pygama

unknown

Mar 01, 2024

CONTENTS

pygama is a Python package for:

- converting physics data acquisition system output to LEGEND LH5-format HDF5 files
- performing bulk digital signal processing on time-series data
- optimizing those digital signal processing (DSP) routines and tuning associated analysis parameters
- generating and selecting high-level event data for further analysis

CHAPTER

ONE

GETTING STARTED

pygama is published on the Python Package Index. Install on local systems with pip:

```
$ pip install pygama
```

```
$ pip install pygama@git+https://github.com/legend-exp/pygama@main
```

Get a LEGEND container with *pygama* pre-installed on Docker hub or follow instructions on the LEGEND wiki.

If you plan to develop *pygama*, refer to the *developer's guide*.

Attention: If installing in a user directory (typically when invoking pip as a normal user), make sure `~/.local/bin` is appended to `PATH`. The `pygama` executable is installed there.

CHAPTER

TWO

NEXT STEPS

2.1 pygama

2.1.1 pygama package

Pygama: decoding and processing digitizer data. Check out the online documentation

Subpackages

pygama.dsp package

Subpackages

pygama.dsp.processors package

pygama.evt package

Utilities for grouping hit data into events.

Subpackages

pygama.evt.modules package

Contains submodules for evt processing

Submodules

pygama.evt.modules.legend module

Module provides LEGEND internal functions

pygama.evt.modules.legend.**metadata**(*params: dict*) → list

Return type
list

pygama.evt.modules.spm module

Module for special event level routines for SiPMs

functions must take as the first 8 args in order: - path to the hit file - path to the dsp <file - path to the tcm file - hit LH5 root group - dsp LH5 root group - tcm LH5 root group - pattern to cast table names to tcm channel ids - list of channels processed additional parameters are free to the user and need to be defined in the JSON

pygama.evt.modules.spm.**cast_trigger**(*trgr*, *tdefault: float*, *length: int | None = None*) → Array

Return type
Array

pygama.evt.modules.spm.**get_energy**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*) → Array

Return type
Array

pygama.evt.modules.spm.**get_energy_dplms**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*) → Array

Return type
Array

pygama.evt.modules.spm.**get_etc**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*, *swin*, *trail*, *min_first_pls_ene*, *max_per_channel*) → Array

Return type
Array

pygama.evt.modules.spm.**get_majority**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*) → Array

Return type
Array

pygama.evt.modules.spm.**get_majority_dplms**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*) → Array

Return type
Array

pygama.evt.modules.spm.**get_masked_tcm_idx**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*, *mode=0*) → VectorOfVectors

Return type
VectorOfVectors

pygama.evt.modules.spm.**get_spm_ene_or_maj**(*f_hit*, *f_tcm*, *hit_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*, *mode*)

pygama.evt.modules.spm.**get_spm_mask**(*lim: float*, *trgr: Array*, *tmin: float*, *tmax: float*, *pe: Array*, *times: Array*) → Array

Return type
Array

pygama.evt.modules.spm.**get_time_shift**(*f_hit*, *f_dsp*, *f_tcm*, *hit_group*, *dsp_group*, *tcm_group*, *tcm_id_table_pattern*, *chs*, *lim*, *trgr*, *tdefault*, *tmin*, *tmax*) → Array

Return type
Array

Submodules

pygama.evt.aggregators module

This module provides aggregators to build the *evt* tier.

pygama.evt.aggregators.**evaluate_at_channel**(*cumulength: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *chns_rm: list*, *expr: str*, *exprl: list*, *ch_comp: Array*, *var_ph: dict | None = None*, *defv: bool | int | float = nan*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → Array

Aggregates by evaluating the expression at a given channel.

Parameters

- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.
- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **chns_rm** (*list*) – list of channels to be skipped from evaluation and set to default value.
- **expr** (*str*) – expression string to be evaluated.
- **exprl** (*list*) – list of *dsp/hit/evt* parameter tuples in expression `(tier, field)`.
- **ch_comp** (*Array*) – array of rawids at which the expression is evaluated.
- **var_ph** (*dict | None*) – dictionary of *evt* and additional parameters and their values.
- **defv** (*bool | int | float*) – default value.
- **tcm_id_table_pattern** (*str*) – pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.
- **dsp_group** (*str*) – LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.
- **evt_group** (*str*) – LH5 root group in *evt* file.

Return type
Array

pygama.evt.aggregators.**evaluate_at_channel_vov**(*cumulength: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *expr: str*, *exprl: list*, *ch_comp: VectorOfVectors*, *chns_rm: list*, *var_ph: dict | None = None*, *defv: bool | int | float = nan*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → VectorOfVectors

Same as *evaluate_at_channel()* but evaluates expression at non flat channels `VectorOfVectors`.

Parameters

- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.
- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **expr** (*str*) – expression string to be evaluated.
- **exprl** (*list*) – list of *dsp/hit/evt* parameter tuples in expression `(tier, field)`.
- **ch_comp** (*VectorOfVectors*) – array of "rawid"s at which the expression is evaluated.
- **chns_rm** (*list*) – list of channels to be skipped from evaluation and set to default value.
- **var_ph** (*dict | None*) – dictionary of *evt* and additional parameters and their values.
- **defv** (*bool | int | float*) – default value.
- **tcm_id_table_pattern** (*str*) –

  pattern to format *tcm* id values to table name in higher tiers. Must have one
  placeholder which is the *tcm* id.

  dsp_group
  LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.
- **evt_group** (*str*) – LH5 root group in *evt* file.

Return type
VectorOfVectors

pygama.evt.aggregators.**evaluate_to_aoesa**(*cumulength: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *chns: list*, *chns_rm: list*, *expr: str*, *exprl: list*, *qry: str | ndarray[Any, dtype[_ScalarType_co]]*, *nrows: int*, *var_ph: dict | None = None*, *defv: bool | int | float = nan*, *missv=nan*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → ArrayOfEqualSizedArrays

Aggregates by returning an `ArrayOfEqualSizedArrays` of evaluated expressions of channels that fulfill a query expression.

Parameters

- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.

- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **chns** (*list*) – list of channels to be aggregated.
- **chns_rm** (*list*) – list of channels to be skipped from evaluation and set to default value.
- **expr** (*str*) – expression string to be evaluated.
- **exprl** (*list*) – list of *dsp/hit/evt* parameter tuples in expression `(tier, field)`.
- **qry** (*str | ndarray[Any, dtype[_ScalarType_co]]*) – query expression to mask aggregation.
- **nrows** (*int*) – length of output `VectorOfVectors`.
- **ch_comp** – array of "rawid"s at which the expression is evaluated.
- **var_ph** (*dict | None*) – dictionary of *evt* and additional parameters and their values.
- **defv** (*bool | int | float*) – default value.
- **missv** – missing value.
- **sorter** – sorts the entries in the vector according to sorter expression.
- **tcm_id_table_pattern** (*str*) – pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.
- **dsp_group** (*str*) – LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.
- **evt_group** (*str*) – LH5 root group in *evt* file.

Return type
ArrayOfEqualSizedArrays

pygama.evt.aggregators.**evaluate_to_first_or_last**(*cumulength: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *chns: list*, *chns_rm: list*, *expr: str*, *exprl: list*, *qry: str | ndarray[Any, dtype[_ScalarType_co]]*, *nrows: int*, *sorter: tuple*, *var_ph: dict | None = None*, *defv: bool | int | float = nan*, *is_first: bool = True*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → Array

Aggregates across channels by returning the expression of the channel with value of *sorter*.

Parameters

- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.
- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **chns** (*list*) – list of channels to be aggregated.
- **chns_rm** (*list*) – list of channels to be skipped from evaluation and set to default value.

- **expr** (*str*) – expression string to be evaluated.
- **exprl** (*list*) – list of *dsp/hit/evt* parameter tuples in expression `(tier, field)`.
- **qry** (*str | ndarray[Any, dtype[_ScalarType_co]]*) – query expression to mask aggregation.
- **nrows** (*int*) – length of output array.
- **sorter** (*tuple*) – tuple of field in *hit/dsp/evt* tier to evaluate `(tier, field)`.
- **var_ph** (*dict | None*) – dictionary of *evt* and additional parameters and their values.
- **defv** (*bool | int | float*) – default value.
- **is_first** (*bool*) – defines if sorted by smallest or largest value of *sorter*
- **tcm_id_table_pattern** (*str*) – pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.
- **dsp_group** (*str*) – LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.
- **evt_group** (*str*) – LH5 root group in *evt* file.

Return type
Array

pygama.evt.aggregators.**evaluate_to_scalar**(*mode: str*, *cumulength: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *chns: list*, *chns_rm: list*, *expr: str*, *exprl: list*, *qry: str | ndarray[Any, dtype[_ScalarType_co]]*, *nrows: int*, *var_ph: dict | None = None*, *defv: bool | int | float = nan*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → Array

Aggregates by summation across channels.

Parameters

- **mode** (*str*) – aggregation mode.
- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.
- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **chns** (*list*) – list of channels to be aggregated.
- **chns_rm** (*list*) – list of channels to be skipped from evaluation and set to default value.
- **expr** (*str*) – expression string to be evaluated.
- **exprl** (*list*) – list of *dsp/hit/evt* parameter tuples in expression `(tier, field)`.
- **qry** (*str | ndarray[Any, dtype[_ScalarType_co]]*) – query expression to mask aggregation.
- **nrows** (*int*) – length of output array
- **var_ph** (*dict | None*) – dictionary of *evt* and additional parameters and their values.

- **defv** (*bool | int | float*) – default value.
- **tcm_id_table_pattern** (*str*) – pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.
- **dsp_group** (*str*) – LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.
- **evt_group** (*str*) – LH5 root group in *evt* file.

Return type
Array

pygama.evt.aggregators.**evaluate_to_vector**(*cumulength: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *chns: list*, *chns_rm: list*, *expr: str*, *exprl: list*, *qry: str | ndarray[Any, dtype[_ScalarType_co]]*, *nrows: int*, *var_ph: dict | None = None*, *defv: bool | int | float = nan*, *sorter: str | None = None*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → VectorOfVectors

Aggregates by returning a `VectorOfVector` of evaluated expressions of channels that fulfill a query expression.

Parameters

- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.
- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **chns** (*list*) – list of channels to be aggregated.
- **chns_rm** (*list*) – list of channels to be skipped from evaluation and set to default value.
- **expr** (*str*) – expression string to be evaluated.
- **exprl** (*list*) – list of *dsp/hit/evt* parameter tuples in expression `(tier, field)`.
- **qry** (*str | ndarray[Any, dtype[_ScalarType_co]]*) – query expression to mask aggregation.
- **nrows** (*int*) – length of output `VectorOfVectors`.
- **ch_comp** – array of "rawids" at which the expression is evaluated.
- **var_ph** (*dict | None*) – dictionary of *evt* and additional parameters and their values.
- **defv** (*bool | int | float*) – default value.
- **sorter** (*str | None*) – sorts the entries in the vector according to sorter expression. `ascend_by:<hit|dsp.field>` results in an vector ordered ascending, `decend_by:<hit|dsp.field>` sorts descending.
- **tcm_id_table_pattern** (*str*) –

  pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.

  dsp_group
  LH5 root group in *dsp* file.

- **hit_group** (*str*) – LH5 root group in *hit* file.
- **evt_group** (*str*) – LH5 root group in *evt* file.

Return type
VectorOfVectors

pygama.evt.build_evt module

This module implements routines to build the *evt* tier.

pygama.evt.build_evt.**build_evt**(*f_tcm: str*, *f_dsp: str*, *f_hit: str*, *evt_config: str | dict*, *f_evt: str | None = None*, *wo_mode: str = 'write_safe'*, *evt_group: str = 'evt'*, *tcm_group: str = 'hardware_tcm_1'*, *dsp_group: str = 'dsp'*, *hit_group: str = 'hit'*, *tcm_id_table_pattern: str = 'ch{}'*) → None | Table

Transform data from the *hit* and *dsp* levels which a channel sorted to a event sorted data format.

Parameters

- **f_tcm** (*str*) – input LH5 file of the *tcm* level.
- **f_dsp** (*str*) – input LH5 file of the *dsp* level.
- **f_hit** (*str*) – input LH5 file of the *hit* level.
- **evt_config** (*str | dict*) – name of configuration file or dictionary defining event fields. Channel lists can be defined by importing a metadata module.
  - `operations` defines the fields `name=key`, where `channels` specifies the channels used to for this field (either a string or a list of strings),
  - `aggregation_mode` defines how the channels should be combined (see *evaluate_expression()*).
  - `expression` defnies the mathematical/special function to apply (see *evaluate_expression()*),
  - `query` defines an expression to mask the aggregation.
  - `parameters` defines any other parameter used in expression.

  For example:

```
{
  "channels": {
    "geds_on": ["ch1084803", "ch1084804", "ch1121600"],
    "spms_on": ["ch1057600", "ch1059201", "ch1062405"],
    "muon": "ch1027202",
  },
  "operations": {
    "energy_id":{
      "channels": "geds_on",
      "aggregation_mode": "gather",
      "query": "hit.cuspEmax_ctc_cal > 25",
      "expression": "tcm.array_id",
      "sort": "ascend_by:dsp.tp_0_est"
    },
    "energy":{
      "aggregation_mode": "keep_at_ch:evt.energy_id",
```

(continues on next page)

(continued from previous page)

```
      "expression": "hit.cuspEmax_ctc_cal > 25"
    }
    "is_muon_rejected":{
      "channels": "muon",
      "aggregation_mode": "any",
      "expression": "dsp.wf_max>a",
      "parameters": {"a":15100},
      "initial": false
    },
    "multiplicity":{
      "channels":  ["geds_on", "geds_no_psd", "geds_ac"],
      "aggregation_mode": "sum",
      "expression": "hit.cuspEmax_ctc_cal > a",
      "parameters": {"a":25},
      "initial": 0
    },
    "t0":{
      "aggregation_mode": "keep_at_ch:evt.energy_id",
      "expression": "dsp.tp_0_est"
    },
    "lar_energy":{
      "channels": "spms_on",
      "aggregation_mode": "function",
      "expression": ".modules.spm.get_energy(0.5, evt.t0, 48000,␣
↪1000, 5000)"
    },
  }
}
```

- **f_evt** (*str | None*) – name of the output file. If `None`, return the output `Table` instead of writing to disk.
- **wo_mode** (*str*) – writing mode.
- **group** (*evt*) – LH5 root group name of *evt* tier.
- **tcm_group** (*str*) – LH5 root group in *tcm* file.
- **dsp_group** (*str*) – LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.
- **tcm_id_table_pattern** (*str*) – pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.

Return type
None | *Table*

pygama.evt.build_evt.**evaluate_expression**(*f_tcm: str*, *f_hit: str*, *f_dsp: str*, *chns: list*, *chns_rm: list*, *mode: str*, *expr: str*, *nrows: int*, *table: Table | None = None*, *para: dict | None = None*, *qry: str | None = None*, *defv: bool | int | float = nan*, *sorter: str | None = None*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*, *tcm_group: str = 'tcm'*) → Array | ArrayOfEqualSizedArrays | VectorOfVectors

Evaluates the expression defined by the user across all channels according to the mode.

Parameters

- **f_tcm** (*str*) – path to *tcm* tier file.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **chns** (*list*) – list of channel names across which expression gets evaluated (form: ch<rawid>).
- **chns_rm** (*list*) – list of channels which get set to default value during evaluation. In function mode they are removed entirely (form: ch<rawid>)
- **mode** (*str*) – The mode determines how the event entry is calculated across channels. Options are:
  - `first_at:sorter`: aggregates across channels by returning the expression of the channel with smallest value of sorter.
  - `last_at`: aggregates across channels by returning the expression of the channel with largest value of sorter.
  - `sum`: aggregates by summation.
  - `any`: aggregates by logical or.
  - `all`: aggregates by logical and.
  - `keep_at_ch:ch_field`: aggregates according to passed ch_field.
  - `keep_at_idx:tcm_idx_field`: aggregates according to passed tcm index field.
  - `gather`: Channels are not combined, but result saved as `VectorOfVectors`.
- **qry** (*str | None*) – a query that can mask the aggregation.
- **expr** (*str*) – the expression. That can be any mathematical equation/comparison. If *mode* is `function`, the expression needs to be a special processing function defined in modules (e.g. *modules.spm.get_energy()*). In the expression parameters from either hit, dsp, evt tier (from operations performed before this one! Dictionary operations order matters), or from the `parameters` field can be used.
- **nrows** (*int*) – number of rows to be processed.
- **table** (*Table | None*) – table of *evt* tier data.
- **para** (*dict | None*) – dictionary of parameters defined in the `parameters` field in the configuration dictionary.
- **defv** (*bool | int | float*) – default value of evaluation.
- **sorter** (*str | None*) – can be used to sort vector outputs according to sorter expression (see `evaluate_to_vector()`).
- **tcm_id_table_pattern** (*str*) – pattern to format tcm id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.
- **group** (*evt*) – LH5 root group name of *evt* tier.
- **tcm_group** (*str*) – LH5 root group in *tcm* file.
- **dsp_group** (*str*) – LH5 root group in *dsp* file.
- **hit_group** (*str*) – LH5 root group in *hit* file.

Return type
Array | *ArrayOfEqualSizedArrays* | *VectorOfVectors*

pygama.evt.build_tcm module

pygama.evt.build_tcm.**build_tcm**(*input_tables: list[tuple[str, str | list[str]]]*, *coin_col: str*, *hash_func: str = '\\d+'*, *coin_window: float = 0*, *window_ref: str = 'last'*, *out_file: str | None = None*, *out_name: str = 'tcm'*, *wo_mode: str = 'write_safe'*) → Table

Build a Time Coincidence Map (TCM).

Given a list of input tables, create an output table containing an entry list of coincidences among the inputs. Uses *evt.tcm.generate_tcm_cols()*. For use with the *DataLoader*.

Parameters

- **input_tables** (*list[tuple[str, str | list[str]]]*) – each entry is `(filename, table_name_pattern)`. All tables matching `table_name_pattern` in `filename` will be added to the list of input tables. `table_name_pattern` can be replaced with a list of patterns to be searched for in the file
- **coin_col** (*str*) – the name of the column in each tables used to build coincidences. All tables must contain a column with this name.
- **hash_func** (*str*) – mapping of table names to integers for use in the TCM. *hash_func* is a regexp pattern that acts on each table name. The default *hash_func* `r"\d+"` pulls the first integer out of the table name. Setting to `None` will use a table's index in *input_tables*.
- **coin_window** (*float*) – the clustering window width.
- **window_ref** (*str*) – Configuration for the clustering window.
- **out_file** (*str | None*) – name (including path) for the output file. If `None`, no file will be written; the TCM will just be returned in memory.
- **out_name** (*str*) – name for the TCM table in the output file.
- **wo_mode** (*str*) – mode to send to `write()`.

