An introduction to ROOT

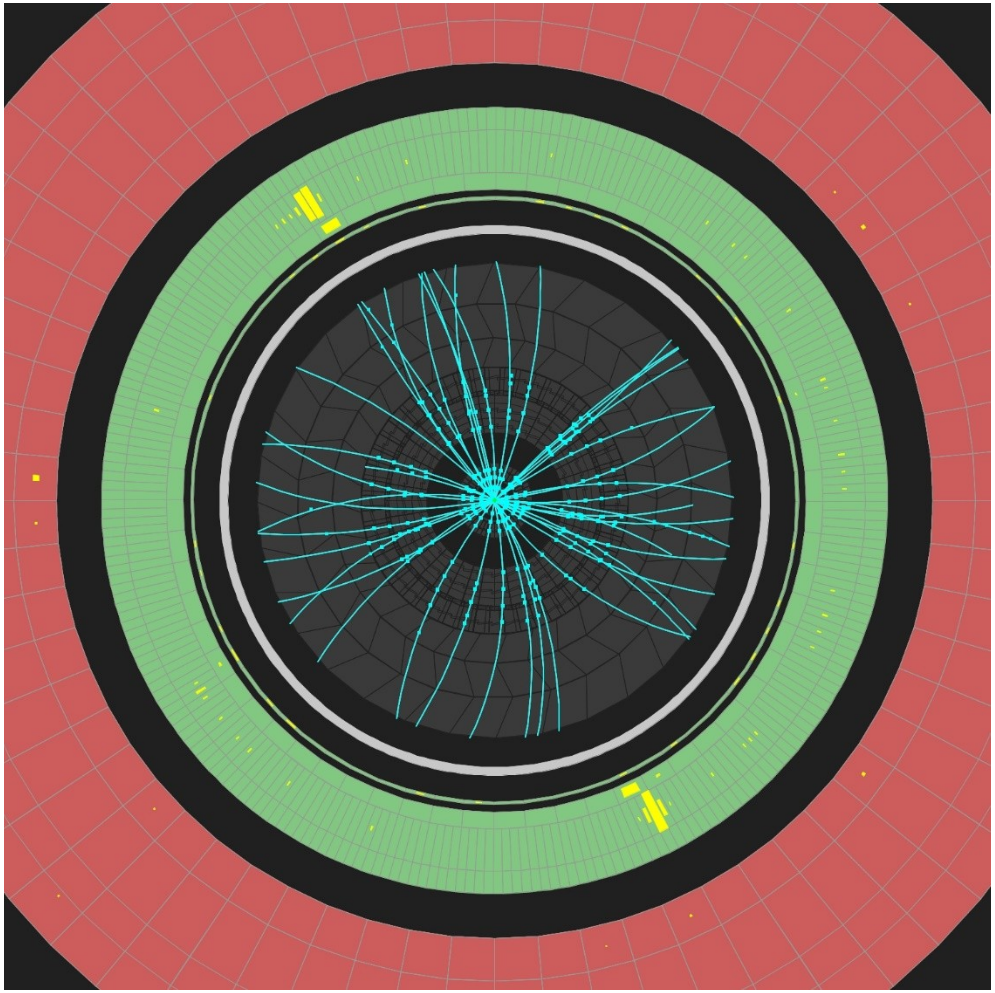
- Lecture 7 of MNXB01
  - Inspired by Oxana’s lecture from last year
- Outline
  - Computing in science
  - ROOT intro
  - ROOT examples

Linux and C++ are tools

- In physics computing is an integral part of the way we do science
  - Calculations – numerical integration, FFT, etc.
  - Simulations – event generators, detector studies
  - Data storage – saving/accessing experimental results
  - Reconstruction – detector signals → physical quantities
  - Analysis – getting results out of the
  - Visualization – results and event displays
  - + many more: e.g. Machine learning, Monitoring, Chip/electronics programming, Readout

