ESIPAP European School of Instrumentation in Particle and Astroparticle Physics

2014 edition

Supervisers: Eric CHABERT, Eric CONTE

Computing session 3

Getting familiar with the framework ROOT: example of a silicon strip detectors analysis

Abstract:
This computing session is dedicated to the so-called ROOT framework. Born at CERN, this framework is used every day by thousands of physicists to analyze data and to perform simulations. This session is an opportunity to have a quick overview of basics but most used functionalities provided by ROOT. More than a simple tutorial, the session also aims to use ROOT for analysis purpose in the context of a silicon strip detectors data analysis. The program to be build could involve class development, however for the sake of simplicity and in order to focus on learning ROOT, simplest developments will be sufficient.

Pedagogical goals:

Software handling

- Browsing efficiently software documentation
- Understanding the description of ROOT classes
- Reading and understanding code examples. Extracting relevant information useful for the development of your programs

ROOT skills

- Reading data file (text and ROOT format)
- Reading and writing trees
- Creating and customizing graphics
- Fitting data distributions according to model functions
- Writing results in several formats

Compiling/linking

- Creating an executable file from a simple source file using ROOT classes.
- Using a Makefile with links to the ROOT libraries.

Requirements:

- Similar to the previous sessions

Contents

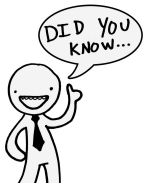
1 Foreword

Computing sessions belong to the educational program of the ESIPAP (European School in Instrumentation for Particle and Astroparticle Physics). Their goal is to teach the secrets of C++ programming through practical work in the context of high energy physics. The session is designed to be pedagogical. Except the *Physics context*, each section of the document is a milestone allowing to acquire computing skills and to validate them. The sections related to C++ programming are ranked in terms of complexity. In order to facilitate the reading of this document and to measure his progress, the student must **fill up the dedicated roadmap** which includes a check-list and empty fields for personal report.

In the document, some graphical tags are used for highlighting some particular points. The list of tags and their description are given below.

The student is invited to perform a pratical work by **writing a piece of code** following some instructions.

Analyzing or interpreting task is requested and the results must be reported in the roadmap.



Some **additional information** is provided for extending the main explanations. It is devoted to curious students.

A piece of **advice** is given to help the student in his task.

Concerning the evaluation of these computing sessions, all source files and other relevant digital documents must be provided to the examiner. Therefore they must be stored in a public folder on the LXPLUS session. The suggested naming convention is the following:

```
$HOME/public/TP1
$HOME/public/TP2
$HOME/public/TP3
$HOME/public/TP4
```

The student is invited to develop his code directly in such folder.

Part I
Getting started with ROOT

2 Introduction to ROOT

2.1 Documentation

The package ROOT is very well documented. Extensive information can be found in its website: http://root.cern.ch/drupal/ and we encourage to visit it. Instructions for downloading and installing the software are also provided for those who would like to install it on their own machine later. On the documentation part, several sources might be very useful depending on the purpose:

- **Users's guide**:
  This is a nice manual that introduces several applications that ROOT covers. A general presentation as well as examples are there presented. This is an interesting document to be partially read as an introduction to the software.

- **Reference guide**:
  This section of the website present all the classes, namespaces, typedef, etc that have been defined in ROOT. The documentation is quite similar to what you have obtained with DOXYGEN in the previous computing session. The Reference guide is then convenient to understand the inheritance, the relationship between classes, discover the available methods with their arguments as well as a general presentation of the class. This is a guide you should refer each time you want to access a class for which you do not remember what it does and the definition of the methods.

- **Tutorials**: Tutorials are available online and also on the installed ROOT version. There are code examples fulfilling specific goals and by the way showing the possibilities of ROOT. In another word, they show in situation how to use ROOT classes.

- **How To's**: That section replies to questions that beginners might face such as "How to Create and Fill a Tree" or even more advanced users such as "How to Use Mathematica From ROOT".

2.2 Overview of the main functionalities

An global overview of what ROOT offers can be reached here. A subset of interesting domains for which ROOT proposes interesting solutions are:

- **Data analysis and visualization**: This is probably the most used functionalities that ROOT provides among the HEP community. It offers possibilities to display histograms or graphs, to stack them, to superimpose them, etc.

