CYTHON TOOLS

Release 1.1

R. Alvárez, A. de la Torre, S. Manthey, L. Pérez

Jun 27, 2023

CONTENTS

1 Contents **3**

1.1 **Introduction** . . . 3

1.2 Getting started - Setup . . . 3

1.3 Look inside the Macros . . . 4

1.4 **Look inside the libraries** . . . 11

2 Indices and tables **21**

Index **23**

Rodrigo Álvarez-Garrote, Andrés de la Torre Rojo, Sergio Manthey Corchado, Laura Pérez-Molina

CHAPTER

ONE

CONTENTS

1.1 Introduction

Warning: This project is still under development. Please, contact the authors for more information.

1.2 Getting started - Setup

We recommend to use VSCode as editor for begginers (and not so beginners).

Download the library by cloning it from GitHub with:

```
(sudo app install git)
git clone https://github.com/CIEMAT-Neutrino/CYTHON_TOOLS.git
cd CYTHON_TOOLS
code .
```

Please, create a branch for including changes in the library and if everything works as it should you could merge with the main one.

The library is arranged in the folders:

$\verb+\scripts+$ → the scripts are used to configure the needed packages for the library

$\verb+\input+$ → input txt files are stored here and they are used as input in all the macros

$\verb+\lib+$ → libraries where all the functions are stored.

$\verb+\macros+$ → macros we will run to analyse the data.

$\verb+\notebooks+$ → some useful notebooks for interactive visualization

If you have never worked with python you need to install it firstly with

```
sudo apt update
sudo apt install python3-pip
```

Second step is installing all the packages needed for the library to run. This is prepared in the $\verb+scripts+$ folder

```
cd scripts
sh setup.sh
```

Which will install all the dependencies added as requirements in $\verb+requirements.txt+$ and $\verb+requirementsTeX.txt+$

Now you will have two new folders $\verb+\data+$ where we will store the raw and processed data and $\verb+\fit_data+$ where we will store txt files with results for the fits we perform.

The next step will be copy the raw data to start working with it. You can configure this in the $\verb+\copy_data.sh+$ script.

```
sh copy_data.sh AFS_USER (AFS_PCAE)
```

You need to provide your afs user to access the folder with the raw data which is stored in /pnfs/ciemat.es/data/neutrinos/. You can also introduce your pcae number otherwise a default configuration will be used.

By default it is configured to copy one run of each type of Feb22_2 campaign (our reference campagin) but later on you can change it! The copied data (from pnfs) are stored in $\verb+/data/Feb22_2/raw/runXX+$

:space_invader: $\color{purple}{Recommendation \rightarrow}$ It is recommended not to change the content of the scripts so that anyone can run the library from scratch with the initial configuration (except to include improvements, of course). If you want to reuse any, copy it to a folder with your name that is not updated with git (see $\verb+.gitignore+$ file). :space_invader:

> :bulb: $\color{white}{Tip:}$
>
> Now you can check the last script to get ideas on how the CYTHON TOOLs library works (`sh MergeDebug.sh`) However this is explained in more detail in the following sections.

> :information_source: $\color{lightblue}{Info}$
>
> Ideally we want to work in $\verb+/pnfs/ciemat.es/data/neutrinos/FOLDER+$ and so we would mount the folder in our computer with:

```
sshfs USER@pcaeXYZ.ciemat.es:/pnfs/ciemat.es/data/neutrinos/FOLDER ../data
```

:warning: Making sure empty data folder exists :warning: (you will need to change the name of your data folder created by the default configuration built with the setup script)

Once this is done we will find in the following distribution:

$\verb+data/MONTH/raw/runXX+$ with the waveY.dat files and

$\verb+data/MONTH/npy/runXX_chYY+$ with npz created. (npz_names=keys of the my_runs dict in the macros' workflow)

1.3 Look inside the Macros

1.3.1 00Raw2Np

Converts the $\verb+.dat+$ files from $\verb+/data/MONTH/raw+$ into $\verb+.npz+$ files at $\verb+/data/MONTH/npy+$

```
runs = []; channels = []
runs = np.append(runs,info["CALIB_RUNS"])
runs = np.append(runs,info["LIGHT_RUNS"])
runs = np.append(runs,info["ALPHA_RUNS"])
runs = np.append(runs,info["MUONS_RUNS"])

channels = np.append(channels,info["CHAN_TOTAL"])

if info["RAW_DATA"][0] == "DAT":
    print("----- Taking a .dat file as input data -----")
    binary2npy(runs.astype(int),channels.astype(int),info=info,compressed=True,
↪force=False)
```

:fast_forward: $\textcolor{orange}{RUN}$ (in `macros` folder)

```
python3 00Raw2Np.py
```

:heavy_check_mark:

If everything is OK you should get new folders $\verb+/data/Feb22_2/npy/runXX_chYY+$

(the same structure is implemented for all the macros)

where we see the destination file and if the different files existed previously, are overwritten etc…

