Universiteit Gent
Faculteit Wetenschappen
Vakgroep Fysica en Sterrenkunde

GIF++ Offline Analysis Tool

An extensive documentation

Alexis Fagot

Offline Analysis Tool v6.0
for GIF++ DAQ files

Table of Contents

List of Figures

List of Acronyms

List of Acronyms

ADC	Anolog-to-Digital Converter
CMS	Compact Muon Solenoid
DAQ	Data Acquisition
DQM	Data Quality Monitoring
FEE	Front-End Electronics
GIF++	new Gamma Irradiation Facility
HV	High Voltage
PT	Pressure and Temperature
RPC	Resistive Plate Chamber
TDC	Time-to-Digital Converter
webDCS	Web Detector Control System

1

Details on the offline analysis package

The data collected in new Gamma Irradiation Facility (GIF++) thanks to the DAQ is difficult to interpret by a human user that doesn't have a clear idea of the raw data architecture of the ROOT data files. In order to render the data human readable, a `C++` offline analysis tool was designed to provide users with detector by detector histograms that give a clear overview of the parameters monitored during the data acquisition [1]. In this appendix, details about this software in the context of GIF++, as of how the software was written and how it functions will be given.

1.1 GIF++ Offline Analysis file tree

GIF++ Offline Analysis source code is fully available on github at `https://github.com/afagot/GIF_OfflineAnalysis`. The software requires ROOT as non-optionnal dependency as it takes ROOT files in input and write an output ROOT file containing histograms. To compile the GIF++ Offline Analysis project is compiled with cmake. To compile, first a `build/` directory must be created to compile from there:

```
mkdir build
cd build
cmake ..
make
make install
```

To clean the directory and create a new build directory, the bash script `cleandir.sh` can be used:

```
./cleandir.sh
```

The source code tree is provided below along with comments to give an overview of the files' content. The different objects created for this project (`Infrastructure`, `Trolley`, `RPC`, `Mapping`, `RPCHit`, `RPCCluster` and `Inifile`) will be described in details in the following sections.

```
GIF_OfflineAnalysis
 ├── bin
 │    └── offlineanalysis ........................ executable
 ├── build ............................ cmake compilation directory
 │    └── ...
 ├── include ............................ list of C++ header files
 │    ├── Cluster.h ..................... declaration of object RPCCluster
 │    ├── Current.h ..................... declaration of GetCurrent analysis macro
 │    ├── GIFTrolley.h .................. declaration of object Trolley
 │    ├── Infrastructure.h .............. declaration of object Infrastructure
 │    ├── IniFile.h ..................... declaration of object IniFile for ini parser
 │    ├── Mapping.h ..................... declaration of object Mapping
 │    ├── MsgSvc.h ...................... declaration of offline log messages
 │    ├── OfflineAnalysis.h ............. declaration of data analysis macro
 │    ├── RPCDetector.h ................. declaration of object RPC
 │    ├── RPCHit.h ...................... declararion of object RPCHit
 │    ├── types.h ....................... definition of useful variable types
 │    └── utils.h ....................... declaration of useful functions
 ├── obj ................................ binary files created by compiler
 │    └── ...
 ├── src ................................ list of C++ source files
 │    ├── Cluster.cc .................... definition of object RPCCluster
 │    ├── Current.cc .................... definition of GetCurrent analysis macro
 │    ├── GIFTrolley.cc ................. definition of object Trolley
 │    ├── Infrastructure.cc ............. definition of object Infrastructure
 │    ├── IniFile.cc .................... definition of object IniFile for ini parser
 │    ├── main.cc ....................... main file
 │    ├── Mapping.cc .................... definition of object Mapping
 │    ├── MsgSvc.cc ..................... definition of offline log messages
 │    ├── OfflineAnalysis.cc ............ definition of data analysis macro
 │    ├── RPCDetector.cc ................ definition of object RPC
 │    ├── RPCHit.cc ..................... definition of object RPCHit
 │    └── utils.cc ...................... definition of useful functions
 ├── cleandir.sh ........................ bash script to clean build directory
 ├── CMakeLists.txt ..................... set of instructions for cmake
 ├── config.h.in ........................ definition of version number
 └── README.md .......................... REAMDE file for github
```

1.2 Usage of the Offline Analysis

In order to use the Offline Analysis tool, it is necessary to know the Scan number and the High Voltage (HV) Step of the run that needs to be analysed. This information needs to be written in the following format:

```
Scan00XXXX_HVY
```

where `XXXX` is the scan ID and `Y` is the high voltage step (in case of a high voltage scan, data will be taken for several HV steps). This format corresponds to the base name of data files in the database of the GIF++ Web Detector Control System (webDCS). Usually, the offline analysis tool is automatically called by the webDCS at the end of data taking or by a user from the webDCS panel if an update of the tool was brought. Nontheless, an expert can locally launch the analysis for tests on the GIF++ computer, or a user can get the code on is local machine from github and download data from the webDCS for is own analysis. To launch the code, the following command can be used from the `GIF_OfflineAnalysis` folder:

```
bin/offlineanalysis /path/to/Scan00XXXX_HVY
```

where, `/path/to/Scan00XXXX_HVY` refers to the local data files. Then, the offline tool will by itself take care of finding all available ROOT data files present in the folder, as listed bellow:

- `Scan00XXXX_HVY_DAQ.root` containing the Time-to-Digital Converter (TDC) data (events, hit and timestamp lists), and
- `Scan00XXXX_HVY_CAEN.root` containing the CAEN mainframe data recorded by the monitoring tool webDCS during data taking (HVs and currents of every HV channels). This file is created independently from the Data Acquisition (DAQ).

1.2.1 Output of the offline tool

1.2.1.1 ROOT file

The analysis gives in output ROOT datafiles that are saved into the data folder and called using the naming convention `Scan00XXXX_HVY_Offline.root`. Inside those, a list of `TH1` histograms can be found. Its size will vary as a function of the number of detectors in the setup as each set of histograms is produced detector by detector. For each partition of each chamber, can be found:

- `Time_Profile_Tt_Sc_p` shows the time profile of all recorded events (number of events per time bin),
- `Hit_Profile_Tt_Sc_p` shows the hit profile of all recorded events (number of events per channel),
- `Hit_Multiplicity_Tt_Sc_p` shows the hit multiplicity (number of hits per event) of all recorded events (number of occurences per multiplicity bin),
- `Strip_Mean_Noise_Tt_Sc_p` shows noise/gamma rate per unit area for each strip in a selected time range. After filters are applied on `Time_Profile_Tt_Sc_p`, the filtered version of `Hit_Profile_Tt_Sc_p` is normalised to the total integrated time and active detection area of a single channel,
- `Strip_Activity_Tt_Sc_p` shows noise/gamma activity for each strip (normalised version of previous histogram - strip activity = strip rate / average partition rate),
- `Strip_Homogeneity_Tt_Sc_p` shows the *homogeneity* of a given partition (homogeneity = exp(-strip rates standard deviation(strip rates in partition/average partition rate))),

- mask_Strip_Mean_Noise_Tt_Sc_p shows noise/gamma rate per unit area for each masked strip in a selected time range. Offline, the user can control the noise/gamma rate and decide to mask the strips that are judged to be noisy or dead. This is done via the *Masking Tool* provided by the webDCS,
- mask_Strip_Activity_Tt_Sc_p shows noise/gamma activity per unit area for each masked strip with repect to the average rate of active strips,
- NoiseCSize_H_Tt_Sc_p shows noise/gamma cluster size, a cluster being constructed out of adjacent strips giving a signal at the *same time* (hits within a time window of 25 ns),
- NoiseCMult_H_Tt_Sc_p shows noise/gamma cluster multiplicity (number of reconstructed clusters per event),
- Chip_Mean_Noise_Tt_Sc_p shows the same information than Strip_Mean_Noise_Tt_Scp using a different binning (1 chip corresponds to 8 strips),
- Chip_Activity_Tt_Sc_p shows the same information than Strip_Activity_Tt_Scp using chip binning,
- Chip_Homogeneity_Tt_Sc_p shows the homogeneity of a given partition using chip binning,
- Beam_Profile_Tt_Sc_p shows the estimated beam profile when taking efficiency scan. This is obtained by filtering Time_Profile_Tt_Sc_p to only consider the muon peak where the noise/gamma background has been subtracted. The resulting hit profile corresponds to the beam profile on the detector channels,
- L0_Efficiency_Tt_Sc_p shows the level 0 efficiency that was estimated **without** muon tracking,
- MuonCSize_H_Tt_Sc_p shows the level 0 muon cluster size that was estimated **without** muon tracking, and
- MuonCMult_H_Tt_Sc_p shows the level 0 muon cluster multiplicity that was estimated **without** muon tracking.

In the histogram labels, t stands for the trolley number (1 or 3), c for the chamber slot label in trolley t and p for the partition label (A, B, C or D depending on the chamber layout) as explained in Chapter **??**.

