Tutorial on ROOT & RooFit

Arun Nayak
Institute of Physics, Bhubneswar

ML4HEP-V4 Online Preschool

ROOT Installation

The easier way to install the latest version of ROOT is using a virtual environment (e.g. Anaconda, https://www.anaconda.com/). You can use Miniconda instead for this tutorial (a lite version, https://www.anaconda.com/docs/getting-started/miniconda/main).
Using a virtual environment helps avoid any interference with other installed packages in your PC/laptop.

For installing the latest version of ROOT, follow the instructions here:
https://root.cern/install/#conda

—--
So, you can use the following commands (in the terminal window) to install root in conda
conda config --set channel_priority strict
conda create -c conda-forge --name root-tutorial root
conda activate root-tutorial
—---

Once ROOT is installed, check the ROOT installed path:
echo $ROOTSYS

Getting Started

To start ROOT, type root -l into the command line.
(The -l flag suppresses the splash screen)

To quit root type .q. If root crashes you can exit by typing the usual linux kill commands such as ctrl+c and ctrl+z.

You can find documents, tutorials, and courses on ROOT at
https://root.cern/learn/

Also, all ROOT classes
https://root.cern.ch/doc/master/annotated.html

Note:
ROOT Data Analysis Framework is a high performance C++ framework:
some knowledge of C++ is required

Using Interactive shell

ROOT has an interpreter for macros (Cling) that you can run from the command line or run like applications.
It is also an interactive shell that can evaluate arbitrary statements and expressions.

Using as a calculator:

```
(root-tutorial) nayak@MacBookAir tutorial_root % root -l
root [0] 2+2
(int) 4
root [1] 2*(4+2)/12.
(double) 1.0000000
root [2] sqrt(2.)
(double) 1.4142136
root [3] 3 > 1
(bool) true
root [4] TMath::Pi()
(double) 3.1415927
root [5] TMath::Erf(.2)
(double) 0.22270259
```

Can even do more elaborated numerical calculation:

```
root [6] double x=.5
(double) 0.50000000
root [7] int N = 10
(int) 10
root [8] double exp_value = 0
(double) 0.0000000
root [9] for (int i = 0; i < N; i++)exp_value += (TMath::Power(x,i)/TMath::Factorial(i))
root [10] cout << "computed exp(0.5) = "<<exp_value<<" and exp(0.5) = "<<TMath::Exp(0.5)<<endl;
computed exp(0.5) = 1.64872 and exp(0.5) = 1.64872
```

Using the Graphical User Interface (GUI)

ROOT provides both a programming interface to use in own applications and a **graphical user interface for interactive data analysis**

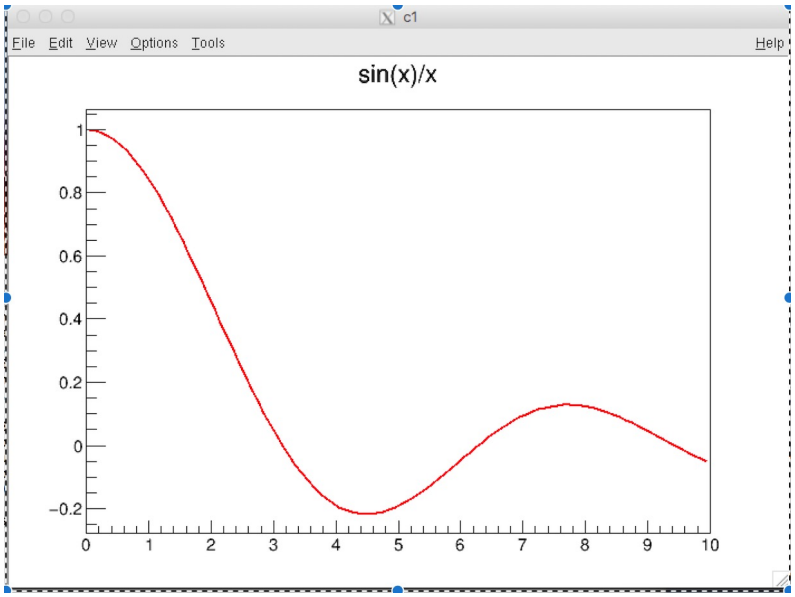
The basic whiteboard on which an object is drawn in ROOT is called a “canvas” (defined by the class **TCanvas**)

Drawing a function:
root [0] **TF1** f1**(**"func1","sin(x)/x",0,10**)**
root [1] f1.Draw**()**
Info in <TCanvas::MakeDefCanvas>: created default TCanvas with name c1

ROOT’s Python interface allows access the full ROOT C++ functionality from Python
https://root.cern/manual/python/

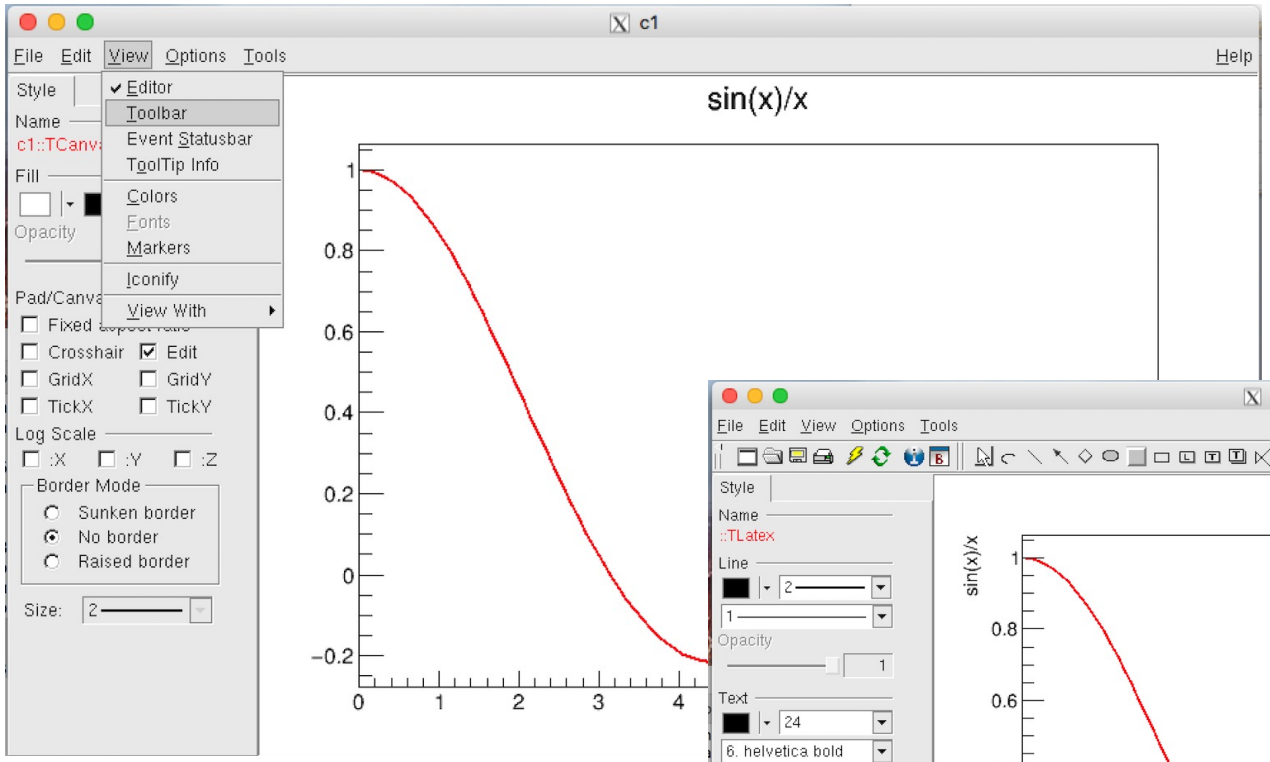
In Python:

```
>>> import ROOT
>>> f = ROOT.TF1("f1", "sin(x)/x", 0., 10.)
>>> f.Draw()
```



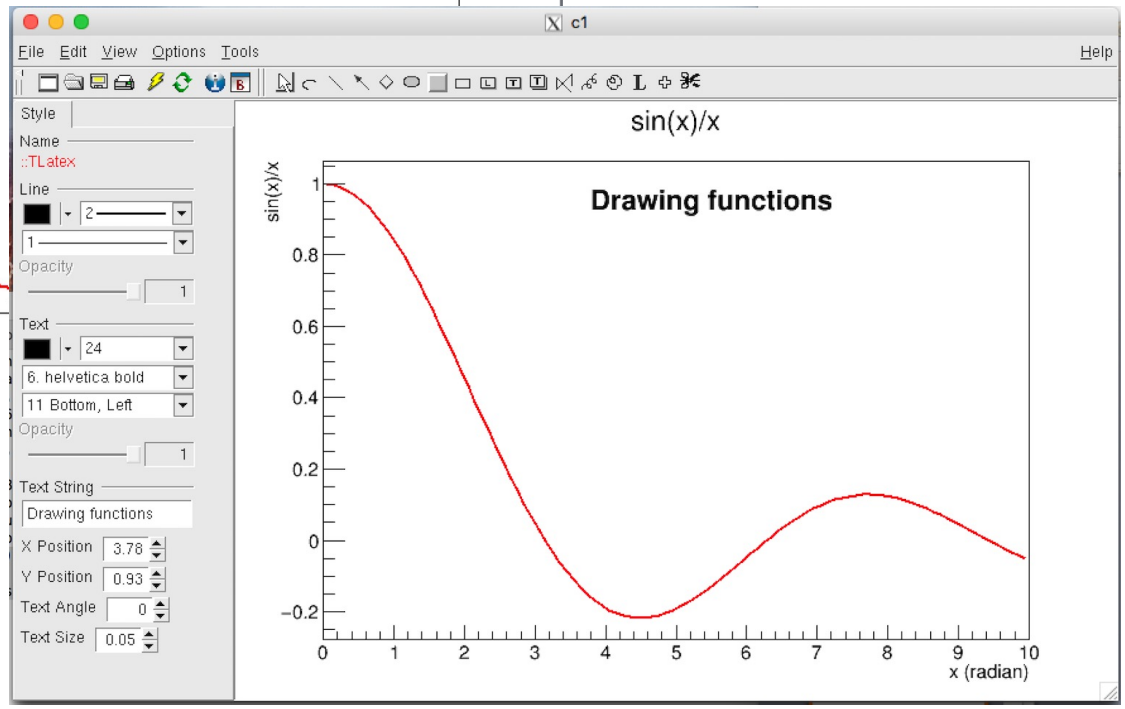
Using the GUI

Toolbar & Editor



Write on canvas using Latex:

Click on “L”,
then click on the canvas,
write any comment that you like.
For latex symbols, use #symbol_name (e.g. #gamma)

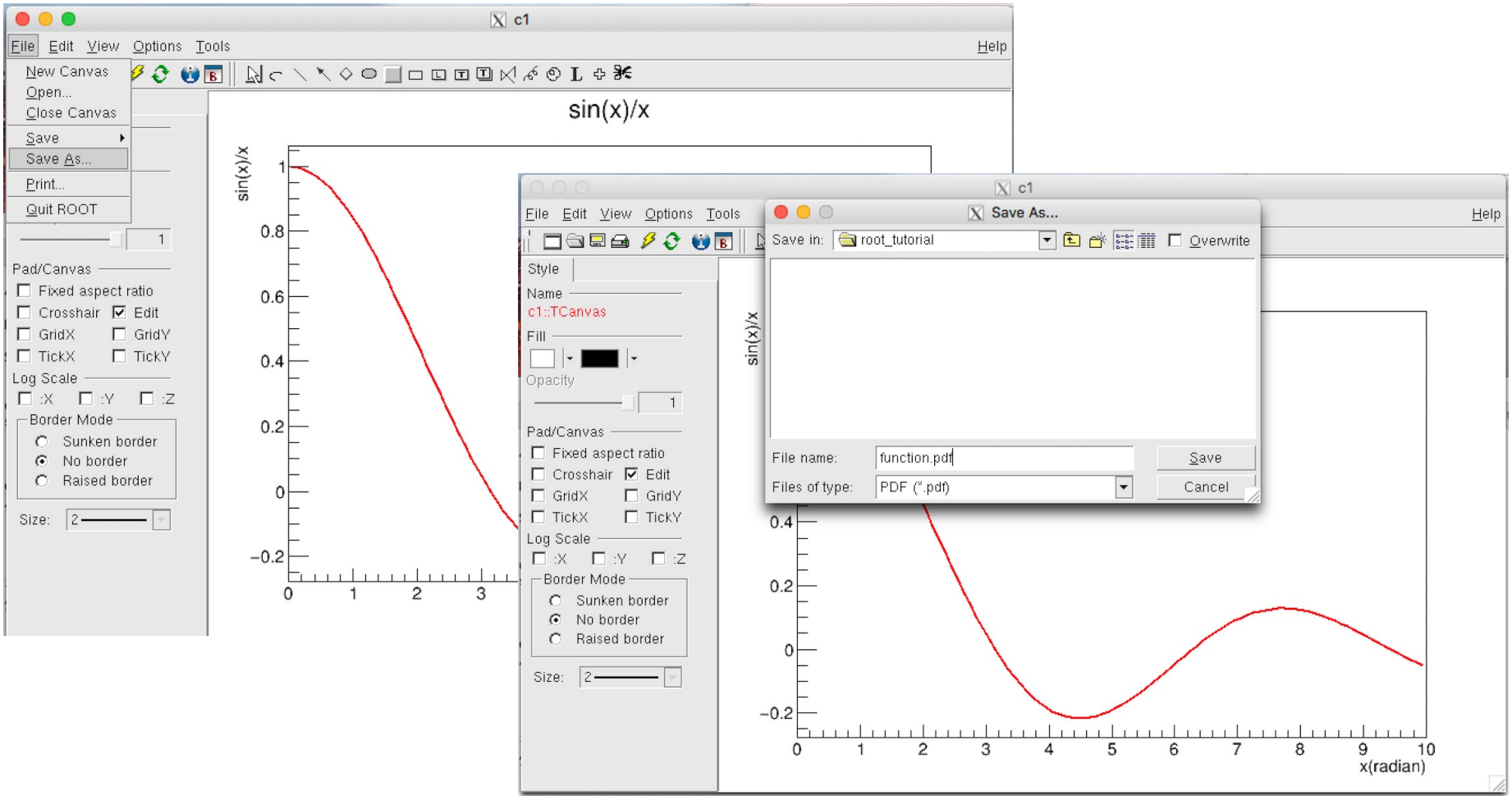


Writing axis title:

