A Python-based Software Stack for the LEGEND Experiment

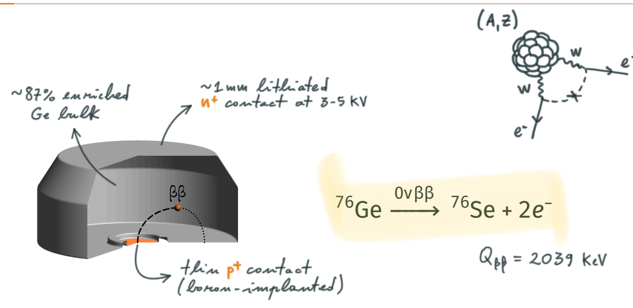
L. Pertoldi <luigi.pertoldi@tum.de>

PyHEP • 9 October 2023

TU München, INFN Padova

IS SEARCHING FOR $0\nu\beta\beta$ WITH ENRICHED GERMANIUM

LEGEND



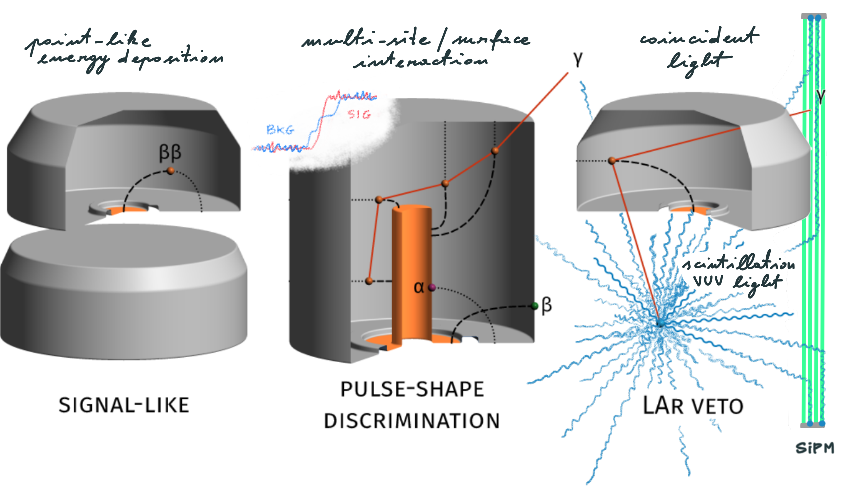
- Enriched High-Purity Germanium detectors
- LAr light collection system: WLS fibers & SiPMs
- μ-veto: water Cherenkov & scintillating panels

μ-panels
H_2O
LAr
SiPMs
HPGe
Fibers
PMTs

SIGNAL AND BACKGROUND EVENT SIGNATURES

--- holes (+)
.... electrons (-)

LEGEND



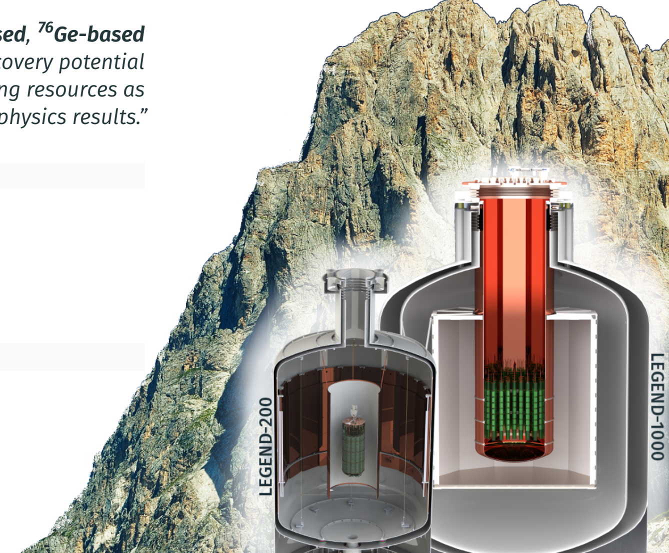
*"The collaboration aims to develop a **phased**, **^{76}Ge-based** double-beta decay experimental program with discovery potential at a **half-life beyond 10^{28} yr**, using existing resources as appropriate to expedite physics results."*

