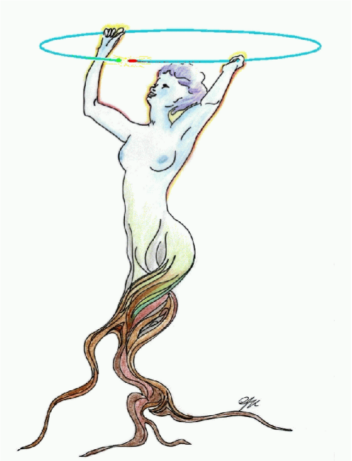
ROOT:
a HEP tool for data analysis

L. Bianchini (INFN Sezione di Pisa)

ROOT is...



- **A scientific software toolkit for data analysis**
  - User code (⇒ macros) is C++ using ROOT classes (⇒ T***).
    - TFile ⇒ basic I/O
    - TH1F ⇒ 1D-histogram with floating-point precision
    - TF1 ⇒ a generic real-valued 1D-function

    ...
  - Macros can be either interpreted on-the-fly or compiled & executed
  - ROOT classes/functions can be used via automatic Python bindings (⇒ **PyROOT**)

```
import ROOT
f = ROOT.TFile.Open(...)
h = ROOT.TH1F()
```

- **Project developed & maintained by CERN** (currently at ROOT 6.X)
  - Online manual, tutorials, forum: https://root.cern.ch/
  - Open-source: can either compile the source code, or use binaries

I/O & operations in ROOT

- **Input/output**
  - Data stored in binary files (⇒ .root). Many compressions possible (ZLIB, LZMA, LZ4, ZSTD)
  - Can save graphic objects into most used formats: .eps, .png, .pdf, .jpeg, .C (⇒ C++ code)
- **Operations**
  - Commands issued into the **ROOT prompt**:

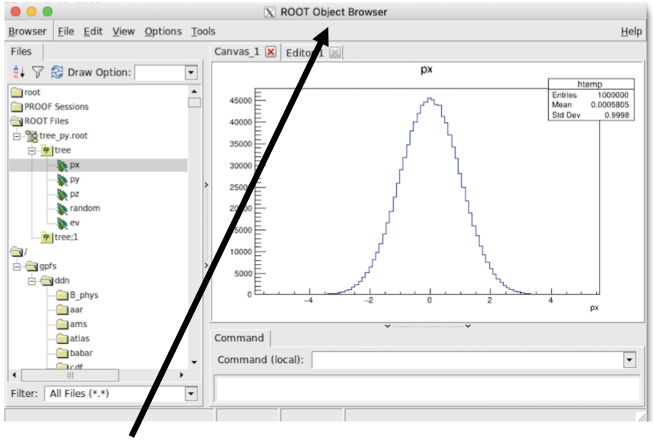
```
> root
root [0] TFile* f = TFile::Open("my_file.root")
```

  - Code interpreted on-the-fly (⇒ Cling):

```
> root
root[0] .L my_macro.C   // func() is defined here
root[1] func()
root[2] .x snippet.C  // C++ code within '{}' brackets
```

  - Code compiled, linked, and executed (⇒ ACLiC)

```
> root
root[0] .L my_macro.C++ // generates my_macro_C.so
root[1] func()
```



ROOT comes with its own **GUI (⇒ TBrowser)** for browsing file content and prompt plotting

Interesting features (from a HEP perspective)

- **Efficient I/O of N-dimensional data of arbitrary type (⇒** TTree **)**
  - Can store objects, arrays, and pointers of basic types, C structures, C++ classes

```
root[0] tree->Scan()
*****************************************************************************
*      Row   *     Var0       *     Var1       *    Var2       *     Var4       *
*****************************************************************************
*      0     *   0.2832548    *  1.0946351     *  1.2784594    *      0         *
*      1     *   1.2402626    * -0.285862      *  1.6199687    *      1         *
```

COLUMNS ⇒ variables

ROWS ⇒ events

- **Native mathematical environment (⇒** ROOT::Math:: **)**
- **Linear algebra** (⇒ TLorentzVector, TMatrixD, TDecomp[SVD,LU,QRH…])
- **N-dimensional minimization and integration (⇒** Minuit, GSL**)**
- **Monte Carlo studies** (random number generation)
- **MVA's (**aNN, BDT, kNN, likelihood, …**) for regression/classification (⇒** TMVA **)**
- **Unfolding (⇒** TUnfold **)**

A concrete example: ROOT-based data for CMS

CMS data stored into ROOT files with different content (data tiers)

(detector….) RAW ⇒ RECO ⇒ AOD ⇒ MiniAOD ⇒ NanoAOD **(…. analysis)**

- Look e.g. at a MiniAOD Monte Carlo file (~50 kB/event):

34705FB5-CE8C-E911-B0E4-00144F45BD0E.root (1 runs, 14 lumis, 13258 events, 565781558 bytes)

```
KEY: TTree    MetaData;1
KEY: TTree    ParameterSets;1
KEY: TTree    Parentage;1
KEY: TTree    Events;1
KEY: TTree    LuminosityBlocks;1
KEY: TTree    Runs;1
```

SW version, file ID, …

Configuration of the driver

Parentage of this file

Run & lumi-block list at which these data were taken

The TTree containing the actual data

A concrete example: ROOT-based data for CMS

Looking at the Event TTree:

User-defined classes

STL containeres of user-defined classes

STL containeres of STL classes

Type	Module	Label	Process
GenEventInfoProduct	"generator"	""	"SIM"
LHEEventProduct	"externalLHEProducer"	""	"SIM"
edm::TriggerResults	"TriggerResults"	""	"SIM"
vector<reco::GenMET>	"genMetTrue"	""	"SIM"
vector<reco::GenMET>	"genMetTrue"	""	"HLT"
vector<pat::Electron>	"slimmedElectrons"	""	"PAT"
vector<pat::IsolatedTrack>	"isolatedTracks"	""	"PAT"
vector<pat::Jet>	"slimmedJets"	""	"PAT"
vector<pat::Jet>	"slimmedJetsAK8"	""	"PAT"
vector<pat::Jet>	"slimmedJetsPuppi"	""	"PAT"
vector<pat::MET>	"slimmedMETs"	""	"PAT"
vector<pat::Muon>	"slimmedMuons"	""	"PAT"
vector<pat::Photon>	"slimmedPhotons"	""	"PAT"
vector<reco::GenJet>	"slimmedGenJets"	""	"PAT"
vector<string>	"slimmedPatTrigger"	""	"PAT"
unsigned int	"bunchSpacingProducer"	""	"PAT"
double	"fixedGridRhoAll"	""	"RECO"

Setup the environment

All the material can be also found here:
https://github.com/bianchini/SNS_DAS

1. **Open a terminal**
2. **Create a working directory**

```
> mkdir ROOT_Tutorial
> cd ROOT_Tutorial/
```

1. **Download the material**

```
> wget https://github.com/bianchini/SNS_DAS/archive/master.zip
> unzip master.zip; cd SNS_DAS-master/
```

For this tutorial, ROOT is made available inside a singularity container.

- To open ROOT:

```
> singularity exec /opt/cern_root/cern_root.sif root -l
```

- To execute Python macros that import the ROOT module:

```
> singularity exec /opt/cern_root/cern_root.sif python –c "import ROOT"
```

FOR TODAY, REMEMBER TO PUT singularity exec /opt/cern_root/cern_root.sif BEFORE ANY CALL TO ROOT/Python IN THE TERMINAL

Basic ROOT commands

- Open ROOT
  > root –l // don't display initial ROOT logo
  > root –b // batch mode (N.B. no graphic windows can be opened)
- List content of the ROOT current directory
  root[0] .ls // objects in memory will be displayed
- Exit ROOT
  root[0] .q
- The ROOT prompt provides auto-completion (TAB), forward-backward search (↑,↓) and regex search in the history (CTRL-R)

Exercise 1: create a dataset (⇒ tree.C)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void make_tree(Int_t nevents=100000)
{

  TFile* file = new TFile("tree_C.root","RECREATE", "");
  TTree* tree = new TTree("tree","A simple Tree");

  Float_t px, py, pz;
  Double_t random;
  Int_t ev;

  tree->Branch("px",&px,"px/F");
  tree->Branch("py",&py,"py/F");
  tree->Branch("pz",&pz,"pz/F");
  tree->Branch("random",&random,"random/D");
  tree->Branch("ev",&ev,"ev/I");

  for (Int_t i = 0 ; i<nevents ; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    tree->Fill();
  }

  tree->Write();

