## CONTENTS

1 Installation .................................................. 3

2 Documentation contents ................................. 5
   2.1 Detector geometry reference .......................... 5
   2.2 Performance notes ....................................... 28
   2.3 AGIPD-1M Geometry .................................... 28
   2.4 LPD-1M geometry ....................................... 31
   2.5 DSSC detector geometry ............................... 33
   2.6 Assembling detector data into images .................. 36
   2.7 Converting array coordinates to physical positions .... 43
   2.8 Making data masks ..................................... 46
   2.9 Release Notes .......................................... 54

3 Indices and tables ..................................... 57

Python Module Index .................................. 59

Index ..................................................... 61
EXtra-geom is a Python library to describe the physical layout of multi-module detectors at European XFEL, and to assemble complete detector images.
EXtra-geom is available on our Anaconda installation on the Maxwell cluster:

```bash
module load exfel exfel_anaconda3
```

You can also install it from PyPI to use in other environments with Python 3.5 or later:

```bash
pip install extra_geom
```

If you get a permissions error, add the `--user` flag to that command.
2.1 Detector geometry reference

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. EXtra-geom 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.

2.1.1 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.

class extra_geom.AGIPD_1MGeometry (modules, filename='No file')

Detector layout for AGIPD-1M

The coordinates used in this class are 3D (x, y, z), and represent metres.

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

Returns a new geometry object.

offset(shift, *, modules=slice(None, None, None), tiles=slice(None, None, None))
Move part or all of the detector, making a new geometry.

By default, this moves all modules & tiles. To move the centre down in the image, move the whole
geometry up relative to it.

Returns a new geometry object of the same type.

```python
# Move the whole geometry up 2 mm (relative to the beam)
geom2 = geom.shift((0, 2e-3))

# Move quadrant 1 (modules 0, 1, 2, 3) up 2 mm
geom2 = geom.shift((0, 2e-3), modules=np.s_[0:4])

# Move each module by a separate amount
shifts = np.zeros((16, 3))
shifts[5] = (0, 2e-3, 0)  # x, y, z for individual modules
shifts[10] = (0, -1e-3, 0)
geom2 = geom.shift(shifts)
```

Parameters

- **shift** (numpy.ndarray or tuple) – (x, y) or (x, y, z) shift to apply in metres. Can be a single shift for all selected modules, a 2D array with a shift per module, or a 3D array with a shift per tile (arr[module, tile, xyz]).

- **modules** (slice) – Select modules to move; defaults to all modules. Like all Python slicing, the end number is excluded, so np.s_[0:4] moves modules 0, 1, 2, 3.

- **tiles** (slice) – Select tiles to move within each module; defaults to all tiles.

quad_positions()
Retrieve the coordinates of the first pixel in each quadrant

The coordinates returned are 2D and in pixel units, compatible with from_quad_positions().

write_crystfel_geom(filename, *, data_path='/entry_1/instrument_1/detector_1/data',
mask_path=None, dims=('frame', 'modno', 'ss', 'fs'), nquads=4,
adu_per_ev=None, clen=None, photon_energy=None)
Write this geometry to a CrystFEL format (.geom) geometry file.

Parameters

- **filename** (str) – Filename of the geometry file to write.

- **data_path** (str) – Path to the group that contains the data array in the hdf5 file. Default: '/entry_1/instrument_1/detector_1/data'.

- **mask_path** (str) – Path to the group that contains the mask array in the hdf5 file.

- **dims** (tuple) – Dimensions of the data. Extra dimensions, except for the defaults, should be added by their index, e.g. ('frame', 'modno', 0, 'ss', 'fs') for raw data. Default: ('frame', 'modno', 'ss', 'fs'). Note: the dimensions must contain frame, ss, fs.

- **adu_per_ev** (float) – ADU (analog digital units) per electron volt for the considered detector.
- **clen** (*float*) – Distance between sample and detector in meters
- **photon_energy** (*float*) – Beam wave length in eV

**get_pixel_positions**(centre=True)
Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

**to_distortion_array**(allow_negative_xy=False)
Return distortion matrix for AGIPD detector, suitable for pyFAI.

Parameters **allow_negative_xy** (*bool*) – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

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, ax=None, figsize=None, colorbar=True, **kwargs)
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.
- **ax** (*~matplotlib.axes.Axes* object, optional) – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.
- **figsize** (*tuple*) – Size of the figure (width, height) in inches to be drawn (default: (10, 10))
- **colorbar** (*bool, dict*) – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.
- **kwargs** – Additional keyword arguments passed to ~matplotlib.imshow

**position_modules_fast**(data, out=None, threadpool=None)
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.

• **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is allocated. Use `output_array_for_position_fast()` to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

• **threadpool** (*concurrent.futures.ThreadPoolExecutor, optional*) – If passed, parallelise copying data into the output image. By default, data for different tiles are copied serially. For a single 1 MPx image, the default appears to be faster, but for assembling a stack of several images at once, multithreading can help.

**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.

`output_array_for_position_fast(extra_shape=(), dtype=<class 'numpy.float32'>)`

Make an empty output array to use with `position_modules_fast()`

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the `out=` parameter to `position_modules_fast()`. By default, it allocates a new array on each call, which can be slow.

**Parameters**

• **extra_shape** (*tuple, optional*) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass `extra_shape=(nframes,)` to get a 3D output array.

• **dtype** (*optional (Default: np.float32]*) –

`position_modules_symmetric(data, out=None, threadpool=None)`

Assemble data with the centre in the middle of the output array.

The assembly process is the same as `position_modules_fast()`, aligning each module to a single pixel grid. But this makes the output array symmetric, with the centre at (height // 2, width // 2).

**Parameters**

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

• **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is created at the minimum size to allow symmetric assembly. If an array is passed in, its last two dimensions must be at least this size.

• **threadpool** (*concurrent.futures.ThreadPoolExecutor, optional*) – If passed, parallelise copying data into the output image. See `position_modules_fast()` for details.

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

**Return type** *ndarray*

`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** (*axis_units='px', frontview=True*)

Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

**Parameters**

- **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.

**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** (*DetectorGeometryBase*) — A second geometry object to compare with this one. It should be for the same kind of detector.
- **scale** (*float*) — Scale the arrows showing the difference in positions. This is useful to show small differences clearly.

**data_coords_to_positions** (*module_no, slow_scan, fast_scan*)

Convert data array coordinates to physical positions

Data array coordinates are how you might refer to a pixel in an array of detector data: module number, and indices in the slow-scan and fast-scan directions. But coordinates in the two pixel dimensions aren’t necessarily integers, e.g. if they refer to the centre of a peak.

module_no, fast_scan and slow_scan should all be numpy arrays of the same shape. module_no should hold integers, starting from 0, so 0: Q1M1, 1: Q1M2, etc.