In the context of GIF++, an extra script called by the webDCS is called to extract the histograms from the ROOT files. The histograms are then stored in PNG and PDF formats into the corresponding folder (a single folder per HV step, so per ROOT file). the goal is to then display the histograms on the Data Quality Monitoring (DQM) page of the webDCS in order for the users to control the quality of the data taking at the end of data taking. An example of histogram organisation is given bellow:

```
Scan001000
├── Scan001000_HV1_DAQ.root
├── Scan001000_HV1_CAEN.root
├── Scan001000_HV1_Offline.root
├── HV1
│   ├── DAQ
│   │   ├── Beam_Profile_T1_S1_A.png
│   │   ├── Beam_Profile_T1_S1_A.pdf
│   │   ├── Beam_Profile_T1_S1_B.png
│   │   ├── Beam_Profile_T1_S1_B.pdf
│   │   └── ...
│   └── CAEN
│       ├── HVapp_Example_RPC1.pdf
│       ├── HVapp_Example_RPC1.png
│       ├── HVapp_Example_RPC1.pdf
│       ├── HVapp_Example_RPC1.png
│       └── ...
├── Scan001000_HV2_DAQ.root
├── Scan001000_HV2_CAEN.root
├── Scan001000_HV2_Offline.root
├── HV2
│   ├── DAQ
│   │   └── ...
│   └── CAEN
│       └── ...
├── Scan001000_HV3_DAQ.root
├── Scan001000_HV3_CAEN.root
├── Scan001000_HV3_Offline.root
├── HV3
│   └── ...
└── ...
```

Here can put some screens from the webDCS to show the DQM and the plots available to users.

1.2.1.2 CSV files

Moreover, up to 3 CSV files can be created depending on which ones of the 3 input files were in the data folder:

- `Offline-Corrupted.csv` , is used to keep track of the amount of data that was corrupted and removed from old data format files that don't contain any data quality flag.

- `Offline-Current.csv` , contains the summary of the currents and voltages applied on each RPC HV channel.

- `Offline-L0-EffCl.csv` , is used to write the efficiencies, cluster size and cluster multiplicity of efficiency runs. Note that `L0` refers here to *Level 0* and means that the results of effiency and clusterization are a first approximation calculated without performing any muon tracking in between the different detectors. This offline tool provides the user with a preliminar calculation of the efficiency and of the muon event parameters. Another analysis software especially dedicated to muon tracking is called on selected data to retrieve the results of efficiency and muon clusterization using a tracking algorithm to discriminate noise or gamma from muons as muons are the only particles that pass through the full setup, leaving hits than can be used to reconstruct their tracks.

- `Offline-Rate.csv` , is used to write the noise or gamma rates measured in the detector readout partitions.

Note that these 4 CSV files are created along with their *headers* (`Offline-[...]-Header.csv` containing the names of each data columns) and are automatically merged together when the offline analysis tool is called from the webDCS, contrary to the case where the tool is runned locally from the terminal as the merging bash script is then not called. Thus, the resulting files, used to make official plots, are:

- `Corrupted.csv` ,
- `Current.csv` ,
- `L0-EffCl.csv` .
- `Rate.csv` .

1.3 Analysis inputs and information handling

The usage of the Offline Analysis tool as well as its output have been presented in the previous section. It is now important to dig further and start looking at the source code and the inputs necessary for the tool to work. Indeed, other than the raw ROOT data files that are analysed, more information needs to be imported inside of the program to perform the analysis such as the description of the setup inside of GIF++ at the time of data taking (number of trolleys, of Resistive Plate Chamber (RPC)s, dimensions of the detectors, etc...) or the mapping that links the TDC channels to the coresponding RPC channels in order to translate the TDC information into human readable data. 2 files are used to transmit all this information:

- `Dimensions.ini`, that provides the necessary setup and RPC information, and
- `ChannelsMapping.csv`, that gives the link between the TDC and RPC channels as well as the *mask* for each channel (masked or not?).

1.3.1 Dimensions file and IniFile parser

This input file, present in every data folder, allows the analysis tool to know of the number of active trolleys, the number of active RPCs in those trolleys, and the details about each RPCs such as the number of RPC gaps, the number of pseudo-rapidity partitions (for prototypes similar to the RPCs used at the Compact Muon Solenoid (CMS)), the number of strips per partion or the dimensions. To do so, there are 3 types of groups in the INI file architecture. A first general group, appearing only once at the head of the document, gives information about the number of active trolleys as well as their IDs, as presented in Source Code 1.1. For each active trolley, a group similar to Source Code 1.2 can be found containing information about the number of active detectors in the trolley and their IDs. Each trolley group as a `Tt` name format, where `t` is the trolley ID. Finally, for each detector stored in slots of an active trolley, there is a group providing information about their names and dimensions, as shown in Source Code 1.3. Each slot group as a `TtSs` name format, where `s` is the slot ID of trolley `t` where the active RPC is hosted.

```
[General]
nTrolleys=2
TrolleysID=13
```

Source Code 1.1: Example of `[General]` *group as might be found in* `Dimensions.ini`*. In GIF++, only 2 trolleys are available to hold RPCs and place them inside of the bunker for irradiation. The IDs of the trolleys are written in a signle string as "13" and then read character by character by the program.*

```
[T1]
nSlots=4
SlotsID=1234
```

Source Code 1.2: Example of trolley group as might be found in `Dimensions.ini`*. In this example, the file tells that there are 4 detectors placed in the holding slots of the trolley* `T1` *and that their IDs, written as a single string variable, are 1, 2, 3 and 4.*

```
[T1S1]
Name=RE2-2-NPD-BARC-8
Partitions=3
Gaps=3
Gap1=BOT
Gap2=TN
Gap3=TW
AreaGap1=11694.25
AreaGap2=6432
AreaGap3=4582.82
Strips=32
ActiveArea-A=157.8
ActiveArea-B=121.69
ActiveArea-C=93.03
```

Source Code 1.3: Example of slot group as might be found in `Dimensions.ini`*. In this example, the file provides information about a detector named* `RE2-2-NPD-BARC-8`*, having 3 pseudo-rapidity readout partitions and stored in slot* `S1` *of trolley* `T1`*. This is a CMS RE2-2 type of detector. This information will then be used for example to compute the rate per unit area calculation.*

This information is readout and stored in a `C++` object called `IniFile`, presented in Source Code 1.4, that parses the information in the INI input file and stores it into a local buffer for later use. This INI parser is the exact same one that was previously developped for the GIF++ DAQ [2]. It contains private methods returning a boolean to check the type of line written in the file, whether a comment, a group header or a key line (`IniFile::CheckIfComment()`, `IniFile::CheckIfGroup()` and `IniFile::CheckIfToken()`). The key may sometimes be referred to as *token* in the source code. Moreover, the private element `FileData` is a map of **`const`** `string` to `string` that allows to store the data contained inside the configuration file via the public method `IniFile::GetFileData()` following the formatting (see method `IniFile::Read()`):

```
string group, token, value;
// Get the field values for the 3 strings.
// Then concatenate group and token together as a single string
// with a dot separation.
token = group + "." + token;
FileData[token] = value;
```

More methods have been written to translate the different keys into the right variable format when used by the Offline Analysis tool. For example, to get a **`float`** value out of the configuration file data, knowing the group and the key needed, the method `IniFile::floatType()` can be used. It takes 3 arguments being the group name and key name (both `string`), and a default **`float`** value used as exception in the case the expected combination of group and key cannot be found in the configuration file. This default value is then used and the DAQ continues on working after sending an alert in the log file for further debugging.

```
typedef map< const string, string > IniFileData;

class IniFile{
    private:
        bool        CheckIfComment(string line);
        bool        CheckIfGroup(string line,string& group);
        bool        CheckIfToken(string line,string& key,string& value);
        string      FileName;
        IniFileData FileData;
        int         Error;

    public:
        IniFile();
        IniFile(string filename);
        virtual    ~IniFile();

        // Basic file operations
        void        SetFileName(string filename);
        int         Read();
        int         Write();
        IniFileData GetFileData();

        // Data readout methods
        Data32      addressType (string groupname, string keyname, Data32
 ↪  defaultvalue);
        long        intType     (string groupname, string keyname, long
 ↪  defaultvalue);
        long long   longType    (string groupname, string keyname, long long
 ↪  defaultvalue );
        string      stringType  (string groupname, string keyname, string
 ↪  defaultvalue );
        float       floatType   (string groupname, string keyname, float
 ↪  defaultvalue );

        // Error methods
        string      GetErrorMsg();
};
```

Source Code 1.4: Description of `C++` *object* `IniFile` *used as a parser for INI file format.*

1.3.2 TDC to RPC link file and Mapping

The same way the INI dimension file information is stored using `map`, the channel mapping and mask information is stored and accessed through `map`. First of all, the mapping CSV file is organised into 3 columns separated by tabulations (and not by comas, as expected for CSV files as it is easier using streams to read tab or space separated data using C++):

```
RPC_channel         TDC_channel         mask
```

using as formatting for each field:

```
TSCCC         TCCC         M
```

`TSCCC` is a 5-digit integer where `T` is the trolley ID, `S` the slot ID in which the RPC is held insite the trolley `T` and `CCC` is the RPC channel number, or *strip* number, that can take values up to 3-digits depending on the detector,

`TCCC` is a 4 digit integer where `T` is the TDC ID, `CCC` is the TDC channel number that can take values in between 0 and 127, and

`M` is a 1-digit integer indicating if the channel should be considered (M = 1) or discarded (M = 0) during analysis.

