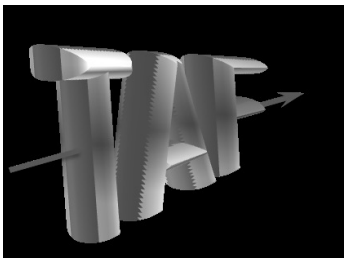
TAF short manual



note edited by J. Baudot (baudot@in2p3.fr)
Universitï¿½ de Strasbourg, IPHC,
CNRS, UMR7178,
Strasbourg, France

for full credits see section 16

2015, January 5

Contents

1 What is TAF?

TAF stands for *TAPI Analysis Framework*, it is the package created and managed by IPHC to characterize CMOS pixel and strip sensors from data acquired with various sources (X-, $\alpha-$, $\beta-$rays, laser) or with particle beams. In the former case, only one sensor is tested. In the latter case, in addition to the device under test (DUT), a telescope made of reference planes is used to identify the particle trajectories (straight tracks) and extrapolate them onto the DUT. Also, because a target can be employed with a beam, vertex reconstruction is required.
Due to the diversity of situations, the software can handle some more or less complex geometries from a single plane to a stack of planes (telescope), where a detection surface can be made of one or several sensors or even double-sided objects. However, only straight tracks are considered.
From the inputs point of view, **TAF** offers the possibility to read a single file or multiple files, possibly produced by various data acquisition system with offline software synchronization. Simulation data can also be analyzed.

This software is meant as a lightweight package, only `ROOT` pre-installation is required. Indeed **TAF** classes are simply added to `ROOT` and you will perform all commands within the `ROOT` environment.
TAF is of course not the only general package for segmented sensor beam test analysis. It is itself an evolution of a previous code named **MAF** (MIMOSA Analysis Framework) developed also at IPHC, [1]. A general package, benefitting from a large user community and freely available, is `EUTelescope` (see eutelescope.web.cern.ch), [2]. Other recent efforts are mentioned as reference [3, 4, 5], but exhaustiveness is simply unreachable here.

This document intends to present the basis to install and start using **TAF** , sections 3, 4, 5, 6. Specific tasks are describe in the following sections 7, 8, 9, 10, 11, 12. Some knowledge of the main algorithm is provided in sections 13, 14, but no detailed descriptions are provided. Much can be learned on the concepts of beam test analysis with the already mentioned **MAF** note, [1], and comparison with **MAF** appears throughout the document. The section 15 gives some hint for potential developers (which can start simply with adding a histogram). Those who will have a look at the source code itself might be stricken (frightened?) by its sometimes unnecessary complexity. The last section 16 might provide some explanations, but more importantly lists hopefully all contributors to **TAF** .

2 General philosophy

The **TAF** workflow is very similar to the one of **MAF** and the same operations come with the same time sequence. The code itself is build upon 3 different parts corresponding to the 3 main functionalities.

1. Tools to decode raw data files from the acquisition systems and build physical events from these data (BoardReader classes): **TAF** allows to plug a new decoding class for any new data acquisition format (contrary to **MAF** which was frozen on a given format). See section 5 for details.

2. Tools for the raw data analysis which produces hits, tracks, mini-vectors or vertices: those are displayed inline event-by-event (see section 9) or massively analyzed and stored in a TTree (see section 10).

3. Tools to perform the final analysis on hits from the DUT or tracks or both (see section 11): this step re-reads hits and tracks from the TTree produced in the previous step and produce final plots for detection efficiency and spatial resolution. For historical reason, it is possible to redo the clustering and alignment for the (In principle, **TAF** allows to analyze hits and tracks reconstructed by another package, the interface is however still to be written.)

For the first two steps, the output files are stored in a directory named `Results/xxxxx` (where `xxxxx` stands for the run number); while the outputs of the third final step are located in the `results_Mxx` (where `xx` stands for the sensor type) directory.

3 Installation

Versions
The code can be obtained from GIT repositories, which are synchronized:

- `https://gitlab.cern.ch/bjerome/taf`,
- `https://github.com/jeromebaudot/taf`.

Two A tar file is also available at the location `http://www.iphc.cnrs.fr/Public-documentation.html`, or from the internal IPHC SVN server (public solution under study).

Directories
Clone the repository or decompressing the archive will create the mandatory basic structure of directories:

- `code`: all the sources and the Makefiles for compilation,

- `Scripts`: useful scripts, see below,
- `config`: contains all the configuration files required for each run, see the section 4,
- `doc`: contains this documentation, the **MAF** write-up as well as the HTML files to browse the code (start with `ClassIndex.html`).

The following directories are created at the configuration, compilation or running steps:

- `bin`: contains libraries and the executable,
- `datDSF`: contains the output TTree,
- `Results`: output files for each run coming from the hit and track reconstruction step,
- `results_Mxx`: output files for the final analysis, where `xx` for the sensor type.

Please read the `README` file which contains useful information on the version, the compilation and running.
A `rootlogon.C` file is also provided with the distribution. There is nothing about histograms appearance, but rather some tricks (libraries loading) to start **TAF** on certain systems.

Configuration
The compilation and running requires the configuration of a number of environment variables, which is performed by the instruction `source Scripts/thistaf.sh`. You shoul edit this file beforehand and possibly change some options.

Compilation
The compilation and running requires the configuration of a number of environment variables, which is performed by the instruction `source Scripts/thistaf.sh`.
On most cases, you will not need to edit the script. But it can happen that editing is necessary, for instance to hard code the ROOT path. In this case, follow the instructions provided in the comments of the script. On Mac-OS, it might still be required to use the old configuration script through the instruction `source Scripts/TAF-config`, but you will need to edit it before.

Compiling can be operated in two ways:

a) issue the `maketaf` command, or equivalently: go to the `code` directory and issue a `make` command, use `make clean` to remove previous compilation results;

b) launch `ROOT` and execute the `Scripts/compilTAF.C` macro, see arguments inside the macro for options.

Be aware that the code does not compile under Windows yet, but runs on various Linux flavors (Debian, Ubuntu, Redhat) or Mac-OS. If the compilation is OK, but **TAF** complains about missing symbols or libraries, usually adding a line in the `rootlogon.C` to force loading the corresponding libraries helps.

4 Configuration or letting TAF knows about your experiment

The **TAF** configuration for a given run, describes in details the geometry of the experiment, the characteristics of the sensors used, the acquisition setup and the final analysis parameters. It is the real "command board" of the software, even if some parameters are hardcoded (see the `README` file for a list). So at first order, you do not need to edit the code at all but rather edit configuration files.

The configuration is written in a text file (directory `config` and extension `.cfg`) which follows the **MAF** data card format `<aField>: <aValue>`. They are many fields, some compulsory and other not. A detailed description of all of them is out of this documentation scope. But their list is provided as appendix 16, which is the reproduction of the comments in the class `DSetup.cxx`. In case of doubt, browse this class, which implements the parsing of the text file.

For your help, the TAF distribution comes with some examples of configuration files, consult the `README` file for their list. Also note that you can generate a copy of an existing configuration file for a new run number with the following script:
`SHELL>source Scripts/copyConfig.sh <oldRunNb> <newRunNb>`

As a baseline, there should be one configuration file per run. However, it is possible to define a generic configuration file which matches several runs. To exploit this generic file feature, the only way is to start **TAF** from the command line with the following options:
`SHELL> TAF -run myRunNumber -cfg myConfigFile`

To know about the various possible running option:
`SHELL> TAF --help`

Some guidances on the five parts of the configuration are provided below, and specific important parameters are explained in the following relevant sections.
As usual, it is difficult to maintain the list of all possible parameters up-to-date. However the description of class `code/src/DSetup.cxx` contains such a list.

4.1 Run Parameters

Run number, date, path to the raw data files or to secondary noise-defining file.
The path to the raw data files is written directly at the beginning of the configuration file as well as the format of the file names (in the acquisition section). There is no need to create symbolic links to the binary file like with **MAF** .

4.2 Parameters of the Tracker

Number of detector planes, tracking and alignment methods and parameters.

4.3 Parameters of the Detector Planes

Defines how one plane is connected to its raw data, parameters of the readout (see subsection 14.1), the analysis (noise, clustering into hits and hit position) and geometry (both position in the laboratory and description of segmentation). One of the fundamental parameter (`AnalysisMode`) indicates wether the sensing elements are strips with analogue output (`AnalysisMode=1`), pixels with analogue outputs (`AnalysisMode=2`) or pixel with binary outputs (`AnalysisMode=3`). "Analogue output" has to be understood as a value coded on more than 1 bit.

Note that planes are declared one after the other. The plane number, used throughout the code to identify planes, is not explicitly defined. Rather the plane number is incremented each time a new plane is declared.
Ladder can also be declared, they are made of several planes. This is a convenient way to define large sensitive area with a single position, made of several smaller areas (planes) with a position defined relatively to the large one (the ladder).

4.4 Parameters of the Data Acquisition

Number of different file types, parameters for each decoder of these files.

4.5 Parameters for Final Analysis

Indicate the goal of the final analysis and contains additional cut definitions, range of histograms and redefinition of sensor segmentation. Regions of interest can also be specified here. This section is new compared to **MAF** , where these information were hardcoded.
Here is a list of available value for the `AnalysisGoal` parameters: `cluster, track, calib, laser, vertex, fake, vector, sitrineo`.

5 Decoding your data: BoardReaders

TAF includes several methods to decode raw data files from various data acquisition system. The idea is that each method corresponds to a specific class. Whatever the decoding class, the input is made of a raw data file or a list of files, and the output is a list of pixels to be used for subsequent analysis. In this way, the decoding is decoupled from the further analysis steps.

This is the current list of decoding possibilities (number in parenthesis is the type flag recognized by **TAF**). This might not be up-to-date, you will have to go through `code/src/DAcq.cxx` (for instance, look at the constructor) to understand what are the existing choices.

- `IMGBoardReader` (1): IPHC USB-Imager board.
- `TNTBoardReader` (3): IPHC TNT board.
- `PXIBoardReader` (4): IPHC National Instrument PXI IO board.
- `PXIeBoardReader` (5): IPHC National Instrument PXIexpress FlexRIO board.
- `GIGBoardReader` (6): output of GEANT4-based simulation, including pixel signal modeling, package by A.Besson and L.Cousin.
- `VMEBoardReader` (7): INFN CAEN VME board for sparsified binary output sensors (MIMOSA-26/28).
- `AliMIMOSA22RawStreamVASingle` (8): INFNF CAEN VME board for binary output sensors (MIMOSA-22).
- `DecoderM18_fb` (9): INFN CAEN VME board for analog output sensors (MIMOSA-18)
- `DecoderGeant` (10): INFN GEANT-based simulation, without pixel signal, package by P. La Rocca.
- `MCBoardReader` (11): Monte-Carlo output from the GEANT4 based full simulation developped by A. Perez Perez.
- `BoardReaderIHEP` (12): IHEP acquisition for binary output sensors MIMOSA-28.