1.3.2 01PreProcess

This macro will process the {RawPedestal, RawPeak} variables for the RawADC that are computed and saved in .npz files. You will see as terminal output the saved files. By default the saving is set to force = True and so if you pre-process files that already existed they will be overwritten.

```
# PRE-PROCESS RAW #
for run, ch in product(runs.astype(int),channels.astype(int)):
    # Start to load_runs
    my_runs = load_npy([run],[ch], preset="RAW", info=info, compressed=True)

    compute_peak_variables(my_runs,key="RawADC",label="Raw")
    compute_pedestal_variables(my_runs,key="RawADC",label="Raw")
    print_keys(my_runs)
    delete_keys(my_runs,["RawADC"]) # Delete previous peak and pedestal variables
    save_proccesed_variables(my_runs,"ALL",info=info, force=False)
    del my_runs
    gc.collect()
```

:fast_forward: $\textcolor{orange}{RUN}$ (in `macros` folder)

```
python3 01PreProcess.py MergeDebug
```

1.3.3 02Processs

This macro will process the {Pedestal, Peak} variables for the ADC are computed and saved in .npz files. You will see as terminal output the saved files. By default the saving is set to force = True and so if you pre-process files that already existed they will be overwritten. However, we delete the RawKeys from the my_runs dictionary to save computing time (RawInfo don't change).

```
# PROCESS WAVEFORMS (Run in loop to avoid memory issues)
for run, ch in product(runs.astype(int),channels.astype(int)):

    my_runs = load_npy([run],[ch],preset=str(info["LOAD_PRESET"][2]),info=info,
↪compressed=True)
    compute_ana_wvfs(my_runs,debug=False)

    insert_variable(my_runs,np.ones(len(channels)),"PChannel") # Change polarity!
    compute_peak_variables(my_runs,key="ADC")                  # Compute new peak␣
↪variables
    compute_pedestal_variables(my_runs,key="ADC",debug=False)  # Compute new ped␣
↪variables

    # print_keys(my_runs)
    average_wvfs(my_runs,centering="NONE") # Compute average wvfs centering (choose␣
↪from: "NONE", "PEAK", "THRESHOLD")

    delete_keys(my_runs,["RawADC",'RawPeakAmp', 'RawPeakTime', 'RawPedSTD', 'RawPedMean',
↪ 'RawPedMax', 'RawPedMin', 'RawPedLim','RawPChannel']) # Delete branches to avoid␣
↪overwritting
    save_proccesed_variables(my_runs,preset=str(info["SAVE_PRESET"][2]),info=info,␣
↪force=False) # Try preset ANA
    del my_runs
    gc.collect()
```

In this macro we also compute the charge with:

$\verb+integrate_wvfs(my_runs, ["ChargeAveRange"], "AveWvf", ["DAQ", 250], [0,100])+$

It will be used for calibrating the sensors in the following macro.

:fast_forward: $\textcolor{orange}{RUN}$ (in `macros` folder)

```
python3 02Process.py MergeDebug
```

1.3.4 03Integration

for run, ch in product(runs.astype(int),channels.astype(int)):

```
    my_runs = load_npy([run],[ch],preset=str(info["LOAD_PRESET"][3]),info=info,
↪compressed=True)
    # print_keys(my_runs)

    ## Align indivual waveforms + Average ##
    # average_wvfs(my_runs,centering="PEAK") # Compute average wvfs VERY COMPUTER␣
```

(continues on next page)

(continued from previous page)

```
↪INTENSIVE!
    # average_wvfs(my_runs,centering="THRESHOLD") # Compute average wvfs EVEN MORE␣
↪COMPUTER INTENSIVE!

    ## Charge Integration ##
    integrate_wvfs(my_runs, info = info) # Compute charge according to selected average␣
↪wvf from input file ("AveWvf", "AveWvfPeak", "AveWvfThreshold")
    save_proccesed_variables(my_runs,preset=str(info["SAVE_PRESET"][3]),info=info,␣
↪force=True)
    del my_runs
    gc.collect()
```

1.3.5 04Calibration

Compute a calibration ($\verb+ChargeAveRange [pC]+$) histogram where the peaks for the PED/1PE/2PE etc are fitted to obtain the gain.

```
for run, ch in product(runs.astype(int),channels.astype(int)):
    my_runs = load_npy([run],[ch], preset=str(info["LOAD_PRESET"][4]), info=info,
↪compressed=True)

    print_keys(my_runs)

    ## Persistence Plot ##
    # vis_persistence(my_runs)

    ## Calibration ##
    print("Run ", run, "Channel ", ch)
    popt, pcov, perr = calibrate(my_runs,int_key,OPT)
    # Calibration parameters = mu,height,sigma,gain,sn0,sn1,sn2 ##
    calibration_txt(run, ch, popt, pcov, filename="gain",info=info)

    ## SPE Average Waveform ##
    if all(x !=-99 for x in popt):
        SPE_min_charge = popt[3]-abs(popt[5])
        SPE_max_charge = popt[3]+abs(popt[5])
        cut_min_max(my_runs, int_key, limits = {int_key[0]: [SPE_min_charge,SPE_max_
↪charge]})
        average_wvfs(my_runs,centering="NONE",cut_label="SPE")

        save_proccesed_variables(my_runs,info=info,branch_list=["AveWvfSPE"])
```

The parameters are computed from the best fit parameters and covariance matrix obtained from the curve_fit function. These parameters are saved in a txt for each channel.

