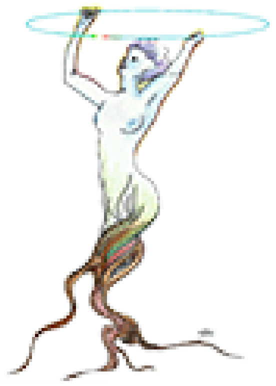
Institut für Experimentelle Kernphysik

A ROOT Guide For Students

"Diving Into ROOT"



http://root.cern.ch

Abstract:
ROOT is an object-oriented framework for data analysis. Among its prominent features are an advanced graphical user interface for visualization and interactive data analysis and an interpreter for the C++ programming language, which allows rapid prototyping of analysis code based on the C++ classes provided by ROOT. Access to ROOT classes is also possible from the very versatile and popular scripting language PYTHON.
This introductory guide shows the main features applicable to typical problems of data analysis in student labs: input and plotting of data from measurements and comparison with and fitting of analytical functions. Although appearing to be quite a heavy gun for some of the simpler problems, getting used to a tool like ROOT at this stage is an optimal preparation for the demanding tasks in state-of-the art, scientific data analysis.

Authors:

Danilo PIPARO,
Günter QUAST,
Manuel ZEISE

Version of December 5, 2013

CHAPTER 1 MOTIVATION AND INTRODUCTION

Welcome to data analysis !

Comparison of measurements to theoretical models is one of the standard tasks in experimental physics. In the most simple case, a "model" is just a function providing predictions of measured data. Very often, the model depends on parameters. Such a model may simply state "the current I is proportional to the voltage U", and the task of the experimentalist consists of determining the resistance, R, from a set of measurements.

As a first step, a visualisation of the data is needed. Next, some manipulations typically have to be applied, e. g. corrections or parameter transformations. Quite often, these manipulations are complex ones, and a powerful library of mathematical functions and procedures should be provided - think for example of an integral or peak-search or a Fourier transformation applied to an input spectrum to obtain the actual measurement described by the model.

One specialty of experimental physics are the inevitable errors affecting each measurement, and visualization tools have to include these. In subsequent analysis, the statistical nature of the errors must be handled properly.

As the last step, measurements are compared to models, and free model parameters need to be determined in this process , see Figure1.1 for an example of a function (model) fit to data points. Several standard methods are available, and a data analysis tool should provide easy access to more than one of them. Means to quantify the level of agreement between measurements and model must also be available.

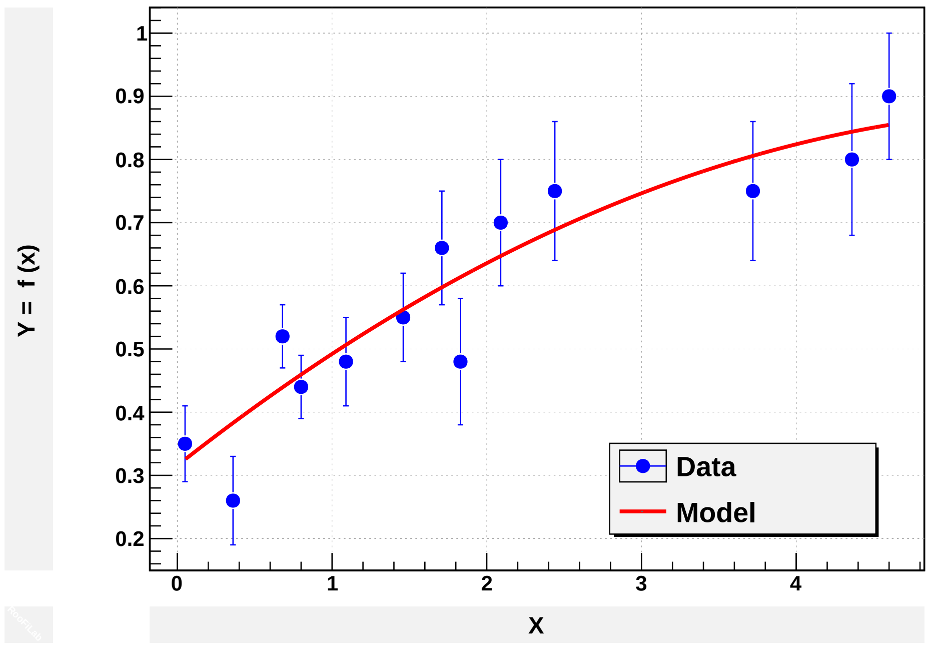


Figure 1.1.: Measured data points with error bars and fitted quadratic function .

Quite often, the data volume to be analyzed is large - think of fine-granular measurements accumulated with the aid of computers. A usable tool therefore must contain easy-to-use and efficient methods for data handling.

In Quantum mechanics, models typically only predict the probability density function ("pdf") of measurements depending on a number of parameters, and the aim of the experimental analysis is to extract the parameters from the observed distribution of frequencies at which certain values of the measurement are observed. Measurements of this kind require means to generate and visualize frequency distributions, so-called histograms, and stringent statistical treatment to extract the model parameters from purely statistical distributions.

Simulation of expected data is another important aspect in data analysis. By repeated generation of "pseudo-data", which are analysed in the same manner as intended for the real data, analysis procedures can be validated or compared. In many cases, the distribution of the measurement errors is not precisely known, and simulation offers the possibility to test the effects of different assumptions.

1.1. Welcome to ROOT

A powerful software framework addressing all of the above requirements is ROOT [1], an open source project coordinated by the European Centre for Particle Physics, CERN in Geneva. ROOT is very flexible and provides both a programming interface to use in own applications and a graphical user interface for interactive data analysis. The purpose of this document is to serve as a beginners guide and provides extendable examples for your own use cases, based on typical problems addressed in student labs. This guide will hopefully lay the ground for more complex applications in your future scientific work building on a modern, state-of the art tool for data analysis.

This guide in form of a tutorial is intended to introduce you to the ROOT package in about 50 pages. This goal will be accomplished using concrete examples, according to the "learning by doing" principle. Also because of this reason, this guide cannot cover the complexity of the ROOT package. Nevertheless, once you feel confident with the concepts presented in the following chapters, you will be able to appreciate the ROOT Users Guide [2] and navigate through the Class Reference [3] to find all the details you might be interested in. You can even look at the code itself, since ROOT is a free, open-source product. Use these documents in parallel to this tutorial!

The ROOT Data Analysis Framework itself is written in and heavily relys on the programming language `C++`, and therefore some knowledge about `C` and`C++` is required. Eventually, just profit from the immense available literature about `C++` if you do not have any idea of what object oriented programming could be.

Recently, an alternative and very powerful way to use and control ROOT classes via the interpreted high-level programming language PYTHON became available. PYTHON itself offers powerful modules and packages for data handling, numerical applications and scienfific computing. A vast number of bindings or wrappers to packages and tools written in other languages is also available. Access to the ROOT functionality is provided by the ROOT package `PyRoot` [5], allowing interactive work as well as scritps based on PYTHON. This is presented at the end of this guide in Chapter 8.

ROOT is available for many platforms (Linux, Mac OS X, Windows...), but in this guide we will implicitly assume that you are using Linux. The first thing you need to do with ROOT is install it. Or do you? Obtaining the latest ROOT version is straightforward. Just seek the "Pro" version on this webpage `http://root.cern.ch/drupal/content/downloading-root`. You will find precompiled versions for the different architectures, or the ROOT source code to compile yourself. Just pick up the flavour you need and follow the installation instructions. Or even simpler: use a virtual machine with ROOT installed ready for use, as availalbe under e. g. `http://www-ekp.physik.uni-karlsruhe.de/~quast`.

Let's dive into ROOT!

CHAPTER 2 ROOT BASICS

Now that you have installed ROOT, what's this interactive shell thing you're running? It's like this: ROOT leads a double life. It has an interpreter for macros (CINT [4]) that you can run from the command line or run like applications. But it is also an interactive shell that can evaluate arbitrary statements and expressions. This is extremely useful for debugging, quick hacking and testing. Let us first have a look at some very simple examples.

2.1. ROOT as calculator

You can even use the ROOT interactive shell in lieu of a calculator! Launch the ROOT interactive shell with the command

```
> root
```

on your Linux box. The prompt should appear shortly:

```
root [1]
```

and let's dive in with the steps shown here:

```
root [0] 1+1
(const int)2
root [1] 2*(4+2)/12.
(const double)1.000000000000000000e+00
root [2] sqrt(3)
(const double)1.73205080756887719e+00
root [3] 1 > 2
(const int)0
root [4] TMath::Pi()
(Double_t)3.14159265358979312e+00
root [5] TMath::Erf(.2)
(Double_t)2.22702589210478447e-01
```

Not bad. You can see that ROOT offers you the possibility not only to type in C++ statements, but also advanced mathematical functions, which live in the TMath namespace.

Now let's do something more elaborated. A numerical example with the well known geometrical series:

```
root [6] double x=.5
root [7] int N=30
root [8] double geom_series=0
root [9] for (int i=0;i<N;++i)geom_series+=TMath::Power(x,i)
root [10] TMath::Abs(geom_series - (1-TMath::Power(x,N-1))/(1-x))
(Double_t)1.86264514923095703e-09
```

Here we made a step forward. We even declared variables and used a *for* control structure. Note that there are some subtle differences between CINT and the standard C++ language. You do not need the ";" at the end of line in interactive mode – try the difference e.g. using the command at line `root [6]`.

2.2. ROOT as Function Plotter

Using one of ROOT's powerful classes, here `TF1` [1], will allow us to display a function of one variable, x. Try the following:

```
root [11] TF1 *f1 = new TF1("f1","sin(x)/x",0.,10.);
root [12] f1->Draw();
```

`f1` is a pointer to an instance of a TF1 class, the arguments are used in the constructor; the first one of type string is a name to be entered in the internal ROOT memory management system, the second string type parameter defines the function, here `sin(x)/x`, and the two parameters of type real define the range of the variable x. The `Draw()` method, here without any parameters, displays the function in a window which should pop up after you typed the above two lines. Note again differences between CINT and C++: you could have omitted the ";" at the end of lines, of CINT woud have accepted the "." to access the method `Draw()`. However, it is best to stick to standard C++ syntax and avoid CINT-specific code, as will become clear in a moment.

A slightly extended version of this example is the definition of a function with parameters, called [0], [1] and so on in ROOT formula syntax. We now need a way to assign values to these parameters; this is achieved with the method `SetParameter(<parameter_number>,<parameter_value>)` of class `TF1`. Here is an example:

```
root [13] TF1 *f1 = new TF1("f2","[0]*sin([1]*x)/x",0.,10.);
root [14] f1->SetParameter(0,1);
root [15] f1->SetParameter(1,1);
root [16] f1->Draw();
```

Of course, this version shows the same results as the initial one. Try playing with the parameters and plot the function again. The class `TF1` has a large number of very useful methods, including integration and differentiation. To make full use of this and other ROOT classes, visit the documentation on the Internet under `http://root.cern.ch/drupal/content/reference-guide`. Formulae in ROOT are evaluated using the class `TFormula`, so also look up the relevant class documentation for examples, implemented functions and syntax.

On many systems, this class reference-guide is available locally, and you should definitely download it to your own system to have it at you disposal whenever you need it.

To extend a little bit on the above example, consider a more complex function you would like to define. You can also do this using standard `C` or `C++` code. In many cases this is the only practical way, as the ROOT formula interpreter has clear limitations concerning complexity and speed of evaluation.

Consider the example below, which calculates and displays the interference pattern produced by light falling on a multiple slit. Please do not type in the example below at the ROOT command line, there is a much simpler way: Make sure you have the file `slits.cxx` on disk, and type `root slits.cxx` in the shell. This will start root and make it read the "macro" `slit.cxx`, i. e. all the lines in the file will be executed one after the other.

```
/* *** example to draw the interference pattern of light
       falling on a grid with n slits
       and ratio r of slit widht over distance between slits  ***  */

/* function code in C */
double single(double *x, double *par) {
 double const pi=4*atan(1.);
 return pow(sin(pi*par[0]*x[0])/(pi*par[0]*x[0]),2); }

double nslit0(double *x,double *par){
 double const pi=4*atan(1.);
 return pow(sin(pi*par[1]*x[0])/sin(pi*x[0]),2); }
```

[1] All ROOT classes start with the letter T.

```
13
14 double nslit(double *x, double *par){
15  return single(x,par) * nslit0(x,par); }
16
17 /* This is the main program */
18 void slits() {
19  float r,ns;
20
21 /* request user input */
22  cout << "slit width / g ? ";
23  scanf("%f",&r);
24  cout << "# of slits? ";
25  scanf("%f",&ns);
26  cout <<"interference pattern for "<< ns<<" slits, width/distance: "<<r<<endl;
27
28 /*define function and set options */
29 TF1 *Fnslit  = new TF1("Fnslit",nslit,-5.001,5.,2);
30  Fnslit->SetNpx(500);          // set number of points to 500
31
32  Fnslit->SetParameter(0,r);     //set parameters, as read in above
33  Fnslit->SetParameter(1,ns);
34
35  Fnslit->Draw();       // draw the interference pattern for a grid with n slits
36 }
```

file: `slits.cxx`

The example first asks for user input, namely the ratio of slit width over slit distance, and the number of slits. After entering this information, you should see the graphical output as is shown in Figure 2.1 below.

This is a more complicated example than the ones we have seen before, so spend some time analysing it carefully, you should have understood it before continuing. Let us go through in detail:

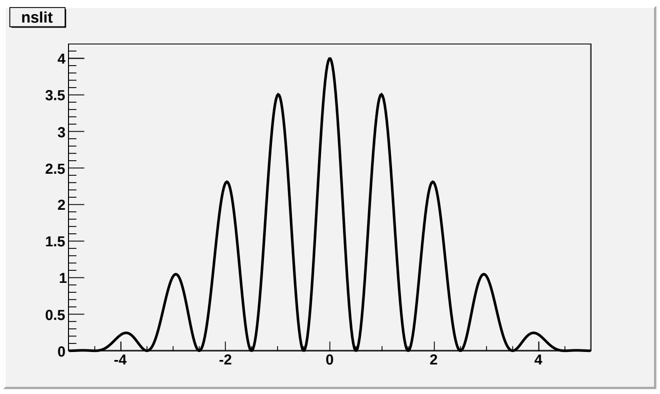


Figure 2.1.: Output of macro slits.cxx with parameters 0.2 and 2.

Lines 6-19 define the necessary functions in `C++` code, split into three separate functions, as suggested by the problem considered. The full interference pattern is given by the product of a function depending on the ratio of the width and distance of the slits, and a second one depending on the number of slits. More important for us here is the definition of the interface of these functions to make them usable for the ROOT class `TF1`: the first argument is the pointer to x, the second one points to the array of parameters.

The main program starts in line 17 with the definition of a function `slits()` of type `void`. After asking for user input, a ROOT function is defined using the C-type function given in the beginning. We can now use all methods of the `TF1` class to control the behaviour of our function – nice, isn't it?

If you like, you can easily extend the example to also plot the interference pattern of a single slit, using function `double single`, or of a grid with narrow slits, function `double nslit0`, in `TF1` instances.

Here, we used a macro, some sort of lightweight program, that the interpreter distributed with ROOT, CINT, is able to execute. This is a rather extraordinary situation, since C++ is not natively an interpreted language! There is much more to say, therefore there is a dedicated chapter on macros.

2.3. Controlling ROOT

One more remark at this point: as every command you type into ROOT is usually interpreted by CINT, an "escape character" is needed to pass commands to ROOT directly. This character is the dot at the beginning of a line:

```
root [1] .<command>
```

To

- **quit root**, simply type `.q`
- obtain a **list of commands**, use `.?`
- **access the shell** of the operating system, type `.!<OS_command>`; try, e. g. `.!ls` or `.!pwd`
- **execute a macro**, enter `.x <file_name>`; in the above example, you might have used `.x slits.cxx` at the ROOT prompt
- **load a macro**, type `.L <file_name>`; in the above example, you might instead have used the command `.L slits.cxx` followed by the function call `slits();`. Note that after loading a macro all functions and procedures defined therein are available at the ROOT prompt.

2.4. Plotting Measurements

To display measurements in ROOT, including errors, there exists a powerful class `TGrapErrors` with different types of constructors. In the example here, we use data from the file `ExampleData.txt` in text format:

```
root [0] TGraphErrors *gr=new TGraphErrors("ExampleData.txt");
root [1] gr->Draw("AP");
```

You should see the output shown in Figure 2.2.

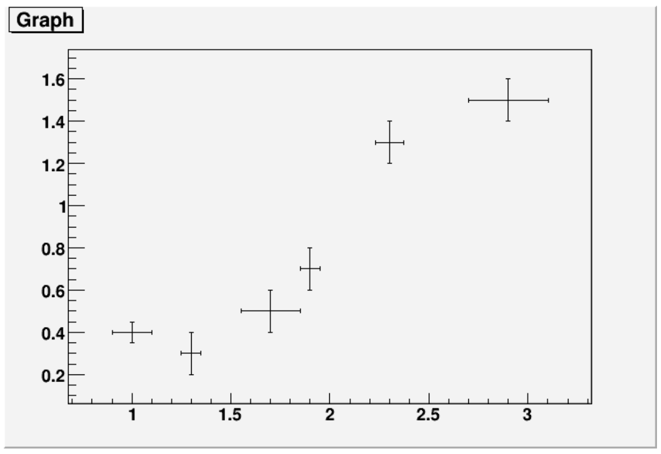


Figure 2.2.: Visualisation of data points with errors using the class TGraphErrors

Make sure the file `ExampleData.txt` is available in the directory from which you started ROOT. Inspect this file now with your favourite editor, or use the command `less ExampleData.txt` to inspect the file, you will see that the format is very simple and easy to understand. Lines beginning with `#` are ignored, very convenient to add some comments on the type of data. The data itself consist of lines with four real numbers each, representing the x- and y- coordinates and their errors of each data point. You should quit

The argument of the method `Draw("AP")` is important here. It tells the `TGraphPainter` class to show the axes and to plot markers at the x and y positions of the specified data points. Note that this simple example relies on the default settings of ROOT, concerning the size of the canvas holding the plot, the marker type and the line colours and thickness used and so on. In a well-written, complete example, all this would need to be specified explicitly in order to obtain nice and reproducible results. A full chapter on graphs will explain many more of the features of the class `TGraphErrors` and its relation to other ROOT classes in much more detail.

2.5. Histograms in ROOT

Frequency distributions in ROOT are handled by a set of classes derived from the histogram class `TH1`, in our case `TH1F`. The letter `F` stands for "float", meaning that the data type `float` is used to store the entries in one histogram bin.

```
root [0] TF1 efunc("efunc","exp([0]+[1]*x)",0.,5.);
root [1] efunc.SetParameter(0,1);
root [2] efunc.SetParameter(1,-1);
root [3] TH1F* h=new TH1F("h","example histogram",100,0.,5.);
root [4] for (int i=0;i<1000;i++) {h->Fill(efunc.GetRandom());}
root [5] h->Draw();
```

The first three lines of this example define a function, an exponential in this case, and set its parameters. In Line 4 a histogram is instantiated, with a name, a title, a certain number of 100 bins (i. e. equidistant, equally sized intervals) in the range from 0. to 5.