6 Generalities on how to run the code

Start with no command line options
To run any algorithm from **TAF** , you first need to instantiate an object of type `MimosaAnalysis` with the name `gTAF`. On LINUX this is done automatically if you have compiled with "make" when you launch **TAF** . On MAC-OS or on LINUX, if you have compiled using the macro `compiltaf.C` you will have to instantiate this object yourself within ROOT:

```
TAF> MimosaAnalysis *gTAF = new MimosaAnalysis()
```

From that point, the running flow is similar to the **MAF** one with some additions. First, you always have to initialize the session (read the configuration file) with:

```
gTAF->InitSession(myRunNumber)
```

Be sure you have the corresponding `config/run<myRunNumber>.cfg` ready.

You may get help on the available commands with: `gTAF->Help()`

Available command line options

- `TAF --help` lists these options.
- `TAF -run myRunNumber` starts **TAF** and launches automatically the `gTAF->InitSession(myRunNumber)` command.
- `TAF -run myRunNumber -cfg myConfigFile` starts **TAF** , sets the run number to `myRunNumber` and initializes with the given configuration file (useful for generic config file).
- `TAF -run myRunNumber -guiw` starts **TAF** and launches automatically the `gTAF->InitSession(myRunNumber)` and then the `gTAF->GetRaw()` commands.
- `TAF -run myRunNumber -guix` starts **TAF** and launches automatically the `gTAF->InitSession(myRunNumber)` and then the `gTAF->GetRax()` commands.

Debugging

You may set the debugging level at any time with (the higher the more messages, 0 to turn back to quite mode, negative levels switch debugging for the decoding methods, while positive levels switch debugging for the clustering and tracking methods): `gTAF->SetDebug(aDebugLevel)`

Workflow

The next batch of commands could consist in:

- compute and store the pedestal and noise for each pixels, see section 7,
- generate a lookup table for the so-called η-algorithm which optimizes the position reconstruction with respect to a center of gravity method (only valid for analog output sensors);
  use the command:
  `gTAF->MakeEta()`
- align the telescope, see section 8,
- perform event-by-event analysis, see section 9;
  use the following command to make a menu appear:
  `gTAF->GetRaw()`
- perform the data mining to reconstruct hits and tracks, see section 10;
  use the command:
  `gTAF->DSFProduction(1000000)`

- perform the final analysis step correlating hits with tracks, see section 11;
  use the command:
  `gTAF->MimosaPro(...)`

It is advised to quit and restart **TAF** between each step.

7 Noise analysis

7.1 Noise with analogue output pixels

Pixel with *analogue* output means that the pixel value has more than 1 digit. This corresponds to the case when you specify a readout mode (`Readout` data card) below 100.
To compute the pedestal and noise, **MAF** was relying on the fact that a value for each channel was available for each event. The pedestal and noise were computed on the first events and then updated regularly.
For other mode, either you do not need the pedestal and noise values or a dedicated run is used to initialize them.
In the later case, this run shall have a configuration file with a `Readout` below 100. Then after the initialization issue the command (equivalent to click on the `NOISE MAP` menu described in section 9) :
`TAF>gTAF->GetRaw()->Noise()`

A file `Noise_run<NoiseRunNb>.root` is generated in the `Results/<NoiseRunNb>` directory containing a map of the pedestals and noises. In order to use these values for a new run `myRunNumber`, two steps are required:

- copy this file into the directory `Results/<myRunNumber>` corresponding to the run with data to analyze,
- edit the configuration file `config/<myRunNumber>.cfg` to set the following datacard:
  `NoiseRun:  NoiseRunNb`

Then, when the hit reconstruction will be performed on the `myRunRunNumber` data, the pedestal and noise values will be taken into account.

7.2 Hot/noisy pixels with binary output

With binary data, readout mode (`Readout` data card) above 100, there is no concept of noise associated to one pixel. However, you can still search for hot or noisy pixels, based on the firing occurrence without any source (dark count or fake rate). **TAF** allows you to compute first the fake hit rate per pixel and then proposes to ways to masking pixels from the analysis.

Fake rate analysis

Use the following method from the Event-by-Event analysis mode (see section 9).
`gTAF->GetRaw()->DisplayCumulatedRawData2D(N,minSN,occ_min,occ_max,nOcc,minOcc,storageO`
Where `N` is the number of events (frames) to process, and `storageOcc` is a bool that needs to be set to true. The other parameters can be set to the default values, `Occ_min = 0.001, Occ_max = 1.0, nOcc = 20, minOcc = 0`.
The system will produce the fake rate map in the column vs row plane and respond with messages like this:
`-------- The HOT PIXEL MAP FOR PLANE 1 HAS BEEN FILLED !`
The results are stored in `Results/NoiseRunNumber/` directory. The 2D histograms containing the fake rate per pixel have the name `hotPixelMap_runNoiseRunNumber_pl1.root`. There is also a text file with the list of pixels above, named `Results/NoiseRunNumber/rawData_runNoiseRunNumber.txt` .

Masking hot pixels, option 1

This first method does not require to re-compile the code. In the text configuration file, precisely in the block of the plane parameters for which you want to mask some pixels, you need to specify two parameters:

- `HotPixelMapFile`: The name of the root file containing the fake rate map
- `FakeRateCut`: A cut on the fake rate (R) per single pixel fake rate, all pixels with rate > `FakeRateCut` will be excluded from further analysis.

Masking hot pixels, option 2

This second method requires to recompile **TAF** . You need to create a specific function in the file `code/include/vetoPixels.c` based on the functions already there and the list of pixels you want to mask, chosen from the list in `rawData_runNoiseRunNumber.txt`. Then, you should edit the class `code/src/DGlobalTools.cxx` and modify the method `DGlobalTools::SetVetoPixel( int noiseRun)` to link your previously defined vetoFunction with a given run number (typically `NoiseRunNumber`). You have to recompile **TAF** then.
Finally, in your configuration file, you should specify in the `Run Parameter` section the following paramter:
`NoiseRun: NoiseRunNumber`.
When TAF will read the raw data, all pixels listed in your vetoFunction will be ignored.

8 Tracker Alignment

General strategy
Alignment has to be understood as the determination, for each detector (planes in the `TAF` naming), of the 6 parameters (3 translations and 3 rotations) which define the position and orientation of a plane in the laboratory or telescope frame (see section 13). This frame is defined by one or two fixed reference planes named. It means that the strategy is always to align a single plane with respect to the tracks built from fixed planes, even though several planes are aligned simultaneously. The data from one plane to be aligned, do not impact the alignment of another plane studied simultaneously. This is **local alignment**, in contrast with global alignment which would determine all plane positions taken into account all plane data.

They are several alignment procedures in **TAF** to align planes, they are all iterative and share the same following basic concepts.

1. **Accumulating**: a pre-defined number of hit-track associations are chosen from the data, within a maximal given distance.
2. **Minimizing:** compute in the frame associated to the plane the translations and rotations which minimize the sum of the squared hit-to-track distances.
3. **Updating:** propagate the new alignment parameters and the maximal distance of the first accumulating step, and be ready / decide for the next iteration.

The various procedures differ by how these steps are actually implemented. Once the iterations stop, the new alignment parameters of each plane are **automatically updated in the configuration file**. So, pay attention when re-editing the configuration file afterwards an alignment, that could overwrite the alignment values.

Note: For **MAF** user, `AlignmentTilt, AlignmentU, AlignmentV` are not used any more, and are set to 0 after the alignment. **TAF** uses and updates the following fields: `PositionsX, PositionsY, PositionsZ, TiltZ, TiltY, TiltX`. If the three parameters are still present in a `AlignmentTilt, AlignmentU, AlignmentV` in a configuration, **TAF** will take them into account to compute a first position, but after alignment they will be set to 0 and so ignored.

Choice of planes aligned or fixed
Planes are used as fixed references for building tracks or aligned depending on their `Status` and the value of the `AlignStatus` flag. The plane `Status` is set in the configuration file (see section 4) according to:

0: "seed" plane, always fixed (also used as a seed point for tracking, see the subsection 14.3);

1: primary reference, could be fixed or not;

2: secondary reference, could be fixed or not;

3: DUT, never fixed.

As underlined previously, **TAF** can accommodate any number of planes fixed or to be aligned, but there shall be at least one fixed plane.
The `AlignStatus` is set by the online command:
`TAF>gTAF->SetlAlignStatus(`*myStatus*`)`.
Any plane with a `Status` strictly greater (lower or equal to) the `AlignStatus` is considered as to be aligned (fixed).

Track selection

Describe the two types of selection: χ^2 cut and geometrical cut

First possible procedure: `AlignTracker`

`AlignTracker` is an automatic procedure, which decides by itself to continue the iterations or to stop. All planes are aligned with the same number of iterations, and the hit-track association is done with the same maximal distance for all of them. The iterations stop when, for all planes, the change for all parameters is below a hardcoded value (in method `MAlign::AlignTracker`). The fit also stops when the maximum number of events allowed for iterations is reached.
The specific features of this procedure are:

- only one hit can be associated per track, and it is the nearest one;
- only adjust 3 parameters, two positions `PositionX, PositionY` and one rotation angle `TiltZ` perpendicular to the beam;
- use an analytic least square minimization formula (possible because of the limited number of parameters).

The procedure is launched by this command:
`TAF>gTAF->AlignTracker(` *myDistance, myAlignEvents, myAdditionalEvents* `)`

myDistance is the maximal distance in μm to associate a hit with a track. It is used for the first iteration and then reduced automatically at each new iteration depending on the calculated residual width.
myAlignEvents is the number of events used for iterations, if reached before the stopping condition, the fit stop anyway. This number can be 0, in which case no fit is performed but plots are produced. This is very useful to have a first picture of the situation.
myAdditionalEvents corresponds to the number of events used after the fit, to produce control plots.
The number of hit-track associations used in the fit of one iteration is given by the parameter `EventsForAlignmentFit` from the configuration file (see appendix 16).
A typical alignment goes like this:

- First **TAF** session:
  `TAF>gTAF->SetlAlignStatus(0)` *(indicate only seed planes are used for tracking)*
  `TAF>gTAF->AlignTracker(20000,0,2000)` *(note the very large distance hit-track required and no events for fit)*
  From the plot, you may change the positions by hand in the configuration file, since the automatic procedures is unable to correct for very large shift (few millimeters).