Slow_scan and fast_scan describe positions within that module. They may hold floats for sub-pixel positions. In both, 0.5 is the centre of the first pixel.

Returns an array of similar shape with an extra dimension of length 3, for (x, y, z) coordinates in metres.

See also:

- **AGIPD-1M Geometry** demonstrates using this method.

**extra_geom.agipdasicseams()**

Make a boolean array marking the double-width pixels in an AGIPD module

This returns a (512, 128) array with False for normal (square) pixels, and True for the 400 x 200 µm pixels at the horizontal joins between ASICs.

See **Making data masks** for an illustration of this.
2.1.2 AGIPD-500K2G

AGIPD-500K2G consists of 8 modules of 512x128 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.

![AGIPD-500K2G detector geometry (No file)](image)

Fig. 2: The approximate layout of AGIPD-500K2G, in a front view (looking along the beam).

class extra_geom.AGIPD_500K2GGeometry(modules, filename='No file')
Detector layout for AGIPD-500k

The coordinates used in this class are 3D (x, y, z), and represent metres.

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

classmethod from_origin(origin=(0, 0), asic_gap=2, panel_gap=(16, 30), unit=0.0002)
Generate an AGIPD-500K2G geometry from origin position.

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

The default origin (0, 0) of the coordinates is the bottom-right corner of the detector. If another coordinate is given as the origin, it is relative to the bottom-right corner. 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.
Returns a new geometry object.

\texttt{write\_crystfel\_geom}(\ast\texttt{args}, \ast\ast\texttt{kwargs})

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

**Parameters**

- **filename** (\texttt{str}) – Filename of the geometry file to write.
- **data\_path** (\texttt{str}) – Path to the group that contains the data array in the hdf5 file. Default: '/entry_1/instrument_1/detector_1/data'.
- **mask\_path** (\texttt{str}) – Path to the group that contains the mask array in the hdf5 file.
- **dims** (\texttt{tuple}) – Dimensions of the data. Extra dimensions, except for the defaults, should be added by their index, e.g. ('frame', 'modno', 0, 'ss', 'fs') for raw data. Default: ('frame', 'modno', 'ss', 'fs'). Note: the dimensions must contain frame, ss, fs.
- **adu\_per\_ev** (\texttt{float}) – ADU (analog digital units) per electron volt for the considered detector.
- **clen** (\texttt{float}) – Distance between sample and detector in meters
- **photon\_energy** (\texttt{float}) – Beam wavelength in eV

\texttt{get\_pixel\_positions}(\texttt{centre=True})

Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

\texttt{to\_distortion\_array}(allow\_negative\_xy=False)

Return distortion matrix for AGIPD500K detector, suitable for pyFAI.

**Parameters**

- **allow\_negative\_xy** (\texttt{bool}) – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

**Returns**

- **out** – Array of float 32 with shape (4096, 128, 4, 3). The dimensions mean:
  - 8192 = 8 modules * 512 pixels (slow scan axis)
  - 128 pixels (fast scan axis)
  - 4 corners of each pixel
  - 3 numbers for z, y, x

**Return type** \texttt{ndarray}

\texttt{plot\_data\_fast}(\texttt{data}, *\texttt{, axis\_units='px', frontview=True, ax=None, figsize=None, colorbar=True, **kwargs})

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** (\texttt{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.

• **ax** (*~matplotlib.axes.Axes* object, optional) – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.

• **figsize** (*tuple*) – Size of the figure (width, height) in inches to be drawn (default: (10, 10))

• **colorbar** (*bool, dict*) – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.

• **kwargs** – Additional keyword arguments passed to ~matplotlib.imshow

**position_modules_fast** (*data, out=None, threadpool=None*)

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.

• **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is allocated. Use output_array_for_position_fast() to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

• **threadpool** (*concurrent.futures.ThreadPoolExecutor, optional*) – If passed, parallelise copying data into the output image. By default, data for different tiles are copied serially. For a single 1 MPx image, the default appears to be faster, but for assembling a stack of several images at once, multithreading can help.

**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.

**output_array_for_position_fast** (*extra_shape=(), dtype=<class 'numpy.float32'>*)

Make an empty output array to use with position_modules_fast

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the *out* parameter to position_modules_fast(). By default, it allocates a new array on each call, which can be slow.

**Parameters**

• **extra_shape** (*tuple, optional*) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass extra_shape=(nframes,) to get a 3D output array.

• **dtype** (*optional (Default: np.float32)*) –

**position_modules_symmetric** (*data, out=None, threadpool=None*)

Assemble data with the centre in the middle of the output array.
The assembly process is the same as `position_modules_fast()`, aligning each module to a single pixel grid. But this makes the output array symmetric, with the centre at (height // 2, width // 2).

**Parameters**

- **data** *(ndarray)* – The last three dimensions should match the modules, then the slow scan and fast scan pixel dimensions.
- **out** *(ndarray, optional)* – An output array to assemble the image into. By default, a new array is created at the minimum size to allow symmetric assembly. If an array is passed in, its last two dimensions must be at least this size.
- **threadpool** *(concurrent.futures.ThreadPoolExecutor, optional)* – If passed, parallelise copying data into the output image. See `position_modules_fast()` for details.

**Returns**

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

**Return type** ndarray

**inspect**(axis_units='px', frontview=True)

Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

**Parameters**

- **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.

**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** *(DetectorGeometryBase)* – A second geometry object to compare with this one. It should be for the same kind of detector.
- **scale** *(float)* – Scale the arrows showing the difference in positions. This is useful to show small differences clearly.

**data_coords_to_positions**(module_no, slow_scan, fast_scan)

Convert data array coordinates to physical positions

Data array coordinates are how you might refer to a pixel in an array of detector data: module number, and indices in the slow-scan and fast-scan directions. But coordinates in the two pixel dimensions aren’t necessarily integers, e.g. if they refer to the centre of a peak.

module_no, fast_scan and slow_scan should all be numpy arrays of the same shape. module_no should hold integers, starting from 0, so 0: Q1M1, 1: Q1M2, etc.

slow_scan and fast_scan describe positions within that module. They may hold floats for sub-pixel positions. In both, 0.5 is the centre of the first pixel.

Returns an array of similar shape with an extra dimension of length 3, for (x, y, z) coordinates in metres.

**See also:**

*AGIPD-1M Geometry* demonstrates using this method.
The `agipd_asic_seams()` also applies to AGIPD-500K2G.