Right click on axis, click on “setTitle”
Write the title on the box, then click “OK”

Using the GUI

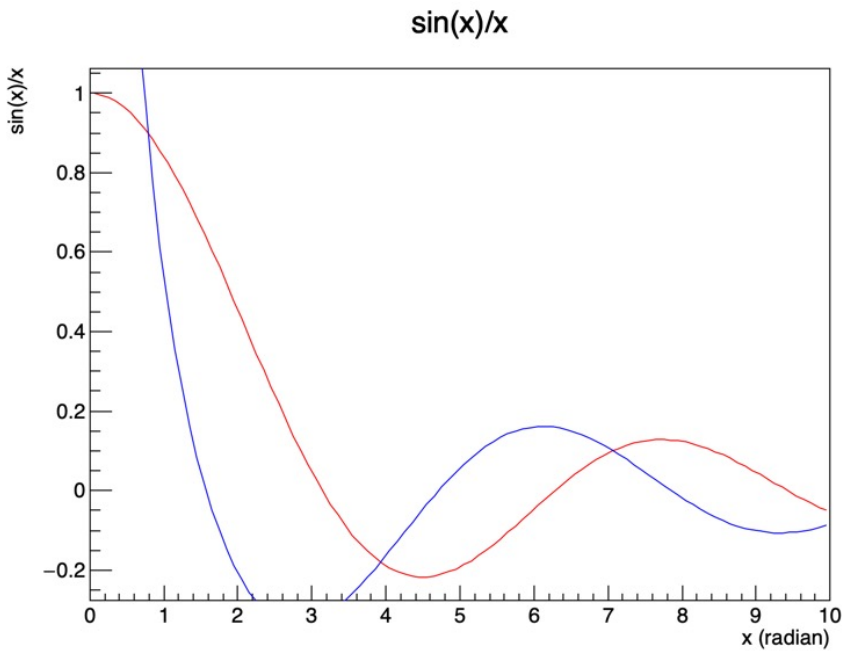
Saving the Canvas



Writing a Script

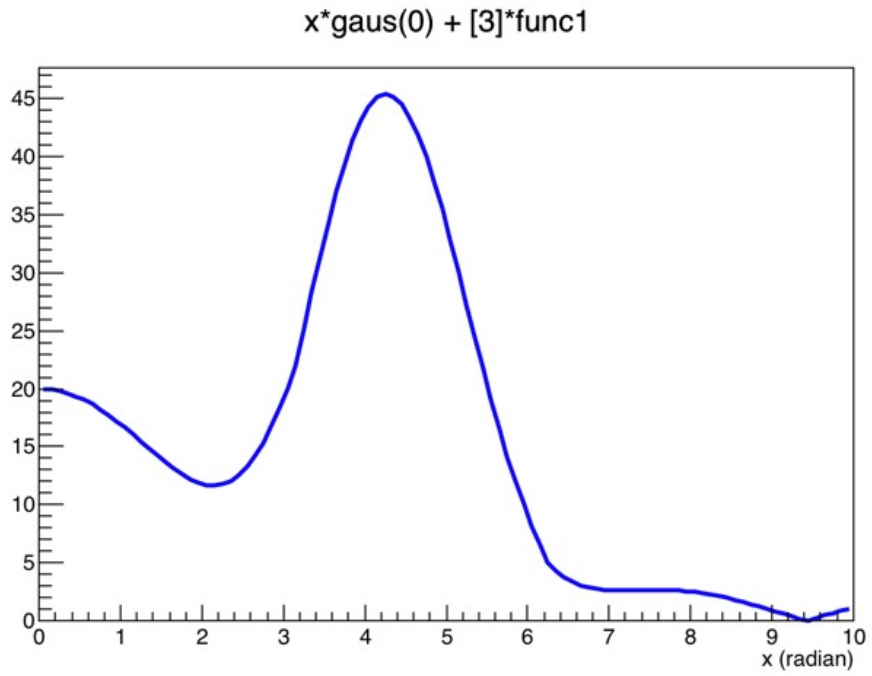
Functionalities can be accessed both from the Python prompt and from a Python script

```
import ROOT

def functions1():

    # Create canvas
    c1 = ROOT.TCanvas("c1", "Canvas", 800, 600)

    # Create functions
    f1 = ROOT.TF1("func1", "sin(x)/x", 0, 10)
    f2 = ROOT.TF1("func2", "cos(x)/x", 0, 10)

    # Axis titles
    f1.GetXaxis().SetTitle("x (radian)")
    f1.GetYaxis().SetTitle("sin(x)/x")

    # Line colors
    f1.SetLineColor(ROOT.kRed)
    f2.SetLineColor(ROOT.kBlue)

    # Draw functions
    f1.Draw()
    f2.Draw("same")

    # Save canvas
    c1.SaveAs("function.png")


if __name__ == "__main__":
    functions1()
```

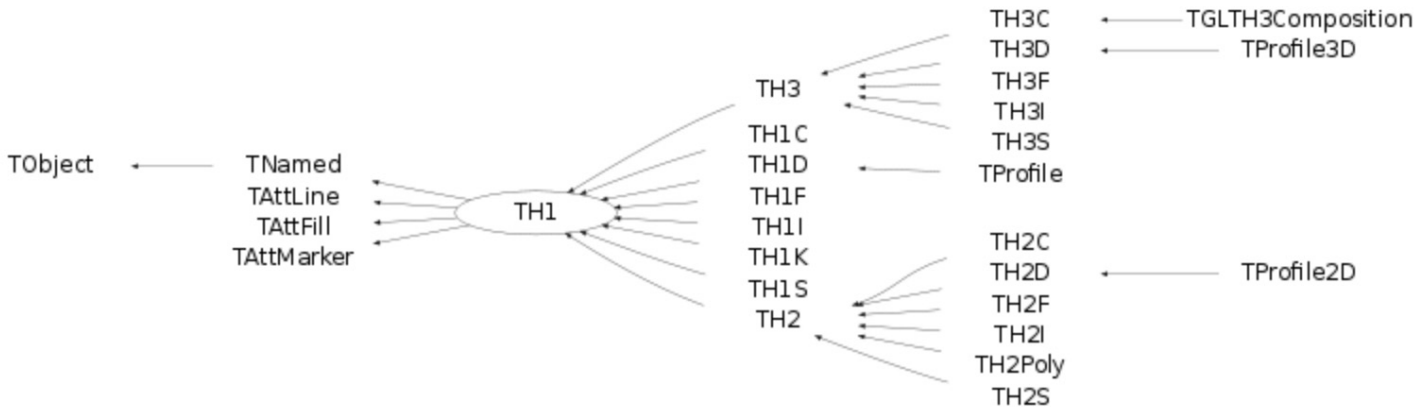


Combining formulas

```python
import ROOT

def functions2():

    # Create canvas
    c1 = ROOT.TCanvas("c1", "Canvas", 800, 600)

    # Define formula
    func1 = ROOT.TFormula("func1", "abs(sin(x)/x)")

    # Define TF1 using the formula
    func2 = ROOT.TF1(
        "func2",
        "x*gaus(0) + [3]*func1",
        0,
        10
    )

    # Set parameters:
    # gaus(0) -> parameters [0], [1], [2]
    # [3]     -> scale factor for func1
    func2.SetParameters(10, 4, 1, 20)

    # Style settings
    func2.SetLineColor(ROOT.kBlue)
    func2.SetLineWidth(6)

    # Axis title
    func2.GetXaxis().SetTitle("x (radian)")

    # Draw function
    func2.Draw()

    # Save canvas
    c1.SaveAs("function2.png")


if __name__ == "__main__":
    functions2()
```



Histograms



The class hierarchy of histogram classes

All ROOT histogram classes are derived from the base class **`TH1`** (see figure above). This means that two-dimensional and three-dimensional histograms are seen as a type of a one-dimensional histogram, in the same way in which multidimensional C arrays are just an abstraction of a one-dimensional contiguous block of memory.

- **`TH1C, TH2C and TH3C`** contain one byte per bin (maximum bin content = 255)
- **`TH1S, TH2S and TH3S`** contain one short per bin (maximum bin content = 65 535).
- **`TH1I, TH2I and TH3I`** contain one integer per bin (maximum bin content = 2 147 483 647).
- **`TH1F, TH2F and TH3F`** contain one float per bin (maximum precision = 7 digits).
- **`TH1D, TH2D and TH3D`** contain one double per bin (maximum precision = 14 digits).
- **`TProfile`** : one dimensional profiles
- **`TProfile2D`** : two dimensional profiles

Histograms

Creating histograms

```
// using various constructors
TH1* h1 = new TH1I("h1", "h1 title", 100, 0.0, 4.0);
TH2* h2 = new TH2F("h2", "h2 title", 40, 0.0, 2.0, 30, -1.5, 3.5);
TH3* h3 = new TH3D("h3", "h3 title", 80, 0.0, 1.0, 100, -2.0, 2.0,
                   50, 0.0, 3.0);

// cloning a histogram
TH1* hc = (TH1*)h1->Clone();

// projecting histograms
// the projections always contain double values !
TH1* hx = h2->ProjectionX(); // ! TH1D, not TH1F
TH1* hy = h2->ProjectionY(); // ! TH1D, not TH1F
```

Variable binning:

```
const Int_t NBINS = 5;
Double_t edges[NBINS + 1] = {0.0, 0.2, 0.3, 0.6, 0.8, 1.0};
TH1* h = new TH1D("h1", "Hist with variable bin width", NBINS, edges);
```

Filling histograms

```
h1->Fill(x);
h1->Fill(x,w); // with weight
h2->Fill(x,y);
h2->Fill(x,y,w);
h3->Fill(x,y,z);
h3->Fill(x,y,z,w);
```

Random Numbers and Histograms

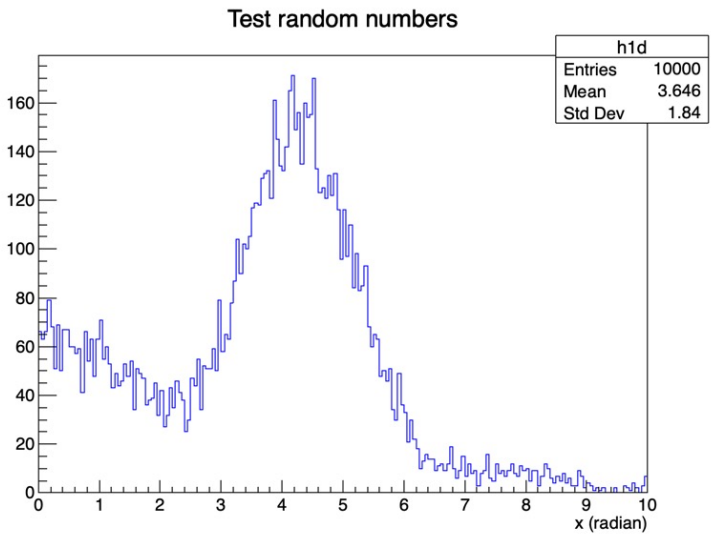
TH1::FillRandom() can be used to randomly fill a histogram using the contents of an existing TF1 function or another TH1 histogram (for all dimensions).

```
root[] TH1F h1("h1","Histo from a Gaussian",100,-3,3);
root[] h1.FillRandom("gaus",10000);
```

```
root[] TH1F h2("h2","Histo from existing histo",100,-3,3);
root[] h2.FillRandom(&h1, 1000);
```

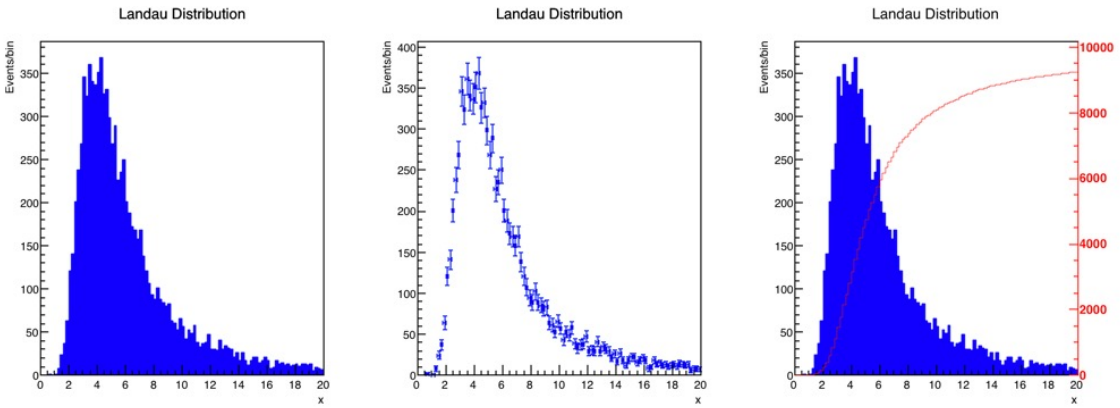
Example:

```
h1d = ROOT.TH1D("h1d", "Test random numbers", nbinsx = 200, xlow = 0.0, xup = 10.0)

# 10000 values sampled from the above distribution.
h1d.FillRandom("func2", 10000)

# Style settings
h1d.SetLineColor(ROOT.kBlue)
h1d.SetLineWidth(2)

# Axis title
h1d.GetXaxis().SetTitle("x (radian)")
# Draw function
h1d.Draw()
```



