alphatims

Mann Labs, MPIB

Jun 09, 2022

CONTENTS

With the introduction of the Bruker TimsTOF and Parallel Accumulation–Serial Fragmentation (PASEF), the inclusion of trapped ion mobility separation (TIMS) between liquid chromatography (LC) and tandem mass spectrometry (MSMS) instruments has gained popularity for both DDA and DIA. However, detection of such five dimensional points (chromatographic retention time (rt), ion mobility, quadrupole mass to charge (m/z), time-of-flight (TOF) m/z and intensity) at GHz results in an increased amount of data and complexity. Efficient accession, analysis and visualisation of Bruker TimsTOF data are therefore imperative. AlphaTims is freely available, open-source and available on all major Operating Systems. It can be used with a graphical user interface (GUI), a command-line interface (CLI) or as a regular Python package.

This documentation is intended as an API for direct Python use. For more information, see AlphaTims on GitHub.

CHAPTER

ONE

ALPHATIMS.UTILS

This module provides generic utilities. These utilities primarily focus on:

- logging
- compilation
- parallelization
- generic io

Classes:

Global_Stack(all_available_options)	A stack that holds multiple option stacks.
Option_Stack(option_name, option_initial_value)	A stack with the option to redo and undo.

Functions:

check_github_version([silent])	Checks and logs the current version of AlphaTims.
create_dict_from_hdf_group(hdf_group[, ...])	Convert the contents of an HDF group and return as normal Python dict.
create_hdf_group_from_dict(hdf_group, ...[, ...])	Save a dict to an open hdf group.
load_parameters(parameter_file_name)	Load a parameter dict from a file.
njit([_func])	A wrapper for the numba.njit decorator.
pjit([_func, thread_count, ...])	A decorator that parallelizes the numba.njit decorator with threads.
progress_callback(iterable[, ...])	A generator that adds progress callback to an iterable.
save_parameters(parameter_file_name, paramaters)	Save parameters to a parameter file.
set_logger(*[, log_file_name, stream, ...])	Set the log stream and file.
set_progress_callback(progress_callback)	Set the global progress callback.
set_threads(threads[, set_global])	Parse and set the (global) number of threads.
show_platform_info()	Log all platform information.
show_python_info()	Log all Python information.
threadpool([_func, thread_count, ...])	A decorator that parallelizes a function with threads and callback.

class alphatims.utils.**Global_Stack**(*all_available_options: dict*)

Bases: object

A stack that holds multiple option stacks.

The current value of each option stack can be retrieved by indexing, i.e. option_value = self[option_key].

Methods:

__init__(all_available_options)	Create a global stack.
lock()	A context manager to lock this stack and prevent modification.
redo()	Increase the stack pointer with 1.
trim()	Remove all elements above of the current stack pointer
undo()	Reduce the stack pointer with 1.
update(option_key, option_value)	Update an option stack with a value.

Attributes:

current_values	A dict with (option_key: option_value) mapping.
is_locked	A flag to check if this stack is modifiable
size	The size of this stack without the initial value.

__init__(*all_available_options: dict*)
Create a global stack.

Parameters **all_available_options** (*dict*) – A dictionary whose items are (str, type), which can be used to create an Option_Stack.

property current_values: dict
A dict with (option_key: option_value) mapping.

Type dict

property is_locked
A flag to check if this stack is modifiable

Type bool

lock()
A context manager to lock this stack and prevent modification.

redo() → tuple
Increase the stack pointer with 1.

Returns (“”, None) if the pointer was already at the maximum. Otherwise (option_name, new_value) if the pointer was increased.

Return type tuple

property size
The size of this stack without the initial value.

Type int

trim() → bool
Remove all elements above of the current stack pointer

Returns True if something was removed, i.e. if stack pointer was not at the top. False if nothing could be deleted, i.e. the stack pointer was at the top.

Return type bool

undo() → tuple
Reduce the stack pointer with 1.

Returns (“”, None) if the pointer was already at the maximum. Otherwise (option_name, new_value) if the pointer was reduced.

Return type tuple

update(*option_key: str*, *option_value*) → tuple
Update an option stack with a value.

Parameters

- **option_key** (*str*) – The name of the option stack to update.
- **option_value** (*type*) – An value to add to this stack. Can be any object that supports the "!=" operator.

Returns ("", None) if the pointer was not updated, i.e. the latest update was equal to the current update. Otherwise (option_name, new_value).

Return type tuple

class alphatims.utils.**Option_Stack**(*option_name: str*, *option_initial_value*)
Bases: object

A stack with the option to redo and undo.

Methods:

__init__(option_name, option_initial_value)	Create an option stack.
redo()	Increase the stack pointer with 1.
trim()	Remove all elements above of the current stack pointer
undo()	Reduce the stack pointer with 1.
update(option_value)	Update this stack with the value.

Attributes:

current_value	The current value of this stack.
option_name	The name of this stack.
size	The size of this stack without the initial value.

__init__(*option_name: str*, *option_initial_value*)
Create an option stack.

Parameters

- **option_name** (*str*) – The name of this option.
- **option_initial_value** (*type*) – The initial value of this stack. Can be any object that supports the "!=" operator.

property current_value
The current value of this stack.

Type type

property option_name: str
The name of this stack.

Type str

redo()
Increase the stack pointer with 1.

Returns None if the pointer was already at the maximum. Otherwise the new value if the pointer was increased.

Return type type

property size: int
The size of this stack without the initial value.

Type int

trim() → bool
Remove all elements above of the current stack pointer

Returns True if something was removed, i.e. if stack pointer was not at the top. False if nothing could be deleted, i.e. the stack pointer was at the top.

Return type bool

undo()
Reduce the stack pointer with 1.

Returns None if the pointer was already at the maximum. Otherwise the new value if the pointer was reduced.

Return type type

update(*option_value*) → bool
Update this stack with the value.

Parameters **option_value** (*type*) – An value to add to this stack. Can be any object that supports the "!=" operator.

Returns True if the stack was updated. False if the provided value equald the current value of this stack.

Return type bool

alphatims.utils.**check_github_version**(*silent=False*) → str
Checks and logs the current version of AlphaTims.

Check if the local version equals the AlphaTims GitHub master branch. This is only possible with an active internet connection and if no credentials are required for GitHub.

Parameters **silent** (*str*) – Use the logger to display the obtained conclusion. Default is False.

Returns The version on the AlphaTims GitHub master branch. "" if no version can be found on GitHub

Return type str

alphatims.utils.**create_dict_from_hdf_group**(*hdf_group*, *mmap_arrays=None*, *parent_file_name: Optional[str] = None*) → dict
Convert the contents of an HDF group and return as normal Python dict.

Parameters

- **hdf_group** (*h5py.File.group*) – An open and readable HDF group.
- **mmap_arrays** (*iterable*) – These array will be mmapped instead of pre-loaded. Default is None
- **parent_file_name** (*str*) – The parent_file_name. This is required when mmap_arrays is not None. Default is None.

Returns A Python dict. Keys of the dict are names of arrays, attrs and subgroups. Values are corresponding arrays and attrs. Subgroups are converted to subdicts. If a subgroup has an "is_pd_dataframe=True" attr, it is automatically converted to a pd.dataFrame.

Return type dict

Raises **ValueError** – When an attr value in the HDF group is not an int, float, str or bool.

alphatims.utils.**create_hdf_group_from_dict**(*hdf_group*, *data_dict: dict*, *, *overwrite: bool = False*, *compress: bool = False*, *recursed: bool = False*, *chunked: bool = False*) → None

Save a dict to an open hdf group.

Parameters

- **hdf_group** (*h5py.File.group*) – An open and writable HDF group.
- **data_dict** (*dict*) – A dict that needs to be written to HDF. Keys always need to be strings. Values are stored as follows:
  - subdicts -> subgroups.
  - np.array -> array
  - pd.dataframes -> subdicts with "is_pd_dataframe: True" attribute.
  - bool, int, float and str -> attrs.
  - None values are skipped and not stored explicitly.
- **overwrite** (*bool*) – If True, existing subgroups, arrays and attrs are fully truncated/overwritten. If False, the existing value in HDF remains unchanged. Default is False.
- **compress** (*bool*) – If True, all arrays are compressed with binary shuffle and "lzf" compression. If False, arrays are saved as provided. On average, compression halves file sizes, at the cost of 2-10 time longer accession times. Default is False.
- **recursed** (*bool*) – If False, the default progress callback is added while itereating over the keys of the data_dict. If True, no callback is added, allowing subdicts to not trigger callback. Default is False.
- **chunked** (*bool*) – If True, all arrays are chunked. If False, arrays are saved as provided. Default is False.

Raises

- **ValueError** – When a value of data_dict cannot be converted to an HDF value (see data_dict).
- **KeyError** – When a key of data_dict is not a string.

alphatims.utils.**load_parameters**(*parameter_file_name: str*) → dict

Load a parameter dict from a file.

Parameters **parameter_file_name** (*str*) – A file name that contains parameters in .json format.

Returns A dict with parameters.