### 2.1.3 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.

![LPD-1M Diagram](image)

**Fig. 3:** The approximate layout of LPD-1M, in a front view (looking along the beam).

```python
class extra_geom.LPD_1MGeometry(modules, filename='No file')
Detector layout for LPD-1M

The coordinates used in this class are 3D (x, y, z), and represent metres.

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.

- `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.

**classmethod from_crystfel_geom** *(filename)*

Read a CrystFEL format (.geom) geometry file.

Returns a new geometry object.

**offset** *(shift, *, modules=slice(None, None, None), tiles=slice(None, None, None))*

Move part or all of the detector, making a new geometry.

By default, this moves all modules & tiles. To move the centre down in the image, move the whole geometry up relative to it.

Returns a new geometry object of the same type.

```python
# Move the whole geometry up 2 mm (relative to the beam)
geom2 = geom.shift((0, 2e-3))

# Move quadrant 1 (modules 0, 1, 2, 3) up 2 mm
```


```python
geom2 = geom.shift((0, 2e-3), modules=np.s_[0:4])
```

# Move each module by a separate amount

```python
shifts = np.zeros((16, 3))
shifts[5] = (0, 2e-3, 0)  # x, y, z for individual modules
shifts[10] = (0, -1e-3, 0)
geom2 = geom.shift(shifts)
```

**Parameters**

- **shift** *(numpy.ndarray or tuple)* – (x, y) or (x, y, z) shift to apply in metres. Can be a single shift for all selected modules, a 2D array with a shift per module, or a 3D array with a shift per tile (arr[module, tile, xyz]).

- **modules** *(slice)* – Select modules to move; defaults to all modules. Like all Python slicing, the end number is excluded, so `np.s_[:4]` moves modules 0, 1, 2, 3.

- **tiles** *(slice)* – Select tiles to move within each module; defaults to all tiles.

**quad_positions** *(h5_file=None)*

Get the positions of the 4 quadrants

Quadrant positions are returned as (x, y) coordinates in millimetres. Their meaning is as in `from_h5_file_and_quad_positions()`.

To use the returned positions with an existing XFEL HDF5 geometry file, the path to that file should be passed in. In that case, the offsets of M4 T16 in each quadrant are read from the file to calculate a suitable quadrant position. The geometry in the file is not checked against this geometry object at all.

**to_h5_file_and_quad_positions** *(path)*

Write this geometry to an XFEL HDF5 format geometry file

The file and quadrant positions are not stored in the file, so they are returned separately. These and the numbers in the file are in millimetres.

The file and quadrant positions produced by this method are compatible with `from_h5_file_and_quad_positions()`.

**write_crystfel_geom** *(filename, *, data_path='/entry_1/instrument_1/detector_1/data', mask_path=None, dims=('frame', 'modno', 'ss', 'fs'), nquads=4, adu_per_ev=None, clen=None, photon_energy=None)*

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

**Parameters**

- **filename** *(str)* – Filename of the geometry file to write.

- **data_path** *(str)* – Path to the group that contains the data array in the hdf5 file. Default: '/entry_1/instrument_1/detector_1/data'.

- **mask_path** *(str)* – Path to the group that contains the mask array in the hdf5 file.

- **dims** *(tuple)* – Dimensions of the data. Extra dimensions, except for the defaults, should be added by their index, e.g. ('frame', 'modno', 0, 'ss', 'fs') for raw data. Default: ('frame', 'modno', 'ss', 'fs'). Note: the dimensions must contain frame, ss, fs.

- **adu_per_ev** *(float)* – ADU (analog digital units) per electron volt for the considered detector.

- **clen** *(float)* – Distance between sample and detector in meters

---

2.1. Detector geometry reference
• **photon_energy** *(float)* – Beam wave length in eV

### get_pixel_positions *(centre=True)*
Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

### to_distortion_array *(allow_negative_xy=False)*
Return distortion matrix for LPD detector, suitable for pyFAI.

**Parameters**
- **allow_negative_xy** *(bool)* – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

**Returns**
- **out** – Array of float 32 with shape (4096, 256, 4, 3). The dimensions mean:
  - 4096 = 16 modules * 256 pixels (slow scan axis)
  - 256 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, ax=None, figsize=None, colorbar=True, **kwargs)*
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.
- **ax** *(~matplotlib.axes.Axes object, optional)* – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.
- **figsize** *(tuple)* – Size of the figure (width, height) in inches to be drawn (default: (10, 10))
- **colorbar** *(bool, dict)* – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.
- **kwargs** – Additional keyword arguments passed to ~matplotlib.imshow

### position_modules_fast *(data, out=None, threadpool=None)*
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.

• **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is allocated. Use `output_array_for_position_fast()` to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

• **threadpool** (*concurrent.futures.ThreadPoolExecutor, optional*) – If passed, parallelise copying data into the output image. By default, data for different tiles are copied serially. For a single 1 MPx image, the default appears to be faster, but for assembling a stack of several images at once, multithreading can help.

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.

`output_array_for_position_fast` (*extra_shape=*, *dtype=*`<class 'numpy.float32'>`)  
Make an empty output array to use with position_modules_fast.

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the `out=` parameter to `position_modules_fast()`. By default, it allocates a new array on each call, which can be slow.

Parameters

• **extra_shape** (*tuple, optional*) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass `extra_shape=(nframes,)` to get a 3D output array.

• **dtype** (*optional* *Default: np.float32*) –

`position_modules_symmetric` (*data*, *out=None*, *threadpool=None*)  
Assemble data with the centre in the middle of the output array.