LEGEND-200

- **200 kg** of ^{enr}Ge (×5 yr), in GERDA cryostat
- Taking data now at LNGS
- $B \sim \mathbf{2 \cdot 10^{-4}}$ cts / (keV kg yr) $\longmapsto T^{0\nu}_{1/2} > \mathbf{10^{27}}$ **yr**

LEGEND-1000 arXiv 2107.11462 "pre-conceptual design report"

- **1 ton** of ^{enr}Ge (×10 yr), awaiting funding
- $B < \mathbf{10^{-5}}$ cts / (keV kg yr) $\longmapsto T^{0\nu}_{1/2} > \mathbf{10^{28}}$ **yr**
- Cover full $\langle m_{\beta\beta} \rangle$ inverted ordering region

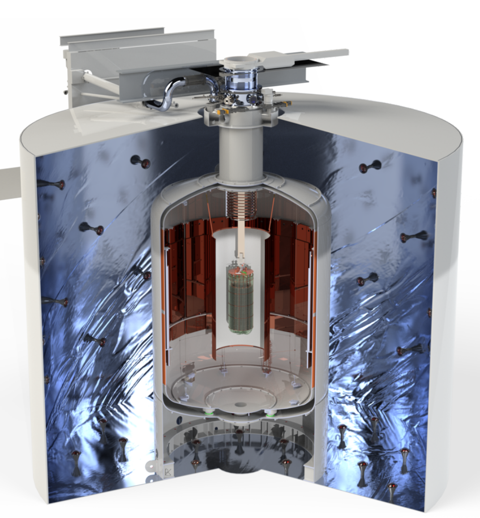


LEGEND-200 started probing the inverted $\langle m_{\beta\beta} \rangle$ region and informs the LEGEND-1000 design

- ^{enr}Ge material (92% enrichment) secured
- Large ICPC detectors > 3 kg
- Improved LAr system efficiency

Status & plans:

- Commissioning completed
- ~ 140 kg deployed, taking data!
- First report at TAUP 2023



In GERDA and MAJORANA

- C++ (ROOT) based
- Closed source
- Written long time ago, un-maintained

In LEGEND we are rewriting everything from scratch!

Primary python stack

- Large scientific computing community
- Easy to write and distribute code
- Becoming increasingly popular also in HEP
- Fast learning curve, good on industry CV

Secondary julia stack

- As fast as C, easy as Python
- Scientific computing friendly
- Cross-check of reference analysis
- Test bench for new technologies

All open source on GitHub: legend-exp

Summary: LEGEND Python packages for data analysis:

- /L/legend-pydataobj ⟼ data objects, HDF5 I/O, waveform compression
- /L/legend-daq2lh5 ⟼ convert digitized data to LEGEND HDF5
- /L/dspeed ⟼ fast digital signal processing for particle detector signals
- /L/pygama ⟼ high-level data analysis/loading routines, glue package
- /L/pylegendmeta ⟼ metadata system interface
- /L/legend-dataflow ⟼ data processing automation
- /L/legend-data-monitor ⟼ near-time data monitoring