- Second **TAF** session:
  `TAF>gTAF->SetlAlignStatus(0)` *(indicate primary, secondary ref. and DUTs will be aligned)*
  `TAF>gTAF->AlignTracker(2000,10000,2000)` *(note the large distance hit-track required)*

- Third **TAF** session:
  `TAF>gTAF->SetlAlignStatus(1)` *(indicate primary ref. are used for tracking all secondary ref. and DUTs will be aligned)*
  `TAF>gTAF->AlignTracker(500,10000,2000)`
  You might repeat this step with smaller track-hit distances to improve the results.

- Last **TAF** session:
  `TAF>gTAF->SetlAlignStatus(2)` *(indicate all ref. are used for tracking only DUTs will be aligned)*
  `TAF>gTAF->AlignTracker(100,10000,2000)`

Second possible procedure: `AlignTrackerMinuit`
`AlignTrackerMinuit` is a semi-automatic procedure, since the user decides when to stop the iteration and which maximal hit-track distance to set at each iteration and for each plane. Consequently, all planes will not be aligned with the same number of iterations, which allow to handle different situation simultaneously. The specific features of this procedure are:

- all hits within the maximal distance allowed are associated to a given track;

- adjust all parameters, the three positions `PositionX, PositionY, PositionZ` and the three rotation angles `TiltZ, TiltY, TiltX`, although `PositionZ` is fixed by default;

- use an interactive `MINUIT` session to perform the minimization at each iteration and for each plane, this is extremely useful since it allows to change the parameters online without editing the configuration file.

The procedure is launched by this command:
`TAF>gTAF->AlignTrackerMinuit(` *0, myDistance, myAlignEvents, myAlignHitsInPlane, myAdditionalEvents, myChi2Limit* `)`

The first parameter is meant to be an option for an automatic procedure (if 1), but it is not very well tested.
myDistance is the maximal distance in μm to associate a hit with a track. It is used for the

first iteration of each plane and then at the end of each iteration the user can set a different distance for this plane. If the user choose a negative distance, the iterations for this plane stop.
myAlignEvents is the number of events used for iterations, if reached iterations for all planes stop.
myAlignHitsInPlane is the number of hit-track association to cumulate before fitting for one iteration on a single plane.
myAdditionalEvents corresponds to the number of events used after the fit, to produce control plots.
myChi2Limit is a potential maximal χ^2 from the track fit allowed if the user wishes to select track.

The alignment strategy follows the same path as with the `AlignTracker` method: successive calls to `AlignTrackerMinuit` with increasing `ALignStatus`.

9 Event-by-event analysis and display

TAF allows to view the raw data and perform the analysis event-by-event through the `MRaw.cxx` class methods; simply issue the command:
`TAF>gTAF->GetRaw()`

A comprehensive menu pops up. The `NOISE MAP` method was already described previously in section 7, other descriptions are provided in the code itself. You can get more printed output with the `Toggle Verbosity` button (for debugging, use `gTAF->GetRaw()->SetDebug(aDebugLevel)`). Note that each method usually has parameters that you may modify from default by calling them directly from the command line with `TAF>gTAF->GetRaw()->[Method]( arg1, arg2, ...)`

Some of these methods, named `CumulateXXXX`, add the results over a given number of events. They are useful to plot characteristics of fired pixels, hits or tracks in order to check there are real hits on the detector planes as well as to optimize the selection cuts and parameters of the clustering and tracking.

Note that the list of methods available in the menu depends on the analysis goal, you have set in the parameters for Final Analysis, see sub-section . For instance, if there is only a single plane, no alignment method will be proposed.
Note that most of those methods write some output files, including histograms and/or text, that you will find in the directory `Results/xxxxx` where `xxxxx` stands for the run number. Also, the method displaying hits does plot the extrapolated track impact as well. Because tracking is involved, its behavior depends on the Tracker alignment status (see sections 8

and 10).

You may notice that there is a `USER PLOT` method. This one is intended for the user to modify as she/he wants following the provided template.

Note: An additional class, `MRax.cxx`, with enhanced graphical user interface and display has been introduced by V.Reithinger. It currently works successfully only for a very specific geometry (4 double planes of MIMOSA-26). But it will be made available for other geometries in 2015.

10 Raw data analysis

To produce the `TTree` considered as the data summary file containing hits and tracks information, use (like in **MAF**):
`TAF>gTAF->DSFProduction(<myEventNumber>)`

The clustering and tracking algorithms are those described in section 14, as well as the selection criteria, which are entirely defined in the configuration file. Note that if you are analyzing a single plane or do not want tracks, you can skip the tracking step by setting the field `TracksMaximum` to 0 in the configuration file.

Outputs
The objects stored in this `TTree` are defined in the class `DEven.h`. It is important to underline that for hits (subclass `DAuthenticHit`), the information (index, signal, noise) of each constitutive pixels are stored. It goes the same with tracks, except that an object `DTransparentPlane` is stored for each plane crossed by the same track.
Further analysis can be done either accessing directly the `TTree` leaves with standard `ROOT` method or with the tools provided by `TAF` like explained in section 11.
To save some disk space, the `TTree` does not store by default the hits from the reference planes (status different from 3). To force the storage of these hits as well, use the following option:
`TAF>gTAF->DSFProcudtion(<myEventNumber>, 0)`

The output `TTree` is stored in the file datDSF/runxxxxx_nn.root where `xxxxx` is the run number and `nn` a number increased each time `DSFProduction` is invoked for the same run, exactly like in **MAF** . In parallel the final printouts, summarizing how many events were read and the number of hits per plane as well as number of tracks, are saved in the `DSFProd.log` file located in **TAF** home directory.

You may ask for a number of events larger than the one available, **TAF** will stop anyway after the last one.

11 Final analysis

Initialization
You can use the methods described here only after the raw data analysis (see section 10) which produces the `TTree`. The goal is to obtain the final histograms for your analysis of the plane corresponding to the DUT. You can add more selections through the arguments of the methods described here or within the dedicated part of the configuration file. It is also possible to re-align the DUT with respect to the trackers.
Before performing the anayzis itself, you shall indicate which plane you choose as DUT (you may have several planes with status 3). You have to method to specify your DUT:

a) at the initialization with: `gTAF->InitSession(`*myRunNumber*`,`*myDUTnumber*`)`;

b) after the initialization with: `gTAF->SetPlaneNumber(`*myDUTnumber*`)`.

Additionally, the file containing the `TTree` which will be analyzed is by default `datDSF/run`*myRunNumber*`_NN.root`, where `NN` is the higher number available in the directory. You may want to choose another file however, for instance if you have renamed it after `DSFProduction`, in that case use:
`gTAF->SetDSFFile(`*myFile*`)`.

Available methods and outputs
This final analysis part is readily used as in the **MAF** case except that they are more methods (get the list by browsing the code or with the command `gTAF->Help()`):

- `MimosaPro`, this is the central method for efficiency and spatial resolution studies with a telescope, identical to **MAF** except that there is no more two different methods for analog or digitized outputs;
- `MimosaCluster`, performs the hit searching on planes only (no tracking) and fill histograms which characterize in detail the reconstructed clusters;
- `MimosaCalibration`, starts with `MimosaCluster` and adds specific histograms useful when calibrating the sensor with a ^{55}Fe source;
- `MimosaFakerate`, computes the fake hit rate per pixel in the absence of beam;

- `MimosaMiniVectors`, is similar to `MimosaPro` but consider the DUT is composite object made of two planes;
- `MimosaPro2Planes`, performs the same analysis as `MimosaPro` but for two planes simultaneously;
- `MimosaProLadder`, performs the same analysis as `MimosaPro` but for all the planes associated to a ladder.
- `MimosaVertex`, tries to find a vertex from all the track present in the events.
- `MimosaVertexFinder`, ??
- `MimosaImaging`, assumes the sensor built an image (from the accumulation of individual impacts) of a resolution chart made of stripes and performs a fit to determine the spatial resolution.

All those methods read the `TTree` generated by `DSFProduction` and then generate a pop-up menu to plot the final histograms, which are partly described in the **MAF** note and in the `MPost.cxx` file. Note that the ranges of these histograms can be adjusted through some configuration parameters, see paragraph `Parameter for Analysis` of the appendix 16.
Whenever you run a menu command, the corresponding plots are stored in a file named `results_ana_M`*XX*`/run`*NNNNN*`Pl`*P*`_ClCharge.root`, where *XX* stands for the DUT type, corresponding to plane number *P* and *NNNNN* stands for the run number. Also the printed information on the output are stored in the file `results_ana_M`*XX*`/Main_results.csv`.

Selection criteria
Selection cuts are adjusted in two ways, some are set through the argument of the `MimosaXXXX` method and others are set in the configuration file, under the section `Parameter for Analysis` (see the appendix 16 for details).
They are basically 4 types of selection cuts available so far:

- **general cuts:** are requirements on the total number of hits or tracks (even number of tracks in a given area) in the event;
- **hit related cuts:** are requirements on the SNR, charge or number of pixels in the hit;
- **track related cuts:** are requirements on the χ^2 or number of hits (even the plane used shall be possible because the information is present in the `TTree` but not yet implemented);
- **geometrical cuts:** allow to select a particular region of a sensor, for the hits and/or for the track. For hits, this is done by selecting a range of rows or columns, or a range of pixel indices (only from the configuration file). For tracks the concept of `geomatrix` is used. Four ranges in the plane coordinate system have to be defined in the configuration file (`GeoMatrix0, GeoMatrix1, GeoMatrix2, GeoMatrix3`). Then, the geomatrix index appears as an argument of the `MimosaXXXX` method used.

The selections listed above will be automatically applied if their cut values are set. It is of course possible to include any other cuts by hard-coding them in the `MCommand::MimosaXXXX` method of interest.

For historical reason, **TAF** manages `submatrix` in the final analysis. These `submatrix` are defined in the configuration file and for each of them the complete pixel matrix have to be redefined (pixel pitch, number of columns, rows, mapping, calibration constant). Also some of the selection cuts are specific to `submatrix`, like the `geomatrix` or number of pixels in a hit. This feature is useful for various purposes.

