An introduction to numerical optimization

ROOT mathematical environment

L. Bianchini (INFN Sezione di Pisa)

Istituto Nazionale di Fisica Nucleare

ROOT (C++): an example

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

=> $f_0 + g^T x + ½\, x^T B\, x$

```
ROOT::Math::Minimizer* minimum =  ROOT::Math::Factory::CreateMinimizer(minName, algoName);
minimum->SetMaxFunctionCalls(10000);
minimum->SetTolerance(0.001);

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();
double *cov[NDIM*NDIM];
minimum->GetCovMat(cov);
```

("Minuit2","migrad")
("Minuit2","simplex"),
("GSLMultiMin","conjugatefr")
("GSLMultiMin","conjugatepr")
("GSLMultiMin","bfgs2")
("GSLMultiMin","steepestdescent")
("GSLMultiFit","") // Levemberg-Marquardt NLLQ
("GSLSimAn","")

GSL (C)

with gradient

```
gsl_multimin_fdfminimizer_conjugate_fr
gsl_multimin_fdfminimizer_conjugate_pr
```

Conjugate-gradients

```
gsl_multimin_fdfminimizer_vector_bfgs2
gsl_multimin_fdfminimizer_vector_bfgs
```

Quasi-Newton (BFGS)

```
gsl_multimin_fdfminimizer_steepest_descent
```

Steepest descent

```
gsl_multimin_fminimizer_nmsimplex2
gsl_multimin_fminimizer_nmsimplex
```

Nelder-Mead Simplex

gradient-free

scipy.optimize (Python)

estimation. Or, objects implementing HessianUpdateStrategy interface can be used to approximate the Hessian. Available quasi-Newton methods implementing this interface are:

- BFGS;
- SR1.

Type of solver. Should be one of

- 'Nelder-Mead' (see here)
- 'Powell' (see here)
- 'CG' (see here)

Gradient-free with linear search

- 'BFGS' (see here)

Quasi-Newton with linear search

- 'Newton-CG' (see here)
- 'L-BFGS-B' (see here)
- 'TNC' (see here)

Approximate Newton with linear search & bound constraints

- 'COBYLA' (see here)
- 'SLSQP' (see here)
- 'trust-constr'(see here)

Constrained fit

- 'dogleg' (see here)

Quasi-Newton (dogleg) with trust-region

- 'trust-ncg' (see here)
- 'trust-exact' (see here)

Quasi-Newton with exact trust-region solution

- 'trust-krylov' (see here)

Approximate Newton with trust-region

scipy.optimize: an example

```
import math
import numpy as np

stddev = 1.0
theta_points = np.array([-1.0, 2.0, 0.0, 0.0, 0.0])
n_theta = theta_points.size
x_points = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0], dtype=np.float32)
n_x = x_points.size

npA = np.array([math.pow(x_points[i], j)/math.factorial(j) for i in range(n_x) for j in range(n_theta) ]).reshape( (n_x,n_theta) )
npV = np.diag( [(stddev*stddev)]*n_x )
npVinv = np.linalg.inv(npV)
npB = np.matmul(npA.T, npVinv)

def npfun(x, args):
    npy = args
    npdelta = npy - np.matmul(npA,x)
    nploss = np.matmul( np.matmul( npdelta.T, npVinv), npdelta )
    return np.sum(nploss)/(n_x-n_theta)

rnd_data = np.random.multivariate_normal( np.matmul(npA, theta_points), npV )
res = minimize(fun=npfun, x0=theta_points, args=ynp, method='BFGS')
print('Loss- BFSG :', res.fun  )
```

=> $(y - Ax)^T V^{-1} (y - Ax)$

tf.keras.optimizers (Python)

`class Adadelta` : Optimizer that implements the Adadelta algorithm.

`class Adagrad` : Optimizer that implements the Adagrad algorithm.

`class Adam` : Optimizer that implements the Adam algorithm.

`class Adamax` : Optimizer that implements the Adamax algorithm.

`class Ftrl` : Optimizer that implements the FTRL algorithm.

`class Nadam` : Optimizer that implements the NAdam algorithm.

`class Optimizer` : Updated base class for optimizers.

`class RMSprop` : Optimizer that implements the RMSprop algorithm.

`class SGD` : Stochastic gradient descent and momentum optimizer.

