European XFEL Python data tools Documentation

Release 1.3.0

European XFEL

Aug 03, 2020

TUTORIALS AND EXAMPLES

EXtra-data is a Python library for accessing and working with data produced at European XFEL.

Note: EXtra-data is the new name for karabo_data. The code to work with detector geometry has been separated as EXtra-geom.

CHAPTER

ONE

INSTALLATION

EXtra-data is available on our Anaconda installation on the Maxwell cluster:

```
module load exfel exfel_anaconda3
```

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

```
pip install extra_data
```

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

CHAPTER

TWO

QUICKSTART

Open a run or a file - see *Opening files* for more:

```
from extra_data import open_run, RunDirectory, H5File

# Find a run on the Maxwell cluster
run = open_run(proposal=700000, run=1)

# Open a run with a directory path
run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0001")

# Open an individual file
file = H5File("RAW-R0017-DA01-S00000.h5")
```

After this step, you'll use the same methods to get data whether you opened a run or a file.

Load data into memory - see *Getting data by source & key* for more:

```
# Get a labelled array
arr = run.get_array("SA3_XTD10_PES/ADC/1:network", "digitizers.channel_4_A.raw.samples
↪")

# Get a pandas dataframe of 1D fields
df = run.get_dataframe(fields=[
    ("*_XGM/*", "*.i[xy]Pos"),
    ("*_XGM/*", "*.photonFlux")
])
```

Iterate through data for each pulse train - see *Getting data by train* for more:

```
for train_id, data in run.select("*/DET/*", "image.data").trains():
    mod0 = data["FXE_DET_LPD1M-1/DET/0CH0:xtdf"]["image.data"]
```

These are not the only ways to get data: *Reading data files* describes various other options.

CHAPTER

THREE

DOCUMENTATION CONTENTS

3.1 Reading data to analyse in memory

It's often quickest and easiest to load data into memory before analysing it.

Some types of data, especially from large pixel detectors, may be bigger than the available memory. Other examples show how to work with very large amounts of data. But the machines in the Maxwell cluster have 250–750 GB of memory, so you can use the simple approach for many cases.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import re
import xarray as xr

from extra_data import RunDirectory
```

3.1.1 Tabular data (with pandas)

A run at European XFEL is stored as a folder containing a number of files. We can open a run with EXtra-data:

[2]:
```
run = RunDirectory('/gpfs/exfel/exp/SA1/201830/p900025/raw/r0150/')
run.info()  # Show overview info about this data
```

```
# of trains:    3721
Duration:       0:06:12.1
First train ID: 142844490
Last train ID:  142848210

0 detector modules ()

2 instrument sources (excluding detectors):
  - SA1_XTD2_XGM/XGM/DOOCS:output
  - SPB_XTD9_XGM/XGM/DOOCS:output

2 control sources:
  - SA1_XTD2_XGM/XGM/DOOCS
  - SPB_XTD9_XGM/XGM/DOOCS
```

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.

pandas is a popular Python library for working with tabular data. We'll 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, as often used for selecting files on Unix platforms.

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

[3]:
```
df = run.get_dataframe(fields=[("*_XGM/*", "*.i[xy]Pos"), ("*_XGM/*", "*.photonFlux
↪")])
df.head()
```

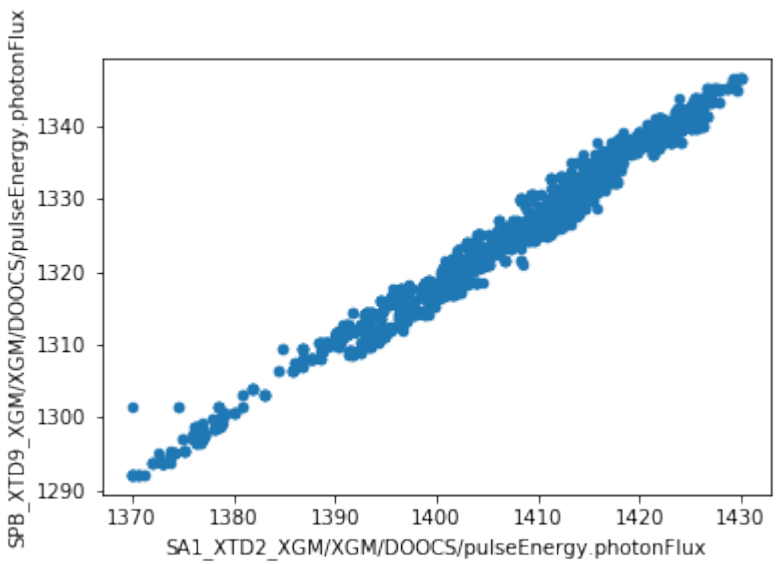
[3]:

	SPB_XTD9_XGM/XGM/DOOCS/beamPosition.iyPos	SPB_XTD9_XGM/XGM/DOOCS/beamPosition.ixPos	SPB_XTD9_XGM/XGM/DOOCS/pulseEnergy.photonFlux	SA1_XTD2_XGM/XGM/DOOCS/beamPosition.iyPos	SA1_XTD2_XGM/XGM/DOOCS/beamPosition.ixPos	SA1_XTD2_XGM/XGM/DOOCS/pulseEnergy.photonFlux
142844490	1.717195	-2.277912	1327.06958	0.161399	2.035218	1410.723755
142844491	1.717195	-2.277912	1327.06958	0.161399	2.035218	1410.137451
142844492	1.717195	-2.277912	1327.06958	0.161399	2.035218	1410.137451
142844493	1.717195	-2.277912	1327.06958	0.161399	2.035218	1410.137451
142844494	1.717195	-2.277912	1327.06958	0.161399	2.035218	1410.137451

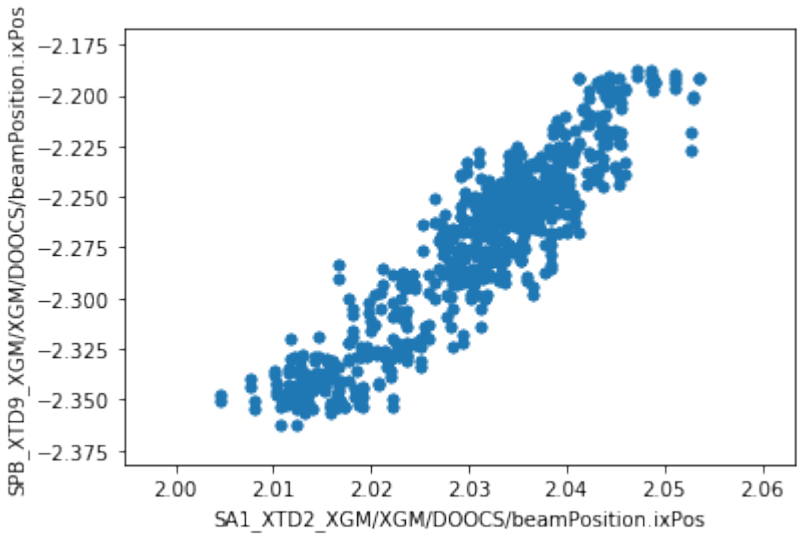
We can now make plots to compare the parameters at the two XGM positions. As expected, there's a strong correlation for each parameter.

[4]:
```
df.plot.scatter(
    x='SA1_XTD2_XGM/XGM/DOOCS/pulseEnergy.photonFlux',
    y='SPB_XTD9_XGM/XGM/DOOCS/pulseEnergy.photonFlux',
)
```

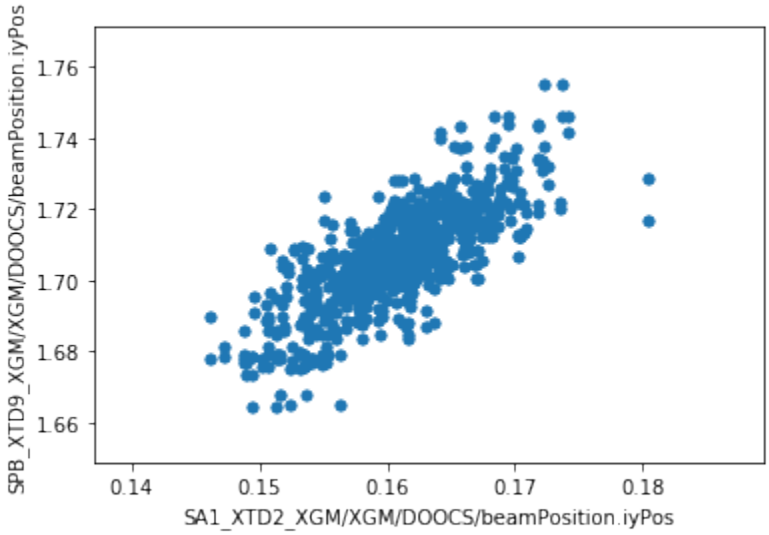
[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2ab99bb80fd0>
```



[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.

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

3.1.2 As arrays (with xarray)

We'll open a different run for this example:

[8]:
```
run = RunDirectory('/gpfs/exfel/exp/SA3/201830/p900027/raw/r0067/')
run.info()  # Show overview info about the data
```

```
# of trains:    1475
Duration:       0:02:27.5
First train ID: 128146446
Last train ID:  128147920

0 detector modules ()

1 instrument sources (excluding detectors):
  - SA3_XTD10_PES/ADC/1:network

11 control sources:
  - SA3_XTD10_PES/ADC/1
  - SA3_XTD10_PES/ASENS/MAGN_X
  - SA3_XTD10_PES/ASENS/MAGN_Y
  - SA3_XTD10_PES/ASENS/MAGN_Z
  - SA3_XTD10_PES/DCTRL/V30300S_NITROGEN
  - SA3_XTD10_PES/DCTRL/V30310S_NEON
  - SA3_XTD10_PES/DCTRL/V30320S_KRYPTON
  - SA3_XTD10_PES/DCTRL/V30330S_XENON
  - SA3_XTD10_PES/GAUGE/G30300F
  - SA3_XTD10_PES/GAUGE/G30310F
  - SA3_XTD10_PES/MCPS/MPOD
```

This run includes data from a Photo-Electron Spectrometer (PES), a monitoring device which records energy spectra for each train. Here's the data from one of its 16 spectrometers:

[9]:
```
run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_4_A.raw.samples')
```

[9]:
```
<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)
Coordinates:
  * trainId  (trainId) uint64 128146446 128146447 ... 128147919 128147920
Dimensions without coordinates: dim_0
```

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.

xarray extends numpy arrays with metadata about the dimensions: we use this to annotate the data with the pulse train IDs they relate to. This is important when correlating data from different sources, as each source may be missing data for some pulse trains, so we need to match them up based on train IDs. The xarray.align() function does this, and by specifying `join='inner'`, we keep only the trains which have data in both sets.

[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 the run to annotate the plot. This uses pandas - see the section above for more about that.

