
vallенаe

Release 0.5.1

Daniel Altmann, Lukas Berbuer (Vallen Systeme GmbH)

Mar 25, 2021

LIBRARY DOCUMENTATION

1	IO	3
1.1	Database classes	3
1.2	Data types	20
1.3	Compression	35
2	Features	37
2.1	Acoustic Emission	37
2.2	Conversion	40
3	Timerpicker	41
3.1	vallенае.timepicker.hinkley	41
3.2	vallенае.timepicker.aic	42
3.3	vallенае.timepicker.energy_ratio	42
3.4	vallенае.timepicker.modified_energy_ratio	43
4	Examples	45
4.1	Read prddb	45
4.2	Read and plot transient data	47
4.3	Timepicker	48
4.4	Timepicker batch processing	53
4.5	Localisation	60
4.6	Go fast with multiprocessing	64
5	Changelog	67
5.1	Unreleased	67
5.2	0.5.1 - 2020-03-25	67
5.3	0.5.0 - 2020-03-18	67
5.4	0.4.0 - 2021-02-14	67
5.5	0.3.0 - 2020-11-05	68
5.6	0.2.4 - 2020-11-01	68
5.7	0.2.3 - 2020-09-01	68
5.8	0.2.2 - 2020-07-10	69
5.9	0.2.1 - 2020-02-10	69
5.10	0.2.0 - 2020-02-06	69
5.11	0.1.0 - 2020-01-24	70
6	ToDoS	71
7	Indices and tables	73
	Python Module Index	75

Extract and analyze Acoustic Emission measurement data.

The IO module `vallenae.io` enables reading (and writing) of Vallen Systeme SQLite database files:

- ***.pridb**: Primary database
- ***.tradb**: Transient data
- ***.trfdb**: Transient features

The remaining modules are system-independent and try to comprise the most common state-of-the-art algorithms in Acoustic Emission:

- `vallenae.features`: Extraction of Acoustic Emission features
- `vallenae.timepicker`: Timepicking algorithms for arrival time estimations

Read/write Vallen Systeme database and setup files.

1.1 Database classes

Classes to read/write pridb, tradb and trfdb database files.

Warning: Writing is still experimental

<i>PriDatabase</i> (filename[, mode])	IO Wrapper for pridb database file.
<i>TraDatabase</i> (filename[, mode, compression])	IO Wrapper for tradb database file.
<i>TrfDatabase</i> (filename[, mode])	IO Wrapper for trfdb (transient feature) database file.

1.1.1 vallenae.io.PriDatabase

class `vallenae.io.PriDatabase` (*filename*, *mode*='ro')
IO Wrapper for pridb database file.

Attributes

<i>connected</i>	Check if connected to SQLite database.
<i>filename</i>	Filename of database.

`vallenae.io.PriDatabase.connected`

property `PriDatabase.connected`
Check if connected to SQLite database.

Return type `bool`

vallenae.io.PriDatabase.filename**property** PriDatabase.filename

Filename of database.

Return type `str`**Methods**

<code>__init__(filename[, mode])</code>	Open pridb database file.
<code>channel()</code>	Get list of channels.
<code>close()</code>	Close database connection.
<code>columns()</code>	Columns of data table.
<code>connection()</code>	Get SQLite connection object.
<code>create(filename)</code>	Create empty pridb.
<code>fieldinfo()</code>	Read fieldinfo table.
<code>globalinfo()</code>	Read globalinfo table.
<code>iread_hits(*[, channel, time_start, ...])</code>	Stream hits with returned iterable.
<code>iread_markers(*[, time_start, time_stop, ...])</code>	Stream markers with returned iterable.
<code>iread_parametric(*[, time_start, time_stop, ...])</code>	Stream parametric data with returned iterable.
<code>iread_status(*[, channel, time_start, ...])</code>	Stream status data with returned iterable.
<code>listen([existing, wait, query_filter])</code>	Listen to database changes and return new records.
<code>read(**kwargs)</code>	Read all data set types (hits, markers, parametric data, status data) from pridb to Pandas DataFrame.
<code>read_hits(**kwargs)</code>	Read hits to Pandas DataFrame.
<code>read_markers(**kwargs)</code>	Read marker to Pandas DataFrame.
<code>read_parametric(**kwargs)</code>	Read parametric data to Pandas DataFrame.
<code>read_status(**kwargs)</code>	Read status data to Pandas DataFrame.
<code>rows()</code>	Number of rows in data table.
<code>tables()</code>	Get table names.
<code>write_fieldinfo(field, info)</code>	Write to fieldinfo table.
<code>write_hit(hit)</code>	Write hit to pridb.
<code>write_marker(marker)</code>	Write marker to pridb.
<code>write_parametric(parametric)</code>	Write parametric data to pridb.
<code>write_status(status)</code>	Write status data to pridb.

vallenae.io.PriDatabase.__init__

PriDatabase.__init__(filename, mode='ro')

Open pridb database file.

Parameters

- **filename** (`str`) – Path to pridb database file
- **mode** (`str`) – Define database access: “ro” (read-only), “rw” (read-write), “rwc” (read-write and create empty database if it does not exist)

vallenae.io.PriDatabase.channel

`PriDatabase.channel()`
 Get list of channels.
Return type `Set[int]`

vallenae.io.PriDatabase.close

`PriDatabase.close()`
 Close database connection.

vallenae.io.PriDatabase.columns

`PriDatabase.columns()`
 Columns of data table.
Return type `Tuple[str,...]`

vallenae.io.PriDatabase.connection

`PriDatabase.connection()`
 Get SQLite connection object.
Raises `RuntimeError` – If connection is closed
Return type `Connection`

vallenae.io.PriDatabase.create

static `PriDatabase.create(filename)`
 Create empty pridb.
Parameters `filename` (`str`) – Path to new pridb database file

vallenae.io.PriDatabase.fieldinfo

`PriDatabase.fieldinfo()`
 Read fieldinfo table.
 The fieldinfo table stores informations about columns of the data table (like units).
Return type `Dict[str,Dict[str,Any]]`
Returns Dict of column names and informations (again a dict)

vallenae.io.PriDatabase.globalinfo

`PriDatabase.globalinfo()`

Read globalinfo table.

Return type `Dict[str, Any]`

vallenae.io.PriDatabase.iread_hits

`PriDatabase.iread_hits(*, channel=None, time_start=None, time_stop=None, set_id=None, query_filter=None)`

Stream hits with returned iterable.

Parameters

- **channel** (`Union[None, int, Sequence[int]]`) – None if all channels should be read. Otherwise specify the channel number or a list of channel numbers
- **time_start** (`Optional[float]`) – Start reading at relative time (in seconds). Start at beginning if *None*
- **time_stop** (`Optional[float]`) – Stop reading at relative time (in seconds). Read until end if *None*
- **set_id** (`Union[None, int, Sequence[int]]`) – Read by SetID
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “Amp > 5000 AND RiseT < 1000”

Return type `SizedIterable[HitRecord]`

Returns Sized iterable to sequential read hits

vallenae.io.PriDatabase.iread_markers

`PriDatabase.iread_markers(*, time_start=None, time_stop=None, set_id=None, query_filter=None)`

Stream markers with returned iterable.

Parameters

- **time_start** (`Optional[float]`) – Start reading at relative time (in seconds). Start at beginning if *None*
- **time_stop** (`Optional[float]`) – Stop reading at relative time (in seconds). Read until end if *None*
- **set_id** (`Union[None, int, Sequence[int]]`) – Read by SetID
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “Number > 11 AND Data LIKE ‘%TimeZone%’”

Return type `SizedIterable[MarkerRecord]`

Returns Sized iterable to sequential read markers

vallenae.io.PriDatabase.iread_parametric

`PriDatabase.iread_parametric` (*, *time_start=None*, *time_stop=None*, *set_id=None*,
query_filter=None)

Stream parametric data with returned iterable.

Parameters

- **time_start** (`Optional[float]`) – Start reading at relative time (in seconds). Start at beginning if *None*
- **time_stop** (`Optional[float]`) – Stop reading at relative time (in seconds). Read until end if *None*
- **set_id** (`Union[None, int, Sequence[int]]`) – Read by SetID
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “PA0 >= -5000 AND PA0 < 5000”

Return type `SizedIterable[ParametricRecord]`

Returns Sized iterable to sequential read parametric data

vallenae.io.PriDatabase.iread_status

`PriDatabase.iread_status` (*, *channel=None*, *time_start=None*, *time_stop=None*, *set_id=None*,
query_filter=None)

Stream status data with returned iterable.

Parameters

- **channel** (`Union[None, int, Sequence[int]]`) – None if all channels should be read. Otherwise specify the channel number or a list of channel numbers
- **time_start** (`Optional[float]`) – Start reading at relative time (in seconds). Start at beginning if *None*
- **time_stop** (`Optional[float]`) – Stop reading at relative time (in seconds). Read until end if *None*
- **set_id** (`Union[None, int, Sequence[int]]`) – Read by SetID
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “RMS < 300 OR RMS > 500”

Return type `SizedIterable[StatusRecord]`

Returns Sized iterable to sequential read status data

vallenae.io.PriDatabase.listen

`PriDatabase.listen` (*existing=False*, *wait=False*, *query_filter=None*)

Listen to database changes and return new records.

Parameters

- **existing** (`bool`) – Return already existing records
- **wait** (`bool`) – Wait for new records even if no acquisition (writer) is active. Otherwise the function returns after all records are read.

- **query_filter** (*Optional[str]*) – Optional query filter provided as SQL clause, e.g. “Time >= 100 AND Chan == 2”

Yields New hit/marker/parametric/status data records

Return type *Iterable[Union[HitRecord, MarkerRecord, ParametricRecord, StatusRecord]]*

vallenae.io.PriDatabase.read

`PriDatabase.read(**kwargs)`

Read all data set types (hits, markers, parametric data, status data) from pridb to Pandas DataFrame.

Parameters ****kwargs** – Arguments passed to *iread_hits*, *iread_markers*, *iread_parametric* and *iread_status*

Return type *DataFrame*

Returns Pandas DataFrame with all pridb data set types

vallenae.io.PriDatabase.read_hits

`PriDatabase.read_hits(**kwargs)`

Read hits to Pandas DataFrame.

Parameters ****kwargs** – Arguments passed to *iread_hits*

Return type *DataFrame*

Returns Pandas DataFrame with hit data

vallenae.io.PriDatabase.read_markers

`PriDatabase.read_markers(**kwargs)`

Read marker to Pandas DataFrame.

Parameters ****kwargs** – Arguments passed to *iread_markers*

Return type *DataFrame*

Returns Pandas DataFrame with marker data

vallenae.io.PriDatabase.read_parametric

`PriDatabase.read_parametric(**kwargs)`

Read parametric data to Pandas DataFrame.

Parameters ****kwargs** – Arguments passed to *iread_parametric*

Return type *DataFrame*

Returns Pandas DataFrame with parametric data

vallenae.io.PriDatabase.read_status

`PriDatabase.read_status(**kwargs)`

Read status data to Pandas DataFrame.

Parameters `**kwargs` – Arguments passed to `iread_status`

Return type `DataFrame`

Returns Pandas DataFrame with status data

vallenae.io.PriDatabase.rows

`PriDatabase.rows()`

Number of rows in data table.

Return type `int`

vallenae.io.PriDatabase.tables

`PriDatabase.tables()`

Get table names.

Return type `Set[str]`

vallenae.io.PriDatabase.write_fieldinfo

`PriDatabase.write_fieldinfo(field, info)`

Write to fieldinfo table.

Parameters

- **field** (`str`) – Column name of data table
- **info** (`Dict[str, Any]`) – Dict of properties and values, e.g. {"Unit": "[Hz]"}

Raises `ValueError` – If field is not a column of data table

vallenae.io.PriDatabase.write_hit

`PriDatabase.write_hit(hit)`

Write hit to pridb.

Caution: `HitRecord.set_id` is ignored and automatically incremented.

Parameters **hit** (`HitRecord`) – Hit data set

Returns Index (`SetID`) of inserted row

Todo: Status flag

vallenae.io.PriDatabase.write_marker

`PriDatabase.write_marker(marker)`

Write marker to pridb.

Caution: `MarkerRecord.set_id` is ignored and automatically incremented.

Parameters `marker` (`MarkerRecord`) – Marker data set

Returns Index (SetID) of inserted row

vallenae.io.PriDatabase.write_parametric

`PriDatabase.write_parametric(parametric)`

Write parametric data to pridb.

Caution: `ParametricRecord.set_id` is ignored and automatically incremented.

Parameters `parametric` (`ParametricRecord`) – Parametric data set

Returns Index (SetID) of inserted row

Todo: Status flag

vallenae.io.PriDatabase.write_status

`PriDatabase.write_status(status)`

Write status data to pridb.

Caution: `StatusRecord.set_id` is ignored and automatically incremented.

Parameters `status` (`StatusRecord`) – Status data set

Returns Index (SetID) of inserted row

Todo: Status flag

1.1.2 vallenae.io.TraDatabase

class `vallenae.io.TraDatabase(filename, mode='ro', *, compression=False)`

IO Wrapper for tradb database file.

Attributes

<code>connected</code>	Check if connected to SQLite database.
<code>filename</code>	Filename of database.

vallenae.io.TraDatabase.connected

property `TraDatabase.connected`
Check if connected to SQLite database.

Return type `bool`

vallenae.io.TraDatabase.filename

property `TraDatabase.filename`
Filename of database.

Return type `str`

Methods

<code>__init__(filename[, mode, compression])</code>	Open tradb database file.
<code>channel()</code>	Get list of channels.
<code>close()</code>	Close database connection.
<code>columns()</code>	Columns of data table.
<code>connection()</code>	Get SQLite connection object.
<code>create(filename)</code>	Create empty tradb.
<code>fieldinfo()</code>	Read fieldinfo table.
<code>globalinfo()</code>	Read globalinfo table.
<code>iread(*[, channel, time_start, time_stop, ...])</code>	Stream transient data with returned Iterable.
<code>listen([existing, wait, query_filter])</code>	Listen to database changes and return new records.
<code>read(**kwargs)</code>	Read transient data to Pandas DataFrame.
<code>read_continuous_wave(channel[, time_start, ...])</code>	Read transient signal of specified channel to a single, continuous array.
<code>read_wave(trai[, time_axis])</code>	Read transient signal for a given TRAI (transient recorder index).
<code>rows()</code>	Number of rows in data table.
<code>tables()</code>	Get table names.
<code>write(trai)</code>	Write transient data to pridb.
<code>write_fieldinfo(field, info)</code>	Write to fieldinfo table.

vallenae.io.TraDatabase.__init__

`TraDatabase.__init__(filename, mode='ro', *, compression=False)`
Open tradb database file.

Parameters

- **filename** (`str`) – Path to tradb database file
- **mode** (`str`) – Define database access: “ro” (read-only), “rw” (read-write), “rwc” (read-write and create empty database if it does not exist)
- **compression** (`bool`) – Enable/disable FLAC compression data BLOBs for writing

vallena.io.TraDatabase.channel

`TraDatabase.channel()`
Get list of channels.
Return type `Set[int]`

vallena.io.TraDatabase.close

`TraDatabase.close()`
Close database connection.

vallena.io.TraDatabase.columns

`TraDatabase.columns()`
Columns of data table.
Return type `Tuple[str,...]`

vallena.io.TraDatabase.connection

`TraDatabase.connection()`
Get SQLite connection object.
Raises `RuntimeError` – If connection is closed
Return type `Connection`

vallena.io.TraDatabase.create

static `TraDatabase.create(filename)`
Create empty tradb.
Parameters `filename` (`str`) – Path to new tradb database file

vallena.io.TraDatabase.fieldinfo

`TraDatabase.fieldinfo()`
Read fieldinfo table.
The fieldinfo table stores informations about columns of the data table (like units).
Return type `Dict[str,Dict[str,Any]]`
Returns Dict of column names and informations (again a dict)

vallenae.io.TraDatabase.globalinfo

`TraDatabase.globalinfo()`

Read globalinfo table.

Return type `Dict[str, Any]`

vallenae.io.TraDatabase.iread

`TraDatabase.iread(*, channel=None, time_start=None, time_stop=None, trai=None, query_filter=None)`
Stream transient data with returned Iterable.