With the cuts functions we also compute the SPE waveform (that can be visualized with the Vis macros)

:fast_forward: $\textcolor{orange}{RUN}$ (in `macros` folder)

```
python3 03Calibration.py MergeDebug
```

If everything is working as it should toy should obtain the following histograms (raw and fitted) and a txt file with the calibration parameters (if confirmed when asked through terminal)

1.3.6 05Scintillation

```
## Visualize average waveforms by runs/channels ##
my_runs = load_npy(runs.astype(int),channels.astype(int),branch_list=["Label","Sampling",
↪"AveWvf"],info=info,compressed=True) #Remember to LOAD your wvf
vis_compare_wvf(my_runs, ["AveWvf"], compare="RUNS", OPT=OPT)

for run, ch in product(runs.astype(int),channels.astype(int)):
    my_runs = load_npy([run],[ch], branch_list=["ADC","TimeStamp","Sampling",
↪"ChargeAveRange", "NEventsChargeAveRange","AveWvf"], info=info,compressed=True)
↪#preset="ANA"
    print_keys(my_runs)

    ## Integrated charge (scintillation runs) ##
    print("Run ", run, "Channel ", ch)

    popt_ch = []; pcov_ch = []; perr_ch = []; popt_nevt = []; pcov_nevt = []; perr_nevt␣
↪= []
    popt, pcov, perr = charge_fit(my_runs, int_key, OPT); popt_ch.append(popt); pcov_ch.
↪append(pcov); perr_ch.append(perr)

    scintillation_txt(run, ch, popt_ch, pcov_ch, filename="pC", info=info) ## Charge␣
↪parameters = mu,height,sigma,nevents ##
```

1.3.7 06Deconvolution

Before runing the deconvolution macro make sure you have a clean laser/led signal that can be used as a template. The macro will load the alpha runs and rescale the light signal to the SPE amplitude according to AveWvfSPE calculated at calibration stage.

- Firstly, the average wvfs are deconvolved. From these the gauss filter cut-off (from the wiener filter fit) is extracted and saved.
- Secondly, the previous calculated gauss filter is applied to deconvolve the ADC wvfs.
- Finally, the deconvolved wvfs are saved for further process using the standard workflow.

```
for idx, run in enumerate(raw_runs):
    for jdx, ch in enumerate(ana_ch):
        my_runs = load_npy([run],[ch],preset=str(info["LOAD_PRESET"][6]),info=info,
↪compressed=True)  # Select runs to be deconvolved (tipichaly alpha)

        if "SiPM" in str(my_runs[run][ch]["Label"]):
            light_runs =  load_npy([dec_runs[SiPM_OV]],[ch],preset="EVA",info=info,
↪compressed=True) # Select runs to serve as dec template (tipichaly light)
```

(continues on next page)

(continued from previous page)

```
            single_runs = load_npy([ref_runs[SiPM_OV]],[ch],preset="EVA",info=info,
↪compressed=True) # Select runs to serve as dec template scaling (tipichaly SPE)
        elif "SC" in str(my_runs[run][ch]["Label"]):
            light_runs =  load_npy([dec_runs[idx]],[ch],preset="EVA",info=info,
↪compressed=True) # Select runs to serve as dec template (tipichaly light)
            single_runs = load_npy([ref_runs[idx]],[ch],preset="EVA",info=info,
↪compressed=True) # Select runs to serve as dec template scaling (tipichaly SPE)
        else:
            print("UNKNOWN DETECTOR LABEL!")

        keys = ["AveWvf","SER","AveWvf"] # keys contains the 3 labels required for␣
↪deconvolution keys[0] = raw, keys[1] = det_response and keys[2] = deconvolution

        generate_SER(my_runs, light_runs, single_runs)

        OPT = {
            "NOISE_AMP": 1,
            "FIX_EXP":True,
            "LOGY":True,
            "NORM":False,
            "FOCUS":False,
            "SHOW": True,
            "SHOW_F_SIGNAL":True,
            "SHOW_F_GSIGNAL":True,
            "SHOW_F_DET_RESPONSE":True,
            "SHOW_F_GAUSS":True,
            "SHOW_F_WIENER":True,
            "SHOW_F_DEC":True,
            "WIENER_BUFFER": 800,
            "THRLD": 1e-4,
            }

        deconvolve(my_runs,keys=keys,OPT=OPT)

        OPT = {
            "SHOW": False,
            "FIXED_CUTOFF": True
            }

        keys[0] = "ADC"
        keys[2] = "ADC"
        deconvolve(my_runs,keys=keys,OPT=OPT)

        save_proccesed_variables(my_runs,preset=str(info["SAVE_PRESET"][6]),info=info,
↪force=True)
        del my_runs,light_runs,single_runs

generate_input_file(input_file,info,label="Gauss")
```