This mapping and masking information is readout and stored thanks to the object `Mapping`, presented in Source Code 1.5. Similarly to `IniFile` objects, this class has private methods. The first one, `Mapping::CheckIfNewLine()` is used to find the newline character `'\n'` or return character `'\r'` (depending on which kind of operating system interacted with the file). This is used for the simple reason that the masking information has been introduced only during the year 2017 but the channel mapping files exist since 2015 and the very beginning of data taking at GIF++. This means that in the older data folders, before the upgrade, the channel mapping file only had 2 columns, the RPC channel and the TDC channel. For compatibility reasons, this method helps controling the character following the readout of the 2 first fields of a line. In case any end of line character is found, no mask information is present in the file and the default M = 1 is used. On the contrary, if the next character was a tabulation or a space, the mask information is present.

Once the 3 fields have been readout, the second private method `Mapping::CheckIfTDCCh()` is used to control that the TDC channel is an existing TDC channel. Finally, the information is stored into 3 different maps (`Link`, `ReverseLink` and `Mask`) thanks to the public method `Mapping::Read()`. `Link` allows to get the RPC channel by knowing the TDC channel while `ReverseLink` does the opposite by returning the TDC channel by knowing the RPC channel. Finally, `Mask` returns the mask associated to a given RPC channel.

```
typedef map<Uint,Uint> MappingData;

class Mapping {
    private:
        bool        CheckIfNewLine(char next);
        bool        CheckIfTDCCh(Uint channel);
        string      FileName;
        MappingData Link;
        MappingData ReverseLink;
        MappingData Mask;
        int         Error;

    public:
        Mapping();
        Mapping(string baseName);
        ~Mapping();

        void SetFileName(const string filename);
        int Read();
        Uint GetLink(Uint tdcchannel);
        Uint GetReverse(Uint rpcchannel);
        Uint GetMask(Uint rpcchannel);
};
```

Source Code 1.5: Description of `C++` *object* `Mapping` *used as a parser for the channel mapping and mask file.*

1.4 Description of GIF++ setup within the Offline Analysis tool

In the previous section, the tool input files have been discussed. The dimension file information is stored in a map hosted by the IniFile object. But this information is then used to create a series of new objects that helps defining the GIF++ infrastructure directly into the Offline Analysis. Indeed, from the `RPC`, to the more general `Infrastructure`, every element of the GIF++ infrastrucutre is recreated for each data analysis based on the information provided in input. All this information about the infrastructure will be used to assign each hit signal to a specific strip channel of a specific detector, and having a specific active area. This way, rate per unit area calculation is possible.

1.4.1 RPC objects

`RPC` objects have been developped to represent physical active detectors in GIF++ at the moment of data taking. Thus, there are as many RPC objects created during the analysis than there were active RPCs tested during a run. Each `RPC` hosts the information present in the corresponding INI slot group, as shown in 1.3, and organises it using a similar architecture. This can been seen from Source Code 1.6.

To make the object more compact, the lists of gap labels, of gap active areas and strip active areas are stored into `vector` dynamical containers. `RPC` objects are always contructed thanks to the dimension file information stored into the `IniFIle` and their ID, using the format `TtSs`. Using the RPC ID, the constructor calls the methods of `IniFile` to initialise the `RPC`. The other constructors are not used but exist in case of need. Finally, some getters have been written to access the different private parameters storing the detector information.

```
class RPC{
    private:
        string          name;        //RPC name as in webDCS database
        Uint            nGaps;       //Number of gaps in the RPC
        Uint            nPartitions; //Number of partitions in the RPC
        Uint            nStrips;     //Number of strips per partition
        vector<string>  gaps;        //List of gap labels (BOT, TOP, etc...)
        vector<float>   gapGeo;      //List of gap active areas
        vector<float>   stripGeo;    //List of strip active areas

    public:
        RPC();
        RPC(string ID, IniFile* geofile);
        RPC(const RPC& other);
        ~RPC();
        RPC& operator=(const RPC& other);

        string GetName();
        Uint   GetNGaps();
        Uint   GetNPartitions();
        Uint   GetNStrips();
        string GetGap(Uint g);
        float  GetGapGeo(Uint g);
        float  GetStripGeo(Uint p);
};
```

Source Code 1.6: Description of `C++` *objects* `RPC` *that describe each active detectors used during data taking.*

1.4.2 Trolley objects

`Trolley` objects have been developped to represent physical active trolleys in GIF++ at the moment of data taking. Thus, there are as many trolley objects created during the analysis than there were active trolleys hosting tested RPCs during a run. Each `Trolley` hosts the information present in the corresponding INI trolley group, as shown in 1.2, and organises it using a similar architecture. In addition to the information hosted in the INI file, these object have a dynamical container of `RPC` objects, representing the active detectors the active trolley was hosting at the time of data taking. This can been seen from Source Code 1.7.

`Trolley` objects are always contructed thanks to the dimension file information stored into the `IniFIle` and their ID, using the format `Tt`. Using the Trolley ID, the constructor calls the methods of `IniFile` to initialise the `Trolley`. Retrieving the information of the RPC IDs via `SlotsID`, a new `RPC` is constructed and added to the container `RPCs` for each character in the ID `string`. The other constructors are not used but exist in case of need. Finally, some getters have been written to access the different private parameters storing the trolley and detectors information.

```
class Trolley{
    private:
        Uint         nSlots;  //Number of active RPCs in the considered trolley
        string       SlotsID; //Active RPC IDs written into a string
        vector<RPC*> RPCs;    //List of active RPCs

    public:
        //Constructors, destructor and operator =
        Trolley();
        Trolley(string ID, IniFile* geofile);
        Trolley(const Trolley& other);
        ~Trolley();
        Trolley& operator=(const Trolley& other);

        //Get GIFTrolley members
        Uint   GetNSlots();
        string GetSlotsID();
        Uint   GetSlotID(Uint s);

        //Manage RPC list
        RPC*   GetRPC(Uint r);
        void   DeleteRPC(Uint r);

        //Methods to get members of RPC objects stored in RPCs
        string GetName(Uint r);
        Uint   GetNGaps(Uint r);
        Uint   GetNPartitions(Uint r);
        Uint   GetNStrips(Uint r);
        string GetGap(Uint r, Uint g);
        float  GetGapGeo(Uint r, Uint g);
        float  GetStripGeo(Uint r, Uint p);
};
```

Source Code 1.7: Description of C++ objects `Trolley` *that describe each active trolley used during data taking.*

1.4.3 Infrastructure object

The `Infrastructure` object has been developped to represent the GIF++ bunker area dedicated to CMS RPC experiments. With this very specific object, all the information about the CMS RPC setup within GIF++ at the moment of data taking is stored. It hosts the information present in the corresponding INI general group, as shown in 1.1, and organises it using a similar architecture. In addition to the information hosted in the INI file, this object have a dynamical container of `Trolley` objects, representing the active tolleys in GIF++ area. This can been seen from Source Code 1.8.

The `Infrastructure` object is always contructed thanks to the dimension file information stored into the `IniFIle`. Retrieving the information of the trolley IDs via `TrolleysID`, a new `Trolley` is constructed and added to the container `Trolleys` for each character in the ID `string`. By extension, it is easy to understand that the process described in Section 1.4.2 for the construction of RPCs takes place when a trolley is constructed. The other constructors are not used but exist in case of need. Finally, some getters have been written to access the different private parameters storing the infrastructure, trolleys and detectors information.

```
class Infrastructure {
    private:
        Uint             nTrolleys;  //Number of active Trolleys in the run
        string           TrolleysID; //Active trolley IDs written into a string
        vector<Trolley*> Trolleys;   //List of active Trolleys (struct)

    public:
        //Constructors and destructor
        Infrastructure();
        Infrastructure(IniFile* geofile);
        Infrastructure(const Infrastructure& other);
        ~Infrastructure();
        Infrastructure& operator=(const Infrastructure& other);

        //Get Infrastructure members
        Uint   GetNTrolleys();
        string GetTrolleysID();
        Uint   GetTrolleyID(Uint t);

        //Manage Trolleys
        Trolley* GetTrolley(Uint t);
        void     DeleteTrolley(Uint t);

        //Methods to get members of GIFTrolley objects stored in Trolleys
        Uint   GetNSlots(Uint t);
        string GetSlotsID(Uint t);
        Uint   GetSlotID(Uint t, Uint s);
        RPC*   GetRPC(Uint t, Uint r);

        //Methods to get members of RPC objects stored in RPCs
        string GetName(Uint t, Uint r);
        Uint   GetNGaps(Uint t, Uint r);
        Uint   GetNPartitions(Uint t, Uint r);
        Uint   GetNStrips(Uint t, Uint r);
        string GetGap(Uint t, Uint r, Uint g);
        float  GetGapGeo(Uint t, Uint r, Uint g);
        float  GetStripGeo(Uint t, Uint r, Uint p);
};
```

Source Code 1.8: Description of C++ object `Infrastructure` *that contains the full information about CMS RPC experiment in GIF++.*

1.5 Handeling of data

The raw data as a `TTree` architecture where every entry is related to a trigger signal provided by a muon or a random pulse, whether the goal of the data taking was to measure the performance of the detector or the noise/gamma background respectively. Each of these entries, referred also as events, contain a more or less full list of hits in the TDC channels to which the detectors are connected. To this list of hits corresponds a list of time stamps, marking the arrival of the hits within the TDC channel.

