ROOT 3 / 4

International School of
Theory & Analysis
in Particle Physics

Istanbul, Turkey
31st – 11th February 2011

Jörg Stelzer
Michigan State University, East Lansing, USA

Outline

- Functions
- Histograms and fitting
- Converting macros to compile-able code
- Designing your class for ROOT based analysis
- Classes and ROOT I/O

Before we start …

It this talk examples have the following color coding

file with source code

file.C

```
// usually source code
...
```

All files can be found at http://cern.ch/stelzer/root/

interactive root session

```
TFile *f = TFile::Open("t.root","recreate");
TTree *tr = new TTree("tr","test");
```

The root prompt is omitted so copy & paste is easier. If you have problem with copy 'n paste (document viewer does funny things) then use http://cern.ch/stelzer/root/interactive.txt

shell command

```
shell> root -l -q fitresult.C
```

screen output (sometimes includes the shell command)

```
Processing macro.C+...
Info in <TUnixSystem::ACLiC>: creating shared library /home/stelzer/root_seminar_istapp/workdir/./macro_C.so
/home/stelzer/root_seminar_istapp/workdir/./macro.C: In function 'void macro()':
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: 'TH1' was not declared in this scope
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: 'h' was not declared in this scope
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: expected type-specifier before 'TH1F'
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: expected `;' before 'TH1F'\
```

Working along …

You can work along, trying the examples your self.

To work along create a workdir:

```
you@istapp2011:~$ mkdir workdir
you@istapp2011:~$ cd workdir
you@istapp2011:~/workdir$
```

To get a certain file (e.g. *logon.C*) from my web space use the command wget

```
you@istapp2011:~/workdir$ wget http://cern.ch/stelzer/root/logon.C
```

Also the wget commands are listed by page under
http://stelzer.web.cern.ch/stelzer/root/interactive.txt

But you can also just watch, as I am going to run some of the examples as well.

Let’s do it with style

Define file logon.C with function logon()

logon.C

```
void logon()  {
   gStyle->SetCanvasBorderMode(0);
   gStyle->SetFrameBorderMode(0);
   gStyle->SetPadBorderMode(0);
   gStyle->SetCanvasColor(0);
   gStyle->SetPadColor(0);
   gStyle->SetTitleFillColor(0);
   gStyle->SetStatColor(0);
   gStyle->SetFillStyle(0);

   gStyle->SetStatW(0.3);
   gStyle->SetStatH(0.25);
}
```

The idea of this is to have a consistent appearance of all your work.

You can define multiple TStyles and switch between them.

Personally I prefer plain white and no visible borders.

And declare as default file to run at startup:

Also we want to keep a local history of what we type in .root_hist

.rootrc

```
Rint.Logon:                  logon.C
Root.History:                ./.root_hist
```

Does it work?

Start ROOT and draw a canvas

```
TCanvas c
```

⇒ It's all grey. Ok, close that.

Download logon.C and .rootrc from my web space

```
you@istapp2011:~/workdir$ wget http://cern.ch/stelzer/root/logon.C
you@istapp2011:~/workdir$ wget http://cern.ch/stelzer/root/.rootrc
```

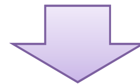
Draw the canvas again.

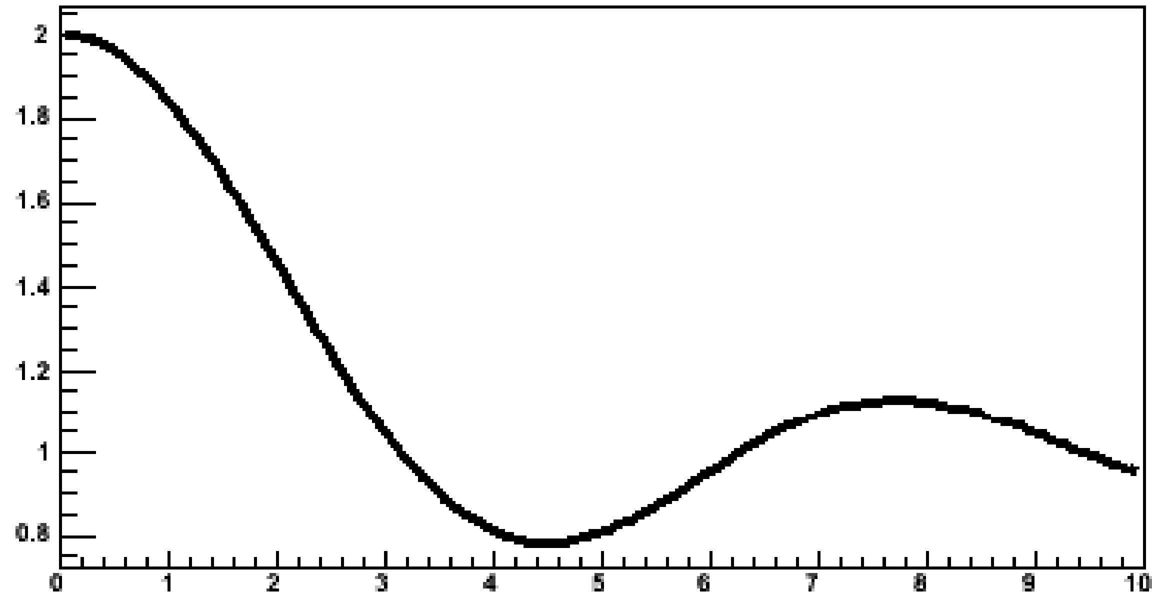
⇒ plain white

Functions in ROOT

Use the class TF1 class to define functions with one variable

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





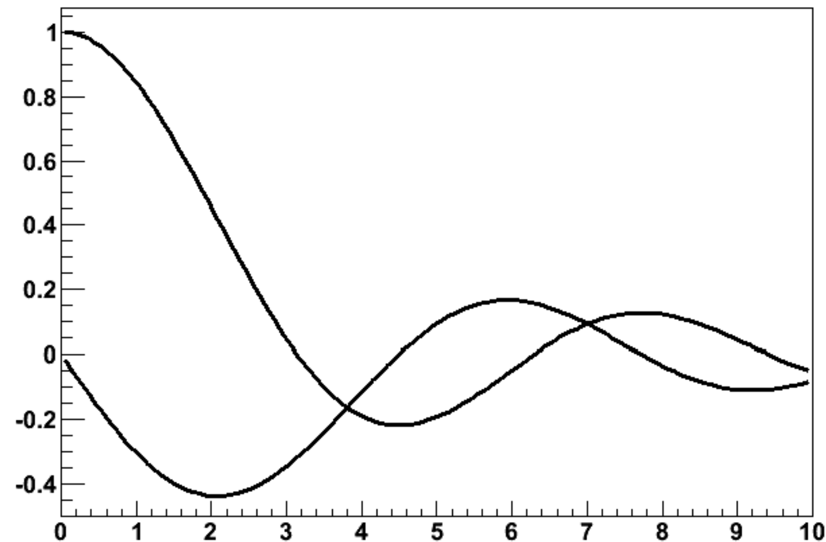
TASK BOX

Try to plot a function yourself !