Parameters

- **channel** (`Union[None, int, Sequence[int]]`) – None if all channels should be read. Otherwise specify the channel number or a list of channel numbers
- **time_start** (`Optional[float]`) – Start reading at relative time (in seconds). Start at beginning if *None*
- **time_stop** (`Optional[float]`) – Stop reading at relative time (in seconds). Read until end if *None*
- **trai** (`Union[None, int, Sequence[int]]`) – Read data by TRAI (transient recorder index)
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “Pretrigger == 500 AND Samples >= 1024”

Return type `SizedIterable[TraRecord]`

Returns Sized iterable to sequential read transient data

vallenae.io.TraDatabase.listen

`TraDatabase.listen(existing=False, wait=False, query_filter=None)`

Listen to database changes and return new records.

Parameters

- **existing** (`bool`) – Return already existing records
- **wait** (`bool`) – Wait for new records even if no acquisition (writer) is active. Otherwise the function returns after all records are read.
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “TRAI >= 100 AND Samples >= 1024”

Yields New transient data records

Return type `Iterable[TraRecord]`

vallenae.io.TraDatabase.read

`TraDatabase.read(**kwargs)`

Read transient data to Pandas DataFrame.

Parameters ****kwargs** – Arguments passed to `iread`

Return type `DataFrame`

Returns Pandas DataFrame with transient data

vallenae.io.TraDatabase.read_continuous_wave

`TraDatabase.read_continuous_wave(channel, time_start=None, time_stop=None, *, time_axis=True, show_progress=True)`

Read transient signal of specified channel to a single, continuous array.

The signal is exactly cropped to the given time range. Time gaps are filled with 0's.

Parameters

- **channel** (`int`) – Channel number to read
- **time_start** (`Optional[float]`) – Start reading at relative time (in seconds). Start at beginning if *None*
- **time_stop** (`Optional[float]`) – Stop reading at relative time (in seconds). Read until end if *None*
- **time_axis** (`bool`) – Create the correspondig time axis. Default: *True*
- **show_progress** (`bool`) – Show progress bar. Default: *True*

Return type `Union[Tuple[ndarray, ndarray], Tuple[ndarray, int]]`

Returns

If *time_axis* is *True*

- Array with transient signal
- Time axis

If *time_axis* is *False*

- Array with transient signal
- Samplerate

vallenae.io.TraDatabase.read_wave

`TraDatabase.read_wave(trai, time_axis=True)`

Read transient signal for a given TRAI (transient recorder index).

This method is useful in combination with `PriDatabase.read_hits`, that will store the TRAI in a DataFrame.

Parameters

- **traid** (`int`) – Transient recorder index (unique key between pridb and tradb)
- **time_axis** (`bool`) – Create the correspondig time axis. Default: *True*

Return type `Union[Tuple[ndarray, ndarray], Tuple[ndarray, int]]`

Returns

If `time_axis` is *True*

- Array with transient signal
- Time axis

If `time_axis` is *False*

- Array with transient signal
- Samplerate

vallenae.io.TraDatabase.rows

`TraDatabase.rows()`

Number of rows in data table.

Return type `int`

vallenae.io.TraDatabase.tables

`TraDatabase.tables()`

Get table names.

Return type `Set[str]`

vallenae.io.TraDatabase.write

`TraDatabase.write(tra)`

Write transient data to pridb.

Parameters `tra` (`TraRecord`) – Transient data set

Return type `int`

Returns Index (SetID) of inserted row

Todo: Status flag

vallenae.io.TraDatabase.write_fieldinfo

`TraDatabase.write_fieldinfo(field, info)`

Write to fieldinfo table.

Parameters

- `field` (`str`) – Column name of data table
- `info` (`Dict[str, Any]`) – Dict of properties and values, e.g. {"Unit": "[Hz]"}

Raises `ValueError` – If field is not a column of data table

1.1.3 vallenae.io.TrfDatabase

class `vallenae.io.TrfDatabase` (*filename*, *mode*='ro')

IO Wrapper for trfdb (transient feature) database file.

Attributes

<code>connected</code>	Check if connected to SQLite database.
<code>filename</code>	Filename of database.

`vallenae.io.TrfDatabase.connected`

property `TrfDatabase.connected`

Check if connected to SQLite database.

Return type `bool`

`vallenae.io.TrfDatabase.filename`

property `TrfDatabase.filename`

Filename of database.

Return type `str`

Methods

<code>__init__(filename[, mode])</code>	Open trfdb database file.
<code>close()</code>	Close database connection.
<code>columns()</code>	Columns of data table.
<code>connection()</code>	Get SQLite connection object.
<code>create(filename)</code>	Create empty trfdb.
<code>fieldinfo()</code>	Read fieldinfo table.
<code>globalinfo()</code>	Read globalinfo table.
<code>iread(*[, trai, query_filter])</code>	Stream features with returned iterable.
<code>listen([existing, wait, query_filter])</code>	Listen to database changes and return new records.
<code>read(**kwargs)</code>	Read features to Pandas DataFrame.
<code>rows()</code>	Number of rows in data table.
<code>tables()</code>	Get table names.
<code>write(feature_set)</code>	Write feature record to trfdb.
<code>write_fieldinfo(field, info)</code>	Write to fieldinfo table.

vallenae.io.TrfDatabase.__init__

`TrfDatabase.__init__(filename, mode='ro')`
 Open trfdb database file.

Parameters

- **filename** (`str`) – Path to trfdb database file
- **mode** (`str`) – Define database access: “ro” (read-only), “rw” (read-write), “rwc” (read-write and create empty database if it does not exist)

vallenae.io.TrfDatabase.close

`TrfDatabase.close()`
 Close database connection.

vallenae.io.TrfDatabase.columns

`TrfDatabase.columns()`
 Columns of data table.

Return type `Tuple[str, ...]`

vallenae.io.TrfDatabase.connection

`TrfDatabase.connection()`
 Get SQLite connection object.

Raises `RuntimeError` – If connection is closed

Return type `Connection`

vallenae.io.TrfDatabase.create

static `TrfDatabase.create(filename)`
 Create empty trfdb.

Parameters **filename** (`str`) – Path to new trfdb database file

vallenae.io.TrfDatabase.fieldinfo

`TrfDatabase.fieldinfo()`
 Read fieldinfo table.

The fieldinfo table stores informations about columns of the data table (like units).

Return type `Dict[str, Dict[str, Any]]`

Returns Dict of column names and informations (again a dict)

vallenae.io.TrfDatabase.globalinfo

`TrfDatabase.globalinfo()`

Read globalinfo table.

Return type `Dict[str, Any]`

vallenae.io.TrfDatabase.iread

`TrfDatabase.iread(*, trai=None, query_filter=None)`

Stream features with returned iterable.

Parameters

- **trai** (`Union[None, int, Sequence[int]]`) – Read data by TRAI (transient recorder index)
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “FFT_CoG >= 150 AND CTP < 20”

Return type `SizedIterable[FeatureRecord]`

Returns Sized iterable to sequential read features

vallenae.io.TrfDatabase.listen

`TrfDatabase.listen(existing=False, wait=False, query_filter=None)`

Listen to database changes and return new records.

Parameters

- **existing** (`bool`) – Return already existing records
- **wait** (`bool`) – Wait for new records even if no acquisition (writer) is active. Otherwise the function returns after all records are read.
- **query_filter** (`Optional[str]`) – Optional query filter provided as SQL clause, e.g. “TRAI >= 100”

Yields New feature records

Return type `Iterable[FeatureRecord]`

vallenae.io.TrfDatabase.read

`TrfDatabase.read(**kwargs)`

Read features to Pandas DataFrame.

Parameters ****kwargs** – Arguments passed to `iread`

Return type `DataFrame`

Returns Pandas DataFrame with features

vallenae.io.TrfDatabase.rows

`TrfDatabase.rows()`
Number of rows in data table.

Return type `int`

vallenae.io.TrfDatabase.tables

`TrfDatabase.tables()`
Get table names.

Return type `Set[str]`

vallenae.io.TrfDatabase.write

`TrfDatabase.write(feature_set)`
Write feature record to trfdb.

Parameters `feature_set` (`FeatureRecord`) – Feature set

Return type `int`

Returns Index (trait) of inserted row

vallenae.io.TrfDatabase.write_fieldinfo

`TrfDatabase.write_fieldinfo(field, info)`
Write to fieldinfo table.

Parameters

- **field** (`str`) – Column name of data table
- **info** (`Dict[str, Any]`) – Dict of properties and values, e.g. {"Unit": "[Hz]"}

Raises `ValueError` – If field is not a column of data table

All database classes implement two different interfaces to access data:

Standard read_*

Read data to `pandas.DataFrame`, e.g. with `PriDatabase.read_hits`

```
>>> pridb = vae.io.PriDatabase("./examples/steel_plate/sample.pridb")
>>> df = pridb.read_hits() # save all hits to pandas dataframe
>>> df[["time", "channel"]] # output columns hit and channel
      time  channel
set_id
10      3.992771      3
11      3.992775      2
12      3.992813      4
13      3.992814      1
```

Streaming iread_*

Iterate through the data row by row. This is a memory-efficient solution ideal for batch processing. The return types are specific `typing.NamedTuple`, see [Data types](#).

Example with `PriDatabase.iread_hits`:

```

>>> prddb = vae.io.PriDatabase("./examples/steel_plate/sample.prddb")
>>> for hit in prddb.iread_hits():
...     print(f"time: {hit.time:0.4f}, channel: {hit.channel}")
...
time: 3.9928,    channel: 3
time: 3.9928,    channel: 2
time: 3.9928,    channel: 4
time: 3.9928,    channel: 1
>>> type(hit)
<class 'vallenae.io.datatypes.HitRecord'>

```

1.2 Data types

Records of the database are represented as `typing.NamedTuple`. Each record implements a class method `from_sql` to init from a SQLite row dictionary (column name: value).

<code>HitRecord(time, channel, param_id, ...)</code>	Hit record in prddb (SetType = 2).
<code>MarkerRecord(time, set_type, data, number, ...)</code>	Marker record in prddb (SetType = 4, 5, 6).
<code>StatusRecord(time, channel, param_id, ...)</code>	Status data record in prddb (SetType = 3).
<code>ParametricRecord(time, param_id, set_id, ...)</code>	Parametric data record in prddb (SetType = 1).
<code>TraRecord(time, channel, param_id, ...)</code>	Transient data record in tradb.
<code>FeatureRecord(trai, features, float)</code>	Transient feature record in trfdb.

1.2.1 vallenae.io.HitRecord

class `vallenae.io.HitRecord` (*time: float, channel: int, param_id: int, amplitude: float, duration: float, energy: float, rms: float, set_id: Optional[int] = None, threshold: Optional[float] = None, rise_time: Optional[float] = None, signal_strength: Optional[float] = None, counts: Optional[int] = None, trai: Optional[int] = None, cascade_hits: Optional[int] = None, cascade_counts: Optional[int] = None, cascade_energy: Optional[int] = None, cascade_signal_strength: Optional[int] = None*)

Hit record in prddb (SetType = 2).

Attributes

<code>amplitude</code>	Peak amplitude in volts
<code>cascade_counts</code>	Summed counts of hits in the same hit-cascade
<code>cascade_energy</code>	Summed energy of hits in the same hit-cascade
<code>cascade_hits</code>	Total number of hits in the same hit-cascade
<code>cascade_signal_strength</code>	disable=line-too-long
<code>channel</code>	Channel number
<code>counts</code>	Number of positive threshold crossings
<code>duration</code>	Hit duration in seconds
<code>energy</code>	Energy (EN 1330-9) in eu (1e-14 V ² s)
<code>param_id</code>	Parameter ID of table ae_params for ADC value conversion
<code>rise_time</code>	Rise time in seconds

continues on next page

Table 9 – continued from previous page

<i>rms</i>	RMS of the noise before the hit in volts
<i>set_id</i>	Unique identifier for data set in pridb
<i>signal_strength</i>	Signal strength in nVs (1e-9 Vs)
<i>threshold</i>	Threshold amplitude in volts
<i>time</i>	Time in seconds
<i>traid</i>	Transient recorder index (foreign key between pridb and tradb)

vallenae.io.HitRecord.amplitude

property `HitRecord.amplitude`
Peak amplitude in volts

vallenae.io.HitRecord.cascade_counts

property `HitRecord.cascade_counts`
Summed counts of hits in the same hit-cascade

vallenae.io.HitRecord.cascade_energy

property `HitRecord.cascade_energy`
Summed energy of hits in the same hit-cascade

vallenae.io.HitRecord.cascade_hits

property `HitRecord.cascade_hits`
Total number of hits in the same hit-cascade

vallenae.io.HitRecord.cascade_signal_strength

property `HitRecord.cascade_signal_strength`
disable=line-too-long

Type Summed signal strength of hits in the same hit-cascade # noqa # pylint

vallenae.io.HitRecord.channel

property `HitRecord.channel`
Channel number

vallena.io.HitRecord.counts

property `HitRecord.counts`
Number of positive threshold crossings

vallena.io.HitRecord.duration

property `HitRecord.duration`
Hit duration in seconds

vallena.io.HitRecord.energy

property `HitRecord.energy`
Energy (EN 1330-9) in eu ($1e-14$ V²s)

vallena.io.HitRecord.param_id

property `HitRecord.param_id`
Parameter ID of table `ae_params` for ADC value conversion

vallena.io.HitRecord.rise_time

property `HitRecord.rise_time`
Rise time in seconds

vallena.io.HitRecord.rms

property `HitRecord.rms`
RMS of the noise before the hit in volts

vallena.io.HitRecord.set_id

property `HitRecord.set_id`
Unique identifier for data set in `pridb`

vallena.io.HitRecord.signal_strength

property `HitRecord.signal_strength`
Signal strength in nVs ($1e-9$ Vs)

vallenae.io.HitRecord.threshold

property `HitRecord.threshold`
Threshold amplitude in volts

vallenae.io.HitRecord.time

property `HitRecord.time`
Time in seconds

vallenae.io.HitRecord.trai

property `HitRecord.trai`
Transient recorder index (foreign key between pridb and tradb)

Methods

<code>__init__()</code>	Initialize self.
<code>count(value, /)</code>	Return number of occurrences of value.
<code>from_sql(row)</code>	Create HitRecord from SQL row.
<code>index(value[, start, stop])</code>	Return first index of value.

vallenae.io.HitRecord.__init__

`HitRecord.__init__()`
Initialize self. See `help(type(self))` for accurate signature.

vallenae.io.HitRecord.count

`HitRecord.count(value, /)`
Return number of occurrences of value.

vallenae.io.HitRecord.from_sql

classmethod `HitRecord.from_sql(row)`
Create HitRecord from SQL row.

Parameters `row` (`Dict[str, Any]`) – Dict of column names and values

Return type `HitRecord`

vallenae.io.HitRecord.index

`HitRecord.index` (*value*, *start*=0, *stop*=9223372036854775807, /)

Return first index of value.

Raises `ValueError` if the value is not present.

1.2.2 vallenae.io.MarkerRecord

class `vallenae.io.MarkerRecord` (*time*: *float*, *set_type*: *int*, *data*: *str*, *number*: *Optional[int]* = *None*, *set_id*: *Optional[int]* = *None*)

Marker record in pridb (SetType = 4, 5, 6).

A marker can have different meanings depending on its `SetType`:

- 4: label
- 5: datetime data set, as it is inserted whenever recording is started by software
- 6: a section start marker. E.g. new sections are started, if acquisition settings changed

Attributes

<i>data</i>	Content of marker (label text or datetime)
<i>number</i>	Marker number
<i>set_id</i>	Unique identifier for data set in pridb
<i>set_type</i>	Marker type (see above)
<i>time</i>	Time in seconds

vallenae.io.MarkerRecord.data

property `MarkerRecord.data`

Content of marker (label text or datetime)

vallenae.io.MarkerRecord.number

property `MarkerRecord.number`

Marker number

vallenae.io.MarkerRecord.set_id

property `MarkerRecord.set_id`

Unique identifier for data set in pridb

vallenae.io.MarkerRecord.set_type

property MarkerRecord.**set_type**
Marker type (see above)

vallenae.io.MarkerRecord.time

property MarkerRecord.**time**
Time in seconds

Methods

<code>__init__()</code>	Initialize self.
<code>count(value, /)</code>	Return number of occurrences of value.
<code>from_sql(row)</code>	Create MarkerRecord from SQL row.
<code>index(value[, start, stop])</code>	Return first index of value.

vallenae.io.MarkerRecord.__init__

MarkerRecord.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

vallenae.io.MarkerRecord.count

MarkerRecord.**count**(value, /)
Return number of occurrences of value.

vallenae.io.MarkerRecord.from_sql

classmethod MarkerRecord.**from_sql**(row)
Create MarkerRecord from SQL row.

