Table of Contents

ROOT Tutorials Exercises for La Plata (26-30 November 2013)....1
Topics....1
Material for the course....1
Start using ROOT (slides)....1
Setting up ROOT....1
Start using the ROOT prompt....2
Exercise 1: Plotting a Function in ROOT....2
Exercise 2: Making an histogram in ROOT....4
Exercise 3: Plotting Points in ROOT (TGraph class)....5
Working with Histograms (slides)....6
Exercise 4: Working with the histogram bins....6
Exercise 5: Histograms Operations....7
Exercise 6: Multi-Dimensional Histograms and Profiles....9
ROOT I/O and Trees (slides)....10
Exercise 7: Writing and Reading Histogram from a file....10
Exercise 8: Creating a ROOT Tree containing a collection of LorentzVector's....11
Exercise 8b: Creating a flat ROOT Tree....14
Exercise 9: Creating a ROOT Tree containing an EventData object....15
Exercise 10: Create and query ROOT Tree from a data text file....19
Exercise 10b: Read a ROOT Tree containing a collection of LorentzVector's....20
Exercise 11: Analyzing a ROOT Tree using TTree::MakeClass....22
Exercise 11b: Analyze a ROOT Tree using the new TTreeReader (Available only on ROOT 5.99)....26
Exercise 12: Using the TSelector class for analysing a TTree....27
Exercise 13: Chaining ROOT Files....31
Interactive Data Analysis with PROOF....31
Exercise 14a: Run....31
Exercise 14b: Run....31
Exercise 14c: Using PROOF to analyze the TTree....32
Exercise 14d: Using PROOF to generate events with Pythia8....33
Fitting in ROOT (slides)....33
Exercise 15: Gaussian fit of an histogram....34
Exercise 16: Fit a peak histogram....35
Exercise 17: Using the Fit Panel GUI....36
Fitting using RooFit (slides)....37
Exercise 18: Gaussian model and fit it to random generated data....37
Exercise 19: Reading a workspace from a file....39
Exercise 20: Fit of a Signal over an Exponential Background....40
Exercise 21: Compute the significance of the signal peak using RooStats....42
Exercise 22: Compute significance (p-value) as function of the signal mass....44
Instructions for the Virtual Machine....45
Accounts....45
Setting up CVMFS....45
Setting up ROOT....45
Opening ports for PROOF....45

ROOT Tutorials Exercises for La Plata (26-30 November 2013

Topics

This tutorial aims to provide a solid base to efficiently analyse data with ROOT. The main features of ROOT are presented: histogramming, data analysis using trees and advanced fitting techniques.

Material for the course

The slides of the lectures are available in electronic form. See each section for the corresponding links. Slides introducing the functionality of ROOT are also available (slides)

As complementary material, one can look at

- ROOT Primer Guide, available in pdf, html or epub format. This introductory guide illustrates the main features of ROOT, relevant for the typical problems of data analysis: input and plotting of data from measurements and fitting of analytical functions.
- ROOT user guide. It can be downloaded in various format (or only individual chatters) from here.
- RooFit User Guide, available in pdf format. A coincide RooFit quick start guide is also available here.
- A tar file with all the exercise solutions (all the ROOT macro required) is available here
- A special section introducing C++ (Exercises , (slides)

Start using ROOT (slides)

We will focus first on introductory exercises for getting started working with the ROOT environment and writing our first ROOT macro. Two levels of help will be provided: a hint and a full solution. Try not to jump to the solution even if you experience some frustration. The help is organised as follow:

Hint Hide
Here the hint is shown.

Solution Hide
Here the solution is shown.

Some points linked to the exercises are optional and marked with a icon: they are useful to scrutinise in more detail some aspects. Try to solve them if you have the time.

Setting up ROOT

ROOT is installed in the virtual machines prepared for the course. It should be automatically available from the Terminal. If not, one should run the fallowing lines (or put in the default login shell script):

```
export ROOTSYS=/usr/local/ROOT
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
. $ROOTSYS/bin/thisroot.sh
```

The version of ROOT available in the virtual machines which should be 5.34.12.

Start using the ROOT prompt

First of all open a terminal window on your computer and type

```
root
```

If this does not work, then it means ROOT is not properly installed in your system. Stop here and ask for help to fix the installation. If it works then start playing with the prompt. Use as a calculator and type:

```
root [0] 2+2
```

or whatever you like (see for example the lecture slide) or page 5 of the booklet ("A ROOT Guide for Beginners"). Note that after having issue a statement from the ROOT prompt, you can omit the `;` required by C++. This will make ROOT printing the returned value, if there is one. For example:

```
root [0] TMath::Pi();
```

will not print anything, while

```
root [0] TMath::Pi()
(Double_t)3.14159265358979312e+00
```

Afterwards start writing your first ROOT macro. A ROOT macro is a file containing some C++ code which can be run from the ROOT prompt. You need to define a function and in the function scope you write the code. For example, you create a file, which you will call `mymacro.C`. Inside the file you define a function `mymacro(int value)` and you write the code of slide 8 of the lecture (or something else, if you prefer). The you can run the macro from the ROOT prompt by typing:

```
root [0] .x mymacro.C(42)
```

Note that if the function name is different than the macro (file) name, you need two steps to run. First you load the macro:

```
root [0] .L mymacro.C
```

Then you run the function in the macro. Let's assume the function name is `test(int value)`:

```
root [1] test(42)
```

After having followed this introduction, you can try to move to the first exercise.

Exercise 1: Plotting a Function in ROOT

Following the lecture slide or page 6 of the Introductory Guide, create a TF1 class using the `sin(x)/x` function and draw it.

Create then a function with parameters, `p0 * sin ( p1 * x) /x` and also draw it for different parameter values. You can try to change the parameter values using `TF1::SetParameters` or by using the ROOT GUI. Try also to change the style of the line and its color. You can either use the ROOT GUI and/or use the methods of the class `TAttLine` (see `TAttLine` reference documentation). Since TF1 derives from `TAttLine`, it inherits all its functions. Try for example to set the colour of the parametric function to blue.

After having drawn the function, compute for the parameter values (p0=1, p1=2):

- function value for x = 1.

- function derivative for x = 1
- integral of the function between 0 and 3.

Hint Hide
To solve the exercise you need to :

- create a TF1 object using a formula expression. In the case of a parametric functions, the two parameters are defined as [0] and [1]
- call `TF1::Draw()`
- call `TF1::Eval(x)`, `TF1::Derivative(x)` and `TF1::Integral(a,b)`

You can also find the available member functions of the class TF1, by using the Tab key on the ROOT prompt, for example

```
root [0] TF1 * f1 = ....
root [1] f1-><TAB>
```

And you can get the full signature of a method by doing for example:

```
root [1] f1->Derivative(<TAB>
```

Solution Hide

```
#include "TF1.h"

void plotFunction() {

   TF1 * f1 = new TF1("f1","sin(x)/x",0,10);
   f1->Draw();


   TF1 * fp = new TF1("fp","[0]*sin([1]*x)/x",0,10);
   fp->SetParameters(1,2);
   fp->Draw("same");
   fp->SetLineColor(kBlue);

   // to change axis y margins
   // (or invert the order of plotting the functions)
   f1->SetMaximum(2);
   f1->SetMinimum(-2);

   std::cout << "Value of f(x) at x = 1 is       " << fp->Eval(1.) << std::endl;
   std::cout << "Derivative of f(x) at x = 1 is  " << fp->Derivative(1.) << std::endl;
   std::cout << "Integral of f(x) in [0,3] is    " << fp->Integral(0,4)  << std::endl;

}
```

You can try to use a different function, for example the Gamma distribution, defined in `ROOT::Math::gamma_pdf`. Try to make a plot as the one in Wikipedia, see here, where the function is plot for different parameter values. See the here the reference documentation of the gamma distribution in ROOT.

Solution Hide

```
#include "TF1.h"
#include "Math/DistFunc.h"


void plotGamma() {

   // Note that parameter [0] is called alpha in definition of gamma_pdf or kappa in Wikipedia
   // Note that parameter [1] is  theta
```

```
   // use range [0,20] as in Wikipedia plot
   TF1 * f1 = new TF1("f","ROOT::Math::gamma_pdf(x,[0],[1])",0,20);

   f1->SetLineColor(kRed);
   f1->SetParameter(0,1);
   f1->SetParameter(1,2);

   // use DrawClone because we will plot many different copies of same object but with different
   // parameter values
   f1->DrawClone();


   // now change parameters and draw at different parameter values
   f1->SetLineColor(kGreen);
   f1->SetParameter(0,2);
   f1->DrawClone("SAME");

   f1->SetLineColor(kBlue);
   f1->SetParameter(0,3);
   f1->DrawClone("SAME");

   f1->SetLineColor(kCyan);
   f1->SetParameter(0,5);
   f1->SetParameter(1,1);
   f1->DrawClone("SAME");


   f1->SetLineColor(kOrange);
   f1->SetParameter(0,9);
   f1->SetParameter(1,0.5);
   f1->DrawClone("SAME");


}
```

Exercise 2: Making an histogram in ROOT

We will learn in this exercise how to create a one-dimensional histogram in ROOT, how to fill it with data and how to plot it.

Create a one-dimensional histogram with 50 bins between 0 10 and fill it with 10000 gaussian distributed random numbers with mean 5 and sigma 2. Plot the histogram and, looking at the documentation in the THistPainter, show in the statistic box the number of entries, the mean, the RMS, the integral of the histogram, the number of underflows, the number of overflows, the skewness and the kurtosis.

Hint Hide
For generating gaussian random numbers use `gRandom->Gaus(mean, sigma)`

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TStyle.h"

void plotHistogram() {

   TH1D * h1 = new TH1D("h1","h1",50,0,10);

   for (int i = 0; i < 10000; ++i) {
      double x = gRandom->Gaus(5,2);
      h1->Fill(x);
   }

   h1->Draw();
```

```
   gStyle->SetOptStat(111111110);

}
```

After calling the function `TH1::ResetStats()` , you will see that the statistics (mean, RMS,..) of the histogram is slightly different. Try to understand the reason for this, by trying for example to compute the mean of the histogram yourself.

Solution Hide
The initial statistics is computed using the original (un-binned) data, while after calling `TH1::ResetStats()`, the statistics is computed using the bin contents and centres (binned data).

The following macro, creating, filling and plotting histogram contains an error, which one ?