[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]:
```
SA3_XTD10_PES/MCPS/MPOD/channels.U200.measurementSenseVoltage   -0.101792
SA3_XTD10_PES/MCPS/MPOD/channels.U201.measurementSenseVoltage   -0.111782
SA3_XTD10_PES/MCPS/MPOD/channels.U202.measurementSenseVoltage   -0.106823
SA3_XTD10_PES/MCPS/MPOD/channels.U203.measurementSenseVoltage   -0.107910
Name: 128146446, dtype: float32
```

[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]
```

(continues on next page)

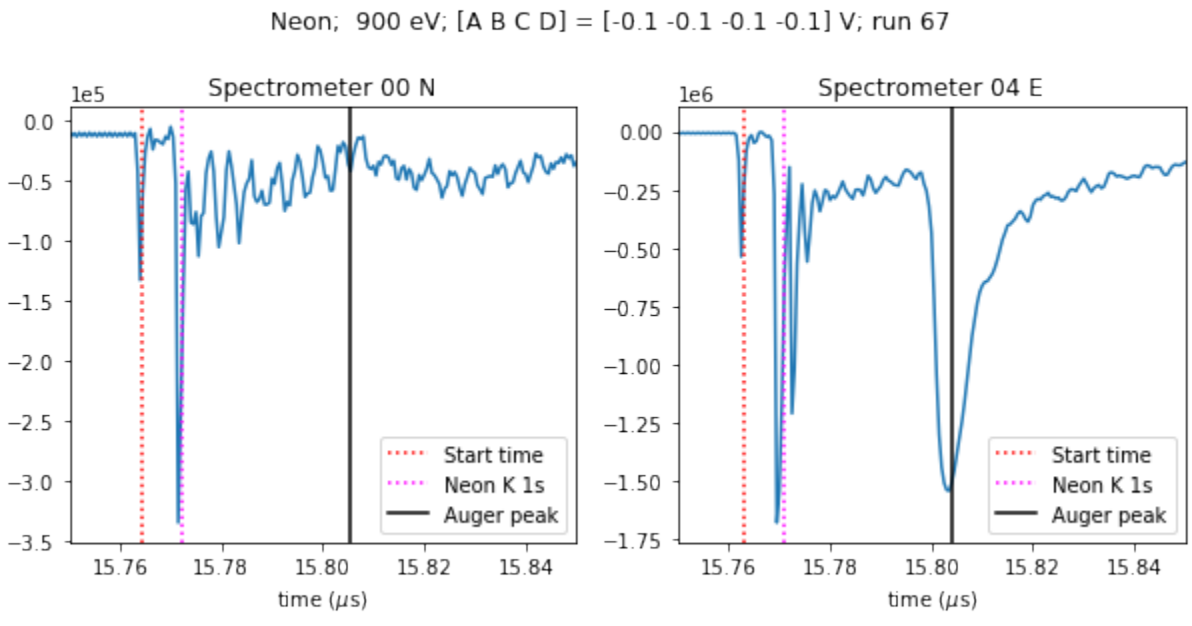
(continued from previous page)

```
    target_gas = re.search(r'(XENON|KRYPTON|NITROGEN|NEON)', key).group(1).title()
else:
    target_gas = 'No gas'
```

```
SA3_XTD10_PES/DCTRL/V30330S_XENON/interlock.AActionState       1
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
Name: 128146446, dtype: uint32
```

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

[13]:
```
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')
    ax.legend()

axes[0].set_title('Spectrometer 00 N')
axes[1].set_title('Spectrometer 04 E')
fig.suptitle('{gas};  900 eV; [A B C D] = [{voltages[0]:.1f} {voltages[1]:.1f}
↪{voltages[2]:.1f} {voltages[3]:.1f}] V; run 67'
             .format(gas=target_gas, voltages=electr_voltages.values), y=1.05);
```



The spectra look different because the beam is horizontally polarised, so the E spectrometer sees a peak that the N spectrometer doesn't.

3.2 Inspecting available data

The `.info()` method provides an overview of the data in an opened run or file:

[1]:
```
from extra_data import RunDirectory

run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0010")
run.info()
```

```
# of trains:    579
Duration:       0:00:57.9
First train ID: 507096934
Last train ID:  507097512

16 detector modules (SPB_DET_AGIPD1M-1)
  e.g. module SPB_DET_AGIPD1M-1 0 : 512 x 128 pixels
  SPB_DET_AGIPD1M-1/DET/0CH0:xtdf
  250 frames per train, up to 144750 frames total

2 instrument sources (excluding detectors):
  - SA1_XTD2_XGM/XGM/DOOCS:output
  - SPB_XTD9_XGM/XGM/DOOCS:output

18 control sources:
  - ACC_SYS_DOOCS/CTRL/BEAMCONDITIONS
  - SA1_XTD2_ATT/MDL/MAIN
  - SA1_XTD2_MIRR-1/MOTOR/HMRY
  - SA1_XTD2_XGM/XGM/DOOCS
  - SPB_IRU_AGIPD1M/MOTOR/Z_STEPPER
  - SPB_IRU_AGIPD1M/PSC/HV
  - SPB_IRU_AGIPD1M/TSENS/H1_T_EXTHOUS
  - SPB_IRU_AGIPD1M/TSENS/H2_T_EXTHOUS
  - SPB_IRU_AGIPD1M/TSENS/Q1_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q2_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q3_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q4_T_BLOCK
  - SPB_IRU_AGIPD1M1/CTRL/MC1
  - SPB_IRU_AGIPD1M1/CTRL/MC2
  - SPB_IRU_VAC/GAUGE/GAUGE_FR_6
  - SPB_RR_SYS/MDL/BUNCH_PATTERN
  - SPB_RR_SYS/TSYS/X2TIMER2
  - SPB_XTD9_XGM/XGM/DOOCS
```

The *lsxfel command* can give similar information at the command line.

The train IDs included in the run are available as a simple list:

[2]:
```
print(run.train_ids[:10])
```

```
[507096934, 507096935, 507096936, 507096937, 507096938, 507096939, 507096940,␣
↪507096941, 507096942, 507096943]
```

And the source names are available as a set:

[3]:
```
run.all_sources
```

[3]:
```
frozenset({'ACC_SYS_DOOCS/CTRL/BEAMCONDITIONS',
           'SA1_XTD2_ATT/MDL/MAIN',
```

(continues on next page)

(continued from previous page)

```
            'SA1_XTD2_MIRR-1/MOTOR/HMRY',
            'SA1_XTD2_XGM/XGM/DOOCS',
            'SA1_XTD2_XGM/XGM/DOOCS:output',
            'SPB_DET_AGIPD1M-1/DET/0CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/10CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/11CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/12CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/13CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/14CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/15CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/1CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/2CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/3CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/4CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/5CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/6CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/7CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/8CH0:xtdf',
            'SPB_DET_AGIPD1M-1/DET/9CH0:xtdf',
            'SPB_IRU_AGIPD1M/MOTOR/Z_STEPPER',
            'SPB_IRU_AGIPD1M/PSC/HV',
            'SPB_IRU_AGIPD1M/TSENS/H1_T_EXTHOUS',
            'SPB_IRU_AGIPD1M/TSENS/H2_T_EXTHOUS',
            'SPB_IRU_AGIPD1M/TSENS/Q1_T_BLOCK',
            'SPB_IRU_AGIPD1M/TSENS/Q2_T_BLOCK',
            'SPB_IRU_AGIPD1M/TSENS/Q3_T_BLOCK',
            'SPB_IRU_AGIPD1M/TSENS/Q4_T_BLOCK',
            'SPB_IRU_AGIPD1M1/CTRL/MC1',
            'SPB_IRU_AGIPD1M1/CTRL/MC2',
            'SPB_IRU_VAC/GAUGE/GAUGE_FR_6',
            'SPB_RR_SYS/MDL/BUNCH_PATTERN',
            'SPB_RR_SYS/TSYS/X2TIMER2',
            'SPB_XTD9_XGM/XGM/DOOCS',
            'SPB_XTD9_XGM/XGM/DOOCS:output'})
```

You can see control and instrument sources separately, but for data analysis this distinction is often not important.

[4]:
```
assert run.all_sources == (run.control_sources | run.instrument_sources)
```

Within each source, the data is organised under keys. The `.keys_for_source()` method lists a source's keys:

[5]:
```
run.keys_for_source('SA1_XTD2_XGM/XGM/DOOCS:output')
```

[5]:
```
{'data.intensityAUXSa1TD',
 'data.intensityAUXSa3TD',
 'data.intensityAUXTD',
 'data.intensitySa1SigmaTD',
 'data.intensitySa1TD',
 'data.intensitySa3SigmaTD',
 'data.intensitySa3TD',
 'data.intensitySigmaTD',
 'data.intensityTD',
 'data.trainId',
 'data.xSa1SigmaTD',
 'data.xSa1TD',
 'data.xSa3SigmaTD',
 'data.xSa3TD',
```

(continues on next page)

(continued from previous page)

```
 'data.xSigmaTD',
 'data.xTD',
 'data.ySa1SigmaTD',
 'data.ySa1TD',
 'data.ySa3SigmaTD',
 'data.ySa3TD',
 'data.ySigmaTD',
 'data.yTD'}
```

Instrument sources may have multiple values recorded for each train, and may be missing data for some trains. You can see how many data points there are for each train with `.get_data_counts()`. E.g. for this AGIPD detector module, the counts are the number of frames in each train:

[6]:
```
run.get_data_counts('SPB_DET_AGIPD1M-1/DET/11CH0:xtdf', 'image.data')
```

[6]:
```
507096934      0
507096935      0
507096936      0
507096937      0
507096938      0
             ...
507097185    250
507097186    250
507097187    250
507097188    250
507097189    250
Length: 256, dtype: uint64
```

This method returns a pandas series. The index (the numbers shown on the left) are train IDs.

3.3 Reading data train by train

If the data you want to work with is too big to load into memory all at once, one simple alternative is to process data from one train at a time.

Other options such as *using Dask* may run faster, or make it easier to do certain kinds of processing. But code that iterates through the trains is probably easier to understand.

[1]:
```
from extra_data import open_run

run = open_run(proposal=700000, run=2)
run.info()  # Show overview info about this data
```

```
# of trains:    3392
Duration:       0:05:39.2
First train ID: 79726751
Last train ID:  79730142

16 detector modules (SPB_DET_AGIPD1M-1)
  e.g. module SPB_DET_AGIPD1M-1 0 : 512 x 128 pixels
  SPB_DET_AGIPD1M-1/DET/0CH0:xtdf
  64 frames per train, up to 217088 frames total

3 instrument sources (excluding detectors):
  - SA1_XTD2_XGM/XGM/DOOCS:output
```

(continues on next page)

(continued from previous page)