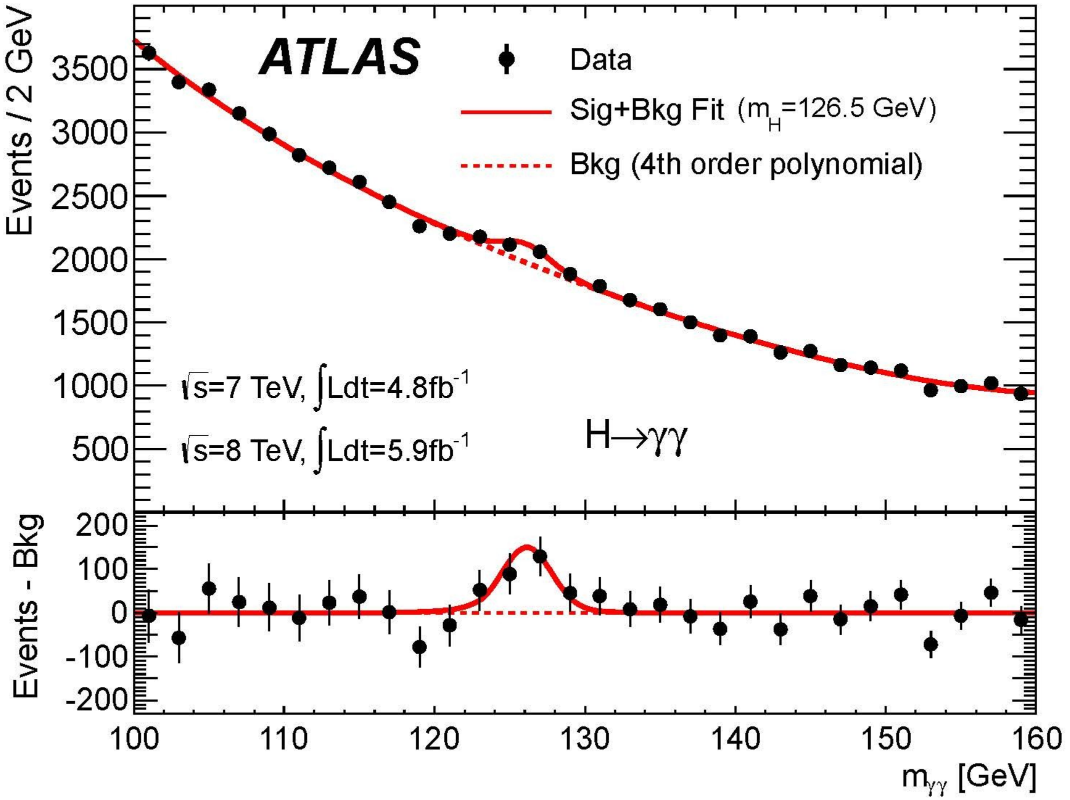
A Higgs → 2 photons candidate

- Reconstruction, e.g., tracks
- Visualization



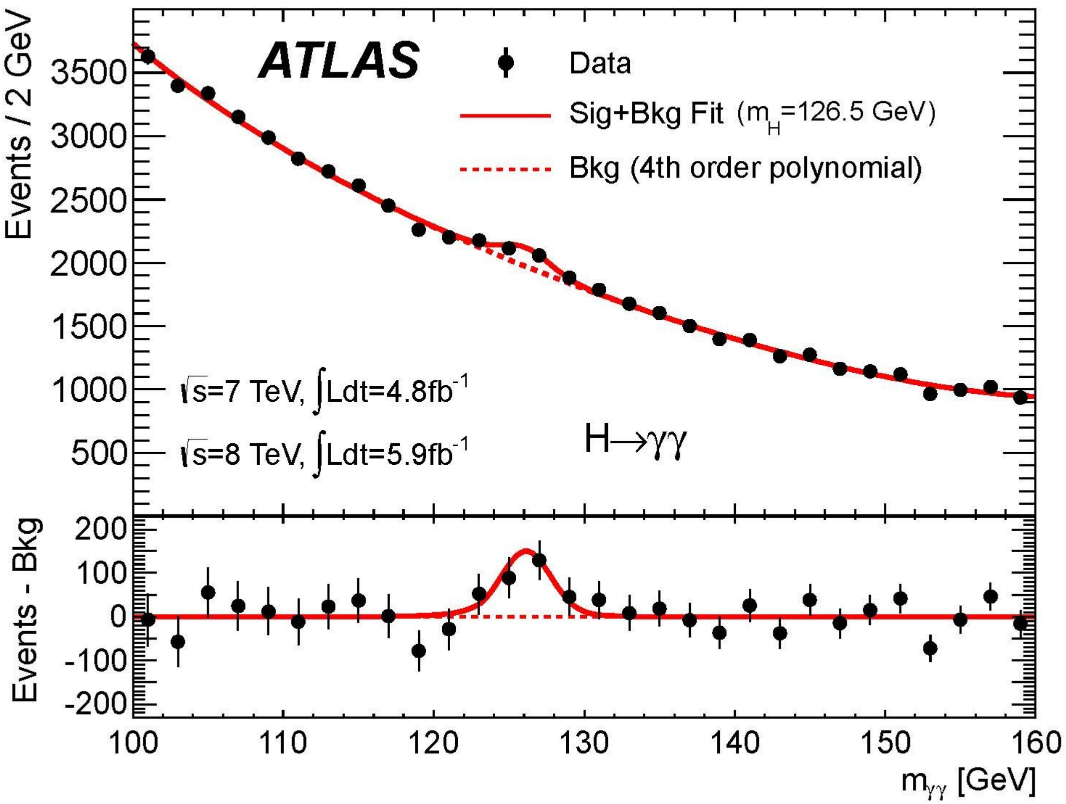
Higgs discovery

- Data analysis
- Visualization



What computing elements were required to make this plot?

Think about the full path from detector to publication



The full path

- Online
  - Detector control system
  - Data acquisition
  - Online monitoring
- Offline
  - Reconstruction
  - Simulation
  - Quality Assurance
  - Data analysis

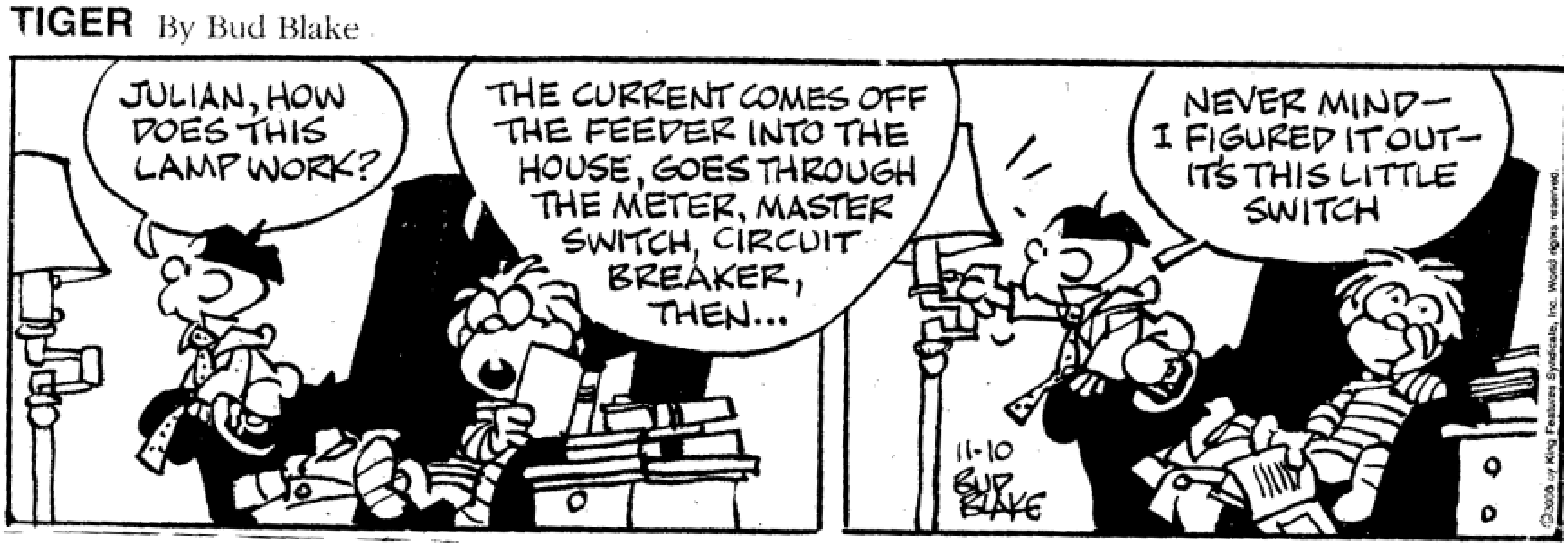
What we will focus on this and next week

- Simulations
- Analysis
- Visualization
- Data storage

Linux and C++ is not enough

- Inefficient to start all projects from scratch and develop the code we need for each project
- We can use existing frameworks to help us
- This week and next we will use ROOT

Frameworks are smart



- But they also hide a lot of things under the hood (be careful) and can restrict what you want to do (choose wisely)

ROOT – an object-oriented analysis framework

- We will focus on **<u>ROOT</u>** – a specialized analysis framework developed at CERN
  - <u>Free</u> and available for almost all platforms (LGPL 2.1 license)
  - Relies on ROOT data format (a hierarchical database, actually)
  - Has built-in C++ interpreter – you can use C++ in ROOT , like Python

- A complete ROOT tutorial normally takes several days; many such tutorials can be found on-line
  - We will give a short introduction, re-using some official slides

What is ROOT?