The assembly process is the same as `position_modules_fast()`, aligning each module to a single pixel grid. But this makes the output array symmetric, with the centre at (height // 2, width // 2).

Parameters

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

• **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is created at the minimum size to allow symmetric assembly. If an array is passed in, its last two dimensions must be at least this size.

• **threadpool** (*concurrent.futures.ThreadPoolExecutor, optional*) – If passed, parallelise copying data into the output image. See `position_modules_fast()` for details.

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

Return type *ndarray*

`inspect` (*axis_units='px', frontview=True*)  
Plot the 2D layout of this detector geometry.
Returns a matplotlib Axes object.

Parameters

- **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.

**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** *(DetectorGeometryBase)* – A second geometry object to compare with this one. It should be for the same kind of detector.
- **scale** *(float)* – Scale the arrows showing the difference in positions. This is useful to show small differences clearly.

**data_coords_to_positions**(module_no, slow_scan, fast_scan)
Convert data array coordinates to physical positions

Data array coordinates are how you might refer to a pixel in an array of detector data: module number, and indices in the slow-scan and fast-scan directions. But coordinates in the two pixel dimensions aren’t necessarily integers, e.g. if they refer to the centre of a peak.

module_no, fast_scan and slow_scan should all be numpy arrays of the same shape. module_no should hold integers, starting from 0, so 0: Q1M1, 1: Q1M2, etc.

slow_scan and fast_scan describe positions within that module. They may hold floats for sub-pixel positions. In both, 0.5 is the centre of the first pixel.

Returns an array of similar shape with an extra dimension of length 3, for (x, y, z) coordinates in metres.

See also:

AGIPD-1M Geometry demonstrates using this method.

2.1.4 DSSC-1M

DSSC-1M consists of 16 modules of 128×512 pixels each. Each module is further subdivided into 2 sensor tiles, which this geometry code can position independently.

The pixels in each DSSC module are tesselating hexagons. This is handled in get_pixel_positions() and to_distortion_array(), but assembling an image treats the pixels as rectangles to simplify processing. This is adequate for previewing detector images, but some pixels will be approximately half a pixel width from their true position.

**class** extra_geom.DSSC_1MGeometry(modules, filename='No file')
Detector layout for DSSC-1M

The coordinates used in this class are 3D (x, y, z), and represent metres.

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

**classmethod** from_h5_file_and_quad_positions(path, positions, unit=0.001)
Load a DSSC geometry from an XFEL HDF5 format geometry file
Fig. 4: The approximate layout of DSSC-1M, in a front view (looking along the beam).
The quadrant positions are not stored in the file, and must be provided separately. The position given should refer to the bottom right (looking along the beam) corner of the quadrant.

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 DSSC.
- **positions**(list of 2-tuples) – (x, y) coordinates of the corner of each quadrant (the one with lowest x and y coordinates).
- **unit**(float, optional) – The conversion factor to put the coordinates into metres. The default 1e-3 means the numbers are in millimetres.

**offset**(shift, *, modules=slice(None, None, None), tiles=slice(None, None, None))

Move part or all of the detector, making a new geometry.

By default, this moves all modules & tiles. To move the centre down in the image, move the whole geometry *up* relative to it.

Returns a new geometry object of the same type.
# Move the whole geometry up 2 mm (relative to the beam)
geom2 = geom.shift((0, 2e-3))

# Move quadrant 1 (modules 0, 1, 2, 3) up 2 mm
geom2 = geom.shift((0, 2e-3), modules=np.s_[:4])

# Move each module by a separate amount
shifts = np.zeros((16, 3))
shifts[5] = (0, 2e-3, 0)  # x, y, z for individual modules
shifts[10] = (0, -1e-3, 0)
geom2 = geom.shift(shifts)

Parameters

- **shift** *(numpy.ndarray or tuple)* – (x, y) or (x, y, z) shift to apply in metres. Can be a single shift for all selected modules, a 2D array with a shift per module, or a 3D array with a shift per tile *(arr[module, tile, xyz])*.

- **modules** *(slice)* – Select modules to move; defaults to all modules. Like all Python slicing, the end number is excluded, so np.s_[:4] moves modules 0, 1, 2, 3.

- **tiles** *(slice)* – Select tiles to move within each module; defaults to all tiles.

**quad_positions** *(h5_file=None)*
Get the positions of the 4 quadrants

Quadrant positions are returned as (x, y) coordinates in millimetres. Their meaning is as in `from_h5_file_and_quad_positions()`.

To use the returned positions with an existing XFEL HDF5 geometry file, the path to that file should be passed in. In that case, the offsets of M1 T1 in each quadrant are read from the file to calculate a suitable quadrant position. The geometry in the file is not checked against this geometry object at all.

**to_h5_file_and_quad_positions** *(path)*
Write this geometry to an XFEL HDF5 format geometry file

The quadrant positions are not stored in the file, so they are returned separately. These and the numbers in the file are in millimetres.

The file and quadrant positions produced by this method are compatible with `from_h5_file_and_quad_positions()`.

**get_pixel_positions** *(centre=True)*
Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

**to_distortion_array** *(allow_negative_xy=False)*
Return distortion matrix for DSSC detector, suitable for pyFAI.

Parameters **allow_negative_xy** *(bool)* – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

Returns

- **out** – Array of float 32 with shape (2048, 512, 6, 3). The dimensions mean:
  - 2048 = 16 modules * 128 pixels (slow scan axis)
- 512 pixels (fast scan axis)
- 6 corners of each pixel
- 3 numbers for z, y, x

**Return type**  ndarray

