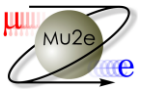

Caltech

Calibration of the Mu2e Calorimeter using Cosmic Ray Events

Italian Summer Student Program 2024

Intern

Giacinto Boccia
Università degli Studi di Napoli Federico II
giacinto.boccia@outlook.it

Supervisors

Robert Kutschke
Fermi National Accelerator Laboratory
kutschke@fnal.gov

Simona Giovannella
INFN - Laboratori Nazionali di Frascati
simona.giovannella@lnf.infn.it

Abstract

As the calorimeter developed for the Mu2e experiment at Fermilab is being completed, a Vertical Slice Test is performed to assess the performance of the entire acquisition chain, refine the calibration procedure and analyze the calorimeter response and resolution. In the framework of this test, 58 crystals from one of the two calorimeter disks have been instrumented, acquiring real cosmic ray data for about 2 hours.

During my internship with the “Italian Summer Student Program”, I developed different programs: the first to display the collected events on the calorimeter disk, the second one to optimize the fitting window of the digitized signals, and the third one to use cosmic rays to calibrate the time response of the read-out channels. The programs have been written in Python making use of CERN’s ROOT data analysis framework, and of parallel execution, taking advantage of the capabilities of Fermilab’s Elastic Analysis Facility.

In this report, I provide a brief overview of the Mu2e Experiment and of its calorimeters; and then describe each of my three project tasks, the approach used to fulfill them, and the results achieved.

All the programs developed during this internship are available on a fork of one of the project’s GitHub repositories: https://github.com/Gianfilippo980/CaloCalibration-VST.

Contents

The Mu2e Experiment

Mu2e aims at measuring or putting an upper limit to the flavor-violating decay of the muon. This process is allowed by the Standard Model of Particle Physics with a branching ratio of 10^{-54}, but beyond the Standard Model theories predict a much higher branching ratio; therefore, Mu2e aims to put an upper limit of about $8 \cdot 10^{-17}$ at a Confidence Level of 90%, if the event is not detected.

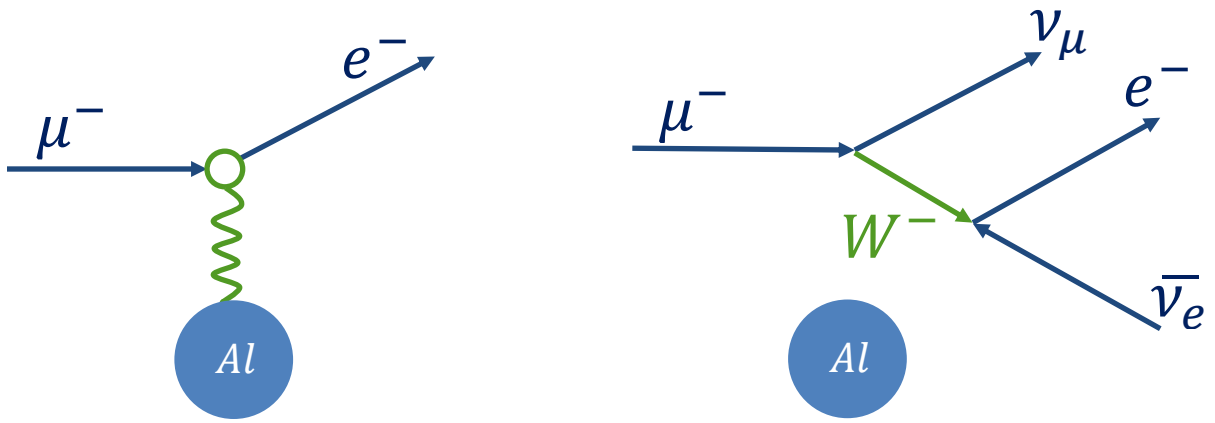

Figure 1. *On the left the flavor violating decay of the muon as it would be observed by Mu2e, on the right the more common process, where the leptonic family numbers are preserved, it will constitute one of Mu2e's backgrounds.*

To distinguish the flavor-violating process from other channels (see **Figure 1**), Mu2e will study the energy of the emitted electrons: for muons that decay while bound in orbit on an aluminum atom, such that the CMS for the decay is the lab reference frame, the kinematics of the decay ensures that:

- Electrons coming from the flavor violating decay of the muon (conversion electrons) appear as a mono-energetic line at $104.9\ MeV$.
- Electrons coming from the regular decay of the muon, but not bound on the Aluminum nucleus, have an energy spectrum up to a maximum of $53\ MeV$.
- Electrons decaying when bound in the nucleus undergo an exchange of momentum with the nucleus that creates recoil tails of up to the same $104.9\ MeV$ energy, with a very small probability. The chance of this phenomenon can be theoretically calculated.

Therefore, Mu2e needs to prepare muonic aluminum atoms and to measure the energy and momentum of the decay electrons.

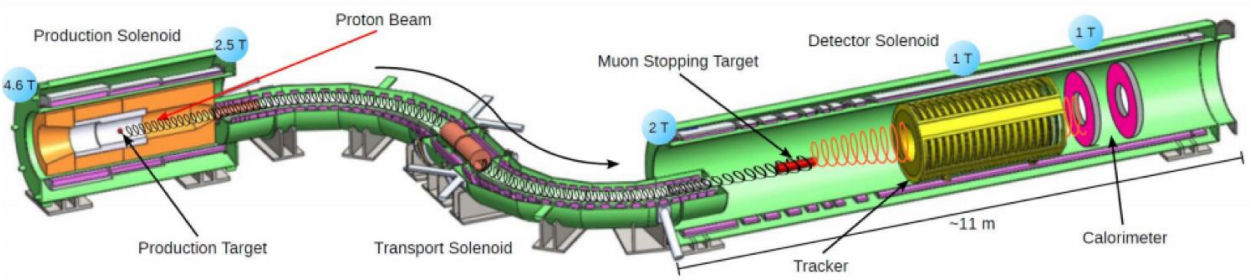

Figure 2. *The general layout of Mu2e, the experiment consists of three superconducting solenoids, producing a graded magnetic field; the first contains the production target, where $\pi^{\pm}$ are produced (and quickly decay); the second selects and transports μ^- to the third, where the muonic atoms are formed and the emitted electrons are studied with a spectrometer and calorimeter.*

Firstly, the 8 GeV pulsed proton beam from the Main Injector and Recycler Ring is directed at the production target, made of Tungsten, which is suspended by thin wires inside the production solenoid. This interaction produces hadrons and Mu2e collects $\pi^{\pm}$ and $K^{\pm}$ emitted backwards (with respect to the proton beam direction). The emitted mesons travel through the s-shaped transport solenoid and decay in $\mu^{\pm}$,; here collimators are designed to select the μ^{-} and direct them to the detector solenoid. The detector solenoid holds a stopping target, made of thin Aluminum disks suspended by wires, where the muons stop and bound forming muonic atoms; traveling further in the detector solenoid, the decay electrons cross the straw tubes-based tracker and two calorimeter disks.