Stochastic Gradient Descent with momentum

tf.keras.optimizers (Python)

`class Adadelta` :

`class Adagrad` : C

`class Adam` : Optir

`class Adamax` : Op

`class Ftrl` : Optir

`class Nadam` : Opt

`class Optimizer`

`class RMSprop` : C

`class SGD` : Stoch

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Minuit + Tensorflow (PyROOT)

Ref: TensorFlowAnalysis

```
def MinuitFit(sess, model, data_sample, integ_sample, call_limit = 50000) :

  tfpars = tf.trainable_variables()  # Create TF variables
  float_tfpars = [ p for p in tfpars if p.floating() ]

  nll = UnbinnedLogLikelihood(model, data_sample, integ_sample)
  gradient = tf.gradients(nll, float_tfpars)

  def fcn(npar, gin, f, par, istatus) :
    for i,p in enumerate(float_tfpars) : p.update(sess, par[i])
    f[0] = sess.run(nll)          # Calculate log likelihood
    if istatus == 2 :             # If gradient calculation is needed
      dnll = sess.run(gradient)  # Calculate analytic gradient
      for i in range(len(float_tfpars)) : gin[i] = dnll[i] # Pass gradient to MINUIT
    fcn.n += 1
    if fcn.n % 10 == 0 : print(fcn.n, istatus, f[0], sess.run(tfpars))

  fcn.n = 0
  minuit = TVirtualFitter.Fitter(0, len(float_tfpars))
  minuit.Clear()
  minuit.SetFCN(fcn)
  arglist = array.array('d', 10*[0])

  for n,p in enumerate(float_tfpars) :
    minuit.SetParameter(n, p.par_name, p.init_value, p.step_size, p.lower_limit, p.upper_limit)

  minuit.ExecuteCommand("SET GRA", arglist, 0)
  arglist[0] = call_limit
  minuit.ExecuteCommand("MIGRAD", arglist, 1)
```

Objective function is a Tensorflow op => benefit from Tensorflow speed-ups

Exploit native *automatic differentiation* of tensorflow

Implementation of objective function is transparent to Minuit

Matlab

Minimization Problems

Type	Formulation
Scalar minimization	$\min f(x)$
Unconstrained minimization	
Linear programming	
Mixed-integer linear programming	such that $A{\cdot}x \le b$, $Aeq{\cdot}x = beq$, $lb \le x \le ub$, x(intcon) is integer-valued.
Quadratic programming	$\min_x \frac{1}{2} x^T H x + c^T x$ such that $A{\cdot}x \le b$, $Aeq{\cdot}x = beq$, $lb \le x \le ub$
Constrained minimization	$\min_x f(x)$ such that $c(x) \le 0$, $ceq(x) = 0$, $A{\cdot}x \le b$, $Aeq{\cdot}x = beq$, $lb \le x \le ub$
Semi-infinite minimization	$\min_x f(x)$ such that $K(x,w) \le 0$ for all w, $c(x) \le 0$, $ceq(x) = 0$, $A{\cdot}x \le b$, $Aeq{\cdot}x = beq$, $lb \le x \le ub$

fminunc Algorithms

`fminunc` has two algorithms:

- `'quasi-newton'` (default) *BFGS or DFP*
- `'trust-region'`

NMinimize

NMinimize[f, x]
minimizes f numerically with respect to x.

NMinimize[f, {x, y, …}]
minimizes f numerically with respect to x, y, ….

NMinimize[{f, $cons$}, {x, y, …}]
minimizes f numerically subject to the constraints $cons$.

NMinimize[…, $x \in reg$]
constrains x to be in the region reg.

```
Optimization`NMinimizeDump`$Methods
 (* -> {Automatic, DifferentialEvolution, NelderMead,
        SimulatedAnnealing, RandomSearch, NonlinearInteriorPoint} *)
```

Interesting features (from a HEP perspective)

- **Efficient storage and read-out of multidimensional, arbitrary-type data (⇒ TTree)**
  - Can store (arrays of, pointers to) 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

- **Mathematical environment (⇒** ROOT::Math:: **)**
- **Linear algebra** (⇒ TLorentzVector, TMatrixD, TDecomp*, SMatrix)
- **N-dimensional minimization and integration (⇒** Minuit, GSL**)**
- **Monte Carlo studies (⇒** TRandom)
- **MVA's for regression/classification (⇒** TMVA **)**
- **Unfolding (⇒** TUnfold **)**

Math with ROOT: a complete example

```
#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

Math with ROOT: a complete example

Check positive-definiteness of the Hessian matrix

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

Use Factory to instantiate the minimzer (Minuit, GSL, ...)