Return type dict

alphatims.utils.**njit**(*_func=None*, **args*, ***kwargs*)

A wrapper for the numba.njit decorator.

The "cache" option is set to True by default. This can be overriden with kwargs.

Parameters

- **_func** (*callable, None*) – The function to decorate.
- ***args** – See numba.njit decorator.
- ****kwargs** – See numba.njit decorator.

Returns A numba.njit decorated function.

Return type function

alphatims.utils.**pjit**(*_func=None*, ***, *thread_count=None*, *include_progress_callback: bool = True*, *cache: bool = True*, ***kwargs*)

A decorator that parallelizes the numba.njit decorator with threads.

The first argument of the decorated function need to be an iterable. A range-object will be most performant as iterable. The original function should accept a single element of this iterable as its first argument. The original function cannot return values, instead it should store results in e.g. one if its input arrays that acts as a buffer array. The original function needs to be numba.njit compatible. Numba argument "nogil" is always set to True.

Parameters

- **_func** (*callable, None*) – The function to decorate.
- **thread_count** (*int, None*) – The number of threads to use. This is always parsed with alphatims.utils.set_threads. Not possible as positional arguments, it always needs to be an explicit keyword argument. Default is None.
- **include_progress_callback** (*bool*) – If True, the default progress callback will be used as callback. (See "progress_callback" function.) If False, no callback is added. See *set_progress_callback* for callback styles. Default is True.
- **cache** (*bool*) – See numba.njit decorator. Default is True (in contrast to numba).

Returns A parallelized numba.njit decorated function.

Return type function

alphatims.utils.**progress_callback**(*iterable*, *include_progress_callback: bool = True*, *total: int = - 1*)

A generator that adds progress callback to an iterable.

Parameters

- **iterable** – An iterable.
- **include_progress_callback** (*bool*) – If True, the default progress callback will be used as callback. If False, no callback is added. See *set_progress_callback* for callback styles. Default is True.
- **total** (*int*) – The length of the iterable. If -1, this will be read as len(iterable), if __len__ is implemented. Default is -1.

Returns A generator over the iterable with added callback.

Return type iterable

alphatims.utils.**save_parameters**(*parameter_file_name: str*, *paramaters: dict*) → None

Save parameters to a parameter file.

IMPORTANT NOTE: This overwrites any existing file.

Parameters

- **parameter_file_name** (*str*) – The file name to where the parameters are written.
- **paramaters** (*dict*) – A dictionary with parameters.

alphatims.utils.**set_logger**(***, *log_file_name=''*, *stream: bool = True*, *log_level: int = 20*, *overwrite: bool = False*) → str

Set the log stream and file.

All previously set handlers will be disabled with this command.

Parameters

- **log_file_name** (*str, None*) – The file name to where the log is written. Folders are automatically created if needed. This is relative to the current path. When an empty string is provided, a log is written to the AlphaTims "logs" folder with the name "log_yymmddhhmmss" (reversed timestamp year to seconds). If None, no log file is saved. Default is "".
- **stream** (*bool*) – If False, no log data is sent to stream. If True, all logging can be tracked with stdout stream. Default is True.
- **log_level** (*int*) – The logging level. Usable values are defined in Python's "logging" module. Default is logging.INFO.
- **overwrite** (*bool*) – If True, overwrite the log_file if one exists. If False, append to this log file. Default is False.

Returns The file name to where the log is written.

Return type str

alphatims.utils.**set_progress_callback**(*progress_callback*)

Set the global progress callback.

Parameters **progress_callback** – The new global progress callback. Options are:

- None, no progress callback will be used
- True, a textual progress callback (tqdm) will be enabled
- Any object that supports a *max* and *value* variable.

alphatims.utils.**set_threads**(*threads: int*, *set_global: bool = True*) → int

Parse and set the (global) number of threads.

Parameters

- **threads** (*int*) – The number of threads. If larger than available cores, it is trimmed to the available maximum. If 0, it is set to the maximum cores available. If negative, it indicates how many cores NOT to use.
- **set_global** (*bool*) – If False, the number of threads is only parsed to a valid value. If True, the number of threads is saved as a global variable. Default is True.

Returns The number of threads.

Return type int

alphatims.utils.**show_platform_info**() → None

Log all platform information.

This is done in the following format:

- [timestamp]> Platform information:
- [timestamp]> system - [...]
- [timestamp]> release - [...]
- [timestamp]> version - [...]
- [timestamp]> machine - [...]
- [timestamp]> processor - [...]
- [timestamp]> cpu count - [...]
- [timestamp]> cpu frequency - [...]

- [timestamp]> ram - [...]/[...] Gb (available/total)

alphatims.utils.**show_python_info**() → None
Log all Python information.

This is done in the following format:

- [timestamp]> Python information:
- [timestamp]> alphatims - [current_version]
- [timestamp]> [required package] - [current_version]
- ...
- [timestamp]> [required package] - [current_version]

alphatims.utils.**threadpool**(*_func=None*, ***, *thread_count=None*, *include_progress_callback: bool = True*, *return_results: bool = False*) → None
A decorator that parallelizes a function with threads and callback.

The original function should accept a single element as its first argument. If the caller function provides an iterable as first argument, the function is applied to each element of this iterable in parallel.

Parameters

- **_func** (*callable, None*) – The function to decorate.
- **thread_count** (*int, None*) – The number of threads to use. This is always parsed with alphatims.utils.set_threads. Not possible as positional arguments, it always needs to be an explicit keyword argument. Default is None.
- **include_progress_callback** (*bool*) – If True, the default progress callback will be used as callback. (See "progress_callback" function.) If False, no callback is added. See *set_progress_callback* for callback styles. Default is True.
- **return_results** (*bool*) – If True, it returns the results in the same order as the iterable. This can be much slower than not returning results. Iti is better to store them in a buffer results array instead (be carefull to avoid race conditions). If the iterable is not an iterable but a single index, a result is always returned. Default is False.

Returns A parallelized decorated function.

Return type function

CHAPTER

TWO

ALPHATIMS.BRUKER

This module provides functions to handle Bruker data. It primarily implements the TimsTOF class, that acts as an in-memory container for Bruker data accession and storage.

Exceptions:

PrecursorFloatError	Used to indicate that a precursor value is not an int but a float.

Classes:

TimsTOF(bruker_d_folder_name, *[, ...])	A class that stores Bruker TimsTOF data in memory for fast access.

Functions:

add_intensity_to_bin(query_index, ...)	Add the intensity of a query to the appropriate bin.
calculate_dia_cycle_mask(dia_mz_cycle, ...)	Calculate a boolean mask for cyclic push indices satisfying queries.
centroid_spectra(index, spectrum_indptr, ...)	Smoothen and centroid a profile spectrum (inplace operation).
convert_slice_key_to_float_array(key)	Convert a key to a slice float array.
convert_slice_key_to_int_array(data, key, ...)	Convert a key of a data dimension to a slice integer array.
filter_indices(frame_slices, scan_slices, ...)	Filter raw indices by slices from all dimensions.
filter_spectra_by_abundant_peaks(index, ...)	Filter a spectrum to retain only the most abundant peaks.
filter_tof_to_csr(tof_slices, push_indices, ...)	Get a CSR-matrix with raw indices satisfying push indices and tof slices.
get_dia_push_indices(frame_slices, ...[, ...])	Filter DIA push indices by slices from LC, TIMS and QUAD.
indptr_lookup(targets, queries[, ...])	Find the indices of queries in targets.
init_bruker_dll([bruker_dll_file_name])	Open a bruker.dll in Python.
open_bruker_d_folder(bruker_d_folder_name[, ...])	A context manager for a bruker dll connection to a .d folder.
parse_decompressed_bruker_binary_type1(...)	Parse a Bruker binary scan buffer into tofs and intensities.
parse_decompressed_bruker_binary_type2(...)	Parse a Bruker binary frame buffer into scans, tofs and intensities.
parse_keys(data, keys)	Convert different keys to a key dict with defined types.
process_frame(frame_id, tdf_bin_file_name, ...)	Read and parse a frame directly from a Bruker .d.analysis.tdf_bin.

continues on next page

Table 3 – continued from previous page

read_bruker_binary(frames, ...[, ...])	Read all data from an "analysis.tdf_bin" of a Bruker .d folder.
read_bruker_sql(bruker_d_folder_name[, ...])	Read metadata, (fragment) frames and precursors from a Bruker .d folder.
set_precursor(precursor_index, offset_order, ...)	Sum the intensities of all pushes belonging to a single precursor.
trim_spectra(index, spectrum_tof_indices, ...)	Trim remaining bytes after merging of multiple pushes.
valid_precursor_index(precursor_index, ...)	Check if a precursor index is included in the slices.
valid_quad_mz_values(low_mz_value, ...)	Check if the low and high quad mz values are included in the slices.

exception alphatims.bruker.**PrecursorFloatError**

Bases: TypeError

Used to indicate that a precursor value is not an int but a float.

class alphatims.bruker.**TimsTOF**(*bruker_d_folder_name: str*, *, *mz_estimation_from_frame: int = 1*, *mobility_estimation_from_frame: int = 1*, *slice_as_dataframe: bool = True*, *use_calibrated_mz_values_as_default: int = 0*, *use_hdf_if_available: bool = True*, *mmap_detector_events: bool = True*, *drop_polarity: bool = True*, *convert_polarity_to_int: bool = True*)

Bases: object

A class that stores Bruker TimsTOF data in memory for fast access.

