
vallenae

Release 0.9.0

Lukas Berbuer, Daniel Altmann (Vallen Systeme GmbH)

Feb 14, 2024

LIBRARY DOCUMENTATION

1	IO	3
1.1	Database classes	3
1.2	Data types	22
1.3	Compression	38
2	Features	41
2.1	Acoustic Emission	41
2.2	Conversion	44
3	Timerpicker	47
3.1	vallenae.timerpicker.hinkley	47
3.2	vallenae.timerpicker.aic	48
3.3	vallenae.timerpicker.energy_ratio	48
3.4	vallenae.timerpicker.modified_energy_ratio	49
4	Examples	51
4.1	Export to WAV (incremental)	51
4.2	Read prddb	53
4.3	Read and plot transient data	55
4.4	Timerpicker	56
4.5	Timerpicker batch processing	61
4.6	Localisation	68
4.7	Go fast with multiprocessing	72
4.8	Custom feature extraction	75
4.9	Spectrogram	80
4.10	Export to WAV	82
5	Changelog	85
5.1	Unreleased	85
5.2	0.9.0 - 2024-02-14	85
5.3	0.8.0 - 2023-07-13	85
5.4	0.7.0 - 2022-11-10	86
5.5	0.6.0 - 2021-09-02	86
5.6	0.5.4 - 2021-05-25	87
5.7	0.5.3 - 2021-05-04	87
5.8	0.5.2 - 2021-05-04	87
5.9	0.5.1 - 2021-03-25	87
5.10	0.5.0 - 2021-03-18	87
5.11	0.4.0 - 2021-02-14	88
5.12	0.3.0 - 2020-11-05	88

5.13	0.2.4 - 2020-11-01	88
5.14	0.2.3 - 2020-09-01	89
5.15	0.2.2 - 2020-07-10	89
5.16	0.2.1 - 2020-02-10	89
5.17	0.2.0 - 2020-02-06	89
5.18	0.1.0 - 2020-01-24	90
6	Indices and tables	91
	Python Module Index	93
	Index	95

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 priddb, tradb and trfdb database files.

Warning: Writing is still experimental

<i>PriDatabase</i> (filename[, mode])	IO Wrapper for priddb 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 vallengae.io.PriDatabase

class vallengae.io.**PriDatabase**(filename, mode='ro')
IO Wrapper for priddb database file.

Attributes

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

vallengae.io.PriDatabase.connected

property PriDatabase.**connected**: **bool**
Check if connected to SQLite database.

vallenae.io.PriDatabase.filename**property** PriDatabase.filename: **str**

Filename of database.

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

vallena.io.PriDatabase.tables

PriDatabase.tables()

Get table names.

Return type

Set[str]

vallena.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

vallena.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

vallena.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

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

1.1.2 vallenae.io.TraDatabase

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

IO Wrapper for tradb database file.

Attributes

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

vallenae.io.TraDatabase.connected

property TraDatabase.**connected**: **bool**

Check if connected to SQLite database.

vallenae.io.TraDatabase.filename

property TraDatabase.**filename**: **str**

Filename of database.

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, raw])</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, raw])</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(tri)</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

vallenae.io.TraDatabase.channel

`TraDatabase.channel()`

Get list of channels.

Return type

`Set[int]`

vallenae.io.TraDatabase.close**TraDatabase.close()**

Close database connection.

vallenae.io.TraDatabase.columns**TraDatabase.columns()**

Columns of data table.

Return type`Tuple[str, ...]`**vallenae.io.TraDatabase.connection****TraDatabase.connection()**

Get SQLite connection object.

Raises`RuntimeError` – If connection is closed**Return type**`Connection`**vallenae.io.TraDatabase.create****static TraDatabase.create(filename)**

Create empty tradb.

Parameters**filename** (`str`) – Path to new tradb database file**vallenae.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, raw=False*)

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”
- **raw** (`bool`) – Return data as ADC values (int16). Default: *False*

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, raw=False*)

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”
- **raw** (`bool`) – Return data as ADC values (int16). Default: *False*

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, raw=False)`

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*
- **raw** (`bool`) – Return data as ADC values (`int16`). Default: *False*

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*, *, *raw=False*)

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

- **trai** (*int*) – Transient recorder index (unique key between pridb and tradb)
- **time_axis** (*bool*) – Create the correspondig time axis. Default: *True*
- **raw** (*bool*) – Return data as ADC values (int16). Default: *False*

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

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: bool`

Check if connected to SQLite database.

vallenae.io.TrfDatabase.filename

property `TrfDatabase.filename: str`

Filename of database.

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 (traí) 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*:

```
>>> pridb = vae.io.PriDatabase("./examples/steel_plate/sample.pridb")
>>> for hit in pridb.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
```

(continues on next page)

(continued from previous page)

```
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).

<i>HitRecord</i> (time, channel, param_id, ...[, ...])	Hit record in pridb (SetType = 2).
<i>MarkerRecord</i> (time, set_type, data[, number, ...])	Marker record in pridb (SetType = 4, 5, 6).
<i>StatusRecord</i> (time, channel, param_id, ...[, ...])	Status data record in pridb (SetType = 3).
<i>ParametricRecord</i> (time, param_id[, set_id, ...])	Parametric data record in pridb (SetType = 1).
<i>TraRecord</i> (time, channel, param_id, ...[, ...])	Transient data record in tradb.
<i>FeatureRecord</i> (tra, features)	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: int | None = None, threshold: float | None =
    None, rise_time: float | None = None, signal_strength: float | None = None,
    counts: int | None = None, tra: int | None = None, cascade_hits: int | None =
    None, cascade_counts: int | None = None, cascade_energy: int | None = None,
    cascade_signal_strength: int | None = None)
```

Hit record in pridb (SetType = 2).

Attributes

<i>amplitude</i>	Peak amplitude in volts
<i>cascade_counts</i>	Summed counts of hits in the same hit-cascade
<i>cascade_energy</i>	Summed energy of hits in the same hit-cascade
<i>cascade_hits</i>	Total number of hits in the same hit-cascade
<i>cascade_signal_strength</i>	Summed signal strength of hits in the same hit-cascade # noqa
<i>channel</i>	Channel number
<i>counts</i>	Number of positive threshold crossings
<i>duration</i>	Hit duration in seconds
<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>rise_time</i>	Rise time in seconds
<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>trai</i>	Transient recorder index (foreign key between pridb and tradb)

vallenae.io.HitRecord.amplitude

HitRecord.**amplitude**: `float`

Peak amplitude in volts

vallenae.io.HitRecord.cascade_counts

HitRecord.**cascade_counts**: `Optional[int]`

Summed counts of hits in the same hit-cascade

vallenae.io.HitRecord.cascade_energy

HitRecord.**cascade_energy**: `Optional[int]`

Summed energy of hits in the same hit-cascade

vallenae.io.HitRecord.cascade_hits

HitRecord.**cascade_hits**: `Optional[int]`

Total number of hits in the same hit-cascade

vallena.io.HitRecord.cascade_signal_strength

HitRecord.cascade_signal_strength: `Optional[int]`
Summed signal strength of hits in the same hit-cascade # noqa

vallena.io.HitRecord.channel

HitRecord.channel: `int`
Channel number

vallena.io.HitRecord.counts

HitRecord.counts: `Optional[int]`
Number of positive threshold crossings

vallena.io.HitRecord.duration

HitRecord.duration: `float`
Hit duration in seconds

vallena.io.HitRecord.energy

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

vallena.io.HitRecord.param_id

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

vallena.io.HitRecord.rise_time

HitRecord.rise_time: `Optional[float]`
Rise time in seconds

vallena.io.HitRecord.rms

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

vallenae.io.HitRecord.set_id**HitRecord.set_id:** `Optional[int]`

Unique identifier for data set in pridb

vallenae.io.HitRecord.signal_strength**HitRecord.signal_strength:** `Optional[float]`

Signal strength in nVs (1e-9 Vs)

vallenae.io.HitRecord.threshold**HitRecord.threshold:** `Optional[float]`

Threshold amplitude in volts

vallenae.io.HitRecord.time**HitRecord.time:** `float`

Time in seconds

vallenae.io.HitRecord.trai**HitRecord.trai:** `Optional[int]`

Transient recorder index (foreign key between pridb and tradb)

Methods

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

vallenae.io.HitRecord.__init__**HitRecord.__init__()**

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*: *int* | *None* = *None*, *set_id*: *int* | *None* = *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**MarkerRecord.data:** `str`

Content of marker (label text or datetime)

vallenae.io.MarkerRecord.number**MarkerRecord.number:** `Optional[int]`

Marker number

vallenae.io.MarkerRecord.set_id**MarkerRecord.set_id:** `Optional[int]`

Unique identifier for data set in pridb

vallenae.io.MarkerRecord.set_type**MarkerRecord.set_type:** `int`

Marker type (see above)

vallenae.io.MarkerRecord.time**MarkerRecord.time:** `float`

Time in seconds

Methods

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

vallenae.io.MarkerRecord.__init__**MarkerRecord.__init__()**

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: int |  
    None = None, threshold: float | None = None, signal_strength: float | None  
    = None)
```

Status data record in pridb (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 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

vallenae.io.StatusRecord.channel

StatusRecord.**channel**: **int**

Channel number

vallenae.io.StatusRecord.energy

StatusRecord.**energy**: **float**

Energy (EN 1330-9) in eu (10^{-14} V²s)

vallenae.io.StatusRecord.param_id

StatusRecord.**param_id**: **int**

Parameter ID of table ae_params for ADC value conversion

vallenae.io.StatusRecord.rms

StatusRecord.**rms**: **float**

RMS in volts

vallenae.io.StatusRecord.set_id

StatusRecord.**set_id**: **Optional[int]**

Unique identifier for data set in pridb

vallenae.io.StatusRecord.signal_strength

StatusRecord.**signal_strength**: **Optional[float]**

Signal strength in nVs (10^{-9} Vs)

vallenae.io.StatusRecord.threshold

StatusRecord.**threshold**: **Optional[float]**

Threshold amplitude in volts

vallenae.io.StatusRecord.time

StatusRecord.**time**: **float**

Time in seconds

Methods

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

vallenae.io.StatusRecord.__init__

`StatusRecord.__init__()`

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: int | None = None, pcta: int | None = None, pa0: int | None = None, pa1: int | None = None, pa2: int | None = None, pa3: int | None = None, pa4: int | None = None, pa5: int | None = None, pa6: int | None = None, pa7: int | None = None)
```

Parametric data record in pridb (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

ParametricRecord.**pa0**: **Optional**[int]
Amplitude of parametric input 0 in volts

vallenae.io.ParametricRecord.pa1

ParametricRecord.**pa1**: **Optional**[int]
Amplitude of parametric input 1 in volts

vallenae.io.ParametricRecord.pa2

ParametricRecord.**pa2**: **Optional**[int]
Amplitude of parametric input 2 in volts

vallenae.io.ParametricRecord.pa3

ParametricRecord.**pa3**: **Optional**[int]
Amplitude of parametric input 3 in volts

vallenae.io.ParametricRecord.pa4

ParametricRecord.**pa4**: **Optional**[int]
Amplitude of parametric input 4 in volts

vallenae.io.ParametricRecord.pa5

ParametricRecord.pa5: `Optional[int]`
Amplitude of parametric input 5 in volts

vallenae.io.ParametricRecord.pa6

ParametricRecord.pa6: `Optional[int]`
Amplitude of parametric input 6 in volts

vallenae.io.ParametricRecord.pa7

ParametricRecord.pa7: `Optional[int]`
Amplitude of parametric input 7 in volts

vallenae.io.ParametricRecord.param_id

ParametricRecord.param_id: `int`
Parameter ID of table ae_params for ADC value conversion

vallenae.io.ParametricRecord.pcta

ParametricRecord.pcta: `Optional[int]`
Analog hysteresis counter

vallenae.io.ParametricRecord.pctd

ParametricRecord.pctd: `Optional[int]`
Digital counter value

vallenae.io.ParametricRecord.set_id

ParametricRecord.set_id: `Optional[int]`
Unique identifier for data set in pridb

vallenae.io.ParametricRecord.time

ParametricRecord.time: `float`
Time in seconds

Methods

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

vallenae.io.ParametricRecord.__init__

`ParametricRecord.__init__()`

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: ndarray, trai: int | None = None, rms: float | None = None, raw: bool = False)`

Transient data record in tradb.