1.3.8 0XVisEvent

We can visualize the individual events from the moment we pre-process the waveforms and obtain RawInfo.

```
info     = read_input_file(input_file)
runs     = [int(r) for r in input_runs.split(",")]
channels = [int(c) for c in input_channels.split(",")]

OPT  = {
    "MICRO_SEC":   True,
    "NORM":        False,               # Runs can be displayed normalised (True/False)
    "LOGY":        False,               # Runs can be displayed in logy (True/False)
    "SHOW_AVE":    "AveWvf",            # If computed, vis will show average (AveWvf,
↪AveWvfSPE,etc.)
    "SHOW_PARAM":  True,                # Print terminal information (True/False)
    "CHARGE_KEY":  "ChargeAveRange",    # Select charge info to be displayed. Default:
↪"ChargeAveRange" (if computed)
    "PEAK_FINDER": False,               # Finds possible peaks in the window (True/
↪False)
    "LEGEND":      False                # Shows plot legend (True/False)
    }
###################################

##### LOAD RUNS #####
my_runs = load_npy(runs,channels,preset="ANA",info=info,compressed=True) # preset could␣
↪be RAW or ANA
#####################

##### EVENT VISUALIZER #####
vis_npy(my_runs, ["ADC"],-1,OPT=OPT) # Remember to change key accordingly (ADC or RawADC)
############################
```

:fast_forward: $\textcolor{orange}{RUN}$ (in `macros` folder)

```
python3 0XVisEvent.py MergeDebug 1 0,1
```

The individual events of different channels can be visualized together and with AveWvf superposed.

This macro could be used to visualize histograms (with/without applying cuts). In the following picture you can see different options (not all at the same time):

:bulb: $\color{white}{Tips:}$

1. Make the zoom you need with the :mag:
2. This will add one point you need to delete with the **right bottom** of your mouse
3. Select the area you want to keep by adding points with the **left bottom**
4. When you are ready click on the **mouse wheel** to confirm your selection

1.4 Look inside the libraries

1.4.1 io_functions

In this library you can find all the functions related to the input and output of data. The functions are divided in the following categories:

General formatting:

These functions are used to format the output of the code, for the moment they are helping us to print colored lines in the output to detect errors or warnings.

---

class io_functions.**color_list**(*color*)

Function which returns the color in ascii.

class io_functions.**print_colored**(*string*, *color*, *bold=False*, *end='\n'*)

Print a string in a specific color.

Input/Output files:

These functions are used to read and write files. The input files are stored in the `input` folder as `.txt` files.

---

class io_functions.**read_input_file**(*input*, *path='../input/'*, *debug=False*)

Obtain the information stored in a .txt input file to load the runs and channels needed.

Once the deconvolution is performed, there are new input files generated that can be used to re-run the analysis workflow using the deconvolved waveforms.

---

class io_functions.**list_to_string**(*input_list*)

Convert a list to a string to be written in a .txt file Used in generate_input_file.

class io_functions.**generate_input_file**(*input_file*, *info*, *path='../input/'*, *label=''*, *debug=False*)

Generate a .txt file with the information needed to load the runs and channels. Used when deconvolving signals to be able to re-start the analysis workflow with the deconvolved waveforms.

In some functions we need to save some results in a file, for example the results of the calibration/charge fit. The output files are stored in the `fit_data` folder as `.txt` files using the following function:

---

class io_functions.**write_output_file**(*run*, *ch*, *output*, *filename*, *info*, *header_list*, *extra_tab=[]*, *not_saved=[1]*, *path='../fit_data/'*)

General function to write a txt file with the outputs obtained. The file name is defined by the given "filename" variable + _chX. If the file existed previously it appends the new fit values (it save the run for each introduced row) By default we dont save the height of the fitted gaussian in the txt.

Raw data to npy/npz files:

These functions are used to convert the raw(`.root` or `.dat`) data files to `.npy` or `.npz` files. For example, with raw data stored as `run00/wave0.dat` after running `binary2npy` we will create a folder `run00_ch0` where we will store the `.npy` files and each `.npy` file will be have the name of the variable we are storing (i.e `ADC.npy`, `AveWvf.npy`, `PedSTD.npy`, `Sampling.npy`).

`class io_functions.binary2npy(runs, channels, info={}, debug=True, compressed=True, header_lines=6, force=False)`

Dumper from binary format to npy tuples. Input are binary input file path and npy outputfile as strings. Depends numpy.

`class io_functions.root2npy(runs, channels, info={}, debug=False)`

Dumper from .root format to npy tuples. Input are root input file path and npy outputfile as strings. Depends on uproot, awkward and numpy. Size increases x2 times. NEEDS UPDATE!! (see binary2npy)

Dictionaries's Keys:

These functions are used to check, print and delete the keys of a dictionary. The keys are the names of the variables stored in the dictionary that correspond to the names of the `.npy` files stored before.

