

# CERN Summer Student report

**Author:** Luca Giommi (EP-SFT), *luca.giommi2@studio.unibo.it*

**Supervisor:** Axel Naumann

**Co-Supervisor:** Danilo Piparo

September 23, 2016

## Abstract

I worked inside the ROOT team of EP-SFT group. My project focuses on writing a ROOT class that has the aim of creating histograms from a TChain. The name of the class is *TSimpleAnalysis* and it is already integrated in ROOT. The work that I have done was to write the source, the header file of the class and also a python script, that allows to the user to use the class through the command line. This represents a great improvement respect to the usual user code that counts lines and lines of code to do the same thing.

(Link for the class: <https://root.cern.ch/doc/master/classTSimpleAnalysis.html>)

## About the project

We can use the new class *TSimpleAnalysis* through a python script that I wrote and, in this way, we can create histograms just in one command line. There are two ways to use the python script. In the first one we can pass the options as arguments, that allow to create and store the histograms (as we can see in Figure 1). Here we are passing the name of the input file *hsimple.root* that we want to use, the output file *output.root* in which we want to store the histograms, and the expression  $x=px$  if  $px > 0$  that represents that we are showing in the histogram (of name  $x$ ) the function  $px$  with the cut  $px > 0$ .

```
rootdrawtree --input hsimple.root --output output.root --histo 'x=px if px > 0'
```

Figure 1: Example of usage of the *TSimpleAnalysis* class through the *rootdrawtree* python script, passing the necessary options.

Another way to use the python script is to pass a configuration file (Figure 2). The configuration file has a particular syntax and an example is shown in Figure 3. I have also written a tutorial inside ROOT and we can use it passing it to the python script as configuration file.

```
rootdrawtree configFile.txt
```

Figure 2: Example of usage of the *TSimpleAnalysis* class through the *rootdrawtree* python script, passing the configuration file.

```
output.root
ntuple
hsimple.root
hsimple2.root
hpx=px if px<-3
hpy=py
hpz=pz if pz>8|
```

Figure 3: Example of configuration file.

In the first line there is the name of the output file in which the histograms will be stored. In the second line there is the name of the tree of which we want to understand better the behaviour of its components. The name of the tree is an optional line if there is only one tree inside the first input file. Then there is a block of lines that represents the input files that we want to use to create the histograms. In the example shown in Figure 3, the *hsimple.root* file is used, and we can find it inside the ROOT tutorial folder. Then we have a block of lines in which the expressions that allow to build the histograms are written. Each one of these lines contains the '=' character that divides the line into the left side, that is the name of the histogram that we will create, and into the right side in which the function that will be shown into the histogram is written, with the optional cut that starts with the word 'if'. Inside the code, the function that is used to create the histograms is the *TTree::Draw* function shown in Figure 4.

```
chain.Draw(expr + ">>" + histoName, cut, "goff");
```

Figure 4: *TTree::Draw* function used to build the histograms.

Then we might have a more complex configuration file, in which we can use spaces, tabs, blank lines in order to give the text more readable to the user. Furthermore we can use comment lines and comments at the end of the lines, starting with the '#' mark, in order to specify what we are writing into the configuration file.

If we have a look at the output file, we see that we can build one, two and three dimensional histograms (TH1F, TH2F, TH3F). In the example in Figure 5, we can see a three-dimensional histogram with a multiple cut.