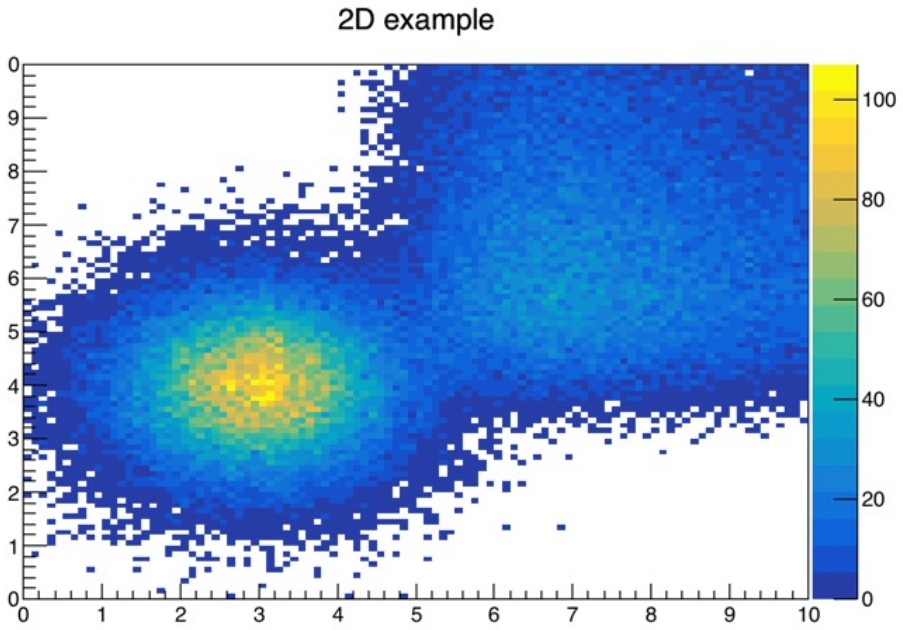
Histograms Example

```
import ROOT

def histograms1():

    # Create canvas
    c1 = ROOT.TCanvas("c1", "", 1000, 400)

    ROOT.gStyle.SetOptStat(False)

    # Divide canvas into 3 pads
    c1.Divide(3, 1)

    # Create histogram
    landau = ROOT.TH1F(
        "landau",
        "Landau Distribution",
        100,
        0.0,
        20.0
    )

    # Fill histogram with Landau random numbers
    for i in range(10000):
        landau.Fill(ROOT.gRandom.Landau(4))

    # Axis titles
    landau.GetXaxis().SetTitle("x")
    landau.GetYaxis().SetTitle("Events/bin")

    # Style
    landau.SetFillColor(ROOT.kBlue)
    landau.SetLineColor(ROOT.kBlue)

    # Pad 1
    c1.cd(1)
    landau.Draw()

    # Pad 2
    c1.cd(2)
    landau.Draw("E1")

    # Pad 3
    c1.cd(3)

    # Create cumulative distribution histogram
    cdf = ROOT.TH1F(
        "cdf",
        "Cumulative distribution",
        100,
        0.0,
        20.0
    )

    cumulative_sum = 0.0

    for i in range(1, 101):
        cumulative_sum += landau.GetBinContent(i)
        cdf.SetBinContent(i, cumulative_sum)

    # Scale cdf to pad coordinates
    landau.Draw()
    c1.Update()

    rightmax = 1.1 * cdf.GetMaximum()
    scale = ROOT.gPad.GetUymax() / rightmax

    cdf.SetLineColor(ROOT.kRed)
    cdf.Scale(scale)

    cdf.Draw("hist same")
```



Histograms Example

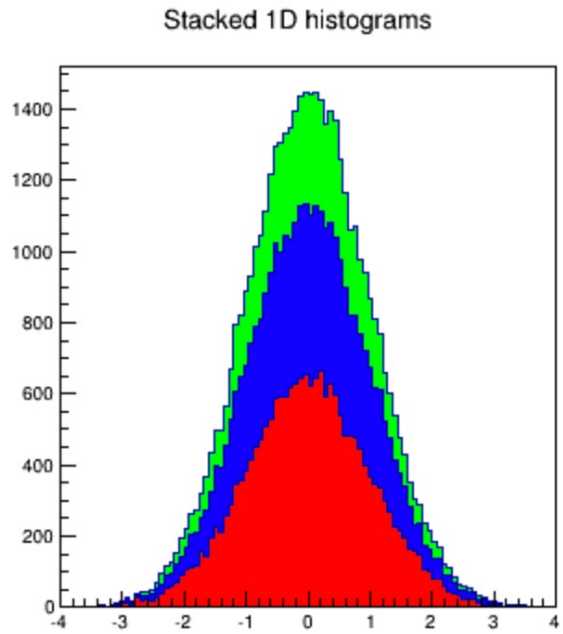
```
import ROOT

def histograms2():

    # Create canvas
    c1 = ROOT.TCanvas("c1", "2D Histogram", 800, 600)

    ROOT.gStyle.SetOptStat(False)

    # Define 2D function
    f2 = ROOT.TF2(
        "f2", "xygaus(0) + xylandau(5)",
        0, 10, 0, 10)

    # Set parameters
    f2.SetParameters(
        1, 3, 1, 4, 1,
        10, 7, 1, 6, 1
    )

    # Create 2D histogram
    h2 = ROOT.TH2F(
        "h2", "2D example",
        100, 0.0, 10.0, 100, 0.0, 10.0)

    # Fill histogram using random samples from f2
    h2.FillRandom("f2", 100000)

    # Draw histogram
    h2.Draw("LEGO2")

    # Save canvas
    c1.SaveAs("histograms2.png")
```

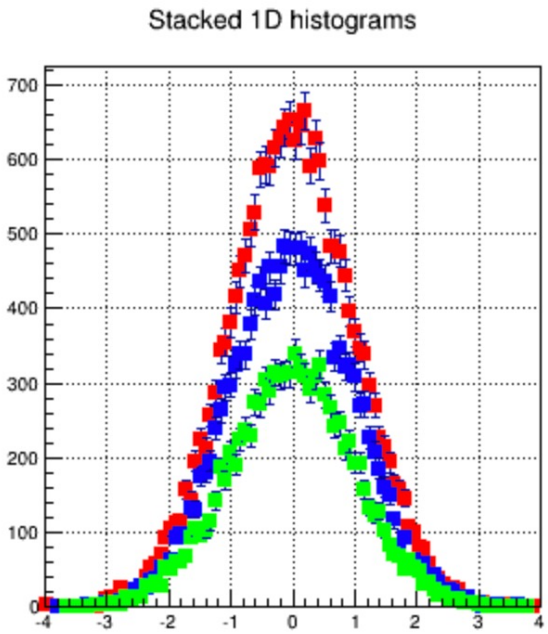


Histogram Stacks

A THStack is a collection of TH1 (or derived) objects. Use THStack::Add(TH1 *h) to add a histogram to the stack. The THStack does not own the objects in the list.

```
$ROOTSYS/tutorials/hist/hist023_THStack_simple.C
```

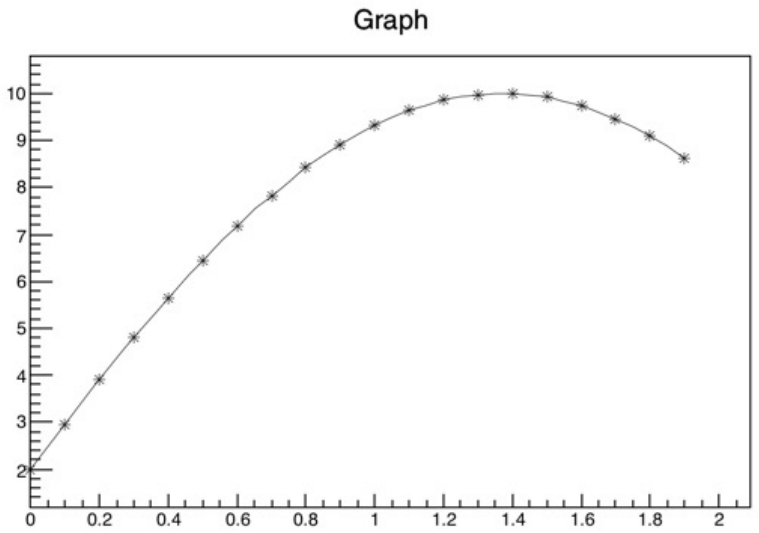




Graphs

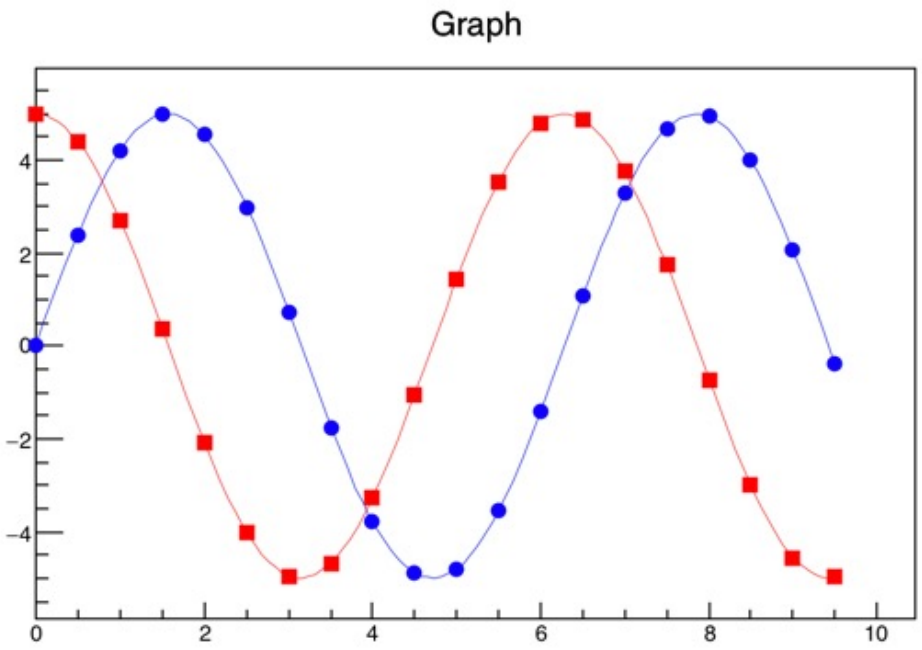
```
import ROOT
import math
from array import array

def graph1():

    n = 20

    # Create arrays for x and y values
    x = array('d')
    y = array('d')

    for i in range(n):
        xv = i * 0.1
        yv = 10 * math.sin(xv + 0.2)

        x.append(xv)
        y.append(yv)

    # Create graph
    gr = ROOT.TGraph(n, x, y)

    # Create canvas
    c1 = ROOT.TCanvas(
        "c1", "Graph Draw Options",
        200, 10, 600, 400)

    # Draw graph with:
    # A = axis
    # C = smooth curve
    # * = star marker
    gr.Draw("AC*")

    # Save canvas
    c1.SaveAs("graph1.png")
```

You can change
Marker size, color, style etc..
Line color, width etc..
Draw options



Superimposing Two Graphs

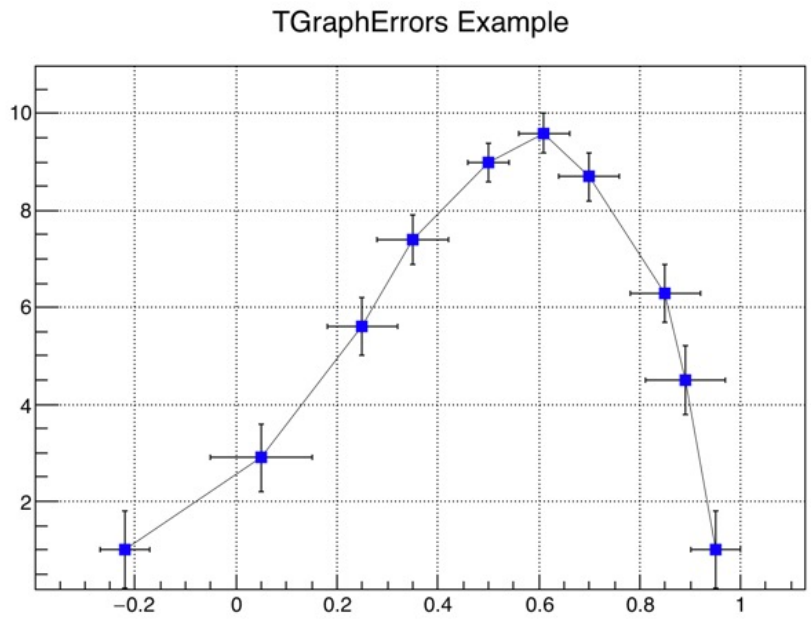
```
import ROOT
import math
from array import array

def graph2():

    n = 20

    # Arrays for graph 1
    x = array('d')
    y = array('d')

    # Arrays for graph 2
    x1 = array('d')
    y1 = array('d')

    # Fill arrays
    for i in range(n):

        xv = i * 0.5

        x.append(xv)
        y.append(5 * math.sin(xv))

        x1.append(xv)
        y1.append(5 * math.cos(xv))

    # Create first graph
    gr1 = ROOT.TGraph(n, x, y)

    gr1.SetLineColor(ROOT.kBlue)
    gr1.SetMarkerStyle(20)
    gr1.SetMarkerSize(1.0)
    gr1.SetMarkerColor(ROOT.kBlue)

    # Create second graph
    gr2 = ROOT.TGraph(n, x1, y1)
```




```
gr2.SetLineColor(ROOT.kRed)
gr2.SetMarkerStyle(21)
gr2.SetMarkerSize(1.0)
gr2.SetMarkerColor(ROOT.kRed)

# Create canvas
c1 = ROOT.TCanvas(
    "c1", "Graph Draw Options",
    200, 10, 600, 400)

# Draw graphs
gr1.Draw("ACP")
gr2.Draw("CP same")

# Save canvas
c1.SaveAs("graph2.png")
```