Return type
Table

See also:

tcm.generate_tcm_cols

pygama.evt.tcm module

pygama.evt.tcm.**generate_tcm_cols**(*coin_data: list[ndarray]*, *coin_window: float = 0*, *window_ref: str = 'last'*, *array_ids: list[int] | None = None*, *array_idxs: list[int] | None = None*) → dict[ndarray]

Generate the columns of a time coincidence map.

Generate the columns of a time coincidence map from a list of arrays of coincidence data (e.g. hit times from different channels). Returns 3 `numpy.ndarray`s representing a vector-of-vector-like structure: two flattened arrays `array_id` (e.g. channel number) and `array_idx` (e.g. hit index) that specify the location in the input `coin_data` of each datum belonging to a coincidence event, and a `cumulative_length` array that specifies which rows of the other two output arrays correspond to which coincidence event. These can be used to retrieve other data at the same tier as the input data into coincidence structures.

The 0'th entry of `cumulative_length` contains the number of hits in the zeroth coincidence event, and the i'th entry is set to `cumulative_length[i-1]` plus the number of hits in the i'th event. Thus, the hits of the i'th event can be found in rows `cumulative_length[i-1]` to `cumulative_length[i] - 1` of `array_id` and `array_idx`.

An example: `cumulative_length = [4, 7, ...]`. Then rows 0 to 3 in *array_id* and *array_idx* correspond to the hits in event 0, rows 4 to 6 correspond to event 1, and so on.

Makes use of `pandas.concat()`, `pandas.DataFrame.sort_values()`, and `pandas.DataFrame.diff()` functions:

- pull data into a `pandas.DataFrame`
- sort events by strictly ascending value of *coin_col*
- group hits if the difference in *coin_data* is less than *coin_window*

Parameters

- **coin_data** (*list[ndarray]*) – a list of arrays of the data to be clustered.
- **coin_window** (*float*) – the clustering window width. *coin_data* within the *coin_window* get aggregated into the same coincidence cluster. A value of `0` means an equality test.
- **window_ref** (*str*) – when testing one datum for inclusion in a cluster, test if it is within *coin_window* of
  - `"first"` – the first element in the cluster (rigid window width)
  - `"last"` – the last element in the cluster (window grows until two data are separated by more than coin_window)
- **array_ids** (*list[int] | None*) – if provided, use *array_ids* in place of "index in coin_data" as the integer corresponding to each element of *coin_data* (e.g. a channel number).
- **array_idxs** (*list[int] | None*) – if provided, use these values in places of the `DataFrame` index for the return values of *array_idx*.

Returns

col_dict – keys are `cumulative_length`, `array_id`, and `array_idx`. `cumulative_length` specifies which rows of the other two output arrays correspond to which coincidence event. `array_id` and `array_idx` specify the location in `coin_data` of each datum belonging to the coincidence event.

Return type

dict[*ndarray*]

pygama.evt.utils module

This module provides utilities to build the *evt* tier.

pygama.evt.utils.**find_parameters**(*f_hit: str*, *f_dsp: str*, *ch: str*, *idx_ch: ndarray[Any, dtype[_ScalarType_co]]*, *exprl: list*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → dict

Wraps `load_vars_to_nda()` to return parameters from *hit* and *dsp* tiers.

Parameters

- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.

- **ch** (*str*) – "rawid" in the tiers.
- **idx_ch** (*ndarray[Any, dtype[_ScalarType_co]]*) – index array of entries to be read from files.
- **exprl** (*list*) – list of tuples `(tier, field)` to be found in the *hit/dsp* tiers.
- **dsp_group** (*str*) – LH5 root group in dsp file.
- **hit_group** (*str*) – LH5 root group in hit file.

Return type
dict

pygama.evt.utils.**get_data_at_channel**(*ch: str*, *ids: ndarray[Any, dtype[_ScalarType_co]]*, *idx: ndarray[Any, dtype[_ScalarType_co]]*, *expr: str*, *exprl: list*, *var_ph: dict*, *is_evaluated: bool*, *f_hit: str*, *f_dsp: str*, *defv*, *tcm_id_table_pattern: str = 'ch{}'*, *evt_group: str = 'evt'*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → ndarray

Evaluates an expression and returns the result.

Parameters

- **ch** (*str*) – "rawid" of channel to be evaluated.
- **idx** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* index array.
- **ids** (*ndarray[Any, dtype[_ScalarType_co]]*) – *tcm* id array.
- **expr** (*str*) – expression to be evaluated.
- **exprl** (*list*) – list of parameter-tuples `(root_group, field)` found in the expression.
- **var_ph** (*dict*) – dict of additional parameters that are not channel dependent.
- **is_evaluated** (*bool*) – if false, the expression does not get evaluated but an array of default values is returned.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **defv** – default value.
- **tcm_id_table_pattern** (*str*) – Pattern to format tcm id values to table name in higher tiers. Must have one placeholder which is the tcm id.
- **dsp_group** (*str*) – LH5 root group in dsp file.
- **hit_group** (*str*) – LH5 root group in hit file.
- **evt_group** (*str*) – LH5 root group in evt file.

Return type
ndarray

pygama.evt.utils.**get_mask_from_query**(*qry: str | ndarray[Any, dtype[_ScalarType_co]]*, *length: int*, *ch: str*, *idx_ch: ndarray[Any, dtype[_ScalarType_co]]*, *f_hit: str*, *f_dsp: str*, *hit_group: str = 'hit'*, *dsp_group: str = 'dsp'*) → ndarray

Evaluates a query expression and returns a mask accordingly.

Parameters

- **qry** (*str | ndarray[Any, dtype[_ScalarType_co]]*) – query expression.
- **length** (*int*) – length of the return mask.

- **ch** (*str*) – "rawid" of channel to be evaluated.
- **idx_ch** (*ndarray[Any, dtype[_ScalarType_co]]*) – channel indices to be read.
- **f_hit** (*str*) – path to *hit* tier file.
- **f_dsp** (*str*) – path to *dsp* tier file.
- **hit_group** (*str*) – LH5 root group in hit file.
- **dsp_group** (*str*) – LH5 root group in dsp file.

Return type
ndarray

pygama.evt.utils.**get_table_name_by_pattern**(*tcm_id_table_pattern: str*, *ch_id: int*) → str

Return type
str

pygama.evt.utils.**get_tcm_id_by_pattern**(*tcm_id_table_pattern: str*, *ch: str*) → int

Return type
int

pygama.evt.utils.**num_and_pars**(*value: str*, *par_dic: dict*)

pygama.flow package

High-level data flow handling routines.

Submodules

pygama.flow.data_loader module

Routines for high-level data loading and skimming.

class pygama.flow.data_loader.**DataLoader**(*config: str | dict*, *filedb: str | dict | FileDB | None = None*, *file_query: str | None = None*)

Bases: object

Facilitate loading of processed data across several tiers.

Where possible, uses a FileDB object so that a user can quickly select a subset of cycle files for interest, and access information at each processing tier.

Example JSON configuration file:

```
{
    "filedb": "path/to/filedb.h5"
    "levels": {
        "hit": {
            "tiers": ["raw", "dsp", "hit"]
        },
        "tcm": {
            "tiers": ["tcm"],
```

(continues on next page)

(continued from previous page)

```
            "parent": "hit",
            "child": "evt",
            "tcm_cols": {
                "child_idx": "coin_idx",
                "parent_tb": "array_id",
                "parent_idx": "array_idx"
            }
        },
        "evt": {
            "tiers": ["evt"]
        }
    }
}
```

Examples

```
>>> from pygama.flow import DataLoader
>>> dl = DataLoader("loader-config.json")
>>> dl.set_files("file_status == 26 and timestamp == '20220716T130443Z'")
>>> dl.set_datastreams([3, 6, 8], "ch")
>>> dl.set_cuts({"hit": "daqenergy > 1000 and AoE > 3", "evt": "muon_veto == False"}
↪)
>>> dl.set_output(fmt="pd.DataFrame", columns=["daqenergy", "channel"])
>>> data = dl.load()
```

Be careful, *load()* loads data in memory regardless of its size. If loading a lot of data (e.g. waveforms), you might want to do it in chunks. `next()` does exactly this:

```
>>> for chunk in dl.load():
...     run_my_processing(chunk)
```

Advanced Usage:

```
>>> from pygama.flow import DataLoader
>>> dl = DataLoader("loader-config.json", filedb="filedb-config.json")  # or any␣
↪value accepted by the FileDB constructor
>>> dl.set_files("all")
>>> dl.set_datastreams([0], "ch")
>>> dl.set_cuts({"hit": "wf_max > 30000"})
>>> el = dl.build_entry_list(tcm_level="tcm", mode="any")
>>> el.query("hit_table == 20", inplace=True)
>>> dl.set_output(fmt="pd.DataFrame", columns=["daqenergy", "channel"])
>>> data = dl.load(el)
```

Parameters

- **config** (*str | dict*) – configuration dictionary or JSON file, see above for a specification. Accepts strings in the following format:

  ```
  path/to/config.json[field1/field2/...]]
  ```

to specify the location of the *DataLoader* configuration in the `config.json` dictionary, if not at the first level.

- **filedb** (*str | dict | FileDB*) – the loader needs a file database. It can be specified in multiple ways:
  - an instance of *FileDB*.
  - an LH5 file containing a *FileDB* (see also *FileDB.to_disk()*).
  - a *FileDB* configuration dictionary or JSON file.

  If `None`, uses the value of the `filedb` key in *config* to instantiate a *FileDB* object.
- **file_query** (*str*) – string query that should operate on columns of a *FileDB*.

Note: No data is loaded in memory at this point.

---

browse(*entry_list: DataFrame | None = None*, *buffer_len: int = 128*, ***kwargs*) → WaveformBrowser

Return a `WaveformBrowser` object for waveform inspection.

Parameters

- **entry_list** (*DataFrame | None*) – the output of *build_entry_list()*. If `None`, builds it according to the current configuration.
- **buffer_len** (*int*) – number of waveforms to keep in memory at a time.
- ****kwargs** – keyword arguments forwarded to `WaveformBrowser`.

Return type
WaveformBrowser

See also:

`WaveformBrowser`

build_entry_list(*tcm_level: str | None = None*, *tcm_table: int | str | None = None*, *mode: str = 'only'*, *save_output_columns: bool = False*, *in_memory: bool = True*, *output_file: str | None = None*) → dict[int, DataFrame] | DataFrame | None

Applies cuts to the tables and files of interest.

Can only load up to two levels, those joined by *tcm_level*.

Parameters

- **tcm_level** (*str | None*) – the type of TCM to be used. If `None`, will only return information from lowest level.
- **tcm_table** (*int | str | None*) – the identifier of the table inside this TCM level that you want to use. If unspecified, there must only be one table inside a TCM file in *tcm_level*.
- **mode** (*str*) – if `any`, returns every hit in the event if any hit in the event passes the cuts. If `only`, only returns hits that pass the cuts.
- **save_output_columns** (*bool*) – if `True`, saves any columns needed for both the cut and the output to the *self.entry_list*.
- **in_memory** (*bool*) – if `True`, returns the generated entry list in memory.
- **output_file** (*str | None*) – HDF5 file name to write the entry list to.

Returns
entries – the entry list containing columns for `{parent}_idx`, `{parent}_table`, `{child}_idx` and output columns if applicable. Only returned if *in_memory* is `True`.

Return type
dict[int, *DataFrame*] | *DataFrame* | None

Note: Does *not* load the column information into memory. This is done by *load()*.

---

build_hit_entries(*save_output_columns: bool = False*, *in_memory: bool = True*, *output_file: str | None = None*) → dict[int, DataFrame] | DataFrame | None

Called by *build_entry_list()* to handle the case when *tcm_level* is unspecified.

Ignores any cuts set on levels above lowest level.

Parameters

- **save_output_columns** (*bool*) – If `True`, saves any columns needed for both the cut and the output to the entry list.
- **in_memory** (*bool*) – If `True`, returns the generated entry list in memory.
- **output_file** (*str | None*) – HDF5 file name to write the entry list to.

Returns
entries – the entry list containing columns for `{low_level}_idx`, `{low_level}_table`, and output columns if applicable. Only returned if *in_memory* is `True`.

Return type
dict[int, *DataFrame*] | *DataFrame* | None

get_file_list() → DataFrame

Returns a copy of the file database with its dataframe pared down to the current file list.

Return type
DataFrame

get_tiers_for_col(*columns: list | ndarray*, *merge_files: bool | None = None*) → dict

For each column given, get the tiers and tables in that tier where that column can be found.

Parameters
columns (*list | ndarray*) – the columns to look for.

Returns
col_tiers – `col_tiers[file]["tables"][tier]` gives a list of tables in `tier` that contain a column of interest. `col_tiers[file]["columns"][column]` gives the tier that `column` can be found in. If *self.merge_file`s then `col_tiers[tier]`* is a list of tables in `tier` that contain a column of interest.

Return type
dict

load(*entry_list: DataFrame | None = None*, *in_memory: bool = True*, *output_file: str | None = None*, *orientation: str = 'hit'*, *tcm_level: str | None = None*) → None | Table | Struct | DataFrame

Loads the requested data from disk.

Loads the requested columns in *self.output_columns* for the entries in the given *entry_list*.

Parameters

- **entry_list** (*DataFrame | None*) – the output of *build_entry_list()*. If None, builds it according to the current configuration.
- **in_memory** (*bool*) – if True, returns the loaded data in memory and stores in *self.data*.
- **output_file** (*str | None*) – if not None, writes the loaded data to the specified file.
- **orientation** (*str*) – specifies the orientation of the output table. Can be hit or evt.
- **tcm_level** (*str | None*) – which TCM was used to create the entry_list.

Returns
data – The data loaded from disk, as specified by *self.output_format*, *self.output_columns*, and *self.merge_files*. Only returned if *in_memory* is True.

Return type
None | *Table* | *Struct* | *DataFrame*

load_cal_pars(*query*)

access the cal_pars parameter database, run a query, and return some tables.

load_detector(*det_id*)

special version of *load* designed to retrieve all file files, tables, column names, and potentially calibration/dsp parameters relevant to one single detector.

load_dsp_pars(*query*)

access the dsp_pars parameter database (probably JSON format) and do some kind of query to retrieve parameters of interest for our file list, and return some tables.

load_evts(*entry_list: DataFrame | None = None*, *in_memory: bool = False*, *output_file: str | None = None*, *tcm_level: str | None = None*) → None | Table | Struct | DataFrame

Called by *load()* when orientation is evt.

Return type
None | *Table* | *Struct* | *DataFrame*

load_hits(*entry_list: DataFrame*, *in_memory: bool = False*, *output_file: str | None = None*, *tcm_level: str | None = None*) → None | Table | Struct | DataFrame

Called by *load()* when orientation is hit.

Return type
None | *Table* | *Struct* | *DataFrame*

load_iterator(*entry_list: DataFrame | None = None*, *tcm_level: str | None = None*, *buffer_len: int = 3200*) → LH5Iterator

Creates an :class:LH5Iterator that will load the requested columns in *self.output_columns* for the entries in the given *entry_list* in chunks. This is more memory efficient than filling a whole table and is recommended for use when loading waveforms.

Parameters

- **entry_list** (*DataFrame | None*) – the output of *build_entry_list()*. If None, builds it according to the current configuration.
- **tcm_level** (*str | None*) – which TCM was used to create the entry_list.
- **buffer_len** (*int*) – how many entries to load in a single chunk

Returns
data – LH5 Iterator, which yields (lh5 table, entry, n_entries) when iterated over.

Return type
LH5Iterator

load_settings()

get metadata stored in raw files, usually from a DAQ machine.

next(*entry_list: DataFrame | None = None*, *chunk_size: int = 10000*, ***kwargs*) → Iterator[Table | Struct | DataFrame]

Loads the requested data from disk in chunks.

This method should be used instead of *load()* to handle large data sets.

Note: It is a user responsibility to optimize the chunk size in order to achieve best performance.

Parameters

- **chunk_size** (*int*) – number of entries to load at each iteration. Adapt based on the size of each entry and the amount of memory available on the system.
- **entry_list** (*DataFrame | None*) – keyword argument forwarded to *load()*.
- ****kwargs** – keyword argument forwarded to *load()*.

Returns
data – see *load()*.

Return type
Iterator[*Table* | *Struct* | *DataFrame*]

Examples

```
>>> for chunk in dl.next():
>>>     # 'chunk' has the same type of the output of dl.load()
```

See also:

load

reset()

Resets all fields to their default values.

As if this is a newly created data loader.

set_config(*config: dict | str*) → None

Load configuration dictionary.

$_ expands to the config file location, if possible, otherwise the current working directory.

set_cuts(*cuts: dict | list*, *append: bool = False*) → None

Apply a selection on columns in the data tables.

Parameters

- **cut** – the cuts on the columns of the data table, e.g. `trapEftp_cal > 1000`. If passing a dictionary, the dictionary should be structured as `dict[tier] = cut_expr`. If passing a list, each item in the array should be able to be applied on one level of tables. The cuts at different levels will be joined with an AND.

- **append** (*bool*) – if True, appends cuts to the existing cuts instead of overwriting

Example

```
>>> dl.set_cuts({"raw": "daqenergy > 1000", "hit": "AoE > 3"})
```

set_datastreams(*ds: list | tuple | ndarray*, *word: str*, *append: bool = False*) → None

Apply selection on data streams (or channels).

Sets *self.table_list*.

Parameters

- **ds** (*list | tuple | ndarray*) – identifies the detectors of interest. Can be a list of detector names, serial numbers, or channels or a list of subsystems of interest e.g. `ged`.
- **word** (*str*) – the type of identifier used in ds. Should be a key in the given channel map or a word defined in the configuration file.
- **append** (*bool*) – if `True`, appends datastreams to the existing *self.table_list* instead of overwriting.

Example

```
>>> dl.set_datastreams(np.arange(40, 45), "ch")
```

set_files(*query: str | list[str]*, *append: bool = False*) → None

Apply a file selection.

Sets *self.file_list*, which is a list of indices corresponding to the rows in the file database.

Parameters

- **query** (*str | list[str]*) – if single string, defines an operation on the file database columns supported by `pandas.DataFrame.query()`. In addition, the `all` keyword is supported to select all files in the database. If list of strings, will be interpreted as key (cycle timestamp) list.
- **append** (*bool*) – if `True`, appends files to the existing *self.file_list* instead of overwriting.

Note: Call this function before any other operation. A second call to *set_files()* does not replace the current file list, which gets instead integrated with the new list. Use *reset()* to reset the file query.

