15 Indices and tables 61

Python Module Index 63

Index 65
karabo_data is a Python library for accessing and working with data produced at European XFEL. To install it on the Maxwell cluster:

```
module load anaconda/3
pip install --user karabo_data
```

If this causes problems for Jupyter, you may need to upgrade ipykernel to fix them:

```
pip install --user --upgrade ipykernel
```

Contents:
CHAPTER ONE

READING DATA FILES

Data is available in *trains*, which arrive ten times per second. Each train has a unique train ID, which is used to match up corresponding data stored separately. Many data sources will make one reading per train, but some, including the main detectors, record data more frequently.

```python
karabo_data.RunDirectory(path)
Open data files from a ‘run’ at European XFEL.
A ‘run’ is a directory containing a number of HDF5 files with data from the same time period.
Returns a DataCollection object.
```

- **Parameters**
  - `path (str)` – Path to the run directory containing HDF5 files.

```python
karabo_data.open_run(proposal, run, data='raw')
Access EuXFEL data on the Maxwell cluster by proposal and run number.
Returns a DataCollection object.
```

- **Parameters**
  - `proposal (str, int)` – A proposal number, such as 2012, ‘2012’, ‘p002012’, or a path such as ‘/gpfs/exfel/exp/SPB/201701/p002012’.
  - `run (str, int)` – A run number such as 243, ‘243’ or ‘r0243’.
  - `data (str)` – ‘raw’ or ‘proc’ (processed) to access data from one of those folders. The default is ‘raw’.

New in version 0.5.

```python
karabo_data.H5File(path)
Open a single HDF5 file generated at European XFEL.
Returns a DataCollection object.
```

- **Parameters**
  - `path (str)` – Path to the HDF5 file

```python
class karabo_data.DataCollection(files, selection=None, train_ids=None, ctx_closes=False)
An assemblage of data generated at European XFEL.
Data consists of *sources* which each have *keys*. It is further organised by *trains*, which are identified by train IDs.
You normally get an instance of this class by calling H5File() for a single file or RunDirectory() for a directory.
```

- **train_ids**
  - A list of the train IDs for which there is any data in this run. The data recorded may not be the same for each train.
control_sources
A set of the control source names in this data, in the format "SA3_XTD10_VAC/TSENS/S30100K".

instrument_sources
A set of the instrument source names in this data, in the format "FXE_DET_LPD1M-1/DET/15CHO:xtdf".

all_sources
A set of names for both instrument and control sources. This is the union of the two sets above.

keys_for_source(source)
Get a set of key names for the given source

If you have used select() to filter keys, only selected keys are returned.

Only one file is used to find the keys. Within a run, all files should have the same keys for a given source, but if you use union() to combine two runs where the source was configured differently, the result can be unpredictable.

info()
Show information about the selected data.

trains(devices=None, train_range=None, *, require_all=False)
Iterate over all trains in the data and gather all sources.

```python
run = Run('/path/to/my/run/r0123')
for train_id, data in run.trains():
    value = data['device']['parameter']
```

Parameters

- **devices**(dict or list, optional) – Filter data by sources and keys. Refer to select() for how to use this.

- **train_range**(by_id or by_index object, optional) – Iterate over only selected trains, by train ID or by index. Refer to select_trains() for how to use this.

- **require_all**(bool) – False (default) returns any data available for the requested trains. True skips trains which don’t have all the selected data; this only makes sense if you make a selection with devices or select().

Yields

- **tid**(int) – The train ID of the returned train

- **data**(dict) – The data for this train, keyed by device name

train_from_id(train_id, devices=None)
Get Train data for specified train ID.

Parameters

- **train_id**(int) – The train ID

- **devices**(dict or list, optional) – Filter data by sources and keys. Refer to select() for how to use this.

Returns

- **tid**(int) – The train ID of the returned train

- **data**(dict) – The data for this train, keyed by device name
European XFEL Python data tools Documentation, Release 0.4.0

Raises **KeyError** – if `train_id` is not found in the run.

`train_from_index(train_index, devices=None)`
Get train data of the nth train in this data.

**Parameters**
- `train_index (int)` – Index of the train in the file.
- `devices (dict or list, optional)` – Filter data by sources and keys. Refer to `select()` for how to use this.

**Returns**
- `tid (int)` – The train ID of the returned train
- `data (dict)` – The data for this train, keyed by device name

`get_dataframe(fields=None, *, timestamps=False)`
Return a pandas dataframe for given data fields.

**Parameters**
- `fields (dict or list, optional)` – Select data sources and keys to include in the dataframe. Selections are defined by lists or dicts as in `select()`.
- `timestamps (bool)` – If false (the default), exclude the timestamps associated with each control data field.

`get_series(source, key)`
Return a pandas Series for a particular data field.

**Parameters**
- `source (str)` – Device name with optional output channel, e.g. “SA1_XTD2_XGM/DOOCS/MAIN” or “SPB_DET_AGIPD1M-1/DET/7CH0:xtdf”
- `key (str)` – Key of parameter within that device, e.g. “beamPosition.iyPos.value” or “header.linkId”. The data must be 1D in the file.

`get_array(source, key, extra_dims=None, roi=by_index[...])`
Return a labelled array for a particular data field.

The first axis of the returned data will be the train IDs.

**Parameters**
- `source (str)` – Device name with optional output channel, e.g. “SA1_XTD2_XGM/DOOCS/MAIN” or “SPB_DET_AGIPD1M-1/DET/7CH0:xtdf”
- `key (str)` – Key of parameter within that device, e.g. “beamPosition.iyPos.value” or “header.linkId”.
- `extra_dims (list of str)` – Name extra dimensions in the array. The first dimension is automatically called ‘train’. The default for extra dimensions is dim_0, dim_1, ...
- `roi (by_index)` – The region of interest. This expression selects data in all dimensions apart from the first (trains) dimension. If the data holds a 1D array for each entry, roi=by_index[:8] would get the first 8 values from every train. If the data is 2D or more at each entry, selection looks like roi=by_index[:,8, 5:10] .

`select(seln_or_source_glob, key_glob='*')`
Select a subset of sources and keys from this data.

There are three possible ways to select data:
1. With two glob patterns (see below) for source and key names:

```python
# Select data in the image group for any detector sources
sel = run.select('*/DET/*, 'image.*')
```

2. With a list of (source, key) glob patterns:

```python
# Select image.data and image.mask for any detector sources
sel = run.select([('*/DET/*, 'image.data'), ('*/DET/*, 'image.mask')])
```

Data is included if it matches any of the pattern pairs.

3. With a dict of source names mapped to sets of key names (or empty sets to get all keys):

```python
# Select image.data from one detector source, and all data from one XGM
sel = run.select({'SPB_DET_AGIPD1M-1/DET/0CH0:xtdf': {'image.data'},
                  'SA1_XTD2_XGM/XGM/DOOCS': set()})
```

Unlike the others, this option _doesn’t_ allow glob patterns. It’s a more precise but less convenient option for code that knows exactly what sources and keys it needs.

Returns a new `DataCollection` object for the selected data.

**Note:** ‘Glob’ patterns may be familiar from selecting files in a Unix shell. `*` matches anything, so `*/DET/*` selects sources with ‘/DET/’ anywhere in the name. There are several kinds of wildcard:

- `*`: anything
- `?`: any single character
- `[xyz]`: one character, “x”, “y” or “z”
- `[0-9]`: one digit character
- `[!xyz]`: one character, _not_ x, y or z

Anything else in the pattern must match exactly. It’s case-sensitive, so “x” does not match “X”.

```python
deselect (seln_or_source_glob, key_glob='*')
```

Select everything except the specified sources and keys.

This takes the same arguments as `select()`, but the sources and keys you specify are dropped from the selection.

Returns a new `DataCollection` object for the remaining data.

```python
select_trains (train_range)
```

Select a subset of trains from this data.

Choose a slice of trains by train ID:

```python
from karabo_data import by_id
sel = run.select_trains(by_id[142844490:142844495])
```

Or select a list of trains:

```python
sel = run.select_trains(by_id[[142844490, 142844493, 142844494]])
```

Or select trains by index within this collection:
from karabo_data import by_index
sel = run.select_trains(by_index[:5])

Returns a new DataCollection object for the selected trains.