```
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]);
}
```

Math with ROOT: a complete example

```
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;
```

Trigger the minimizer

Retrieve the information about the minimum

Alternatively, find the solution by one Newton step

Evaluate the objective function at the analytical solution

Math with ROOT: a complete example

```
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

Integrate the function in the box [-1,1]

Several MC algorithms available from GSL library (kVEGAS, kADAPTIVE, kPLAIN, ...)

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/
```

3. Download the material

```
> wget https://github.com/bianchini/SNS_DAS/archive/master.zip
> unzip master.zip; cd SNS_DAS-master/
> sudo apt-get install libgsl23 libgsl-dev
```

Basic ROOT commands

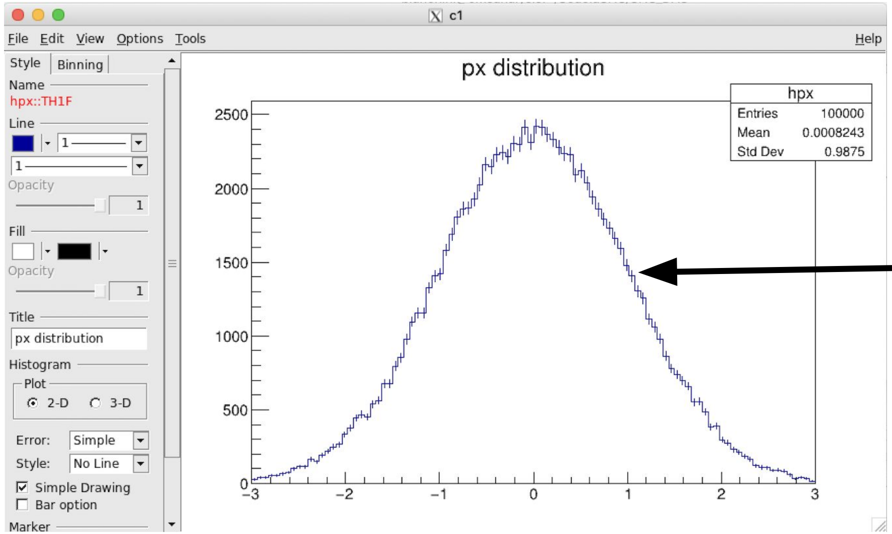
- Open ROOT without displaying initial logo
  > root -l
- Open ROOT in batch mode (no graphic windows)
  > root -b
- List content of the ROOT current directory
  root[0] .ls
- Exit ROOT
  root[0] .q
- The ROOT prompt provides auto-completion (TAB), forward-backward search (↑,↓) and regex search in the history (CTRL-R)

Fit Panel

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

  > root -l -e 'gROOT->ProcessLine(".L tree.C"); gROOT->ProcessLine("read_tree()")'

- **A default** TCanvas **is created and displayed**
  - On the top, select "View" -> "Editor", then click on any point of the histogram



Let's fit this histogram with a Gaussian
Right-click on any point of the histogram and choose "Fit Panel"

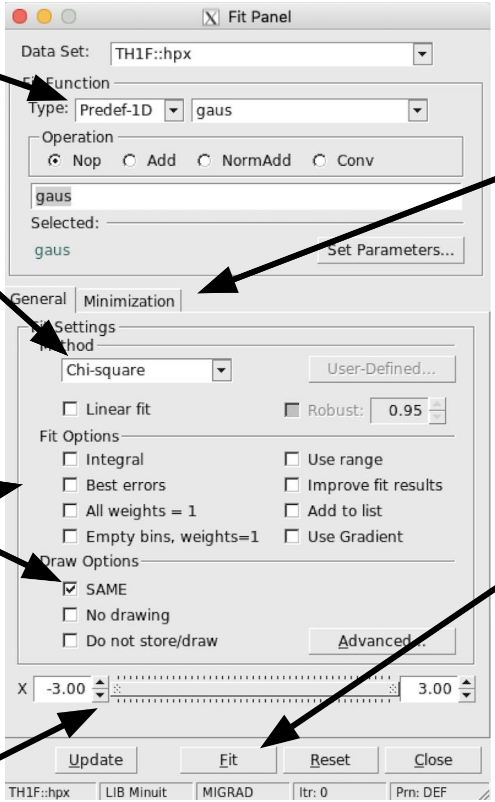
A simple 1D fit to a binned data set

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



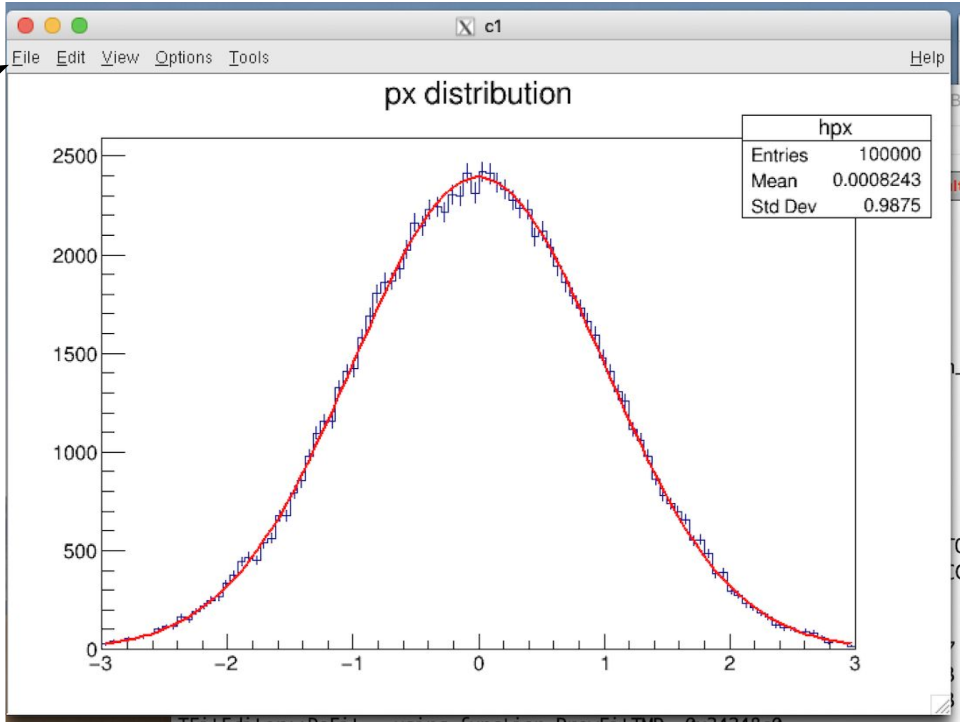
Minimizer

- Minuit(2): 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
```