The infrastructure of the CMS RPC experiment within GIF++ being defined, combining the information about the raw data with the information provided by both the mapping/mask file and the dimension file allows to build new physical objects that will help in computing efficiency or rates.

1.5.1 RPC hits

The raw data stored in the ROOT file as output of the GIF++ DAQ, is readout by the analysis tool using the structure `RAWData` presented in Source Code 1.10. In this sense, this structure is in the case of the offline analysis tool not a dynamical object and will only be storing a single event contained in a single entry of the `TTree`.

```
class RPCHit {
    private:
        Uint  Channel;    //RPC channel according to mapping (5 digits)
        Uint  Trolley;    //0, 1 or 3 (1st digit of the RPC channel)
        Uint  Station;    //Slot where is held the RPC in Trolley (2nd digit)
        Uint  Strip;      //Physical RPC strip where the hit occured (last 3
 ↪  digits)
        Uint  Partition;  //Readout partition along eta segmentation
        float TimeStamp;  //Time stamp of the arrival in TDC

    public:
        //Constructors, destructor & operator =
        RPCHit();
        RPCHit(Uint channel, float time, Infrastructure* Infra);
        RPCHit(const RPCHit& other);
        ~RPCHit();
        RPCHit& operator=(const RPCHit& other);

        //Get RPCHit members
        Uint  GetChannel();
        Uint  GetTrolley();
        Uint  GetStation();
        Uint  GetStrip();
        Uint  GetPartition();
        float GetTime();
};

typedef vector<RPCHit> HitList;
typedef struct GIFHitList { HitList rpc[NTROLLEYS][NSLOTS][NPARTITIONS]; }
 ↪  GIFHitList;

bool SortHitbyStrip(RPCHit h1, RPCHit h2);
bool SortHitbyTime(RPCHit h1, RPCHit h2);
```

Source Code 1.9: Description of `C++` *object* `RPCHit`.

```
struct RAWData{
    int              iEvent;    //Event i
    int              TDCNHits;  //Number of hits in event i
    int              QFlag;     //Quality flag list (1 flag digit per TDC)
    vector<Uint>   *TDCCh;     //List of channels giving hits per event
    vector<float>  *TDCTS;     //List of the corresponding time stamps
};
```

Source Code 1.10: Description of `C++` *structure* `RAWData`.

Each member of the structure in then linked to the corresponding branch of the ROOT data tree, as shown in the example of Source Code 1.11, and using the method `GetEntry(int i)` of the ROOT class `TTree` will update the state of the members of `RAWData`.

```
TTree*  dataTree = (TTree*)dataFile.Get("RAWData");
RAWData data;

dataTree->SetBranchAddress("EventNumber",    &data.iEvent);
dataTree->SetBranchAddress("number_of_hits", &data.TDCNHits);
dataTree->SetBranchAddress("Quality_flag",   &data.QFlag);
dataTree->SetBranchAddress("TDC_channel",    &data.TDCCh);
dataTree->SetBranchAddress("TDC_TimeStamp",  &data.TDCTS);
```

Source Code 1.11: Example of link in between RAWData and TTree.

The data is then analysed entry by entry and to each element of the TDC channel list, a `RPCHit` is constructed by linking each TDC channel to the corresponding RPC channel thanks to the `Mapping` object. The information carried by the RPC channel format allows to easily retrieve the trolley and slot from which the hit was recorded (see section 1.3.2). Using these 2 values, the readout partition can be found by knowing the strip channel and comparing it with the number of partitions and strips per partition stored into the `Infrastructure` object.

Thus `RPCHit` objects are then stored into 3D dynamical list called `GIFHitList` (Source Code 1.10) where the 3 dimensions refer to the 3 layers of the readout in GIF++ : in the bunker there are *trolleys* (`T`) holding detectors in *slots* (`S`) and each detector readout is divided into 1 or more pseudo-rapidity *partitions* (`p`). Using these 3 information allows to assign an address to each readout partition and this address will point to a specific hit list.

1.5.2 Clusters of hits

All the hits contained in the ROOT file have been sorted into the different hit lists through the `GIFHitList`. At this point, it is possible to start looking for clusters. A cluster is a group of adjacent strips getting hits within a time window of 25 ns. These strips are then assumed to be part of the same physical avalanch signal generated by a muon passing through the chamber or by the interaction of a gamma stopping into the electrodes of the RPCs.

To keep the cluster information, `RPCCluster` objects have been defined as shown in Source Code 1.12. Using the information of each individual `RPCHit` taken out of the hit list, it stores the cluster size (number of adjacent strips composing the cluster), the first and last hit, the center for spatial reconstruction and finally the start and stop time stamps as well as te time spread in between the first and last hit.

```cpp
class RPCCluster{
    private:
        Uint  ClusterSize;  //Size of cluster #ID
        Uint  FirstStrip;   //First strip of cluster #ID
        Uint  LastStrip;    //Last strip of cluster #ID
        float Center;       //Center of cluster #ID ((first+last)/2)
        float StartStamp;   //Time stamp of the earliest hit of cluster #ID
        float StopStamp;    //Time stamp of the latest hit of cluster #ID
        float TimeSpread;   //Time difference between earliest and latest hits
                            //of cluster #ID
    public:
        //Constructors, destructor & operator =
        RPCCluster();
        RPCCluster(HitList List, Uint cID, Uint cSize, Uint first, Uint firstID);
        RPCCluster(const RPCCluster& other);
        ~RPCCluster();
        RPCCluster& operator=(const RPCCluster& other);

        //Get Cluster members
        Uint GetID();
        Uint GetSize();
        Uint GetFirstStrip();
        Uint GetLastStrip();
        float GetCenter();
        float GetStart();
        float GetStop();
        float GetSpread();
};

typedef vector<RPCCluster> ClusterList;

//Other functions to build cluster lists out of hit lists
void BuildClusters(HitList &cluster, ClusterList &clusterList);
void Clusterization(HitList &hits, TH1 *hcSize, TH1 *hcMult);
```

Source Code 1.12: Description of C++ *object* `Cluster`*.*

To investigate the hit list of a given detector partition, the function `Clusterization()` definied in `include/Cluster.h` needs the hits in the list to be time sorted. This is achieved by calling function `sort()` of library `<algorithm>` using the comparator `SortHitbyTime(RPCHit h1, RPCHit h2)` defined in `include/RPCHit.h` that returns `true` if the time stamp of hit `h1` is lower than that of `h2`. A first isolation of strips is made only based on time information. All the hits within the 25 ns window are taken separately from the rest. Then, this sub-list of hits is sorted this time by ascending strip number, using this time the comparator `SortHitbyStrip(RPCHit h1, RPCHit h2)`. Finally, the groups of adjacent strips are used to construct `RPCCluster` objects that are then stored in a temporary list of clusters that is at the end of the process used to know how many clusters were reconstructed and to fill their sizes into an histogram that will allows to know the mean size of muon or gamma clusters.

1.6 DAQ data Analysis

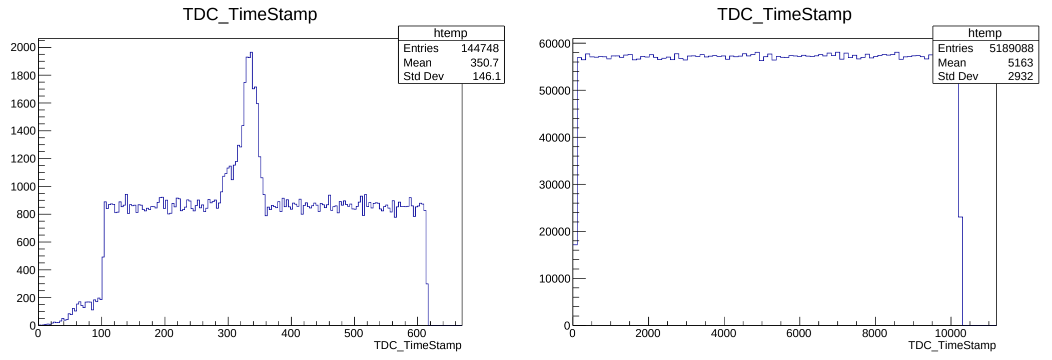
All the ingredients to analyse GIF++ data have been defined. This section will focus on the different part of the analysis performed on the data, from determining the type of data the tool is dealing with to calculating the rate in each detector or reconstructing muon or gamma clusters.

1.6.1 Determination of the run type

In GIF++, both the performance of the detectors in detecting muons in an irradiated environment and the gamma background can be independantly measured. These corresponds to different run types and thus, to different TDC settings giving different data to look at.

In the case of performance measurements, the trigger for data taking is provided by the coïncidence of several scintillators when muons from the beam passing through the area are detected. Data is collected in a 600 ns wide window around the arrival of muons in the RPCs. The expected time distribution of hits is shown in Figure 1.1a. The muon peak is clearly visible in the center of the distribution and is to be extracted from the gamma background that composes the flat part of the distribution.

On the other hand, gamma background or noise measurements are focussed on the non muon related physics and the trigger needs to be independant from the muons to give a good measurement of the gamma/noise distribution as seen by the detectors. The trigger is then provided by a pulse generator at a frequency of 300 Hz whose pulse is not likely to be on time with a muon. In order to increase the integrated time without increasing the acquisition time too much, the width of the acquisition windows are increased to 10 µs. The time distribution of the hits is expected to be flat, as shown by Figure 1.1b.