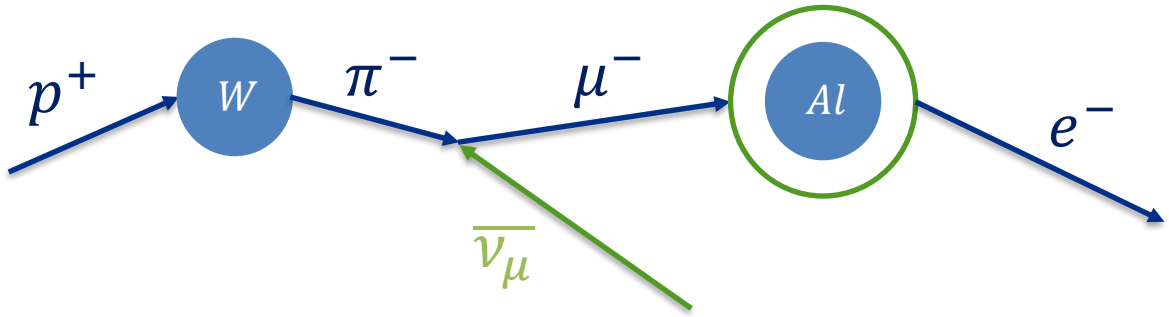

Figure 3. *Schematic 'path' of a signal particle form the original proton to the decay electron.*

Several measures are taken to reduce the backgrounds; among these, the trigger is only enabled after the end of the proton beam pulses with a delay calibrated for the expected life of the muonic atom. This ensures that most other prompt particles, especially the pions, that could constitute a significant noise have already decayed and/or travelled through the detectors when the experiment acquires data. The detectors do not cover the region closest to the solenoid axis, as the signal in those regions would mostly detect noise, while the signal electrons travel in wide spirals in the magnetic field of the Detector Solenoid.

The Calorimeter

Structure

The calorimeter designed for the Mu2e experiment consists of two identical disks, each containing 674 CsI crystals measuring $34.4 \times 34.4 \times 200\ mm^3$. Each of these crystals is read by two SiPMs whose electric signals reach digitizing boards capable of processing up to 20 channels each with a sampling period of $5\ ns$.

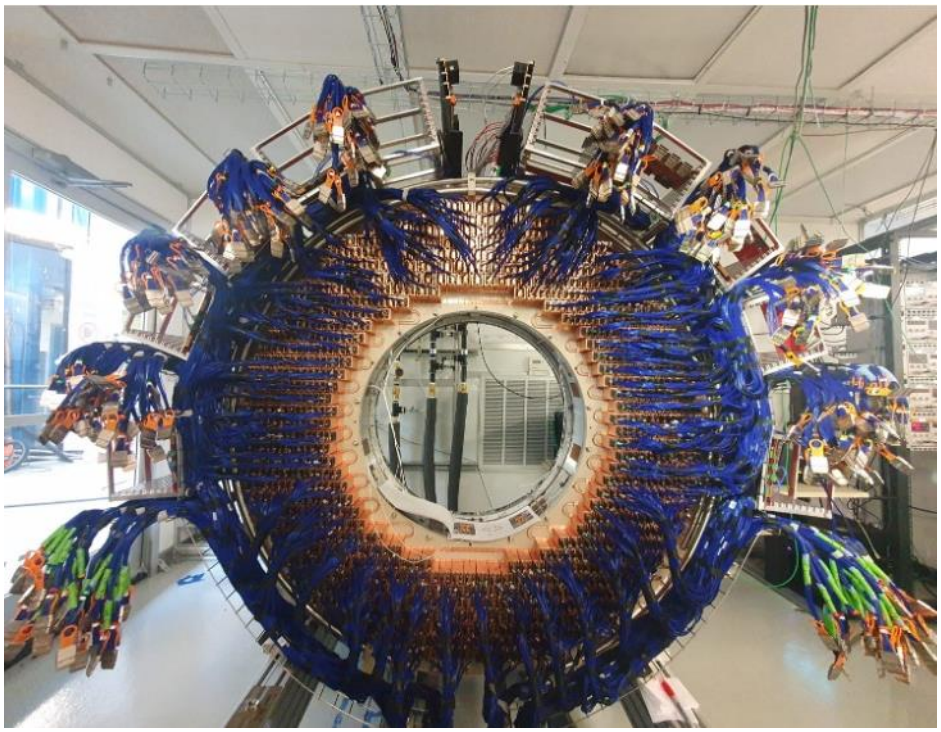

Figure 4. *Back view of one of the calorimeter disks, each of the copper boxes shields and helps cooling two SiPMs with their little boards, while digitizing boards are to be placed in the racks that surround the disk.*

To calibrate the energy and time response of the calorimeter, both during the commissioning phase and once installed in the Mu2e apparatus, three main systems are used:

- The light of a green laser system is fitted to each calorimeter crystal via optical fibers, allowing to inject a known light signal.
- An activated fluid (Fluorinert) can be circulated in a tubing system that covers the frontal face of each disk.
- Cosmic rays traveling through the calorimeter disks will release energies whose distribution is well known, as well as their travel speed.

During my internship, my work focused on the time calibration of the calorimeter using cosmic ray events.

Vertical Slice Test

The Vertical Slice Test of the calorimeter, done in July 2024, consisted of reading out the full chain of 58 instrumented crystals from one of the two disks, allowing for the collection of about 2 hours of real cosmic data. This allowed us to test the intended calibration algorithms, confirming that they work as expected in simulation on the data from the real hardware. At this stage, as the acquired data is not yet reconstructed to the final production format, ad-hoc computer programs are needed to reconstruct them. In particular, Binary files produced by otsDAQ are unpacked with C++ routines and transformed to ROOT Trees.

Note that the crystals used during the test were split into three groups in the right half of the disk, spaced to facilitate the recognition and selection of cosmic rays.

Cosmic Event Display Program

The first part of my project was to develop a practical visualization program for the cosmic events recorded. This was achieved with a Python script that makes extensive use of PyROOT to perform the analysis and produce the drawings (as in **Figure 6**). A different script is responsible for the simple graphical user interface.

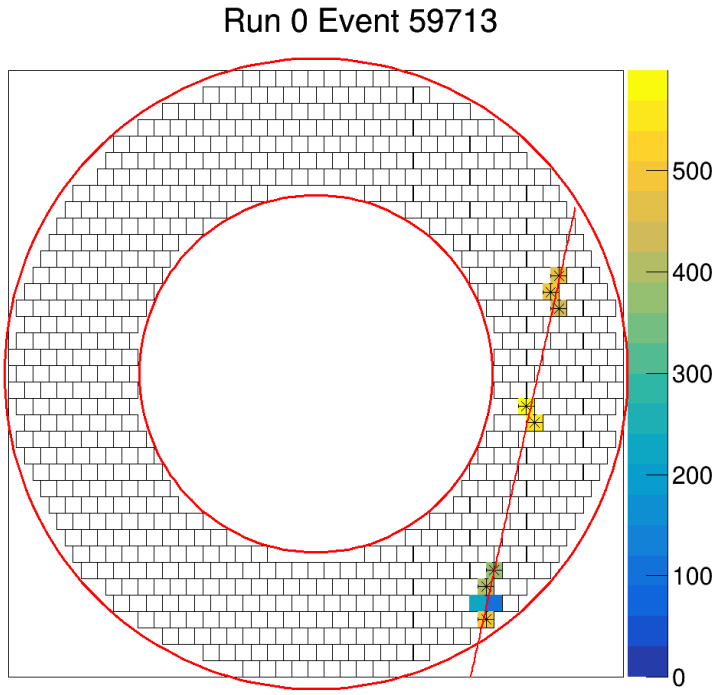

***Figure 6**. An example of the graphs produced by the Event Display for an event, showing a frontal view of the calorimeter, each black box represents a crystal, they are colored with the peak signal detected, averaged between the two SiPMs; hits above the threshold are marked with an asterisk and fitted with a straight line (in red).*