Data objects and I/O


```
Table(dict={'packet_id': Array([1 2 ... 9 10], attrs={'datatype': 'array<1>{real}'}), 'eventnumber'
9], attrs={'datatype': 'array<1>{real}'}), 'timestamp': Array([1.63836878e+09 1.63836878e+09 ... 1.
6878e+09], attrs={'datatype': 'array<1>{real}', 'units': 's'}), 'runtime': Array([0.53781918 0.5529
 0.85165747], attrs={'datatype': 'array<1>{real}', 'units': 's'}), 'numtraces': Array([6 6 ... 6 6]
: 'array<1>{real}'}), 'tracelist': VectorOfVectors(flattened_data=Array([0 1 ... 4 5], attrs={'data
al}'}), cumulative_length=Array([6 12 ... 54 60], attrs={'datatype': 'array<1>{real}'}), attrs={'da
array<1>{real}}'}), 'baseline': Array([7722 7725 ... 7726 7725], attrs={'datatype': 'array<1>{real}
rray([85 230 ... 54 85], attrs={'datatype': 'array<1>{real}'}), 'channel': Array([5 5 ... 5 5], att
ray<1>{real}'}), 'ts_pps': Array([0 0 ... 0 0], attrs={'datatype': 'array<1>{real}'}), 'ts_ticks':
8230812 ... 211326880 212914368], attrs={'datatype': 'array<1>{real}'}), 'ts_maxticks': Array([2499
 249999999 249999999], attrs={'datatype': 'array<1>{real}'}), 'mu_offset_sec': Array([1638368781 16
8781 1638368781], attrs={'datatype': 'array<1>{real}'}), 'mu_offset_usec': Array([654536 654536 ...
trs={'datatype': 'array<1>{r                 c': Array([1638368781 1638368781 ... 1638368781 1
'datatype': 'array<1>{real}'}), 'delta_mu_usec': Array([-345463 -345463 ... -345463 -345463], attrs
y<1>{real}'}), 'abs_delta_mu_usec': Array([345463 345463 ... 345463 345463], attrs={'datatype': 'ar
o_start_sec': Array([0 0 ... 0 0], attrs={'datatype': 'array<1>{real}'}), 'to_start_usec': Array([5
0185 530185], attrs={'datatype': 'array<1>{real}'}), 'dr_start_pps': Array([0 0 ... 0 0], attrs={'d
{real}'}), 'dr_start_ticks': Array([0 0 ... 0 0], attrs={'datatype': 'array<1>{real}'}), 'dr_stop_p
 0 0], attrs={'datatype': 'array<1>{real}'}), 'dr_stop_ticks': Array([0 0 ... 0 0], attrs={'datatyp
'}), 'dr_maxticks': Array([249999999 249999999 ... 249999999 249999999], attrs={'datatype': 'array<
ime': Array([0. 0. ... 0. 0.], attrs={'datatype': 'array<1>{real}'}), 'waveform': WaveformTable(dic
0. ... 0. 0.], attrs={'datatype': 'array<1>{real}', 'units': 'ns'}), 'dt': Array([16. 16. ... 16. 1
pe': 'array<1>{real}', 'units': 'ns'}), 'values': ArrayOfEqualSizedArrays([[7730 7730 ... 7717 7720
25 7724] ... [7721 7721 ... 7726 7723] [7721 7721 ... 7729 7723]], attrs={'datatype': 'array_of_equ
{real}'})}, attrs={'datatype': 'table{t0,dt,values}'})}, attrs={'datatype': 'table{packet_id,eventn
time,numtraces,tracelist,baseline,daqenergy,channel,ts_pps,ts_ticks,ts_maxticks,mu_offset_sec,mu_of
_sec,delta_mu_usec,abs_delta_mu_usec,to_start_sec,to_start_usec,dr_start_pps,dr_start_ticks,dr_stop
```

THE LEGEND DATA FORMAT

LEGEND defines an abstract data model and implementation into the file format

- Public at legend-exp.github.io/legend-data-format-specs
- Long-term data accessibility and FAIR data principles

Binary data format: HDF5

- Hierarchical (like ROOT files)
- Efficient: parallel I/O, compression
- Portable, well supported, self-describing

Implementation:

- in python ⟼ /legend-pydataobj
- in julia ⟼ /LegendHDF5IO.jl

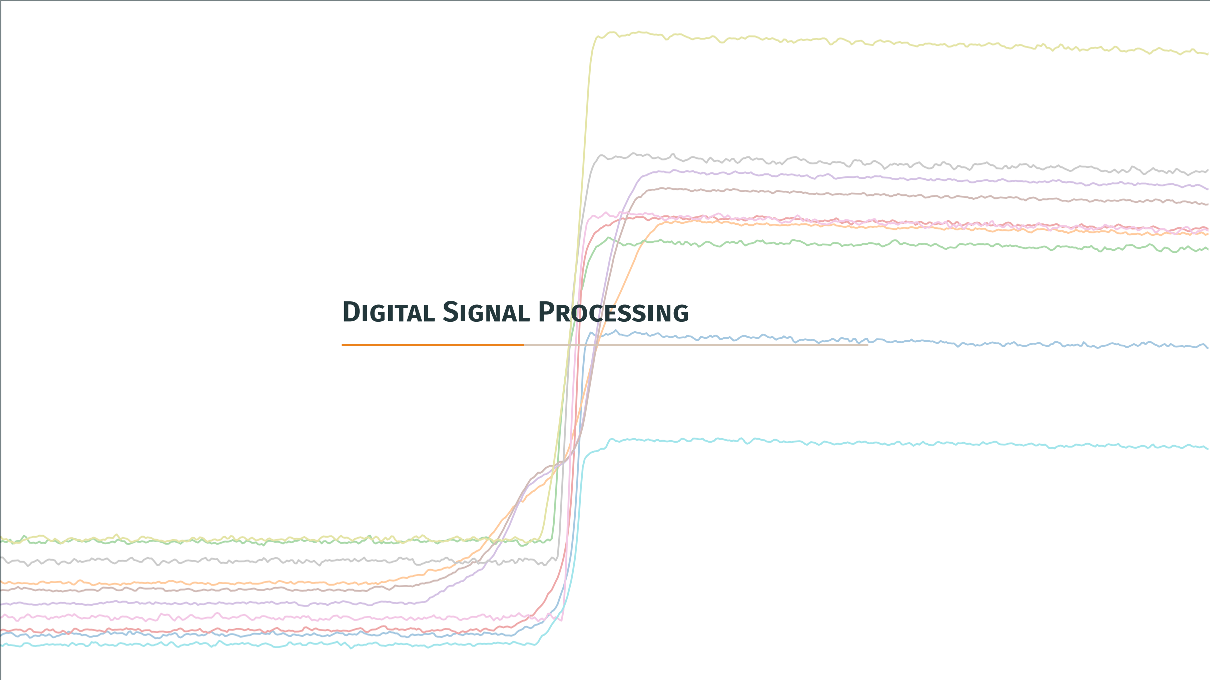
Quick reference

Data type	datatype attribute
Scalar	real, string, symbol, bool, ...
Flat n-dimensional array	array<n>{ELTYPE}
Fixed-sized n-dimensional array	fixedsize_array<n>{ELTYPE}
n-dimensional array of m-dimensional arrays of the same size	array_of_equalsized_arrays<n,m>{ELTYPE}
Vector of vectors of different size	array<1>{array<1>{ELTYPE}}
Struct	struct{FIELDNAME_1,FIELDNAME_2,...}
Table	table{COLNAME_1,COLNAME_2,...}
Enum	enum{NAME_1=INT_VAL_1,NAME_2=INT_VAL_2,...}
Encoded vector of vectors of different size	array<1>{encoded_array<1>{ELTYPE}}
Encoded array of arrays of the same size	array_of_equalsized_encoded_arrays<n,m>{ELTYPE}

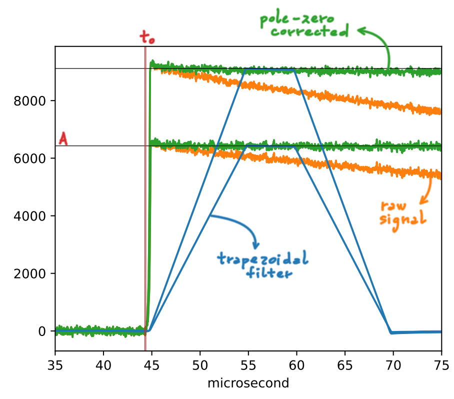
- LEGEND Data Objects (LGDO) in Python
  - Types derived from Python/Numpy classes: `Array`, `Table`, `WaveformTable`, ...
  - Metadata attributes (HDF5, physical units, compression, ...)
  - Other convenience methods
- HDF5 I/O (via h5py) and iteration routines
  - Including sliced reading
- Hardware-accelerated (via Numba) custom lossless compression algorithms for particle detector signals [spec]
  - `ULEB128ZigZagDiff`: variable-length base-128 encoding of waveform differences
  - `RadwareSigcompress`: adaptive encoding algorithm by D. Radford
- Tutorial online legend-pydataobj

