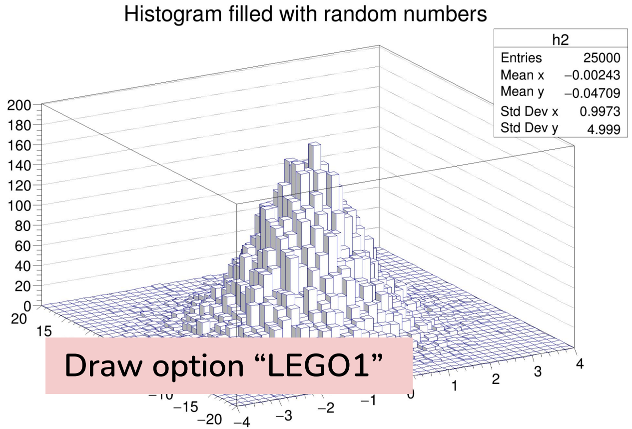
Examples of histograms

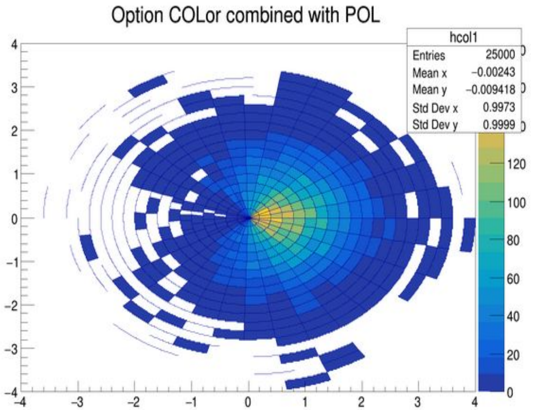






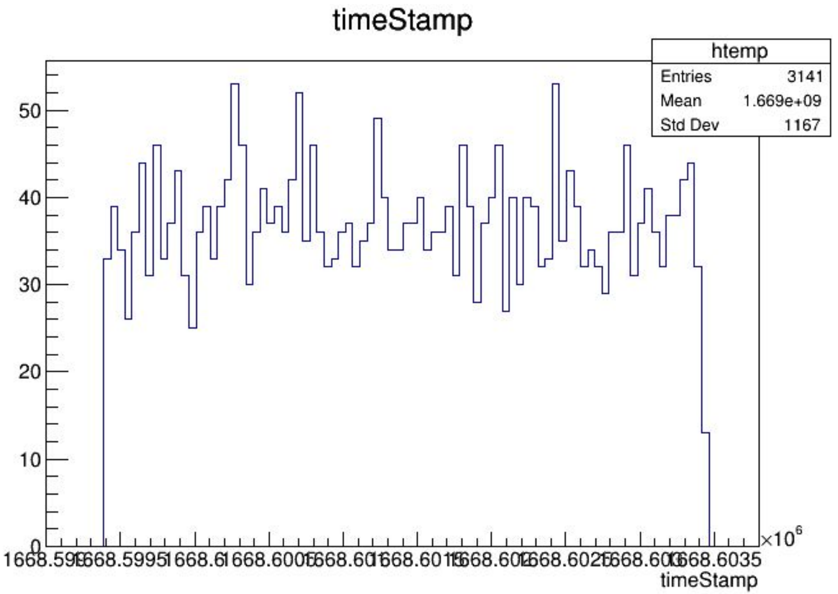






Create a histogram of timestamps from a measurement, and estimate the run duration

```
> root myFile.root
root[0] AnalysisTree->Draw("timeStamp")
```



IAXO-D0: Run 1855, subrun 0

Quick histogram created from the tree, but information is poorly presented

- UNIX time (seconds since Jan 01, 1970 UTC)
- Overlapping tick labels
- Only a single sub-run (can't plot data from multiple files)
- Unknown binwidth → we can't quickly see the count rate

We can do better!