```
#include "TH1.h"

void testPlotHistogram() {

   TH1D h1("h1","h1",50,-5,5);
   h1.FillRandom("gaus",10000);
   h1.Draw();

}
```

Solution Hide
The histogram objected is deleted at the end of the macro, therefore it is not shown in the plot after exiting the macro. To fix it, either call `TH1::DrawClone` or create the histogram using operator new as in the previous example.

Exercise 3: Plotting Points in ROOT (TGraph class)

We will learn in this exercise how to plot a set of points in ROOT using the TGraph class.

Suppose you have this set of points defined in the attached file graphdata.txt

Plot these points using the TGraph class. Use as a marker point a black box. Looking at the possible options for drawing the TGraph in TGraphPainter, plot a line connecting the points.

Hint Hide
To solve the exercise you need to :

- create the Graph using the constructor where you can specify the text file name containing the points.

Otherwise you can also

- create an array for the X points: `double x[] = {1,2...`
- create an array for the y points
- create the TGraph from the X and Y arrays

You can also create first an empty TGraph object and set the point one by one by calling `TGraph::SetPoint`

Solution Hide

```
void plotGraph() {

   TGraph * g = new TGraph("graphdata.txt");
```

```
   g->Draw("AP");
   g->SetMarkerStyle(21);

}
```

Make a TGraphError and display it by using the attached data set, graphdata_error.txt, containing error in `x` and `y`.

Solution Hide

```
void plotGraphError() {

   TGraphErrors * g = new TGraphErrors("graphdata_error.txt");

   g->Draw("AP");
   g->SetMarkerStyle(21);

}
```

Working with Histograms (slides)

We will focus in the next exercises on the histograms and their operations

Exercise 4: Working with the histogram bins

Create an histogram with 200 bins between 0 and 10. Fill then the histogram with 1000 gaussian random numbers with mean 5 and sigma 0.3 and 10000 uniform number between 0 and 10. Plot this histogram. Find the bin number of the histogram with the maximum content. What is its bin content ? What is the bin center and the bin error ? Afterwards zoom the histogram plot between 4 and 6 using either `TAxis::SetRange` or the mouse (clicking on the axis).

Hint Hide
To find the maxim bin you can use `TH1::GetMaximumBin` if you don't want to loop at the bins yourself.

For zooming an histogram you can use `TAxis::SetRange(firstbin,last bin)` or `TAxis::SetRangeUser(x1,x2)`. You can also do it with the GUI by selecting the axis range with the mouse.

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TPad.h"
#include <iostream>

void exerciseHistogram1() {


   TH1D * h1 = new TH1D("h1","flat+gaus histogram",200,0,10);

   for (int i = 0; i < 10000; ++i) {         h1->Fill( gRandom->Uniform(0,10) );
   }
   for (int i = 0; i < 1000; ++i) {         h1->Fill( gRandom->Gaus(5,0.2) );
   }

   h1->Draw();

   double ibinMax = h1->GetMaximumBin();
   std::cout << "Bin with maximum is " << ibinMax << " at x = " << h1->GetBinCenter(ibinMax) << s
   axis->SetRange( axis->FindBin(4.0001), axis->FindBin(5.9999) ); // add or subtract a small eps
   // axis->SetRangeUser(4.,6. );   // alternativly you can use SetRangeUser

   gPad->Update();
```

```
}
```

Instead of zooming create a sub-histogram between 4 and 6 using `TH1::GetBinContent` on the original histogram and `TH1::SetBinContent` on the new histogram. How many bins has the new histogram ?

Hint Hide
To create the sub-histogram, you need first to find the bin numbers in the original histogram corresponding to 4 and 6. You can use for this `TAxis:FindBin` for this. Then you create the sub-histogram using as lower value for the axis, the lower edge of the bin corresponding to 4, and, as upper value, the upper edge of the bin corresponding to 6. Afterwards, by looping on the bins, you can copy the bin content from the original histogram into the new one. It is better to use a slightly value larger than 4 (e.g. 4.0001) and a value a little bit smaller than 6 (e.g. 5.999) to avoid the bin edges.

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TPad.h"
#include <iostream>


void exerciseHistogram1b() {


   TH1D * h1 = new TH1D("h1","flat+gaus histogram",200,0,10);

   for (int i = 0; i < 10000; ++i) {         h1->Fill( gRandom->Uniform(0,10) );
   }
   for (int i = 0; i < 1000; ++i) {         h1->Fill( gRandom->Gaus(5,0.2) );
   }

   h1->Draw();

   double ibinMax = h1->GetMaximumBin();
   std::cout << "Bin with maximum is " << ibinMax << " at x = " << h1->GetBinCenter(ibinMax) << s
   int ifirst =  axis->FindBin(4.001);
   int ilast =  axis->FindBin(5.9999);
   int nbins =  ilast - ifirst;
   TH1D * h2 = new TH1D("h2","flat+gaus zoomed histogram",nbins, axis->GetBinLowEdge(ifirst), axi
   std::cout << " Number of bins of zoomed histogram is " << nbins << std::endl;    for (int i1 =
   }
   h2->Draw();
   gPad->Update();

   // to check if we did not screw-up,  we recompute maximum position
   double ibinMax2 = h2->GetMaximumBin();
   std::cout << "Bin with maximum is " << ibinMax2 << " at x = " << h2->GetBinCenter(ibinMax2) <<

}
```

Exercise 5: Histograms Operations

We will work in this exercise on the histogram operations like addition, subtraction and scaling.

- Make a gaussian filled histogram between 0 and 10 with 100 bis and 1000 entries with mean 5 and sigma 1.
- Make another histogram uniformly distributed between 0 and 10 with 100 bins and 10000 entries.
- Add the two histogram into a new one using `TH1::Add`
- Make another uniformly distributed histogram, still with 100 bins but with 100000 entries. Normalize this histogram to have a total integral of 10000 using `TH1::Scale`.

- Subtract now from the histogram which contains the sum of the flat and the gaussian histograms, the flat normalised histogram, using `TH1::Add`
- Plot the result using the error option (`h1->Draw("E")`). Do the error make sense ? If not, how can you get the correct bin errors ?

Hint Hide

- Use `TH1::Add` to add the two histogram.
- Use `TH1::Scale(10000/ 100000)` to re-normalise the histogram.
- Use again `TH1::Add` to subtract the histogram, but with a second coefficient equal to -1. (`TH1::Add(h1,h2,1,-1)`).
- For getting the right errors you must call `TH1::Sumw2` before doing the operations on the histograms (i.e. before scaling and before subtracting them)

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TCanvas.h"
#include "TLine.h"
#include <iostream>

void HistoOperations() {


   TH1D * h1 = new TH1D("h1","flat histogram",100,0,10);
   for (int i = 0; i < 10000; ++i) {         h1->Fill( gRandom->Uniform(0,10) );
   }

   TH1D * h2 = new TH1D("h2","gaus histogram",100,0,10);
   for (int i = 0; i < 1000; ++i) {         h2->Fill( gRandom->Gaus(5,1) );
   }

   TH1D * h3 = new TH1D("h3","flat+gaus histogram",100,0,10);
   h3->Add(h1,h2);

   h3->Draw();


   TH1D * h4 = new TH1D("h4","second flat histogram",100,0,10);
   for (int i = 0; i < 100000; ++i) {         h4->Fill( gRandom->Uniform(0,10) );
   }

   // renormalize histogram
   //!! VERY IMPORTANT - NEED TO CALL Sumw2() to get right ERRORS!!!
   h4->Sumw2();
   h3->Sumw2();
   h4->Scale(0.1);

   // plot in a new Canvas
   new TCanvas("c2","c2");


   TH1D * h5 = new TH1D("h5","(flat+gaus) histogram - flat histogram",100,0,10);
   h5->Add(h3,h4,1.,-1.);
   h5->Draw("E");

   TLine * line = new TLine(0,0,10,0);
   line->Draw();


}
```

Rebin now of the histogram (e.g. the one resulting at the end from the subtraction) in a new histogram with bins 4 times larger.

Hint Hide
For rebinning in 4 times larger bins, use `TH1::Rebin` with `ngroup=4`.

Solution Hide
Add these lines of code at the end of the macro

```
new TCanvas("c2","c2");

  TH1 * h6 = h5->Rebin(4,"h6");
  h6->SetTitle("Rebinned histogram");
  h6->Draw();
```

Exercise 6: Multi-Dimensional Histograms and Profiles

Create a 2 dimensional histogram with x and y in the range [-5,5] and [-5,5] and 40 bins in each axis. Fill the histogram with correlated random normal numbers. To do this generate 2 random normal numbers (mean=0, sigma=1) u and w. Then use `x = u` and `y=w+0.5*u` for filling the histogram. Plot the histogram using color boxes (See documentation in THistPainter class) or choose what ever option you prefer. After having filled the histogram, compute the correlation using `TH1::GetCorrelationFactor`.

Hint Hide
The option for plotting colour boxes is "COLZ", which draws also a palette for the scale on the Z axis (the bin content)

Solution Hide

```
#include "TH2.h"
#include "TProfile.h"
#include "TRandom.h"
#include "TCanvas.h"
#include <iostream>

void Histogram2D() {


   TH2D * h2d = new TH2D("h2d","2d histogram",40,-5,5, 40, -5, 5);
   for (int i = 0; i < 100000; ++i) {        double u =  gRandom->Gaus(0,1);
      double w =  gRandom->Gaus(0,1);
      double x = u;
      double y = w + 0.5 * u;
      h2d->Fill( x,y );
   }

   h2d->Draw("COLZ");

   std::cout << "correlation factor " << h2d->GetCorrelationFactor() << std::endl;


}
```

Make then a projection of the 2-dimensional histogram on the x. Make also a projection of the y axis into a profile. Plot the resulting projected histograms in a new canvas separated in 2 pads.

Hint Hide
For making the projection call `TH1::ProjectionX` and for making the profile call `TH1::ProfileX`

For dividing the canvas call `TCanvas::Divide(1,2)` and navigate in the pad contained in the canvas by calling `TCanvas::cd(pad_number)`.

Solution Hide
Add these lines of code at the end of the macro

```
TH1 * hx = h2d->ProjectionX();
   TH1 * px = h2d->ProfileX();

   TCanvas * c2 = new TCanvas("c2","c2");
   // divide in 2 pad in x and one in y
   c2->Divide(2,1);
   c2->cd(1);
   hx->Draw();

   c2->cd(2);
   px->Draw();
```

Look at the bin error of the profile. Do you know how it is computed ? You can find this answer in the TProfile reference documentation.

If you have still time, after having finished the exercises you can look at some of the tutorials in the ROOT distribution directory, $ROOTSYS/tutorials/hist. They are available on the Web at this location. For example look at:

- hlabels1.C
- hstack.C
- rebin.C
- sparsehist.C

ROOT I/O and Trees (slides)

Set of Exercises working with the Trees in ROOT. First will start with an exercise on the I/O of ROOT by storing and reading an histogram from a file. Then we will move to exercises using the `TTree` class. The first one is very simple and it could be skipped by somebody already knowledgeable of ROOT.

Exercise 7: Writing and Reading Histogram from a file

Open a file then create a simple histogram, for example an histogram generated with exponential distribution. Fit it and write it in a file. Why the ROOT Canvas does not show the histogram ? Do you know what to do to have the histogram displayed ?

Hint Hide
Use `TFile::Open` to open the file or just create a `TFile object`. Call `TH1::Write` to write the histogram in the file after having filled it.

Solution Hide

```
#include "TFile.h"
#include "TH1.h"
#include "TRandom.h"

void histogramWrite() {

   TFile f("histogram.root","RECREATE");

   TH1D * h1 = new TH1D("h1","h1",100,0,10);
   for (int i = 0; i < 10000; ++i)
      h1->Fill(gRandom->Exp(5) );
```

```
   h1->Fit("expo");
   h1->Draw();

   f.Write("h1");
   f.Close();
}
```

The histogram is not shown, becausem when the file is close, it is automatically deleted.

Now read the histogram from the file and plot it.

Hint Hide
Create a file object (or call `TFile::Open`) and then `TFile::Get`

Solution Hide

```
void histogramRead() {

   TFile * file = new TFile("histogram.root");

   TH1 * h1 = 0;
   file->GetObject("h1",h1);
   // you can also use nut you need to cast if you compile the code
   //TH1 * h1 = (TH1*) file->Get("h1");

   h1->Draw();
}
```

You can also use the `TBrowser` to open the file and display the histogram.

What is going to happen if you delete the file after having retrieved the histogram from the file ?

Exercise 8: Creating a ROOT Tree containing a collection of LorentzVector's

Create a TTree containing a collection of 4D LorentzVectors. For example one could generated a list of pions (let's suppose 20/event in average) with an exponential distribution in pt and a uniform distribution in phi and Eta.

Measure the time to write a TTree with 100000 events.

Hint Hide
Create the Tree class and then define a branch containing a std::vector<ROOT::Math::XYZTVector> > (or if you prefer a std::vector). In this second case you need to generate the dictionary for the type written in the tree. You can do this by adding at the beginning of the macro these following lines

```
#ifdef __MAKECINT__
#pragma link C++ class std::vector<TLorentzVector>+;
#endif
```

For measuring the writing time you can use for example the TStopwatch class. Before the loop you create the TStopwatch class and you call TStopwatch::Start(). At the end of the macro you call TStopwatch::Stop() and then TStopwatch::Print() to get the elapsed time. You can also use the TTreePerfStats class (see its reference documentation), which will make also a summary performance graph.