`class io_functions.check_key(OPT, key)`

Checks if the given key is included in the dictionary OPT. Returns a bool. (True if it finds the key)

`class io_functions.print_keys(my_runs)`

Prints the keys of the run_dict which can be accesed with run_dict[runs][channels][BRANCH]

`class io_functions.delete_keys(my_runs, keys)`

Delete the keys list introduced as 2nd variable

Load/Save npy/npz files:

These functions are used to load and save the `.npy` files. Firstly, we have a function used to get the list of preset names that we want to load or save. This output list is used then in the `load/save` functions.

`class io_functions.get_preset_list(my_run, path, folder, preset, option, debug=False)`

Return as output presets lists for load/save npy files.

VARIABLES:

- my_run: my_runs[run][ch]
- path: saving path
- folder: saving in/out folder
- preset:
  (a) "ALL": all the existing keys/branches
  (b) "ANA": only Ana keys/branches (removing RAW info)

(c) "INT": only Charge*, Ave* keys/branches

(d) "RAW": only Raw information i.e loaded from Raw2Np + Raw* keys

(e) "EVA": all the existing keys/branches EXCEPT ADC

(f) "DEC": only DEC info i.e Wiener*, Gauss*, Dec* or Charge* keys

(g) "CAL": only Charge* keys

- option:

(a) "LOAD": takes the os.listdir(path+folder) as brach_list (IN)

(b) "SAVE": takes the my_run.keys() as branch list (OUT)

class io_functions.**load_npy**(*runs*, *channels*, *preset=''*, *branch_list=[]*, *info={}*, *debug=False*, *compressed=True*)

Structure: run_dict[runs][channels][BRANCH] Loads the selected channels and runs, for simplicity, all runs must have the same number of channels Presets can be used to only load a subset of desired branches. ALL is default.

class io_functions.**save_proccesed_variables**(*my_runs*, *preset=''*, *branch_list=[]*, *info={}*, *force=False*, *debug=False*, *compressed=True*)

Saves the processed variables an npx file.

VARIABLES:

- my_runs: dictionary with the runs and channels to be saved
- preset: preset to be used to save the variables
- branch_list: list of branches to be saved
- info: dictionary with the path and month to be used
- force: if True, the files will be overwritten
- debug: if True, the function will print the branches that are being saved
- compressed: if True, the files will be saved as npz, if False, as npy

---

1.4.2 ana_functions

General analysis:

These functions are used to perform general analysis of the data. For example, we can use the `insert_variable` function to insert a new variable in the dictionary. The `generate_cut_array` function is used to generate the cut array that we will use to select the events that we want to analyze. The `get_units` function is used to get the units of the variables stored in the dictionary.

---

class ana_functions.**insert_variable**(*my_runs*, *var*, *key*, *debug=False*)

Insert values for each type of signal.

class ana_functions.**generate_cut_array**(*my_runs*, *debug=False*)

This function generates an array of bool = True with length = NEvts. If cuts are applied and then you run this function, it resets the cuts.

class ana_functions.**get_units**(*my_runs*, *debug=False*)

Computes and store in a dictionary the units of each variable.

Computing peak/pedestal variables:

These functions are used to compute the peak and pedestal varibales of the raw waveforms.

---

class ana_functions.**compute_peak_variables**(*my_runs*, *key='ADC'*, *label=''*, *debug=False*)

Computes the peaktime and amplitude of a collection of a run's collection in standard format

class ana_functions.**compute_pedestal_variables**(*my_runs*, *key='ADC'*, *label=''*, *buffer=200*, *debug=False*)

Computes the pedestal variables of a collection of a run's collection in standard format

class ana_functions.**compute_pedestal_variables_sliding_window**(*my_runs*, *key='ADC'*, *label=''*, *ped_lim=400*, *sliding=50*, *pretrigger=800*, *start=0*, *debug=False*)

Computes the pedestal variables of a collection of a run's collection in several windows.

** VARIABLES:**

- label: string added to the new variables. Eg: label = Raw, variable = PedSTD –> RawPedSTD
- ped_lim: size in bins of the sliding window
- sliding: bins moved between shifts of the window
- pretrigger: amount of bins to study. Eg: ped_lim = 400, sliding = 50, pretrigger = 800 –> 8 windows to compute
- start: the bin where starts the window. This way you can check the end of the window

class ana_functions.**compute_pedestal_sliding_windows**(*ADC*, *ped_lim=400*, *sliding=50*, *pretrigger=800*, *start=0*)

Taking the best between different windows in pretrigger. Same variables than "compute_pedestal_variables_sliding_window". It checks for the best window.

We can also compute the processed waveforms with the pedestal subtracted as well as computing the power spectrum of the waveforms.

---

class ana_functions.**compute_ana_wvfs**(*my_runs*, *debug=False*)

Computes the peaktime and amplitude of a collection of a run's collection in standard format

class ana_functions.**compute_power_spec**(*ADC*, *timebin*, *debug=False*)

Computes the power spectrum of the given events. It returns both axis.

Finally, we are trying to optimize the functions by introducing `numba` libraries. The `shift_ADCs` function is used to shift the ADCs of the waveforms to the pedestal value. The `shift4_numba` function is used to shift the ADCs of the waveforms to the pedestal value using `numba` library.