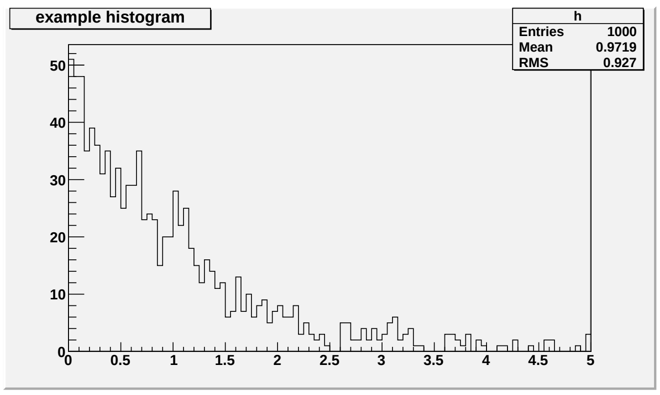


Figure 2.3.: Visualisation of a histogram filled with exponentially distributed, random numbers.

We use yet another new feature of ROOT to fill this histogram with data, namely pseudo-random numbers generated with the method `TF1::GetRandom`, which in turn uses an instance of the ROOT class `TRandom` created when ROOT is started. Data is entered in the histogram in line 5 using the method `TH1F::Fill` in a loop construct. As a result, the histogram is filled with 1000 random numbers distributed according to the defined function. The histogram is displayed using the method `TH1F::Draw()`. You may think of this example as repeated measurements of the life time of a quantum mechanical state, which are entered into the histogram, thus giving a visual impression of the probability density distribution. The plot is shown in Figure 2.3.

Note that you will not obtain an identical plot when executing the above lines, depending on how the random number generator is initialised.

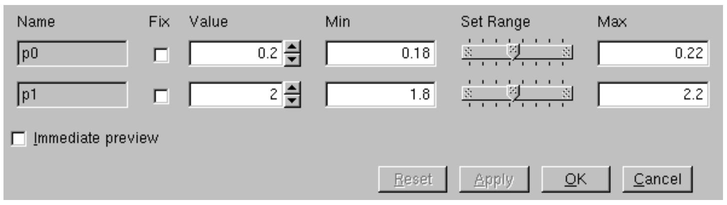
The class `TH1F` does not contain a convenient input format from plain text files. The following lines of C++ code do the job. One number per line stored in the text file “expo.dat” is read in via an input stream and filled in the histogram until end of file is reached.

```
root [1] TH1F* h=new TH1F("h","example histogram",100,0.,5.);
root [2] ifstream inp; double x;
root [3] inp.open("expo.dat");
root [4] while(!(inp >> x)==0){h->Fill(x);}
root [5] h->Draw();
root [6] inp.close();
```

Histograms and random numbers are very important tools in statistical data analysis, and the whole Chapter 5 will be dedicated to this.

2.6. Interactive ROOT

Look at one of your plots again and move the mouse across. You will notice that this is much more than a static picture, as the mouse pointer changes its shape when touching objects on the plot. When the mouse is over an object, a right-click opens a pull-down menu displaying in the top line the name of the ROOT class you are dealing with, e.g. `TCanvas` for the display window itself, `TFrame` for the frame of the plot, `TAxis` for the axes, `TPaveText` for the plot name. Depending on which plot you are investigating, menus for the ROOT classes `TF1`, `TGraphErrors` or `TH1F` will show up when a right-click is performed on the respective graphical representations. The menu items allow direct access to the members of the various classes, and you can even modify them, e.g. change colour and size of the axis ticks or labels, the function lines, marker types and so on. Try it!



Name	Fix	Value	Min	Set Range	Max
p0	☐	0.2	0.18		0.22
p1	☐	2	1.8		2.2

☐ Immediate preview

Figure 2.4.: Interactive ROOT panel for setting function parameters.

You will probably like the following: in the output produced by the example `slits.cxx`, right-click on the function line and select "SetLineAttributes", then left-click on "Set Parameters". This gives access to a panel allowing you to interactively change the parameters of the function, as shown in Figure 2.4. Change the slit width, or go from one to two and then three or more slits, just as you like. When clicking on "Apply", the function plot is updated to reflect the actual value of the parameters you have set.

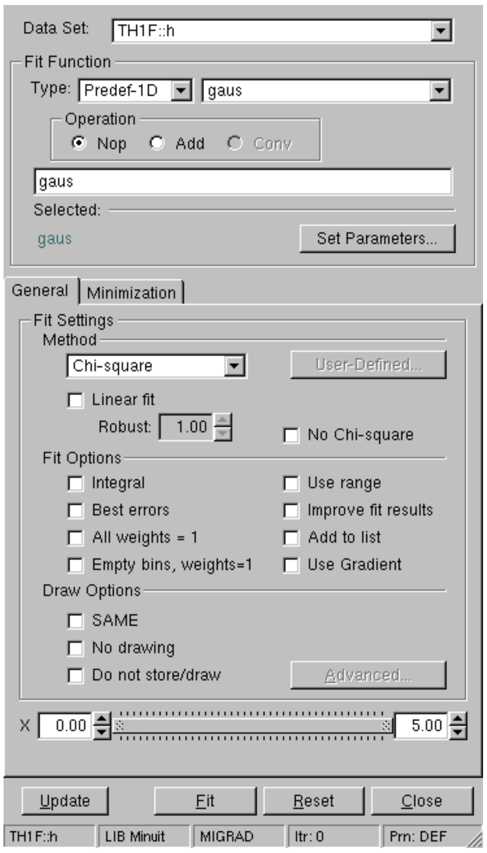
Another very useful interactive tool is the `FitPanel`, available for the classes `TGraphErrors` and `TH1F`. Predefined fit functions can be selected from a pull-down menu, including "`gaus`", "`expo`" and "`pol0`" - "`pol9`" for Gaussian and exponential functions or polynomials of degree 0 to 9, respectively. In addition, user-defined functions using the same syntax as for functions with parameters are possible.

After setting the initial parameters, a fit of the selected function to the data of a graph or histogram can be performed and the result displayed on the plot. The fit panel is shown in Figure 2.5. The fit panel has a large number of control options to select the fit method, fix or release individual paramters in the fit, to steer the level of output printed on the console, or to extract and display additional information like contour lines showing parameter correlations. Most of the methods of the class `TVirtualFitter` are easily available through the latest version of the graphical interface. As function fitting is of prime importance in any kind of data analysis, this topic will again show up in later chapters.

If you are satisfied with your plot, you probably want to save it. Just close all selector boxes you opened previously, and select the menu item $\boxed{\text{Save as}}$ from the menu line of the window, which will pop up a file selector box to allow you to choose the format, file name and target directory to store the image.

There is one very noticeable feature here: you can store a plot as a root macro! In this macro, you find the C++ representation of all methods and classes involved in generating the plot. This is a very valuable source of information for your own macros, which you will hopefully write after having worked through this tutorial.

Using the interactive capabilities of ROOT is very useful for a first exploration of possibilities. Other ROOT classes you will be encountering in this tutorial have such graphical interfaces as well. We will not comment further on this, just be aware of the existence of interactive features in ROOT and use them if you find convenient. Some trial-and-error is certainly necessary to find your way through the enormous number of menus and possible parameter settings.



Data Set: TH1F::h
Fit Function
Type: Predef-1D gaus
Operation: [✓ Nop] Add Conv
gaus
Selected: gaus
General | Minimization
Fit Settings
Method: Chi-square
Linear fit
Robust: 1.00
No Chi-square
Fit Options: Integral, Use range, Best errors, Improve fit results, All weights = 1, Add to list, Empty bins, weights=1, Use Gradient
Draw Options: SAME, No drawing, Do not store/draw
X 0.00 5.00
TH1F::h | LIB Minuit | MIGRAD | Itr: 0 | Prn: DEF

Figure 2.5.: Fit functions to graphs and histograms.

2.7. ROOT Beginners' FAQ

At this point of the guide, some basic question could have already come to your mind. We will try to clarify some of them with further explanations in the following.

2.7.1. ROOT type declarations for basic data types

In the official ROOT documentation, you find special data types replacing the normal ones, e. g. `Double_t`, `Float_t` or `Int_t` replacing the standard `double`, `float` or `int` types. Using the ROOT types makes it easier to port code between platforms (64/32 bit) or operating systems (windows/Linux), as these types are mapped to suitable ones in the ROOT header files. If you want adaptive code of this type, use the ROOT type declarations. However, usually you do not need such adaptive code, and you can safely use the standard C type declarations for your private code, as we did and will do throughout this guide. If you intend to become a ROOT developer, however, you better stick to the official coding rules!

2.7.2. Configure ROOT at start-up

If the file `.rootlogon.C` exists in your home directory, it is executed by ROOT at start-up. Such a file can be used to set preferred options for each new ROOT session. The ROOT default for displaying graphics looks OK on the computer screen, but rather ugly on paper. If you want to use ROOT graphs in documents, you should change some of the default options. This is done most easily by creating a new `TStyle` object with your preferred settings, as described in the class reference guide, and then use the command `gROOT->SetStyle("MyStyle");` to make this new style definition the default one. As an example, have a look in the file `rootlogon.C` coming with this tutorial.

There is also a possibility to set many ROOT features, in particular those closely related to the operating and window system, like e.g. the fonts to be used, where to find start-up files, or where to store a file containing the command history, and many others. The file searched for at ROOT start-up is called `.rootrc` and must reside in the user's home directory; reading and interpeting this file is handled by the ROOT class `TEnv`, see its documentation if you need such rather advanced features.

2.7.3. ROOT command history

Every command typed at the ROOT prompt is stored in a file `.root_hist` in your home directory. ROOT uses this file to allow for navigation in the command history with the up-arrow and down-arrow keys. It is also convenient to extract successful ROOT commands with the help of a text editor for use in your own macros.

2.7.4. ROOT Global Variables

All global variables in ROOT begin with a small "g". Some of them were already implicitly introduced (for example in session 2.7.2). The most important among them are presented in the following:

- **gROOT**: the gROOT variable is the entry point to the ROOT system. Technically it is an instance of the `TROOT` class. Using the gROOT pointer one has access to basically every object created in a ROOT based program. The `TROOT` object is essentially a container of several lists pointing to the main `ROOT` objects.
- **gRandom**: the gRandom variable is a variable that points to a random number generator instance of the type `TRandom3`. Such a variable is useful to access in every point of a program the same random number generator, in order to achieve a good quality of the random sequence.
- **gStyle**: By default ROOT creates a default style that can be accessed via the `gStyle` pointer. This class includes functions to set some of the following object attributes.
  - Canvas
  - Pad
  - Histogram axis
  - Lines
  - Fill areas
  - Text
  - Markers
  - Functions
  - Histogram Statistics and Titles
- **gSystem**: An instance of a base class defining a generic interface to the underlying Operating System, in our case `TUnixSystem`.

At this point you have already learnt quite a bit about some basic features of ROOT.

Please move on to become an expert!

CHAPTER 3 ROOT MACROS

You know how other books go on and on about programming fundamentals and finally work up to building a complete, working program? Let's skip all that. In this part of the guide, we will describe macros executed by the ROOT C++ interpreter CINT.

An alternative way to access ROOT classes interactively or in a script will be shown in Chapter 8, where we describe how to use the scritping language Python. This is most suitable for smaller analysis projects, as some overhead of the C++ language can be avoided. It is very easy to convert ROOT macros into python scripts using the *pyroot* interface.

Since ROOT itself is written in C++ let us start with Root macros in C++. As an additional advantage, it is relatively easy to turn a ROOT C++ macro into compiled – and hence much faster – code, either as a pre-compiled library to load into ROOT, or as a stand-alone application, by adding some include statements for header files or some "dressing code" to any macro.

3.1. General Remarks on ROOT macros

If you have a number of lines which you were able to execute at the ROOT prompt, they can be turned into a ROOT macro by giving them a name which corresponds to the file name without extension. The general structure for a macro stored in file `MacroName.cxx` is

```
void MacroName() {
        <          ...
          your lines of CINT code
                   ...            >
}
```

The macro is executed by typing

```
 > root MacroName.cxx
```

at the system prompt, or it can be loaded into a ROOT session and then be executed by typing

```
root [0].L MacroName.cxx
root [1] MacroName();
```

at the ROOT prompt. Note that more than one macro can be loaded this way, as each macro has a unique name in the ROOT name space. Because many other macros may have been executed in the same shell before, it is a good idea to reset all ROOT parameters at the beginning of a macro and define your preferred graphics options, e. g. with the code fragment

```
// re-initialise ROOT
gROOT->Reset();                // re-initialize ROOT
gROOT->SetStyle("Plain");      // set empty TStyle (nicer on paper)
gStyle->SetOptStat(111111);    // print statistics on plots, (0) for no output
gStyle->SetOptFit(1111);       // print fit results on plot, (0) for no ouput
gStyle->SetPalette(1);         // set nicer colors than default
gStyle->SetOptTitle(0);        // suppress title box
   ...
```

Next, you should create a canvas for graphical output, with size, subdivisions and format suitable to your needs, see documentation of class `TCanvas`:

```
// create a canvas, specify position and size in pixels
TCanvas c1("c1","<Title>",0,0,400,300);
c1.Divide(2,2); //set subdivisions, called pads
c1.cd(1); //change to pad 1 of canvas c1
```

These parts of a well-written macro are pretty standard, and you should remember to include pieces of code like in the examples above to make sure your output always comes out as you had intended.

Below, in section3.4, some more code fragments will be shown, allowing you to use the system compiler to compile macros for more efficient execution, or turn macros into stand-alone applications linked against the ROOT libraries.

3.2. A more complete example

Let us now look at a rather complete example of a typical task in data analysis, a macro that constructs a graph with errors, fits a (linear) model to it and saves it as an image. To run this macro, simply type in the shell:

```
> root macro1.cxx
```

The code is build around the ROOT class `TGraphErrors`, which was already introduced previously. Have a look at it in the class reference guide, where you will also find further examples. The macro shown below uses additional classes, `TF1` to define a function, `TCanvas` to define size and properties of the window used for our plot, and `TLegend` to add a nice legend. For the moment, ignore the commented include statements for header files, they will only become important at the end (section 3.4).

```
/* **** Builds a graph with errors, displays it and saves it as image. *** */
// first, include some header files (within CINT, these will be ignored)
#include "TCanvas.h"
#include "TROOT.h"
#include "TGraphErrors.h"
#include "TF1.h"
#include "TLegend.h"
#include "TArrow.h"
#include "TLatex.h"

void macro1(){
    // The values and the errors on the Y axis
    const int n_points=10;
    double x_vals[n_points]=
            {1,2,3,4,5,6,7,8,9,10};
    double y_vals[n_points]=
            {6,12,14,20,22,24,35,45,44,53};
    double y_errs[n_points]=
            {5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43};

    // Instance of the graph
    TGraphErrors graph(n_points,x_vals,y_vals,NULL,y_errs);
    graph.SetTitle("Measurement XYZ;lenght [cm];Arb.Units");

    // Make the plot estetically better
    gROOT->SetStyle("Plain");
    graph.SetMarkerStyle(kOpenCircle);
    graph.SetMarkerColor(kBlue);
    graph.SetLineColor(kBlue);

    // The canvas on which we'll draw the graph
    TCanvas* mycanvas = new TCanvas();

    // Draw the graph !
    graph.DrawClone("APE");

37     // Define a linear function
38     TF1 f("Linear law","[0]+x*[1]",.5,10.5);
39     // Let's make the funcion line nicer
40     f.SetLineColor(kRed); f.SetLineStyle(2);
41     // Fit it to the graph and draw it
42     graph.Fit(&f);
43     f.DrawClone("Same");
44 
45     // Build and Draw a legend
46     TLegend leg(.1,.7,.3,.9,"Lab. Lesson 1");
47     leg.SetFillColor(0);
48     graph.SetFillColor(0);
49     leg.AddEntry(&graph,"Exp. Points");
50     leg.AddEntry(&f,"Th. Law");
51     leg.DrawClone("Same");
52 
53     // Draw an arrow on the canvas
54     TArrow arrow(8,8,6.2,23,0.02,"----|>");
55     arrow.SetLineWidth(2);
56     arrow.DrawClone();
57 
58     // Add some text to the plot
59     TLatex text(8.2,7.5,"#splitline{Maximum}{Deviation}");
60     text.DrawClone();
61 
62     mycanvas->Print("graph_with_law.pdf");
63 }
64 
65 #ifndef __CINT__
66 int main(){
67     macro1();
68     }
69 #endif
```

file: `macro1.cxx`

Let's comment it in detail:

- Line 11: the name of the principal function (it plays the role of the "main" function in compiled programs) in the macro file. It has to be the same as the file name without extension.
- Line 22 − 23: instance of the `TGraphErrors` class. The constructor takes the number of points and the pointers to the arrays of x values, y values, x errors (in this case none, represented by the NULL pointer) and y errors. The second line defines in one shot the title of the graph and the titles of the two axes, separated by a ";".
- Line 26 − 29: the first line refers to the style of the plot, set as *Plain*. This is done through a manipulation of the global variable `gSystem` (ROOT global variables begin always with "g"). The following three lines are rather intuitive right? To understand better the enumerators for colours and styles see the reference for the `TColor` and `TMarker` classes.
- Line 32: the canvas object that will host the drawn objects. The "memory leak" is intentional, to make the object existing also out of the macro1 scope.
- Line 35: the method *DrawClone* draws a clone of the object on the canvas. It *has to be* a clone, to survive after the scope of `macro1`, and be displayed on screen after the end of the macro execution. The string option "APE" stands for:
  - *A* imposes the drawing of the Axes.
  - *P* imposes the drawing of the graphs markers.
  - *E* imposes the drawing of the graphs markers errors.
- Line 38: define a mathematical function. There are several ways to accomplish this, but in this case the constructor accepts the name of the function, the formula, and the function range.
- Line 40: maquillage. Try to give a look to the line styles at your disposal visiting the documentation of the `TLine` class.
- Line 42: fits the f function to the graph, observe that the pointer is passed. It is more interesting to look at the output on the screen to see the parameters values and other crucial information that we will learn to read at the end of this guide.

- Line 43: again draws the clone of the object on the canvas. The "Same" option avoids the cancellation of the already drawn objects, in our case, the graph.
- Line 46 – 51: completes the plot with a legend, represented by a `TLegend` instance. The constructor takes as parameters the lower left and upper right corners coordinates with respect to the total size of the canvas, assumed to be 1, and the legend header string. You can add to the legend the objects, previously drawn or not drawn, through the `addEntry` method. Observe how the legend is drawn at the end: looks familiar now, right?
- Line 54 – 56: defines an arrow with a triangle on the right hand side, a thickness of 2 and draws it.
- Line 59 – 61: interpret a Latex string which hast its lower left corner located in the specified coordinate. The "#splitline{}{}" construct allows to store multiple lines in the same `TLatex` object.
- Line 62: save the canvas as image. The format is automatically inferred from the file extension (it could have been eps, gif, ...).

Let's give a look to the obtained plot in figure 3.1. Beautiful outcome for such a small bunch of lines, isn't it?