Data can be read directly from a Bruker .d folder. All OS's are supported, but reading mz_values and mobility_values from a .d folder requires Windows or Linux due to availability of Bruker libraries. On MacOS, they are estimated based on metadata, but these values are not guaranteed to be correct. Often they fall within 0.02 Th, but errors up to 6 Th have already been observed!

A TimsTOF object can also be exported to HDF for subsequent access. This file format is portable to all OS's. As such, initial reading on Windows with correct mz_values and mobility_values can be done and the resulting HDF file can safely be read on MacOS. This HDF file also provides improved accession times for subsequent use.

After reading, data can be accessed with traditional Python slices. As TimsTOF data is 5-dimensional, the data can be sliced in 5 dimensions as well. These dimensions follows the design of the TimsTOF Pro:

1 LC: rt_values, frame_indices The first dimension allows to slice retention_time values or frames indices. These values and indices have a one-to-one relationship.

2 TIMS: mobility_values, scan_indices The second dimension allows to slice mobility values or scan indices (i.e. a single push). These values and indices have a one-to-one relationship.

3 QUAD: quad_mz_values, precursor_indices The third dimension focusses on the quadrupole and indirectly on the collision cell. It allows to slice lower and upper quadrupole mz values (e.g. the m/z of unfragmented ions / precursors). If set to -1, the quadrupole and collision cell are assumed to be inactive, i.e. precursor ions are detected instead of fragments. Equally, this dimension allows to slice precursor indices. Precursor index 0 defaults to all precusors (i.e. quad mz values equal to -1). In DDA, precursor indices larger than 0 point to ddaPASEF MSMS spectra. In DIA, precursor indices larger than 0 point to windows, i.e. all scans in a frame with equal quadrupole and collision settings that is repeated once per full cycle. Note that these values do not have a one-to-one relationship.

4 TOF: mz_values, tof_indices The fourth dimension allows to slice (fragment) mz_values or tof indices. Note that the quadrupole dimension determines if precursors are detected or fragments. These values and indices have a one-to-one relationship.

5 DETECTOR: intensity_values The fifth dimension allows to slice intensity values.

Note that all dimensions except for the detector have both (float) values and (integer) indices. For each dimension, slices can be provided in several different ways:

- **int:** A single int can be used to select a single index. If used in the fifth dimension, it still allows to select intensity_values
- **float:** A single float can be used to select a single value. As the values arrays are discrete, the smallest index with a value equal to or larger than this value is actually selected. For intensity_value slicing, the exact value is used.
- **slice:** A Python slice with start, stop and step can be provided. Start and stop values can independently be set to int or float. If a float is provided it conversed to an int as previously described. The step always needs to be provided as an int. Since there is not one-to-one relation from values to indices for QUAD and DETECTOR, the step value is ignored in these cases and only start and stop can be used.

  IMPORTANT NOTE: negative start, step and stop integers are not supported!
- **iterable:** An iterable with (mixed) floats and ints can also be provided, in a similar fashion as Numpy's fancy indexing.

  IMPORTANT NOTE: The resulting integers after float->int conversion need to be sorted in ascending order!
- **np.ndarray:** Multiple slicing is supported by providing either a np.int64[:, 3] array, where each row is assumed to be a (start, stop, step) tuple or np.float64[:, 2] where each row is assumed to be a (start, stop) tuple.

  IMPORTANT NOTE: These arrays need to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(precursor_slices[:, :2].ravel()) >= 0) = True).

Alternatively, a dictionary can be used to define filters for each dimension (see examples).

The result of such slicing is a pd.DataFrame with the following columns:

- raw_indices
- frame_indices
- scan_indices
- precursor_indices
- tof_indices
- rt_values
- mobility_values
- quad_low_mz_values
- quad_high_mz_values
- mz_values
- intensity_values

Instead of returning a pd.DataFrame, raw indices can be returned by setting the last slice element to "raw".

Examples

```
>>> data[:100.0]
# Return all datapoints with rt_values < 100.0 seconds
```

```
>>> data[:, 450]
# Return all datapoints with scan_index = 450
```

```
>>> data[:, :, 700.: 710.]
# Return all datapoints with 700.0 <= quad_mz_values < 710.0
```

```
>>> data[:, :, :, 621.9: 191000]
# Return all datapoints with 621.9 <= mz_values and
# tof_indices < 191000
```

```
>>> data[[1, 8, 10], :, 0, 621.9: np.inf]
# Return all datapoints from frames 1, 8 and 10, which are unfragmented
# (precursor_index = 0) and with 621.9 <= mz_values < np.inf
```

```
>>> data[:, :, 999]
# Return all datapoints from precursor 999
# (for diaPASEF this is a traditional MSMS spectrum)
```

```
>>> scan_slices = np.array([[10, 20, 1], [100, 200, 10]])
>>> data[:, scan_slices, :, :, :]
# Return all datapoints with scan_indices in range(10, 20) or
# range(100, 200, 10)
```

```
>>> df = data[
...     {
...         "frame_indices": [1, 191],
...         "scan_indices": slice(300, 800, 10),
...         "mz_values": slice(None, 400.5),
...         "intensity_values": 50,
...     }
... ]
# Slice by using a dictionary
```

```
>>> data[:, :, 999, "raw"]
# Return the raw indices of datapoints from precursor 999
```

Methods:

`__init__`(bruker_d_folder_name, *[, ...])	Create a Bruker TimsTOF object that contains all data in-memory.
`as_dataframe`(indices, *[, raw_indices, ...])	Convert raw indices to a pd.DataFrame.
`bin_intensities`(indices, axis)	Sum and project the intensities of the indices along 1 or 2 axis.
`calculate_global_calibrated_mz_values`([...])	Calculate global calibrated_mz_values based on two calibrant ions.

continues on next page

Table 4 – continued from previous page

convert_from_indices(raw_indices, *[, ...])	Convert selected indices to a dict.
convert_to_indices(values, *[, ...])	Convert selected values to an array in the requested dimension.
estimate_strike_count(frame_slices, ...)	Estimate the number of detector events, given a set of slices.
index_precursors([centroiding_window, ...])	Retrieve all MS2 spectra acquired with DDA.
save_as_hdf(directory, file_name[, ...])	Save the TimsTOF object as an hdf file.
save_as_mgf(directory, file_name[, ...])	Save profile spectra from this TimsTOF object as an mgf file.
use_calibrated_mz_values_as_default(...)	Override the default mz_values with the global calibrated_mz_values.

Attributes:

accumulation_times	The accumulation times of all frames.
acquisition_mode	The acquisition mode.
calibrated_mz_max_value	The maximum calibrated mz value.
calibrated_mz_min_value	The minimum calibrated mz value.
calibrated_mz_values	np.float64[:] : The global calibrated mz values.
dia_mz_cycle	np.float64[:, 2] : The mz_values of a DIA cycle.
dia_precursor_cycle	np.int64[:] : The precursor indices of a DIA cycle.
directory	The directory of this TimsTOF object.
fragment_frames	The fragment frames table.
frame_max_index	The maximum frame index.
frames	The frames table of the analysis.tdf SQL.
intensity_corrections	The intensity_correction per frame.
intensity_max_value	The maximum intensity value.
intensity_min_value	The minimum intensity value.
intensity_values	np.uint16[:] : The intensity values.
is_compressed	HDF array is compressed or not.
max_accumulation_time	The maximum accumulation time of all frames.
meta_data	The metadata for the acquisition.
mobility_max_value	The maximum mobility value.
mobility_min_value	The minimum mobility value.
mobility_values	np.float64[:] : The mobility values.
mz_max_value	The maximum mz value.
mz_min_value	The minimum mz value.
mz_values	np.float64[:] : The mz values.
precursor_indices	np.int64[:] : The precursor indices.
precursor_max_index	The maximum precursor index.
precursors	The precursor table.
push_indptr	np.int64[:] : The tof indptr.
quad_indptr	np.int64[:] : The quad indptr (tof_indices).
quad_mz_max_value	The maximum quad mz value.
quad_mz_min_value	The minimum quad mz value.
quad_mz_values	np.float64[:, 2] : The (low, high) quad mz values.
raw_quad_indptr	np.int64[:] : The raw quad indptr (push indices).
rt_max_value	The maximum rt value.
rt_values	np.float64[:] : The rt values.
sample_name	The sample name of this TimsTOF object.

continues on next page

Table 5 – continued from previous page

scan_max_index	The maximum scan index.
tof_indices	np.uint32[:] : The tof indices.
tof_max_index	The maximum tof index.
version	AlphaTims version used to create this TimsTOF object.
zeroth_frame	A blank zeroth frame is present so frames are 1-indexed.