1.4.3 wvf_functions

These are more specific functions for analysing the waveforms.

Average Waveforms:

---

class wvf_functions.**average_wvfs**(*my_runs*, *centering='NONE'*, *key='ADC'*, *threshold=0*, *cut_label=''*, *OPT={}*)

It calculates the average waveform of a run. Select centering:

- "NONE" -> AveWvf: each event is added without centering.
- "PEAK" -> AveWvfPeak: each event is centered according to wvf argmax.
- "THRESHOLD" -> AveWvfThreshold: each event is centered according to first wvf entry exceding a threshold.

class wvf_functions.**expo_average**(*my_run*, *alpha*)

This function calculates the exponential average with a given alpha. **returns**: average[i+1] = (1-alpha) * average[i] + alpha * my_run[i+1]

class wvf_functions.**unweighted_average**(*my_run*)

This function calculates the unweighted average. **returns**: average[i+1] = (my_run[i] + my_run[i+1] + my_run[i+2]) / 3

class wvf_functions.**smooth**(*my_run*, *alpha*)

This function calculates the exponential average and then the unweighted average. **returns**: average[i+1] = (my_run[i] + my_run[i+1] + my_run[i+2]) / 3 with my_run = (1-alpha) * average[i] + alpha * my_run[i+1]

Integration:

---

class wvf_functions.**find_baseline_cuts**(*raw*)

It finds the cuts with the x-axis. It returns the index of both bins.

VARIABLES:

- raw: the .root that you want to analize.

class wvf_functions.**find_amp_decrease**(*raw*, *thrld*)

It finds bin where the amp has fallen above a certain threshold relative to the main peak. It returns the index of both bins.

VARIABLES:

- raw: the np array that you want to analize.
- thrld: the relative amp that you want to analize.

class wvf_functions.**integrate_wvfs**(*my_runs*, *info={}*, *key=''*, *cut_label=''*)

This function integrates each event waveform. There are several ways to do it and we choose it with the argument "types".

VARIABLES:

- my_runs: run(s) we want to use
- info: input information from .txt with DAQ characteristics and Charge Information.

- key: waveform we want to integrate (by default any ADC)

In txt Charge Info part we can indicate the type of integration, the reference average waveform and the ranges we want to integrate. If I_RANGE == -1 it fixes t0 to pedestal time and it integrates the time indicated in F_RANGE, e.g. I_RANGE = -1 F_RANGE = 6e-6 it integrates 6 microsecs from pedestal time. If I_RANGE != -1 it integrates from the indicated time to the F_RANGE value, e.g. I_RANGE = 2.1e-6 F_RANGE = 4.3e-6 it integrates in that range. I_RANGE must have same length than F_RANGE!

1.4.4 vis_functions

These functions are used to visualize the data.

Individual Events and Waveforms:

`class vis_functions.vis_npy(my_run, keys, evt_sel=-1, same_plot=False, OPT={}, debug=False)`

This function is a event visualizer. It plots individual events of a run, indicating the pedestal level, pedestal std and the pedestal calc limit. We can interact with the plot and pass through the events freely (go back, jump to a specific event...)

VARIABLES:

- my_run: run(s) we want to check
- KEYS: choose between ADC or AnaADC to see raw (as get from ADC) or Analyzed events (starting in 0 counts), respectively. Type: List
- OPT: several options that can be True or False. Type: List (a) MICRO_SEC: if True we multiply Sampling by 1e6 (b) NORM: True if we want normalized waveforms (c) LOGY: True if we want logarithmic y-axis (d) SHOW_AVE: if computed and True, it will show average (e) SHOW_PARAM: True if we want to check calculated parameters (pedestal, amplitude, charge...) (f) CHARGE_KEY: if computed and True, it will show the parametre value (g) PEAK_FINDER: True if we want to check how many peaks are
- evt_sel: choose the events we want to see. If -1 all events are displayed, if 0 only uncutted events are displayed, if 1 only cutted events are displayed
- same_plot: True if we want to plot different channels in the SAME plot

`class vis_functions.vis_compare_wvf(my_run, keys, compare='RUNS', OPT={})`

This function is a waveform visualizer. It plots the selected waveform with the key and allow comparisson between runs/channels.

VARIABLES:

- my_run: run(s) we want to check
- KEYS: waveform to plot (AveWvf, AveWvdSPE, ...). Type: List
- OPT: several options that can be True or False. Type: List (a) MICRO_SEC: if True we multiply Sampling by 1e6 (b) NORM: True if we want normalized waveforms (c) LOGY: True if we want logarithmic y-axis
- compare: (a) "RUNS" to get a plot for each channel and the selected runs. Type: String (b) "CHANNELS" to get a plot for each run and the selected channels. Type: String

---

Histograms:

---

class vis_functions.**vis_var_hist**(*my_run*, *key*, *compare='NONE'*, *percentile=[0.1, 99.9]*, *OPT={'SHOW': True}*, *select_range=False*)

This function takes the specified variables and makes histograms. The binning is fix to 600, so maybe it is not the appropriate. Outliers are taken into account with the percentile. It discards values below and above the indicated percetiles. It returns values of counts, bins and bars from the histogram to be used in other function.