```
  - SPB_IRU_SIDEMIC_CAM:daqOutput
  - SPB_XTD9_XGM/XGM/DOOCS:output

13 control sources:
  - ACC_SYS_DOOCS/CTRL/BEAMCONDITIONS
  - SA1_XTD2_XGM/XGM/DOOCS
  - SPB_IRU_AGIPD1M/PSC/HV
  - SPB_IRU_AGIPD1M/TSENS/H1_T_EXTHOUS
  - SPB_IRU_AGIPD1M/TSENS/H2_T_EXTHOUS
  - SPB_IRU_AGIPD1M/TSENS/Q1_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q2_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q3_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q4_T_BLOCK
  - SPB_IRU_AGIPD1M1/CTRL/MC1
  - SPB_IRU_AGIPD1M1/CTRL/MC2
  - SPB_IRU_VAC/GAUGE/GAUGE_FR_6
  - SPB_XTD9_XGM/XGM/DOOCS
```

To iterate through the trains in this run, we need the `.trains()` method.

But first, it's always a good idea to select the sources and keys we want, so we don't waste time loading irrelevant data. Let's select the image data from all AGIPD modules:

[2]:
```
sel = run.select('SPB_DET_AGIPD1M-1/DET/*CH0:xtdf', 'image.data')
sel.all_sources
```

[2]:
```
frozenset({'SPB_DET_AGIPD1M-1/DET/0CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/10CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/11CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/12CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/13CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/14CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/15CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/1CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/2CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/3CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/4CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/5CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/6CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/7CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/8CH0:xtdf',
           'SPB_DET_AGIPD1M-1/DET/9CH0:xtdf'})
```

[3]:
```
for tid, data in sel.trains():
    print("Processing train", tid)
    print("Detctor data module 0 shape:", data['SPB_DET_AGIPD1M-1/DET/0CH0:xtdf'][
↪'image.data'].shape)

    break  # Stop after the first train to keep the demo quick
```

```
Processing train 79726751
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-3-630f9647c3c0> in <module>
      1 for tid, data in sel.trains():
      2     print("Processing train", tid)
```

(continues on next page)

(continued from previous page)

```
----> 3     print("Detctor data module 0 shape:", data['SPB_DET_AGIPD1M-1/DET/0CH0:
↪xtdf']['image.data'].shape)
      4
      5     break  # Stop after the first train to keep the demo quick

KeyError: 'image.data'
```

Oops, we're missing data for this detector module. We can use the `require_all=True` parameter to skip over trains where some modules are missing data.

[4]:
```
for tid, data in sel.trains(require_all=True):
    print("Processing train", tid)
    print("Detctor data module 0 shape:", data['SPB_DET_AGIPD1M-1/DET/0CH0:xtdf'][
↪'image.data'].shape)

    break  # Stop after the first train to keep the demo quick
```

```
Processing train 79726787
Detctor data module 0 shape: (64, 2, 512, 128)
```

The data for each train is organised in nested dictionaries: `data[source][key]`. As this is often used with multi-module detectors like AGIPD, the `stack_detector_data` function is a convenient way to combine data from multiple similar modules.

[5]:
```
from extra_data import stack_detector_data

for tid, data in sel.trains(require_all=True):
    print("Detctor data module 0 shape:", data['SPB_DET_AGIPD1M-1/DET/0CH0:xtdf'][
↪'image.data'].shape)
    stacked = stack_detector_data(data, 'image.data')
    print("Stacked data shape:", stacked.shape)

    break  # Stop after the first train to keep the demo quick
```

```
Detctor data module 0 shape: (64, 2, 512, 128)
Stacked data shape: (64, 2, 16, 512, 128)
```

There are also methods which can get one train in the same format, from either a train ID or an index within this data:

[6]:
```
tid, data = sel.train_from_id(79726787)
tid, data = sel.train_from_index(36)
```

3.4 Averaging detector data with Dask

We often want to average large detector data across trains, keeping the pulses within each train separate, so we have an average image for pulse 0, another for pulse 1, etc.

This data may be too big to load into memory at once, but using Dask we can work with it like a numpy array. Dask takes care of splitting the job up into smaller pieces and assembling the result.

[1]:
```
from extra_data import open_run

import dask.array as da
from dask.distributed import Client, progress
```

(continues on next page)

(continued from previous page)

```
from dask_jobqueue import SLURMCluster
import numpy as np
```

First, we use Dask-Jobqueue to talk to the Maxwell cluster.

[2]:
```
partition = 'exfel'  # For EuXFEL staff
#partition = 'upex'   # For users

cluster = SLURMCluster(
    queue=partition,
    local_directory='/scratch',  # Local disk space for workers to use

    # Resources per SLURM job (per node, the way SLURM is configured on Maxwell)
    # processes=16 runs 16 Dask workers in a job, so each worker has 1 core & 32 GB␣
↪RAM.
    processes=16, cores=16, memory='512GB',
)

# Get a notbook widget showing the cluster state
cluster
```

```
VBox(children=(HTML(value='<h2>SLURMCluster</h2>'), HBox(children=(HTML(value='\n<div>
↪\n  <style scoped>\n    ...
```

[3]:
```
# Submit 2 SLURM jobs, for 32 Dask workers
cluster.scale(32)
```

If the cluster is busy, you might need to wait a while for the jobs to start. The cluster widget above will update when they're running.

Next, we'll set Dask up to use those workers:

[4]:
```
client = Client(cluster)
print("Created dask client:", client)
```

```
Created dask client: <Client: scheduler='tcp://131.169.193.102:44986' processes=32␣
↪cores=32>
```

Now Dask is ready, let's open the run we're going to operate on:

[5]:
```
run = open_run(proposal=700000, run=2)
run.info()
```

```
# of trains:    3392
Duration:       0:05:39.2
First train ID: 79726751
Last train ID:  79730142

16 detector modules (SPB_DET_AGIPD1M-1)
  e.g. module SPB_DET_AGIPD1M-1 0 : 512 x 128 pixels
  SPB_DET_AGIPD1M-1/DET/0CH0:xtdf
  64 frames per train, up to 217088 frames total

3 instrument sources (excluding detectors):
  - SA1_XTD2_XGM/XGM/DOOCS:output
  - SPB_IRU_SIDEMIC_CAM:daqOutput
  - SPB_XTD9_XGM/XGM/DOOCS:output
```

(continues on next page)

(continued from previous page)

```
13 control sources: (1 entry per train)
  - ACC_SYS_DOOCS/CTRL/BEAMCONDITIONS
  - SA1_XTD2_XGM/XGM/DOOCS
  - SPB_IRU_AGIPD1M/PSC/HV
  - SPB_IRU_AGIPD1M/TSENS/H1_T_EXTHOUS
  - SPB_IRU_AGIPD1M/TSENS/H2_T_EXTHOUS
  - SPB_IRU_AGIPD1M/TSENS/Q1_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q2_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q3_T_BLOCK
  - SPB_IRU_AGIPD1M/TSENS/Q4_T_BLOCK
  - SPB_IRU_AGIPD1M1/CTRL/MC1
  - SPB_IRU_AGIPD1M1/CTRL/MC2
  - SPB_IRU_VAC/GAUGE/GAUGE_FR_6
  - SPB_XTD9_XGM/XGM/DOOCS
```

We're working with data from the AGIPD detector. In this run, it's recording 64 frames for each train - this is part of the info above.

We can get a dask array for each module. This doesn't load the data yet, but it knows what shape it is:

[6]:
```
run.get_dask_array('SPB_DET_AGIPD1M-1/DET/0CH0:xtdf', 'image.data')
```

[6]:
```
dask.array<concatenate, shape=(191872, 2, 512, 128), dtype=uint16, chunksize=(8192, 2,
↪ 512, 128), chunktype=numpy.ndarray>
```

Now, we'll define how we're going to average over trains for each module:

[7]:
```
def average_module(modno, run, pulses_per_train=64):
    source = f'SPB_DET_AGIPD1M-1/DET/{modno}CH0:xtdf'
    counts = run.get_data_counts(source, 'image.data')

    arr = run.get_dask_array(source, 'image.data')[:, :1]
    # Make a new dimension for trains
    arr_trains = arr.reshape(-1, pulses_per_train, 512, 128)
    if modno == 0:
        print("array shape:", arr.shape)  # frames, dummy, 512, 128
        print("Reshaped to:", arr_trains.shape)

    return arr_trains.mean(axis=0, dtype=np.float32)
```

[8]:
```
mod_averages = [
    average_module(i, run, pulses_per_train=64)
    for i in range(16)
]

mod_averages
```

```
array shape: (191872, 1, 512, 128)
Reshaped to: (2998, 64, 512, 128)
```

[8]:
```
[dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
 dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
 dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
 dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
```

(continues on next page)

(continued from previous page)

```
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>,
dask.array<mean_agg-aggregate, shape=(64, 512, 128), dtype=float32, chunksize=(64,␣
↪512, 128), chunktype=numpy.ndarray>]
```

[9]:
```
# Stack the averages into a single array
all_average = da.stack(mod_averages)
all_average
```

[9]:
```
dask.array<stack, shape=(16, 64, 512, 128), dtype=float32, chunksize=(1, 64, 512,␣
↪128), chunktype=numpy.ndarray>
```

Dask shows us what shape the result array will be, but so far, no real computation has happened. Now that we've defined what we want, let's tell Dask to compute it.

This will take a minute or two. If you're running it, scroll up to the Dask cluster widget and click the status link to see what it's doing.

[10]:
```
%%time
all_average_arr = all_average.compute()   # Get a concrete numpy array for the result
```

```
CPU times: user 29.5 s, sys: 1.99 s, total: 31.4 s
Wall time: 1min 50s
```

`all_average_arr` is a regular numpy array with our results. Here are the values from the corner of module 0, frame 0:

[11]:
```
print(all_average_arr[0, 0, :5, :5])
```

```
[[5172.2964 5027.5137 5265.5615 4804.2617 4851.1353]
 [5534.881  5434.519  5051.8687 4966.505  5019.865 ]
 [5271.6772 5522.6396 5437.736  5310.047  5110.2173]
 [5560.7173 5607.7104 4831.513  4956.135  5309.423 ]
 [5209.8374 5452.4673 5573.138  5163.399  4962.6494]]
```

Please shut down the cluster (or scale it down to 0 workers) if you won't be using it for a while. This releases the resources for other people.

[12]:
```
client.close()
cluster.close()
```

3.5 Parallel processing with a virtual HDF5 dataset

This example demonstrates splitting up some data to be processed by several worker processes, and collecting the results back together.

For this example, we'll use data from an XGM, and find the average intensity of each pulse across all the trains in the run. This doesn't actually need parallel processing: we can easily do it directly in the notebook. But the same techniques should work with much more data and more complex calculations.

[1]:
```
from extra_data import RunDirectory
import multiprocessing
import numpy as np
```

The data that we want is separated over these seven sequence files:

[2]:
```
!ls /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S*.h5
```

```
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00000.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00001.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00002.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00003.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00004.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00005.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00006.h5
```

[3]:
```
run = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/')
```

By making a virtual dataset, we can see the shape of it, as if it was one big numpy array:

[4]:
```
vds_filename = 'xgm_vds.h5'
xgm_vds = run.get_virtual_dataset(
    'SA1_XTD2_XGM/XGM/DOOCS:output', 'data.intensityTD',
    filename=vds_filename
)
xgm_vds
```

[4]:
```
<HDF5 dataset "intensityTD": shape (3391, 1000), type "<f4">
```

Let's read this into memory and calculate the means directly, to check our parallel calculations against. We can do this for this example because the calculation is simple and the data is small; it wouldn't be practical in real situations where parallelisation is useful.

These data are recorded in 32-bit floats, but to minimise rounding errors we'll tell numpy to give the results as 64-bit floats. Try re-running this example with 32-bit floats to see how much the results change!

[5]:
```
simple_mean = xgm_vds[:, :40].mean(axis=0, dtype=np.float64)
simple_mean.round(4)
```

[5]:
```
array([834.2744, 860.0754, 869.2637, 891.4351, 899.6227, 899.3759,
       900.3555, 899.1162, 898.4991, 904.4979, 910.5669, 914.1612,
       922.5737, 925.8734, 930.093 , 935.3124, 938.9643, 941.4609,
       946.1351, 950.6574, 951.855 , 954.2491, 956.6414, 957.5584,
       961.7528, 961.1457, 958.9655, 957.6415, 953.8603, 947.9236,
```

(continues on next page)

(continued from previous page)