- The ROOT system is an object-oriented (OO) framework for large scale data analysis (and even simulation)
  - Written in C++
  - Provides, among others,
    - An efficient hierarchical OO **database**
    - A C++ interpreter (**CINT**)
    - Advanced statistical **analysis** (multi-dimensional histogramming, fitting, minimization and cluster finding algorithms)
    - **Visualization** tools
    - And much, much more
  - The user interacts with ROOT via a graphical user interface, the command line or scripts
  - The command and scripting language is C++ (thanks to the embedded CINT C++ interpreter)
    - Large scripts can be compiled and dynamically loaded

How to get and set up ROOT

- On Ubuntu, it is available from **`universe`** repositories
  - Install package **`root-system`**
- Otherwise, go to

  http://root.cern.ch

  and download what you need
  - Current stable version is **6.xx**
    - Versions 5.xx are widely used, too (does not matter for simple code)
  - Installation from source is for brave people: will take some time and may produce odd error messages
- You can configure your ROOT preferences using **`~/.rootrc`** file
- There are also scripts **`rootlogon.C`**, **`rootlogoff.C`** (executed on logon and logoff) and **`rootalias.C`** (loaded on logon)
- History is saved in **`~/.root_hist`** file
- Read ROOT documentation for details (or Google "ROOT getting started")

Built-in ROOT C and C++ interpreter: **CINT**

- Main goal: provide a framework for C and C++ “scripting” – somewhat like Python
- As a separate software, CINT code is available under an Open Source license
- It implements about 95% of ANSI C and 90% of ANSI C++
- It is robust and complete enough to interpret itself (90000 lines of C, 5000 lines of C++)
- Has good debugging facilities
- Has a byte code compiler
- In many cases it is faster than tcl, Perl and Python
  - Large scripts can still be compiled for optimal performance (always recommended)
- CINT is used in ROOT:
  - As command line interpreter
  - As script interpreter
  - To generate class dictionaries
  - To generate function/method calling stubs
- In ROOT, the command line, script and programming language become the same
  - But it does accepts also non-C++ statements (avoid this)

Working with ROOT

- Type root at the command line prompt
  - This starts a new “shell” from which you can work with data by using C++ instructions and scripts
  - To exit, type .q
  - To run a script (e.g. a tutorial), type
    .x <scriptname.C>
  - To load functions from a file, type
    .L <scriptname.C>
    - **To compile (best!!!!!): .L <scriptname.C>+**
  - To execute a regular shell command, type
    .! <command>

Simple ROOT warm-up examples

```
root [] 35 + 89.3
(const double)1.24299999999999997e+02
root [] float x = 45.6
root [] float y = 56.2 + sqrt(x);
root [] float z = x+y;
root [] x
(float)4.55999984741210938e+01
root [] y
(float)6.29527778625488281e+01
root [] z
(float)1.08552780151367188e+02

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

- Note that by default ROOT uses double precision
- TF1 is a ROOT class for functions of 1 variable (1-dimensional functions)
  - `Draw` is a method of the class
  - Use TAB to show all methods: `root [] f1.<TAB>`

Some ROOT conventions

- ROOT classes begin with **`T`** (like **`TF1`** above)
- Non-class types end with **`_t`** (for example, **`Int_t`**)
- Constants begin with **`k`** (for example, color red: **`kRed`**)
- ROOT uses machine-independent types, e.g.:
  - **`Bool_t`** – Boolean (0=false 1=true)
  - **`Char_t`** – signed character 1 byte
  - **`Int_t`** – Signed integer 4 bytes
  - **`Short_t`** – Signed short integer 2 bytes
  - **`Long64_t`** – Signed long integer 8 bytes
  - **`Float_t`** – Float 4 bytes
  - **`Double_t`** – Float 8 bytes (a.k.a. double precision)
- But it also accepts int, float etc. BUT these can be machine dependent

Scripts in ROOT

- Un-named Script: a simple short-cut (like a bash script)
  - Starts with **{** and ends with **}**
  - All variables are in the global scope
  - No class definitions
  - No function declarations
  - No parameters
- **Named Script: essentially, a C++ program (recommended)**
  - C++ functions
  - Scope rules follow standard C++
  - Function with the same name as the file is executed with a *.x*
  - Parameters
  - Class definitions (derived from a compiled class at your own risk)

Examples of scripts

- "Macro" is a historical way of denoting scripts in ROOT
- **Un-named** Macro: `hello.C`