```python
plot_data_fast(data, *, axis_units='px', frontview=True, ax=None, figsize=None, colorbar=False, **kwargs)
```

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

**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.
- **ax (matplotlib.axes.Axes object, optional)** – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.
- **figsize (tuple)** – Size of the figure (width, height) in inches to be drawn (default: (10, 10))
- **colorbar (bool, dict)** – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.
- **kwargs** – Additional keyword arguments passed to `~matplotlib.imshow`

```python
position_modules_fast(data, out=None, threadpool=None)
```

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.
- **out (ndarray, optional)** – An output array to assemble the image into. By default, a new array is allocated. Use `output_array_for_position_fast()` to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.
- **threadpool (concurrent.futures.ThreadPoolExecutor, optional)** – If passed, parallelise copying data into the output image. By default, data for different tiles are copied serially. For a single 1 MPx image, the default appears to be faster, but for assembling a stack of several images at once, multithreading can help.

**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.
output_array_for_position_fast (extra_shape=(), dtype=<class 'numpy.float32'>)
Make an empty output array to use with position_modules_fast
You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the out= parameter to position_modules_fast(). By default, it allocates a new array on each call, which can be slow.

Parameters
• extra_shape (tuple, optional) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass extra_shape=(nframes,) to get a 3D output array.
• dtype (optional (Default: np.float32)) –

inspect (axis_units='px', frontview=True)
Plot the 2D layout of this detector geometry.
Returns a matplotlib Axes object.

Parameters
• 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.

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 (DetectorGeometryBase) – A second geometry object to compare with this one. It should be for the same kind of detector.
• scale (float) – Scale the arrows showing the difference in positions. This is useful to show small differences clearly.

2.1.5 JUNGFRAU

JUNGFRAU detectors can be made with varying numbers of 512×1024 pixel modules. Each module is further subdivided into 8 sensor tiles.

Note: Reading & writing geometry files for JUNGFRAU is not yet implemented.

class extra_geom.JUNGFRAUGeometry (modules, filename='No file')
Detector layout for flexible Jungfrau arrangements
The base JUNGFRAU unit (and rigid group) in combined arrangements is the JF-500K module, which is an independent detector unit of 2 x 4 ASIC tiles.
In the default orientation, the slow-scan dimension is y and the fast-scan dimension is x, so the data shape for one module is (y, x).

classmethod from_module_positions (offsets=((0, 0)), orientations=None, asic_gap=2, unit=7.5e-05)
Generate a Jungfrau geometry object from module positions
Parameters

• offsets (iterable of tuples) – iterable of length n_modules containing a coordinate tuple (x,y) for each offset to the global origin. Coordinates are in pixel units by default.

These offsets are positions for the bottom, beam-left corner of each module, regardless of its orientation.

• orientations (iterable of tuples) – list of length n_modules containing a unit-vector tuple (x,y) for each orientation wrt. the axes

Orientations default to (1,1) for each module if this optional keyword argument is lacking; if not, the number of elements must match the number of modules as per offsets

•asic_gap (float) – The gap between the 8 ASICs within each module. This is in pixel units by default.

• unit (float) – The unit for offsets andasic_gap, in metres. Defaults to the pixel size (75 um).

get_pixel_positions (centre=True)

Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

to_distortion_array (allow_negative_xy=False)

Generate a distortion array for pyFAI from this geometry.

plot_data_fast (data, *, axis_units='px', frontview=True, ax=None, figsize=None, colorbar=True, **kwargs)

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.

• ax (~matplotlib.axes.Axes object, optional) – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.

• figsize (tuple) – Size of the figure (width, height) in inches to be drawn (default: (10, 10))

• colorbar (bool, dict) – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.

• kwargs – Additional keyword arguments passed to ~matplotlib.imshow

position_modules_fast (data, out=None, threadpool=None)

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.

- **out** *(ndarray, optional)* – An output array to assemble the image into. By default, a new array is allocated. Use `output_array_for_position_fast()` to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

- **threadpool** *(concurrent.futures.ThreadPoolExecutor, optional)* – If passed, parallelise copying data into the output image. By default, data for different tiles are copied serially. For a single 1 MPx image, the default appears to be faster, but for assembling a stack of several images at once, multithreading can help.

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.

`output_array_for_position_fast(extra_shape=(), dtype=<class 'numpy.float32'>)`

Make an empty output array to use with position_modules_fast

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the `out=` parameter to `position_modules_fast()`. By default, it allocates a new array on each call, which can be slow.

Parameters

- **extra_shape** *(tuple, optional)* – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass `extra_shape=(nframes,)` to get a 3D output array.

- **dtype** *(optional (Default: np.float32))*

`inspect(axis_units='px', frontview=True)`

Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

Parameters

- **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.

`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** *(DetectorGeometryBase)* – A second geometry object to compare with this one. It should be for the same kind of detector.

- **scale** *(float)* – Scale the arrows showing the difference in positions. This is useful to show small differences clearly.
2.2 Performance notes

These are some notes on how to load and process data efficiently.

2.2.1 Reduce before assembling

Assembling detector images (see Detector geometry reference) is relatively slow. If your analysis involves a reduction step like summing or averaging over a number of images, try to do this on the data from separate modules before assembling them into images.

This also applies more generally: if a step in your processing makes the data smaller, you want to do that step as near the start as possible.

2.3 AGIPD-1M Geometry

The AGIPD-1M detector, which is used at the SPB & MID experiments, consists of 16 modules of 512×128 pixels each. Each module is further divided into 16 ASICs, but these are arranged in pairs, so we treat a module as 8 tiles.

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

```
[1]: %matplotlib inline
    from extra_geom import AGIPD_1MGeometry

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.

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

[3]: geom.inspect()

[3]: <matplotlib.axes._subplots.AxesSubplot at 0x2b4e041f9588>
```
We can also load AGIPD-1M geometry from a CrystFEL format geometry file. These are text based files, often saved with a .geom extension. EXtra-geom expects the tiles in the file to be named \texttt{p0a0} (panel 0, ASIC 0) up to \texttt{p15a7}.

[15]: \[\texttt{# Create a CrystFEL format geometry file to demonstrate loading it}
\texttt{geom.write_crystfel_geom('agipd_example.geom', clen=0.3, adu_per_ev=1)}\]

[16]: \[\texttt{# Load geometry from CrystFEL format file}
\texttt{geom2 = AGIPD_1MGeometry.from_crystfel_geom('agipd_example.geom')}
\texttt{geom2.inspect()}\]

[16]: \[\texttt{<matplotlib.axes._subplots.AxesSubplot at 0x2b4e0ace8be0>}\]
The code above creates a geometry object for AGIPD-1M. See these examples for how to use a geometry object:

- Assemble images
- Convert positions in the data array into physical positions
- Create masks
2.4 LPD-1M geometry

The LPD-1M detector, used in FXE, consists of 16 supermodules of 256×256 pixels each. Each module is divided into 16 tiles.

```python
%matplotlib inline
from extra_geom import LPD_1MGeometry

We can generate a simple geometry from (x, y) coordinates of the top, beam-left (left looking along the beam) corner of each quadrant:

```python
quadpos = [(11.4, 299), (-11.5, 8), (254.5, -16), (278.5, 275)]  # in mm
g = LPD_1MGeometry.from_quad_positions(quadpos)
g.inspect()
```

```python
<matplotlib.axes._subplots.AxesSubplot at 0x2afb34c5af28>
```
We can also load geometry information within each quadrant from an EuXFEL HDF5 geometry file. These files do not store the quadrant positions, so we need to use it along with the quadrant positions, in the same format as above.

```python
# From March 18; converted to XFEL standard coordinate directions
geom2 = LPD_1MGeometry.from_h5_file_and_quad_positions('lpd_mar_18_axesfixed.h5', quadpos)
geom2.inspect()
```

Q2M2 was missing and not measured for this geometry file, so there’s a gap at the bottom right. In fact, all of the tiles for this module have ended up overlapped inside Q2M4.