TF1 can do quite a few things

Evaluate, derivative, integral

```
root [3] f1->Eval(3);
(const Double_t)4.70400026866224020e-002
root [4] f1->Derivative(3);
(const Double_t)(-3.45677499760625890e-001)
root [5] f1->Integral(0,3);
(Double_t)1.84865252799946810e+000
root [6] f1->DrawDerivate("sameL");
```



Doesn't look like this? Try

```
root [7] f1->SetMinimum(-0.5);
```

In root, always the first thing plotted sets the scale

Making nicer looking functions

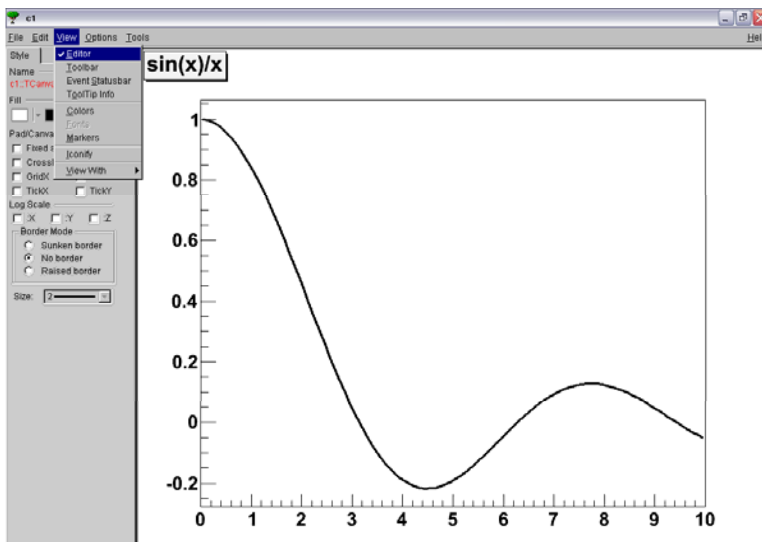
Use the View → Editor

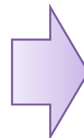
Select object (canvas, title, hist-area, function, axis)

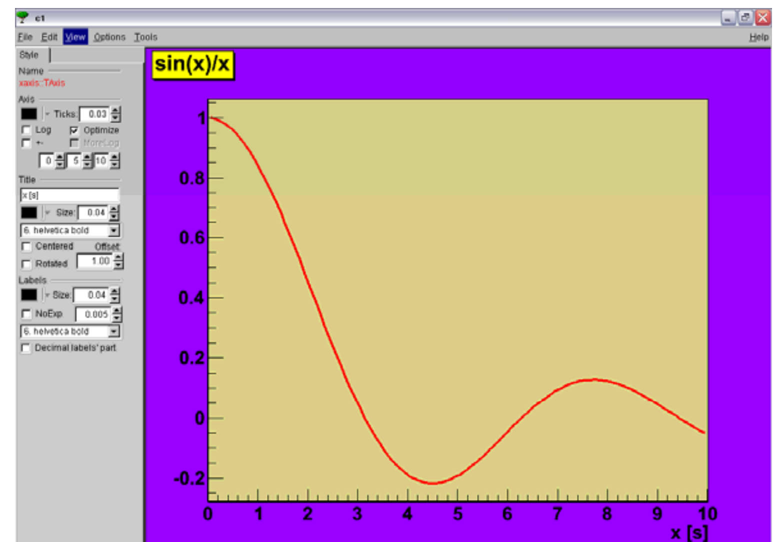
Editor on the left adapts to the selection

Chose colors, styles, etc.

Axis can also get a title







TASK BOX

Try yourself making your function look nice! Give the axis a title. Check out the update and X-Range for the function.

Making nicer looking functions

Programmatic (setting the style from the source code)

```
TF1 *f1 = new TF1("sinc", "sin(x)/x",0,10);
f1->Draw();
f1->SetLineColor(kRed);
f1->GetXaxis()->SetTitle("x[s]");
gPad->SetFillColor(13);
gPad->GetFrame()->SetFillColor(17);
((TPaveText*)gPad->GetListOfPrimitives()->FindObject("title"))->SetFillColor(5);
gPad->Draw();
```

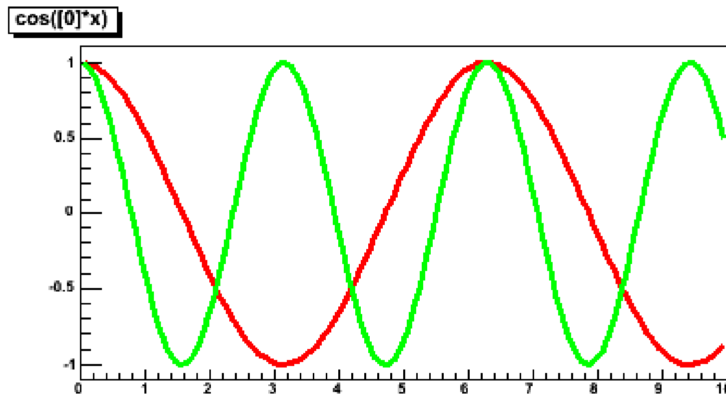
A trick: when you plot something for the first time, change the appearance interactively. Then right click on object , and select either "Dump", or "Inspect", or " SaveAs ->c1.C"
⇒ gives you the properties of the object on the screen or in a window, or even generates source code

For common style use gStyle.

Functions with parameters

A function can have parameters

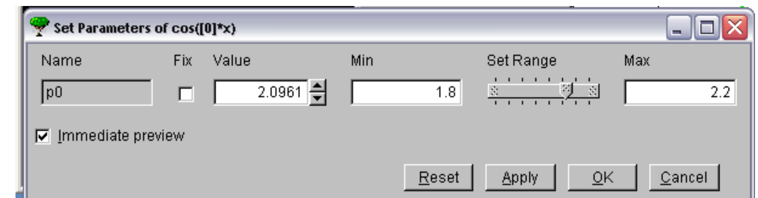
```
TF1 *f1 = new TF1("fcos","cos([0]*x)",0,10);
f1->SetParameter(0,1);
f1->DrawCopy()->SetLineColor(2);
f1->SetParameter(0,2);
f1->DrawCopy("sameL")->SetLineColor(3);
```



Note how the parameters are defined as numbers in brackets (e.g. [0] for the first parameter) !

TASK BOX

Repeat this. Then 'View' → 'Editor' → Select function → 'Set Parameters' → 'immediate preview' !
Watch as you move the slider



Functions synonyms predefined in TFormula

TFormula is the base class for TF1 (TF2, TF3, RooFormula). It takes care of the evaluation of the formula specification, e.g. "sin(x)/x"

These predefined expressions can be used

log	cos	exp	cosh	expo
sq	sin	log10	sinh	gaus
sqrt	tan		tanh	polN
min	acos	abs	asinh	landau
max	asin	sign	acosh	
	atan	int	atanh	fmod
	atan2			rndm

There is a special way to pass on parameters to these function, e.g. for gauss, landau, polN,…:

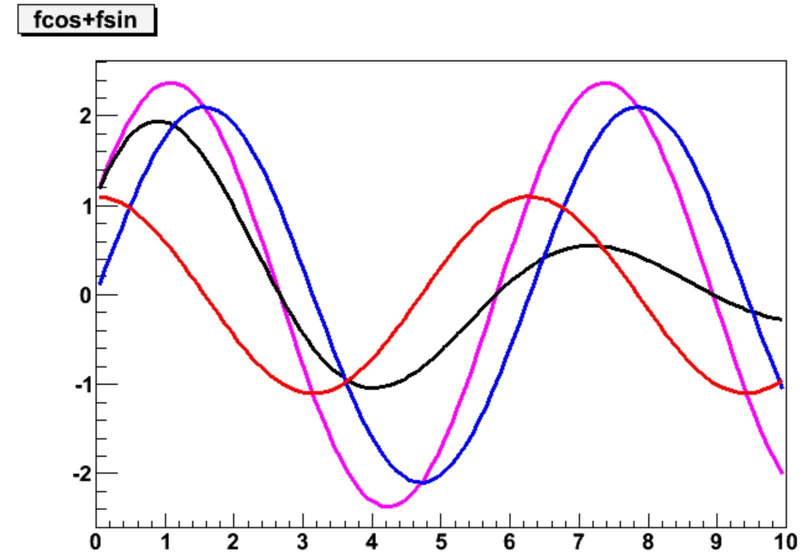
```
TF1("fnc", "[0]*gaus(2)*expo(5)+[1]*pol3(7)*x", 0, 10);
```

translates:
```
gaus(2) → [2]*exp(-0.5*((x-[3])/[4])**2
expo(5) → exp([5]+[6]*x)
pol3(7) → par[7]+par[8]*x+par[9]*x**2+par[10]*x**3
```

Combine your functions

You can define combinations of functions

```
TF1 *fcos = new TF1 ("fcos", "[0]*cos(x)", 0., 10.);
fcos->SetParameter( 0, 1.1);
fcos->SetLineColor(kRed);

TF1 *fsin = new TF1 ("fsin", "[0]*sin(x)", 0., 10.);
fsin->SetParameter( 0, 2.1);
fsin->SetLineColor(kBlue);

TF1 *fsincos = new TF1 ("fsc", "fcos+fsin",0.,10.);
fsincos->SetLineColor(kMagenta);

TF1 *fenv= new TF1 ("fenv", "exp(-x/5.)*fsc",0.,10.);

fsincos->Draw(); fenv->Draw("same");
fsin->Draw("same"); fcos->Draw("same");
```



TASK BOX

Give the plot a better title

```
fsincos->SetTitle("Demo combined functions")
```

Functions predefined in TMath

Before you define your own function, look at what's there!