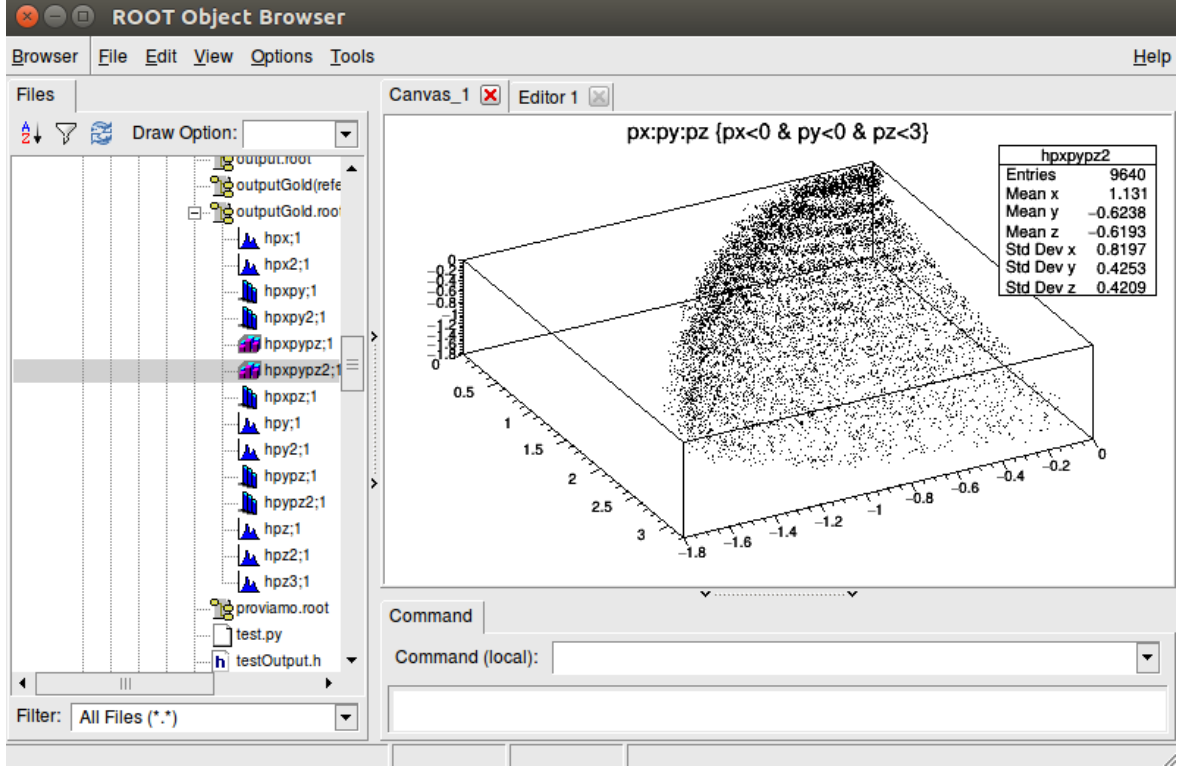


Figure 5: Example of three-dimensional histogram created using *TSimpleAnalysis* class.

Another important improvement of the class respect to the usual user code is the part of handling errors. In fact usually the user doesn't write a proper part of handling errors. On the contrary using the *TSimpleAnalysis* class, for each kind of error it will print out an error message that tell us very clearly what is the problem, helping us in its correction. For example an error message will be printed if we fail to open the configuration file, or the output file, or if there are problems in the chain (when for example there isn't one of the required input files inside the user folder, as shown in Figure 6) or if there are problems in the expression lines.

```
Error in <TFile::TFile>: file hsimpleNOTEXISTS.root does not exist
Error in <TSimpleAnalysis::Run>: Load failure in file hsimpleNOTEXISTS.root: cannot open the file
```

Figure 6: Example of error when an input file can not be opened.

In fact we might miss the '=', or we might miss the name of the histogram, or we might miss the function that we want to show, or we might miss the cut expression even if the word 'if' is in the expression. In these error cases about expressions, the kind of error and also the line of the configuration file (or the expression that we passed as argument to the script) will be printed as shown in Figure 7. This is very useful for the user because in this way it is easier to correct errors.

How it is possible to understand if the class works correctly? To do it I had to check if the output file produced using the class is the same of a reference, that we know is correct. Here we have to test if they have the same number of keys, with the same name, and if they

```

// Report any errors if occur during the configuration proceedings
if (!errorMessage.empty()) {
    ::Error("TSimpleAnalysis::Configure", "%s in %s:%d", errorMessage.c_str(),
            fConfigFile.c_str(), numblLine);
    return false;
}

```

Figure 7: Example of handling errors in the expressions.

have the same bin content with same errors for the same histograms. Then I had to check also if the python script worked correctly. In order to do it, I had to compare if the output file produced by the script passing the configuration file is the same of the reference. Then I had also to check if the output file produced by the script passing the configuration file is the same produced passing all the options.

The last part of the project is the parallelization part. What we want to do is to use the number of cores that we have in the computer in order to create histograms from multiple input files in parallel way. As shown in Figure 8, we have  $n$  input files that we want to use, and we have  $m$  cores inside the computer. For each file we create a vector of histograms in parallel way using the number of cores that we want. After this parallel part we add the content of the histograms with the same name into a final histogram that will be stored inside an output file. So this file will contain some histograms that come from the addition of the  $n$  ones that were built from the  $n$  input files, after we had used  $m$  cores to produce them. This parallelization part is very important because if we want to use multiple input files and if we use multiple cores to create the histograms, we can gain in time of execution and this is a great improvement.