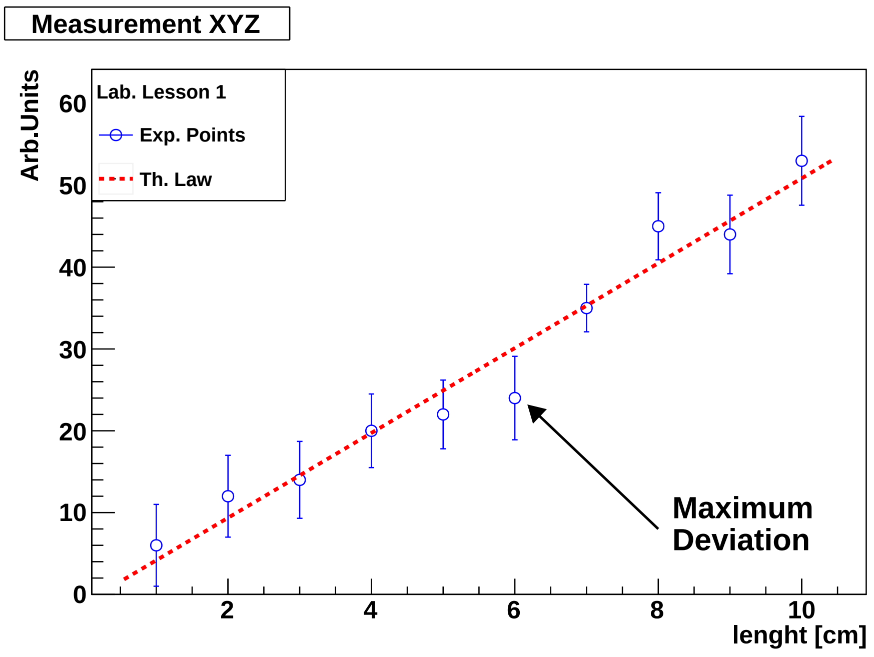


Figure 3.1.: Your first plot with data points.

A version of the same macro in PYTHON is available in the file `macro1.py`; you may want to open it in the editor and have a look at the differences right now - please consult the introductory sections of Chapter 8 first. This example shows how easy it is to change a ROOT macro from C++ to PYTHON.

3.3. Summary of Visual effects

3.3.1. Colours and Graph Markers

We have seen that to specify a colour, some identifiers like kWhite, kRed or kBlue can be specified for markers, lines, arrows etc. The complete summary of colours is represented by the ROOT "colour wheel", shown in appendix in figure B.1. To know more about the full story, refer to the online documentation of `TColor`.
ROOT provides an analogue of the colour wheel for the graphics markers. Select the most suited symbols for your plot (see Figure B.1) among dots, triangles, crosses or stars. An alternative set of names for the markers is summarised in Table B.1.

3.3.2. Arrows and Lines

The macro line 56 shows how to define an arrow and draw it. The class representing arrows is `TArrow`, which inherits from `TLine`. The constructors of lines and arrows always contain the coordinates of the endpoints. Arrows also foresee parameters to specify their shapes (see Figure B.2). Do not underestimate the role of lines and arrows in your plots. Since each plot should contain a message, it is convenient to stress it with additional graphics primitives.

3.3.3. Text

Also text plays a fundamental role in making the plots self-explanatory. A possibility to add text in your plot is provided by the `TLatex` class. The objects of this class are constructed with the coordinates of the bottom-left corner of the text and a string which contains the text itself. The real twist is that ordinary Latex mathematical symbols are automatically interpreted, you just need to replace the "\" by a "#" (see Figure B.3).

3.4. Interpretation and Compilation

As you observed, up to now we heavily exploited the capabilities of ROOT for interpreting our code, more than compiling and then executing. This is sufficient for a wide range of applications, but you might have already asked yourself "how can this code be compiled?". There are two answers.

3.4.1. Compile a Macro with ACLiC

ACLiC will create for you a compiled dynamic library for your macro, without any effort from your side, except the insertion of the appropriate header files in lines 3–9. In this example, they are already included. This does not harm, as they are not loaded by CINT. To generate an object libary from the macro code, from inside the interpreter type (please note the "+"):

```
 root [1] .L macro1.cxx+
```

Once this operation is accomplished, the macro symbols will be available in memory and you will be able to execute it simply by calling from inside the interpreter:

```
 root [2] macro1()
```

3.4.2. Compile a Macro with g++

In this case, you have to include the appropriate headers in the code and then exploit the *root-config* tool for the automatic settings of all the compiler flags. *root-config* is a script that comes with ROOT; it prints all flags and libraries needed to compile code and link it with the ROOT libraries. In order to make the code executable stand-alone, an entry point for the operating system is needed, in C++ this is the procedure `int main();`. The easiest way to turn a ROOT macro code into a stand-alone application is to add the following "dressing code" at the end of the macro file. This defines the procedure main, the only purpose of which is to call your macro:

```
#ifndef __CINT__
int main() {
  ExampleMacro();
  return 0;
}
#endif
```

Within ROOT, the symbol `__CINT__` is defined, and the code enclosed by `#ifndef __CINT__` and `#endif` is not executed; on the contrary, when running the system compiler `g++`, this symbol is not defined, and the code is compiled. To create a stand-alone program from a macro called `ExampleMacro.C`, simply type

```
 > g++ -o ExampleMacro.exe ExampleMacro.C `root-config --cflags --libs`
```

and execute it by typing

```
> ./ExampleMacro.exe
```

This procedure will, however, not give access to the ROOT graphics, as neither control of mouse or keyboard events nor access to the graphics windows of ROOT is available. If you want your stand-alone application have display graphics output and respond to mouse and keyboard, a slightly more complex piece of code can be used. In the example below, a macro `ExampleMacro_GUI` is executed by the ROOT class TApplication. As a further feature, this code example offers access to parameters eventually passed to the program when started from the command line. Here is the code fragment:

```
#ifndef __CINT__
void StandaloneApplication(int argc, char** argv) {
  // eventually, evaluate the application parameters argc, argv
  // ==>> here the ROOT macro is called
  ExampleMacro_GUI();
}
```

```
  // This is the standard "main" of C++ starting a ROOT application
int main(int argc, char** argv) {
   gROOT->Reset();
   TApplication app("Root Application", &argc, argv);
   StandaloneApplication(app.Argc(), app.Argv());
   app.Run();
   return 0;
}
#endif
```

Compile the code with

```
 > g++ -o ExampleMacro_GUI.exe ExampleMacro_GUI `root-config --cflags --libs`
```

and execute the program with

```
> ./ExampleMacro_GUI.exe
```

CHAPTER 4 GRAPHS

In this Chapter we will learn how to exploit some of the functionalities that ROOT provides to display data based on the class `TGraphErrors`, which you already got to know previously.

4.1. Read Graph Points from File

The fastest way in which you can fill a graph with experimental data is to use the constructor which reads data points and their errors from a file in ASCII (i. e. standard text) format:

```
TGraphErrors(const char *filename, const char *format="%lg %lg %lg %lg", ↩
    Option_t *option="");
```

The format string can be:

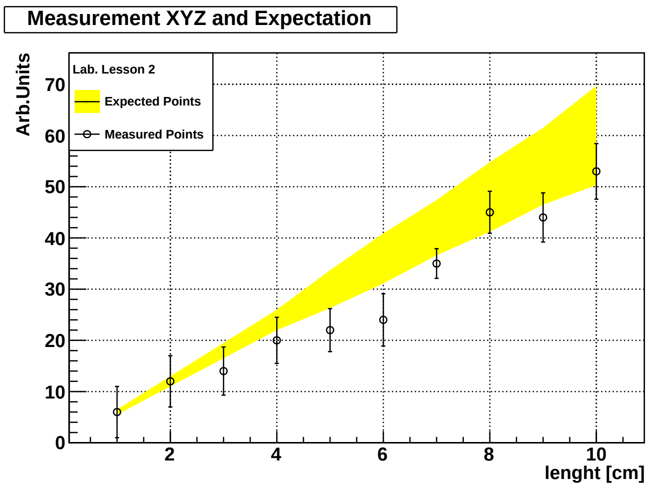
- `"\%lg \%lg"` read only 2 first columns into X,Y
- `"\%lg \%lg \%lg"` read only 3 first columns into X,Y and EY
- `"\%lg \%lg \%lg \%lg"` read only 4 first columns into X,Y,EX,EY

This approach has a the nice feature of allowing the user to reuse the macro for many different data sets. Here is an example of an input file. The nice graphic result shown is produced by the macro below, which reads two such input files and uses different options to display the data points.

```
# Measurement of Friday 26 March
# Experiment 2 Physics Lab

1   6   5
2   12  5
3   14  4.7
4   20  4.5
5   22  4.2
6   24  5.1
7   35  2.9
8   45  4.1
9   44  4.8
10  53  5.43
```

file: `macro2_input.txt`



```
/* Reads the points from a file and produces a simple graph. */
int macro2(){

    gROOT->SetStyle("Plain");
    TCanvas* c=new TCanvas();
    c.SetGrid();

    TGraphErrors graph_expected("./macro2_input_expected.txt","%lg %lg %lg");
    graph_expected.SetTitle("Measurement XYZ and Expectation;lenght [cm];Arb.Units");
    graph_expected.SetFillColor(kYellow);
    graph_expected.DrawClone("E3AL"); // E3 draws the band

    TGraphErrors graph("./macro2_input.txt","%lg %lg %lg");
    graph.SetMarkerStyle(kCircle);
    graph.SetFillColor(0);
    graph.DrawClone("PESame");

    // Draw the Legend
    TLegend leg(.1,.7,.3,.9,"Lab. Lesson 2");
    leg.SetFillColor(0);
    leg.AddEntry(&graph_expected,"Expected Points");
    leg.AddEntry(&graph,"Measured Points");
    leg.DrawClone("Same");

    c.Print("graph_with_band.pdf");
}
```

file: `macro2.cxx`

Beyond looking at the plot, you can check the actual contents of the graph with the `TGraph::Print()` method at any time, obtaining a printout of the coordinates of data points on screen. The macro also shows us how to print a coloured band around a graph instead of error bars, quite useful for example to represent the errors of a theoretical prediction.

4.2. Polar Graphs

With ROOT you can profit from rather advanced plotting routines, like the ones implemented in the `TPolarGraph`, a class to draw graphs in polar coordinates. It is very easy to use, as you see in the example macro and the resulting plot 4.1:

```
 1 /* Builds a polar graph in a square Canvas
 2 */
 3 void macro3(){
 4     double rmin=0;
 5     double rmax=TMath::Pi()*6;
 6     const int npoints=300;
 7     Double_t r[npoints];
 8     Double_t theta[npoints];
 9     for (Int_t ipt = 0; ipt < npoints; ipt++) {
10         r[ipt] = ipt*(rmax-rmin)/(npoints-1.)+rmin;
11         theta[ipt] = TMath::Sin(r[ipt]);
12     }
13     TCanvas* c = new TCanvas("myCanvas","myCanvas",600,600);
14     TGraphPolar grP1 (npoints,r,theta);
15     grP1.SetTitle("A Fan");
16     grP1.SetLineWidth(3);
17     grP1.SetLineColor(2);
18     grP1.DrawClone("AOL");
19 }
```

file: `macro3.cxx`

A new element was added on line 4, the size of the canvas: it is sometimes optically better to show plots in specific canvas sizes.

Some PYTHON variants of this macro are shown and discussed in Chapter 8.

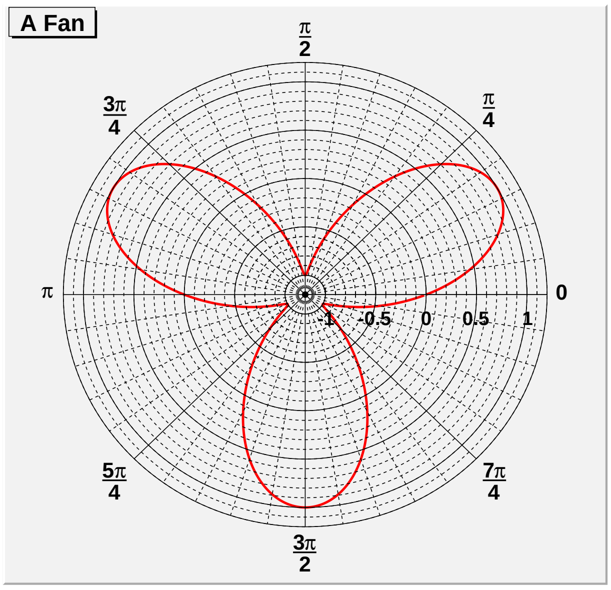


Figure 4.1.: The graph of a fan obtained with ROOT.

4.3. 2D Graphs

On some occasions it might be useful to plot some quantities versus two variables, therefore creating a bi-dimensional graph. Of course ROOT can help you in this task, with the `TGraph2DErrors` class. The following macro produces a bi-dimensional graph representing a hypothetical measurement, fits a bi-dimensional function to it and draws it together with its x and y projections. Some points of the code will be explained in detail. This time, the graph is populated with data points using random numbers, introducing a new and very important ingredient, the ROOT `TRandom3` random number generator using the Mersenne Twister algorithm [6].

```
/* Create, Draw and fit a TGraph2DErrors */
void macro4(){
    gStyle->SetPalette(1);
    gROOT->SetStyle("Plain");

    const double e = 0.3;
    const int nd = 500;

    TRandom3 my_random_generator;
    TF2  *f2 = new TF2("f2","1000*(([0]*sin(x)/x)*([1]*sin(y)/y))+200"↩
        ,-6,6,-6,6);
    f2->SetParameters(1,1);
    TGraph2DErrors *dte = new TGraph2DErrors(nd);
// Fill the 2D graph
    double rnd, x, y, z, ex, ey, ez;
    for (Int_t i=0; i<nd; i++) {
        f2->GetRandom2(x,y);
        rnd = my_random_generator.Uniform(-e,e); // A random number in [-e,e]
        z = f2->Eval(x,y)*(1+rnd);
        dte->SetPoint(i,x,y,z);
        ex = 0.05*my_random_generator.Uniform();
        ey = 0.05*my_random_generator.Uniform();
        ez = TMath::Abs(z*rnd);
        dte->SetPointError(i,ex,ey,ez);
    }
  // Fit function to generated data
    f2->SetParameters(0.7,1.5);  // set initial values for fit
    f2->SetTitle("Fitted 2D function");
    dte->Fit(f2);
29 // Plot the result
30     TCanvas *c1 = new TCanvas();
31     f2->Draw("Surf1");
32     dte->Draw("P0 Same");
33 // Make the x and y projections
34     TCanvas* c_p= new TCanvas("ProjCan"," The Projections",1000,400);
35     c_p->Divide(2,1);
36     c_p->cd(1);
37     dte->Project("x")->Draw();
38     c_p->cd(2);
39     dte->Project("y")->Draw();
40 }
```

file: `macro4.cxx`

- Line 3: This sets the palette colour code to a much nicer one than the default. Comment this line to give it a try.
- Line 4: sets a style type without fill color and shadows for pads. Looks much nicer on paper than the default setting.
- Line 9: The instance of the random generator. You can then draw out of this instance random numbers distributed according to different probability density functions, like the Uniform one at lines 25,26. See the on-line documentation to appreciate the full power of this ROOT feature.
- Line 10: You are already familiar with the `TF1` class. This is its two-dimensional correspondent. At line 21 two random numbers distributed according to the `TF2` formula are drawn with the method `TF2::GetRandom2(double& a, double&b)`.
- Line 26–28: Fitting a 2-dimensional function just works like in the one-dimensional case, i.e. initialisation of parameters and calling of the `Fit()` method.
- Line 31: The *Surf1* option draws the `TF2` objects (but also bi-dimensional histograms) as coloured surfaces with a wire-frame on three-dimensional canvases.
- Line 34–39: Here you learn how to create a canvas, partition it in two sub-pads and access them. It is very handy to show multiple plots in the same window or image.

CHAPTER 5 HISTOGRAMS

Histograms play a fundamental role in any type of Physics analysis, not only displaying measurements but being a powerful form of data reduction. ROOT presents many classes that represent histograms, all inheriting from the `TH1` class. We will focus in this chapter on uni- and bi- dimensional histograms whose bin-contents are represented by floating point numbers [1] , the `TH1F` and `TH2F` classes respectively.

5.1. Your First Histogram

Let's suppose that you want to measure the counts of a Geiger detector put in proximity of a radioactive source in a given time interval. This would give you an idea of the activity of your source. The count distribution in this case is a Poisson distribution. Let's see how operatively you can fill and draw a histogram in the following example macro.

```
/*Create, Fill and draw an Histogram which reproduces the
counts of a scaler linked to a Geiger counter.*/

void macro5(){
    TH1F* cnt_r_h=new TH1F("count_rate",
                "Count Rate;N_{Counts};# occurencies",
                100, // Number of Bins
                -0.5, // Lower X Boundary
                15.5); // Upper X Boundary

    const float mean_count=3.6;
    TRandom3 rndgen;
    // simulate the measurements
    for (int imeas=0;imeas<400;imeas++)
        cnt_r_h->Fill(rndgen.Poisson(mean_count));

    gROOT->SetStyle("Plain");
    TCanvas* c= new TCanvas();
    cnt_r_h->Draw();

    TCanvas* c_norm= new TCanvas();
    cnt_r_h->DrawNormalized();

    // Print summary
    cout << "Moments of Distribution:\n"
         << " - Mean = " << cnt_r_h->GetMean() << " +- "
                         << cnt_r_h->GetMeanError() << "\n"
         << " - RMS = " << cnt_r_h->GetRMS() << " +- "
                        << cnt_r_h->GetRMSError() << "\n"
         << " - Skewness = " << cnt_r_h->GetSkewness() << "\n"
         << " - Kurtosis = " << cnt_r_h->GetKurtosis() << "\n";
}
```

file: `macro5.cxx`

[1] To optimise the memory usage you might go for one byte (TH1C), short (TH1S), integer (TH1I) or double-precision (TH1D) bin-content.

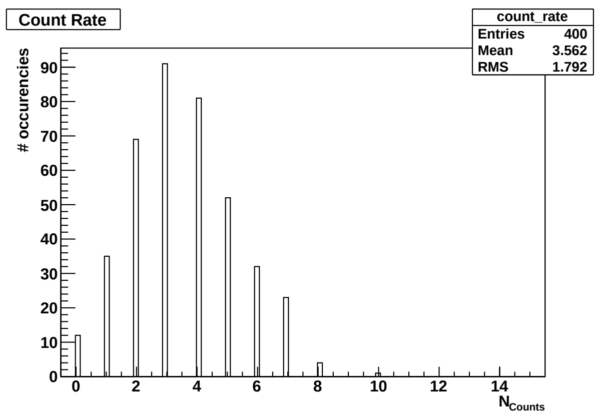


Figure 5.1.: The result of a counting (pseudo) experiment.

Which gives you the following plot 5.1: Using histograms is rather simple. The main differences with respect to graphs that emerge from the example are:

- line 5: The histograms have a name and a title right from the start, no predefined number of entries but a number of bins and a lower-upper range.
- line 15: An entry is stored in the histogram through the *TH1F::Fill* method.
- line 19 and 22: The histogram can be drawn also normalised, ROOT automatically takes cares of the necessary rescaling.
- line 25 to 31: This small snippet shows how easy it is to access the moments and associated errors of a histogram.