- **Fitting and statistics**: ROOT give tools to represent mathematical functions and also to perform fits. Additional packages have been more recently developed to handle those particular tasks. They are called ROOFIT and ROOSTAT. We will not discuss them during that session but use the simplest fitting tools that ROOT gives access to.

- **Randomization and Monte Carlo**: Randomization is mainly used for Monte Carlo techniques. ROOT provides pseudo-random generators as well as more advanced tools to perform pseudo-experiments. We will not discuss those functionalities due to the lack of time.

- **Storing information/ROOT Format**: ROOT provides tools for input/output management. If offers classes to manipulate files, folders, and to store data in a given format. We will use during this session the so class TTree and TBranch classes.

- **Mathematical libraries**: Those libraries include mathematical functions (Poison, Gauss,etc), computation of statistic quantities (mean, rms), basic algorithms (search min, max elements, etc) and over features. Moreover a Linear Algebra package is available allowing the description and manipulation of matrices like their decomposition.

- **Physics vectors**: As ROOT is used in HEP, and it gives a description of physics vectors such as the energy-impulsion four-momentum called TLorentzVector in ROOT. This is commonly used to describe the particles colliding or produced in our experiments.

- **GUI**: It is possible to create your own Graphical User Interface based on ROOT classes. ROOT provides also an editor to easy the creation of such applications.

- **Detector**: The description of geometries are available in ROOT and are used to also describe detector. They might have an overlap with the software GEANT4 functionalities that we will be studied in the next session.

- **Event display**: ROOT as also been used to create event display applications, proofing that it might be used in many applications.

2.3 ROOT classes

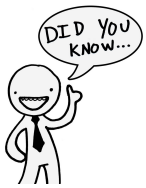
Inheritance and ROOT principles

As discussed during the lectures, ROOT respects C++ coding conventions that can be found here. By example, the name of all ROOT classes starts with a capital T. They all inherit from a mother class TObject. Inheritance with many generation as well as multiple inheritance is commonly used in that software.

In the Reference guide, browse in the documentation of classes that will be used later in the sessions such as TH1F, TGraph, TF1. See how the concepts mentioned in the lectures and applied in the previous sessions have been used in the structure of that software.

Global variables and environment

`ROOT` defines many global variables that control dedicated functionalities such as the memory management, the graphic style management, etc. Some of them are listed below:



- gEnv
- gApplication
- gSystem
- gStyle
- gObjectTable
- gClassTable
- gFile, gDirectory, gPad, gRootDir, gProgName, gProgPath

3 Software Handling

3.1 Using ROOT intepreter

If ROOT has been correctly installed on the machine or server you are accessing as it is with lxplus, the software can be launch as following:

```
bash$ root
```

A window showing the ROOT version will appear on the screen. Then a user console is openend with a prompt where instructions can be launched:

```
root [0]
```

You can now use ROOT interpreter, called CINT, to execute sequentially C++ instructions as shown below:

```
root [0] float a = 1.2345;
root [1] cout<<"a^0.5 = "<<sqrt(a)<<endl;
```

The interpreter is sometimes more permissive that g++ compiler would be. This might sometimes be useful but it has to be used with cautious. If C++ rules are not followed, the results might not be the one expected. Some examples:

- if you forget to type a semi-colon at the end of the line, the instruction will be anyway executed.
- the use of std namespace is not required.
- if you affect a value to a variable that has not yet been declared, it will give you a warning but will still be executed.

The interpreter might be convenient for small applications, quick tests, but its usage should be discouraged for more advanced applications.

The interpreter provides also tools too than can be used in complement of additional documentation. Let's consider a simple example with a class that describes an histogram of double: TH1D. Instantiate an object h as following:

```
root [0] TH1D h;
```

If you want to know what are the available public methods and data members of TH1D, you can simplify type h. and then use the tabulation. A long list should appear. If you remember that the name of the methods starts with GetBin, you can type h.GetBin and use the tabulation and then a shorter list should appear. If the name of the methods you were looking for is GetBinContent, type h.GetBinContent(and all the existing methods with their arguments will appear.