Attributes

<i>channel</i>	Channel number
<i>data</i>	Transient signal in volts or ADC values if <i>raw</i> = <i>True</i>
<i>param_id</i>	Parameter ID of table tr_params for ADC value conversion
<i>pretrigger</i>	Pretrigger samples
<i>raw</i>	<i>data</i> is stored as ADC values (int16)
<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 prddb and tradb)

vallenae.io.TraRecord.channel

TraRecord.**channel**: **int**

Channel number

vallenae.io.TraRecord.data

TraRecord.**data**: **ndarray**

Transient signal in volts or ADC values if *raw* = *True*

vallenae.io.TraRecord.param_id

TraRecord.**param_id**: **int**

Parameter ID of table tr_params for ADC value conversion

vallenae.io.TraRecord.pretrigger

TraRecord.**pretrigger**: **int**

Pretrigger samples

vallenae.io.TraRecord.raw

TraRecord.**raw**: **bool**

data is stored as ADC values (int16)

vallenae.io.TraRecord.rms**TraRecord.rms:** `Optional[float]`

RMS of the noise before the hit

vallenae.io.TraRecord.samplerate**TraRecord.samplerate:** `int`

Samplerate in Hz

vallenae.io.TraRecord.samples**TraRecord.samples:** `int`

Number of samples

vallenae.io.TraRecord.threshold**TraRecord.threshold:** `float`

Threshold amplitude in volts

vallenae.io.TraRecord.time**TraRecord.time:** `float`

Time in seconds

vallenae.io.TraRecord.trai**TraRecord.trai:** `Optional[int]`

Transient recorder index (foreign key between pridb and tradb)

Methods

<code>__init__()</code>	
<code>count(value, /)</code>	Return number of occurrences of value.
<code>from_sql(row, *[, raw])</code>	Create <i>TraRecord</i> from SQL row.
<code>index(value[, start, stop])</code>	Return first index of value.

vallenae.io.TraRecord.__init__

TraRecord.__init__()

vallenae.io.TraRecord.count

TraRecord.count(*value*, /)

Return number of occurrences of value.

vallenae.io.TraRecord.from_sql

classmethod TraRecord.from_sql(*row*, *, *raw=False*)

Create *TraRecord* from SQL row.

Parameters

- **row** (*Dict[str, Any]*) – Dict of column names and values
- **raw** (*bool*) – Provide *data* as ADC values (int16)

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(*trai: int*, *features: Dict[str, float]*)

Transient feature record in trfdb.

Attributes

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

vallenae.io.FeatureRecord.features

FeatureRecord.features: *Dict[str, float]*

Feature dictionary (feature name -> value)

vallenae.io.FeatureRecord.traiFeatureRecord.**tra**i: **int**

Transient recorder index

Methods

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

vallenae.io.FeatureRecord.__init__FeatureRecord.**__init__**()**vallenae.io.FeatureRecord.count**FeatureRecord.**count**(*value*, /)

Return number of occurrences of value.

vallenae.io.FeatureRecord.from_sql**classmethod** FeatureRecord.**from_sql**(*row*)Create *FeatureRecord* from SQL row.**Parameters****row** (**Dict**[**str**, **Any**]) – Dict of column names and values**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(data_blob, data_format, ...)</code>	Decodes (compressed) 16-bit ADC values from BLOB to array of voltage values.
<code>encode_data_blob(data, data_format, ..., raw)</code>	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, *, raw=False)`

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'
- **raw** (`bool`) – Return data as ADC values (`np.int16`), *factor_millivolts* will be ignored

Return type

`ndarray`

Returns

Array of voltage values or ADC values if *raw* is *True*

1.3.2 vallenae.io.encode_data_blob

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

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'
- **raw** (`bool`) – Provide *data* as ADC values (`int16`), *factor_millivolts* will be ignored

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 `vallena.features.peak_amplitude`

`vallena.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 `vallena.features.peak_amplitude_index`

`vallena.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 $1e-14 \text{ V}^2\text{s}$.

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 ($1e-9 \text{ Vs}$).

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)

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 Export to WAV (incremental)

Generate WAV files from tradb. We use the `vallенаe.io.TraDatabase.read_continuous_wave` method to read the transient data as a continuous array.

This example reads and writes the data incrementally to handle big file sizes that don't fit into memory at once.

```
Export channel 1 to /tmp/tmpixwlv5gd/bearing_plain_ch1.wav
Generated WAV files:
/tmp/tmpixwlv5gd/bearing_plain_ch1.wav
```

```
from pathlib import Path
from tempfile import TemporaryDirectory
from typing import Optional

import vallенаe as vae
from soundfile import SoundFile

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()

def export_wav_incremental(
    filename_wav: Path,
    tradb: vae.io.TraDatabase,
    channel: int,
    time_start: Optional[float] = None,
    time_stop: Optional[float] = None,
    time_block: int = 1,
):
    """
    Export data from tradb to a WAV file.
```

(continues on next page)

(continued from previous page)

```

Args:
    filename_wav: Path to the generated WAV file
    tradb: `TraDatabase` instance
    channel: Channel number
    time_start: Start reading at relative time (in seconds). Start at beginning if None
    time_stop: Stop reading at relative time (in seconds). Read until end if None
    time_block: Block size in seconds
"""
con = tradb.connection()
if time_start is None:
    time_start = 0
if time_stop is None:
    time_stop = con.execute("SELECT MAX(Time) FROM view_tr_data").fetchone()[0]

samplerates = con.execute("SELECT DISTINCT(SampleRate) FROM tr_data").fetchone()
assert len(samplerates) == 1
samplerate = samplerates[0]

blocks = int((time_stop - time_start) // time_block + 1)
with SoundFile(filename_wav, "w", samplerate=samplerate, channels=1, subtype="PCM_16") as f:
    for block in range(blocks):
        time_block_start = time_start + block * time_block
        time_block_stop = min(time_start + (block + 1) * time_block, time_stop)
        y, _ = tradb.read_continuous_wave(
            channel=channel,
            time_start=time_block_start,
            time_stop=time_block_stop,
            time_axis=False,
            show_progress=False,
            raw=True, # read as ADC values (int16)
        )
        f.write(y)
        f.flush()

def main():
    filename_tradb = HERE / "bearing" / "bearing_plain.tradb"
    # use a temporary file for this example
    with TemporaryDirectory() as tmpdir:
        with vae.io.TraDatabase(filename_tradb) as tradb:
            for channel in tradb.channel():
                filename_wav = Path(tmpdir) / f"{filename_tradb.stem}_ch{channel}.wav"
                print(f"Export channel {channel} to {filename_wav}")
                export_wav_incremental(
                    filename_wav=filename_wav,
                    tradb=tradb,
                    channel=channel,
                    time_start=0, # read from t = 0 s
                    time_stop=None, # read until end if None
                    time_block=1, # read/write in block sizes of 1 s

```

(continues on next page)

(continued from previous page)

```

    )

    # list all generated wav files
    print("Generated WAV files:")
    for file in Path(tmpdir).iterdir():
        print(file)

if __name__ == "__main__":
    main()

```

Total running time of the script: (0 minutes 0.064 seconds)

4.2 Read prddb

```

from pathlib import Path

import matplotlib.pyplot as plt
import vallenae as vae

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
PRIDB = HERE / "steel_plate" / "sample.pridb"

```

4.2.1 Open prddb

```

prddb = vae.io.PriDatabase(PRIDB)

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

```

```

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

```

4.2.2 Read hits to Pandas DataFrame

```

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

```

```

Hits:   0%|          | 0/4 [00:00<?, ?it/s]
Hits: 100%|| 4/4 [00:00<00:00, 7509.94it/s]
      time  channel  amplitude  counts      energy
set_id

```

(continues on next page)

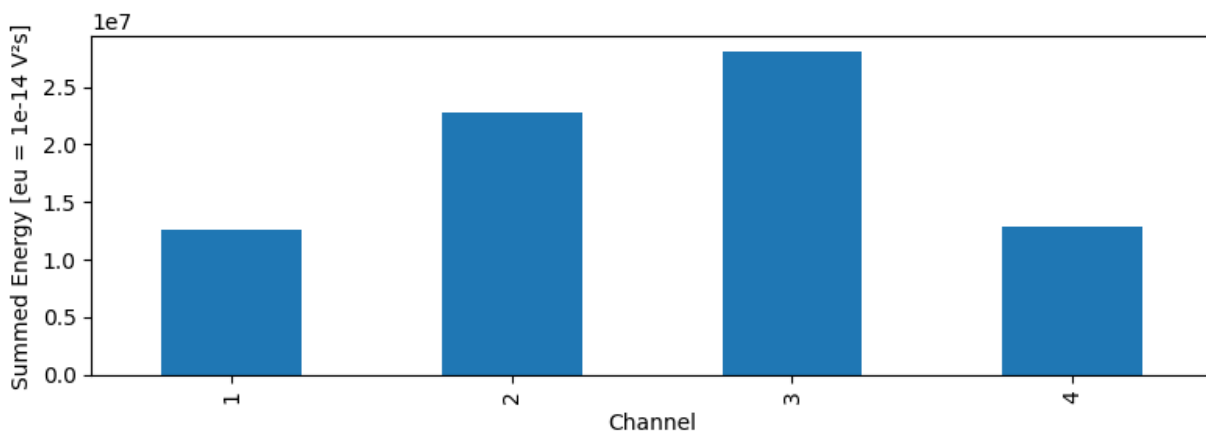
(continued from previous page)

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

4.2.3 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()
```



4.2.4 Read markers

```
df_markers = pridb.read_markers()
print(df_markers)
```

```
Marker: 0%|          | 0/5 [00:00<?, ?it/s]
Marker: 100%|| 5/5 [00:00<00:00, 14324.81it/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
```

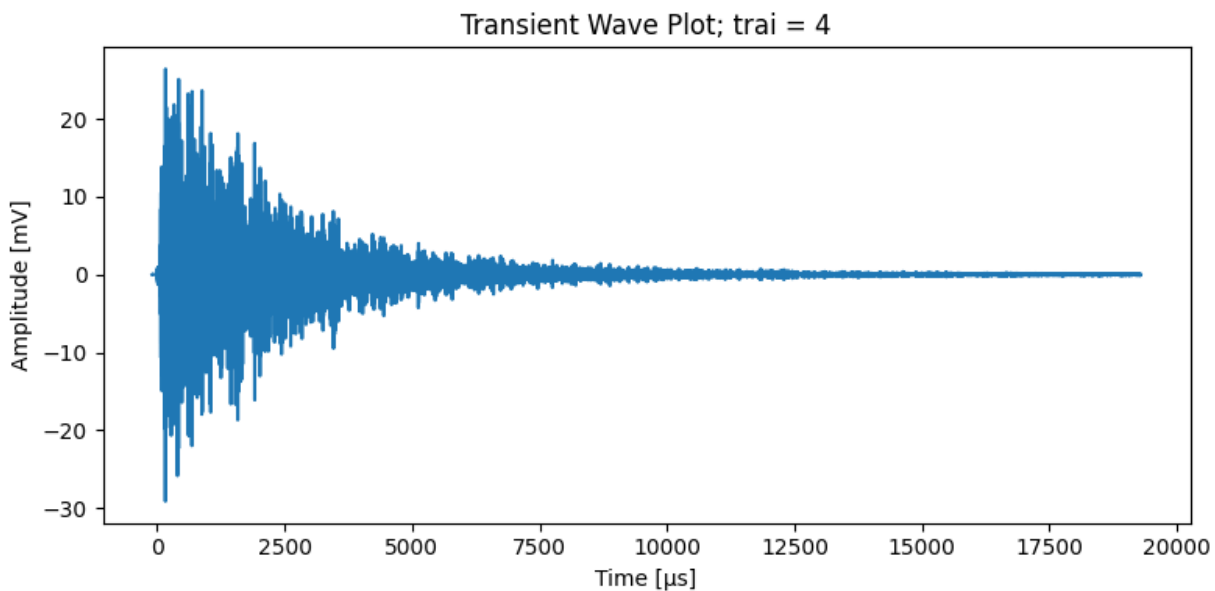
4.2.5 Read parametric data

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