}
```

Exercise 1: create a dataset (⇒ tree.C)

No need to set full path. ROOT has its own collection of predefined include paths (⇒ root[0] .include)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void make_tree(Int_t nevents=100000)
{

  TFile* file = new TFile("tree_C.root","RECREATE", "");
  TTree* tree = new TTree("tree","A simple Tree");

  Float_t px, py, pz;
  Double_t random;
  Int_t ev;

  tree->Branch("px",&px,"px/F");
  tree->Branch("py",&py,"py/F");
  tree->Branch("pz",&pz,"pz/F");
  tree->Branch("random",&random,"random/D");
  tree->Branch("ev",&ev,"ev/I");

  for (Int_t i = 0 ; i<nevents ; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    tree->Fill();
  }

  tree->Write();

}
```

Exercise 1: create a dataset (⇒ tree.C)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void make_tree(Int_t nevents=100000)
{

  TFile* file = new TFile("tree_C.root","RECREATE", "");
  TTree* tree = new TTree("tree","A simple Tree");

  Float_t px, py, pz;
  Double_t random;
  Int_t ev;

  tree->Branch("px",&px,"px/F");
  tree->Branch("py",&py,"py/F");
  tree->Branch("pz",&pz,"pz/F");
  tree->Branch("random",&random,"random/D");
  tree->Branch("ev",&ev,"ev/I");

  for (Int_t i = 0 ; i<nevents ; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    tree->Fill();
  }

  tree->Write();

}
```

A classical C++ function.
N.B. ROOT defines new types for int, float, …
E.g.: int → Int_t

No need to call explicit destructors here.
⇒ after execution, objects remains "alive"
(will be destroyed when ROOT is quitted)

Exercise 1: create a dataset (⇒ tree.C)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void make_tree(Int_t nevents=100000)
{

  TFile* file = new TFile("tree_C.root","RECREATE");
  TTree* tree = new TTree("tree","A simple Tree");

  Float_t px, py, pz;
  Double_t random;
  Int_t ev;

  tree->Branch("px",&px,"px/F");
  tree->Branch("py",&py,"py/F");
  tree->Branch("pz",&pz,"pz/F");
  tree->Branch("random",&random,"random/D");
  tree->Branch("ev",&ev,"ev/I");
```

ROOT file created from scratch (⇒ "RECREATE")
A new TTree object is also created

```
  for (Int_t i = 0 ; i<nevents ; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    tree->Fill();
  }

  tree->Write();

}
```

Objects that we want to be persistently save (e.g. tree) will be written into the currently open writable file

Exercise 1: create a dataset (⇒ tree.C)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void make_tree(Int_t nevents=100000)
{

  TFile* file = new TFile("tree_C.root","RECREATE", "");
  TTree* tree = new TTree("tree","A simple Tree");

  Float_t px, py, pz;
  Double_t random;
  Int_t ev;

  tree->Branch("px",&px,"px/F");
  tree->Branch("py",&py,"py/F");
  tree->Branch("pz",&pz,"pz/F");
  tree->Branch("random",&random,"random/D");
  tree->Branch("ev",&ev,"ev/I");
```

Auxiliary variables to store intermediate values

```
  for (Int_t i = 0 ; i<nevents ; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    tree->Fill();
  }

  tree->Write();

}
```

Define the TBranches (⇒ columns) for this TTree

Variable type is specified here (e.g. "/F", "/I", "/D")

Exercise 1: create a dataset (⇒ tree.C)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void make_tree(Int_t nevents=100000)
{

  TFile* file = new TFile("tree_C.root","RECREATE", "");
  TTree* tree = new TTree("tree","A simple Tree");

  Float_t px, py, pz;
  Double_t random;
  Int_t ev;

  tree->Branch("px",&px,"px/F");
  tree->Branch("py",&py,"py/F");
  tree->Branch("pz",&pz,"pz/F");
  tree->Branch("random",&random,"random/D");
  tree->Branch("ev",&ev,"ev/I");

  for (Int_t i = 0 ; i<nevents ; i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    tree->Fill();
  }

  tree->Write();

}
```

Start the "event" loop.
At each iterate i, the Fill() command fills one row of the TTree using current data found at the branch address

Inspecting a ROOT file (⇒ tree_C.root)

- **Let's run the macro** tree.C

```
> root -l
root[0] .L tree.C
root[1] make_tree(100000)
root[2] .q
```

Equivalent to:

```
> root -l -b -e 'gROOT->ProcessLine(".L tree.C"); gROOT->ProcessLine("make_tree(100000)")'
```

We could also have compiled the code (⇒ tree_C.so) and run it:

```
> root -l -b -e 'gROOT->ProcessLine(".L tree.C++"); gROOT->ProcessLine("make_tree(100000)")'
```

- **Let's inspect the file content** (Q: what is the size of the output file?)

```
> root -l tree_C.root // equivalent to starting ROOT prompt and then opening the TFile
root[0] .ls
root[1] tree->Print()
root[2] new TBrowser()
```

Inspecting a ROOT file (⇒ tree_C.root)

- **From the ROOT prompt, you can draw the content of any** TBranch

```
root[0] tree->Draw("px")
root[1] tree->Draw("px", "py>0 && random<0.5") // 2nd argument is a 0/1 weight per-event
root[2] tree->Draw("px:py", "py>0 && random<0.5", "colz”) //  “:” means “one vs the other”
root[3] tree->Draw("px:py:pz", "py>0 && random<0.5", "lego")
```

- **The output of** Draw() **can be also redirected into a named histogram:**

```
root[4] tree->Draw("px>>h(100,-3,3)") // TH1F object “h” is created
root[5] h->GetMean()
```

- **You can draw any function of the input branches:**

```
root[6] tree->Draw("px*TMath::Sin(py)/TMath::Log(pz+1)")
```

When functions are too complicated to fit into a line:

1. Write the function in a separate macro (⇒ func.C)
2. Load the function macro within ROOT (⇒ root[1] .L func.C)
3. Now, you can now use func(…) within Draw(). E.g.:

```
root[7] tree->Draw("px*TMath::Sin(py)/TMath::Log(pz+1):func(px,py,pz)", "", "COLZ")
```

Exercise 2: read a dataset and fill a histogram (⇒ tree.C)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void read_tree()
{

  TFile *infile = new TFile("tree_C.root", "READ");
  TTree *tree = (TTree*)infile->Get("tree");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  tree->SetBranchAddress("px",&px);
  tree->SetBranchAddress("py",&py);
  tree->SetBranchAddress("pz",&pz);
  tree->SetBranchAddress("random",&random);
  tree->SetBranchAddress("ev",&ev);
```

```
  TH1F *hpx   = new TH1F("hpx","px distribution",100,-3,3);
  TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);

  Long64_t nentries = tree->GetEntries();
  for (Long64_t i = 0; i < nentries; i++) {
    tree->GetEntry(i);
    hpx->Fill(px);
    hpxpy->Fill(px,py);
  }

  hpx->Draw("HISTE");
  TFile* outfile = new TFile("histos.root", "RECREATE");
  hpx->Write();
  outfile->Close();

}
```

Exercise 2: read a dataset and fill a histogram (⇒ tree.C)

This is analogous to make_tree() **with the replacements:**
"RECREATE" ⇒ "READ"
tree->Branch(...) ⇒ tree->SetBranchAddress(...)

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void read_tree()
{

  TFile *infile = new TFile("tree_C.root", "READ");
  TTree *tree = (TTree*)infile->Get("tree");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  tree->SetBranchAddress("px",&px);
  tree->SetBranchAddress("py",&py);
  tree->SetBranchAddress("pz",&pz);
  tree->SetBranchAddress("random",&random);
  tree->SetBranchAddress("ev",&ev);

  TH1F *hpx   = new TH1F("hpx","px distribution",100,-3,3);
  TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);

  Long64_t nentries = tree->GetEntries();
  for (Long64_t i = 0; i < nentries; i++) {
    tree->GetEntry(i);
    hpx->Fill(px);
    hpxpy->Fill(px,py);
  }

  hpx->Draw("HISTE");
  TFile* outfile = new TFile("histos.root", "RECREATE");
  hpx->Write();
  outfile->Close();

}
```

Exercise 2: read a dataset and fill a histogram (⇒ tree.C)