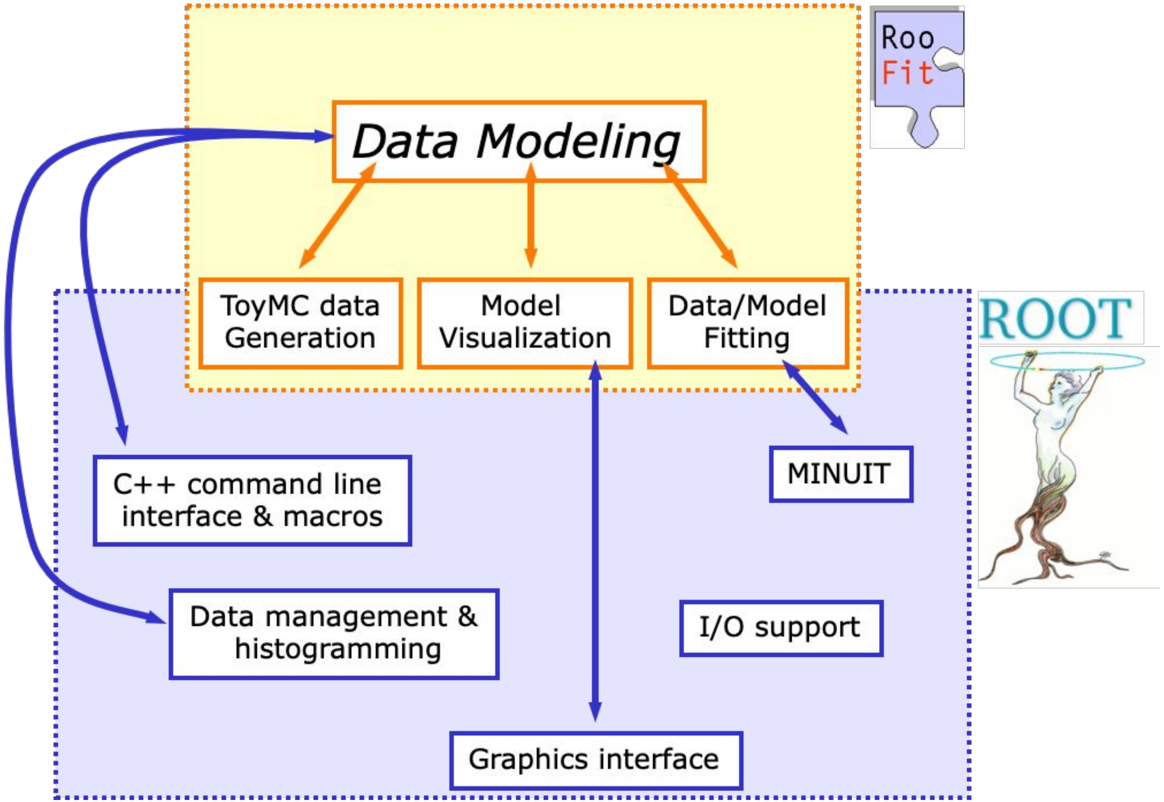
RooFit: a ROOT package for statistical analysis

