ActRoot manual

Miguel Lozano González

Contents

1 Introduction

ActRoot aims at being a simple program to convert, process, and obtain the physical data for an ACTAR TPC experiment. It follows the structure of *NPTools*[1], creating different *detector* classes that perform custom tasks on each component of the experiment. A range of different data classes stores meaningful information at all stages of the analysis.

The following picture depicts a simplified scheme of the code.

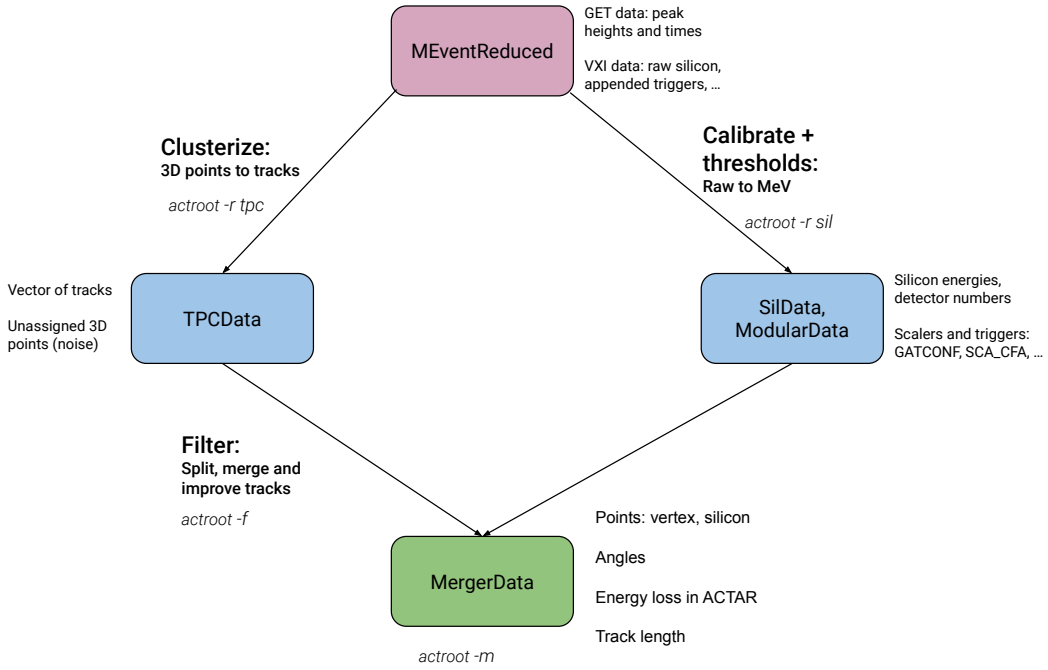

Figure 1: Actions tree of the analysis code.

A brief description of each element is presented in the following lines.

1. `MEventReduced`: minimal processed data. It merges the information from the pad plane (the GET electronics) and the silicons (VXI), appending the external variables such as the scalers or triggers. This is our source data and it is further processed in two separate ways:
   - **Cluster**: Handles the TPC data by merging the cloud of 3D points arising from the GET data to form groups of points (*clusters*) that indeed constitute **tracks**.
   - **Calibrate**: In charge of calibrating the silicon data and applying thresholds. It also reads the desired trigger variables.

2. `TPCData`: Contains the vector of tracks result of the cluster process. It also stores points not assigned to any cluster. Afterwards:
    - **Filter**: Complex action that transforms the vector of clusters to end up with only 2 or 3 that can be subsequently identified with the components of a **binary reaction**. Depending on the nature of the experiment, several actions might be needed to fulfill this goal: merging broken clusters, splitting artificially merged tracks, erasing noisy clusters (**contamination**, i.e., delta electrons), etc.
3. `SilData`, `ModularData`: Just a collection of silicon energies, detector index, and other useful data per event. By *modular* we understand variables such as the `GATCONF` (which unambiguously tells the detectors that triggered the acquisition) or the `SCA_CFA` (that provides the number of beam particles in the CFA detector). Both classes do not require further processing.
4. `MergerData`: Merges the three aforementioned classes to build a physical event. Given a set of conditions, it determines whether the event is good or not (provided the track or silicon multiplicity, the existence of a vertex, `GATCONF` value, etc) and saves the following information:
    - Angles of recoils with respect to the beam.
    - Energy at the silicon and the detector index.
    - Average energy loss in the pad plane, which is of paramount importance for the PID.
    - Track length. Also key to account for energy losses in the gas.
    - The run and entry number, to unequivocally identify the event later.
    - Other variables with debug purposes.

One of the main goals of the code is to remain versatile: by using configuration files, the actions can be independently modified, permitting the analysis of different experiments without modifying the code. This is achieved by using a wide range of *.conf* file, as follows:

- *detector.conf*: Configures general settings of the detectors: size of ACTAR pad plane, algorithms to be used, action files, and the merger conditions, so to speak.
- *calibration.conf*: Calibrations to be used in the silicon energy conversion but also the look-up table for the pad plane (channel to $<x, y>$) and the gain matching parameters.
- Since there are different algorithms available to perform the *Cluster* and *Filter* actions, each of them reads a separate file:
    - For the clustering: *ransac.conf* or *climb.conf*. Essentially, they set the minimum number of points to form a track.
    - For the filtering: *multiregion.conf* and *multistep.conf*. This step is more complex. An attempt to explain these algorithms is given in section 2.