Moreover two specific virtual methods inheriting from TObject, the base class of all ROOT classes, might be useful:

- Print(): it summaries few information on the object

```
root [3] h.Print()
TH1.Print Name  = , Entries= 0, Total sum= 0
```

- `Dump()`: it gives you the list of all data members with their value and their meaning

```
root [4] h.Dump();
==> Dumping object at: 0x00000000017c4310, name=, class=TH1F
```

- `Inspect()`: it does something equivalent to `Dump()` but in a graphic way.

- `DrawClass()`: it opens a window on which all inherited classes and the current class present their public methods.

The above mentioned procedures can be applied to any `ROOT` class.
Finally to quit `ROOT` interpreter, you simply have to to do:

```
root [5] .q
```

The instructions that have been launched are stored in a file /.root_history. Moreover, in a similar way as you would do in a Terminal, you can use the top and bottom array to browse in the history from the `ROOT` interpreter.

.

3.2 Using `ROOT` macros

Compared to the interpreted way, a more elaborate way to use `ROOT` consists in typing directly all the instructions in a file. The instructions might be encapsulated in brackets. Then to launch it, two procedures could be used. If the file is called myMacroFile.cpp:

```
root  myMacroFile.cpp
```

or

```
root
root [0] .X myMacroFile.cpp
```

To manage more complex applications, one could decide to use functions defined in macro files. They can be loaded and launched with specific arguments as following:

```
root
root [0] .L myMacroFileWithFunctions.cpp // the file is loaded
root [1] myFunction(3.2,"histoName");
```

The second instruction call the function `myFunction(float x, char* name)`.

- Copy the macro files in your working folder by the commands:

```
cp ~echabert/public/ESIPAP/myMacroFile.cpp ./
cp ~echabert/public/ESIPAP/myMacroWithFunctions.cpp ./
```

- Open the files and study the structure of these macros.
- Test the commands given for running macros (or functions from macros).

3.2.1 ROOT compilation: ACLIC

The performances obtained with an interpreted code are poor compared to the ones obtained with the same code but compiled. ROOT proposes a compiler to take that in charge internally:ACLIC. Using the previous example but in a compiled way, one could execute the following instructions:

```
root -l
root [0] .L myMacroFile.cpp++ // the file will be compiled
Info in <TUnixSystem::ACLiC>: creating shared library
/home/login/path/myMacroFile_Cpp.so
root [1] myFunction(3.2,"histoName")
```

ACLIC compilation options can be defined, for additional information, refer to the dedicated documentation. One interesting feature of ACLIC is that you do not have to take care of loading the ROOT libraries ourself.

Use the previous example and execute it in a compiled way. Quantify what are the difference in term of execution time.

It is interesting to notice that ROOT can be launch with several options. To know then, you can simply type:

```
root --help
```

As you can see from the output, it is also possible to use dedicated file to configure ROOT usage.

- **.rootrc** is a file containing some global default for your ROOT session. There are three locations where the system looks for this file: $ROOTSYS/system.rootrc, /.rootrc and ./.rootrc (the latter taking precedence over the former). If you type in a root session the command `gEnv->Print()` you see which defaults are active.
- **.rootalias.C**: It is loaded and executed at ROOT startup. If can define some often used functions.
- **.rootlogon.C**: It contains the code that will be executed at ROOT startup.
- **.rootlogoff.C**: it is called when the session is finishing.

These last 3 files are always taken from the current working directory. If you would like to have a global version of one or all of these files make the change in your /.rootrc.

3.2.2 $ROOTSYS and access to tutorials

If the ROOT installation has been properly done, an environment variable ROOTSYS should have been set up. It refers to the directory where headers, source code, libraries and tutorials are installed. It might be interesting for pedagogical reasons, to read some tutorials. They are contained in the directory `$ROOTSYS/tutorials`. You can scrutinize them, execute them and also copy them.

Execute the tutorial `$ROOTSYS/tutorials/demoshelp.C`

3.3 Using ROOT classes in a standalone program

The most general and flexible way to use ROOT functionalities is to write your own standalone compiled code linked to ROOT libraries. To do that, you should follow those recommendations:

- Include all the appropriate header file.
  If you use a TH1F histogram, you should include "TH1F.h".
- Use compiler options to link your code with the ROOT libraries. Let's consider the example of a program main.cpp that use histograms, you should compile it as following:

```
1 g++ -o main main.cpp `root-config --cflags --glibs`
```

- Type `root-config` to observe exactly what it does.
- Write a simple code that instantiates a TH1F object, fill it with few entries and then print on screen the mean and the rms of the distribution.
- Compile this simple code as a standalone program.