- **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 Minuit transparent to the user
- Many pre-compiled models
- Takes care of p.d.f. normalization, variable marginalization, conditioning, simultaneous fit. Using optimizations whenever possible
- 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`

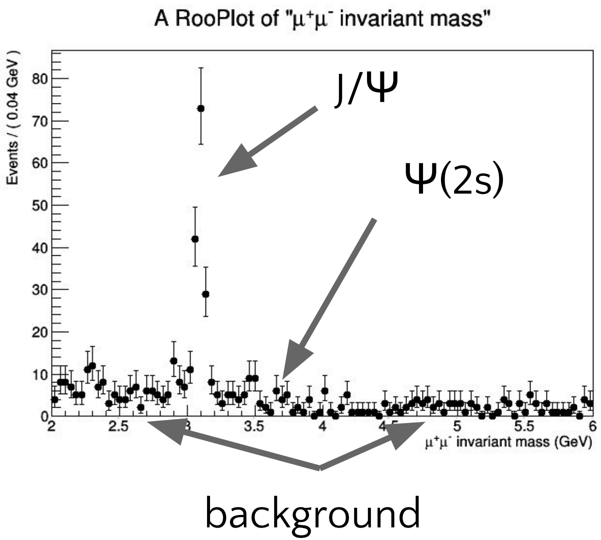
`RooAbsData` → RooDataHist (binned)

`RooAbsData` → RooDataSet (unbinned)

Cross section measurement

- **J/Ψ: a particle with mass of 3.096 GeV. It can be produced in pp collisions**
  - J/Ψ -> μμ is a neat narrow peak on top of a smooth background
  - A second excited state (⇒ Ψ(2s)) exists at the slightly larger mass of 3.686 GeV
- **The experiment produces a certain number of binary collisions (⇒** L**), and has an efficiency (⇒** ε**) of collecting Ψ(2s) -> μμ events.**
  We wish to determine:

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



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

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))
```

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))
```

The name of the observable saved into data is mass
(⇒ data imported by name)

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)
```

```
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

An empirical p.d.f. for the J/Ψ (⇒ Crystal Ball function)

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()
```

Fitting with RooFit (⇒ roofit.py)

Total yields (for extended ML fit)

```
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

The fit is done here!

```
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()
```