The code above creates a geometry object for LPD-1M. See these examples for how to use a geometry object:

- **Assemble images**
- **Convert positions in the data array into physical positions**
• Create masks

2.5 DSSC detector geometry

The DSSC detector used at SCS consists of 16 modules of 128×512 pixels each. Each module consists of two tiles.

```
from extra_geom import DSSC_1MGeometry
```

We can load geometry information for DSSC from an EuXFEL HDF5 geometry file. These files do not store the quadrant positions, so we need to use it along with the quadrant positions referring to the bottom, beam-right (right looking along the beam) corner of each quadrant.

```
# Made up numbers!
quad_pos = [(-130, 5), (-130, -125), (5, -125), (5, 5), ]
path = 'dssc_geo_june19.h5'

g = DSSC_1MGeometry.from_h5_file_and_quad_positions(path, quad_pos)
```

```
g.inspect()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2abd47b6fe10>
```
The code above creates a geometry object for DSSC. See these examples for how to use a geometry object:

- *Assemble images*
- *Create masks*
2.5.1 Hexagonal pixels

DSSC has unusual hexagonal pixels. Assembling an image treats the pixels as uniform rectangles so that one input pixel is one output pixel, but the geometry object does know about the real shape and layout of the pixels.

Let’s have a close up look at some pixels in Q1M1. `get_pixel_positions()` gives us pixel centres. `to_distortion_array()` gives pixel corners in a slightly different format, suitable for PyFAI.

PyFAI requires non-negative x and y coordinates. But we want to plot them along with the centre positions, so we pass `allow_negative_xy=True` to get comparable coordinates.

```python
[4]:
   pixel_pos = g.get_pixel_positions()
   print("Pixel positions array shape:", pixel_pos.shape,
         "= (modules, slow_scan, fast_scan, x/y/z)"
   q1ml_centres = pixel_pos[0]
   cx = q1ml_centres[..., 0]
   cy = q1ml_centres[..., 1]

   distortn = g.to_distortion_array(allow_negative_xy=True)
   print("Distortion array shape:", distortn.shape,
         "= (modules * slow_scan, fast_scan, corners, z/y/x)"
   q1ml_corners = distortn[:128]

   print("Pixel positions array shape: (16, 128, 512, 3) = (modules, slow_scan, fast_scan, x/y/z)"
   print("Distortion array shape: (2048, 512, 6, 3) = (modules * slow_scan, fast_scan, corners, z/y/x)"

[5]:
   import matplotlib.pyplot as plt
   from matplotlib.patches import Polygon
   from matplotlib.collections import PatchCollection

   fig, ax = plt.subplots(figsize=(10, 10))
   hexes = []
   for ss_pxl in range(4):
       for fs_pxl in range(5):
           # Create hexagon
           corners = q1ml_corners[ss_pxl, fs_pxl]
           corners = corners[:, 1:][..., ::-1] # Drop z, flip x & y
           hexes.append(Polygon(corners))

           # Draw text label near the pixel centre
           ax.text(cx[ss_pxl, fs_pxl], cy[ss_pxl, fs_pxl],
                   '{} {}'.format(ss_pxl, fs_pxl),
                   verticalalignment='bottom', horizontalalignment='left')

           # Add the hexagons to the plot
           pc = PatchCollection(hexes, facecolor=(0.75, 1.0, 0.75), edgecolor='k')
           ax.add_collection(pc)

           # Plot the pixel centres
           ax.scatter(cx[:5, :6], cy[:5, :6], marker='x')

           # matplotlib is reluctant to show such a small area, so we need to set the limits
           ax.set_xlim(-0.007, -0.0085) # To match the convention elsewhere, draw x right-to-left
```

(continues on next page)
Every second row of pixels is offset half a pixel width relative to its neighbours.

### 2.6 Assembling detector data into images

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

This example focuses mainly on LPD data, but the same methods work for all supported detectors. The example loads data from EuXFEL HDF5 files using EXtra-data, but you can assemble an image from any stored or streamed data that can give a NumPy array of the correct shape.

```python
import numpy as np
import matplotlib.pyplot as plt
import h5py
from extra_data import RunDirectory, stack_detector_data
from extra_geom import LPD_1MGeometry

Load an example run containing LPD data:
```
```python
run = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/proc/r0007/')
run.info()
```

- # of trains: 507
- Duration: 0:00:50.7
- First train ID: 1487289920
- Last train ID: 1487290426

13 detector modules (FXE_DET_LPD1M-1)
- e.g. module FXE_DET_LPD1M-1 0 : 256 x 256 pixels
- FXE_DET_LPD1M-1/DET/0CH0:xtdf
  - 30 frames per train, up to 15210 frames total

0 instrument sources (excluding detectors):

0 control sources: (1 entry per train)

Find the first train with data from all the LPD modules in use:

```python
sel = run.select('*LPD1M-1/DET/*', 'image.data')
for tid, train_data in sel.trains(require_all=True):
    print(f"Found detector data in train {tid}"),
    break
```

```plaintext
Found detector data in train 1487289920
FXE_DET_LPD1M-1/DET/0CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/11CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/12CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/13CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/14CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/15CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/1CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/2CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/4CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/6CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/7CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/8CH0:xtdf (30, 256, 256)
FXE_DET_LPD1M-1/DET/9CH0:xtdf (30, 256, 256)
```

Extract the detector images into a single Numpy array:

```python
modules_data = stack_detector_data(train_data, 'image.data')
modules_data.shape
```

```plaintext
(30, 16, 256, 256)
```

Load the geometry from a file (see LPD geometry for more about this specific detector):

```python
# From March 18; converted to XFEL standard coordinate directions
quadpos = [(11.4, 299), (-11.5, 8), (254.5, -16), (278.5, 275)]  # mm
geom = LPD_1MGeometry.from_quad_positions(quadpos)
```

Reassemble and show a detector image using the geometry:
Reassemble detector data into a numpy array for further analysis. The areas without data have the special value `"nan"` to mark them as missing.
Let’s have a closer at the image from a single module. You can see where it’s divided up into tiles:

```python
[8]: plt.figure(figsize=(8, 8))
plt.imshow(modules_data[10, 2], vmin=0, vmax=3000, origin='lower')
```

```python
[8]: <matplotlib.image.AxesImage at 0x2b22dba700f0>
```
The geometry object has a `.split_tiles()` method which cuts the array up into tiles, and puts them in the order that the tiles are numbered.

```python
[9]: tiles = LPD_1MGeometry.split_tiles(modules_data[10, 2])
   plt.figure(figsize=(8, 8))
   plt.imshow(tiles[11], vmin=0, vmax=3000)
[9]: <matplotlib.image.AxesImage at 0x2b22dbac4c88>
```
2.6.1 AGIPD example

We can do the same with AGIPD data:

```python
agipd_run = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/proc/r0005/')
tid, train_data = agipd_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)
```

Create an **AGIPD geometry object**:

```python
from extra_geom import AGIPD_1MGeometry
agipd_geom = AGIPD_1MGeometry.from_quad_positions(quad_pos=[
    (-525, 625),
    (-550, -10),
    (520, -160),
    (542.5, 475),
])
```

