COMPUTING METHODS IN HIGH ENERGY PHYSICS

S. Lehti

Helsinki Institute of Physics

Spring 2026

ROOT

Introduction

ROOT is an object-oriented framework aimed at solving data analysis challenges of high energy physics. The commonly used components of ROOT are

- Command line interpreter
- Histogramming and fitting
- Writing a graphical user interface
- 2D graphics
- Input/output
- Collection classes
- Script processor

ROOT web pages:
http://root.cern.ch.

The reference manual is most useful, and when facing a problem, the ROOT talk digest may already contain the answers to your questions. The ROOT developers are also very helpful, and they are gladly answering questions how to use ROOT.

You can start a ROOT session by typing

$ root or: $ root -l

assuming ROOT is installed on your system. This will give you the system prompt

root [0]

You can execute your ROOT script (myScript.C) by typing

ROOT

root [0] .x myScript.C

or directly from the shell

$ root myScript.C

One can exit the system prompt by typing

root [0] .q

It is possible to type CINT commands directly to the system prompt

```
root [0] int i = 0
root [1] cout << i << endl
0
root [2]
```

The command line interpreter CINT looks and feels just like C++, and it does about 95% of what C++ does. Not everything, though.

Some practicalities

Root scripts are text files which contain the commands enclosed within curly brackets:

```
{
  cout << "Hello world!" << endl;
}
```

When executing ROOT files from the command line, functions need to be named as the file. If the file name is abc.cxx, then

```
void abc(){}
```

is the only allowed name for the function. If the function name and file name are different, then one can load the script first

root [] .L abc.cxx

ROOT

and then use the function

root [] bcd()

It is possible to include header files just like in C++. Quite a lot is already available, but if it's not then one can add e.g. for maps
#include <map>

ROOT is known to be sometimes unstable. Therefore is it sometimes better to run scripts with a new ROOT session starting every time instead of using

root [] .x myScript.C++

time after time in the same session.

A very powerful method to create and modify ROOT scripts is to save the current script in a ROOT file, and then copy-paste the relevant section into your script. For example when adding text in a figure, one can add it interactively, save the script in a file, open the saved file and copy the text part in the original file. The original file is good to be available if you have e.g. classes written in it, or if it includes fits. Those files may contain code you need again later. The saved scripts drop all the fancy structures, and sometimes fits dont work in the saved scripts.

Best way to work with ROOT is not scripting alone, or graphical interface alone, but a combination of both.

ROOT

Conventions and types

ROOT uses a set of coding conventions.

- Classes begin with T
- Non-class types end with _t
- Member functions begin with a capital
- Constants begin with k
- Global variables begin with g
- Getters and setters begin with Get and Set

In ROOT there are machine independent types available. For example int may be 16 bytes in one machine, but 32 bytes in another. To ensure the size of your variables, there are predefined types in ROOT:

- Int_t,
- Float_t,
- Double_t,
- Bool_t etc.

You can, however, use also the C++ types int, double etc if you wish.

Global variables

ROOT has a set of global variables that apply to the session. For example the single instance of TROOT is accessible via the global gROOT and hold information relative to the current session. Examples:

```
gROOT->Reset();
```

ROOT

gROOT->LoadMacro("ftions.cxx");

gSystem->Load("libMyEvent.so")

History file

When you do something with command line, you may want to save what you have done. You can use the arrow up and down to browse the used commands. They are recorded in a file $HOME/.root_hist, which contains the last 100 commands. If needed, you can cut and past your commands from that text file.

Histograms

ROOT supports 1-D, 2-D 3-D and profile histograms. There are several histogram classes available. For example the class TH1D is for plotting double precision numbers in a one dimensional histogram. Inheritance is heavily used, which means that the ROOT reference manual is often needed to find suitable member functions.

Histograms are created with constructors:

root [] TH1F* n = new TH1F("Name","Title",10,0,5);

Then filled

root [] n->Fill(value);

The histogram drawing is done with Draw() member function

root [] n->Draw();

ROOT

Drawing option can be used to plot several histograms in the same figure

```
root [] n2->Draw("same");
```

The full list of drawing options can be found in the ROOT users guide.

The histogram classes inherit from the attribute classes, which provide member functions for setting the line, fill, marker and text attributes.

```
root [] n->SetFillColor(4);
root [] n->SetLineWidth(3);
root [] n->SetLineStyle(2);
```

The axis titles can be given by

```
root []
n->GetXaxis()->SetTitle("x");
root []
n->GetYaxis()->SetTitle("y");
```

Histograms can be added, divided and multiplied. They can also be cloned

```
root [] TH1F* m =
(TH1F*)n->Clone();
```

Cloning is useful when there are several identical histograms for different variables. When one histogram binning is changed, all are changed accordingly. For example: histograms for reconstructed mass (background only and signal+background)

Sometimes histograms need rescaling

```
root [] double scale =
1/n->Integral();
root [] n->Scale(scale);
```

ROOT

Canvases

Canvases may be seen as windows. In ROOT a graphical entity that contains graphical objects is called a Pad. A Canvas is a special kind of Pad. A pad can contain e.g. histograms.