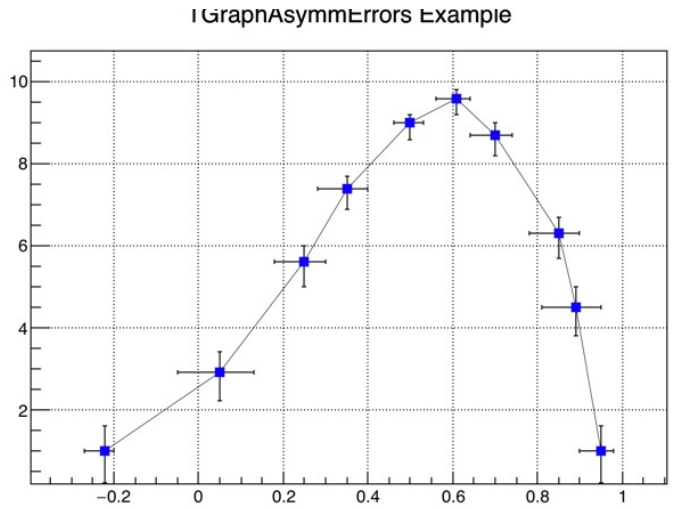
Graphs with Error Bars

```
# Create canvas
c1 = ROOT.TCanvas(
    "c1", "A Simple Graph with Error Bars",
    200, 10, 700, 500)

c1.SetGrid()

# Number of points
n = 10

# Coordinate arrays
x = array('f', [-0.22, 0.05, 0.25, 0.35, 0.5,
                0.61, 0.7, 0.85, 0.89, 0.95])

y = array('f', [1, 2.9, 5.6, 7.4, 9,
                9.6, 8.7, 6.3, 4.5, 1])

# Error arrays
ex = array('f', [0.05, 0.1, 0.07, 0.07, 0.04,
                 0.05, 0.06, 0.07, 0.08, 0.05])

ey = array('f', [0.8, 0.7, 0.6, 0.5, 0.4,
                 0.4, 0.5, 0.6, 0.7, 0.8])

# Create TGraphErrors
gr = ROOT.TGraphErrors(n, x, y, ex, ey)

# Graph styling
gr.SetTitle("TGraphErrors Example")
gr.SetMarkerColor(ROOT.kBlue)
gr.SetMarkerStyle(21)

# Draw graph
gr.Draw("ALP")
```



Graphs with Asymmetric Error Bars

```
import ROOT
from array import array

def graph4():

    # Create canvas
    c1 = ROOT.TCanvas(
        "c1", "A Simple Graph with Asymmetric Error Bars",
        200, 10, 700, 500)

    c1.SetGrid()

    # Number of points
    n = 10

    # Coordinate arrays
    x = array('d', [
        -0.22, 0.05, 0.25, 0.35, 0.5,
        0.61, 0.7, 0.85, 0.89, 0.95
    ])

    y = array('d', [
        1, 2.9, 5.6, 7.4, 9,
        9.6, 8.7, 6.3, 4.5, 1
    ])

    # Lower x errors
    exl = array('d', [
        0.05, 0.1, 0.07, 0.07, 0.04,
        0.05, 0.06, 0.07, 0.08, 0.05
    ])

    # Lower y errors
    eyl = array('d', [
        0.8, 0.7, 0.6, 0.5, 0.4,
        0.4, 0.5, 0.6, 0.7, 0.8
    ])

    # Upper x errors
    exh = array('d', [
        0.02, 0.08, 0.05, 0.05, 0.03,
        0.03, 0.04, 0.05, 0.06, 0.03
    ])

    # Upper y errors
    eyh = array('d', [
        0.6, 0.5, 0.4, 0.3, 0.2,
        0.2, 0.3, 0.4, 0.5, 0.6
    ])

    # Create TGraphAsymmErrors
    gr = ROOT.TGraphAsymmErrors(
        n, x, y, exl, exh, eyl, eyh)

    # Styling
    gr.SetTitle("TGraphAsymmErrors Example")
    gr.SetMarkerColor(ROOT.kBlue)
    gr.SetMarkerStyle(21)

    # Draw graph
    gr.Draw("ALP")
```



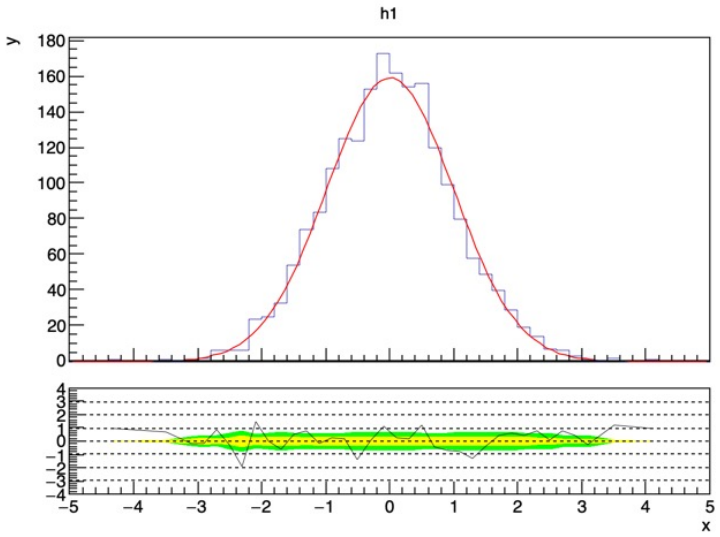
More details:
https://root.cern.ch/doc/master/classTGraphAsymmErrors.html

Fitting

Fitting a function to Histograms and Graphs,
Using TH1::Fit() and TGraph::Fit() methods

Returns a TFitResultPtr

```
TFitResultPtr Fit(TF1 *f1,
                  const char *option="",
                  const char *goption="",
                  double xxmin=0.0,
                  double xxmax=0.0);
```

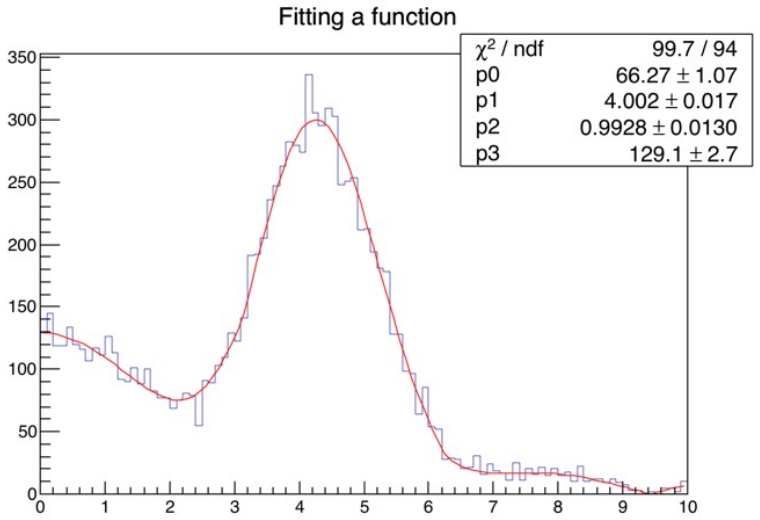


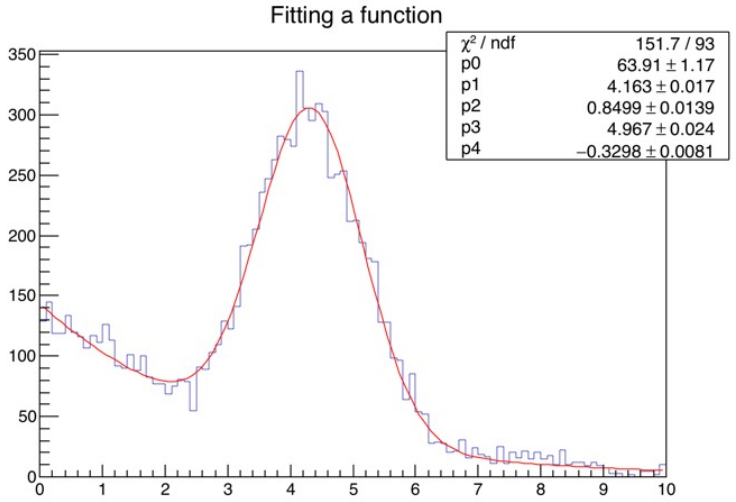
Example Macro

```
gStyle->SetOptStat(0);
auto c1 = new TCanvas("c1", "fit residual simple");
auto h1 = new TH1D("h1", "h1", 50, -5, 5);
h1->FillRandom("gaus", 2000);
h1->Fit("gaus", "0");
h1->GetXaxis()->SetTitle("x");
h1->GetYaxis()->SetTitle("y");
auto rp1 = new TRatioPlot(h1);
std::vector<double> lines = {-3, -2, -1, 0, 1, 2, 3};
rp1->SetGridlines(lines);
rp1->Draw();
rp1->GetLowerRefGraph()->SetMinimum(-4);
rp1->GetLowerRefGraph()->SetMaximum(4);
```

Histogram Fitting

```
# Style options
ROOT.gStyle.SetOptStat(0)
ROOT.gStyle.SetOptFit(1)

# Define first fitting function
func = ROOT.TF1(
    "func", "x*gaus(0) + [3]*abs(sin(x)/x)",
    0, 10
)

# Set parameters
func.SetParameters(10, 4, 1, 20)

# Create histogram
h1 = ROOT.TH1F(
    "h1", "Fitting a function",
    100, 0.0, 10.0
)

# Fill histogram with random numbers
h1.FillRandom("func", 10000)

# Draw histogram
h1.Draw()

# Fit histogram with first function
h1.Fit("func")

    # Define second fitting function
    func2 = ROOT.TF1(
        "func2", "x*gaus(0) + expo(3)",
        0, 10
    )

    # Set parameters
    func2.SetParameters(10, 4, 10, -2)

    # Fit histogram with second function
    h1.Fit("func2", "R")
```

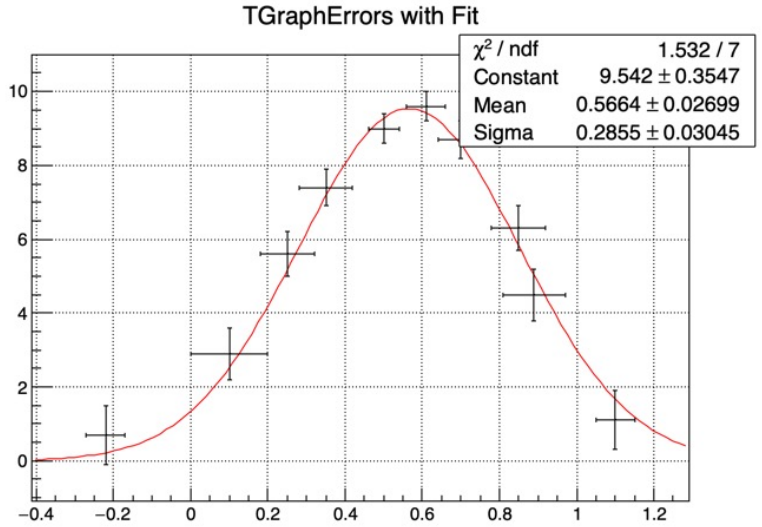




You can also fit an user defined function

Fitting a Graph

```
#
# Draw a graph with error bars and fit a function to it
#
from ROOT import gStyle, TCanvas, TGraphErrors
from array import array
gStyle.SetOptFit (111) # superimpose fit results
c1=TCanvas("c1" ,"Data" ,200 ,10 ,700 ,500) #make nice
c1.SetGrid ()
#define some data points . . .
x = array('f', (-0.22, 0.1, 0.25, 0.35, 0.5, 0.61, 0.7, 0.85, 0.89, 1.1) )
y = array('f', (0.7, 2.9, 5.6, 7.4, 9., 9.6, 8.7, 6.3, 4.5, 1.1) )
ey = array('f', (.8 ,.7 ,.6 ,.5 ,.4 ,.4 ,.5 ,.6 ,.7 ,.8) )
ex = array('f', (.05 ,.1 ,.07 ,.07 ,.04 ,.05 ,.06 ,.07 ,.08 ,.05) )
nPoints=len ( x )
# . . . and hand over to TGraphErrors object
gr=TGraphErrors ( nPoints , x , y , ex , ey )
gr.SetTitle("TGraphErrors with Fit")
gr.Draw ( "AP" ) ;
gr.Fit("gaus")
c1.Update ()
# request user action before ending (and deleting
raw_input('Press <ret> to end -> ')
```



Tree

ROOT uses a columnar storage format designed for efficient compression and random access of large datasets

ROOT Tree (TTree class) is designed to store large quantities of same-class objects, and is optimized to reduce disk space and enhance access speed. It can hold all kind of data, such as objects or arrays, in addition to simple types.