Parameters **row** (Dict[str, Any]) – Dict of column names and values

Return type MarkerRecord

vallenae.io.MarkerRecord.index

MarkerRecord.**index**(value, start=0, stop=9223372036854775807, /)
Return first index of value.

Raises ValueError if the value is not present.

1.2.3 vallenae.io.StatusRecord

```
class vallenae.io.StatusRecord(time: float, channel: int, param_id: int, energy: float, rms:  
                               float, set_id: Optional[int] = None, threshold: Optional[float]  
                               = None, signal_strength: Optional[float] = None)
```

Status data record in prddb (SetType = 3).

Attributes

<i>channel</i>	Channel number
<i>energy</i>	Energy (EN 1330-9) in eu (1e-14 V ² s)
<i>param_id</i>	Parameter ID of table ae_params for ADC value conversion
<i>rms</i>	RMS in volts
<i>set_id</i>	Unique identifier for data set in prddb
<i>signal_strength</i>	Signal strength in nVs (1e-9 Vs)
<i>threshold</i>	Threshold amplitude in volts
<i>time</i>	Time in seconds

vallenae.io.StatusRecord.channel

```
property StatusRecord.channel  
    Channel number
```

vallenae.io.StatusRecord.energy

```
property StatusRecord.energy  
    Energy (EN 1330-9) in eu (1e-14 V2s)
```

vallenae.io.StatusRecord.param_id

```
property StatusRecord.param_id  
    Parameter ID of table ae_params for ADC value conversion
```

vallenae.io.StatusRecord.rms

```
property StatusRecord.rms  
    RMS in volts
```

vallenae.io.StatusRecord.set_id

```
property StatusRecord.set_id  
    Unique identifier for data set in prddb
```

vallenae.io.StatusRecord.signal_strength

property StatusRecord.**signal_strength**
Signal strength in nVs (1e-9 Vs)

vallenae.io.StatusRecord.threshold

property StatusRecord.**threshold**
Threshold amplitude in volts

vallenae.io.StatusRecord.time

property StatusRecord.**time**
Time in seconds

Methods

<code>__init__()</code>	Initialize self.
<code>count(value, /)</code>	Return number of occurrences of value.
<code>from_sql(row)</code>	Create StatusRecord from SQL row.
<code>index(value[, start, stop])</code>	Return first index of value.

vallenae.io.StatusRecord.__init__

StatusRecord.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

vallenae.io.StatusRecord.count

StatusRecord.**count** (value, /)
Return number of occurrences of value.

vallenae.io.StatusRecord.from_sql

classmethod StatusRecord.**from_sql** (row)
Create StatusRecord from SQL row.

Parameters **row** (Dict[str, Any]) – Dict of column names and values

Return type StatusRecord

vallenae.io.StatusRecord.index

StatusRecord.**index** (*value*, *start*=0, *stop*=9223372036854775807, /)

Return first index of value.

Raises ValueError if the value is not present.

1.2.4 vallenae.io.ParametricRecord

```
class vallenae.io.ParametricRecord(time: float, param_id: int, set_id: Optional[int] = None,
                                   pctd: Optional[int] = None, pcta: Optional[int] = None,
                                   pa0: Optional[int] = None, pa1: Optional[int] = None,
                                   pa2: Optional[int] = None, pa3: Optional[int] = None,
                                   pa4: Optional[int] = None, pa5: Optional[int] = None,
                                   pa6: Optional[int] = None, pa7: Optional[int] = None)
```

Parametric data record in prddb (SetType = 1).

Attributes

<i>pa0</i>	Amplitude of parametric input 0 in volts
<i>pa1</i>	Amplitude of parametric input 1 in volts
<i>pa2</i>	Amplitude of parametric input 2 in volts
<i>pa3</i>	Amplitude of parametric input 3 in volts
<i>pa4</i>	Amplitude of parametric input 4 in volts
<i>pa5</i>	Amplitude of parametric input 5 in volts
<i>pa6</i>	Amplitude of parametric input 6 in volts
<i>pa7</i>	Amplitude of parametric input 7 in volts
<i>param_id</i>	Parameter ID of table ae_params for ADC value conversion
<i>pcta</i>	Analog hysteresis counter
<i>pctd</i>	Digital counter value
<i>set_id</i>	Unique identifier for data set in prddb
<i>time</i>	Time in seconds

vallenae.io.ParametricRecord.pa0

property ParametricRecord.**pa0**

Amplitude of parametric input 0 in volts

vallenae.io.ParametricRecord.pa1

property ParametricRecord.pa1
Amplitude of parametric input 1 in volts

vallenae.io.ParametricRecord.pa2

property ParametricRecord.pa2
Amplitude of parametric input 2 in volts

vallenae.io.ParametricRecord.pa3

property ParametricRecord.pa3
Amplitude of parametric input 3 in volts

vallenae.io.ParametricRecord.pa4

property ParametricRecord.pa4
Amplitude of parametric input 4 in volts

vallenae.io.ParametricRecord.pa5

property ParametricRecord.pa5
Amplitude of parametric input 5 in volts

vallenae.io.ParametricRecord.pa6

property ParametricRecord.pa6
Amplitude of parametric input 6 in volts

vallenae.io.ParametricRecord.pa7

property ParametricRecord.pa7
Amplitude of parametric input 7 in volts

vallenae.io.ParametricRecord.param_id

property ParametricRecord.param_id
Parameter ID of table ae_params for ADC value conversion

vallenae.io.ParametricRecord.pcta

property `ParametricRecord.pcta`
Analog hysteresis counter

vallenae.io.ParametricRecord.pctd

property `ParametricRecord.pctd`
Digital counter value

vallenae.io.ParametricRecord.set_id

property `ParametricRecord.set_id`
Unique identifier for data set in prddb

vallenae.io.ParametricRecord.time

property `ParametricRecord.time`
Time in seconds

Methods

<code>__init__()</code>	Initialize self.
<code>count(value, /)</code>	Return number of occurrences of value.
<code>from_sql(row)</code>	Create ParametricRecord from SQL row.
<code>index(value[, start, stop])</code>	Return first index of value.

vallenae.io.ParametricRecord.__init__

`ParametricRecord.__init__()`
Initialize self. See `help(type(self))` for accurate signature.

vallenae.io.ParametricRecord.count

`ParametricRecord.count(value, /)`
Return number of occurrences of value.

vallenae.io.ParametricRecord.from_sql

classmethod `ParametricRecord.from_sql(row)`
Create ParametricRecord from SQL row.

Parameters `row` (`Dict[str, Any]`) – Dict of column names and values

Return type `ParametricRecord`

vallenae.io.ParametricRecord.index

`ParametricRecord.index` (*value*, *start*=0, *stop*=9223372036854775807, /)

Return first index of value.

Raises ValueError if the value is not present.

1.2.5 vallenae.io.TraRecord

class `vallenae.io.TraRecord` (*time*: *float*, *channel*: *int*, *param_id*: *int*, *pretrigger*: *int*, *threshold*: *float*, *samplerate*: *int*, *samples*: *int*, *data_format*: *int*, *data*: *numpy.ndarray*, *trai*: *Optional[int]* = None, *rms*: *Optional[float]* = None)

Transient data record in tradb.

Todo: Remove RMS

Attributes

<i>channel</i>	Channel number
<i>data</i>	Transient signal in volts
<i>data_format</i>	Data format (0 = uncompressed, 2 = FLAC compression)
<i>param_id</i>	Parameter ID of table tr_params for ADC value conversion
<i>pretrigger</i>	Pretrigger samples
<i>rms</i>	RMS of the noise before the hit
<i>samplerate</i>	Samplerate in Hz
<i>samples</i>	Number of samples
<i>threshold</i>	Threshold amplitude in volts
<i>time</i>	Time in seconds
<i>trai</i>	Transient recorder index (foreign key between pridb and tradb)

vallenae.io.TraRecord.channel

property `TraRecord.channel`
Channel number

vallenae.io.TraRecord.data

property `TraRecord.data`
Transient signal in volts

vallena.io.TraRecord.data_format

property `TraRecord.data_format`
Data format (0 = uncompressed, 2 = FLAC compression)

vallena.io.TraRecord.param_id

property `TraRecord.param_id`
Parameter ID of table `tr_params` for ADC value conversion

vallena.io.TraRecord.pretrigger

property `TraRecord.pretrigger`
Pretrigger samples

vallena.io.TraRecord.rms

property `TraRecord.rms`
RMS of the noise before the hit

vallena.io.TraRecord.samplerate

property `TraRecord.samplerate`
Samplerate in Hz

vallena.io.TraRecord.samples

property `TraRecord.samples`
Number of samples

vallena.io.TraRecord.threshold

property `TraRecord.threshold`
Threshold amplitude in volts

vallena.io.TraRecord.time

property `TraRecord.time`
Time in seconds

vallenae.io.TraRecord.trai**property** TraRecord.**tra**i

Transient recorder index (foreign key between pridb and tradb)

Methods

<code>__init__()</code>	Initialize self.
<code>count(value, /)</code>	Return number of occurrences of value.
<code>from_sql(row)</code>	rtype TraRecord
<code>index(value[, start, stop])</code>	Return first index of value.

vallenae.io.TraRecord.__init__TraRecord.**__init__**()

Initialize self. See help(type(self)) for accurate signature.

vallenae.io.TraRecord.countTraRecord.**count**(value, /)

Return number of occurrences of value.

vallenae.io.TraRecord.from_sql**classmethod** TraRecord.**from_sql**(row)**Return type** TraRecord**vallenae.io.TraRecord.index**TraRecord.**index**(value, start=0, stop=9223372036854775807, /)

Return first index of value.

Raises ValueError if the value is not present.

1.2.6 vallenae.io.FeatureRecord**class** vallenae.io.**FeatureRecord**(*tra*: int, *features*: Dict[str, float])

Transient feature record in trfdb.

Attributes

<i>features</i>	Feature dictionary (feature name -> value)
<i>tra</i> i	Transient recorder index

vallenae.io.FeatureRecord.features

property FeatureRecord.**features**
Feature dictionary (feature name -> value)

vallenae.io.FeatureRecord.tra

property FeatureRecord.**tra**i
Transient recorder index

Methods

<i>__init__</i> ()	Initialize self.
<i>count</i> (value, /)	Return number of occurrences of value.
<i>from_sql</i> (row)	rtype FeatureRecord
<i>index</i> (value[, start, stop])	Return first index of value.

vallenae.io.FeatureRecord.__init__

FeatureRecord.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

vallenae.io.FeatureRecord.count

FeatureRecord.**count**(value, /)
Return number of occurrences of value.

vallenae.io.FeatureRecord.from_sql

classmethod FeatureRecord.**from_sql**(row)
Return type FeatureRecord

vallenae.io.FeatureRecord.index

`FeatureRecord.index` (*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises `ValueError` if the value is not present.

1.3 Compression

Transient signals in the tradb are stored as BLOBs of 16-bit ADC values – either uncompressed or compressed (FLAC). Following functions convert between BLOBs and arrays of voltage values.

<code>decode_data_blob</code> (<i>data_blob</i> , <i>data_format</i> , ...)	Decodes (compressed) 16-bit ADC values from BLOB to array of voltage values.
<code>encode_data_blob</code> (<i>data</i> , <i>data_format</i> , ...)	Encodes array of voltage values to BLOB of 16-bit ADC values for memory-efficient storage.

1.3.1 vallenae.io.decode_data_blob

`vallenae.io.decode_data_blob` (*data_blob*, *data_format*, *factor_millivolts*)

Decodes (compressed) 16-bit ADC values from BLOB to array of voltage values.

Parameters

- **data_blob** (`bytes`) – Blob from tradb
- **data_format** (`int`) –
 - 0: uncompressed
 - 2: FLAC compression
- **factor_millivolts** (`float`) – Factor from int16 representation to millivolts. Stored in tradb -> tr_params as 'TR_mV'

Return type `ndarray`

Returns Array of voltage values

1.3.2 vallenae.io.encode_data_blob

`vallenae.io.encode_data_blob` (*data*, *data_format*, *factor_millivolts*)

Encodes array of voltage values to BLOB of 16-bit ADC values for memory-efficient storage.

Parameters

- **data** (`ndarray`) – Array with voltage values
- **data_format** (`int`) –
 - 0: uncompressed
 - 2: FLAC compression
- **factor_millivolts** (`float`) – Factor from int16 representation to millivolts. Stored in tradb -> tr_params as 'TR_mV'

Return type `bytes`

Returns Data blob

FEATURES

2.1 Acoustic Emission

<code>peak_amplitude(data)</code>	Compute maximum absolute amplitude.
<code>peak_amplitude_index(data)</code>	Compute index of peak amplitude.
<code>is_above_threshold(data, threshold)</code>	Checks if absolute amplitudes are above threshold.
<code>first_threshold_crossing(data, threshold)</code>	Compute index of first threshold crossing.
<code>rise_time(data, threshold, samplerate[, ...])</code>	Compute the rise time.
<code>energy(data, samplerate)</code>	Compute the energy of a hit.
<code>signal_strength(data, samplerate)</code>	Compute the signal strength of a hit.
<code>counts(data, threshold)</code>	Compute the number of positive threshold crossings of a hit (counts).
<code>rms(data)</code>	Compute the root mean square (RMS) of an array.

2.1.1 `vallenae.features.peak_amplitude`

`vallenae.features.peak_amplitude(data)`

Compute maximum absolute amplitude.

Parameters `data` (`ndarray`) – Input array

Return type `float`

Returns Peak amplitude of the input array

2.1.2 `vallenae.features.peak_amplitude_index`

`vallenae.features.peak_amplitude_index(data)`

Compute index of peak amplitude.

Parameters `data` (`ndarray`) – Input array

Return type `int`

Returns Index of peak amplitude

2.1.3 vallenae.features.is_above_threshold

`vallenae.features.is_above_threshold(data, threshold)`

Checks if absolute amplitudes are above threshold.

Parameters

- **data** (`ndarray`) – Input array
- **threshold** (`float`) – Threshold amplitude

Return type `bool`

Returns True if input array is above threshold, otherwise False

2.1.4 vallenae.features.first_threshold_crossing

`vallenae.features.first_threshold_crossing(data, threshold)`

Compute index of first threshold crossing.

Parameters

- **data** (`ndarray`) – Input array
- **threshold** (`float`) – Threshold amplitude

Return type `Optional[int]`

Returns Index of first threshold crossing. None if threshold was not exceeded

2.1.5 vallenae.features.rise_time

`vallenae.features.rise_time(data, threshold, samplerate, first_crossing=None, index_peak=None)`

Compute the rise time.

The rise time is the time between the first threshold crossing and the peak amplitude.

Parameters

- **data** (`ndarray`) – Input array (hit)
- **threshold** (`float`) – Threshold amplitude (in volts)
- **samplerate** (`int`) – Sample rate of the input array
- **first_crossing** (`Optional[int]`) – Precomputed index of first threshold crossing to save computation time
- **index_peak** (`Optional[int]`) – Precomputed index of peak amplitude to save computation time

Return type `float`

2.1.6 vallenae.features.energy

`vallenae.features.energy` (*data*, *samplerate*)

Compute the energy of a hit.

Energy is the integral of the squared AE-signal over time (EN 1330-9). The unit of energy is eu. 1 eu corresponds to $1e-14$ V²s.

Parameters

- **data** (`ndarray`) – Input array (hit)
- **samplerate** (`int`) – Sample rate of input array in Hz

Return type `float`

Returns Energy of input array (hit)

2.1.7 vallenae.features.signal_strength

`vallenae.features.signal_strength` (*data*, *samplerate*)

Compute the signal strength of a hit.

Signal strength is the integral of the rectified AE-signal over time. The unit of Signal Strength is nVs ($1e-9$ Vs).

