Table of Contents

ROOT Tutorials Exercises for Desy C++ School

Topics

This tutorial aims to provide a complement to the lecture shown at the Desy C++ School, November 2013. This tutorial is focus on data analysis in ROOT using Tree's and Proof. It does not cover other features of ROOT such as histogramming or fitting techniques. 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 from the school Agenda page. 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 set of simpler exercises covering also histogramming and fitting for RootGridKaTutorial2013

Introduction

We will focus first on simpler exercises. The solution of the exercise is often to write a running ROOT macro. Two levels of help will be provided: a hint and a full solution. Try not to jump straight 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 school virtual machines which should be 5.34.11.

Setting up ROOT pre-release version 6

Some exercises require some new functionality of ROOT, which will be available only in the new version 6, expected for the end of the year. In order to solve for this exercise and try the new version, you need to set-up a different environment variable for $ROOTSYS.

ROOT I/O and Trees

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 1: 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 2: Creating a ROOT Tree

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 3: 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.

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
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 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). 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"
#include "Math/VectorUtil.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 now to use a `TClonesArray` and/or 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 4: Read a ROOT Tree containing a collection of LorentzVector's

Use `TTree::Draw` to plot:

- The number of tracks per event in an histogram with 50 bins between 0 and 50.
- A profile plot showing the pt of the number of tracks vs the event number.
- A candle plot with the pt of the first 5 tracks when the number of tracks is larger or equal 5.

Use then C++ code to plot the Px distribution of the tracks. Measure the time to read the tree and make the plot and compare to when the tree was generated with splitting or not splitting. Plot also the invariant mass of all 2-tracks combinations. This you cannot do with `TTree;:Draw`

Hint Hide

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

See also TTree::Draw documentation

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 4: Analyze a ROOT Tree using `MakeProxy`, `MakeClass` or `MakeSelector`.

Plot the invariant mass of the 2-track combinations for all tracks with a pt > 3 GeV using a proxy function (see TTree::MakeProxy)/

Hint Hide
Using the proxy function. Create a file proxy.C as following:

```
TH1D *h1;

void proxy_Begin(TTree*) {
  // create here histogram
   h1 =  ..
}

double proxy() {
   // put here user code

   // for example
   int ntracks = tracks->size();
   for (int i = 0; i < ntracks; ++i) {
     double pt = tracks[i].Pt()
     if (pt > something) {

       }
      }
   }
   return 0;
}

void proxy_Terminate() {
   // draw the histogram
  h1->Draw();
}
```

Note that when using a proxy function the tree must be generated in split mode to have access to all information. This is a limitation. Also you cannot use weight/cut expression in TTree::Draw.

Solution Hide

```
TH1D *h1;

void proxy_Begin(TTree*) {
   h1 = new TH1D("h1","inv mass",100,0,100);
}

double proxy() {
   int ntracks = tracks->size();
   for (int i = 0; i < ntracks; ++i) {        for (int j = i+1; j < ntracks; ++j) {          if
            h1->Fill( (tracks[i] + tracks[j]).M() );
      }
   }
   return 0;
}

void proxy_Terminate() {
```

```
  h1->Draw();
}
```

If you have more time generate the skeleton code using `TTree::MakeClass` or `TTree::MakeSelector`. Note that now to have the full access to the vector the tree must be generated in non-split mode. Currently MakeClass/MakeSelector do not work if this particular tree is generated in split mode.

Exercise 5: Creating a ROOT Tree containing an 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. Write the macro creating the tree and filling with them with the events containing the EventData object.

Solution Hide

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

#include "EventData.h"

void CreateEventTree(Long64_t numEvents = 200, int splitlevel = 99) {
   TFile* f = new TFile("eventdata_s99.root", "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. Run the macro to create and write the tree in the file.

Exercise 6: Analyzing a ROOT Tree using for example `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 the future this limitation should be removed. in case of 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.

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 6b: 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 7: Chaining ROOT Trees.

Using the macro to create the EventData tree, we run 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.