We define 1D and 2D histograms to be filled with the TTree content

```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void read_tree()
{

  TFile *infile = new TFile("tree_C.root", "READ");
  TTree *tree = (TTree*)infile->Get("tree");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  tree->SetBranchAddress("px",&px);
  tree->SetBranchAddress("py",&py);
  tree->SetBranchAddress("pz",&pz);
  tree->SetBranchAddress("random",&random);
  tree->SetBranchAddress("ev",&ev);

  TH1F *hpx   = new TH1F("hpx","px distribution",100,-3,3);
  TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);

  Long64_t nentries = tree->GetEntries();
  for (Long64_t i = 0; i < nentries; i++) {
    tree->GetEntry(i);
    hpx->Fill(px);
    hpxpy->Fill(px,py);
  }

  hpx->Draw("HISTE");
  TFile* outfile = new TFile("histos.root", "RECREATE");
  hpx->Write();
  outfile->Close();

}
```

Draw a bar histogram ("HIST") with Poisson error ("E")

Histogram persistently written into file, e.g. for future use

Exercise 2: read a dataset and fill a histogram (⇒ tree.C)

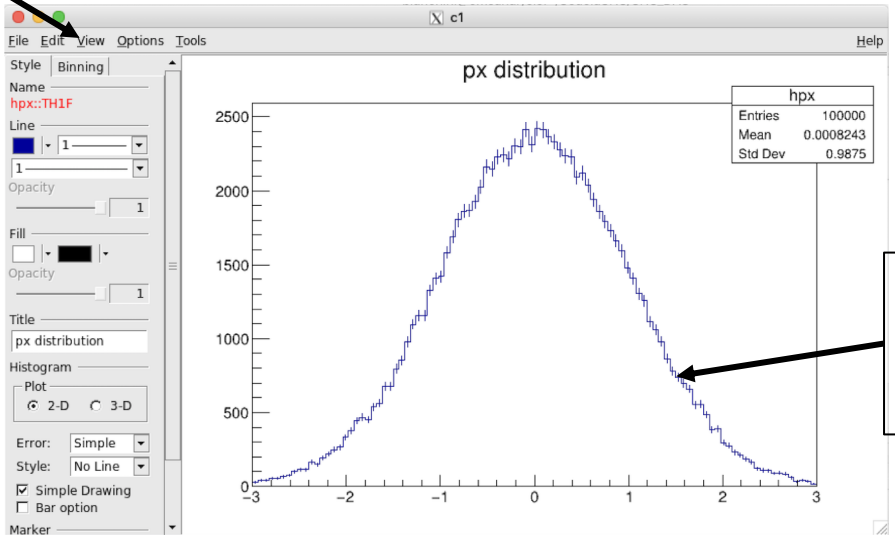
```
#include "TFile.h"
#include "TTree.h"
#include "TH2.h"
#include "TRandom.h"

void read_tree()
{

  TFile *infile = new TFile("tree_C.root", "READ");
  TTree *tree = (TTree*)infile->Get("tree");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  tree->SetBranchAddress("px",&px);
  tree->SetBranchAddress("py",&py);
  tree->SetBranchAddress("pz",&pz);
  tree->SetBranchAddress("random",&random);
  tree->SetBranchAddress("ev",&ev);

  TH1F *hpx   = new TH1F("hpx","px distribution",100,-3,3);
  TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);

  Long64_t nentries = tree->GetEntries();
  for (Long64_t i = 0; i < nentries; i++) {
    tree->GetEntry(i);
    hpx->Fill(px);
    hpxpy->Fill(px,py);
  }

  hpx->Draw("HISTE");
  TFile* outfile = new TFile("histos.root", "RECREATE");
  hpx->Write();
  outfile->Close();

}
```

Start the "event" loop. At each iterate i, the GetEntry(i) command will read data row i for <u>all the "addressed" branches</u>.
Not-addressed branches are NOT read (big speed-up when partial data needed)

A simple 1D fit to a binned data set (⇒ tree.C)

- **Run** read_tree()
- **The default** TCanvas **is displayed showing** hpx

On the top-left, select "View" -> "Editor"



Right-click on any point of the histogram and choose "Fit Panel"

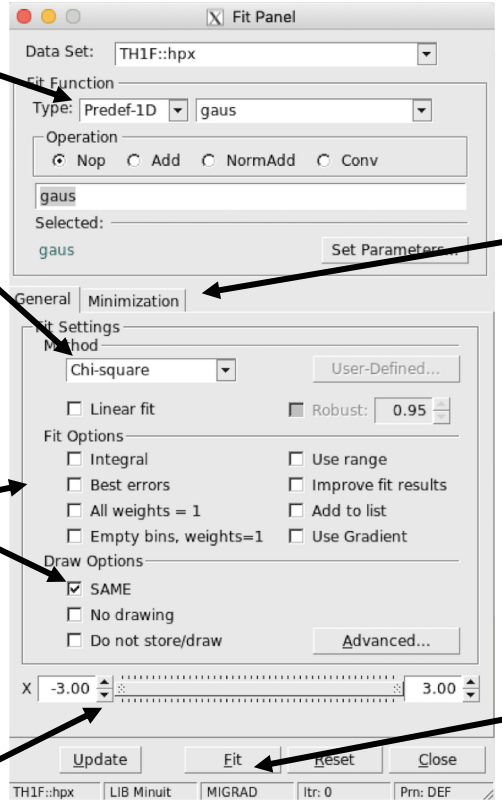
The Fit Panel

Fit function
(pre-defined or user-defined)

Loss function
(χ^2 or binned likelihood)

Fit options
Click on SAME so that fit result is visualized

Adjustable fit range



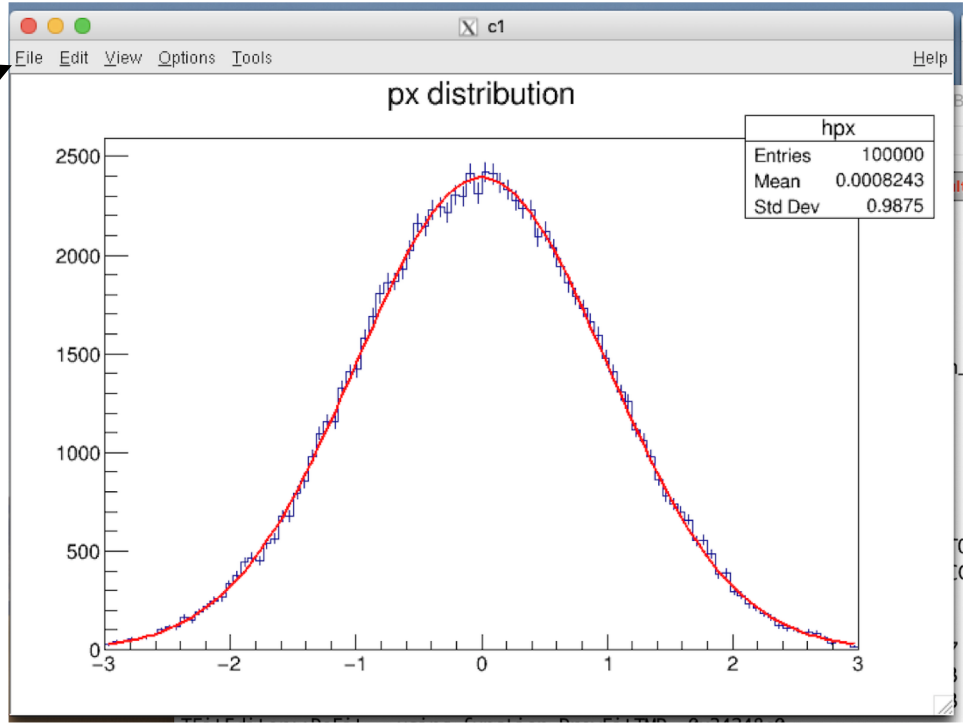
Minimizer
- Minuit: Migrad, Simplex, Fumili
- GSL: conjugate-gradient (FR, PR) Lavenberg-Marquardt, Steepest descent

Choose:
Type -> Predef-1D -> gaus
Minimization -> Minuit -> MIGRAD

Click on Fit

Saving graphic data into a .C file

You can customize the plot as you wish. Then: "File" -> "Save As..." to save it in graphic form.



Select the ".C" format, and save it as example.C

Now do:

```
root[1] .q
> root -l example.C
```

A simple 1D fit to a binned data set

This way of doing a fit is clear, but inefficient (shown only for illustration!)