Parameters

- **data** (`ndarray`) – Input array (hit)
- **samplerate** (`int`) – Sample rate of input array in Hz

Return type `float`

Returns Signal strength of input array (hit)

2.1.8 vallenae.features.counts

`vallenae.features.counts` (*data*, *threshold*)

Compute the number of positive threshold crossings of a hit (counts).

Parameters

- **data** (`ndarray`) – Input array
- **threshold** (`float`) – Threshold amplitude

Return type `int`

Returns Number of positive threshold crossings

2.1.9 vallenae.features.rms

`vallenae.features.rms` (*data*)

Compute the root mean square (RMS) of an array.

Parameters **data** (`ndarray`) – Input array

Return type `float`

Returns RMS of the input array

References

https://en.wikipedia.org/wiki/Root_mean_square

2.2 Conversion

<code>amplitude_to_db(amplitude[, reference])</code>	Convert amplitude from volts to decibel (dB).
<code>db_to_amplitude(amplitude_db[, reference])</code>	Convert amplitude from decibel (dB) to volts.

2.2.1 `vallenae.features.amplitude_to_db`

`vallenae.features.amplitude_to_db(amplitude, reference=1e-06)`

Convert amplitude from volts to decibel (dB).

Parameters

- **`amplitude`** (`float`) – Amplitude in volts
- **`reference`** (`float`) – Reference amplitude. Defaults to 1 μ V for dB(AE)

Return type `float`

Returns Amplitude in dB(ref)

2.2.2 `vallenae.features.db_to_amplitude`

`vallenae.features.db_to_amplitude(amplitude_db, reference=1e-06)`

Convert amplitude from decibel (dB) to volts.

Parameters

- **`amplitude_db`** (`float`) – Amplitude in dB
- **`reference`** (`float`) – Reference amplitude. Defaults to 1 μ V for dB(AE)

Return type `float`

Returns Amplitude in volts

TIMERPICKER

The determination of signal arrival times has a major influence on the localization accuracy. Usually, arrival times are determined by the first threshold crossing (either fixed or adaptive). Following popular methods have been proposed in the past to automatically pick time of arrivals:

<code>hinkley(arr[, alpha])</code>	Hinkley criterion for arrival time estimation.
<code>aic(arr)</code>	Akaike Information Criterion (AIC) for arrival time estimation.
<code>energy_ratio(arr[, win_len])</code>	Energy ratio for arrival time estimation.
<code>modified_energy_ratio(arr[, win_len])</code>	Modified energy ratio method for arrival time estimation.

3.1 `vallena.timepicker.hinkley`

`vallena.timepicker.hinkley(arr, alpha=5)`

Hinkley criterion for arrival time estimation.

The Hinkley criterion is defined as the partial energy of the signal (cumulative square sum) with an applied negative trend (characterized by alpha).

The starting value of alpha is reduced iteratively to avoid wrong picks within the pre-trigger part of the signal. Usually alpha values are chosen to be between 2 and 200 to ensure minimal delay. The chosen alpha value for the Hinkley criterion influences the results significantly.

Parameters

- **arr** (`ndarray`) – Transient signal of hit
- **alpha** (`int`) – Divisor of the negative trend. Default: 5

Return type `Tuple[ndarray, int]`

Returns

- Array with computed detection function
- Index of the estimated arrival time (max value)

Todo: Weak performance, if used with default parameter alpha

References

- Molenda, M. (2016). Acoustic Emission monitoring of laboratory hydraulic fracturing experiments. Ruhr-Universität Bochum.
- van Rijn, N. (2017). Investigating the Behaviour of Acoustic Emission Waves Near Cracks: Using the Finite Element Method. Delft University of Technology.

3.2 vallenae.timepicker.aic

`vallenae.timepicker.aic(arr)`

Akaike Information Criterion (AIC) for arrival time estimation.

The AIC picker basically models the signal as an autoregressive (AR) process. A typical AE signal can be subdivided into two parts. The first part containing noise and the second part containing noise and the AE signal. Both parts of the signal contain non deterministic parts (noise) describable by a Gaussian distribution.

Parameters `arr` (`ndarray`) – Transient signal of hit

Return type `Tuple[ndarray, int]`

Returns

- Array with computed detection function
- Index of the estimated arrival time (max value)

References

- Molenda, M. (2016). Acoustic Emission monitoring of laboratory hydraulic fracturing experiments. Ruhr-Universität Bochum.
- Bai, F., Gagar, D., Foote, P., & Zhao, Y. (2017). Comparison of alternatives to amplitude thresholding for onset detection of acoustic emission signals. *Mechanical Systems and Signal Processing*, 84, 717–730.
- van Rijn, N. (2017). Investigating the Behaviour of Acoustic Emission Waves Near Cracks: Using the Finite Element Method. Delft University of Technology.

3.3 vallenae.timepicker.energy_ratio

`vallenae.timepicker.energy_ratio(arr, win_len=100)`

Energy ratio for arrival time estimation.

Method based on preceding and following energy collection windows.

Parameters

- `arr` (`ndarray`) – Transient signal of hit
- `win_len` (`int`) – Samples of sliding windows. Default: 100

Return type `Tuple[ndarray, int]`

Returns

- Array with computed detection function
- Index of the estimated arrival time (max value)

References

- Han, L., Wong, J., & Bancroft, J. C. (2009). Time picking and random noise reduction on microseismic data. CREWES Research Report, 21, 1–13.

3.4 vallenae.timepicker.modified_energy_ratio

`vallenae.timepicker.modified_energy_ratio(arr, win_len=100)`

Modified energy ratio method for arrival time estimation.

The modifications improve the ability to detect the onset of a seismic arrival in the presence of random noise.

Parameters

- **arr** (`ndarray`) – Transient signal of hit
- **win_len** (`int`) – Samples of sliding windows. Default: 100

Return type `Tuple[ndarray, int]`

Returns

- Array with computed detection function
- Index of the estimated arrival time (max value)

References

- Han, L., Wong, J., & Bancroft, J. C. (2009). Time picking and random noise reduction on microseismic data. CREWES Research Report, 21, 1–13.

EXAMPLES

A collection of examples how to read and analyse Acoustic Emission data.

4.1 Read pridb

```
import os

import matplotlib.pyplot as plt

import vallenae as vae

HERE = os.path.dirname(__file__) if "__file__" in locals() else os.getcwd()
PRIDB = os.path.join(HERE, "steel_plate/sample.pridb")
```

4.1.1 Open pridb

```
pridb = vae.io.PriDatabase(PRIDB)

print("Tables in database: ", pridb.tables())
print("Number of rows in data table (ae_data): ", pridb.rows())
print("Set of all channels: ", pridb.channel())
```

Out:

```
Tables in database: {'ae_params', 'ae_markers', 'data_integrity', 'ae_fieldinfo',
→ 'acq_setup', 'ae_data', 'ae_globalinfo'}
Number of rows in data table (ae_data): 18
Set of all channels: {1, 2, 3, 4}
```

4.1.2 Read hits to Pandas DataFrame

```
df_hits = pridb.read_hits()
# Print a few columns
print(df_hits[["time", "channel", "amplitude", "counts", "energy"]])
```

Out:

```

Hits: 0%|          | 0/4 [00:00<?, ?it/s]
Hits: 100%|#####| 4/4 [00:00<00:00, 10761.52it/s]
      time  channel  amplitude  counts      energy
set_id
10    3.992771      3    0.046539    2180  2.799510e+07
11    3.992775      2    0.059621    2047  2.276279e+07
12    3.992813      4    0.034119    1854  1.286700e+07
13    3.992814      1    0.029115    1985  1.265275e+07

```

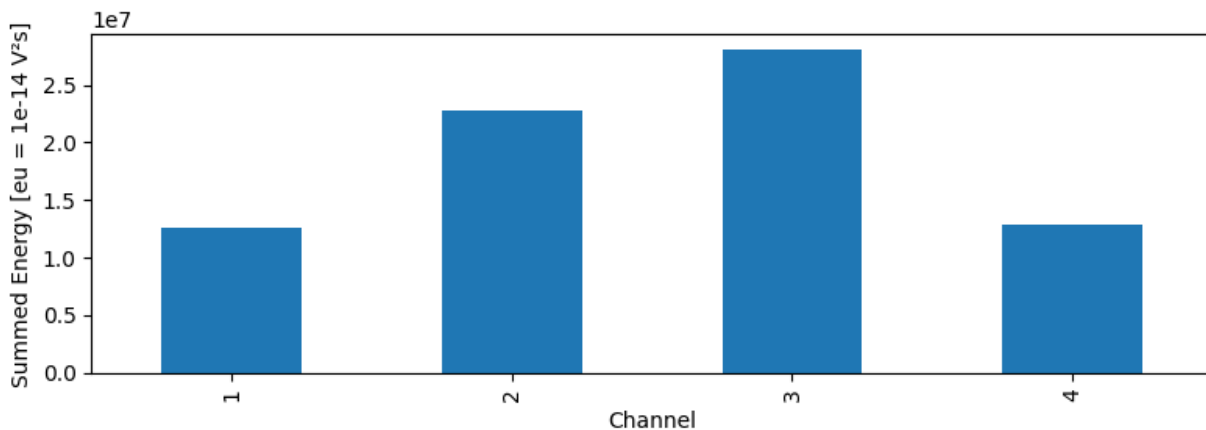
4.1.3 Query Pandas DataFrame

DataFrames offer powerful features to query and aggregate data, e.g. plot summed energy per channel

```

ax = df_hits.groupby("channel").sum()["energy"].plot.bar(figsize=(8, 3))
ax.set_xlabel("Channel")
ax.set_ylabel("Summed Energy [eu = 1e-14 V²s]")
plt.tight_layout()
plt.show()

```



4.1.4 Read markers

```

df_markers = pridb.read_markers()
print(df_markers)

```

Out:

```

Marker: 0%|          | 0/5 [00:00<?, ?it/s]
Marker: 100%|#####| 5/5 [00:00<00:00, 17697.49it/s]
      time  set_type      data  number
set_id
1      0.00        6              1
2      0.00        4      10:52 Resume  1
3      0.00        5      2019-09-20 10:54:52 <NA>
4      0.00        4  TimeZone: +02:00 (W. Europe Standard Time)  2
18    100.07        4      10:56 Suspend  3

```

4.1.5 Read parametric data

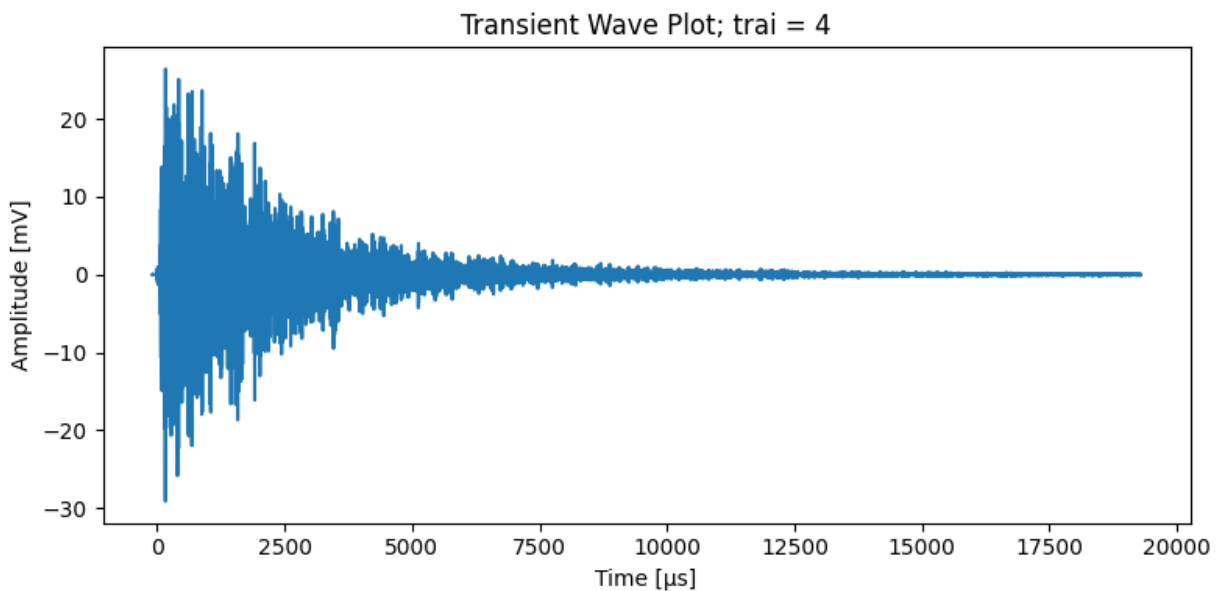
```
df_parametric = pridb.read_parametric()
print(df_parametric)
```

Out:

```
Parametric: 0%|          | 0/9 [00:00<?, ?it/s]
Parametric: 100%|#####| 9/9 [00:00<00:00, 25488.68it/s]
   time  param_id  pctd  pcta
set_id
5      0.00        1    0    0
6      1.00        1    0    0
7      2.00        1    0    0
8      3.00        1    0    0
9      3.99        1    0    0
14     4.00        1    0    0
15     5.00        1    0    0
16     6.00        1    0    0
17     6.45        1    0    0
```

Total running time of the script: (0 minutes 0.233 seconds)

4.2 Read and plot transient data



```
import os

import matplotlib.pyplot as plt

import vallenae as vae

HERE = os.path.dirname(__file__) if "__file__" in locals() else os.getcwd()
```

(continues on next page)

(continued from previous page)

```
TRADB = os.path.join(HERE, "steel_plate/sample_plain.tradb") # uncompressed
TRAI = 4 # just an example, no magic here

def main():
    # Read waveform from tradb
    with vae.io.TraDatabase(TRADB) as tradb:
        y, t = tradb.read_wave(TRAI)

    y *= 1e3 # in mV
    t *= 1e6 # for μs

    # Plot waveforms
    plt.figure(figsize=(8, 4), tight_layout=True)
    plt.plot(t, y)
    plt.xlabel("Time [μs]")
    plt.ylabel("Amplitude [mV]")
    plt.title(f"Transient Wave Plot; trai = {TRAI}")
    plt.show()

if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.194 seconds)

4.3 Timepicker

Following example showcases the results of different timepicking methods. For more informations, please refer to the functions documentation ([vallenae.timepicker](#)).