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()");
```

Exercise 8: Using Tree Friends

Suppose we have the Tree with the vector of LorentzVector. We want now to add a new branch to this tree containing an `std::vector< std::vector< double> >` which contains the mass invariant of each track with respect all the other tracks. Make a new Tree. Read from the file the Tree used before and add as a friend to this tree. Plot then the invariant mass where at least one of the track has a pT larger than 3 GeV.

Solution Hide
Here is the macro to create a second tree, containing the invariant mass

```
//
// 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 MakeInvMassTree(const char * fileIn = "vectorCollection.root", const char * fileOut = "invMa


  TH1D * hinv = new TH1D("hinv","Track invariant mass",100,0,100);


  TFile f1(fileIn);

  // create tree
  TTree *t1 = (TTree*)f1.Get("t1");

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

  TFile f2(fileOut,"RECREATE");

  TTree *t2 = new TTree("t2","Tree with Inv Mass branch");

  std::vector< std::vector< double> >  invMassVec;
  std::vector< std::vector< double> >  * pInvMassVec;

  t2->Branch("invMass",&pInvMassVec);
  double CUT = 10;

  TStopwatch timer;
  timer.Start();
  int n = (int) t1->GetEntries();
  std::cout << " Tree Entries = " << n << std::endl;    std::vector<double> m;

  for (int i = 0; i < n; ++i) {      t1->GetEntry(i);
    int ntrk = pTracks->size();
    pInvMassVec->clear();
```

```
    pInvMassVec->reserve(ntrk);
    for (int j = 0; j < ntrk; ++j) {        XYZTVector v = (*pTracks)[j];        if (v.Pt() > CUT
         m.clear();
         m.reserve(ntrk-j-1);
         // compute invariant mass
         for (int k = j+1; k < ntrk; ++k) {              XYZTVector q = (*pTracks)[k];
               double minv = (v+q).M();
               hinv->Fill(minv);
               m.push_back(minv);
            }
         }
         pInvMassVec->push_back(m);
      }

    }
    t2->Fill();
  }


  timer.Stop();
  std::cout << " Time to read and write new  Tree " << timer.RealTime() << "  " << timer.CpuTime(
  t2->Print();
  t2->Write();
  f2.Close();
}
```

Here are the lines of codes to Draw the invariant mass You can run these lines from the ROOT prompt.

```
TFile f("vectorCollection.root");    // to get  the first tree
 tree->AddFriend("t2","invMassCollection.root");
tree->Draw("t2.invMass", "@tracks.size()  > 20");
```

Exercise 9: 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

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

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 * 1;
```

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 10: Using PROOF to analyze the TTree

Using the Selector defined in the previous exercise run the selection using PROOF light. Try to run for different cores and see how the rate of processing events scales with the number of workers.

Hint Hide
Modify the `EventDataSelector` file you have created before to add in the output list (`fOutputList` data member) the list of histogram you create. When drawing the histogram, retrieve them from the output list.

Create first a TChain containing the ROOT files you wan to analyze. PROOF works only on TChain. Create then a TProof instance with `TProof::Open()` and then call `TChain::SetProof`. Thats all, now by running `TChain::Process`, PROOF Lite will be used for the analysis. The number of workers can be defined by doing `TProof::Open("nworkers= ")`. Run for different workers and see the processing rate reported in the GUI.

Solution Hide

```
void runProof(int nworkers=8) {

   TChain * chain = new TChain("tree");
   chain->Add("eventdata_s1.root");
   bool on = nworkers>0;
   if (on) {
      //TProof::Open("workers=8");
      TProof::Open(TString::Format("workers=%d",nworkers) );

      chain->SetProof();
   }

   TStopwatch w; w.Start();
   chain->Process("EventDataSelector.C+");
   w.Print();

}
```

If you still have time for the exercise you can fit the generated invariant mass histograms to measure the number of pi0, the K0 or the J/Psi events. You can use directly ROOT fitting or RooFit.

This topic: Main > ROOTDesyTutorial2013
Topic revision: r10 - 2013-11-19 - LorenzoMoneta