Example

```
>>> dl.set_files("file_status == 26 and timestamp == '20220716T130443Z'")
```

set_output(*fmt: str | None = None*, *merge_files: bool | None = None*, *columns: list | None = None*, *aoesa_to_vov: bool | None = None*) → None

Set the parameters for the output format of load

Parameters

- **fmt** (*str | None*) – `lgdo.Table` or `pd.DataFrame`.

- **merge_files** (*bool | None*) – if `True`, information from multiple files will be merged into one table.
- **columns** (*list | None*) – the columns that should be copied into the output.
- **aoesa_to_vov** (*bool | None*) – output `ArrayOfEqualSizedArrays` as `VectorOfVectors`.

Example

```
>>> dl.set_output(
...   fmt="pd.DataFrame",
...   merge_files=False,
...   columns=["daqenergy", "trapEmax", "channel"]
... )
```

skim_waveforms(*mode: str = 'hit'*, *hit_list=None*, *evt_list=None*)

handle this one separately because waveforms can easily fill up memory.

pygama.flow.data_loader.**iskeyword**()

x.__contains__(y) <==> y in x.

pygama.flow.file_db module

Utilities for LH5 file inventory.

class pygama.flow.file_db.**FileDB**(*config: str | dict | list[str]*, *scan: bool = True*)

Bases: `object`

LH5 file database.

A class containing a `pandas.DataFrame` that has additional functions to scan the data directory, fill the dataframe's columns with information about each file, and read or write to disk in an LGDO format.

The database contains the following columns:

- file keys: the fields specified in the configuration file's `file_format` that are required to generate a file name e.g. `run`, `type`, `timestamp` etc.
- `{tier}_file`: generated file name for the tier.
- `{tier}_size`: size of file on disk, if applicable.
- `file_status`: contains a bit corresponding to whether or not a file for each tier exists for a given cycle e.g. If we have tiers *raw*, *dsp*, and *hit*, but only the *raw* file has been produced, `file_status` would be `0b100`.
- `{tier}_tables`: available data streams (channels) in the tier.
- `{tier}_col_idx`: `file_db.columns[{tier}_col_idx]` will return the list of columns available in the tier's file.

The database must be configured by a JSON file (or corresponding dictionary), which defines the data file names, paths and LH5 layout. For example:

```
{
    "data_dir": "prod-ref-l200/generated/tier",
    "tier_dirs": {
        "raw": "/raw",
        "dsp": "/dsp",
        "hit": "/hit",
        "tcm": "/tcm",
        "evt": "/evt"
    },
    "file_format": {
        "raw": "/{type}/{period}/{run}/{exp}-{period}-{run}-{type}-{timestamp}-tier_
↪raw.lh5",
        "dsp": "/{type}/{period}/{run}/{exp}-{period}-{run}-{type}-{timestamp}-tier_
↪dsp.lh5",
        "hit": "/{type}/{period}/{run}/{exp}-{period}-{run}-{type}-{timestamp}-tier_
↪hit.lh5",
        "evt": "/{type}/{period}/{run}/{exp}-{period}-{run}-{type}-{timestamp}-tier_
↪evt.lh5",
        "tcm": "/{type}/{period}/{run}/{exp}-{period}-{run}-{type}-{timestamp}-tier_
↪tcm.lh5"
    },
    "table_format": {
        "raw": "ch{ch:03d}/raw",
        "dsp": "ch{ch:03d}/dsp",
        "hit": "{ch}/hit",
        "evt": "{grp}/evt",
        "tcm": "hardware_tcm"
    },
    "tables": {
        "raw": [0, 1, 2, 4, 5, 6, 7],
        "dsp": [0, 1, 2, 4, 5, 6, 7],
        "hit": [0, 1, 2, 4, 5, 6, 7],
        "tcm": [""],
        "evt": [""]
    },
    "columns": {
        "raw": ["baseline", "waveform", "daqenergy"],
        "dsp": ["trapEftp", "AoE", "trapEmax"],
        "hit": ["trapEftp_cal", "trapEmax_cal"],
        "tcm": ["cumulative_length", "array_id", "array_idx"],
        "evt": ["lar_veto", "muon_veto", "ge_mult"]
    }
}
```

FileDB objects can be also stored on disk and read-in at later times.

Examples

```
>>> from pygama.flow import FileDB
>>> db = FileDB("./filedb_config.json")
>>> db.scan_tables_columns()  # read in also table columns names
>>> print(db)
<< Columns >>
[['baseline', 'card', 'ch_orca', 'channel', 'crate', 'daqenergy', 'deadtime', 'dr_
↪maxticks', 'dr_start_pps', 'dr_start_ticks', 'dr_stop_pps', 'dr_stop_ticks',
↪'eventnumber', 'fcid', 'numtraces', 'packet_id', 'runtime', 'timestamp', 'to_abs_
↪mu_usec', 'to_dt_mu_usec', 'to_master_sec', 'to_mu_sec', 'to_mu_usec', 'to_start_
↪sec', 'to_start_usec', 'tracelist', 'ts_maxticks', 'ts_pps', 'ts_ticks', 'waveform
↪'], ['bl_intercept', 'bl_mean', 'bl_slope', 'bl_std', 'tail_slope', 'tail_std',
↪'wf_blsub'], ['array_id', 'array_idx', 'cumulative_length']]
<< DataFrame >>
   exp period   run         timestamp type  ... hit_col_idx tcm_tables tcm_col_idx␣
↪evt_tables evt_col_idx
0  l60    p01  r014  20220716T105236Z  cal  ...        None         []         [2] ␣
↪     None        None
1  l60    p01  r014  20220716T104550Z  cal  ...        None         []         [2] ␣
↪     None        None
>>> db.to_disk("file_db.lh5")
```

Parameters

- **config** (*str | dict | list[str]*) – dictionary or path to JSON file specifying data directories, tiers, and file name templates. Can also be path (or list of paths or regular expression) to existing LH5 file containing *FileDB* object serialized by *to_disk()*.
- **scan** (*bool*) – whether the file database should scan the directory containing *raw* files to fill its rows with file keys.

from_disk(*path: str | list[str]*) → None

Read FileDBs from disk.

Overrides the dataframe, configuration dictionary and columns with the information from a file created by *to_disk()*.

Parameters

path (*str | list[str]*) – file or file pattern (or list of the latter).

get_table_columns(*table: str | int*, *tier: str*, *ifile: int = 0*) → list[str]

Return list of columns in table *table*, tier *tier*.

Assumes that the table contents do not change across data files. If desired, *ifile* (default is 0) can be used to select a different file.

Return type

list[str]

get_table_name(*tier: str*, *tb: str*) → str

Get the table name for a tier given its table identifier.

Parameters

- **tier** (*str*) – specify the tier whose table format will be used.
- **tb** (*str*) – the table identifier that will be passed to the table format.

Returns
table_name – the name of the table in *tier* with table identifier *tb*

Return type
str

scan_daq_files(*daq_dir: str*, *daq_template: str*) → None

Does the exact same thing as *scan_files()* but with extra configuration arguments for a DAQ directory and template instead of using the lowest tier.

scan_files(*dirs: list[str] | None = None*) → None

Scan the directory containing files from the lowest tier and fill the dataframe.

The lowest tier is defined as the first element of the *tiers* array. Only fills columns that can be populated with just these files.

Parameters
dirs (*list[str] | None*) – restrict search to this list of directories. Specified paths can be absolute, relative to *self.data_dir* or relative to the root directory of the lowest-tier files. If `None`, the whole root lowest-tier directory is scanned. Useful to build a partial database.

scan_tables_columns(*to_file: str | None = None*, *override: bool = False*, *dir_files_conform: bool = False*) → list[str]

Open files to read (and store) available tables (and columns therein) names.

Adds the available table names in each tier as a column in the dataframe by searching for group names that match the configured `table_format` and saving the associated keyword values.

Returns a list with each unique list of columns found in each table and adds a column `{tier}_col_idx` to the dataframe that maps to the column table.

Parameters

- **to_file** (*str | None*) – Optionally write the column table to an LH5 file (as a `VectorOfVectors`).
- **override** (*bool*) – If the *FileDB* already has a *columns* field, the scan will not run unless this parameter is set to `True`.
- **dir_files_conform** (*bool*) – if `True`, assume that all files in a directory contain tables with the same columns (i.e. all file contents conform to the same format) and scan only the first file. Significantly reduces processing time.

Return type
list[str]

set_config(*config: dict*, *config_path: str | None = None*) → None

Read in the configuration dictionary.

set_file_sizes() → None

Add columns for each tier containing the corresponding file size in bytes.

As reported by `os.path.getsize()`.

set_file_status() → None

Add a column with a bit corresponding to whether each tier's file exists.

For example, if we have tiers *raw*, *dsp*, and *hit*, but only the *raw* file has been produced, `file_status` would be 4 (`0b100` in binary representation).

to_disk(*filename: str*, *wo_mode='write_safe'*) → None

Serializes database to disk.

Parameters

- **filename** (*str*) – output LH5 file name.
- **wo_mode** – passed to `write()`.

pygama.flow.utils module

Utility functions.

pygama.flow.utils.**dict_to_table**(*col_dict: dict*, *attr_dict: dict*)

pygama.flow.utils.**fill_col_dict**(*tier_table: Table*, *col_dict: dict*, *attr_dict: dict*, *tcm_idx: list | RangeIndex*, *table_length: int*, *aoesa_to_vov: bool*)

pygama.flow.utils.**inplace_sort**(*df: DataFrame*, *by: str*) → None

pygama.flow.utils.**to_datetime**(*key: str*) → datetime

Convert LEGEND cycle key to `datetime`.

Assumes *key* is formatted as `YYYYMMDDTHHMMSSZ` (UTC).

Return type
datetime

pygama.flow.utils.**to_unixtime**(*key: str*) → int

Convert LEGEND cycle key to POSIX timestamp.

Return type
int

pygama.hit package

Routines for applying columnar transformations to tabular data. Specifically, to produce the hit-tier from the dsp-tier.

Submodules

pygama.hit.build_hit module

This module implements routines to evaluate expressions to columnar data.

pygama.hit.build_hit.**_reorder_table_operations**(*config: Mapping[str, Mapping]*) → OrderedDict[str, Mapping]

Reorder operations in *config* according to mutual dependency.

Return type
OrderedDict[str, *Mapping*]

pygama.hit.build_hit.**build_hit**(*infile: str*, *outfile: str | None = None*, *hit_config: str | Mapping | None = None*, *lh5_tables: Iterable[str] | None = None*, *lh5_tables_config: str | Mapping[str, Mapping] | None = None*, *n_max: int = inf*, *wo_mode: str = 'write_safe'*, *buffer_len: int = 3200*) → None

Transform a `Table` into a new `Table` by evaluating strings describing column operations.

Operates on columns only, not specific rows or elements. Relies on `eval()`.

Parameters

- **infile** (*str*) – input LH5 file name containing tables to be processed.
- **outfile** (*str | None*) – name of the output LH5 file. If `None`, create a file in the same directory and append *_hit* to its name.
- **hit_config** (*str | Mapping | None*) – dictionary or name of JSON file defining column transformations. Must contain an `outputs` and an `operations` keys. For example:

```
{
  "outputs": ["calE", "AoE"],
  "operations": {
    "calE": {
      "expression": "sqrt(a + b * trapEmax**2)",
      "parameters": {"a": "1.23", "b": "42.69"},
    },
    "AoE": {"expression": "A_max/calE"},
  }
}
```

  The `outputs` array lists columns that will be effectively written in the output LH5 file. Add here columns that will be simply forwarded as they are from the DSP tier.
- **lh5_tables** (*Iterable[str] | None*) – tables to consider in the input file. if `None`, tables with name *dsp* will be searched for in the file, even nested by one level.
- **lh5_tables_config** (*str | Mapping[str, Mapping] | None*) – dictionary or JSON file defining the mapping between LH5 tables in *infile* and hit configuration. Table names can be directly mapped to configuration blocks or to JSON files containing them. This option is mutually exclusive with *hit_config* and *lh5_tables*.
- **n_max** (*int*) – maximum number of rows to process
- **wo_mode** (*str*) – forwarded to `lgdo.lh5.store.LH5Store.write()`.

See also:

`lgdo.types.table.Table.eval`

pygama.lgdo package

pygama.math package

Statistical and mathematical utilities.

Submodules

pygama.math.histogram module

pygama convenience functions for 1D histograms.

1D hists in pygama require 3 things available from all implementations of 1D histograms of numerical data in python: hist, bins, and var: - hist: an array of histogram values - bins: an array of bin edges - var: an array of variances in each bin If var is not provided, pygama assuems that the hist contains "counts" with variance = counts (Poisson stats)

These are just convenience functions, provided for your convenience. Hopefully they will help you if you need to do something trickier than is provided (e.g. 2D hists).

pygama.math.histogram.**better_int_binning**(*x_lo=0*, *x_hi=None*, *dx=None*, *n_bins=None*)

Get a good binning for integer data.

Guarantees an integer bin width.

At least two of x_hi, dx, or n_bins must be provided.

Parameters

- **x_lo** (*float*) – Desired low x value for the binning
- **x_hi** (*float*) – Desired high x value for the binning
- **dx** (*float*) – Desired bin width
- **n_bins** (*float*) – Desired number of bins

Returns

- **x_lo** (*int*) – int values for best x_lo
- **x_hi** (*int*) – int values for best x_hi, returned if x_hi is not None
- **dx** (*int*) – best int bin width, returned if arg dx is not None
- **n_bins** (*int*) – best int n_bins, returned if arg n_bins is not None

pygama.math.histogram.**find_bin**(*x*, *bins*)

Returns the index of the bin containing x Returns -1 for underflow, and len(bins) for overflow For uniform bins, jumps directly to the appropriate index. For non-uniform bins, binary search is used.

pygama.math.histogram.**get_bin_centers**(*bins*)

Returns an array of bin centers from an input array of bin edges. Works for non-uniform binning. Note: a new array is allocated

Parameters:

pygama.math.histogram.**get_bin_widths**(*bins*)

Returns an array of bin widths from an input array of bin edges. Works for non-uniform binning.

pygama.math.histogram.**get_fwfm**(*fraction*, *hist*, *bins*, *var=None*, *mx=None*, *dmx=0*, *bl=0*, *dbl=0*, *method='bins_over_f'*, *n_slope=3*)

Estimate the full width at some fraction of the max of data in a histogram

Typically used by sending slices around a peak. Searches about argmax(hist) for the peak to fall by [fraction] from mx to bl

Parameters

- **fraction** (*float*) – The fractional amplitude at which to evaluate the full width
- **hist** (*array-like*) – The histogram data array containing the peak
- **bins** (*array-like*) – An array of bin edges for the histogram
- **var** (*array-like (optional)*) – An array of histogram variances. Used with the 'fit_slopes' method
- **mx** (*float or tuple(float, float) (optional)*) – The value to use for the max of the peak. If None, np.amax(hist) is used.
- **dmx** (*float (optional)*) – The uncertainty in mx
- **bl** (*float or tuple (float, float) (optional)*) – Used to specify an offset from which to estimate the FWFM.
- **dbl** (*float (optional)*) – The uncertainty in the bl
- **method** (*string*) –

  'bins_over_f'
  [the simplest method: just take the difference in the bin] centers that are over [fraction] of max. Only works for high stats and FWFM/bin_width >> 1

  'interpolate'
  [interpolate between the bins that cross the [fration]] line. Works well for high stats and a reasonable number of bins. Uncertainty incorporates var, if provided.

  'fit_slopes'
  [fit over n_slope bins in the vicinity of the FWFM and] interpolate to get the fractional crossing point. Works okay even when stats are moderate but requires enough bins that dx traversed by n_slope bins is approximately linear. Incorporates bin variances in fit and final uncertainties if provided.

- **n_slope** (*int*) – DOCME

Returns

fwfm, dfwfm (*float, float*) – fwfm: the full width at [fraction] of the maximum above bl dfwfm: the uncertainty in fwfm

Examples

```
>>> import pygama.analysis.histograms as pgh
>>> from numpy.random import normal
>>> hist, bins, var = pgh.get_hist(normal(size=10000), bins=100, range=(-5,5))
>>> pgh.get_fwfm(0.5, hist, bins, var, method='bins_over_f')
(2.2, 0.15919638684132664) # may vary
```

```
>>> pgh.get_fwfm(0.5, hist, bins, var, method='interpolate')
(2.2041666666666666, 0.09790931254396479) # may vary
```

```
>>> pgh.get_fwfm(0.5, hist, bins, var, method='fit_slopes')
(2.3083363869003466, 0.10939486522749278) # may vary
```

pygama.math.histogram.**get_fwhm**(*hist*, *bins*, *var=None*, *mx=None*, *dmx=0*, *bl=0*, *dbl=0*, *method='bins_over_f'*, *n_slope=3*)

Estimate the FWHM of data in a histogram

See also:

get_fwfm
for parameters and return values

pygama.math.histogram.**get_gaussian_guess**(*hist*, *bins*)

given a hist, gives guesses for mu, sigma, and amplitude

pygama.math.histogram.**get_hist**(*data*, *bins=None*, *range=None*, *dx=None*, *wts=None*)

return hist, bins, var after binning data

This is just a wrapper for numpy.histogram, with optional weights for each element and proper computing of variances.

Note: there are no overflow / underflow bins.

Available binning methods:

- Default (no binning arguments) : 100 bins over an auto-detected range
- bins=N, range=(x_lo, x_hi) : N bins over the specified range (or leave range=None for auto-detected range)
- bins=[str] : use one of np.histogram's automatic binning algorithms
- bins=bin_edges_array : array lower bin edges, supports non-uniform binning
- dx=dx, range=(x_lo, x_hi): bins of width dx over the specified range. Note: dx overrides the bins argument!

Parameters

- **data** (*array like*) – The array of data to be histogrammed
- **bins** (*int, array, or str (optional)*) – int: the number of bins to be used in the histogram array: an array of bin edges to use str: the name of the np.histogram automatic binning algorithm to use If not provided, np.histogram's default auto-binning routine is used
- **range** (*tuple (float, float) (optional)*) – (x_lo, x_high) is the tuple of low and high x values to uses for the very ends of the bin range. If not provided, np.histogram chooses the ends based on the data in data
- **dx** (*float (optional)*) – Specifies the bin width. Overrides "bins" if both arguments are present
- **wts** (*float or array like (optional)*) – Array of weights for each bin. For example, if you want to divide all bins by a time T to get the bin contents in count rate, set wts = 1/T. Variances will be computed for each bin that appropriately account for each data point's weighting.