VARIABLES:

- my_run: run(s) we want to check
- keys: variables we want to plot as histograms. Type: List (a) PeakAmp: histogram of max amplitudes of all events. The binning is 1 ADC. There are not outliers. (b) PeakTime: histogram of times of the max amplitude in events. The binning is the double of the sampling. There are not outliers. (c) Other variable: any other variable. Here we reject outliers.
- percentile: percentile used for outliers removal

WARNING! Maybe the binning stuff should be studied in more detail.

class vis_functions.**vis_two_var_hist**(*my_run*, *keys*, *compare='NONE'*, *percentile=[0.1, 99.9]*, *select_range=False*, *OPT={}*)

This function plots two variables in a 2D histogram. Outliers are taken into account with the percentile. It plots values below and above the indicated percetiles, but values are not removed from data.

VARIABLES:

- my_run: run(s) we want to check
- keys: variables we want to plot as histograms. Type: List
- percentile: percentile used for outliers removal
- select_range: if we still have many outliers we can select the ranges in x and y axis.

1.4.5 dec_functions

class dec_functions.**generate_SER**(*my_runs*, *dec_runs*, *SPE_runs*, *scaling_type='Amplitude'*)

This function rescales AveWvfs from light runs to SPE level to be used for wvf deconvolution:

VARIABLES:

- my_runs: DICTIONARY containing the wvf to be deconvolved.
- dec_runs: DICTIONARY containing the wvfs that work as detector response (light runs).
- SPE_runs: DICTIONARY containing the SPE wvf that serve as reference to rescale dec_runs.

class dec_functions.**deconvolve**(*my_runs*, *keys=[]*, *noise_run=[]*, *peak_buffer=20*, *OPT={}*)

This function deconvolves any given number of arrays according to a provided SPE template. By default it uses a gaussian filter fitted to a wiener assuming gaussian noise at 0.5 amp. SPE level.

VARIABLES:

- my_runs: DICTIONARY containing the wvf to be deconvolved.
- keys: LIST containing the keys of [wvf, template, outputkey].
- peak_buffer: INT with left distance from peak to calculate baseline.
- OPT: DICTIONARY with settings and vis options (“SHOW”, “LOGY”, “NORM”, “FILTER”: Gauss/Wiener, etc.).

class dec_functions.**convolve**(*my_runs*, *keys=[]*, *OPT={}*)

class dec_functions.**check_array_len**(*wvf1*, *wvf2*)

class dec_functions.**check_array_even**(*wvf*)

class dec_functions.**conv_func2**(*wvf*, *t0*, *sigma*, *tau1*, *a1*, *tau2*, *a2*)

class dec_functions.**gauss**(*f*, *fc*, *n*)

class dec_functions.**fit_gauss**(*f*, *fc*, *n*)

1.4.6 fit_functions

class fit_functions.**chi_squared**(*x*, *y*, *popt*)

class fit_functions.**pure_scint**(*time*, *t0*, *a1*, *a2*, *tau1*, *tau2*)

class fit_functions.**gauss**(*x*, *a*, *x0*, *sigma*)

class fit_functions.**gaussian_train**(*x*, **params*)

class fit_functions.**loggaussian_train**(*x*, **params*)

class fit_functions.**gaussian**(*x*, *height*, *center*, *width*)

class fit_functions.**pmt_spe**(*x*, *height*, *center*, *width*)

class fit_functions.**loggaussian**(*x*, *height*, *center*, *width*)

class fit_functions.**func**(*t*, *t0*, *sigma*, *a*, *tau*)

class fit_functions.**func2**(*t*, *p*, *t0*, *sigma*, *a1*, *tau1*, *sigma2*, *a2*, *tau2*)

class fit_functions.**logfunc2**(*t*, *p*, *t0*, *sigma1*, *a1*, *tau1*, *sigma2*, *a2*, *tau2*)

class fit_functions.**logfunc3**(*t*, *p*, *t0*, *sigma*, *a1*, *tau1*, *a2*, *tau2*, *a3*, *tau3*)

class fit_functions.**func3**(*t*, *p*, *t0*, *sigma*, *a1*, *tau1*, *a2*, *tau2*, *a3*, *tau3*)

class fit_functions.**scfunc**(*t*, *a*, *b*, *c*, *d*, *e*, *f*)

class fit_functions.**gaussian_fit**(*counts*, *bins*, *bars*, *thresh*, *fit_function='gaussian'*, *custom_fit=[0]*)

This function fits the histogram, to a gaussians, which has been previoulsy visualized with: **counts, bins, bars = vis_var_hist(my_runs, run, ch, key, OPT=OPT)** And return the parameters of the fit (if performed)

class fit_functions.**gaussian_train_fit**(*counts*, *bins*, *bars*, *thresh*, *fit_function='gaussian'*)