Raises ValueError – If given train IDs do not overlap with the trains in this data.

def union(*others)
    Join the data in this collection with one or more others.
    This can be used to join multiple sources for the same trains, or to extend the same sources with data for
    further trains. The order of the datasets doesn’t matter.
    Returns a new DataCollection object.

def write(filename)
    Write the selected data to a new HDF5 file
    You can choose a subset of the data using methods like select() and select_trains(), then use
    this write it to a new, smaller file.
    The target filename will be overwritten if it already exists.

def write_virtual(filename)
    Write an HDF5 file with virtual datasets for the selected data.
    This doesn’t copy the data, but each virtual dataset provides a view of data spanning multiple sequence
    files, which can be accessed as if it had been copied into one big file.
    This is not the same as building virtual datasets to combine multi-module detector data. That more complex
    operation will be integrated into karabo_data in the future.
    Creating and reading virtual datasets requires HDF5 version 1.10.
    The target filename will be overwritten if it already exists.

1.1 Missing data

What happens if some data was not recorded for a given train?

Control data is duplicated for each train until it changes. If the device cannot send changes, the last values will be
recorded for each subsequent train until it sends changes again. There is no general way to distinguish this scenario
from values which genuinely aren’t changing.

Parts of instrument data may be missing from the file. These will also be missing from the data returned by
karabo_data:

- The train-oriented methods trains(), train_from_id(), and train_from_index() give you dic-
  tionaries keyed by source and key name. Sources and keys are only included if they have data for that train.
- get_array(), and get_series() skip over trains which are missing data. The indexes on the returned
  DataArray or Series objects link the returned data to train IDs. Further operations with xarray or pandas may
  drop misaligned data or introduce fill values.
- get_dataframe() includes rows for which any column has data. Where some but not all columns have
data, the missing values are filled with NaN by pandas’ missing data handling.

Missing data does not necessarily mean that something has gone wrong: some devices send data at less than 10 Hz
(the train rate), so they always have gaps between updates.
1.2 Data problems

If you encounter problems accessing data with karabo_data, there may be problems with the data files themselves. Use the karabo-data-validate command to check for this (see Checking data files).

Here are some problems we’ve seen, and possible solutions or workarounds:

- Indexes point to data beyond the end of datasets: this has previously been caused by bugs in the detector calibration pipeline. If you see this in calibrated data (in the proc/ folder), ask for the relevant runs to be re-calibrated.

- Train IDs are not strictly increasing: issues with the timing system when the data is recorded can create an occasional train ID which is completely out of sequence. Usually it seems to be possible to ignore this and use the remaining data, but if you have any issues, please let us know.
  - In one case, a train ID had the maximum possible value ($2^{64} - 1$), causing info() to fail. You can select everything except this train using select_trains():

    ```python
    from karabo_data import by_id
    sel = run.select_trains(by_id[:2**64-1])
    ```

If you’re having problems with karabo_data, you can also try searching previously reported issues to see if anyone has encountered similar symptoms.
AGIPD & LPD DATA

These data from AGIPD and LPD is spread out in separate files. karabo_data includes convenient interfaces to access this data, pulling together the separate modules into a single array.

class karabo_data.components.AGIPD1M(data: karabo_data.reader.DataCollection, detector_name=None, modules=None, *, min_modules=1)

An interface to AGIPD-1M data.

Parameters

- **data** *(DataCollection)* – A data collection, e.g. from RunDirectory.
- **modules** *(set of ints, optional)* – Detector module numbers to use. By default, all available modules are used.
- **detector_name** *(str, optional)* – Name of a detector, e.g. ‘SPB_DET_AGIPD1M-1’. This is only needed if the dataset includes more than one AGIPD detector.
- **min_modules** *(int)* – Include trains where at least n modules have data. Default is 1.

The methods of this class are identical to those of LPD1M, below.

class karabo_data.components.LPD1M(data: karabo_data.reader.DataCollection, detector_name=None, modules=None, *, min_modules=1)

An interface to LPD-1M data.

Parameters

- **data** *(DataCollection)* – A data collection, e.g. from RunDirectory.
- **modules** *(set of ints, optional)* – Detector module numbers to use. By default, all available modules are used.
- **detector_name** *(str, optional)* – Name of a detector, e.g. ‘FXE_DET_LPD1M-1’. This is only needed if the dataset includes more than one LPD detector.
- **min_modules** *(int)* – Include trains where at least n modules have data. Default is 1.

get_array *(key, pulses=by_index[:]*)

Get a labelled array of detector data

Parameters

- **key** *(str)* – The data to get, e.g. ‘image.data’ for pixel values.
- **pulses** *(by_id or by_index)* – Select the pulses to include from each train. by_id selects by pulse ID, by_index by index within the data being read. The default includes all pulses. Only used for per-train data.
trains (pulses=by_index[:])
  Iterate over trains for detector data.

  Parameters pulses (by_index or by_id) – Select which pulses to include for each
  train. The default is to include all pulses.

  Yields train_data (dict) – A dictionary mapping key names (e.g. image.data) to labelled
  arrays.

write_virtual_cxi (filename)
  Write a virtual CXI file to access the detector data.

  The virtual datasets in the file provide a view of the detector data as if it was a single huge array, but
  without copying the data. Creating and using virtual datasets requires HDF5 1.10.

  Parameters filename (str) – The file to be written. Will be overwritten if it already exists.

See also:
Accessing LPD data: An example using the class above.

If you get data for a train from the main DataCollection interface, there is also another way to combine detector
modules from AGIPD or LPD:

karabo_data.reader.stack_detector_data (train, data, axis=-3, modules=16, only='', xcept=(),
fillvalue=nan)

  Stack data from detector modules in a train.

  Parameters
  • train (dict) – Train data.
  • data (str) – The path to the device parameter of the data you want to stack, e.g. ‘image.data’.
  • axis (int) – Array axis on which you wish to stack (default is -3).
  • modules (int) – Number of modules composing a detector (default is 16).
  • only (str) – Deprecated: Only use devices in train containing this substring.
  • xcept (list) – Deprecated: list of devices to ignore, if you have recorded slow data with
detector data in the same run).
  • fillvalue (number) – Value to use in place of data for missing modules. The default is
  nan (not a number) for floating-point data, and 0 for integers.

  Returns combined – Stacked data for requested data path.

  Return type numpy.array
CHAPTER THREE

STREAMING DATA OVER ZEROMQ

Karabo Bridge provides access to live data during the experiment over a ZeroMQ socket. The karabo_data Python package can stream data from files using the same protocol. You can use this to test code which expects to receive data from Karabo Bridge, or use the same code for analysing live data and stored data.

To stream the data from a file or run unmodified, use the command:

```
karabo-bridge-serve-files /gpfs/exfel/exp/SPB/201830/p900022/raw/r0034 4545
```

The number (4545) must be an unused TCP port above 1024. It will bind to this and stream the data to any connected clients.

We provide Karabo bridge clients as Python and C++ libraries.

If you want to do some processing on the data before streaming it, you can use this Python interface to send it out:

```python
class karabo_data.ZMQStreamer(port, maxlen=10, protocol_version='2.2', dummy_timestamps=False)
```

ZeroMQ interface sending data over a TCP socket.

```
# Server:
serve = ZMQStreamer(1234)
serve.start()

for tid, data in run.trains():
    result = important_processing(data)
    serve.feed(result)

# Client:
from karabo_bridge import Client
client = Client('tcp://server.hostname:1234')
data = client.next()
```

Parameters

- **port** (int) – Local TCP port to bind socket to
- **maxlen** (int, optional) – How many trains to cache before sending (default: 10)
- **protocol_version** ("1.0" / "2.1") – Which version of the bridge protocol to use. Defaults to the latest version implemented.
- **dummy_timestamps** (bool) – Some tools (such as OnDA) expect the timestamp information to be in the messages. We can’t give accurate timestamps where these are not in the file, so this option generates fake timestamps from the time the data is fed in.
start()  
Start a zmq.REP socket.

feed(data, metadata=None)  
Push data to the sending queue.

This blocks if the queue already has maxlen items waiting to be sent.

Parameters

- **data** (dict) – Contains train data. The dictionary has to follow the karabo_bridge protocol structure:
  - keys are source names
  - values are dict, where the keys are the parameter names and values must be python built-in types or numpy.ndarray.