```
Parametric:  0%|          | 0/9 [00:00<?, ?it/s]
Parametric: 100%|| 9/9 [00:00<00:00, 13657.29it/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 0.219 seconds)

4.3 Read and plot transient data



```
from pathlib import Path

import matplotlib.pyplot as plt
import vallenae as vae

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
TRADB = HERE / "steel_plate" / "sample_plain.tradb" # uncompressed
```

(continues on next page)

(continued from previous page)

```

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 0.266 seconds)

4.4 Timepicker

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

```

import time
from pathlib import Path

import matplotlib.pyplot as plt
import vallenae as vae

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
TRADB = HERE / "steel_plate" / "sample_plain.tradb"
TRAI = 4
SAMPLES = 2000

```

4.4.1 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
```

4.4.2 Prepare plotting with time-picker results

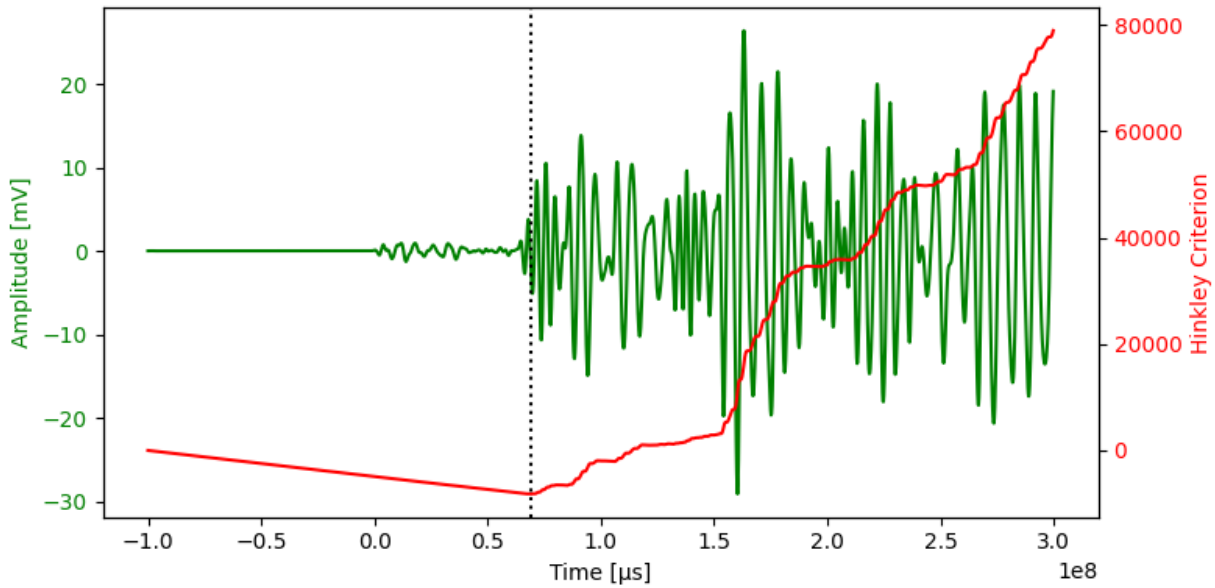
```
def plot(t_wave, y_wave, y_picker, index_picker, name_picker):
    _, ax1 = plt.subplots(figsize=(8, 4), tight_layout=True)
    ax1.set_xlabel("Time [ $\mu$ s]")
    ax1.set_ylabel("Amplitude [mV]", color="g")
    ax1.plot(t_wave, y_wave, color="g")
    ax1.tick_params(axis="y", labelcolor="g")

    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()
```

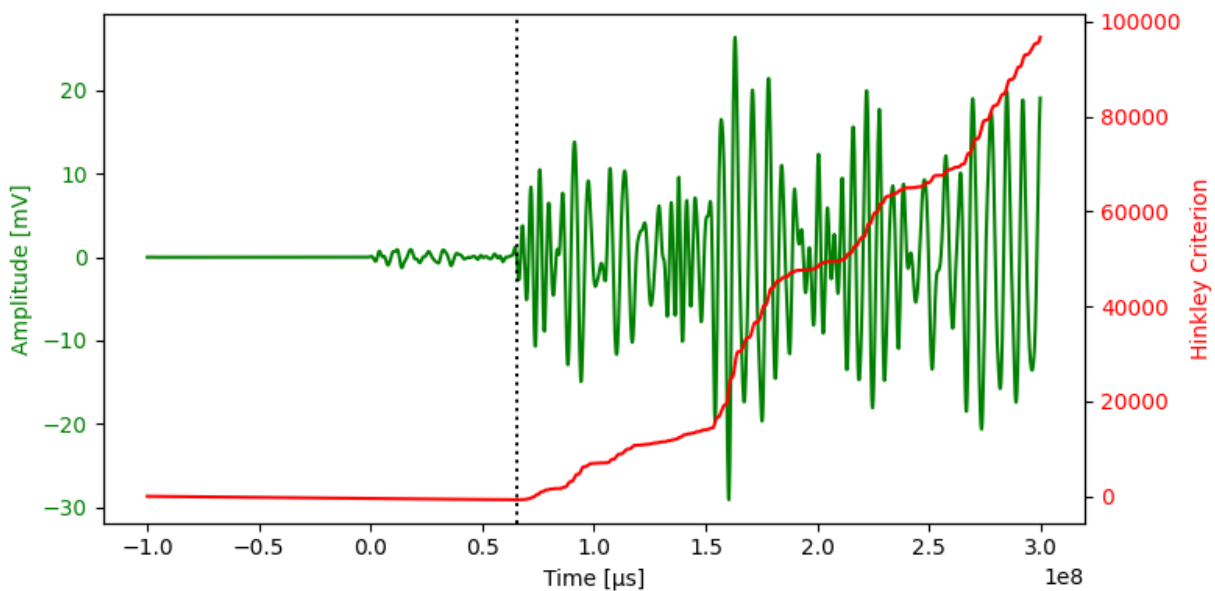
4.4.3 Hinkley Criterion

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



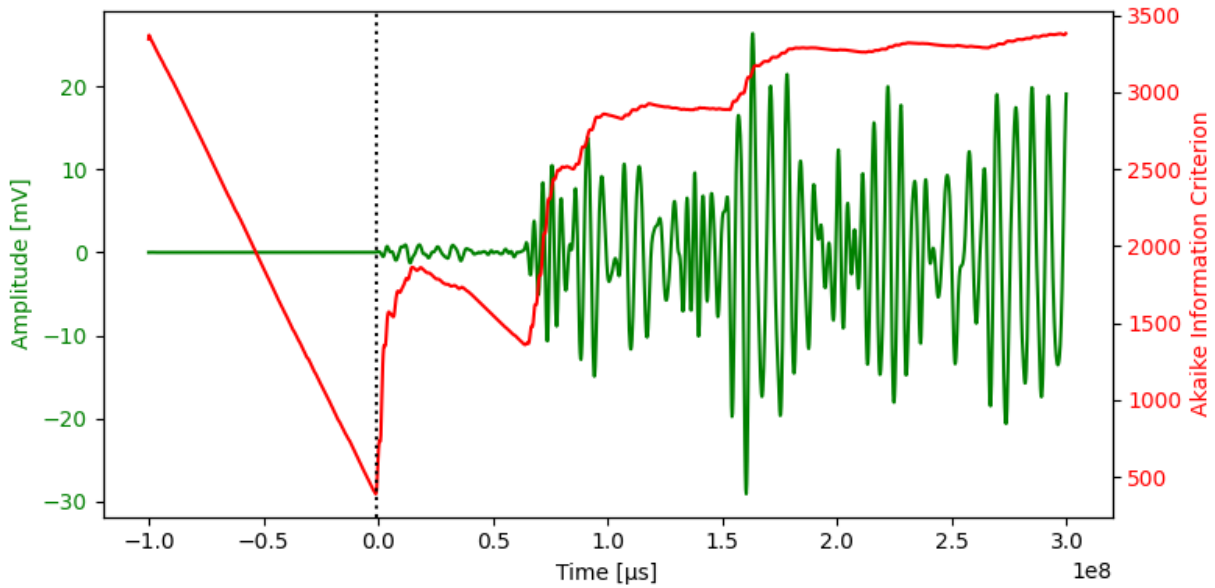
The negative trend correlates to the chosen alpha value and can influence the results strongly. Results with **alpha = 50** (less negative trend):

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



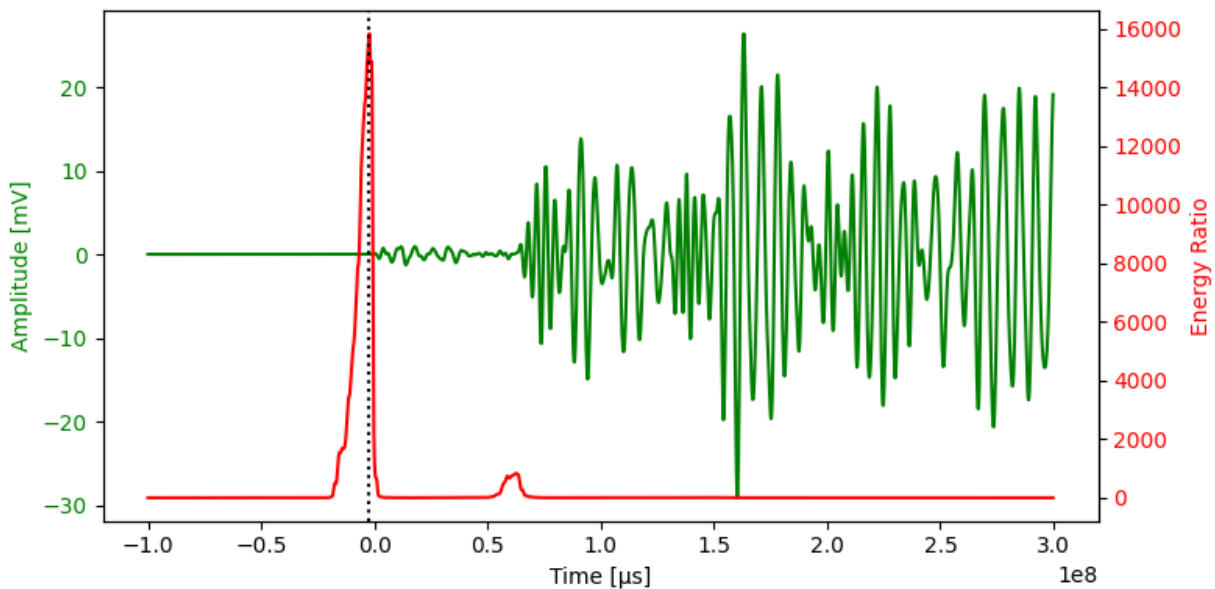
4.4.4 Akaike Information Criterion (AIC)