5.2. Add and Divide Histograms

Quite a large number of operations can be carried out with histograms. The most useful are addition and division. In the following macro we will learn how to manage these procedures within ROOT.

```
/*Divide and add 1D Histograms*/

void format_h(TH1F* h, int linecolor){
    h->SetLineWidth(3);
    h->SetLineColor(linecolor);
    }

void macro6(){
    gROOT->SetStyle("Plain");

    TH1F* sig_h=new TH1F("sig_h","Signal Histo",50,0,10);
    TH1F* gaus_h1=new TH1F("gaus_h1","Gauss Histo 1",30,0,10);
    TH1F* gaus_h2=new TH1F("gaus_h2","Gauss Histo 2",30,0,10);
    TH1F* bkg_h=new TH1F("exp_h","Exponential Histo",50,0,10);

    // simulate the measurements
    TRandom3 rndgen;
    for (int imeas=0;imeas<4000;imeas++){
        exp_h->Fill(rndgen.Exp(4));
        if (imeas%4==0) gaus_h1->Fill(rndgen.Gaus(5,2));
        if (imeas%4==0) gaus_h2->Fill(rndgen.Gaus(5,2));
        if (imeas%10==0)sig_h->Fill(rndgen.Gaus(5,.5));}

    // Format Histograms
    TH1F* histos[4]={sig_h,bkg_h,gaus_h1,gaus_h2};
    for (int i=0;i<4;++i){
        histos[i]->Sumw2(); // *Very* Important
```

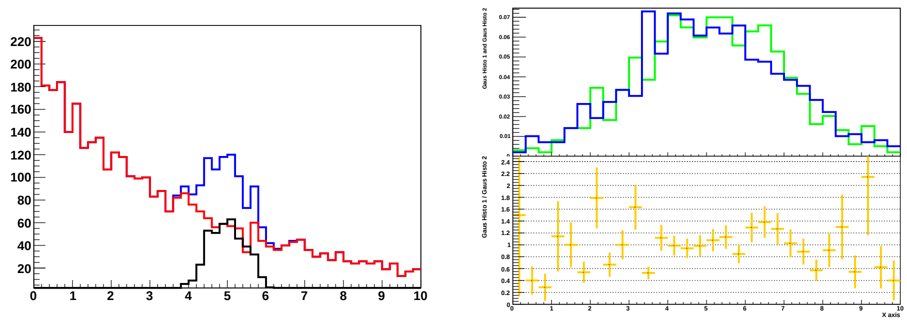


Figure 5.2.: The sum of two histograms and the ratio.

```
28            format_h(histos[i],i+1);
29            }
30
31        // Sum
32        TH1F* sum_h= new TH1F(*bkg_h);
33        sum_h->Add(sig_h,1.);
34        sum_h->SetTitle("Exponential + Gaussian");
35        format_h(sum_h,kBlue);
36
37        TCanvas* c_sum= new TCanvas();
38        sum_h->Draw("hist");
39        bkg_h->Draw("SameHist");
40        sig_h->Draw("SameHist");
41
42        // Divide
43        TH1F* dividend=new TH1F(*gaus_h1);
44        dividend->Divide(gaus_h2);
45
46        // Graphical Maquillage
47        dividend->SetTitle(";X axis;Gaus Histo 1 / Gaus Histo 2");
48        format_h(dividend,kOrange);
49        gaus_h1->SetTitle(";;Gaus Histo 1 and Gaus Histo 2");
50        gStyle->SetOptStat(0);
51        gStyle->SetOptTitle(0);
52
53        TCanvas* c_divide= new TCanvas();
54        c_divide->Divide(1,2,0,0);
55        c_divide->cd(1);
56        c_divide->GetPad(1)->SetRightMargin(.01);
57        gaus_h1->DrawNormalized("Hist");
58        gaus_h2->DrawNormalized("HistSame");
59        c_divide->cd(2);
60        dividend->GetYaxis()->SetRangeUser(0,2.49);
61        c_divide->GetPad(2)->SetGridy();
62        c_divide->GetPad(2)->SetRightMargin(.01);
63        dividend->Draw();
64    }
```

file: `macro6.cxx`

The plots that you will obtain are shown in 5.2 Some lines now need a bit of clarification:

- line 3: CINT, as we know, is also able to interpret more than one function per file. In this case the function simply sets up some parameters to conveniently set the line of histograms.
- line 20 to 22: Some contracted C++ syntax for conditional statements is used to fill the histograms with different numbers of entries inside the loop.
- line 27: This is a crucial step for the sum and ratio of histograms to handle errors properly. The method *TH1::Sumw2* causes the squares of weights to be stored inside the histogram (equivalent to the number of

entries per bin if weights of 1 are used). This information is needed to correctly calculate the errors of each bin entry when the methods *TH1::Add* and *TH1::Divide* are applied.

- line 33: The sum of two histograms. A weight can be assigned to the added histogram, for example to comfortably switch to subtraction.
- line 44: The division of two histograms is rather straightforward.
- line 53 to 63: When you draw two quantities and their ratios, it is much better if all the information is condensed in one single plot. These lines provide a skeleton to perform this operation.

5.3. Two-dimensional Histograms

Two-dimensional histograms are a very useful tool, for example to inspect correlations between variables. You can exploit the bi-dimensional histogram classes provided by ROOT in a very simple way. Let's see how in the following macro:

```
/* Draw a Bidimensional Histogram in many ways
together with its profiles and projections*/

void macro7(){
    gROOT->SetStyle("Plain");
    gStyle->SetPalette(1);
    gStyle->SetOptStat(0);
    gStyle->SetOptTitle(0);

    TH2F bidi_h("bidi_h",
           "2D Histo;Guassian Vals;Exp. Vals",
           30,-5,5,  // X axis
           30,0,10); // Y axis

    TRandom3 rndgen;
    for (int i=0;i<500000;i++)
        bidi_h.Fill(rndgen.Gaus(0,2),
                    10-rndgen.Exp(4));

    TCanvas* c=new TCanvas("Canvas","Canvas",800,800);
    c->Divide(2,2);
    c->cd(1);bidi_h.DrawClone("Contz");
    c->cd(2);bidi_h.DrawClone("Colz");
    c->cd(3);bidi_h.DrawClone("lego2");
    c->cd(4);bidi_h.DrawClone("surf3");

    // Profiles and Projections
    TCanvas* c2=new TCanvas("Canvas2","Canvas2",800,800);
    c2->Divide(2,2);
    c2->cd(1);bidi_h.ProjectionX()->DrawClone();
    c2->cd(2);bidi_h.ProjectionY()->DrawClone();
    c2->cd(3);bidi_h.ProfileX()->DrawClone();
    c2->cd(4);bidi_h.ProfileY()->DrawClone();
}
```

file: `macro macro7.cxx`

Two kinds of plots are provided by the code, the first one containing three-dimensional representations (Figure 5.3) and the second one projections and profiles (5.4) of the bi-dimensional histogram. When a projection is performed along the x (y) direction, for every bin along the x (y) axis, all bin contents along the y (x) axis are summed up (upper the plots of figure 5.4). When a profile is performed along the x (y) direction, for every bin along the x (y) axis, the average of all the bin contents along the y (x) is calculated together with their RMS and displayed as a symbol with error bar (lower two plots of figure). 5.4).

Correlations between the variables are quantified by the methods `Double_T GetCovariance()` and `Double_t GetCorrelationFactor()`.

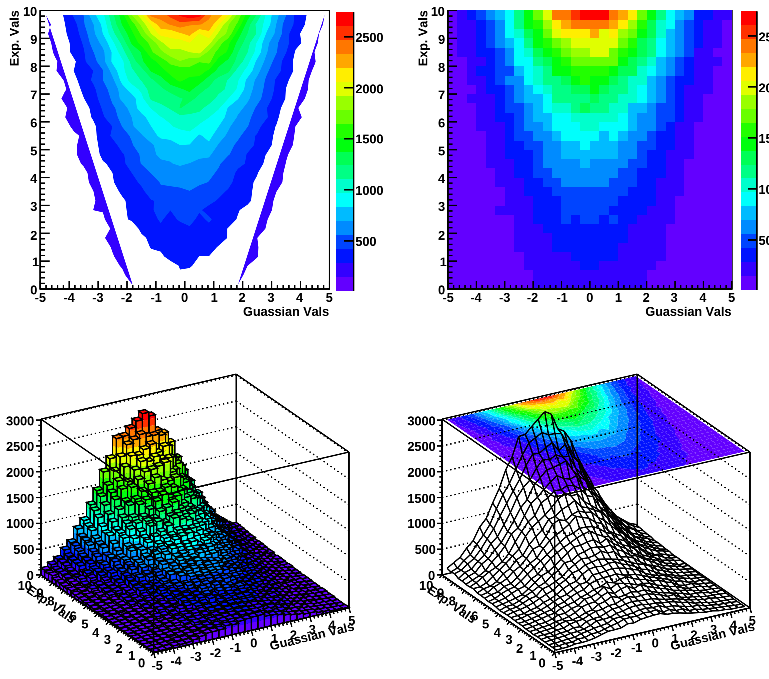


Figure 5.3.: Different ways of representing bi-dimensional histograms.

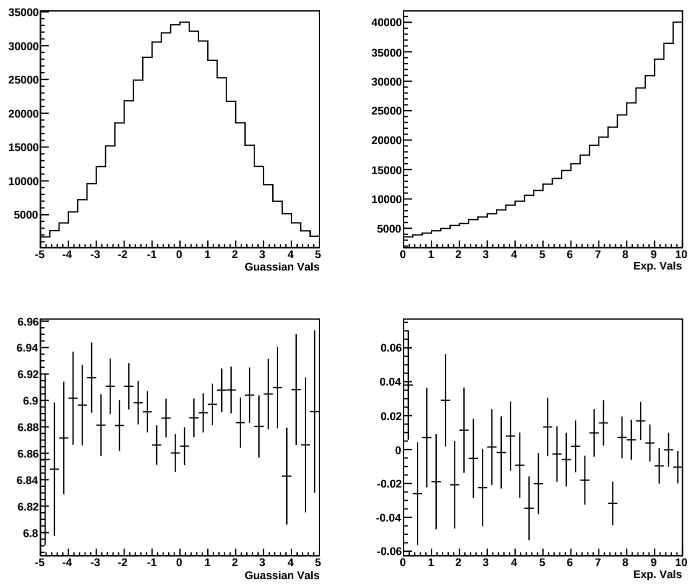


Figure 5.4.: The projections and profiles of bi-dimensional histograms.

CHAPTER 6 FILE I/O

6.1. Storing ROOT Objects

ROOT offers the possibility to write the instances of all the classes inheriting from the class `TObject` (basically all classes in ROOT) on disk, into what is referred to as *ROOT-file*, a file created by the `TFile` class. One says that the object is made "persistent" by storing it on disk. When reading the file back, the object can be restored to memory.

We can explore this functionality with histograms and two simple macros.

```
void write_to_file(){

    // Istance of our histogram
    TH1F h("my_histogram","My Title;X;# of entries",100,-5,5);

    // Let's fill it randomly
    h.FillRandom("gaus");

    // Let's open a TFile
    TFile out_file("my_rootfile.root","RECREATE");

    // Write the histogram in the file
    h.Write();

    // Close the file
    out_file.Close();
}
```

file: `write_to_file.cxx`

The *RECREATE* option forces ROOT to create a new file even if a file with the same name exists on disk.

Now, you may use the CINT command line to access information in the file and draw the previously written histogram:

```
>>>  root my_rootfile.root
root [0]
Attaching file my_rootfile.root as _file0...
root [1] _file0.ls()
TFile**         my_rootfile.root
 TFile*         my_rootfile.root
  KEY: TH1F     my_histogram;1  My Title
root [2] my_histogram.Draw()
```

Alternatively, you can use a simple macro to carry out the job:

```
void read_from_file(){

    // Let's open the TFile
    TFile* in_file= new TFile("my_rootfile.root");

    // Get the Histogram out
    TH1F* h = in_file.GetObjectChecked("my_histogram","TH1F");

    // Draw it
    h->DrawClone();

}
```

file: `read_from_file.cxx`

Please note that the order of opening files for write access and creating objects determines whether the objects are stored or not. You can avoid this behaviour by using the `Write()` function as shown in the previous example.

Although you could access an object within a file also with the `Get` function and a dynamic type cast, it is advisable to use `GetObjectChecked`.

6.2. N-tuples in ROOT

6.2.1. Storing simple N-tuples

Up to now we have seen how to manipulate input read from ASCII files. ROOT offers the possibility to do much better than that, with its own n-tuple classes. Among the many advantages provided by these classes one could cite

- Optimised disk I/O.
- Possibility to store many n-tuple rows (Millions).
- Write the n-tuples in ROOT files.
- Interactive inspection with `TBrowser`.
- Store not only numbers, but also *objects* in the columns.

In this section we will discuss briefly the `TNtuple` class, which is a simplified version of the `TTree` class. A ROOT `TNtuple` object can store rows of float entries. Let's tackle the problem according to the usual strategy commenting a minimal example

```
/*
Fill an n-tuple and write it to a file simulating measurement of
conductivity of a material in different conditions of pressure and temperature.
*/

void write_ntuple_to_file(){

    // Initialise the TNtuple
    TNtuple cond_data("cond_data",
                      "Example N-Tuple",
                      "Potential:Current:Temperature:Pressure");

    // Fill it randomly to fake the acquired data
    float pot,cur,temp,pres;
    for (int i=0;i<10000;++i){
        pot=gRandom->Uniform(0.,10.);      // get voltage
        temp=gRandom->Uniform(250.,350.);  // get temperature
        pres=gRandom->Uniform(0.5,1.5);    // get pressure
        cur=pot/(10.+0.05*(temp-300.)-0.2*(pres-1.)); //calculate current
// add some random smearing (measurement errors)
        pot*=gRandom->Gaus(1.,0.01);       // 1% error on voltage
        temp+=gRandom->Gaus(0.,0.3);       // 0.3 absolute error on temperature
        pres*=gRandom->Gaus(1.,0.02);      // 1% error on pressure
        cur*=gRandom->Gaus(1.,0.01);       // 1% error on current
// write to ntuple
        cond_data.Fill(pot,cur,temp,pres);
        }

    // Open a file, save the ntuple and close the file
    TFile ofile("conductivity_experiment.root","RECREATE");
    cond_data.Write();
    ofile.Close();
    }
```

file: `write_ntuple_to_file.cxx`

This data written to this example n-tuple represents, in the statistical sense, three independent variables (Potential or Voltage, Pressure and Temperature), and one variable (Current) which depends on the the others according to very simple laws, and an additional Gaussian smearing. This set of variables mimics a measurement of an electrical resistance while varying pressure and temperature.

Imagine your task now consists in finding the relations among the variables – of course without knowing the code used to generate them. You will see that the possibilities of the `NTuple` class enable you to perform this analysis task. Open the ROOT file (`cond_data.root`) written by the macro above in an interactive section and use a `TBrowser` to interactively inspect it:

```
root[0] new TBrowser()
```

You find the columns of your n-tuple written as *leafs*. Simply clicking on them you can obtain histograms of the variables!

Next, try the following commands at the shell prompt and in the interactive ROOT shell, respectively:

```
> root conductivity_experiment.root
Attaching file conductivity_experiment.root as _file0...
root [0] cond_data.Draw("Current:Potential")
```

You just produced a correlation plot with one single line of code!

Try to extend the syntax typing for example

```
root [1] cond_data.Draw("Current:Potential","Temperature<270")
```

What do you obtain?

Now try

```
root [2] cond_data.Draw("Current/Potential:Temperature")
```

It should have become clear from these examples how to navigate in such a multi-dimensional space of variables and uncover relations between variables using n-tuples.

6.2.2. Reading N-tuples

For completeness, you find here a small macro to read the data back from a ROOT n-tuple

```
/*
Read the previously produced N-Tuple and print on screen its content
*/

void read_ntuple_from_file(){

    // Open a file, save the ntuple and close the file
    TFile in_file("conductivity_experiment.root");
    TNtuple* my_tuple = in_file.GetObjectChecked("cond_data","TNtuple");

    float pot,cur,temp,pres;
    float* row_content;

    cout << "Potential\tCurrent\tTemperature\tPressure\n";
    for (int irow=0;irow<my_tuple->GetEntries();++irow){
        my_tuple->GetEntry(irow);
        row_content = my_tuple->GetArgs();
        pot = row_content[0];
        cur = row_content[1];
        temp = row_content[2];
        pres = row_content[3];
        cout << pot << "\t" << cur << "\t" << temp << "\t" << pres << endl;
        }

    }
```

file: `read_ntuple_from_file.cxx`

The macro shows the easiest way of accessing the content of a n-tuple: after loading the n-tuple, its branches are assigned to variables and `GetEntry(long)` automatically fills them with the content for a specific row. By doing so, the logic for reading the n-tuple and the code to process it can be split and the source code remains clear.

6.2.3. Storing Arbitrary N-tuples

It is also possible to write n-tuples of arbitrary type by using ROOT's `TBranch` class. This is especially important as `TNtuple::Fill()` accepts only floats. The following macro creates the ame n-tuple as before but the branches are booked directly. The `Fill()` function then fills the current values of the connected variables to the tree.

```
/*
Fill an n-tuple and write it to a file simulating measurement of
conductivity of a material in different conditions of pressure and temperature.
using branches
*/

void write_ntuple_to_file_advanced(std::string outputFileName = "↩
  conductivity_experiment.root", unsigned int numDataPoints = 10000){
  // Initialise the TNtuple
  TTree cond_data("cond_data", "Example N-Tuple");

  // define the variables and book them for the ntuple
  float pot,cur,temp,pres;
  cond_data.Branch("Potential", &pot, "Potential/F");
  cond_data.Branch("Current", &cur, "Current/F");
  cond_data.Branch("Temperature", &temp, "Temperature/F");
  cond_data.Branch("Pressure", &pres, "Pressure/F");

  for (int i=0;i<numDataPoints;++i){
    // Fill it randomly to fake the acquired data
    pot=gRandom->Uniform(0.,10.)*gRandom->Gaus(1.,0.01);
    temp=gRandom->Uniform(250.,350.)+gRandom->Gaus(0.,0.3);
    pres=gRandom->Uniform(0.5,1.5)*gRandom->Gaus(1.,0.02);
    cur=pot/(10.+0.05*(temp-300.)-0.2*(pres-1.))*gRandom->Gaus(1.,0.01);

    // write to ntuple
    cond_data.Fill();
    }

  // Open a file, save the ntuple and close the file
  TFile ofile(outputFileName.c_str(),"RECREATE");
  cond_data.Write();
  ofile.Close();
}
```

file: `write_ntuple_to_file_advanced.cxx`

The `Branch()` function requires a pointer to a variable and a definition of the variable type. Table 6.1 lists some of the possible values. Please note that ROOT is not checking the input and mistakes are likely to result in serious problems. This holds especially if values are read as another type than they have been written, e.g. when storing a variable as float and reading it as double.

Table 6.1.: List of variable types that can be used to define the type of a branch in ROOT.

type	size	C++	identifier
signed integer	32 bit	int	I
	64 bit	long	L
unsigned integer	32 bit	unsigned int	i
	64 bit	unsigned long	l
floating point	32 bit	float	F
	64 bit	double	D
boolean	-	bool	O

6.2.4. Processing N-tuples Spanning over Several Files

Usually n-tuples or trees span over many files and it would be difficult to add them manually. ROOT thus kindly provides a helper class in the form of `TChain`. Its usage is shown in the following macro which is very similar to the previous example. The constructor of a `TChain` takes the name of the `TTree` (or `TNuple`) as an argument. The files are added with the function `Add(fileName)`, where one can also use wild-cards as shown in the example.

```
/*
  Read several previously produced N-Tuples and print on screen its content

  you can easily create some files with the following statement:
  for i in 0 1 2 3 4 5; do root -l -x -b -q "write_ntuple_to_file.cxx(\"conductivity_experiment_${i}.root\", 100)"; done
*/

void read_ntuple_with_chain(){
  // initiate a TChain with the name of the TTree to be processed
  TChain in_chain("cond_data");
  in_chain.Add("conductivity_experiment*.root"); // add files , wildcards work

  // define variables and assign them to the corresponding branches
  float pot, cur, temp, pres;
  my_tuple->SetBranchAddress("Potential", &pot);
  my_tuple->SetBranchAddress("Current", &cur);
  my_tuple->SetBranchAddress("Temperature", &temp);
  my_tuple->SetBranchAddress("Pressure", &pres);

  cout << "Potential\tCurrent\tTemperature\tPressure\n";
  for (size_t irow=0; irow<in_chain.GetEntries(); ++irow){
    in_chain.GetEntry(irow); // loads all variables that have been connected to branches
    cout << pot << "\t" << cur << "\t" << temp << "\t" << pres << endl;
  }
}
```

file: `read_ntuple_with_chain.cxx`

6.2.5. *For the advanced user:* Processing trees with a selector script

Another very general and powerful way of processing a `TChain` is provided via the method `TChain::Process()`. This method takes as arguments an instance of a – user-implemented– class of type `TSelector`, and – optionally – the number of entries and the first entry to be processed. A template for the class `TSelector` is provided by the method `TTree::MakeSelector`, as is shown in the little macro `makeSelector.C` below.