TMath is part of ROOT that contains definitions of a large number of

- constants: C(), E(), G(), Pi(), PiOver2(), G(), H(), K(), Na(), Qe(), …
- simple functions: Abs(x), Sin(x), Floor(x), Max(x), Sqrt(x),…
- complex functions: Bessel(x), Landau(x,mpv,s,n), BreitWigner(x,m,g), Erf(x), Gaus(x,m,s,n),Poisson(x,l),…
- operations: Factorial(n), GeomMean([]), MaxElement([]), RMS([]), Sort(…),…

TMath has no constructor, all functions are static

```
Double_t bessel = TMath::BesselK(x);
Int_t ax = TMath::Abs(x);
```

Note: ROOT ensures platform compatibility, it is recommended to use the functions in TMath instead of the native ones the c++ distribution. E.g. don't use std::fabs.

Define your own simple function

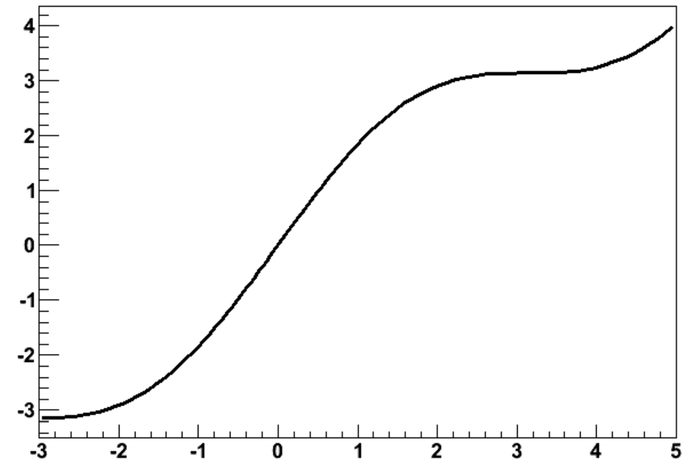
You can write some function *y = myFunc(x)* and turn it into an TF1.

myFunc.C

```
Double_t myFunc(Double_t x) {
  return x+TMath::Sin(x);
}
```

Note: string of func_name(x) !

```
.L myFunc.C

TF1 *f = new TF1("myFunc", "myFunc(x) ", -3, 5);
f->Draw();
```



Definition of function myFunc can not be done interactively ! ⇒ Write a file and load into session !

Three ways to include a file:

```
.L myFunc.C
```

```
gROOT->ProcessLine(".L myFunc.C");
```

```
#include "myFunc.C"
```

gROOT->Processline("**command**") is the same as doing root[]: **command**. But it can be compiled!

Define your own parameterized function

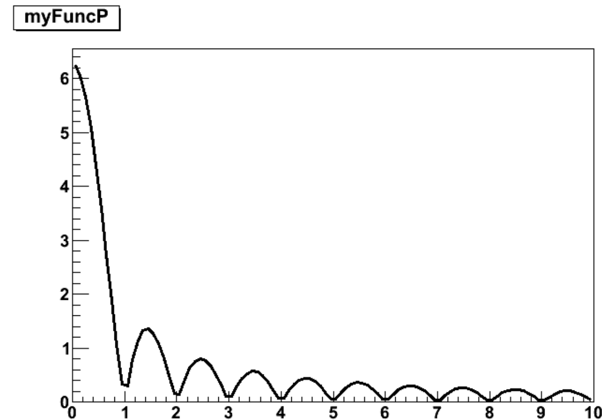
You can write some parameterized function *y = myFunc(x;p)* and turn this into an TF1.

myFuncPar.C

```
Double_t myFuncP(Double_t *x, Double_t *p)
{
  Float_t xx =x[0];
  Double_t f = TMath::Abs(p[0]*sin(p[1]*xx)/xx);
  return f;
}
```

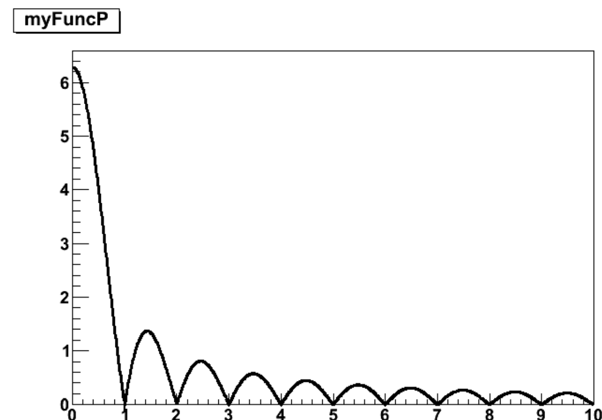
Note: argument is function itself !

```
.L myFuncPar.C
TF1 *f1 = new TF1("myfunc",myFuncP, 0, 10, 2);
f1->SetParameters(2,TMath::Pi());
f1->Draw();
```



Not smooth? Use more points when drawing !

```
f1->SetNpx(1000);
f1->Draw();
```



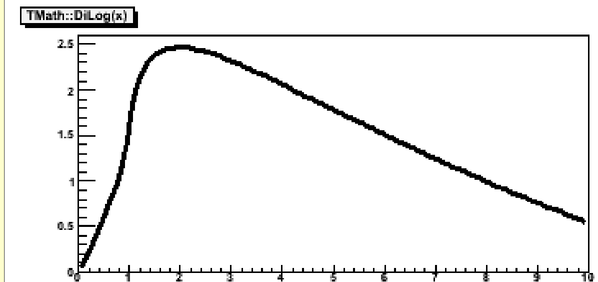
Note: in the definition of myFuncP we used only the first component of the vector x: x[0].
The constructor is like this TF1(name, fnc, xmin, xmax, n_parameter)

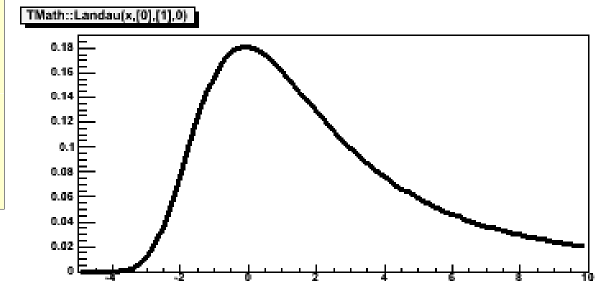
A little function library

It is useful to define the functions that you need repeatedly in a separate file, your function library. ROOT stores all defined functions in a separate list. Then you can use these functions in various fitting tests, distribution samplings, etc.

funclib.C

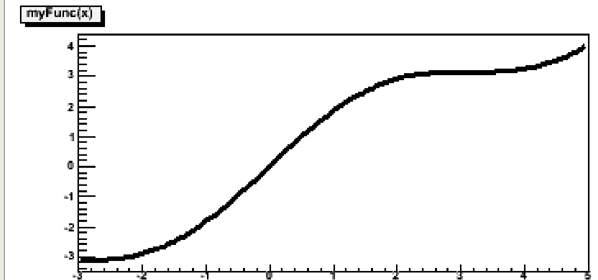
```
Bool_t funclib() {
    TF1 *f1 = new TF1("myDilog", "TMath::DiLog(x)", 0, 10);

    TF1 *f2 = new TF1("myLandau",  "TMath::Landau(x,[0],[1],0)", -5, 10);
    f2->SetParameters(0.2,1.3);
    f2->SetParName(0,"Constant");

    Double_t myFunc(Double_t x) { return x+sin(x); }
    TF1 *f3 = new TF1("myFunc","myFunc(x)", -3, 5);
}
Bool_t a = funclib();
```





Access like this:

```
#include "funclib.C"
TCanvas *c = new TCanvas("c","c",400,600);
c->Divide(1,3);
c->cd(1); gROOT->GetFunction("myDilog")->Draw();
c->cd(2); gROOT->GetFunction("myLandau")->Draw();
c->cd(3); gROOT->GetFunction("myFunc")->Draw();
```