- Convert digitized data from various FADC output formats to HDF5 (LEGEND format)
- Control decoding/output layout through JSON configuration files
- Optional waveform pre-processing (trimming, down-sampling etc.) (through / /dspeed)
- Compression (through / /legend-pydataobj)
- Tutorial online legend-daq2lh5

```
raw-data.lh5
└── ch1121605 · HDF5 group → stream / detector
    └── raw · table{ ... }
        ├── baseline · array<1>{real}
        ├── board_id · array<1>{real}
        ├── channel · array<1>{real}
        ├── crate · array<1>{real}
        ├── daqenergy · array<1>{real}
        ├── deadtime · array<1>{real}
        ├── event_type · array<1>{real}
        ├── eventnumber · array<1>{real}
        ├── numtraces · array<1>{real}
        ├── packet_id · array<1>{real}
        ├── runtime · array<1>{real}
        ├── timestamp · array<1>{real}
        ├── tracelist · array<1>{array<1>{real}}
        │   ├── cumulative_length · array<1>{real}
        │   └── flattened_data · array<1>{real}
        └── waveform · table{t0,dt,values}
            ├── dt · array<1>{real}
            ├── t0 · array<1>{real}
            └── values · array_of_equalsized_arrays<1,1>{real}
```


Digital Signal Processing

- *Pulse Shape Discrimination* and *excellent energy resolution* require high-resolution traces
  - ...and a rather long chain of (often computationally expensive) DSP filters
- Traces are stored on disk for offline processing
- Being able to reprocess large data sets in a couple of days is key to deliver timely physics results
- Need for fast software solutions



- Waveforms stored in rectangular structures separately for each data stream

```
>>> print(channel_42.waveform_table.values)
[[13712 13712 13683 ... 15445 15380 15400]
 [13072 13072 12992 ... 18842 18819 18806]
 [13575 13575 13496 ... 18409 18384 18457]
 ...
 [15405 15405 15366 ... 36208 36214 36233]
 [ 9962  9962  9949 ... 13269 13326 13269]
 [16918 16918 16962 ... 18715 18877 18933]]
```

- We use Numba to hardware-accelerate DSP filters on the CPU
  - SIMD vectorization
  - NumPy broadcasting
  - Efficient memory management
  - Simple to implement

```python
import numpy as np
from numba import guvectorize

@guvectorize(
    [
        "void(float32[:], int32, int32, float32[:])",
        "void(float64[:], int32, int32, float64[:])",
    ],
    "(n),(),()→(n)",
    **nb_kwargs,
)
def trap_filter(
    w_in: np.ndarray,
    rise: int,
    flat: int,
    w_out: np.ndarray
) → None:
    """Apply a symmetric trapezoidal filter to the waveform.
```