- **metadata** (dict, optional) – Contains train metadata. The dictionary has to follow the karabo_bridge protocol structure:
  - keys are (str) source names
  - values (dict) should contain the following items:
    * 'timestamp' Unix time with subsecond resolution
    * 'timestamp.sec' Unix time with second resolution
    * 'timestamp.frac' fractional part with attosecond resolution
    * 'timestamp.tid' is European XFEL train unique ID

```json
{
    'source': 'sourceName'  # str
    'timestamp': 1234.567890  # float
    'timestamp.sec': '1234'  # str
    'timestamp.frac': '567890000000000000'  # str
    'timestamp.tid': 1234567890  # int
}
```

If the metadata dict is not provided it will be extracted from ‘data’ or an empty dict if ‘metadata’ key is missing from a data source.
CHAPTER
FOUR

CHECKING DATA FILES

`karabo_data` includes a tool to check the integrity of data files. You can pass it a run:

```
karabo-data-validate /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0803
```

Or a single data file:

```
karabo-data-validate /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0803/RAW-R0803-AGIPD00-S00000.h5
```

The checks are informed by problems we have encountered with data files in the past. Currently, it checks that:

- All .h5 files in a run can be opened, and the run contains at least one usable file.
- The list of train IDs in a file has no zeros except for padding at the end.
- Each train ID in a file is greater than the one before it.
- The indexes do not point to data beyond the end of a dataset.
- The indexes point to the start of the dataset, and then to successive chunks for successive trains, without gaps or overlaps between them.

If any checks fail, the output will contain details, and the exit code will be non-zero. An exit code of 0 means that the checks all passed. This is the standard convention for command line tools to indicate success or failure.
AGIPD & LPD GEOMETRY

The AGIPD and LPD detectors are made up of several sensor modules, from which separate streams of data are recorded. Inspecting or processing data from these detectors therefore depends on knowing how the modules are arranged. The module `karabo_data.geometry2` handles this information.

All the coordinates used in this module are from the detector centre. This should be roughly where the beam passes through the detector. They follow the standard European XFEL axis orientations, with x increasing to the left (looking along the beam), and y increasing upwards.

**Note:** This module includes methods to assemble data into a single array. This is sufficient for a quick examination of detector images, but the detector pixels may not line up with the grid imposed by a single array. For accurate analysis, it’s best to use a tool that can process geometry internally with sub-pixel precision.

### 5.1 AGIPD-1M

AGIPD-1M consists of 16 modules of 512×128 pixels each. Each module is further subdivided into 8 tiles. The layout of tiles within a module is fixed by the manufacturing process, but this geometry code works with a position for each tile.

```python
class karabo_data.geometry2.AGIPD_1MGeometry(modules, filename='No file')
```

Detector layout for AGIPD-1M

The coordinates used in this class are 3D (x, y, z), and represent multiples of the pixel size.

You won’t normally instantiate this class directly; use one of the constructor class methods to create or load a geometry.

**classmethod from_quad_positions**(quad_pos,asic_gap=2,panel_gap=29,unit=0.0002)

Generate an AGIPD-1M geometry from quadrant positions.

This produces an idealised geometry, assuming all modules are perfectly flat, aligned and equally spaced within their quadrant.

The quadrant positions are given in pixel units, referring to the first pixel of the first module in each quadrant, corresponding to data channels 0, 4, 8 and 12.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

To give positions in units other than pixels, pass the `unit` parameter as the length of the unit in metres. E.g. `unit=1e-3` means the coordinates are in millimetres.

**classmethod from_crystfel_geom**(filename)

Read a CrystFEL format (.geom) geometry file.
Fig. 1: The approximate layout of AGIPD-1M, in a front view (looking along the beam).
Returns a new geometry object.

write_crystfel_geom(filename)
Write this geometry to a CrystFEL format (.geom) geometry file.

to_distortion_array()
Return distortion matrix for AGIPD detector, suitable for pyFAI.

Returns
out – Array of float 32 with shape (8192, 128, 4, 3). The dimensions mean:
• 8192 = 16 modules * 512 pixels (slow scan axis)
• 128 pixels (fast scan axis)
• 4 corners of each pixel
• 3 numbers for z, y, x

Return type ndarray

plot_data_fast(data, axis_units='px', frontview=True)
Plot data from the detector using this geometry.
This approximates the geometry to align all pixels to a 2D grid.
Returns a matplotlib axes object.

Parameters
• data (ndarray) – Should have exactly 3 dimensions, for the modules, then the slow scan and fast scan pixel dimensions.
• axis_units (str) – Show the detector scale in pixels (‘px’) or metres (‘m’).
• frontview (bool) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a ‘looking into the beam’ view.

position_modules_fast(data)
Assemble data from this detector according to where the pixels are.
This approximates the geometry to align all pixels to a 2D grid.

Parameters data (ndarray) – The last three dimensions should match the modules, then the slow scan and fast scan pixel dimensions.

Returns
• out (ndarray) – Array with one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.
• centre (ndarray) – (y, x) pixel location of the detector centre in this geometry.

position_modules_interpolate(data)
Assemble data from this detector according to where the pixels are.
This performs interpolation, which is very slow. Use position_modules_fast() to get a pixel-aligned approximation of the geometry.

Parameters data (ndarray) – The three dimensions should be channelno, pixel_ss, pixel_fs (lengths 16, 512, 128). ss/fs are slow-scan and fast-scan.

Returns
• out (ndarray) – Array with the one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.
• `centre (ndarray)` – (y, x) pixel location of the detector centre in this geometry.

`inspect (frontview=True)`
Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

**Parameters**

- `frontview (bool)` – If True (the default), x increases to the left, as if you were looking along the beam. False gives a ‘looking into the beam’ view.

`compare (other, scale=1.0)`
Show a comparison of this geometry with another in a 2D plot.

This shows the current geometry like `inspect ()`, with the addition of arrows showing how each panel is shifted in the other geometry.

**Parameters**

- `other (AGIPD_1MGeometry)` – A second geometry object to compare with this one.
- `scale (float)` – Scale the arrows showing the difference in positions. This is useful to show small differences clearly.

### 5.2 LPD-1M

LPD-1M consists of 16 supermodules of 256×256 pixels each. Each supermodule is further subdivided into 16 sensor tiles, which this geometry code can position independently.

```python
class karabo_data.geometry2.LPD_1MGeometry (modules, filename='No file')
```

Detector layout for LPD-1M

The coordinates used in this class are 3D (x, y, z), and represent multiples of the pixel size.

You won’t normally instantiate this class directly: use one of the constructor class methods to create or load a geometry.

```python
classmethod from_quad_positions (quad_pos, *, unit=0.001,asic_gap=None,panel_gap=None)
```

Generate an LPD-1M geometry from quadrant positions.

This produces an idealised geometry, assuming all modules are perfectly flat, aligned and equally spaced within their quadrant.

The quadrant positions refer to the corner of each quadrant where module 4, tile 16 is positioned. This is the corner of the last pixel as the data is stored. In the initial detector layout, the corner positions are for the top left corner of the quadrant, looking along the beam.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

**Parameters**

- `quad_pos (list of 2-tuples)` – (x, y) coordinates of the last corner (the one by module 4) of each quadrant.
- `unit (float, optional)` – The conversion factor to put the coordinates into metres. The default 1e-3 means the numbers are in millimetres.
- `asic_gap (float, optional)` – The gap between adjacent tiles/ASICS. The default is 4 pixels.
Fig. 2: The approximate layout of LPD-1M, in a front view (looking along the beam).
• **panel_gap** (*float, optional*) – The gap between adjacent modules/panels. The default is 4 pixels.

**classmethod from_h5_file_and_quad_positions** *(path, positions, unit=0.001)*

Load an LPD-1M geometry from an XFEL HDF5 format geometry file.

The quadrant positions are not stored in the file, and must be provided separately. By default, both the quadrant positions and the positions in the file are measured in millimetres; the unit parameter controls this.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

This version of the code only handles x and y translation, as this is all that is recorded in the initial LPD geometry file.

**Parameters**

• **path** (*str*) – Path of an EuXFEL format (HDF5) geometry file for LPD.

• **positions** (*list of 2-tuples*) – (x, y) coordinates of the last corner (the one by module 4) of each quadrant.

• **unit** (*float, optional*) – The conversion factor to put the coordinates into metres. The default 1e-3 means the numbers are in millimetres.

**plot_data_fast** *(data, axis_units='px', frontview=True)*

Plot data from the detector using this geometry. This approximates the geometry to align all pixels to a 2D grid.