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



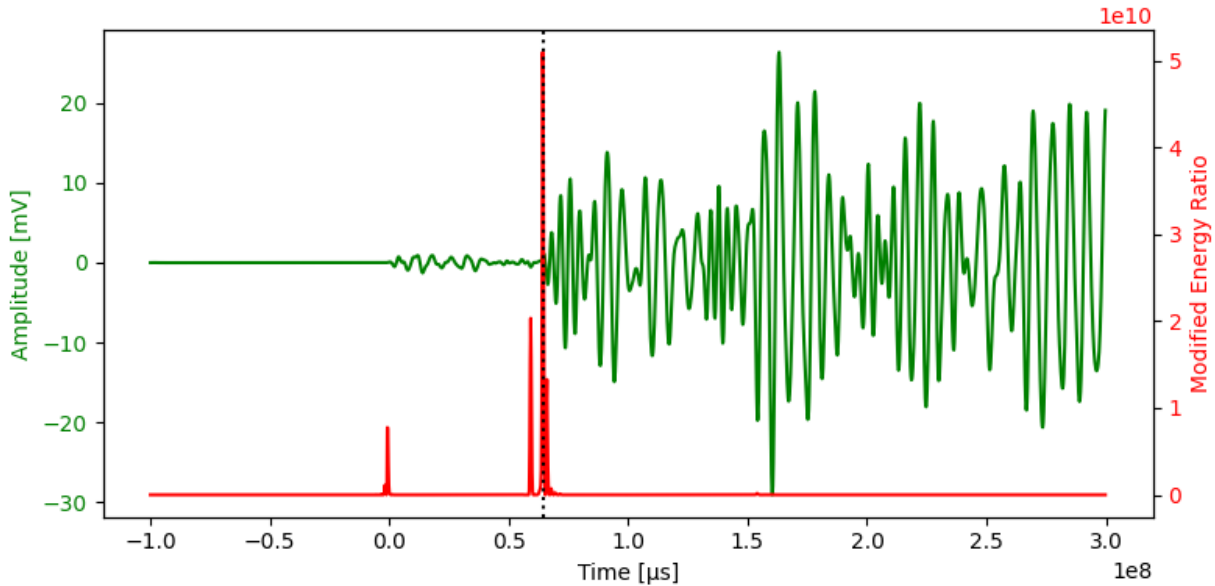
4.4.5 Energy Ratio

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



4.4.6 Modified Energy Ratio

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



4.4.7 Performance comparison

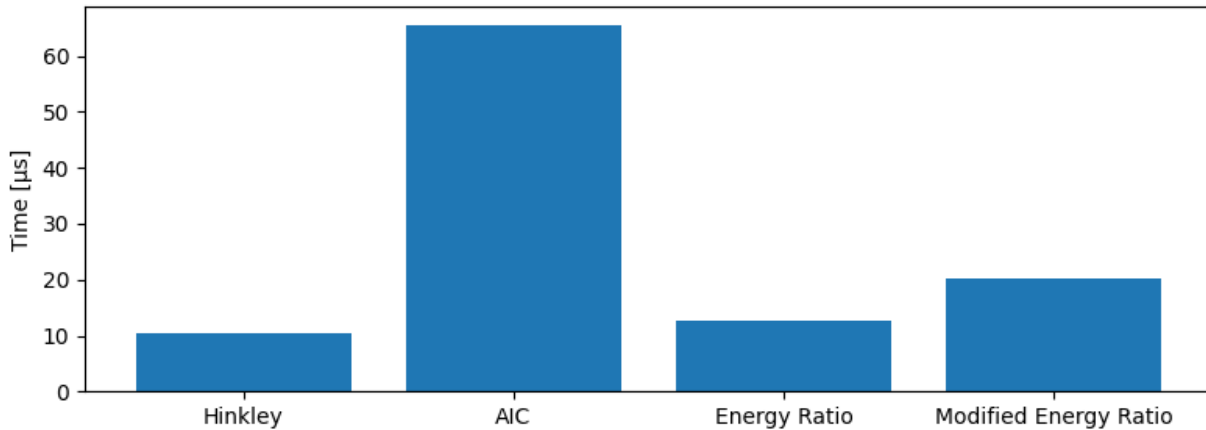
All timepicker implementations are using Numba for just-in-time (JIT) compilations. Usually the first function call is slow, because it will trigger the JIT compiler. To compare the performance to a native or numpy implementation, the average of multiple executions should be compared.

```
def timeit(func, loops=100):
    time_start = time.perf_counter()
    for _ in range(loops):
        func()
    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, execution_time in timer_results.items():
    print(f"{name}: {execution_time:0.3f} μs")

plt.figure(figsize=(8, 3), tight_layout=True)
plt.bar(timer_results.keys(), timer_results.values()) # noqa: SIM911
plt.ylabel("Time [μs]")
plt.show()
```

```
Hinkley: 10.544 μs
AIC: 65.591 μs
Energy Ratio: 12.616 μs
Modified Energy Ratio: 20.098 μs
```

Total running time of the script: (0 minutes 3.200 seconds)

4.5 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).

```
from pathlib import Path
from shutil import copyfile
from tempfile import gettempdir

import matplotlib.pyplot as plt
import pandas as pd
import vallenae as vae

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
TRADB = HERE / "steel_plate" / "sample_plain.tradb"
TRFDB = HERE / "steel_plate" / "sample.trfdb"
TRFDB_TMP = Path(gettempdir()) / "sample.trfdb"
```

4.5.1 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
```

4.5.2 Read current trfdb

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

```
Trf: 0%|          | 0/4 [00:00<?, ?it/s]
Trf: 100%| 4/4 [00:00<00:00, 26462.49it/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]
```

4.5.3 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
    return (index_timepicker - index_ref) * 1e6 / tra.samplerate
```

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

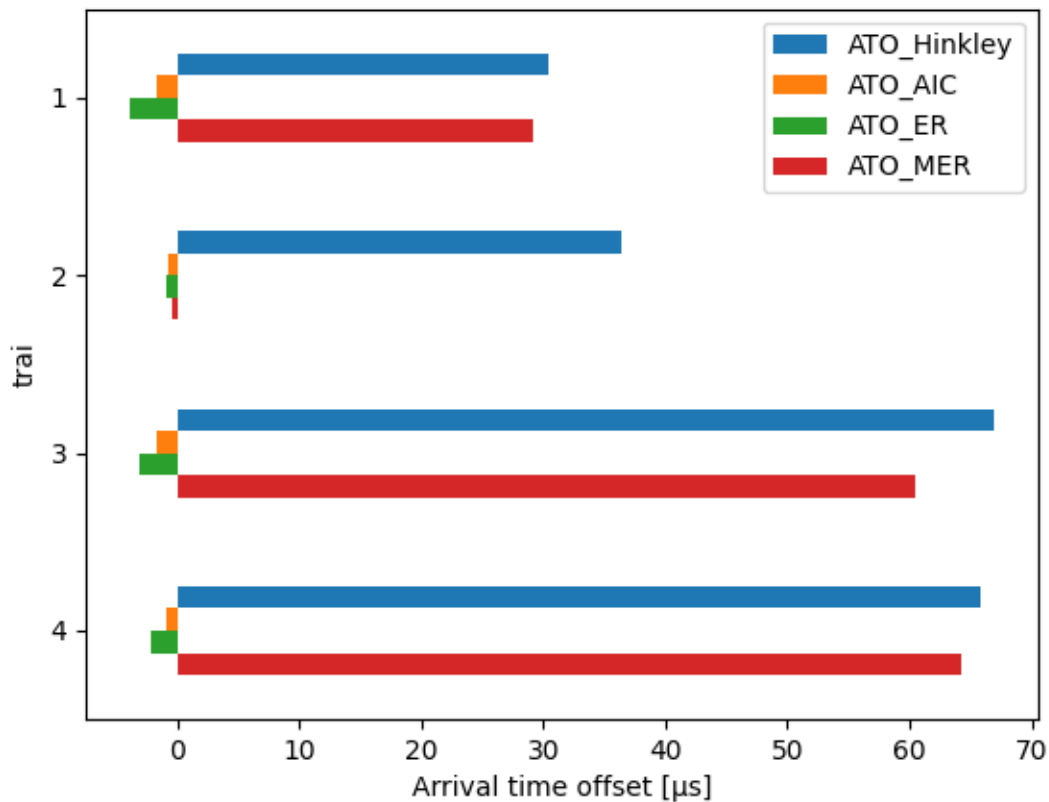
4.5.4 Read results from trfdb

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

```
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
traï
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
```

4.5.5 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()
```



```
Trf:  0%|          | 0/4 [00:00<?, ?it/s]
Trf: 100%| 4/4 [00:00<00:00, 94254.02it/s]
```

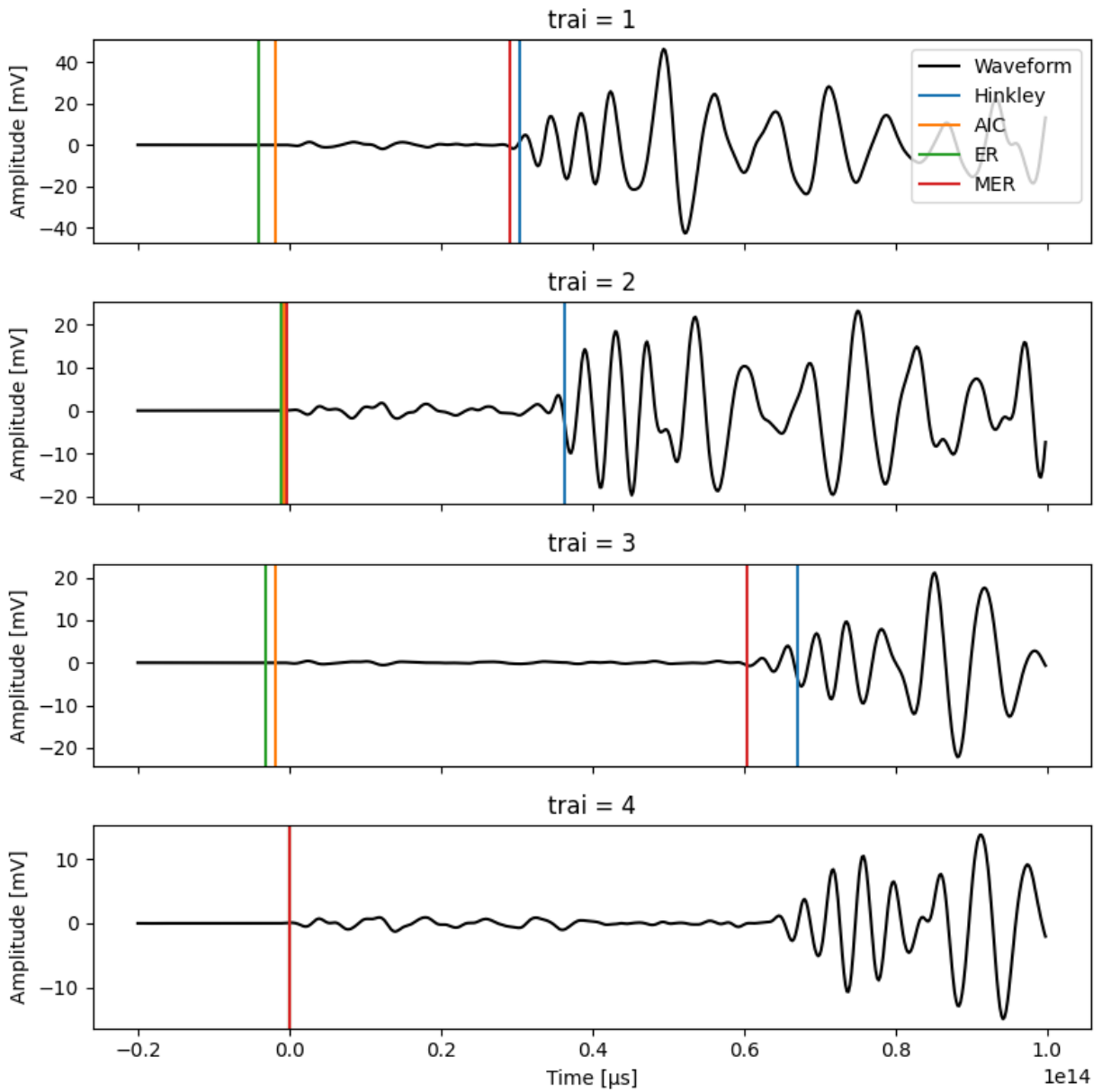
4.5.6 Plot waveforms and arrival times

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

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

    # plot waveform
    ax.plot(t[400:1000] * 1e6, y[400:1000] * 1e3, "k") # crop and convert to  $\mu$ s/mV
    ax.set_title(f"trai = {trai}")
    ax.set_xlabel("Time [ $\mu$ s]")
    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")
    ax.axvline(row.ATO_MER, color="C3")

axes[0].legend(["Waveform", "Hinkley", "AIC", "ER", "MER"])
plt.show()
```