```
{
  cout << "Hello" << endl;
}
```

- **Named** Macro: `say.C`

```
void say(char * what = "Hello")
{
  cout << what << endl;
}
```

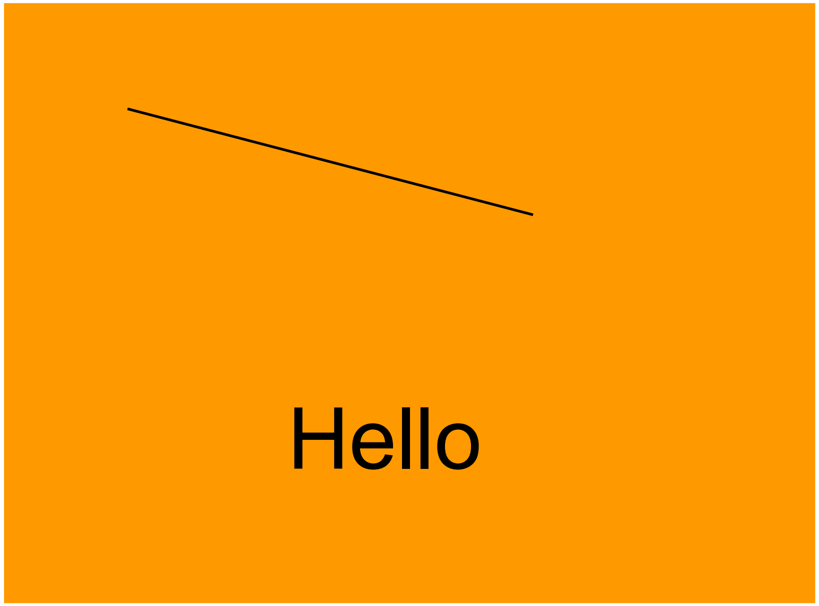
- Executing the Named Macro