If you want to display graphical elements on screen while running your standalone compiled code, you could follow the instructions below:

```
#include "TApplication.h"
#include "TSystem.h"
#include "TCanvas.h"

int main(int argc, char **argv) {
    TApplication* theApp = new TApplication("App", &argc, argv);
    // Write the core of your program
    // You can display graphical elements
    TCanvas* c = new TCanvas();
    c->Draw();
    // Run the application - the program will not stop
    theApp->Run();
    delete theApp;
}
```

3.4 ROOT Graphical User Interface

Many manipulation of `ROOT` objects can be done in a graphic way. This is sometimes really convenient especially for the graphical display like the histograms, the graphs, etc.

TBrowser

ROOT provides a class that is dedicated to browse directories, files and even `ROOT` file content. Type the following command in `ROOT`:

```
TBrowser b;
```

A window will appear. You can now browse the directories and files.

Let's take an example file for explaining how to browse a ROOT file. First copy in your working folder the ROOT example file:

```
cp ~echabert/public/ESIPAP/demo.root ./
```

Then with an instance of `TBrowser`, try to access on the file and double-click on it. You should now be able to browse its content. Double click on the histogram `myHisto`. You can now use the GUI to modify the properties of the histogram. On the window Canvas_c1, you can use right click over different elements and access to their public methods:

- TH1F
- TCanvas
- TFrame
- TPaveStats

Use the `Rebin` method of TH1F and see what happens.

On the top menu, activate Editor from the top menu. The content on the new frame that appeared depends on the latest graphical object you have clicked on. You can now try to modify many parameters like the style, width, color of the lines, background, etc.

- Take few minutes to get familiar with the GUI if you are not used to before.
- Modify the properties of the histogram with the Editor and observe the result

Saving histograms in several formats

Once you have a modified displayed histogram, you should think about saving it. This can be done with several formats:

- .root: this will allow you to reopen a file that contain the histogram later on.
- .C: this might be useful to store the code that describe your histogram.
- .pdf .ps .esp: could be convenient to include it in latex file later on.
- image format(.jpg,.png,.tiff,...)

- Try to save your histogram in different format.
- Open the .C file that you have created and see what it contains. You should observe a long list of lines. This .C file might be useful to understand how you could modify your macros/programs later in order to have a visualization that corresponds to what you wanted.

Part II
Silicon strip modules commissioning

4 Physics context

The subject proposed during that session concerns the characterization of silicon strips detectors. Those detectors are commonly used in many HEP experiments and beyond. Part of the tracker subdetector of the LHC experiments are based on such technology. By example, the CMS tracker is composed of around 15000 silicon strip modules corresponding to a total surface of about 200 m^2. The figure 1 show the barrel and end cap pictures of the CMS tracker. The whole description of those detectors and their associated readout chain will not be given in that document but interested students could find more details in many references such as the Technical Design Report of the LHC experiments. (link). An incoming particle crossing a silicon strip module deposits a small fraction of its energy that can lead to the observation of hits. Those latter can be clustered and then associate among different layers composing the detector in order to reconstruct tracks with dedicated algorithms.