Trf: 0%| | 0/4 [00:00<?, ?it/s]
 Trf: 100%| 4/4 [00:00<00:00, 68478.43it/s]

4.5.7 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())
```

```
{'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()))
```

	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"))
```

	ATO_Hinkley	...	ATO_MER
SetTypes	None	...	None

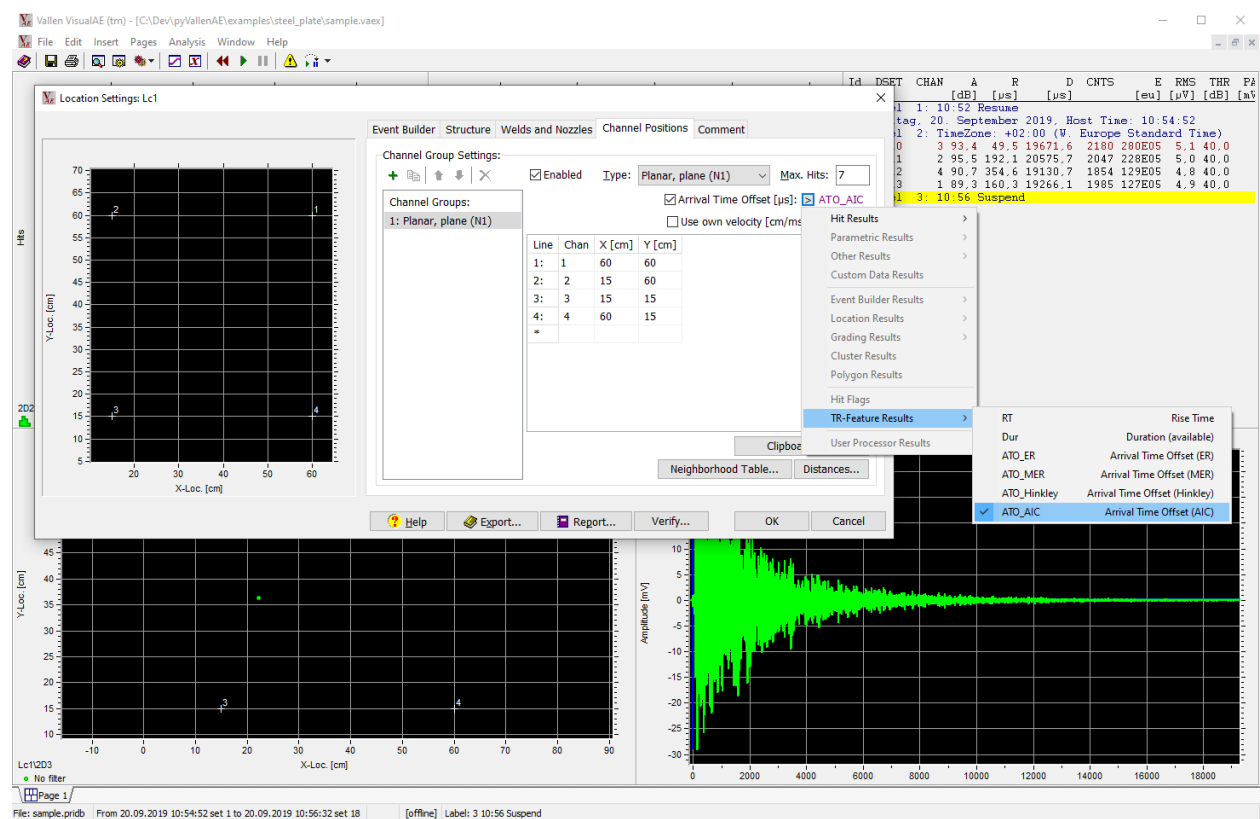
(continues on next page)

(continued from previous page)

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]			

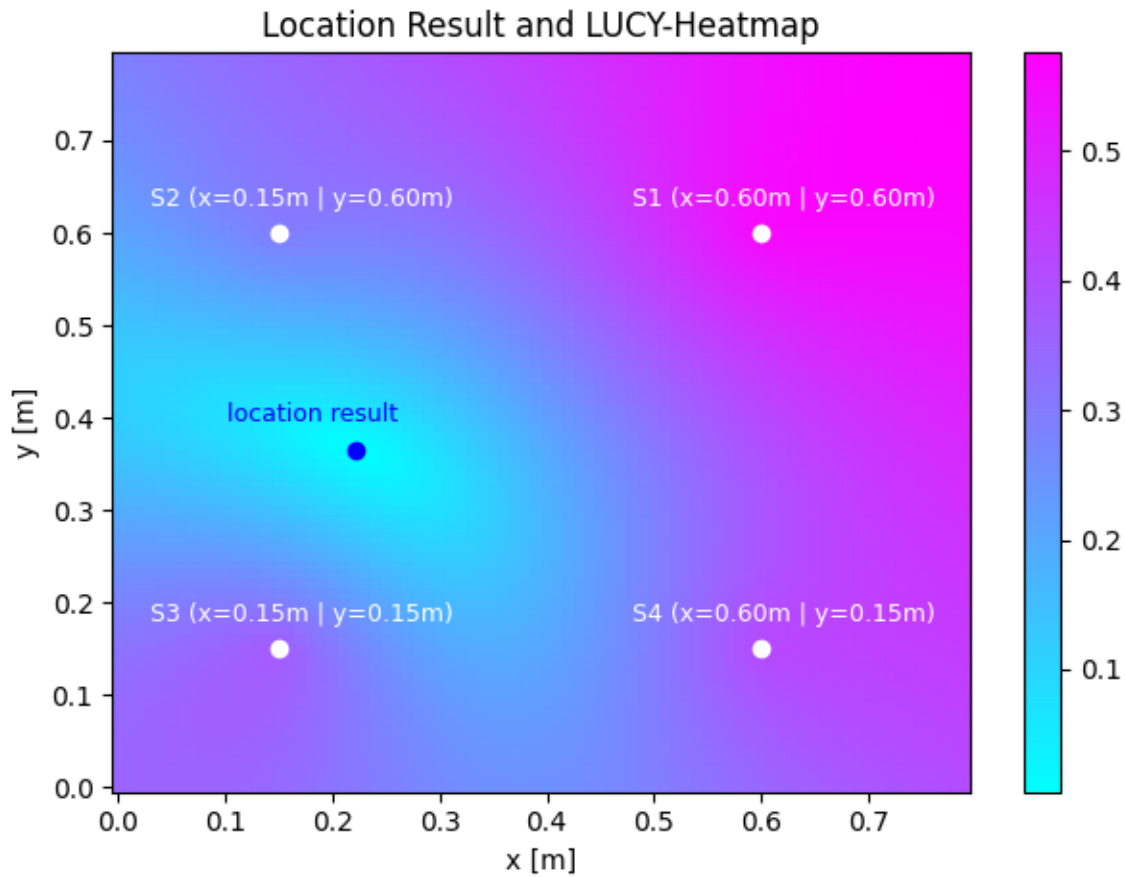
Load results in VisualAE

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 pridb.)



Total running time of the script: (0 minutes 0.832 seconds)

4.6 Localisation



```
Hits: 0%|          | 0/4 [00:00<?, ?it/s]
Hits: 100%| 4/4 [00:00<00:00, 9664.29it/s]
Runtime for 1 call to differential_evolution(): 0.6849 s
      message: Optimization terminated successfully.
      success: True
          fun: 0.001115888406323039
          x: [ 2.217e-01  3.657e-01]
          nit: 102
          nfev: 8249
      population: [[ 2.217e-01  3.657e-01]
                  [ 2.218e-01  3.656e-01]
                  ...
                  [ 2.215e-01  3.657e-01]
                  [ 2.216e-01  3.656e-01]]
      population_energies: [ 1.116e-03  1.125e-03 ... 1.125e-03  1.124e-03]
          jac: [ 6.477e-04  1.395e-03]
```



```

import math
import time
from pathlib import Path
from typing import Dict, Optional, Tuple
from xml.etree import ElementTree

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

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
PRIDB = HERE / "steel_plate" / "sample.pridb"
SETUP = HERE / "steel_plate" / "sample.vaex"
NUMBER_SENSORS = 4

@njit(f8(f8[:], f8, f8[:, :], f8[:]))
def lucy_error_fun(
    test_pos: np.ndarray,
    speed: float,
    sens_poss: 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_poss: 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_poss[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:
    return norm(theo_delta_dists - measured_delta_dists) / math.sqrt(n - 1)

```

(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)
    return {
        int(elem.get("Chan")): (float(elem.get("X")), float(elem.get("Y"))) # type: ignore
        for elem in nodes if elem is not None
    }

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 pridb
    pridb = vae.io.PriDatabase(PRIDB)
    hits = pridb.read_hits()
    pridb.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
    def lucy_instance_2args(x, y):
        return lucy_error_fun(np.array([x, y]), velocity, pos_ordered, delta_ts)

```

(continues on next page)

(continued from previous page)

```

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)
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
def lucy_instance_single_arg(pos):
    return 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="rand1bin",
    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")

```

(continues on next page)

(continued from previous page)

```
plt.show()

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

Total running time of the script: (0 minutes 1.861 seconds)

4.7 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 multiprocessing
import os
import time
from itertools import cycle
from pathlib import Path
from typing import Iterable

import matplotlib.pyplot as plt
import vallenae as vae

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
TRADB = HERE / "steel_plate" / "sample_plain.tradb"
```

4.7.1 Prepare streaming reads

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]:
    with vae.io.TraDatabase(TRADB) as tradb:
        for loop, tra in enumerate(cycle(tradb.iread())):
            if loop > loops:
                break
            yield tra
```

4.7.2 Define feature extraction function

A simple function from the module `_feature_extraction` is applied to all data sets and returns computed features. The function is defined in another module to work with `multiprocessing.Pool`: <https://bugs.python.org/issue25053>

```
from _feature_extraction import feature_extraction # noqa
```

4.7.3 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:

```
def time_elapsed_ms(t0):
    return 1000.0 * (time.perf_counter() - t0)

if __name__ == "__main__": # guard needed for multiprocessing on Windows
    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")
```

```
Time single thread: 1215.43 ms
```

4.7.4 Compute with multiple processes/cores

First get number of available cores in your machine:

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

```
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:

```
if __name__ == "__main__": # guard needed for multiprocessing on Windows
    with multiprocessing.Pool(4) as pool:
        time_start = time.perf_counter()
```

(continues on next page)

(continued from previous page)

```

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}")

```

```

Time multiprocessing: 1190.77 ms
Speedup: 1.02

```

Variation of the chunksize

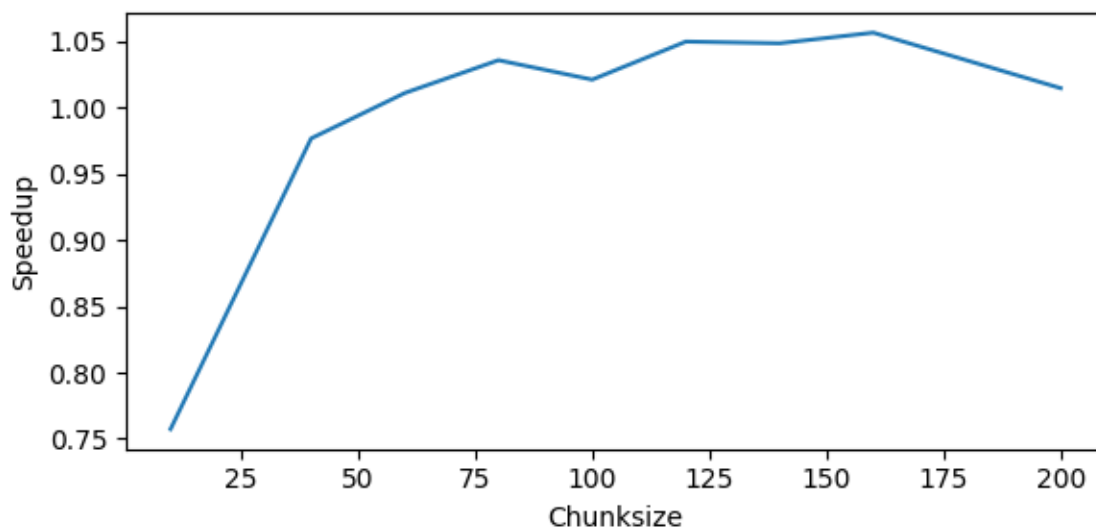
Following results show how the chunksize impacts the overall performance. The speedup is measured for different chunksizes and plotted against the chunksize:

```

if __name__ == "__main__": # guard needed for multiprocessing on Windows
    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 13.960 seconds)

4.8 Custom feature extraction

Following examples shows how to compute custom features and save them in the transient feature database (trfdb) to visualize them in VisualAE.