```
root [3] .x say.C
Hello
root [4] .x say.C("Hi there")
Hi there
```

Graphics in ROOT

- ROOT is no Photoshop, and graphics is designed for scientific results representation

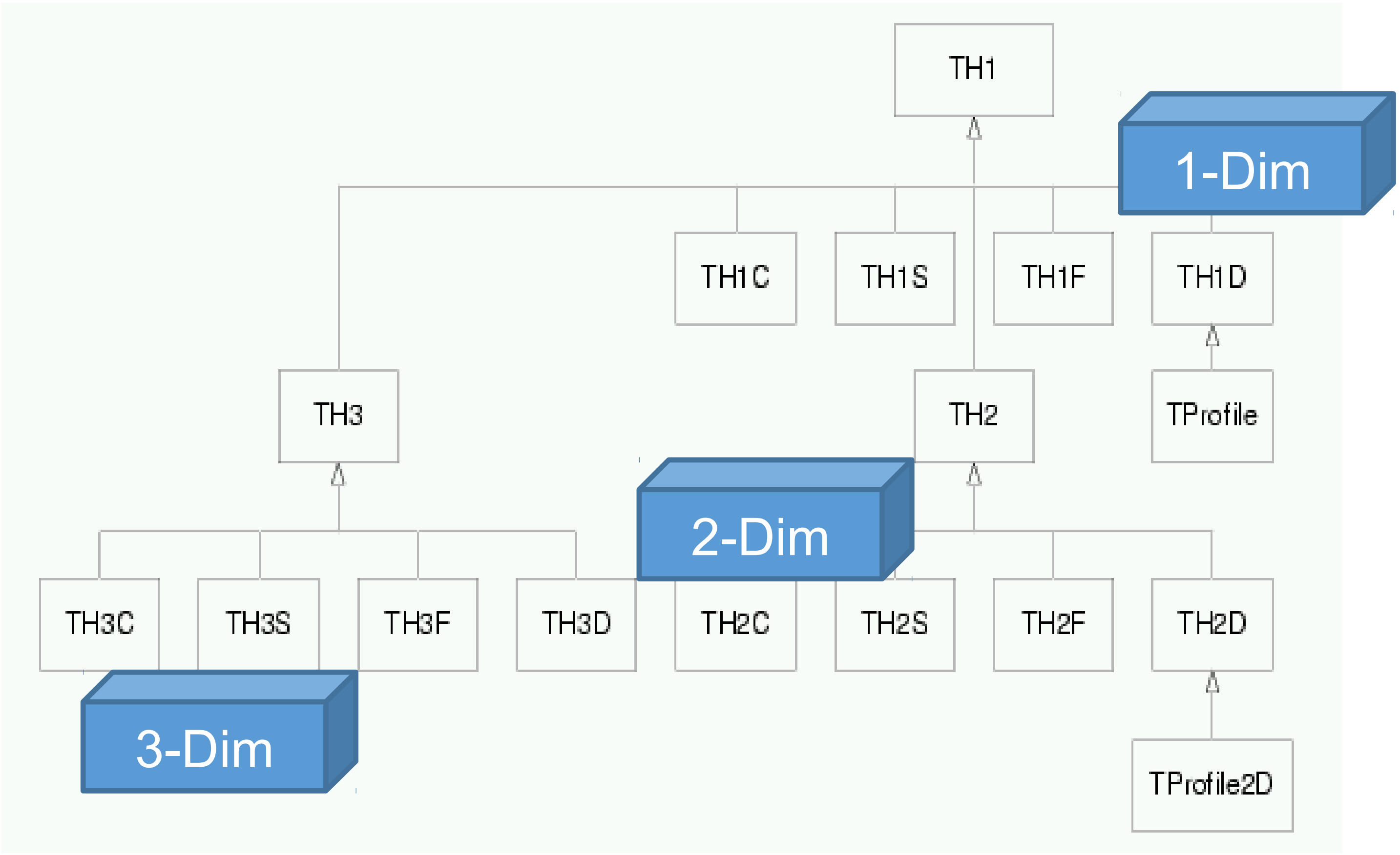
```
root [] TLine myline(.1,.9,.6,.6)
root [] myline.Draw()
root [] TText mytxt(.5,.2,"Hello")
root [] mytxt.Draw()
```



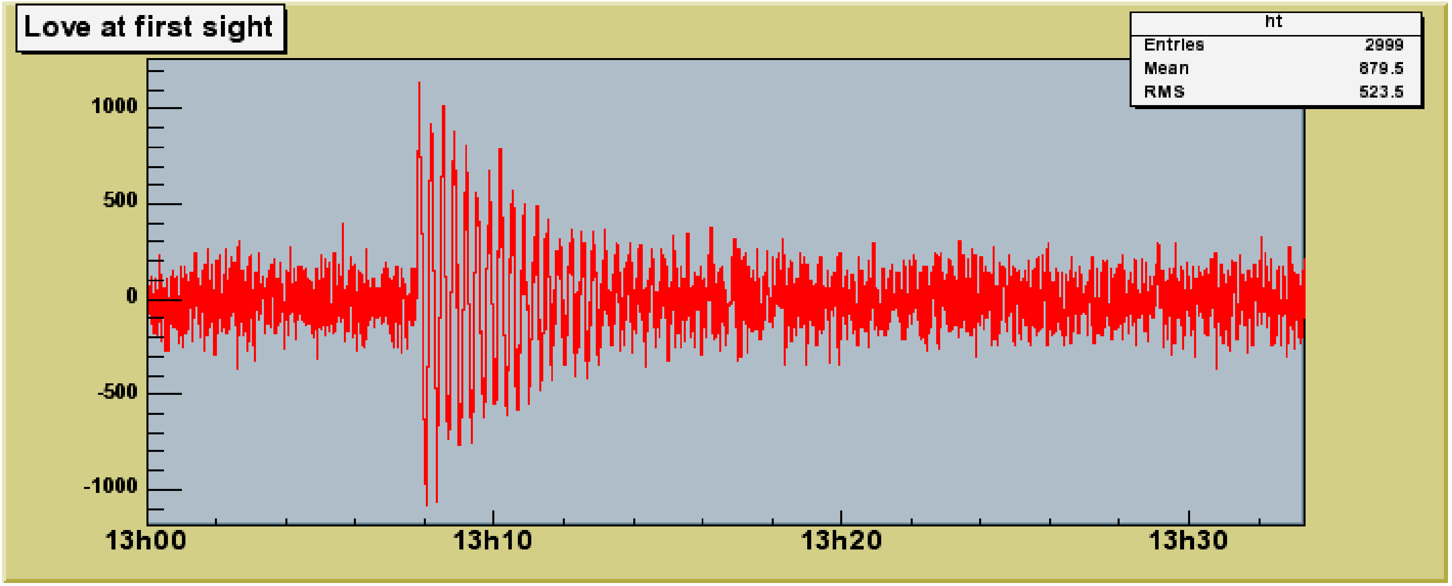
- The `Draw` function adds the object to the list of primitives of the current graphics "**pad**"
- If a pad does not exist, it is automatically created with a default range [0,1]
- When the pad needs to be drawn or redrawn, the `Paint` function is called

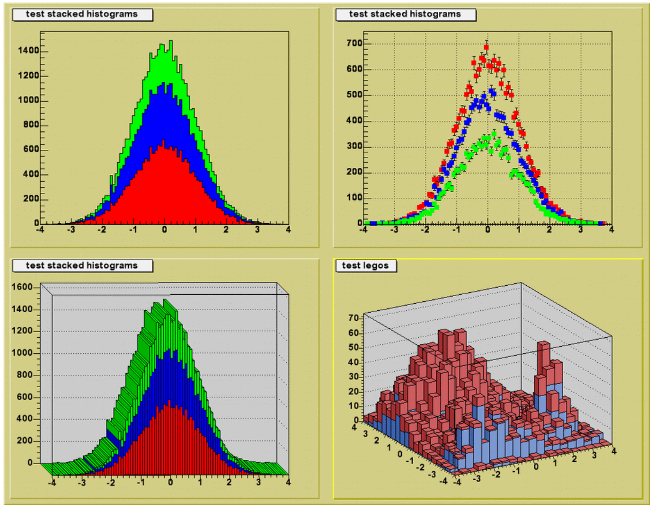
Histogram classes in ROOT

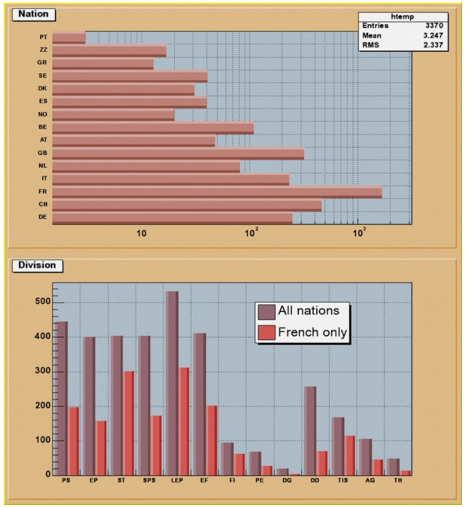
- 1- and 2-dimensional histograms are most common
  - **C**, **S**, **F** and **D** stand for the content type: **D** is double and recommended
- Profile histograms are 2-dim histograms “compressed” into 1-dim by calculating mean values
- 3-dimensional histograms are essentially graphs