Solution Hide

```
//
// Example showing how to write and read a std vector of ROOT::Math LorentzVector in a ROOT tree.
```

```
// a variable number of track Vectors is generated
// according to a Poisson distribution with random momentum exponentially distributed in pt and u
// in phi and eta.
// distributions are displayed  in a canvas.
//
// To execute the macro type in:
//
//   root[0]: .x  WriteVectorCollection.C



#include "TRandom2.h"
#include "TSystem.h"
#include "TStopwatch.h"
#include "TFile.h"
#include "TTree.h"
#include "TLorentzVector.h"
#include "TTreePerfStats.h"
#include <iostream>

#include "Math/Vector4D.h"


// if using TLorentzVector we need to generate the dictionary
// ROOT has already dictionary for vector<XYZTVector>
// #ifdef __MAKECINT__
// #pragma link C++ class std::vector<TLorentzVector>+;
// #endif


using namespace ROOT::Math;



void WriteVectorCollection(int n, int splitlevel = 0) {


  // to load dictionary library for std::vector<XYZTVector>
  gSystem->Load("libGenVector");

  TRandom2 R;
  TStopwatch timer;


  TFile f1("vectorCollection.root","RECREATE");

  // create tree
  TTree t1("t1","Tree with new LorentzVector");

  std::vector<ROOT::Math::XYZTVector>  tracks;
  std::vector<ROOT::Math::XYZTVector> * pTracks = &tracks;

  // std::vector<TLorentzVector>  tlv;
  // std::vector<TLorentzVector> * pTLV = &tlv;

  t1.Branch("tracks",&pTracks,32000,splitlevel);

  double M = 0.13957;  // set pi+ mass


  timer.Start();
  for (int i = 0; i < n; ++i) {      int nPart = R.Poisson(20);     pTracks->clear();
    pTracks->reserve(nPart);
    for (int j = 0; j < nPart; ++j) {       double pt = R.Exp(10);       double eta = R.Uniform(-

    }
```

```
    t1.Fill();
  }

  f1.Write();
  timer.Stop();
  std::cout << " Time for writing Tree with collection of LorentzVector " << timer.RealTime() <<

  t1.Print();

}
```

Try to generate the tree in split mode (default) and no-split mode. Use TTree->Print() to see the content of the generated trees. Did you see a difference in performances in writing when using or not using splitting ?

Hint Hide
For creating a branch not split, use a plot level of zero, while for splitting use 99 (the default value). The split level is passed as last parameter in TTree::Branch (see the reference documentation).

Try now to use a `TClonesArray` a `TObjArray` and measure the time to create the tree. Did you see an increase/decrease in performances ?

Hint Hide
Now you must add in the Branch the TClonesArray (or TObjArray object). Remember that for the TClonesArray you must construct them by passing the class name of the contained object. You must also use only classes deriving from TObject. Thus you can only use TLorentzVector and not the template ROOT::Math::LorentzVector class. Remember you need to use a special syntax to create the new object and to fill the TClonesArray. See the TClonesArray documentation

Solution Hide

```
//
// Example showing how to write and read a std vector of ROOT::Math LorentzVector in a ROOT tree.
// a variable number of track Vectors is generated
// according to a Poisson distribution with random momentum exponentially distributed in pt and u
// in phi and eta.
// distributions are displayed  in a canvas.
//
// To execute the macro type in:
//
//   root[0]: .x  WriteVectorCollection.C



#include "TRandom2.h"
#include "TStopwatch.h"
#include "TFile.h"
#include "TTree.h"
#include "TLorentzVector.h"
#include "TTreePerfStats.h"
#include <iostream>
#include "TClonesArray.h"


#include "Math/Vector4D.h"



using namespace ROOT::Math;
```

```
void WriteArrayCollection(int n, int splitlevel = 0) {


  TRandom2 R;
  TStopwatch timer;


  TFile f1("arrayCollection.root","RECREATE");

  // create tree
  TTree t1("t1","Tree with new LorentzVector");

  //N.B. one must specify the name of the contained class in the TClonesArray constructor
  TClonesArray * pTracks = new TClonesArray("TLorentzVector");

  //TObjArray * pTracks = new TObjArray();


  t1.Branch("tracks",&pTracks,32000,splitlevel);

  double M = 0.13957;  // set pi+ mass


  timer.Start();
  for (int i = 0; i < n; ++i) {      int nPart = R.Poisson(20);     pTracks->Clear();
    for (int j = 0; j < nPart; ++j) {       double pt = R.Exp(10);       double eta = R.Uniform(-

    }


    t1.Fill();
  }

  f1.Write();
  timer.Stop();
  std::cout << " Time for writing Tree with array of LorentzVector " << timer.RealTime() << "   "

  t1.Print();

}
```

Exercise 8b: Creating a flat ROOT Tree

This exercise show a different way to create a ROOT tree in case of a flat data structure. An simple tuple (`TNtuple` class could also be used in this case).

Create a simple ROOT tree containing 4 variables (for example x,y,z,t). Fill the tree with data (for example 10000 events) where x is generated according to a uniform distribution, y a gaussian and z an exponential and t a Landau distribution. Write also the tree in the file.

Hint Hide
Create the Tree class and then declare each branch for each simple variables as described in the lecture slides. See the documentation of TTree::Branch on how to declare branches for simple variables (fundamental types). You can also look at the tutorial `tutorials/tree/tree1.C` as example, on how `TTree:Branch` is used to define the tree branches containing the variables. Alternatively you can use also the TNtuple class. An example for the tuple class is the tutorial `tutorials/hsimple.C`.

Solution Hide

```
#include "TRandom.h"
```

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

void SimpleTree(int  n = 10000) {

// open a file
   TFile f("SimpleTree.root","RECREATE");

   TTree * data = new TTree("tree","Example TTree");
   double x, y, z, t;
   data->Branch("x",&x,"x/D");
   data->Branch("y",&y,"y/D");
   data->Branch("z",&z,"z/D");
   data->Branch("t",&t,"t/D");

// fill it with random data
   for (int i = 0; i<n; ++i) {       x = gRandom->Uniform(-10,10);
      y = gRandom->Gaus(0,5);
      z = gRandom->Exp(10);
      t = gRandom->Landau(0,2);

      data->Fill();
   }
   data->Write();
   f.Close();

}
```

Afterwards having saved the file, re-open the file and get the tree. Plot each single variable and also one variable versus another one (for example x versus y) using TTree::Draw. You can also use the TBrowser

Exercise 9: Creating a ROOT Tree containing an EventData object

Create a ROOT tree which contains an EventData object. The EventData object is defined in the file below

Show Hide

```
#ifndef EventData_h
#define EventData_h


#include <vector>
#include "Math/Vector4D.h"
#include "Math/Vector3D.h"


class Particle {
public:
   Particle() { //memset(fTags, 0, sizeof(fTags));
 }



   ROOT::Math::XYZVector  fPosition;    // vertex position
   ROOT::Math::PtEtaPhiMVector fVector;     // particle vector
   int  fCharge;         // particle charge
   int fType;            // particle type (0=pho, 1 ele, 2...)
};

class TF1;

class EventData {
public:

   std::vector<Particle> fParticles; // particles of the event
   int fEventSize; // size (in bytes) of the event
```

```
   void SetSize() {
      fEventSize = sizeof(EventData) + fParticles.size() * sizeof(Particle);
   }
   void Clear() {
      fParticles.clear();
   }

   void AddParticle(const Particle& p) { fParticles.push_back(p); }

   void Generate();

   //ClassDef(EventData,1); // Data for an event
};


#ifdef __MAKECINT__
#pragma link C++ class Particle+;
#pragma link C++ class EventData+;
#pragma link C++ class std::vector<Particle>+;
#endif

#endif
```

It contains a collection (a `std::vector`) of `Particle` object. The `Particle` object contains the initial vertex position of the particle, the particle momentum , its charge and a different code, depending if it is a photon, an electron, a muon, a charged pion or a changed kaon.

The EventData object provides a method (`Generate()`) to generate one event and it is implemented in the file below

Show Hide

```
#include "EventData.h"
#include "TRandom.h"
#include "TGenPhaseSpace.h"
#include "Math/Vector4D.h"
#include "TLorentzVector.h"

// generate the events

// parameters

int NTRACKS = 40;
// parameter for pt distribution
double PtAvg = 20;
int NTYPES = 8;
double masses[] = { 0, 0.0005, 0.105, 0.135, 0.139, 0.4937, 0.4976, 3.096 };
int charge [] = { 0, 1, 1, 0, 1, 1, 0, 0 };
double fractions[] = { 0.1, 0.12, 0.08, 0.05, 0.5, 0.1, 0.049 , 0.01 };
double sigmax = 10.E-6;
double sigmay = 10.E-6;
double sigmaz = 5.;

using namespace ROOT::Math;

ROOT::Math::PtEtaPhiMVector SmearVector(const ROOT::Math::XYZTVector & v) {
   double x = v.X()*(1. + gRandom->Gaus(0, 0.1) );
   double y = v.Y()*(1. + gRandom->Gaus(0, 0.1) );
   double z = v.Z()*(1. + gRandom->Gaus(0, 0.1) );
   ROOT::Math::PxPyPzMVector tmp(x,y,z,v.M() );
   return PtEtaPhiMVector(tmp);
}
```

```
void EventData::Generate()  {

   // get expected value for each type
   for (int i = 0; i < NTYPES; ++i) {        double nexp = fractions[i] * NTRACKS;        int np
      for (int j = 0; j < np; ++j) {           Particle p;           p.fPosition = XYZVector( gRa
         double pt = gRandom->Exp(PtAvg);
         double eta = gRandom->Uniform(-3,3);
         double phi = gRandom->Uniform(-TMath::Pi(), TMath::Pi() );
         double mass = masses[i];
         p.fVector = PtEtaPhiMVector(pt, eta, phi, mass);
         p.fType = i;
         p.fCharge = charge[i];
         if (p.fCharge) {
            int tmp = gRandom->Integer(2);
            if (tmp == 0) p.fCharge = -1;
         }
         // special case for decays
         if (i == 3 ) {
            // pi0
            TGenPhaseSpace evt;
            double m[2] = {0,0};
            TLorentzVector W( p.fVector.X(), p.fVector.Y(), p.fVector.Z(), p.fVector.E() );
            evt.SetDecay(W, 2, m);
            evt.Generate();
            TLorentzVector * v1 = evt.GetDecay(0);
            TLorentzVector * v2 = evt.GetDecay(1);
            Particle p1;
            Particle p2;
            p1.fPosition = p.fPosition;
            p2.fPosition = p.fPosition;
            p1.fCharge = 0;
            p2.fCharge = 0;
            p1.fType = 0;
            p2.fType = 0;

            p1.fVector = SmearVector(XYZTVector(v1->X(), v1->Y(), v1->Z(), v1->E() ));
            p2.fVector = SmearVector(XYZTVector(v2->X(), v2->Y(), v2->Z(), v2->E() ));
            AddParticle(p1);
            AddParticle(p2);
         }
         if (i == 6) {
            // Ks
            TGenPhaseSpace evt;
            double m[2] = {0.139,0.139};
            TLorentzVector W( p.fVector.X(), p.fVector.Y(), p.fVector.Z(), p.fVector.E() );
            evt.SetDecay(W, 2, m);
            evt.Generate();
            TLorentzVector * v1 = evt.GetDecay(0);
            TLorentzVector * v2 = evt.GetDecay(1);
            double ctau = 2.6844 * p.fVector.Gamma();
            double disp = gRandom->Exp(ctau);
            double dispX = disp * p.fVector.X()/p.fVector.P();
            double dispY = disp * p.fVector.Y()/p.fVector.P();
            double dispZ = disp * p.fVector.Z()/p.fVector.P();
            Particle p1;
            Particle p2;
            p1.fPosition = XYZVector( p.fPosition.X() + dispX,  p.fPosition.Y() + dispY,  p.fPosi
            p2.fPosition = p1.fPosition;
            p1.fCharge = 1;
            p2.fCharge = -1;
            p1.fType = 4;
            p2.fType = 4;

            p1.fVector = SmearVector(XYZTVector(v1->X(), v1->Y(), v1->Z(), v1->E() ));
            p2.fVector = SmearVector(XYZTVector(v2->X(), v2->Y(), v2->Z(), v2->E() ));
```