It opens the n-tuple `conductivity_experiment.root` from the example above and creates from it the header file `MySelector.h` and a template to insert your own analysis code, `MySelector.C`.

```
{
```

```
// create template class for Selector to run on a tree
//////////////////////////////////////////////////////
//
// open root file containing the Tree
  TFile *f = TFile::Open("conductivity_experiment.root");
// create TTree object from it
  TTree *t = (TTree *) f->Get("cond_data");
// this generates the files MySelector.h and MySelector.C
  t->MakeSelector("MySelector");
  }
```

file: `makeMySelector.C`

The template contains the entry points `Begin()` and `SlaveBegin()` called before processing of the `TChain` starts, `Process()` called for every entry of the chain, and `SlaveTerminate()` and `Terminate()` called after the last entry has been processed. Typically, initialization like booking of histograms is performed in `SlaveBegin()`, the analysis, i. e. the selection of entries, calculations and filling of histograms, is done in `Process()`, and final operations like plotting and storing of results happen in `SlaveTerminate()` or `Terminate()`.

The entry points `SlaveBegin()` and `SlaveTerminate()` are called on so-called slave nodes only if parallel processing via `PROOF` or `PROOF lite` is enabled, as will be explained below.

A simple example of a selector class is shown in the macro `MySelector.C`. The example is executed with the following sequence of commands:

```
> TChain *ch=new TChain("cond_data", "My Chain for Example N-Tuple");
> ch->Add("conductivity_experiment*.root");
> ch->Process("MySelector.C+");
```

As usual, the "+" appended to the name of the macro to be executed initiates the compilation of the `MySelector.C` with the system compiler in order to improve performance.

The code in `MySelector.C`, shown in the listing below, books some histograms in `SlaveBegin()` and adds them to the instance `fOutput`, which is of the class `TList`[1] The final processing in `Terminate()` allows to access histograms and store, display or save them as pictures. This is shown in the example via the `TList fOutput`. See the commented listing below for more details; most of the text is actually comments generated automatically by `TTree::MakeSelector`.

```
#define MySelector_cxx
// The class definition in MySelector.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("MySelector.C")
// Root > T->Process("MySelector.C","some options")
// Root > T->Process("MySelector.C+")
//

#include "MySelector.h"
#include <TH2.h>
```

[1]The usage of `fOutput` is not really needed for this simple example, but it allows re-usage of the exact code in parallel processing with `PROOF` (see next section).

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

// user defined variables may come here:
UInt_t fNumberOfEvents; TDatime tBegin, tNow;

TH1F *h_pot,*h_cur,*h_temp,*h_pres,*h_resistance;

void MySelector::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();

   // some time measurement
   tBegin.Set(); printf("*==* ---------- Begin of Job ---------- ");
   tBegin.Print();
}

void MySelector::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();

   //book some histograms
   h_pot=new TH1F("pot","potential",100,-0.5,10.5);
   h_cur=new TH1F("cur","current",100,-0.1,1.5);
   h_temp=new TH1F("temp","temperature",100,200.,400.);
   h_pres=new TH1F("pres","pressure",100,-0.,2.);
   h_resistance=new TH1F("resistance","resistance",100,5.,15.);

 // add all booked histograms to output list (only really needed for PROOF)
   fOutput->AddAll(gDirectory->GetList());
}

Bool_t MySelector::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 MySelector::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.

// - - - - - - - - - begin processing
   GetEntry(entry);

 // count number of entries (=events) ...
   ++fNumberOfEvents;

 // analsiys code comes here - fill histograms
   h_pot->Fill(Potential);
   h_cur->Fill(Current);
   h_temp->Fill(Temperature);
   h_pres->Fill(Pressure);
   h_resistance->Fill(Potential/Current);

   return kTRUE;  //kFALSE would abort processing
}

void MySelector::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.

  // some statistics at end of job
  printf("\n *==* ---------- End of Slave Job ----------   ");
  tNow.Set(); tNow.Print();
  printf("Number of Events: %i, elapsed time: %i sec, rate: %g evts/sec\n",
   fNumberOfEvents,
         tNow.Convert()-tBegin.Convert(),
         float(fNumberOfEvents)/(tNow.Convert()-tBegin.Convert()) );
}

void MySelector::Terminate()
{
   // The Terminate() function is the last function to be called during
   // a query. It always runs on the client, it can be used to present
   // the results graphically or save the results to file.

// finally, store all output
   TFile hfile("MySelector_Result.root","RECREATE","MuonResults");
   fOutput->Write();

   //Example to retrieve output from output list
   h_resistance=dynamic_cast<TH1F *>(fOutput->FindObject("resistance"));
   TCanvas c_result("cresult","Resistance",100,100,300,300);
   h_resistance->Draw();
   c_result.SaveAs("ResistanceDistribution.png");

   tNow.Set(); printf("*==* ---------- End of Job ---------- ");
   tNow.Print();
}
```

file: MySelector.C

6.2.6. *For power-users:* Multi-core processing with PROOF lite

The processing of n-tuples via a selector function of type `TSelector` through `TChain::Process()`, as described at the end of the previous section, offers an additional advantage in particular for very large data sets: on distributed systems or multi-core architectures, portions of data can be processed in parallel, thus significantly reducing the execution time. On modern computers with multi-core CPUs or hyper-threading enabled, this allows a much faster turnaround of analyses, since all the available CPU power is used.

On distributed systems, a PROOF server and worker nodes have to be set up, as is described in detail in the ROOT documentation. On a single computer with multiple cores, `PROOF lite` can be used instead. Try the following little macro, `RunMySelector.C`, which contains two extra lines compared to the example above (adjust the number of workers according to the number of CPU cores):

```
{
```

```
// set up a TChain
TChain *ch=new TChain("cond_data", "My Chain for Example N-Tuple");
 ch->Add("conductivity_experiment*.root");
//
// eventually, start Proof Lite on cores
TProof::Open("workers=4");
ch->SetProof();
//
ch->Process("MySelector.C+");
}
```

file: `RunMySelector.C`

The first command, `TProof::Open()` starts a local PROOF server, and the command `ch->SetProof();` enables processing of the chain using PROOF. Now, when issuing the command `ch->Process("MySelector.C+);`, the code in `MySelector.C` is compiled and executed on each slave node. The methods `Begin()` and `Terminate()` are executed on the master only. The list of n-tuple files is analysed, and portions of the data are assigned to the available slave processes. Histograms booked in `SlaveBegin()` exist in the processes on the slave nodes, and are filled accordingly. Upon termination, the PROOF master collects the histograms from the slaves and merges them. In `Terminate()` all merged histograms are available and can be inspected, analysed or stored. The histograms are handled via the instances `fOutput` of class `TList` in each slave process, and can be retrieved from this list after merging in `Terminate`.

To explore the power of this mechanism, generate some very large n-tuples using the script from Section 6.2.3 - you could try 10 000 000 events (this results in a large n-tuple of about 160 MByte in size). You could also generate a large number of files and use wildcards to add the to the `TCHain`. Now execute

```
> root -l RunMySelector.C
```

and watch what happens:

```
Processing RunMySelector.C...
 +++ Starting PROOF-Lite with 4 workers +++
Opening connections to workers: OK (4 workers)
Setting up worker servers: OK (4 workers)
PROOF set to parallel mode (4 workers)

Info in <TProofLite::SetQueryRunning>: starting query: 1
Info in <TProofQueryResult::SetRunning>: nwrks: 4
Info in <TUnixSystem::ACLiC>: creating shared library ~/DivingROOT/macros/↩
    MySelector_C.so
*==* ---------- Begin of Job ---------- Date/Time = Wed Feb 15 23:00:04 2012
Looking up for exact location of files: OK (4 files)
Looking up for exact location of files: OK (4 files)
Info in <TPacketizerAdaptive::TPacketizerAdaptive>: Setting max number of ↩
    workers per node to 4
Validating files: OK (4 files)
Info in <TPacketizerAdaptive::InitStats>: fraction of remote files 1.000000
Info in <TCanvas::Print>: file ResistanceDistribution.png has been created
*==* ---------- End of Job ---------- Date/Time = Wed Feb 15 23:00:08 2012
Lite-0: all output objects have been merged
```

Log files of the whole processing chain are kept in the directory `~.proof` for each worker node. This is very helpful for debugging or if something goes wrong. As the the method described here also works without using PROOF, the development work on an analysis script can be done in the standard way on a small subset of the data, and only for the full processing one would use parallelism via PROOF.

6.2.7. Optimisation Regarding N-tuples

ROOT automatically applies compression algorithms on n-tuples to reduce the memory consumption. A value that is in most cases only zero will consume only small space on your disk (but it has to be deflated on reading). Nevertheless, you should think about the design of your n-tuples and your analyses as soon as the processing time exceeds some minutes.

- Try to keep your n-tuples simple and use appropriate variable types. If your measurement has only a limited precision, it is needless to store it with double precision.
- Experimental conditions that do not change with every single measurement should be stored in a separate tree. Although the compression can handle redundant values, the processing time increase with every variable that has to be filled.

- The function `SetCacheSize(long)` specifies the size of the cache for reading a `TTree` object from a file. The default value is 30MB. A manual increase may help in certain situations. Please note that the caching mechanism can cover only one `TTree` object per `TFile` object.
- You can select the branches to be covered by the caching algorithm with `AddBranchToCache` and deactivate unneeded branches with `SetBranchStatus`. This mechanism can result in a significant speed-up for simple operations on trees with many branches.
- You can measure the performance easily with `TTreePerfStats`. The ROOT documentation on this class also includes an introductory example. For example, `TTreePerfStats` can show you that it is beneficial to store meta data and payload data separately, i. e. write the meta data tree in a bulk to a file at the end of your job instead of writing both trees interleaved.

CHAPTER 7 FUNCTIONS AND PARAMETER ESTIMATION

After going through the previous chapters, you already know how to use mathematical functions (class TF1), and you got some insight into the graph (TGraphErrors) and histogram classes (TH1F) for data visualisation. In this chapter we will add more detail to the previous approximate explanations to face the fundamental topic of parameter estimation by fitting functions to data. For graphs and histograms, ROOT offers an easy-to-use interface to perform fits - either the fit panel of the graphical interface, or the Fit method. The class TVirtualFitter allows access to the detailed results, and can also be used for more general tasks with user-defined minimisation functions.

Very often it is necessary to study the statistical properties of analysis procedures. This is most easily achieved by applying the analysis to many sets of simulated data (or "pseudo data"), each representing one possible version of the true experiment. If the simulation only deals with the final distributions observed in data, and does not perform a full simulation of the underlying physics and the experimental apparatus, the name "Toy Monte Carlo" is frequently used[1]. Since the true values of all parameters are known in the pseudo-data, the differences between the parameter estimates from the analysis procedure w. r. t. the true values can be determined, and it is also possible to check that the analysis procedure provides correct error estimates.

7.1. Fitting Functions to Pseudo Data

In the example below, a pseudo-data set is produced and a model fitted to it.

ROOT offers various fit methods, all inheriting from a virtual class TVirtualFitter. The default fitter in ROOT is MINUIT, a classical fitting package originally implemented in the FORTRAN programming language. Recently, a C++ version, MINUIT2, has been added, and the new package FUMILI. All of these methods determine the best-fit parameters, their errors and correlations by minimising a χ^2 or a negative log-likelihood function. A pointer to the active fitting method is accessible via an instance of class TVirtualFitter. Methods of this class allow to set initial values or allowed ranges for the fit parameters, provide means for fixing and releasing of parameters and offer steering options for the numerical precision, and - most importantly - allow to retrieve the status of the fit upon completion and the fit results. The documentation of the class TVirtualFitter gives a list of all currently implemented methods.

The complication level of the code below is intentionally a little higher than in the previous examples. The graphical output of the macro is shown in Figure 7.1:

```
/* Define and play with TF1s */

void format_line(TAttLine* line,int col,int sty){
    line->SetLineWidth(5);
    line->SetLineColor(col);
    line->SetLineStyle(sty);}

double the_gausppar(double* vars, double* pars){
    return pars[0]*TMath::Gaus(vars[0],pars[1],pars[2])+
        pars[3]+pars[4]*vars[0]+pars[5]*vars[0]*vars[0];}

```

[1] "Monte Carlo" simulation means that random numbers play a role here which is as crucial as in games of pure chance in the Casino of Monte Carlo.

```
12 int macro8(){
13     gROOT->SetStyle("Plain");
14     gStyle->SetOptTitle(0);
15     gStyle->SetOptStat(0);
16     gStyle->SetOptFit(1111);
17     gStyle->SetStatX(.89); gStyle->SetStatY(.89);
18     gStyle->SetStatBorderSize(0);
19 
20     TF1 parabola("parabola","[0]+[1]*x+[2]*x**2",0,20);
21     format_line(&parabola,kBlue,2);
22 
23     TF1 gaussian("gaussian","[0]*TMath::Gaus(x,[1],[2])",0,20);
24     format_line(&gaussian,kRed,2);
25 
26     TF1 gausppar("gausppar",the_gausppar,-0,20,6);
27     double a=15; double b=-1.2; double c=.03;
28     double norm=4; double mean=7; double sigma=1;
29     gausppar.SetParameters(norm,mean,sigma,a,b,c);
30     gausppar.SetParNames("Norm","Mean","Sigma","a","b","c");
31     format_line(&gausppar,kBlue,1);
32 
33     TH1F histo("histo",
34                "Signal plus background;X vals;Y Vals",
35                50,0,20);
36     histo.SetMarkerStyle(8);
37 
38     // Fake the data
39     for (int i=1;i<=5000;++i)
40         histo.Fill(gausppar.GetRandom());
41 
42     /* Reset the parameters before the fit and set
43     by eye a peak at 6 with an area of more or less 50 */
44     gausppar.SetParameter(0,50);
45     gausppar.SetParameter(1,6);
46     int npar=gausppar.GetNpar();
47     for (int ipar=2;ipar<npar;++ipar)
48        gausppar.SetParameter(ipar,1);
49 
50     // perform fit ...
51     histo.Fit(&gausppar);
52 
53     // ... and retrieve fit results
54     TVirtualFitter *fit = TVirtualFitter::GetFitter(); // get fit method
55     fit->PrintResults(2,0.); // print fit results
56     // get covariance Matrix an print it
57     TMatrixD *covMatrix = new TMatrixD(npar,npar,fit->GetCovarianceMatrix());
58     covMatrix->Print();
59 
60     // Set the values of the gaussian and parabola
61     for (int ipar=0;ipar<3;ipar++){
62         gaussian.SetParameter(ipar,gausppar.GetParameter(ipar));
63         parabola.SetParameter(ipar,gausppar.GetParameter(ipar+3));}
64 
65     histo.GetYaxis()->SetRangeUser(0,250);
66     histo.DrawClone("PE");
67     parabola.DrawClone("Same"); gaussian.DrawClone("Same");
68     TLatex latex(2,220,"#splitline{Signal Peak over}{background}");
69     latex.DrawClone("Same");
70 }
```

file:`macro8.cxx`

- Line 3-6: A simple function to ease the make-up of lines. Remember that the class `TF1` inherits from

`TAttLine`.

- Line 8-10: Definition of a customised function, namely a Gaussian (the "signal") plus a parabolic function, the "background".
- Line 13-18: Some maquillage for the Canvas. In particular we want that the parameters of the fit appear very clearly and nicely on the plot.
- Line 26-31: define and initialise an instance of `TF1`.
- Line 33-40: define and fill a histogram.
- Line 42-48: for convenience, the same function as for the generation of the pseudo-data is used in the fit; hence, we need to reset the function parameters. This part of the code is very important for each fit procedure, as it sets the initial values of the fit.
- Line 51: A very simple command, well known by now: fit the function to the histogram.
- Line 53–58: retrieve the output from the fit Here, we simply print the fit result and access and print the covariance matrix of the parameters.
- Line 60–end: plot the pseudo-data, the fitted function and the signal and background components at the best-fit values.

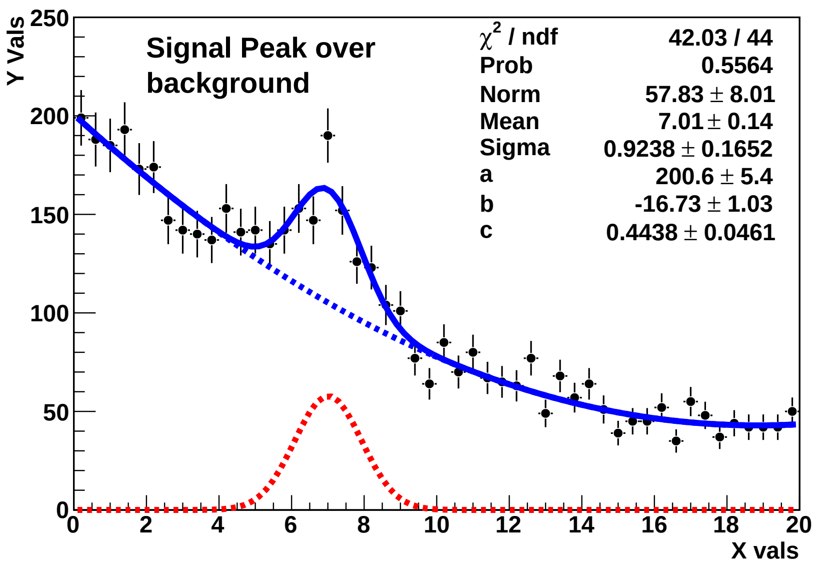


Figure 7.1.: Function fit to pseudo-data

7.2. Toy Monte Carlo Experiments

Let us look at a simple example of a toy experiment comparing two methods to fit a function to a histogram, the χ^2 method and a method called "binned log-likelihood fit", both available in ROOT.

As a very simple yet powerful quantity to check the quality of the fit results, we construct for each pseudo-data set the so-called "pull", the difference of the estimated and the true value of a parameter, normalised to the estimated error on the parameter, $(p_{estim} - p_{true})/\sigma_p$. If everything is OK, the distribution of the pull values is a standard normal distribution, i. e. a Gaussian distribution centred around zero with a standard deviation of one.

The macro performs a rather big number of toy experiments, where a histogram is repeatedly filled with Gaussian distributed numbers, representing the pseudo-data in this example. Each time, a fit is performed according to the selected method, and the pull is calculated and filled into a histogram. Here is the code:

```
/* Toy Monte Carlo example
   check pull distribution to compare chi2 and binned log-likelihood methods
*/
pull( int n_toys = 10000,
      int n_tot_entries = 100,
        int nbins = 40,
        bool do_chi2=true ){

    gROOT->SetStyle("Plain");

    TString method_prefix("Log-Likelihood ");
    if (do_chi2)
        method_prefix="#chi^{2} ";