In real life applications (e.g. >>1 histograms, multiple fit options, complicated functions...), all of these operations will be coded in a C++ macro and run in batch mode:

```
> root -b fit.C
```

Homework: write your own version of `fit.C`

Hint: you should use the `.Fit()` *method of the* `TH1F` *class*

Exercise 3: make_tree() revisited (⇒ tree.py)

```
import ROOT
import numpy as np

def make_tree(nevents):
    outfile = ROOT.TFile("tree_py.root", "RECREATE")
    tree = ROOT.TTree("tree", "A simple Tree")
    px = np.empty((1), dtype="float32")
    py = np.empty((1), dtype="float32")
    pz = np.empty((1), dtype="float32")
    random = np.empty((1), dtype="float64")
    ev = np.empty((1), dtype="int32")
    tree.Branch("px", px, "px/F")
    tree.Branch("py", py, "py/F")
    tree.Branch("pz", pz, "pz/F")
    tree.Branch("random", random, "random/D")
    tree.Branch("ev", ev, "ev/I")
    for i in range(nevents):
        px[0] = np.random.normal()
        py[0] = np.random.normal()
        pz[0] =  px[0]**2 + py[0]**2
        random[0] = np.random.random_sample()
        ev[0] = i
        tree.Fill()
    outfile.Write()
    return (outfile), tree
```

```
if __name__ == '__main__':
    _, tree = make_tree(100000)
    array, labels = tree.AsMatrix(return_labels=True)
```

Exercise 3: make_tree() revisited (⇒ tree.py)

```
import ROOT
import numpy as np

def make_tree(nevents):
    outfile = ROOT.TFile("tree_py.root", "RECREATE")
    tree = ROOT.TTree("tree", "A simple Tree")
    px = np.empty((1), dtype="float32")
    py = np.empty((1), dtype="float32")
    pz = np.empty((1), dtype="float32")
    random = np.empty((1), dtype="float64")
    ev = np.empty((1), dtype="int32")
    tree.Branch("px", px, "px/F")
    tree.Branch("py", py, "py/F")
    tree.Branch("pz", pz, "pz/F")
    tree.Branch("random", random, "random/D")
    tree.Branch("ev", ev, "ev/I")
    for i in range(nevents):
        px[0] = np.random.normal()
        py[0] = np.random.normal()
        pz[0] =  px[0]**2 + py[0]**2
        random[0] = np.random.random_sample()
        ev[0] = i
        tree.Fill()
    outfile.Write()
    return (outfile), tree
```

```
if __name__ == '__main__':
    _, tree = make_tree(100000)
    array, labels = tree.AsMatrix(return_labels=True)
```

That's it! From now on, any ROOT class T* accessible using ROOT.T*

numpy **or** array **only needed for <u>writing</u> branches**

Exercise 3: make_tree() revisited (⇒ tree.py)

```
import ROOT
import numpy as np

def make_tree(nevents):
    outfile = ROOT.TFile("tree_py.root", "RECREATE")
    tree = ROOT.TTree("tree", "A simple Tree")
    px = np.empty((1), dtype="float32")
    py = np.empty((1), dtype="float32")
    pz = np.empty((1), dtype="float32")
    random = np.empty((1), dtype="float64")
    ev = np.empty((1), dtype="int32")
    tree.Branch("px", px, "px/F")
    tree.Branch("py", py, "py/F")
    tree.Branch("pz", pz, "pz/F")
    tree.Branch("random", random, "random/D")
    tree.Branch("ev", ev, "ev/I")
    for i in range(nevents):
        px[0] = np.random.normal()
        py[0] = np.random.normal()
        pz[0] =  px[0]**2 + py[0]**2
        random[0] = np.random.random_sample()
        ev[0] = i
        tree.Fill()
    outfile.Write()
    return (outfile), tree
```

```
if __name__ == '__main__':
    _, tree = make_tree(100000)
    array, labels = tree.AsMatrix(return_labels=True)
```

Same as before, with all the benefits of Python

Exercise 3: make_tree() revisited (⇒ tree.py)

```
import ROOT
import numpy as np

def make_tree(nevents):
    outfile = ROOT.TFile("tree_py.root", "RECREATE")
    tree = ROOT.TTree("tree", "A simple Tree")
    px = np.empty((1), dtype="float32")
    py = np.empty((1), dtype="float32")
    pz = np.empty((1), dtype="float32")
    random = np.empty((1), dtype="float64")
    ev = np.empty((1), dtype="int32")
    tree.Branch("px", px, "px/F")
    tree.Branch("py", py, "py/F")
    tree.Branch("pz", pz, "pz/F")
    tree.Branch("random", random, "random/D")
    tree.Branch("ev", ev, "ev/I")
    for i in range(nevents):
        px[0] = np.random.normal()
        py[0] = np.random.normal()
        pz[0] =  px[0]**2 + py[0]**2
        random[0] = np.random.random_sample()
        ev[0] = i
        tree.Fill()
    outfile.Write()
    return (outfile), tree
```

```
if __name__ == '__main__':
    _, tree = make_tree(100000)
    array, labels = tree.AsMatrix(return_labels=True)
```

A nice feature: automatic conversion
TTree -> numpy.array(nevents,nbranches)
(working in multithread mode, if supported)

Exercise 3: make_tree() revisited (⇒ tree.py)

```
import ROOT
import numpy as np

def make_tree(nevents):
    outfile = ROOT.TFile("tree_py.root", "RECREATE")
    tree = ROOT.TTree("tree", "A simple Tree")
    px = np.empty((1), dtype="float32")
    py = np.empty((1), dtype="float32")
    pz = np.empty((1), dtype="float32")
    random = np.empty((1), dtype="float64")
    ev = np.empty((1), dtype="int32")
    tree.Branch("px", px, "px/F")
    tree.Branch("py", py, "py/F")
    tree.Branch("pz", pz, "pz/F")
    tree.Branch("random", random, "random/D")
    tree.Branch("ev", ev, "ev/I")
    for i in range(nevents):
        px[0] = np.random.normal()
        py[0] = np.random.normal()
        pz[0] =  px[0]**2 + py[0]**2
        random[0] = np.random.random_sample()
        ev[0] = i
        tree.Fill()
    outfile.Write()
    return (outfile), tree
```

```
if __name__ == '__main__':
    _, tree = make_tree(100000)
    array, labels = tree.AsMatrix(return_labels=True)
```

<u>Caveats</u>:

PyROOT can be much slower than compiled C++*(see e.g. the two examples just discussed)*

Handling of memory sometimes problematic. Extra care needed to avoid memory leaks.

Exercise 3: make_tree() revisited (⇒ tree.py)

Let's run it:

```
> python tree.py 10
> python tree.py 100000
```

Now, let's modify tree.py so that the output tree also contains variable-length arrays of floats…

N.B.: convertion into numpy array will fail due to array-like branch

Exercise 4: a tree with variable-length branches (⇒ tree_array.py)

```
import ROOT
import numpy as np

def make_tree(nevents):
    outfile = ROOT.TFile("tree_py.root", "RECREATE")
    tree = ROOT.TTree("tree", "A simple Tree")
    px = np.empty((1), dtype="float32")
    py = np.empty((1), dtype="float32")
    pz = np.empty((1), dtype="float32")
    random = np.empty((1), dtype="float64")
    ev = np.empty((1), dtype="int32")
    nHits = np.empty((1), dtype="int32")
    hits = np.empty((100), dtype="float32")
    tree.Branch("px", px, "px/F")
    tree.Branch("py", py, "py/F")
    tree.Branch("pz", pz, "pz/F")
    tree.Branch("random", random, "random/D")
    tree.Branch("ev", ev, "ev/I")
    tree.Branch("nHits", nHits, "nHits/I")
    tree.Branch("hits", hits, "hits[nHits]/F")
```

```
for i in range(nevents):
    px[0] = np.random.normal()
    py[0] = np.random.normal()
    pz[0] =  px[0]**2 + py[0]**2
    random[0] = np.random.random_sample()
    ev[0] = i
    nHits[0] = 2 if i%2==0 else 4
    for h in range(nHits[0]):
        hits[h] = np.random.randint(0,20)
    tree.Fill()
outfile.Write()
return (outfile), tree
```

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

ROOT comes with a native mathematical environment

- **Linear algebra**: matrix inversion, eigendecomposition, determinant, matrix-matrix and matrix-vector multiplication.
- **N-dimensional integration**: MC integration methods (VEGAS, Adaptive, Miser, Gauss-Legendre (1D))
- **N-dimensional minimization**: Minuit2 (re-written from original FORTRAN code by F. James) and wrapper to GSL minimizers