__init__(*bruker_d_folder_name: str*, *, *mz_estimation_from_frame: int = 1*, *mobility_estimation_from_frame: int = 1*, *slice_as_dataframe: bool = True*, *use_calibrated_mz_values_as_default: int = 0*, *use_hdf_if_available: bool = True*, *mmap_detector_events: bool = True*, *drop_polarity: bool = True*, *convert_polarity_to_int: bool = True*)

Create a Bruker TimsTOF object that contains all data in-memory.

Parameters

- **bruker_d_folder_name** (*str*) – The full file name to a Bruker .d folder. Alternatively, the full file name of an already exported .hdf can be provided as well.
- **mz_estimation_from_frame** (*int*) – If larger than 0, mz_values from this frame are read as default mz_values with the Bruker library. If 0, mz_values are being estimated with the metadata based on "MzAcqRangeLower" and "MzAcqRangeUpper". IMPORTANT NOTE: MacOS defaults to 0, as no Bruker library is available. Default is 1.
- **mobility_estimation_from_frame** (*int*) – If larger than 0, mobility_values from this frame are read as default mobility_values with the Bruker library. If 0, mobility_values are being estimated with the metadata based on "OneOverK0AcqRangeLower" and "OneOverK0AcqRangeUpper". IMPORTANT NOTE: MacOS defaults to 0, as no Bruker library is available. Default is 1.
- **slice_as_dataframe** (*bool*) – If True, slicing returns a pd.DataFrame by default. If False, slicing provides a np.int64[:] with raw indices. This value can also be modified after creation. Default is True.
- **use_calibrated_mz_values** (*int*) – If not 0, the mz_values are overwritten with global calibrated_mz_values. If 1, calibration at the MS1 level is performed. If 2, calibration at the MS2 level is performed. Default is 0.
- **use_hdf_if_available** (*bool*) – If an HDF file is available, use this instead of the .d folder. Default is True.
- **mmap_detector_events** (*bool*) – Do not save the intensity_values and tof_indices in memory, but use an mmap instead. Default is True
- **drop_polarity** (*bool*) – The polarity column of the frames table contains "+" or "-" and is not numerical. If True, the polarity column is dropped from the frames table. this ensures a fully numerical pd.DataFrame. If False, this column is kept, resulting in a pd.DataFrame with dtype=object. Default is True.
- **convert_polarity_to_int** (*bool*) – Convert the polarity to int (-1 or +1). This allows to keep it in numerical form. This is ignored if the polarity is dropped. Default is True.

property accumulation_times

The accumulation times of all frames.

Type np.ndarray

property acquisition_mode
The acquisition mode.

Type str

as_dataframe(*indices: numpy.ndarray*, ***, *raw_indices: bool = True*, *frame_indices: bool = True*, *scan_indices: bool = True*, *quad_indices: bool = False*, *tof_indices: bool = True*, *precursor_indices: bool = True*, *rt_values: bool = True*, *rt_values_min: bool = True*, *mobility_values: bool = True*, *quad_mz_values: bool = True*, *push_indices: bool = True*, *mz_values: bool = True*, *intensity_values: bool = True*, *corrected_intensity_values: bool = True*, *raw_indices_sorted: bool = False*)
Convert raw indices to a pd.DataFrame.

Parameters

- **indices** (*np.int64[:]*) – The raw indices for which coordinates need to be retrieved.
- **raw_indices** (*bool*) – If True, include "raw_indices" in the dataframe. Default is True.
- **frame_indices** (*bool*) – If True, include "frame_indices" in the dataframe. Default is True.
- **scan_indices** (*bool*) – If True, include "scan_indices" in the dataframe. Default is True.
- **quad_indices** (*bool*) – If True, include "quad_indices" in the dataframe. Default is False.
- **tof_indices** (*bool*) – If True, include "tof_indices" in the dataframe. Default is True.
- **precursor_indices** (*bool*) – If True, include "precursor_indices" in the dataframe. Default is True.
- **rt_values** (*bool*) – If True, include "rt_values" in the dataframe. Default is True.
- **rt_values_min** (*bool*) – If True, include "rt_values_min" in the dataframe. Default is True.
- **mobility_values** (*bool*) – If True, include "mobility_values" in the dataframe. Default is True.
- **quad_mz_values** (*bool*) – If True, include "quad_low_mz_values" and "quad_high_mz_values" in the dict. Default is True.
- **push_indices** (*bool*) – If True, include "push_indices" in the dataframe. Default is True.
- **mz_values** (*bool*) – If True, include "mz_values" in the dataframe. Default is True.
- **intensity_values** (*bool*) – If True, include "intensity_values" in the dataframe. Default is True.
- **corrected_intensity_values** (*bool*) – If True, include "corrected_intensity_values" in the dataframe. Default is True.
- **raw_indices_sorted** (*bool*) – If True, raw_indices are assumed to be sorted, resulting in a faster conversion. Default is False.

Returns A dataframe with all requested columns.

Return type pd.DataFrame

bin_intensities(*indices: numpy.ndarray*, *axis: tuple*)
Sum and project the intensities of the indices along 1 or 2 axis.

Parameters

- **indices** (*np.int64[:]*) – The selected indices whose coordinates need to be summed along the selected axis.
- **axis** (*tuple*) – Must be length 1 or 2 and can only contain the elements "rt_values", "mobility_values" and "mz_values".

Returns **np.float64[** – An array or heatmap that express the summed intensity along the selected axis.

Return type], np.float64[:, 2]

calculate_global_calibrated_mz_values(*calibrant1: tuple = (922.009798, 1.1895, slice(0, 1, None))*, *calibrant2: tuple = (1221.990637, 1.382, slice(0, 1, None))*, *mz_tolerance: float = 10*, *mobility_tolerance: float = 0.1*) → None

Calculate global calibrated_mz_values based on two calibrant ions.

Parameters

- **calibrant1** (*tuple*) – The first calibrant ion. This is a tuple with (mz, mobility, precursor_slice) foat values. Default is (922.009798, 1.1895, slice(0, 1)).
- **calibrant2** (*tuple*) – The first calibrant ion. This is a tuple with (mz, mobility, precursor_slice) foat values. Default is (1221.990637, 1.3820, slice(0, 1)).
- **mz_tolerance** (*float*) – The tolerance window (in Th) with respect to the uncalibrated mz_values. If this is too large, the calibrant ion might not be the most intense ion anymore. If this is too small, the calibrant ion might not be contained. Default is 10.
- **mobility_tolerance** (*float*) – The tolerance window with respect to the uncalibrated mobility_values. If this is too large, the calibrant ion might not be the most intense ion anymore. If this is too small, the calibrant ion might not be contained. Default is 0.1.

property calibrated_mz_max_value

The maximum calibrated mz value.

Type float

property calibrated_mz_min_value

The minimum calibrated mz value.

Type float

property calibrated_mz_values

np.float64[:] : The global calibrated mz values.

Type np.ndarray

convert_from_indices(*raw_indices*, *, *frame_indices=None*, *quad_indices=None*, *scan_indices=None*, *tof_indices=None*, *return_raw_indices: bool = False*, *return_frame_indices: bool = False*, *return_scan_indices: bool = False*, *return_quad_indices: bool = False*, *return_tof_indices: bool = False*, *return_precursor_indices: bool = False*, *return_rt_values: bool = False*, *return_rt_values_min: bool = False*, *return_mobility_values: bool = False*, *return_quad_mz_values: bool = False*, *return_push_indices: bool = False*, *return_mz_values: bool = False*, *return_intensity_values: bool = False*, *return_corrected_intensity_values: bool = False*, *raw_indices_sorted: bool = False*) → dict

Convert selected indices to a dict.

Parameters

- **raw_indices** (*np.int64[:], None*) – The raw indices for which coordinates need to be retrieved.

- **frame_indices** (*np.int64[:], None*) – The frame indices for which coordinates need to be retrieved.
- **quad_indices** (*np.int64[:], None*) – The quad indices for which coordinates need to be retrieved.
- **scan_indices** (*np.int64[:], None*) – The scan indices for which coordinates need to be retrieved.
- **tof_indices** (*np.int64[:], None*) – The tof indices for which coordinates need to be retrieved.
- **return_raw_indices** (*bool*) – If True, include "raw_indices" in the dict. Default is False.
- **return_frame_indices** (*bool*) – If True, include "frame_indices" in the dict. Default is False.
- **return_scan_indices** (*bool*) – If True, include "scan_indices" in the dict. Default is False.
- **return_quad_indices** (*bool*) – If True, include "quad_indices" in the dict. Default is False.
- **return_tof_indices** (*bool*) – If True, include "tof_indices" in the dict. Default is False.
- **return_precursor_indices** (*bool*) – If True, include "precursor_indices" in the dict. Default is False.
- **return_rt_values** (*bool*) – If True, include "rt_values" in the dict. Default is False.
- **return_rt_values_min** (*bool*) – If True, include "rt_values_min" in the dict. Default is False.
- **return_mobility_values** (*bool*) – If True, include "mobility_values" in the dict. Default is False.
- **return_quad_mz_values** (*bool*) – If True, include "quad_low_mz_values" and "quad_high_mz_values" in the dict. Default is False.
- **return_push_indices** (*bool*) – If True, include "push_indices" in the dict. Default is False.
- **return_mz_values** (*bool*) – If True, include "mz_values" in the dict. Default is False.
- **return_intensity_values** (*bool*) – If True, include "intensity_values" in the dict. Default is False.
- **return_corrected_intensity_values** (*bool*) – If True, include "corrected_intensity_values" in the dict. Default is False.
- **raw_indices_sorted** (*bool*) – If True, raw_indices are assumed to be sorted, resulting in a faster conversion. Default is False.