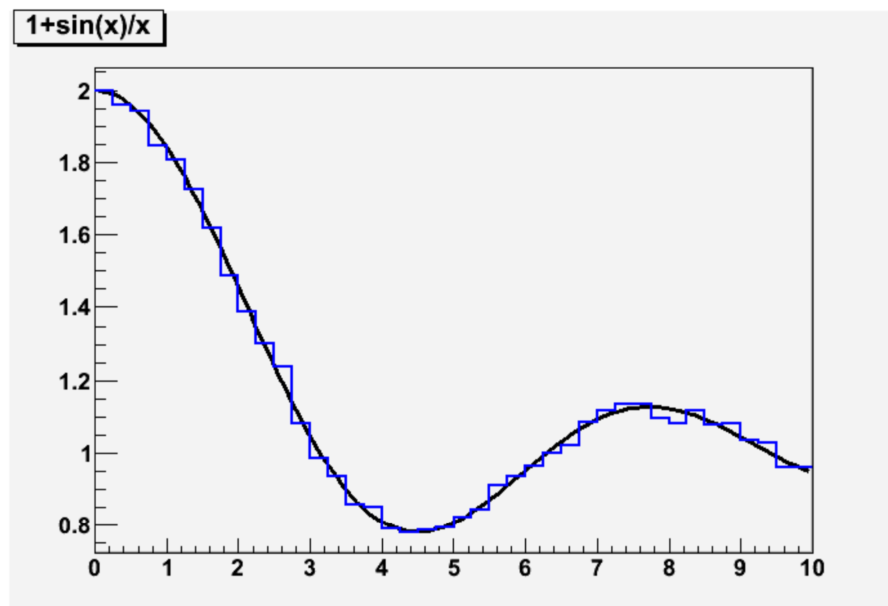


Functions into histograms

A function can be used to fill a histogram. The normalized function describes the probability for a certain bin to get filled.

```
TF1 *f1 = new TF1("sinc", "1+sin(x)/x",0,10);
TH1F *h = new TH1F("hsinc", "test",40,0,10);
h->FillRandom("sinc",200000);
h->Scale(f1->GetMaximum()/h->GetMaximum());
h->SetLineColor(4); h->SetLineWidth(2);
f1->Draw();  h->Draw("same");
```

Note : the histogram has not the same normalization a priori



Try
h->Sumw2();
a) before h->Scale
b) after h-Scale

Sumw2() calculates the error in each bin, which is then correctly scaled. Always calculate error before you scale your histogram!

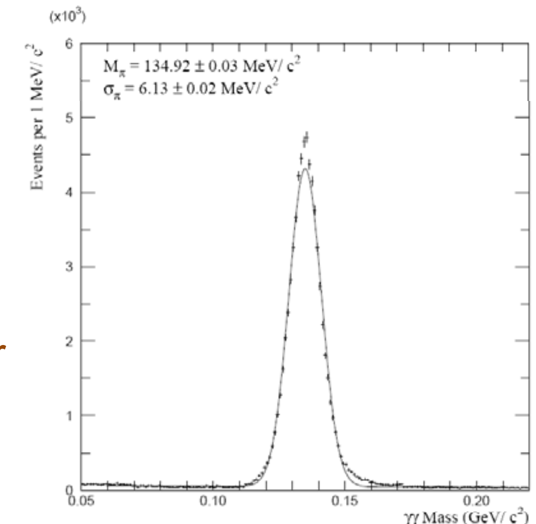
The idea of fitting

- Data sample S={ x_1, x_2, ..., x_N} as the results of repeated measurements of a certain physical property

  E.g. the mass of a π^0 for which we measure the momentum of the two decay photons of $\pi^0 \rightarrow \gamma\gamma$.

- Idea of the underlying physical process, ie. a distribution of x in terms of certain parameters p_i: $f(x;p_i)$

  For instance we know that the π^0-decay produces primarily a Gaussian around the π^0 mass with a resolution that is given by the detector.



So for this case we could

1. put our data sample into a histogram
2. fit a Gaussian function with parameters μ and σ free to the histogram
3. use the fitted μ to calibrate our electromagnetic calorimeter (energy correction for pions) and σ to describe our energy resolution.

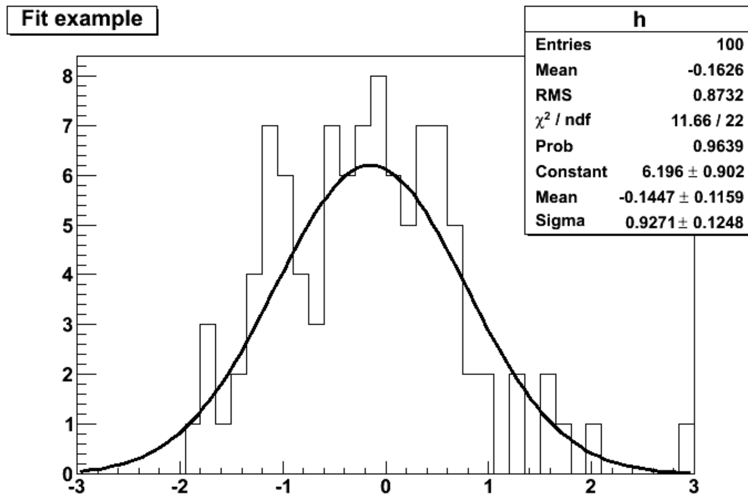
We would also be able to see if the Gaussian is a good way to describe the π^0-mass distribution !

A simple fit example

Define generating function: gen_gaus

Generate distribution (size:100)

Fit selected model: fit_gaus



Stat-box:

entries, mean, RMS of histogram

χ^2: mean quadratic deviation

ndof: 25 filled bins, 3 parameters

Constant, Mean, Sigma: fitted parameters

simplefit.C

```
void generatingFnc() {
  TF1* f=new TF1("gen_gaus", "gaus",-3,3);
  f->SetParameters(1,0,1);
}

TH1* generateSample(const TString& fnc, Int_t n) {
  TH1F *h = new TH1F("h", "Fit example",40,-3,3);
  h->FillRandom(fnc,n);
  return h;
}

TF1* fittingFnc() {
  return new TF1("fit_gaus", "gaus", -3, 3);
}

void simplefit() {
  generatingFnc();
  TH1* h = generateSample("gen_gaus",100);
  h->Draw();

  gStyle->SetOptFit(11111);  // print out of fit in stat box
  h->Fit(fittingFnc());
}
```

Accessing the fit results

Information about the fit appears on the screen:

```
 FCN=22.9704 FROM MIGRAD    STATUS=CONVERGED      60 CALLS          61 TOTAL
                     EDM=1.47586e-13    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  Constant     5.96885e+02   7.47520e+00   1.42856e-02  -1.88776e-08
   2  Mean         8.43250e-03   1.01851e-02   2.43388e-05  -1.54111e-05
   3  Sigma        1.00302e+00   7.75718e-03   4.95861e-06  -2.26585e-04
```

By default all fits are attached to the histogram, so you can access them !

If you need the numbers from the fit for further computation, get them through the fitted function.

```
.x simplefit.C
TF1 * fitfnc = h->GetFunction("fit_gaus");
cout << "Chi-square: " << fitfnc->GetChisquare() << endl;
for(Int_t p=0; p<3; p++) {
  cout << fitfnc->GetParName(p) << ": "
       << fitfnc->GetParameter(p)
       << " +/- " << fitfnc->GetParError(p) << endl;
}
```

```
Chi-square: 22.9704
Constant: 596.885 +/- 7.4752
Mean: 0.0084325 +/- 0.0101851
Sigma: 1.00302 +/- 0.00775718
```

The random seed

Let's try something. Run the fitresult.C a few times from the command-line:

```
shell> root -l -q fitresult.C
```

You will always get the same result (compare the chi-squares).

```
.x fitresult.C
.x fitresult.C
```

This gives you two different results, since the random generators are only initialized at start of root.