```
            AddParticle(p1);
            AddParticle(p2);
         }
         if (i == 7 ) {
            // J/psi
            TGenPhaseSpace evt;
            double m1[2] = {0.0005,0.0005};
            double m2[2] = {0.105,0.105};
            int tmp = gRandom->Integer(2);
            TLorentzVector W( p.fVector.X(), p.fVector.Y(), p.fVector.Z(), p.fVector.E() );
            if (tmp == 0)
               evt.SetDecay(W, 2, m1);
            else
               evt.SetDecay(W, 2, m2);

            evt.Generate();

            TLorentzVector * v1 = evt.GetDecay(0);
            TLorentzVector * v2 = evt.GetDecay(1);
            Particle p1;
            Particle p2;
            p1.fPosition = p.fPosition;
            p2.fPosition = p.fPosition;
            p1.fCharge = 1;
            p2.fCharge = -1;
            p1.fType = 1;
            p2.fType = 1;
            if (tmp == 1) {
               p1.fType = 2;
               p2.fType = 2;
            }

            p1.fVector = SmearVector(XYZTVector(v1->X(), v1->Y(), v1->Z(), v1->E() ));
            p2.fVector = SmearVector(XYZTVector(v2->X(), v2->Y(), v2->Z(), v2->E() ));
            AddParticle(p1);
            AddParticle(p2);
         }
         else
            AddParticle(p);

      }
   }
}
```

Look at the macro defining the EventData class and the implementation generating the events. Afterwards, write the macro creating the tree and fill it with the EventData objects, which has to be generated for each event.

Solution Hide

```
// Prefer compiled:
#include "TTree.h"
#include "TFile.h"
#include "TRandom.h"
#include "TMath.h"
#include <vector>

#include "EventData.h"

void CreateEventTree(const char *fn = "eventdata_s0.root",
                     Long64_t numEvents = 10000, int splitlevel = 1) {
   TFile* f = new TFile(fn, "RECREATE");
   TTree* tree = new TTree("tree", "Tutorial tree");

   EventData* event = new EventData();
   tree->Branch("event", &event,32000,splitlevel);
```

```
   Particle p;

   for (Long64_t i = 0; i < numEvents; ++i) {        event->Clear();
      event->Generate();
      event->SetSize();

      tree->Fill();

   }

   tree->Print();
   tree->Write();
   delete f;
}
```

Look at the macro and try to understand it. Run the macro to create and write the tree in the file.

Exercise 10: Create and query ROOT Tree from a data text file

This exercise show how you can create a Tree from a data file in text format (e.g. csv file). The aim is to create a Tree from a CMS public LHC data set which is is text format. You can use for example the J/Psi dimuon data available at this link.

- Create a TTree from the given CMS text data using the function TTree::ReadFile
- plot the di-muon invariant mass using `TTree::Draw` (variable `M`) for all the events
- plot the di-muon invariant mass for the same charged and opposite charged muons
- plot the pt distribution as function of eta in a profile plot for the first and the second muon
- save the tree in a root binary file.

Hint Hide
Use the function `TTree::ReadFile` to fill a tree from data from a text file. In principle the function should be able to read the branch names directly from the first line of the text file, if the first line contains also the description of the branch type (e.g. `"x/F"`). Since this is not the case It is recommended then to create a string defining the branches (`"Type/C:RunNo/I:EvtNo/I:E1:/F:px1:py1.....")` and pass it as second argument to `TTree::ReadFile`. Note that if the following branches are of the same type one can omit the branch type descriptor. For example a valid string for that CMS file is

```
Type/C:RunNo/I:EvNo/I:E1/F:px1:py1:pz1:pt1:eta1:phi1:Q1/I:E2/F:px2:py2:pz2:pt2:eta2:phi2:Q2/I:M/F
```

Use then `TTree::Draw` to query the variable in the Tree.

Solution Hide

```
// example of creating a TTree from a csv file

void CreateFromCSV(const char * filename) {

   // remove prefix from filename
   TString tmp(filename);
   TString fileOut = tmp(0,tmp.First("."));
   fileOut += TString(".root");

   TFile * file = new TFile(fileOut,"RECREATE");
   TTree * t = new TTree("tree","tree");


   // define first the branch
   TString branchDescription("Type/C:RunNo/I:EvNo/I:E1/F:px1:py1:pz1:pt1:eta1:phi1:Q1/I:E2/F:px2:

   t->ReadFile(filename,branchDescription,',');

   t->Print();
```

```
   t->Draw("M");
   t->Write();
   file->Close();

}
```

Here is the code to analyze the tree using `TTree::Draw`

```
// to draw invariant mass
tree->Draw("M");
tree->Draw("M","Q1*Q2==-1");
tree->Draw("M","Q1*Q2==1");
 // to draw the profile
tree->Draw("pt1:eta1 >> p1(40,-3,3)","","prof");
tree->Draw("pt2:eta2 >> p2(40,-3,3)","","prof");
```

Exercise 10b: Read a ROOT Tree containing a collection of LorentzVector's

Use `TTree::Draw` to plot:

- The pt distribution of the tracks
- The number of tracks per event in an histogram with 50 bins between 0 and 50.
- The E distribution for |eta| < 2
- A profile plot showing the pt of the number of tracks vs eta.

Hint Hide

- To plot the size of the collection use the special keyword "@".
- To make a profile plot use the graphics option "prof" (3rd parameter in TTree::Draw).

Solution Hide

```
{
     TFile f("vectorCollection.root");
      t1->Draw("tracks.Pt()");
      t1->Draw("@tracks.size() >> h1(50,0,50)");
      t1->Draw("tracks.E()","abs(tracks.Eta() < 2)");
      t1->Draw("tracks.Pt():tracks.Eta() >> prof(50,-4,4)","","prof");
}
```

See also TTree::Draw documentation

Use then C++ code to plot the invariant mass of all 2-tracks combinations. This you cannot do with `TTree;:Draw`

Hint Hide

To read the tree using C++ code, write a macro, where you retrieve the `TTree` from the file and then loop on its entry, retrieve the needed object data and then fill the histograms. In more detail, this is a suggestion on how to write this code:

- Open the file using its file name in `TFile::Open()` and get the Tree. Remember to check if the file pointer is not null. If it is null means the file is not existing.
- Get then a pointer to the tree.
- Connect a Tree Branch with the Data Member.We have to somehow connect the branch we want to read with the variables used to actually store the data by calling `TTree::SetBranchAddress()`.
- Load the TTree data. For the analysis example we need to access the vector of tracks, which is stored in the branch with name `"tracks"`. But the TTree first needs to load the data for each event it

contains. For that call TBranch::GetEntry(entry) in a loop, passing the TTree entry number from the loop index to `GetEntry()`. Again `TBranch` is the class name, but you obviously need to call it on an object. To know how many entries the tree contains, simply call TTree::GetEntries().

- Without the call to `GetEntry()`, the variables will not contain data. `GetEntry()` loads the data into the variables connected with the tree by the call to `SetBranchAddress()`.
- Once you have the event data (the vector of tracks) you can loop on its elements.
- Make all the combination of 2-elements (2 tracks) and add them together to retrieve the invariant mass. Just use the M() function of the added LorentzVector to get the invariant mass.
- Fill an histogram with the obtained value
- Plot the histogram at the end of the loop

Solution Hide

```
//
// Example showing how to write and read a std vector of ROOT::Math LorentzVector in a ROOT tree.
// In the write() function a variable number of track Vectors is generated
// according to a Poisson distribution with random momentum uniformly distributed
// in phi and eta.
// In the read() the vectors are read back and the content analyzed and
// some information such as number of tracks per event or the track pt
// distributions are displayed  in a canvas.
//
// To execute the macro type in:
//
//   root[0]: .x  vectorCollection.C



#include "TStopwatch.h"
#include "TFile.h"
#include "TTree.h"
#include "TH1.h"
#include "TCanvas.h"
#include "TMath.h"
#include "TLorentzVector.h"
#include "TTreePerfStats.h"
#include <iostream>

// CINT does not understand some files included by LorentzVector
#include "Math/Vector4D.h"


// #ifdef __MAKECINT__
// #pragma link C++ class std::vector<TLorentzVector>+;
// #endif


using namespace ROOT::Math;



void ReadVectorCollection(const char * fileName = "vectorCollection.root") {


  TH1D * h1 = new TH1D("h1","Number of track per event",51,-0.5,50.5);
  TH1D * h2 = new TH1D("h2","Track Px",100,0,100);
  TH1D * hinv = new TH1D("hinv","Track invariant mass",100,0,100);


  TFile f1(fileName);

  // create tree
```

```
  TTree *t1 = (TTree*)f1.Get("t1");

  std::vector<ROOT::Math::LorentzVector<ROOT::Math::PxPyPzE4D<double> > > * pTracks = 0;
  t1->SetBranchAddress("tracks",&pTracks);

  TStopwatch timer;
  timer.Start();
  int n = (int) t1->GetEntries();
  std::cout << " Tree Entries " << n << std::endl;    for (int i = 0; i < n; ++i) {      t1->GetE
    int ntrk = pTracks->size();
    h1->Fill(ntrk);
    for (int j = 0; j < ntrk; ++j) {        XYZTVector v = (*pTracks)[j];        h2->Fill(v.X());
      // compute invariant mass
      for (int k = j+1; k < ntrk; ++k) {          XYZTVector q = (*pTracks)[k];          double
      }
    }
  }


  timer.Stop();
  std::cout << " Time to read Tree " << timer.RealTime() << "  " << timer.CpuTime() << std::endl;

  c1->cd(1);
  h1->Draw();
  c1->cd(2);
  h2->Draw();
  c1->cd(3);
  hinv->Draw();
}
```

Exercise 11: Analyzing a ROOT Tree using `TTree::MakeClass`

Read the tree containing the EventData class. The aim is to get the invariant mass distribution of the photons, of the opposite charged particles and of the opposite charged leptons (electrons and muons). You can write code to read the Tree using `TTree::MakeClass`. But you can also write yourself or use `TTree::MakeProxy`. We will see in one of the next exercise on how to use `TTree::MakeSelector`. Note that in order to get the right definition of the top-level branches, one needs to generate the tree with split level=1. In case of a tree with a different splitting level, one needs to declare itself the needed branches and the contained variables and call `TTree::SetBranchAddress` in the initialisation of the analysis class. Future versions of ROOT should have this limitation removed and should be able to correctly define the contained branches.

Hint Hide
Use =TTree::MakeClass("myclassname") to generate the header file and the implementation of the class code required to analyze the Tree. Fill the implementation file with the needed code to get the invariant masses distributions. To run the code just do from the ROOT prompt (let's suppose the classname generated is called `EventDataClass`)

```
root[0]  .L EventDataClass.C
root[1]  EventDataClass c;
root[2] c.Loop();
```

Solution Hide
Header file obtained from `MakeClass`. It did not require any changes by the user.