```
        0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
        0.    ,   0.    ,   0.    ,   0.    ])
```

Now, we're going to define chunks of the data for each of 4 worker processes.

[6]:
```
N_proc = 4
cuts = [int(xgm_vds.shape[0] * i / N_proc) for i in range(N_proc + 1)]
chunks = list(zip(cuts[:-1], cuts[1:]))
chunks
```

[6]:
```
[(0, 847), (847, 1695), (1695, 2543), (2543, 3391)]
```

3.5.1 Using `multiprocessing`

This is the function we'll ask each worker process to run, adding up the data and returning a 1D numpy array.

We're using default arguments as a convenient way to copy the filename and the dataset path into the worker process.

[7]:
```
def sum_chunk(chunk, filename=vds_filename, ds_name=xgm_vds.name):
    start, end = chunk
    # Reopen the file in the worker process:
    import h5py, numpy
    with h5py.File(filename, 'r') as f:
        ds = f[ds_name]
        data = ds[start:end]  # Read my chunk

    return data.sum(axis=0, dtype=numpy.float64)
```

Using Python's `multiprocessing` module, we start four workers, farm the chunks out to them, and collect the results back.

[8]:
```
with multiprocessing.Pool(N_proc) as pool:
    res = pool.map(sum_chunk, chunks)
```

`res` is now a list of 4 arrays, containing the sums from each chunk. To get the mean, we'll add these up to get a grand total, and then divide by the number of trains we have data from.

[9]:
```
multiproc_mean = (np.stack(res).sum(axis=0, dtype=np.float64)[:40] / xgm_vds.shape[0])
np.testing.assert_allclose(multiproc_mean, simple_mean)

multiproc_mean.round(4)
```

[9]:
```
array([834.2744, 860.0754, 869.2637, 891.4351, 899.6227, 899.3759,
       900.3555, 899.1162, 898.4991, 904.4979, 910.5669, 914.1612,
       922.5737, 925.8734, 930.093 , 935.3124, 938.9643, 941.4609,
       946.1351, 950.6574, 951.855 , 954.2491, 956.6414, 957.5584,
       961.7528, 961.1457, 958.9655, 957.6415, 953.8603, 947.9236,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ])
```

3.5.2 Using SLURM

What if we need more power? The example above is limited to one machine, but we can use SLURM to spread the work over multiple machines on the Maxwell cluster.

This is massive overkill for this example calculation - we'll only use one CPU core for a fraction of a second on each machine. But we could do something similar for a much bigger problem.

[10]:
```
from getpass import getuser
import h5py
import subprocess
```

We'll write a Python script for each worker to run. Like the `sum_chunk` function above, this reads a chunk of data from the virtual dataset and sums it along the train axis. It saves the result into another HDF5 file for us to collect.

[11]:
```
%%writefile parallel_eg_worker.py
#!/gpfs/exfel/sw/software/xfel_anaconda3/1.1/bin/python
import h5py
import numpy as np
import sys

filename = sys.argv[1]
ds_name = sys.argv[2]
chunk_start = int(sys.argv[3])
chunk_end = int(sys.argv[4])
worker_idx = sys.argv[5]

with h5py.File(filename, 'r') as f:
    ds = f[ds_name]
    data = ds[chunk_start:chunk_end]  # Read my chunk

chunk_totals = data.sum(axis=0, dtype=np.float64)

with h5py.File(f'parallel_eg_result_{worker_idx}.h5', 'w') as f:
    f['chunk_totals'] = chunk_totals
```

```
Writing parallel_eg_worker.py
```

The Maxwell cluster is divided into various partitions for different groups of users. If you're running this as an external user, comment out the 'Staff' line below.

[12]:
```
partition = 'upex'   # External users
partition = 'exfel'  # Staff
```

Now we submit 4 jobs with the `sbatch` command:

[13]:
```
for i, (start, end) in enumerate(chunks):
    cmd = ['sbatch', '-p', partition, 'parallel_eg_worker.py', vds_filename, xgm_vds.
↪name, str(start), str(end), str(i)]
    print(subprocess.check_output(cmd))
```

```
b'Submitted batch job 2631813\n'
b'Submitted batch job 2631814\n'
b'Submitted batch job 2631815\n'
b'Submitted batch job 2631816\n'
```

We can use `squeue` to monitor the jobs running. Re-run this until all the jobs have disappeared, meaning they're finished.

[14]:
```
!squeue -u {getuser()}
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

Now, so long as all the workers succeeded, we can collect the results.

If any workers failed, you'll find tracebacks in `slurm-*.out` files in the working directory.

[15]:
```
res = []

for i in range(N_proc):
    with h5py.File(f'parallel_eg_result_{i}.h5', 'r') as f:
        res.append(f['chunk_totals'][:])
```

Now `res` is once again a list of 1D numpy arrays, representing the totals from each chunk. So we can finish the calculation as in the previous section:

[16]:
```
slurm_mean = np.stack(res).sum(axis=0)[:40] / xgm_vds.shape[0]
np.testing.assert_allclose(slurm_mean, simple_mean)

slurm_mean.round(4)
```

[16]:
```
array([834.2744, 860.0754, 869.2637, 891.4351, 899.6227, 899.3759,
       900.3555, 899.1162, 898.4991, 904.4979, 910.5669, 914.1612,
       922.5737, 925.8734, 930.093 , 935.3124, 938.9643, 941.4609,
       946.1351, 950.6574, 951.855 , 954.2491, 956.6414, 957.5584,
       961.7528, 961.1457, 958.9655, 957.6415, 953.8603, 947.9236,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ])
```

3.6 Accessing LPD data

The Large Pixel Detector (LPD) is made of 16 modules which record data separately. `extra_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 other examples, including *Reading data to analyse in memory* and *Reading data train by train*.

The example uses some empty sample files which are generated by this cell:

[1]:
```
!python3 -m extra_data.tests.make_examples
```

```
Written examples.
```

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

[2]:
```
from extra_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
```

[2]:
```
frozenset({'FXE_DET_LPD1M-1/DET/0CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/10CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/11CH0:xtdf',
```

(continues on next page)

(continued from previous page)

```
            'FXE_DET_LPD1M-1/DET/12CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/13CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/14CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/15CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/1CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/2CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/3CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/4CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/5CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/6CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/7CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/8CH0:xtdf',
            'FXE_DET_LPD1M-1/DET/9CH0:xtdf',
            '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'})
```

Normal access methods give us each module separately:

[3]:
```
data_module0 = run.get_array('FXE_DET_LPD1M-1/DET/0CH0:xtdf', 'image.data')
data_module0.shape
```

[3]:
```
(384, 1, 256, 256)
```

The class `extra_data.components.LPD1M` can piece these together:

[4]:
```
from extra_data.components import LPD1M
lpd = LPD1M(run)
lpd
```

[4]:
```
<LPD1M: Data interface for detector 'FXE_DET_LPD1M-1' with 16 modules>
```

[5]:
```
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:

[6]:
```
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')
```

(continues on next page)

(continued from previous page)

```
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
  * module   (module) int64 2 7 8 13
Dimensions without coordinates: slow_scan, fast_scan
```

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:

[7]:
```
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']
```

(continues on next page)

(continued from previous page)

```
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)
```

[ ]:

3.7 Combining data from separate but concurrent runs

Here we will look at XGM (X-ray Gas Monitor) data that was recorded in the same short time interval, but in different parts of EuXFEL. We will compare an XGM in SASE1 (XTD2) to another one in SASE3 (XTD10). These data were stored in two different runs, belonging to two different proposals.

Conceptually, this section makes use of the data-object format *xarray.DataArray*.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

from extra_data import RunDirectory
```

3.7.1 SASE1

Load the SASE1 run:

[2]:
```
sa1_data = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0008')
sa1_data.info()
```

```
# of trains:    6296
Duration:       0:10:29.500000
First train ID: 38227866
Last train ID:  38234161

0 detector modules ()

1 instrument sources (excluding detectors):
  - SA1_XTD2_XGM/XGM/DOOCS:output

0 control sources:
```

We are interested in fast, i.e. pulse-resolved data from the instrument source `SA1_XTD2_XGM/DOOCS:output`.

[3]:
```
sa1_data.keys_for_source('SA1_XTD2_XGM/XGM/DOOCS:output')
```

[3]:
```
{'data.intensityTD'}
```

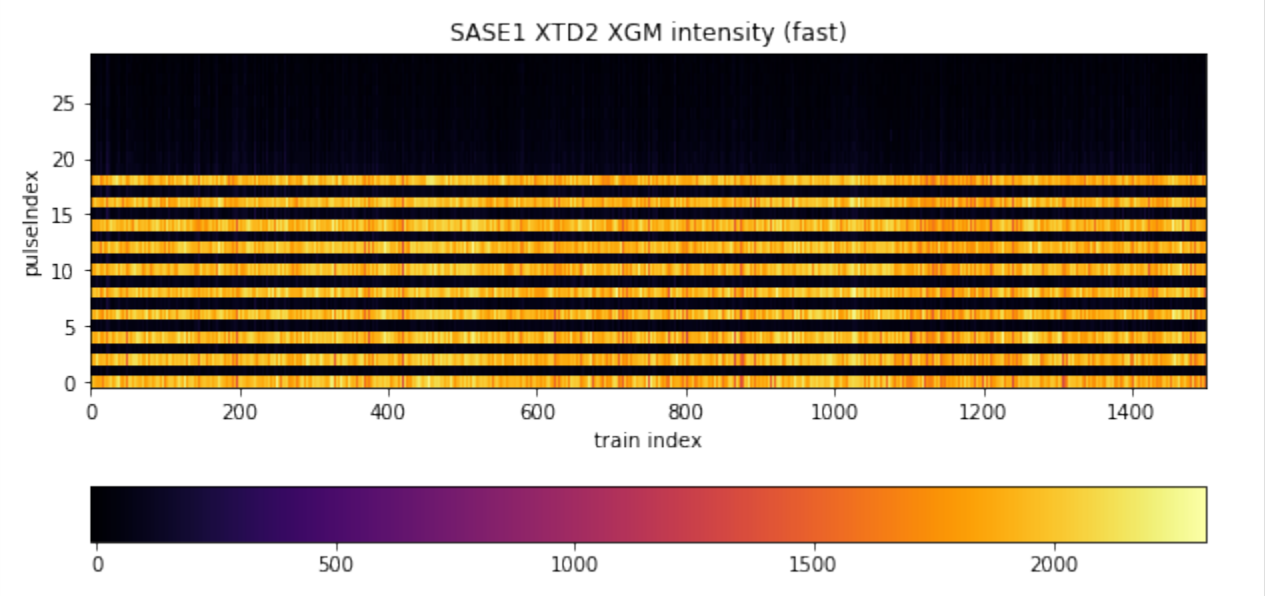
We are particularly interested in data for quantity "intensityTD". The ***xarray DataArray*** class is suited for work with axis-labeled data, and the extra_data method `get_array()` serves the purpose of shaping a 2D array of that type from pulse-resolved data (which is originally stored "flat" in terms of pulses: there is one dimension of N(train) x N(pulse) values in HDF5, and the same number of train and pulse identifiers for reference). The unique train identifier values are taken as coordinate values ("labels").