Returns a matplotlib axes object.

**Parameters**

• **data** (*ndarray*) – Should have exactly 3 dimensions, for the modules, then the slow scan and fast scan pixel dimensions.

• **axis_units** (*str*) – Show the detector scale in pixels (‘px’) or metres (‘m’).

• **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a ‘looking into the beam’ view.

**position_modules_fast** *(data)*

Assemble data from this detector according to where the pixels are.

This approximates the geometry to align all pixels to a 2D grid.

**Parameters**

• **data** (*ndarray*) – The last three dimensions should match the modules, then the slow scan and fast scan pixel dimensions.

**Returns**

• **out** (*ndarray*) – Array with one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.

• **centre** (*ndarray*) – (y, x) pixel location of the detector centre in this geometry.

**inspect** *(frontview=True)*

Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

**Parameters**

• **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a ‘looking into the beam’ view.
6.1 lsxfel

Examine the contents of an EuXFEL proposal directory, run directory, or HDF5 file:

```
# Proposal directory
lsxfel /gpfs/exfel/exp/XMPL/201750/p700000

# Run directory
lsxfel /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002

# Single file
lsxfel /gpfs/exfel/exp/XMPL/201750/p700000/proc/r0002/CORR-R0034-AGIPD00-S00000.h5
```

6.2 karabo-data-validate

Check the structure of an EuXFEL run or HDF5 file:

```
karabo-data-validate /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002
```

If it finds problems with the data, the program will produce a list of them and exit with status 1.

6.3 karabo-bridge-serve-files

Stream data from files in the Karabo bridge format. See Streaming data over ZeroMQ for more information.

6.4 karabo-data-make-virtual-cxi

Make a virtual CXI file to access AGIPD/LPD detector data from a specified run:

```
karabo-data-make-virtual-cxi /gpfs/exfel/exp/XMPL/201750/p700000/proc/r0003 -o xmpl-3.cxi
```

-o <path>, --output <path>
The filename to write. Defaults to creating a file in the proposal’s scratch directory.

--min-modules <number>
Include trains where at least N modules have data (default 9).
CHAPTER
SEVEN

DATA FILES FORMAT

The main unit of data this tool works with is a run. A run is data collected in a specific period, and each research proposal given beantime at European XFEL may collect hundreds of runs.

A run is stored as a directory containing HDF5 data files from different sources. These fall into two important categories:

1. Detector data, from the main X-ray detectors in the various experiments.
   - Each detector module writes separate files, e.g. RAW-R0348-AGIPD00-S00000.h5. The number in the third part of the filename identifies the module (0 in this example).
   - The detectors in use as of April 2018 are LPD and AGIPD in the file names. Each has 16 modules numbered 0–15.

2. All the other data, such as motor positions, beam measurements, etc., are recorded through a data aggregator, and stored in a file with the letters DA in the name, e.g. RAW-R0450-DA01-S00000.h5.

The last part of the file name (e.g. S00000) is a sequence number. The data within a run may be broken into a number of sequences. So RAW-R0450-DA01-S00000.h5 and RAW-R0450-DA01-S00001.h5 will contain data from the same set of devices, with sequence 1 continuing just after the end of sequence 0. Though all data within a run may be broken into sequences, different data sets do not necessarily break at the same point, so the various ‘sequence 0’ data files in a run do not have corresponding data.

7.1 HDF5 file structure

7.1.1 METADATA

The METADATA group in an HDF5 file contains three datasets, each of which is a 1D array of strings:

- METADATA/dataSourceId lists data groups in the file. The values are either:
  - CONTROL/ followed by a Karabo device name, e.g. CONTROL/SA1_XTD2_XGM/DOOCS/MAIN.
  - INSTRUMENT/ followed by a Karabo device name, a colon, the name of the output channel, a slash, and the name of a data group (?), e.g. INSTRUMENT/SA1_XTD2_XGM/DOOCS/MAIN:output/data
- METADATA/deviceId lists the part of each dataSourceId after the first slash.
- METADATA/root lists the parts before the first slash, so concat(root, "/", deviceId) == dataSourceId.

These three data sets always have the same number of values. They may be padded with empty strings, so empty entries are ignored.
7.1.2 INDEX

INDEX/trainId is a 1D array of uint64, listing the pulse trains which the file holds data for. This is crucial, since all other data has to be matched up according to train IDs.

For each entry in METADATA/deviceId, the INDEX group contains two datasets, both uint64 data with the same length as the train IDs:

• INDEX/{ deviceId }/count: for each train ID, how many data samples did this device record. This may be 0 if no data was recorded for this train.
• INDEX/{ deviceId }/first: for each train ID, the index at which the corresponding data starts in the arrays for this device.

Thus, to find the data for a given train ID, we could do:

```python
train_index = trainIds.index(train_id)
first = device_firsts[train_index]
count = device_counts[train_index]
train_data = data[first : first+count]
```

Control data is always (?) recorded once per train, so count is 1 and first counts up from 0 to the number of trains. Instrument data is more variable.

Some older files use a different index format with first/last/status instead of first/count. In this case, a status of 0 means that no data was recorded for that train.

7.1.3 CONTROL and RUN

For each CONTROL entry in METADATA/dataSourceId, there is a group with that name in the file. This may have further arbitrarily nested subgroups representing different properties of that device, e.g. /CONTROL/SA1_XTD2_XGM/DOOCS/MAIN/current/bottom/output.

The leaves of this tree are pairs of datasets called timestamp and value. Each dataset has one entry per train, and the timestamp record when the value was updated, which is typically less than once per train. The value dataset may have extra dimensions, but in most cases it is 1D.

(Does timestamp update if value is re-read but doesn’t change?)

RUN holds a complete duplicate of the CONTROL hierarchy, but each pair of timestamp and value contain only one entry, taken at the start of the run. There is still a dimension for this, so 2D value datasets in CONTROL have corresponding 2D datasets in RUN, but the first dimension has length 1.

(Is RUN exactly duplicated in subsequent sequence files?)

7.1.4 INSTRUMENT

For each INSTRUMENT entry in METADATA/dataSourceId, there is a group with that name in the file. Each such group holds a 1D trainId dataset, and a number of other datasets (possibly nested in subgroups). All these datasets have the same length in the first dimension: this represents the successive readings taken. The slices defined by the corresponding datasets in INDEX work on this dimension.

The trainId dataset for each instrument group thus appears to be redundant with the information in INDEX.
This command creates the sample data files used in the rest of this example. These files contain no real data, but they have the same structure as European XFEL's HDF5 data files.

8.1 Single files

```python
[2]: !h5ls fxe_control_example.h5
CONTROL Group
INDEX Group
INSTRUMENT Group
METADATA Group
RUN Group

[3]: from karabo_data import H5File
f = H5File('fxe_control_example.h5')

[4]: f.control_sources
frozenset({'FXE_XAD_GEC/CAM/CAMERA', 'SA1_XTD2_XGM/DOOCS/MAIN', 'SPB_XTD9_XGM/DOOCS/MAIN'})

[5]: f.instrument_sources
frozenset({'FXE_XAD_GEC/CAM/CAMERA:daqOutput', 'SA1_XTD2_XGM/DOOCS/MAIN:output', 'SPB_XTD9_XGM/DOOCS/MAIN:output'})

8.1.1 Get data by train

```python
[6]: for tid, data in f.trains():
    print("Processing train", tid)
    print("beam iyPos:", data['SA1_XTD2_XGM/DOOCS/MAIN']"beamPosition.iyPos.value")
    break
```
Processing train 10000
beam iyPos: 0.0

```
[7]: tid, data = f.train_from_id(10005)
data['FXE_XAD_GEC/CAM/CAMERA:daqOutput']['data.image.dims']
[7]: array([1024, 255], dtype=uint64)
```

These are just a few of the ways to access data. The attributes and methods described below for run directories also work with individual files. We expect that it will normally make sense to access a run directory as a single object, rather than working with the files separately.

## 8.2 Run directories

An experimental run is recorded as a collection of files in a directory.

Another dummy example:

```
[8]: !ls fxe_example_run/
RAW-R0450-DA01-S00000.h5  RAW-R0450-LPD04-S00000.h5  RAW-R0450-LPD10-S00000.h5
RAW-R0450-DA01-S00001.h5  RAW-R0450-LPD05-S00000.h5  RAW-R0450-LPD11-S00000.h5
RAW-R0450-LPD00-S00000.h5  RAW-R0450-LPD06-S00000.h5  RAW-R0450-LPD12-S00000.h5
RAW-R0450-LPD01-S00000.h5  RAW-R0450-LPD07-S00000.h5  RAW-R0450-LPD13-S00000.h5
RAW-R0450-LPD02-S00000.h5  RAW-R0450-LPD08-S00000.h5  RAW-R0450-LPD14-S00000.h5
RAW-R0450-LPD03-S00000.h5  RAW-R0450-LPD09-S00000.h5  RAW-R0450-LPD15-S00000.h5
```

```
[9]: from karabo_data import RunDirectory
run = RunDirectory('fxe_example_run/
```

```
[10]: run.files[:3]  # The objects for the individual files (see above)
[10]: [FileAccess(<HDF5 file "RAW-R0450-LPD04-S00000.h5" (mode r)>,
FileAccess(<HDF5 file "RAW-R0450-LPD11-S00000.h5" (mode r)>),
FileAccess(<HDF5 file "RAW-R0450-LPD15-S00000.h5" (mode r)>))
```

What devices were recording in this run?

*Control* devices are slow data, recording once per train. *Instrument* devices includes detector data, but also some other data sources such as cameras. They can have more than one reading per train.