```
//////////////////////////////////////////////////////////
// This class has been automatically generated on
// Thu Nov 14 18:50:55 2013 by ROOT version 5.34/11
// from TTree tree/Tutorial tree
// found on file: eventdata.root
//////////////////////////////////////////////////////////
```

```
#ifndef EventDataClass_h
#define EventDataClass_h

#include <TROOT.h>
#include <TChain.h>
#include <TFile.h>

// Header file for the classes stored in the TTree if any.
#include "./EventData.h"

// Fixed size dimensions of array or collections stored in the TTree if any.

class EventDataClass {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

   // Declaration of leaf types
 //EventData       *event;
   vector<Particle> fParticles;
   Int_t           fEventSize;

   // List of branches
   TBranch        *b_event_fParticles;   //!
   TBranch        *b_event_fEventSize;   //!

   EventDataClass(TTree *tree=0);
   virtual ~EventDataClass();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};

#endif

#ifdef EventDataClass_cxx
EventDataClass::EventDataClass(TTree *tree) : fChain(0)
{
// if parameter tree is not specified (or zero), connect the file
// used to generate this class and read the Tree.
   if (tree == 0) {
      TFile *f = (TFile*)gROOT->GetListOfFiles()->FindObject("eventdata.root");
      if (!f || !f->IsOpen()) {
         f = new TFile("eventdata.root");
      }
      f->GetObject("tree",tree);

   }
   Init(tree);
}

EventDataClass::~EventDataClass()
{
   if (!fChain) return;
   delete fChain->GetCurrentFile();
}

Int_t EventDataClass::GetEntry(Long64_t entry)
{
// Read contents of entry.
   if (!fChain) return 0;
   return fChain->GetEntry(entry);
}
```

```
Long64_t EventDataClass::LoadTree(Long64_t entry)
{
// Set the environment to read one entry
   if (!fChain) return -5;
   Long64_t centry = fChain->LoadTree(entry);
   if (centry < 0) return centry;    if (fChain->GetTreeNumber() != fCurrent) {
      fCurrent = fChain->GetTreeNumber();
      Notify();
   }
   return centry;
}

void EventDataClass::Init(TTree *tree)
{
   // The Init() function is called when the selector needs to initialize
   // a new tree or chain. Typically here the branch addresses and branch
   // pointers of the tree will be set.
   // It is normally not necessary to make changes to the generated
   // code, but the routine can be extended by the user if needed.
   // Init() will be called many times when running on PROOF
   // (once per file to be processed).

   // Set branch addresses and branch pointers
   if (!tree) return;
   fChain = tree;
   fCurrent = -1;
   fChain->SetMakeClass(1);

   fChain->SetBranchAddress("fParticles", &fParticles, &b_event_fParticles);
   fChain->SetBranchAddress("fEventSize", &fEventSize, &b_event_fEventSize);
   Notify();
}

Bool_t EventDataClass::Notify()
{
   // The Notify() function is called when a new file is opened. This
   // can be either for a new TTree in a TChain or when when a new TTree
   // is started when using PROOF. It is normally not necessary to make changes
   // to the generated code, but the routine can be extended by the
   // user if needed. The return value is currently not used.

   return kTRUE;
}

void EventDataClass::Show(Long64_t entry)
{
// Print contents of entry.
// If entry is not specified, print current entry
   if (!fChain) return;
   fChain->Show(entry);
}
Int_t EventDataClass::Cut(Long64_t entry)
{
// This function may be called from Loop.
// returns  1 if entry is accepted.
// returns -1 otherwise.
   return 1;
}
#endif // #ifdef EventDataClass_cxx
```

Implementation file of class `EventDataClass` containing the code to plot the invariant masses

```
#define EventDataClass_cxx
#include "EventDataClass.h"
#include <TH2.h>
#include <TStyle.h>
```

```
#include <TCanvas.h>
#include <TStopwatch.h>

void EventDataClass::Loop()
{
//   In a ROOT session, you can do:
//      Root > .L EventDataClass.C
//      Root > EventDataClass t
//      Root > t.GetEntry(12); // Fill t data members with entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//

//     This is the loop skeleton where:
//    jentry is the global entry number in the chain
//    ientry is the entry number in the current Tree
//  Note that the argument to GetEntry must be:
//    jentry for TChain::GetEntry
//    ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//    fChain->SetBranchStatus("*",0);  // disable all branches
//    fChain->SetBranchStatus("branchname",1);  // activate branchname
// METHOD2: replace line
//    fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch
   if (fChain == 0) return;

   TH1 * h1 = new TH1F("h1","inv mass neutrals",1000,0,1);
   TH1 * h2 = new TH1F("h2","inv mass charged",1000,0,1);
   TH1 * h3 = new TH1F("h3","inv mass leptons",1000,0,10);

   TStopwatch timer; timer.Start();

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {       Long64_t ientry = LoadTree(jentry);

      nb = b_event_fParticles->GetEntry(jentry); nbytes += nb;


      int npart = fParticles.size();
      for (int i = 0; i< npart; ++i) {           for (int j = i+1; j< npart; ++j) {
            if (fParticles[i].fCharge * fParticles[j].fCharge  == -1 )
               h2->Fill( (fParticles[i].fVector + fParticles[j].fVector).M() );
            if ( (fParticles[i].fCharge * fParticles[j].fCharge  == -1 ) &&
                 ( (fParticles[i].fType == 1 && fParticles[j].fType == 1 ) ||
                   (fParticles[i].fType == 2 && fParticles[j].fType == 2 ) ) )
               h3->Fill( (fParticles[i].fVector + fParticles[j].fVector).M() );

         }
      }
      // if (Cut(ientry) < 0) continue;    }    timer.Stop();    std::cout << "Total number of by
   c1->cd(1);
   h1->Draw();
   c1->cd(2);
   h2->Draw();
   c1->cd(3);
   h3->Draw();

}
```

Exercise 11b: Analyze a ROOT Tree using the new `TTreeReader` (Available only on ROOT 5.99)

Read the tree containing the `EventData` objects using the `TTreeReader` class. See as example the tutorial `tutorials/tree/TreeReaderSimple.cxx`. You need to have ROOT 5.99 installed to run this tutorial, since the `TTreeReaderClass` is not available in the ROOT version 5.34.

Show Hide

```
// program to test treereader functionality

#include "TFile.h"
#include "TH1F.h"
#include "TTreeReader.h"
#include "TTreeReaderValue.h"
#include "Math/LorentzVector.h"
#include "Math/Vector4d.h"
#include <iostream>

#include "TStopwatch.h"

#include "TCanvas.h"
#include "EventData.h"

void EventDataReader() {

   TH1 * h1 = new TH1F("h1","inv mass neutrals",1000,0,1);
   TH1 * h2 = new TH1F("h2","inv mass charged",1000,0,1);
   TH1 * h3 = new TH1F("h3","inv mass leptons",1000,0,10);

   TFile *myFile = TFile::Open("eventdata_s99.root");

   TTreeReader myReader("tree", myFile);

   TTreeReaderValue<std::vector< Particle> > particles_value(myReader, "fParticles");


   std::cout << myReader.GetEntries(1) << std::endl;    TStopwatch w;     w. Start();     while (
      int npart = fParticles.size();


      for (int i = 0; i< npart; ++i) {           for (int j = i+1; j< npart; ++j) {
            if (fParticles[i].fCharge * fParticles[j].fCharge  == -1 )
               h2->Fill( (fParticles[i].fVector + fParticles[j].fVector).M() );
            if ( (fParticles[i].fCharge * fParticles[j].fCharge  == -1 ) &&
                 ( (fParticles[i].fType == 1 && fParticles[j].fType == 1 ) ||
                   (fParticles[i].fType == 2 && fParticles[j].fType == 2 ) ) )
               h3->Fill( (fParticles[i].fVector + fParticles[j].fVector).M() );

         }
      }


   }

   std::cout << "Time to read the tree:  ";    w.Print();    TCanvas * c1 = new TCanvas();     c1
   c1->cd(1);
   h1->Draw();
   c1->cd(2);
   h2->Draw();
   c1->cd(3);
   h3->Draw();

}
```

Exercise 12: Using the TSelector class for analysing a TTree

Create the Selector for the EventData TTree made before. Use TTree::MakeSelector to create your own Selector class. The aim as in a previous exercise is to plot the invariant masses for:

- the photons,
- the opposite charged particles
- for the muon and electrons with opposite charge

Inside the code of your Selector do the following:

- book the histograms in the initialisation routine
- fill the histogram in the Process function
- draw the histogram in the Terminate function

Remember to generate first the TTree with a split-level = 0 to be able to have `TTree::MakeSelector` generating correctly the code for reading the collection (`std::vector`) object. Otherwise, you will have to declare its branch definition by hand using `TTree::SetBranchAddress`.

Hint Hide
Here is what you need to do, after having opened the file with the tree, first load the dictionary library for the `EventData` class

```
.L EventData.cxx+;
```

then create your `Selector` class

```
tree->MakeSelector("EventDataSelector");
```

The file `EventDataSelector.h` and `EventDataSelector.C` will be created. Add in `EventDataSelector.h`, inside the class `EventDataSelector`, a new data member, the histograms you want to create,

```
TH1D *  h_t;
```

Edit then the file EventDataSelector.C and add in `EventDataSelector::SlaveBegin` the booking of the histograms. For example:

```
h_t = new TH1D("h_t","t",100,0,100);
```

In `EventDataSelector::Process` the filling of the histogram after calling `TSelector::GetEntry()`

```
GetEntry(entry);
h_t->Fill(t);
```

In `EventDataSelector::Terminate` add the drawing of the histogram.

After having saved the file run the selection by doing (for example from the ROOT prompt):

```
TFile f("tree.root");
tree->Process("EventDataSelector.C+");
```

Solution Hide
Header file

```
//////////////////////////////////////////////////////////
// This class has been automatically generated on
```

```
// Mon Nov 11 18:52:15 2013 by ROOT version 5.34/11
// from TTree tree/Tutorial tree
// found on file: eventdata_s1.root
//////////////////////////////////////////////////////////

#ifndef EventDataSelector_h
#define EventDataSelector_h

#include <TROOT.h>
#include <TChain.h>
#include <TFile.h>
#include <TSelector.h>

// Header file for the classes stored in the TTree if any.
#include "./EventData.h"

#ifdef __MAKECINT__
#pragma link C++ class Particle+;
#pragma link C++ class EventData+;
#pragma link C++ class std::vector<Particle>+;
#endif



// Fixed size dimensions of array or collections stored in the TTree if any.

class EventDataSelector : public TSelector {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain

   // Declaration of leaf types
 //EventData       *event;
   vector<Particle> fParticles;
   Int_t           fEventSize;

   // List of branches
   TBranch        *b_event_fParticles;   //!
   TBranch        *b_event_fEventSize;   //!

   EventDataSelector(TTree * /*tree*/ =0) : fChain(0) { }
   virtual ~EventDataSelector() { }
   virtual Int_t   Version() const { return 2; }
   virtual void    Begin(TTree *tree);
   virtual void    SlaveBegin(TTree *tree);
   virtual void    Init(TTree *tree);
   virtual Bool_t  Notify();
   virtual Bool_t  Process(Long64_t entry);
   virtual Int_t   GetEntry(Long64_t entry, Int_t getall = 0) { return fChain ? fChain->GetTree()
   virtual void    SetOption(const char *option) { fOption = option; }
   virtual void    SetObject(TObject *obj) { fObject = obj; }
   virtual void    SetInputList(TList *input) { fInput = input; }
   virtual TList  *GetOutputList() const { return fOutput; }
   virtual void    SlaveTerminate();
   virtual void    Terminate();


   TH1 * h1;
   TH1 * h2;
   TH1 * h3;


   ClassDef(EventDataSelector,0);
};

#endif

#ifdef EventDataSelector_cxx
```

```
void EventDataSelector::Init(TTree *tree)
{
   // The Init() function is called when the selector needs to initialize
   // a new tree or chain. Typically here the branch addresses and branch
   // pointers of the tree will be set.
   // It is normally not necessary to make changes to the generated
   // code, but the routine can be extended by the user if needed.
   // Init() will be called many times when running on PROOF
   // (once per file to be processed).