Step 1: extract a `vector<double>` **`timestamps`** from the file. Simplest way is with **`RDataFrame`** (we will look at this class later)

```
ROOT::RDataFrame data("AnalysisTree", “myFile.root”);
auto result = data.Take<double>("timeStamp");
vector<double> timestamps = result.GetValue();
```

Also get max and min value of timestamps:

```
double tmax = data.Max("timeStamp").GetValue();
double tmin = data.Min("timeStamp").GetValue();
```

Step 2: initialize some variables and define some parameters

```
double binw = 60;
int rate_threshold = 10;


int bincount;
int run_duration = 0;
```

// bin width in seconds, set to 1 minute
// rate threshold in counts per binw. If the rate is below, the detector is regarded to off, and the bin is not counted towards the measurement time.
Set to 10 counts / minute

// counts in bin
// total duration of run

Step 4: fill histogram with data

```
TH1D* h = new TH1D("timestamps", "timestamps", (tmax - tmin) / binw, tmin, tmax);
for (auto i : timestamps) h->Fill(i);
```

Example: Histogram

The number of bins need to be an integer → we have to round the duration to a multiple of 60!

Step 4: fill histogram with data

```
TH1D* h = new TH1D("timestamps", "timestamps", (tmax - tmin) / binw, tmin, tmax);
for (auto i : timestamps) h->Fill(i);
```

Step 3: round tmax - tmin to the next larger multiple of the bin width (60)

```
tmax = tmin + (round((tmax - tmin + binw / 2) / binw) * binw);
```

Step 4: fill histogram with data

```
TH1D* h = new TH1D("timestamps", "timestamps", (tmax - tmin) / binw, tmin, tmax);
for (auto i : timestamps) h->Fill(i);
binw = h->GetBinWidth(1);
```

Example: Histogram

Step 3: round tmax - tmin to the next larger multiple of the bin width (60)

```
tmax = tmin + (round((tmax - tmin + binw / 2) / binw) * binw);
```

Step 4: fill histogram with data

```
TH1D* h = new TH1D("timestamps", "timestamps", (tmax - tmin) / binw, tmin, tmax);
for (auto i : timestamps) h->Fill(i);
binw = h->GetBinWidth(1);
```

Step 5: check each bin if it is above the threshold and if yes, add it to the run duration

```
int nbins = h->GetNbinsX();
for (unsigned int i = 0; i < nbins; i++){
      bincount = h->GetBinContent(i);
      if (bincount > rate_threshold) run_duration += binw;
}
```

Step 3: round tmax - tmin to the next larger multiple of the bin width (60)

```
tmax = tmin + (round((tmax - tmin + binw / 2) / binw) * binw);
```

Step 4: fill histogram with data

```
TH1D* h = new TH1D("timestamps", "timestamps", (tmax - tmin) / binw, tmin, tmax);
for (auto i : timestamps) h->Fill(i);
binw = h->GetBinWidth(1);
```

Step 5: check each bin if it is above the threshold and if yes, add it to the run duration

```
int nbins = h->GetNbinsX();
for (unsigned int i = 0; i < nbins; i++){
      bincount = h->GetBinContent(i);
      if (bincount > rate_threshold) run_duration += binw;
}
```

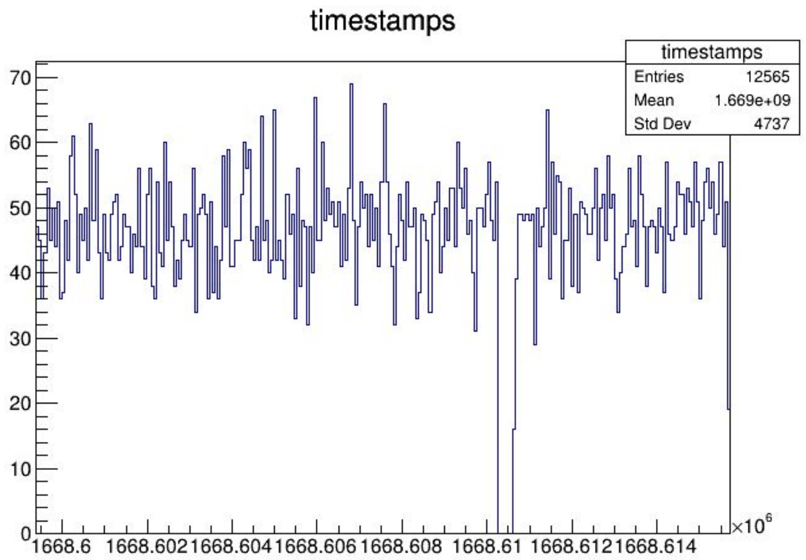
A histogram is not only useful for plots, but also for calculations!

The variable **run_duration** now contains the duration of the measurement in seconds!