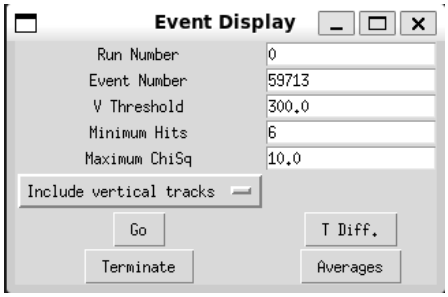

***Figure 5.** The main graphical user interface window, where the user can read and set the thresholds as well as the number of the event currently displayed. Pressing "Go" will show the next event matching the cuts or, if the user changed the Event number, will jump straight to the selected event. A drop-down menu allows to pick a policy with vertical tracks, that don't have a χ^2 value.*

Firstly, the program asks for the file to open, it expects the file to contain a ROOT TTree called "sidet". Once the tree has been successfully retrieved, the main user interface window (see **Figure 5**) is shown. With this window, the user can set the thresholds that each event needs to pass to be displayed; as cosmic ray events are expected to be constituted by relatively high energy hits arranged along a straight line across the entire disk, the following cuts were selected:

1. Peak amplitude: for each event, only the hits with a signal peak above the user set threshold are considered.
2. Number of hits: only events with a number of hits greater than this cut are displayed.
3. χ^2: for inclined events, a linear fit is performed, and the event is only drawn if the χ^2 is below the set value. If an event does not span x values greater than the side of a crystal ($34.4\ mm$), it is flagged as vertical and it is not fitted.
4. Vertical policy: the user can select three policies with vertical events, either they are excluded, or they can be included (meaning that they will be drawn if they pass any other filter a part form # 3), or the user can decide to only show vertical events,

in which case filters 1 and 2 are still applied, but any event that is not vertical will be skipped.

Once selected the cuts, the user can use the “Go” button to jump to the next event that will be drawn as shown in **Figure 6**. If the selected policy allows for the display of vertical events, these will be drawn as in **Figure 7**.

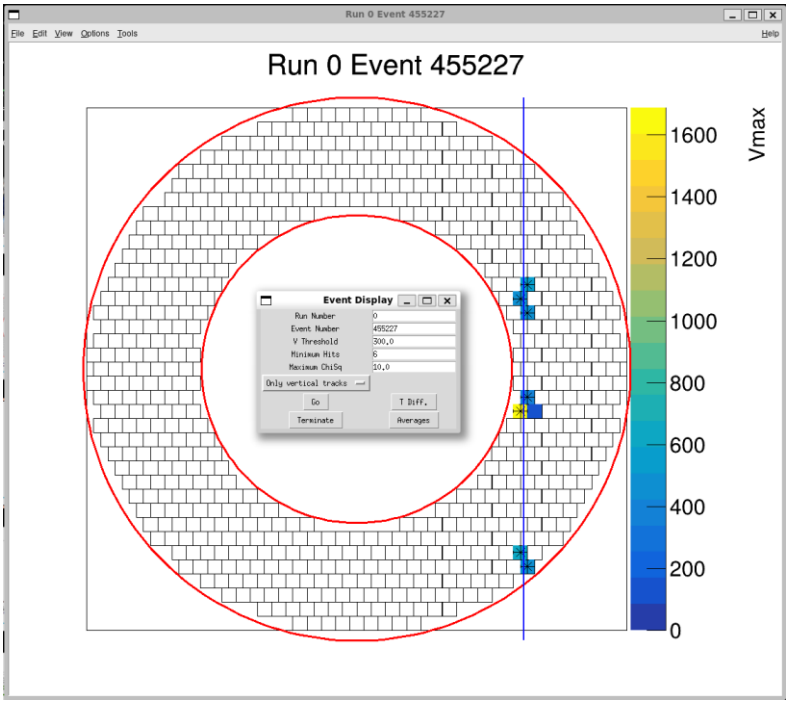

Figure 7. *The Event Display program drawing a vertical event. Black boxes are a frontal view of the crystals, colors represent the peak signal averaged between the two SiPMs and hits above the threshold are marked with an asterisk; the vertical blue line is not a fit, but a vertical line drawn through the mean x value for the vertical event. The graphical user interface is shown superimposed in the middle, note the vertical event policy used to select this event.*

Of course, the program should be able to display an arbitrary event, indexed by the Run and Event number, so, while the relevant fields are updated each time that the “Go” button is pressed, the user can input any value there. When the program detects a change in the run and event number values, this is interpreted as an indication that the user wishes to display a specific event. The program tries to find the specified event in the TTreeIndex that ROOT builds for the Tree, using the nearest greater value if an exact match does not exist. When displaying this user selected event, the program skips all the selection cuts. After jumping to a specific event, pressing “Go” without changing the event number will bring the program back to its normal behavior; this means that starting again the display of the same file is achieved by simply jumping back to event 0 and moving ahead from there, without the need to close and restart the program.

Once an event is displayed, by pressing the “T Diff.” button, the user can change the visualization from the peak value of the signal to the time difference recorded, for each crystal, between the two SiPMs, as shown in **Figure 8**.

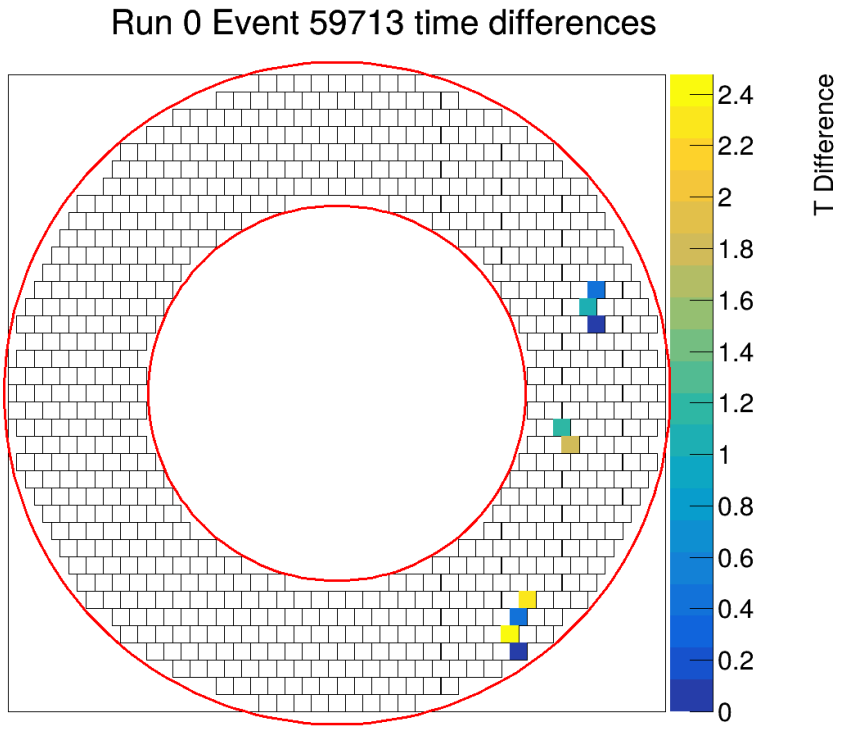

Figure 8. *Example of the display showing the time difference between the signals coming from the two SiPMs for the event. The black boxes represent the crystals and are colored to show the time difference (note that the time response for the event shown has not been calibrated) in ns.*

From the main graphical user interface, the user can press the "Averages" button, causing the program to operate in a different mode. Upon activation of this mode, the program will open each of the events in the tree, storing, for each crystal, the number of hits, the sum of the peak values, and that of the timing differences. This produces three statistical displays, selected from a smaller control interface (**Figure 9**) that only appears after the completion of the loading.