    // Create histo
    TH1F* h4 = new TH1F(method_prefix+"h4",method_prefix+" Random Gauss",nbins↩
        ,-4,4);
    h4->SetMarkerStyle(21);
    h4->SetMarkerSize(0.8);
    h4->SetMarkerColor(kRed);

    // Histogram for sigma and pull
    TH1F* sigma = new TH1F(method_prefix+"sigma",method_prefix+"sigma from gaus ↩
        fit",50,0.5,1.5);
    TH1F* pull = new TH1F(method_prefix+"pull",method_prefix+"pull from gaus fit↩
        ",50,-4.,4.);

    // Make nice canvases
    TCanvas* c0 = new TCanvas(method_prefix+"Gauss",method_prefix+"Gauss"↩
        ,0,0,320,240);
    c0->SetGrid();

    // Make nice canvases
    TCanvas* c1 = new TCanvas(method_prefix+"Result",method_prefix+"Sigma-↩
        Distribution",0,300,600,400);

    c0->cd();

    float sig, mean;
    for (int i=0; i<n_toys; i++){
     // Reset histo contents
        h4->Reset();
     // Fill histo
        for ( int j = 0; j<n_tot_entries; j++ )
        h4->Fill(gRandom->Gaus());
     // perform fit
        if (do_chi2) h4->Fit("gaus","q"); // Chi2 fit
        else h4->Fit("gaus","lq"); // Likelihood fit
     // some control output on the way
        if (!(i%100)){
            h4->Draw("EP");
            c0->Update();
        }
     // Get sigma from fit
        TF1 *fitfunc = h4->GetFunction("gaus");
        sig = fitfunc->GetParameter(2);
        mean= fitfunc->GetParameter(1);
        sigma->Fill(sig);
        pull->Fill(mean/sig * sqrt(n_tot_entries));
       } // end of toy MC loop
     // print result
        c1->cd();
        pull->Fit("gaus");
        pull->Draw("EP");
        c1->Update();
}

void macro9(){
```

```
64     int n_toys=10000;
65     int n_tot_entries=100;
66     int n_bins=40;
67     cout << "Performing Pull Experiment with chi2 \n";
68     pull(n_toys,n_tot_entries,n_bins,true);
69     cout << "Performing Pull Experiment with Log Likelihood\n";
70     pull(n_toys,n_tot_entries,n_bins,false);
71     }
```

file: `macro9.cxx`

Your present knowledge of ROOT should be enough to understand all the technicalities behind the macro. Note that the variable `pull` in line 54 is different from the definition above: instead of the parameter error on `mean`, the fitted standard deviation of the distribution divided by the square root of the number of entries, `sig/sqrt(n_tot_entries)`, is used.

- What method exhibits the better performance with the default parameters?
- What happens if you increase the number of entries per histogram by a factor of ten? Why?

7.3. Fitting in General

In the examples above, we used the simplified fitting interface of ROOT, and the default minimisation functions. In general, however, fitting tasks often require special, user-defined minimisation functions. This is the case when data cannot be represented as one- or two-dimensional histograms or graphs, when errors are correlated and covariance matrices must be taken into account, or when external constrains on some of the fit parameters exist. The default minimiser in ROOT is MINUIT, a package that has been in use since decades. It offers several minimisation methods and a large number of features accessible through the class `TMinuit`. A more modern, generalised interface allowing to use other minimises also exists (see class `TVirtualFitter`), but still lacks some of the original features offered by `TMinuit`. The macro below provides a very general example, consisting of a part to be written specifically for each problem, and a more general part at the bottom for executing the fit and retrieving the results. Data is read from a file, stored in an n-tuple for repeated access, and an extended negative log-likelihood function is calculated and minimized with respect to the fit parameters.

```
/* Example of an extended log likelihood fit
      control part of this macro is general   */

// ———— begin of user code ————
// global variables for this macro
TF1 *nPDF;          // probability density function for the fit
TNtuple *inpdata; //n-tuple to hold input data
// Info for initialisation of MINUIT
int NFitPar=3; // specify number of fit parameters
//———————————————————————————————————————————————————————————————
int initialize_fit(TMinuit* minfit){ // initialisation of FIT
 // Define a probability density function, normalized to N !
 //        exponential in range [0,5.] plus off-set
    nPDF=new TF1("eplusconstPDF","[2]*((1.-[1])*(exp(-x/[0])-exp(-5./[0]))/[0]+[1]/(5.))",0.,5.);
 // input data come from a file and are stored in an NTuple
    inpdata=new TNtuple("InputData","InputData","x");
    cout << "\nNtuple contains " << inpdata->ReadFile("expob.dat")
         << " entries.\n\n";
    minfit->DefineParameter(0,    // Param index
                            "tau", // Param name
                            1,     // Param initial value
                            0.1,   // Param initial error, 0 for fix marameter
                            0,     // Param lower limit
                            0);    // Param upper limit
    minfit->DefineParameter(1,"offset",0.5, 0.1, 0, 1);
    minfit->DefineParameter(2,"norm",  150, 10,  0, 0);
    return 0;}
//———————————————————————————————————————————————————————————————
//The function to be minimized, called by MINUIT, must have this form.
```

```
void the_function(Int_t &npar,                    // Optional
                  Double_t* derivatives_array, // optional
                  Double_t& function_val,      // the function value
                  Double_t* par,               // the array of parameters
                  Int_t internal_flag){        // internal flag
 // calculate extended negative log likelihood
    function_val=0.;
    // pass on parameters to PDF
    nPDF->SetParameters(par[0],par[1],par[2]);
    // calculate -log L, i.e. loop over ntuple
    float *ntrow;
    for (int i=0; i < inpdata->GetEntries(); ++i){
        inpdata->GetEntry(i); ntrow=inpdata->GetArgs();
        function_val -= log(nPDF->Eval(ntrow[0]));}
    // add a Poission-term to take into account normalisation
    function_val -=inpdata->GetEntries()*log(par[2])-par[2];
    function_val *=2.; // mult. by 2, as usual in ROOT, i.e. Dchi2=D(-logL)
}
//------------------------------------------------------------------------------
void end_of_fit(TMinuit* minfit){
   // compare data with fit at the end
   TCanvas *cfit = new TCanvas("cfit","results",10,10,400,400);
   cfit->cd();
   inpdata->Draw("x"); TH1F ht(*htemp); // access to histogram
   ht.SetLineWidth(2); ht.SetLineColor(kBlue);
   // PDF must be scaled to take into account bin width
   ht.Eval(nPDF); ht.Scale(ht.GetBinWidth(1));
   ht.SetName("Data");ht.SetTitle("Fit to data;x;N_{Events}");
   ht.DrawClone("C SAME");}
// -------- end of user code --------
// ************************************************************************
// -------- start of general code  --------
// Function to access info on fit (and print it)
void printFit(TMinuit *minfit) {
  using namespace TMath;
  using namespace std;
    char line[200];
    Double_t vline[25];Double_t eline[25];
    cout << "\n\n\n";
    // ---------------------------------------------------------------------
    cout << "Fitted parameters: " << endl;
    cout << "       NO.      NAME        VALUE     ERROR " << endl;
    for (int n = 0; n < minfit->fNu; n++) {
        sprintf(line, "     %4d %9s %12.5g %8.3g",n+1,
         (const char*)minfit->fCpnam[n],minfit->fU[n],minfit->fWerr[n]);
    cout << line << endl; }
    // ---------------------------------------------------------------------
    cout << "  Correlation Matrix: " << endl;
    cout << "  NO.    GLOBAL";
    for (Int_t id = 1; id <= minfit->fNu; ++id)
      cout<< "        " <<id; cout<< endl;
        for (int i = 1; i <= minfit->fNu; ++i) {
            int ix  = minfit->fNexofi[i-1];
            int ndi = i*(i + 1) / 2;
            for (Int_t j = 1; j <= minfit->fNu; ++j){
                int m = Max(i,j); int n = Min(i,j);
                int ndex=m*(m-1)/2+n; int ndj=j*(j+1)/2;
                vline[j-1] = minfit->fVhmat[ndex-1]/
        sqrt(fabs(minfit->fVhmat[ndi-1]*minfit->fVhmat[ndj-1]));}
            sprintf(line, "  %2d   %8.3g ",ix,minfit->fGlobcc[i-1]);
            cout << line;
            for (Int_t it = 1; it <= minfit->fNu; ++it) {
                sprintf(line, "  %6.3f",vline[it-1]); cout << line; }
```

```
             cout << endl; }
    // ---------------------------------------------------------------------
    cout << "  Covariance Matrix: " << endl;
    double dxdi, dxdj;
    for (int i = 1; i <= minfit->fNu; ++i) {
        int ix  = minfit->fNexofi[i-1];
        int ndi = i*(i + 1) / 2;
        minfit->mndxdi(minfit->fX[i-1], i-1, dxdi);
        for (Int_t j = 1; j <= minfit->fNu; ++j) {
          minfit->mndxdi(minfit->fX[j-1], j-1, dxdj);
          int m=TMath::Max(i,j);int n=TMath::Min(i,j);
          int ndex=m*(m-1)/2+n;int ndj=j*(j+1)/2;
          eline[j-1] = dxdi*minfit->fVhmat[ndex-1]*dxdj*minfit->fUp; }
        for (Int_t it = 1; it <= minfit->fNu; ++it) {
         sprintf(line, " %10.3e",eline[it-1]); cout << line; }
         cout << endl; }
}
//------------------------------------------------------------------------------
void formatGraph(TGraph*g,int col,int msize,int lwidth){
     g->SetLineColor(col);    g->SetMarkerColor(col);
     g->SetMarkerSize(msize); g->SetLineWidth(lwidth);}

void plotContours(TMinuit* minfit, int p1, int p2) {
 // Get confidence contours of parameters
    int ic=0;
    minfit->SetPrintLevel(0); // not print all countour points
    minfit->mncomd("Set ERR 4",ic); // Set the the contour level
    TGraph* cont_2sigma = (TGraph*) minfit->Contour(50);// contour w. 50 points
    minfit->mncomd("Set ERR 1",ic); // Set the the contour level
    TGraph* cont_1sigma = (TGraph*) minfit->Contour(50);// contour w. 50 points

    // The minimum of the graph and its 1 sigma error
    TGraphErrors min_g(1); min_g.SetMarkerStyle(22);
    min_g.SetPoint(0,minfit->fU[0],minfit->fU[1]);
    min_g.SetPointError(0,minfit->fWerr[0],minfit->fWerr[1]);

    // Maquillage of the Graphs
    formatGraph(cont_1sigma,kRed,p2,p1);
    formatGraph(cont_2sigma,kGreen,p2,p1);
    cont_2sigma->SetTitle("Contours;#tau;off-set");

    TCanvas *cresult = new TCanvas("cresult","",10,410,400,400);
    cresult->cd();
    cont_2sigma->DrawClone("APC");cont_1sigma->DrawClone("SamePC");
    min_g.DrawClone("PSame");}
//------------------------------------------------------------------------------
// main program for MINUIT fit
int example_minuit(){
    TMinuit* myminuit=new TMinuit(NFitPar); //initialize global pointer
    if(initialize_fit(myminuit)!=0) return -1;
   // Standard control of a fit with MINUIT
    int ic=0;  // integer for condition code
    myminuit->SetFCN(the_function);
    myminuit->mncomd("MIN", // Start minimization (SIMPLEX first, then MIGRAD)
                     ic);    // 0 if command executed normally
    myminuit->mncomd("MINOS",ic); // Call MINOS for asymmetric errors
    myminuit->mncomd("HESSE",ic); // Call HESSE for correct error matrix
    end_of_fit(myminuit);   // Call user-defined fit summary
    printFit(myminuit);     // retrieve output from minuit
    plotContours(myminuit,2,3); //contour lines of fit parameters (2 and 3)
    return 0; }
```

file: `example_minuit.cxx`

You already know most of the code fragments used above. The new part is the user-defined minimisation function `the_function`, made known to the minimiser via the method `SetFCN(void *f)`.

- Lines 6–27: initialisation of the fit: definition of a probability density function as a `TF1`, creation and filling of an n-tuple containing the data read from a file, and the definition of the fit parameters and their initial values and ranges. Note that the main program at the bottom must be called first, as it sets up the minimizer.
- Lines 31–47: definition of function to be minimised; the parameter list (number of parameters, eventually analytically calculated derivatives w.r.t. the parameters, the return value of the function, the array of parameters, and a control flag) is fixed, as it is expected by the minimisation package. This function is repeatedly called by the minimisation package with different values of the function parameters.
- Lines 49–58: procedure called upon completion of the fit; this part needs access to the data and serves for a comparison of the fit result with the data - here, we show the fitted function on top of a histogram of the input data. Note that the PDF of a likelihood fit needs to be scaled to take into account the bin width of the histogram.
- Line 61: the general part of the code starts here; it contains some helper functions to extract and display information from the class `TMinuit`, and the overall control of the different steps of the fit.
- Lines 140–end: Main program, control of the different steps of the fitting process.
- Line 63–109: The function `printFit` illustrates how to access the best-fit values of the parameters and their errors and correlations from an object of `TMinuit`. Here, they are written to standard output; it is easy to redirect this into a file or some other data structure, if required.
- Line 115–137: Retrieval of the contour lines of two fit parameters, one and two σ in this example. The correlation of the two variables `tau` and `off-set` is clearly visible (Figure 7.2).

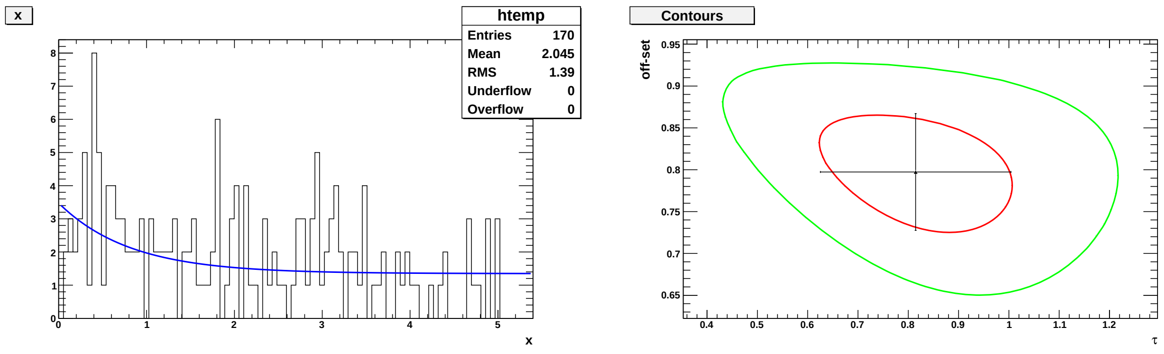


Figure 7.2.: Histogrammed input data with overlayed scaled fit function, and one- and 2-σ contour lines from extended log-likelihood fit.

CHAPTER 8 ROOT IN PYTHON

ROOT also offers an interface named PyRoot, see http://root.cern.ch/drupal/content/pyroot, to the PYTHON programming language. PYTHON is used in a wide variety of application areas and one of the most used scripting languages today. With its very high-level data types with dynamic typing, its intuitive object orientation and the clear and efficient syntax PYTHON is very suited to control even complicated analysis work flows. With the help of PyROOT it becomes possible to combine the power of a scripting language with ROOT methods.

Introductory material to PYTHON is available from many sources in the Internet, see e. g. http://docs.python.org/. There are additional very powerful PYTHON packages, like *numpy*, providing high-level mathematical functions and handling of large multi-dimensional matrices, or *matplotlib*, providing plotting tools for publication-quality graphics. PyROOT additionally adds to this access to the vast capabilities of the ROOT universe.

To use ROOT from PYTHON, the environment variable `PYTHONPATH` must include the path to the library path, `$ROOTSYS/lib`, of a ROOT version with PYTHON support. Then, *PyROOT* provides direct interactions with ROOT classes from PYTHON by importing ROOT.py into PYTHON scrips via the command `import ROOT`; it is also possible to import only selected classes from ROOT, e. g. `from ROOT import TF1`.

8.1. PyROOT

The access to ROOT classes and their methods in PyROOT is almost identical to C++ macros, except for the special language features of PYTHON, most importantly dynamic type declaration at the time of assignment.

Coming back to our first example, simply plotting a function in ROOT, the following C++ code:

```
TF1 *f1 = new TF1("f2","[0]*sin([1]*x)/x",0.,10.);
f1->SetParameter(0,1);
f1->SetParameter(1,1);
f1->Draw();
```

in PYTHON becomes:

```
import ROOT
f1 = ROOT.TF1("f2","[0]*sin([1]*x)/x",0.,10.)
f1.SetParameter(0,1)
f1.SetParameter(1,1)
f1.Draw();
```

A slightly more advanced example hands over data defined in the macro to the ROOT class `TGraphErrors`. Note that a PYTHON array can be used to pass data between PYTHON and ROOT. The first line in the PYTHON script allows it to be executed directly from the operating system, without the need to start the script from `python` or the highly recommended powerful interactive shell `ipython`. The last line in the python script is there to allow you to have a look at the graphical output in the ROOT canvas before it disappears upon termination of the script.

Here is the C++ version:

```
void TGraphFit() {
//
//Draw a graph with error bars and fit a function to it
//
gStyle->SetOptFit(111); //superimpose fit results
// make nice Canvas
TCanvas *c1 = new TCanvas("c1","Daten",200,10,700,500);
c1->SetGrid();
//define some data points ...
const Int_t n = 10;
Float_t x[n]  = {-0.22, 0.1, 0.25, 0.35, 0.5, 0.61, 0.7, 0.85, 0.89, 1.1};
Float_t y[n]  = {0.7, 2.9, 5.6, 7.4, 9., 9.6, 8.7, 6.3, 4.5, 1.1};
Float_t ey[n] = {.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
Float_t ex[n] = {.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
// and hand over to TGraphErros object
TGraphErrors *gr = new TGraphErrors(n,x,y,ex,ey);
gr->SetTitle("TGraphErrors with Fit");
gr->Draw("AP");
// now perform a fit (with errors in x and y!)
gr->Fit("gaus");
c1->Update();
}
```
file: `TGraphFit.C`

In PYTHON it looks like this:

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

Comparing the C++ and PYTHON versions in these two examples, it now should be clear how easy it is to convert any ROOT Macro in C++ to a PYTHON version.

As another example, let us revisit *macro3* from Chapter 4. A straight-forward PYTHON version relying on the ROOT class `TMath`:

```
#!/usr/bin/env python
#      (the first line allows execution directly from the linux shell)
#
#--------- macro3 as python script ------------------------------
```

```
# Author:        G. Quast   Oct. 2013
# dependencies:  PYTHON v2.7, pyroot
# last modified:
#------------------------------------------------------------------
#
# *** Builds a polar graph in a square Canvas

from ROOT import TCanvas,TGraphPolar,TMath
from array import array

rmin=0.
rmax=6.*TMath.Pi()
npoints=300
r=array('d',npoints*[0.])
theta=array('d',npoints*[0.])
e=array('d',npoints*[0.])
for ipt in range(0,npoints):
    r[ipt] = ipt*(rmax-rmin)/(npoints-1.)+rmin
    theta[ipt]=TMath.Sin(r[ipt])
c=TCanvas("myCanvas","myCanvas",600,600)
grP1=TGraphPolar(npoints,r,theta,e,e)
grP1.SetTitle("A Fan")
grP1.SetLineWidth(3)
grP1.SetLineColor(2)
grP1.Draw("AOL")

raw_input('Press <ret> to end -> ')
```

file: `macro3.py`

8.1.1. More PYTHON- less ROOT

You may have noticed already that there are some PYTHON modules providing functionality similar to ROOT classes, which fit more seamlessly into your PYTHON code.

A more “pythonic” version of the above macro3 would use a replacement of the ROOT class `TMath` for the provisoining of data to `TGraphPolar`. With the *math* package, the part of the code becomes

```
import math
from array import array
from ROOT import TCanvas,TGraphPolar
  ...
ipt=range(0,npoints)
r=array('d',map(lambda x: x*(rmax-rmin)/(npoints-1.)+rmin,ipt))
theta=array('d',map(math.sin,r))
e=array('d',npoints*[0.])
  ...
```

Using the very powerful package *numpy* and the built-in functions to handle numerical arrays makes the PYTHON code more compact and readable:

```
import numpy as np
from ROOT import TCanvas,TGraphPolar
  ...
r=np.linspace(rmin,rmax,npoints)
theta=np.sin(r)
e=np.zeros(npoints)
  ...
```

file: `macro3_numpy.py`

Customised Binning

This example combines comfortable handling of arrays in PYTHON to define variable bin sizes of a ROOT histogram. All we need to know is the interface of the relevant ROOT class and its methods (from the ROOT documentation):

```
TH1F(const char* name, const char* title, Int_t nbinsx, const Double_t* xbins)
```

Here is the PYTHON code:

```
import ROOT
from array import array
arrBins = array('d',(1,4,9,16) ) # array of bin edges
histo = ROOT.TH1F("hist", "hist", len(arrBins)-1, arrBins)
# fill it with equally spaced numbers
for i in range(1,16):
  histo.Fill(i)
histo.Draw()
```
file: `histrogram.py`

A fit example in PYTHON using TMinuit from ROOT

One may even wish to go one step further and do most of the implementation directly in PYTHON, while using only some ROOT classes. In the example below, the ROOT class `TMinuit` is used as the minimizer in a χ^2-fit. Data are provided as PYTHON arrays, the function to be fitted and the χ^2-function are defined in PYTHON and iteratively called by `Minuit`. The results are extracted to PYTHON objects, and plotting is done via the very powerful and versatile python package `matplotlib`.