Returns A dict with all requested columns.

Return type dict

convert_to_indices(*values: numpy.ndarray*, ***, *return_frame_indices: bool = False*, *return_scan_indices: bool = False*, *return_tof_indices: bool = False*, *side: str = 'left'*, *return_type: str = ''*)

Convert selected values to an array in the requested dimension.

Parameters

- **values** (*float, np.float64[...], iterable*) – The raw values for which indices need to be retrieved.
- **return_frame_indices** (*bool*) – If True, convert the values to "frame_indices". Default is False.
- **return_scan_indices** (*bool*) – If True, convert the values to "scan_indices". Default is False.
- **return_tof_indices** (*bool*) – If True, convert the values to "tof_indices". Default is False.
- **side** (*str*) – If there is an exact match between the values and reference array, which index should be chosen. See also np.searchsorted. Options are "left" or "right". Default is "left".
- **return_type** (*str*) – Alternative way to define the return type. Options are "frame_indices", "scan_indices" or "tof_indices". Default is "".

Returns An array with the same shape as values or iterable or an int which corresponds to the requested value.

Return type np.int64[...], int

Raises *PrecursorFloatError* – When trying to convert a quad float other than np.inf or -np.inf to precursor index.

property dia_mz_cycle

np.float64[:, 2] : The mz_values of a DIA cycle.

Type np.ndarray

property dia_precursor_cycle

np.int64[:] : The precursor indices of a DIA cycle.

Type np.ndarray

property directory

The directory of this TimsTOF object.

Type str

estimate_strike_count(*frame_slices: numpy.ndarray*, *scan_slices: numpy.ndarray*, *precursor_slices: numpy.ndarray*, *tof_slices: numpy.ndarray*, *quad_slices: numpy.ndarray*) → int

Estimate the number of detector events, given a set of slices.

Parameters

- **frame_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(frame_slices[:, :2].ravel()) >= 0) = True).
- **scan_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(scan_slices[:, :2].ravel()) >= 0) = True).
- **precursor_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(precursor_slices[:, :2].ravel()) >= 0) = True).
- **tof_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(tof_slices[:, :2].ravel()) >= 0) = True).

- **quad_slices** (*np.float64[:, 2]*) – Each row of the array is assumed to be (lower_mz, upper_mz) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(quad_slices.ravel()) >= 0) = True).

Returns The estimated number of detector events given these slices.

Return type int

property fragment_frames
The fragment frames table.

Type pd.DataFrame

property frame_max_index
The maximum frame index.

Type int

property frames
The frames table of the analysis.tdf SQL.

Type pd.DataFrame

index_precursors(*centroiding_window: int = 0*, *keep_n_most_abundant_peaks: int = - 1*) → tuple
Retrieve all MS2 spectra acquired with DDA.

IMPORTANT NOTE: This function is intended for DDA samples. While it in theory works for DIA sample too, this probably has little value.

Parameters

- **centroiding_window** (*int*) – The centroiding window to use. If 0, no centroiding is performed. Default is 0.
- **keep_n_most_abundant_peaks** (*int*) – Keep the n most abundant peaks. If -1, all peaks are retained. Default is -1.

Returns The spectrum_indptr array, spectrum_tof_indices array and spectrum_intensity_values array.

Return type tuple (np.int64[:], np.uint32[:], np.uint32[:])

property intensity_corrections
The intensity_correction per frame.

Type np.ndarray

property intensity_max_value
The maximum intensity value.

Type float

property intensity_min_value
The minimum intensity value.

Type float

property intensity_values
np.uint16[:] : The intensity values.

Type np.ndarray

property is_compressed
HDF array is compressed or not.

Type bool

property max_accumulation_time
The maximum accumulation time of all frames.

Type float

property meta_data
The metadata for the acquisition.

Type dict

property mobility_max_value
The maximum mobility value.

Type float

property mobility_min_value
The minimum mobility value.

Type float

property mobility_values
np.float64[:] : The mobility values.

Type np.ndarray

property mz_max_value
The maximum mz value.

Type float

property mz_min_value
The minimum mz value.

Type float

property mz_values
np.float64[:] : The mz values.

Type np.ndarray

property precursor_indices
np.int64[:] : The precursor indices.

Type np.ndarray

property precursor_max_index
The maximum precursor index.

Type int

property precursors
The precursor table.

Type pd.DataFrame

property push_indptr
np.int64[:] : The tof indptr.

Type np.ndarray

property quad_indptr
np.int64[:] : The quad indptr (tof_indices).

Type np.ndarray

property quad_mz_max_value
The maximum quad mz value.

Type float

property quad_mz_min_value
The minimum quad mz value.

Type float

property quad_mz_values
np.float64[:, 2] : The (low, high) quad mz values.

Type np.ndarray

property raw_quad_indptr
np.int64[:] : The raw quad indptr (push indices).

Type np.ndarray

property rt_max_value
The maximum rt value.

Type float

property rt_values
np.float64[:] : The rt values.

Type np.ndarray

property sample_name
The sample name of this TimsTOF object.

Type str

save_as_hdf(*directory: str*, *file_name: str*, *overwrite: bool = False*, *compress: bool = False*, *return_as_bytes_io: bool = False*)
Save the TimsTOF object as an hdf file.

Parameters

- **directory** (*str*) – The directory where to save the HDF file. Ignored if return_as_bytes_io == True.
- **file_name** (*str*) – The file name of the HDF file. Ignored if return_as_bytes_io == True.
- **overwrite** (*bool*) – If True, an existing file is truncated. If False, the existing file is appended to only if the original group, array or property does not exist yet. Default is False.
- **compress** (*bool*) – If True, compression is used. This roughly halves files sizes (on-disk), at the cost of taking 3-6 longer accession times. See also alphatims.utils.create_hdf_group_from_dict. If False, no compression is used Default is False.
- **return_as_bytes_io** – If True, the HDF file is only created in memory and returned as a bytes stream. If False, the file is written to disk. Default is False.

Returns The full file name or a bytes stream containing the HDF file.

Return type str, io.BytesIO

save_as_mgf(*directory: str*, *file_name: str*, *overwrite: bool = False*, *centroiding_window: int = 5*, *keep_n_most_abundant_peaks: int = - 1*)
Save profile spectra from this TimsTOF object as an mgf file.

Parameters

- **directory** (*str*) – The directory where to save the mgf file.

- **file_name** (*str*) – The file name of the mgf file.
- **overwrite** (*bool*) – If True, an existing file is truncated. If False, nothing happens if a file already exists. Default is False.
- **centroiding_window** (*int*) – The centroiding window to use. If 0, no centroiding is performed. Default is 5.
- **keep_n_most_abundant_peaks** (*int*) – Keep the n most abundant peaks. If -1, all peaks are retained. Default is -1.

Returns The full file name of the mgf file.

Return type str

property scan_max_index
The maximum scan index.

Type int

property tof_indices
np.uint32[:] : The tof indices.

Type np.ndarray

property tof_max_index
The maximum tof index.

Type int

use_calibrated_mz_values_as_default(*use_calibrated_mz_values: int*) → None
Override the default mz_values with the global calibrated_mz_values.

Calibrated_mz_values will be calculated if they do not exist yet.

Parameters **use_calibrated_mz_values** (*int*) – If not 0, the mz_values are overwritten with global calibrated_mz_values. If 1, calibration at the MS1 level is performed. If 2, calibration at the MS2 level is performed.

property version
AlphaTims version used to create this TimsTOF object.

Type str

property zeroth_frame
A blank zeroth frame is present so frames are 1-indexed.

Type bool

alphatims.bruker.**add_intensity_to_bin**(*query_index: int*, *intensities: numpy.ndarray*, *parsed_indices: numpy.ndarray*, *intensity_bins: numpy.ndarray*) → None
Add the intensity of a query to the appropriate bin.

IMPORTANT NOTE: This function is decorated with alphatims.utils.pjit. The first argument is thus expected to be provided as an iterable containing ints instead of a single int.

Parameters

- **query_index** (*int*) – The query whose intensity needs to be binned The first argument is thus expected to be provided as an iterable containing ints instead of a single int.
- **intensities** (*np.float64[:]*) – An array with intensities that need to be binned.
- **parsed_indices** (*np.int64[:], np.int64[:, :]*) – Description of parameter *parsed_indices*.