Figure 9. *The averages mode user interface, it is only shown after the user presses the "Averages" button on the main GUI and after the loading of all the events in the opened tree.*

By pressing the relevant buttons, the user can display the number of hits recorded in each crystal (or occupancy), the average peak value and the average time difference; the latter two being computed by dividing the sums by the number of hits. The decision to store only the sums instead of all the hit data reduces the chances for future developments but allows to limit the memory footprint of the program regardless of the size of the data tree studied.

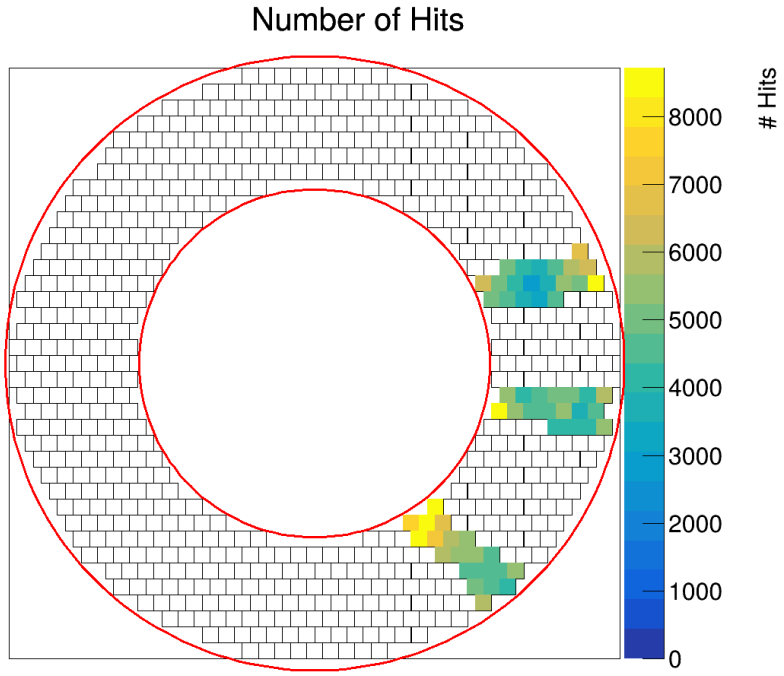

Figure 10. *The display of the occupancy for the VST data, black boxes represent the crystals, while color displays the number of recorded hits. Note the distribution of the instrumented crystals.*

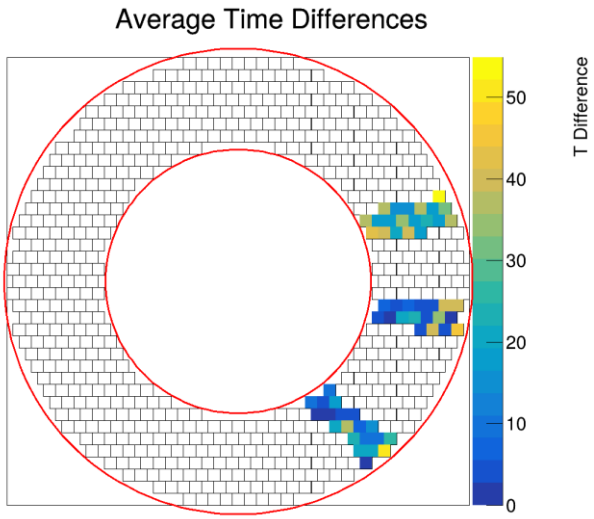

Figure 12. *Display of the average time differences between the signals from the two SiPMs of each crystal. Note that the calibration has not yet been applied here.*

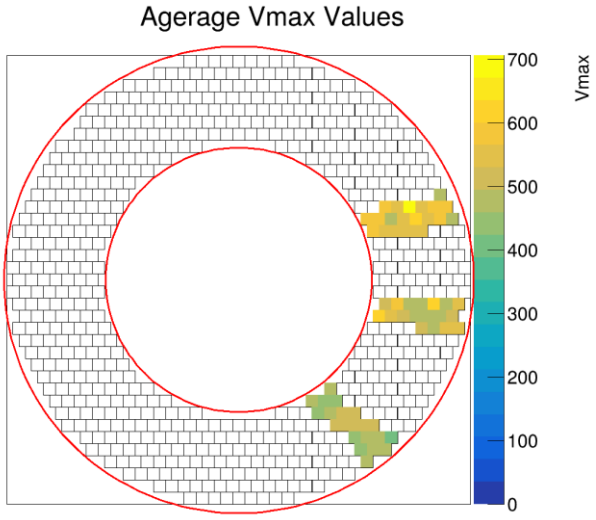

Figure 11. *Display of the average peak value recorded for each crystal, for each event an average is computed between the two SiPMs of each crystal and then averaged across all the events.*

The main Python program is designed to be used as a library for future display implementations. To allow for future extensions to different fitting functions, the entire fitting operation is performed in a nested class that gets instantiated and stored inside the disk class, allowing the use of multiple fits, each of which can be displayed independently or together with the other; or the same linear fit could be instantiated with several thresholds.

Fit Window Optimization

Template fitting

The signals from the SiPMs and front-end electronics are sampled and digitized every $5\,ns$, the timing of the peak sample is stored and represents a first timing information. This timing has an obvious uncertainty of $\pm 5\,ns$, too big for the Mu2e calorimeter. For each signal, the sampled and digitized values around the peak are stored; fitting them with an appropriate template function greatly reduces the uncertainty. Such a function has been devised in previous tests and is shown in **Figure 13**.