The feature extraction can be live during acquisition. VisualAE will be notified, that a writer to the trfdb is active and waits for the features to be computed. Therefore, the computed features can be visualized in real time.

```
from pathlib import Path
from tempfile import gettempdir

import matplotlib.pyplot as plt
import numpy as np
import vallenae as vae

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
PRIDB = HERE / "bearing" / "bearing.pridb"
TRADB = HERE / "bearing" / "bearing_plain.tradb"
# TRFDB = HERE / "bearing" / "bearing.trfdb"
TRFDB_TMP = Path(gettempdir()) / "bearing_custom.trfdb" # use a temp file for demo
```

4.8.1 Custom feature extraction algorithms

```
def rms(data: np.ndarray) -> float:
    """Root mean square (RMS)."""
    return np.sqrt(np.mean(data ** 2))

def crest_factor(data: np.ndarray) -> float:
    """Crest factor (ratio of peak amplitude and RMS)."""
    return np.max(np.abs(data)) / rms(data)

def spectral_peak_frequency(spectrum_: np.ndarray, samplerate: int) -> float:
    """
    Peak frequency in a spectrum.

    Args:
        spectrum: FFT amplitudes
        samplerate: Sample rate of the spectrum in Hz

    Returns:
        Peak frequency in Hz
    """
    def bin_to_hz(samplerate: int, samples: int, index: int):
        return 0.5 * samplerate * index / (samples - 1)

    peak_index = np.argmax(spectrum_)
    return bin_to_hz(samplerate, len(spectrum_), peak_index)
```

4.8.2 Open tradb and trfdb

```
tradb = vae.io.TraDatabase(TRADB)
trfdb = vae.io.TrfDatabase(TRFDB_TMP, mode="rwc")
```

Helper function to notify VisualAE, that the transient feature database is active/closed

```
def set_file_status(trfdb_: vae.io.TrfDatabase, status: int):
    """Notify VisualAE that trfdb is active/closed."""
    trfdb_.connection().execute(
        f"UPDATE trf_globalinfo SET Value = {status} WHERE Key == 'FileStatus'"
    )
```

4.8.3 Read tra records, compute features and save to trfdb

The `vallenae.io.TraDatabase.listen` method will read the tradb row by row and can be used during acquisition. Only if the acquisition is closed and no new records are available, the function returns.

```
set_file_status(trfdb, 2) # 2 = active

for tra in tradb.listen(existing=True, wait=False):
    spectrum = np.fft.rfft(tra.data)
    features = vae.io.FeatureRecord(
        trai=tra.trai,
        features={
            "RMS": rms(tra.data),
            "CrestFactor": crest_factor(tra.data),
            "SpectralPeakFreq": spectral_peak_frequency(spectrum, tra.samplerate),
        }
    )
    trfdb.write(features)

set_file_status(trfdb, 0) # 0 = closed
```

Write field infos to trfdb

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

```
trfdb.write_fieldinfo("RMS", {"Unit": "[V]", "LongName": "Root mean square"})
trfdb.write_fieldinfo("CrestFactor", {"Unit": "[]", "LongName": "Crest factor"})
trfdb.write_fieldinfo("SpectralPeakFreq", {"Unit": "[Hz]", "LongName": "Spectral peak_
↪ frequency"})
```


4.8.4 Read results from trfdb

```
df_trfdb = trfdb.read()
print(df_trfdb)
```

```
Trf: 0%|          | 0/490 [00:00<?, ?it/s]
Trf: 100%|| 490/490 [00:00<00:00, 1892457.61it/s]
      RMS  CrestFactor  SpectralPeakFreq
traid
1      0.000129      3.684550      63800.0
2      0.000134      3.843248      65500.0
3      0.000108      3.906766      58500.0
4      0.000115      3.876217      58000.0
5      0.000106      3.548410      72000.0
...      ...      ...      ...
486    0.000006      3.624268      162900.0
487    0.000006      3.486689      250000.0
488    0.000006      3.530853      164100.0
489    0.000006      4.059815      174000.0
490    0.000006      3.545406      164700.0

[490 rows x 3 columns]
```

4.8.5 Plot AE features and custom features

Read pridb and join it with trfdb:

```
with vae.io.PriDatabase(PRIDB) as pridb:
    df_pridb = pridb.read_hits()

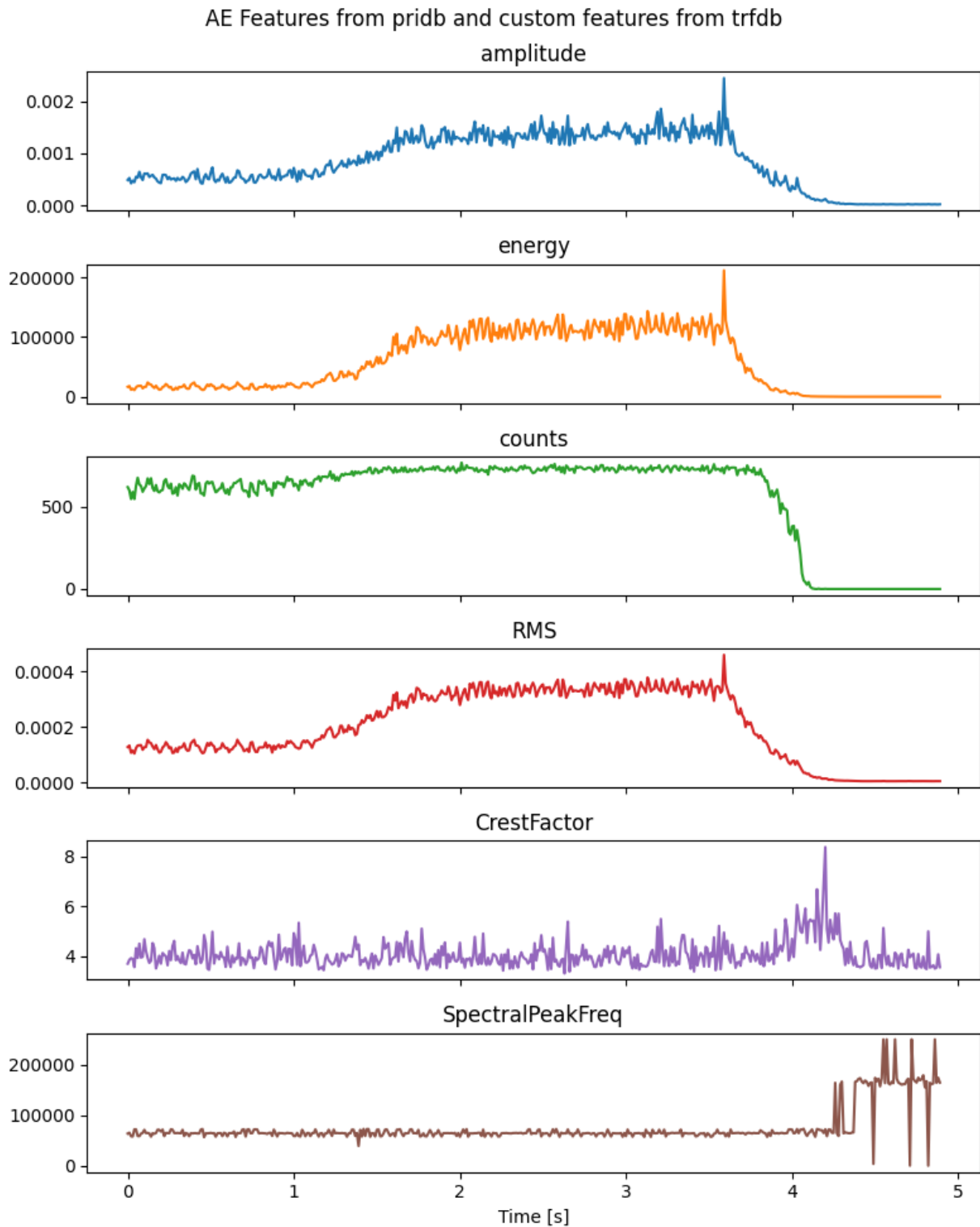
df_combined = df_pridb.join(df_trfdb, on="traid", how="left")
print(df_combined)
```

```
Hits: 0%|          | 0/490 [00:00<?, ?it/s]
Hits: 100%|| 490/490 [00:00<00:00, 105568.57it/s]
      time  channel  param_id  ...      RMS  CrestFactor  SpectralPeakFreq
set_id
6      0.00      1      2 ...  0.000129      3.684550      63800.0
8      0.01      1      2 ...  0.000134      3.843248      65500.0
10     0.02      1      2 ...  0.000108      3.906766      58500.0
12     0.03      1      2 ...  0.000115      3.876217      58000.0
14     0.04      1      2 ...  0.000106      3.548410      72000.0
...      ...      ...      ...      ...      ...      ...
976    4.85      1      2 ...  0.000006      3.624268      162900.0
978    4.86      1      2 ...  0.000006      3.486689      250000.0
980    4.87      1      2 ...  0.000006      3.530853      164100.0
982    4.88      1      2 ...  0.000006      4.059815      174000.0
984    4.89      1      2 ...  0.000006      3.545406      164700.0

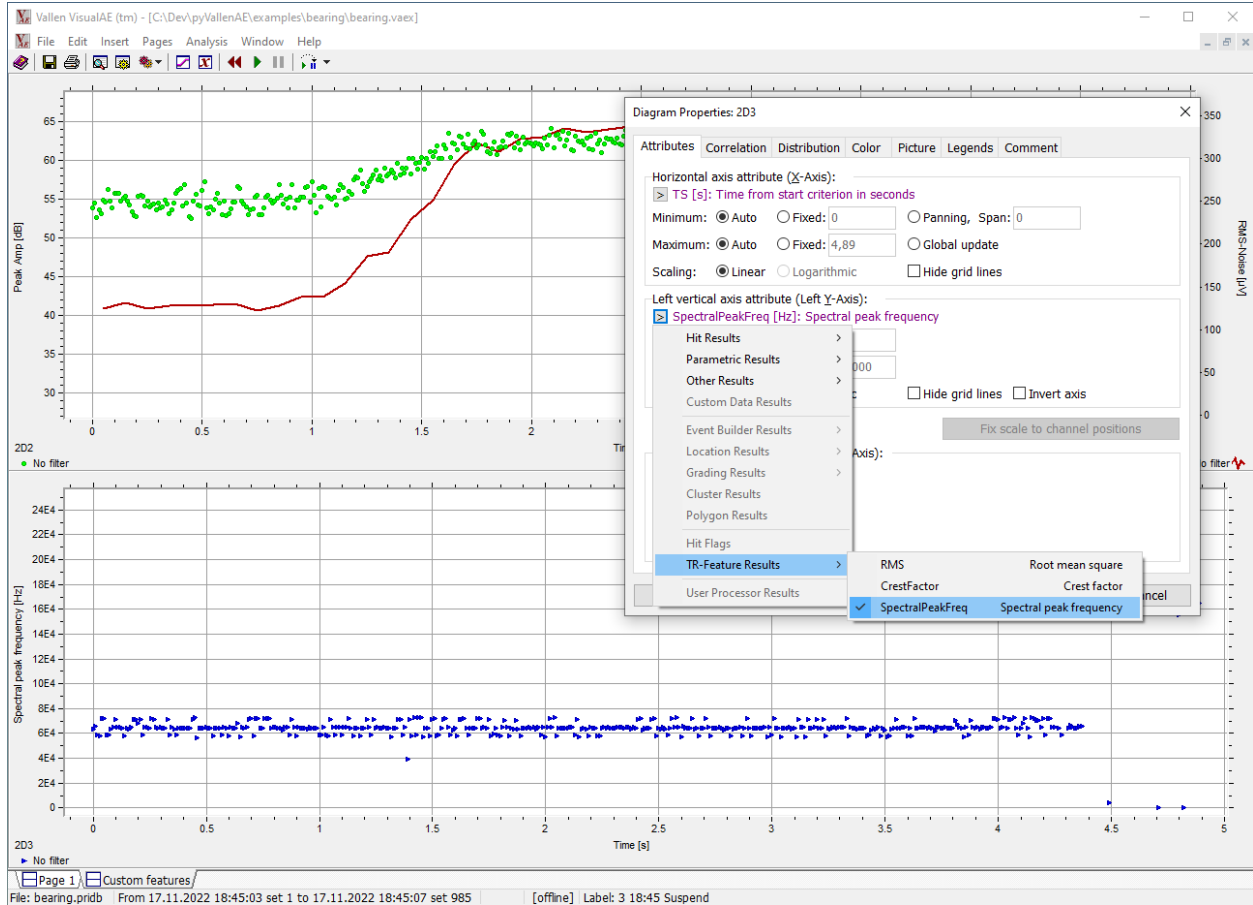
[490 rows x 15 columns]
```