```
[11]: run.control_sources
[11]: frozenset({'FXE_XAD_GEC/CAM/CAMERA',
'FXE_XAD_GEC/CAM/CAMERA_NODATA',
'SA1_XTD2_XGM/DOOCS/MAIN',
'SPB_XTD9_XGM/DOOCS/MAIN'})
```

```
[12]: run.instrument_sources
[12]: frozenset({'FXE_DET_LPD1M-1/DET/0CH0:xtdf',
'FXE_DET_LPD1M-1/DET/10CH0:xtdf',
'FXE_DET_LPD1M-1/DET/11CH0:xtdf',
'FXE_DET_LPD1M-1/DET/12CH0:xtdf',
'FXE_DET_LPD1M-1/DET/13CH0:xtdf',
'FXE_DET_LPD1M-1/DET/14CH0:xtdf',
```

(continues on next page)
Which trains are in this run?

```python
[13]: print(run.train_ids[:10])
[10000, 10001, 10002, 10003, 10004, 10005, 10006, 10007, 10008, 10009]
```

See the available keys for a given source:

```python
[14]: run.keys_for_source('SPB_XTD9_XGM/DOOCS/MAIN:output')
```

```python
[14]: {'data.intensityAUXTD',
       'data.intensitySigma.x_data',
       'data.intensitySigma.y_data',
       'data.intensityTD',
       'data.trainId',
       'data.xTD',
       'data.yTD'}
```

This collects data from across files, including detector data:

```python
[15]: for tid, data in run.trains():
    print("Processing train", tid)
    print("Detector data module 0 shape:", data['FXE_DET_LPD1M-1/DET/0CH0:xtdf'][˓→'image.data'].shape)
    break  # Stop after the first train to keep the demo short
```

Processing train 10000  
Detector data module 0 shape: (128, 1, 256, 256)

Train IDs are meant to be globally unique (although there were some glitches with this in the past). A train index is only within this run.

```python
[16]: tid, data = run.train_from_id(10005)
    tid, data = run.train_from_index(5)
```

### 8.2.1 Series data to pandas

Data which holds a single number per train (or per pulse) can be extracted to as *series* (individual columns) and *dataframes* (tables) for *pandas*, a widely-used tool for data manipulation.
karabo_data chains sequence files, which contain successive data from the same source. In this example, trains 10000–10399 are in one sequence file (DA01-S00000.h5), and 10400–10479 are in another (DA01-S00001.h5). They are concatenated into one series:

```python
ixPos = run.get_series('SA1_XTD2_XGM/DOOCS/MAIN', 'beamPosition.ixPos.value')
ixPos.tail(10)
```

```plaintext
trainId
10470  0.0
10471  0.0
10472  0.0
10473  0.0
10474  0.0
10475  0.0
10476  0.0
10477  0.0
10478  0.0
10479  0.0
Name: SA1_XTD2_XGM/DOOCS/MAIN/beamPosition.ixPos, dtype: float32
```

To extract a dataframe, you can select interesting data fields with `glob` syntax, as often used for selecting files on Unix platforms.

- `[abc]`: one character, a/b/c
- `?`: any one character
- `*`: any sequence of characters

```python
run.get_dataframe(fields=[("*_XGM/*", "*_i[xy]Pos")])
```

```plaintext
trainId
SA1_XTD2_XGM/DOOCS/MAIN/beamPosition.ixPos
10000 0.0
10001 0.0
10002 0.0
10003 0.0
10004 0.0
10005 0.0
10006 0.0
10007 0.0
10008 0.0
10009 0.0
10010 0.0
10011 0.0
10012 0.0
10013 0.0
10014 0.0
10015 0.0
10016 0.0
10017 0.0
10018 0.0
10019 0.0
10020 0.0
10021 0.0
10022 0.0
10023 0.0
10024 0.0
10025 0.0
```

(continues on next page)
SA1_XTD2_XGM/DOOCS/MAIN/beamPosition.iyPos \

trainId
10000 0.0
10001 0.0
10002 0.0
10003 0.0
10004 0.0
10005 0.0
10006 0.0
10007 0.0
10008 0.0
10009 0.0
10010 0.0
10011 0.0
10012 0.0
10013 0.0
10014 0.0
10015 0.0
10016 0.0
10017 0.0
10018 0.0
10019 0.0

(continues on next page)
8.2. Run directories
(continued from previous page)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10006</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10007</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10008</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10009</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10012</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10013</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10014</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10015</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10016</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10017</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10018</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10019</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10020</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10021</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10022</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10023</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10024</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10025</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10026</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10027</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10028</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10029</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>10450</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10451</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10452</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10453</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10454</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10455</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10456</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10457</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10458</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10459</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10460</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10461</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10462</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10463</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10464</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10465</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10466</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10467</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10468</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10469</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10470</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10471</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10472</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10473</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10474</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10475</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10476</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10477</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10478</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10479</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

[480 rows x 4 columns]
8.2.2 Labelled arrays

Data with extra dimensions can be handled as xarray labelled arrays. These are a wrapper around Numpy arrays with indexes which can be used to align them and select data.

```
[19]: xtd2_intensity = run.get_array('SA1_XTD2_XGM/DOOCS/MAIN:output', 'data.intensityTD',
extra_dims=['pulseID'])
xtd2_intensity
```

```
[19]: <xarray.DataArray (trainId: 480, pulseID: 1000)>
    array([[0., 0., 0., ..., 0., 0., 0.],
           [0., 0., 0., ..., 0., 0., 0.],
           [0., 0., 0., ..., 0., 0., 0.],
           ...
           [0., 0., 0., ..., 0., 0., 0.],
           [0., 0., 0., ..., 0., 0., 0.],
           [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
    Coordinates:
    * trainId (trainId) uint64 10000 10001 10002 10003 ... 10477 10478 10479
    Dimensions without coordinates: pulseID
```

Here’s a brief example of using xarray to align the data and select by train ID. See the examples in the xarray docs for more on what it can do.

In this example data, all the data sources have the same range of train IDs, so aligning them doesn’t change anything. In real data, devices may miss some trains that other devices did record.