[4]:
```
sa1_flux = sa1_data.get_array('SA1_XTD2_XGM/XGM/DOOCS:output', 'data.intensityTD')
print(sa1_flux)
```

```
<xarray.DataArray (trainId: 6295, dim_0: 1000)>
array([[2.045129e+03, 7.820441e+01, 1.964445e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [2.091464e+03, 4.242367e+01, 1.915582e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [1.872965e+03, 4.368253e+01, 1.984025e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       ...,
       [1.611342e+03, 5.569377e+01, 1.811418e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [1.536590e+03, 6.418680e+01, 1.643087e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [1.871557e+03, 5.983860e+01, 1.738864e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00]], dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227866 38227867 38227868 ... 38234160 38234161
Dimensions without coordinates: dim_0
```

Next, we will plot a portion of the data in two dimensions, taking the first 1500 trains for the x-Axis and the first 30 pulses per train for the y-Axis (1500, 30). Because the Matplotlib convention takes the slow axis to be *y*, we have to transpose to (30, 1500):

[5]:
```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
image = ax.imshow(sa1_flux[:1500, :30].transpose(), origin='lower', cmap='inferno')
ax.set_title('SASE1 XTD2 XGM intensity (fast)')
fig.colorbar(image, orientation='horizontal')
ax.set_xlabel('train index')
ax.set_ylabel('pulseIndex')
ax.set_aspect(15)
```



The pattern tells us what was done in this experiment: the lasing scheme was set to provide an alternating X-ray pulse delivery within a train, where every "even" electron bunch caused lasing in SASE1 and every "odd" bunch caused lasing in SASE3. This scheme was applied for the first 20 pulses. Therefore, we see signal only for data at even pulses here (0,2,...18), throughout all trains, of which 1500 are depicted. The intensity varies somewhat around 2000 units, but for odd pulses it is suppressed and neglegibly small.

A relevant measure to judge the efficiency of pulse suppression is the ratio of mean intensity between the odd and even set. The numpy `mean` method can work with DataArray objects and average over a specified dimension.
We make use of the numpy indexing and slicing syntax with square brackets and comma to seperate axes (dimensions). We specify `[:, :20:2]` to take every element of the slow axis (trains) and every second pulse up to but excluding # 20. That is, start:end:step = 0:20:2 (start index 0 is default, thus not put, and stop means first index beyond range). We specify `axis=1` to explicitly average over that dimension. The result is a DataArray reduced to the "trainId" dimension.

[6]:
```
sa1_mean_on = np.mean(sa1_flux[:, :20:2], axis=1)
sa1_stddev_on = np.std(sa1_flux[:, :20:2], axis=1)
print(sa1_mean_on)
```

```
<xarray.DataArray (trainId: 6295)>
array([1931.4768, 1977.8414, 1873.7828, ..., 1771.5828, 1697.2053, 1857.7439],
      dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227866 38227867 38227868 ... 38234160 38234161
```

Accordingly for the odd "off" pulses:

[7]:
```
sa1_mean_off = np.mean(sa1_flux[:, 1:21:2], axis=1)
sa1_stddev_off = np.std(sa1_flux[:, 1:21:2], axis=1)
print(sa1_mean_off)
```

```
<xarray.DataArray (trainId: 6295)>
array([96.10835 , 84.489044, 59.212048, ..., 90.2944  , 84.33766 , 85.03202 ],
      dtype=float32)
```

(continues on next page)

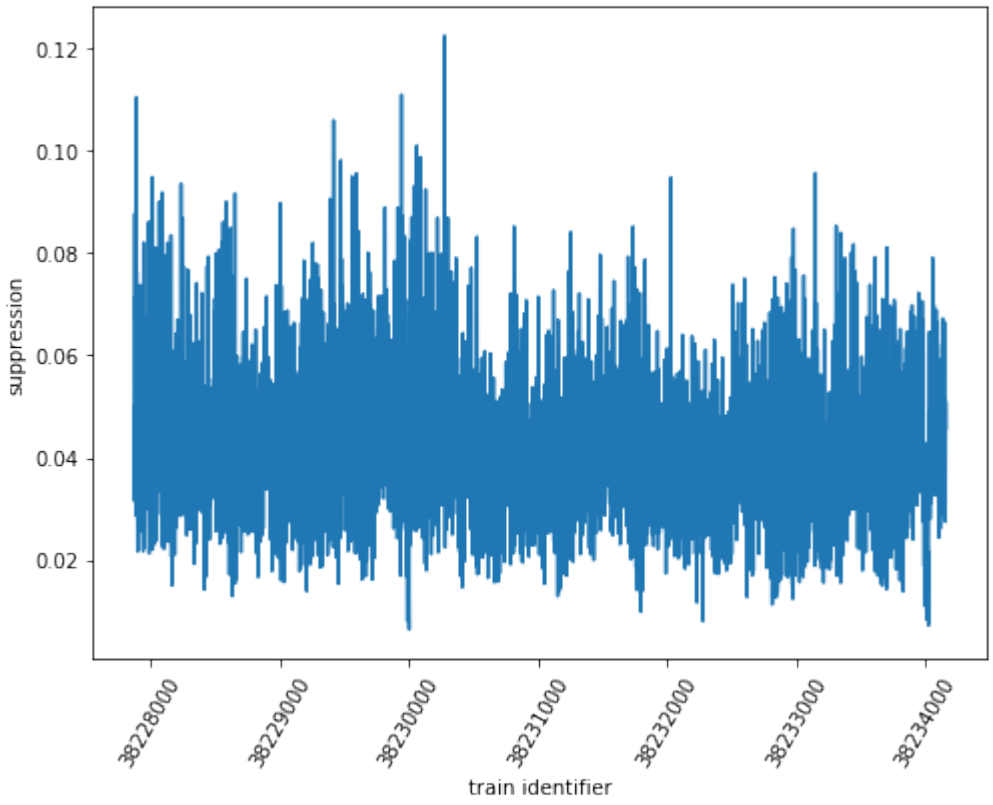
(continued from previous page)

```
Coordinates:
  * trainId  (trainId) uint64 38227866 38227867 38227868 ... 38234160 38234161
```

Now we can calculate the ratio of averages for every train - data types like *numpy ndarray* or *xarray DataArray* may be just divided "as such", a shortcut notation for dividing every corresponding element - and plot.

[8]:
```
sa1_suppression = sa1_mean_off / sa1_mean_on
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(sa1_suppression.coords['trainId'].values, sa1_suppression)
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

[8]:
```
Text(0, 0.5, 'suppression')
```



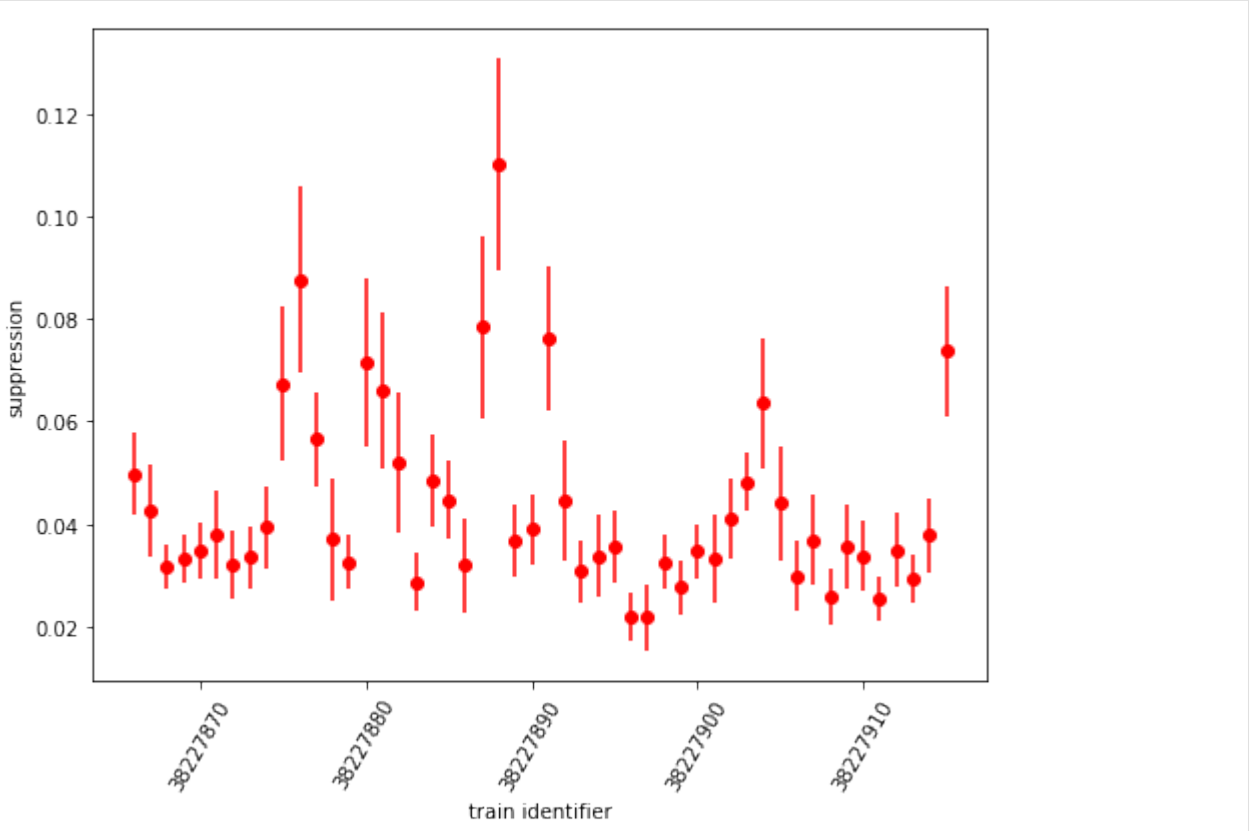
Moreover, the relative error of this ratio can be calculated by multiplicative error propagation as the square root of the sum of squared relative errors (enumerator and denominator), and from it the absolute error. The Numpy functions "sqrt" and "square" applied to array-like structures perform these operations element-wise, so the entire calculation can be conveniently done using the arrays as arguments, and we obtain individual errors for every train in the end.

[9]:
```
sa1_rel_error = np.sqrt(np.square(sa1_stddev_off / sa1_mean_off) + np.square(sa1_
↪stddev_on / sa1_mean_on))
sa1_abs_error = sa1_rel_error * sa1_suppression
```

We can as well plot the suppression ratio values with individual error bars according to the respective absolute error. Here, we restrict ourselves to the first 50 trains for clarity:

[10]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.errorbar(sa1_suppression.coords['trainId'].values[:50], sa1_suppression[:50],␣
↪yerr=sa1_abs_error[:50], fmt='ro')
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

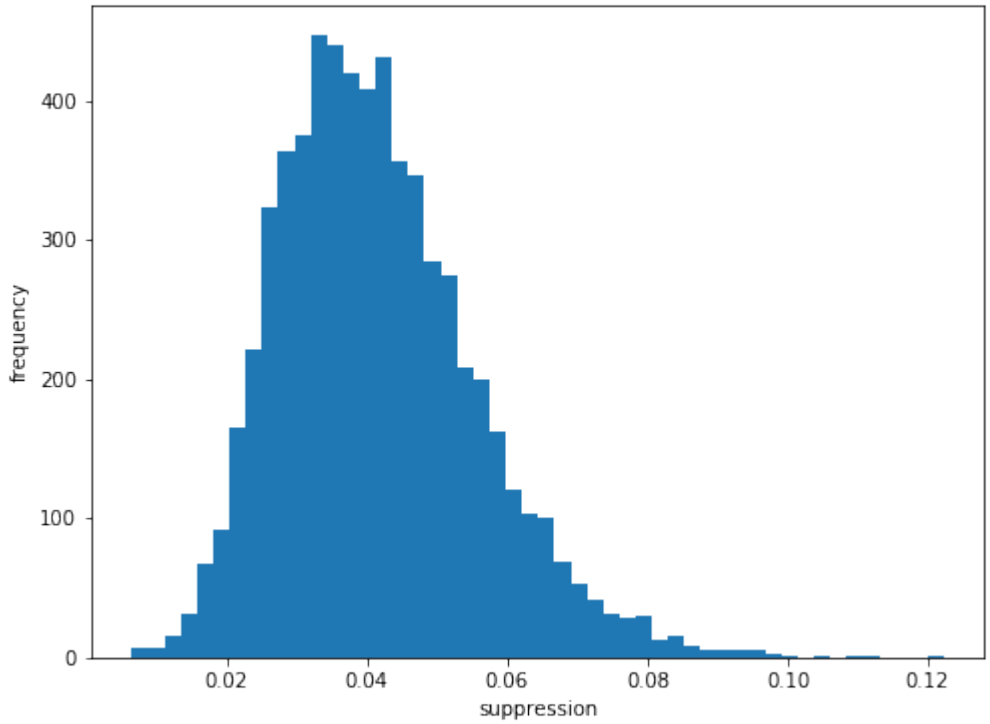
[10]:
```
Text(0, 0.5, 'suppression')
```



Finally, we draw a histogram of suppression ratio values:

[11]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
_ = ax.hist(sa1_suppression, bins=50)
ax.set_xlabel('suppression')
ax.set_ylabel('frequency')
```