- **intensity_bins** (*np.float64[:]*) – A buffer with intensity bins to which the current query will be added.

alphatims.bruker.**calculate_dia_cycle_mask**(*dia_mz_cycle: numpy.ndarray*, *quad_slices: numpy.ndarray*, *dia_precursor_cycle: numpy.ndarray = None*, *precursor_slices: numpy.ndarray = None*)

Calculate a boolean mask for cyclic push indices satisfying queries.

Parameters

- **dia_mz_cycle** (*np.float64[:, 2]*) – An array with (upper, lower) mz values of a DIA cycle (per push).
- **quad_slices** (*np.float64[:, 2]*) – Each row of the array is assumed to be (lower_mz, upper_mz) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(quad_slices.ravel()) >= 0) = True).
- **dia_precursor_cycle** (*np.int64[:]*) – An array with precursor indices of a DIA cycle (per push).
- **precursor_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(precursor_slices[:, :2].ravel()) >= 0) = True).

Returns A mask that determines if a cyclic push index is valid given the requested slices.

Return type np.bool_[:]

alphatims.bruker.**centroid_spectra**(*index: int*, *spectrum_indptr: numpy.ndarray*, *spectrum_counts: numpy.ndarray*, *spectrum_tof_indices: numpy.ndarray*, *spectrum_intensity_values: numpy.ndarray*, *window_size: int*)

Smoothen and centroid a profile spectrum (inplace operation).

IMPORTANT NOTE: This function will overwrite all input arrays.

IMPORTANT NOTE: This function is decorated with alphatims.utils.pjit. The first argument is thus expected to be provided as an iterable containing ints instead of a single int.

Parameters

- **index** (*int*) – The push index whose intensity_values and tof_indices will be centroided.
- **spectrum_indptr** (*np.int64[:]*) – An index pointer array defining the (untrimmed) spectrum boundaries.
- **spectrum_counts** (*np. int64[:]*) – The original array defining how many distinct tof indices each spectrum has.
- **spectrum_tof_indices** (*np.uint32[:]*) – The original array containing tof indices.
- **spectrum_intensity_values** (*np.float64[:]*) – The original array containing intensity values.
- **window_size** (*int*) – The window size to use for smoothing and centroiding peaks.

alphatims.bruker.**convert_slice_key_to_float_array**(*key*)

Convert a key to a slice float array.

NOTE: This function should only be used for QUAD or DETECTOR dimensions.

Parameters **key** (*slice, int, float, None, iterable*) – The key that needs to be converted.

Returns Each row represent a a (start, stop) slice.

Return type np.float64[:, 2]

Raises **ValueError** – When the key is an np.ndarray with more than 2 columns.

alphatims.bruker.**convert_slice_key_to_int_array**(*data:* alphatims.bruker.TimsTOF, *key*, *dimension: str*)

Convert a key of a data dimension to a slice integer array.

Parameters

- **data** (*alphatims.bruker.TimsTOF*) – The TimsTOF objext for which to get slices.
- **key** (*slice, int, float, None, iterable*) – The key that needs to be converted.
- **dimension** (*str*) – The dimension for which the key needs to be retrieved

Returns Each row represent a a (start, stop, step) slice.

Return type np.int64[:, 3]

Raises

- **ValueError** – When the key contains elements other than int or float.
- ***PrecursorFloatError*** – When trying to convert a quad float to precursor index.

alphatims.bruker.**filter_indices**(*frame_slices: numpy.ndarray*, *scan_slices: numpy.ndarray*, *precursor_slices: numpy.ndarray*, *tof_slices: numpy.ndarray*, *quad_slices: numpy.ndarray*, *intensity_slices: numpy.ndarray*, *frame_max_index: int*, *scan_max_index: int*, *push_indptr: numpy.ndarray*, *precursor_indices: numpy.ndarray*, *quad_mz_values: numpy.ndarray*, *quad_indptr: numpy.ndarray*, *tof_indices: numpy.ndarray*, *intensities: numpy.ndarray*)

Filter raw indices by slices from all dimensions.

Parameters

- **frame_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(frame_slices[:, :2].ravel()) >= 0) = True).
- **scan_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(scan_slices[:, :2].ravel()) >= 0) = True).
- **precursor_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(precursor_slices[:, :2].ravel()) >= 0) = True).
- **tof_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(tof_slices[:, :2].ravel()) >= 0) = True).
- **quad_slices** (*np.float64[:, 2]*) – Each row of the array is assumed to be (lower_mz, upper_mz) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(quad_slices.ravel()) >= 0) = True).
- **intensity_slices** (*np.float64[:, 2]*) – Each row of the array is assumed to be (lower_mz, upper_mz) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(intensity_slices.ravel()) >= 0) = True).
- **frame_max_index** (*int*) – The maximum frame index of a TimsTOF object.
- **scan_max_index** (*int*) – The maximum scan index of a TimsTOF object.
- **push_indptr** (*np.int64[:]*) – The self.push_indptr array of a TimsTOF object.

- **precursor_indices** (*np.int64[:]*) – The self.precursor_indices array of a TimsTOF object.
- **quad_mz_values** (*np.float64[:, 2]*) – The self.quad_mz_values array of a TimsTOF object.
- **quad_indptr** (*np.int64[:]*) – The self.quad_indptr array of a TimsTOF object.
- **tof_indices** (*np.uint32[:]*) – The self.tof_indices array of a TimsTOF object.
- **intensities** (*np.uint16[:]*) – The self.intensity_values array of a TimsTOF object.

Returns The raw indices that satisfy all the slices.

Return type np.int64[:]

alphatims.bruker.**filter_spectra_by_abundant_peaks**(*index: int*, *spectrum_indptr: numpy.ndarray*, *spectrum_counts: numpy.ndarray*, *spectrum_tof_indices: numpy.ndarray*, *spectrum_intensity_values: numpy.ndarray*, *keep_n_most_abundant_peaks: int*)

Filter a spectrum to retain only the most abundant peaks.

IMPORTANT NOTE: This function will overwrite all input arrays.

IMPORTANT NOTE: This function is decorated with alphatims.utils.pjit. The first argument is thus expected to be provided as an iterable containing ints instead of a single int.

Parameters

- **index** (*int*) – The push index whose intensity_values and tof_indices will be centroided.
- **spectrum_indptr** (*np.int64[:]*) – An index pointer array defining the (untrimmed) spectrum boundaries.
- **spectrum_counts** (*np. int64[:]*) – The original array defining how many distinct tof indices each spectrum has.
- **spectrum_tof_indices** (*np.uint32[:]*) – The original array containing tof indices.
- **spectrum_intensity_values** (*np.float64[:]*) – The original array containing intensity values.
- **keep_n_most_abundant_peaks** (*int*) – Keep only this many abundant peaks.

alphatims.bruker.**filter_tof_to_csr**(*tof_slices: numpy.ndarray*, *push_indices: numpy.ndarray*, *tof_indices: numpy.ndarray*, *push_indptr: numpy.ndarray*) → tuple

Get a CSR-matrix with raw indices satisfying push indices and tof slices.

Parameters

- **tof_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(tof_slices[:, :2].ravel()) >= 0) = True).
- **push_indices** (*np.int64[:]*) – The push indices from where to retrieve the TOF slices.
- **tof_indices** (*np.uint32[:]*) – The self.tof_indices array of a TimsTOF object.
- **push_indptr** (*np.int64[:]*) – The self.push_indptr array of a TimsTOF object.

Returns **(np.int64[** – An (indptr, values, columns) tuple, where indptr are push indices, values raw indices, and columns the tof_slices.

Return type], np.int64[:], np.int64[:],)

alphatims.bruker.**get_dia_push_indices**(*frame_slices: numpy.ndarray*, *scan_slices: numpy.ndarray*, *quad_slices: numpy.ndarray*, *scan_max_index: int*, *dia_mz_cycle: numpy.ndarray*, *dia_precursor_cycle: numpy.ndarray = None*, *precursor_slices: numpy.ndarray = None*, *zeroth_frame: bool = True*)

Filter DIA push indices by slices from LC, TIMS and QUAD.

Parameters

- **frame_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(frame_slices[:, :2].ravel()) >= 0) = True).
- **scan_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(scan_slices[:, :2].ravel()) >= 0) = True).
- **quad_slices** (*np.float64[:, 2]*) – Each row of the array is assumed to be (lower_mz, upper_mz) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(quad_slices.ravel()) >= 0) = True).
- **scan_max_index** (*int*) – The maximum scan index of a TimsTOF object.
- **dia_mz_cycle** (*np.float64[:, 2]*) – An array with (upper, lower) mz values of a DIA cycle (per push).
- **dia_precursor_cycle** (*np.int64[:]*) – An array with precursor indices of a DIA cycle (per push).
- **precursor_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(precursor_slices[:, :2].ravel()) >= 0) = True).
- **zeroth_frame** (*bool*) – Indicates if a zeroth frame was used before a DIA cycle.

Returns The raw push indices that satisfy all the slices.

Return type np.int64[:]

alphatims.bruker.**indptr_lookup**(*targets: numpy.ndarray*, *queries: numpy.ndarray*, *momentum_amplifier: int = 2*)

Find the indices of queries in targets.