```
[20]: import xarray as xr
xtd9_intensity = run.get_array('SPB_XTD9_XGM/DOOCS/MAIN:output', 'data.intensityTD',
extra_dims=['pulseID'])

# Align two arrays, keep only trains which they both have data for:
xtd2_intensity, xtd9_intensity = xr.align(xtd2_intensity, xtd9_intensity, join='inner')

# Select data for a single train by train ID:
xtd2_intensity.sel(trainId=10004)

# Select data from a range of train IDs.
# This includes the end value, unlike normal Python indexing
xtd2_intensity.loc[10004:10006]
```

```
[20]: <xarray.DataArray (trainId: 3, pulseID: 1000)>
    array([[0., 0., 0., ..., 0., 0., 0.],
           [0., 0., 0., ..., 0., 0., 0.],
           [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
    Coordinates:
    * trainId (trainId) uint64 10004 10005 10006
    Dimensions without coordinates: pulseID
```

You can also specify a region of interest from an array to load only part of the data:

```
[21]: from karabo_data import by_index

    # Select the first 5 trains in this run:
sel = run.select_trains(by_index[:5])

    # Get the whole of this array:
arr = sel.get_array('FXE_XAD_GEC/CAM/CAMERA:daqOutput', 'data.image.pixels')
```

(continues on next page)
print("Whole array shape:", arr.shape)

# Get a region of interest
arr2 = sel.get_array('FXE_XAD_GEC/CAM/CAMERA:daqOutput', 'data.image.pixels', roi=by_index[100:200, :512])
print("ROI array shape:", arr2.shape)

Whole array shape: (5, 255, 1024)
ROI array shape: (5, 100, 512)

8.3 General information

karabo_data provides a few ways to get general information about what’s in data files. First, from Python code:

[22]: run.info()

# of trains: 480
Duration: 0:00:47.900000
First train ID: 10000
Last train ID: 10479

16 detector modules (FXE_DET_LPD1M-1)
  e.g. module FXE_DET_LPD1M-1 0 : 256 x 256 pixels
  128 frames per train, 61440 total frames

4 instrument sources (excluding detectors):
  - FXE_XAD_GEC/CAM/CAMERA:daqOutput
  - FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput
  - SA1_XTD2_XGM/DOOCS/MAIN:output
  - SPB_XTD9_XGM/DOOCS/MAIN:output

4 control sources:
  - FXE_XAD_GEC/CAM/CAMERA
  - FXE_XAD_GEC/CAM/CAMERA_NODATA
  - SA1_XTD2_XGM/DOOCS/MAIN
  - SPB_XTD9_XGM/DOOCS/MAIN

[23]: run.detector_info('FXE_DET_LPD1M-1/DET/0CH0:xtdf')

[23]: {'dims': (256, 256), 'frames_per_train': 128, 'total_frames': 61440}

The lsxfel command provides similar information at the command line:

[24]: !lsxfel fxe_example_run/RAW-R0450-LPD00-S00000.h5

RAW-R0450-LPD00-S00000.h5 : Raw detector data from LPD module 00
480 trains
256 × 256 pixels
128 frames per train, 61440 total

[25]: !lsxfel fxe_example_run/RAW-R0450-DA01-S00000.h5
4 instrument sources
- FXE_XAD_GEC/CAM/CAMERA:daqOutput
- FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput
- SA1_XTD2_XGM/DOOCS/MAIN:output
- SPB_XTD9_XGM/DOOCS/MAIN:output

4 control sources
- FXE_XAD_GEC/CAM/CAMERA
- FXE_XAD_GEC/CAM/CAMERA_NODATA
- SA1_XTD2_XGM/DOOCS/MAIN
- SPB_XTD9_XGM/DOOCS/MAIN

8.3. General information 35
The Large Pixel Detector (LPD) is made of 16 modules which record data separately. karabo_data includes convenient interfaces to access this data together.

This example stands by itself, but if you need more generic access to the data, please see Reading data with karabo_data.

First, let’s load a run containing LPD data:

```python
from karabo_data import RunDirectory, by_index

run = RunDirectory('fxe_example_run/')
# Using only the first three trains to keep this example light:
run = run.select_trains(by_index[:3])

run.instrument_sources
```

Normal access methods give us each module separately:

```python
data_module0 = run.get_array('FXE_DET_LPD1M-1/DET/0CH0:xtdf', 'image.data')
data_module0.shape
```

The class karabo_data.components.LPD1M can piece these together:
from karabo_data.components import LPD1M
lpd = LPD1M(run)
lpd

Data interface for detector 'FXE_DET_LPD1M-1' with 16 modules

image_data = lpd.get_array('image.data')
print("Data shape:", image_data.shape)
print("Dimensions:", image_data.dims)
Data shape: (16, 3, 128, 256, 256)
Dimensions: ('module', 'train', 'pulse', 'slow_scan', 'fast_scan')

Note: This class pulls the data together, but it doesn't know how the modules are physically arranged, so it can't produce a detector image. Other examples show how to use detector geometry to produce images.

You can also select only certain modules of the detector. For example, modules 2 (Q1M3), 7 (Q2M4), 8 (Q3M1) and 13 (Q4M2) are the four modules around the center of the detector:

lpd = LPD1M(run, modules=[2, 7, 8, 13])
image_data = lpd.get_array('image.data')
print("Data shape:", image_data.shape)
print("Dimensions:", image_data.dims)

print()
print("Data for one pulse:")
print(image_data.sel(train=10000, pulse=0))
Data shape: (4, 3, 128, 256, 256)
Dimensions: ('module', 'train', 'pulse', 'slow_scan', 'fast_scan')
Data for one pulse:
<xarray.DataArray (module: 4, slow_scan: 256, fast_scan: 256)>
array([[[0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0],
       ...,
       [0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0]],
       [[0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0],
       ...,
       [0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0]],
       [[0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0],
       ...,
       [0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0]],
       [[0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0],
       ...,
       [0, 0, ..., 0, 0],
       [0, 0, ..., 0, 0]]], dtype=uint16)
Coordinates:
pulse    uint64 0
train    uint64 10000

(continues on next page)
The returned array is an *xarray* object with labelled axes. See Indexing and selecting data in the *xarray* docs for more on what you can do with it.

This interface also supports iterating train-by-train through detector data, giving labelled arrays again:

```python
[6]:
    for tid, train_data in lpd.trains(pulses=by_index[:16]):
        print("Train", tid)
        print("Keys in data:", sorted(train_data.keys()))
        print("Image data shape:", train_data['image.data'].shape)
        print()
```

Train 10000
Keys in data: ['detector.data', 'detector.trainId', 'header.dataId', 'header.linkId',
  'header.magicNumberBegin', 'header.majorTrainFormatVersion', 'header.
  minorTrainFormatVersion', 'header.pulseCount', 'header.reserved', 'header.trainId',
  'image.cellId', 'image.data', 'image.length', 'image.pulseId', 'image.status',
  'image.trainId', 'trailer.checksum', 'trailer.magicNumberEnd', 'trailer.status',
  'trailer.trainId']
Image data shape: (4, 1, 16, 256, 256)

Train 10001
Keys in data: ['detector.data', 'detector.trainId', 'header.dataId', 'header.linkId',
  'header.magicNumberBegin', 'header.majorTrainFormatVersion', 'header.
  minorTrainFormatVersion', 'header.pulseCount', 'header.reserved', 'header.trainId',
  'image.cellId', 'image.data', 'image.length', 'image.pulseId', 'image.status',
  'image.trainId', 'trailer.checksum', 'trailer.magicNumberEnd', 'trailer.status',
  'trailer.trainId']
Image data shape: (4, 1, 16, 256, 256)

Train 10002
Keys in data: ['detector.data', 'detector.trainId', 'header.dataId', 'header.linkId',
  'header.magicNumberBegin', 'header.majorTrainFormatVersion', 'header.
  minorTrainFormatVersion', 'header.pulseCount', 'header.reserved', 'header.trainId',
  'image.cellId', 'image.data', 'image.length', 'image.pulseId', 'image.status',
  'image.trainId', 'trailer.checksum', 'trailer.magicNumberEnd', 'trailer.status',
  'trailer.trainId']
Image data shape: (4, 1, 16, 256, 256)
ASSEMBLING DETECTOR DATA INTO IMAGES

The X-ray detectors at XFEL are made up of a number of small pieces. To get an image from the data, or analyse it spatially, we need to know where each piece is located.

This example reassembles some commissioning data from LPD, a detector which has 4 quadrants, 16 modules, and 256 tiles. Elements (especially the quadrants) can be repositioned; talk to the detector group to ensure that you have the right geometry information for your data.