Plot joined features from pridb and trfdb

```
features = [  
    # from pridb  
    "amplitude",  
    "energy",  
    "counts",  
    # from trfdb - custom  
    "RMS",  
    "CrestFactor",  
    "SpectralPeakFreq",  
]  
df_combined.plot(  
    x="time",  
    y=features,  
    xlabel="Time [s]",  
    title=features,  
    legend=False,  
    subplots=True,  
    figsize=(8, 10),  
)  
plt.suptitle("AE Features from pridb and custom features from trfdb")  
plt.tight_layout()  
plt.show()
```



4.8.6 Display custom features in VisualAE



Total running time of the script: (0 minutes 0.922 seconds)

4.9 Spectrogram

Generate spectrogram from tradb. The `vallenae.io.TraDatabase.read_continuous_wave` method is used to read the transient data as a continuous array.

```
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import vallenae as vae
from matplotlib.colors import LogNorm
from scipy import signal

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()
TRADB = HERE / "bearing" / "bearing_plain.tradb"
```

4.9.1 Read transient data as continuous array

The signal is exactly cropped to the given time range (*time_start*, *time_stop*). Time gaps are filled with 0's.

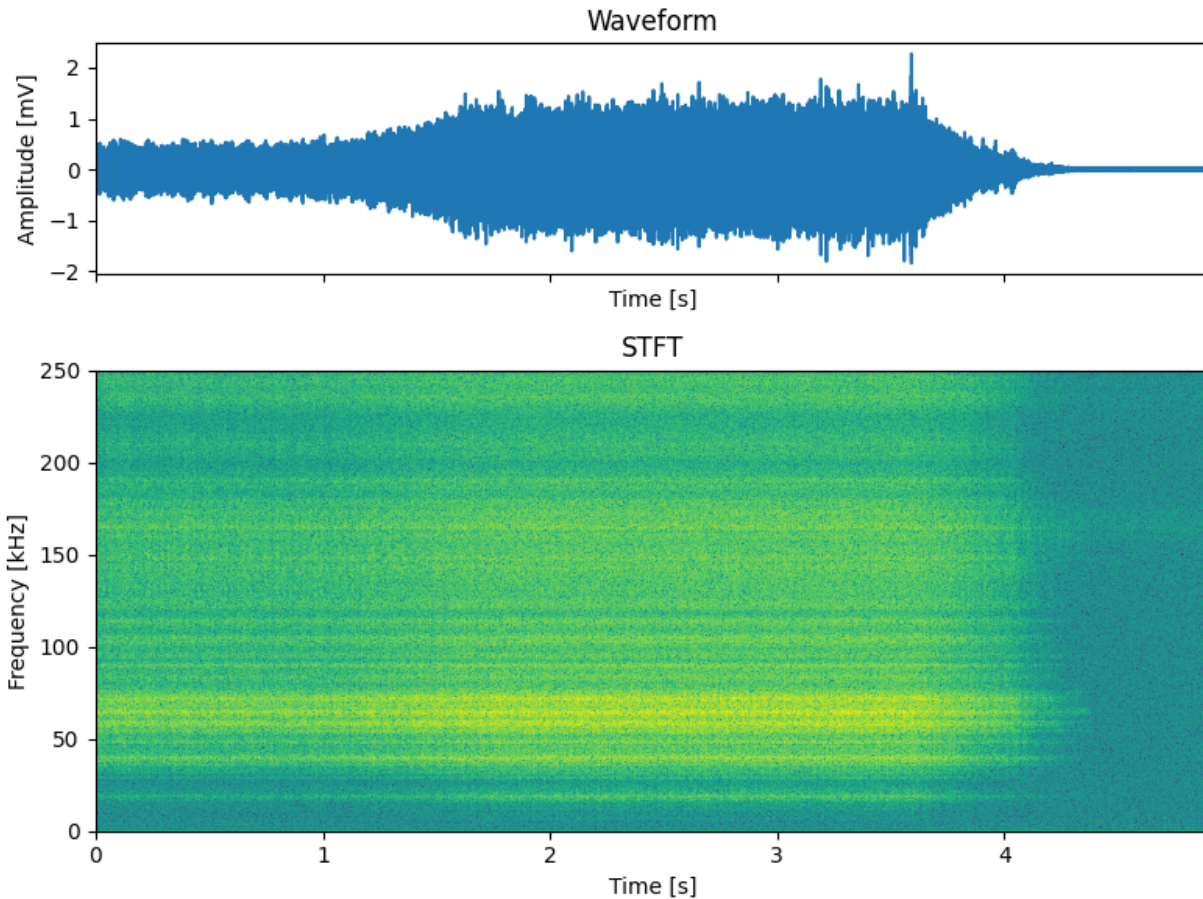
```
with vae.io.TraDatabase(TRADB) as tradb:
    y, fs = tradb.read_continuous_wave(
        channel=1,
        time_start=None,
        time_stop=None,
        time_axis=False, # return samplerate instead of time axis
        show_progress=False,
    )
    t = np.arange(0, len(y)) / fs # generate time axis
```

4.9.2 Compute Short-Time Fourier Transform (STFT)

```
nfft = 4096
noverlap = 2048
fz, tz, zxx = signal.stft(y, fs=fs, window="hann", nperseg=nfft, noverlap=noverlap)
```

4.9.3 Plot time data and spectrogram

```
fig, ax = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(8, 6),
    tight_layout=True,
    gridspec_kw={"height_ratios": (1, 2)},
)
ax[0].plot(t, y * 1e3)
ax[0].set(xlabel="Time [s]", ylabel="Amplitude [mV]", title="Waveform")
ax[1].pcolormesh(tz, fz / 1e3, np.abs(zxx), norm=LogNorm())
ax[1].set(xlabel="Time [s]", ylabel="Frequency [kHz]", title="STFT")
plt.show()
```



Total running time of the script: (0 minutes 2.368 seconds)

4.10 Export to WAV

Generate WAV files from tradb. We use the `vallenae.io.TraDatabase.read_continuous_wave` method to read the transient data as a continuous array.

The signal can optionally be decimated to reduce the size of the generated WAV files (using the `scipy.signal.decimate` function).

```
Export channel 1 to /tmp/tmp96qdcmt/bearing_plain_ch1.wav
Generated WAV files:
/tmp/tmp96qdcmt/bearing_plain_ch1.wav
```

```
from pathlib import Path
from tempfile import TemporaryDirectory
from typing import Optional
```

(continues on next page)

(continued from previous page)

```

import numpy as np
import vallenae as vae
from scipy import signal
from scipy.io import wavfile

HERE = Path(__file__).parent if "__file__" in locals() else Path.cwd()

def export_wav(
    filename_wav: Path,
    tradb: vae.io.TraDatabase,
    channel: int,
    time_start: Optional[float] = None,
    time_stop: Optional[float] = None,
    decimation_factor: int = 1,
):
    """
    Export data from tradb to a WAV file.

    Args:
        filename_wav: Path to the generated WAV file
        tradb: `TraDatabase` instance
        channel: Channel number
        time_start: Start reading at relative time (in seconds). Start at beginning if None
        time_stop: Stop reading at relative time (in seconds). Read until end if None
        decimation_factor: Decimate signal with given factor
    """
    y, fs = tradb.read_continuous_wave(
        channel=channel,
        time_start=time_start,
        time_stop=time_stop,
        time_axis=False,
        show_progress=False,
        raw=True, # read as ADC values (int16)
    )

    if decimation_factor > 1:
        y = signal.decimate(y, decimation_factor).astype(np.int16)
        fs //= decimation_factor

    wavfile.write(filename_wav, fs, y)

def main():
    filename_tradb = HERE / "bearing" / "bearing_plain.tradb"

    # use a temporary file for this example
    with TemporaryDirectory() as tmpdir:
        with vae.io.TraDatabase(filename_tradb) as tradb:
            for channel in tradb.channel():

```

(continues on next page)