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

```
#include "Math/Minimizer.h"
#include "Math/Factory.h"
#include "Math/Functor.h"
#include "TRandom3.h"
#include "Math/IntegratorMultiDim.h"

const int NDIM = 4;

TMatrixD B = THilbertMatrixD(NDIM,NDIM);

double grad[NDIM] = {};
TVectorD g(NDIM, grad);
const double f0 = 1.0;

double Quadratic(const double *xx)
{
  TVectorD x(NDIM, xx);
  return f0 + g*x + 0.5*(x*(B*x));
}

void minimizer(const char * minName = "Minuit", const char *algoName =
"Migrad"){

  TRandom3 ran(1234);
  double gmin = -1.;
  double gmax = +1.;

 for(unsigned int i=0 ; i<NDIM; ++i){
    double r = ran.Rndm();
    g[i] = gmin*r + gmax*(1-r);
  }

TVectorD eig(NDIM);
TMatrixD eigs = B.EigenVectors(eig);
Double_t det = B.Determinant();
```

```
 ROOT::Math::Minimizer* minimum =      ROOT::Math::Factory::CreateMinimizer(minName,
algoName);
  minimum->SetMaxFunctionCalls(1000000);
  minimum->SetTolerance(0.001);
  minimum->SetPrintLevel(1);

  ROOT::Math::Functor f( &Quadratic, NDIM);
  minimum->SetFunction(f);

  double step[NDIM] = {};
  for(unsigned int i=0 ; i<NDIM; ++i) step[i] += 0.01;
  double start[NDIM] = {};
  for(unsigned int i=0 ; i<NDIM; ++i){
     minimum->SetVariable(i, Form("x%d",i), start[i], step[i]);
  }

minimum->Minimize();
  const double *xs = minimum->X();

  std::cout <<  std::endl;
  std::cout << "NUMERICAL:  f(";
  for(unsigned int i=0 ; i<NDIM; ++i) std::cout << xs[i] << ",";
  std::cout <<  "): " << minimum->MinValue()  << std::endl;

  TVectorD x0(NDIM, start);
  TMatrixD Binv(B);
  TVectorD xs_ana = x0 - Binv.InvertFast()*(B*x0 + g);
  std::cout << "ANALYTICAL: f(";
  for(unsigned int i=0 ; i<NDIM; ++i) std::cout << xs_ana[i] << ",";
  std::cout <<  "): " << Quadratic(xs_ana.GetMatrixArray()) << std::endl;

  return;

}
```

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

```
#include "Math/Minimizer.h"
#include "Math/Factory.h"
#include "Math/Functor.h"
#include "TRandom3.h"
#include "Math/IntegratorMultiDim.h"

const int NDIM = 4;

TMatrixD B = THilbertMatrixD(NDIM,NDIM);

double grad[NDIM] = {};
TVectorD g(NDIM, grad);
const double f0 = 1.0;

double Quadratic(const double *xx)
{
  TVectorD x(NDIM, xx);
  return f0 + g*x + 0.5*(x*(B*x));
}

void minimizer(const char * minName = "Minuit", const
char *algoName = "Migrad"){

  TRandom3 ran(1234);
  double gmin = -1.;
  double gmax = +1.;
```

```
for(unsigned int i=0 ; i<NDIM; ++i){
  double r = ran.Rndm();
  g[i] = gmin*r + gmax*(1-r);
}
```

Use an ill-conditioned Hessian matrix to make the task more challenging

The objective function we wish to minimize.
xx[i] is *i*-th variable

Create non-zero gradient to make the solution non-trivial

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

Check positive-definiteness of the Hessian matrix

```
TVectorD eig(NDIM);
TMatrixD eigs = B.EigenVectors(eig);
Double_t det = B.Determinant();
```

Use ROOT::Math::Factory to instantiate the minimizer

```
ROOT::Math::Minimizer* minimum =
ROOT::Math::Factory::CreateMinimizer(minName, algoName);
minimum->SetMaxFunctionCalls(1000000);
minimum->SetTolerance(0.001);
minimum->SetPrintLevel(1);
```

Define which function to minimize

```
ROOT::Math::Functor f( &Quadratic, NDIM);
minimum->SetFunction(f);
```

Declare variable and suitable starting values & initial step

```
double step[NDIM] = {};
for(unsigned int i=0 ; i<NDIM; ++i) step[i] += 0.01;
double start[NDIM] = {};
for(unsigned int i=0 ; i<NDIM; ++i){
   minimum->SetVariable(i, Form("x%d",i), start[i], step[i]);
}
```

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

```
minimum->Minimize();
const double *xs = minimum->X();

std::cout <<  std::endl;
std::cout << "NUMERICAL:  f(";
for(unsigned int i=0 ; i<NDIM; ++i) std::cout << xs[i] << ",";
std::cout <<  "): " << minimum->MinValue()  << std::endl;

TVectorD x0(NDIM, start);
TMatrixD Binv(B);
TVectorD xs_ana = x0 - Binv.InvertFast()*(B*x0 + g);
std::cout << "ANALYTICAL: f(";
for(unsigned int i=0 ; i<NDIM; ++i) std::cout << xs_ana[i] << ",";
std::cout <<  "): " << Quadratic(xs_ana.GetMatrixArray()) << std::endl;
```

Start the minimizer

Retrieve the information about the minimum

Alternatively, find the solution by one Newton step

Evaluate the objective function at the analytical solution

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

```
  double xL[NDIM] = {};
  double xU[NDIM] = {};
  for(unsigned int i=0 ; i<NDIM; ++i) xL[i] = -1.0;
  for(unsigned int i=0 ; i<NDIM; ++i) xU[i] = +1.0;

  double val = 0.;
  ROOT::Math::IntegratorMultiDim ig1(ROOT::Math::IntegrationMultiDim::kVEGAS);
  ig1.SetFunction(f);
  val = ig1.Integral(xL,xU);

  return;
}
```

Integration range

Several MC algorithms available.
Unique interface.

Integrate the function in a box

Exercise 5: ROOT mathematical environment (⇒ minimizer.C)

Let's run it:

```
> root -l
root[1] .L minimizer.C
root[2] minimizer("Minuit2","migrad")                    // quasi-Newton line-search with Flatcher's switching rank-2 update
root[3] minimizer("Minuit2","simplex")                   // Nelder-Mead simplex algorithm
root[4] minimizer("GSLMultiMin","conjugatefr")           // Fletcher-Reeves conjugate-gradient line-search
root[5] minimizer("GSLMultiMin","conjugatepr")           // Poljak-Ribiere conjugate-gradient line-search
root[6] minimizer("GSLMultiMin","bfgs2")                 // quasi-Newton method with BFGS rank-2 update
root[7] minimizer("GSLMultiMin","steepestdescent")// steepest-descent line search
```

You can try two functions: "Quadratic" (NDIM<7) **and** "Rosenbrock" (NDIM>100)

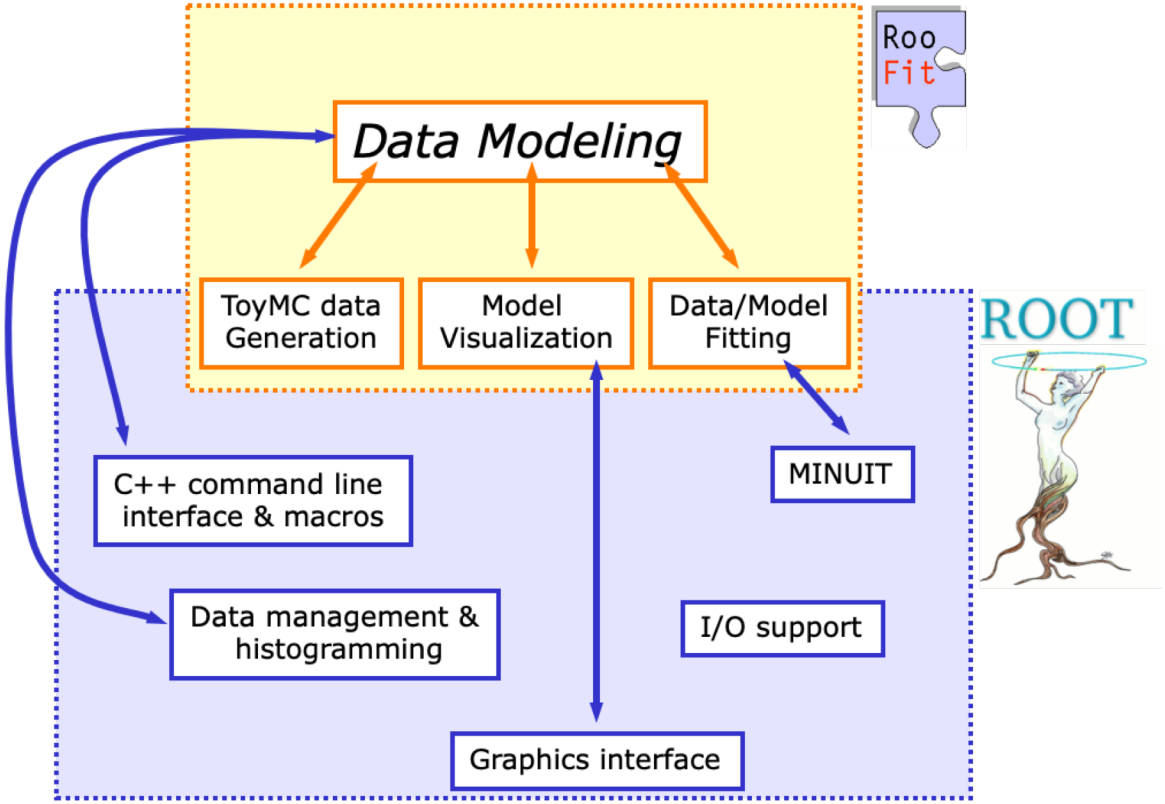
RooFit: a ROOT package for statistical analysis