An example for creating a simple Tree

```
void tree_example1() {

  TFile *f = new TFile("myfile.root", "RECREATE");
  TTree *T = new TTree("T","simple tree");
  TRandom r;
  Float_t px,py,pz,pt;
  Double_t random;
  UShort_t i;
  T->Branch("px",&px,"px/F");
  T->Branch("py",&py,"py/F");
  T->Branch("pz",&pz,"pz/F");
  T->Branch("pt",&pt,"pt/F");
  T->Branch("random",&random,"random/D");

  for (i = 0; i < 10000; i++) {
    r.Rannor(px,py);
    pz = r.Gaus(0, 2);
    pt = std::sqrt(px*px + py*py);
    random = r.Rndm();
    T->Fill();
  }

  f->Write();
  f->Close();
}
```

```
import ROOT
from array import array
import math


f = ROOT.TFile.Open("myfile.root", "RECREATE")
T = ROOT.TTree("T","simple tree")
r = ROOT.TRandom()

px = array('f', [ 0 ])
py = array('f', [ 0 ])
pz = array('f', [ 0 ])
pt = array('f', [ 0 ])
random = array('d', [ 0 ])

T.Branch("px",px,"px/F")
T.Branch("py",py,"py/F")
T.Branch("pz",pz,"pz/F")
T.Branch("pt",pt,"pt/F")
T.Branch("random",random,"random/D")

for iEntry in range(10000):
    r.Rannor(px,py)
    pz[0] = r.Gaus(0, 2)
    pt[0] = math.sqrt(px[0] * px[0] + py[0] * py[0])
    random[0] = r.Rndm()
    T.Fill()


f.Write()
f.Close()
```

More on TBranch

- The class for a branch is called TBranch
- A variable in a TBranch is called a leaf (TLeaf)
- If two variables are independent they should be placed on separate branches. However, if they are related it is efficient to put them on same branch
- TTree::Branch() method is used to add a TBranch to TTree
- The branch type is decided by the object stored in it
- A branch can hold an entire object, a list of simple variables, contents of a folder, contents of a TList, or an array of objects.

For example:
tree->Branch("Ev_Branch",&event, "temp/F:ntrack/I:nseg:nvtex:flag/i");

Where “event” is structure with one float, three integers, and one unsigned integer.

Accessing TTree

Show an entry:

```
root [ ] TFile f("myfile.root")
root [ ] T->Show(10)
======> EVENT:10
 px             = 0.680243
 py             = 0.198578
 pt             = 0.708635
 random         = 0.586894
```

Scan a few variables:

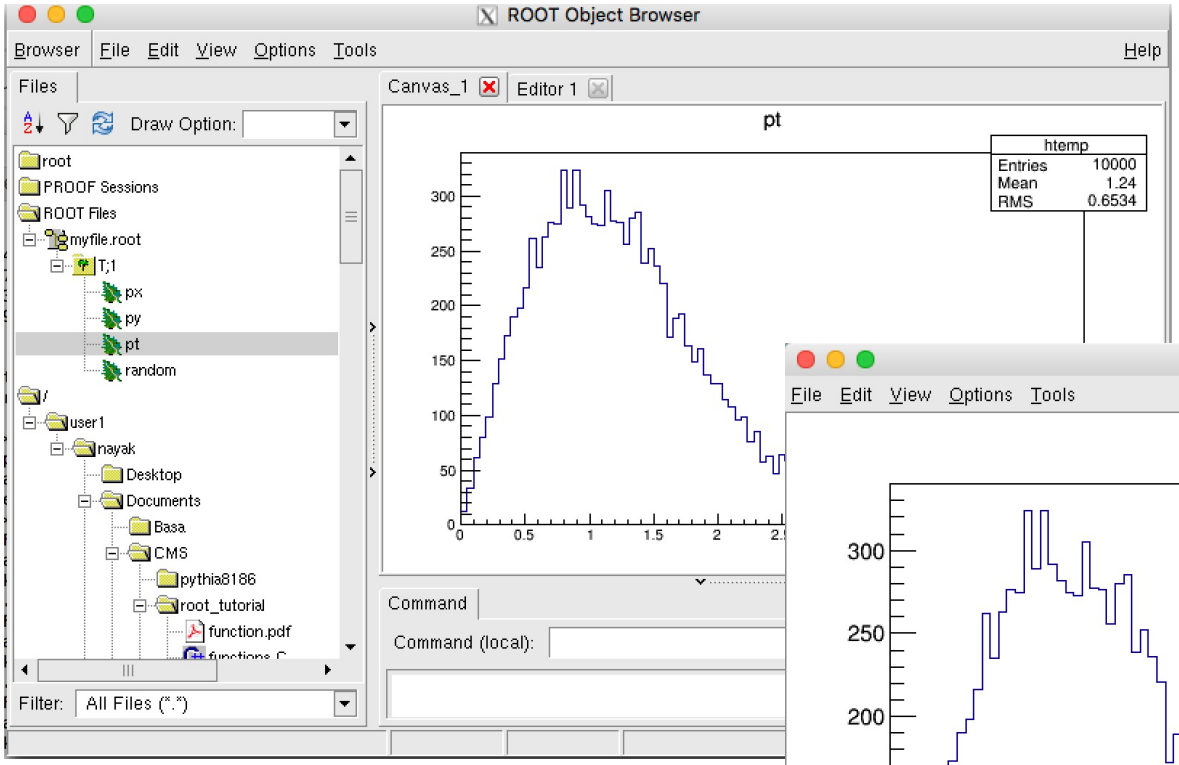
```
root [ ] T->Scan("px:py:pt")
***************************************
*    Row   *        px *        py *        pt *
***************************************
*        0 * 0.8966467 * -1.712815 * 1.9333161 *
*        1 * 1.5702210 * 0.5797516 * 1.6738297 *
*        2 * 0.6975117 * 0.1442547 * 0.7122724 *
*        3 * 0.0616207 * -1.009907 * 1.0117853 *
*        4 * -0.054552 * 1.3832200 * 1.3842953 *
*        5 * -2.017178 * 1.4682819 * 2.4949667 *
*        6 * 0.8903368 * 2.5101616 * 2.6633834 *
*        7 * -1.098390 * -0.318103 * 1.1435256 *
*        8 * 0.3865155 * 0.0235152 * 0.3872301 *
*        9 * 1.8970719 * 1.9546536 * 2.7238855 *
*       10 * 0.6802427 * 0.1985776 * 0.7086347
```

Print the Tree structure:

```
root [ ] T->Print()
******************************************************************
*Tree    :T        : simple tree                                   *
*Entries :    10000 : Total =          202834 bytes  File  Size =     169658 *
*       :          : Tree compression factor =   1.19              *
******************************************************************
*Br    0 :px       : px/F                                          *
*Entries :    10000 : Total  Size=      40594 bytes  File Size  =      37262 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.08 *
*.....                                                             *
*Br    1 :py       : py/F                                          *
*Entries :    10000 : Total  Size=      40594 bytes  File Size  =      37265 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.08 *
*.....                                                             *
*Br    2 :pt       : pt/F                                          *
*Entries :    10000 : Total  Size=      40594 bytes  File Size  =      35874 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.12 *
*.....                                                             *
*Br    3 :random   : random/D                                      *
*Entries :    10000 : Total  Size=      80696 bytes  File Size  =      58600 *
*Baskets :        3 : Basket Size=      32000 bytes  Compression=   1.37 *
*.....                                                             *
```

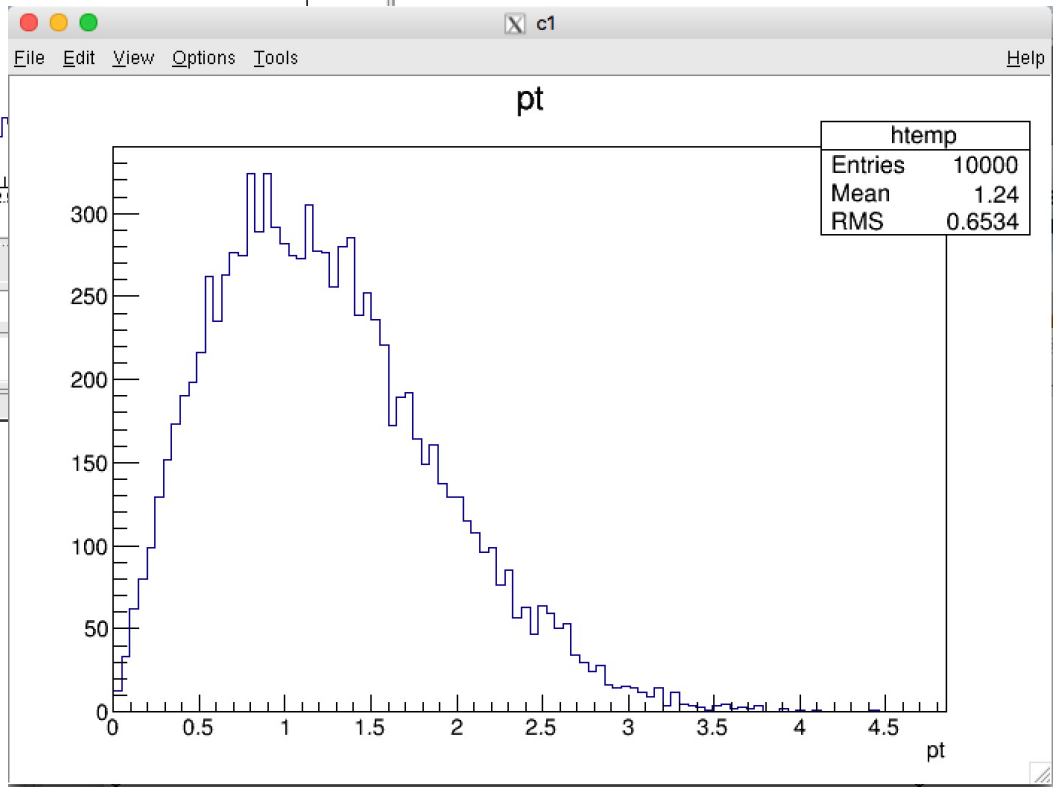
Accessing TTree

Open and Draw branches using TBrowser:



Using TTree::Draw()

T->Draw("pt")



2D plot:

T->Draw("py:px")

Simple Analysis using TTree::Draw()

Applying cuts:
T->Draw(“pt”, “px>0&&py>0”)

For applying “weights” to the distribution:
Selection = “weight * (boolean expression)”
Where “weight” is a event-by-event weight stored as a branch in the ntuple (e.g. cross section * luminosity for the normalization)

Using TCuts:

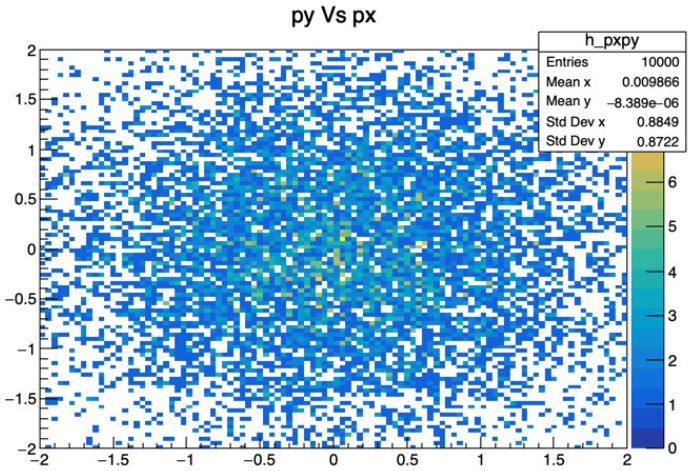
```
TCut c1 = “px > 0”
TCut c2 = “py < 0”
TCut c3 = c1 && c2
T->Draw(“pt”, c3)
T->Draw(“pt”, c1 && c2)
T->Draw(“pt”, c1 && “py > 0”)
TCut c4 = c1 || c2
T->Draw(“pt”, c4)
T->Draw(“pt”, c1 || c2)
```

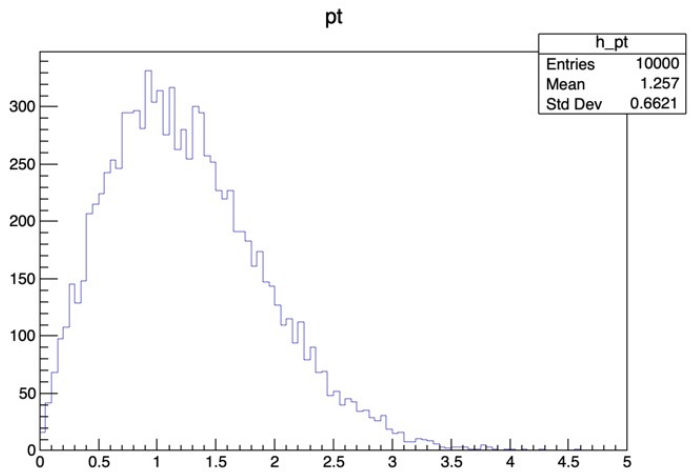
Filling Histograms:

```
TH1F *hist = new TH1F("hist", "", 100, 0., 2.);
T->Draw("pt>>hist");
hist->SetLineColor(2);
hist->Draw("same")
```

A Simple Analyzer

```
import ROOT
import array as array

h_pxpy = ROOT.TH2F("h_pxpy", "py Vs px", 100, -2.0, 2.0, 100, -2.0, 2.0)
h_pt = ROOT.TH1F("h_pt", "pt", 100, 0., 5.0)

f = ROOT.TFile.Open("myfile.root")
t1 = f["T"]

for iEntry in t1:
    h_pxpy.Fill(iEntry.px, iEntry.py)
    h_pt.Fill(iEntry.pt)


c1 = ROOT.TCanvas()
h_pxpy.Draw("colz")
c1.SaveAs("tree_example2_pxpy.png")

c1.Update()
h_pt.Draw()
c1.SaveAs("tree_example2_pt.png")

f.Close()
```





Run the macro:
python3 tree_example2.py

Task: Add histogram attributes (axis title, line color etc..)