   // Set branch addresses and branch pointers
   if (!tree) return;
   fChain = tree;
   fChain->SetMakeClass(1);

   fChain->SetBranchAddress("fParticles", &fParticles, &b_event_fParticles);
   fChain->SetBranchAddress("fEventSize", &fEventSize, &b_event_fEventSize);
}

Bool_t EventDataSelector::Notify()
{
   // The Notify() function is called when a new file is opened. This
   // can be either for a new TTree in a TChain or when when a new TTree
   // is started when using PROOF. It is normally not necessary to make changes
   // to the generated code, but the routine can be extended by the
   // user if needed. The return value is currently not used.

   return kTRUE;
}

#endif // #ifdef EventDataSelector_cxx
```

Implementation file. The output list is used for the histograms, so this selector is ready to be used by PROOF (see next exercise).

```
#define EventDataSelector_cxx
// The class definition in EventDataSelector.h has been generated automatically
// by the ROOT utility TTree::MakeSelector(). This class is derived
// from the ROOT class TSelector. For more information on the TSelector
// framework see $ROOTSYS/README/README.SELECTOR or the ROOT User Manual.

// The following methods are defined in this file:
//    Begin():        called every time a loop on the tree starts,
//                    a convenient place to create your histograms.
//    SlaveBegin():   called after Begin(), when on PROOF called only on the
//                    slave servers.
//    Process():      called for each event, in this function you decide what
//                    to read and fill your histograms.
//    SlaveTerminate: called at the end of the loop on the tree, when on PROOF
//                    called only on the slave servers.
//    Terminate():    called at the end of the loop on the tree,
//                    a convenient place to draw/fit your histograms.
//
// To use this file, try the following session on your Tree T:
//
// Root > T->Process("EventDataSelector.C")
// Root > T->Process("EventDataSelector.C","some options")
// Root > T->Process("EventDataSelector.C+")
//

#include "EventDataSelector.h"
#include <TH2.h>
#include "TCanvas.h"
#include <TStyle.h>
#include "TStopwatch.h"
```

```
TStopwatch w;

void EventDataSelector::Begin(TTree * /*tree*/)
{
   // The Begin() function is called at the start of the query.
   // When running with PROOF Begin() is only called on the client.
   // The tree argument is deprecated (on PROOF 0 is passed).

   TString option = GetOption();
   w.Start();
}

void EventDataSelector::SlaveBegin(TTree * /*tree*/)
{
   // The SlaveBegin() function is called after the Begin() function.
   // When running with PROOF SlaveBegin() is called on each slave server.
   // The tree argument is deprecated (on PROOF 0 is passed).

   TString option = GetOption();

   h1 = new TH1F("h1","inv mass neutrals",1000,0,1);
   h2 = new TH1F("h2","inv mass charged",1000,0,1);
   h3 = new TH1F("h3","inv mass leptons",1000,0,10);

   fOutput->Add(h1);
   fOutput->Add(h2);
   fOutput->Add(h3);

}

Bool_t EventDataSelector::Process(Long64_t entry)
{
   // The Process() function is called for each entry in the tree (or possibly
   // keyed object in the case of PROOF) to be processed. The entry argument
   // specifies which entry in the currently loaded tree is to be processed.
   // It can be passed to either EventDataSelector::GetEntry() or TBranch::GetEntry()
   // to read either all or the required parts of the data. When processing
   // keyed objects with PROOF, the object is already loaded and is available
   // via the fObject pointer.
   //
   // This function should contain the "body" of the analysis. It can contain
   // simple or elaborate selection criteria, run algorithms on the data
   // of the event and typically fill histograms.
   //
   // The processing can be stopped by calling Abort().
   //
   // Use fStatus to set the return value of TTree::Process().
   //
   // The return value is currently not used.

   b_event_fParticles->GetEntry(entry);

   int npart = fParticles.size();
   for (int i = 0; i< npart; ++i) {        for (int j = i+1; j< npart; ++j) {           if (fPart
         if (fParticles[i].fCharge * fParticles[j].fCharge  == -1 )
            h2->Fill( (fParticles[i].fVector + fParticles[j].fVector).M() );
         if ( (fParticles[i].fCharge * fParticles[j].fCharge  == -1 ) &&
            ( (fParticles[i].fType == 1 && fParticles[j].fType == 1 ) ||
              (fParticles[i].fType == 2 && fParticles[j].fType == 2 ) ) )
            h3->Fill( (fParticles[i].fVector + fParticles[j].fVector).M() );

      }
   }

   return kTRUE;
}
```

```
void EventDataSelector::SlaveTerminate()
{
   // The SlaveTerminate() function is called after all entries or objects
   // have been processed. When running with PROOF SlaveTerminate() is called
   // on each slave server.
   std::cout << "terminate slave " << std::endl; } void EventDataSelector::Terminate() {     // Th
   c1->cd(1);
   TObject * oh1 = fOutput->FindObject("h1");
   if (oh1) oh1->Draw();
   c1->cd(2);
   TObject * oh2 = fOutput->FindObject("h2");
   if (oh2) oh2->Draw();
   c1->cd(3);
   TObject * oh3 = fOutput->FindObject("h3");
   if (oh3) oh3->Draw();

}

#include "EventData.cxx"
```

Exercise 13: Chaining ROOT Files.

Using the macro to create the EventData tree, run it few times (e.g. 2 or 3 times) using a different file name each time. Afterwards use then the `TChain` class to merge the trees and analyse the obtained chain as in Exercise 11.

Hint Hide
See the example in the lecture slide on how to use `TTree::Chain` or its reference documentation. The `TChain` must be created passing the name of the `TTree` existing in the files.

Solution Hide
This are the few lines to create the TChain, that you can run directly from the prompt. You can also use wildcard's to chain many files

```
%CODE{"cpp" style="background: yellow;"  }%
TChain chain("tree");
chain.Add("event*.root");
chain.Draw("@fParticles.size()");
```

Interactive Data Analysis with PROOF

Here are the slides for the PROOF lecture: Introduction, Terminology, Basics

Exercise 14a: Run

Download these files ProofSimple.C and ProofSimple.h and run the exercise shown on the slides.

Exercise 14b: Run

The TProofBench package can be used to measure the performance of a PROOF 'cluster' (lap/desktop or a real cluster) for a given task. By default it runs a CPU-intensive and basic data-intensive task. The exercise consist in running the default code in PROOF-Lite.

Hint Hide
Check the TProofBench pages.

Solution Hide

For the CPU-intensive benchmark do:

```
root [] TProofBench pb("lite://", "cpu-test.root")
root [] pb.RunCPU()
```

For the data-intensive benchmark do:

```
root [] TProofBench pb("lite://", "io-test.root")
root [] pb.MakeDataSet()
root [] pb.RunDataSet()
```

We will learn in this exercise how we can analyse a data set (a chain of ROOT files containing a Tree)

Exercise 14c: Using PROOF to analyze the TTree

Generate 10 files with CreateEventTree in a subdirectory called 'data'. Create a TChain with these files. Using the TSelector defined in the exercise on TTrees - `EventDataSelector` - run the selector using PROOF-Lite. Try to compare the processing time with the standard TChain processing.

Hint Hide
Make sure the selector compiles with

```
root [] TSelector::GetSelector("EventDataSelector.C+")
```

Check the TProofBench pages.

Hint Hide
Look at the slides for ways to createthe TChain and to the TChain reference documentation for enabling PROOF. Use gROOT->Time() to measure times.

Solution Hide
Create the files

```
$ mkdir data
$ root -l
root [] .L EventData.cxx+
root [] .L CreateEventTree.C
root [] for(Int_t i = 0; i < 10; ++i) { CreateEventTree(Form("data/evtree_%d.root", i)); }
```

Create the TChain (use the correct path ... /home/admin if required ...)

```
root [] TChain chain("tree")
root [] for(Int_t i = 0; i < 10; ++i) { chain.AddFile(Form("file:///home/user/data/evtree_%d.root
root [] chain.ls()
```

Process locally

```
root [] gROOT->Time()
root [] chain.Process("EventDataSelector.C+")
```

Process in PROOF-Lite

```
root [] TProof::Open("lite://")
root [] gProof->Load("EventData.cxx+")
root [] chain.SetProof()
root [] gROOT->Time()
root [] chain.Process("EventDataSelector.C+")
```

After having done this, try to run the same as a dataset. You need to create a TFileCollection for that, to register it as 'eventdatads', verify it and process it.

Hint Hide
The best way to create a TFileCollection for a limited number of files is to create a text file with one file path per line. Look at the refeference documentation for TFileCollection. Remember to put the full path.

Solution Hide
Download the file eventdatads.txt; make sure that the paths are correct for your working area.

Create the TFileCollection, start PROOF-Lite and register it

```
root [] TFileCollection *fc = new TFileCollection("","","eventdatads.txt")
root [] TProof::Open("lite://")
root [] gProof->RegisterDataSet("eventdatads", fc)
root [] gProof->ShowDataSets()
```

Verify the dataset, i.e. open the files and get some info from inside

```
root [] gProof->VerifyDataSet("eventdatads")
root [] gProof->ShowDataSets()
```

Process the dataset by name

```
root [] gProof->Load("EventData.cxx+")
root [] gProof->Process("eventdatads", "EventDataSelector.C+")
```

Exercise 14d: Using PROOF to generate events with Pythia8

The purpose of the exercise is to run a TSelector which generated events using Pythia8. You need the following files: ProofPythia.C, ProofPythia.h and the PAR file pythia8.par.txt (save `pythia8.par.txt` as `pythia8.par`). We need to make the PYTHIA8 and PYTHIA8DATA variables to point to the pythia8 installation under `/cvmfs/sft.cern.ch/lcg/dev/root-pythia8-xrootd/x86_64-slc6-gcc47-python27/`.

Solution Hide
First we export the environment variable in the shell

```
$ export PYTHIA8=/cvmfs/sft.cern.ch/lcg/dev/root-pythia8-xrootd/x86_64-slc6-gcc47-python27/pythia
$ export PYTHIA8DATA=/cvmfs/sft.cern.ch/lcg/dev/root-pythia8-xrootd/x86_64-slc6-gcc47-python27/py
```

Then we start ROOT and we set the variables to be transferred to PROOF

```
$ root -l
root [] TProof::AddEnvVar("PYTHIA8", "/cvmfs/sft.cern.ch/lcg/dev/root-pythia8-xrootd/x86_64-slc6-
root [] TProof::AddEnvVar("PYTHIA8DATA", "/cvmfs/sft.cern.ch/lcg/dev/root-pythia8-xrootd/x86_64-s
root [] TProof::Open("lite://")
```

We can now load the package and run

```
root [] gProof->UploadPackage("pythia8.par")
root [] gProof->EnablePackage("pythia8")
root [] gProof->Process("ProofPythia.C+", 10000)
```

Fitting in ROOT (slides)

Welcome to the hands-on session dedicated on fitting in ROOT

Exercise 15: Gaussian fit of an histogram

We will start with an exercise where we fin an histogram with a simple function, to get familiar with the fitting options in ROOT.

- Start creating an histogram with 50 bins in [-5,5] and fill with 1000 Gaussian distributed number
- Fit the histogram using a Gaussian function
- Get the value and error of the width of the gaussian
- Retrieve the fit result and print the correlation matrix.

Hint Hide
To solve the exercise you need first to create first the TF1 object, either using the pre-defined gaus function or by using a formula expression with `"[0]*ROOT::Math::normal_pdf(x,[2],[1])"`. Remember that in the second case you need to set the initial function parameters (e.g. `f1->SetParameters(1,0,1)`).

To get access to the `TFitResult` object after fitting use option "S", as shown in slide 10 of the lecture.

Use `gStyle->SetOptFit(1)` to display the fit result in the statistics box.

Solution Hide