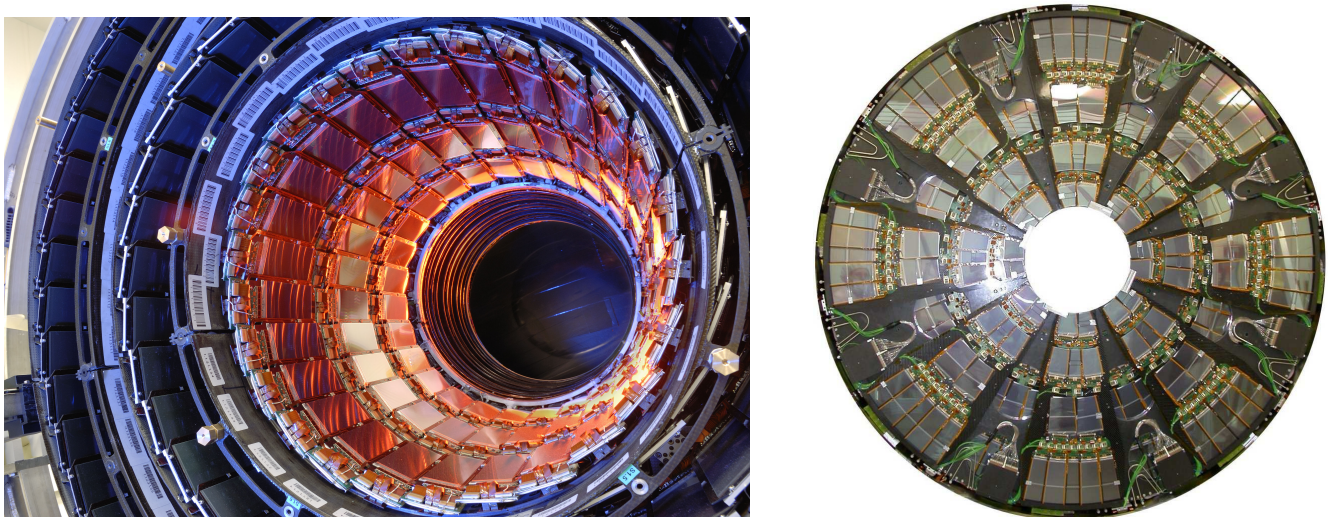


Figure 1: Pictures of the silicon strip CMS tracker composed of a barrel (left) and tow end-caps (right)

4.1 Strip sensor description

Silicon strip sensors are semi-conductors whose aim to amplify the charge induced by an incoming particle in the sensitive volume of the detector, *ie* a depleted region created under the application of a high voltage. The collected charged leads to a signal preamplified and shaped at the end of each strip by dedicated front-end ASICs. This signal is then sent to the control room often via optical fibers. The signal could be digitized at front-end or back-end level depending on the design chosen by the experiment. In our case, the signal is digitized by a 10 bits ASIC leading to a value in ADC count ranging from 0 to 1023. The figure 2 presents a transverse scheme of a silicon strip module illustrating the principle of operation. Additionally a picture of a CMS silicon strip detector with its electronics is also presented.

Silicon strip sensors have properties that depends on many parameters corresponding to their geometry (strip length, thickness,etc) , their production process, the choice of the silicon doping, etc.

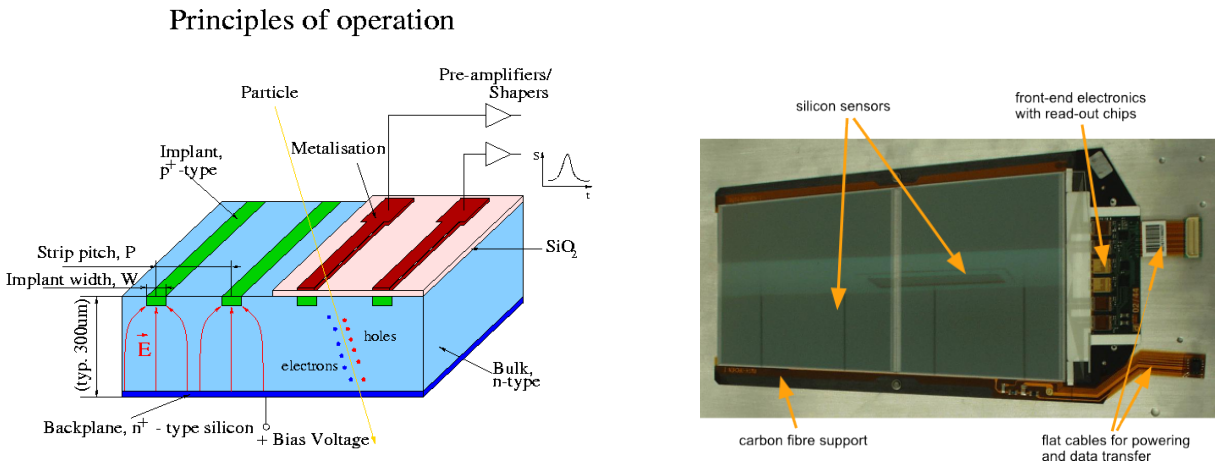


Figure 2: Traversal scheme of a silicon strip module on the right and picture of CMS silicon strip module with its front-end electronics on the the right)