Step 6: draw the histogram

```
TCanvas* c = new TCanvas();
h->Draw();
```

```
cout << "Run duration:" << run_duration << " seconds = " << (double)run_duration/60 << " minutes = " <<
(double)run_duration/3600  << " hours" << endl;
```




```
Run duration:15900 seconds = 265 minutes
= 4.41667 hours
```

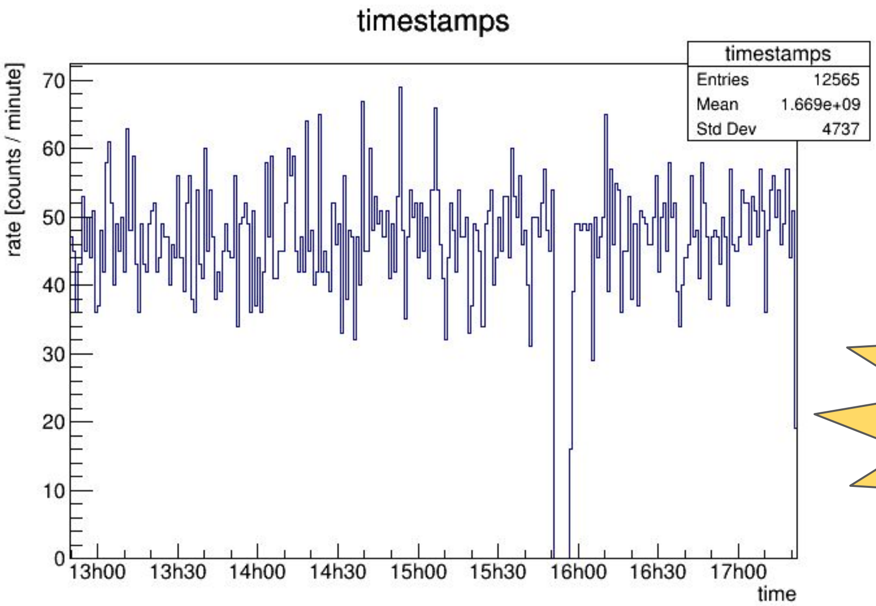
- Meaningful bin width (we see counts per minute)
- Still bad Xticks

Step 6: draw the histogram

```
TCanvas* c = new TCanvas();
h->GetXaxis()->SetTimeDisplay(1);
h->GetXaxis()->SetTitle(“time”);
h->GetYaxis()->SetTitle(“rate [counts / minute]”);
h->Draw();
```

Step 6: draw the histogram

```
TCanvas* c = new TCanvas();
h->GetXaxis()->SetTimeDisplay(1);
h->GetXaxis()->SetTitle("time");
h->GetYaxis()->SetTitle("rate [counts / minute]");
h->Draw();
```



Ready for a Nature paper!

The most important classes: Fitting

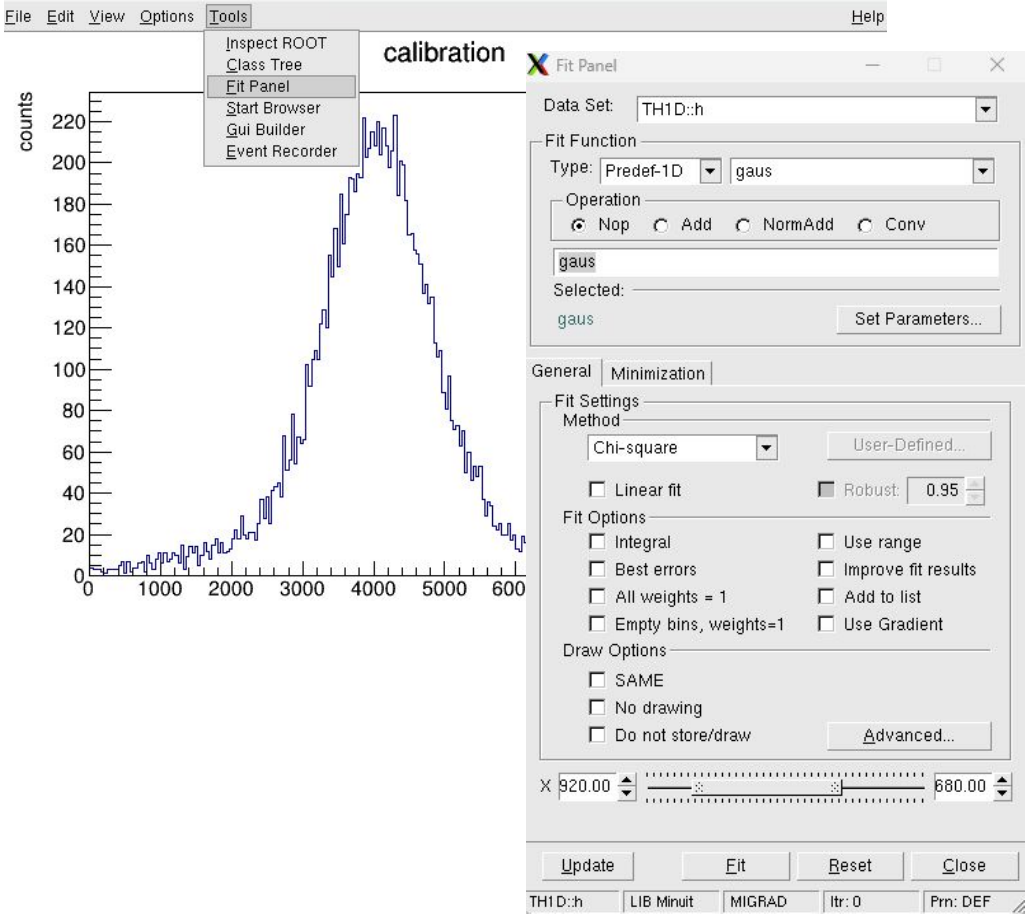
Several options to choose from:

- `Fit()` method, implemented for **`TH1`**, **`TGraph`** and more
  - A large number of fitting algorithms to choose from
  - You can use it with a GUI after drawing a histogram

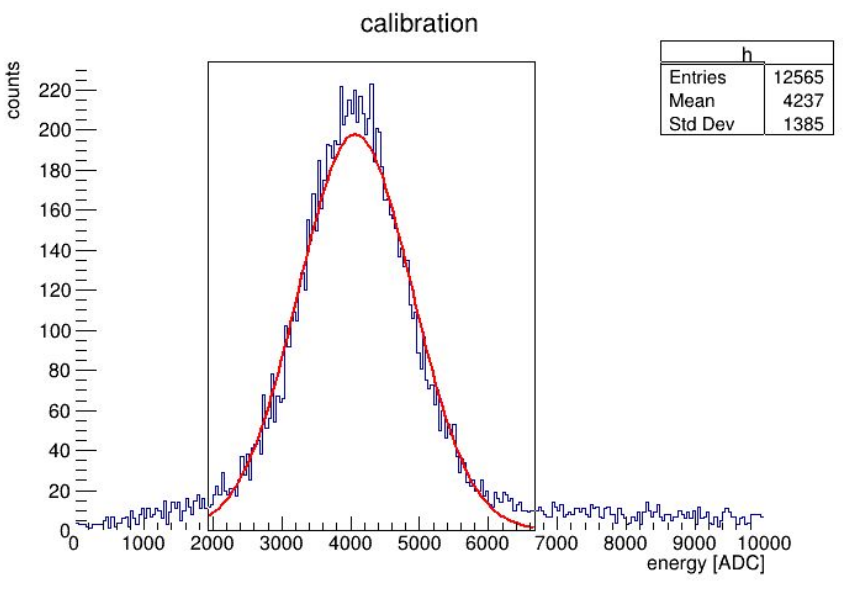
- Minimization packages:
  - E.g. **`minuit2`**, which is very useful for multivariate analysis (actually a C++ minimization engine that can be used stand-alone or with other software)

- **`ROOT::Fit`** classes
  - For more specialized fitting methods
- **`ROOFit`** library
  - More physics based
  - Modelling of expected data, “toy Monte Carlo” studies

Using the fit panel in a histogram



Result




```
FCN=238.091 FROM MIGRAD    STATUS=CONVERGED      71 CALLS          72 TOTAL
                     EDM=1.24005e-10    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  Constant     1.97848e+02   2.64557e+00   1.47841e-02   2.23009e-07
  2  Mean         4.06596e+03   8.38836e+00   6.33087e-02  -1.82997e-06
  3  Sigma        8.28022e+02   7.79012e+00   1.73490e-05   1.37552e-03
```

What you should know about fitting

What algorithm are there?