- Several DUTs are present, one `submatrix` per DUT can be defined.
- Various analysis have to be performed for the same DUT, each selections can be defined as different `submatrix`.
- The DUT plane features several submatrices with different pixel sizes and numbers (or same pixel size but different treatment micro-circuits which affect noise and gain) but was declared as a single plane, each one can be addressed by a specific `submatrix`. This is actually the way it was done in **MAF** . In **TAF** , it is always possible to define several DUT planes corresponding to the different physical submatrices, if preferred.

Note: For **MAF** users, the differences is that the definitions previously hardcoded in the `MPara.cxx` file have been replaced by parameters in the configuration file, under the `Parameter for Analysis` section (see the appendix 16). Additional cuts are also available in this section.

For the convenience of users developing a piece of code in the final analysis (either MAnalysis.cxx or MCommands.cxx files), a specific variable, names UserFlag, can be set in the configuration file and is available as a data member of the MAnalysis class, so available anywhere.

11.1 `MimosaPro` analysis

More detailed description (additional alignment, options to save good and/or missed hits, option to avoid hot pixels) ...

11.2 `MimosaCluster/Calibration` analysis

More detailed description (fit of the calibration peak) ...

11.3 `MimosaFakerate` analysis

More detailed description on the various histos filled ...

11.4 `MimosaMiniVectors` analysis

More detailed description on the creation of minivectors ...

12 Simulated data

TAF does not provide a way to simulate data, but rather offers the possibility to read some simulated data (typically obtained with `GEANT`) and perform the full analysis. Two `BoardReaders` have been developed depending on wether hits have been digitized (signals on each pixel known) or not (only true energy deposition and position known).

Simulated data already digitized
Work by Loic Cousin and Auguste Besson , consul class `GIGBoardReader`.

Simulated data with true hits
Work by Paola La Rocca, consult class `DecoderGeant`.

Full GEANT4 simulation
A full GEANT4 based simulation package, using the same text configuration file as **TAF** and a CMOS digitizer algorithm, was developped by Alejandro Perez Perez. It is mainly dedicated to reproduce test beam configurations.
More explanations are needed here.

13 Geometry description

Coordinate systems and transformations
TAF uses two types of coordinates. Local frame for each plane, corresponds to the $\vec{U}(u, v, w)$ coordinate systems. While the Laboratory or telescope frame uses $\vec{x}(x, y, z)$ system. The telescope frame is defined by the positions (`PositionsX, Y, Z`) and orientations (`TiltZ, Y, X`) of "seed" plane, the one with `Status: 0`.

The position and rotation parameters written in the configuration files for other planes are the alignment constants. They provide the quantitative information to perform coordinate transformation from one frame to another. The three positions define a translation vector $\vec{T}$, while the three angles define a 3×3 rotation matrix $\mathbf{R}$:

$$\vec{T} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & \sin\theta_X \\ 0 & -\sin\theta_X & \cos\theta_X \end{pmatrix} \times \begin{pmatrix} \cos\theta_Y & 0 & -\sin\theta_Y \\ 0 & 1 & 0 \\ \sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix} \times \begin{pmatrix} \cos\theta_Z & \sin\theta_Z & 0 \\ -\sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Transformations are consequently expressed like this:

$$\begin{aligned} \vec{U} &= \mathbf{R}\,(\vec{X} - \vec{T}), \\ \vec{X} &= \mathbf{R}^{-1}\,\vec{U} + \vec{T}. \end{aligned}$$

The class `DPrecAlign` contains the alignment parameters and the methods for the transformations. There is one such object for each `DPlane` object. Additionally, `DPrecAlign` proposes methods to compute the intersection of a plane with a line in space and to convolute or deconvolute two sets of alignments (for instance if you want to know the position-orientation of a given plane with respect to another).

Note: During the final analysis, you may perform an additional refined alignment of the DUT. In this case, the alignment parameters are stored (as a `DPrecAlign` object) in a specific file: `results_ana_M`XX`/CorPar_`P`.root` where XX is the sensor type and P is the index of the plane. So the parameters in the configuration file are not anymore up to date, however it is always the class `DPrecAlign` that does the transformation. Also you can figure out which parameters are stored in the `CorPar_[P].root` file by opening it within TAF and use the method DPrecAlign::PrintAll() on the object inside.

Planes at same `z`

There is no problem with defining two or more planes at the same `z` coordinate, i.e. same position with respect to the beam.
However, to ensure tracking (if needed) is performed on all these planes, they should feature the same `Status`. For instance, imagine you have several stations along the beam, each made of two butted sensors. Let's say, there are the top and bottom sensors. Then you shall set a `Status: 0` for at least one bottom sensor **and** one top sensor.

Set of planes: `Ladder`

The `Ladder` object allows to group a number of planes and define their relative position with respect to a single point. This point becomes the center of the `Ladder` and alignment

will solely modify it, leaving untouched the relative positions. You need to use the dedicated command for such an alignment:
`TAF>gTAF->AlignLadder(...)`
Note that the relative positions of the planes belonging to the ladder, can be absolutely arbitrary. Hence the term "ladder" is not entirely appropriate and comes from the first application of this feature to the PLUME double-sided ladders.
The implementation is done in the `DLadder.cxx` class, mostly developed by Loic Cousin.

In the configuration file, a new section is needed for each `Ladder` declared. It contains the list of associated planes, geometrical informations (center position and all relative positions) and the `Status`. Any position or `Status` information specified for a plane belonging to a `Ladder` will be ignored and superseded by the `Ladder` one. Consult the appendix 16 to get the list of `Ladder` configuration parameters. If your plane positions are already known from a given configuration, you may also use the command: `TAF>gTAF->GetRaw()->MakeLadderGeometry(...)` to generate the ladder parameters needed by the configuration file. Consult the class `MRaw.cxx` to learn about the arguments of the method.

Plane deformation
It is possible to take into account the deformation of the surface of a plane, which otherwise is assumed flat.

More complex situations
Currently it is not possible to have several telescopes (aka `Trackers`) with different orientation. For instance that would be the case with a first telescope on a beam prior a target and a second telescope located after the target. One would then want to reconstruct a single track from the beam to estimate the vertex position in the target and simultaneously for the same event track all the outgoing particles from the vertex.
The structure of the code would easily allow it but it is not yet developed (configuration, loop to update each trackers or visualize them). Please, feel free to contribute!

Visualization
In order to visualize the plane positions and orientation, use the command: `gTAF->GetRaw()->DisplayGeometry()`
The implementation of the transformation from one frame to the other is done in the `DPrecAlign` class. However the object `DPlane`, representing a detector plane, provides the interface to these transformation with the methods: `TrackerToPlane`, `PlaneToTracker` and `Intersection`.

14 Basic algorithms description

We provide here a basic description of the structure of the main algorithms, clustering and tracking, and where you will find their implementation in the code. Prior to this short introduction, a note is given on the object used by these methods.

14.1 Containers for sensing elements

Pixels and/or Strips: `DStrip` **and** `DPixel`
Difference between the `DStrip` and `DPixel` objects.

Hits or clusters: `DHit`

Tracks: `DTracks`

14.2 Clustering

Strategy
Clustering identify pixels groups to make a hit and compute the charge associated to this hit as well as its position. The strategy in **TAF** consists in selecting a seed pixel and gather neighboring pixels to it, that is clustering. The selection of the seed pixel is implemented in the method `DPlane::find_hits()`. The various algorithms for clustering are implemented in `DHit`, which choice is controlled by the configuration variable `HitFinder` (default being `0`).

Analog output clustering: `DHit::Analyze`
If the plane has analogue outputs (`AnalysisMode=1` or `2`), then seed pixels are identified as the ones with the highest signal. In this case, the hit finder algorithm starts with the highest signal pixel, tries to build a cluster around, marks any pixels associated to the hit as such. Then it resumes the same procedure with the next highest signal pixel which is not yet associated to a hit. And so on, up to the exhaustion of available pixels.

There is only one clustering algorithm in the analogue case, so the `HitFinder` parameter is ignored. Association of a new pixel to a hit works on the basis of distance cuts between the new pixel and the seed pixel. For pixel (2D segmented planes) the distances in both direction are tested against the cut value defined in the configuration file `ClusterLimitU` and `ClusterLimitV`. There is no cut on the pixel value. So for instance, if `ClusterLimitU=2×PitchU` and `ClusterLimitV=2×PitchV`, then hits can be as large as a 5×5 pixels area with 25 pixels.
Currently, each pixel can only be associated to a single hit, there is no final procedure to separate what might be merged clusters.

After the clustering algorithm, the hit selection is performed (see below). If it fails, all the pixels but the seed in the tested cluster are released so as to be available for another

potential hit.

Binary output clustering: `DHit::Analyze`

If the plane has binary outputs (`AnalysisMode=3`), the first available pixel is tried as a seed pixel. Then the clustering behavior depends on the value of `HitFinder`.

- `HitFinder=0` (default): Pixels are associated with potential neighboring pixels according to the same algorithm used for analogue output (`DHit::Analyze`). During this gathering process, it might appear that the seed pixel is no more the central pixel due to the addition of new pixels. In this case, the seed pixel is redefined as the central one (in terms of position barycenter) and the association process restarts from the beginning of available pixels.

- `HitFinder=1`: dynamic clustering algorithm implemented in the method `DHit::Analyse_Dynamic`, where new pixels are associated if they are a direct neighbor of one of the pixels already associated to the hit.

- `HitFinder=2`: corresponds to method `DHit::Analyse_2_cgo` which requires an additional parameter `ClusterLimitRadius` in the configuration file.

Hit selection

There are two types of selection cuts to decide to keep a hit or not, all cut values are defined in the configuration file.
The first type concerns the signal values, and thus they are used only for analogue outputs:

- signal-over-noise ratio for the seed pixel $\leq$ `ThreshSeedSN`,

- signal-over-noise ratio for the neighbor pixels $\leq$ `ThreshNeighbourSN`, where "neighbor" means all pixels of the cluster but the seed.

The second type concerns the number of pixels in the hit that shall be within the inclusive limits: `MinNStrips` and `MaxNStrips`. In fact the variable `MaxNStrips` is not used.