  The choice of algorithm is done in the *detector.conf* file.
- *data.conf*: Sets the vector of runs to analyze and the paths to read and store the `.root` files.

All these files must lay under a `configs/` directory in what will be called a **project** folder: the parent directory for all the analysis. It is in this folder that we can run the executables provided by the code, as noted in Figure 1 (`actroot -flag OPTIONS`).

The goal of this manual is to provide a quick insight into the different algorithm settings for the *Filtering* part.

2 Algorithms

Two different types of algorithms to deal with voxels exist in the *Algorithm* folder of *ActRoot*:

- Inheriting from `VCluster`, they perform the basic clustering operation: joining voxels to form sets of voxels, i.e., clusters. Two have been implemented:
  - **RANSAC**: randomly samples two points from the cloud to form a line. Points close to it by a given distance threshold constitute the cluster. The process is repeated N times and only the *best-ranked* clusters are kept.
  - **Climb**: clusters points based on the continuity principle: neighbor points *should* belong to the same cluster. It was developed by J. Lois-Fuentes during his PhD [2], thus for more information see his thesis.

  Both can be configured in the `ransac.conf` or `climb.conf` files, respectively.
- From `VFilter`, this set of classes performs the treatment of the clusters, meaning this:
  - Cleaning noise
  - Merging broken clusters
  - Splitting clusters
  - Finding the **reaction point**
  - Improving the quality of the line fits

  Two are available now: `MultiStep` and `MultiRegion`. The former was employed for J. Lois experiment E796, while the latter is an *easier* version.

  Both leverage common functions that execute particular tasks. It is to this that we are going to explain in detail its workflow.

2.1 Filtering algorithms

Following the `VFilter` class, the shared structure for all the filtering classes is:

1. A pointer to the `TPCData` has to be set by `void VFilter::SetTPVData(TPCData* data)`
2. The algorithm is executed by `void VFilter::Run()`. This virtual function (i.e., mandatory to be implemented in any derived class) **sets the order of the inner functions**.
3. Other methods are available depending on the derived class.

2.1.1 Common tasks

There are a few steps that are usually needed by both classes. Hence, they are implemented in a *global* fashion under `ActAlgoFuncs.h`, `.cxx`.

These are:

- **Merging similar clusters**: a cluster might be broken into smaller parts as a result of charge collection inefficacies or damaged pads. Therefore, it is essential to remerge them. This function iterates over the cluster vector and, by pairs, joins clusters if some conditions are met:
  1. The distance between the central points of the two clusters is bellow `MergeDistThresh`.
  2. Both are quite parallel: their scalar product of fits is above `MergeMinParallel` $\in [0, 1]$.
  3. If 1 **and** 2 are met, the combined fit is done.

4. If, and only if, the $\chi^2_{new} \leq$ MergeChi2Factor $\cdot \chi^2_{old}$, the larger cluster is merged into the smaller one and deleted. The scaling factor is usually ~ 1.5.

- **Compute a reaction point (RP) in 3D**: In general two lines in 3D do not perfectly intersect. On the contrary, the goal is to find the minimum approach distance and the two points on each line at which this occurs.

  This function is schematically defined as:

  ```
  std::tuple<Point on A, Point on B, double min dist>
  ComputeRPIn3D(Point A, Vector A, Point B, Vector B);
  ```

  Specifying the points and the direction vectors of each line, one is given the two points and the distance between them. The RP is afterward calculated as the mean.

- **Cylinder cleaning** of a cluster: Some tracks may present *transversal* anomalies that worsen the 3D linear fit. To delete them, a cylinder centered around the fit is defined and points outside it are tagged to be erased. This threshold, given by the radius of the cylinder, is known as `CleanCylinderR`.

  Before deleting the voxels, a **check** is executed to determine whether the cluster will have enough remaining points. If not, it is kept unmodified. This threshold in the number of points is usually linked to the clustering algorithm by the function `VCluster::GetMinPoints()`.

- **Separate recoil from beam**: In most reactions, there is a heavy recoil whose angle is so small that it remains undistinguishable from the beam particle. In these cases, the light particle is used to find the RP and, subsequently, this can be used to separate the beam from the secondary track. The procedure is as follows:

  1. In the vector of clusters, find the **beam-like**.
  2. Split its voxels according to the X value: lower or greater than the RP.
  3. If enough voxels to form a new cluster (`VFilter::GetMinPoints()`), erase.
  4. Refit the remaining cluster (and the new one if it is the case).

  The flag `RPKeepSplit == false` allows to **delete** newly-formed cluster.

References

[1] NPTools team. *NPTools*. Version 3. URL: https://nptool.in2p3.fr/.

[2] Juan Lois Fuentes. "Complete spectroscopy of 16C and 20O with solid and active targets using transfer reactions". Available at http://hdl.handle.net/10347/30947. PhD thesis. Universidade de Santiago de Compostela, July 2023.