Adding Branch with Arrays

Fixed sized array

```
import ROOT
from array import array
import math


f = ROOT.TFile.Open("myfile.root", "RECREATE")
T = ROOT.TTree("T","simple tree")
r = ROOT.TRandom()

px = array('f', [0]*5)
py = array('f', [0]*5)
pz = array('f', [0]*5)
pt = array('f', [0]*5)
random = array('d', [0]*5)

T.Branch("px",px,"px[5]/F")
T.Branch("py",py,"py[5]/F")
T.Branch("pz",pz,"pz[5]/F")
T.Branch("pt",pt,"pt[5]/F")
T.Branch("random",random,"random[5]/D")

for iEntry in range(10000):
    for j in range(5):
        px[j] = r.Gaus()
        py[j] = r.Gaus()
        pz[j] = r.Gaus(0, 2)
        pt[j] = math.sqrt(px[j] * px[j] + py[j] * py[j]
        random[j] = r.Rndm()
    T.Fill()

f.Write()
f.Close()
```

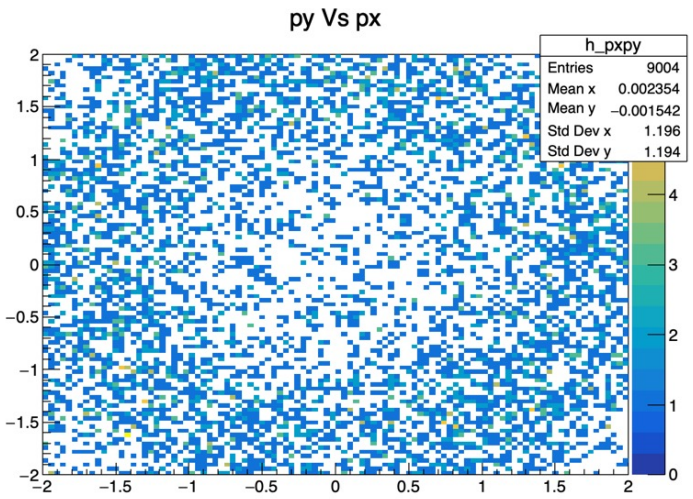
Variable sized array

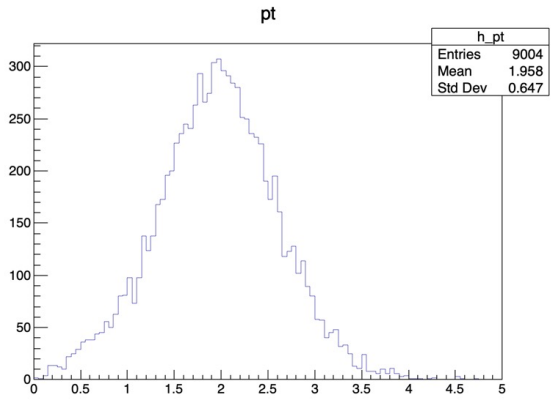
```
import ROOT
from array import array
import math


f = ROOT.TFile.Open("myfile.root", "RECREATE")
T = ROOT.TTree("T","simple tree")
r = ROOT.TRandom()

px = array('f', [0]*10)
py = array('f', [0]*10)
pz = array('f', [0]*10)
pt = array('f', [0]*10)
np = array('i', [0]) #for storing the array size

T.Branch("np",np,"np/I")
T.Branch("px",px,"px[np]/F")
T.Branch("py",py,"py[np]/F")
T.Branch("pz",pz,"pz[np]/F")
T.Branch("pt",pt,"pt[np]/F")

for iEntry in range(10000):
    np[0] = int(r.Rndm()*10)
    for j in range(np[0]):
        px[j] = r.Gaus()
        py[j] = r.Gaus()
        pz[j] = r.Gaus(0, 2)
        pt[j] = math.sqrt(px[j] * px[j] + py[j] * py[j])
    T.Fill()

f.Write()
f.Close()
```

Reading Branch with Arrays

```
# Create histograms
h_pxpy = ROOT.TH2F("h_pxpy", "py Vs px", 100, -2.0, 2.0, 100, -2.0, 2.0)
h_pt = ROOT.TH1F("h_pt", "pt", 100, 0.0, 5.0)

# Open ROOT file
f = ROOT.TFile("myfile.root")

# Get tree
t1 = f.Get("T")

# Define arrays
px = array('f', [0.] * 10)
py = array('f', [0.] * 10)
pz = array('f', [0.] * 10)
pt = array('f', [0.] * 10)

np = array('i', [0])

# Set branch addresses
t1.SetBranchAddress("np", np)
t1.SetBranchAddress("px", px)
t1.SetBranchAddress("py", py)
t1.SetBranchAddress("pz", pz)
t1.SetBranchAddress("pt", pt)
```

```
# Number of entries
nentries = t1.GetEntries()

# Loop over events
for i in range(nentries):
    t1.GetEntry(i)

    # Find object with highest pt
    highest_pt = 0.0
    highest_index = -1

    if np[0] > 0:
        for j in range(np[0]):
            if pt[j] > highest_pt:
                highest_pt = pt[j]
                highest_index = j

        # Fill histograms
        h_pxpy.Fill(
            px[highest_index],
            py[highest_index]
        )

        h_pt.Fill(pt[highest_index])
```





RooFit: Introduction

Purpose

Model distributions of observables $\vec{x}$ in terms of

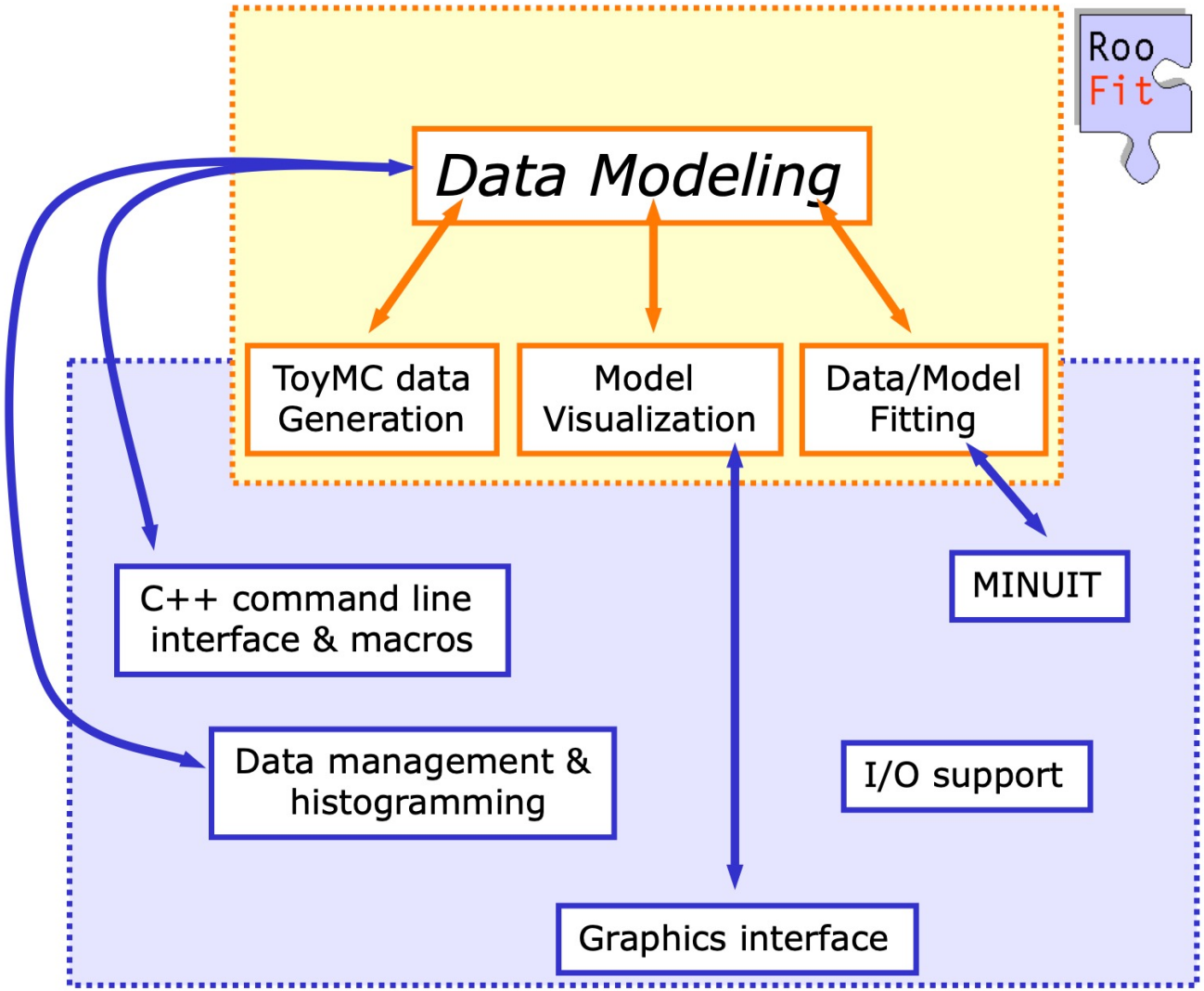
- Physical parameters of interest $\vec{p}$, as well as
- Other (nuisance) parameters $\vec{q}$

Probability density function (PDF) $F(\vec{x}, \vec{p}, \vec{q})$

- Normalized over the allowed range of the observable $\vec{x}$ with respect to the parameters $\vec{p}$ and $\vec{q}$

RooFit: Introduction

An extension to ROOT – Complimentary to existing functionality



ROOT

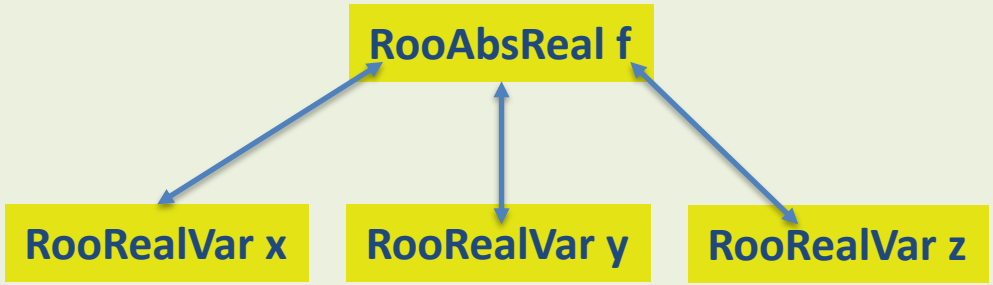
Object-oriented data modeling

Mathematical objects are represented as C++ objects

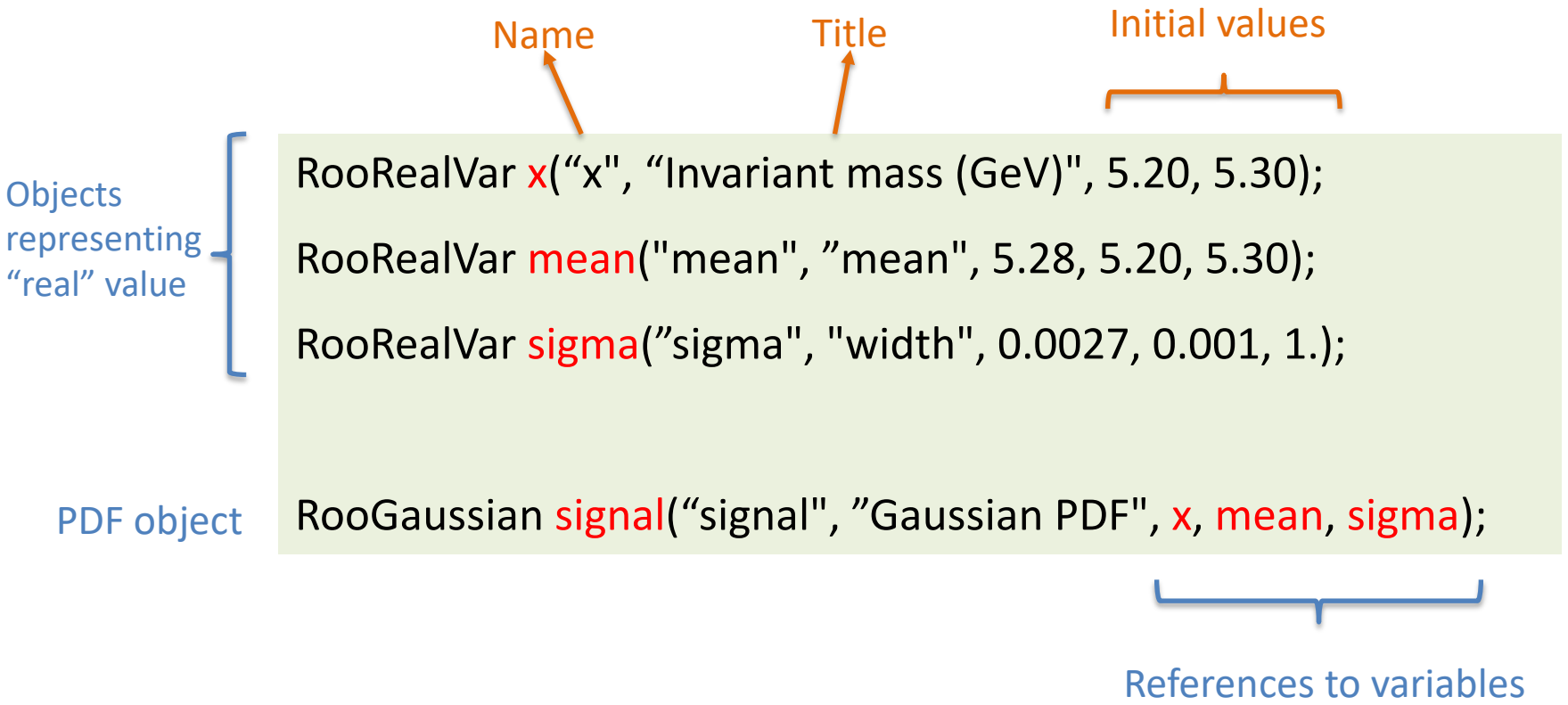
Mathematical concept		RooFit class
variable	x	**RooRealVar**
function	$f(x)$	**RooAbsReal**
PDF	$f(x)$	**RooAbsPdf**
space point	$\vec{x}$	**RooArgSet**
integral	$\int_{x_{\min}}^{x_{\max}} f(x)dx$	**RooRealIntegral**
list of space points		**RooAbsData**