- **ROOT has some limitations when dealing with complicated fit**
  - Standard ROOT function framework insufficient to handle complicated functions
  - P.d.f's are normalized densities, not just functions
  - Computation performances important when NDIM>>1, unbinned data, many events, ...
- **You can write a specific fit using ROOT interface to Minuit (⇒ minimizer.C)**
  - can requires a lot of coding and *ad hoc* optimization

⇒ **RooFit**

- Interface to Minuit2 transparent to the user
- Many precompiled physics-inspired p.d.f.'s
- Takes care of p.d.f. normalization, variable marginalization, conditioning, simultaneous fit. Using optimizations whenever possible
- Automatization of profile-likelihood scans, likelihood contours, GoF, MC studies, ...

RooFit: a ROOT package for statistical analysis



RooFit: a ROOT package for statistical analysis

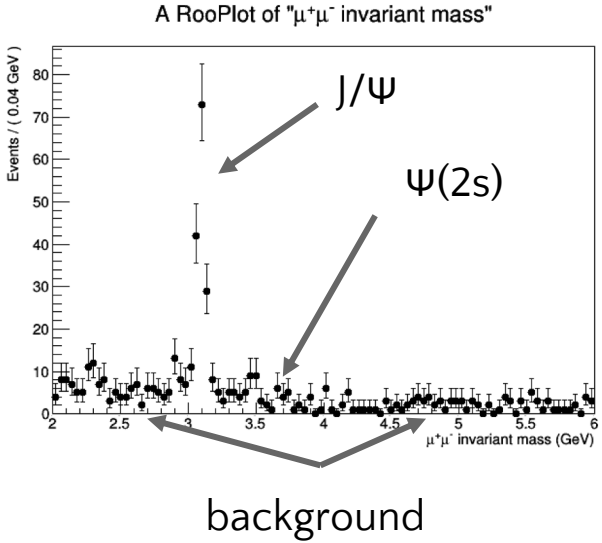
Mathematical concept		RooFit class
variable	x	`RooRealVar`
function	$f(x)$	`RooAbsReal`
PDF	$f(x)$	`RooAbsPdf`
space point	$\vec{x}$	`RooArgSet`
integral	$\int_{x_{\min}}^{x_{\max}} f(x)dx$	`RooRealIntegral`
list of space points		`RooAbsData`

`RooAbsData` → RooDataHist (binned)

`RooAbsData` → RooDataSet (unbinned)

A more sophisticated fit: measuring a cross section

- **A sample of di-muon events is collected and their invariant mass is measured. Three processes:**
  - J/Ψ, with a mass of **3.096** GeV and FWHM ~ 1%
  - Ψ(2s) with a mass of **3.686** GeV, same width as J/Ψ
  - smooth and broad combinatorial background



- **What is the cross section σ of Ψ(2s)?**

  $\sigma = N/(L\,\varepsilon)$, N = number of measured Ψ(2s) (TBD)

  L = integrated luminosity (0.64/pb)

  ε = reconstruction efficiency (75%)

- **The data is a** RooDataSet **(⇒** DataSet_lowstat.root**) containing the di-muon mass of 500 events.**

Exercise 6: fitting with RooFit (⇒ roofit.py)

```
import ROOT

fInput = ROOT.TFile("DataSet_lowstat.root")
dataset = fInput.Get("data")

mass = ROOT.RooRealVar("mass","mumu mass", 2.0,
6.0,"GeV")

meanJpsi = ROOT.RooRealVar("meanJpsi","The mean of the Jpsi
Gaussian",3.1,2.8,3.2)
sigmaJpsi = ROOT.RooRealVar("sigmaJpsi","The width of the Jpsi
Gaussian",0.3,0.0001,1.)
alphaJpsi = ROOT.RooRealVar("alphaJpsi","The alpha of the Jpsi
Gaussian",1.5,-5.,5.)
nJpsi = ROOT.RooRealVar("nJpsi","The alpha of the Jpsi
Gaussian",1.5,0.5,5.)

CBJpsi = ROOT.RooCBShape("CBJpsi","The Jpsi Crystall
Ball",mass,meanJpsi,sigmaJpsi,alphaJpsi,nJpsi)
```

```
meanpsi2S = ROOT.RooRealVar("meanpsi2S","The mean of
the psi(2S) Gaussian",3.7,3.65,3.75)
gausspsi2S = ROOT.RooGaussian("gausspsi2S","The psi(2S)
Gaussian",mass,meanpsi2S,sigmaJpsi)

a1 = ROOT.RooRealVar("a1","The a1 of background",-0.7,-
2.,2.)
a2 = ROOT.RooRealVar("a2","The a2 of background",0.3,-
2.,2.)
a3 = ROOT.RooRealVar("a3","The a3 of background",-0.03,-
2.,2.)
backgroundPDF =
ROOT.RooChebychev("backgroundPDF","The background
PDF",mass,ROOT.RooArgList(a1,a2,a3))
```

Exercise 6: fitting with RooFit (⇒ roofit.py)

Get the data (same I/O than ROOT!)

```
import ROOT

fInput = ROOT.TFile("DataSet_lowstat.root")
dataset = fInput.Get("data")

mass = ROOT.RooRealVar("mass","mumu mass", 2.0,
6.0,"GeV")

meanJpsi = ROOT.RooRealVar("meanJpsi","The mean of the Jpsi
Gaussian",3.1,2.8,3.2)
sigmaJpsi = ROOT.RooRealVar("sigmaJpsi","The width of the Jpsi
Gaussian",0.3,0.0001,1.)
alphaJpsi = ROOT.RooRealVar("alphaJpsi","The alpha of the Jpsi
Gaussian",1.5,-5.,5.)
nJpsi = ROOT.RooRealVar("nJpsi","The alpha of the Jpsi
Gaussian",1.5,0.5,5.)

CBJpsi = ROOT.RooCBShape("CBJpsi","The Jpsi Crystall
Ball",mass,meanJpsi,sigmaJpsi,alphaJpsi,nJpsi)
```

```
meanpsi2S = ROOT.RooRealVar("meanpsi2S","The mean of
the psi(2S) Gaussian",3.7,3.65,3.75)
gausspsi2S = ROOT.RooGaussian("gausspsi2S","The psi(2S)
Gaussian",mass,meanpsi2S,sigmaJpsi)

a1 = ROOT.RooRealVar("a1","The a1 of background",-0.7,-
2.,2.)
a2 = ROOT.RooRealVar("a2","The a2 of background",0.3,-
2.,2.)
a3 = ROOT.RooRealVar("a3","The a3 of background",-0.03,-
2.,2.)
backgroundPDF =
ROOT.RooChebychev("backgroundPDF","The background
PDF",mass,ROOT.RooArgList(a1,a2,a3))
```

"data" contains a column with title "mass"
(N.B. data imported by name, so they have to match)

Exercise 6: fitting with RooFit (⇒ roofit.py)