And use it to assemble an image. The geometry object has the same methods as the LPD examples above.

```python
agipd_geom.plot_data_fast(stacked_pulse, vmin=0, vmax=1000)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xb22dbb21a90>
```
You can control the plot using keyword arguments for axis and colorbar. For example, to plot two images in the same figure:

```python
fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=2, figsize=(12, 7.5))
ax_cbar = fig.add_axes([0.15, 0.08, 0.7, 0.02])  # Create extra axes for the colorbar

# Plot a single pulse in the left axes
agipd_geom.plot_data_fast(stacked_pulse, vmin=0, vmax=1000, ax=ax0, colorbar={
    'cax': ax_cbar,
    'shrink': 0.6,
    'pad': 0.1,
    'orientation': 'horizontal'
})
ax0.set_title('11th pulse')

# Label the colorbar associated with the first image
for colorbar in ax0.images[0].colorbar
```
2.7 Converting array coordinates to physical positions

If you find features of interest in the data for individual modules, these are likely to be located in array coordinates - e.g. module 5, centred at 57.1 pixels across and 32.8 pixels up.

You can use EXtra-geom to convert these array coordinates into physical (x, y, z) positions in metres in the assembled detector. There is also a separate method to get (x, y, z) coordinates for every pixel - see Making masks for that.

This example uses an AGIPD geometry, but the same method should work for other supported detectors. However, this method does not yet work for DSSC, due to complications with its hexagonal pixels.

```python
import matplotlib.pyplot as plt
import numpy as np
from extra_geom import AGIPD_1MGeometry
```

[1]: Text(0.5, 1.0, 'Average of pulses in one train')
Coordinates within the module are described as slow-scan and fast-scan dimensions, referring to the order in which the data is stored. The ‘First row’ lines in the diagram above run along the fast-scan dimension for each module, with slow-scan = 0.

Let’s generate 16 points in array coordinates.

```python
# Place one point in each module
module_no = np.arange(0, 16)
```
# For AGIPD, slow-scan is the x dimension, increasing from the edges towards the centre
slow_scan = np.linspace(10, 500, num=16)
fast_scan = np.full(fill_value=40.1, shape=16)  # Fixed y position in each module

geom.data_coords_to_positions() converts these array positions into physical coordinates. The points we generated above are plotted on top of the detector layout below.

```python
positions = geom.data_coords_to_positions(module_no, slow_scan, fast_scan)
print("positions.shape =", positions.shape)  # (point, x/y/z)

ax = geom.inspect(axis_units='m')
px = positions[:, 0]
py = positions[:, 1]
ax.scatter(px, py);
positions.shape = (16, 3)
```

2.7. Converting array coordinates to physical positions
This example shows how to make pixel masks using simple expressions. These masks can be used on the data without assembling detector images, which is useful as assembly is relatively slow, and reduces accuracy slightly by rounding pixel coordinates.

This example uses AGIPD geometry, but the same technique should work for any supported detector.

```python
import matplotlib
import numpy as np
from extra_geom import AGIPD_1MGeometry
```
The `get_pixel_positions()` method gives (x, y, z) coordinates for the centre of each pixel. Here, as is typical, the z coordinates are zero (on the detector plane), so we’ll only use x and y.
A rectangular mask is defined by four limits. These numbers are in metres:

```
rect_mask = (0.01 < px) & (px < 0.05) & (-0.05 < py) & (py < -0.02)
```

This makes a mask which is true (1) inside the rectangle. Multiplying `mask * data` will zero out everything outside the selected area. If you need a mask where 0 indicates pixels to keep, invert it with `~mask`.

By plotting the mask itself like an image, we can see what area it includes.

```
def visualise_mask(mask_arr):
    return geom.plot_data_fast(
        # converting to float allows gaps to be distinguished as NaN.
        mask_arr.astype(float), colorbar=None, axis_units='m'
    )
visualise_mask(rect_mask)
```
We can make circular shapes by using Pythagoras’ theorem to get the distance from the centre to each point \( c = \sqrt{a^2 + b^2} \):

\[\text{6:} \quad \text{radius} = \text{np.sqrt(px**2 + py**2)} \]

\[\text{ring_mask} = (0.04 < \text{radius}) \& (\text{radius} < 0.05) \]

\[\text{visualise_mask(ring_mask)} \]

\[\text{6:} \quad <\text{AxesSubplot:xlabel='metres', ylabel='metres'}> \]
arctan2() converts x, y coordinates to angles in radians, which enables things like this:

```
[7]: angle = np.arctan2(py, px)

wedge_mask = (np.pi * 5/8 < angle) & (angle < np.pi * 7/8)
visualise_mask(wedge_mask)
```

```
[7]: <AxesSubplot:xlabel='metres', ylabel='metres'>
```
We can combine masks with \& (intersection) and | (union) to create more complex shapes:

```python
[8]: complex_mask = (ring_mask & wedge_mask) | rect_mask
visualise_mask(complex_mask)
```

```
<AxesSubplot:xlabel='metres', ylabel='metres'>
```
This mask only includes data in four detector modules. If we’re selecting that data, we might be able to skip loading the data from the other modules. We can check which module numbers the mask includes:

```python
[9]: modules_included = np.any(complex_mask, axis=(1, 2))
modules_included.nonzero()[0]

[9]: array([2, 3, 8, 9])
```
### 2.8.1 Mask AGIPD wide pixels