4.2 Characterization of silicon strip detector noise

The noise is one of the most important characteristics of the detector. It has to be low enough to ensure that energy deposited by the incoming minimum ionizing particles will lead to an good enough detection efficiency. One interesting quantity used as estimator is the so-called S/B ratio where S is the signal strength and B is the average noise. By example, S/B is about 30 for the CMS silicon strip modules.

The noise arise from electronic activities that appear both on the sensor and the electronic circuits. The electrical behavior of a silicon strip module can be simply as the scheme presented on the figure 3.

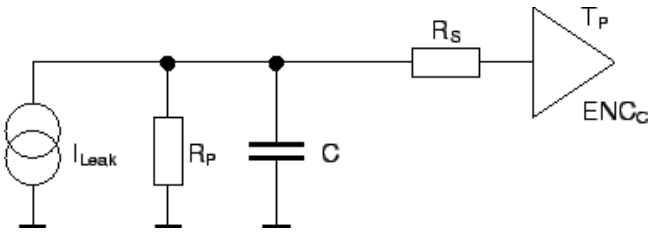


Figure 3: A simplified equivalent electrical scheme of a silicon strip module.)

Noise can be divided into several sources:

- noise arising from the capacitance (B_C)
- shot noise arising from leakage current ($B_{I_{leak}}$)
- parallel thermal noise arising from bias resistance (B_{R_p})
- serial thermal noise term arising from the metal strip resistance (B_{R_s})

For the sake of simplicity, we will not consider the B_{R_s} term during that session. Moreover, we will consider that most of the module characteristics are fixed parameters and we will focus on only few noise dependencies as express below:

$$B^2 = B^2_{I_{leak}} + B^2_C + B^2_{R_p} \quad (1)$$

$$B_C = a + b \times L \quad (2)$$

$$B_{I_{leak}} = c \times \sqrt{I_{leak}} \quad (3)$$

$$B_{R_p} = d \times \sqrt{e \times \frac{T}{R_p}} \quad (4)$$

where T is the temperature, L is the strip length and a,b,c,d are constant parameters. The units are arbitrary.

4.3 Commissioning runs

Before using silicon strip detectors for physics analysis, physicists in charge of the detector operation should perform commissioning runs. They are done in absence of signal, *ie* while beam is off. The data provided later in that session originate from a very simple simulation tool, however they are sufficiently realistic to reflect the behavior of silicon strip modules.

Instrumental setup

We will consider that the data registered correspond to a bunch of modules having different length. The table 1 describes the modules used, their length and their ID range.

Number of modules	Module Id range	Strip length (cm)
50	[1-50]	9
50	[51-100]	11
50	[101-150]	13
50	[151-200]	15

Table 1: Description of the modules used during commissioning runs.

Data format

The data analyzed later in the session follow the format described below:

- event id (integer)
- detector id (integer)
- leakage current measured (real)
- signal measured in each of the 128 channels expressed in ADC counts: 128 integers belonging to [0-1023]

Commissioning tasks

The commissioning runs have to fulfill many goals. We will not describe all of them (initialization, synchronization, alignment,etc), but only discuss the ones relevant for the exercise proposed today.

- `Measuring the pedestal`: the average activity in each of the channel in absence of signal is not null and has to be measured. It is the quantity that will has to be subtracted later to check if a signal has been observed
- `Measuring the noise`: it corresponds to the fluctuation around the pedestal for each channel. To detect a signal, the response of a channel will be compare to the noise fluctuations on top of the pedestal. A typical 3 σ criterion could be apply to reconstruct hits.
- `Detecting noisy and dead channels`. For many physical reasons, such as bonding problems, channels could have a really low response or at the contrary a too important noise. Such channels can be spotted and then masked during the physics runs.