A simpler (⇒ Gaussian) function for Ψ(2s)

```
import ROOT

fInput = ROOT.TFile("DataSet_lowstat.root")
dataset = fInput.Get("data")

mass = ROOT.RooRealVar("mass","mumu mass", 2.0,
6.0,"GeV")

meanJpsi = ROOT.RooRealVar("meanJpsi","The mean of the Jpsi
Gaussian",3.1,2.8,3.2)
sigmaJpsi = ROOT.RooRealVar("sigmaJpsi","The width of the Jpsi
Gaussian",0.3,0.0001,1.)
alphaJpsi = ROOT.RooRealVar("alphaJpsi","The alpha of the Jpsi
Gaussian",1.5,-5.,5.)
nJpsi = ROOT.RooRealVar("nJpsi","The alpha of the Jpsi
Gaussian",1.5,0.5,5.)

CBJpsi = ROOT.RooCBShape("CBJpsi","The Jpsi Crystall
Ball",mass,meanJpsi,sigmaJpsi,alphaJpsi,nJpsi)
```

P.d.f. for the J/Ψ (⇒ Crystal Ball function)

```
meanpsi2S = ROOT.RooRealVar("meanpsi2S","The mean of
the psi(2S) Gaussian",3.7,3.65,3.75)
gausspsi2S = ROOT.RooGaussian("gausspsi2S","The psi(2S)
Gaussian",mass,meanpsi2S,sigmaJpsi)

a1 = ROOT.RooRealVar("a1","The a1 of background",-0.7,-
2.,2.)
a2 = ROOT.RooRealVar("a2","The a2 of background",0.3,-
2.,2.)
a3 = ROOT.RooRealVar("a3","The a3 of background",-0.03,-
2.,2.)
backgroundPDF =
ROOT.RooChebychev("backgroundPDF","The background
PDF",mass,ROOT.RooArgList(a1,a2,a3))
```

An empirical (⇒ polynomial) function for the background

Exercise 6: fitting with RooFit (⇒ roofit.py)

```
NJpsi = ROOT.RooRealVar("NJpsi","The Jpsi signal
events",1500.,0.1,10000.)
Nbkg = ROOT.RooRealVar("Nbkg","The bkg
events",5000.,0.1,50000.)

eff_psi = ROOT.RooRealVar("eff_psi","The psi
efficiency",0.75,0.00001,1.)
lumi_psi  = ROOT.RooRealVar("lumi_psi","The CMS
luminosity",0.64,0.00001,50.,"pb-1")
cross_psi = ROOT.RooRealVar("cross_psi","The psi
xsec",3.,0.,40.,"pb")
Npsi =
ROOT.RooFormulaVar("Npsi","@0*@1*@2",ROOT.RooArgList(eff
_psi,lumi_psi,cross_psi))
eff_psi.setConstant(1)
lumi_psi.setConstant(1)

totPDF = ROOT.RooAddPdf("totPDF","The total
PDF",ROOT.RooArgList(CBJpsi,gausspsi2S,backgroundPDF),ROOT
.RooArgList(NJpsi,Npsi,Nbkg))

totPDF.fitTo(dataset, ROOT.RooFit.Extended(1))
```

```
xframe = mass.frame()
dataset.plotOn(xframe)
totPDF.plotOn(xframe)
totPDF.plotOn(xframe,
ROOT.RooFit.Components("backgroundPDF"),
ROOT.RooFit.LineStyle(ROOT.kDashed),
ROOT.RooFit.LineColor(ROOT.kRed))

c1 = ROOT.TCanvas()
xframe.Draw()
c1.SaveAs("exercise_0.png")

fOutput =
ROOT.TFile("Workspace_mumufit.root","RECREATE")
fInput.cd()
ws = ROOT.RooWorkspace("ws")
getattr(ws,'import')(totPDF)
getattr(ws,'import')(dataset)
ws.writeToFile("Workspace_mumufit.root")
del ws
fOutput.Write()
fOutput.Close()
```

Exercise 6: fitting with RooFit (⇒ roofit.py)

Total yields for changing the relative S & B fractions

```
NJpsi = ROOT.RooRealVar("NJpsi","The Jpsi signal
events",1500.,0.1,10000.)
Nbkg = ROOT.RooRealVar("Nbkg","The bkg
events",5000.,0.1,50000.)

eff_psi = ROOT.RooRealVar("eff_psi","The psi
efficiency",0.75,0.00001,1.)
lumi_psi  = ROOT.RooRealVar("lumi_psi","The CMS
luminosity",0.64,0.00001,50.,"pb-1")
cross_psi = ROOT.RooRealVar("cross_psi","The psi
xsec",3.,0.,40.,"pb")
Npsi =
ROOT.RooFormulaVar("Npsi","@0*@1*@2",ROOT.RooArgList(eff
_psi,lumi_psi,cross_psi))
eff_psi.setConstant(1)
lumi_psi.setConstant(1)

totPDF = ROOT.RooAddPdf("totPDF","The total
PDF",ROOT.RooArgList(CBJpsi,gausspsi2S,backgroundPDF),ROOT
.RooArgList(NJpsi,Npsi,Nbkg))

totPDF.fitTo(dataset, ROOT.RooFit.Extended(1))
```

Total p.d.f. is S+B

$$-\log\left(L(\vec{p})\right) = -\sum_{D} \log\left(g(\vec{x}_i, \vec{p})\right) + N_{\exp} - N_{obs} \log\left(N_{\exp}\right)$$

```
xframe = mass.frame()
dataset.plotOn(xframe)
ROOT.RooFit.LineColor(ROOT.kRed))

c1 = ROOT.TCanvas()
xframe.Draw()
c1.SaveAs("exercise_0.png")

fOutput =
ROOT.TFile("Workspace_mumufit.root","RECREATE")
fInput.cd()
ws = ROOT.RooWorkspace("ws")
getattr(ws,'import')(totPDF)
getattr(ws,'import')(dataset)
ws.writeToFile("Workspace_mumufit.root")
del ws
fOutput.Write()
fOutput.Close()
```

Construct the NLL and minimize it w.r.t. all non-constant RooRealVar

Exercise 6: fitting with RooFit (⇒ roofit.py)

```
NJpsi = ROOT.RooRealVar("NJpsi","The Jpsi signal
events",1500.,0.1,10000.)
Nbkg = ROOT.RooRealVar("Nbkg","The bkg
events",5000.,0.1,50000.)

eff_psi = ROOT.RooRealVar("eff_psi","The psi
efficiency",0.75,0.00001,1.)
lumi_psi  = ROOT.RooRealVar("lumi_psi","The CMS
luminosity",0.64,0.00001,50.,"pb-1")
cross_psi = ROOT.RooRealVar("cross_psi","The psi
xsec",3.,0.,40.,"pb")
Npsi =
ROOT.RooFormulaVar("Npsi","@0*@1*@2",ROOT.RooArgList(eff
_psi,lumi_psi,cross_psi))
eff_psi.setConstant(1)
lumi_psi.setConstant(1)

totPDF = ROOT.RooAddPdf("totPDF","The total
PDF",ROOT.RooArgList(CBJpsi,gausspsi2S,backgroundPDF),ROOT
.RooArgList(NJpsi,Npsi,Nbkg))

totPDF.fitTo(dataset, ROOT.RooFit.Extended(1))
```

A fix for bad memory handling…

Draw the data & fit result

```
xframe = mass.frame()
dataset.plotOn(xframe)
totPDF.plotOn(xframe)
totPDF.plotOn(xframe,
ROOT.RooFit.Components("backgroundPDF"),
ROOT.RooFit.LineStyle(ROOT.kDashed),
ROOT.RooFit.LineColor(ROOT.kRed))

c1 = ROOT.TCanvas()
xframe.Draw()
c1.SaveAs("exercise_0.png")

fOutput =
ROOT.TFile("Workspace_mumufit.root","RECREATE")
fInput.cd()
ws = ROOT.RooWorkspace("ws")
getattr(ws,'import')(totPDF)
getattr(ws,'import')(dataset)
ws.writeToFile("Workspace_mumufit.root")
del ws
fOutput.Write()
fOutput.Close()
```