(a) *(b)*

Figure 1.1: Example of expected hit time distributions in the cases of efficiency (Figure 1.1a) and noise/gamma rate per unit area (Figure 1.1b) measurements as extracted from the raw ROOT files. The unit along the x-axis corresponds to ns*. The fact that "the" muon peak is not well defined in Figure 1.1a is due to the contribution of all the RPCs being tested at the same time that don't necessarily have the same signal arrival time. Each individual peak can have an offset with the ones of other detectors. The inconsistancy in the first* 100 ns *of both time distributions is an artefact of the TDCs and are systematically rejected during the analysis.*

The ROOT files include a `TTree` called `RunParameters` containing, among other things, the information related to the type of run. The run type can then be accessed as described by Source Code 1.13 and the function `IsEfficiencyRun()` is then used to determine if the run file is an efficiency run or, on the contrary, another type of run (noise or gamma measurement).

```
TTree* RunParameters = (TTree*)dataFile.Get("RunParameters");
TString* RunType = new TString();
RunParameters->SetBranchAddress("RunType",&RunType);
RunParameters->GetEntry(0);
```

Source Code 1.13: Access to the run type contained in `TTree* RunParameters`*.*

Finally, the data files will have a slightly different content whether it was collected before or after October 2017 and the upgrade of the DAQ software that brought a new information into the ROOT output. This implies that the analysis will differ a little depending on the data format. Indeed, as no information on the data quality is stored, in older data files, the corrections for missing events has to be done at the end of the analysis. The information about the type of data format is stored in the variable `bool isNewFormat` by checking the list of branches contained in the data tree via the methods `TTree::GetListOfBranches()` and `TCollection::Contains()`.

1.6.2 Beam time window calculation for efficiency runs

Knowing the run type is important first of all to know the width of the acquisition window to be used for the rate calculation and finally to be able to seek for muons. Indeed, the peak that appears in the time distribution for each detectors is then fitted to extract the most probable time window in which the tool should look for muon hits. The data outside of this time window in then used to evaluate the noise or gamma background the detector was subbjected to during the data taking. Computing the position of the peak is done calling the function `SetBeamWindow()` defined in file `src/RPCHit.cc` that loops a first time on the data. The data is first sorted in a 3D array of 1D histograms (`GIFH1Array`, see `include/types.h`). Then the location of the highest bin is determined using `TH1::GetMaximumBin()` and is used to define a window in which a gaussian fit will be applied to compute the peak width. This window is a 80 ns defined by Formula 1.1 around the central bin.

$$\begin{aligned} t_{center}(ns) &= bin \times width_{bin}(ns) \qquad (1.1a) \\ [t_{low}; t_{high}] &= [t_{center} - 40; t_{center} + 40] \qquad (1.1b) \end{aligned}$$

Before the fit is performed, the average number of noise/gamma hits per bin is evaluated using the data outside of the fit window. Excluding the first 100 ns, the average number of hits per bin due to the noise or gamma is defined by Formula 1.2 after extracting the amount of hits in the time windows $[100; t_{low}]$ and $[t_{high}; 600]$ thanks to the method `TH1::Integral()`. This average number of hits is then subtracted to every bin of the 1D histogram, in order to *clean* it from the noise or gamma contribution as much as possible to improve the fit quality. Bins where $\langle n_{hits} \rangle$ is greater than the actual bin content are set to 0.

$$\begin{aligned} \Delta t_{noise}(ns) &= 600 \overbrace{-t_{high} + t_{low}}^{-80ns} -100 = 420ns \qquad (1.2a) \\ \langle n_{hits} \rangle &= width_{bin}(ns) \times \frac{\sum_{t=100}^{t_{low}} + \sum_{t=t_{high}}^{600}}{\Delta t_{noise}(ns)} \qquad (1.2b) \end{aligned}$$

Finally, the fit parameters are extracted and saved for each detector in 3D arrays of `float` (`muonPeak`, see `include/types.h`), a first one for the mean arrival time of the muons, `PeakTime`, and a second one for the width of the peak, `PeakWidth`. The width is defined as 6σ of the gaussian fit. The same settings are applied to every partitions of the same detector. To determine which one of the detector's partitions is directly illuminated by the beam, the peak height of each partition is compared and the highest one is then used to define the peak settings.

1.6.3 Data loop and histogram filling

3D arrays of histogram are created to store the data and display it on the DQM of GIF++ webDCS for the use of shifters. These histograms, presented in section 1.2.1.1, are filled while looping on the data. Before starting the analysis loop, it is necessary to control the entry quality for the new file formats featuring `QFlag`. If the `QFlag` value for this entry shows that 1 TDC or more have a `CORRUPTED` flag, then this event is discarded. The loss of statistics is low enough to be neglected. `QFlag` is controled using the function `IsCorruptedEvent()` defined in `src/utils.cc`. Each digit of this integer represent a TDC flag that can be 1 or 2. Each 2 is the sign of a `CORRUPTED` state. Then, the data is accessed entry by entry in the ROOT `TTree` using `RAWData` and each hit in the hit list is assigned to a detector channel and saved in the corresponding histograms. In the first part of the analysis, in which the loop over the ROOT file's content is performed, the different steps are:

1- RPC channel assignment and control: a check is done on the RPC channel extracted thanks to the mapping via the method `Mapping::GetLink()`. If the channel is not initialised and is 0, or if the TDC channel was greater than 5127, the hit is discarded. This means there was a problem in the mapping. Often a mapping problem leads to the crash of the offline tool.

2- Creation of a `RPCHit` object: to easily get the trolley, slot and partition in which the hit has been assigned, this object is particularly helpful.

3- General histograms are filled: the hit is filled into the time distribution and the general hit distribution histograms, and if the arrival time is within the first 100 ns, it is discarded and nothing else happens and the loop proceeds with the next hit in the list.

4- Multiplicity counter: the hit multiplicity counter of the corresponding detectors incremented.

5-a- *Effiency runs* - Is the hit within the peak window? : if the peak is contained in the peak window previously defined in section 1.6.2, the hit is filled into the beam hit profile histogram of the corresponding chamber, added into the list of muon hits and increments the counter of *in time* hits. The term *in time* here refers to the hits that are likely to be muons by arriving in the expected time window. If the hit is outside of the peak window, it is filled into the noise profile histogram of the corresponding detector, added into the list of noise/gamma hits and increments the counter of noise/gamma hits.

5-b- *Noise/gamma rate runs* - Noise histograms are filled: the hit is filled into the noise profile histogram of the corresponding detector, added into the list of noise/gamma hits and increments the counter of noise/gamma hits.

After the loop on the hit list of the entry is over, the next step is too clusterize the 3D lists filled in the previous steps. A 3D loop is then started over the active trolley, slot and RPC partitions to access these objects. Each `NoiseHitList` and `MuonHitList`, in case of efficiency run, are clusterized as decribed in section 1.5.2. There corresponding cluster size and multiplicity histograms are filled at the end of the clustering process. Then, the effiency histogram is filled in case of effieincy run. The selection is simply made by checking whether the RPC detected signals in the peak window during this event. Nevertheless, it is useful to highlight that at this level, it is not possible yet to discriminate in between a muon hit and noise or gamma hit. Thus, `MuonCSize_H`, `MuonCMult_H` and `Efficiency0_H` are subjected to noise and gamma contamination. This contamination will be estimated and corrected at the moment the results will be written into output CSV files. Finally, the loop ends on the filling of the general hit multiplicity histogram.

1.6.4 Results calculation

As mentioned in section 1.2.1, the analysis of DAQ data provides the user with 3 CSV files and a ROOT file associated to each and every ROOT data file. The fourth CSV file is provided by the extraction of the CEAN main frame data monitored during data tacking and will be discussed later. After looping on the data in the previous part of the analysis macro, the output files are created and a 3D loop on each RPC readout partitions is started to extract the histograms parameters and compute the final results.

1.6.4.1 Rate normalisation

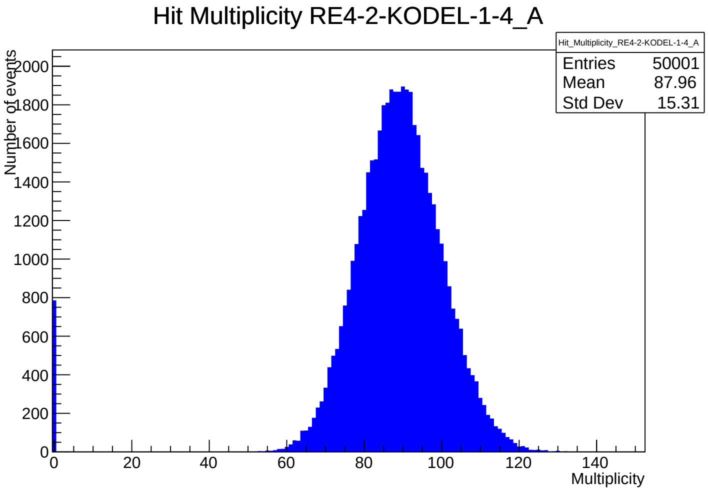


Figure 1.2: The effect of the quality flag is explained by presenting the reconstructed hit multiplicity of a data file without `Quality_flag`. The artificial high content of bin 0 is the effect of corrupted data.