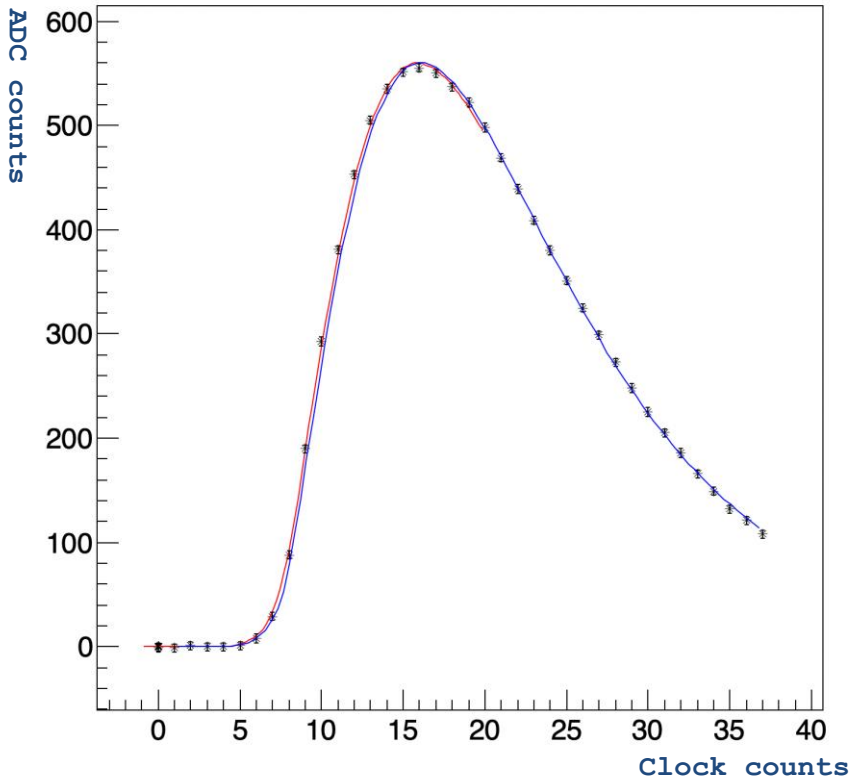

Figure 13. *The template function developed in previous tests fitted over a signal. Blue and red lines represent the fitting over a different range, while dots are the samples.*

Given the sets of samples of the signal, the template function can be fitted over many different subsets of them. We define as a timing fitting window a starting and ending point from the peak time measured in ns. Different fitting windows may give better timing or amplitude results.

Window optimization

For the VST data, a ROOT macro had already been developed to perform the template fitting of the sampled signal to evaluate the arrival time for each hit. For each hit, the complete timing is given by:

$$t_{hit} = T_{hit}^{sampl} + t_{hit}^{fit}$$

Equation 1.

Now, for each crystal involved in a Cosmic Ray Event, both SiPMs should record a hit, and the timing difference between these two hits depends only on the different delays in the signal chains of the two channels; therefore its main value should remain constant across various events. Assuming a uniform illumination among the two SiPMs, the spread of this time difference provides an estimate of the SiPM timing resolution:

$$\sigma(\Delta t) = \sqrt{2} \cdot \sigma t$$

Equation 2.

So, to optimize the fitting window for the best calorimeter timing resolution, the spread of $\sigma(\Delta t)$ was minimized.

Firstly, a fitting window is picked; then, for each crystal and each event, we compute the timing difference between the two SiPMs:

$$\Delta t_{event}^{crystal} = t_{event}^{SiPM\ 0} - t_{event}^{SiPM\ 1}$$

Equation 3.

For each crystal, this timing difference has a distribution as in **Figure 14**:

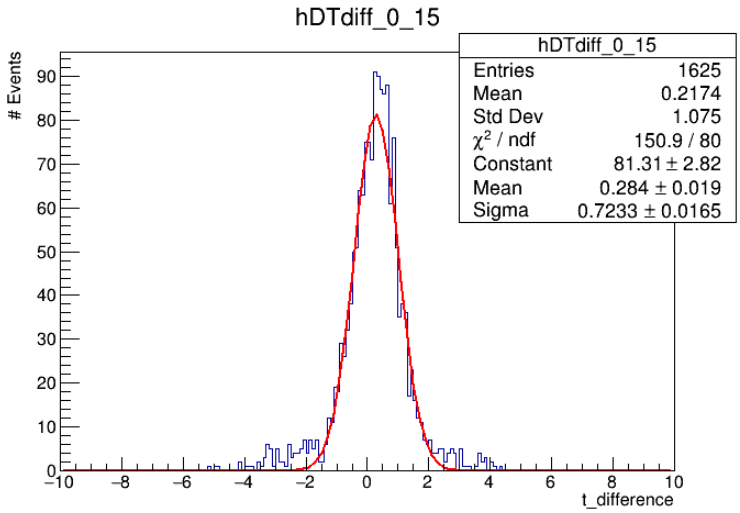

Figure 14. *Distribution of the timing differences between the two SiPMs of a specific crystal across all the available events, with a Gaussian fit in red.*

This distribution is then fit with a Gaussian for each instrumented crystal, with the sigma of this Gaussian representing the timing resolution. So, for each fitting window, we can fill the distribution of $\sigma(\Delta t)$ of each crystal across all the available events as in **Figure 15**.

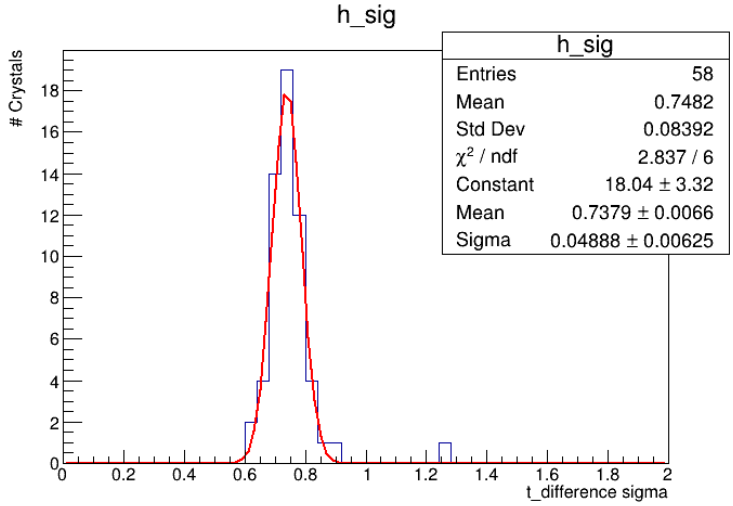

Figure 15. *Distribution of the sigmas of the time differences between the two SiPMs of each crystal across all the available events, for one fitting window.*

Of this distribution, again fitted with a Gaussian, we are interested in the mean that we use as a proxy of the performance of the fitting window under test.

Since each of the windows can be tested independently; for the second part of my project, I developed a Python script that leverages the parallel computation capabilities of the Fermilab Elastic Analysis Facility to run the ROOT macros (some of which I have adapted to this new task) to perform this analysis over a range of possible windows. Each time window is labeled with a starting and ending time (in ns) relative to the peak time, and the results are shown in **Figure 16**.

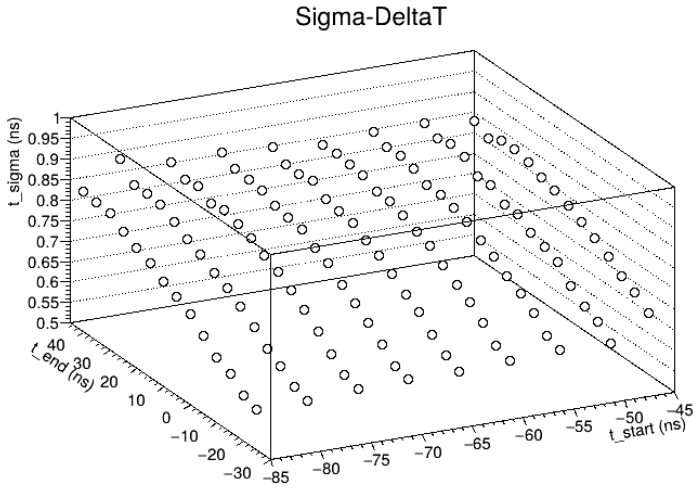

Figure 16. *Results of the fitting window test; for each fitting window (defined by a start and end time relative to the peak) the mean sigma in time differences between the SiPMs of each crystal is plotted.*

The timing step was set to $5\,ns$ since it was not useful to reduce it further below the sampling of the signal. As shown, the best resolution is achieved when the fitting function is applied on the baseline and the rising edge of the signal (see **Figure 13**) i.e. when both the start and end times are negative. The window selected for the subsequent analysis is $[-80\,ns, -20\,ns]$ and the resulting sigma is:

$$\sigma t_{-80,-20} = 0.61\,ns$$

Equation 4.