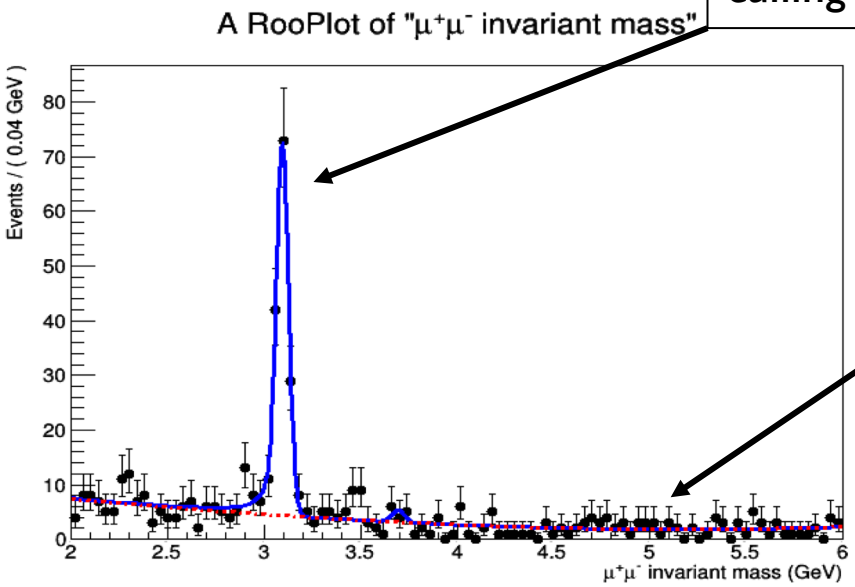
Save the data & fit result into a RooWorkspace

Exercise 6: fitting with RooFit (⇒ roofit.py)

Let's run it:

```
> python roofit.py
```

p.d.f. automatically normalized when calling plotOn()



Asymmetric poisson errors shown by default

Exercise 6: fitting with RooFit (⇒ Workspace_mumufit.root)

What is the measured cross section of Ψ(2s)->μμ?

```
> root -l -b Workspace_mumufit.root
root[1] w = (RooWorkspace*) gDirectory->Get("ws")
root[2] w->Print()
root[3] w->var("cross_psi")->Print()
```

```
RooRealVar::cross_psi = 9.65042 +/- 7.94308  L(0 - 40) // [pb]
```

Exercise 7: profile likelihood scans

Suppose we want to look at the profile log-likelihood **for one POI (**say, J/Ψ mass**).**
This is an important sanity check of the fit (and remember Wilks' theorem!)

- An intensive task (one minimization per value of the POI)
- RooFit has a built-in method for doing it

$$PL(p) = \frac{L(p, \hat{q})}{L(\hat{p}, \hat{q})}$$

Exercise 7: profile likelihood (⇒ profile.C)

```
{
 TFile* f = TFile::Open("Workspace_mumufit.root");
 RooWorkspace* w = (RooWorkspace*)f->Get("ws");
 w->Print();

 double xL = w->var("meanJpsi")->getVal()  - 2.2* w->var("meanJpsi")->getError();
 double xU = w->var("meanJpsi")->getVal() + 2.2* w->var("meanJpsi")->getError();
 w->var("meanJpsi")->setRange(xL, xU);

 RooNLLVar nll("nll","nll",*(w->pdf("totPDF")),*(w->data("data"))) ;
 RooProfileLL pll("pll","pll", nll,*w->var("meanJpsi"));

 RooPlot* frame = w->var("meanJpsi")->frame(xL, xU) ;
 pll.plotOn(frame) ;
 frame->Draw();
}
```

Exercise 7: profile likelihood (⇒ profile.C)

```
{
 TFile* f = TFile::Open("Workspace_mumufit.root");
 RooWorkspace* w = (RooWorkspace*)f->Get("ws");
 w->Print();

 double xL = w->var("meanJpsi")->getVal()  - 2.2* w->var("meanJpsi")->getError();
 double xU = w->var("meanJpsi")->getVal() + 2.2* w->var("meanJpsi")->getError();
 w->var("meanJpsi")->setRange(xL, xU);

 RooNLLVar nll("nll","nll",*(w->pdf("totPDF")),*(w->data("data"))) ;
 RooProfileLL pll("pll","pll", nll,*w->var("meanJpsi"));

 RooPlot* frame = w->var("meanJpsi")->frame(xL, xU) ;
 pll.plotOn(frame) ;
 frame->Draw();
}
```

Restrict scan to reasonable interval around the best-fit

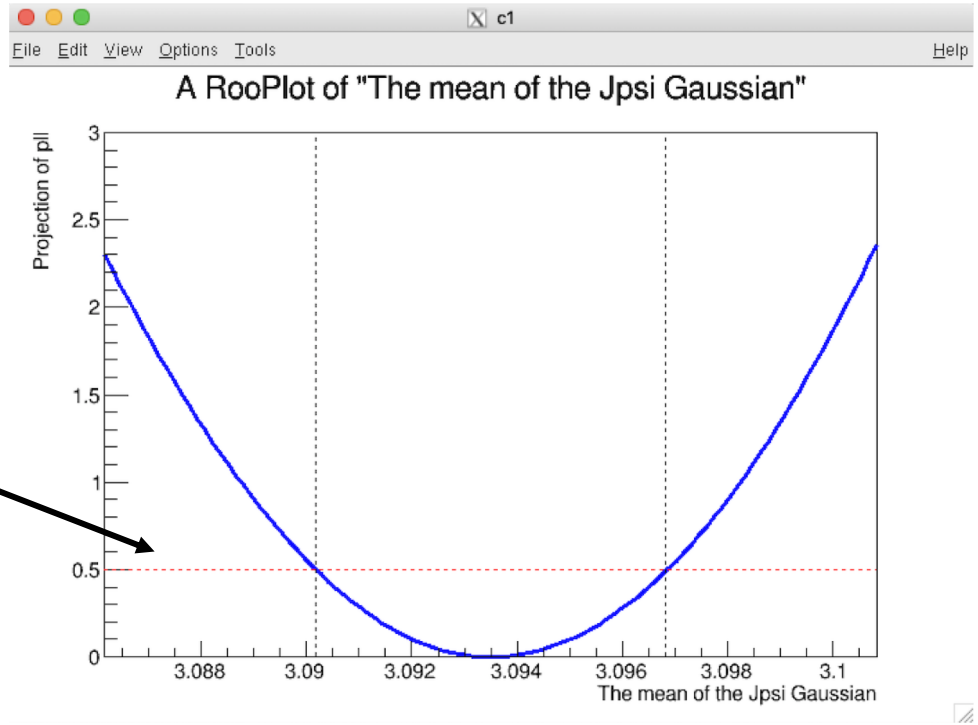
NLL(μ,θ | data).
(μ,θ are the best-fit results)

NLL(μ, θ_μ | data) - NLL(μ, θ | data)
(θ_μ is the best-fit result given μ)

Exercise 7: profile likelihood (⇒ profile.C)

Let's run it:

```
> root -l profile.C
```



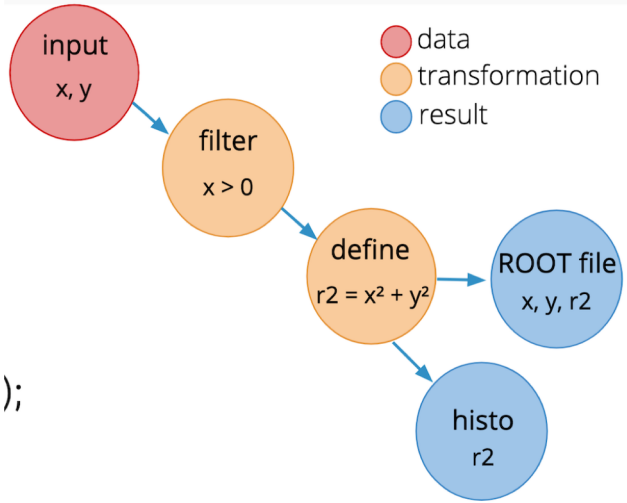
Add line segmemts (TLine) to visualize the 1σ CL intervals.

More into the future: RDataFrame (⇒ test_RDF.C, read_RDF.C)

A tool for declarative analysis

- *"Users say what, ROOT chooses how"*
- Analysis as a computational graph of connected nodes: data -> operations -> results
- Computation of the graph can be parallelized in a transparent way

```
{
  ROOT::EnableImplicitMT();
  ROOT::RDataFrame df("tree", "tree_py.root", {"px", "py"});
  auto df2 = df.Filter("px > 0").Define("pT2", "px*px + py*py");
  auto rHist = df2.Histo1D("pT2");
  rHist->Draw();
  df2.Snapshot("newtree", "out.root");
}
```



);

Lat's run it:

```
> root -l test_RDF.C
```

Backup