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))
```

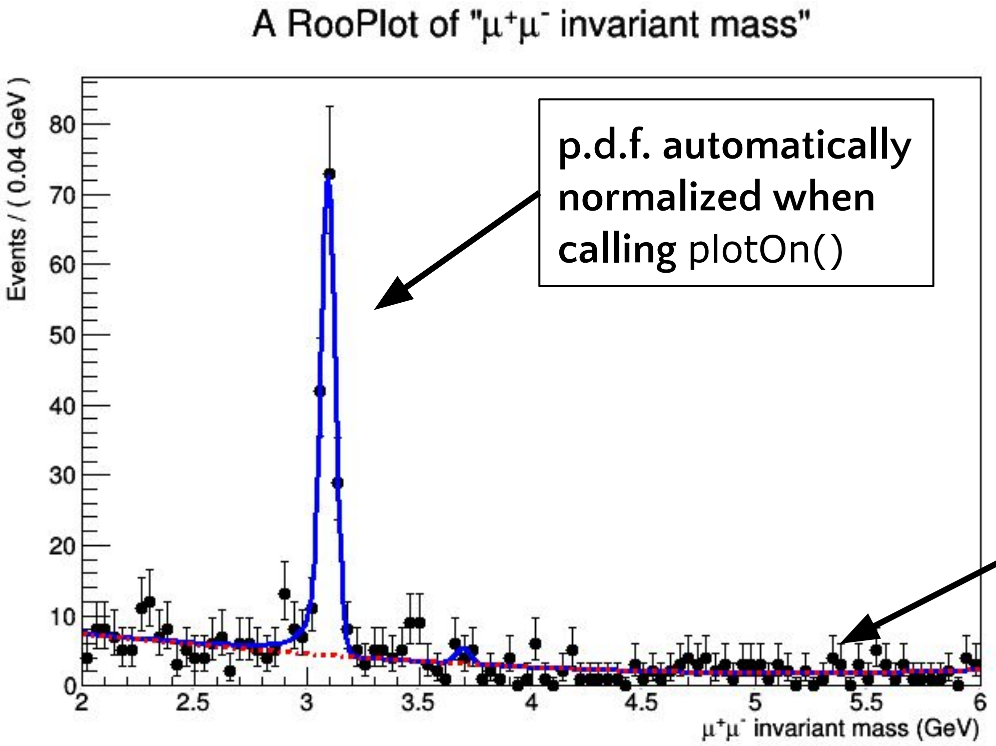
A fix for bad memory handling…

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 a comprehensive summary into a RooWorkspace

Fitting with RooFit (⇒ roofit.py)



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

Asymmetric poisson errors shown by default

Pulls/residuals readily obtained as:

```
frame->makePullHist()
frame->makePullHist()
```

Fitting with RooFit (⇒ roofit.py)

So, what is the measured cross section of Ψ(2s)->μμ?

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

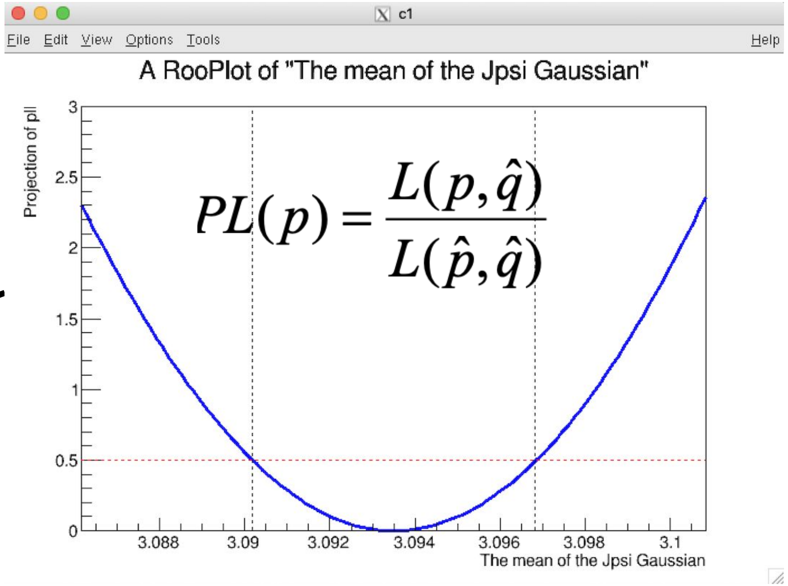
```
RooRealVar::cross_psi = 9.65042 +/- 7.94308  L(0 - 40) // [pb]
```

Fitting with RooFit (⇒ profile.C)

Suppose we want to look at the profile log-likelihood **for one POI (**say, J/Ψ mass**)**

- an <u>intensive task</u> (one minimization per value of the POI)
- RooFit has a built-in method for doing it