5 First step: reading data

Pedagogical goals:

- Reading data: transforming data contained in a text file into a tree
- Writing a code skeleton to analyze a tree
- Manipulating root files

Ressources:

Documentation	• Users guide: Tree and I/O sections • Tutorials:tree section • How To's: create, fill and read trees
ROOT classes	• TTree • TBranch • TFile • TBrowser

5.1 Converting a data stored in a text file into a ROOT tree

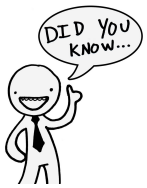
Start that section by copying the file `~echabert/public/ESIPAP/Test.dat` in your working directory. This file contains hundred events corresponding to the data acquisition of the 200 silicon strip modules of the setup, totalizing 20000 lines. Each line corresponds to the acquisition of a given module, specify by its module-id, taken at a given event labeled by an event-id. Then the current leakage measurement precedes the signals measured in each of the 128 channels of module expressed in ADC counts (128 integers ranging in [0-1023]). The program developed in that section and the next ones must have to be compiled.

- Read the basic documentation about TTree manipulation linked above in a partial an efficient way. Find the relevant information and the instructions to perform the tasks listed below.
- **Read the `~echabert/public/ESIPAP/Test.dat` and load it into a TTree by calling the method `ReadFile`.**
- **Is the first line of the text file useful ? If yes, what does it do?**
- **Save the tree into a `ROOT` file. Note that you can give a name to the TTree via the construtor or the method `SetName()`**

5.2 Reading data from the tree

- **Create a skeleton of code that can load that tree, and set a loop over its entries to be able to access the tree content. You will have to use `SetBranchAddress()` method. Find an automated way to load the branches containing the digital signal of each channel. Variables referring to the tree content will be contained in a C structure or in a simple C++ class.**
- **Check that you are able to access the data and that you are ready to implement your future analysis. You could simply try to print on screen one entry and compare it with the original file.**

It is possible to create an automatic code skeleton from a tree. Following the instructions below you should be able to run on a given tree according its data structure.




```
1 TTree* tree;
2 //here you need to access to a tree,
3 //by example from a TFile
4 tree->MakeClass("ClassName");
5 //The previous line will generate two files:
6 //ClassName.h and ClassName.cc
```

Open the file `ClassName.cc` and read the instructions to launch the program. This example can help you to undestand the structure of a code that reads a tree. As you can see, this solution is written in a way to be executed in the ROOT interpreter. Moreover, it might be convenient to write an equivalent code in a more condensed way, especially for reading the vXXX branches of the tree.

6 Data analysis: creating histograms and graphs

Pedagogical goals:
The goals of that section is to be trained to obtain basic graphical results that are commonly used in our disciplines. That knowledge will be directly useful for data analysis latter.

- Getting familiar with TTreeViewer
- Creating and customizing one-dimensional and bi-dimensional histograms
- Manipulating histograms in an automated way

Ressources:

Documentation	• Users guide: Histograms and Graphs sections • Tutorials: histo and graph sections • How To's: Histograms and Graphics and Graphical User Interface sections
ROOT classes	• TCanvas • TH1F • TH2F • TGraph • TGraphErrors • TLegend • TStyle • TTreeViewer

6.1 Using the TTreeViewer

`ROOT` offers a graphical solution to run simple analysis interactively on a `TTree`. The class in charge of that is called `TTreeViewer`. This is the two options to launch it on your tree:

1. From the `TBrowser` : Select a tree in the `TBrowser`, then call the `StartViewer()` method from its context menu (right-click on the tree).
2. From the command line : Start a `ROOT` session in the directory where you have your tree. You will need first to load the library for `TTreeViewer` and optionally other libraries for user defined classes (you can do this later in the session) :

```
root [0] gSystem->Load("TTreeViewer");
//For a tree MyTree contained in a file MyFile, you can do :
root [1] TFile file("Myfile");
root [2] new TTreeViewer("Mytree");
```

For further documentation, you can read the class description here

- Take few minutes to play with the TTreeViewer and display quantities.
- Display 1-D or 2-D histogram while dragging variables of interest.
- Even cuts can be added. By example you can apply a selection based on the detector id.