- MIGRAD: “standard” in ROOT
  - Heavily relies on first derivatives of data → can crash if your model contains e.g. a step
- SIMPLEX:
  - Creates a triangle (in the 2D case) around a point and checks in which direction the function value is lower and continues there
  - Not depending on derivatives → very robust
  - Can take a long time, but it WILL find your minimum

What to keep in mind with constrained fits:

- The errors of the fitted parameters are obtained by varying around the minimum
- If you constrain a parameter, and the minimum value is close to the constraint, the returned error might be meaningless

Fit a custom function

- Define a custom function (normal distribution):

```
double fitf(double* x, double *par){
      double fitval = par[0] * TMath::Exp(-TMath::Power(*x - par[1],2)/(2*TMath::Power(par[2],2)));
      return fitval;}
```

- Create a **TF1 object** using the fit function:

```
TF1 *func = new TF1("fit",fitf,0,10000,3);
```

range   number of pars

- Set start parameters:

```
func->SetParameters(200,4000,800);
```

- Set parameter limits:

```
func->SetParLimits(1,0,10000);
```

limit for par 1

- Give the parameters names:

```
func->SetParNames("Constant","Mean_value","Sigma");
```

- Call `TH1::Fit`:

```
h->Fit(func, "L", "", 2000, 6000);
```

fit option “likelihood method” (chi squared)

- Many more options to customize the fit (algorithms, fitting methods, ...)!

Many predefined functions available in **`TF1`**, e.g. `h->Fit("gaus");`

- **`gaus`**, **`gausn`** normal distribution (the latter normalized)
- **`landau`**, **`landaun`** landau distribution
- **`expo`** exponential distribution
- **`pol1,...9`**, **`chebychev1,...9`** polynomials of various degrees

Many fitting options (see [here]):

- **“L”** log likelihood method (chi squared)
- **“B”** when you use a predefined function, but want to set the parameters
- **“I”** uses bin integral instead of bin center
- **“E”** improves error estimation with the Minos technique
- **“S”** returns full result including covariance matrix
- **“Multithread”**
- …

The most important classes: `RDataFrame`

`RDataFrame` offers a modern, high-level interface for analysis of data stored in `TTree`, CSV and other data formats, in C++ or Python.

- Easy multithreading
- Modular and flexible work flow
- Quickly perform common analysis tasks
- Fully customize the data processing by integrating any C++ code
- Lazy actions: operations are not executed on the spot, but when a results is accessed

Super simple syntax (if you do simple things):

```
ROOT::RDataFrame df("TreeName", "filename.root");
ROOT::RDF::RNode df2 = ROOT::RDataFrame(0);
df2 = df.Define("energy_calibrated","energy * 0.00145");
df2 = df2.Define("radius","xMean*xMean + yMean*yMean");
df2 = df2.Filter("radius<100");
h = df2.Histo1D({"h","energy_calibrated",100,0,10},"energy");
h->Draw();
```

// Load tree from file into dataframe
// Initialize new dataframe for processed data
// Define a new column with calibrated energy
// Define a new column with event radius
// Cut events outside a certain radius
// Create a histogram
// Draw it

Column definitions and filters accept as expression

- A string with the condition, e.g.

  ```
  df2.Filter("radius < 10 && energy < 10")
  ```

- Or any function or callable object, allowing complex calculations, e.g. a C++ lambda
  - Integrate RDataFrame in a function:

```
double calculation(double l,double u){
      df = df.Filter([l,u](double c) {return l <= c && c <= u;}, {“columName”});
      // calculate something from the data
      }
```

  - Define custom functions to define a new column:

```
// assuming a function with signature:
double myComplexCalculation(const RVec<float> &datapoints);
// we can pass it directly to Define
auto df_with_define = df.Define("newColumn", myComplexCalculation, {"datapoints"});
```

The most useful RDataFrame functions

- `Define` Define a new column
- `DefinePerSample` Define a new column with parameters depending on the input file (when you combine several files)
- `Filter` Filter your data
- `Snapshot` Save some (filtered) columns to a file with a root tree
- `Count` Get the number of events in the dataframe that survived so far
- `Histo1D, Histo2D,...` Produce a histogram from selected columns
- `Max, Min, Mean, Sum` Self explanatory
- `Take` Take a column and return a vector with its content
- `GetValue` Lazy actions produce a pointer with the result. The value is accessed with this command

TMath provides an extensive library of commonly used math functions, constants, operations, ...