```
import os
import time

import matplotlib.pyplot as plt
import numpy as np

import vallenae as vae

HERE = os.path.dirname(__file__) if "__file__" in locals() else os.getcwd()
TRADB = os.path.join(HERE, "steel_plate/sample_plain.tradb")

TRAI = 4
SAMPLES = 2000

plt.ioff() # Turn interactive mode off; plt.show() is blocking
```

4.3.1 Read waveform from tradb

```
tradb = vae.io.TraDatabase(TRADB)

y, t = tradb.read_wave(TRAI)
# crop first samples
t = t[:SAMPLES]
y = y[:SAMPLES]
# unit conversion
t *= 1e6 # convert to  $\mu$ s
y *= 1e3 # convert to mV
```

4.3.2 Prepare plotting with time-picker results

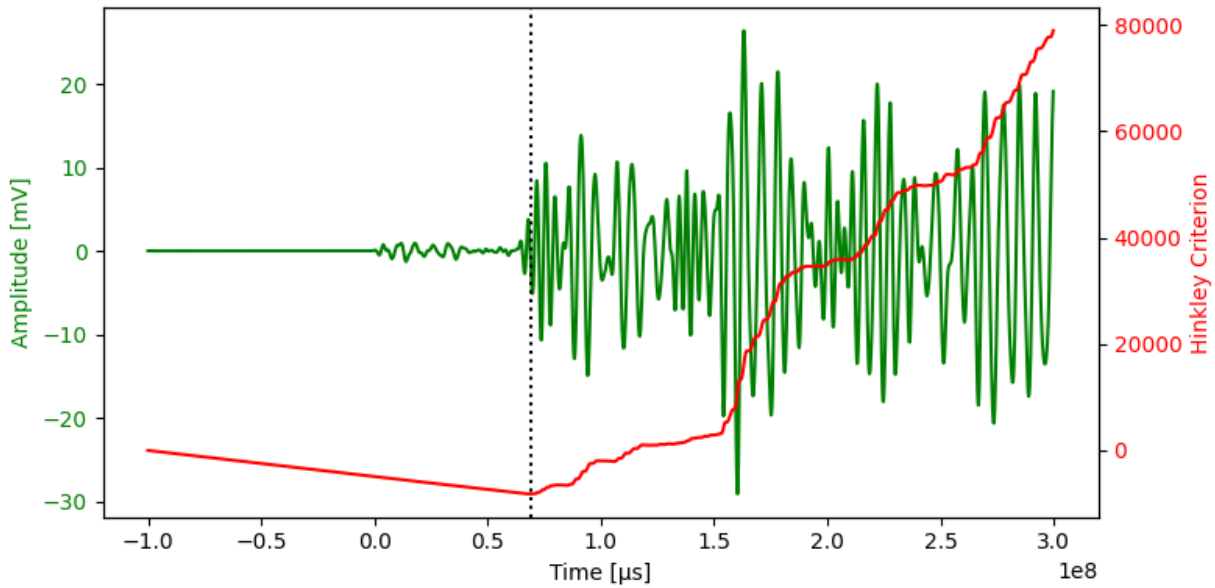
```
def plot(t_wave, y_wave, y_picker, index_picker, name_picker):
    _, ax1 = plt.subplots(figsize=(8, 4), tight_layout=True)
    ax1.set_xlabel("Time [ $\mu$ s]")
    ax1.set_ylabel("Amplitude [mV]", color="g")
    ax1.plot(t_wave, y_wave, color="g")
    ax1.tick_params(axis="y", labelcolor="g")

    ax2 = ax1.twinx()
    ax2.set_ylabel(f"{name_picker}", color="r")
    ax2.plot(t_wave, y_picker, color="r")
    ax2.tick_params(axis="y", labelcolor="r")

    plt.axvline(t_wave[index_picker], color="k", linestyle=":")
    plt.show()
```

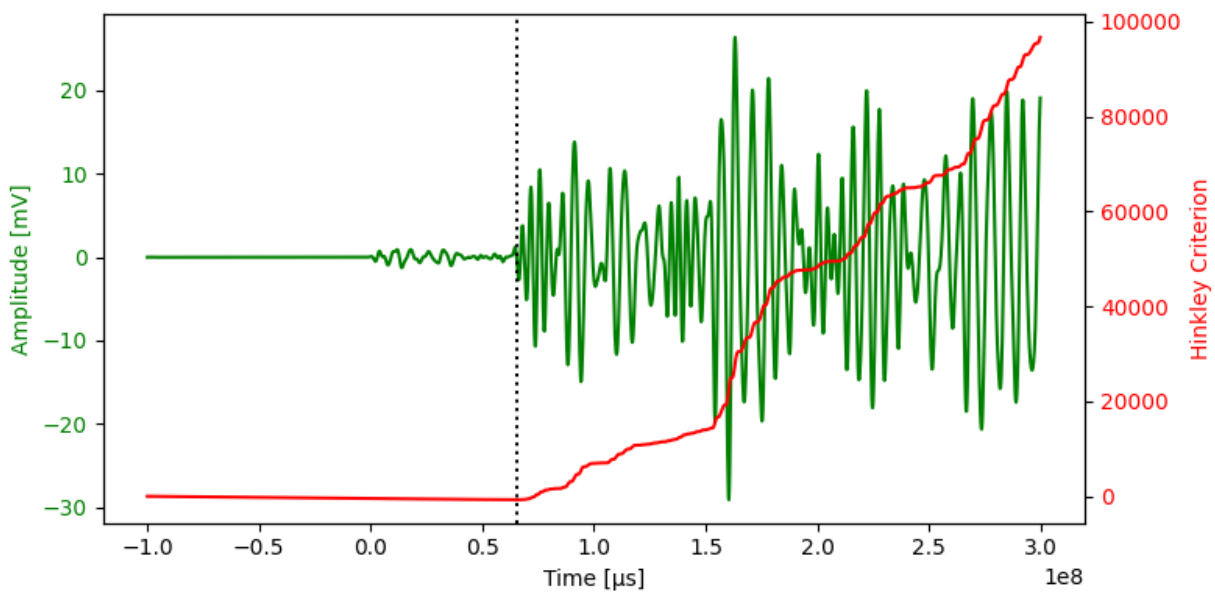
4.3.3 Hinkley Criterion

```
hc_arr, hc_index = vae.timepicker.hinkley(y, alpha=5)
plot(t, y, hc_arr, hc_index, "Hinkley Criterion")
```



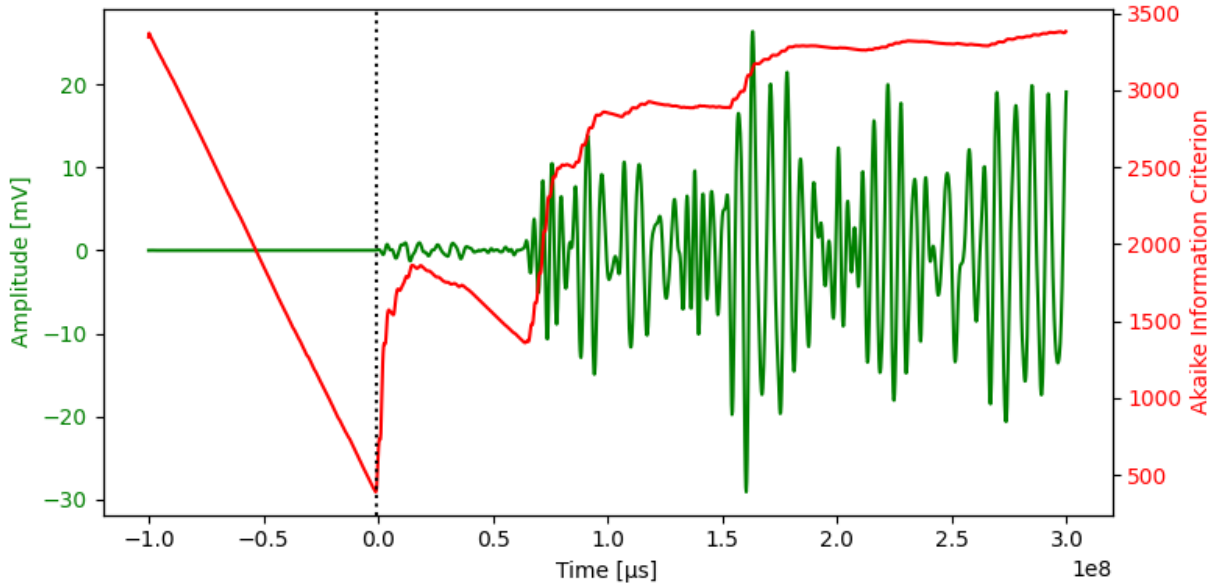
The negative trend correlates to the chosen alpha value and can influence the results strongly. Results with **alpha = 50** (less negative trend):

```
hc_arr, hc_index = vae.timepicker.hinkley(y, alpha=50)
plot(t, y, hc_arr, hc_index, "Hinkley Criterion")
```



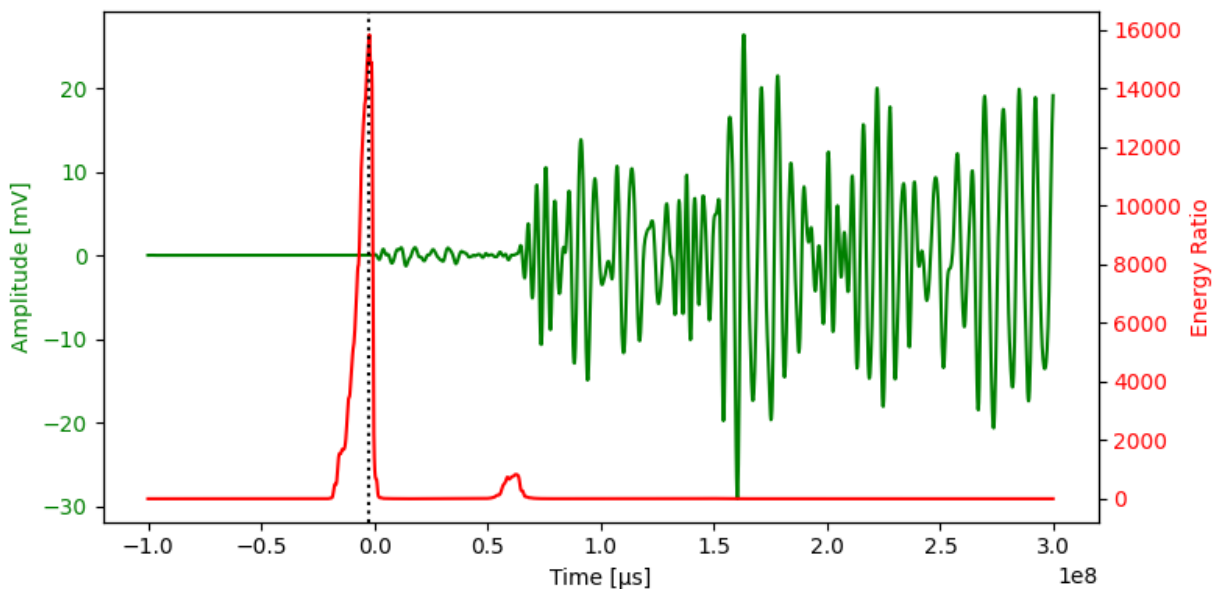
4.3.4 Akaike Information Criterion (AIC)

```
aic_arr, aic_index = vae.timepicker.aic(y)
plot(t, y, aic_arr, aic_index, "Akaike Information Criterion")
```



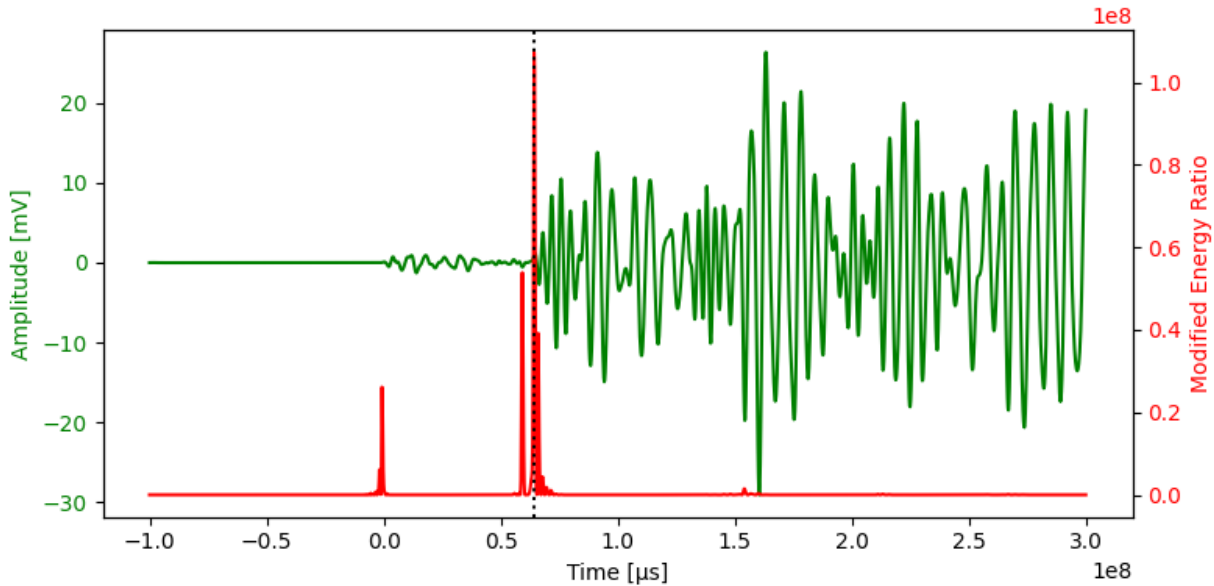
4.3.5 Energy Ratio

```
er_arr, er_index = vae.timepicker.energy_ratio(y)
plot(t, y, er_arr, er_index, "Energy Ratio")
```



4.3.6 Modified Energy Ratio

```
mer_arr, mer_index = vae.timepicker.modified_energy_ratio(y)
plot(t, y, mer_arr, mer_index, "Modified Energy Ratio")
```



4.3.7 Performance comparison

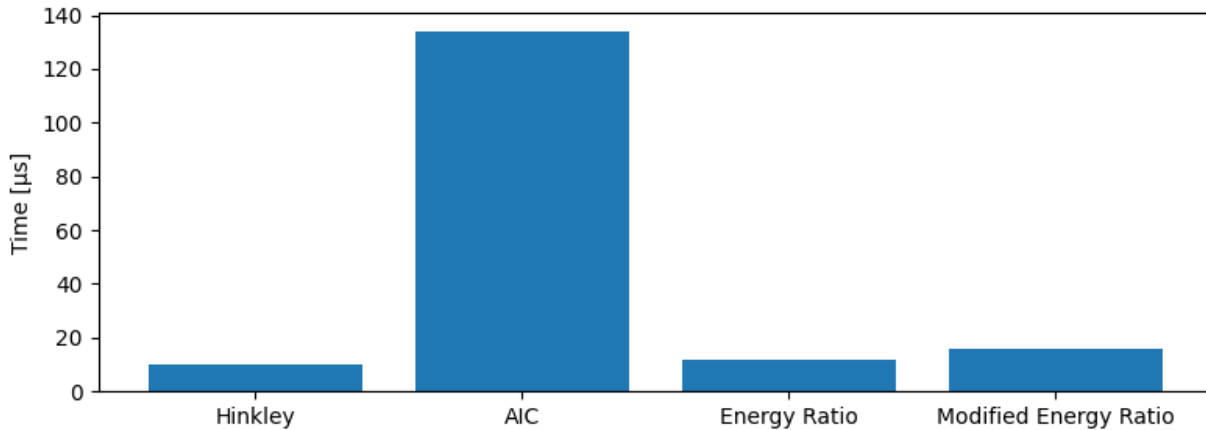
All timepicker implementations are using Numba for just-in-time (JIT) compilations. Usually the first function call is slow, because it will trigger the JIT compiler. To compare the performance to a native or numpy implementation, the average of multiple executions should be compared.

```
def timeit(callable, loops=100):
    time_start = time.perf_counter()
    for _ in range(loops):
        callable()
    return 1e6 * (time.perf_counter() - time_start) / loops # elapsed time in μs

timer_results = {
    "Hinkley": timeit(lambda: vae.timepicker.hinkley(y, 5)),
    "AIC": timeit(lambda: vae.timepicker.aic(y)),
    "Energy Ratio": timeit(lambda: vae.timepicker.energy_ratio(y)),
    "Modified Energy Ratio": timeit(lambda: vae.timepicker.modified_energy_ratio(y)),
}

for name, time in timer_results.items():
    print(f"{name}: {time:0.3f} μs")

plt.figure(figsize=(8, 3), tight_layout=True)
plt.bar(timer_results.keys(), timer_results.values())
plt.ylabel("Time [μs]")
plt.show()
```

Out:

```
Hinkley: 10.045 μs
AIC: 133.962 μs
Energy Ratio: 11.592 μs
Modified Energy Ratio: 15.917 μs
```

Total running time of the script: (0 minutes 3.110 seconds)

4.4 Timepicker batch processing

Following examples shows how to stream transient data row by row, compute timepicker results and save the results to a feature database (trfdb).

```
import os
from shutil import copyfile
from tempfile import gettempdir

import matplotlib.pyplot as plt
import pandas as pd

import vallenae as vae

HERE = os.path.dirname(__file__) if "__file__" in locals() else os.getcwd()
TRADB = os.path.join(HERE, "steel_plate/sample_plain.tradb")
TRFDB = os.path.join(HERE, "steel_plate/sample.trfdb")
TRFDB_TMP = os.path.join(gettempdir(), "sample.trfdb")
```

4.4.1 Open tradb (readonly) and trfdb (readwrite)

```
copyfile(TRFDB, TRFDB_TMP) # copy trfdb, so we don't overwrite it

tradb = vae.io.TraDatabase(TRADB)
trfdb = vae.io.TrfDatabase(TRFDB_TMP, mode="rw") # allow writing
```

4.4.2 Read current trfdb

```
print(trfdb.read())
```

Out:

```
Trf: 0%|          | 0/4 [00:00<?, ?it/s]
Trf: 100%|#####| 4/4 [00:00<00:00, 86480.49it/s]
      FFT_CoG      FFT_FoM      PA  ...  CTP      FI      FR
traï
1    147.705078  134.277344  46.483864  ...  11  222.672058  110.182449
2    144.042969  139.160156  59.450512  ...  35  182.291672   98.019981
3    155.029297  164.794922  33.995209  ...  55  155.191879   95.493233
4    159.912109  139.160156  29.114828  ...  29  181.023727  101.906227

[4 rows x 8 columns]
```

4.4.3 Compute arrival time offsets with different timepickers

To improve localisation, time of arrival estimates using the first threshold crossing can be refined with timepickers. Therefore, arrival time offsets between the first threshold crossings and the timepicker results are computed.