Implementation

The seed finding is implemented in the method `find_hits()` within the `DPlane.cxx` class. The gathering of neighboring pixels and the estimate of the hit properties are done in the `DHit.cxx` class with the `Analyse` method. Note there are two such methods depending on wether the analysis is performed with `DStrip` or `DPixel` objects.

14.3 Tracking

Strategy

The tracking strategy in **TAF** follows the standard approach to extrapolate the track from one plane to the other in an iterative way. The implementation is currently limited only to a straight track model and does not take into account multiple scattering (see later on an alternative strategy to deal with it). The track starts with a single hit and a zero slope. Then, the track seed extrapolation to the next (with respect to the index, not necessarily

with respect to geometry) plane defines the center of a circular search area (which size is fixed by the configuration field `SearchHitDistance`). If there are hits on this plane within the search area, the nearest one to the center is associated to the track. The parameters of the track are recomputed, and the iteration goes on with the next plane. Note you have two options for the extrapolation once at least two hits have been selected, either the track is kept extrapolated with a zero slope (configuration `UseSlopeInTrackFinder: 0`); either the extrapolation uses the computed slope (configuration `UseSlopeInTrackFinder: 1`) Once all planes have been scanned, the track is tested against selection cuts.

Role of planes in the tracking depends on their status in the configuration file and on the status of the alignment:

- `Status: 3`, the plane is ignored by tracking,
- `Status: 0`. all hits of the plane are used as track seed,
- `Status: 1` or `2`, hits of the plane are considered for association to an existing tested track, if the alignment status is lower or equal to the plane status.

Note that it is perfectly fine to set a `Status 0` for all planes entering the tracking. In this latter case, each hit of each `Status 0` plane, will be tested as a track seed (unless it has already been associated to a previous track).
Consequently the following commands will have the following effects:

- `TAF>gTAF->SetAlignStatus(2)`: this is the **default**, you don't need to issue the command if you want this option, all reference planes (`Status` 0, 1 or 2) are used for the tracking and the minimum number of hits to build a track is the one defined by the field `PlanesForTrackMinimum` of the configuration file;
- `TAF>gTAF->SetAlignStatus(1)`: only the primary reference planes (`Status` 0 or 1) are used for the tracking and the minimum number of hits to build a track is reduced with respect to the one defined by the field `PlanesForTrackMinimum`;
- `TAF>gTAF->SetAlignStatus(0)`: only the "seed" plane(s) (`Status` 0) is(are) used for the tracking and the minimum number of hits to build a track is set to 1.

Alternative strategies
The strategy described above is selected by default or corresponds to the configuration parameter `TracksFinder: 0`. Other strategies are available and briefly described here.

Ordering plane for tracking: `TracksFinder: 1`
This method `DTracker::find_tracks_1_opt` allows you to change the order through which planes are searched for hits to associate to a given track. Either the plane index (which comes from the order in which planes are declared in the configuration file) is used (configuration `TrackingPlaneOrderType: 0`), either the plane status defines which comes first (status 0 first, then status 1and finally status 2) if the configuration `TrackingPlaneOrderType: 1`

is set.
Another features allows you to change the search distance between the track extrapolation and the hit in the next plane depending on the number of hits already associated. If the track is just starting and has only one hit, the value `SearchHitDistance` is used; but if the tracks holds already 2 hits, then the value `SearchMoreHitDistance` is used.

Tracking dedicated to ion vertex imaging: `TracksFinder: 2`
Look in the implementation `DTracker::find_tracks_2_ivi` to understand what happens there. Pay attention, many parameters shall be set in the configuration.

Tracking dedicated to ion vertex imaging: `VertexMaximum: 1`
If vertexing is required then the method `DTracker::find_tracks_and_vertex` is used.

Tracking with strip sensors:
When the first plane is a strip sensor, case recognize in the configuration for this plane has `AnalysisMode: 1`, then the tracking proceeds with the default strategy but with some specificities, due to the fact that hits contain only 1D information. The method `DTracker::find_tracks_withStrips` is used, BUT IT HAS NOT YET BEEN FULLY VALIDATED ← use at your own risk! (Sorry for that.)

Track fit
They are four track parameters to describe a straight line in space (x, y at $z = 0$ and slopes $\mathrm{d}x/dz$, $\mathrm{d}y/dz$). They are obtained by a least square fit, where uncertainties match each plane resolution. Hence it is important to specify the spatial resolution you expect for each plane used for the tracking. Use the configuration file in three different ways depending on the situation.

- If all planes share the same resolution, use the `Resolution` field in the `Parameters of the Tracker` section.
- If planes feature various resolutions, use the `PlaneResolution` field in the `Parameter of the Detector Planes` section.
- If plane resolutions depend on the direction, use both the `PlaneResolutionU` and `PlaneResolutionV` fields in the `Parameter of the Detector Planes` section.

Three χ^2 values are available after the fit, two for each direction and the last one being their sum.

Use `TAF>gTAF->GetRaw->DisplayCumulatedTracks()` (see section 9) to check tracking performances before launching long data mining with `DSFProduction`.

Track selection
The few parameters driving the tracking behavior are defined in the `Parameters of the Tracker` section of the configuration file. Their list is:

- `TracksMaximum`: maximum number of track to reconstruct,
- `PlanesForTrackMinimum`: minimum of hits (1 hit per plane) to be associated to a track, if not reached the tested track is discarded,
- `HitsInPlaneTrackMaximum`: excludes a plane from tracking for a given event, if the number of hits reconstructed in this plane is larger than this value,
- `SearchHitDistance`: defines the hit search area from the track extrapolation.

Alternative strategy with large multiple scattering
In the case of the evaluation of the single point resolution of a DUT with a telescope made of several (reference) planes and particles suffering from non negligible multiple scattering, the overall track fit exploiting all the reference points will certainly not yield the best track extrapolation accuracy at the DUT. Hence the residual width might be severely impacted by the multiple scattering.
Using only the set of two planes on both sides of the DUT to define the track parameters could help reducing this impact. **TAF** is able to generate such a "subtrack", based on a subset of points from the full track.
The exact number of planes allowed in the subtrack and their identifier have to be specified in the `Parameters of the Tracker` section of the configuration file. Here comes an example with two planes (number 3 and 4):

```
SubtrackNplanes:            2
SubtrackPlanes:             3: 4
```

When the command `TAF>gTAF->DSFProduction` is issued and `SubtrackNplanes` is non-zero, then the subtrack extrapolation and slopes are stored in the output `TTree`.
You can also check the behavior of the subtrack with the command:
`TAF>gTAF->GetRaw->DisplayCumulatedSubtracks( nnnn, p)`,
where `nnnn` is the number of events to analyze and `p` the DUT plane number to consider.

Implementation
The track finding based on iterative extrapolation is implemented in the method `find_tracks()` within the `DTracker.cxx` class. The computation of the track parameters is done in the `DTrack.cxx` class with the `Analyze` method.

14.4 Track-hit correlation analysis

The methods for the final analysis (listed in section 11) are defined in the `MCommands.cxx` file. All use common methods are implemented in the `MAnalysis.cxx` file. Note that both groups of methods belong to the same class `MimosaAnalysis`. Another class `MHist.cxx` defines all the histograms filled by the previously mentioned methods.
There are template `User` methods in the `MAnalysis.cxx` and `MHist.cxx` files, which are called from `MimosaPro`.

15 Developers' corner

This section intends to provide hints to modify the code itself. Once you feel confident, please consult the `TO DO LIST` in the `README` file to contribute.

IMPORTANT ADVISE AND REQUEST: avoid creating something (a method, an histogram, a configuration parameter...) which already exists somewhere, otherwise you are just making this code more messy and difficult to maintain and document. This documentation is certainly not exhaustive, so please in doubt ask us (baudot@in2p3.fr).

15.1 Commenting and documenting

Whenever you modify some lines of code, the following comments shall be added for the sake of bookkeeping and understanding what you have done.

- You may of course add some comments nearby the lines themselves, and terminate these lines with your initials and the date in the following format (`YYYY/MM/DD`).
- It is compulsory to add a line like:
  `Modified by [initials] [date] short explanations`
  in the beginning comments of the method, and obviously also to modify the explanatory text describing the method if its operation has been affected.
- To monitor which method has changed, add a line at the beginning of the class like `Last Modification [initials] [date] [method name]`.
- Finally, in the `README` file, describe the modification if you think it is important enough.

If you implement a new method, the following information are required.

- The most important, is the explanations on the new method, which shall appear as commented lines at the beginning of the implementation. Also indicate your name and the date.
- To monitor which method was added, add a line at the beginning of the class like `Last Modification [initials] [date] [method name]`.
- In the `README` file, describe what brings the method.
- And finally, edit and compile this file `doc/taf_shortDoc.tex` to add a paragraph or section describing what your method does.

It is useful to regenerate the `.html` files corresponding to all classes after your modification. For that, you need to compile within `ROOT`. Use in sequence, the two following scripts:

```
ROOT>.x Scripts/compiltaf.C
ROOT>.x Scripts/makeHtmlDoc.C
```

15.2 Adding a new class

- Write your class with a heritage from `TObject`.
- Write proper comments (general and for each method) to document your code!
- Add your class in one of the `LinkDef` files: either `code/include/DTLinkDef.h` or `code/include/MLinkDef.h`.
- Add your class for compilation in the `code/Makefile` and `Scripts/compiltaf.C` files.
- Add your class in the `Scripts/makeHtmlDoc.C` script to get automatically the `HTML` documentation.

15.3 Implementing a new sensor

- Check which decoder to use, you might need to create a new one, see the following subsection.
- The sensor will be associated with a `Readout` mode, `AnalysisMode`, `Mimosatype` and `Mapping`.
- if the `Readout` mode is a new one, the method `DPlane::Update()` has to be updated.

Note: However, the management of the `Mimosatype` is currently not robust.

15.4 Implementing a new `BoardReader` decoder

As underlined in section 5, the goal is to provide a list of pixels with an address and an associated value, from the binary files written by the acquisition system. Before creating some new reader, check that the existing ones do no fit your needs.
The steps below are specific to `BoardReader` classes, but of course you should also follows the recommendations in subsections 15.1 and 15.2.