Now try with:

```
shell> root -l -q fitresultrdm.C
```

You will always get a different result.

fitresultrdm.C

```
TH1* generateSample(const TString& fnc, Int_t n=200000) {
  TH1F *h = new TH1F("h", "Fit example",40,-3,3);
  gRandom->SetSeed(0);  // based on system time
  h->FillRandom(fnc,n);
  return h;
}
```

The random seed 0 is different every time. Any other number gives you repeatedly the same random sequence.

Decide wisely when you want predictable results and when not !

Generating and fitting – a toy MC

What if we repeat the experiment of generating and fitting data, let's say 2000 times?

The generated distribution should look different each time.

The fitted mean should be different each time.

Lets histogram the fitted mean for each of the 2000 trials …

… and fit a Gaussian at the end

toy.C

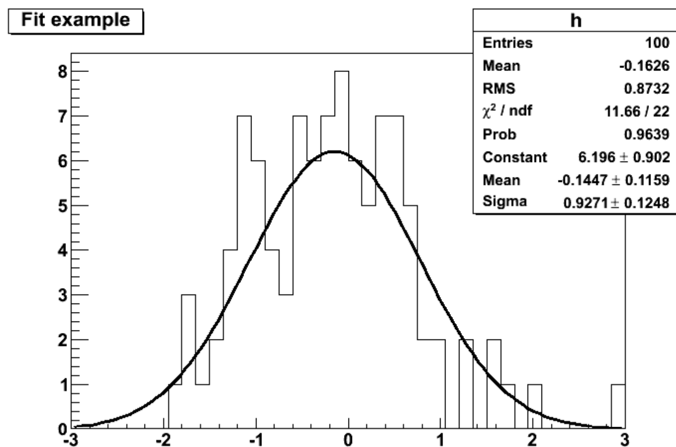
```
void generatingFnc() {…}

TH1* generateSample(…) {…}

TF1* fittingFnc() {…}

void toy() {
  Int_t sample_size = 100;
  Int_t nToy = 2000;

  generatingFnc();
  TF1 * fitfnc = fittingFnc();
  TH1F *hmean = new TH1F("hmean", "Fit mean",40,-1,1);

  for(Int_t ifit=0; ifit<nToy; ifit++) {  // the loop
    TH1* h = generateSample("gen_gaus",sample_size);
    h->Fit(fitfnc);
    hmean->Fill(fitfnc->GetParameter(1));
  }

  hmean->Draw();
  gStyle->SetOptFit(11111);
  hmean->Fit("gaus");
}
```

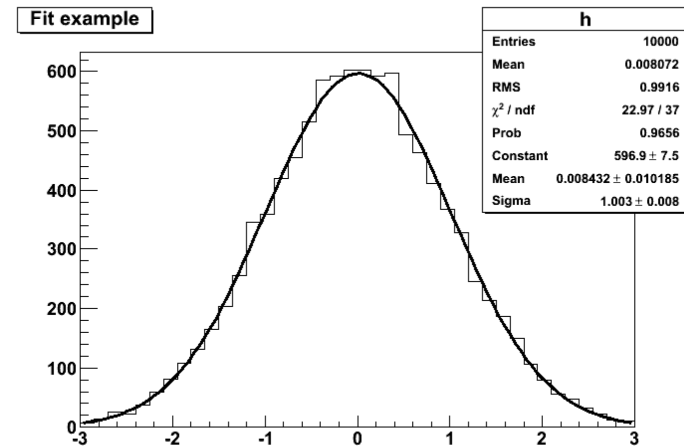
A toy MC

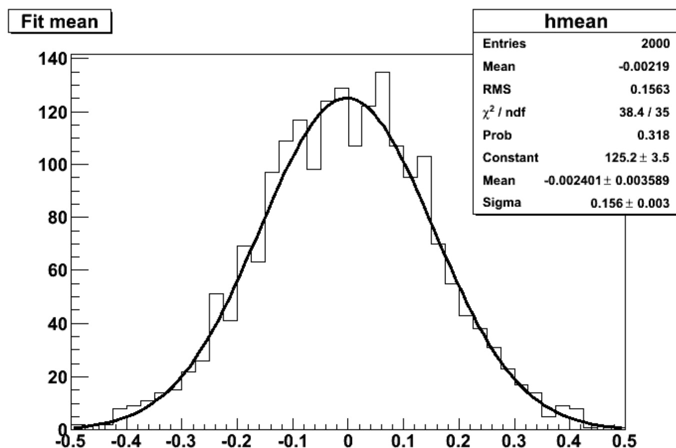
We find that fitted mean is consistent with 0 (-0.0024±0.0036). But how much can we trust a single measurement?

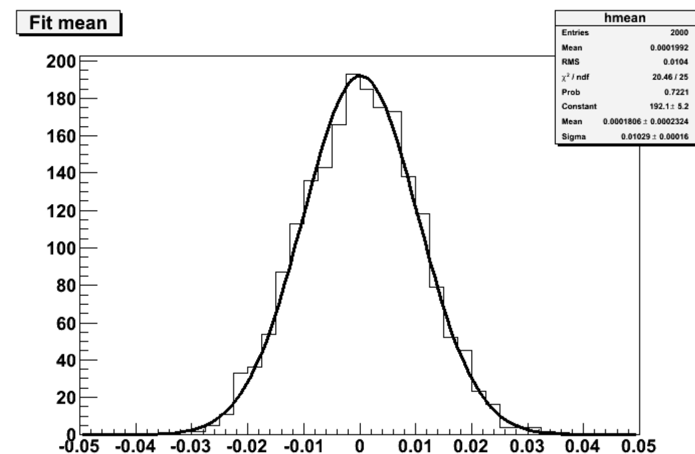
TASK BOX

Repeat this for a sample_size of 10000



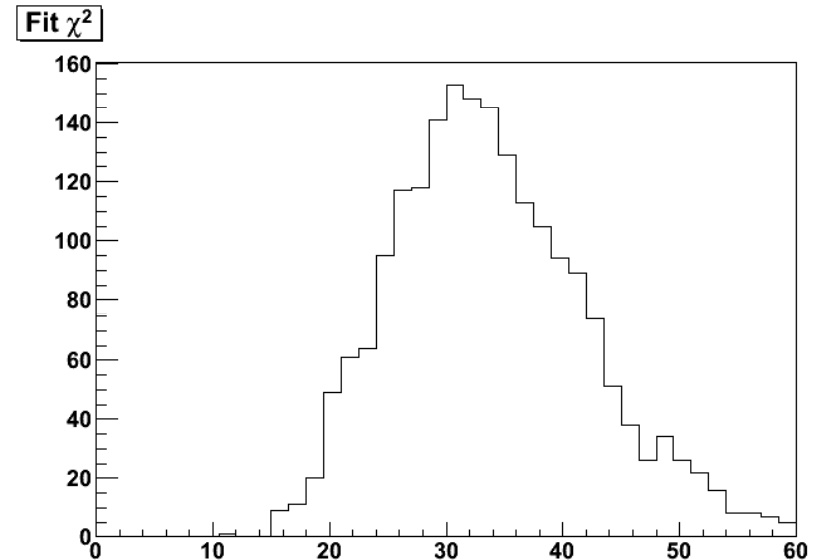






Note the different scale on the x-axis

Chi Square - a preview



Let's repeat the toy exercise, but let's histogram the chi-square

chi2.C

```
…
hchisq->Fill(fitfnc->GetChisquare());
…
```

This is the ChiSquare distribution for *n* degrees of freedom:

$$f(x;n) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2}$$

In our case *n*~35:

Remember, we fitted a Gaussian to a histogram with 40 bins. A Gaussian has 3 parameters. A few bins ~2 are empty. So n=40-3-(~2)=~35

Small Homework: Define the chi-square function (see page 16) and fit to the chi-square distribution. Do you fit n~35?