To analyse old data format files, not containing any quality flag, it is needed to estimate the amount of corrupted data via a fit as the corrupted data will always fill events with a fake ”0 multiplicity”. Indeed, as no hits were stored in the DAQ ROOT files, these events artificially contribute to fill the bin corresponding to a null multiplicity, as shown in Figure 1.2. In the case the mean of the hit multiplicity distribution is high, the contribution of the corrupted data can easily be evaluated for later correction by comparing the level of the bin at multiplicity 0 and of a skew fit curve that should indicate a value consistent with 0. A skew fit has been chosen over a Poisson fit as it was giving better results for lower mean multiplicity values. Nevertheless, for low irradiation cases, the hit multiplicity distribution mean is, on the contrary, rather small and the probability to record events without hits can't be considered small anymore, leading to a difficult and non-reliable estimation of the corruption. As can be seen in Source Code 1.14, conditions have been applied to prevent bad fits and wrong corruption estimation in cases where :

- The difference in between the data for multiplicity 1 and the corresponding fit value should be lower than 1% of the total amount of data : $\frac{|n_{m=1} - sk(1)|}{N_{tot}} < 0.01$ where $n_{m=1}$ is the number of entries with multiplicity 1, $sk(1)$ the value of the skew fit, as defined by Formula **??**, for multiplicity 1 and N_{tot} the total number of entries.

- The amount of data contained in the multiplicity 0 bin should not exceed 40% : $\frac{n_{m=0}}{N_{tot}} \leq 0.4$ where $n_{m=0}$ is the number of entries with multiplicity 0. This number has been determined to be the maximum to be able to separate the excess of data due to corruption from the hit multiplicity distribution.

Those 2 conditions need to be fulfilled to estimate the corruption of old data format files. If the fit was successful, the level of corruption is written in `Offline-Corrupted.csv` and the number of corrupted entries, refered as the integer `nEmptyEvent`, is subtracted from the total number of entries when the rate normalisation factor is computed as explicited in Source Code 1.14. Note that for new data format files, the number of corrupted entries being set to 0, the definition of `rate_norm` stays valid.

```
if(!isNewFormat){
    TF1* GaussFit = new TF1("gaussfit","[0]*exp(-0.5*((x-[1])/[2])**2)",0,Xmax);
    GaussFit->SetParameter(0,100);
    GaussFit->SetParameter(1,10);
    GaussFit->SetParameter(2,1);
    HitMultiplicity_H.rpc[T][S][p]->Fit(GaussFit,"LIQR","",0.5,Xmax);

    TF1* SkewFit = new TF1("skewfit","[0]*exp(-0.5*((x-[1])/[2])**2) / (1 + exp(-[3]*(x-[4])))",0,Xmax);
    SkewFit->SetParameter(0,GaussFit->GetParameter(0));
    SkewFit->SetParameter(1,GaussFit->GetParameter(1));
    SkewFit->SetParameter(2,GaussFit->GetParameter(2));
    SkewFit->SetParameter(3,1);
    SkewFit->SetParameter(4,1);
    HitMultiplicity_H.rpc[T][S][p]->Fit(SkewFit,"LIQR","",0.5,Xmax);

    double fitValue = SkewFit->Eval(1,0,0,0);
    double dataValue = (double)HitMultiplicity_H.rpc[T][S][p]->GetBinContent(2);
    double difference = TMath::Abs(dataValue - fitValue);
    double fitTOdataVSentries_ratio = difference / (double)nEntries;
    bool isFitGOOD = fitTOdataVSentries_ratio < 0.01;

    double nSinglehit = (double)HitMultiplicity_H.rpc[T][S][p]->GetBinContent(1);
    double lowMultRatio = nSinglehit / (double)nEntries;
    bool isMultLOW = lowMultRatio > 0.4;

    if(isFitGOOD && !isMultLOW){
        nEmptyEvent = HitMultiplicity_H.rpc[T][S][p]->GetBinContent(1);
        nPhysics = (int)SkewFit->Eval(0,0,0,0);
        if(nPhysics < nEmptyEvent)
        nEmptyEvent = nEmptyEvent-nPhysics;
    }
}

double corrupt_ratio = 100.*(double)nEmptyEvent / (double)nEntries;
outputCorrCSV << corrupt_ratio << '\t';

float rate_norm = 0.;
float stripArea = GIFInfra->GetStripGeo(tr,sl,p);

if(IsEfficiencyRun(RunType)){
    float noiseWindow = BMTDCWINDOW - TIMEREJECT - 2*PeakWidth.rpc[T][S][p];
    rate_norm = (nEntries-nEmptyEvent)*noiseWindow*1e-9*stripArea;
} else
    rate_norm = (nEntries-nEmptyEvent)*RDMNOISEWDW*1e-9*stripArea;
```

Source Code 1.14: Definition of the rate normalisation variable. It takes into account the number of non corrupted entries and the time window used for noise calculation, to estimate the total integrated time, and the strip active area to express the result as rate per unit area.

1.6.4.2 Rate and activity

At this point, the strip rate histograms, `StripNoiseProfile_H.rpc[T][S][p]`, only contain an information about the total number of noise or rate hits each channel received during the data taking. As described in Source Code 1.15, a loop on the strip channels will be used to normalise the content of the rate distribution histogram for each detector partitions. The initial number of hits recorded for a given bin will be extracted and 2 values will be computed:

- the strip rate, defined as the number of hits recorded in the bin normalised like described in the previous section, using the variable `rate_norm`, and

- the strip activity, defined as the number of hits recorded in the bin normalised to the average number of hits per bin contained in the partition histogram, using the variable `averageNhit`. This value provides an information on the homogeneity of the detector response to the gamma background or of the detector noise. An activity of 1 corresponds to an average response. Above 1, the channel is more active than the average and bellow 1, the channel is less active.

```
int nNoise = StripNoiseProfile_H.rpc[T][S][p]->GetEntries();
float averageNhit = (nNoise>0) ? (float)(nNoise/nStripsPart) : 1.;

for(Uint st = 1; st <= nStripsPart; st++){
    float stripRate =
        StripNoiseProfile_H.rpc[T][S][p]->GetBinContent(st)/rate_norm;
    float stripAct =
        StripNoiseProfile_H.rpc[T][S][p]->GetBinContent(st)/averageNhit;

    StripNoiseProfile_H.rpc[T][S][p]->SetBinContent(st,stripRate);
    StripActivity_H.rpc[T][S][p]->SetBinContent(st,stripAct);
}
```

Source Code 1.15: Description of the loop that allows to set the content of each strip rate and strip activity channel for each detector partition.

On each detector partitions, which are readout by a single Front-End Electronics (FEE), all the channels are not processed by the same chip. Each chip can give a different noise response and thus, histograms using a chip binning are used to investigate chip related noise behaviours. The average values of the strip rate or activity grouped into a given chip are extracted using the using the function `GetChipBin()` and stored in dedicated histograms as described in Source Codes 1.16 and 1.17 respectively.

```
float GetChipBin(TH1* H, Uint chip){
    Uint start = 1 + chip*NSTRIPSCHIP;
    int nActive = NSTRIPSCHIP;
    float mean = 0.;

    for(Uint b = start; b <= (chip+1)*NSTRIPSCHIP; b++){
        float value = H->GetBinContent(b);
        mean += value;
        if(value == 0.) nActive--;
    }

    if(nActive != 0) mean /= (float)nActive;
    else mean = 0.;

    return mean;
}
```

Source Code 1.16: Function used to compute the content of a bin for an histogram using chip binning.

```
for(Uint ch = 0; ch < (nStripsPart/NSTRIPSCHIP); ch++){
    ChipMeanNoiseProf_H.rpc[T][S][p]->
        SetBinContent(ch+1,GetChipBin(StripNoiseProfile_H.rpc[T][S][p],ch));
    ChipActivity_H.rpc[T][S][p]->
        SetBinContent(ch+1,GetChipBin(StripActivity_H.rpc[T][S][p],ch));
}
```

Source Code 1.17: Description of the loop that allows to set the content of each chip rate and chip activity bins for each detector partition knowing the information contained in the corresponding strip distribution histograms.

The activity variable is used to evaluate the homogeneity of the detector response to background or of the detector noise. The homogeneity h_p of each detector partition can be evaluated using the formula $h_p = exp(-\sigma_p^R/\langle R\rangle_p)$, where $\langle R\rangle_p$ is the partition mean rate and σ_p^R is the rate standard deviation calculated over the partition channels. The more homogeneously the rates are distributed and the smaller will σ_p^R be, and the closer to 1 will h_p get. On the contrary, if the standard deviation of the channel's rates is large, h_p will rapidly get to 0. This value is saved into histograms as shown in Source Code 1.18 and could in the future be used to monitor through time, once extracted, the evolution of every partition homogeneity. This could be of great help to understand the apparition of eventual hot spots due to ageing of the chambers subjected to high radiation levels. The monitored homogeneity information could then be combined with a monitoring of the activity of each individual channel in order to have a finer information. Monitoring tools have been suggested and need to be developped for this purpose.