```
def dt_from_timepicker(timepicker_func, tra: vae.io.TraRecord):
    # Index of the first threshold crossing is equal to the pretrigger samples
    index_ref = tra.pretrigger
    # Only analyse signal until peak amplitude
    index_peak = vae.features.peak_amplitude_index(tra.data)
    data = tra.data[:index_peak]
    # Get timepicker result
    _, index_timepicker = timepicker_func(data)
    # Compute offset in µs
    dt_us = (index_timepicker - index_ref) * 1e6 / tra.samplerate
    return dt_us
```

Transient data is streamed from the database row by row using `vallenae.io.TraDatabase.iread`. Only one transient data set is loaded into memory at a time. That makes the streaming interface ideal for batch processing. The timepicker results are saved to the trfdb using `vallenae.io.TrfDatabase.write`.

```
for tra in tradb.iread():
    # Calculate arrival time offsets with different timepickers
    feature_set = vae.io.FeatureRecord(
        tra=tra.traï,
        features={
            "ATO_Hinkley": dt_from_timepicker(vae.timepicker.hinkley, tra),
            "ATO_AIC": dt_from_timepicker(vae.timepicker.aic, tra),
            "ATO_ER": dt_from_timepicker(vae.timepicker.energy_ratio, tra),
```

(continues on next page)

(continued from previous page)

```

        "ATO_MER": dt_from_timepicker(vae.timepicker.modified_energy_ratio, tra),
    }
)
# Save results to trfdb
trfdb.write(feature_set)

```

4.4.4 Read results from trfdb

```
print(trfdb.read().filter(regex="ATO"))
```

Out:

```

Trf:   0%|          | 0/4 [00:00<?, ?it/s]
Trf: 100%|#####| 4/4 [00:00<00:00, 85598.04it/s]
      ATO_Hinkley  ATO_AIC  ATO_ER  ATO_MER
traid
1              30.4      -1.8    -4.0     29.2
2              36.4      -0.8    -1.0     -0.4
3              67.0      -1.8    -3.2     60.4
4              65.8      -1.0    -2.2     64.2

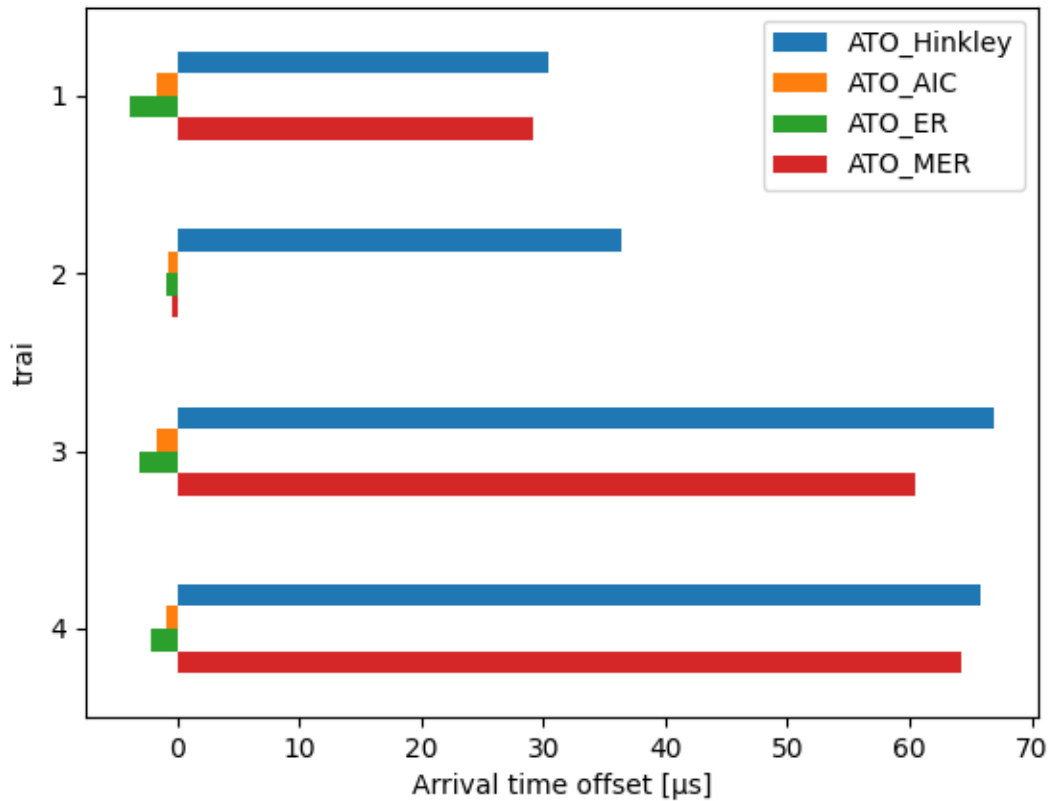
```

4.4.5 Plot results

```

ax = trfdb.read()[["ATO_Hinkley", "ATO_AIC", "ATO_ER", "ATO_MER"]].plot.barh()
ax.invert_yaxis()
ax.set_xlabel("Arrival time offset [µs]")
plt.show()

```



Out:

```
Trf: 0%|          | 0/4 [00:00<?, ?it/s]
Trf: 100%|#####| 4/4 [00:00<00:00, 102927.71it/s]
```

4.4.6 Plot waveforms and arrival times

```
_, axes = plt.subplots(4, 1, tight_layout=True, figsize=(8, 8))
for row, ax in zip(trfdb.read().itertuples(), axes):
    traid = row.Index

    # read waveform from tradb
    y, t = tradb.read_wave(traid)

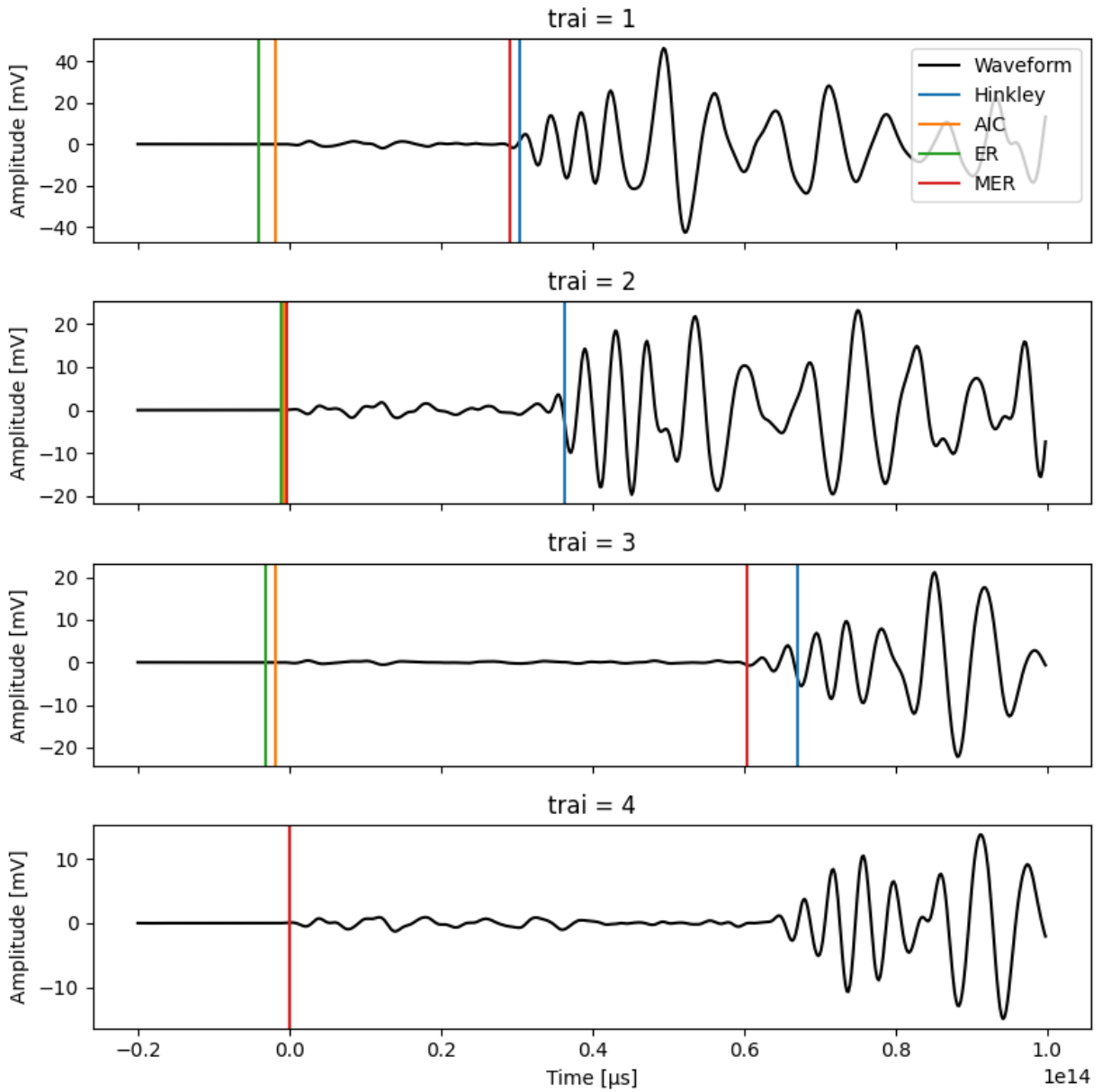
    # plot waveform
    ax.plot(t[400:1000] * 1e6, y[400:1000] * 1e3, "k") # crop and convert to μs/mV
    ax.set_title(f"traid = {traid}")
    ax.set_xlabel("Time [μs]")
    ax.set_ylabel("Amplitude [mV]")
    ax.label_outer()

    # plot arrival time offsets
    ax.axvline(row.ATO_Hinkley, color="C0")
    ax.axvline(row.ATO_AIC, color="C1")
    ax.axvline(row.ATO_ER, color="C2")
```

(continues on next page)

(continued from previous page)

```
ax.axvline(row.ATO_MER, color="C3")
axes[0].legend(["Waveform", "Hinkley", "AIC", "ER", "MER"])
plt.show()
```



Out:

```
Trf: 0%|          | 0/4 [00:00<?, ?it/s]
Trf: 100%|#####| 4/4 [00:00<00:00, 80273.76it/s]
```

4.4.7 Use results in VisualAE

The computed arrival time offsets can be directly used in VisualAE. We only need to specify the unit. VisualAE requires them to be in μs . Units and other column-related meta data is saved in the `trf_fieldinfo` table. Field infos can be retrieved with `vallenae.io.TrfDatabase.fieldinfo`:

```
print(trfdb.fieldinfo())
```

Out:

```
{'FFT_CoG': {'SetTypes': 2, 'Unit': '[kHz]', 'LongName': 'F(C.o.Gravity)',
  ↳ 'Description': 'Center of gravity of spectrum', 'ShortName': None, 'FormatStr': '␣
  ↳ None}, 'FFT_FoM': {'SetTypes': 2, 'Unit': '[kHz]', 'LongName': 'F(max. Amp.)',
  ↳ 'Description': 'Frequency of maximum of spectrum', 'ShortName': None, 'FormatStr': '␣
  ↳ None}, 'PA': {'SetTypes': 8, 'Unit': '[mV]', 'LongName': 'Peak Amplitude',
  ↳ 'Description': None, 'ShortName': None, 'FormatStr': None}, 'RT': {'SetTypes': 8,
  ↳ 'Unit': '[μs]', 'LongName': 'Rise Time', 'Description': None, 'ShortName': None,
  ↳ 'FormatStr': None}, 'Dur': {'SetTypes': 8, 'Unit': '[μs]', 'LongName': 'Duration␣
  ↳ (available)', 'Description': None, 'ShortName': None, 'FormatStr': None}, 'CTP': {
  ↳ 'SetTypes': 8, 'Unit': None, 'LongName': 'Cnts to peak', 'Description': None,
  ↳ 'ShortName': None, 'FormatStr': '#'}, 'FI': {'SetTypes': 8, 'Unit': '[kHz]',
  ↳ 'LongName': 'Initiation Freq.', 'Description': None, 'ShortName': None, 'FormatStr
  ↳ ': None}, 'FR': {'SetTypes': 8, 'Unit': '[kHz]', 'LongName': 'Reverberation Freq.',
  ↳ 'Description': None, 'ShortName': None, 'FormatStr': None}}
```

Show results as table:

```
print(pd.DataFrame(trfdb.fieldinfo()))
```

Out:

	FFT_CoG	...	FR
SetTypes	2	...	8
Unit	[kHz]	...	[kHz]
LongName	F(C.o.Gravity)	...	Reverberation Freq.
Description	Center of gravity of spectrum	...	None
ShortName	None	...	None
FormatStr	None	...	None

[6 rows x 8 columns]

Write units to trfdb

Field infos can be written with `vallenae.io.TrfDatabase.write_fieldinfo`:

```
trfdb.write_fieldinfo("ATO_Hinkley", {"Unit": "[μs]", "LongName": "Arrival Time␣
  ↳ Offset (Hinkley)"})
trfdb.write_fieldinfo("ATO_AIC", {"Unit": "[μs]", "LongName": "Arrival Time Offset␣
  ↳ (AIC)"})
trfdb.write_fieldinfo("ATO_ER", {"Unit": "[μs]", "LongName": "Arrival Time Offset (ER)
  ↳ "})
trfdb.write_fieldinfo("ATO_MER", {"Unit": "[μs]", "LongName": "Arrival Time Offset␣
  ↳ (MER)"})