Defining a probability density function

Represent relations between variables and functions as client/server links between objects

Math	$f(x, y, z)$
RooFit Diagram	
RooFit code	**RooRealVar x("x","x",5) ;** **RooRealVar y("y","y",5) ;** **RooRealVar z("z","z",5) ;** **RooFunction f("f","f",x,y,z) ;**

Example – Creating a Gaussian PDF

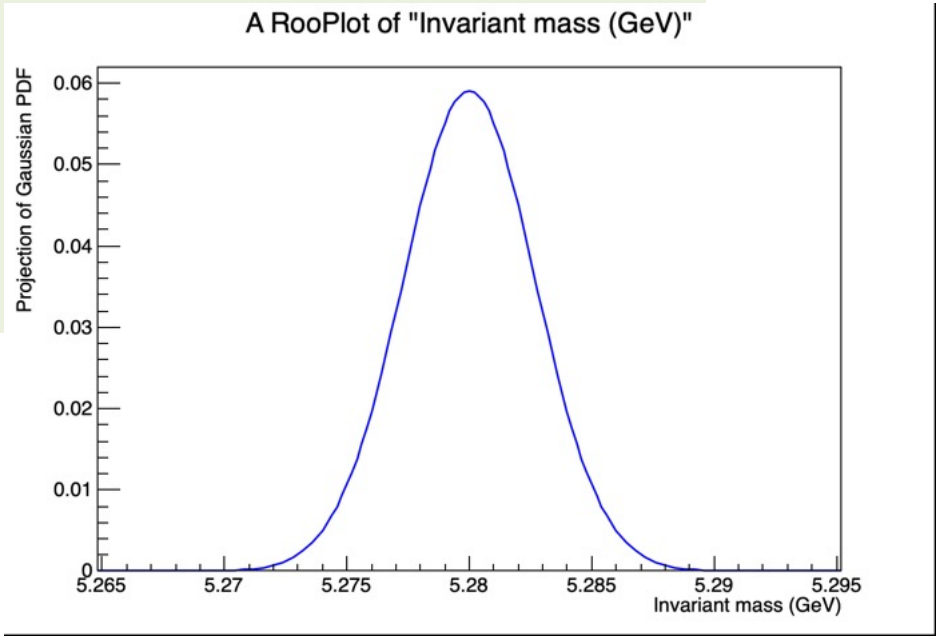


Example – Creating a Gaussian PDF

Creating and plotting Gaussian PDF

```
# Define observable
x = ROOT.RooRealVar("x", "Invariant mass (GeV)", 5.20, 5.30)

# Define Gaussian mean parameter
mean = ROOT.RooRealVar("mean", "mean", 5.28, 5.20, 5.30)

# Define Gaussian width parameter
sigma = ROOT.RooRealVar("sigma", "width", 0.0027, 0.001, 1.0)

# Create Gaussian PDF
signal = ROOT.RooGaussian("signal", "Gaussian PDF", x, mean, sigma)

# Create a RooPlot frame
xframe = x.frame()

# Plot PDF on frame
signal.plotOn(xframe)

# Draw frame
xframe.Draw()
```

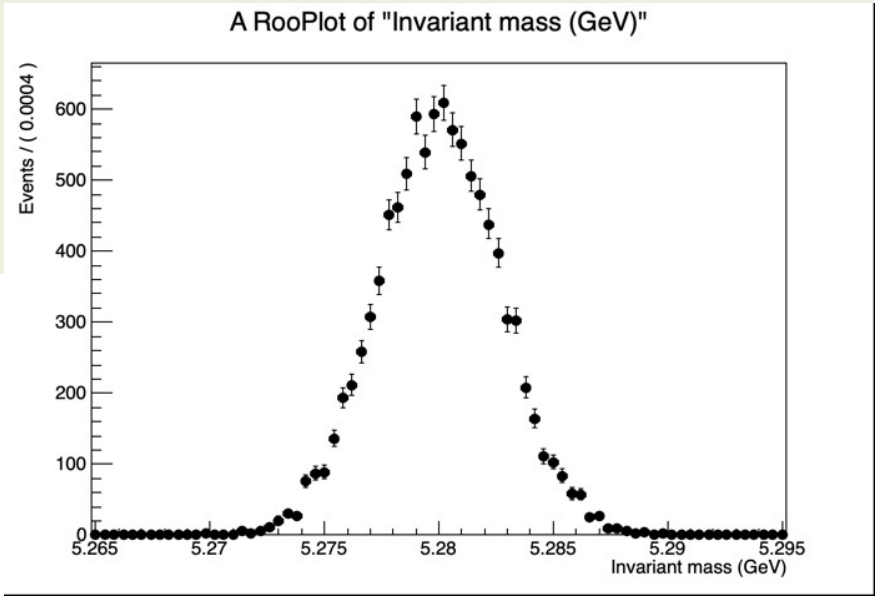
A **`RooPlot`** is an empty frame capable of holding anything plotted versus it variable



Example – generating toy MC

Generate toy MC from PDF and plot the distribution

```
x = ROOT.RooRealVar("x", "Invariant mass (GeV)", 5.20, 5.30)
mean = ROOT.RooRealVar("mean", "mean", 5.28, 5.20, 5.30)
sigma = ROOT.RooRealVar("sigma", "width", 0.0027, 0.001, 1.0)
signal = ROOT.RooGaussian("signal", "Gaussian PDF", x, mean, sigma)

# Generate toy MC events
data = signal.generate(ROOT.RooArgSet(x), 10000)

# Create frame
xframe = x.frame()

# Plot toy dataset
data.plotOn(xframe)

# Draw frame
xframe.Draw()
```



Generated dataset is an unbinned dataset

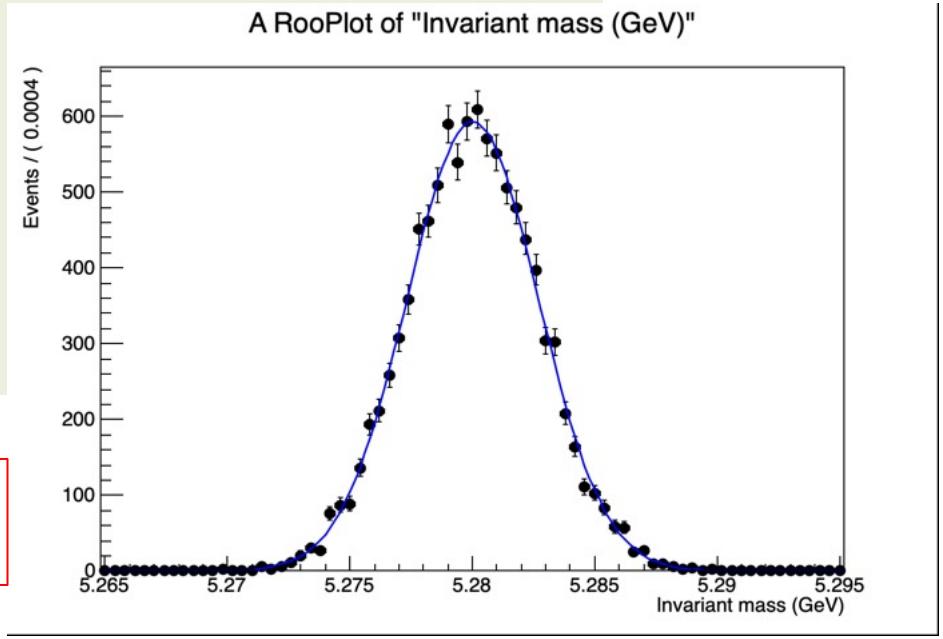
Binning is performed in `data.plotOn()` call

Example – ML fit PDF to unbinned data

Perform Unbinned Maximum Likelihood fit of PDF to data

```
# Generate toy MC events
data = signal.generate(ROOT.RooArgSet(x), 10000)

# Perform maximum likelihood fit
signal.fitTo(data)

# Create frame
xframe = x.frame()

# Plot toy dataset
data.plotOn(xframe)

# Overlay fitted PDF
signal.plotOn(xframe)

# Draw frame
xframe.Draw()
```

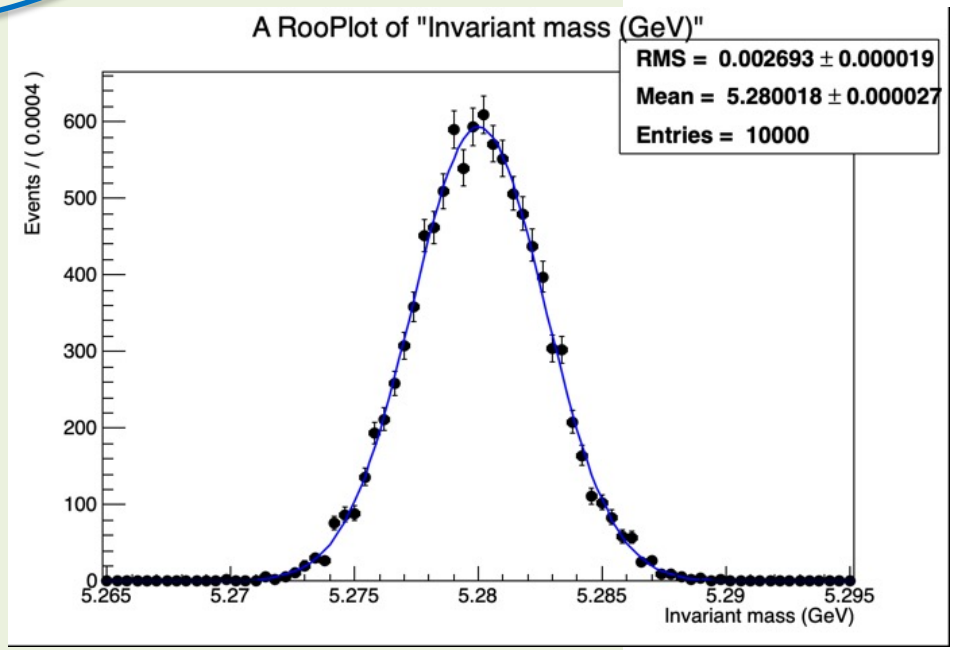
```
mean = 5.28002 +/- 2.69288e-05
sigma = 0.00269288 +/- 1.90418e-05
```



Stat info on RooPlot

Creating and plotting Gaussian PDF

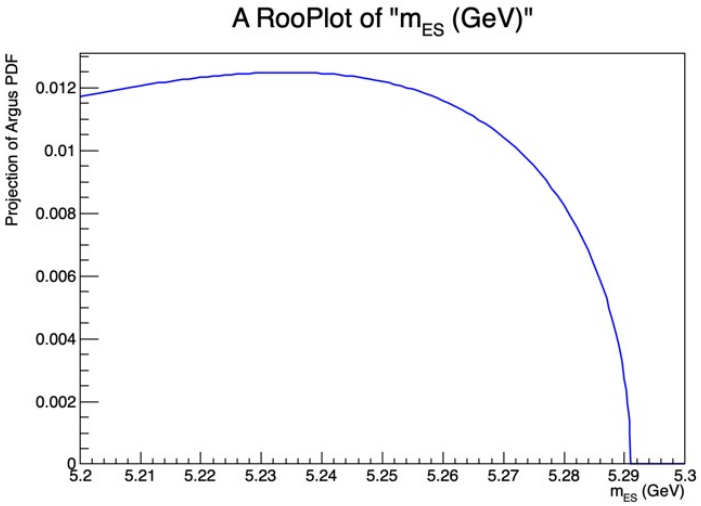
```
# Perform maximum likelihood fit
signal.fitTo(data)

# Print fitted parameter values
mean.Print()
sigma.Print()

# Create frame
xframe = x.frame()

# Plot toy dataset
data.plotOn(xframe)

# Overlay fitted PDF
signal.plotOn(xframe)

#add fit statistics
data.statOn(xframe)

# Draw frame
xframe.Draw()
```




```
mean = 5.28002 +/- 2.69288e-05
sigma = 0.00269288 +/- 1.90418e-05
```

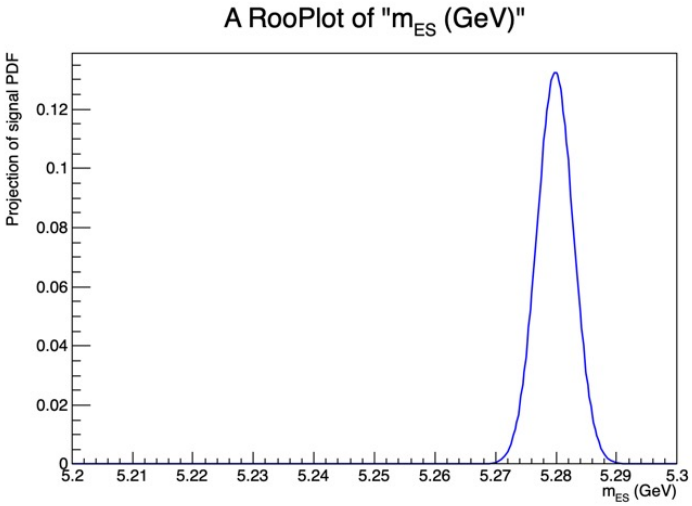
Addition of PDF

Most realistic models can be constructed as the sum of one or more p.d.f.s (e.g. signal and background)