```
#include "TF1.h"
#include "TH1.h"
#include "TFitResult.h"
#include "TMatrixDSym.h"
#include "TStyle.h"

void gausFit() {

   TH1D * h1 = new TH1D("h1","h1",50,-5,5);

   h1->FillRandom("gaus",1000);

   TF1 * f1 = new TF1("f1","gaus");

   // add also option "Q" (quite) to avoid prrinting two time the result
   TFitResultPtr r = h1->Fit(f1,"S Q");

   // print the result
   r->Print();

   // get the correlation matrix and print it
   TMatrixDSym corrMatrix = r->GetCorrelationMatrix();
   corrMatrix.Print();

   gStyle->SetOptFit(1);

  // to get the sigma of the gaussian
  std::cout  << "Gaussian sigma = " << f1->GetParameter("sigma") << "  +/- " << f1->GetParError(f

}
```

If you repeat the fit few times (without exiting ROOT) you will see that the sigma you obtain is almost always less than 1. The result is then slightly bias. Try to perform a likelihood fit (option "L"). Is the result better ?

Solution Hide
The reason is that the lest square fit is not correct in case of low statistics. The bins with zero events are not included in the fit and this bias the result. The likelihood fit is the correct method for fitting count histograms.

You can notice that the sigma and the amplitude of the gaussian are quite correlated. Can you have a better parametrisation ?

Solution Hide
You can fit using a normalised Gaussian. In this case you get a much smaller correlation. To do this create the TF1 as following:

```
TF1 * f1 = new TF1("f1","[0]*ROOT::Math::normal_pdf(x,[2],[1])");
// set the parameters (needed if not using a pre-defined function as "gauss")
 f1->SetParameters(1,0,1);
```

Exercise 16: Fit a peak histogram

We are going to fit the histogram with a more complicated function. We can use the histogram obtained from the CMS tree data of Exercise 10. The aim is to compute the mass and width of the peak (in this case the J/Psi).

- Create (or read from the ROOT file obtained in Exercise 10) the tree with the dimuon CMS data between 2 and 5 GeV (text data file).
- Fill an histogram with 60 bins between 2 and 5 with the invariant mass for the events when the two muons have opposite charge.
- Create a function composed of the gaussian plus the exponential and fit to the histogram. Do the fit works ? What do you need to do to make the fit working ?
- Compute the number of peak events, by using the integral of the Gaussian function. Use `TF1::IntegralError` to compute also its error.

Hint Hide
See Exercise 10 on how to create the tree. To fill and draw the histogram do for example:

```
tree->Draw("M >> h1(60,2,5)","Q1*Q2==-1")
```

Before fitting you need to set sensible parameter values. You can do this by fitting first a single gaussian in the range [2.7,3.2] and then the exponential separately. If you don't set good initial parameter values, the fit will probably not converge.

After the fit works, you can compute the number of peak events, by using `TF1::Integral` on the Gaussian only function. For the error you can use `TF1::IntegralError`, but you need to extract the correlation matrix from the fit and use the sub-matrix referring to the gaussian part.

Solution Hide

```
#include "TF1.h"
#include "TH1.h"
#include "TFitResult.h"
#include "TMatrixDSym.h"
#include "TRandom.h"
#include "TStyle.h"

void JPsiPeakFit() {


   TTree * t = new TTree("tree","tree");
   // define first the branch
   TString branchDescription("Type/C:RunNo/I:EvNo/I:E1/F:px1:py1:pz1:pt1:eta1:phi1:Q1/I:E2/F:px2:

   tree->ReadFile("dimuon_2-5Gev.csv",branchDescription,',');



   TH1D * h1 = new TH1D("h1","h1",60,2,5);
```

```
   // to draw on the h1 histogram
   t->Draw("M >> h1");



   // fit first a single gaussian in range [0,5]
   TFitResultPtr r1 = h1->Fit("gaus","S","",2.7,3.3);  // first fit of gaussian
   TFitResultPtr r2 = h1->Fit("expo","S");  // first exponential

   TF1 * f1 = new TF1("fitFunc","gaus(0)+expo(3)");
   // get parameters and set in global TF1

   // parameters of first gaussian
   f1->SetParameter(0,r1->Parameter(0));
   f1->SetParameter(1,r1->Parameter(1));
   f1->SetParameter(2,r1->Parameter(2));
   // parameters of second gaussian
   f1->SetParameter(3,r2->Parameter(0));
   f1->SetParameter(4,r2->Parameter(1));



   TFitResultPtr res = h1->Fit(f1,"SL");

   TMatrixDSym cov = res->GetCovarianceMatrix();

   // compute number of signal events
   // fitted gaussian function
   TF1 * peakFunc = new TF1("peakFunc","gaus");
   peakFunc->SetParameter(0, f1->GetParameter(0));
   peakFunc->SetParameter(1, f1->GetParameter(1));
   peakFunc->SetParameter(2, f1->GetParameter(2));

   TMatrixDSym covPeak = cov.GetSub(0,2,0,2);

   double nsignal = peakFunc->Integral(0,10) / h1->GetBinWidth(1);
   double err  = peakFunc->IntegralError(0,10,peakFunc->GetParameters(), covPeak.GetMatrixArray()

   std::cout << "number of signal events = " << nsignal << " +/- " << err << std::endl;




}
```

Exercise 17: Using the Fit Panel GUI

Repeat the previous exercise, but by using the Fit Panel GUI

Hint Hide

- Select the fit panel in the Canvas/Tools menu or by right-clicking on the histogram
- Make the fit function (gaussian plus exponential) using the Operation/Add button
- Set the initial parameters by playing with the sliders
- Press the Fit button

Explore the other functionalities of the fit panel like changing the fit method, use a different minimiser, plotting contours in the parameters.

If you want to know more about fitting, you can look at some of the tutorials in the ROOT distribution directory, `$ROOTSYS/tutorials/fit`. They are available on the Web at this location. For example look at:

- FittingDemo.C
- NumericalMinimization.C
- ErrorIntegral.C
- fitMultiGraph.C

Fitting using RooFit (slides)

Welcome to the hands-on session dedicated on fitting using RooFit. The aim is to start familiarizing with RooFit and trying understand the basic syntax of creating models using the workspace factory. We will also see how to save a workspace in a ROOT file which, allowing to perform the fitting analysis at a later stage or to share the models with other people.

RooFit provides also a separate User Guide (unfortunately it does not cover yet the workspace factory syntax), but it exists also a reduced booklet, showing the main functionality, which you can find here.

Exercise 18: Gaussian model and fit it to random generated data

We will start with a similar exercise we did for Root fitting. We will create a Gaussian model from which we will generate a pseudo-data set and then we will fit this data set.

Start directly creating the Gaussian model using the workspace factory, thus the syntax introduced in the lecture slide 15. One you have created the model, use the `generate()` method of the `RooAbsPdf` class to generate 1000 events. Try to plot the data set using `RooPlot` as shown in slide 12.

After, fit the model to the data and show the resulting fitted function as in slide 14.

At the end save the `RooWorkspace` object in a file, but before remember to import, by calling `RooWorkspace::import(data)`, the data set you have generated in the workspace. The workspace does not contains only the model, but also the data, allowing then to re-perform the analysis later on.

Hint Hide

- Use the syntax of the `RooWorkspace` factory to create first the variables (observables and parameters) of the Gaussian probability density function (p.d.f.), as shown in slide 15.

Every variable in RooFIt, when created needs to be created with a value a min and a max allowed range. Use very large value if you don't know the range. If you provided only a value, the variable is considered constant. If you provide only the minimum and the maximum, the initial value will be taken as half the range. To avoid undesired side effect the value given should be defined between the min and max of the given range.

- You need to define the [value, min, max] of a variable only the first time you create in the factory. Afterwards you can reference the variable by its name.

See also slide 21 to create the p.d.f.

- After you have created the variable and p.d.f in the workspace, you can access their pointers, by using `RooWorkspace::var` for variables or `=RooWorkspace::pdf` for p.d.f.
- You can then do as in slide 12 to generate the data set and plot it.

- To fit the data set with the pdf, you need to call the `RooAbsPdf::fitTo`. See slide 14 on how to fit and on how to plot the function after the fit.
- To save the model call, `RooWorkspace::write(file_name)`. See slide 19.

Below is the code solution:

Solution Hide

```
// make a simple Gaussian model

#include "RooWorkspace.h"
#include "RooRealVar.h"
#include "RooAbsPdf.h"
#include "RooDataSet.h"
#include "RooPlot.h"

#include "TCanvas.h"

using namespace RooFit;

void GaussianModel(int n = 1000) {


   RooWorkspace w("w");
   // define a Gaussian pdf
   w.factory("Gaussian:pdf(x[-10,10],mu[1,-1000,1000],s[2,0,1000])");

   RooAbsPdf * pdf = w.pdf("pdf");   // access object from workspace
   RooRealVar * x = w.var("x");   // access object from workspace


   // generate n gaussian measurement for a Gaussian(x, 1, 1);
   RooDataSet * data = pdf->generate( *x, n);

   data->SetName("data");

   // RooFit plotting capabilities
   RooPlot * pl = x->frame();
   data->plotOn(pl);
   pl->Draw();


   // remove this line if you want to fit the data
   return;

   // now fit the data set
   pdf->fitTo(*data);


   // plot the pdf on the same RooPlot object we have plotted the data
   pdf->plotOn(pl);

   pl->Draw();

   // import data in workspace (IMPORTANT for saving it )
   w.import(*data);

   w.Print();

   // write workspace in the file (recreate file if already existing)
   w.writeToFile("GaussianModel.root", true);

   cout << "model written to file " << endl;

}
```

Exercise 19: Reading a workspace from a file

Open the file you have just created in the previous exercise and get the `RooWorkspace` object ffrom the file. Get a pointer to the p.d.f describing your model, and a pointer to the data. Re-fit the data, but this time in the range [0,10] and plot the result.

Hint Hide
To read and analyse the workspace you need to do:

- Open the `TFile` object and use `TFile::Get` to get a pointer to the workspace using its name
- Once you have the workspace in memory, retrieve from it the p.d.f. and the data set with their names using `RooWorkspace::pdf` and `RooWorkspace::data`.
- Re-issue again the call to `RooAbsPdf::fitTo`. You can set the fit range using the `RooFit::Range(xmim,xmas)` command arg option in `fitTo`. See the reference documentation for all the possible options that you can pass (some are shown in the solution code).

Here is the solution. This macro (apart form the Range fit) can work and fit whatever workspace you have in the file. You just need to set the right names for file, workspace, global p.d.f. and data set.

Solution Hide

```
#include "RooWorkspace.h"
#include "RooAbsPdf.h"
#include "RooRealVar.h"
#include "RooPlot.h"
#include "RooDataSet.h"
#include "RooFitResult.h"

#include "TFile.h"


// roofit tutorial showing how to fit whatever model we get from a file

// we assume the name of the workspace is w
// the name of the pdf is pdf
// the name of the data is data
const char * workspaceName = "w";
const char * pdfName = "pdf";
const char * dataName = "data";


using namespace RooFit;



void fitModel(const char * filename = "GaussianModel.root" ) {

   // read file:
   // following lines are for reading workspace
   // and to check that is fine

   // Check if example input file exists
   TFile *file = TFile::Open(filename);

   // if input file was specified but not found, quit
   if(!file ){
      cout <<"file " << filename << " not found" << endl;
      return;
   }

   // get the workspace out of the file
   RooWorkspace* w = (RooWorkspace*) file->Get(workspaceName);
```

```
   if(!w){
      cout <<"workspace with name " << workspaceName << " not found" << endl;
      return;
   }

   // fit a pdf from workspace with name pdfName

   RooAbsPdf * pdf = w->pdf(pdfName);
   if (!pdf) {
      w->Print();
      cout << "pdf with name " << pdfName << " does not exist in workspace " << endl;
      return;
   }

   // get the data out of the file
   RooAbsData* data = w->data(dataName);

   if(!data ){
      w->Print();
      cout << "data " << dataName << " was not found" <<endl;
      return;
   }