Time Calibration

To calibrate the time response of each channel using Cosmic Ray Events, a procedure was first developed and tested on a Monte Carlo data sample [see Mu2e-doc-38991 In-situ time calibration of the Mu2e calorimeter]. My project was to write a Python script to perform the intended calibration procedure with Sidet collected data, while performing as much as possible all the operations in parallel.

Calibration Procedure

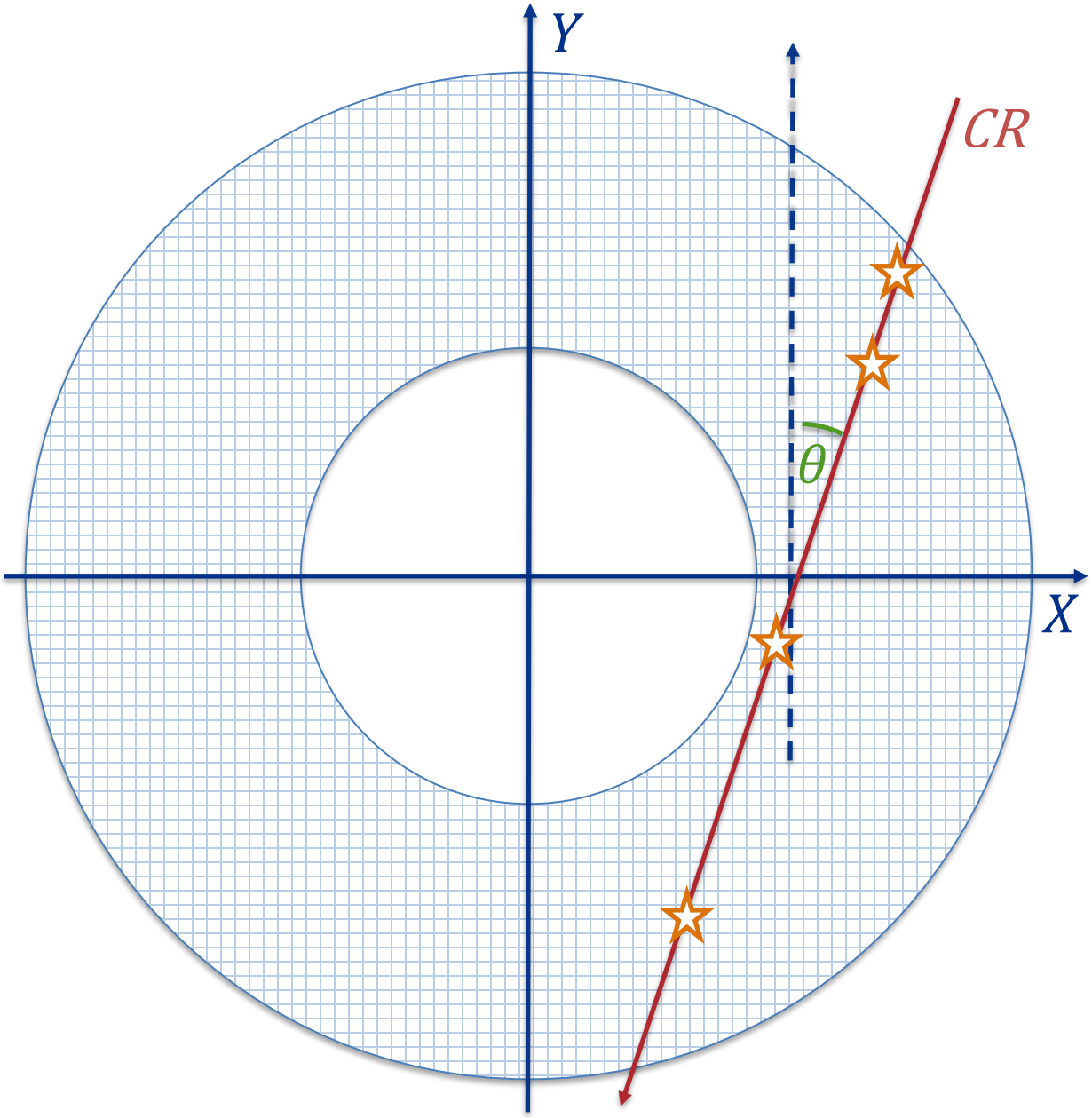

Figure 17. *A schematic representation of a Cosmic Ray (red) crossing the Mu2e calorimeter and leaving some hits (orange stars) on the instrumented crystals. The angle between the Cosmic Ray and the Y axis, θ, is in green.*

The calibration procedure utilizes cosmic rays crossing the calorimeter disk in a straight line constraining the arrival times on each point with the cosmic rays speed (approximated by the speed of light in vacuum). The following set of cuts is used to select the events for this procedure:

- The y-span of the track (i.e. $\max y - \min y$) must be greater than $200\ mm$.
- The number of hits must be greater than 3.
- The digitized peak of the signal V_{max} must be comprised in $300 < V_{max} < 800$ (see **Figure 20**).
- Considering a linear fit of the event, and θ being the angle between the fit-line and the vertical, we require either $\cos\theta > 0.2$ or a vertical like track (if $\max x - \min x < 34.4\ mm$ the event is considered vertical and the linear fit is not performed).
- Considering only the linear fits with $\frac{\chi^2}{ndf} < 5$.

For each hit of the events satisfying the cuts, the first step is removing the dependence from the point along the Cosmic Ray track where the hit occurred. Applying this "geometrical" correction means translating each hit along the fit line to the calorimeter vertical center, $y = 0$.

$$t_i^{geom} = T_i + \frac{y_i}{c \cdot \cos\theta}$$

Equation 5.

Here T_i is the measured time of the hit plus the current calibration for the channel, y_i is the measured y, and θ comes from the linear fit of all the (filtered) hits in the event.

Across all the hits of the event, the weighted average of the times is computed:

$$t_0^{ev} = \frac{\sum_i v_i \cdot t_i^{geom}}{\sum_i v_i}$$

Equation 6.

Here v_i is the digitized peak value (V_{max}) of each hit.

Now we can compute the "raw" residuals for each hit:

$$t_i^{res} = t_i^{geom} - t_0^{ev}$$

Equation 7.

These residuals are labeled by channel-ID (row, column, and SiPM) for a given hit, and the process is repeated for all the available events resulting in a distribution (see **Figure 22**) of the residuals for each channel across all the events. If this distribution contains more than 10

entries, it is fitted with a Gaussian, and its mean is used as a first estimate of the T0 for the channel and subtracted to the current time calibration for the channel, for another iteration.

The entire process needs to be repeated iteratively to consistently minimize the residuals of each channel.