For example

- **Abs, Min, Max, Mean,** ... common operations
- **Pi, G, C, Na,** ... constants
- **Sin, Cos, Tan,** ... trigonometric functions
- **BesselI, BreitWigner, Erf,** ... functions
- **DegToRad,** ... conversions
- **Gcgs, GhbarC,** ... print commonly used units like $g^3cm^{-1}s^{-2}$

Using ROOT with Python: PyRoot

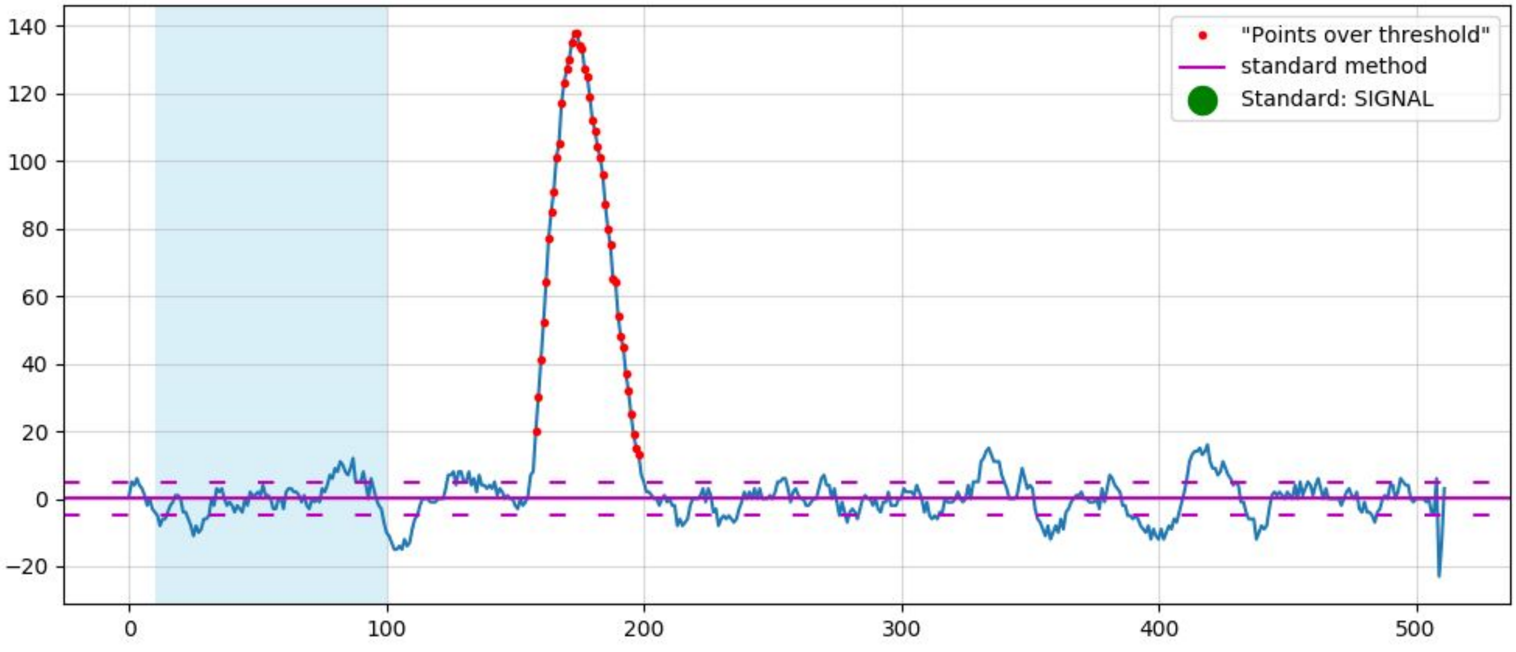
With PyROOT you can access the full ROOT from Python while benefiting from the performance of the ROOT C++ libraries.

- Very useful for analysis prototyping (hit “run” and get your result)
- Also REST libraries can be accessed

To start:

```
import ROOT
import REST
```

Example / demonstration: use PyROOT together with REST to test parameters for the IAXO-D0 noise reduction



Writing ROOT Macros (2): Combining everything we learned

Exercise: calibrating the detector data

Data Files: `/data/R01855/*root`

The files contains data from IAXO-D0. In this data a ^{55}Fe source was used.