AGIPD modules contain double width pixels at the boundaries between ASICs. Being larger, these catch more photons, so they can affect results. EXtra-geom contains a function to select them for masking:

```python
[10]: from extra_geom import agipd_asic_seams

[11]: # Get the mask, and repeat it for 16 modules
module_mask = agipd_asic_seams()
all_modules_mask = np.repeat(module_mask[np.newaxis], 16, axis=0)

ax = visualise_mask(all_modules_mask)

# Zoom in to see the masked edges
ax.set_xlim(0.04, 0)
ax.set_ylim(0, 0.04)

[11]: (0.0, 0.04)
2.9 Release Notes

2.9.1 1.1

- New `position_modules_symmetric()` method to assemble data with the detector centre at the midpoint of the output array (PR #31).
- New `offset()` method to move part or all of a geometry in 2 or 3 dimensions (PR #27).
- New function `agipd_asic_seams()` to select or mask the double-width pixels where AGIPD tiles touch.
- Examples in documentation rearranged and improve (PR #32, PR #36).
- CI moved to Github Actions (PR #34) and integrated with Dependabot to control new versions of dependencies (PR #35).
2.9.2 1.0

- Added support for AGIPD ‘mini-half’ detector (8 modules) - see AGIPD-500K2G (PR #26).
- Added methods to write XFEL HDF5 geometry files and get quadrant positions from geometry objects (PR #24).
- Fixed y-axis scale in metres for plotting DSSC data (PR #23).
- Faster image assembly with less overhead (PR #16).
- Allow parallel image assembly using a thread pool (PR #17), which can speed up assembling several images to a single 3D array.

2.9.3 0.10

- Added support for pnCCD detector (PR #13).

2.9.4 0.9

- Initial support for JUNGFRAU detectors (PR #6).
- Fix compare() method to draw arrows the right size (PR #4).
- New example showing how to construct masks: Making data masks (PR #1).
- Correct code in LPD_1MGeometry.from_h5_file_and_quad_positions() which was working only by numeric coincidence (PR #7).

2.9.5 0.8

First separated version. No functional changes from karabo_data 0.7.

2.9.6 Earlier history

The code in EXtra-geom was previously released as karabo_data, up to version 0.7. See the karabo_data release notes for changes before the separation.

See also:

Data Analysis at European XFEL
CHAPTER
THREE

INDICES AND TABLES

- genindex
- search
e
extra_geom, 5
INDEX

A
AGIPD_1MGeometry (class in extra_geom), 5
AGIPD_500K2GGeometry (class in extra_geom), 11
agipd_asic_seams() (in module extra_geom), 10

C
compare() (extra_geom.AGIPD_1MGeometry method), 10
compare() (extra_geom.AGIPD_500K2GGeometry method), 14
compare() (extra_geom.DSSC_1MGeometry method), 25
compare() (extra_geom.JUNGFRAUGeometry method), 27
compare() (extra_geom.LPD_1MGeometry method), 20

D
data_coords_to_positions() (extra_geom.AGIPD_1MGeometry method), 10
data_coords_to_positions() (extra_geom.AGIPD_500K2GGeometry method), 14
data_coords_to_positions() (extra_geom.LPD_1MGeometry method), 20
DSSC_1MGeometry (class in extra_geom), 20

E
extra_geom module, 5

F
from_crystfel_geom() (extra_geom.AGIPD_1MGeometry class method), 5
from_crystfel_geom() (extra_geom.AGIPD_500K2GGeometry class method), 11
from_crystfel_geom() (extra_geom.LPD_1MGeometry class method), 16
from_h5_file_and_quad_positions() (extra_geom.DSSC_1MGeometry class method), 20
from_h5_file_and_quad_positions() (extra_geom.LPD_1MGeometry class method), 16
from_module_positions() (extra_geom.JUNGFRAUGeometry class method), 25
from_origin() (extra_geom.AGIPD_500K2GGeometry class method), 11
from_quad_positions() (extra_geom.AGIPD_1MGeometry class method), 5
from_quad_positions() (extra_geom.LPD_1MGeometry class method), 15

G
get_pixel_positions() (extra_geom.AGIPD_1MGeometry method), 8
get_pixel_positions() (extra_geom.AGIPD_500K2GGeometry method), 12
get_pixel_positions() (extra_geom.DSSC_1MGeometry method), 23
get_pixel_positions() (extra_geom.JUNGFRAUGeometry method), 26
get_pixel_positions() (extra_geom.LPD_1MGeometry method), 18

I
inspect() (extra_geom.AGIPD_1MGeometry method), 10
inspect() (extra_geom.AGIPD_500K2GGeometry method), 14
inspect() (extra_geom.DSSC_1MGeometry method), 25
inspect() (extra_geom.JUNGFRAUGeometry method), 27
inspect() (extra_geom.LPD_1MGeometry method), 19

J
JUNGFRAUGeometry (class in extra_geom), 25

L
LPD_1MGeometry (class in extra_geom), 15

M
module extra_geom, 5

O
offset() (extra_geom.AGIPD_1MGeometry method), 7
offset() (extra_geom.DSSC_1MGeometry method), 22
offset() (extra_geom.LPD_1MGeometry method), 16
output_array_for_position_fast() (extra_geom.AGIPD_1MGeometry method), 9
output_array_for_position_fast() (extra_geom.AGIPD_500K2GGeometry method), 13
output_array_for_position_fast() (extra_geom.DSSC_1MGeometry method), 24
output_array_for_position_fast() (extra_geom.JUNGFRAUGeometry method), 26
output_array_for_position_fast() (extra_geom.LPD_1MGeometry method), 19

P
plot_data_fast() (extra_geom.AGIPD_1MGeometry method), 8
plot_data_fast() (extra_geom.AGIPD_500K2GGeometry method), 12
plot_data_fast() (extra_geom.DSSC_1MGeometry method), 24
plot_data_fast() (extra_geom.JUNGFRAUGeometry method), 26
plot_data_fast() (extra_geom.LPD_1MGeometry method), 18
position_modules_fast() (extra_geom.AGIPD_1MGeometry method), 8
position_modules_fast() (extra_geom.AGIPD_500K2GGeometry method), 13
position_modules_fast() (extra_geom.DSSC_1MGeometry method), 24
position_modules_fast() (extra_geom.JUNGFRAUGeometry method), 26
position_modules_fast() (extra_geom.LPD_1MGeometry method), 18
position_modules_interpolate() (extra_geom.AGIPD_1MGeometry method), 9
position_modules_symmetric() (extra_geom.AGIPD_1MGeometry method), 9
position_modules_symmetric() (extra_geom.AGIPD_500K2GGeometry method), 13
position_modules_symmetric() (extra_geom.LPD_1MGeometry method), 19

Q
quad_positions() (extra_geom.AGIPD_1MGeometry method), 7
quad_positions() (extra_geom.DSSC_1MGeometry method), 23
quad_positions() (extra_geom.LPD_1MGeometry method), 17

to_distortion_array() (extra_geom.AGIPD_1MGeometry method), 8
to_distortion_array() (extra_geom.AGIPD_500K2GGeometry method), 12
to_distortion_array() (extra_geom.DSSC_1MGeometry method), 23
to_distortion_array() (extra_geom.JUNGFRAUGeometry method), 26
to_distortion_array() (extra_geom.LPD_1MGeometry method), 18
to_h5_file_and_quad_positions() (extra_geom.DSSC_1MGeometry method), 23
to_h5_file_and_quad_positions() (extra_geom.LPD_1MGeometry method), 17
write_crystfel_geom() (extra_geom.AGIPD_1MGeometry method), 7
write_crystfel_geom() (extra_geom.AGIPD_500K2GGeometry method), 12
write_crystfel_geom() (extra_geom.LPD_1MGeometry method), 17