```
[1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import h5py
from karabo_data import RunDirectory, stack_detector_data
from karabo_data.geometry2 import LPD_1MGeometry

[2]: run = RunDirectory('/gpfs/exfel/exp/FXE/201830/p900020/proc/r0221/')
run.info()
# of trains: 513
Duration: 0:00:51.200000
First train ID: 54861753
Last train ID: 54862265

14 detector modules (FXE_DET_LPD1M-1)
e.g. module FXE_DET_LPD1M-1 0 : 256 x 256 pixels
128 frames per train, 39040 total frames

0 instrument sources (excluding detectors):
0 control sources:

[3]: # Find a train with some data in
empty = np.asarray([])
for tid, train_data in run.trains():
    module_imgs = sum(d.get('image.data', empty).shape[0] for d in train_data.values())
    if module_imgs:
        print(tid, module_imgs)
        break
54861797 1792
```
```python
[4]: tid, train_data = run.train_from_id(54861797)
print(tid)
for dev in sorted(train_data.keys()):
    print(dev, end='\t')
    try:
        print(train_data[dev]['image.data'].shape)
    except KeyError:
        print("No image.data")

54861797
FXE_DET_LPD1M-1/DET/0CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/10CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/11CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/12CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/13CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/14CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/15CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/1CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/2CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/3CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/4CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/6CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/8CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/9CH0:xtdf (128, 256, 256)

Extract the detector images into a single Numpy array:

[5]: modules_data = stack_detector_data(train_data, 'image.data')
modules_data.shape

[5]: (128, 16, 256, 256)

To show the images, we sometimes need to ‘clip’ extreme high and low values, otherwise the colour map makes everything else the same colour.

[6]: def clip(array, min=-10000, max=10000):
    x = array.copy()
    finite = np.isfinite(x)
    # Suppress warnings comparing numbers to nan
    with np.errstate(invalid='ignore'):
        x[finite & (x < min)] = np.nan
        x[finite & (x > max)] = np.nan
    return x

[7]: plt.figure(figsize=(10, 5))

    a = modules_data[5][2]
    plt.subplot(1, 2, 1).hist(a[np.isfinite(a)])

    a = clip(a, min=-400, max=400)
    plt.subplot(1, 2, 2).hist(a[np.isfinite(a))];
```

Chapter 10. Assembling detector data into images
Let’s look at the image from a single module. You can see where it’s divided up into tiles:

```python
plt.figure(figsize=(8, 8))
clipped_mod = clip(modules_data[10][2], -400, 500)
plt.imshow(clipped_mod, origin='lower')
```

```python
<matplotlib.image.AxesImage at 0x2ab829b330b8>
```
Here’s a single tile:

```python
[9]: splitted = LPD_1MGeometry.split_tiles(clipped_mod)
plt.figure(figsize=(8, 8))
plt.imshow(splitted[11])
```

Load the geometry from a file, along with the quadrant positions used here.
In the future, geometry information will be stored in the calibration catalogue.

```python
[10]: quadpos = [(11.4, 299), (-11.5, 8), (254.5, -16), (278.5, 275)]  # mm
```

(continues on next page)
Reassemble and show a detector image using the geometry:

```python
[11]: geom.plot_data_fast(clip(modules_data[12], max=5000))
```

Reassemble detector data into a numpy array for further analysis. The areas without data have the special value ‘nan’ to mark them as missing.

```python
[12]: res, centre = geom.position_modules_fast(modules_data)
    print(res.shape)
    plt.figure(figsize=(8, 8))
    plt.imshow(clip(res[12, 250:750, 450:850], min=-400, max=5000), origin='lower')
```
EXAMINING DETECTOR GEOMETRY

The *Applying geometry notebook* shows how to use detector geometry to assemble data into an image. We can also examine geometry information without any data, to check for problems.

```python
from itertools import product
import numpy as np
import matplotlib.pyplot as plt
import h5py
from karabo_data import RunDirectory
from karabo_data.geometry2 import LPD_1MGeometry

This is some geometry for LPD. You can see that Q2M2 is ‘missing’ - in fact all its tiles are showing up in Q2M4.

Each module has tiles 1-16 running anticlockwise from the top left (looking into the beam). To make it visually clearer, only three corner tiles of each module are numbered.

Here we are loading the geometry from a file, but in the future it will be possible to get the information from the calibration database directly.

```
The AGIPD detector, which is already in use at the SPB experiment, consists of 16 modules of 512×128 pixels each. Each module is further divided into 8 ASICs.

To view or analyse detector data, we need to apply geometry to find the positions of pixels.

```
%matplotlib inline
import numpy as np
from karabo_data import RunDirectory, stack_detector_data
from karabo_data.geometry2 import AGIPD_1MGeometry

Fetch AGIPD detector data for one pulse to test with:

```
run = RunDirectory('/gpfs/exfel/exp/SPB/201831/p900039/proc/r0273/')
```

```
tid, train_data = run.select('*/DET/*', 'image.data').train_from_index(60)
```

```
stacked = stack_detector_data(train_data, 'image.data')
stacked_pulse = stacked[10]
stacked_pulse.shape
```

```
(16, 512, 128)
```

Generate a simple geometry given the (x, y) coordinates of the first pixel in the first module of each quadrant, in pixel units relative to the centre, where the beam passes through the detector.

There are also methods to load and save CrystFEL format geometry files.

```
geom = AGIPD_1MGeometry.from_quad_positions(quad_pos=[
    (-525, 625),
    (-550, -10),
    (520, -160),
    (542.5, 475),
])
```

```
geom.inspect()
```
The pixels are not necessarily all aligned, so precisely assembling data in a 2D array requires interpolation, which is slow:

```
[7]: %%time
data, centre_yx = geom.position_modules_interpolate(stacked_pulse)
print(data.shape)
(1258, 1094)
CPU times: user 12.7 s, sys: 499 ms, total: 13.2 s
Wall time: 7.2 s
```

But we know that the modules are closely aligned with the axes, so we can ‘snap’ the geometry to the grid and copy data more efficiently:

```
[8]: %%time
data, centre_yx = geom.position_modules_fast(stacked_pulse)
print(data.shape)
```
geom.plot_data_fast(np.clip(stacked_pulse, 0, 1000))
The biggest and often most important data at European XFEL comes from X-ray pixel detectors, but there are many other data sources which may be of interest. This data is often small enough to load it completely into memory, making it much easier to work with.

13.1 Using pandas

This example works with data from two X-Ray Gas Monitors (XGMs). These measure properties of the X-ray beam in different parts of the tunnel. This data refers to one XGM in XTD2 and one in XTD9.

We create a pandas dataframe containing the beam x and y position at each XGM, and the photon flux. We select the columns using ‘glob’ patterns: * is a wildcard matching anything.

pandas makes it very convenient to work with tabular data like this, though we’re limited to datasets that have a single value per train.
We can now make plots to compare the parameters at the two XGM positions. As expected, there’s a strong correlation for each parameter.

```python
[4]: df.plot.scatter(x='SA1_XTD2_XGM/XGM/DOOCS/pulseEnergy.photonFlux', y='SPB_XTD9_XGM/XGM/DOOCS/pulseEnergy.photonFlux')
[4]: <matplotlib.axes._subplots.AxesSubplot at 0x2b2de8e244a8>
```
[5]: ax = df.plot.scatter(x='SA1_XTD2_XGM/XGM/DOOCS/beamPosition.ixPos', y='SPB_XTD9_XGM/→XGM/DOOCS/beamPosition.ixPos')

[6]: ay = df.plot.scatter(x='SA1_XTD2_XGM/XGM/DOOCS/beamPosition.iyPos', y='SPB_XTD9_XGM/→XGM/DOOCS/beamPosition.iyPos')
We can also export the dataframe to a CSV file - or any other format pandas supports - for further analysis with other tools.

```python
[7]: df.to_csv('xtd2_xtd9_xgm_r150.csv')
```

### 13.2 Using xarray

xarray adds pandas-style axis labelling to multidimensional numpy arrays. We can get xarray arrays for data which has multiple values per train. For example, the Photo-Electron Spectrometer (PES) is a monitoring device which records energy spectra for each train. Here’s the data from one of its 16 spectrometers:

```python
[8]: run = RunDirectory('/gpfs/exfel/exp/SA3/201830/p900027/raw/r0067/')
[9]: run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_4_A.raw.samples')
```

```
<xarray.DataArray (trainId: 1475, dim_0: 40000)>
array([[  -6,  -10,  -7, ...,  -10,  -8,  -9],
       [  -8,  -8,  -7, ...,  -9,  -2,  -11],
       [  -8,  -10,  -7, ...,  -6,  -8,  -11],
       ...
       [  -7,  -9,  -8, ...,  -9,  -2,  -5],
       [  -5,  -10,  -8, ...,  -5,  -4,  -10],
       [  -7,  -8,  -7, ...,  -6,  -5,  -8]], dtype=int16)
```

The PES consists of 16 spectrometers arranged in a circle around the beamline. We’ll retrieve the data for two of these, separated by 90°. N and E refer to their positions in the circle, although these are not literally North and East.