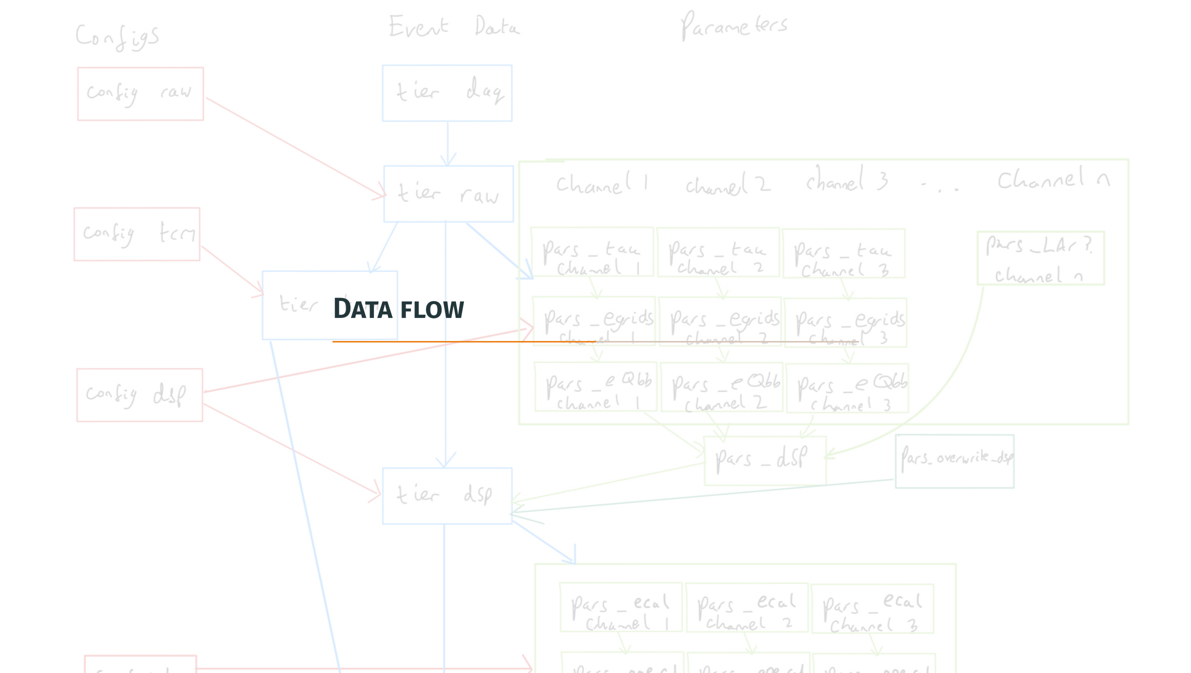
- Complex JSON-configurable DSP chains
    - Can import external ufuncs
    - Support for physical units
    - Waveform slicing
    - Support for optimization loops
    - Support for array inputs/output
- Tutorial/examples online dspeed

Potentially interesting package for the community!

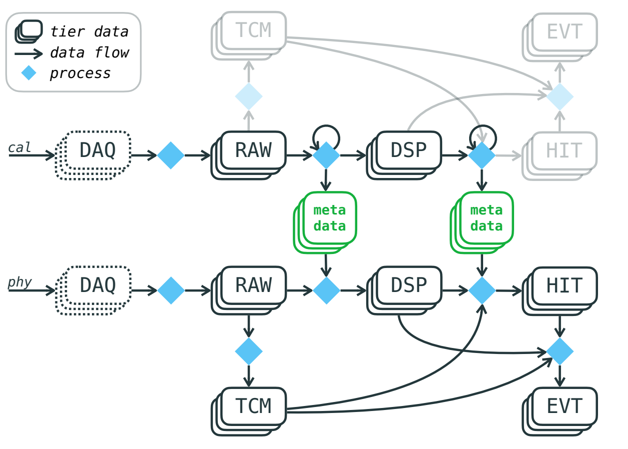
```
1  {
2    "outputs": ["tp_min", "tp_max", "wf_min", "wf_max", "trapEmax"],
3    "processors": {
4      "tp_min, tp_max, wf_min, wf_max": {
5        "function": "min_max",
6        "module": "dspeed.processors",
7        "args": ["waveform", "tp_min", "tp_max", "wf_min", "wf_max"],
8        "unit": ["ns", "ns", "ADC", "ADC"]
9      },
10     "wf_pz": {
11       "function": "pole_zero",
12       "module": "dspeed.processors",
13       "args": ["wf_blsub", "db.pz.tau", "wf_pz"],
14       "unit": "ADC",
15       "defaults": { "db.pz.tau": "27460.5" }
16     },
17     "wf_t0_filter": {
18       "function": "t0_filter",
19       "module": "dspeed.processors",
20       "args": ["wf_pz", "wf_t0_filter(len(wf_pz), 'f', grid=wf_pz.grid)"],
21       "init_args": ["128*ns/wf_pz.period", "2*us/wf_pz.period"],
22       "unit": "ADC"
23     },
24     "trapEmax": {
25       "function": "amax",
26       "module": "numpy",
27       "args": ["wf_etrap", 1, "trapEmax"],
28       "kwargs": { "signature": "(n),()->()", "types": ["fi->f"] },
29       "unit": "ADC"
30     },
31     ...
```