- Write your class with a name ending with `BoardReader`. Get some inspiration from the existing classes. In principle, all parameters needed shall be passed through the arguments of the constructor. But additional parameters, optional ones for instance or list of input files, could be set with additional methods. There shall be at least one public method `HasData()`, which will trigger the retrieving of the next event. The other compulsory method is the one giving access to the list of pixels.
- Decide a type (integer format) to identify your board within other `TAF classes`. In December 2014, the last type is number 10.
- Edit the `DAcq.cxx` class where you shall introduce your new class at least in the constructor, the `SetDebug`, `NextEvent` and `PrintStatistics` methods. The `NextEvent` method is extremely important since this is where the list of pixels from your BoardReader are transferred to the list of `DPixel` objects which will be analyzed by the rest of `TAF` classes.

15.5 Adding a method of global interest

Global interest means the method could be used at different places in the code, like computing a χ^2 probability, or getting a path for files.
Add such algorithm as a method of the `DGlobalTools` class. In any other `TAF` class, there is usually a `fTool` object, an instance of `DGlobalTools`, which can use to call your method.

15.6 Changing the basic algorithms

Basic algorithms are the ones described in section 14. Their main behavior are driven by configuration parameters like `HitFinder` and `TracksFinder`. Additional configuration parameters are used to set useful search distances, resolutions, ...
If you intend to add new behavior to the already existing algorithm or a new algorithm, it is probably a matter of replicating an existing method (`DPlane::find_hits()` or `DTracker::find_tracks()`) and / or adding new options to them. Follow subsection 15.7 to add any new parameters you might need and update the documentation if existing parameters get new potential values.
PLEASE DOCUMENT (see 15.1), your additions, otherwise others will not be able to use your new smart algorithms.

15.7 Adding a configuration parameter

Before adding a new parameter, check that an existing one cannot help you first, see appendix 16.

- Choose a key name for your parameter to be represented in the configuration text file. Please do not choose a name already taken by another parameter (check from the `DSetup` class description).
- Create the corresponding variable as a data member of the `DSetup` class, including it in the best appropriated C-structures defined in the file `DSetup.h`. There are one such C-structure to store the list of parameters of the planes, trackers, acquisition, ...
- Edit the corresponding method in `DSetup.cxx`, to read this parameter from the configuration file.
- Edit the constructor (or potentially other method) of the class where you wish to use this parameter. In most of the classes, there is a pointer, `fc`, to the unique `DSetup` object. So you can easily initialize the variable in the final class from the configuration parameter.

15.8 Adding plots to the final analysis

Histograms are defined in the `MHist` class. You need to create a pointer in the `MHist.h` file and then book it in the function `BookingHistograms` of the `MHist.cxx` file. Please, try to place the definition of your new histograms nearby other related histograms.

Histograms are filled within the different methods of the `MAnalysis.cxx` file. All the available variables, related to pixels, hits or tracks, are data member of the class `MAnalysis` defined in the `MAnalysis.h` file. Look there first before defining a new variable.
It is expected that 4 methods are needed to perform all the operations required. Let's assume the new histograms can be refered under the name `XXXX`, then you can have:

`void XXXX_init` : to initialize any variables needed (declared in `MAnalysis.h`);
`void XXXX_compute` : to actually do the computation;
`void XXXX_fill` : to fill the histograms (created in `MHist`) with the results;
`void XXXX_end` : to finalize, for instance normalize histograms.

The display of histograms is done in the functions of the `MPost.cxx` file, which the user access through the menu displayed at the end of each final analysis methods like `MimosaPro()`. The histograms are also saved in the output file in these functions.

15.9 Adding plots to the event-by-event analysis

All display related to the event-by-event analysis is implemented in the `MRaw` class. Each functions there are independent and contains the booking and filling of their own histograms. So, simply change one function at a time.
If you add a new function, don't forget to add it to the menu defined in the `PrepareRaw()` function.

15.10 Adding a method for final analysis

The existing methods are listed in section 11. If you really need something new, it shall be one of the following:

a) add new histograms or specific analysis in an existing method `MimosaXXXX`;

b) a new general method `MimosaXXXX`.

a) addition to an existing method `MimosaXXXX`
You want to perform some specific computations and probably store them in some histograms. The best way is to follow the instructions in subsection 15.8, which explain how to modify the required files `MHist`, `MAnalysis` and `MPost`.
Once you are done, call the methods (you have just created in `MAnalysis`) inside the `MimosaXXXX` method you are using (file `MCommand.cxx`).

b) new method `MimosaXXXX`
This is a complete new final analysis. You will both need to create a new `MimosaXXXX` method in the `MCommands.cxx` file, and also to reuse methods from the `MAnalysis.cxx` file or create new ones as explained in **a)**.
What you do in this final step is basically an analysis of the `ROOT TTree` stored in the

datDSF/runNNNN_nn.root file. Get some inspiration from existing methods.

Note there is a mechanism to adjust the pop-up menu proposed by MPost. Indeed some displays do not make sense for some analysis, so options are displayed according to boolean flags of type fMimosaXXXXDone, which are declared in MAnalysis.h and initialized with false. So at the end of your new MimosaXXXX method, set the corresponding flag to true.

15.11 Adding new leafs in the output TTree

This is not yet permitted because there is no mechanism yet to insure backward compatibility with older TTree.

15.12 Adding a method for alignment

Still to be written...

16 Credits

It is a pleasure to write a few lines about all the people who have participated and are participating to the development of this package.
The early version in C++ was written in the 90's by Dirk Meyer at CERN within the RD42 collaboration to study diamond detectors with a silicon strip telescope. The code itself inherited from earlier FORTRAN versions developed in Strasbourg by **Renato Turcheta** and then **Farez Djama** for silicon strip sensors.

The C++ package was brought back in Strasbourg by **Christophe Suire** and **Jï¿½rï¿½me Baudot** in the late 90's for the characterization of silicon strip sensors within the STAR and ALICE collaborations. At the turn of the new century, the CMOS sensor group took over the code to study pixel detectors, still with the same silicon strip telescope. That was the creation of **MAF** by **Youri Gornushkin, Gilles Orazi, Auguste Besson, Damien Grandjean** and **Marc Winter**. Later on additional functionalities were brought by **Alexandre Shabetai, Rita De Masi** and Jï¿½rï¿½me Baudot.

The advent of telescope using pixel sensors as reference planes generated the need for **TAF** in 2005. Among the **TAF** contributors we find: Rita De Masi, **Loic Cousin, Serhyi Senuykov, Yorgos Voutsinas, Mario Bachaalany, Marie Gelin, Rhorry Gauld** (Oxford) and Jï¿½rï¿½me Baudot. Some of the raw data decoding methods have been provided in 2014 by Italian colleagues from INFN working in the ALICE collaboration, **Ilaria Aimo, Cristina Bedda, Paola La Rocca, Serena Mattiazio, Eleuterio Spiriti**, and by **Christian Finck** from IPHC.
In order to analyze data from Carbon ion beams, many additions were done by **Valerian Reithinger** (IPN Lyon) between 2012 and 2014.
Since 2014, **Alejandro Perez Perez** became a strong developer correcting and adding a lot of methods and plots. Specific additions to handle objects (called ladder in the code) made of several sensors were made by Loic Cousin. Methods to correct deformation of plane sensors were brought by **Benjamin Boitrelle**.

A reformatted version of the package, more compliant with C++ rules and cleaned of the heavy historical layers, has been re-written by Christian Finck and Regina Rescigno to serve as the reconstruction package of the vertex detector of the FIRST experiment and for the simulation and analysis of experiment with Carbon beams, [].

The various versions of this package would not have been successful in producing a load of published results if it had not benefited from the advises of semiconductor detector experts: **Wojtek Dulinski, Harris Kagan, Renato Turchetta** and **Marc Winter**.

References

[1] A. Besson, D. Greandjean and A. Shabetai, *MAF - Mimosa Analysis Framework Documentation*, June 2006, http://www.iphc.cnrs.fr/IMG/ps/mimosa_doc.ps.

[2] I. Rubinski, *An EUDET/AIDA Pixel Beam Telescope for Detector Development*, Phys.Procedia **37** (2012) 923?931, doi:10.1016/j.phpro.2012.02.434.

[3] R. Rescigno, *et al*, *Performance of the reconstruction algorithms of the FIRST experiment pixel sensors vertex detector*, Nucl.Instr.Meth. **A 767** (2014) 34?40, doi:10.1016/j.nima.2014.08.024.

[4] , R. Frï¿½hwirth, *et al.*, *Analysis of beam test data by global optimization methods*, Nucl.Instr.Meth. **A 732** (2013) 79?82, http://dx.doi.org/10.1016/j.nima.2013.05.038.

[5] G. McGoldrick, *et al.*, *Synchronized analysis of testbeam data with the Judith software*, Nucl.Instr.Meth. **A 765** (2014) 140?145, doi:10.1016/j.nima.2014.05.033.