This function fits the histogram, to a train of gaussians, which has been previoulsy visualized with: **counts, bins, bars = vis_var_hist(my_runs, run, ch, key, OPT=OPT)** And return the parameters of the fit (if performed)

class fit_functions.**pmt_spe_fit**(*counts*, *bins*, *bars*, *thresh*)

This function fits the histogram, to a train of gaussians, which has been previoulsy visualized with: **counts, bins, bars = vis_var_hist(my_runs, run, ch, key, OPT=OPT)** And return the parameters of the fit (if performed) [es muy parecida a gaussian_train_fit; hay algunas cosas que las coge en log pero igual se pueden unificar] [se le puede dedicar un poco mas de tiempo para tener un ajuste mas fino pero parece que funciona]

class fit_functions.**peak_fit**(*fit_raw*, *raw_x*, *buffer*, *thrld*, *sigma_fast=1e-09*, *a_fast=1*, *tau_fast=1e-08*, *OPT={}*)

This function fits the peak to a gaussian function, and returns the parameters

class fit_functions.**sipm_fit**(*raw*, *raw_x*, *fit_range*, *thrld=1e-06*, *OPT={}*)

DOC

class fit_functions.**scint_fit**(*raw*, *raw_x*, *fit_range*, *thrld=1e-06*, *i_param={}*, *OPT={}*)

DOC

class fit_functions.**sc_fit**(*raw*, *raw_x*, *fit_range*, *thrld=1e-06*, *OPT={}*)

class fit_functions.**fit_wvfs**(*my_runs*, *signal_type*, *thrld*, *fit_range=[0, 200]*, *i_param={}*, *in_key=['ADC']*, *out_key=''*, *OPT={}*)

DOC

class fit_functions.**get_initial_parameters**(*i_param*)

DOC

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

average_wvfs (*class in wvf_functions*), 15

B

binary2npy (*class in io_functions*), 12

C

check_array_even (*class in dec_functions*), 18
check_array_len (*class in dec_functions*), 18
check_key (*class in io_functions*), 12
chi_squared (*class in fit_functions*), 18
color_list (*class in io_functions*), 11
compute_ana_wvfs (*class in ana_functions*), 14
compute_peak_variables (*class in ana_functions*), 14
compute_pedestal_sliding_windows (*class in ana_functions*), 14
compute_pedestal_variables (*class in ana_functions*), 14
compute_pedestal_variables_sliding_window (*class in ana_functions*), 14
compute_power_spec (*class in ana_functions*), 14
conv_func2 (*class in dec_functions*), 18
convolve (*class in dec_functions*), 17

D

deconvolve (*class in dec_functions*), 17
delete_keys (*class in io_functions*), 12

E

expo_average (*class in wvf_functions*), 15

F

find_amp_decrease (*class in wvf_functions*), 15
find_baseline_cuts (*class in wvf_functions*), 15
fit_gauss (*class in dec_functions*), 18
fit_wvfs (*class in fit_functions*), 19
func (*class in fit_functions*), 18
func2 (*class in fit_functions*), 18
func3 (*class in fit_functions*), 18

G

gauss (*class in dec_functions*), 18
gauss (*class in fit_functions*), 18
gaussian (*class in fit_functions*), 18
gaussian_fit (*class in fit_functions*), 18
gaussian_train (*class in fit_functions*), 18
gaussian_train_fit (*class in fit_functions*), 18
generate_cut_array (*class in ana_functions*), 13
generate_input_file (*class in io_functions*), 11
generate_SER (*class in dec_functions*), 17
get_initial_parameters (*class in fit_functions*), 19
get_preset_list (*class in io_functions*), 12
get_units (*class in ana_functions*), 13

I

insert_variable (*class in ana_functions*), 13
integrate_wvfs (*class in wvf_functions*), 15

L

list_to_string (*class in io_functions*), 11
load_npy (*class in io_functions*), 13
logfunc2 (*class in fit_functions*), 18
logfunc3 (*class in fit_functions*), 18
loggaussian (*class in fit_functions*), 18
loggaussian_train (*class in fit_functions*), 18

P

peak_fit (*class in fit_functions*), 18
pmt_spe (*class in fit_functions*), 18
pmt_spe_fit (*class in fit_functions*), 18
print_colored (*class in io_functions*), 11
print_keys (*class in io_functions*), 12
pure_scint (*class in fit_functions*), 18

R

read_input_file (*class in io_functions*), 11
root2npy (*class in io_functions*), 12

S

save_proccesed_variables (*class in io_functions*), 13
sc_fit (*class in fit_functions*), 19
scfunc (*class in fit_functions*), 18
scint_fit (*class in fit_functions*), 19

sipm_fit (*class in fit_functions*), 19
smooth (*class in wvf_functions*), 15

U

unweighted_average (*class in wvf_functions*), 15

V

vis_compare_wvf (*class in vis_functions*), 16
vis_npy (*class in vis_functions*), 16
vis_two_var_hist (*class in vis_functions*), 17
vis_var_hist (*class in vis_functions*), 16

W

write_output_file (*class in io_functions*), 11