Here is a simple example to draw a histogram, assuming the histogram is already created and filled.

```
 TCanvas* c = new
TCanvas("c","",500,500);
 c->SetFillColor(0);
 histo->Draw();
 c->Print("mygraph.C");
```

Pads can contain pads. Canvas can be divided into subpads

```
canvas->Divide(2,1);
canvas->cd(1);
histo1->Draw();
canvas->cd(2);
histo2->Draw();
```

The Divide member function creates two parallel figures one containing histo1 and the other containing histo2. The arguments determine the pad matrix, e.g.

```
canvas->Divide(2,3);
```

will create six pads, two in x direction and 3 in y. Navigation between the pads is done with member function cd(padnumber);

The log/linear scale is switched on/off using pads, not histograms. In the current pad the y scale is changed to log scale by

```
gPad->SetLogy();
```

Graphs

A graph is a graphics object made of two arrays holding the coordinates of n points. The coordinates can be arrays of doubles or floats.

```
int N = 3;
double x[N] = {1,2,3},
       y[N] = {1,2,3};
TGraph* g = new TGraph(N,x,y);
TCanvas* c = new
TCanvas("c","",300,300);
c->cd();
hframe = new
TH2F("hframe","",2,0,4,2,0,4);
hframe->Draw();
g->Draw("C");
```

Here the draw area is determined with a 2D histogram hframe. If automatic draw area is enough, one can also use the draw option "A" for including the axis drawing. Other draw options are e.g. "C" for continuous and "P" for polymarkers. If polymarkers are used, the default marker is a tiny dot, so to get the graph visible, one should change the marker/marker size. Draw options can be combined , e.g "CP" will produce both continuous line and polymarkers.

Several graphs can be superimposed in the same figure by leaving out the axis option "A".

Errors can be added using class TGraphErrors. The errors are given

in error arrays for x and y errors.

Trees

Trees are used to store large quantities of data for later analysis. In HEP the event reconstruction takes usually quite some time, and if the analysis is done within the reconstruction program, we need to run the time consuming program every time we change our analysis. Trees (ntuples) is a way to solve this problem. The reconstructed object can be stored in a tree, saved in a file and then later analysed separately. That way the reconstruction is done only once, and when changing e.g. cuts in the analysis all we have to do is to read the ROOT files again.

ROOT manual Example 1:

Writing a Tree is done in three parts, Tree object creation, filling, and writing. Tree creation is done using a constructor

```
TTree t1("t1","simple tree");
```

Once the tree is created, we need to select the variables to be stored. In the example they are px, py, pz and ev. Each variable is linked to a branch, and when creating the branch the stored object address is needed:

```
Float_t px;
t1.Branch("px",&px,"px/F");
```

Here the branch is called px, with value address &px, and leaf list and the type of each leaf. Here we have

ROOT

only one leaf of type Float_t.

Filling the tree is done by giving the variable(s)-to-be-filled a value, and then calling

```
t1.Fill();
```

This will store the current value of px, py, pz and ev into the tree. Once all the values are filled, the tree can be written in a file by calling the member function TTree::Write().

A tree is read from the root file by TFile::Get member function. Let's assume we have a ROOT file open and it contains a tree called t1.

```
TTree *t1 =
(TTree*)gFile->Get("t1");
```

The branch addresses must be set

```
Float_t px;
t1->SetBranchAddress("px",&px);
```

The number of entries in a tree is returned with member function TTree::GetEntries()

```
int entries = t1->GetEntries();
for( int i = 0; i < entries; i++){
  t1->GetEntry(i);
  cout << px << endl;
```

Here the stored variable was Float_t. One can also store ADS's in root trees using the same method.

Fitting

To fit a histogram, one can use the member function TH1::Fit. For example to fit a gaussian,

```
histo->Fit("gaus");
```

ROOT

Often more complicated functions are needed. It is possible to create a custom formula, which is used as the fit function

```
TF1* f = new
TF1("f","[0]*x+[1]",0,1);
histo->Fit("f");
```

Here [0] and [1] are the parameters, which will be varied to get the best fit.

The third way is to define a function which is of the form

```
Double_t name(Double_t* x,
Double_t* par){}
```

The two parameters x and par contain a pointer to the variable array, and a pointer to the parameters array.

Example:

```
Double_t func1(Double_t *x,Double_t *par){
  return
par[0]/sqrt(2*TMath::Pi()*par[2]*par[2]) *
        exp(-0.5*(x[0]-par[1])*(x[0]-
par[1])/(par[2]*par[2]));
}
Double_t func2(Double_t *x,Double_t *par){
  return
par[0]*TMath::Landau(x[0],par[1],par[2])/par[2];
}
Double_t fitFunction(Double_t *x,Double_t
*par){
  return func1(x,par) + func2(x,&par[3]);
}

TF1* theFit = new TF1("theFit", fitFunction,
    fitRangeMin, fitRangeMax, nPar);

histo->Fit("theFit","R");
```

Often to get the fit to converge, one needs to have an educated guess: to set the parameters to some initial values not too far from the expected results. This can be achieved with

ROOT

```
theFit->SetParameter(0,1.234);
```

which sets the initial value of the parameter par[0] to 1.234.
Parameters can also be fixed to a constant value so that they are not varied in the fitting procedure

```
theFit->FixParameter(0,1.234);
```