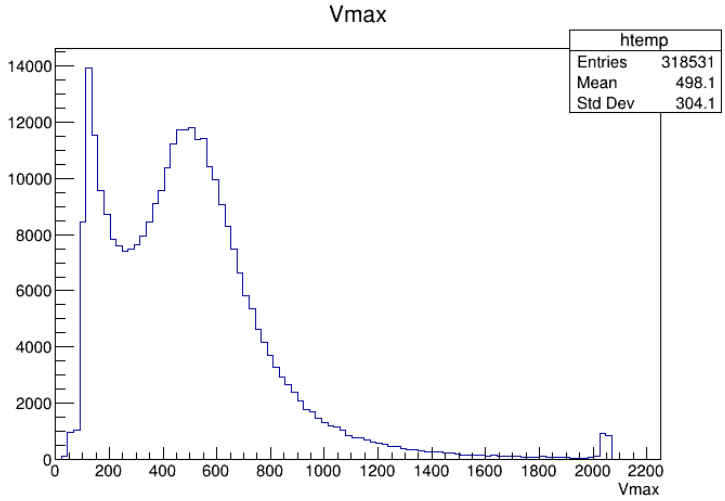

Figure 20. *Distribution of the V_{max} across all the recorded hits, used to determine the values for the signal peak cuts.*

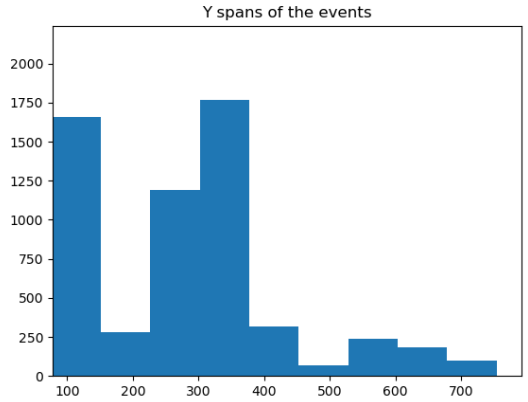

Figure 18. *Distribution of the y-spans (i.e. $\max y - \min y$) for the recorded events in mm.*

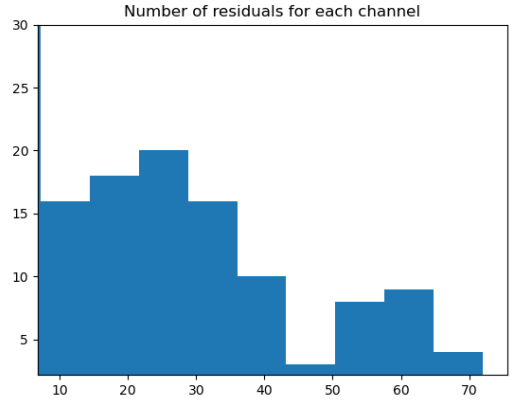

Figure 19. *Distribution of the number of residuals computed for each channel (the 0 bin is not shown as it contains 1232 channels that are not yet instrumented).*

Execution

The calibration procedure explained above was implemented in a Python script, with the only difference of eliminating large residuals, where $|t_i^{res}| > 10\,ns$, that are occasionally produced by events affected by reconstruction errors (this being beyond my project scope).

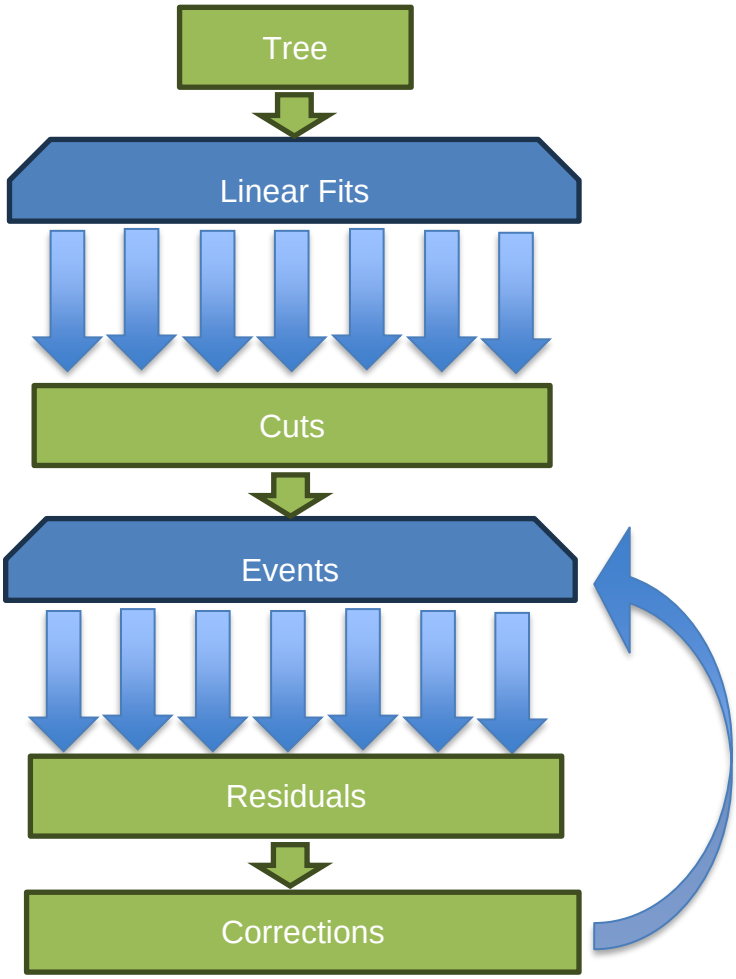

Figure 21. *A schematic diagram of the implemented program, notice the operations performed in parallel (parallel blue arrows).*

Since the amount of data to process is expected to grow significantly (about 100 folds) as the entire calorimeter is instrumented and the acquisition time grows, the program was designed to reduce all the iterative operations to a minimum, performing them in parallel as much as possible. Great use was made of slicing techniques, as suggested by best practice, moving the application of the necessary filters to the libraries instead to the main code. To the same goal, the operations needed only to produce graphs are executed only if the user requires explicitly to produce graphs at the beginning of the execution.

The program uses Uproot to extract the data from a .root file to an Awkward Array structure (as ROOT's TTree can store a different number of values in the same branch across

the entries, reading it as a more common data-frame would require a lot of padding, while the Awkward Array module supports this structure in a memory-efficient way). To enforce dimensional consistency, the program uses the quantities module; the use of it gave mixed results, as it works very well with NumPy structures, while for use with Awkward Arrays or matplotlib the units had to be dropped most of the time. Operations like the linear fitting of the events (that is called only once at the beginning of the program, as its results are necessary to implement the cuts) are performed with NumPy as it does not require the instantiation of a complex plot object for each fit, since this could be heavy on the memory and computational requirements.

When the option to produce plots is not selected, the program only requires indexing the events to compute the fits (and this operation is performed once) and to compute the residuals; both operations are executed in parallel (see **Figure 21**). A third iteration over all the events is required in graphical mode to collect the distribution of the y-spans. The final calibration parameters are saved to a .csv file for easy access.