[11]:
```
Text(0, 0.5, 'frequency')
```



We see that there is a suppression of signal from odd pulses to approximately 4% of the intensity of even pulses.

3.7.2 SASE3

We repeat everything for the second data set from the different run - SASE3:

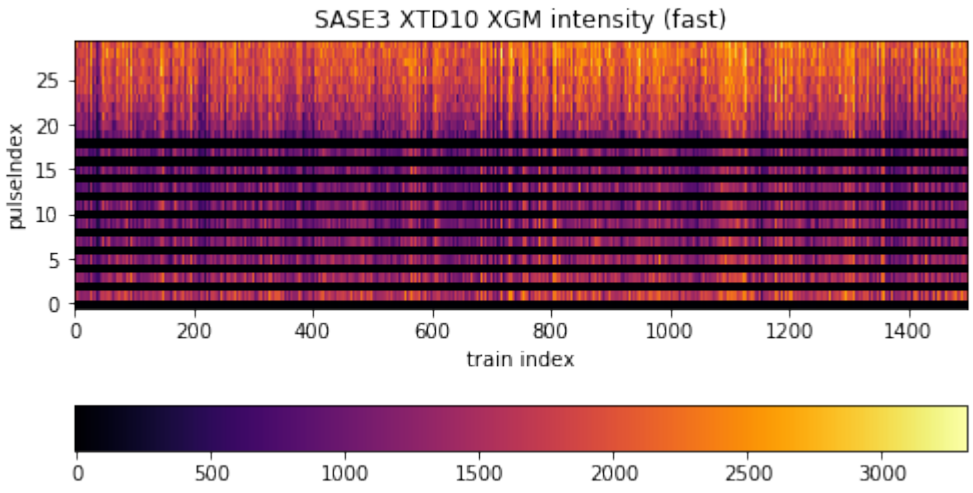
[12]:
```
sa3_data = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0009')
sa3_data.info()
```

```
# of trains:    6236
Duration:       0:10:23.500000
First train ID: 38227850
Last train ID:  38234085

0 detector modules ()

1 instrument sources (excluding detectors):
  - SA3_XTD10_XGM/XGM/DOOCS:output

0 control sources:
```

[13]:
```
sa3_flux = sa3_data.get_array('SA3_XTD10_XGM/XGM/DOOCS:output', 'data.intensityTD')
print(sa3_flux.shape)
```

```
(6235, 1000)
```

[14]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
image = ax.imshow(sa3_flux[:1500, :30].transpose(), origin='lower', cmap='inferno')
ax.set_title('SASE3 XTD10 XGM intensity (fast)')
fig.colorbar(image, orientation='horizontal')
ax.set_xlabel('train index')
ax.set_ylabel('pulseIndex')
ax.set_aspect(15)
```



The difference here is that the selection scheme (indexing and slicing) shifts by one with respect to SASE1 data: odd pulses are "on", even pulses are "off". Moreover, while the alternating scheme is upheld to pulse # 19, pulses beyond that exclusively went to SASE3. There is signal up to pulse # 70, which we could see with a wider plotting range (but not done due to emphasis on the alternation).

[15]:
```
sa3_mean_on = np.mean(sa3_flux[:, 1:21:2], axis=1)
sa3_stddev_on = np.std(sa3_flux[:, 1:21:2], axis=1)
print(sa3_mean_on)
```

```
<xarray.DataArray (trainId: 6235)>
array([ 963.89746, 1073.1758 ,  902.22656, ...,  883.9881 ,  960.5875 ,
        889.625  ], dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227850 38227851 38227852 ... 38234084 38234085
```

[16]:
```
sa3_mean_off = np.mean(sa3_flux[:, :20:2], axis=1)
sa3_stddev_off = np.std(sa3_flux[:, :20:2], axis=1)
print(sa3_mean_off)
```

```
<xarray.DataArray (trainId: 6235)>
array([5.435107, 6.615537, 8.361802, ..., 2.378666, 7.135999, 4.612433],
      dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227850 38227851 38227852 ... 38234084 38234085
```

The suppression ratio calculation and its plot:

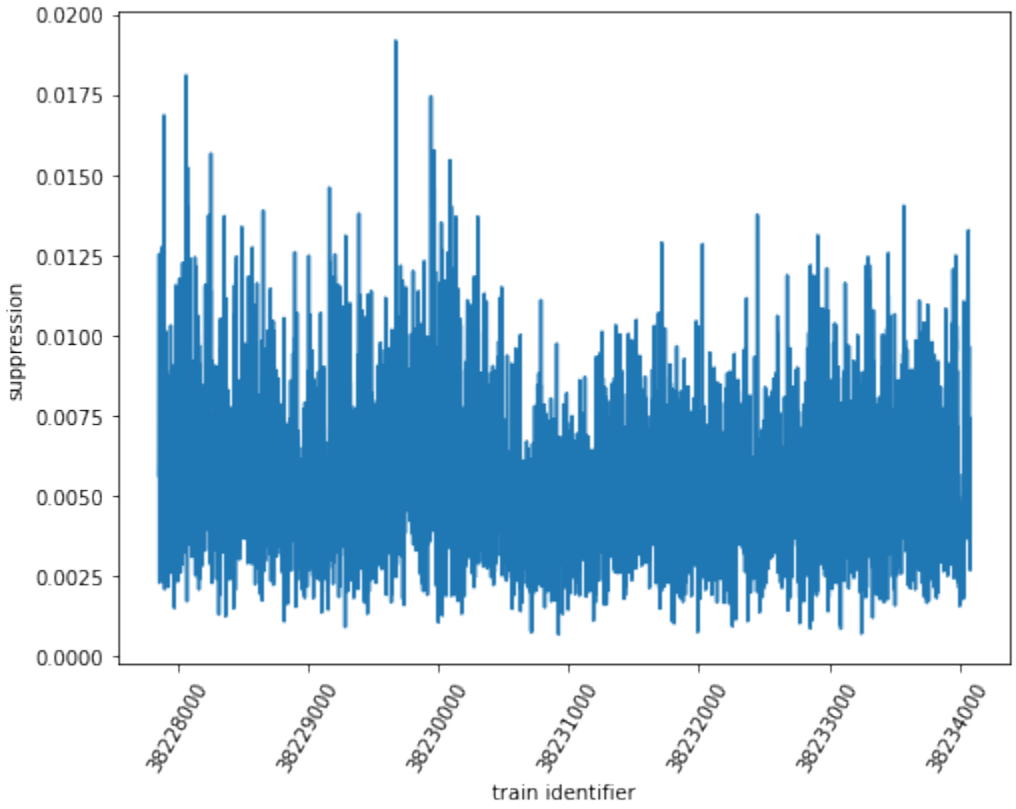
[17]:
```
sa3_suppression = sa3_mean_off / sa3_mean_on
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(sa3_suppression.coords['trainId'].values, sa3_suppression)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

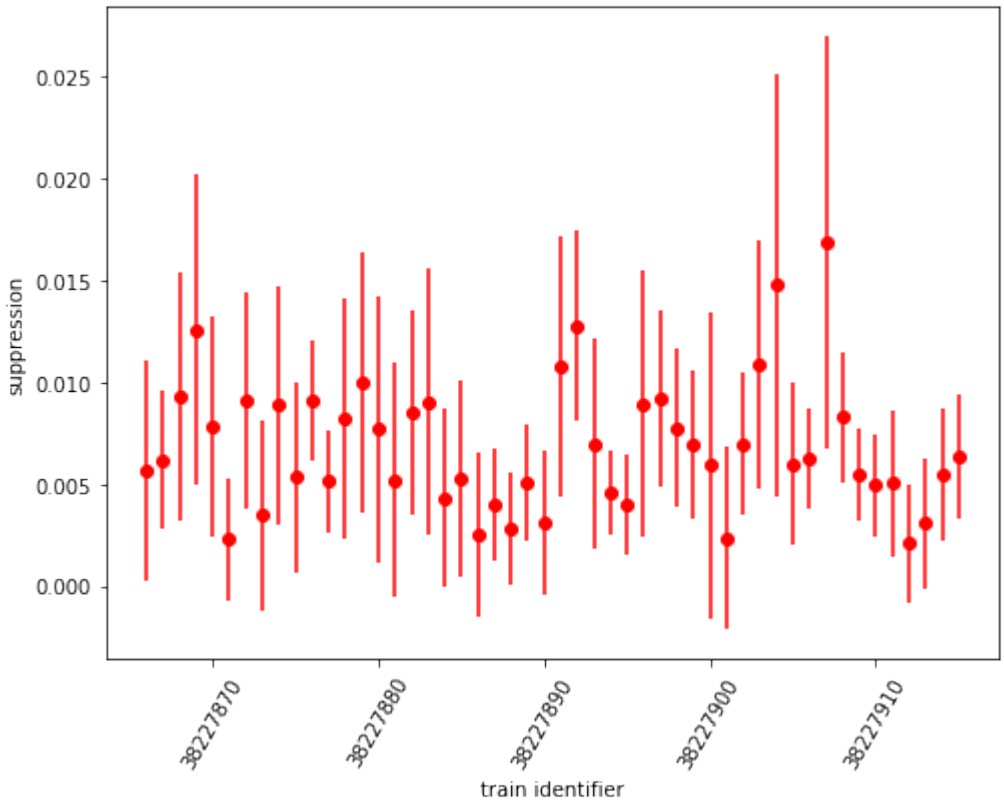
[17]:
```
Text(0, 0.5, 'suppression')
```