```
{
  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();
}
```

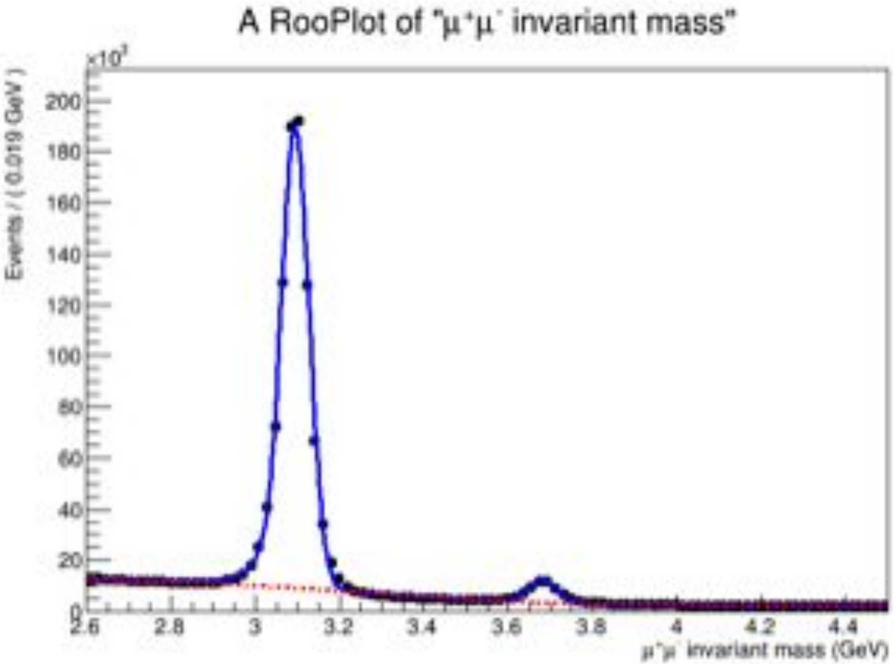


$$PL(p) = \frac{L(p,\hat{q})}{L(\hat{p},\hat{q})}$$

Hands-on

- **Download a large sample of CMS open data (L=11.6/fb)**
  - Pre-processed for you into a flat TTree with one branch only
    > wget -O DataSet_highstat.root cern.ch/arizzi/out.root
- **Modify** roofit.py **to run on the new input**
  - The input file contains a TTree, not a RooDataSet. *Hint: RooDataSet reference guide*
  - Remember that data is imported by name. *Hint: Events->Print()*
  - Is it necessary to perform an unbinned fit? *Hint: RooDataHist reference guide*
  - Are the RooRealVar ranges feasible? *Hint: what is the total number of events?*
  - Is the fitting range adequate? *Hint: take a quick look at the data before making a fit*
  - Can you improve the background/signal modeling? *Hint: add more d.o.f.*

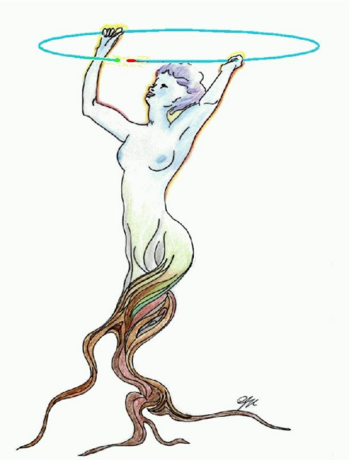
Hands-on



A RooPlot of "μ+μ- invariant mass"
Events / (0.019 GeV)
μ+μ- invariant mass (GeV)

Backup

ROOT is...



- **A scientific software toolkit for data analysis**
  - User code (⇒ macros) is standard C++ using ROOT classes (⇒ T***)
    - TFile ⇒ basic I/O
    - TH1F ⇒ 1D-histogram with floating-point precision
    - TF1 ⇒ a generic real-valued 1D-function
  - Class methods start with capital letters (⇒ histo.Draw();)
  - The code can be either interpreted or compiled
  - Any ROOT class can be used through automatic Python binding (⇒ PyROOT)

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

- **Project developed & maintained by CERN**
  - 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)
- **Operations**
  - Commands issued into the ROOT prompt:

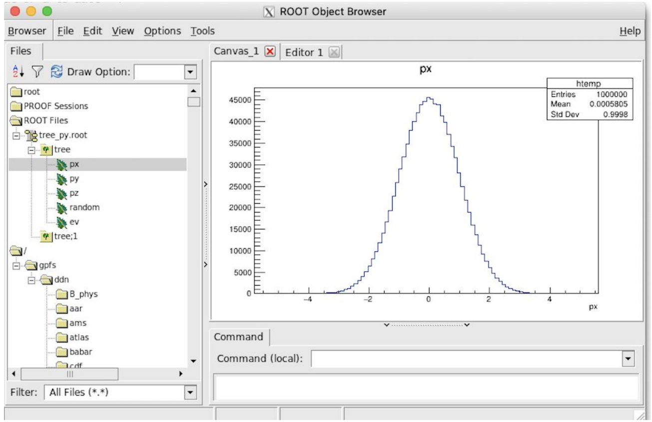
```
> root
root [0] TFile* f = TFile::Open("my_file.root")
```

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