```
#!/usr/bin/env python
#
#--------- python script ---------------------------------------------
# EXAMPLE showing how to set up a fit with MINUIT using pyroot
#---------------------------------------------------------------------
# Author:        G. Quast   May 2013
# dependencies:  PYTHON v2.7, pyroot, numpy, matplotlib, array
# last modified: Oct. 6, 2013
#---------------------------------------------------------------------
#
from ROOT import TMinuit,Double,Long
import numpy as np
from array import array as arr
import matplotlib.pyplot as plt

# --> define some data
ax = arr( 'f', ( ↩
    0.05,0.36,0.68,0.80,1.09,1.46,1.71,1.83,2.44,2.09,3.72,4.36,4.60) )
ay = arr( 'f', ( ↩
    0.35,0.26,0.52,0.44,0.48,0.55,0.66,0.48,0.75,0.70,0.75,0.80,0.90) )
ey = arr( 'f', ( ↩
    0.06,0.07,0.05,0.05,0.07,0.07,0.09,0.10,0.11,0.10,0.11,0.12,0.10) )
nPoints = len(ax)

# --> Set parameters and function to fit
#  a list with convenient names,
name = ["a","m","b"]
# the initial values,
vstart = arr( 'd', (1.0, 1.0, 1.0) )
# and the initial step size
step =   arr( 'd', (0.001, 0.001, 0.001) )
npar =len(name)
#
# this defines the function we want to fit:
def fitfunc(x, npar, apar):
    a = apar[0]
    m = apar[1]
    b = apar[2]
    f = Double(0)
    f=a*x*x + m*x + b
    return f
#
```

```

# --> this is the definition of the function to minimize, here a chi^2-function
def calcChi2(npar, apar):
    chisq = 0.0
    for i in range(0,nPoints):
        x = ax[i]
        curFuncV = fitfunc(x, npar, apar)
        curYV = ay[i]
        curYE = ey[i]
        chisq += ( (curYV - curFuncV) * (curYV - curFuncV) ) / (curYE*curYE)
    return chisq

#--- the function fcn - called by MINUIT repeatedly with varying parameters
#       NOTE: the function name is set via TMinuit.SetFCN
def fcn(npar, deriv, f, apar, iflag):
    """ meaning of parametrs:
        npar:   number of parameters
        deriv:  aray of derivatives df/dp_i (x), optional
        f:      value of function to be minimised (typically chi2 or negLogL)
        apar:   the array of parameters
        iflag:  internal flag: 1 at first call, 3 at the last, 4 during ↩
            minimisation
    """
    f[0] = calcChi2(npar,apar)
#

# --> set up MINUIT
myMinuit = TMinuit(npar)  # initialize TMinuit with maximum of npar parameters
myMinuit.SetFCN(fcn)      # set function to minimize
arglist = arr('d', 2*[0.01]) # set error definition
ierflg = Long(0)
arglist[0] = 1.               # 1 sigma is Delta chi^2 = 1
myMinuit.mnexcm("SET ERR", arglist ,1,ierflg)

# --> Set starting values and step sizes for parameters
for i in range(0,npar):                  # Define the parameters for the fit
  myMinuit.mnparm(i, name[i], vstart[i], step[i], 0,0,ierflg)
arglist[0] = 6000 # Number of calls to FCN before giving up.
arglist[1] = 0.3  # Tolerance
myMinuit.mnexcm("MIGRAD", arglist ,2,ierflg)  # execute the minimisation

# --> check TMinuit status
amin, edm, errdef = Double(0.), Double(0.), Double(0.)
nvpar, nparx, icstat = Long(0), Long(0), Long(0)
myMinuit.mnstat(amin,edm,errdef,nvpar,nparx,icstat)
# meaning of parameters:
#   amin: value of fcn at minimum (=chi^2)
#   edm:  estimated distance to mimimum
#   errdef: delta_fcn used to define 1 sigma errors
#   nvpar: number of variable parameters
#   nparx: total number of parameters
#   icstat: status of error matrix:
#           3=accurate
#           2=forced pos. def
#           1= approximative
#           0=not calculated
myMinuit.mnprin(3,amin) # print-out by Minuit

# --> get results from MINUIT
finalPar = []
finalParErr = []
p, pe = Double(0), Double(0)
for i in range(0,npar):
    myMinuit.GetParameter(i, p, pe)  # retrieve parameters and errors
    finalPar.append(float(p))
    finalParErr.append(float(pe))
# get covariance matrix
buf = arr('d', npar*npar*[0.])
myMinuit.mnemat(buf,npar) # retrieve error matrix
emat=np.array(buf).reshape(npar,npar)

# --> provide formatted output of results
print "\n"
print "*==* MINUIT fit completed:"
print ' fcn@minimum = %.3g' %(amin)," error code =",ierflg," status =",icstat
print "  Results: \t    value       error          corr. mat."
for i in range(0,npar):
    print '     %s: \t%10.3e +/- %.1e    '%(name[i],finalPar[i],finalParErr[i]),
    for j in range (0,i):
      print '%+.3g ' %(emat[i][j]/np.sqrt(emat[i][i])/np.sqrt(emat[j][j])),
    print ' '

# --> plot result using matplotlib
plt.figure()
plt.errorbar(ax, ay, yerr=ey, fmt="o", label='data') # the data
x=np.arange(ax[0],ax[nPoints-1],abs((ax[nPoints-1]-ax[0])/100.) )
y=fitfunc(x,npar,finalPar) # function at best-fit-point
plt.title("Fit Result")
plt.grid()
plt.plot(x,y, label='fit function')
plt.legend(loc=0)
plt.show()
```

file: fitting-example.py

CONCLUDING REMARKS

This is the end of our guided tour through ROOT for beginners. There is still a lot coming to mind to be said, but by now you are experienced enough to use the ROOT documentation, most importantly the **ROOT home page** and the **ROOT reference guide** with the documentation of all ROOT classes, or the **ROOT users guide**.

A very useful way for you to continue exploring ROOT is to study the examples in the sub-directory `tutorials/` of any ROOT installation.

There are some powerful additions to ROOT, e. g. packages named RooFit and RooStats providing a frame work for model building, fitting and statistical analysis. The ROOT class `TMVA` offers multi-variate analysis tools including an artificial neural network and many other advanced methods for classification problems. The remarkable ability of ROOT to handle large data volumes was already mentioned in this guide, implemented through the class `TTree`. But there is still much more for you to explore ...

End of this guide ... but hopefully not of your interaction with ROOT !

APPENDIX A

ROOFILAB

A.1. Root-based tool for fitting: RooFiLab

Although simple in principle, the fomulation of a problem in `C++` and the complex environment of the ROOT framework pose a relativly high hurdle to overcome for the beginner. A simplification and extension of avialable standard methods for function fitting to one-dimesional distributions is the package RooFiLab ("Root Fits for Laboratory courses"). Based on ROOT, this program developed at KIT (Karlsruhe Institute of Technology, URL http://www-ekp.physik.uni-Karlsruhe.de/ quast/RooFiLab) offers an easy-to-use, structured graphical user interface and an ASCII input format for typical use cases in student laboratory courses. Correlated erros on both the x- and y-coordinate are also supported. In the most general case, covariance matrices of the x- and y-coordinates can be specified. There is also a simplified possibility for special cases of fully correlated absolute or relative errors on the measurements. An example fit is shown in Figure A.1.

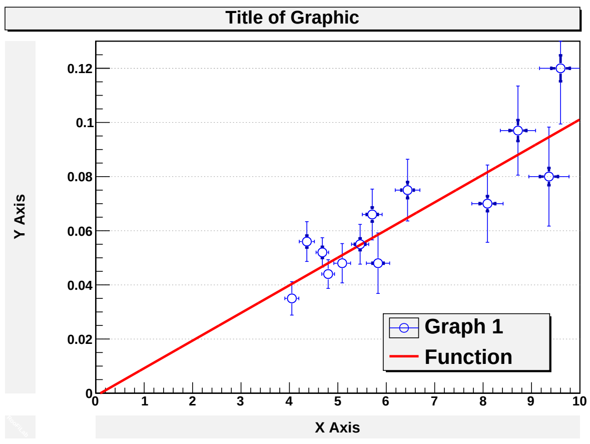


Figure A.1.: Example of a straight-line fit with independent and correlated (systematic) errors on both the x- and y-directions.

High flexibility in the definition of the model is achieved by direct usage of the ROOT interpreter, which has been extended to use named parameters instead of parameter numbers. In addition, more complex models can be implemented as C or C++ functions, wich are compiled and linked at run-time.

The elements of the grafical user interface (see Figure A.2) and control via the input file are described in the manual (file `RooFiLab.pdf` in the subdirectory `RooFiLab/doc`, in German language). A brief overview is given here.

A.1.1. Installation

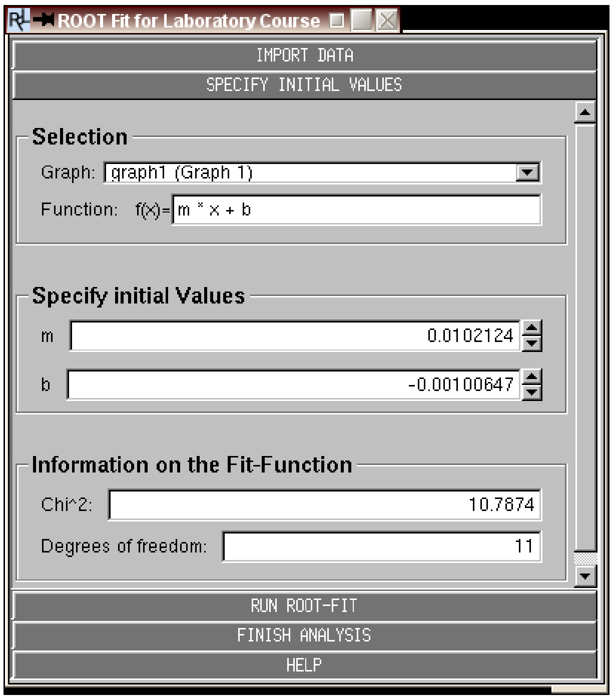
RooFiLab is availalbe, fully installed along with ROOT in a virtual machine[1] based on the Ubuntu distribution. The compressed disk image is most easily imported into the freely available virtualisation tool *VirtualBox* for the most common Linux distributions, for Windows versions XP and later and for Macintosh operating systems.

The program code of *RooFiLab* is distributed from the URL given above as a compressed archive `RooFiLab.tar.gz`. After unpacking, the installation under Linux proceeds by executing `make`; the file `Makefile` contains all neccessary instructions. A ROOT installation must be present and initialized, i.e. the environment variable `PATH` must contain the path to the ROOT executable and `LD_LIBRARY_PATH` must point to the ROOT libraries.

A.1.2. Usage of *RooFiLab*

RooFiLab offers two windows: one is used for control, the other is for graphics output. The control window, as depicted in FigureA.2, is separated into four Shutters, offering the following actions

- data input and definition of functions and parameters
- fixing of start values and “Fit-by-Eye”
- execution of the fit, eventually iteratively by fixing some of the free parameters
- options for graphical output



Selection

Graph: graph1 (Graph 1)

Function: f(x)= m * x + b

Specify initial Values

m 0.0102124

b -0.00100647

Information on the Fit-Function

Chi^2: 10.7874

Degrees of freedom: 11

Figure A.2.: The grafical user interface of RooFiLab.

During execution, ROOT functionality is also available. Of particular importance are procedures for interactive manilulations of the output graphcis and their export. As usual, the context menu is opened by right-klicking of the components of the graph or via the Toolbar at the top of the graphics window.

In addition to interactive usage of the controls of the graphical interface, fits can also be executed automatically by specification of control options in the input file definig the data inputs. After an interactive fit, options can thus be archived in the input file and then be used for repeated, automated fits.

A.2. Examples with *RooFiLab*

The following subsections show simple examples illustrating the usage of *RooFiLab* and may serve as the basis for own applications.

[1] `http://www-ekp.physik.uni-karlsruhe.de/~quast/VMroot`

A.2.1. Straight-line fit with correlated erros in x and y

This *RooFiLab* input file contains several control lines and documents the available options. Control lines are comment lines starting with `#!` followed by a keyword. The control command `#! dofit = true` triggers an automated fit defined by the input data and the control options in the file.

```
# straight-line fit to data with errors in x and y, incl. simple correlations
# ===========================================================================
#! staterrors = xy
#! systerrors = 0.02 0.04 rel rel

#! fit = "m*x+b" "m,b" "roofilab.fit"
#! initialvalues = 0.015 0

### command to execute fit
#! dofit = true
### show systematic erros as second error bar
#! secondgraph = syst

#! title = "Fit to data with correlated errors"
#! graphlegend = "Data" bottom right
#! functionlegend = "Model" bottom right
#! xaxis = "X-values"
#! yaxis = "Y-values or f(x)"

#! markersettings = 1.5 4 24
#! functionsettings = 1 3 2
#! grid = y
#! logscale = 0
#! savegraphic = "roofilab.eps"

# =================Eingabe der Daten ===================================
# values in up to four columns separated by whitespaces
#       (except for linebreaks or linefeeds)
# x       y      ex      ey
4.05 0.035 0.12 0.006
4.36 0.056 0.13 0.007
4.68 0.052 0.09 0.005
4.80 0.044 0.09 0.005
5.09 0.048 0.14 0.007
5.46 0.055 0.14 0.007
5.71 0.066 0.17 0.009
5.83 0.048 0.21 0.011
6.44 0.075 0.22 0.011
8.09 0.070 0.28 0.014
8.72 0.097 0.32 0.016
9.36 0.080 0.37 0.018
9.60 0.120 0.39 0.020
```

A.2.2. Averaging correlated measurements

Averaging correlated measurements formally corresponds to a fit of a constant. The measurements in this example are the individual measurements of the mass of the Z Boson at the electron-positron collider LEP at CERN. The common error of 1.7 MeV results from uncertainties in the centre-of-mass energy of the accelerator. The line `#! systerrors = 0 0.0017 abs abs` specifies this common aboslute error on each measurement.

```
# Mesurements of Z-Mass by AELPH, DELPHI, L3 and OPAL
# ---------------------------------------------------

# graphics options
#! markersettings = 1.5 4 24
#! functionsettings = 1 3 3
#! grid = y
# logscale = 0
# savegraphic = "roofilab.eps"
# saverfl = "data.rfl"

# plot lables
#! title = "averaging measurements"
#! xaxis = "n"
#! yaxis = "Mass of Z boson"
#! graphlegend = "Z mass measurements" bottom right
#! functionlegend = "average Z mass" bottom right

# fit control
#! fit = "m" "m" "average.fit"
#! initialvalues =  91.2
#! dofit = true

#! staterrors = y # control-command
#! systerrors = 0 0.0017 abs abs
# the data, LEP electroweak working group, CERN 2000
1 91.1893 0.0031
2 91.1863 0.0028
3 91.1894 0.0030
4 91.1853 0.0029
```

A.2.3. Fit of a polynomyal to data with Poisson errors

This example show the fit of a fourth-order polynomial to data with uncorrelated, Poissonian errors, i. e. erros given by the square root of the data points. Although the errors are non-Gaussion in this case, a χ^2-fit often results in acceptable results. With the option `#! fitmethod = likelihood` a likelihood method can be selected. In this case, the statistical errors are ignored and may be ommitted. For technical reasons, the x-values must be equi-distant in this case (due to usage of ROOT-class `TH1`).

```
##########################################################
#     example: fit of an angular distribution
##########################################################

# plot commands
#! title = "angular distribution "
#! xaxis = "cos(theta)"
#! yaxis = "number of events"
#! graphlegend ="observed rate " top left
#! functionlegend ="fitted cos(theta) distribution " top left
#! markersettings = 1.5 2 5
#! functionsettings = 1 3 3
# fit control
#! fit = "a4*x^4+a3*x^3+a2*x^2+a1*x+a0" "a0,a1,a2,a3,a4" "v_vs_cost.fit"
#! dofit = true

# fitmethod = likelihood # uncomment to perform a Log Likelihood fit

# definition of data
#! staterrors = y
# cost   N    sqrt(N)
-0.9    81.    9.0
-0.7    50.    7.1
-0.5    35.    5.9
-0.3    27.    5.2
-0.1    26.    5.1
 0.1    60.    7.7
 0.3   106.    10.3
 0.5   189.    13.7
 0.7   318.    17.8
 0.9   520.    22.8
```

A.2.4. Correlated measurements with full covariance matrix

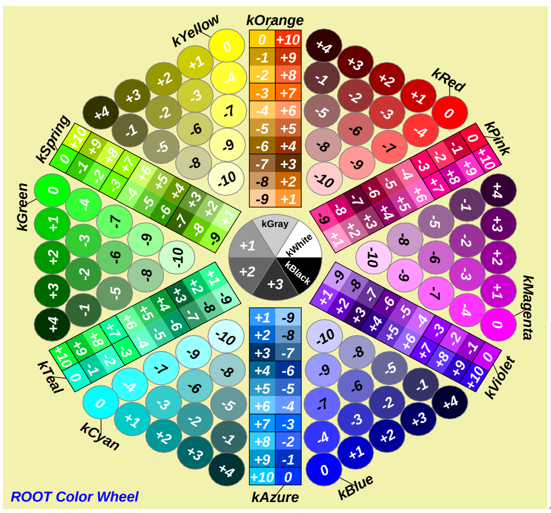
As a more complex example the averaging procedure for measurements of the W Boson mass is shown here. Measurements of the four LEP experiments in two final states have different systematic errors, which are correlated among groups of measurements. These are specified in the full 8×8 covariance matrix, which is composed of 4×4 block matrices. The control line `#! covmatrices = 0 wmass.cov` . specifies that not covariance matrix in x and the matrix `wmass.cov` are to be used in the fit.