Appendix A

List of parameters available for the configuration file

This list is a carbon copy of the comments in the `DSetup.cxx` source file.

```
// ##########################################################################
//                              Run Parameters
// ##########################################################################
// Affiliation      = [optional] (char) {""}
// Signature        = [optional] (char) {""}
// BeamTime         = [optional] (char) {""}
// Confidence       = [optional] (char) {""}
// DataPath         = [MANDATORY](char)      directory of the raw data
// DataSubDirPrefix = [optional] (char) {""} sub directory for data files, stored in
// Extension        = [optional] (char) {""}
// RunNumber        = [OBSOLETE]   RunNumber is get from DSession, set via gTAF->Init
// EventsInFile     = [MANDATORY for IMGBoardReader] (int) {0}
// StartIndex       = [MANDATORY for IMGBoardReader] (int) {0}
// EndIndex         = [MANDATORY for IMGBoardReader] (int) {0}
// NoiseRun         = [optional] (int) {0}  defines either the file with the noise fo
// --------------------------------------------------------------------------

// ##########################################################################
//                         Parameters of the Tracker
// ##########################################################################
// "Planes" or "Ladders" has to be the first field
// ---------------------------------
//     Planes and their parameters
// ---------------------------------
// Planes                   = [MANDATORY] (int)       the nb of planes in the setup
// Ladders                  = [optional]  (int) {0}   the nb of ladders in the setup (
// TimeLimit                = [optional]  (int) {0}   for sensor with timestamping, de
// Resolution               = [optional]  (float,um) {-1.} the knonw spatial resolutio
//
// ---------------------------------
//     Clustering in planes
// ---------------------------------
// HitsInPlaneMaximum       = [MANDATORY] (int)       the nb hits which will be recons
// KeepUnTrackedHitsBetw2evts =[optional]   (int) {0}
//                                                    1 memorise untracked hits between
//
// ---------------------------------
//     Tracking parameters
// ---------------------------------
// TracksMaximum            = [optional]  (int) {0}   the maximum nb of tracks which w
```

```
// TracksFinder             = [optional]  (int) {0}   select the tracking method
//                                                    0 for the original track finder
//                                                    1 for the one based on find_trac
//                                                    2 for the one adapted for Intera
// PlanesForTrackMinimum   = [MANDATORY if TracksMaximum!=0 && TracksFinder==0 ] (int
//                                                    the min nb hits required to make
// HitsNbMinForPreTrack    = [MANDATORY if TracksMaximum!=0 && TracksFinder==2 ] (int
//                                                    the min nb hits required to make
// HitsNbMinForFullTrack   = [MANDATORY if TracksMaximum!=0 && TracksFinder==2 ] (int
//                                                    the min nb hits required to make
// HitsInPlaneTrackMaximum = [MANDATORY if TracksMaximum!=0 && TracksFinder==0] (int)
//                                                    the max nb hits in a plane allow
// SearchHitDistance       = [MANDATORY if TracksMaximum<2] (float,um)
//                                                    the search distance to associate
// SearchMoreHitDistance   = [MANDATORY if TracksMaximum<2]   (float,um)
//                                                    the search distance to associate
// HitsDistanceForPreTrack = [MANDATORY if TracksMaximum==2]   (float,um)
//                                                    the search distance of hits to b
// HitsDistanceForFullTrack= [MANDATORY if TracksMaximum==2]   (float,um)
//                                                    the search distance between hits
// UseSlopeInTrackFinder   = [optional for TracksFinder=0]  (int) {1}
//                                                    use the track slope to extrapola
// TrackingPlaneOrderType  = [optional for TracksFinder=1]  (int) {0} Try to make a t
//                                                    0 in the same order than in the
//                                                    1 ordered with the status (0,1,2
// TrackingOrderLines      = [MANDATORY if TracksFinder=2] (int)
//                                                   number of lines like TrackingOrde
// TrackingOrderLineN      = [MANDATORY if TrackingOrderLines>=1]
//                            definition : TrackingOrderLineN:   [P#] {P1 P2 P3 ...}
//                               example :  TrackingOrderLine1:   [3] {5 6 7} [4] {1
//                                 means : There is [3] planes to build a pre-track
//                                                 and [4] planes to build the full-trac
// TrackingPass            = [MANDATORY if TracksFinder=2] (int)
//                                                  number of pass of the tracking al
// SubtrackNplanes         = [optional] {0}: if non-zero indicates that each track wi
// SubtrackPlanes          = [MANDATORY if SubtrackNplanes!=0] list of planes (separa
//
// ---------------------------------
//     Vertexing parameters
// ---------------------------------
// VertexMaximum           = [optional]  (int) {0}   0 : no vertexing
// VertexConstraint        = [optional]  (int) {0}   use a vertex constraint to start
//
// ---------------------------------
//     Alignement parameters
```

```
// ---------------------------------
// EventsForAlignmentFit   = [optional]  (int) {0}   the nb pairs (track-hit) for one

// ###############################################################################
//                          Parameters of the Ladder
// ###############################################################################
// "LadderID" has to be the first field
// LadderID            = [MANDATORY] (int), ID of the ladder
// LadderName          = [MANDATORY] (char), name AND type of the ladder, used to def
//                  could be "Plume", "Simple", "Salat", "Custom"
// Status              = [MANDATORY] (int), same meaning as for Plane,
// LadderPositionX/Y/Z = [MANDATORY] (float,mm), same meaning as for Plane,
// LadderTiltX/Y/Z     = [MANDATORY] (float,deg), same meaning as for Plane,
// NbOfPlanes          = [MANDATORY] (int), nb of planes associated to this ladder
// If using "Custom", you need to define each planes associated
//   with the following lines
//  IncludedPlane   = [MANDATORY if "Custom"] (int) the plane number as defined later
//  PlaneShiftU,V,W = [MANDATORY if "Custom"] (float,mm) relative shifts of this plan
//  PlaneTiltU,V,W  = [MANDATORY if "Custom"] (float,deg) relative tilts of this plan
//
// NOTE:
//  The positions/tilts of planes belonging to a ladder are defined
//   by the ladder definition. Any position settings of these planes
//   in the configuration files are superseeded.
//


// ###############################################################################
//                    Parameters for each detector plane
// ###############################################################################
// "Inputs" has to be the first field
//
// ---------------------------------
//     Data decoding
// ---------------------------------
// Inputs          = [MANDATORY] (int), 1st parameter = number of inputs needed to bu
//    -> for each input, specify in the following order:
//      ModuleType    = [MANDATORY] (int) index for the desired type of modules
//                      according to the declaration order in Acquisition module
//      ModuleNumber  = [MANDATORY] (int) id in the set of modules of this type
//      InputNumber   = [MANDATORY] (int) id of the input of this module used
//      ChannelNumber = [MANDATORY] (int) channel shift so that
//                      plane_channel_nb = input_channel_nb + ChannelNumber
//      ChannelOffset = [optional] (int) {1} first channel in the input related
//                       to the plane
```

```
//      Channels      = [MANDATORY] (int) number of channels to read from this input
// StripselUse     = [obsolete] used to define dead strips
//
// ---------------------------------
//     Readout and Analysis of pixels
// ---------------------------------
// Name            = [MANDATORY] (char) just for display for now
// Purppose        = [MANDATORY] (char) just for display for now
// ParentLadder      = [optional] (int) indicate if the plane belong to a ladder
//                                  and which one
// Readout         = [MANDATORY] (int) specifies the type of raw data
//                   0 -> not read,
//                   1<Readout<100 -> data not sparsified,
//                   100<Readout -> sparsified data.
// AnalysisMode    = [MANDATORY] (int) controls the analysis
//                   0 -> data read but no clustering,
//                   1 -> strips,
//                   2 -> pixels with analog output,
//                   3 -> pixels with binary output.
// MimosaType      = [MANDATORY] (int) not clear (sorry!)
// InitialPedestal = [obsolete] (int) superseded by InitialNoise
//                   # events to analyze before the first computation
// InitialNoise    = [MANDATORY only if Readout<100] (int)
//                   # events to analyze before the first computation
// CacheSize       = [MANDATORY only if Readout<100] (int)
//                   size of the set of events from which one is picked
//                   for computing the noise and pedestal
// HotPixelMapFile = [optional] (char) ROOT file name with fake rate map
//   FakeRateCut     = [MANDATORY if HotPixelMapFile] (float) Single pixel fake rate
// IfDigitize      = [optional] (int) {0} # thresholds to emulate the digitization
//                   0 (default) means no-digitization
//     DigitizeThresholds = [MANDATORY if IfDigitize>0] (array of int)
//                          as many values as indicated, separated with ':'
//
// ---------------------------------
//     Position and size
// ---------------------------------
// PositionsXYZ    = [MANDATORY] (3 float,mm) position of the center of the plane,
//                    changed by the alignment procedure
// TiltZYX         = [MANDATORY] (3 float,deg)
//                    rotation angles defining plane orientation,
//                    changed by the alignment procedure
// PitchUVW        = [MANDATORY] (3 float,um) pitch in all directions
//                   (pitchW=sensitive layer thickness, not used yet)
// StripsUVW       = [MANDATORY] (3 int) # collecting noted in all directions
```

```
// StripSizeUVW    = [obsolete] computed from PitchUVW and StripsUVW
//                    size of the sensitive area, set to PitchUVW if not provided
// StripSizeUVW    = [optional] (3 float,um) not used yet
// AlignementU/V   = [obsolete] (float,mm) old alignment shift parameter
// AlignementTilt  = [obsolete] (float,deg) old alignment angle parameter
// Deformed        = [optional] (int) {0} if 0 then no deformation applied,
//                     if 1 then deformation applied from following coeff.
//  coeffLegendreU = [MANDATORY if Deformed==1] (6 floats) values (separated by ":")
//                     coeff of the 6 first Legendre polynomials
//                    used to parametrize delat_W with respect to U coordinate
//  coeffLegendreV = [MANDATORY if Deformed==1] (6 floats) 6 values (separated by ":"
//                     coeff of the 6 first Legendre polynomials
//                    used to parametrize delat_W with respect to V coordinate
// Mapping         = [MANDATORY] (int) drives pixel position computation,
//                   1 -> position at pixel center,
//                   2 -> position considers staggering
//                   3,4,5 -> ?
// Status            = [MANDATORY] controls how this plane is used by the tracking