Returns

- **hist** (*array*) – the values in each bin of the histogram
- **bins** (*array*) – an array of bin edges: bins[i] is the lower edge of the ith bin. Note: it includes the upper edge of the last bin and does not include underflow or overflow bins. So len(bins) = len(hist) + 1

- **var** (*array*) – array of variances in each bin of the histogram

pygama.math.histogram.**plot_hist**(*hist*, *bins*, *var=None*, *show_stats=False*, *stats_hloc=0.75*, *stats_vloc=0.85*, *fill=False*, *fillcolor='r'*, *fillalpha=0.2*, ***kwargs*)

plot a step histogram, with optional error bars

pygama.math.histogram.**range_slice**(*x_min*, *x_max*, *hist*, *bins*, *var=None*)

pygama.math.peak_fitting module

pygama.math.peak_fitting.**Am_double**(*x*, *n_sig1*, *mu1*, *sigma1*, *n_sig2*, *mu2*, *sigma2*, *n_sig3*, *mu3*, *sigma3*, *n_bkg1*, *hstep1*, *n_bkg2*, *hstep2*, *lower_range=inf*, *upper_range=inf*, *components=False*)

A Fit function exclusevly for a 241Am 99keV and 103keV lines situation Consists of

- three gaussian peaks (two lines + one bkg line in between)
- two steps (for the two lines)
- two tails (for the two lines)

pygama.math.peak_fitting.**cal_slope**(*x*, *m1*, *m2*)

Fit the calibration values

pygama.math.peak_fitting.**double_gauss_pdf**(*x*, *n_sig1*, *mu1*, *sigma1*, *n_sig2*, *mu2*, *sigma2*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

A Fit function exclusevly for a 133Ba 81keV peak situation Consists of

- two gaussian peaks (two lines)
- one step

pygama.math.peak_fitting.**extended_Am_double**(*x*, *n_sig1*, *mu1*, *sigma1*, *n_sig2*, *mu2*, *sigma2*, *n_sig3*, *mu3*, *sigma3*, *n_bkg1*, *hstep1*, *n_bkg2*, *hstep2*, *lower_range=inf*, *upper_range=inf*, *components=False*)

pygama.math.peak_fitting.**extended_double_gauss_pdf**(*x*, *n_sig1*, *mu1*, *sigma1*, *n_sig2*, *mu2*, *sigma2*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

A Fit function exclusevly for a 133Ba 81keV peak situation Consists of

- two gaussian peaks (two lines)
- one step

pygama.math.peak_fitting.**extended_gauss_pdf**(*x*, *mu*, *sigma*, *n_sig*)

Basic Gaussian pdf args; mu, sigma, n_sig (number of signal events)

pygama.math.peak_fitting.**extended_gauss_step_pdf**(*x*, *n_sig*, *mu*, *sigma*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

Pdf for Gaussian on step background for Compton spectrum, returns also the total number of events for extended unbinned fits args: n_sig mu, sigma for the signal and n_bkg, hstep for the background

pygama.math.peak_fitting.**extended_radford_pdf**(*x*, *n_sig*, *mu*, *sigma*, *htail*, *tau*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

Pdf for gaussian with tail signal and step background, also returns number of events

pygama.math.peak_fitting.**fit_binned**(*func*, *hist*, *bins*, *var=None*, *guess=None*, *cost_func='LL'*, *Extended=True*, *simplex=False*, *bounds=None*, *fixed=None*)

Do a binned fit to a histogram.

Default is Extended Log Likelihood fit, with option for either Least Squares or other cost function.

Parameters

- **func** (*the function to fit, if using LL as method needs to be a cdf*) –
- **hist** (*histogrammed data*) –
- **bins** (*histogrammed data*) –
- **var** (*histogrammed data*) –
- **guess** (*initial guess parameters*) –
- **cost_func** (*cost function to use*) –
- **Extended** (*run extended or non extended fit*) –
- **simplex** (*whether to include a round of simpson minimisation before main minimisation*) –
- **bounds** (*list of tuples with bounds can be None, e.g. [(0,None), (0, 10)]*) –
- **fixed** (*list of parameter indices to fix*) –

Returns

- **coeff** (*array*)
- **error** (*array*)
- **cov_matrix** (*array*)

pygama.math.peak_fitting.**fit_hist**(*func*, *hist*, *bins*, *var=None*, *guess=None*, *poissonLL=False*, *integral=None*, *method=None*, *bounds=None*)

Deprecated since version 0.8: Replaced by *fit_binned()*. Will be removed in future releases.

do a binned fit to a histogram (nonlinear least squares). can either do a poisson log-likelihood fit (jason's favourite) or use curve_fit w/ an arbitrary function.

Parameters

- **func** – function to be fitted
- **hist** – as in return value of pygama.histograms.get_hist()
- **bins** – as in return value of pygama.histograms.get_hist()
- **var** – as in return value of pygama.histograms.get_hist()
- **guess** – initial parameter guesses. Should be optional – we can auto-guess for many common functions. But not yet implemented.
- **poissonLL** – use Poisson stats instead of the Gaussian approximation in each bin. Requires integer stats. You must use parameter bounds to make sure that func does not go negative over the x-range of the histogram.
- **integral** – DOCME
- **method** – options to pass to scipy.optimize.minimize()
- **bounds** – options to pass to scipy.optimize.minimize()

Returns
coeff, cov_matrix (*tuple(array, matrix)*)

pygama.math.peak_fitting.**fit_unbinned**(*func*, *data*, *guess=None*, *Extended=True*, *cost_func='LL'*, *simplex=False*, *bounds=None*, *fixed=None*)

Do a unbinned fit to data. Default is Extended Log Likelihood fit, with option for other cost functions.

Parameters

- **func** (*the function to fit*) –
- **data** (*the data*) –
- **guess** (*initial guess parameters*) –
- **Extended** (*run extended or non extended fit*) –
- **cost_func** (*cost function to use*) –
- **simplex** (*whether to include a round of simpson minimisation before main minimisation*) –
- **bounds** (*list of tuples with bounds can be None, e.g. [(0,None), (0, 10)]*) –
- **fixed** (*list of parameter indices to fix*) –
- **coeff** (*tuple(array, matrix)*) –
- **cov_matrix** (*tuple(array, matrix)*) –

@numba.jit pygama.math.peak_fitting.**gauss**(*x*, *mu*, *sigma*)

Gaussian, unnormalised for use in building pdfs, w/ args: mu, sigma.

@numba.jit pygama.math.peak_fitting.**gauss_amp**(*x*, *mu*, *sigma*, *a*)

Gaussian with height as a parameter for fwhm etc. args mu sigma, amplitude

@numba.jit pygama.math.peak_fitting.**gauss_cdf**(*x*, *mu*, *sigma*)

gaussian cdf, w/ args: mu, sigma.

pygama.math.peak_fitting.**gauss_linear**(*x*, *n_sig*, *mu*, *sigma*, *n_bkg*, *b*, *m*, *components=False*)

gaussian signal + linear background function args: n_sig mu, sigma for the signal and n_bkg,b,m for the background

pygama.math.peak_fitting.**gauss_mode**(*hist*, *bins*, ***kwargs*)

Alias for gauss_mode_max that just returns the mode (position) of a peak

See also:

gauss_mode_max, *gauss_mode_width_max*

Returns

- **mode** (*the estimated x-position of the maximum*)
- **dmode** (*the uncertainty in the mode*)

pygama.math.peak_fitting.**gauss_mode_max**(*hist*, *bins*, ***kwargs*)

Alias for gauss_mode_width_max that just returns the max and mode

See also:

gauss_mode_width_max

Returns

- **pars** (*ndarray with the parameters (mode, maximum) of the gaussian fit*) – mode : the estimated x-position of the maximum maximum : the estimated maximum value of the peak
- **cov** (*2x2 ndarray of floats*) – The covariance matrix for the 2 parameters in pars

Examples

```
>>> import pygama.math.histogram as pgh
>>> from numpy.random import normal
>>> import pygama.math.peak_fitting as pgf
>>> hist, bins, var = pgh.get_hist(normal(size=10000), bins=100, range=(-5,5))
>>> pgf.gauss_mode_max(hist, bins, var=var, n_bins=20)
```

pygama.math.peak_fitting.**gauss_mode_width_max**(*hist*, *bins*, *var=None*, *mode_guess=None*, *n_bins=5*, *cost_func='Least Squares'*, *inflate_errors=False*, *gof_method='var'*)

Get the max, mode, and width of a peak based on gauss fit near the max Returns the parameters of a gaussian fit over n_bins in the vicinity of the maximum of the hist (or the max near mode_guess, if provided). This is equivalent to a Taylor expansion around the peak maximum because near its maximum a Gaussian can be approximated by a 2nd-order polynomial in x:

$$A\exp[-(x-\mu)^2/2\sigma^2] \simeq A[1-(x-\mu)^2/2\sigma^2] = A-(1/2!)(A/\sigma^2)(x-\mu)^2$$

The advantage of using a gaussian over a polynomial directly is that the gaussian parameters are the ones we care about most for a peak, whereas for a poly we would have to extract them after the fit, accounting for covariances. The gaussian also better approximates most peaks farther down the peak. However, the gauss fit is nonlinear and thus less stable.

Parameters

- **hist** (*array-like*) – The values of the histogram to be fit
- **bins** (*array-like*) – The bin edges of the histogram to be fit
- **var** (*array-like (optional)*) – The variances of the histogram values. If not provided, square-root variances are assumed.
- **mode_guess** (*float (optional)*) – An x-value (not a bin index!) near which a peak is expected. The algorithm fits around the maximum within +/- n_bins of the guess. If not provided, the center of the max bin of the histogram is used.
- **n_bins** (*int (optional)*) – The number of bins (including the max bin) to be used in the fit. Also used for searching for a max near mode_guess
- **cost_func** (*str (optional)*) – Passed to fit_binned()
- **inflate_errors** (*bool (optional)*) – If true, the parameter uncertainties are inflated by sqrt(chi2red) if it is greater than 1
- **gof_method** (*str (optional)*) – method flag for goodness_of_fit

Returns

- **pars** (*ndarray containing the parameters (mode, sigma, maximum) of the gaussian fit*) –
  - mode : the estimated x-position of the maximum

  - sigma : the estimated width of the peak. Equivalent to a gaussian width (sigma), but based only on the curvature within n_bins of the peak. Note that the Taylor-approxiamted curvature of the underlying function in the vicinity of the max is given by max / sigma^2
  - maximum : the estimated maximum value of the peak
- **cov** (*3x3 ndarray of floats*) – The covariance matrix for the 3 parameters in pars

@numba.jit pygama.math.peak_fitting.**gauss_norm**(*x*, *mu*, *sigma*)

Normalised Gaussian, w/ args: mu, sigma.

@numba.jit pygama.math.peak_fitting.**gauss_pdf**(*x*, *mu*, *sigma*, *n_sig*)

Basic Gaussian pdf args; mu, sigma, n_sig (number of signal events)

pygama.math.peak_fitting.**gauss_step_cdf**(*x*, *n_sig*, *mu*, *sigma*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

Cdf for Gaussian on step background args: n_sig mu, sigma for the signal and n_bkg,hstep for the background

pygama.math.peak_fitting.**gauss_step_pdf**(*x*, *n_sig*, *mu*, *sigma*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

Pdf for Gaussian on step background args: n_sig mu, sigma for the signal and n_bkg,hstep for the background

@numba.jit pygama.math.peak_fitting.**gauss_tail_approx**(*x*, *mu*, *sigma*, *tau*)

@numba.jit pygama.math.peak_fitting.**gauss_tail_cdf**(*x*, *mu*, *sigma*, *tau*, *lower_range=inf*, *upper_range=inf*)

CDF for gaussian tail

@numba.jit pygama.math.peak_fitting.**gauss_tail_exact**(*x*, *mu*, *sigma*, *tau*)

@numba.jit pygama.math.peak_fitting.**gauss_tail_integral**(*x*, *mu*, *sigma*, *tau*)

Integral for gaussian tail

@numba.jit pygama.math.peak_fitting.**gauss_tail_norm**(*x*, *mu*, *sigma*, *tau*, *lower_range=inf*, *upper_range=inf*)

Normalised gauss tail. Note: this is only needed when the fitting range does not include the whole tail

@numba.jit pygama.math.peak_fitting.**gauss_tail_pdf**(*x*, *mu*, *sigma*, *tau*)

A gaussian tail function template Can be used as a component of other fit functions w/args mu,sigma,tau

pygama.math.peak_fitting.**gauss_uniform**(*x*, *n_sig*, *mu*, *sigma*, *n_bkg*, *components=False*)

define a gaussian signal on a uniform background, args: n_sig mu, sigma for the signal and n_bkg for the background

pygama.math.peak_fitting.**gauss_with_tail_cdf**(*x*, *mu*, *sigma*, *htail*, *tau*, *components=False*)

Cdf for gaussian with tail

pygama.math.peak_fitting.**gauss_with_tail_pdf**(*x*, *mu*, *sigma*, *htail*, *tau*, *components=False*)

Pdf for gaussian with tail

pygama.math.peak_fitting.**get_bin_estimates**(*pars*, *func*, *hist*, *bins*, *integral=None*, ***kwargs*)

Bin expected means are estimated by f(bin_center)*bin_width. Supply an integrating function to compute the integral over the bin instead. TODO: make default integrating function a numerical method that is off by default.

pygama.math.peak_fitting.**get_fwhm_func**(*func*, *pars*, *cov=None*)

pygama.math.peak_fitting.**get_mu_func**(*func*, *pars*, *cov=None*, *errors=None*)

pygama.math.peak_fitting.**get_total_events_func**(*func*, *pars*, *cov=None*, *errors=None*)

pygama.math.peak_fitting.**goodness_of_fit**(*hist*, *bins*, *var*, *func*, *pars*, *method='var'*, *scale_bins=False*)

Compute chisq and dof of fit

Parameters

- **hist** (*array, array, array or None*) – histogram data. var can be None if hist is integer counts
- **bins** (*array, array, array or None*) – histogram data. var can be None if hist is integer counts
- **var** (*array, array, array or None*) – histogram data. var can be None if hist is integer counts
- **func** (*function*) – the function that was fit to the hist
- **pars** (*array*) – the best-fit pars of func. Assumes all pars are free parameters
- **method** (*str*) – Sets the choice of "denominator" in the chi2 sum 'var': user passes in the variances in var (must not have zeros) 'Pearson': use func (hist must contain integer counts) 'Neyman': use hist (hist must contain integer counts and no zeros)

Returns

- **chisq** (*float*) – the summed up value of chisquared
- **dof** (*int*) – the number of degrees of freedom

@numba.jit pygama.math.peak_fitting.**nb_erf**(*x*)

Numba version of error function

@numba.jit pygama.math.peak_fitting.**nb_erfc**(*x*)

Numba version of complementary error function

pygama.math.peak_fitting.**poisson_gof**(*pars*, *func*, *hist*, *bins*, *integral=None*, ***kwargs*)

The Poisson likelihood does not give a good GOF until the counts are very high and all the poisson stats are roughly gaussian and you don't need it anyway. But the G.O.F. is calculable for the Poisson likelihood. So we do it here.

pygama.math.peak_fitting.**poly**(*x*, *pars*)

A polynomial function with pars following the polyfit convention

pygama.math.peak_fitting.**radford_cdf**(*x*, *n_sig*, *mu*, *sigma*, *htail*, *tau*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

Cdf for gaussian with tail signal and step background

pygama.math.peak_fitting.**radford_fwhm**(*sigma*, *htail*, *tau*, *cov=None*)

Return the FWHM of the radford_peak function, ignoring background and step components. If calculating error also need the normalisation for the step function.

pygama.math.peak_fitting.**radford_parameter_gradient**(*E*, *pars*, *step_norm*)

pygama.math.peak_fitting.**radford_pdf**(*x*, *n_sig*, *mu*, *sigma*, *htail*, *tau*, *n_bkg*, *hstep*, *lower_range=inf*, *upper_range=inf*, *components=False*)

David Radford's HPGe peak shape PDF consists of a gaussian with tail signal on a step background

pygama.math.peak_fitting.**radford_peakshape_derivative**(*E*, *pars*, *step_norm*)

@numba.jit pygama.math.peak_fitting.**step_cdf**(*x*, *mu*, *sigma*, *hstep*, *lower_range=inf*, *upper_range=inf*)

CDF for step function w/args mu, sigma, hstep

@numba.jit pygama.math.peak_fitting.**step_int**(*x*, *mu*, *sigma*, *hstep*)

Integral of step function w/args mu, sigma, hstep

@numba.jit pygama.math.peak_fitting.**step_pdf**(*x*, *mu*, *sigma*, *hstep*, *lower_range=inf*, *upper_range=inf*)

Normalised step function w/args mu, sigma, hstep Can be used as a component of other fit functions

pygama.math.peak_fitting.**taylor_mode_max**(*hist*, *bins*, *var=None*, *mode_guess=None*, *n_bins=5*, *poissonLL=False*)

Get the max and mode of a peak based on Taylor exp near the max Returns the amplitude and position of a peak based on a poly fit over n_bins in the vicinity of the maximum of the hist (or the max near mode_guess, if provided)

Parameters

- **hist** (*array-like*) – The values of the histogram to be fit. Often: send in a slice around a peak
- **bins** (*array-like*) – The bin edges of the histogram to be fit
- **var** (*array-like (optional)*) – The variances of the histogram values. If not provided, square-root variances are assumed.
- **mode_guess** (*float (optional)*) – An x-value (not a bin index!) near which a peak is expected. The algorithm fits around the maximum within +/- n_bins of the guess. If not provided, the center of the max bin of the histogram is used.
- **n_bins** (*int*) – The number of bins (including the max bin) to be used in the fit. Also used for searching for a max near mode_guess
- **poissonLL** – DOCME

Returns

- **pars** (*2-tuple with the parameters (mode, max) of the fit*) – mode : the estimated x-position of the maximum maximum : the estimated maximum value of the peak
- **cov** (*2x2 matrix of floats*) – The covariance matrix for the 2 parameters in pars

Examples

```
>>> import pygama.analysis.histograms as pgh
>>> from numpy.random import normal
>>> import pygama.analysis.peak_fitting as pgf
>>> hist, bins, var = pgh.get_hist(normal(size=10000), bins=100, range=(-5,5))
>>> pgf.taylor_mode_max(hist, bins, var, n_bins=5)
```

@numba.jit pygama.math.peak_fitting.**unnorm_step_pdf**(*x*, *mu*, *sigma*, *hstep*)

Unnormalised step function for use in pdfs

pygama.math.peak_fitting.**xtalball**(*x*, *mu*, *sigma*, *A*, *beta*, *m*)

power-law tail plus gaussian https://en.wikipedia.org/wiki/Crystal_Ball_function

pygama.math.units module

pygama.math.utils module

pygama convenience functions.

pygama.math.utils.**fit_simple_scaling**(*x*, *y*, *var=1*)

Fast computation of weighted linear least squares fit to a simple scaling

I.e. y = scale * x. Returns the best fit scale parameter and its variance.