(continued from previous page)

```
filename_wav = Path(tmpdir) / f"{filename_tradb.stem}_ch{channel}.wav"
print(f"Export channel {channel} to {filename_wav}")
export_wav(
    filename_wav=filename_wav,
    tradb=tradb,
    channel=channel,
    time_start=0, # read from t = 0 s
    time_stop=None, # read until end if None
    decimation_factor=5, # custom decimation factor
)

# list all generated wav files
print("Generated WAV files:")
for file in Path(tmpdir).iterdir():
    print(file)

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

Total running time of the script: (0 minutes 0.096 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 0.9.0 - 2024-02-14

5.2.1 Added

- Example for incremental WAV export
- Python 3.12 support

5.3 0.8.0 - 2023-07-13

5.3.1 Added

- Flag `TraRecord.raw` if data is stored as ADC values (int16)
- Flag `raw` for `TraDatabase` read methods to read data as ADC values:
 - `TraDatabase.iread`
 - `TraDatabase.read`
 - `TraDatabase.read_wave`
 - `TraDatabase.read_continuous_wave`
 - `TraDatabase.listen`
- Bearing example data
- Spectrogram example
- WAV export example using the new `raw` flag
- CI for Python 3.11

5.3.2 Changed

- Remove scipy dependency (only needed for examples)
- Migrate from setuptools to hatch (replace `setup.py` with `pyproject.toml`)

5.3.3 Fixed

- Multiprocessing example for Windows

5.4 0.7.0 - 2022-11-10

5.4.1 Added

- Example for custom feature extraction
- PyInstaller hook
- CI for Python 3.10

5.4.2 Changed

- Make Numba dependency optional (fallback timepicker implementations with NumPy)

5.4.3 Fixed

- Counts computations (first sample above threshold is not a count)

5.5 0.6.0 - 2021-09-02

5.5.1 Added

- CI for Python 3.9

5.5.2 Changed

- Remove superfluous `data_format` field from `TraRecord` data type

5.6 0.5.4 - 2021-05-25

5.6.1 Fixed

- Limit number of buffered records in `listen` methods
- Time axis rounding errors, e.g. for `TraDatabase.read_wave` with `time_axis=True`

5.7 0.5.3 - 2021-05-04

5.7.1 Fixed

- SQLite URI for absolute linux paths

5.8 0.5.2 - 2021-05-04

5.8.1 Fixed

- SQLite URI for special characters (`#`, `?`)

5.9 0.5.1 - 2021-03-25

5.9.1 Fixed

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

5.10 0.5.0 - 2021-03-18

5.10.1 Added

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

5.10.2 Changed

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

5.11 0.4.0 - 2021-02-14

5.11.1 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, ...

5.11.2 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)

5.11.3 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`

5.12 0.3.0 - 2020-11-05

5.12.1 Added

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

5.13 0.2.4 - 2020-11-01

5.13.1 Fixed

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

5.14 0.2.3 - 2020-09-01

5.14.1 Fixed

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

5.15 0.2.2 - 2020-07-10

5.15.1 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`)

5.15.2 Fixed

- Catch possible `global_info` table parsing errors

5.16 0.2.1 - 2020-02-10

5.16.1 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

5.17 0.2.0 - 2020-02-06

5.17.1 Added

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

5.17.2 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)

5.18 0.1.0 - 2020-01-24

Initial public release

INDICES AND TABLES

- `genindex`
- `modindex`

PYTHON MODULE INDEX

V

`vallенае.features`, 41
`vallенае.io`, 3
`vallенае.timepicker`, 47

Symbols

__init__() (vallenae.io.FeatureRecord method), 37
 __init__() (vallenae.io.HitRecord method), 25
 __init__() (vallenae.io.MarkerRecord method), 27
 __init__() (vallenae.io.ParametricRecord method), 33
 __init__() (vallenae.io.PriDatabase method), 4
 __init__() (vallenae.io.StatusRecord method), 30
 __init__() (vallenae.io.TraDatabase method), 12
 __init__() (vallenae.io.TraRecord method), 36
 __init__() (vallenae.io.TrfDatabase method), 18

A

aic() (in module vallenae.timepicker), 48
 amplitude (vallenae.io.HitRecord attribute), 23
 amplitude_to_db() (in module vallenae.features), 44

C

cascade_counts (vallenae.io.HitRecord attribute), 23
 cascade_energy (vallenae.io.HitRecord attribute), 23
 cascade_hits (vallenae.io.HitRecord attribute), 23
 cascade_signal_strength (vallenae.io.HitRecord attribute), 24
 channel (vallenae.io.HitRecord attribute), 24
 channel (vallenae.io.StatusRecord attribute), 29
 channel (vallenae.io.TraRecord attribute), 34
 channel() (vallenae.io.PriDatabase method), 5
 channel() (vallenae.io.TraDatabase method), 12
 close() (vallenae.io.PriDatabase method), 5
 close() (vallenae.io.TraDatabase method), 13
 close() (vallenae.io.TrfDatabase method), 18
 columns() (vallenae.io.PriDatabase method), 5
 columns() (vallenae.io.TraDatabase method), 13
 columns() (vallenae.io.TrfDatabase method), 18
 connected (vallenae.io.PriDatabase property), 3
 connected (vallenae.io.TraDatabase property), 11
 connected (vallenae.io.TrfDatabase property), 17
 connection() (vallenae.io.PriDatabase method), 5
 connection() (vallenae.io.TraDatabase method), 13
 connection() (vallenae.io.TrfDatabase method), 19
 count() (vallenae.io.FeatureRecord method), 37
 count() (vallenae.io.HitRecord method), 26
 count() (vallenae.io.MarkerRecord method), 28

count() (vallenae.io.ParametricRecord method), 33
 count() (vallenae.io.StatusRecord method), 30
 count() (vallenae.io.TraRecord method), 36
 counts (vallenae.io.HitRecord attribute), 24
 counts() (in module vallenae.features), 43
 create() (vallenae.io.PriDatabase static method), 5
 create() (vallenae.io.TraDatabase static method), 13
 create() (vallenae.io.TrfDatabase static method), 19

D

data (vallenae.io.MarkerRecord attribute), 27
 data (vallenae.io.TraRecord attribute), 34
 db_to_amplitude() (in module vallenae.features), 44
 decode_data_blob() (in module vallenae.io), 38
 duration (vallenae.io.HitRecord attribute), 24

E

encode_data_blob() (in module vallenae.io), 38
 energy (vallenae.io.HitRecord attribute), 24
 energy (vallenae.io.StatusRecord attribute), 29
 energy() (in module vallenae.features), 43
 energy_ratio() (in module vallenae.timepicker), 48

F

FeatureRecord (class in vallenae.io), 36
 features (vallenae.io.FeatureRecord attribute), 36
 fieldinfo() (vallenae.io.PriDatabase method), 5
 fieldinfo() (vallenae.io.TraDatabase method), 13
 fieldinfo() (vallenae.io.TrfDatabase method), 19
 filename (vallenae.io.PriDatabase property), 4
 filename (vallenae.io.TraDatabase property), 11
 filename (vallenae.io.TrfDatabase property), 17
 first_threshold_crossing() (in module vallenae.features), 42
 from_sql() (vallenae.io.FeatureRecord class method), 37
 from_sql() (vallenae.io.HitRecord class method), 26
 from_sql() (vallenae.io.MarkerRecord class method), 28
 from_sql() (vallenae.io.ParametricRecord class method), 33
 from_sql() (vallenae.io.StatusRecord class method), 30

`from_sql()` (*vallenae.io.TraRecord* class method), 36

G

`globalinfo()` (*vallenae.io.PriDatabase* method), 6

`globalinfo()` (*vallenae.io.TraDatabase* method), 14

`globalinfo()` (*vallenae.io.TrfDatabase* method), 19

H

`hinkley()` (in module *vallenae.timepicker*), 47

HitRecord (class in *vallenae.io*), 22

I

`index()` (*vallenae.io.FeatureRecord* method), 37

`index()` (*vallenae.io.HitRecord* method), 26

`index()` (*vallenae.io.MarkerRecord* method), 28

`index()` (*vallenae.io.ParametricRecord* method), 33

`index()` (*vallenae.io.StatusRecord* method), 30

`index()` (*vallenae.io.TraRecord* method), 36

`iread()` (*vallenae.io.TraDatabase* method), 14

`iread()` (*vallenae.io.TrfDatabase* method), 19

`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*), 42

L

`listen()` (*vallenae.io.PriDatabase* method), 8

`listen()` (*vallenae.io.TraDatabase* method), 14

`listen()` (*vallenae.io.TrfDatabase* method), 20

M

MarkerRecord (class in *vallenae.io*), 26

`modified_energy_ratio()` (in module *vallenae.timepicker*), 49

module

vallenae.features, 39

vallenae.io, 1

vallenae.timepicker, 45

N

`number` (*vallenae.io.MarkerRecord* attribute), 27

P

`pa0` (*vallenae.io.ParametricRecord* attribute), 31

`pa1` (*vallenae.io.ParametricRecord* attribute), 31

`pa2` (*vallenae.io.ParametricRecord* attribute), 31

`pa3` (*vallenae.io.ParametricRecord* attribute), 31

`pa4` (*vallenae.io.ParametricRecord* attribute), 31

`pa5` (*vallenae.io.ParametricRecord* attribute), 32

`pa6` (*vallenae.io.ParametricRecord* attribute), 32

`pa7` (*vallenae.io.ParametricRecord* attribute), 32

`param_id` (*vallenae.io.HitRecord* attribute), 24

`param_id` (*vallenae.io.ParametricRecord* attribute), 32

`param_id` (*vallenae.io.StatusRecord* attribute), 29

`param_id` (*vallenae.io.TraRecord* attribute), 34

ParametricRecord (class in *vallenae.io*), 30

`pcta` (*vallenae.io.ParametricRecord* attribute), 32

`pctd` (*vallenae.io.ParametricRecord* attribute), 32

`peak_amplitude()` (in module *vallenae.features*), 41

`peak_amplitude_index()` (in module *vallenae.features*), 41

`pretrigger` (*vallenae.io.TraRecord* attribute), 34

PriDatabase (class in *vallenae.io*), 3

R

`raw` (*vallenae.io.TraRecord* attribute), 34

`read()` (*vallenae.io.PriDatabase* method), 8

`read()` (*vallenae.io.TraDatabase* method), 15

`read()` (*vallenae.io.TrfDatabase* method), 20

`read_continuous_wave()` (*vallenae.io.TraDatabase* method), 15

`read_hits()` (*vallenae.io.PriDatabase* method), 8

`read_markers()` (*vallenae.io.PriDatabase* method), 9

`read_parametric()` (*vallenae.io.PriDatabase* method), 9

`read_status()` (*vallenae.io.PriDatabase* method), 9

`read_wave()` (*vallenae.io.TraDatabase* method), 16

`rise_time` (*vallenae.io.HitRecord* attribute), 24

`rise_time()` (in module *vallenae.features*), 42

`rms` (*vallenae.io.HitRecord* attribute), 24

`rms` (*vallenae.io.StatusRecord* attribute), 29

`rms` (*vallenae.io.TraRecord* attribute), 35

`rms()` (in module *vallenae.features*), 44

`rows()` (*vallenae.io.PriDatabase* method), 9

`rows()` (*vallenae.io.TraDatabase* method), 16

`rows()` (*vallenae.io.TrfDatabase* method), 20

S

`samplerate` (*vallenae.io.TraRecord* attribute), 35

`samples` (*vallenae.io.TraRecord* attribute), 35

`set_id` (*vallenae.io.HitRecord* attribute), 25

`set_id` (*vallenae.io.MarkerRecord* attribute), 27

`set_id` (*vallenae.io.ParametricRecord* attribute), 32

`set_id` (*vallenae.io.StatusRecord* attribute), 29

`set_type` (*vallenae.io.MarkerRecord* attribute), 27

`signal_strength` (*vallenae.io.HitRecord* attribute), 25

`signal_strength` (*vallenae.io.StatusRecord* attribute), 29

`signal_strength()` (in module *vallenae.features*), 43

StatusRecord (class in *vallenae.io*), 28

T

`tables()` (*vallenae.io.PriDatabase* method), 10

`tables()` (*vallenae.io.TraDatabase* method), 16

[tables\(\)](#) (*vallenae.io.TrfDatabase* method), 20
[threshold](#) (*vallenae.io.HitRecord* attribute), 25
[threshold](#) (*vallenae.io.StatusRecord* attribute), 29
[threshold](#) (*vallenae.io.TraRecord* attribute), 35
[time](#) (*vallenae.io.HitRecord* attribute), 25
[time](#) (*vallenae.io.MarkerRecord* attribute), 27
[time](#) (*vallenae.io.ParametricRecord* attribute), 32
[time](#) (*vallenae.io.StatusRecord* attribute), 29
[time](#) (*vallenae.io.TraRecord* attribute), 35
[TraDatabase](#) (class in *vallenae.io*), 11
[tra](#) (*vallenae.io.FeatureRecord* attribute), 37
[tra](#) (*vallenae.io.HitRecord* attribute), 25
[tra](#) (*vallenae.io.TraRecord* attribute), 35
[TraRecord](#) (class in *vallenae.io*), 33
[TrfDatabase](#) (class in *vallenae.io*), 17

V

[vallenae.features](#)
 module, 39
[vallenae.io](#)
 module, 1
[vallenae.timepicker](#)
 module, 45

W

[write\(\)](#) (*vallenae.io.TraDatabase* method), 16
[write\(\)](#) (*vallenae.io.TrfDatabase* method), 21
[write_fieldinfo\(\)](#) (*vallenae.io.PriDatabase* method),
 10
[write_fieldinfo\(\)](#) (*vallenae.io.TraDatabase* method),
 17
[write_fieldinfo\(\)](#) (*vallenae.io.TrfDatabase* method),
 21
[write_hit\(\)](#) (*vallenae.io.PriDatabase* method), 10
[write_marker\(\)](#) (*vallenae.io.PriDatabase* method), 10
[write_parametric\(\)](#) (*vallenae.io.PriDatabase*
 method), 11
[write_status\(\)](#) (*vallenae.io.PriDatabase* method), 11