This function is equivalent to "np.searchsorted(targets, queries, "right") - 1". By utilizing the fact that queries are also sorted, it is significantly faster though.

Parameters

- **targets** (*np.int64[:]*) – A sorted list of index pointers where queries needs to be looked up.
- **queries** (*np.int64[:]*) – A sorted list of queries whose index pointers needs to be looked up.
- **momentum_amplifier** (*int*) – Factor to add momentum to linear searching, attempting to quickly discard empty range without hits. Invreasing it can speed up searching of queries if they are sparsely spread out in targets.

Returns The indices of queries in targets.

Return type np.int64[:]

alphatims.bruker.**init_bruker_dll**(*bruker_dll_file_name: str = '/home/docs/checkouts/readthedocs.org/user_builds/alphatims/checkouts/latest/alphatims/ex*

Open a bruker.dll in Python.

Five functions are defined for this dll:

- tims_open: [c_char_p, c_uint32] -> c_uint64
- tims_close: [c_char_p, c_uint32] -> c_uint64
- tims_read_scans_v2: [c_uint64, c_int64, c_uint32, c_uint32, c_void_p, c_uint32] -> c_uint32
- tims_index_to_mz: [c_uint64, c_int64, POINTER(c_double), POINTER(c_double), c_uint32] -> None
- tims_scannum_to_oneoverk0: Same as "tims_index_to_mz"

Parameters **bruker_dll_file_name** (*str*) – The absolute path to the timsdata.dll. Default is alphatims.utils.BRUKER_DLL_FILE_NAME.

Returns The Bruker dll library.

Return type ctypes.cdll

alphatims.bruker.**open_bruker_d_folder**(*bruker_d_folder_name: str*, *bruker_dll_file_name='/home/docs/checkouts/readthedocs.org/user_builds/alphatims/* → tuple

A context manager for a bruker dll connection to a .d folder.

Parameters

- **bruker_d_folder_name** (*str*) – The name of a Bruker .d folder.
- **bruker_dll_file_name** (*str, ctypes.cdll*) – The path to Bruker' timsdata.dll library. Alternatively, the library itself can be passed as argument. Default is alphatims.utils.BRUKER_DLL_FILE_NAME, which in itself is dependent on the OS.

Returns The opened bruker dll and identifier of the .d folder.

Return type tuple (ctypes.cdll, int).

alphatims.bruker.**parse_decompressed_bruker_binary_type1**(*decompressed_bytes: bytes*, *scan_indices_: numpy.ndarray*, *tof_indices_: numpy.ndarray*, *intensities_: numpy.ndarray*, *scan_start: int*, *scan_index: int*) → int

Parse a Bruker binary scan buffer into tofs and intensities.

Parameters

- **decompressed_bytes** (*bytes*) – A Bruker scan binary buffer that is already decompressed with lzf.
- **scan_indices** (*np.ndarray*) – The **scan_indices_** buffer array.
- **tof_indices** (*np.ndarray*) – The **tof_indices_** buffer array.
- **intensities** (*np.ndarray*) – The **intensities_** buffer array.
- **scan_start** (*int*) – The offset where to start new tof_indices and intensity_values.
- **scan_index** (*int*) – The scan index.

Returns The number of peaks in this scan.

Return type int

alphatims.bruker.**parse_decompressed_bruker_binary_type2**(*decompressed_bytes: bytes*) → tuple
Parse a Bruker binary frame buffer into scans, tofs and intensities.

Parameters **decompressed_bytes** (*bytes*) – A Bruker frame binary buffer that is already decompressed with pyzstd.

Returns The scan_indices, tof_indices and intensities present in this binary array

Return type tuple (np.uint32[:], np.uint32[:], np.uint32[:]).

alphatims.bruker.**parse_keys**(*data:* alphatims.bruker.TimsTOF, *keys*) → dict
Convert different keys to a key dict with defined types.

NOTE: Negative slicing is not supported and all indiviudal keys are assumed to be sorted, disjunct and strictly increasing

Parameters

- **data** (alphatims.bruker.TimsTOF) – The TimsTOF objext for which to get slices.
- **keys** (*tuple*) – A tuple of at most 5 elemens, containing slices, ints, floats, Nones, and/or iterables. See *alphatims.bruker.convert_slice_key_to_int_array* and *alphatims.bruker.convert_slice_key_to_float_array* for more details.

Returns

The resulting dict always has the following items:

- ”frame_indices”: np.int64[:, 3]
- ”scan_indices”: np.int64[:, 3]
- ”tof_indices”: np.int64[:, 3]
- ”precursor_indices”: np.int64[:, 3]
- ”quad_values”: np.float64[:, 2]
- ”intensity_values”: np.float64[:, 2]

Return type dict

alphatims.bruker.**process_frame**(*frame_id: int*, *tdf_bin_file_name: str*, *tims_offset_values: numpy.ndarray*, *scan_indptr: numpy.ndarray*, *intensities: numpy.ndarray*, *tof_indices: numpy.ndarray*, *frame_indptr: numpy.ndarray*, *max_scan_count: int*, *compression_type: int*, *max_peaks_per_scan: int*) → None
Read and parse a frame directly from a Bruker .d.analysis.tdf_bin.

Parameters

- **frame_id** (*int*) – The frame number that should be processed. Note that this is interpreted as 1-indixed instead of 0-indexed, so that it is compatible with Bruker.
- **tdf_bin_file_name** (*str*) – The full file name of the SQL database “analysis.tdf_bin” in a Bruker .d folder.
- **tims_offset_values** (*np.int64[:]*) – The offsets that indicate the starting indices of each frame in the binary. These are contained in the “TimsId” column of the frames table in “analysis.tdf_bin”.
- **scan_indptr** (*np.int64[:]*) – A buffer containing zeros that can store the cumulative number of detections per scan. The size should be equal to max_scan_count * len(frames) + 1. A dummy 0-indexed frame is required to be present for len(frames). The last + 1 allows to explicitly interpret the end of a scan as the start of a subsequent scan.

- **intensities** (*np.uint16[:]*) – A buffer that can store the intensities of all detections. It's size can be determined by summing the "NumPeaks" column from the frames table in "analysis.tdf_bin".
- **tof_indices** (*np.uint32[:]*) – A buffer that can store the tof indices of all detections. It's size can be determined by summing the "NumPeaks" column from the frames table in "analysis.tdf_bin".
- **frame_indptr** (*np.int64[:]*) – The cumulative sum of the number of detections per frame. The size should be equal to len(frames) + 1. A dummy 0-indexed frame is required to be present for len(frames). The last + 1 allows to explicitly interpret the end of a frame as the start of a subsequent frame.
- **max_scan_count** (*int*) – The maximum number of scans a single frame can have.
- **compression_type** (*int*) – The compression type. This must be either 1 or 2. Should be treieved from the global metadata.
- **max_peaks_per_scan** (*int*) – The maximum number of peaks per scan. Should be retrieved from the global metadata.

alphatims.bruker.**read_bruker_binary**(*frames: numpy.ndarray*, *bruker_d_folder_name: str*, *compression_type: int*, *max_peaks_per_scan: int*, *mmap_detector_events: Optional[bool] = None*) → tuple

Read all data from an "analysis.tdf_bin" of a Bruker .d folder.

Parameters

- **frames** (*pd.DataFrame*) – The frames from the "analysis.tdf" SQL database of a Bruker .d folder. These can be acquired with e.g. alphatims.bruker.read_bruker_sql.
- **bruker_d_folder_name** (*str*) – The full path to a Bruker .d folder.
- **compression_type** (*int*) – The compression type. This must be either 1 or 2.
- **max_peaks_per_scan** (*int*) – The maximum number of peaks per scan. Should be treieved from the global metadata.
- **mmap_detector_events** (*bool*) – Do not save the intensity_values and tof_indices in memory, but use an mmap instead. Default is True

Returns The scan_indptr, tof_indices and intensities.

Return type tuple (np.int64[:], np.uint32[:], np.uint16[:]).

alphatims.bruker.**read_bruker_sql**(*bruker_d_folder_name: str*, *add_zeroth_frame: bool = True*, *drop_polarity: bool = True*, *convert_polarity_to_int: bool = True*) → tuple

Read metadata, (fragment) frames and precursors from a Bruker .d folder.

Parameters

- **bruker_d_folder_name** (*str*) – The name of a Bruker .d folder.
- **add_zeroth_frame** (*bool*) – Bruker uses 1-indexing for frames. If True, a zeroth frame is added without any TOF detections to make Python simulate this 1-indexing. If False, frames are 0-indexed. Default is True.
- **drop_polarity** (*bool*) – The polarity column of the frames table contains "+" or "-" and is not numerical. If True, the polarity column is dropped from the frames table. this ensures a fully numerical pd.DataFrame. If False, this column is kept, resulting in a pd.DataFrame with dtype=object. Default is True.

- **convert_polarity_to_int** (*bool*) – Convert the polarity to int (-1 or +1). This allows to keep it in numerical form. This is ignored if the polarity is dropped. Default is True.