Parameters

- **x** (*array like*) – x values for the fit
- **y** (*array like*) – y values for the fit
- **var** (*array like (optional)*) – The variances for each y-value

Returns

scale, scale_var (*tuple (float, float)*) – The scale parameter and its variance

pygama.math.utils.**get_dataset_from_cmdline**(*args*, *run_db*, *cal_db*)

make it easier to call this from argparse:

arg("-ds", nargs='*', action="store", help="load runs for a DS") arg("-r", "–run", nargs=1, help="load a single run")

pygama.math.utils.**get_formatted_stats**(*mean*, *sigma*, *ndigs=2*)

convenience function for formatting mean +/- sigma to the right number of significant figures.

pygama.math.utils.**get_par_names**(*func*)

Return a list containing the names of the arguments of "func" other than the first argument. In pygamaland, those are the function's "parameters."

pygama.math.utils.**linear_fit_by_sums**(*x*, *y*, *var=1*)

Fast computation of weighted linear least squares fit to a linear model

Note: doesn't compute covariances. If you want covariances, just use polyfit

Parameters

- **x** (*array like*) – x values for the fit
- **y** (*array like*) – y values for the fit
- **var** (*array like (optional)*) – The variances for each y-value

Returns

(m, b) (*tuple (float, float)*) – The slope (m) and y-intercept (b) of the best fit (in the least-squares sense) of the data to y = mx + b

pygama.math.utils.**peakdet**(*v*, *delta*, *x=None*)

Converted from MATLAB script at: http://billauer.co.il/peakdet.html Returns two arrays: [maxtab, mintab] = peakdet(v, delta, x) An updated (vectorized) version is in pygama.dsp.transforms.peakdet

pygama.math.utils.**plot_func**(*func*, *pars*, *range=None*, *npx=None*, ***kwargs*)

plot a function. take care of the x-axis points automatically, or user can specify via range and npx arguments.

pygama.math.utils.**print_fit_results**(*pars*, *cov*, *func=None*, *title=None*, *pad=True*)

convenience function for scipy.optimize.curve_fit results

pygama.math.utils.**set_plot_style**(*style*)

Choose a pygama plot style. Current options: 'clint', 'root' Or add your own [label].mpl file in the pygama directory!

pygama.math.utils.**sh**(*cmd*, *sh=False*)

input a shell command as you would type it on the command line.

pygama.math.utils.**sizeof_fmt**(*num*, *suffix='B'*)

given a file size in bytes, output a human-readable form.

pygama.math.utils.**tree_draw**(*tree*, *vars*, *tcut*)

if you have to debase yourself and use ROOT, this is an easy convenience function for quickly extracting data from TTrees. TTree::Draw can only handle groups of 4 variables at a time, but here we can put in as many as we want, and return a list of numpy.ndarrays for each one

pygama.pargen package

Utilities to generate and optimize parameters of interest from data (e.g. calibration routines)

Submodules

pygama.pargen.AoE_cal module

This module provides functions for correcting the a/e energy dependence, determining the cut level and calculating survival fractions.

class pygama.pargen.AoE_cal.**PDF**

Bases: object

Base class for A/E pdfs.

_replace_values(***kwargs*)

pdf()

class pygama.pargen.AoE_cal.**cal_aoe**(*cal_dicts: dict = {}*, *cal_energy_param: str = 'cuspEmax_ctc_cal'*, *eres_func: callable = <function cal_aoe.<lambda>>*, *pdf=<class 'pygama.pargen.AoE_cal.standard_aoe'>*, *selection_string: str = ''*, *dt_corr: bool = False*, *dep_acc: float = 0.9*, *dep_correct: bool = False*, *dt_cut: dict | None = None*, *dt_param: str = 'dt_eff'*, *high_cut_val: int = 3*, *mean_func: ~typing.Callable = <class 'pygama.pargen.AoE_cal.pol1'>*, *sigma_func: ~typing.Callable = <class 'pygama.pargen.AoE_cal.sigma_fit'>*, *comptBands_width: int = 20*, *plot_options: dict = {}*)

Bases: object

AoEcorrection(*data: DataFrame*, *aoe_param: str*, *display: int = 0*)

Calculates the corrections needed for the energy dependence of the A/E. Does this by fitting the compton continuum in slices and then applies fits to the centroid and variance.

aoe_timecorr(*df*, *aoe_param*, *output_name='AoE_Timecorr'*, *display=0*)

calibrate(*df*, *initial_aoe_param*)

drift_time_correction(*data: DataFrame*, *aoe_param*, *display: int = 0*)

Calculates the correction needed to align the two drift time regions for ICPC detectors

fill_plot_dict(*data*, *plot_dict={}*)

get_aoe_cut_fit(*data: DataFrame*, *aoe_param: str*, *peak: float*, *ranges: tuple*, *dep_acc: float*, *display: int = 1*)

Determines A/E cut by sweeping through values and for each one fitting the DEP to determine how many events survive. Then interpolates to get cut value at desired DEP survival fraction (typically 90%)

get_results_dict()

update_cal_dicts(*update_dict*)

pygama.pargen.AoE_cal.**compton_sf**(*cut_param*, *low_cut_val*, *high_cut_val=None*, *mode='greater'*, *dt_mask=None*)

pygama.pargen.AoE_cal.**compton_sf_sweep**(*energy: np.array*, *cut_param: np.array*, *final_cut_value: float*, *peak: float*, *eres: list[float, float]*, *dt_mask: np.array = None*, *cut_range=(-5, 5)*, *n_samples=51*, *mode='greater'*)

Determines survival fraction for compton continuum by basic counting

Return type
tuple(float, np.array, list)

class pygama.pargen.AoE_cal.**drift_time_distribution**

Bases: *PDF*

bounds(***kwargs*)

extended_pdf(*n_sig1*, *mu1*, *sigma1*, *htail1*, *tau1*, *n_sig2*, *mu2*, *sigma2*, *htail2*, *tau2*, *components*)

fixed()

guess(*bins: array*, *var: array*, ***kwargs*) → list

Guess for fitting dt spectrum

Return type
list

pdf(*n_sig1*, *mu1*, *sigma1*, *htail1*, *tau1*, *n_sig2*, *mu2*, *sigma2*, *htail2*, *tau2*, *components*)

pygama.pargen.AoE_cal.**drifttime_corr_plot**(*aoe_class*, *data*, *aoe_param='AoE_Timecorr'*, *aoe_param_corr='AoE_DTcorr'*, *figsize=[12, 8]*, *fontsize=12*)

pygama.pargen.AoE_cal.**energy_guess**(*hist*, *bins*, *var*, *func_i*, *peak*, *eres*, *fit_range*)

Simple guess for peak fitting

pygama.pargen.AoE_cal.**fit_time_means**(*tstamps*, *means*, *reses*)

class pygama.pargen.AoE_cal.**gaussian**

Bases: *PDF*

bounds(***kwargs*)

extended_pdf(*n_events: float*, *mu: float*, *sigma: float*)

Extended PDF for A/E consists of a gaussian signal with gaussian tail background

Return type
tuple(float, np.array)

fixed()

guess(*bins*, *var*, ***kwargs*)

pdf(*n_events: float*, *mu: float*, *sigma: float*) → array

PDF for A/E consists of a gaussian signal with tail with gaussian tail background

Return type
array

pygama.pargen.AoE_cal.**get_peak_label**(*peak: float*) → str

Return type
str

pygama.pargen.AoE_cal.**get_sf_sweep**(*energy: np.array*, *cut_param: np.array*, *final_cut_value: float*, *peak: float*, *eres_pars: list*, *dt_mask=None*, *cut_range=(-5, 5)*, *n_samples=51*, *mode='greater'*)

Calculates survival fraction for gamma lines using fitting method as in cut determination

Return type
tuple(pd.DataFrame, float, float)

pygama.pargen.AoE_cal.**get_survival_fraction**(*energy*, *cut_param*, *cut_val*, *peak*, *eres_pars*, *high_cut=None*, *guess_pars_cut=None*, *guess_pars_surv=None*, *dt_mask=None*, *mode='greater'*, *display=0*)

pygama.pargen.AoE_cal.**plot_aoe_mean_time**(*aoe_class*, *data*, *time_param='AoE_Timecorr'*, *figsize=[12, 8]*, *fontsize=12*)

pygama.pargen.AoE_cal.**plot_aoe_res_time**(*aoe_class*, *data*, *time_param='AoE_Timecorr'*, *figsize=[12, 8]*, *fontsize=12*)

pygama.pargen.AoE_cal.**plot_classifier**(*aoe_class*, *data*, *aoe_param='AoE_Classifier'*, *xrange=(900, 3000)*, *yrange=(-50, 10)*, *xn_bins=700*, *yn_bins=500*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.AoE_cal.**plot_compt_bands_overlayed**(*aoe_class*, *data*, *eranges: list[tuple]*, *aoe_param='AoE_Timecorr'*, *aoe_range: list[float] | None = None*, *title='Compton Bands'*, *density=True*, *n_bins=50*, *figsize=[12, 8]*, *fontsize=12*) → None

Function to plot various compton bands to check energy dependence and corrections

pygama.pargen.AoE_cal.**plot_cut_fit**(*aoe_class*, *data*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.AoE_cal.**plot_dt_dep**(*aoe_class*, *data*, *eranges: list[tuple]*, *titles: list | None = None*, *aoe_param='AoE_Timecorr'*, *bins=[200, 100]*, *dt_max=2000*, *figsize=[12, 8]*, *fontsize=12*) → None

Function to produce 2d histograms of A/E against drift time to check dependencies

pygama.pargen.AoE_cal.**plot_mean_fit**(*aoe_class*, *data*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.AoE_cal.**plot_sf_vs_energy**(*aoe_class*, *data*, *xrange=(900, 3000)*, *n_bins=701*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.AoE_cal.**plot_sigma_fit**(*aoe_class*, *data*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.AoE_cal.**plot_spectra**(*aoe_class*, *data*, *xrange=(900, 3000)*, *n_bins=2101*, *xrange_inset=(1580, 1640)*, *n_bins_inset=200*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.AoE_cal.**plot_survival_fraction_curves**(*aoe_class*, *data*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

class pygama.pargen.AoE_cal.**pol1**

Bases: object

func(*a*, *b*)

guess(*means*, *mean_errs*)

string_func()

class pygama.pargen.AoE_cal.**sigma_fit**

Bases: object

func(*a*, *b*, *c*)

guess(*sigmas*, *sigma_errs*)

string_func()

class pygama.pargen.AoE_cal.**sigmoid_fit**

Bases: object

func(*a*, *b*, *c*, *d*)

guess(*ys*, *y_errs*)

class pygama.pargen.AoE_cal.**standard_aoe**

Bases: *PDF*

bounds(***kwargs*)

centroid(*errs*, *cov*)

extended_pdf(*n_sig: float*, *mu: float*, *sigma: float*, *n_bkg: float*, *tau_bkg: float*, *lower_range: float = inf*, *upper_range: float = inf*, *components: bool = False*)

Extended PDF for A/E consists of a gaussian signal with gaussian tail background

Return type
tuple(float, np.array)

fixed()

guess(*bins*, *var*, ***kwargs*)

pdf(*n_sig: float*, *mu: float*, *sigma: float*, *n_bkg: float*, *tau_bkg: float*, *lower_range: float = inf*, *upper_range: float = inf*, *components: bool = False*) → array

PDF for A/E consists of a gaussian signal with gaussian tail background

Return type
array

width(*errs*, *cov*)

class pygama.pargen.AoE_cal.**standard_aoe_bkg**

Bases: *PDF*

bounds(***kwargs*)

extended_pdf(*n_events: float*, *mu: float*, *sigma: float*, *tau_bkg: float*, *lower_range: float = inf*, *upper_range: float = inf*)

Extended PDF for A/E consists of a gaussian signal with gaussian tail background

Return type
tuple(float, np.array)

fixed()

guess(*bins*, *var*, ***kwargs*)

pdf(*n_events: float*, *mu: float*, *sigma: float*, *tau_bkg: float*, *lower_range: float = inf*, *upper_range: float = inf*) → array

PDF for A/E consists of a gaussian signal with tail with gaussian tail background

Return type
array

class pygama.pargen.AoE_cal.**standard_aoe_with_high_tail**

Bases: *PDF*

bounds(***kwargs*)

centroid(*errs*, *cov*)

extended_pdf(*n_sig: float*, *mu: float*, *sigma: float*, *htail: float*, *tau_sig: float*, *n_bkg: float*, *tau_bkg: float*, *lower_range: float = inf*, *upper_range: float = inf*, *components: bool = False*)

Extended PDF for A/E consists of a gaussian signal with gaussian tail background

Return type
tuple(float, np.array)

fixed()

guess(*bins*, *var*, ***kwargs*)

pdf(*n_sig: float*, *mu: float*, *sigma: float*, *htail: float*, *tau_sig: float*, *n_bkg: float*, *tau_bkg: float*, *lower_range: float = inf*, *upper_range: float = inf*, *components: bool = False*) → array

PDF for A/E consists of a gaussian signal with tail with gaussian tail background

Return type
array

width(*errs*, *cov*)

pygama.pargen.AoE_cal.**unbinned_aoe_fit**(*aoe: np.array*, *pdf=<class 'pygama.pargen.AoE_cal.standard_aoe'>*, *display: int = 0*, *verbose: bool = False) -> tuple(np.array*, *np.array*)

Fitting function for A/E, first fits just a gaussian before using the full pdf to fit if fails will return NaN values

Return type
tuple(np.array, np.array)

pygama.pargen.AoE_cal.**unbinned_energy_fit**(*energy: np.array*, *peak: float*, *eres: list*, *simplex=False*, *guess=None*, *display=0*, *verbose: bool = False*)

Fitting function for energy peaks used to calculate survival fractions

Return type
tuple(np.array, np.array)

pygama.pargen.cuts module

This module provides routines for calculating and applying quality cuts

pygama.pargen.cuts.**cut_dict_to_hit_dict**(*cut_dict*, *final_cut_field='is_valid_cal'*)

pygama.pargen.cuts.**find_pulser_properties**(*df*, *energy='daqenergy'*)

pygama.pargen.cuts.**generate_cuts**(*data: dict[str, ndarray]*, *parameters: dict[str, int]*, *rounding: int = 4*) → dict

Finds double sided cut boundaries for a file for the parameters specified

Parameters

- **data** (*lh5 table or dictionary of arrays*) – data to calculate cuts on
- **parameters** (*dict*) – dictionary with the parameter to be cut and the number of sigmas to cut at

Return type
dict

pygama.pargen.cuts.**get_cut_indexes**(*all_data: dict[str, ndarray]*, *cut_dict: dict*, *energy_param: str = 'trapTmax'*) → list[int]

Returns a mask of the data, for a single file, that passes cuts based on dictionary of cuts in form of cut boundaries above :param File: dictionary of parameters + array such as load_nda or lh5 table of params :type File: dict or lh5_table :param Cut_dict: Dictionary file with cuts :type Cut_dict: string

Return type
list[int]

pygama.pargen.cuts.**get_keys**(*in_data*, *parameters*)

pygama.pargen.cuts.**tag_pulsers**(*df*, *chan_info*, *window=0.01*)

pygama.pargen.data_cleaning module

mainly pulser tagging - gaussian_cut (fits data to a gaussian, returns mean +/- cut_sigma values) - xtalball_cut (fits data to a crystalball, returns mean +/- cut_sigma values) - find_pulser_properties (find pulser by looking for which peak has a constant time between events) - tag_pulsers

pygama.pargen.data_cleaning.**find_pulser_properties**(*df*, *energy='trap_max'*)

pygama.pargen.data_cleaning.**gaussian_cut**(*data*, *cut_sigma=3*, *plotAxis=None*)

fits data to a gaussian, returns mean +/- cut_sigma values for a cut

pygama.pargen.data_cleaning.**tag_pulsers**(*df*, *chan_info*, *window=250*)

pygama.pargen.data_cleaning.**xtalball_cut**(*data*, *cut_sigma=3*, *plotFigure=None*)

fits data to a crystalball, returns mean +/- cut_sigma values for a cut

pygama.pargen.dplms_ge_dict module

This module is for creating dplms dictionary for ge processing

pygama.pargen.dplms_ge_dict.**dplms_ge_dict**(*lh5_path: str*, *raw_fft: Table*, *raw_cal: Table*, *dsp_config: dict*, *par_dsp: dict*, *par_dsp_lh5: str*, *dplms_dict: dict*, *decay_const: float = 0*, *ene_par: str = 'dplmsEmax'*, *display: int = 0*) → dict

This function calculates the dplms dictionary for HPGe detectors.