ACLIC

Automatic Compiler of Libraries for CINT

Simple macro.C can be run via

```
shell> root macro.C
```

or from inside a root session via

```
.x macro.C
```

macro.C

```
void macro() {
  TH1* h = new TH1F("h","hist",10,0,1);
  for(Int_t i=0; i<100; i++) {
    h->Fill(gRandom->Rndm());
  }
  h->Draw();
}
```

First step to get all your developments into a compiled library is to get your macro to compile. This can be done using ROOTs ACLIC. Just put a + behind the filename:

```
shell> root macro.C+
```

That is what you get:

```
Processing macro.C+...
Info in <TUnixSystem::ACLiC>: creating shared library /home/stelzer/root_seminar_istapp/workdir/./macro_C.so
/home/stelzer/root_seminar_istapp/workdir/./macro.C: In function 'void macro()':
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: 'TH1' was not declared in this scope
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: 'h' was not declared in this scope
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: expected type-specifier before 'TH1F'
/home/stelzer/root_seminar_istapp/workdir/./macro.C:3: error: expected `;' before 'TH1F'
/home/stelzer/root_seminar_istapp/workdir/./macro.C:5: error: 'gRandom' was not declared in this scope
g++: /home/stelzer/root_seminar_istapp/workdir/macro_C_ACLiC_dict.o: No such file or directory
Error in <ACLiC>: Compilation failed!
Error: Function macro() is not defined in current scope  :0:
*** Interpreter error recovered ***
```

Everything must be included

Complain was about undeclared classes TH1, TH1F and names gRandom.

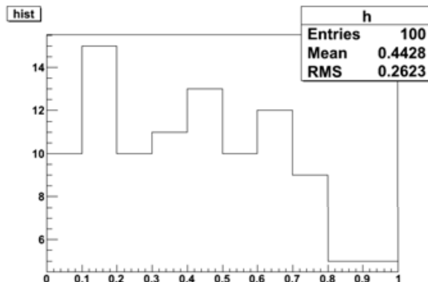
$\Rightarrow$ Must specify correct header files to include. (can be found on ROOT web)

macro.C

```
#include "TH1.h"
#include "TH1F.h"
#include "TRandom.h"

void macro() {
  TH1* h = new TH1F("h","hist",10,0,1);
  for(Int_t i=0; i<100; i++) {
    h->Fill(gRandom->Rndm());
  }
  h->Draw();
}
```

That is what you get now:

```
root -l macro.C+
zsh: correct 'macro.C+' to 'macro.C~' [nyae]?
root [0]
Processing macro.C+...
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```



where to find

ROOT

Quick Links: ROOT Homepage | Class Index | Class Hierarchy
Source: header file | source file | viewVC header | viewVC source
Sections: class description | function members | data members | class charts

ROOT » MATH » MATHCORE » TRandom

class TRandom: public TNamed

class TRandom
library: libMathCore
#include "TRandom.h"
Display options:
Show inherited
Show non-public
[↑ Top] [? Help]

A standalone example

First lets compile a standalone program

```
shell> g++ `root-config --cflags` `root-config --libs` executable.cxx
```

Compared to macro.C only main() was added

This creates a file a.out

```
workdir> a.out
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with
name c1
```

executable.cxx

```
#include "TH1.h"
#include "TH1F.h"
#include "TRandom.h"

void macro() {
  TH1* h = new TH1F("h","hist",10,0,1);
  for(Int_t i=0; i<100; i++) {
    h->Fill(gRandom->Rndm());
  }
  h->Draw();
}

int main() {
  macro();
}
```

root-config provides you convenient access to includes and libs, which are needed:

```
shell> root-config --cflags
```

```
-pthread -m64 -I/usr/local/root/trunk/include
```

```
shell> root-config --libs
```

```
-L/usr/local/root/trunk/lib -lCore -lCint -lRIO -lNet -lHist -
lGraf -lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -
lPhysics -lMathCore -lThread -pthread -lm -ldl -rdynamic
```

To build complete libraries and executable it is a good and common practice to use the **build system make.** ⇒ compilation of code and building of libraries is done in correct order

Building your own library

If you have macro A.cxx that you would like in a library mylib.so for quick access

Build the object files

```
shell> g++ -Wall -fPIC `root-config --cflags` -c A.cxx -o A.o
```

And put them into a shared lib:

```
shell> g++ -shared A.o -o mylib.so
```

However, unlike when using ACLIC you can not yet use this library inside CINT

```
root [0] gSystem->Load("mylib.so");
root [1] A()
Error: Function A() is not defined in current scope  (tmpfile):1:
*** Interpreter error recovered ***
```

⇒ The **CINT dictionary** is missing. This dictionary holds the entry points to the functions that CINT ought to know.

LinkDef.h

Generating the CINT dictionary requires a definition file

Generate the dictionary:

the dictionary must be specified last after all definition files

```
shell> rootcint -f Dict.C -c -p -I./ A.h LinkDef.h
```

```
workdir> ls Dict.*
Dict.C   Dict.h
```

compile it

```
shell> g++ -Wall -fPIC `root-config --cflags` -c Dict.C -o Dict.o
```

and add it to the library

```
shell> g++ -shared A.o Dict.o -o mylib.so
```

```
root [0] gSystem->Load("mylib.so");
root [1] A()
function A() called
```

LinkDef.h

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link off all namespaces;

#pragma link C++ function A;

#endif
```

These tell CINT about function A()

A.h

```
void A();
```

Designing our own class "Event"

As an exercise we will define a class "Event" for our data analysis.

Don't worry, you don't have to do this for the experiment you might be working at, many people are most likely working on this already. But:

- you do your own analysis,
- you implement your own ideas, and – having gotten a liking for object oriented programming –
- you want to work with your own information in self-designed C++ objects

⇒ what better way to keep your objects in root files ?

I am not advertizing that everybody is to invent the wheel again. After all we work in collaborations. But sometimes you are the first with a good idea, or the old way of doing things wasn't good.

So here we go!

Writing our own Event class

The light-weight TObject class provides the default behavior and interface for the objects in the ROOT system.

Primary interface to classes providing object I/O, error handling, inspection, introspection, and drawing.

You also need a public default constructor

When ROOT reads objects from a TTree, it needs to create an empty object of this type

Adding **ClassDef** is needed for

- Extensive RTTI (Run Time Type Information)
- ROOT object I/O
- Inspection

Adding **ClassImp** is needed for

- HTML documentation but somewhat obsolete

Event.h

```
#include "TObject.h"
#include "LorentzVector.h"

class Event : public TObject {
public:
  Event();  // default constructor
  virtual ~Event(){};
private:
  Float_t        fMissingEnergy;
  Int_t          fNTracks;
  TLorentzVector  fMostEnergeticMuon;
  ClassDef(Event,1); //My analysis event
};
```

Event.cxx

```
#include "Event.h"

ClassImp(Event)

Event::Event() :
  fMissingEnergy(0),
  fNTracks(0),
  fMostEnergeticMuon(0,0,0,0)
{}
```

The role of TObject and ClassDef

ClassDef enables introspection / reflection: a class can look inside itself at runtime

- TClass *cl = obj->**IsA()**, provides all information about the class
- const char* name = obj->**ClassName()**
- Bool_t b = obj->**InheritsFrom("TLine")**
- obj->**ShowMembers()** access parameters, methods, comments, …

Collections

Only TObjects can be stored in ROOT collections (e.g. TClonesArray)

ROOT I/O

obj->Write() inherited from TObject to write object to file

Uses Streamer(): Object ←→ TBuffer (=flat data structure for storage)

You need a public default constructor and a virtual destructor

Using our Event.so library

Let's build the library:

```
shell> source build.sh
```

build.sh

```
g++ -Wall -fPIC `root-config --cflags` -c Event.cxx -o Event.o
rootcint -f EventDict.C -c -p -I. Event.h EventLinkDef.h
g++ -Wall -fPIC `root-config --cflags` -c EventDict.C -o EventDict.o
g++ -shared Event.o EventDict.o -o libEvent.so
```

EventLinkDef.h

```
#ifdef __CINT__
#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link off all namespaces;
#pragma link C++ class Event;
#endif
```

And use it within root:

```
gSystem->Load("libEvent.so");
Event *ev = new Event();
ev->Dump();
```

It works !

We have to load the libraries by hand every time, that's a bit annoying.

ROOT provides a feature, called root-maps to lookup the libraries needed for each class.
Let's add this Event.rootmap to our directory:

Event.rootmap

```
Library.Event:      libEvent.so
```

Now we can use the Event out of the box:

```
Event *ev = new Event();
```

Event I/O

Let's extent Event a bit. We add a constructor with values and a print helper function

These will just help us to easily fill events with values and visualize them.