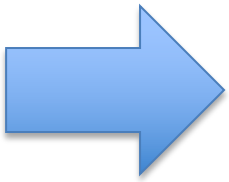


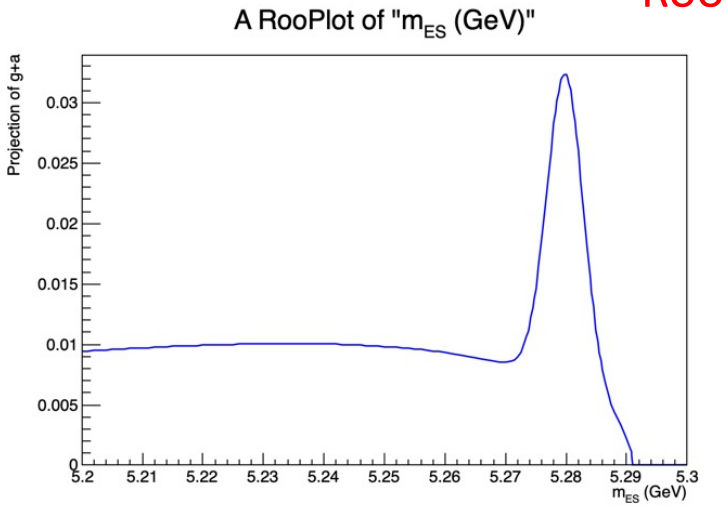
RooGaussian

RooAddPdf



RooArgusBG





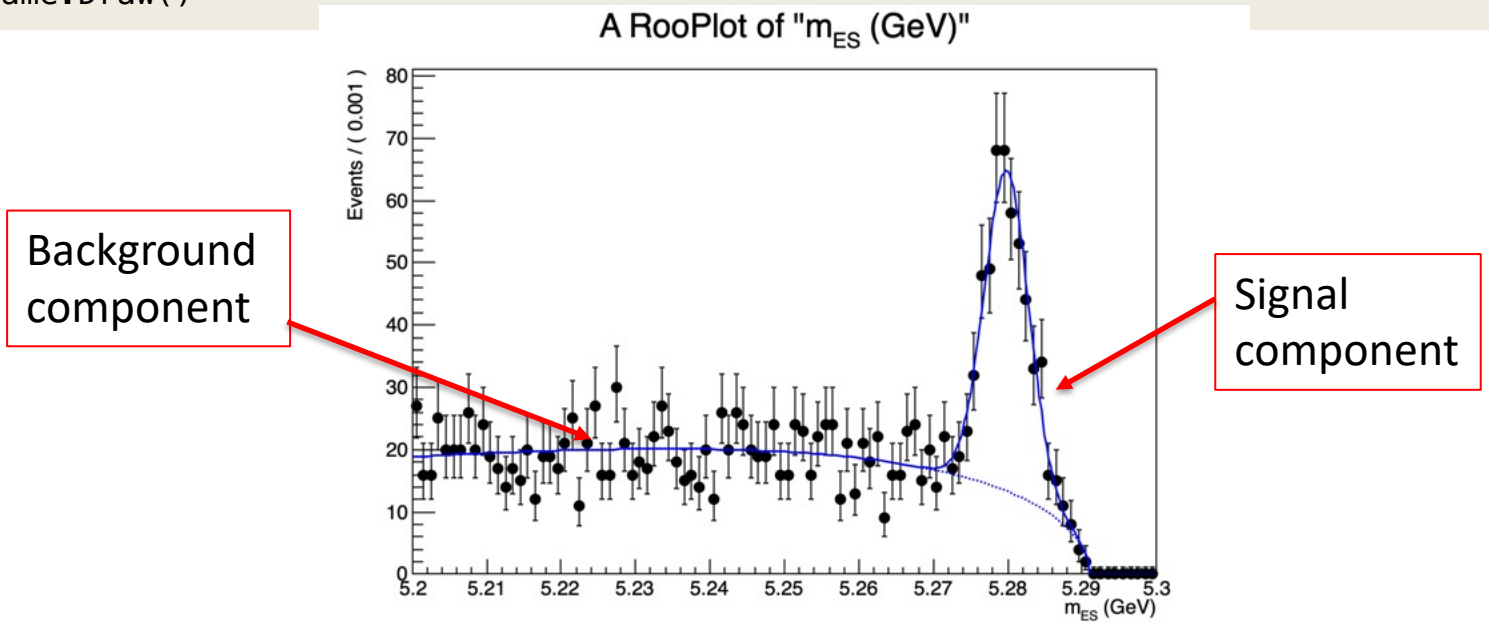
$$S(x) = f.F(x) + (1 - f)G(x)$$

Signal + Background Model

```
    # Use RooFit namespace
    RF = ROOT.RooFit

    # Observable
    mes = ROOT.RooRealVar("mes", "m_{ES} (GeV)", 5.20, 5.30)

    # Signal parameters
    sigmean = ROOT.RooRealVar("sigmean", "B^{#pm} mass", 5.28, 5.20, 5.30)
    sigwidth = ROOT.RooRealVar("sigwidth", "B^{#pm} width", 0.0027, 0.001, 1.0)

    # Gaussian signal PDF
    signalModel = ROOT.RooGaussian("signal", "signal PDF", mes, sigmean, sigwidth)

    # Argus background PDF
    argpar = ROOT.RooRealVar("argpar", "argus shape parameter", -20.0, -100.0, -1.0)
    background = ROOT.RooArgusBG("background", "Argus PDF", mes, ROOT.RooFit.RooConst(5.291), argpar)

    # Construct a signal and background PDF (Composite model)
    nsig = ROOT.RooRealVar("nsig", "#signal events", 200, 0.0, 10000)
    nbkg = ROOT.RooRealVar("nbkg", "#background events", 800, 0.0, 10000)
    model = ROOT.RooAddPdf("model", "g+a", ROOT.RooArgList(signalModel, background),
ROOT.RooArgList(nsig, nbkg))

    # Generate toy Monte Carlo dataset
    data = model.generate(ROOT.RooArgSet(mes), 2000)

    # Perform extended maximum likelihood fit
    model.fitTo(data)

    # Plot toy data and composite PDF overlaid
    mesframe = mes.frame()
    data.plotOn(mesframe)
    model.plotOn(mesframe)
    model.plotOn(mesframe, RF.Components("background"), RF.LineStyle(ROOT.kDashed)
    mesframe.Draw()
```

Signal + Background Model

```
    # Construct a signal and background PDF (Composite model)
    model = ROOT.RooAddPdf("model", "g+a", ROOT.RooArgList(signalModel, background),
ROOT.RooArgList(nsig, nbkg))

    # Generate toy Monte Carlo dataset
    data = model.generate(ROOT.RooArgSet(mes), 2000)

    # Perform extended maximum likelihood fit
    model.fitTo(data)

    # Plot toy data and composite PDF overlaid
    mesframe = mes.frame()
    data.plotOn(mesframe)
    model.plotOn(mesframe)
    model.plotOn(mesframe, RF.Components("background"), RF.LineStyle(ROOT.kDashed)
    mesframe.Draw()
```



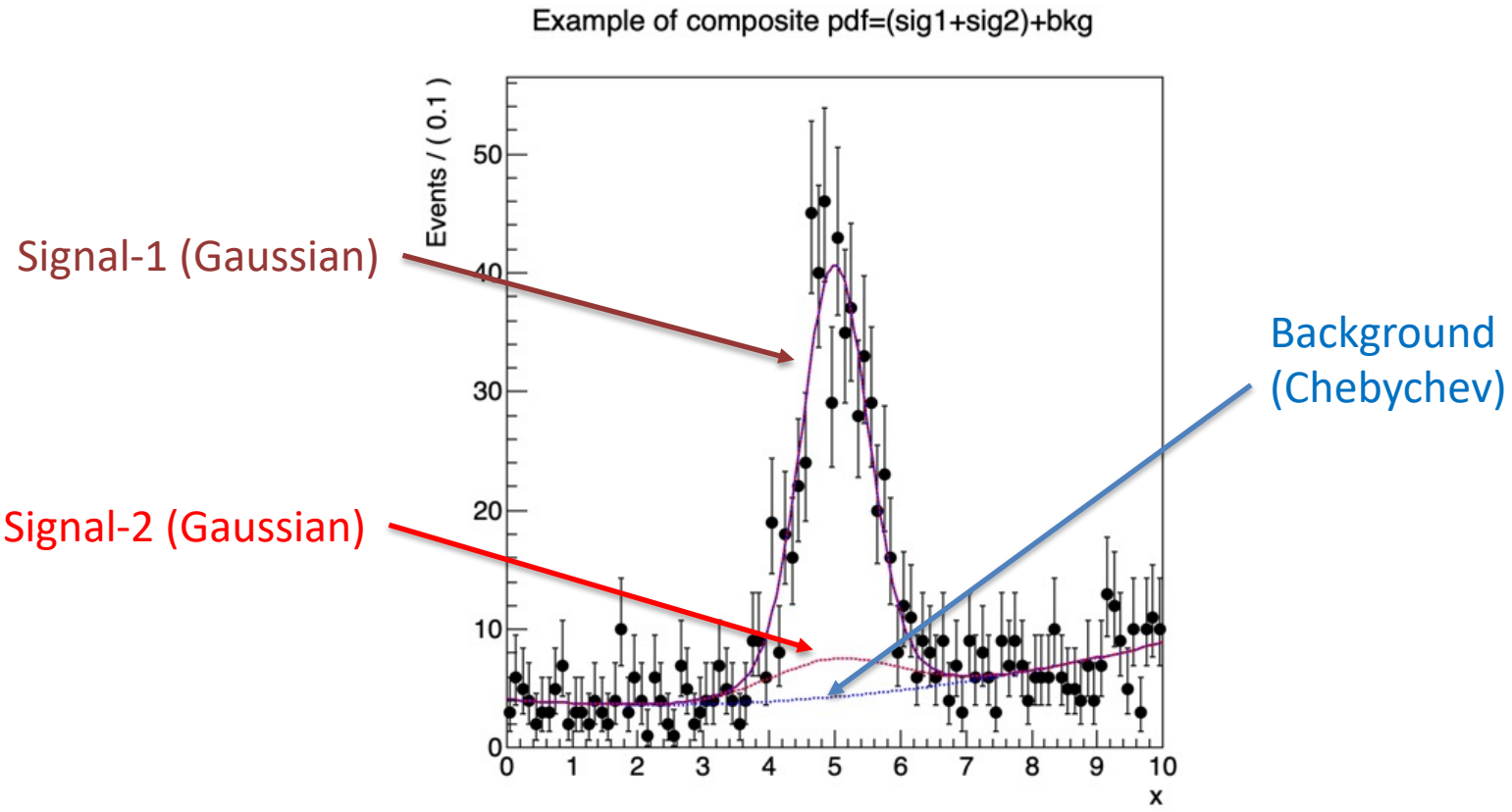
Composite PDF

Adding more than two PDFs, e.g. multiple signal components + background

`See example $ROOTSYS/tutorials/roofit/roofit/rf201_composite.py`

Two signals (Gaussian) + one background (Chebychev)

$$f(x) = f_b B(x) + (1 - f_b)(f_{s1} S_1(x) + (1 - f_{s1}) S_2(x))$$



Composite PDF

Adding more than two PDFs, e.g. multiple signal components + background

See example
$ROOTSYS/tutorials/roofit//roofit/rf201_composite.py

```
// Declare observable x
x = ROOT.RooRealVar("x", "x", 0,

// Create two Gaussian PDFs g1(x,mean1,sigma) anf g2(x,mean2,sigma) and their parameters
mean = ROOT.RooRealVar("mean", "mean of gaussians", 5)
sigma1 = ROOT.RooRealVar("sigma1", "width of gaussians", 0.5)
sigma2 = ROOT.RooRealVar("sigma2", "width of gaussians", 1)

sig1 = ROOT.RooGaussian("sig1", "Signal component 1", x, mean, sigma1)
sig2 = ROOT.RooGaussian("sig2", "Signal component 2", x, mean, sigma2)

// Build Chebychev polynomial pdf
a0 = ROOT.RooRealVar("a0", "a0", 0.5, 0.0, 1.0)
a1 = ROOT.RooRealVar("a1", "a1", -0.2, 0.0, 1.0)
bkg = ROOT.RooChebychev("bkg", "Background", x, [a0, a1])

// Sum the signal components into a composite signal pdf
sig1frac = ROOT.RooRealVar("sig1frac", "fraction of component 1 in signal", 0.8, 0.0, 1.0)
sig = ROOT.RooAddPdf("sig", "Signal", [sig1, sig2], [sig1frac])

// Sum the composite signal and background
bkgfrac = ROOT.RooRealVar("bkgfrac", "fraction of background", 0.5, 0.0, 1.0)

model = ROOT.RooAddPdf("model", "g1+g2+a", [bkg, sig], [bkgfrac])
// OR
// Construct sum of models on one go using recursive fraction
interpretations
//   model2 = bkg + (sig1 + sig2)
model2 = ROOT.RooAddPdf("model", "g1+g2+a", [bkg, sig1, sig2], [bkgfrac, sig1frac], True)
```