Parameters

- **lh5_path** (*str*) – Name of channel to process, should be name of lh5 group in raw files
- **fft_files** – table with fft data
- **raw_cal** (*Table*) – table with cal data
- **dsp_config** (*dict*) – dsp config file
- **par_dsp** (*dict*) – Dictionary with db parameters for dsp processing
- **par_dsp_lh5** (*str*) – Path for saving dplms coefficients
- **dplms_dict** (*dict*) – Dictionary with various parameters

Returns
out_dict

Return type
dict

pygama.pargen.dplms_ge_dict.**filter_synthesis**(*ref: array*, *nmat: array*, *rmat: array*, *za: int*, *pmat: array*, *fmat: array*, *length: int*, *size: int*, *flip: bool = True*) → array

Return type
array

pygama.pargen.dplms_ge_dict.**is_not_pile_up**(*peak_pos: array*, *peak_pos_neg: array*, *thr: int*, *lim: int*, *size: int*) → list[bool]

Return type
list[bool]

pygama.pargen.dplms_ge_dict.**is_valid_centroid**(*centroid: array*, *lim: int*, *size: int*, *full_size: int*) → list[bool]

Return type
list[bool]

pygama.pargen.dplms_ge_dict.**is_valid_risetime**(*risetime: array*, *llim: int*, *perc: float*)

pygama.pargen.dplms_ge_dict.**noise_matrix**(*bls: array*, *length: int*) → array

Return type
array

pygama.pargen.dplms_ge_dict.**signal_matrices**(*wfs: array*, *length: int*, *decay_const: float*, *ff: int = 2*) → array

Return type
array

pygama.pargen.dplms_ge_dict.**signal_selection**(*dsp_cal*, *dplms_dict*, *coeff_values*)

pygama.pargen.dsp_optimize module

class pygama.pargen.dsp_optimize.**ParGrid**

Bases: object

Parameter Grid class Each ParGrid entry corresponds to a dsp parameter to be varied. The ntuples must follow the pattern: (name parameter value_strs) : (str, str, list of str) where name and parameter are the same as 'db.name.parameter' in the processing chain, value_strs is the array of strings to set the argument to.

add_dimension(*name*, *parameter*, *value_strs*)

get_data(*i_dim*, *i_par*)

get_n_dimensions()

get_n_grid_points()

get_n_points_of_dim(*i*)

get_par_meshgrid(*copy=False*, *sparse=False*)

return a meshgrid of parameter values Always uses Matrix indexing (natural for par grid) so that mg[i1][i2][...] corresponds to index order in self.dims Note copy is False by default as opposed to numpy default of True

get_shape()

get_zero_indices()

iterate_indices(*indices*)

iterate given indices [i1, i2, ...] by one. For easier iteration. The convention here is arbitrary, but its the order the arrays would be traversed in a series of nested for loops in the order appearin in dims (first dimension is first for loop, etc): Return False when the grid runs out of indices. Otherwise returns True.

print_data(*indices*)

set_dsp_pars(*db_dict*, *indices*)

class pygama.pargen.dsp_optimize.**ParGridDimension**(*name*, *parameter*, *value_strs*)

Bases: tuple

Create new instance of ParGridDimension(name, parameter, value_strs)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('name', 'parameter', 'value_strs')

classmethod _make(*iterable*)

Make a new ParGridDimension object from a sequence or iterable

_replace(***kwds*)

Return a new ParGridDimension object replacing specified fields with new values

name

Alias for field number 0

parameter

Alias for field number 1

value_strs

Alias for field number 2

pygama.pargen.dsp_optimize.**get_grid_points**(*grid*)

Generates a list of the indices of all possible grid points

pygama.pargen.dsp_optimize.**run_grid**(*tb_data*, *dsp_config*, *grid*, *fom_function*, *db_dict=None*, *verbosity=1*, ***fom_kwargs*)

Extract a table of optimization values for a grid of DSP parameters The grid argument defines a list of parameters and values over which to run the DSP defined in dsp_config on tb_data. At each point, a scalar figure-of-merit is extracted.

Returns a N-dimensional ndarray of figure-of-merit values, where the array axes are in the order they appear in grid.

Parameters

- **tb_data** (*lh5 Table*) – An input table of lh5 data. Typically a selection is made prior to sending tb_data to this function: optimization typically doesn't have to run over all data
- **dsp_config** (*dict*) – Specifies the DSP to be performed (see build_processing_chain()) and the list of output variables to appear in the output table for each grid point
- **grid** (ParGrid) – See ParGrid class for format
- **fom_function** (*function*) – When given the output lh5 table of this DSP iteration, the fom_function must return a scalar figure-of-merit. Should accept verbosity as a second keyword argument
- **db_dict** (*dict (optional)*) – DSP parameters database. See build_processing_chain for formatting info
- **verbosity** (*int (optional)*) – verbosity for the processing chain and fom_function calls
- ****fom_kwargs** – Any keyword arguments for fom_function

Returns

grid_values (*ndarray of floats*) – An N-dimensional numpy ndarray whose Mth axis corresponds to the Mth row of the grid argument

pygama.pargen.dsp_optimize.**run_grid_multiprocess_parallel**(*tb_data*, *dsp_config*, *grid*, *fom_function*, *db_dict=None*, *verbosity=1*, *processes=5*, *fom_kwargs=None*)

run one iteration of DSP on tb_data with multiprocessing, can handle multiple grids if they are the same dimensions

Optionally returns a value for optimization

Parameters

- **tb_data** (*lh5 Table*) – An input table of lh5 data. Typically a selection is made prior to sending tb_data to this function: optimization typically doesn't have to run over all data
- **dsp_config** (*dict*) – Specifies the DSP to be performed for this iteration (see build_processing_chain()) and the list of output variables to appear in the output table
- **grid** (*pargrid, list of pargrids*) – Grids to run optimization on
- **db_dict** (*dict (optional)*) – DSP parameters database. See build_processing_chain for formatting info
- **fom_function** (*function or None (optional)*) – When given the output lh5 table of this DSP iteration, the fom_function must return a scalar figure-of-merit value upon which the optimization will be based. Should accept verbosity as a second argument. If multiple grids provided can either pass one fom to have it run for each grid or a list of fom to run different fom on each grid.
- **verbosity** (*int (optional)*) – verbosity for the processing chain and fom_function calls
- **processes** (*int*) – DOCME
- **fom_kwargs** – any keyword arguments to pass to the fom, if multiple grids given will need to be a list of the fom_kwargs for each grid

Returns

- **figure_of_merit** (*float*) – If fom_function is not None, returns figure-of-merit value for the DSP iteration
- **tb_out** (*lh5 Table*) – If fom_function is None, returns the output lh5 table for the DSP iteration

pygama.pargen.dsp_optimize.**run_grid_point**(*tb_data*, *dsp_config*, *grids*, *fom_function*, *iii*, *db_dict=None*, *verbosity=1*, *fom_kwargs=None*)

Runs a single grid point for the index specified

pygama.pargen.dsp_optimize.**run_one_dsp**(*tb_data*, *dsp_config*, *db_dict=None*, *fom_function=None*, *verbosity=0*, *fom_kwargs=None*)

run one iteration of DSP on tb_data

Optionally returns a value for optimization

Parameters

- **tb_data** (*lh5 Table*) – An input table of lh5 data. Typically a selection is made prior to sending tb_data to this function: optimization typically doesn't have to run over all data
- **dsp_config** (*dict*) – Specifies the DSP to be performed for this iteration (see build_processing_chain()) and the list of output variables to appear in the output table
- **db_dict** (*dict (optional)*) – DSP parameters database. See build_processing_chain for formatting info
- **fom_function** (*function or None (optional)*) – When given the output lh5 table of this DSP iteration, the fom_function must return a scalar figure-of-merit value upon which the optimization will be based. Should accept verbosity as a second argument
- **verbosity** (*int (optional)*) – verbosity for the processing chain and fom_function calls
- **fom_kwargs** – any keyword arguments to pass to the fom

Returns

- **figure_of_merit** (*float*) – If fom_function is not None, returns figure-of-merit value for the DSP iteration
- **tb_out** (*lh5 Table*) – If fom_function is None, returns the output lh5 table for the DSP iteration

pygama.pargen.ecal_th module

This module provides a routine for running the energy calibration on Th data

pygama.pargen.ecal_th.**apply_cuts**(*data: DataFrame*, *hit_dict*, *cut_parameters=None*, *final_cut_field: str = 'is_valid_cal'*, *pulser_field='is_pulser'*)

pygama.pargen.ecal_th.**bin_pulser_stability**(*ecal_class*, *data*, *pulser_field='is_pulser'*, *time_slice=180*)

pygama.pargen.ecal_th.**bin_spectrum**(*ecal_class*, *data*, *cut_field='is_valid_cal'*, *pulser_field='is_pulser'*, *erange=[0, 3000]*, *dx=2*)

pygama.pargen.ecal_th.**bin_stability**(*ecal_class*, *data*, *time_slice=180*, *energy_range=[2585, 2660]*)

pygama.pargen.ecal_th.**bin_survival_fraction**(*ecal_class*, *data*, *cut_field='is_valid_cal'*, *pulser_field='is_pulser'*, *erange=[0, 3000]*, *dx=6*)

class pygama.pargen.ecal_th.**calibrate_parameter**(*energy_param*, *selection_string=''*, *plot_options: dict | None = None*, *guess_keV: float | None = None*, *threshold: int = 0*, *p_val: float = 0*, *n_events: int | None = None*, *simplex: bool = True*, *deg: int = 1*, *cal_energy_param: str | None = None*, *tail_weight=100*)

Bases: object

calibrate_parameter(*data*)

fill_plot_dict(*data*, *plot_dict={}*)

fit_energy_res()

funcs = [<function extended_radford_pdf>, <function extended_radford_pdf>, <function extended_radford_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_radford_pdf>]

gen_pars_dict()

get_results_dict(*data*)

glines = [583.191, 727.33, 860.564, 1592.53, 1620.5, 2103.53, 2614.5]

gof_funcs = [<function radford_pdf>, <function radford_pdf>, <function radford_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>, <function radford_pdf>]

range_keV = [(20, 20), (30, 30), (30, 30), (40, 20), (20, 40), (40, 40), (60, 60)]

class pygama.pargen.ecal_th.**fwhm_linear**

Bases: object

bounds()

func(*a*, *b*)

guess(*ys*, *y_errs*)

string_func()

class pygama.pargen.ecal_th.**fwhm_quadratic**

Bases: object

bounds()

func(*a*, *b*, *c*)

guess(*ys*, *y_errs*)

string_func()

pygama.pargen.ecal_th.**fwhm_slope**(*x: array*, *m0: float*, *m1: float*, *m2: float | None = None*) → array

Fit the energy resolution curve

Return type
array

pygama.pargen.ecal_th.**gen_pars_dict**(*pars*, *deg*, *energy_param*)

pygama.pargen.ecal_th.**get_peak_label**(*peak: float*) → str

Return type
str

pygama.pargen.ecal_th.**get_peak_labels**(*labels: list[str]*, *pars: list[float]*)

Return type
tuple(list[float], list[float])

class pygama.pargen.ecal_th.**high_stats_fitting**(*energy_param*, *selection_string*, *threshold*, *p_val*, *plot_options={}*, *simplex=False*, *tail_weight=20*, *cal_energy_param=None*, *deg=2*, *fixed=None*)

Bases: *calibrate_parameter*

binning = [0.02, 0.02, 0.02, 0.02, 0.2, 0.2, 0.02, 0.2, 0.2, 0.2, 0.1, 0.1, 0.1, 0.02, 0.2, 0.2, 0.2]

fit_peaks(*data*)

funcs = [<function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_radford_pdf>, <function extended_radford_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_radford_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_radford_pdf>, <function extended_radford_pdf>, <function extended_radford_pdf>, <function extended_radford_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>, <function extended_gauss_step_pdf>]

get_results_dict(*data*)

glines = [238.632, 511, 583.191, 727.33, 763, 785, 860.564, 893, 1079, 1513, 1592.53, 1620.5, 2103.53, 2614.5, 3125, 3198, 3474]

gof_funcs = [<function gauss_step_pdf>, <function gauss_step_pdf>, <function radford_pdf>, <function radford_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>, <function radford_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>, <function radford_pdf>, <function radford_pdf>, <function radford_pdf>, <function radford_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>, <function gauss_step_pdf>]

range_keV = [(10, 10), (30, 30), (30, 30), (30, 30), (30, 15), (15, 30), (30, 25), (25, 30), (30, 30), (30, 30), (30, 20), (20, 30), (30, 30), (30, 30), (30, 30), (30, 30), (30, 30)]

run_fit(*data*)

update_calibration(*data*)

pygama.pargen.ecal_th.**plot_2614_timemap**(*ecal_class*, *data*, *figsize=[12, 8]*, *fontsize=12*, *erange=[2580, 2630]*, *dx=1*, *time_dx=180*)

pygama.pargen.ecal_th.**plot_cal_fit**(*ecal_class*, *data*, *figsize=[12, 8]*, *fontsize=12*, *erange=[200, 2700]*)

pygama.pargen.ecal_th.**plot_eres_fit**(*ecal_class*, *data*, *erange=[200, 2700]*, *figsize=[12, 8]*, *fontsize=12*)

pygama.pargen.ecal_th.**plot_fits**(*ecal_class*, *data*, *figsize=[12, 8]*, *fontsize=12*, *ncols=3*, *nrows=3*, *binning_keV=5*)

pygama.pargen.ecal_th.**plot_pulser_timemap**(*ecal_class*, *data*, *pulser_field='is_pulser'*, *figsize=[12, 8]*, *fontsize=12*, *dx=0.2*, *time_dx=180*, *n_spread=3*)

pygama.pargen.energy_cal module

routines for automatic calibration.

- hpge_find_E_peaks (Find uncalibrated E peaks whose E spacing matches the pattern in peaks_keV)
- hpge_get_E_peaks (Get uncalibrated E peaks at the energies of peaks_keV)
- hpge_fit_E_peaks (fits the energy peals)
- hpge_E_calibration (main routine – finds and fits peaks specified)

pygama.pargen.energy_cal.**calibrate_tl208**(*energy_series*, *cal_peaks=None*, *plotFigure=None*)

energy_series: array of energies we want to calibrate cal_peaks: array of peaks to fit 1.) we find the 2614 peak by looking for the tallest peak at >0.1 the max adc value 2.) fit that peak to get a rough guess at a calibration to find other peaks with 3.) fit each peak in peak_energies 4.) do a linear fit to the peak centroids to find a calibration

pygama.pargen.energy_cal.**get_calibration_energies**(*cal_type*)

pygama.pargen.energy_cal.**get_hpge_E_bounds**(*func*, *parguess*)

pygama.pargen.energy_cal.**get_hpge_E_fixed**(*func*)

Returns: Sequence list of fixed indexes for fitting and mask for parameters

pygama.pargen.energy_cal.**get_hpge_E_peak_par_guess**(*hist*, *bins*, *var*, *func*, *mode_guess*)

Get parameter guesses for func fit to peak in hist

Parameters

- **hist** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist(). Should be windowed around the peak.
- **bins** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist(). Should be windowed around the peak.
- **var** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist(). Should be windowed around the peak.
- **func** (*function*) – The function to be fit to the peak in the (windowed) hist

pygama.pargen.energy_cal.**get_i_local_extrema**(*data*, *delta*)

Get lists of indices of the local maxima and minima of data

The "local" extrema are those maxima / minima that have heights / depths of at least delta. Converted from MATLAB script at: http://billauer.co.il/peakdet.html

Parameters

- **data** (*array-like*) – the array of data within which extrema will be found
- **delta** (*scalar*) – the absolute level by which data must vary (in one direction) about an extremum in order for it to be tagged

Returns

imaxes, imins (*2-tuple (array, array)*) – A 2-tuple containing arrays of variable length that hold the indices of the identified local maxima (first tuple element) and minima (second tuple element)

pygama.pargen.energy_cal.**get_i_local_maxima**(*data*, *delta*)

pygama.pargen.energy_cal.**get_i_local_minima**(*data*, *delta*)

pygama.pargen.energy_cal.**get_most_prominent_peaks**(*energySeries*, *xlo*, *xhi*, *xpb*, *max_num_peaks=inf*, *test=False*)

find the most prominent peaks in a spectrum by looking for spikes in derivative of spectrum energySeries: array of measured energies max_num_peaks = maximum number of most prominent peaks to find return a histogram around the most prominent peak in a spectrum of a given percentage of width

pygama.pargen.energy_cal.**hpge_E_calibration**(*E_uncal*, *peaks_keV*, *guess_keV*, *deg=0*, *uncal_is_int=False*, *range_keV=None*, *funcs=<function gauss_step_cdf>*, *gof_funcs=None*, *method='unbinned'*, *gof_func=None*, *n_events=None*, *simplex=False*, *allowed_p_val=0.05*, *tail_weight=100*, *verbose=True*)

Calibrate HPGe data to a set of known peaks

Parameters

- **E_uncal** (*array*) – unbinned energy data to be calibrated
- **peaks_keV** (*array*) – list of peak energies to be fit to. Each must be in the data
- **guess_keV** (*float*) – a rough initial guess at the conversion factor from E_uncal to keV. Must be positive
- **deg** (*non-negative int*) – degree of the polynomial for the E_cal function E_keV = poly(E_uncal). deg = 0 corresponds to a simple scaling E_keV = scale * E_uncal. Otherwise follows the convention in np.polyfit
- **uncal_is_int** (*bool*) – if True, attempts will be made to avoid picket-fencing when binning E_uncal
- **range_keV** (*float, tuple, array of floats, or array of tuples of floats*) – ranges around which the peak fitting is performed if tuple(s) are supplied, they provide the left and right ranges
- **funcs** – DOCME
- **gof_funcs** (*function or array of functions*) – functions to use for calculation goodness of fit if unspecified will use same func as fit
- **method** (*str*) – default is unbinned fit can specify to use binned fit method instead
- **gof_func** – DOCME
- **n_events** (*int*) – number of events to use for unbinned fit
- **simplex** (*bool*) – DOCME
- **allowed_p_val** – lower limit on p_val of fit
- **verbose** (*bool*) – print debug statements

Returns

- **pars, cov** (*array, 2D array*) – array of calibration function parameters and their covariances. The form of the function is E_keV = poly(E_uncal). Assumes poly() is overwhelmingly

dominated by the linear term. pars follows convention in np.polyfit unless deg=0, in which case it is the (lone) scale factor

- **results** (*dict with the following elements*) –

  'detected_peaks_locs', 'detected_peaks_keV'
  [array, array] array of rough uncalibrated/calibrated energies at which the fit peaks were found in the initial peak search

  'pt_pars', 'pt_cov'
  [list of (array), list of (2D array)] arrays of gaussian parameters / covariances fit to the peak tops in the first refinement

  'pt_cal_pars', 'pt_cal_cov'
  [array, 2D array] array of calibration parameters E_uncal = poly(E_keV) for fit to means of gausses fit to tops of each peak

  'pk_pars', 'pk_cov', 'pk_binws', 'pk_ranges'
  [list of (array), list of (2D array), list, list of (array)] the best fit parameters, covariances, bin width and energy range for the local fit to each peak

  'pk_cal_pars', 'pk_cal_cov'
  [array, 2D array] array of calibration parameters E_uncal = poly(E_keV) for fit to means from full peak fits

  'fwhms', 'dfwhms'
  [array, array] the numeric fwhms and their uncertainties for each peak.

pygama.pargen.energy_cal.**hpge_find_E_peaks**(*hist*, *bins*, *var*, *peaks_keV*, *n_sigma=5*, *deg=0*, *Etol_keV=None*, *var_zero=1*, *verbose=False*)

Find uncalibrated E peaks whose E spacing matches the pattern in peaks_keV Note: the specialization here to units "keV" in peaks and Etol is unnecessary. However it is kept so that the default value for Etol_keV has an unambiguous interpretation.