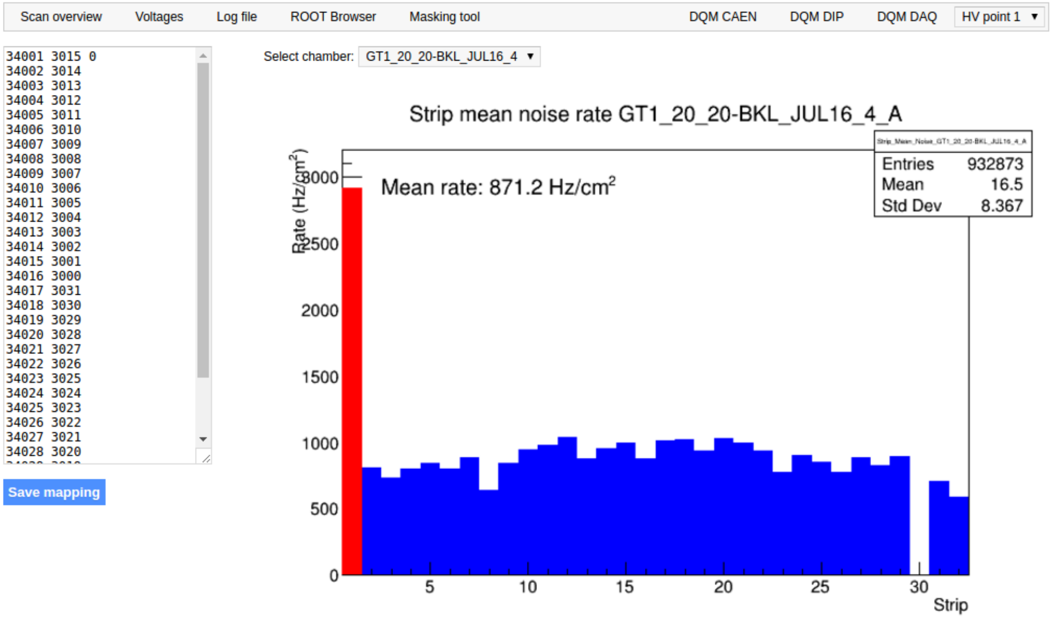
```
float MeanPartSDev = GetTH1StdDev(StripNoiseProfile_H.rpc[T][S][p]);
float strip_homog = (MeanPartRate==0)
    ? 0.
    : exp(-MeanPartSDev/MeanPartRate);
StripHomogeneity_H.rpc[T][S][p]->Fill("exp -#left(#frac{#sigma_{Strip
 ↪  Rate}}{#mu_{Strip Rate}}#right)",strip_homog);
StripHomogeneity_H.rpc[T][S][p]->GetYaxis()->SetRangeUser(0.,1.);

float ChipStDevMean = GetTH1StdDev(ChipMeanNoiseProf_H.rpc[T][S][p]);
float chip_homog = (MeanPartRate==0)
    ? 0.
    : exp(-ChipStDevMean/MeanPartRate);
ChipHomogeneity_H.rpc[T][S][p]->Fill("exp -#left(#frac{#sigma_{Chip
 ↪  Rate}}{#mu_{Chip Rate}}#right)",chip_homog);
ChipHomogeneity_H.rpc[T][S][p]->GetYaxis()->SetRangeUser(0.,1.);
```

Source Code 1.18: Storage of the homogeneity into dedicated histograms.

1.6.4.3 Strip masking tool

The offline tool is automatically called at the end of each data taking to analyse the data and offer the shifter DQM histograms to control the data quality. After the histograms have been published online in the DQM page, the shifter can decide to mask noisy or dead channels that will contribute to bias the final rate calculation by editing the mask column of `ChannelsMapping.csv` as can be seen in Figure 1.3.



```
34001 3015 0
34002 3014
34003 3013
34004 3012
34005 3011
34006 3010
34007 3009
34008 3008
34009 3007
34010 3006
34011 3005
34012 3004
34013 3003
34014 3002
34015 3001
34016 3000
34017 3031
34018 3030
34019 3029
34020 3028
34021 3027
34022 3026
34023 3025
34024 3024
34025 3023
34026 3022
34027 3021
34028 3020
```

Select chamber: GT1_20_20-BKL_JUL16_4 ▼

Figure 1.3: Display of the maskting tool page on the webDCS. The window on the left allows the shifter to edit `ChannelsMapping.csv`*. To mask a channel, it only is needed to set the 3rd field corresponding to the strip to mask to 0. It is not necessary for older mapping file formats to add a 1 for each strip that is not masked as the code is versatile and the default behaviour is to consider missing mask fields as active strips. The effect of the mask is directly visible for noisy channels as the corresponding bin turns red. The global effect of masking strips will be an update of the rate value showed on the histogram that will take into consideration the rejected channels.*

From the code point of view, the function `GetTH1Mean()` is used to retrieve the mean rate partition by partition after the rates have been calculated strip by strip and filled into the histograms `StripNoiseProfile_H.rpc[T][S][p]`, as described through Source Code 1.19.

Once the mask for each rejected channel has been updated, the shifter can manually run the offline tool again to update the DQM plots, now including the masked strips, as well the rate results written in the output CSV file `Offline-Rate.csv`. If not done during the shifts, the strip masking procedure needs to be carefully done by the person in charge of data analysis on the scans that were selected to produce the final results.

```
float GetTH1Mean(TH1* H){
    int nBins = H->GetNbinsX();
    int nActive = nBins;
    float mean = 0.;

    for(int b = 1; b <= nBins; b++){
        float value = H->GetBinContent(b);
        mean += value;
        if(value == 0.) nActive--;
    }

    if(nActive != 0) mean /= (float)nActive;
    else mean = 0.;

    return mean;
}
```

Source Code 1.19: The function `GetTH1Mean()` is used to return the mean along the y-axis of `TH1` histograms containing rate information. In order to take into account masked strips whose rate is set to 0, the function looks for masked channels and decrement the number of active channels for each null value found.

1.6.4.4 Output CSV files filling

All the histograms have been filled. Parameters will then be extracted from them to compute the final results that will later be used to produce plots. Once the results have been computed, the very last step of the offline macro is to write these values into the corresponding CSV outputs. Aside of the file `Offline-Corrupted.csv`, 2 CSV files are being written by the macro `OfflineAnalysis()`, `Offline-Rates.csv` and `Offline-L0-EffCl.csv` that respectively contain information about noise or gamma rates, cluster size and multiplicity, and about level 0 reconstruction of the detector efficiency, muon cluster size and multiplicity. Details on the computation and file writing are respectively given in Sources Codes 1.20 and 1.21.

Noise/gamma background variables are computed and written in the output file for each detector partitions. A detector average of the hit and cluster rate is also provided, as shown through Sources Code 1.20. The variables that are written for each partition are:

- The mean partition hit rate per unit area, `MeanPartRate`, that is extracted from the histogram `StripNoiseProfile_H` as the mean value along the y-axis, as described in section 1.6.4.3. No error is recorded for the hit rate as this is considered a single measurement. No statistical error can be associated to it and the systematics are unknown.

- The mean cluster size, `cSizePart`, is extracted from the histogram `NoiseCSize_H` and it's statistical error, `cSizePartErr`, is taken to be 2σ of the total distribution.

- The mean cluster multiplicity per trigger, `cMultPart`, is extracted from the histogram `NoiseCMult_H` and it's statistical error, `cMultPartErr`, is taken to be 2σ of the total distribution. It is important to point to the fact that this variable gives an information that is dependent on the buffer window width used for each trigger for the calculation.

- The mean cluster rate per unit area, `ClustPartRate`, is defined as the mean hit rate normalised to the mean cluster size and it’s statistical error, `ClustPartRateErr`, is then obtained using the relative statistical error on the mean cluster size.

```
for (Uint tr = 0; tr < GIFInfra->GetNTrolleys(); tr++){
    Uint T = GIFInfra->GetTrolleyID(tr);

    for (Uint sl = 0; sl < GIFInfra->GetNSlots(tr); sl++){
        Uint S = GIFInfra->GetSlotID(tr,sl) - 1;

        float MeanNoiseRate = 0.;
        float ClusterRate   = 0.;
        float ClusterSDev   = 0.;

        for (Uint p = 0; p < GIFInfra->GetNPartitions(tr,sl); p++){
            float MeanPartRate = GetTH1Mean(StripNoiseProfile_H.rpc[T][S][p]);
            float cSizePart = NoiseCSize_H.rpc[T][S][p]->GetMean();
            float cSizePartErr = (NoiseCSize_H.rpc[T][S][p]->GetEntries()==0)
                ? 0.
                : 2*NoiseCSize_H.rpc[T][S][p]->GetStdDev() /
                            sqrt(NoiseCSize_H.rpc[T][S][p]->GetEntries());
            float cMultPart = NoiseCMult_H.rpc[T][S][p]->GetMean();
            float cMultPartErr = (NoiseCMult_H.rpc[T][S][p]->GetEntries()==0)
                ? 0.
                : 2*NoiseCMult_H.rpc[T][S][p]->GetStdDev() /
                            sqrt(NoiseCMult_H.rpc[T][S][p]->GetEntries());
            float ClustPartRate = (cSizePart==0) ? 0.
                : MeanPartRate/cSizePart;
            float ClustPartRateErr = (cSizePart==0) ? 0.
                : ClustPartRate * cSizePartErr/cSizePart;

            outputRateCSV << MeanPartRate << '\t'
                << cSizePart << '\t' << cSizePartErr << '\t'
                << cMultPart << '\t' << cMultPartErr << '\t'
                << ClustPartRate << '\t' << ClustPartRateErr << '\t';

            RPCarea       += stripArea * nStripsPart;
            MeanNoiseRate += MeanPartRate * stripArea * nStripsPart;
            ClusterRate   += ClustPartRate * stripArea * nStripsPart;
            ClusterSDev   += (cSizePart==0)
                ? 0.
                : ClusterRate*cSizePartErr/cSizePart;
        }

        MeanNoiseRate /= RPCarea;
        ClusterRate   /= RPCarea;
        ClusterSDev   /= RPCarea;

        outputRateCSV << MeanNoiseRate << '\t'
            << ClusterRate << '\t' << ClusterSDev << '\t';
    }
}
```