Event.cxx

```
…
Event::Event(Float_t missingEnergy, Int_t nTracks, const TLorentzVector& mostEnergeticMuon) :
  fMissingEnergy(missingEnergy),
  fNTracks(nTracks),
  fMostEnergeticMuon(mostEnergeticMuon)
{}

std::ostream& operator<< ( ostream& os, const Event& event ) {
  os << "Missing energy     : " << event.fMissingEnergy << std::endl;
  os << "Number of tracks    : " << event.fNTracks << std::endl;
  os << "Most energetic muon : " << event.fMostEnergeticMuon.E()
    << " (" << event.fMostEnergeticMuon.X() << "," << event.fMostEnergeticMuon.Y()
    << "," << event.fMostEnergeticMuon.Z() << ")" << std::endl;
return os;
}
```

The function **operator<<** is not part of the Event class. So it needs to declared separately in EventLinkDef.h

EventLinkDef.h

```
#pragma link C++ function operator<<( std::ostream&, const Event& );
```

Event I/O – Write

writeEvent.C

```
void writeEvent() {
  TFile *f= TFile::Open("SingleEvent.root","RECREATE");
  Event ev(3.5,7,TLorentzVector(1,2,3,4));
  cout << ev << endl;
  ev.Write("EventNr1");
  f->Close();
}
```

Let's save an event in a file

- Open file in "recreate" mode

  Careful, this overwrites old files
- Create an event with some values
- Print
- Write with key "*EventNr1*"

  writing always happens in the current directory of the current file

```
workdir> root -l writeEvent.C
root [0]
Processing writeEvent.C...
Missing energy       : 3.5
Number of tracks     : 7
Most energetic muon : 4 (1,2,3)
workdir> ls *.root
SingleEvent.root
```

try:

```
TFile *_file0 = TFile::Open("SingleEvent.root")
EventNr1->Dump()
```

Event I/O – Read

Let's read the event back

- Open file in "read" mode
- Get the event with key "*EventNr1*"
- Print

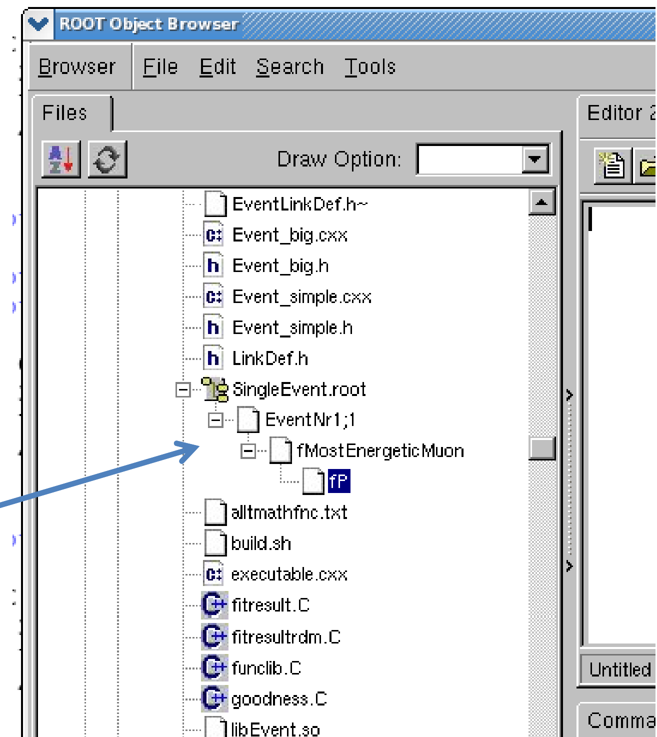
readEvent.C

```
void readEvent() {
  TFile *f= TFile::Open("SingleEvent.root","READ");
  Event *ev = f->Get("EventNr1");
  cout << *ev << endl;
  f->Close();
}
```

```
workdir> root -l readEvent.C
root [0]
Processing readEvent.C...
Missing energy        : 3.5
Number of tracks      : 7
Most energetic muon : 4 (1,2,3)
```

Can inspect the **EventNr1** with the TBrowser

It shows the structure of the Event, but not the primary types



Filling an Event tree

fillEventTree.C

Let's save 10000 events

Preparation

- Open file and create new tree
- Create ***empty Event*** object on the heap
- Create a branch in the tree for the events

  &event is a reference to the event pointer (double redirection)