The error calculation with (selective) plot

[18]:
```
sa3_rel_error = np.sqrt(np.square(sa3_stddev_off / sa3_mean_off) + np.square(sa3_
↪stddev_on / sa3_mean_on))
sa3_abs_error = sa3_rel_error * sa3_suppression
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.errorbar(sa1_suppression.coords['trainId'].values[:50], sa3_suppression[:50],␣
↪yerr=sa3_abs_error[:50], fmt='ro')
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

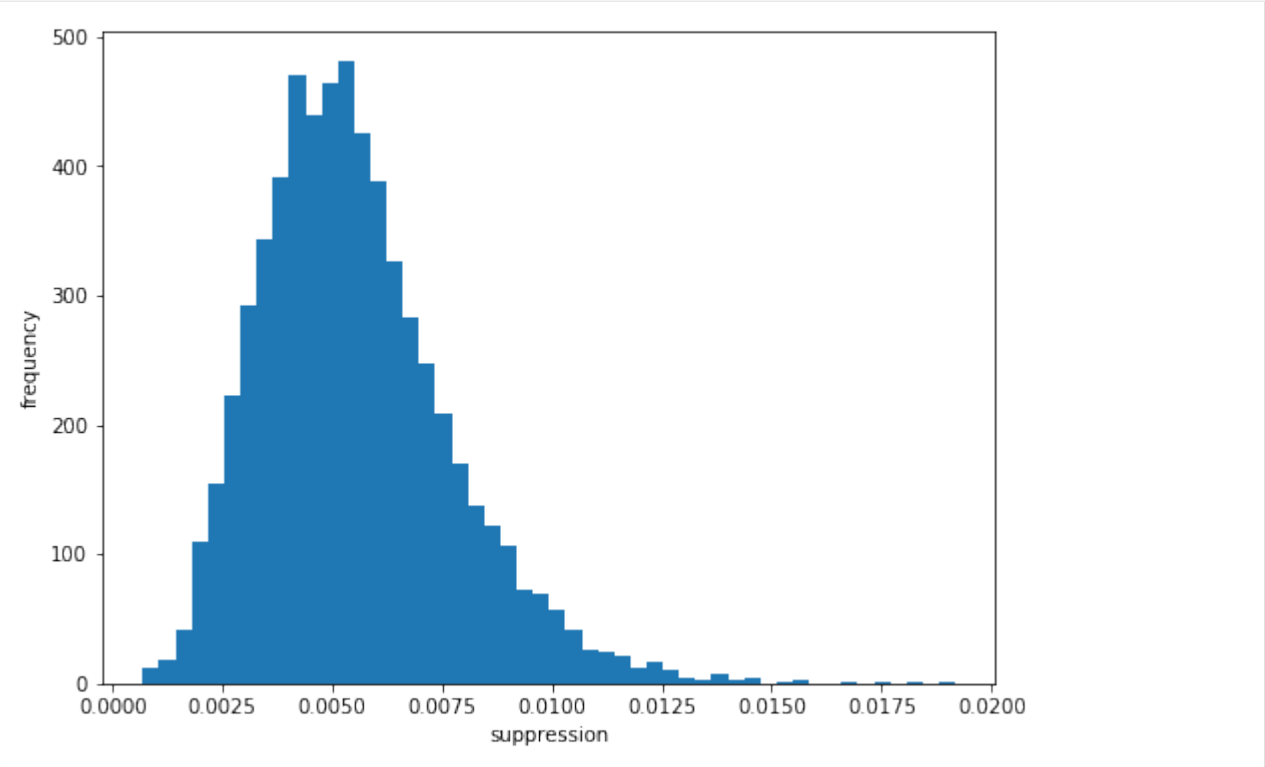
[18]:
```
Text(0, 0.5, 'suppression')
```



The histogram:

[19]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
_ = ax.hist(sa3_suppression, bins=50)
ax.set_xlabel('suppression')
ax.set_ylabel('frequency')
```

[19]:
```
Text(0, 0.5, 'frequency')
```



Here, suppression of signal for even "off" pulses is to approximately 0.5% of intensity from odd "on" pulses. The "suppression factor" is almost 10 times the value of SASE1. However, the relative error of these values is larger as well, as can be seen in the error-bar plot. For the smaller quantities, it is ~ 100% (!).

3.7.3 Overall comparison of suppression ratio (with error)

We ultimately want a single overall compression ratio with error for both beamlines, to complement the error-bar plots. In order to keep the error calculation simple, we do not average the mean values, but create one mean and standard deviation from a flat array of original values.

Because labeled axes are not required for this purpose, we can afford to move from the xarray.DataArray regime to Numpy array.

[20]:
```
sa1_on_all = np.array(sa1_flux[:, :20:2]).flatten()
sa1_on_all.shape
```

[20]:
```
(62950,)
```

[21]:
```
sa1_mean_on_overall = np.mean(sa1_on_all)
sa1_stddev_on_overall = np.std(sa1_on_all)
```

[22]:
```
sa1_off_all = np.array(sa1_flux[:, 1:21:2]).flatten()
sa1_off_all.shape
```

[22]:
```
(62950,)
```

[23]:
```
sa1_mean_off_overall = np.mean(sa1_off_all)
sa1_stddev_off_overall = np.std(sa1_off_all)
```

[24]:
```
sa1_suppression_overall = sa1_mean_off_overall / sa1_mean_on_overall
sa1_rel_error_overall = np.sqrt(np.square(sa1_stddev_off_overall / sa1_mean_off_
↪overall) + \
                        np.square(sa1_stddev_on_overall / sa1_mean_on_overall))
sa1_abs_error_overall = sa1_rel_error_overall * sa1_suppression_overall
print('SA1 suppression ratio =', sa1_suppression_overall, '\u00b1', sa1_abs_error_
↪overall)
```

```
SA1 suppression ratio = 0.04107769 ± 0.016009845
```

[25]:
```
sa3_on_all = np.array(sa3_flux[:, 1:21:2]).flatten()
sa3_on_all.shape
```

[25]:
```
(62350,)
```

[26]:
```
sa3_mean_on_overall = np.mean(sa3_on_all)
sa3_stddev_on_overall = np.std(sa3_on_all)
```

[27]:
```
sa3_off_all = np.array(sa3_flux[:, :20:2]).flatten()
sa3_off_all.shape
```

[27]:
```
(62350,)
```

[28]:
```
sa3_mean_off_overall = np.mean(sa3_off_all)
sa3_stddev_off_overall = np.std(sa3_off_all)
```

[29]:
```
sa3_suppression_overall = sa3_mean_off_overall / sa3_mean_on_overall
sa3_rel_error_overall = np.sqrt(np.square(sa3_stddev_off_overall / sa3_mean_off_
↪overall) + \
                        np.square(sa3_stddev_on_overall / sa3_mean_on_overall))
sa3_abs_error_overall = sa3_rel_error_overall * sa3_suppression_overall
print('SA3 suppression ratio =', sa3_suppression_overall, '\u00b1', sa3_abs_error_
↪overall)
```

```
SA3 suppression ratio = 0.005213415 ± 0.0040653846
```

3.7.4 References

1. K. Tiedtke et al., Gas-detector for X-ray lasers , J. Appl. Phys. 103, 094511 (2008) - DOI 10.1063/1.2913328
2. A. A. Sorokin et al., J. Synchrotron Rad. 26 (4), DOI 10.1107/S1600577519005174 (2019)
3. Th. Maltezopoulos et al., J. Synchrotron Rad. 26 (4), DOI 10.1107/S1600577519003795 (2019)

3.8 Reading data files

3.8.1 Opening files

You will normally access data from a run, which is stored as a directory containing HDF5 files. You can open a run using `RunDirectory()` with the path of the directory, or using `open_run()` with the proposal number and run number to look up the standard data paths on the Maxwell cluster.

extra_data.**RunDirectory**(*path*, *include='*'*, *file_filter=<function lc_any>*)

Open data files from a 'run' at European XFEL.

```
run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0001")
```

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.
- **include** (*str*) – Wildcard string to filter data files.
- **file_filter** (*callable*) – Function to subset the list of filenames to open. Meant to be used with functions in the extra_data.locality module.

extra_data.**open_run**(*proposal*, *run*, *data='raw'*, *include='*'*, *file_filter=<function lc_any>*)

Access EuXFEL data on the Maxwell cluster by proposal and run number.

```
run = open_run(proposal=700000, run=1)
```

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'.
- **include** (*str*) – Wildcard string to filter data files.
- **file_filter** (*callable*) – Function to subset the list of filenames to open. Meant to be used with functions in the extra_data.locality module.

New in version 0.5.

You can also open a single file. The methods described below all work for either a run or a single file.

extra_data.**H5File**(*path*)

Open a single HDF5 file generated at European XFEL.

```
file = H5File("RAW-R0017-DA01-S00000.h5")
```

Returns a *DataCollection* object.

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

3.8.2 Data structure

A run (or file) contains data from various *sources*, each of which has *keys*. For instance, `SA1_XTD2_XGM/XGM/DOOCS` is one source, for an 'XGM' device which monitors the beam, and its keys include `beamPosition.ixPos` and `beamPosition.iyPos`.

European XFEL produces ten *pulse trains* per second, each of which can contain up to 2700 X-ray pulses. Each pulse train has a unique train ID, which is used to refer to all data associated with that 0.1 second window.

class extra_data.**DataCollection**

train_ids
A list of the train IDs included in this data. 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"`. Control data is always recorded exactly once per train.

instrument_sources
A set of the instrument source names in this data, in the format `"FXE_DET_LPD1M-1/DET/15CH0:xtdf"`. Instrument data may be recorded zero to many times per train.

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.

get_data_counts(*source*, *key*)
Get a count of data points in each train for the given data field.

Returns a pandas series with an index of train IDs.

Parameters

- **source** (*str*) – Source name, e.g. "SPB_DET_AGIPD1M-1/DET/7CH0:xtdf"
- **key** (*str*) – Key of parameter within that device, e.g. "image.data".

info(*details_for_sources=()*)
Show information about the selected data.