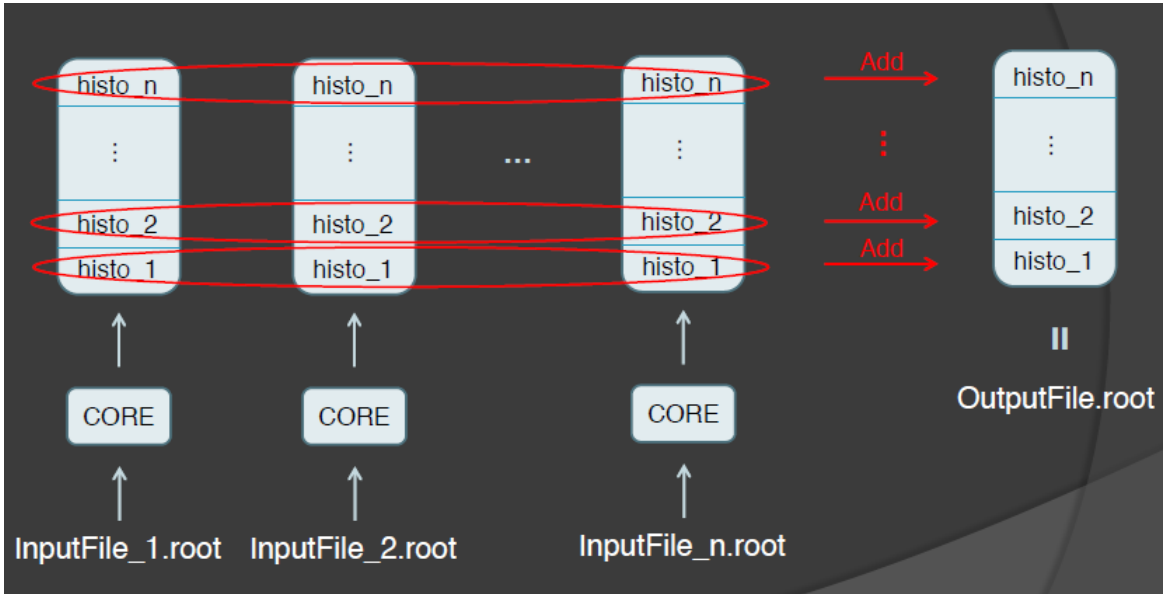


Figure 8: Scheme of the parallel proceedings.

Unfortunately in this moment there is a problem that locks this gain in time, that in future will be discussed inside the ROOT team. The problem is that when we use an input file and we want to create an object of the class, we need a dictionary for this build, otherwise

we need to emulate the layout. So if we have a dictionary, the thing will be super-fast but unfortunately we don't have it in the most of cases. This causes that each time, each core has to emulate what is inside the input file and so there is a waste of time in creation of these emulations every time. This lock causes loss of time and for the moment, with this problem, the parallelization part is disabled but is present inside the class. So the serial part now is enabled but when this thing will be improved, the parallelization part could be used easily, just with a change of modality inside the class.

## Considerations

Summing up, this tool addresses a common use case for physicists, making it significantly simpler to have a quick look at data stored in ROOT files. The class has been uploaded to the master, and will be part of the upcoming release 6.08.

Here at CERN, I have had the opportunity to work close to the main experts of ROOT and this has been a great improvement for my person. In fact I could talk with some of them, comparing our ideas and learn more how to work inside a group, how to share our knowledge. In these months I learned more C++, more ROOT, working with a big piece of software and I have had the opportunity to integrate my work inside ROOT, writing a new class that will improve in future the work of the physicists. I am a physicist but this melting of 'Lectures' (with more physics) and project (focused on computing) gave me the possibility to improve my physics background and above all my computing science field, that I know will be an important part of me in future.

Thanks to the help of the people here at CERN, in particular my supervisor and the members of the ROOT team, I have had the possibility to improve myself, to share ideas with people from different countries, improving the English and my ability to relate with the others. I think that this experience has been fantastic and also the project was great and I am very glad to have written and to have contributed to something that will be used in future and will help the other physicists.

*Luca Giommi*