Returns (str, dict, pd.DataFrame, pd.DataFrame, pd.DataFrame). The acquisition_mode, global_meta_data, frames, fragment_frames and precursors. For diaPASEF, precursors is None.

Return type tuple

alphatims.bruker.**set_precursor**(*precursor_index: int*, *offset_order: numpy.ndarray*, *precursor_offsets: numpy.ndarray*, *quad_indptr: numpy.ndarray*, *tof_indices: numpy.ndarray*, *intensities: numpy.ndarray*, *spectrum_tof_indices: numpy.ndarray*, *spectrum_intensity_values: numpy.ndarray*, *spectrum_indptr: numpy.ndarray*, *spectrum_counts: numpy.ndarray*) → None

Sum the intensities of all pushes belonging to a single precursor.

IMPORTANT NOTE: This function is decorated with alphatims.utils.pjit. The first argument is thus expected to be provided as an iterable containing ints instead of a single int.

Parameters

- **precursor_index** (*int*) – The precursor index indicating which MS2 spectrum to determine.
- **offset_order** (*np.int64[:]*) – The order of self.precursor_indices, obtained with np.argsort.
- **precursor_offsets** (*np.int64[:]*) – An index pointer array for precursor offsets.
- **quad_indptr** (*np.int64[:]*) – The self.quad_indptr array of a TimsTOF object.
- **tof_indices** (*np.uint32[:]*) – The self.tof_indices array of a TimsTOF object.
- **intensities** (*np.uint16[:]*) – The self.intensity_values array of a TimsTOF object.
- **spectrum_tof_indices** (*np.uint32[:]*) – A buffer array to store tof indices of the new spectrum.
- **spectrum_intensity_values** (*np.float64[:]*) – A buffer array to store intensity values of the new spectrum.
- **spectrum_indptr** (*np.int64[:]*) – An index pointer array defining the original spectrum boundaries.
- **spectrum_counts** (*np. int64[:]*) – An buffer array defining how many distinct tof indices the new spectrum has.

alphatims.bruker.**trim_spectra**(*index: int*, *spectrum_tof_indices: numpy.ndarray*, *spectrum_intensity_values: numpy.ndarray*, *spectrum_indptr: numpy.ndarray*, *trimmed_spectrum_tof_indices: numpy.ndarray*, *trimmed_spectrum_intensity_values: numpy.ndarray*, *new_spectrum_indptr: numpy.ndarray*) → None

Trim remaining bytes after merging of multiple pushes.

IMPORTANT NOTE: This function is decorated with alphatims.utils.pjit. The first argument is thus expected to be provided as an iterable containing ints instead of a single int.

Parameters

- **index** (*int*) – The push index whose intensity_values and tof_indices will be trimmed.
- **spectrum_tof_indices** (*np.uint32[:]*) – The original array containing tof indices.
- **spectrum_intensity_values** (*np.float64[:]*) – The original array containing intensity values.

- **spectrum_indptr** (*np.int64[:]*) – An index pointer array defining the original spectrum boundaries.
- **trimmed_spectrum_tof_indices** (*np.uint32[:]*) – A buffer array to store new tof indices.
- **trimmed_spectrum_intensity_values** (*np.float64[:]*) – A buffer array to store new intensity values.
- **new_spectrum_indptr** (*np.int64[:]*) – An index pointer array defining the trimmed spectrum boundaries.

alphatims.bruker.**valid_precursor_index**(*precursor_index: int*, *precursor_slices: numpy.ndarray*) → bool

Check if a precursor index is included in the slices.

Parameters

- **precursor_index** (*int*) – The precursor index to validate.
- **precursor_slices** (*np.int64[:, 3]*) – Each row of the array is assumed to be a (start, stop, step) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(precursor_slices[:, :2].ravel()) >= 0) = True).

Returns True if the precursor index is present in any of the slices. False otherwise.

Return type bool

alphatims.bruker.**valid_quad_mz_values**(*low_mz_value: float*, *high_mz_value: float*, *quad_slices: numpy.ndarray*) → bool

Check if the low and high quad mz values are included in the slices.

NOTE: Just a part of the quad range needs to overlap with a part of a single slice.

Parameters

- **low_mz_value** (*float*) – The lower mz value of the current quad selection.
- **high_mz_value** (*float*) – The upper mz value of the current quad selection.
- **quad_slices** (*np.float64[:, 2]*) – Each row of the array is assumed to be (lower_mz, upper_mz) tuple. This array is assumed to be sorted, disjunct and strictly increasing (i.e. np.all(np.diff(quad_slices.ravel()) >= 0) = True).

Returns True if some part of the quad overlaps with some part of some slice. False if there is no overlap in the range.

Return type bool

CHAPTER

THREE

ALPHATIMS.PLOTTING

This module provides basic LC-TIMS-Q-TOF plots.

Functions:

heatmap(df, x_axis_label, y_axis_label[, ...])	Create a scatterplot / heatmap for a dataframe.
line_plot(timstof_data, selected_indices, ...)	Plot an XIC, mobilogram or spectrum as a lineplot.
tic_plot(timstof_data[, title, width, height])	Create a total ion chromatogram (TIC) for the data.

alphatims.plotting.**heatmap**(*df*, *x_axis_label: str*, *y_axis_label: str*, *title: str = ''*, *z_axis_label: str = 'intensity'*, *width: int = 1000*, *height: int = 320*, *rescale_to_minutes: bool = True*, ***kwargs*)

Create a scatterplot / heatmap for a dataframe.

The coordinates of the dataframe are projected (i.e. their intensities are summed) on the requested axes.

Parameters

- **df** (*pd.DataFrame*) – A dataframe with coordinates. This should be obtained by slicing an alphatims.bruker.TimsTOF object.
- **x_axis_label** (*str*) – A label that is used for projection (i.e. intensities are summed) on the x-axis. Options are:
  - mz
  - rt
  - mobility
- **y_axis_label** (*str*) – A label that is used for projection (i.e. intensities are summed) on the y-axis. Options are:
  - mz
  - rt
  - mobility
- **title** (*str*) – The title of this plot. Will be prepended with "Heatmap". Default is "".
- **z_axis_label** (*str*) – Should not be set for a 2D scatterplot / heatmap. Default is "intensity".
- **width** (*int*) – The width of this plot. Default is 1000.
- **height** (*int*) – The height of this plot. Default is 320.
- **rescale_to_minutes** (*bool*) – If True, the rt_values of the dataframe will be divided by 60. WARNING: this updates the dataframe directly and is persistent! Default is True.

- ****kwargs** – Additional keyword arguments that will be passed to hv.Scatter.

Returns A scatter plot projected on the 2 dimensions.

Return type hv.Scatter

alphatims.plotting.**line_plot**(*timstof_data*, *selected_indices*, *x_axis_label: str*, *title: str = ''*, *y_axis_label: str = 'intensity'*, *remove_zeros: bool = False*, *trim: bool = True*, *width: int = 1000*, *height: int = 320*, ***kwargs*)

Plot an XIC, mobilogram or spectrum as a lineplot.

Parameters

- **timstof_data** (alphatims.bruker.TimsTOF) – An alphatims.bruker.TimsTOF data object.
- **selected_indices** (*np.int64[:]*) – The raw indices that are selected for this plot. These are typically obtained by slicing the TimsTOF data object with e.g. data[..., "raw"].
- **x_axis_label** (*str*) – A label that is used for projection (i.e. intensities are summed) on the x-axis. Options are:
  - rt
  - mobility
  - mz
- **title** (*str*) – The title of this plot. Will be prepended with "Spectrum", "Mobilogram" or "XIC". Default is "".
- **y_axis_label** (*str*) – Should not be set for a 1D line plot. Default is "intensity".
- **remove_zeros** (*bool*) – If True, zeros are removed. Note that a line plot connects consecutive points, which can lead to misleading plots if non-zeros are removed. If False, use the full range of the appropriate dimension of the timstof_data. Default is False.
- **trim** (*bool*) – If True, zeros on the left and right are trimmed. Default is True.
- **width** (*int*) – The width of this plot. Default is 1000.
- **height** (*int*) – The height of this plot. Default is 320.
- ****kwargs** – Additional keyword arguments that will be passed to hv.Curve.

Returns A curve plot that represents an XIC, mobilogram or spectrum.

Return type hv.Curve

alphatims.plotting.**tic_plot**(*timstof_data*, *title: str = ''*, *width: int = 1000*, *height: int = 320*, ***kwargs*)

Create a total ion chromatogram (TIC) for the data.

Parameters

- **timstof_data** (alphatims.bruker.TimsTOF) – An alphatims.bruker.TimsTOF data object.
- **title** (*str*) – The title of this plot. Will be prepended with "TIC". Default is False
- **width** (*int*) – The width of this plot. Default is 1000.
- **height** (*int*) – The height of this plot. Default is 320.
- ****kwargs** – Additional keyword arguments that will be passed to hv.Curve.

Returns The TIC of the provided dataset.

Return type hv.Curve

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

Z