Source Code 1.20: Description of rate result calculation and writing into the CSV output `Offline-Rate.csv`*. Are saved into the file for each detector, the mean partition rate, cluster size and cluster mutiplicity, along with their errors, for each partition and as well as a detector average.*

```
for (Uint tr = 0; tr < GIFInfra->GetNTrolleys(); tr++){
    Uint T = GIFInfra->GetTrolleyID(tr);
    for (Uint sl = 0; sl < GIFInfra->GetNSlots(tr); sl++){
        Uint S = GIFInfra->GetSlotID(tr,sl) - 1;
        for (Uint p = 0; p < GIFInfra->GetNPartitions(tr,sl); p++){
            float noiseWindow =
                BMTDCWINDOW - TIMEREJECT - 2*PeakWidth.rpc[T][S][p];
            float peakWindow = 2*PeakWidth.rpc[T][S][p];
            float windowRatio = peakWindow/noiseWindow;

            float PeakCM = MuonCMult_H.rpc[T][S][p]->GetMean();
            float PeakCS = MuonCSize_H.rpc[T][S][p]->GetMean();
            float NoiseCM = NoiseCMult_H.rpc[T][S][p]->GetMean()*windowRatio;
            float NoiseCS = NoiseCSize_H.rpc[T][S][p]->GetMean();
            float MuonCM = (PeakCM<NoiseCM) ? 0. : PeakCM-NoiseCM;
            float MuonCS = (MuonCM==0 || PeakCM*PeakCS<NoiseCM*NoiseCS)
                ? 0.
                : (PeakCM*PeakCS-NoiseCM*NoiseCS)/MuonCM;
            float PeakCM_err = (MuonCMult_H.rpc[T][S][p]->GetEntries()==0.)
                ? 0.
                : 2*MuonCMult_H.rpc[T][S][p]->GetStdDev() /
                            sqrt(MuonCMult_H.rpc[T][S][p]->GetEntries());
            float PeakCS_err = (MuonCSize_H.rpc[T][S][p]->GetEntries()==0.)
                ? 0.
                : 2*MuonCSize_H.rpc[T][S][p]->GetStdDev() /
                            sqrt(MuonCSize_H.rpc[T][S][p]->GetEntries());
            float NoiseCM_err = (NoiseCMult_H.rpc[T][S][p]->GetEntries()==0.)
                ? 0.
                : windowRatio*2*NoiseCMult_H.rpc[T][S][p]->GetStdDev() /
                            sqrt(NoiseCMult_H.rpc[T][S][p]->GetEntries());
            float NoiseCS_err = (NoiseCSize_H.rpc[T][S][p]->GetEntries()==0.)
                ? 0.
                : 2*NoiseCSize_H.rpc[T][S][p]->GetStdDev() /
                            sqrt(NoiseCSize_H.rpc[T][S][p]->GetEntries());
            float MuonCM_err = (MuonCM==0) ? 0. : PeakCM_err+NoiseCM_err;
            float MuonCS_err = (MuonCS==0 || MuonCM==0) ? 0.
                : (PeakCS*PeakCM_err + PeakCM*PeakCS_err +
                  NoiseCS*NoiseCM_err + NoiseCM*NoiseCS_err +
                  MuonCS*MuonCM_err)/MuonCM;

            float DataRatio = MuonCM/PeakCM;
            float DataRatio_err = (MuonCM==0) ? 0.
                : DataRatio*(MuonCM_err/MuonCM + PeakCM_err/PeakCM);
            float eff = DataRatio*Efficiency0_H.rpc[T][S][p]->GetMean();
            float eff_err =  DataRatio*2*Efficiency0_H.rpc[T][S][p]->GetStdDev()/
                        sqrt(Efficiency0_H.rpc[T][S][p]->GetEntries()) +
                        Efficiency0_H.rpc[T][S][p]->GetMean()*DataRatio_err;

            outputEffCSV << eff << '\t' << eff_err << '\t'
                << MuonCS << '\t' << MuonCS_err << '\t'
                << MuonCM << '\t' << MuonCM_err << '\t';
        }
    }
}
```

Source Code 1.21: Description of efficiency result calculation and writing into the CSV output `Offline-L0-EffCl.csv`*. Are saved into the file for each detector, the efficiency, corrected taking into account the background in the peak window of the time profile, muon cluster size and muon cluster mutiplicity, along with their errors, for each partition and as well as a detector average.*

Muon performance variables are computed and written in the output file for each detector partitions as shown through Sources Code 1.21. The variables that are written for each partition are:

- The muon efficiency, `eff`, extracted from the histogram `Efficiency0_H`. It is reminded that this offline tool doesn't include any tracking algorithm to identify muons from the beam and only relies on the hits arriving in the time window corresponding to the beam time. The content of the efficiency histogram is thus biased by the noise/gamma background contribution into this window and is thus corrected by estimating the muon data content in the peak region knowing the noise/gamma content in the rate calculation region. Both time windows being different, the choice was made to normalise the noise/gamma background calculation window to it's equivalent beam window in order to have comparable values using the variable `windowRatio`. Finally, to estimate the data ratio in the peak region, the variable `DataRatio` is defined as the ratio in between the estimated mean cluster multiplicity of the muons in the peak region, `MuonCM`, and of the total mean cluster multiplicity in the peak region, `PeakCM`. `MuonCM` is itself defined as the difference in between the total mean cluster multiplicity in the peak region and the normalised mean noise/gamma cluster multiplicity calculated outside of the peak region. The statistical error related to the efficiency, `eff_err`, is computed using a binomial distribution, as the efficiency measure the probability of "success" and "failure" to detect muons.

- The mean muon cluster size, `MuonCS`, is calculated using the total mean cluster size and multiplicity in the peak region, respectively extracted from histograms `MuonCSize_H` and `MuonCMult_H`, the noise/gamma background mean cluster size and normalised multiplicity, extracted from `NoiseCSize_H` and `NoiseCMult_H`, and of the estimated muon cluster multiplicity `MuonCM` previously explicited. The associated statistical error, `MuonCM_err`, is calculated using the propagation of errors of the mentioned variables.

- The mean muon cluster multiplicity in the peak region, `MuonCM`, explicited above whose statistical error, `MuonCM_err`, is the sum of statistical error associated to the total mean cluster multiplicity in the peak reagion, `PeakCM_err`, and of the mean noise/gamma cluster size, `NoiseCM_err`.

In addition to these 2 CSV files, the histograms are saved in ROOT file `Scan00XXXX_HVY_Offline.root` as explained in section 1.2.1.1.

1.7 Current data Analysis

Detectors under test at GIF++ are connected both to a CAEN HV power supply and to a CAEN Anolog-to-Digital Converter (ADC) that reads the currents inside of the RPC gaps bypassing the supply cable. During data taking, the webDCS records into a ROOT file called `Scan00XXXX_HVY_CAEN.root` histograms with the monitored parameters of both CAEN devices. Are recorded for each RPC channels (in most cases, a channel corresponds to an RPC gap):

- the effective voltage, HV_{eff}, set by the webDCS using the Pressure and Temperature (PT) correction on the CAEN power supply,

- the applied voltage, HV_{app}, monitored by the CAEN power supply, and the statistical error related to the variations of this value through time to follow the variation of the environmental parameters defined as the RMS of the histogram divided by the square root of the number of recorded points,

- the monitored current, I_{mon}, monitored by the CAEN power supply, and the statistical error related to the variations of this value through time to follow the variation of the environmental parameters defined as the RMS of the histogram divided by the square root of the number of recorded points,

- the corresponding current density, J_{mon}, defined as the monitored current per unit area, $J_{mon} = I_{mon}/A$, where A is the active area of the corresponding gap,

- the ADC current, I_{ADC}, recorded through the CAEN ADC module that monitors the dark current in the gap itself. First of all, the resolution of such a module is better than that of CAEN power supplies and moreover, the current is not read-out through the HV supply line but directly at the chamber level giving the real current inside of the detector. The statistical error is defined as the RMS of the histogram distribution divided by the square root of the number of recorded points.

Once extracted through a loop over the element of GIF++ infrastructure via the `C++` macro `GetCurrent()`, these parameters, organised in 9 columns per detector HV supply line, are written in the output CSV file `Offline-Current.csv`. The macro can be found in the file `Current.cc`.

References

[1] S. Carrillo A. Fagot. *GIF++ Offline Analysis v6*. 2017. URL: `https://github.com/afagot/GIF_OfflineAnalysis`.

[2] A. Fagot. *GIF++ DAQ v4*. 2017. URL: `https://github.com/afagot/GIF_DAQ`.