Task: Calibrate the detector data.

Preparation:

- what lines does the ^{55}Fe source provide? → table of isotopes

Exercise: calibrating the detector data

1. Create a macro. Include the necessary classes.
2. Define the fitting function.
3. Define the main function.
4. Load the data into a RDataFrame.
5. Initialize a filtered dataframe.
6. Filter the data. Only keep the data from a spot with 10 mm radius in the detector center (necessary columns: hitsAna_xMean and hitsAna_yMean).
7. Create a histogram of the energy (column tckAna_MaxTrackEnergy).
8. Use TSpectrum::Search() to find peak candidates. Get the position of the peak to use it to define the start parameters of the fit.
9. Create a TF1-object from the fitting function
10. Configure the fit by interacting with the TF1-object: setting parameter values (start parameters), setting parameter constraints, naming the parameters, ...
11. Execute the fit by calling TH1::Fit().
12. Draw the histogram and the fitted function on a canvas.
13. Access the best fit parameters and calculate the calibration factor. Note that the fit has to be performed with the "S" option to extract the parameters.
14. Define a new column in the dataframe with the calibrated energy. Use a C++ lambda function in the definition.
15. Create a new plot with the calibrated energy. Try out some options to adjust style and colors to your liking.
16. (BONUS) Create a 2-D histogram with the detector hit map.

Exercise: calibrating the detector data

- Create a macro. Include the necessary classes.
- Define the fitting function

```
??? fitf(???* x, ???* par){

     ??? fitval = par[0] * exp[(*x - par[1])²/(2*par[2]²)];
     return fitval;
}
```

- Define the main function
- Load the data into a RDataFrame
- Initialize a filtered dataframe

```
ROOT::RDataFrame df(???);
ROOT::RDF::RNode df_filtered = ROOT::RDataFrame(0);
```

- Filter the data. Only keep the data from a spot with 10 mm radius in the detector center
- Create a histogram

```
auto h1 = df_filtered.???({"name","title", nbins, min, max},"columnName");
auto h = h1.???(); // to convert the RDataFrame ROOT::RDF::RResultPtr<TH1D> to TH1D
```

Exercise: calibrating the detector data

- Use `TSpectrum::Search()` to find peak candidates. Get the position of the peak to use it to define the start parameters of the fit.

```
TSpectrum *s = new TSpectrum();
int sigma = 10; // a parameter of the peak finder that defines the sensitivity
Int_t nfound = s->Search(???);
printf("Found %d candidate peaks to fit\n", nfound);
auto peak = s->???(); // get x position of peak
```

- Create a TF1-object from the fitting function

```
TF1 *func = new TF1(???);
```

- Configure the fit by interacting with the TF1-object: setting parameter values (start parameters), setting parameter constraints, naming the parameters, ...
- Execute the fit

```
auto fitresult = h.Fit(func, "fit option", "", range, range);
```

- Draw the histogram and the fitted function on a canvas.

Exercise: calibrating the detector data

- Access the best fit parameters and calculate the calibration factor.

```
??? mean = fitresult->Parameter(???);
??? calfactor = line energy / mean;
```

- Define a new column in the dataframe with the calibrated energy. Use a C++ lambda function in the definition.

```
df_filtered = df_filtered.Define("calibratedEnergy", [capture](double c){function;},{"columName"});
```

- Create a new plot with the calibrated energy. Try out some options to adjust style and colors to your liking.

- (BONUS) Create a 2-D histogram with the detector hit map.

ROOT Resources

How to learn more:

- https://root.cern/get_started/
  - Beginner's guide
  - Extensive manual
- Tutorials with many helpful examples: https://root.cern/tutorials/

- ROOT forum: https://root-forum.cern.ch/

ROOT
Data Analysis Framework

How does fitting work?

- Minimization problem.
  The proper fitting method (i.e. the value you want to minimize) depends heavily on the nature of your data!
  - Least squares: minimize sum of squared residuals (data point - model)
    - Not suitable for histogram data, because it doesn't take into account the statistical error of the data points
    - Very sensitive to outliers
  - Chi squared: minimizes $$X^2 = \sum_{i=1}^{k} \frac{(x_i - m_i)^2}{m_i}$$
    - Accounts for statistical errors, assuming a normal distribution
  - Maximum likelihood estimation: can take the Poissonian error of the data
    - When the bin counts are very low