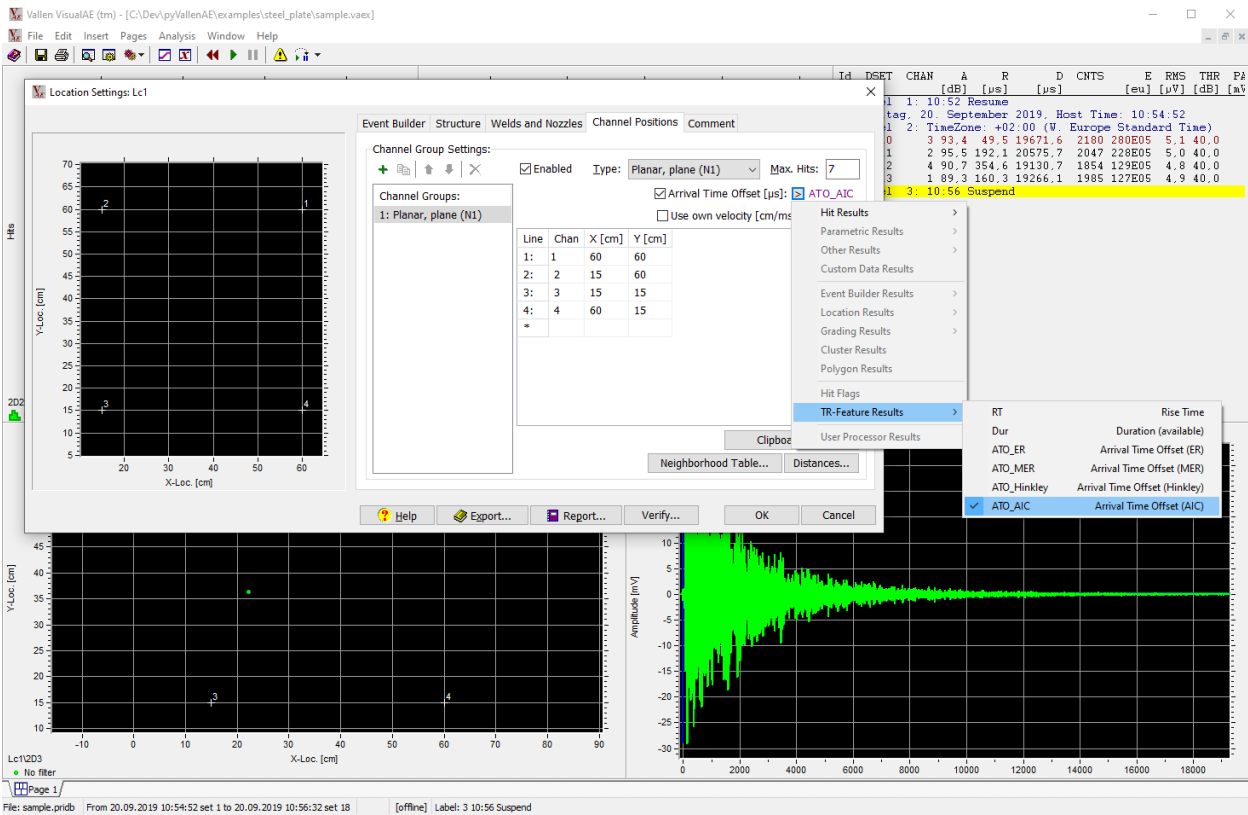
print(pd.DataFrame(trfdb.fieldinfo()).filter(regex="ATO"))
```

Out:

	ATO_Hinkley	...	ATO_MER
SetTypes	None	...	None
Unit	[μs]	...	[μs]
LongName	Arrival Time Offset (Hinkley)	...	Arrival Time Offset (MER)
Description	None	...	None
ShortName	None	...	None
FormatStr	None	...	None
[6 rows x 4 columns]			

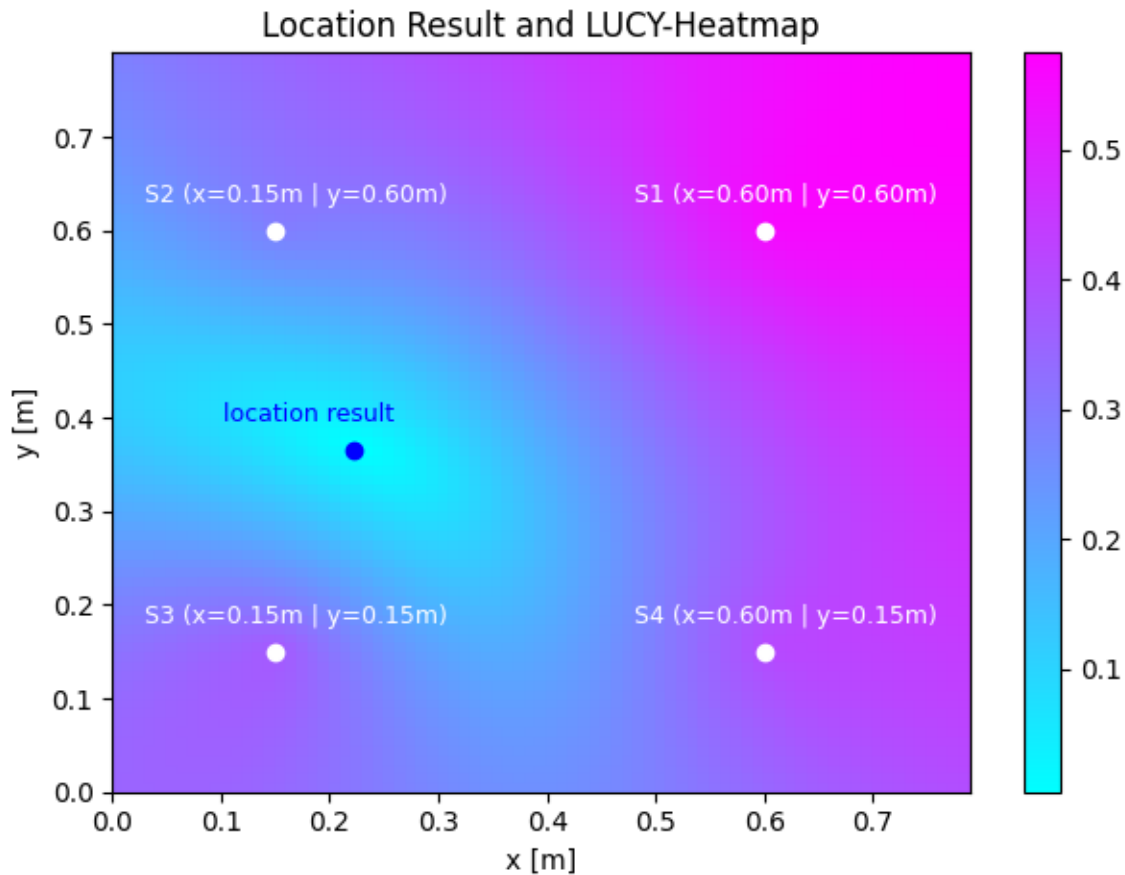
Load results in VisualAE

Time arrival offsets can be specified in the settings of *Location Processors - Channel Positions - Arrival Time Offset*. (Make sure to rename the generated trfdb to match the filename of the prfdb.)



Total running time of the script: (0 minutes 0.889 seconds)

4.5 Localisation



Out:

```
Hits: 0%|          | 0/4 [00:00<?, ?it/s]
Hits: 100%|#####| 4/4 [00:00<00:00, 13400.33it/s]
/home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/checkouts/0.5.1/examples/
→ex5_location.py:122: MatplotlibDeprecationWarning: shading='flat' when X and Y have
→the same dimensions as C is deprecated since 3.3. Either specify the corners of
→the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or
→set rcParams['pcolor.shading']. This will become an error two minor releases later.
plt.pcolormesh(x_grid, y_grid, z_grid, cmap="cool")
Runtime for 1 call to differential_evolution(): 0.608 s
fun: 0.0011158885067467825
jac: array([0.00072356, 0.00151068])
message: 'Optimization terminated successfully.'
nfev: 8169
nit: 101
success: True
x: array([0.22165341, 0.36565977])
```



```

import math
import os
import time
import xml.etree.ElementTree as ElementTree
from typing import Dict, Optional, Tuple

import matplotlib.pyplot as plt
import numpy as np
from numba import f8, njit
from numpy.linalg import norm
from scipy.optimize import differential_evolution

import vallenae as vae

HERE = os.path.dirname(__file__) if "__file__" in locals() else os.getcwd()
SETUP = os.path.join(HERE, "steel_plate/sample.vaex")
PRIDB = os.path.join(HERE, "steel_plate/sample.pridb")
NUMBER_SENSORS = 4

@njit(f8(f8[:, :], f8, f8[:, :]), f8[:, :])
def lucy_error_fun(
    test_pos: np.ndarray,
    speed: float,
    sens_pos: np.ndarray,
    measured_delta_ts: np.ndarray,
) -> float:
    """
    Implementation of the LUCY computation in 2D as documented in
    the Vallen online help.

    Args:
        test_pos: Emitter position to test.
        speed: Assumed speed of sound in a plate-like structure.
        sens_pos: Sensor positions, often a 4x2 array, has to match
            the sorting of the delta-ts.
        measured_delta_ts: The measured time differences in seconds, has to
            match the order of the sensor positions.

    Returns:
        The LUCY value as a float. Ideally 0, in practice never 0, always positive.
    """
    m = len(measured_delta_ts)
    n = m + 1
    measured_delta_dists = speed * measured_delta_ts
    theo_dists = np.zeros(n)
    theo_delta_dists = np.zeros(m)
    for i in range(n):
        theo_dists[i] = norm(test_pos - sens_pos[i, :])
    for i in range(m):
        theo_delta_dists[i] = theo_dists[i + 1] - theo_dists[0]

    # LUCY definition taken from the vallen online help:
    lucy_val = norm(theo_delta_dists - measured_delta_dists) / math.sqrt(n - 1)
    return lucy_val

```

(continues on next page)

(continued from previous page)

```

def get_channel_positions(setup_file: str) -> Dict[int, Tuple[float, float]]:
    tree = ElementTree.parse(setup_file)
    nodes = tree.getroot().findall("./ChannelPos")
    if nodes is None:
        raise RuntimeError("Can not retrieve channel positions from %s", setup_file)
    channel_positions = {
        int(elem.get("Chan")): (float(elem.get("X")), float(elem.get("Y"))) # type: ignore
        for elem in nodes if elem is not None
    }
    return channel_positions

def get_velocity(setup_file: str) -> Optional[float]:
    tree = ElementTree.parse(setup_file)
    node = tree.getroot().find("./Location")
    if node is not None:
        velocity_str = node.get("Velocity")
        if velocity_str is not None:
            return float(velocity_str) * 1e3 # convert to m/s
    raise RuntimeError("Can not retrieve velocity from %s", setup_file)

def main():
    # Consts plotting
    text_delta_y = 0.03
    text_delta_x = -0.12

    # Consts LUCY grid
    grid_delta = 0.01
    location_search_bounds = [(0.0, 0.80), (0.0, 0.80)]

    # Read from prddb
    prddb = vae.io.PriDatabase(PRIDB)
    hits = prddb.read_hits()
    prddb.close()

    channel_order = hits["channel"].to_numpy()
    arrival_times = hits["time"].to_numpy()
    delta_ts = (arrival_times - arrival_times[0])[1:]

    # Get localisation parameters from .vaex file
    velocity = get_velocity(SETUP)
    pos_dict = get_channel_positions(SETUP)

    # Order sensor positions by hit occurrence
    pos_ordered = np.array([pos_dict[ch] for ch in channel_order])

    # Compute heatmap
    lucy_instance_2args = lambda x, y: lucy_error_fun(
        np.array([x, y]), velocity, pos_ordered, delta_ts
    )

    x_range = np.arange(location_search_bounds[0][0], location_search_bounds[0][1],
        grid_delta)
    y_range = x_range
    x_grid, y_grid = np.meshgrid(x_range, y_range)

```

(continues on next page)

(continued from previous page)

```

z_grid = np.vectorize(lucy_instance_2args)(x_grid, y_grid)

# Plot heatmap
plt.figure(tight_layout=True)
plt.pcolormesh(x_grid, y_grid, z_grid, cmap="cool")
plt.colorbar()
plt.title("Location Result and LUCY-Heatmap")
plt.xlabel("x [m]")
plt.ylabel("y [m]")

# Compute location
lucy_instance_single_arg = lambda pos: lucy_error_fun(
    pos, velocity, pos_ordered, delta_ts
)

start = time.perf_counter()
# These are excessive search / overkill parameters:
location_result = differential_evolution(
    lucy_instance_single_arg,
    location_search_bounds,
    popsize=40,
    polish=True,
    strategy="randlbin",
    recombination=0.1,
    mutation=1.3,
)
end = time.perf_counter()
print(f"Runtime for 1 call to differential_evolution(): {(end - start):0.4} s")
print(location_result)

# Plot location result
x_res = location_result.x[0]
y_res = location_result.x[1]
plt.plot([x_res], [y_res], "bo")
plt.text(
    x_res + text_delta_x,
    y_res + text_delta_y,
    "location result",
    fontsize=9,
    color="b",
)

# Plot sensor positions
for channel, (x, y) in pos_dict.items():
    text = f"S{channel} (x={x:0.2f}m | y={y:0.2f}m)"
    plt.scatter(x, y, marker="o", color="w")
    plt.text(x + text_delta_x, y + text_delta_y, text, fontsize=9, color="w")

plt.show()

if __name__ == "__main__":
    main()

```

Total running time of the script: (0 minutes 1.521 seconds)

4.6 Go fast with multiprocessing

The streaming interfaces with iterables allow efficient batch processing as shown [here](#). But still only one core/thread will be utilized. We will change that will multiprocessing.

Following example shows a batch feature extraction procedure using multiple CPU cores.

```
import os
import time
import multiprocessing
from typing import Dict, Iterable
from itertools import cycle
import __main__

import numpy as np
from scipy import stats
import matplotlib.pyplot as plt

import vallenae as vae

HERE = os.path.dirname(__file__) if "__file__" in locals() else os.getcwd()
TRADB = os.path.join(HERE, "steel_plate/sample_plain.tradb")
```

4.6.1 Prepare streaming reads

```
tradb = vae.io.TraDatabase(TRADB)
```

Our sample tradb only contains four data sets. That is not enough data for demonstrating batch processing. Therefore, we will simulate more data by looping over the data sets with following generator/iterable:

```
def tra_generator(loops: int = 1000) -> Iterable[vae.io.TraRecord]:
    for loop, tra in enumerate(cycle(tradb.iread())):
        if loop > loops:
            break
        yield tra
```

4.6.2 Define feature extraction function

Following function will be applied to all data sets and returns computed features:

```
def feature_extraction(tra: vae.io.TraRecord) -> Dict[str, float]:
    # compute random statistical features
    return {
        "Std": np.std(tra.data),
        "Skew": stats.skew(tra.data),
    }

# Fix to use pickle serialization in sphinx gallery
setattr(__main__, feature_extraction.__name__, feature_extraction)
```

4.6.3 Compute with single thread/core

Note: The examples are executed on the CI / readthedocs server with limited resources. Therefore, the shown computation times and speedups are below the capability of modern machines.

Run computation in a single thread and get the time:

```
time_elapsed_ms = lambda t0: 1e3 * (time.perf_counter() - t0)

time_start = time.perf_counter()
for tra in tra_generator():
    results = feature_extraction(tra)
    # do something with the results
time_single_thread = time_elapsed_ms(time_start)

print(f"Time single thread: {time_single_thread:.2f} ms")
```

Out:

```
Time single thread: 773.09 ms
```

4.6.4 Compute with multiple processes/cores

First get number of available cores in your machine:

```
print(f"Available CPU cores: {os.cpu_count()}")
```

Out:

```
Available CPU cores: 2
```

But how can we utilize those cores? The common answer for most programming languages is multithreading. Threads run in the same process and heap, so data can be shared between them (with care). Sadly, Python uses a global interpreter lock (GIL) that locks heap memory, because Python objects are not thread-safe. Therefore, threads are blocking each other and no speedups are gained by using multiple threads.

The solution for Python is multiprocessing to work around the GIL. Every process has its own heap and GIL. Multiprocessing will introduce overhead for interprocess communication and data serialization/deserialization. To reduce the overhead, data is sent in bigger chunks.

Run computation on 4 cores with chunks of 128 data sets and get the time / speedup:

```
with multiprocessing.Pool(4) as pool:
    time_start = time.perf_counter()
    for results in pool.imap(feature_extraction, tra_generator(), chunksize=128):
        pass # do something with the results
    time_multiprocessing = time_elapsed_ms(time_start)

print(f"Time multiprocessing: {time_multiprocessing:.2f} ms")
print(f"Speedup: {(time_single_thread / time_multiprocessing):.2f}")
```

Out:

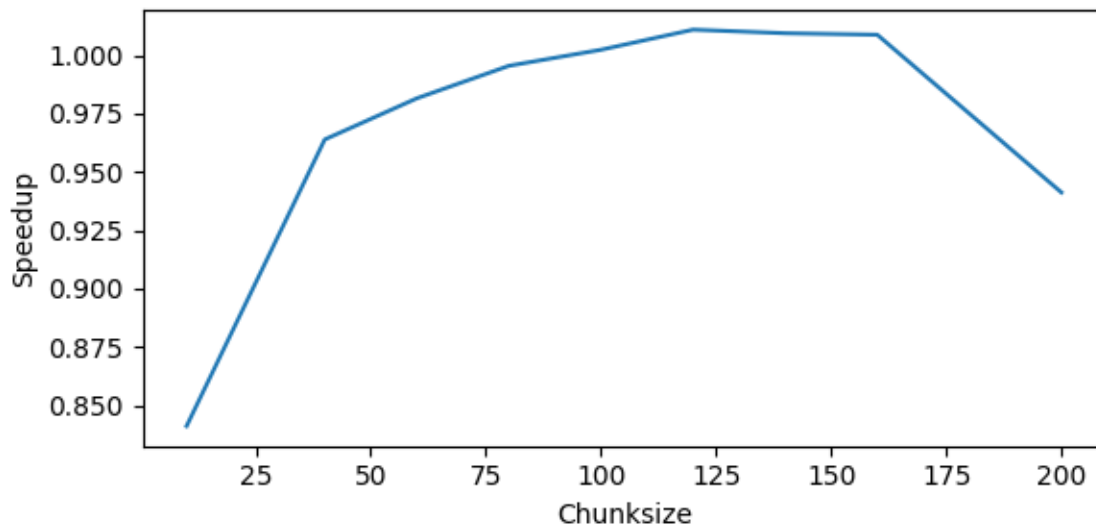
```
Time multiprocessing: 780.51 ms
Speedup: 0.99
```

Variation of the chunksize

Following results show how the chunksize impacts the overall performance. The speedup is measured for different chunksizes and plotted against the chunksize:

```
chunksizes = (10, 40, 60, 80, 100, 120, 140, 160, 200)
speedup_chunksizes = []
with multiprocessing.Pool(4) as pool:
    for chunksize in chunksizes:
        time_start = time.perf_counter()
        for results in pool.imap(feature_extraction, tra_generator(),
    ↪ chunksize=chunksize):
            pass # do something with the results
            speedup_chunksizes.append(time_single_thread / time_elapsed_ms(time_start))

plt.figure(tight_layout=True, figsize=(6, 3))
plt.plot(chunksizes, speedup_chunksizes)
plt.xlabel("Chunksize")
plt.ylabel("Speedup")
plt.show()
```



Total running time of the script: (0 minutes 9.105 seconds)

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

5.1 Unreleased

5.2 0.5.1 - 2020-03-25

5.2.1 Fixed

- Buffering of SQL results in `listen` methods to allow SQL queries in between

5.3 0.5.0 - 2020-03-18

5.3.1 Added

- Query filter parameter to `TrfDatabase.read` and `TrfDatabase.iread`
- `listen` method for `PriDatabase`, `TraDatabase` and `TrfDatabase` to retrieve new records live

5.3.2 Changed

- Order feature records by TRAI for `TrfDatabase.read` and `TrfDatabase.iread`

5.4 0.4.0 - 2021-02-14

5.4.1 Added

- CI with GitHub actions on Linux, MacOS and Windows
- Workflow with GitHub actions to publish to PyPI on new releases
- `pyproject.toml` as the main config file for `pylint`, `pytest`, `tox`, `coverage`, ...

5.4.2 Changed

- Return exact time range with `TraDatabase.read_continuous_wave`
- Return “absolute” time axis with `TraDatabase.read_continuous_wave` (instead of starting at $t = 0$ s)

5.4.3 Fixed

- Fix database close if exception raised in `__init__` (e.g. file not found)
- Example `ex6_multiprocessing` for MacOS
- Find lower/upper bounds for same values (times) in binary search (used by `TraDatabase.iread`)
- Stop condition for `time_stop` in `TraDatabase.iread`
- Use TRAI for `TraDatabase.iread` as a time sorted index for binary search (SetID is not!)
- Check for empty time ranges in `TraDatabase.iread`

5.5 0.3.0 - 2020-11-05

5.5.1 Added

- Query filter for `pridb/tradb` (i)read functions

5.6 0.2.4 - 2020-11-01

5.6.1 Fixed

- SQL schemas for `pridb/tradb/trfdb` creation, add `fieldinfos`

5.7 0.2.3 - 2020-09-01

5.7.1 Fixed

- AIC timepicker
- Add threshold for monotonic time check (1 ns) to ignore rounding issues
- Suppress exception chaining

5.8 0.2.2 - 2020-07-10

5.8.1 Added

- Database classes are now pickable and can be used in multiprocessing
- SQLite transactions for all writes
- Faster blob encoding (`vallenae.io.encode_data_blob`)
- Faster RMS computation with Numba (`vallenae.features.rms`)

5.8.2 Fixed

- Catch possible `global_info` table parsing errors

5.9 0.2.1 - 2020-02-10

5.9.1 Fixed

- Examples outputs if not run as notebook
- Out-of-bound `time_start`, `time_stop` with SQL binary search
- Optional signal strength for e.g. `spotWave` data acquisition

5.10 0.2.0 - 2020-02-06

5.10.1 Added

- Database creation with `mode="rwc"`, e.g. `vallenae.io.PriDatabase.__init__`

5.10.2 Fixed

- Number field in `vallenae.io.MarkerRecord` optional
- Scaling of parametric inputs optional
- Keep column order of query if new columns are added to the database
- Return array with `float32` from `vallenae.io.TraDatabase.read_continuous_wave` (instead of `float64`)

5.11 0.1.0 - 2020-01-24

Initial public release

TODOS

Todo: Status flag

(The [original entry](#) is located in /home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/envs/0.5.1/lib/python3.7/site-packages/vallenae/io/pridb.py:docstring of vallenae.io.pridb.PriDatabase.write_hit, line 10.)

Todo: Status flag

(The [original entry](#) is located in /home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/envs/0.5.1/lib/python3.7/site-packages/vallenae/io/pridb.py:docstring of vallenae.io.pridb.PriDatabase.write_parametric, line 10.)

Todo: Status flag

(The [original entry](#) is located in /home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/envs/0.5.1/lib/python3.7/site-packages/vallenae/io/pridb.py:docstring of vallenae.io.pridb.PriDatabase.write_status, line 10.)

Todo: Status flag

(The [original entry](#) is located in /home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/envs/0.5.1/lib/python3.7/site-packages/vallenae/io/tradb.py:docstring of vallenae.io.tradb.TraDatabase.write, line 9.)

Todo: Remove RMS

(The [original entry](#) is located in /home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/envs/0.5.1/lib/python3.7/site-packages/vallenae/io/datatypes.py:docstring of vallenae.io.datatypes.TraRecord, line 3.)

Todo: Weak performance, if used with default parameter alpha

(The [original entry](#) is located in /home/docs/checkouts/readthedocs.org/user_builds/pyvallenae/envs/0.5.1/lib/python3.7/site-packages/vallenae/timepicker/timepicker.py:docstring of vallenae.timepicker.timepicker.hinkley, line 22.)

INDICES AND TABLES

- `genindex`
- `modindex`

PYTHON MODULE INDEX

V

`vallenae.features`, [36](#)
`vallenae.io`, [1](#)
`vallenae.timepicker`, [40](#)

Symbols

__init__() (vallenae.io.FeatureRecord method), 34
 __init__() (vallenae.io.HitRecord method), 23
 __init__() (vallenae.io.MarkerRecord method), 25
 __init__() (vallenae.io.ParametricRecord method), 30
 __init__() (vallenae.io.PriDatabase method), 4
 __init__() (vallenae.io.StatusRecord method), 27
 __init__() (vallenae.io.TraDatabase method), 11
 __init__() (vallenae.io.TraRecord method), 33
 __init__() (vallenae.io.TrfDatabase method), 17

A

aic() (in module vallenae.timepicker), 42
 amplitude() (vallenae.io.HitRecord property), 21
 amplitude_to_db() (in module vallenae.features), 40

C

cascade_counts() (vallenae.io.HitRecord property), 21
 cascade_energy() (vallenae.io.HitRecord property), 21
 cascade_hits() (vallenae.io.HitRecord property), 21
 cascade_signal_strength() (vallenae.io.HitRecord property), 21
 channel() (vallenae.io.HitRecord property), 21
 channel() (vallenae.io.PriDatabase method), 5
 channel() (vallenae.io.StatusRecord property), 26
 channel() (vallenae.io.TraDatabase method), 12
 channel() (vallenae.io.TraRecord property), 31
 close() (vallenae.io.PriDatabase method), 5
 close() (vallenae.io.TraDatabase method), 12
 close() (vallenae.io.TrfDatabase method), 17
 columns() (vallenae.io.PriDatabase method), 5
 columns() (vallenae.io.TraDatabase method), 12
 columns() (vallenae.io.TrfDatabase method), 17
 connected() (vallenae.io.PriDatabase property), 3
 connected() (vallenae.io.TraDatabase property), 11
 connected() (vallenae.io.TrfDatabase property), 16
 connection() (vallenae.io.PriDatabase method), 5

connection() (vallenae.io.TraDatabase method), 12
 connection() (vallenae.io.TrfDatabase method), 17
 count() (vallenae.io.FeatureRecord method), 34
 count() (vallenae.io.HitRecord method), 23
 count() (vallenae.io.MarkerRecord method), 25
 count() (vallenae.io.ParametricRecord method), 30
 count() (vallenae.io.StatusRecord method), 27
 count() (vallenae.io.TraRecord method), 33
 counts() (in module vallenae.features), 39
 counts() (vallenae.io.HitRecord property), 22
 create() (vallenae.io.PriDatabase static method), 5
 create() (vallenae.io.TraDatabase static method), 12
 create() (vallenae.io.TrfDatabase static method), 17

D

data() (vallenae.io.MarkerRecord property), 24
 data() (vallenae.io.TraRecord property), 31
 data_format() (vallenae.io.TraRecord property), 32
 db_to_amplitude() (in module vallenae.features), 40
 decode_data_blob() (in module vallenae.io), 35
 duration() (vallenae.io.HitRecord property), 22

E

encode_data_blob() (in module vallenae.io), 35
 energy() (in module vallenae.features), 39
 energy() (vallenae.io.HitRecord property), 22
 energy() (vallenae.io.StatusRecord property), 26
 energy_ratio() (in module vallenae.timepicker), 42

F

FeatureRecord (class in vallenae.io), 33
 features() (vallenae.io.FeatureRecord property), 34
 fieldinfo() (vallenae.io.PriDatabase method), 5
 fieldinfo() (vallenae.io.TraDatabase method), 12
 fieldinfo() (vallenae.io.TrfDatabase method), 17
 filename() (vallenae.io.PriDatabase property), 4
 filename() (vallenae.io.TraDatabase property), 11
 filename() (vallenae.io.TrfDatabase property), 16
 first_threshold_crossing() (in module vallenae.features), 38

from_sql() (*vallenae.io.FeatureRecord* class method), 34
 from_sql() (*vallenae.io.HitRecord* class method), 23
 from_sql() (*vallenae.io.MarkerRecord* class method), 25
 from_sql() (*vallenae.io.ParametricRecord* class method), 30
 from_sql() (*vallenae.io.StatusRecord* class method), 27
 from_sql() (*vallenae.io.TraRecord* class method), 33

G

globalinfo() (*vallenae.io.PriDatabase* method), 6
 globalinfo() (*vallenae.io.TraDatabase* method), 13
 globalinfo() (*vallenae.io.TrfDatabase* method), 18

H

hinkley() (in module *vallenae.timepicker*), 41
 HitRecord (class in *vallenae.io*), 20

I

index() (*vallenae.io.FeatureRecord* method), 35
 index() (*vallenae.io.HitRecord* method), 24
 index() (*vallenae.io.MarkerRecord* method), 25
 index() (*vallenae.io.ParametricRecord* method), 31
 index() (*vallenae.io.StatusRecord* method), 28
 index() (*vallenae.io.TraRecord* method), 33
 iread() (*vallenae.io.TraDatabase* method), 13
 iread() (*vallenae.io.TrfDatabase* method), 18
 iread_hits() (*vallenae.io.PriDatabase* method), 6
 iread_markers() (*vallenae.io.PriDatabase* method), 6
 iread_parametric() (*vallenae.io.PriDatabase* method), 7
 iread_status() (*vallenae.io.PriDatabase* method), 7
 is_above_threshold() (in module *vallenae.features*), 38

L

listen() (*vallenae.io.PriDatabase* method), 7
 listen() (*vallenae.io.TraDatabase* method), 13
 listen() (*vallenae.io.TrfDatabase* method), 18

M

MarkerRecord (class in *vallenae.io*), 24
 modified_energy_ratio() (in module *vallenae.timepicker*), 43
 module
 vallenae.features, 36
 vallenae.io, 1
 vallenae.timepicker, 40

N

number() (*vallenae.io.MarkerRecord* property), 24

P

pa0() (*vallenae.io.ParametricRecord* property), 28
 pa1() (*vallenae.io.ParametricRecord* property), 29
 pa2() (*vallenae.io.ParametricRecord* property), 29
 pa3() (*vallenae.io.ParametricRecord* property), 29
 pa4() (*vallenae.io.ParametricRecord* property), 29
 pa5() (*vallenae.io.ParametricRecord* property), 29
 pa6() (*vallenae.io.ParametricRecord* property), 29
 pa7() (*vallenae.io.ParametricRecord* property), 29
 param_id() (*vallenae.io.HitRecord* property), 22
 param_id() (*vallenae.io.ParametricRecord* property), 29
 param_id() (*vallenae.io.StatusRecord* property), 26
 param_id() (*vallenae.io.TraRecord* property), 32
 ParametricRecord (class in *vallenae.io*), 28
 pcta() (*vallenae.io.ParametricRecord* property), 30
 pcta() (*vallenae.io.ParametricRecord* property), 30
 peak_amplitude() (in module *vallenae.features*), 37
 peak_amplitude_index() (in module *vallenae.features*), 37
 pretrigger() (*vallenae.io.TraRecord* property), 32
 PriDatabase (class in *vallenae.io*), 3

R

read() (*vallenae.io.PriDatabase* method), 8
 read() (*vallenae.io.TraDatabase* method), 14
 read() (*vallenae.io.TrfDatabase* method), 18
 read_continuous_wave() (*vallenae.io.TraDatabase* method), 14
 read_hits() (*vallenae.io.PriDatabase* method), 8
 read_markers() (*vallenae.io.PriDatabase* method), 8
 read_parametric() (*vallenae.io.PriDatabase* method), 8
 read_status() (*vallenae.io.PriDatabase* method), 9
 read_wave() (*vallenae.io.TraDatabase* method), 14
 rise_time() (in module *vallenae.features*), 38
 rise_time() (*vallenae.io.HitRecord* property), 22
 rms() (in module *vallenae.features*), 39
 rms() (*vallenae.io.HitRecord* property), 22
 rms() (*vallenae.io.StatusRecord* property), 26
 rms() (*vallenae.io.TraRecord* property), 32
 rows() (*vallenae.io.PriDatabase* method), 9
 rows() (*vallenae.io.TraDatabase* method), 15
 rows() (*vallenae.io.TrfDatabase* method), 19

S

samplerate() (*vallenae.io.TraRecord* property), 32
 samples() (*vallenae.io.TraRecord* property), 32
 set_id() (*vallenae.io.HitRecord* property), 22

set_id() (*vallenae.io.MarkerRecord* property), 24
 set_id() (*vallenae.io.ParametricRecord* property), 30
 set_id() (*vallenae.io.StatusRecord* property), 26
 set_type() (*vallenae.io.MarkerRecord* property), 25
 signal_strength() (*in module vallenae.features*),
 39
 signal_strength() (*vallenae.io.HitRecord* prop-
 erty), 22
 signal_strength() (*vallenae.io.StatusRecord*
 property), 27
 StatusRecord (*class in vallenae.io*), 26

T

tables() (*vallenae.io.PriDatabase* method), 9
 tables() (*vallenae.io.TraDatabase* method), 15
 tables() (*vallenae.io.TrfDatabase* method), 19
 threshold() (*vallenae.io.HitRecord* property), 23
 threshold() (*vallenae.io.StatusRecord* property), 27
 threshold() (*vallenae.io.TraRecord* property), 32
 time() (*vallenae.io.HitRecord* property), 23
 time() (*vallenae.io.MarkerRecord* property), 25
 time() (*vallenae.io.ParametricRecord* property), 30
 time() (*vallenae.io.StatusRecord* property), 27
 time() (*vallenae.io.TraRecord* property), 32
 TraDatabase (*class in vallenae.io*), 10
 trai() (*vallenae.io.FeatureRecord* property), 34
 trai() (*vallenae.io.HitRecord* property), 23
 trai() (*vallenae.io.TraRecord* property), 33
 TraRecord (*class in vallenae.io*), 31
 TrfDatabase (*class in vallenae.io*), 16

V

vallenae.features
 module, 36
 vallenae.io
 module, 1
 vallenae.timepicker
 module, 40

W

write() (*vallenae.io.TraDatabase* method), 15
 write() (*vallenae.io.TrfDatabase* method), 19
 write_fieldinfo() (*vallenae.io.PriDatabase*
 method), 9
 write_fieldinfo() (*vallenae.io.TraDatabase*
 method), 15
 write_fieldinfo() (*vallenae.io.TrfDatabase*
 method), 19
 write_hit() (*vallenae.io.PriDatabase* method), 9
 write_marker() (*vallenae.io.PriDatabase* method),
 10
 write_parametric() (*vallenae.io.PriDatabase*
 method), 10