```
# Mesurements of W-Mass by AELPH, DELPHI, L3 and OPAL
# ---------------------------------------------------
# ### example of fit with covariance matrix#
# --- graphics options
#! markersettings = 1.5 4 24
#! functionsettings = 1 3 3
#! grid = y
#! title = "averaging measurements"
#! xaxis = "n"
#! yaxis = "Mass of W boson"
#! graphlegend = "W mass measurements" top right
#! functionlegend = "average W mass" top right
# --- fit control
#! fit = "m" "m" "Wmittelung.fit"
#! initialvalues =  80.5
#! dofit = true
# --- the data (LEP electroweak working group, CERN 2006)
#! staterrors = 0
#! systerrors = 0 0 abs abs
#! covmatrices = 0 wmass.cov
1 80.429  0.059 # qqlv ALEPH
2 80.340  0.076 # qqlv DELPHI
3 80.213  0.071 # qqlv L3
4 80.449  0.062 # qqlv OPAL
5 80.475  0.082 # qqqq ALEPH
6 80.310  0.102 # qqqq DELPHI
7 80.323  0.091 # qqqq L3
8 80.353  0.081 # qqqq OPAL

//file wmass.cov
  0.003481 0.000316 0.000316 0.000316 0.000383 0.000383 0.000383 0.000383
  0.000316 0.005776 0.000316 0.000316 0.000383 0.000383 0.000383 0.000383
  0.000316 0.000316 0.005041 0.000316 0.000383 0.000383 0.000383 0.000383
  0.000316 0.000316 0.000316 0.003844 0.000383 0.000383 0.000383 0.000383
  0.000383 0.000383 0.000383 0.000383 0.006724 0.001741 0.001741 0.001741
  0.000383 0.000383 0.000383 0.000383 0.001741 0.010404 0.001741 0.001741
  0.000383 0.000383 0.000383 0.000383 0.001741 0.001741 0.008281 0.001741
  0.000383 0.000383 0.000383 0.000383 0.001741 0.001741 0.001741 0.006561
```

APPENDIX B MARKERS, COLOURS, SYMBOLS

B.1. Colour Wheel and Graph Markers



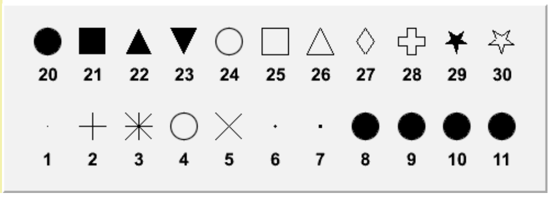


Figure B.1.: The wheel shows all available colours in ROOT and the codes to specify them and The markers provided by ROOT.

Table B.1.: Alternative symbols to select the ROOT markers for graphs.

Integer	Description	Literal	Integer	Description	Literal
1	dot	kDot	21	full square	kFullSquare
2	+	kPlus	22	full triangle up	kFullTriangleUp
3	*	kStar	23	full triangle down	kFullTriangleDown
4	o	kCircle	24	open circle	kOpenCircle
5	x	kMultiply	25	open square	kOpenSquare
6	small dot	kFullDotSmall	26	open triangle up	kOpenTriangleUp
7	medium dot	kFullDotMedium	27	open diamond	kOpenDiamond
8	large scalable dot	kFullDotLarge	28	open cross	kOpenCross
20	full circle	kFullCircle	29	open star	kOpenStar

B.2. Lines and Arrows

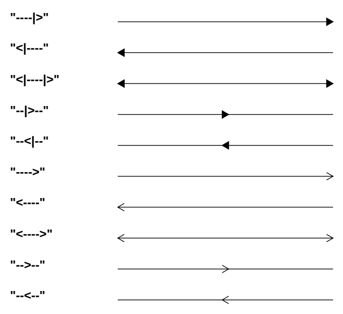
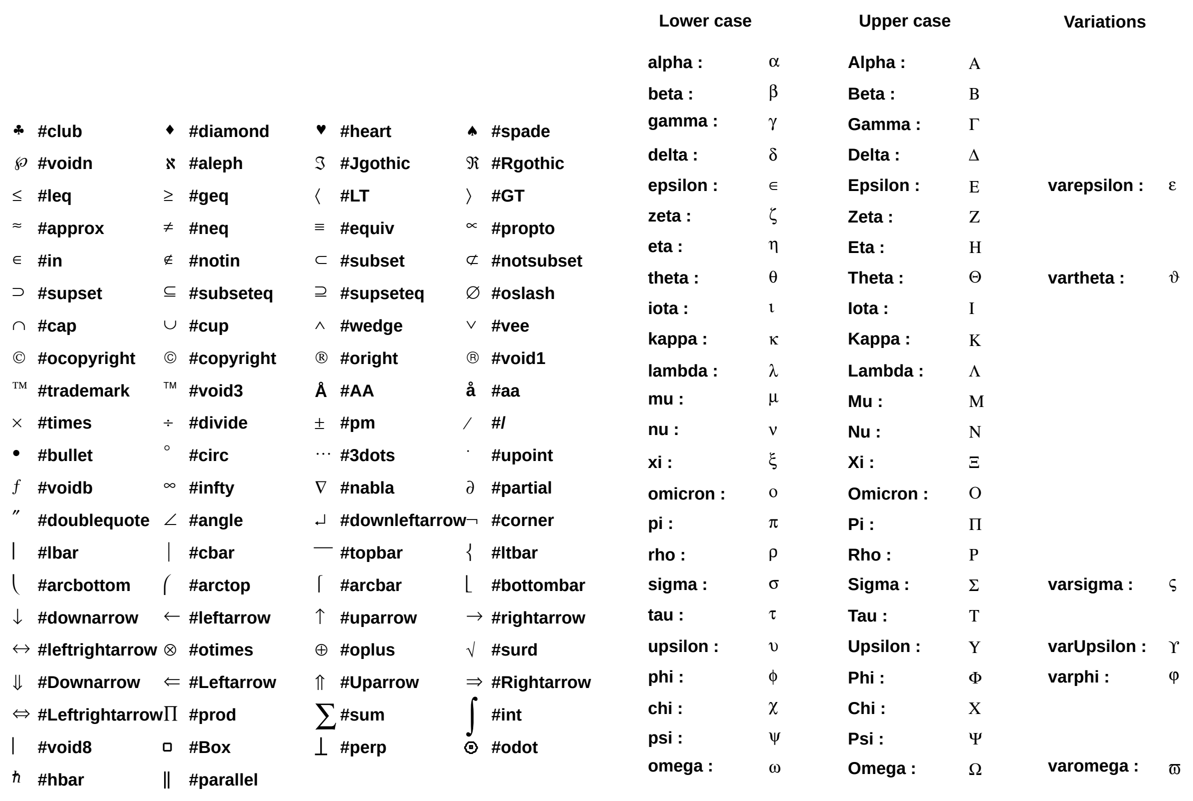


Figure B.2.: The arrows styles available in ROOT.

B.3. Latex Symbols



♣ #club	♦ #diamond	♥ #heart	♠ #spade
℘ #voidn	ℵ #aleph	ℑ #Jgothic	ℜ #Rgothic
≤ #leq	≥ #geq	〈 #LT	〉 #GT
≈ #approx	≠ #neq	≡ #equiv	∝ #propto
∈ #in	∉ #notin	⊂ #subset	⊄ #notsubset
⊃ #supset	⊆ #subseteq	⊇ #supseteq	∅ #oslash
∩ #cap	∪ #cup	∧ #wedge	∨ #vee
© #ocopyright	© #copyright	® #oright	® #void1
™ #trademark	™ #void3	Å #AA	å #aa
× #times	÷ #divide	± #pm	⁄ #/
• #bullet	° #circ	⋯ #3dots	· #upoint
ƒ #voidb	∞ #infty	∇ #nabla	∂ #partial
″ #doublequote	∠ #angle	↵ #downleftarrow	¬ #corner
| #lbar	| #cbar	‾ #topbar	{ #ltbar
⎝ #arcbottom	⎛ #arctop	⌈ #arcbar	⌊ #bottombar
↓ #downarrow	← #leftarrow	↑ #uparrow	→ #rightarrow
↔ #leftrightarrow	⊗ #otimes	⊕ #oplus	√ #surd
⇓ #Downarrow	⇐ #Leftarrow	⇑ #Uparrow	⇒ #Rightarrow
⇔ #Leftrightarrow	∏ #prod	∑ #sum	∫ #int
| #void8	□ #Box	⊥ #perp	⊙ #odot
ℏ #hbar	∥ #parallel		

Lower case		Upper case		Variations	
alpha :	α	Alpha :	Α		
beta :	β	Beta :	Β		
gamma :	γ	Gamma :	Γ		
delta :	δ	Delta :	Δ		
epsilon :	ϵ	Epsilon :	Ε	varepsilon :	ε
zeta :	ζ	Zeta :	Ζ		
eta :	η	Eta :	Η		
theta :	θ	Theta :	Θ	vartheta :	ϑ
iota :	ι	Iota :	Ι		
kappa :	κ	Kappa :	Κ		
lambda :	λ	Lambda :	Λ		
mu :	μ	Mu :	Μ		
nu :	ν	Nu :	Ν		
xi :	ξ	Xi :	Ξ		
omicron :	ο	Omicron :	Ο		
pi :	π	Pi :	Π		
rho :	ρ	Rho :	Ρ		
sigma :	σ	Sigma :	Σ	varsigma :	ς
tau :	τ	Tau :	Τ		
upsilon :	υ	Upsilon :	Υ	varUpsilon :	ϒ
phi :	ϕ	Phi :	Φ	varphi :	φ
chi :	χ	Chi :	Χ		
psi :	ψ	Psi :	Ψ		
omega :	ω	Omega :	Ω	varomega :	ϖ

Figure B.3.: The main Latex symbols that can be interpreted by the TLatex class.

APPENDIX C MOST RELEVANT CLASSES AND THEIR METHODS

This list of classes and methods shows the most relevant ones, which have been considered in this guide. It is an excerpt from the ROOT class reference guide.

TGraphErrors: the graph class with error bars

create Graph frm file	`TGraphErrors(const char* filename, const char* format = "%lg %lg %lg %lg", Option_t* option = "")`
create graph fom C-arrays	`TGraphErrors(Int_t n, const Float_t* x, const Float_t* y, const Float_t* ex = 0, const Float_t* ey = 0)`
create graph from histogram	`TGraphErrors(const TH1* h)`
fit a function	`.Fit(TF1* f1, Option_t* option = "", Option_t* goption = "", Axis_t xmin = 0, Axis_t xmax = 0)`
	`.Fit(const char* formula, Option_t* option = "", Option_t* goption = "", Axis_t xmin = 0, Axis_t xmax = 0)`
draw	`.Draw("AP")` and `.DrawClone("AP")`
draw options	methods of classes `TGraph`, `TGraphPainter`

TH1F: the histogram class with float bin contents

create ("book") histogram	`TH1F(const char* name, const char* title, Int_t nbinsx, Double_t xlow, Double_t xup)`
store also squared weights	`.Sumw2()`
fill a value	`.Fill(Double_t x)`
fill with weight	`.Fill(Double_t x, Double_t w)`
set bin content	`.SetBinContent(Int_t bin, Double_t content)`
get bin content	`Double_t .GetBinContent(Int_t bin) const`
fill with random numbers	`.FillRandom(const char* fname, Int_t ntimes)`
clear	`.Reset()`
copy to C-array	`Float_t* .GetArray()`
set maximum on y-axis	`.SetMaximum(Double_t ymax)`
set minimum on y-axix	`.SetMinimum(Double_t ymin)`
get mean	`Double_t GetMean(1)`
get RMS	`Double_t GetRMS(1)`
draw	`.Draw(Option_t* option = "")`
useful draw options	"SAME" "E" "P"
	see documentation of class `THistPainter`

TH2F: 2-dimensional histogram class with float bin contents

book	`TH2F(const char* name, const char* title, Int_t nbinsx, Double_t xlow, Double_t xup, Int_t nbinsy, Double_t ylow, Double_t yup)`
fill	`Fill(Double_t x, Double_t y)`
fill with weight	`Fill(Double_t x, Double_t y, Double_t w)`
get mean along axis i	`Double_t GetMean(i)`
get RMS along axis i	`Double_t GetRMS(i)`
get covariance	`Double_t GetCovariance()`
get correlation	`Double_t GetCorrelationFactor()`
draw	`.Draw(Option_t* option = "")` and `.DrawClone`
useful draw options	"" "SAME" "BOX" "COL" "LEGO" "SURF"
	see documentation of class `THistPainter`

TProfile: "profile representation" for 2-dim histograms

book profile histogram	`TProfile(const char* name,const char* title,Int_t nbinsx,Double_t xlow,Double_t xup,Double_t ylow,Double_t yup,Option_t* option = "")`
fill a value	`.Fill(Double_t x)`
fill with weight	`.Fill(Double_t x, Double_t w)`
draw	`.Draw()` and `.DrawClone()`

TF1: the mathematical function

define function in `TFormula` syntax	`TF1(const char* name, const char* formula, Double_t xmin = 0, Double_t xmax = 1)`
predefined functions	"gaus" "expo" "pol0" ... "pol9" "landau"
define function via pointer	`TF1(const char* name, void* fcn, Double_t xmin, Double_t xmax, Int_t npar)`
evaluate at x	`.Eval(Double_t x)`
calculate derivative	`Double_t .Derivative(Double_t x)`
calculate integral a to b	`Double_t .Integral(Double_t a, Double_t b)`
get random number	`Double_t .GetRandom()`
set parameter i	`.SetParameter(Int_t i, Double_t parvalue)`
set parameters	`.SetParameters(const Double_t* params)`
fit function `*f` to graph `*gr` or histogram `*h`	`gr->Fit(TF1 *f)` or h->Fit(TF1 *f)|
get parameter i	`Double_t .GetParameter(Int_t i)`
get error on parameter i	`Double_t .GetParError(Int_t i)`

TRandom3: the calss used to generate random sequences of high quality

initialize random generator with random seed	`TRandom(0)`
initialize random generator with seed	`TRandom(UInt_t seed)`
get actual seed	`UInt_t .GetSeed()`
uniform random number]0,x1]	`Double_t .Uniform(Double_t x1=1)`
uniform random number]x1,x2]	`Double_t .Uniform(Double_t x1, Double_t x2)`
random number from binomial distribution	`Int_t .Binomial(Int_t ntot, Double_t prob)`
random Poisson number	`Int_t .Poisson(Double_t mean)`
random number from exponential	`Double_t .Exp(Double_t tau)`
random number from Gaussian distribution	`Double_t .Gaus(Double_t mean=0, Double_t sigma=1)`
pre-initialised random generator	`gRandom` points to global instance of `TRandom3`

TCanvas: configuring the graphics canvas

create canvas of size `ww x wh`	`TCanvas(const char* name, const char* title, Int_t ww, Int_t wh)`
subdivide into pads	`.Divide(Int_t nx = 1, Int_t ny = 1, Float_t xmargin = 0.01, Float_t ymargin = 0.01, Int_t color = 0)`
chage to subpad	`.cd(Int_t subpadnumber = 0)`
update canvas	`.Update()`
mark as modified to trigger re-draw	`.Modified(Bool_t flag = 1)`
draw canvas	`.Draw(Option_t* option = "")` and `.DrawClone`

TLegend: the legend in a plot. Fundamental for the understanding of the contents

create Legend	`TLegend(Double_t x1,Double_t y1,Double_t x2,Double_t y2,const char* header, Option_t* option = brNDC)`
add an entry	`.AddEntry(TObject* obj, const char* label, Option_t* option = lpf)`
add text entry	`.AddEntry(const char* name, const char* label, Option_t* option = lpf)`
draw	`.Draw()` and `.DrawClone();`

TLatex: LaTEX formatting

create Text	`TLatex(Double_t x, Double_t y, const char* text)`
draw	`.Draw()` and `.DrawClone();`

TFile: file I/O

create file	`TFile(const char* fname, Option_t* option = "", const char* ftitle = "", Int_t compress = 1)`
	options " NEW" "CREATE" "RECREATE" "READ"
change direcotry to fle	`.cd()`
write histogram `*h` to file	`h1->Write()`
close file at the end	`.Close()`
read histogram `*h` from file `*f`	TH1F *h1=(TH1F*)f.Get(const char* histname)

TNtuple: variables in ntuples

create	`TNtuple(const char* name, const char* title, const char* varlist)`
	format varlist: `"x0:x2:...:xn"` (n<15)
fill	`.Fill(Float_t x0,Float_t x1=0,Float_t x2=0, ... ,Float_t x14=0)`
initialize from file	`.ReadFile(const char* filename)`
plot variables	`.Draw(const char* varexp, const char* selection)`
e.g. plot variable `xi`	`.Draw("xi")`
e.g. plot variable with cut on others	`.Draw("xi","xj<3")`
e.g. 2-dim plot of variables `xi` and `xj`	`.Draw("xi:xj")`
fill existing histogram from ntuple	`.Project(const char* hname, const char* varexp, const char* selection = "")`

global pointers `gStyle` and `gSystem` as instances of classes **TStyle** and **TSystem**

show statistics box	`gStyle->SetOptStat(11...1)`
show fit parameters in statistics box	`gStyle->SetOptFit(11...1)`
suppress title boxes on graphs and histograms	`gStyle->SetOptTitle(0)`
for animations: add pause in milliseconds	`gSystem->Sleep(UInt_t t)`

TVirtualFitter: Fitting

set default fitter, e. g. `name="Minuit"`	`TVirtualFitter::SetDefaultFitter("(const char* name = "")`
create Fitter instance	`TVirtualFitter::Fitter(0,Int_t maxpar=25);`
define a parameter	`Int_t .SetParameter(Int_t ipar,const char* parname,Double_t value,Double_t verr,Double_t vlow,Double_t vhigh)`
set function to be minimized	`.SetFCN(void (*)(Int_t&, Double_t*,Double_t&f,Double_t*,Int_t) fcn)`
fix a parameter	`.FixParameter(Int_t ipar)`
release parameter	`.ReleaseParameter(Int_t ipar)`
get pointer to active fitter instance	`static TVirtualFitter* .GetFitter()`
interaction with fitter	`Int_t .ExecuteCommand(const char* command, Double_t* args, Int_t nargs)`
example: start fit with MINUIT:	`double arglist[2]={5000,0.01}; .ExecuteCommand("MINIMIZE",arglist,2)`
example: error evaluation MINUIT / MINOS:	`ExecuteCommand("MINOS",arglist,0)`
get pointer to covariance matrix	`Double_t* .GetCovarianceMatrix() const`

interaction with MINUIT via global pointer gMinuit of class **TMinuit**

set DeltaChi2 value for error determination	`gMinuit->SetErrorDef(float DeltaChi2)`
get coutour line as `TGraph`	`(TGraph*)gMinuit->Contour(npoints, int par1, int par2)`

Contents

BIBLIOGRAPHY

[1] ReneBrun and Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. and Meth. in Phys. Res. A 389 (1997) 81-86. See also `http://root.cern.ch`.

[2] `http://root.cern.ch/drupal/content/users-guide`

[3] `http://root.cern.ch/drupal/content/reference-guide`

[4] `http://root.cern.ch/drupal/content/cint`

[5] `http://root.cern.ch/drupal/category/package-context/pyroot`

[6] `http://www.math.keio.ac.jp/~matumoto/emt.html`