- Event building routines (`build_hit()`, `build_tcm()`, ...)
- `pargen`: data cleaning, calibration, optimization routines *et al.*
- `DataLoader`: High-level, generic data loading of tier data
  - Lazy/sliced loading
  - File database
  - Event/hit output orientation
  - Chunked reading
  - JSON configuration
- Development/test of new software stack components
- Glues together the LEGEND framework

Data flow



LEGEND



raw raw waveforms, DAQ parameters

tcm event building info

dsp digital signal processing outputs

hit calibrated DSP outputs

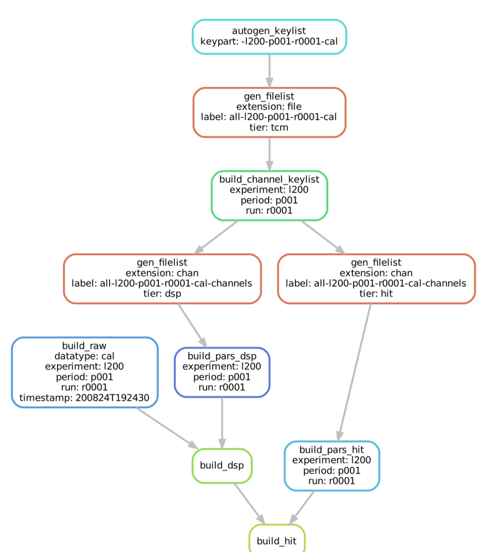
evt event-level classifiers

Note

Array-programming-ready flat/rectangular data layout

Workflow characteristics:

- Massive concurrency at the DAQ cycle (file) level
- Data streams (detectors) processed concurrently
- Manage calibration → physics data feedback and optimization loops
- Needed for automatic data production (monitoring) and offline re-processing



- GNU Make like logic
- Mini-language extending on python
- Integrates well with LEGEND software stack
- Seamlessly scalable to servers, clusters or grids
- Tasks run in the LEGEND software container

```
1  rule build_dsp:
2      input:
3          raw_file = get_pattern_tier_raw(setup),
4          tcm_file = get_pattern_tier_tcm(setup),
5          pars_file = ancient(get_pars_dsp_file)
6      params:
7          timestamp = "{timestamp}",
8          datatype = "{datatype}"
9      output:
10         get_pattern_tier_dsp(setup)
11     log:
12         get_pattern_log(setup, "tier_dsp")
13     shell:
14         """
15         {swenv} scripts/build_dsp.py
16           --log {log}
17           --pars_file {input.pars_file}
18           --datatype {params.datatype}
19           --timestamp {params.timestamp}
20           --input {input.raw_file}
21           --output {output}
22         """
```

Metadata

Metadata system requirements

Metadata is ubiquitous

physics/simulated data processing • detector production and characterization • hardware configuration • slow control • ...

Need for:

- Standard format, human and machine readable
- Version control
- Distribution system
- Accessibility (especially from code)
- Validity management (time, DAQ system, ...)

- JSON file system
  ⟼ transparent database [spec]
- Version control: Git
- Distribution: GitHub
- Multiple sub-repositories (scope or responsibility)
- /L/legend-metadata *[private]* collects them as submodules
  - Central access/distribution point
  - Global version control

```
legend-exp/legend-metadata
├── dataprod
│   ├── config -> legend-exp/legend-dataflow-config
│   ├── overrides -> legend-exp/legend-dataflow-overrides
│   └── runinfo.json
├── hardware
│   ├── configuration -> legend-exp/legend-hardware-config
│   │   └── channelmaps
│   │       ├── l200-p01-r001-T%-all-config.json
│   │       ├── ...
│   │       └── validity.jsonl
│   └── detectors -> legend-exp/legend-detectors
│       ├── germanium
│       │   ├── crystals
│       │   └── diodes
│       │       ├── V00048.json
│       │       ├── V00050.json
│       │       └── ...
│       └── ...
└── ...
```