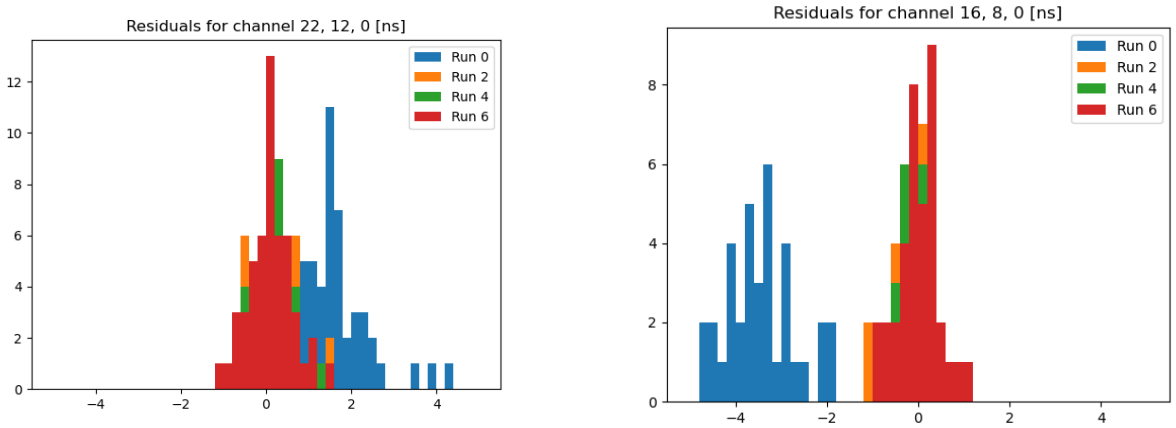

Figure 22. *The distributions of the residuals computed for two of the instrumented channels (labeled with row, column and SiPM) across all the filtered events; colors indicate different runs of the calibration procedure.*

While the number of available events might still be small, the results in **Figure 23** and **Figure 24**) show that the time calibration procedure works and that the time response of the calorimeter can be calibrated to less than $0.1\ ns$.

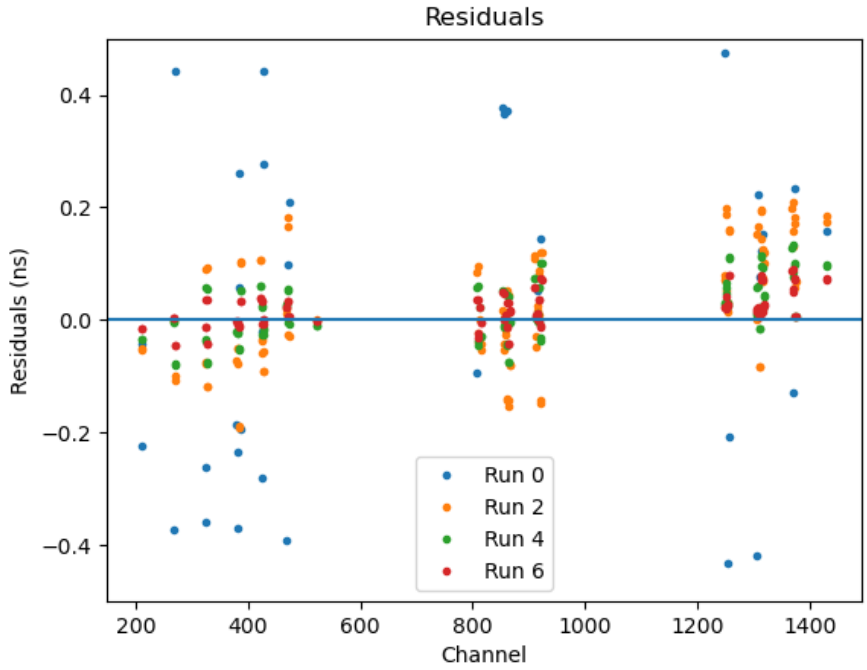

Figure 24. *Scatter plot of the residuals, colors represent runs of the calibration procedure. Note that some of the residuals for Run 0 are outside the plotted region.*

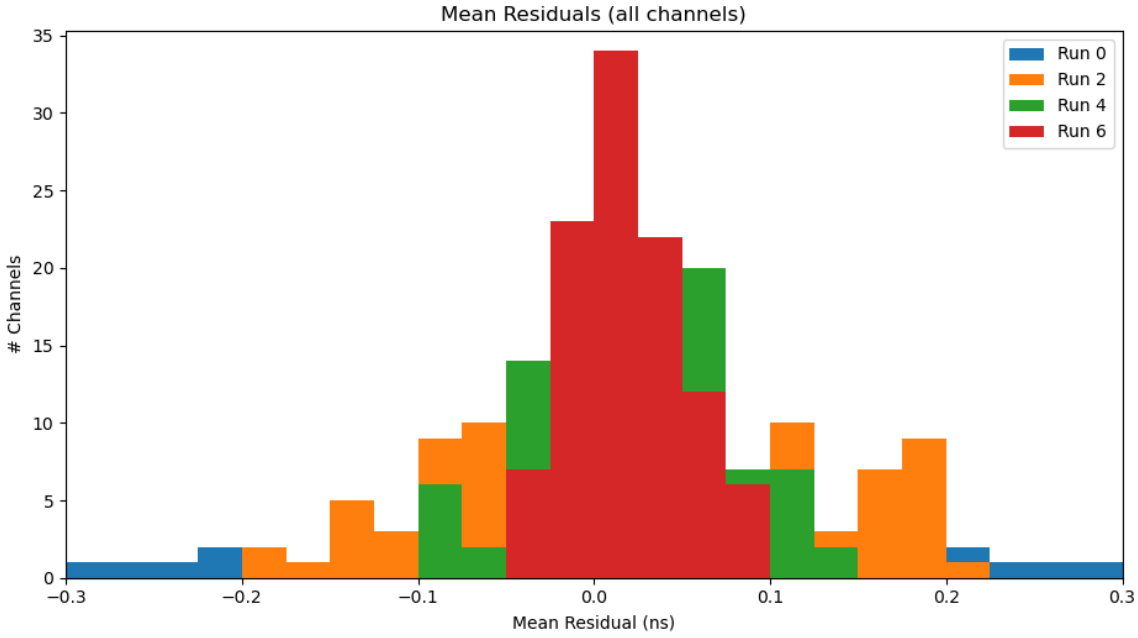

Figure 23. *Distribution of the residuals, colors represent different runs of the calibration procedure. Note that some of the residuals for Run 0 are outside the plotted range.*

Two Versions

While with the current amount of data to process the program can run in a few minutes on any PC, knowing that the number of instrumented crystals and hours of acquisition is going to increase, the program is aimed to run on the Fermilab Elastic Analysis Facility. For this reason, besides the Python script, a Jupiter Notebook (EAF's default format) is provided.

The two programs are not identical: an issue exists when running in Parallel Python programs that use libraries with internal use of lock variables. In this case, when the process forks, it can copy these variables in a locked state without any threads (Python has a process-level parallelism to optimize CPU usage and a thread-level parallelism that runs inside the same process sharing such lock variables) to release the locks; thus stalling the program. This issue is what probably blocks the EAF when trying to perform the linear fits in parallel; something that works on other computers where the program was tested. Tests on the EAF with a capped number of processes to reduce its parallel capability to that of the less performant machines indicated no difference. My humble opinion is that some differences in the versions of the Python interpreter or of the involved libraries can explain what happens.

For this reason, the Jupiter Notebook version of the program does not use process-level parallelism when computing the linear fits (see **Figure 21**) while still using it when computing the residuals; note that while the computation of the linear fits is an onerous task, it only needs to be performed once per event, while the residuals are computed again for each Run. Further work could bring this issue to a better solution, removing the need for the two different versions.

Naples, 17 October 2024.