  99 is the "splitlevel", more later

Loop

Generate random values

Set them in the event

Call tree->Fill(): this copies the values into the tree structure

Final

Write the tree and close the file

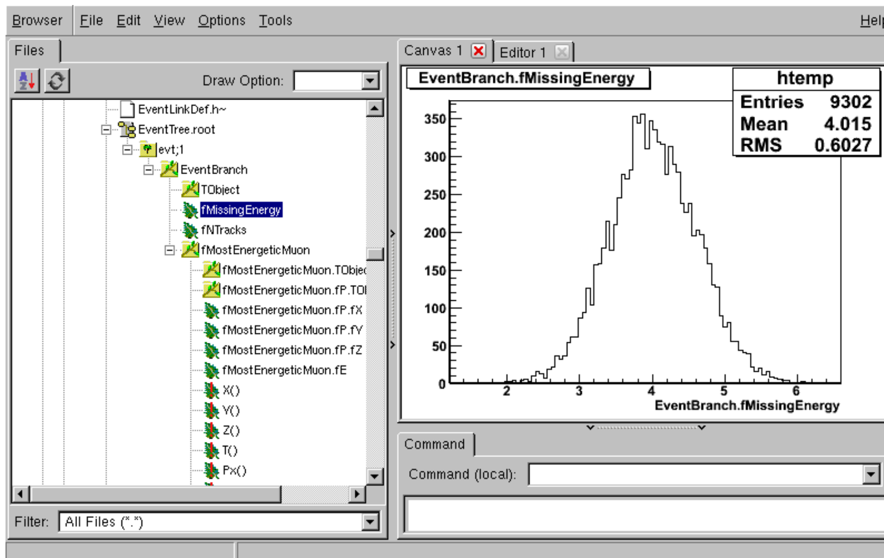
```
void fillEventTree() {
  TFile *f = TFile::Open("EventTree.root","RECREATE");
  TTree *tree = new TTree("evt","EventTree");

  Event *event = new Event();
  tree->Branch("EventBranch","Event",&event,32000,99);

  for(Int_t i=0; i<10000; i++) {

    Float_t E = gRandom->Landau(20,5);
    Float_t m = gRandom->Gaus(0.105,0.002);
    if(E>100) continue;
    Double_t px,py,pz;
    gRandom->Sphere(px,py,pz,TMath::Sqrt(E*E-m*m));

    event->SetMissingEnergy(gRandom->Gaus(4,0.6));
    event->SetNTracks(gRandom->Poisson(7.5));
    event->SetMostEnergeticMuon(
                  TLorentzVector(px,py,pz,E));
    tree->Fill();
  }

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

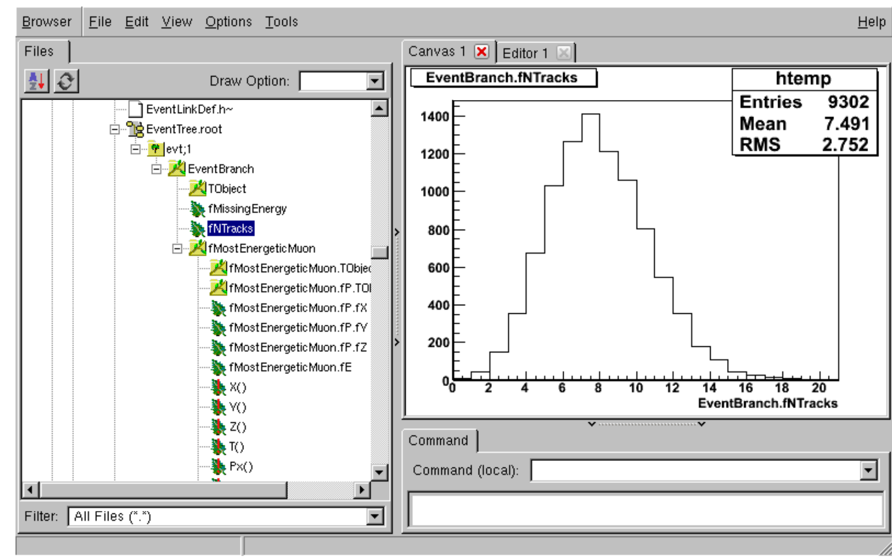
Plotting Event properties

With the TBrowser we can look at the EventTree

- EventTree.root — file
  - evt — tree
    - EventBranch — event branch
      - fMissingEnergy, … — variable branches

and plot the variables





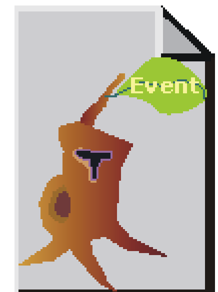
Split Level

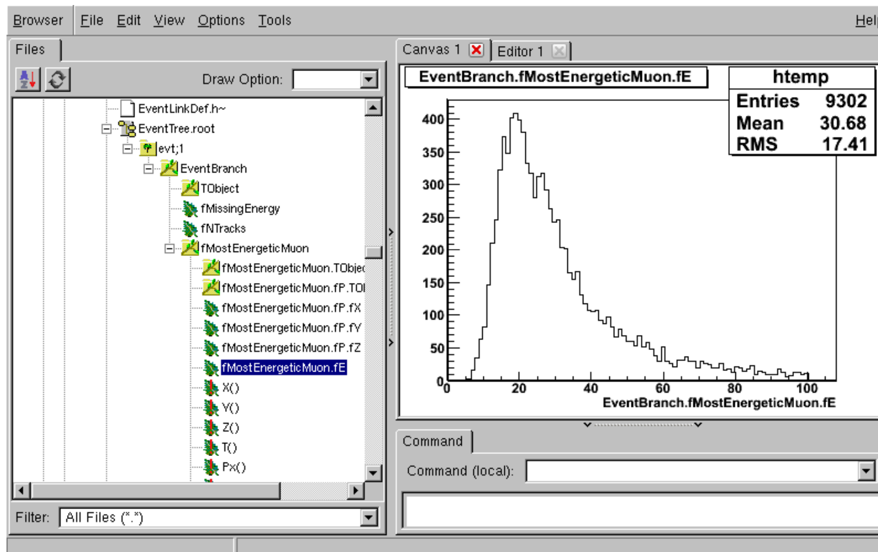
When creating a branch for an object type one can chose if that object should be split into its components, such that each component gets it own branch.

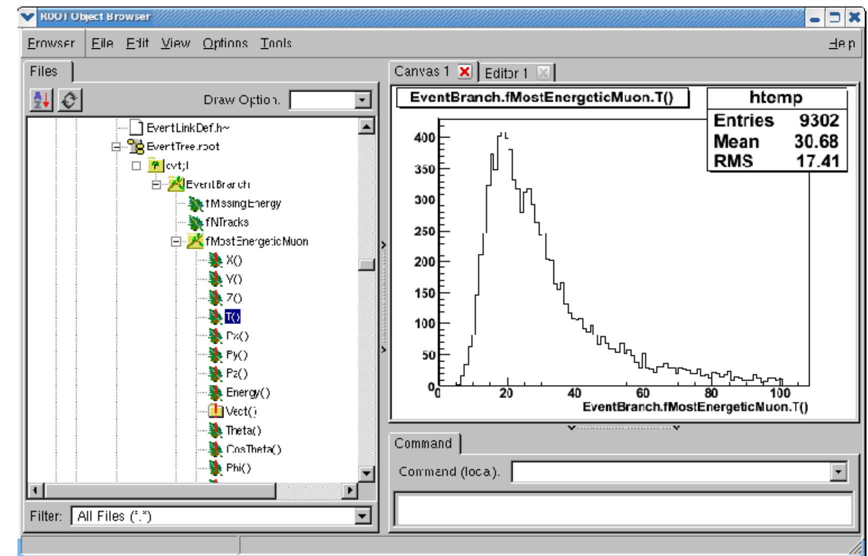
Generally a split branch is faster to read but a bit slower to write. A split branch needs more memory.

Split (SL=99): e.g. all 4 variables of the TLorentzVector are visible

Unsplit (SL=0): e.g. no internals of the TLorentzVector are visible. Have to use functions like E() to access fE



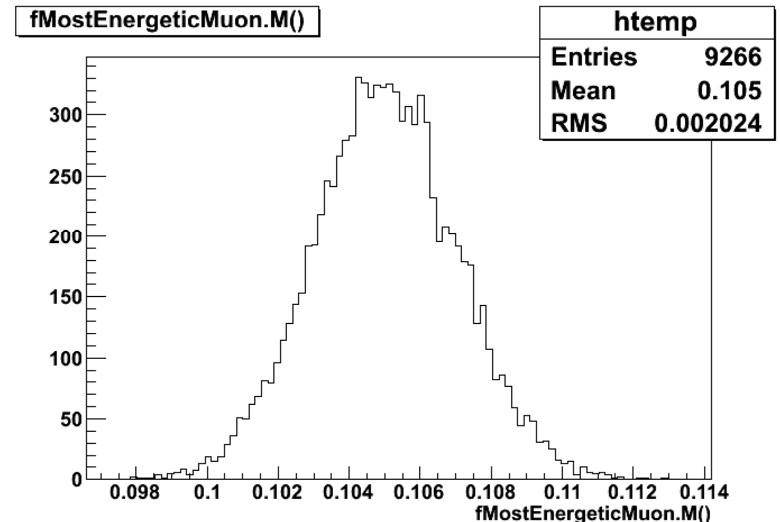




Plotting of functions variables

Simple macro to plot the mass of the most energetic muon:

```
TFile *f = TFile::Open("EventTree.root","READ");
TTree *tree = f->Get("evt");
tree->Draw("fMostEnergeticMuon.M()");
```



So we can use functions of the stored object, e.g. TLorentzVector.M() !

Note: this does not work when objects are in split branches. The macro fails for splitlevels>1 !

Functions of objects only work with Draw() when the objects is not split!

You can prevent splitting by adding "//||" as comment behind the field

This way you can always use fMostEnergeticMuon.M()

Event.h

```
class Event : public TObject {
…
  TLorentzVector  fMostEnergeticMuon; //||
};
```

How I/O works – object streaming

For classes rootcint creates:

- Entry points for class members: root[] event->SetMissingEnergy(3.1);
- Class streamer: Event::**Streamer**(). TBuffer::**operator>>**(Event)
  ⇒ These auto-generated functions do all the hard work

Sometimes you want to modify the default behavior

In Class.h:

Class.h

```
Float_t fTmp;  //! do not stream
TH1F *fH; //-> pointer to non-zero object with separate streamer
TLorentzVector   pMax;  //||  prevent splitting
```

In LinkDef.h:

EventLinkDef.h

```
#pragma link C++ class ClassA!;    // no streamer
#pragma link C++ class ClassB-;    // no operator<<
```

Then you can and have to define these function yourself.

That's it

We have learned two extremely useful things, which will help us doing a good analysis

How to define functions and fit them to the binned data

- We can judge if we have chosen a good fitting model
- We can extract parameters of the generating distribution
- We can study how accurate our parameter estimate will be
- For unbinned fits ROOT contains a package RooFit

How to design and build your own data analysis class, and how to persistify this information in ROOT files:

- This allows you to structure your analysis results as you wish and make you more efficient

Homework: extending the Event class

At the moment the event class is very limited. We would like to store a variety of different information:

Event.h

```
class Event : public TObject {
…
  TH1F*  fh;              //->          a histogram (pointer to other object)
  TDirectory   *fCurDir;  //!           the current directory (temporary)
  Float_t   *p;           //[fNTracks]  the momentum of each track (array of variable size)
  Float_t  ptmiss[2];     //            the missing transverse momentum (array of fixed size)
};
```

1. add the new member variables
2. adapt the print function
3. extend writeEvent to fill the new variables
4. Write event to file and read it back and compare

Hint: you will need these seven files: Event.h, Event.cxx, EventDict.h, build.sh, writeEvent.C, readEvent.C, Event.rootmap. Three of those need modification.

Hint 2: Clone() uses streamer to make a copy

```
Event * newEv = ev->Clone();
```