6.2 Using ROOT graphics in a program

This section can be realized with some freedom. However, beginners are encourage to build their program following successive steps:

1. Using TTreeViewer to check the expected results for basic histograms

2. Writing code in a macro, running it interactively and checking that the histograms performed correspond to the ones obtained previously

3. Adapting the code to become a standalone program link to `ROOT` libraries

The list of histograms we are interested in will be given below. The displayed quantities can be obtained by running on the whole modules or such on one subset or few subsets depending on the analysis purpose. You should think in advance of a way to implement a program that could accommodate to such changes. You are encourage to use functions that takes arguments such as the module id, the event id, etc. In view of time execution "optimization", think about a solution where you don't loop over the `TTree` each time you want to produce a new histogram ...

Producing your first histograms

The first histogram you could create is simply the silicon strip signal measured on a given module for a given event. This histogram should contain 128 bins corresponding to the 128 channels.

Histograms at module level

- Write functions that can create and display the following quantities:
  - leakage current distribution
  - pedestal distribution
  - noise distribution
- Test your code interactively and check the distributions obtained
- `TCanvas` can be divided in order to display the 3 distributions of a given module at once.
- Automatize the creation of such distributions for all modules and save them into folders of a `TFile`. This will imply the manipulation of string for the name of histograms and folders. Note that `ROOT` implemented a class `TString` than could eventually be used instead of std::string.
- Define your own graphical chart (line width, colors, etc) and apply it to all your distributions. One should have a look to the class `TStyle`.
- Save all you distributions in the same pdf file. For readiness think about relevant title for your histograms (variable, module id)

Histograms of mean values for all modules

We are now interested to check the distributions of quantities average for each module. `TH1` class provides functions that gives access to the mean and rms of the distributions. Check on the documentation and use them to fill the above mentioned distributions

- Write the code to obtained the following quantities
  - mean and rms leakage current distributions
  - mean and rms pedestal distributions
  - mean noise distribution
  - mean leakage current versus mean noise (TH2F)
- For the 2D-histogram, check the available display options. Moreover, you will see on the documentation that `TH2` class provided a function to return linear correlation. Use it and interpret the result. One could think about producing additional 2D-histograms.
- As for the previous subsection, save the histograms into folders in the `TFile` as well as in a pdf.

Analysis depending on the strip length

We are now interesting in observing quantities for subset of modules that have the same strip length.

- Compute the mean noise for each group of modules having the same length and create a graph from the 4 points obtained. Propagate the uncertainty on each point into the graph.
- Similar graphs could be obtained from over histograms.
- Save it into a `TFile` and a pdf.

7 Data analysis: fitting

Pedagogical goals:

- Handling fit functionality and getting used to the fit options
- Fitting distribution with predefined or user-defined functions
- Automation of fits other a collection of histograms

Ressources:

Documentation	• Users guide: Fitting Histograms sections • Tutorials: fit sections • How To's: Fitting Histograms section
ROOT classes	• TF1 • FitPanel

7.1 Using the FitPanel

ROOT offers a graphical solution to perform fit interactively on histograms. Read the appropriate documentation here.

- Train yourself on the following distributions:
  - leakage current distribution for a given module
  - Distribution of mean leakage current averaged per module
- For each distribution:
  - Find the appropriate model function
  - Check the value of the χ^2 returned
  - Test the stability of the fit depending on the range used
  - Try several minimization options and compare the results

7.2 Automatized fits from a compiled program

The goal is now to reuse the conclusion obtained from the the previous tries using the FitPanel and to perform the same fit in our program.

- Call the Fitter and fit the histograms previously used
- Retrieve the parameters of the fitted function as well as their errors
- Compute a probability (p-value) from the χ^2 returned (see documentation in TMath)
- Superimpose the fitted function to the histogram produced. Customize the display (color, size, etc)

Once you are able to reproduce the previously obtained results in our program, you could envisage to extend that for a bunch of histograms. Automatized fits might however be dangerous if one doesn't properly look to some meaningful quantities like the errors of the p-value.

- Fit the leakage current distribution for each module, retrieve the mean, the σ and the p-value and fill an histogram for each of those 3 quantities
- Fit the distribution of the mean noise as function of the strip length
- Fit the mean leakage current versus mean noise per module (2D distribution)

- How would you interpret the previous results obtained ?
- Can you extract noise parameters described in the beginning of that document from your fit ? If not, what would you propose.