//                 0 = Primary Reference, never aligned and used as track seed,
//                 1 = Primary Reference, never aligned and used in tracking (not for
//                 2 = Secondary Reference, aligned and used in tracking (not for see
//                 3 = Device Under Test (DUT), aligned but never used in tracking
//
// ----------------------------------
//     Cluster (=Hit) finder
// ----------------------------------
// HitFinder       = [optional] (int) {0} select the hit finder method,
//                   0 -> standard
//                   1 -> connected pixel
//                   2 -> cog based with search radius, requires additional parameter
// ThreshNeighbourSN = [MANDATORY] (float) S/N or S cut on all the pixels
//                     (seed excluded) in the cluster for the hit finding
// ThreshSeedSN      = [MANDATORY] (float) S/N or S cut on the seed pixel
//                     for the hit finding
// MaxNStrips        = [optional] (int) {1000} maximal #strips allowed in cluster
//                     (corrected or set automaticaly by DCut depending on HitFinder)
// MinNStrips        = [optional] (int) {1} minimal #strips required in cluster
//                     (corrected or set automaticaly by DCut depending on HitFinder)
// ClusterLimitU     = [MANDATORY if HitFinder!=2] (float,um) maximal distance
//                     between the seed pixel and any other pixel in the hit
// ClusterLimitRadius= [MANDATORY if HitFinder==2] (float,um) maximal distance
//                     between the center of gravity and any other pixel in the hit
// CommonRegions     = [optional] (int) {1} # regions to define
//                     for the common noise shift computation per event
// Position Algorithm= [MANDATORY] (int) controls how the hit position
```

```
//                      is estimated from the pixels info
//                  1 = Center of Gravity,
//                  2 = eta,
//                  3 = kappa (not implemented yet)
// PlaneResolution   = [optional] (float,um) expected plane resolution in both direct
// PlaneResolutionU  = [optional] (float,um) expected plane resolution in U direction
// PlaneResolutionV  = [optional] (float,um) expected plane resolution in V direction
// ResolutionFile    = [optional] (char) not implemented yet
// ResolutionRegion  = [optional] (?) not implemented yet
//


// ###############################################################################
//                    Parameters for DAQ
// ###############################################################################
// "AcqModuleTypes" or "FileHeaderSize" has to be the first field
// AcqModuleTypes     = [MANDATORY] (int) # different boards used in the DAQ
// TriggerMode        = [optional] (int) method to deal with trigger info
//                      0 -> ignore trigger info
//                      1 -> each new trigger generates an event
//                      2, 3 -> ?
// EventBuildingMode  = [obsolote] (int) {0} replaced by EventBuildingBoardMode
// BinaryCoding       = [optional] (int) {0} 0 for one Endian, 1 for the other
// FileHeaderSize     = [obsolete] (int) {0} size of additional header file
// EventBufferSize    = [optional] (int) event size in Bytes
// FileHeaderLine     = [optional] (int) event header size in Bytes


// ###############################################################################
//                    Parameters for each Acquisition module type
// ###############################################################################
// "Name" has to be the first field
// Name                  = [MANDATORY] (char) generic name of such modules
//                        known types: "IMG", "TNT", "PXI", "PXIe", "GIG", "VME"
//                                     "DecoderM18", "ALI22", "DecoderGeant"
// Type                  = [MANDATORY] (int) unique identifier for the module type
// Devices               = [MANDATORY] (int) # module instances of this type,
//                        typically, one instance decode one file
// Inputs                = [optional] (int) # identical data block (one per sensor ty
// Channels              = [optional] (int) # channels in one input (data block)
// EventBuildingBoardMode= [optional] (int) used to pass a flag
// Bits                  = [optional] (int) size in bits of words to read from file
// SignificantBits       = [optional] (int) # significant bits per word
// FirstTriggerChannel   = [optional] (int)
// LastTriggerChannel    = [optional] (int)
```

```
// NbOfFramesPerChannel  = [optional] (int) # frames stored for each trigger/event
// DataFile or DataFile1 = [optional] (char) core name of data file
//
// --- Name: "IMG"
//  Type = 10 for pixels / 11 for strips / 12 for pixels with multi-frame
//         13 for pixels with
//  Inputs             = [MANDATORY] (int) # identical data block (one per sensor typ
//  EventsInFile       = [MANDATORY] (int) expeted # events in a file
//  StartIndex         = [MANDATORY] (int) index of first data file
//  EndIndex           = [MANDATORY] (int) index of last data file
//  Extension          = [MANDATORY] (char) {"rz"} extension of data file
//  TriggerMode        = [MANDATORY] (int) method to deal with trigger info
//  EventBuildingBoardMode or EventBuildingMode = [MANDATORY]
//  EventBufferSize    = [MANDATORY] (int)
//  FileHeaderLine     = [MANDATORY] (int)
//  Bits: [MANDATORY] (int) if negative
//  SignificantBits: [MANDATORY] (int) if negative
//  DataFile: [MANDATORY] (char) typically "RUN_32844_"
//
// --- Name: "TNT"
//  Type = 30 or 31
//  StartIndex         = [MANDATORY] (int) index of first data file
//  EndIndex           = [MANDATORY] (int) index of last data file
//  Extension          = [MANDATORY] (char) {"rz"} extension of data file
//  TimeLimit          = [MANDATORY] (int)
//  EventBufferSize    = [MANDATORY] (int)
//  TriggerMode        = [MANDATORY] (int) method to deal with trigger info
//  BinaryCoding       = [MANDATORY] (int)
//  Bits               = [MANDATORY] (int)
//  SignificantBits    = [MANDATORY] (int)
//  DataFile           = [MANDATORY] (char) typically "CardXXXX_000"
//  TriggerMode ->unused
//  EventBuildingBoardMode -> unused
//  FirstTriggerChannel, LastTriggerChannel, NbOfFramesPerChannel -> unused
//
// --- Name: "PXI"
//  Type = 40
//  TriggerMode        = [MANDATORY] (int) method to deal with trigger info
//  BinaryCoding       = [MANDATORY] (int)
//  DataFile           = [MANDATORY] typically "run_XXXX_"
//  EventBufferSize, FileHeaderLine -> unused
//  Channels, Bits, SignificantBits, EventBuildingBoardMode -> unused
//  FirstTriggerChannel, LastTriggerChannel, NbOfFramesPerChannel -> unused
//
// --- Name: "PXIe"
```

```
//  Type = 50
//  TriggerMode        = [MANDATORY] (int) method to deal with trigger info
//  BinaryCoding       = [MANDATORY] (int)
//  EventBuildingBoardMode
//  EventBufferSize, FileHeaderLine -> unused
//  Channels, Bits, SignificantBits -> unused
//  FirstTriggerChannel, LastTriggerChannel, NbOfFramesPerChannel, DataFile -> unused
//
// --- Name: "GIG"
//  Type = 60
//  TriggerMode, BinaryCoding, EventBufferSize, FileHeaderLine -> unused
//  Channels, Bits, SignificantBits, EventBuildingBoardMode -> unused
//  FirstTriggerChannel, LastTriggerChannel NbOfFramesPerChannel, DataFile -> unused
//
// --- Name: "VME"
//  Type = 70
//  Extension        = [MANDATORY] (char) {"rz"} extension of data file
//  Inputs           = [MANDATORY] (int) # identical data block (one per sensor typic
//  DataFile         = [MANDATORY] (char) typically "FIFOdata_M28_RUN3_ch"
//  TriggerMode, BinaryCoding, EventBufferSize, FileHeaderLine -> unused
//  Channels, Bits, SignificantBits, EventBuildingBoardMode -> unused
//  FirstTriggerChannel, LastTriggerChannel NbOfFramesPerChannel -> unused
//
// --- Name: "ALI22"
//  Type = 80
//  DataFile         = [MANDATORY] (char) typically "FIFOdata_M22"
//  NbOfFramesPerChannel = [MANDATORY] (int) # frames stored for each trigger/event
//  TriggerMode, BinaryCoding, EventBufferSize, FileHeaderLine -> unused
//  Channels, Bits, SignificantBits, EventBuildingBoardMode -> unused
// FirstTriggerChannel, LastTriggerChannel, NbOfFramesPerChannel -> unused
//
// --- Name: "DecoderM18"
//  Type = 90
//  Extension       = [MANDATORY] (char) {"dat"} extension of data file
//  DataFile        = [MANDATORY] (char) typically "SIS3301dataZS_ch"
//  TriggerMode, BinaryCoding, EventBufferSize, FileHeaderLine -> unused
//  Channels, Bits, SignificantBits, EventBuildingBoardMode -> unused
// FirstTriggerChannel, LastTriggerChannel, NbOfFramesPerChannel -> unused
//
// --- Name: "DecoderGeant"
//  Type = 100
//  DataFile        = [MANDATORY] (char) typically "FILEdata_Geant_RUN1_ch"
//  TriggerMode, BinaryCoding, EventBufferSize, FileHeaderLine -> unused
//  Inputs, Channels, Bits, SignificantBits, EventBuildingBoardMode -> unused
//  FirstTriggerChannel, LastTriggerChannelNbOfFramesPerChannel -> unused
```

```
//


// ##########################################################################
//                      Parameter for Final Analysis
// ##########################################################################
// "AnalysisGoal" or "MaxNbOfHits" or "StatisticCells" or "CmsNoiseCut" has to be the
// AnalysisGoal     = [optional] (char) {""} drives type of histograms created, could
//              cluster, track, calib, laser, vertex, fake, vector
// StatisticCells   = [obsolete]
// CmsNoiseCut      = [optional] (float) {3} upper SNR cut to use a pixel value in th
// MaxNbOfHits      = [MANDATORY] (int) max # hits allowed to consider events for ana
// MinNbOfHits      = [MANDATORY] (int) min # hits allowed to consider events for ana
// TrackChi2Limit   = [MANDATORY] (float) upper chi2 cut to consider a track for anal
// MinHitsPerTrack  = [optional] (int) minimum #hits per track (default is set with P
// MaxTracksExGeom  = [optional] (int) {-1} inclusive max # tracks allowed in the ExG
// ExGeomatrix      = [optional] (int) {0} geomatrix (of submatrix 0) to be used in t
// UserFlag         = [optional] (int) whatever you want to use in User's stuff
// HistoChargeRange = [optional] (float) {5000} max charge displayed in histo
// HistoSNRRange    = [optional] (float) {250} max SNR displayed in histo
// HistoNoiseRange  = [optional] (int) {40} max noise displayed in histo
// Submatrices      = [MANDATORY] (int)  # submatrices defined below, at least 1 shal
//
// Then for each submatrix declared
//   PixelSizeU     = [MANDATORY] (float)
//   PixelSizeV     = [MANDATORY] (float)
//   PixelsInRaw    = [MANDATORY] (int)
//   PixelsInColumn = [MANDATORY] (int)
//   Matrixtype     = [optional] (int) {1} defines the mapping pixel-position
//                1 = orthogonal pixel network,
//                2 = staggered pixel network for elongated pixel,
//                3,4,5 = ?
//   MaxNofPixelsInCluster = [optional] (int) {0} maximum # pixel allowed to consider
//   MinNofPixelsInCluster = [optional] (int) {1} minimum # pixel allowed to consider
//   MinSeedCharge         = [optional] (int) {-1000} minimal charge on the seed pixe
//   MinClusterCharge      = [optional] (int) {-1000} minimal total charge to conside
//   MinNeighbourCharge    = [optional] (int) {-1000} minimal charge on pixels neighb
//   MinSeedIndex          = [optional] (int) {0} defines limit index of pixels to ta
//   MaxSeedIndex          = [optional] (int) {0} defines limit index of pixels to ta
//   MinSeedCol            = [optional] (int) {0} defines limit index of col to take
//   MaxSeedCol            = [optional] (int) {0} defines limit index of col to take
//   MinSeedRow            = [optional] (int) {0} defines limit index of row to take
//   MaxSeedRow            = [optional] (int) {0} defines limit index of row to take
//   Calibration           = [optional] (float) {1.} conversion factor between ADCuni
//   NoiseScope            = [MANDATORY] (float) noise multiplication factor, if 0. -
```

```
//   4 geomatrix shall be specified, they define 4 ROI (region of interest)
//     GeoMatrix0/1/2/3 = [MANDATORY] (float,um)  minU: maxU: minV: maxV
//
```