Parameters

- **hist** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist() var cannot contain any zero entries.
- **bins** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist() var cannot contain any zero entries.
- **var** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist() var cannot contain any zero entries.
- **peaks_keV** (*array*) – Energies of peaks to search for (in keV)
- **n_sigma** (*float*) – Threshold for detecting a peak in sigma (i.e. sqrt(var))
- **deg** (*int*) – deg arg to pass to poly_match
- **Etol_keV** (*float*) – absolute tolerance in energy for matching peaks
- **var_zero** (*float*) – number used to replace zeros of var to avoid divide-by-zero in hist/sqrt(var). Default value is 1. Usually when var = 0 its because hist = 0, and any value here is fine.
- **verbose** (*bool*) – print debug messages

Returns

- **detected_peak_locations** (*list*) – list of uncalibrated energies of detected peaks
- **detected_peak_energies** (*list*) – list of calibrated energies of detected peaks

- **pars** (*list of floats*) – the parameters for poly(peaks_uncal) = peaks_keV (polyfit convention)

pygama.pargen.energy_cal.**hpge_fit_E_cal_func**(*mus*, *mu_vars*, *Es_keV*, *E_scale_pars*, *deg=0*, *fixed=None*)

Find best fit of E = poly(mus +/- sqrt(mu_vars)) This is an inversion of hpge_fit_E_scale. E uncertainties are computed from mu_vars / dmu/dE where mu = poly(E) is the E_scale function

Parameters

- **mus** (*array*) – uncalibrated energies
- **mu_vars** (*array*) – variances in the mus
- **Es_keV** (*array*) – energies to fit to, in keV
- **E_scale_pars** (*array*) – Parameters from the escale fit (keV to ADC) used for calculating uncertainties
- **deg** (*int*) – degree for energy scale fit. deg=0 corresponds to a simple scaling mu = scale * E. Otherwise deg follows the definition in np.polyfit
- **fixed** (*dict*) – dict where keys are index of polyfit pars to fix and vals are the value to fix at, can be None to fix at guess value

Returns

- **pars** (*array*) – parameters of the best fit. Follows the convention in np.polyfit
- **cov** (*2D array*) – covariance matrix for the best fit parameters.

pygama.pargen.energy_cal.**hpge_fit_E_peak_tops**(*hist*, *bins*, *var*, *peak_locs*, *n_to_fit=7*, *cost_func='Least Squares'*, *inflate_errors=False*, *gof_method='var'*)

Fit gaussians to the tops of peaks

Parameters

- **hist** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist()
- **bins** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist()
- **var** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist()
- **peak_locs** (*array*) – locations of peaks in hist. Must be accurate two within +/- 2*n_to_fit
- **n_to_fit** (*int*) – number of hist bins near the peak top to include in the gaussian fit
- **cost_func** (*bool (optional)*) – Flag passed to gauss_mode_width_max()
- **inflate_errors** (*bool (optional)*) – Flag passed to gauss_mode_width_max()
- **gof_method** (*str (optional)*) – method flag passed to gauss_mode_width_max()

Returns

- **pars_list** (*list of array*) – a list of best-fit parameters (mode, sigma, max) for each peak-top fit
- **cov_list** (*list of 2D arrays*) – a list of covariance matrices for each pars

pygama.pargen.energy_cal.**hpge_fit_E_peaks**(*E_uncal*, *mode_guesses*, *wwidths*, *n_bins=50*, *funcs=<function gauss_step_cdf>*, *method='unbinned'*, *gof_funcs=None*, *n_events=None*, *allowed_p_val=0.05*, *uncal_is_int=False*, *simplex=False*, *tail_weight=100*)

Fit the Energy peaks specified using the function given

Parameters

- **E_uncal** (*array*) – unbinned energy data to be fit
- **mode_guesses** (*array*) – array of guesses for modes of each peak
- **wwidths** (*float or array of float*) – array of widths to use for the fit windows (in units of E_uncal), typically on the order of 10 sigma where sigma is the peak width
- **n_bins** (*int or array of ints*) – array of number of bins to use for the fit window histogramming
- **funcs** (*function or array of functions*) – funcs to be used to fit each region
- **method** (*str*) – default is unbinned fit can specify to use binned fit method instead
- **gof_funcs** (*function or array of functions*) – functions to use for calculation goodness of fit if unspecified will use same func as fit
- **uncal_is_int** (*bool*) – if True, attempts will be made to avoid picket-fencing when binning E_uncal
- **simplex** (*bool determining whether to do a round of simpson minimisation before gradient minimisation*) –
- **n_events** (*int number of events to use for unbinned fit*) –
- **allowed_p_val** (*lower limit on p_val of fit*) –

Returns

- **pars** (*list of array*) – a list of best-fit parameters for each peak fit
- **covs** (*list of 2D arrays*) – a list of covariance matrices for each pars
- **binwidths** (*list*) – a list of bin widths used for each peak fit
- **ranges** (*list of array*) – a list of [Euc_min, Euc_max] used for each peak fit

pygama.pargen.energy_cal.**hpge_fit_E_scale**(*mus*, *mu_vars*, *Es_keV*, *deg=0*, *fixed=None*)

Find best fit of poly(E) = mus +/- sqrt(mu_vars) Compare to hpge_fit_E_cal_func which fits for E = poly(mu)

Parameters

- **mus** (*array*) – uncalibrated energies
- **mu_vars** (*array*) – variances in the mus
- **Es_keV** (*array*) – energies to fit to, in keV
- **deg** (*int*) – degree for energy scale fit. deg=0 corresponds to a simple scaling mu = scale * E. Otherwise deg follows the definition in np.polyfit
- **fixed** (*dict*) – dict where keys are index of polyfit pars to fix and vals are the value to fix at, can be None to fix at guess value

Returns

- **pars** (*array*) – parameters of the best fit. Follows the convention in np.polyfit
- **cov** (*2D array*) – covariance matrix for the best fit parameters.

pygama.pargen.energy_cal.**hpge_get_E_peaks**(*hist*, *bins*, *var*, *cal_pars*, *peaks_keV*, *n_sigma=3*, *Etol_keV=5*, *var_zero=1*, *verbose=False*)

Get uncalibrated E peaks at the energies of peaks_keV

Parameters

- **hist** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist() var cannot contain any zero entries.
- **bins** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist() var cannot contain any zero entries.
- **var** (*array, array, array*) – Histogram of uncalibrated energies, see pgh.get_hist() var cannot contain any zero entries.
- **cal_pars** (*array*) – Estimated energy calibration parameters used to search for peaks
- **peaks_keV** (*array*) – Energies of peaks to search for (in keV)
- **n_sigma** (*float*) – Threshold for detecting a peak in sigma (i.e. sqrt(var))
- **Etol_keV** (*float*) – absolute tolerance in energy for matching peaks
- **var_zero** (*float*) – number used to replace zeros of var to avoid divide-by-zero in hist/sqrt(var). Default value is 1. Usually when var = 0 its because hist = 0, and any value here is fine.
- **verbose** (*bool*) – print debug messages

Returns

- **got_peak_locations** (*list*) – list of uncalibrated energies of found peaks
- **got_peak_energies** (*list*) – list of calibrated energies of found peaks
- **pars** (*list of floats*) – the parameters for poly(peaks_uncal) = peaks_keV (polyfit convention)

pygama.pargen.energy_cal.**match_peaks**(*data_pks*, *cal_pks*)

Match uncalibrated peaks with literature energy values.

pygama.pargen.energy_cal.**poly_match**(*xx*, *yy*, *deg=-1*, *rtol=1e-05*, *atol=1e-08*)

Find the polynomial function best matching pol(xx) = yy

Finds the poly fit of xx to yy that obtains the most matches between pol(xx) and yy in the np.isclose() sense. If multiple fits give the same number of matches, the fit with the best gof is used, where gof is computed only among the matches. Assumes that the relationship between xx and yy is monotonic

Parameters

- **xx** (*array-like*) – domain data array. Must be sorted from least to largest. Must satisfy len(xx) >= len(yy)
- **yy** (*array-like*) – range data array: the values to which pol(xx) will be compared. Must be sorted from least to largest. Must satisfy len(yy) > max(2, deg+2)
- **deg** (*int*) – degree of the polynomial to be used. If deg = 0, will fit for a simple scaling: scale * xx = yy. If deg = -1, fits to a simple shift in the data: xx + shift = yy. Otherwise, deg is equivalent to the deg argument of np.polyfit()
- **rtol** (*float*) – the relative tolerance to be sent to np.isclose()
- **atol** (*float*) – the absolute tolerance to be sent to np.isclose(). Has the same units as yy.

Returns

- **pars** (*None or array of floats*) – The parameters of the best fit of poly(xx) = yy. Follows the convention used for the return value "p" of polyfit. Returns None when the inputs are bad.
- **i_matches** (*list of int*) – list of indices in xx for the matched values in the best match

pygama.pargen.energy_cal.**poly_wrapper**(*x*, **pars*)

pygama.pargen.energy_cal.**staged_fit**(*energies*, *hist*, *bins*, *var*, *func_i*, *gof_func_i*, *simplex*, *mode_guess*, *tail_weight=100*)

class pygama.pargen.energy_cal.**tail_prior**(*data*, *model*, *tail_weight=100*)

Bases: object

Generic least-squares cost function with error.

_call(**pars*)

errordef = 0.5

verbose = 0

pygama.pargen.energy_optimisation module

This module contains the functions for performing the energy optimisation. This happens in 2 steps, firstly a grid search is performed on each peak separately using the optimiser, then the resulting grids are interpolated to provide the best energy resolution at Qbb

class pygama.pargen.energy_optimisation.**BayesianOptimizer**(*acq_func*, *batch_size*, *kernel=None*)

Bases: object

_extend_prior_with_posterior_data(*x*, *y*, *yerr*)

_get_expected_improvement(*x_new*)

_get_lcb(*x_new*)

_get_next_probable_point()

_get_ucb(*x_new*)

add_dimension(*name*, *parameter*, *min_val*, *max_val*, *rounding=2*, *unit=None*)

add_initial_values(*x_init*, *y_init*, *yerr_init*)

eta_param = 0

get_best_vals()

get_first_point()

get_n_dimensions()

iterate_values()

lambda_param = 0.01

plot(*init_samples=None*)

plot_acq(*init_samples=None*)

update(*results*)

update_db_dict(*db_dict*)

class pygama.pargen.energy_optimisation.**OptimiserDimension**(*name*, *parameter*, *min_val*, *max_val*, *rounding*, *unit*)

Bases: tuple

Create new instance of OptimiserDimension(name, parameter, min_val, max_val, rounding, unit)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('name', 'parameter', 'min_val', 'max_val', 'rounding', 'unit')

classmethod _make(*iterable*)

Make a new OptimiserDimension object from a sequence or iterable

_replace(***kwds*)

Return a new OptimiserDimension object replacing specified fields with new values

max_val

Alias for field number 3

min_val

Alias for field number 2

name

Alias for field number 0

parameter

Alias for field number 1

rounding

Alias for field number 4

unit

Alias for field number 5

pygama.pargen.energy_optimisation.**event_selection**(*raw_files*, *lh5_path*, *dsp_config*, *db_dict*, *peaks_keV*, *peak_idxs*, *kev_widths*, *cut_parameters={'bl_mean': 4, 'bl_std': 4, 'pz_std': 4}*, *pulser_mask=None*, *energy_parameter='trapTmax'*, *wf_field: str = 'waveform'*, *n_events=10000*, *threshold=1000*)

pygama.pargen.energy_optimisation.**find_lowest_grid_point_save**(*grid*, *err_grid*, *opt_dict*)

Finds the lowest grid point, if more than one with same value returns shortest filter.

pygama.pargen.energy_optimisation.**fom_FWHM**(*tb_in*, *kwarg_dict*, *ctc_parameter*, *alpha*, *idxs=None*, *display=0*)

FOM for sweeping over ctc values to find the best value, returns the best found fwhm

pygama.pargen.energy_optimisation.**fom_FWHM_fit**(*tb_in*, *kwarg_dict*)

FOM with no ctc sweep, used for optimising ftp.

pygama.pargen.energy_optimisation.**fom_FWHM_with_dt_corr_fit**(*tb_in*, *kwarg_dict*, *ctc_parameter*, *idxs=None*, *display=0*)

FOM for sweeping over ctc values to find the best value, returns the best found fwhm with its error, the corresponding alpha value and the number of events in the fitted peak, also the reduced chisquare of the

pygama.pargen.energy_optimisation.**fom_all_fit**(*tb_in*, *kwarg_dict*)

FOM to run over different ctc parameters

pygama.pargen.energy_optimisation.**fwhm_slope**(*x*, *m0*, *m1*, *m2*)

Fit the energy resolution curve

pygama.pargen.energy_optimisation.**get_best_vals**(*peak_grids*, *peak_energies*, *param*, *opt_dict*, *save_path=None*)

Finds best filter parameters

pygama.pargen.energy_optimisation.**get_ctc_grid**(*grids*, *ctc_param*)

Reshapes optimizer grids to be in easier form

pygama.pargen.energy_optimisation.**get_filter_params**(*grids*, *matched_configs*, *peak_energies*, *parameters*, *save_path=None*)

Finds best parameters for filter

pygama.pargen.energy_optimisation.**get_peak_fwhm_with_dt_corr**(*Energies*, *alpha*, *dt*, *func*, *gof_func*, *peak*, *kev_width*, *guess=None*, *kev=False*, *display=0*)

Applies the drift time correction and fits the peak returns the fwhm, fwhm/max and associated errors, along with the number of signal events and the reduced chi square of the fit. Can return result in ADC or keV.

pygama.pargen.energy_optimisation.**get_wf_indexes**(*sorted_indexs*, *n_events*)

pygama.pargen.energy_optimisation.**index_data**(*data*, *indexes*, *wf_field='waveform'*)

pygama.pargen.energy_optimisation.**interpolate_energy**(*peak_energies*, *points*, *err_points*, *energy*)

pygama.pargen.energy_optimisation.**interpolate_energy_old**(*peak_energies*, *grids*, *error_grids*, *energy*, *nevents_grids*)

Interpolates fwhm vs energy for every grid point

pygama.pargen.energy_optimisation.**interpolate_grid**(*energies*, *grids*, *int_energy*, *deg*, *nevents_grids*)

Interpolates energy vs parameter for every grid point using polynomial.

pygama.pargen.energy_optimisation.**new_fom**(*data*, *kwarg_dict*)

pygama.pargen.energy_optimisation.**run_optimisation**(*tb_data*, *dsp_config*, *fom_function*, *optimisers*, *fom_kwargs=None*, *db_dict=None*, *nan_val=10*, *n_iter=10*)

pygama.pargen.energy_optimisation.**run_optimisation_multiprocessed**(*file*, *opt_config*, *dsp_config*, *cuts*, *lh5_path*, *fom=None*, *db_dict=None*, *processes=5*, *n_events=8000*, ***fom_kwargs*)

Runs optimisation on .lh5 file, this version multiprocesses the grid points, it also can handle multiple grids being passed as long as they are the same dimensions.

Parameters

- **file** (*string*) – path to raw .lh5 file

- **opt_config** (*str*) – path to JSON dictionary to configure optimisation
- **dsp_config** (*str*) – path to JSON dictionary specifying dsp configuration
- **fom** (*function*) – When given the output lh5 table of a DSP iteration, the fom_function must return a scalar figure-of-merit value upon which the optimization will be based. Should accept verbosity as a second argument
- **n_events** (*int*) – Number of events to run over
- **db_dict** (*dict*) – Dictionary specifying any values to put in processing chain e.g. pz constant
- **processes** (*int*) – Number of separate processes to run for the multiprocessing

pygama.pargen.energy_optimisation.**set_par_space**(*opt_config*)

Generates grid for optimizer from dictionary of form {param : {start: , end: , spacing: }}

pygama.pargen.energy_optimisation.**set_values**(*par_values*)

Finds values for grid

pygama.pargen.energy_optimisation.**simple_guess**(*hist*, *bins*, *var*, *func_i*, *fit_range*)

Simple guess for peak fitting

pygama.pargen.energy_optimisation.**single_peak_fom**(*data*, *kwarg_dict*)

pygama.pargen.energy_optimisation.**unbinned_energy_fit**(*energy*, *func*, *gof_func*, *gof_range*, *fit_range=(inf, inf)*, *guess=None*, *tol=None*, *verbose=False*, *display=0*)

Unbinned fit to energy. This is different to the default fitting as it will try different fitting methods and choose the best. This is necessary for the lower statistics.

pygama.pargen.extract_tau module

This module is for extracting a single pole zero constant from the decay tail

pygama.pargen.extract_tau.**dsp_preprocess_decay_const**(*tb_data*, *dsp_config: dict*, *double_pz: bool = False*, *display: int = 0*, *opt_dict: dict | None = None*, *wf_field: str = 'waveform'*, *wf_plot: str = 'wf_pz'*, *norm_param: str = 'pz_mean'*, *cut_parameters: dict = {'bl_mean': 4, 'bl_slope': 4, 'bl_std': 4}*) → dict

This function calculates the pole zero constant for the input data

Parameters

- **f_raw** (*str*) – The raw file to run the macro on
- **dsp_config** (*str*) – Path to the dsp config file, this is a stripped down version which just includes cuts and slope of decay tail
- **channel** (*str*) – Name of channel to process, should be name of lh5 group in raw files

Returns

tau_dict (*dict*)

Return type

dict

pygama.pargen.extract_tau.**fom_dpz**(*tb_data*, *verbosity=0*, *rand_arg=None*)

pygama.pargen.extract_tau.**get_decay_constant**(*slopes: array*, *wfs: WaveformTable*, *display: int = 0*) → dict

Finds the decay constant from the modal value of the tail slope after cuts and saves it to the specified json.