Examples of histograms



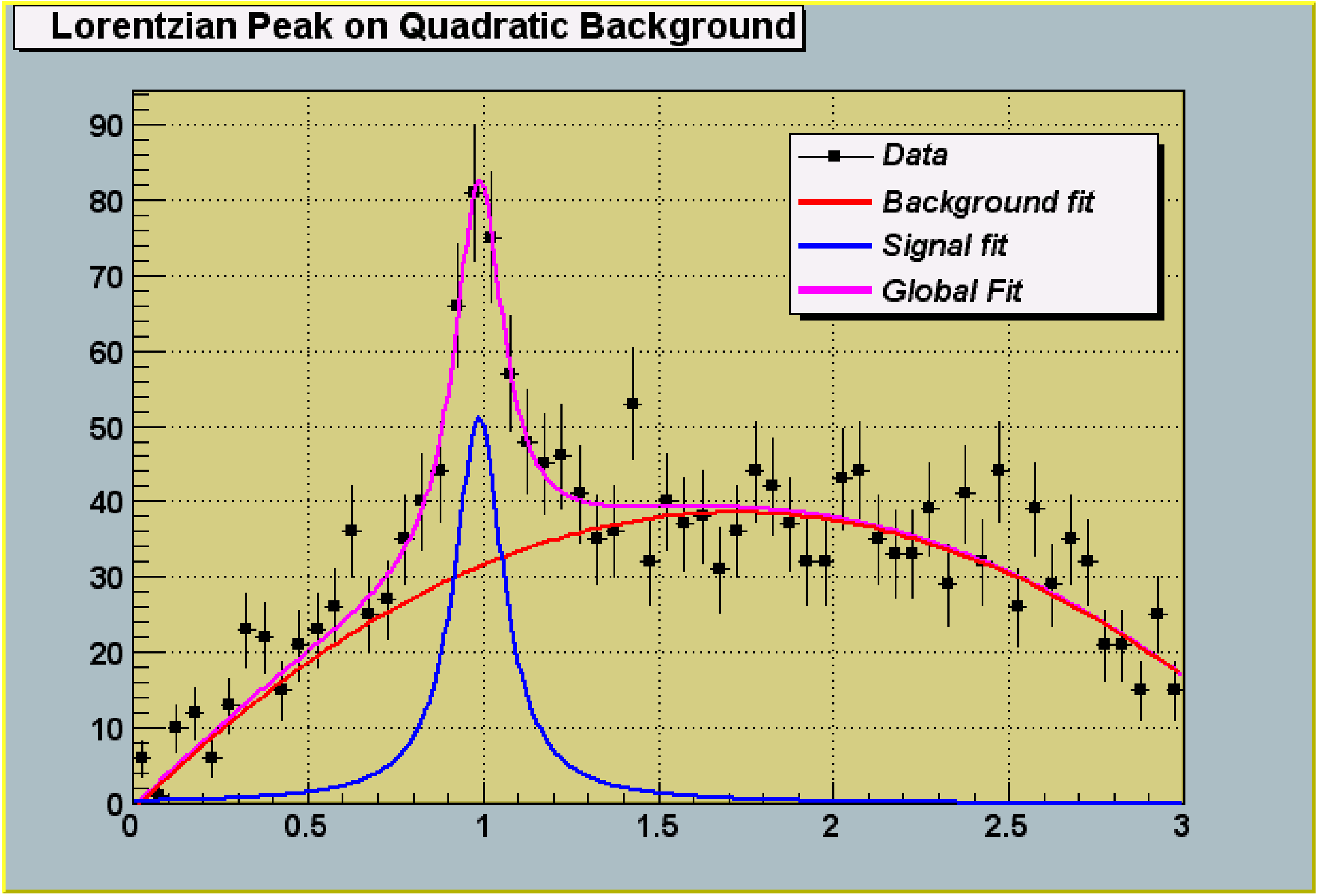




Fitting in ROOT

- Histograms can be fitted with any function via `TH1::Fit`. Two fitting algorithms are supported: **Chi-square** method and **Log Likelihood**
- The user functions may be of the following types:
  - standard functions: `gaus`, `landau`, `expo`, `poln`
  - combination of standard functions; `poln + gaus`
  - A C++ interpreted function or a C++ precompiled function
- When an histogram is fitted, the resulting function with its parameters is added to the list of functions of this histogram. If the histogram is made *persistent* (saved as a file), the list of associated functions is also persistent.
- One can retrieve the function/fit parameters with calls such as:
  - `Double_t chi2 = myfunc->GetChisquare();`
  - `Double_t par0 = myfunc->GetParameter(0);` //value of 1st parameter
  - `Double_t err0 = myfunc->GetParError(0);` //error on first parameter

Fitting example



Random numbers and histograms

- **`TH1::FillRandom`** can be used to randomly fill an histogram using either of:
  - the contents of an existing **`TF1`** analytic function
  - another histogram

- Example: the following two statements create and fill an histogram 10000 times with a default Gaussian distribution of mean 0 and sigma 1:

```
TH1F h1("h1","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",10000);
```

- **`TH1::GetRandom`** can be used to return a random number distributed according the contents of an histogram

The tree most important classes

- Histogram: binned → well defined and easy to manipulate
  - Examples: TH1D, TH2D
- Graph: unbinned → can be used for anythings (but more difficult to manipulate)
  - Examples: Tgraph, TGraphErrors
- Function (can e..g fit both histograms and graphs)
  - Exaples: TF1, TF2

Interactive examples

- If you have a full installation you have tutorials under $ROOTSYS/tutorials
- You can also find them online: https://root.cern.ch/doc/master/group__Tutorials.html

Explanation of day 1 and 2 exercises

- http://www.hep.lu.se/staff/christiansen/MNXB01/