The `xarray.align()` function aligns data using the axes. This is important if you’re comparing data from different sources, because it matches up the train IDs. By specifying `join='inner'`, we keep only the trains which have data in both sets.
```python
[10]:
data_n = run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_4_A.raw.
→samples')
data_e = run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_3_A.raw.
→samples')
data_n, data_e = xr.align(data_n, data_e, join='inner')
nsamples = data_n.shape[1]
data_n.shape

[10]: (1475, 40000)
```

We’ll get a few other values from slow data to annotate the plot.

```python
[11]: # Get the first values from four channels measuring voltage
electr = run.get_dataframe([('SA3_XTD10_PES/MCPS/MPOD', 'channels.U20[0123].
→measurementSenseVoltage')])
electr_voltages = electr.iloc[0].sort_index()
electr_voltages

[11]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SA3_XTD10_PES/MCPS/MPOD/channels.U200.measurementSenseVoltage</td>
<td>-0.101792</td>
</tr>
<tr>
<td>SA3_XTD10_PES/MCPS/MPOD/channels.U201.measurementSenseVoltage</td>
<td>-0.111782</td>
</tr>
<tr>
<td>SA3_XTD10_PES/MCPS/MPOD/channels.U202.measurementSenseVoltage</td>
<td>-0.106823</td>
</tr>
<tr>
<td>SA3_XTD10_PES/MCPS/MPOD/channels.U203.measurementSenseVoltage</td>
<td>-0.107910</td>
</tr>
</tbody>
</table>
Name: 128146446, dtype: float32
```

```python
[12]: gas_interlocks = run.get_dataframe([('SA3_XTD10_PES/DCTRL/*/interlock.AActionState
→')])

# Take the first row of the gas interlock data and find which gas was unlocked
row = gas_interlocks.iloc[0]
print(row)
if (row == 0).any():
    key = row[row == 0].index[0]
target_gas = re.search(r'(XENON|KRYPTON|NITROGEN|NEON)', key).group(1).title()
else:
    target_gas = 'No gas'
```

```text
SA3_XTD10_PES/DCTRL/V30300S_NITROGEN/interlock.AActionState 1
SA3_XTD10_PES/DCTRL/V30320S_KRYPTON/interlock.AActionState 1
SA3_XTD10_PES/DCTRL/V30310S_NEON/interlock.AActionState 0
SA3_XTD10_PES/DCTRL/V30330S_XENON/interlock.AActionState 1
Name: 128146446, dtype: uint32
```

Now we can average the spectra across the trains in this run, and plot them.

```python
[14]: x = np.linspace(0, 0.0005*nsamples, nsamples, endpoint=False)
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for ax, dataset, start_time in zip(axes, [data_n, data_e], [15.76439411, 15.76289411]):
   ax.plot(x, dataset.sum(axis=0))
   ax.yaxis.major.formatter.set_powerlimits((0, 0))
   ax.set_xlim(15.75, 15.85)
   ax.set_xlabel('time ($\mu$s)')
   ax.axvline(start_time, color='red', linestyle='dotted', label='Start time')
   ax.axvline(start_time + 0.0079, color='magenta', linestyle='dotted', label='Neon
→K 1s')
   ax.axvline(start_time + 0.041, color='black', label='Auger peak')
```

(continues on next page)
The spectra look different because the beam is horizontally polarised, so the E spectrometer sees a peak that the N spectrometer doesn’t.
14.1 0.4

- Python 3.5 is now the minimum required version.
- Fix compatibility with numpy 1.14 (the version installed in Anaconda on the Maxwell cluster).
- Better error message from `stack_detector_data()` when passed non-detector data.

14.2 0.3

New features:

- New interfaces for working with AGIPD & LPD Geometry.
- New interfaces for accessing AGIPD & LPD data.
- `select_trains()` can now select arbitrary specified trains, not just a slice.
- `get_array()` can take a region of interest (roi) parameter to select a slice of data from each train.
- A newly public `keys_for_source()` method to list keys for a given source.

Fixes:

- `stack_detector_data()` can handle missing detector modules.
- Source sets have been changed to frozen sets. Use `select()` to choose a subset of sources.
- `get_array()` now only loads the data for selected trains.
- `get_array()` works with data recorded more than once per train.

14.3 0.2

- New command `karabo-data-validate` to check the integrity of data files.
- New methods to select a subset of data: `select()`, `deselect()`, `select_trains()`, `union()`.
- Selected data can be written back to a new HDF5 file with `write()`.
- `RunDirectory()` and `H5File()` are now functions which return a `DataCollection` object, rather than separate classes. Most code using these should still work, but checking the type with e.g. `isinstance()` may break.
See also:

Data Analysis at European XFEL
CHAPTER
FIFTEEN

INDICES AND TABLES

• genindex
• search
k

karabo_data, 3
karabo_data.components, 9
karabo_data.geometry2, 15
Symbols
-min-modules <number>  
  karabo-data-make-virtual-cxi  
  command line option, 21
-o <path>, -output <path>  
  karabo-data-make-virtual-cxi  
  command line option, 21

A
AGIPD1M (class in karabo_data.components), 9
AGIPD_1MGeometry (class in karabo_data.geometry2), 15
all_sources (karabo_data.DataCollection attribute), 4

C
compare() (karabo_data.geometry2.AGIPD_1MGeometry method), 18
control_sources (karabo_data.DataCollection attribute), 3

D
DataCollection (class in karabo_data), 3
deselect() (karabo_data.DataCollection method), 6

F
feed() (karabo_data.ZMQStreamer method), 12
from_crystfel_geom()  
  (karabo_data.geometry2.AGIPD_1MGeometry class method), 15
from_h5_file_and_quad_positions()  
  (karabo_data.geometry2.LPD_1MGeometry class method), 20
from_quad_positions()  
  (karabo_data.geometry2.AGIPD_1MGeometry class method), 15
from_quad_positions()  
  (karabo_data.geometry2.LPD_1MGeometry class method), 18

G
get_array() (karabo_data.components.LPD1M method), 9
get_array() (karabo_data.DataCollection method), 5
get_dataframe() (karabo_data.DataCollection method), 5
gt_series() (karabo_data.DataCollection method), 5

H
H5File() (in module karabo_data), 3

I
info() (karabo_data.DataCollection method), 4
inspect() (karabo_data.geometry2.AGIPD_1MGeometry method), 18
inspect() (karabo_data.geometry2.LPD_1MGeometry method), 20
instrument_sources (karabo_data.DataCollection attribute), 4

K
karabo-data-make-virtual-cxi command line option
  -min-modules <number>, 21
  -o <path>, -output <path>, 21
karabo_data (module), 3
karabo_data.components (module), 9
karabo_data.geometry2 (module), 15
keys_for_source() (karabo_data.DataCollection method), 4

L
LPD1M (class in karabo_data.components), 9
LPD_1MGeometry (class in karabo_data.geometry2), 18

O
open_run() (in module karabo_data), 3

P
plot_data_fast() (karabo_data.geometry2.AGIPD_1MGeometry method), 17
plot_data_fast() (karabo_data.geometry2.LPD_1MGeometry
method), 20
position_modules_fast()
(karabo_data.geometry2.AGIPD_1MGeometry
method), 17
position_modules_fast()
(karabo_data.geometry2.LPD_1MGeometry
method), 20
position_modules_interpolate()
(karabo_data.geometry2.AGIPD_1MGeometry
method), 17

R
RunDirectory() (in module karabo_data), 3

S
select() (karabo_data.DataCollection method), 5
select_trains() (karabo_data.DataCollection
method), 6
stack_detector_data() (in module
karabo_data.reader), 10
start() (karabo_data.ZMQStreamer method), 11

to_distortion_array()
(karabo_data.geometry2.AGIPD_1MGeometry
method), 17
train_from_id() (karabo_data.DataCollection
method), 4
train_from_index() (karabo_data.DataCollection
method), 5
train_ids (karabo_data.DataCollection attribute), 3
trains() (karabo_data.components.LPD1M method),
9
trains() (karabo_data.DataCollection method), 4

U
union() (karabo_data.DataCollection method), 7

W
write() (karabo_data.DataCollection method), 7
write_crystfel_geom()
(karabo_data.geometry2.AGIPD_1MGeometry
method), 17
write_virtual() (karabo_data.DataCollection
method), 7
write_virtual_cxi()
(karabo_data.components.LPD1M method), 10

Z
ZMQStreamer (class in karabo_data), 11