Parameters

- **slopes** (*array*) – tail slope array
- **dict_file** (*str*) – path to json file to save decay constant value to. It will be saved as a dictionary of form {'pz': {'tau': decay_constant}}

Returns
tau_dict (*dict*)

Return type
dict

pygama.pargen.extract_tau.**get_dpz_consts**(*grid_out*, *opt_dict*)

pygama.pargen.extract_tau.**load_data**(*raw_file: list[str]*, *lh5_path: str*, *pulser_mask=None*, *n_events: int = 10000*, *threshold: int = 5000*, *wf_field: str = 'waveform'*) → Table

Return type
Table

pygama.pargen.lq_cal module

pygama.pargen.lq_cal.**binned_lq_fit**(*df: ~pandas.core.frame.DataFrame*, *lq_param: str*, *cal_energy_param: str*, *peak: float*, *cdf=CPUDispatcher(<function gauss_cdf>)*, *sidebands: bool = True*)

Function for fitting a distribution of LQ values within a specified
energy peak. Fits a gaussian to the distribution

Parameters

- **df** (*pd.DataFrame()*) – Dataframe containing the data for fitting. Data must contain the desired lq parameter and the calibrated energy
- **lq_param** (*string*) – Name of the LQ parameter to fit
- **cal_energy_param** (*string*) – Name of the calibrated energy parameter of choice
- **peak** (*float*) – Energy value, in keV, of the peak who's LQ distribution will be fit
- **cdf** (*callable*) – Function to be used for the binned fit
- **sidebands** (*bool*) – Whether or not to perform a sideband subtraction when fitting the LQ distribution

Returns

- **m1.values** (*array-like object*) – Resulting parameter values from the peak fit
- **m1.errors** (*array-like object*) – Resulting parameter errors from the peak fit
- **hist** (*array*) – Histogram that was used for the binned fit
- **bins** (*array*) – Array of bin edges used for the binned fit

class pygama.pargen.lq_cal.**cal_lq**(*cal_dicts: dict*, *cal_energy_param: str*, *eres_func: callable*, *cdf: callable = CPUDispatcher(<function gauss_cdf>)*, *selection_string: str = 'is_valid_cal&is_not_pulser'*, *plot_options: dict = {}*)

Bases: object

A class for calibrating the LQ parameter and determining the LQ cut value

Parameters

- **cal_dicts** (*dict*) – A dictionary containing the hit-level operations to apply to the data.
- **cal_energy_param** (*string*) – The calibrated energy parameter of choice
- **eres_function** (*callable*) – The energy resolutions function
- **cdf** (*callable*) – The CDF used for the binned fits
- **selection_string** (*string*) – A string of flags to apply the data when running the calibration
- **plot_options** (*dict*) – A dict containing the plot functions the user wants to run, and any user options to provide those plot functions

calibrate(*df*, *initial_lq_param*)

Run the LQ calibration and calculate the cut value

drift_time_correction(*df: pd.DataFrame()*, *lq_param*, *cal_energy_param: str*, *display: int = 0*)

Deterimines the drift time correction parameters for LQ by fitting a degree 1 polynomial to the LQ vs drift time distribution for DEP events. Corrects for any linear dependence and centers the final LQ distribution to a mean of 0.

fill_plot_dict(*data*, *plot_dict={}*)

get_cut_lq_dep(*df: pd.DataFrame()*, *lq_param: str*, *cal_energy_param: str*)

Determines the cut value for LQ. Value is calculated by fitting the LQ distribution for events in the DEP to a gaussian. The cut value is set at 3*sigma of the fit. Sideband subtraction is used to determine the LQ distribution for DEP events. Events greater than the cut value fail the cut.

get_results_dict()

lq_timecorr(*df*, *lq_param*, *output_name='LQ_Timecorr'*, *display=0*)

Calculates the average LQ value for DEP events for each specified run run_timestamp. Applies a time normalization based on the average LQ value in the DEP across all run_timestamps.

update_cal_dicts(*update_dict*)

pygama.pargen.lq_cal.**fit_time_means**(*tstamps*, *means*, *reses*)

pygama.pargen.lq_cal.**get_fit_range**(*lq: np.array*)

Function for determining the fit range for a given distribution of lq values

Return type
tuple(float, float)

pygama.pargen.lq_cal.**get_lq_hist**(*df: pd.DataFrame()*, *lq_param: str*, *cal_energy_param: str*, *peak: float*, *sidebands: bool = True*)

Function for getting a distribution of LQ values for a given peak. Returns a histogram of the LQ distribution as well as an array of bin edges

pygama.pargen.lq_cal.**plot_classifier**(*lq_class*, *data*, *lq_param='LQ_Classifier'*, *xrange=(800, 3000)*, *yrange=(-2, 8)*, *xn_bins=700*, *yn_bins=500*, *figsize=[12, 8]*, *fontsize=12*) → figure

Return type
figure

pygama.pargen.lq_cal.**plot_drift_time_correction**(*lq_class*, *data*, *lq_param='LQ_Timecorr'*, *figsize=[12, 8]*, *fontsize=12*) → figure

Plots a 2D histogram of LQ versus effective drift time in a 6 keV window around the DEP. Additionally plots the fit results for the drift time correction.

Return type
figure

pygama.pargen.lq_cal.**plot_lq_cut_fit**(*lq_class*, *data*, *figsize=[12, 8]*, *fontsize=12*) → figure

Plots the final histogram of LQ values for events in the DEP, and the fit results used for determining the cut value

Return type
figure

pygama.pargen.lq_cal.**plot_lq_mean_time**(*lq_class*, *data*, *lq_param='LQ_Timecorr'*, *figsize=[12, 8]*, *fontsize=12*) → figure

Plots the mean LQ value calculated for each given timestamp

Return type
figure

pygama.pargen.lq_cal.**plot_sf_vs_energy**(*lq_class*, *data*, *xrange=(900, 3000)*, *n_bins=701*, *figsize=[12, 8]*, *fontsize=12*) → figure

Plots the survival fraction as a function of energy

Return type
figure

pygama.pargen.lq_cal.**plot_spectra**(*lq_class*, *data*, *xrange=(900, 3000)*, *n_bins=2101*, *xrange_inset=(1580, 1640)*, *n_bins_inset=200*, *figsize=[12, 8]*, *fontsize=12*) → figure

Plots a 2D histogram of the LQ classifier vs calibrated energy

Return type
figure

pygama.pargen.lq_cal.**plot_survival_fraction_curves**(*lq_class*, *data*, *figsize=[12, 8]*, *fontsize=12*) → figure

Plots the survival fraction curves as a function of LQ cut values for every peak of interest

Return type
figure

pygama.pargen.mse_psd module

- get_avse_cut (does AvsE)
- get_ae_cut (does A/E)

pygama.pargen.mse_psd.**get_ae_cut**(*e_cal*, *current*, *plotFigure=None*)

pygama.pargen.mse_psd.**get_avse_cut**(*e_cal*, *current*, *plotFigure=None*)

pygama.pargen.noise_optimization module

This module contains the functions for performing the filter optimisation. This happens with a grid search performed on ENC peak.

pygama.pargen.noise_optimization.**calculate_spread**(*energies*, *percentile_low*, *percentile_high*, *n_samples*)

pygama.pargen.noise_optimization.**noise_optimization**(*tb_data: Table*, *dsp_proc_chain: dict*, *par_dsp: dict*, *opt_dict: dict*, *lh5_path: str*, *verbose: bool = False*, *display: int = 0*) → dict

This function calculates the optimal filter par. :param tb_data: raw table to run the macro on :type tb_data: str :param dsp_proc_chain: Path to minimal dsp config file :type dsp_proc_chain: str :param par_dsp: Dictionary with default dsp parameters :type par_dsp: str :param opt_dict: Dictionary with parameters for optimization :type opt_dict: str :param lh5_path: Name of channel to process, should be name of lh5 group in raw files :type lh5_path: str

Returns
res_dict (*dict*)

Return type
dict

pygama.pargen.noise_optimization.**simple_gaussian_fit**(*energies*, *dx=1*, *sigma_thr=4*, *allowed_p_val=1e-20*)

pygama.pargen.noise_optimization.**simple_gaussian_guess**(*hist*, *bins*, *func*, *toll=0.2*)

pygama.pargen.utils module

pygama.pargen.utils.**get_params**(*file_params*, *param_list*)

pygama.pargen.utils.**get_tcm_pulser_ids**(*tcm_file*, *channel*, *multiplicity_threshold*)

pygama.pargen.utils.**load_data**(*files: list*, *lh5_path: str*, *cal_dict: dict*, *params=['cuspEmax']*, *cal_energy_param: str = 'cuspEmax_ctc_cal'*, *threshold=None*, *return_selection_mask=False*)

Loads in the A/E parameters needed and applies calibration constants to energy

Return type
tuple(np.array, np.array, np.array, np.array)

pygama.pargen.utils.**return_nans**(*input*)

pygama.raw package

pygama.skm package

Utilities for grouping hit data into events.

Submodules

pygama.skm.build_skm module

This module implements routines to build the *skm* tier, consisting of skimmed data from lower tiers.

pygama.skm.build_skm.**build_skm**(*f_evt: str*, *f_hit: str*, *f_dsp: str*, *f_tcm: str*, *skm_conf: dict | str*, *f_skm: str | None = None*, *wo_mode: str = 'w'*, *skm_group: str = 'skm'*, *evt_group: str = 'evt'*, *tcm_group: str = 'hardware_tcm_1'*, *dsp_group: str = 'dsp'*, *hit_group: str = 'hit'*, *tcm_id_table_pattern: str = 'ch{}'*) → None | Table

Builds a skimmed file from a (set) of *evt/hit/dsp* tier file(s).

Parameters

- **f_evt** (*str*) – path of *evt* file.
- **f_hit** (*str*) – path of *hit* file.
- **f_dsp** (*str*) – path of *dsp* file.
- **f_tcm** (*str*) – path of *tcm* file.
- **skm_conf** (*dict | str*) – name of configuration file or dictionary defining *skm* fields.
  - `multiplicity` defines up to which row length `VectorOfVector` fields should be kept.
  - `postfixes` list of postfixes must be list of `len(multiplicity)`. If not given, numbers from 0 to `multiplicity -1` are used
  - `operations` are forwarded from lower tiers and clipped/padded according to `missing_value` if needed. If the forwarded field is not an evt tier, `tcm_idx` must be passed that specifies the value to pick across channels.

  For example:

```
{
  "multiplicity": 2,
  "postfixes":["", "aux"],
  "operations": {
    "timestamp":{
      "forward_field": "evt.timestamp"
    },
    "multiplicity":{
      "forward_field": "evt.multiplicity"
    },
    "energy":{
      "forward_field": "hit.cuspEmax_ctc_cal",
      "missing_value": "np.nan",
      "tcm_idx": "evt.energy_idx"
    },
```

(continues on next page)

(continued from previous page)

```
    "energy_id":{
      "forward_field": "tcm.array_id",
      "missing_value": 0,
      "tcm_idx": "evt.energy_idx"
    }
  }
}
```

- **f_skm** (*str | None*) – name of the *skm* output file. If None, return the output class:.*Table* instead of writing to disk.
- **wo_mode** (*str*) – writing mode.
  - write_safe or w: only proceed with writing if the file does not already exists.
  - append or a: append to file.
  - overwrite or o: replaces existing file.
- **skm_group** (*str*) – *skm* LH5 root group name.
- **evt_group** (*str*) – *evt* LH5 root group name.
- **hit_group** (*str*) – *hit* LH5 root group name.
- **dsp_group** (*str*) – *dsp* LH5 root group name.
- **tcm_group** (*str*) – *tcm* LH5 root group name.
- **tcm_id_table_pattern** (*str*) – pattern to format *tcm* id values to table name in higher tiers. Must have one placeholder which is the *tcm* id.

Return type
None | *Table*

pygama.vis package

This subpackage implements utilities to visualize data.

Submodules

pygama.cli module

pygama's command line interface utilities.

pygama.cli.**add_build_dsp_parser**(*subparsers*)
Configure dsp.build_dsp.build_dsp() command line interface

pygama.cli.**add_build_hit_parser**(*subparsers*)
Configure *hit.build_hit.build_hit()* command line interface

pygama.cli.**add_build_raw_parser**(*subparsers*)
Configure raw.build_raw.build_raw() command line interface

pygama.cli.**add_lh5ls_parser**(*subparsers*)
Configure lgdo.lh5.show() command line interface.

pygama.cli.**build_dsp_cli**(*args*)

Passes command line arguments to `dsp.build_dsp.build_dsp()`.

pygama.cli.**build_hit_cli**(*args*)

Passes command line arguments to *hit.build_hit.build_hit()*.

pygama.cli.**build_raw_cli**(*args*)

Passes command line arguments to `raw.build_raw.build_raw()`.

pygama.cli.**lh5_show_cli**(*args*)

Passes command line arguments to `lgdo.lh5.show()`.

pygama.cli.**pygama_cli**()

pygama's command line interface.

Defines the command line interface (CLI) of the package, which exposes some of the most used functions to the console. This function is added to the `entry_points.console_scripts` list and defines the `pygama` executable (see `setuptools`' documentation). To learn more about the CLI, have a look at the help section:

```
$ pygama --help
$ pygama build-raw --help  # help section for a specific sub-command
```

pygama.logging module

This module implements some helpers for setting up logging.

pygama.logging.**setup**(*level: int = 20*, *logger: Logger | None = None*) → None

Setup a colorful logging output.

If *logger* is None, sets up only the `pygama` logger.

Parameters

- **level** (*int*) – logging level (see `logging` module).
- **logger** (*Logger | None*) – if not *None*, setup this logger.

Examples

```
>>> from pygama import logging
>>> logging.setup(level=logging.DEBUG)
```

2.2 Developer's guide

Note: The https://learn.scientific-python.org webpages are an extremely valuable learning resource for Python software developer. The reader is referred to that for any detail not covered in the following guide.

The following rules and conventions have been established for the package development and are enforced throughout the entire code base. Merge requests that do not comply to the following directives will be rejected.

To start developing *pygama*, fork the remote repository to your personal GitHub account (see About Forks). If you have not set up your ssh keys on the computer you will be working on, please follow GitHub's instructions. Once you have your own fork, you can clone it via (replace "yourusername" with your GitHub username):

```
$ git clone git@github.com:yourusername/pygama.git
```

All extra tools needed to develop *pygama* are listed as optional dependencies and can be installed via pip by running:

```
$ cd pygama
$ pip install -e '.[all]'  # single quotes are not needed on bash
```

Important: Pip's `--editable | -e` flag let's you install the package in "developer mode", meaning that any change to the source code will be directly propagated to the installed package and importable in scripts.

Tip: It is strongly recommended to work inside a virtual environment, which guarantees reproductibility and isolation. For more details, see learn.scientific-python.org.

2.2.1 Code style

- All functions and methods (arguments and return types) must be type-annotated. Type annotations for variables like class attributes are also highly appreciated.
- Messaging to the user is managed through the `logging` module. Do not add `print()` statements. To make a logging object available in a module, add this:

  ```
  import logging
  log = logging.getLogger(__name__)
  ```

  at the top. In general, try to keep the number of `logging.debug()` calls low and use informative messages. `logging.info()` calls should be reserved for messages from high-level routines (like `pygama.dsp.build_dsp()`) and very sporadic. Good code is never too verbose.
- If an error condition leading to undefined behavior occurs, raise an exception. try to find the most suitable between the built-in exceptions, otherwise `raise RuntimeError("message")`. Do not raise `Warnings`, use `logging.warning()` for that and don't abort the execution.
- Warning messages (emitted when a problem is encountered that does not lead to undefined behavior) must be emitted through `logging.warning()` calls.

A set of pre-commit hooks is configured to make sure that *pygama* coherently follows standard coding style conventions. The pre-commit tool is able to identify common style problems and automatically fix them, wherever possible. Configured hooks are listed in the `.pre-commit-config.yaml` file at the project root folder. They are run remotely on the GitHub repository through the pre-commit bot, but should also be run locally before submitting a pull request:

```
$ cd pygama
$ pip install '.[test]'
$ pre-commit run --all-files  # analyse the source code and fix it wherever possible
$ pre-commit install          # install a Git pre-commit hook (strongly recommended)
```

For a more comprehensive guide, check out the learn.scientific-python.org documentation about code style.

2.2.2 Testing

- The *pygama* test suite is available below `tests/`. We use pytest to run tests and analyze their output. As a starting point to learn how to write good tests, reading of the relevant learn.scientific-python.org webpage is recommended.
- Unit tests are automatically run for every push event and pull request to the remote Git repository on a remote server (currently handled by GitHub actions). Every pull request must pass all tests before being approved for merging. Running the test suite is simple:

```
$ cd pygama
$ pip install '.[test]'
$ pytest
```

- Additionally, pull request authors are required to provide tests with sufficient code coverage for every proposed change or addition. If necessary, high-level functional tests should be updated. We currently rely on codecov.io to keep track of test coverage. A local report, which must be inspected before submitting pull requests, can be generated by running:

```
$ pytest --cov=pygama
```

2.2.3 Documentation

We adopt best practices in writing and maintaining *pygama*'s documentation. When contributing to the project, make sure to implement the following:

- Documentation should be exclusively available on the Project website pygama.readthedocs.io. No READMEs, GitHub/LEGEND wiki pages should be written.
- Pull request authors are required to provide sufficient documentation for every proposed change or addition.
- Documentation for functions, classes, modules and packages should be provided as Docstrings along with the respective source code. Docstrings are automatically converted to HTML as part of the *pygama* package API documentation.
- General guides, comprehensive tutorials or other high-level documentation (e.g. referring to how separate parts of the code interact between each other) must be provided as separate pages in `docs/source/` and linked in the table of contents.
- Jupyter notebooks should be added to the main Git repository below `docs/source/notebooks`.
- Before submitting a pull request, contributors are required to build the documentation locally and resolve and warnings or errors.

Writing documentation

We adopt the following guidelines for writing documentation:

- Documentation source files must formatted in reStructuredText (reST). A reference format specification is available on the Sphinx reST usage guide. Usage of Cross-referencing syntax in general and for Python objects in particular is recommended. We also support cross-referencing external documentation via sphinx.ext.intersphinx, when referring for example to `pandas.DataFrame`.
- To document Python objects, we also adopt the NumPy Docstring style. Examples are available here.
- We support also the Markdown format through the MyST-Parser.

- Jupyter notebooks placed below `docs/source/notebooks` are automatically rendered to HTML pages by the nbsphinx extension.

Building documentation

Scripts and tools to build documentation are located below `docs/`. To build documentation, `sphinx` and a couple of additional Python packages are required. You can get all the needed dependencies by running:

```
$ cd pygama
$ pip install '.[docs]'
```

Pandoc is also required to render Jupyter notebooks. To build documentation, run the following commands:

```
$ cd docs
$ make clean
$ make
```

Documentation can be then displayed by opening `build/html/index.html` with a web browser. Documentation for the *pygama* website is built and deployed by Read the Docs.

2.2.4 Versioning

Collaborators with push access to the GitHub repository that wish to release a new project version must implement the following procedures:

- Semantic versioning is adopted. The version string uses the `MAJOR.MINOR.PATCH` format.
- To release a new **minor** or **major version**, the following procedure should be followed:
  1. A new branch with name `releases/vMAJOR.MINOR` (note the `v`) containing the code at the intended stage is created
  2. The commit is tagged with a descriptive message: `git tag vMAJOR.MINOR.0 -m 'short descriptive message here'` (note the `v`)
  3. Changes are pushed to the remote:

     ```
     $ git push origin releases/vMAJOR.MINOR
     $ git push origin refs/tags/vMAJOR.MINOR.0
     ```

- To release a new **patch version**, the following procedure should be followed:
  1. A commit with the patch is created on the relevant release branch `releases/vMAJOR.MINOR`
  2. The commit is tagged: `git tag vMAJOR.MINOR.PATCH` (note the `v`)
  3. Changes are pushed to the remote:

     ```
     $ git push origin releases/vMAJOR.MINOR
     $ git push origin refs/tags/vMAJOR.MINOR.PATCH
     ```

- To upload the release to the Python Package Index, a new release must be created through the GitHub interface, associated to the just created tag. Usage of the "Generate release notes" option is recommended.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X