Access with /L/pylegendmeta

LEGEND

- Git revision management
- `JsonDB` object representing a JSON database
- In memory it's an augmented Python `dict` (`AttrsDict`)
- Request metadata validity on time, category, ...
- Remote LEGEND slow control PosgreSQL database interface

```
>>> db = LegendMetadata("path/to/legend-metadata")
>>> isinstance(db, JsonDB)
True
>>> db["hardware"]
JsonDB("path/to/legend-metadata/hardware")
>>> db["hardware"]["detectors"]["germanium"]["diodes"]["V00048"]["production"]["mass_in_g"]
2000
>>> db.hardware.detectors.germanium.diodes.V00048.production.mass_in_g # attribute access
2000
>>> chmaps = db.hardware.configuration.channelmaps
>>> chmaps.on(datetime.now()) # time validity
>>> chmaps.map("daq.channel_id")[253] # re-index with label
>>> chmaps.map("detector.name").V00000A
{ ... }
```

More examples online pylegendmeta

Simulation software stack

THE LEGEND SIMULATIONS WORKFLOW

Geant4 application

post processing tools

probability density functions

optical response model

HPGe response model

hardware configuration

background model

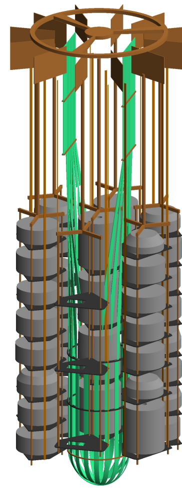
optical photon simulator

material assay

geometry model

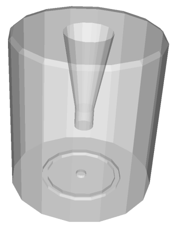
pulse shape simulator

data



Geometry development is time consuming and cumbersome

- Many experiments (LEGEND-200, LEGEND-1000, test stands, ...)
  $\longmapsto$ a lot of code
- C++ (GEANT4) is not the best language to implement them (neither GDML for complex geometries)
- What about re-using CAD models from engineers?



Idea:

- Geometry agnostic GEANT4 application (/L/remage)
- Use g4edge/pyg4ometry[1] to develop geometries in Python
- Export to GDML for GEANT4

LEGEND geometry modeling Python packages:

/L/legend-pygeom-hpges ⟼ germanium detectors
/L/legend-pygeom-optics ⟼ material optical properties (for GEANT4)
/L/legend-pygeom-l200 *[private]* ⟼ as-built LEGEND-200 setup
/L/legend-pygeom-l1000 *[private]* ⟼ LEGEND-1000 setups

[1] Don't miss tomorrow's talk at PyHEP by S. Boogert.

build_tier_raw jobid: 0001 simid: l200a-hpge-bulk-2vbb
build_tier_raw jobid: 0000 simid: l200a-hpge-bulk-2vbb
build_tier_raw jobid: 0002 simid: l200a-hpge-bulk-2vbb
build_tier_raw jobid: 0003 simid: l200a-hpge-bulk-2vbb
build_tier_raw jobid: 0004 simid: l200a-hpge-bulk-2vbb

l200a-optical-map-fiber.root
l200a-optical-map-lar.root
l200a-optical-map-pen.root

build_tier_hit
build_tier_hit
build_tier_hit
build_tier_hit
build_tier_hit

make_tier_evt_config_file runid: l200-p04-r002-phy
make_run_partition_file
make_tier_evt_config_file runid: l200-p04-r003-phy

build_tier_evt
build_tier_evt

legend-metadata

build_tier_pdf

The whole simulation production chain, from GEANT4 simulations to *pdfs*, is also automated via Snakemake (workflow: /legend/legend-simflow)

- Modern packaging practices
- pre-commit style checks
- Unit tests, code coverage tracking
- Documentation on Read The Docs
- PyPI deployment
- CI/CD via GitHub Actions

learn.scientific-python.org

- LEGEND is building an end-to-end Python software stack for the experiment
    - Leverage new technologies from the Python community
    - Improve software maintainability/preservation
    - A benefit for early-career members
- Strong interplay with HEP community packages
- LEGEND packages are potentially interesting for other experiments

Thanks for listening!