   //------------------------------------------------------------------------
   //// real code starts here

   // get variable x (is the first of the data)
   RooRealVar * x = w->var("x");
   RooPlot * plot = x->frame();

   data->plotOn(plot);
   plot->Draw();

   // fit pdf - (example using option: save result and using diffferent minimizer

   // global fit

   pdf->fitTo( *data );

   // for doing a reduce fit in a Range (plus other options)

   RooFitResult * r = pdf->fitTo( *data, Save(true), Minimizer("Minuit2","Migrad"),Range(0.,10.)

   pdf->plotOn(plot);
   pdf->paramOn(plot);

   plot->Draw();

   // if we have a result we can do

   r->Print();

   r->correlationMatrix().Print();


}
```

Exercise 20: Fit of a Signal over an Exponential Background

The aim of tis exercise is to learn how to build a composite model in RooFit made of two p.d.f, one representing a signal and one a background distributions. We want to determine the number of signal events. For this we need to perform an extended maximum likelihood fit, where the signal events is one of the fit parameter. We will create the composite model formed by a Gaussian signal over a falling exponential background. We will have ~ 100 Gaussian-distributed signal event over ~ 1000 exponential-distributed

background. We assume that the location and width of the signal are known. The "mass" observable is distributed in range [0 - 10] and the signal has mean ~ 2 and sigma 0.3. The exponential decay of the background is 2.

We will follow the same step as in the previous exercise:

- Start creating the model using the workspace factory. Create the separate signal and background pdf and then combine together using the `RooAddPdf` class.
- Use the `generate(...)` method of the `RooAbsPdf` class to generate a data set with 1000 events.
- Plot the data set using `RooPlot` in a ROOT Canvas.
- Fit the generate data using the `fitTo(...)` method of the `RooAbsPdf` class
- Plot the resulting fit function on top of the data
- Save the workspace in a file

Hint Hide
Creating the model

- Use the syntax of the `RooWorkspace` factory to create first the Gaussian p.d.f. function with the variables (observables and parameters) and the Exponential
- Create then a `RooAddPdf` using the Gaussian and the Exponential and the relative number of events. Note that we could instead of the number of events a relative fraction. In this last case we would fit only for the shape and not the normalisation.
- The `RooAddPdf` is created using the `SUM` operator of the factory syntax.

```
// create model
RooWorkspace w("w");
 w.factory("Exponential:bkg_pdf(x[0,10], a[-0.5,-2,-0.2])");
 w.factory("Gaussian:sig_pdf(x, mass[2], sigma[0.3])");

// create RooAddPdf
 w.factory("SUM:model(nsig[100,0,10000]*sig_pdf, nbkg[1000,0,10000]*bkg_pdf)");  // for extended
```

Retrieving objects from the workspace

- After you have created the variable and p.d.f in the workspace, you can access their pointers, by using `RooWorkspace::var` for variables or `RooWorkspace::pdf` for p.d.f. Example:

```
// retrieving model components
RooAbsPdf * pdf = w.pdf("pdf");   // access object from workspace
RooRealVar * x = w.var("x");   // access object from workspace
```

Generating a dataset

- We generate the dataset using the `generate` function passing the observables we want to generate and the number of events. In case of an extended pdf (as the `RooAddPdf` we have created), if we don't pass the number of events, the number of events will be generated according to a Poisson distribution given the expected events (`nsig+nbkg`).
- Note that you can generate also binned data sets. In that case you have two options:
  - ◊ use a `RooDataHist` and `generateBinned`: `RooDataHist * binData = pdf->generateBinned(*x, 1000);`.
  - ◊ use the option `AllBinned()` which will generate a weighted data set: `RooDataSet * data = pdf->generate(*x, NumEvents(1000), AllBinned());`.

```
// generate an unbinned dataset of 1000 events
RooDataSet * data = pdf->generate( *x, 1000);
```

Plot the data:

```
// create a RooPlot object and plot the data
RooPlot * plot = x->frame();
data->plotOn(plot);
```

Fit the data: you need to call the `RooAbsPdf::fitTo`.

```
// fit the data and save its result. Use the optional =Minuit2= minimiser
RooFitResult * res = pdf->fitTo(*data, Minimizer("Minuit2","Migrad"), Save(true) );
```

- Plot the resulting fit function in the same plot.
- Plot also the signal and background fit components with different colour and style

```
pdf->plotOn(plot);
//draw the two separate pdf's
pdf->plotOn(plot, RooFit::Components("bkg_pdf"), RooFit::LineStyle(kDashed) );
pdf->plotOn(plot, RooFit::Components("sig_pdf"), RooFit::LineColor(kRed), RooFit::LineStyle(kDash
plot->Draw();   // to show the RooPlot in the current ROOT Canvas
```

Save the workspace in a file for re-using it without the need to generate the data

```
w.writeToFile("GausExpModel.root",true);  // true means the file is re-created
```

Solution Hide
Note that in the solution we also create a `ModelConfig` object that we save in the workspace. This will be useful for the RooStats exercises (see later the RooStats exercise 1)

```
<span class='twikiAlert'>
     Error: File attachment at https://twiki.cern.ch/twiki/pub/Main/ROOTLaPlataTutorial/GausExpMo
</span>
```

- Gaussian plus Exponential fit result:

Error: (3) can't find GausExpModelFit.png in Main

Exercise 21: Compute the significance of the signal peak using RooStats

The aim of this exercise is to compute the significance of observing a signal in the Gaussian plus Exponential model. It is better to re-generate the model using a lower value for the number of signal events (e.g. 50), in order not to have a too high significance, which will then be difficult to estimate if we use pseudo-experiments. To estimate the significance, we need to perform an hypothesis test. We want to disprove the null model, i.e the background only model against the alternate model, the background plus the signal. In RooStats, we do this by defining two two `ModelConfig` objects, one for the null model (the background only model in this case) and one for the alternate model (the signal plus background).

We have defined in the workspace saved in the ROOT file only the signal plus background model. We can create the background model from the signal plus background model by setting the value of `nsig=0`. We do this by cloning the S+B model and by defining a snapshot in the `ModelConfig` for `nsig` with a different value:

```
ModelConfig*  sbModel = (RooStats::ModelConfig*) w->obj(modelConfigName);
RooRealVar* poi = (RooRealVar*) sbModel->GetParametersOfInterest()->first();
// define the S+B snapshot (this is used for computing the expected significance)
poi->setVal(50);
sbModel->SetSnapshot(*poi);
// create B only model
ModelConfig * bModel = (ModelConfig*) sbModel->Clone();
bModel->SetName("B_only_model");
poi->setVal(0);
```

```
bModel->SetSnapshot( *poi  );
```

Using the given models plus the observed data set, we can create a RooStats hypothesis test calculator to compute the significance. We have the choice between

- Frequentist calculator
- Hybrid Calculator
- Asymptotic calculator

The first two require generate pseudo-experiment, while the Asymptotic calculator, requires only a fit to the two models. For the Frequentist and the Hybrid calculators we need to choose also the test statistics. We have various choices in RooStats. Since it is faster, we will use for the exercise the Asymptotic calculator. The Asymptotic calculator it is based on the Profile Likelihood test statistics, the one used at LHC (see slides for its definition). For testing the significance, we must use the one-side test statistic (we need to configure this explicitly in the class). Summarising, we will do:

- create the `AsymptoticCalculator` class using the two models and the data set;
- run the test of hypothesis using the `GetHypoTest` function.
- Look at the result obtained as a `HypoTestResult` object

Hint Hide

```
AsymptoticCalculator   ac(*data, *sbModel, *bModel);
ac.SetOneSidedDiscovery(true);  // for one-side discovery test

// this run the hypothesis test and print the result
HypoTestResult * result = ac.GetHypoTest();
result->Print();
```

Asymptotic calculator Solution Hide

```
<span class='twikiAlert'>
     Error: File attachment at https://twiki.cern.ch/twiki/pub/Main/ROOTLaPlataTutorial/SimpleHyp
</span>
```

- Result of the significance test on the Gaussian Signal plus Background model (nsig=50, nbckg=1000):

```
Results HypoTestAsymptotic_result:
 - Null p-value = 0.00118838
 - Significance = 3.0386
 - CL_b: 0.00118838
 - CL_s+b: 0.502432
 - CL_s: 422.787
```

Run the Frequentist calculator

Hint Hide
For the Frequentist calculator we need to :

- set the test statistic we want to use
- set the number of toys for the null and alternate model

```
FrequentistCalculator   fc(*data, *sbModel, *bModel);
fc.SetToys(2000,500);    // 2000 for null (B) and 500 for alt (S+B)

// create the test statistics
ProfileLikelihoodTestStat profll(*sbModel->GetPdf());
// use one-sided profile likelihood
profll.SetOneSidedDiscovery(true);
```

```
// configure  ToyMCSampler and set the test statistics
ToyMCSampler *toymcs = (ToyMCSampler*)fc.GetTestStatSampler();
toymcs->SetTestStatistic(&profll);

if (!sbModel->GetPdf()->canBeExtended())
   toymcs->SetNEventsPerToy(1);
```

We run now the hypothesis test

```
HypoTestResult * fqResult = fc.GetHypoTest();
fqResult->Print();
```

And we can also plot the test statistic distributions obtained from the pseudo-experiments for the two models. We use the `HypTestPlot` class for this.

```
// plot test statistic distributions
HypoTestPlot * plot = new HypoTestPlot(*fqResult);
plot->SetLogYaxis();
plot->Draw();
```

Frequentist calculator Solution Hide
Use the macro SimpleHypoTest.C remove the `return` statement in the middle and run it again

- Result of the significance test on the Gaussian Signal plus Background model (nsig=50, nbckg=1000) using the Frequentist calculator:

```
Results HypoTestCalculator_result:
 - Null p-value = 0.001 +/- 0.000706753
 - Significance = 3.09023 +/- 0.2099 sigma
 - Number of Alt toys: 500
 - Number of Null toys: 2000
 - Test statistic evaluated on data: 4.61654
 - CL_b: 0.001 +/- 0.000706753
 - CL_s+b: 0.496 +/- 0.02236
 - CL_s: 496 +/- 351.262
```

Error: (3) can't find FrequentistHTResult.png in Main

Exercise 22: Compute significance (p-value) as function of the signal mass

As an optional exercise to lear more about computing significance in RooStats, we will study the significance (or equivalent the p-value) we obtain in our Gaussian signal plus exponential background model as function of the signal mass hypothesis. For doing this we perform the test of hypothesis, using the AsymptoticCalculator, for several mass points. We can then plot the obtained p-value for the null hypothesis (`p0`). We will also estimate the expected significance for our given S+B model as function of its mass. The expected significance can be obtained using a dedicated function in the `AsymptoticCalculator`: `AsymptoticCalculator::GetExpectedPValues` using as input the observed p-values for the null and the alternate models. See the Asymptotic formulae paper for the details.

Solution Hide

```
<span class='twikiAlert'>
     Error: File attachment at https://twiki.cern.ch/twiki/pub/Main/ROOTLaPlataTutorial/p0Plot.C,
</span>
```

Error: (3) can't find p0Plot.png in Main

Instructions for the Virtual Machine

Accounts

The VM has two users defined: admin, administrator account with sudo access, and user, meant to be used for the tutorial. Passwords are distributed locally in the room.

Setting up CVMFS

Setting up ROOT

Opening ports for PROOF

Acquire the credentials by first logging as 'admin'; if you are logged as 'user' you can do 'ssh admin@0' . Do 'sudo su' to be superuser. Replace the file /etc/sysconfig/iptables with this version of the file: iptables.txt. Restart the service 'service iptables restart'.

-- LorenzoMoneta - 26 Nov 2013

This topic: Main > ROOTLaPlataTutorial
Topic revision: r19 - 2015-11-16 - LorenzoMoneta