```
> root
root[0] .L my_macro.C
root[1] my_function()
root[2] .x snippet.C
```

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

```
> root
root[0] .L my_macro.C++
root[1] my_function()
```



ROOT comes with its own ***GUI*** *for inspecting files, visualize, and customize plots*

A concrete example: ROOT-based data for CMS

Extract of a typical file content for simulated data (~50 kB/event)

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

SW version, file ID, …

PSet specifying how file was made

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

Where this file comes from

The TTree containing the actual data

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

A concrete example: ROOT-based data for CMS

Extract of a typical Event content for simulated data

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"

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)

A classical C++ function. No need to write main()
N.B. ROOT defines new types for int, float, …
E.g.: int → Int_t

```
#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();

}
```

No need to call explicit destructors here.
⇒ object will be destroyed once out-of-scope.
⇒ it allows to work interactively with the TTree within the ROOT prompt

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();

}
```

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

Objects that we want to be persistently save (e.g. tree) will be written into 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 the output

Define the TBranches for this TTree

```
  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();

}
```

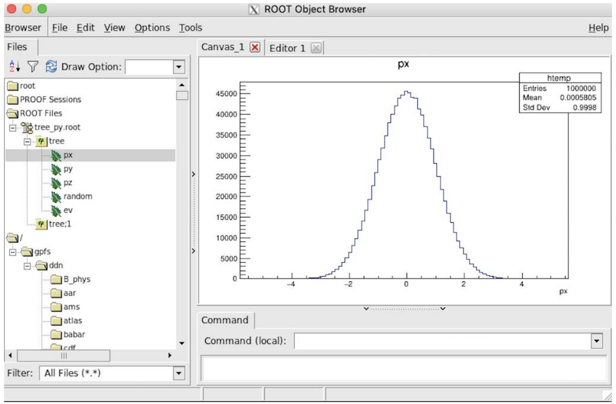
Leaf type gets specified here (e.g. "/F", "/I", "/D")

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. Quit ROOT, then:

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

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

```
> root -l tree_C.root
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")
root[2] tree->Draw("px:py", "py>0 && random<0.5", "colz")
root[3] tree->Draw("px:py:pz", "py>0 && random<0.5", "lego")
```

- **The output of** Draw() **can be put into a binned histogram (**e.g. for future work**):**

```
root[4] tree->Draw("px>>h(100,-3,3)")
root[5] h->GetMean()
```

- **Actually, 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 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 all the 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)
```

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 Txxx accessible using ROOT.Txxx

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 advantages 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 multicore 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)
```

Caveats:

1. PyROOT can be much slower than compiled C++ *(see e.g. the two examples just discussed)*
2. Handling of memory sometimes problematic. Extra care needed to avoid memory leaks.

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
```

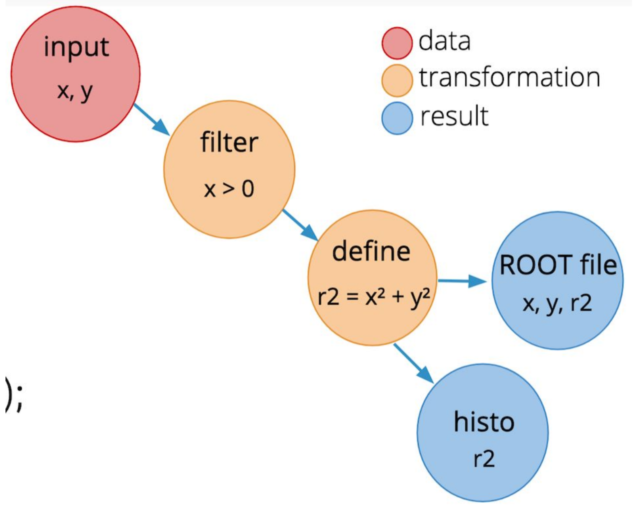
N.B.: convertion into **numpy** array will fail due to array-like branch

More into the future: RDataFrame (⇒ test_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 test_RDF.C
```