3.8.3 Getting data by source & key

Where data will fit into memory, it's usually quickest and most convenient to load it like this.

class extra_data.**DataCollection**

get_array(*source*, *key*, *extra_dims=None*, *roi=()*)
Return a labelled array for a particular data field.

```
arr = run.get_array("SA3_XTD10_PES/ADC/1:network", "digitizers.channel_4_A.
↪raw.samples")
```

This should work for any data. The first axis of the returned data will be labelled with 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** (*slice, tuple of slices, or 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=np.s_[:8] would get the first 8 values from every train. If the data is 2D or more at each entry, selection looks like roi=np.s_[:8, 5:10] .

See also:

xarray documentation How to use the arrays returned by *get_array()*

Reading data to analyse in memory Examples using xarray & pandas with EuXFEL data

get_dask_array(*source*, *key*, *labelled=False*)

Get a Dask array for the specified data field.

Dask is a system for lazy parallel computation. This method doesn't actually load the data, but gives you an array-like object which you can operate on. Dask loads the data and calculates results when you ask it to, e.g. by calling a `.compute()` method. See the Dask documentation for more details.

If your computation depends on reading lots of data, consider creating a dask.distributed.Client before calling this. If you don't do this, Dask uses threads by default, which is not efficient for reading HDF5 files.

Parameters

- **source** (*str*) – Source name, e.g. "SPB_DET_AGIPD1M-1/DET/7CH0:xtdf"
- **key** (*str*) – Key of parameter within that device, e.g. "image.data".
- **labelled** (*bool*) – If True, label the train IDs for the data, returning an xarray.DataArray object wrapping a Dask array.

See also:

Dask Array documentation How to use the objects returned by *get_dask_array()*

Averaging detector data with Dask An example using Dask with EuXFEL data

get_series(*source*, *key*)

Return a pandas Series for a particular data field.

```
s = run.get_series("SA1_XTD2_XGM/XGM/DOOCS", "beamPosition.ixPos")
```

This only works for 1-dimensional data.

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

```
df = run.get_dataframe(fields=[
    ("*_XGM/*", "*.i[xy]Pos"),
    ("*_XGM/*", "*.photonFlux")
])
```

This links together multiple 1-dimensional datasets as columns in a table.

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.

See also:

pandas documentation How to use the objects returned by *get_series()* and *get_dataframe()*

Reading data to analyse in memory Examples using xarray & pandas with EuXFEL data

get_virtual_dataset(*source*, *key*, *filename=None*)
Create an HDF5 virtual dataset for a given source & key

A dataset looks like a multidimensional array, but the data is loaded on-demand when you access it. So it's suitable as an interface to data which is too big to load entirely into memory.

This returns an h5py.Dataset object. This exists in a real file as a 'virtual dataset', a collection of links pointing to the data in real datasets. If *filename* is passed, the file is written at that path, overwriting if it already exists. Otherwise, it uses a new temp file.

To access the dataset from other worker processes, give them the name of the created file along with the path to the dataset inside it (accessible as `ds.name`). They will need at least HDF5 1.10 to access the virtual dataset, and they must be on a system with access to the original data files, as the virtual dataset points to those.

New in version 0.5.

See also:

Parallel processing with a virtual HDF5 dataset

3.8.4 Getting data by train

Some kinds of data, e.g. from AGIPD, are too big to load a whole run into memory at once. In these cases, it's convenient to load one train at a time.

When accessing data like this, it's worth selecting which sources you're interested in, either using *select()*, or the `devices=` parameter. This avoids reading all the other data.

class extra_data.**DataCollection**

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

```
run = Run('/path/to/my/run/r0123')
for train_id, data in run.select("*/DET/*", "image.data").trains():
    mod0 = data["FXE_DET_LPD1M-1/DET/0CH0:xtdf"]["image.data"]
```

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

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

3.8.5 Selecting & combining data

These methods all return a new *DataCollection* object with the selected data, so you use them like this:

```
sel = run.select("*/XGM/*")
# sel includes only XGM sources
# run still includes all the data
```

class extra_data.**DataCollection**

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:

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

2. With an iterable of (source, key) glob patterns:

   ```
   # 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):

   ```
   # 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".

---

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.

select_trains(*train_range*)
Select a subset of trains from this data.

Choose a slice of trains by train ID:

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

Or select a list of trains:

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

Or select trains by index within this collection:

```
sel = run.select_trains(np.s_[: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.

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.

3.8.6 Writing selected data

class extra_data.**DataCollection**

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.

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. See *AGIPD, LPD & DSSC data* for that.

Creating and reading virtual datasets requires HDF5 version 1.10.

The target filename will be overwritten if it already exists.

3.8.7 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 `extra_data`:

- The train-oriented methods *`trains()`*, *`train_from_id()`*, and *`train_from_index()`* give you dictionaries 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.

3.8.8 Data problems

If you encounter problems accessing data with `extra_data`, there may be problems with the data files themselves. Use the `extra-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()`*:

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

If you're having problems with extra_data, you can also try searching previously reported issues to see if anyone has encountered similar symptoms.

3.8.9 Cached run data maps

When you open a run in extra_data, it needs to know what data is in each file. Each file has metadata describing its contents, but reading this from every file is slow, especially on GPFS. extra_data therefore tries to cache this information the first time a run is opened, and reuse it when opening that run again.

This should happen automatically, without the user needing to know about it. You only need these details if you think caching may be causing problems.

- Caching is triggered when you use *RunDirectory()* or *open_run()*.
- There are two possible locations for the cached data map:
  - In the run directory: `(run dir)/karabo_data_map.json`.
  - In the proposal scratch directory: `(proposal dir)/scratch/.karabo_data_maps/raw_r0032.json`. This will normally be the one used on Maxwell, as users can't write to the run directory.
- The format is a JSON array, with an object for each file in the run.
  - This holds the list of train IDs in the file, and the lists of control and instrument sources.
  - It also stores the file size and last modified time of each data file, to check if the file has changed since the cache was created. If either of these attributes doesn't match, extra_data ignores the cached information and reads the metadata from the HDF5 file.
- If any file in the run wasn't listed in the data map, or its entry was outdated, a new data map is written automatically. It tries the same two locations described above, but it will continue without error if it can't write to either.

JSON was chosen as it can be easily inspected manually, and it's reasonably efficient to load the entire file.

3.8.10 Issues reading archived data

Files at European XFEL storage migrate over time from GPFS (designed for fast access) to PNFS (designed for archiving). The data on PNFS is usually always available for reading. But sometimes, this may require staging from the tape to disk. If there is a staging queue, the operation can take an indefinitely long time (days or even weeks) and any IO operations will be blocked for this time.

To determine the files which require staging or are lost, use the script:

```
extra-data-locality <run directory>
```

It returns a list of files which are currently located only on slow media for some reasons and, separately, any which have been lost.

If the files are not essential for analysis, then they can be filtered out using filter `lc_ondisk()` from `extra_data.locality`:

```
from extra_data.locality import lc_ondisk
run = open_run(proposal=700000, run=1, file_filter=lc_ondisk)
```

`file_filter` must be a callable which takes a list as a single argument and returns filtered list.

Note: Reading the file locality on PNFS is an expensive operation. Use it only as a last resort.

If you find any files which are located only on tape or unavailable, please let know to ITDM. If you need these files for analysis mentioned that explicitly.

3.9 AGIPD, LPD & DSSC data

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

class extra_data.components.**AGIPD1M**(*data: extra_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 extra_data.components.**DSSC1M**(*data: extra_data.reader.DataCollection*, *detector_name=None*, *modules=None*, ***, *min_modules=1*)

An interface to DSSC-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. 'SCS_DET_DSSC1M-1'. This is only needed if the dataset includes more than one DSSC 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 extra_data.components.**LPD1M**(*data: extra_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_dask_array(*key*, *subtrain_index='pulseId'*)

Get a labelled Dask array of detector data

Dask does lazy, parallelised computing, and can work with large data volumes. This method doesn't immediately load the data: that only happens once you trigger a computation.

Parameters

- **key** (*str*) – The data to get, e.g. 'image.data' for pixel values.
- **subtrain_index** (*str*) – Specify 'pulseId' (default) or 'cellId' to label the frames recorded within each train. Pulse ID should allow this data to be matched with other devices, but depends on how the detector was manually configured when the data was taken. Cell ID refers to the memory cell used for that frame in the detector hardware.

get_array(*key*, *pulses=slice(None, None, None)*, *unstack_pulses=True*)
Get a labelled array of detector data

Parameters

- **key** (*str*) – The data to get, e.g. 'image.data' for pixel values.
- **pulses** (*slice, array, 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.
- **unstack_pulses** (*bool*) – Whether to separate train and pulse dimensions.

trains(*pulses=slice(None, None, None)*, *require_all=True*)
Iterate over trains for detector data.

Parameters

- **pulses** (*slice, array, by_index or by_id*) – Select which pulses to include for each train. The default is to include all pulses.
- **require_all** (*bool*) – If True (default), skip trains where any of the selected detector modules are missing data.

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

write_frames(*filename*, *trains*, *pulses*)
Write selected detector frames to a new EuXFEL HDF5 file

trains and pulses should be 1D arrays of the same length, containing train IDs and pulse IDs (corresponding to the pulse IDs recorded by the detector). i.e. (trains[i], pulses[i]) identifies one frame.

write_virtual_cxi(*filename*, *fillvalues=None*)
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.
- **fillvalues** (*dict, optional*) – keys are datasets names (one of: data, gain, mask) and associated fill value for missing data (default is np.nan for float arrays and zero for integer arrays)

See also:

Accessing LPD data: An example using the class above.

extra_data.components.**identify_multimod_detectors**(*data*, *detector_name=None*, ***, *single=False*, *clses=None*)
Identify multi-module detectors in the data

Various detectors record data in a similar format, and we often want to process whichever detector was used in a run. This tries to identify the detector, so a user doesn't have to specify it manually.

If `single=True`, this returns a tuple of (detector_name, access_class), throwing `ValueError` if there isn't exactly 1 detector found. If `single=False`, it returns a set of these tuples.

clses may be a list of acceptable detector classes to check.

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:

`extra_data.`**`stack_detector_data`**(*train*, *data*, *axis=-3*, *modules=16*, *fillvalue=nan*, *real_array=True*)

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).
- **`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.
- **`real_array`** (*bool*) – If True (default), copy the data together into a real numpy array. If False, avoid copying the data and return a limited array-like wrapper around the existing arrays. This is sufficient for assembling images using detector geometry, and allows better performance.

Returns **combined** – Stacked data for requested data path.

Return type numpy.array

3.10 Streaming data over ZeroMQ

Karabo Bridge provides access to live data during the experiment over a ZeroMQ socket. The `extra_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.

Command-line options are explained on the *command reference* page.

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:

`class` `extra_data.export.`**`ZMQStreamer`**(*port*, *sock='REP'*, *maxlen=10*, *protocol_version='2.2'*, *dummy_timestamps=False*)

`start`()

`feed`(*data*, *metadata=None*, *block=True*, *timeout=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 see `serialize()` for details
- **metadata** (*dict, optional*) – Contains train metadata. 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. see `serialize()` for details
- **block** (*bool*) – If True, block if necessary until a free slot is available or 'timeout' has expired. If False and there is no free slot, raises 'queue.Full' (timeout is ignored)
- **timeout** (*float*) – In seconds, raises 'queue.Full' if no free slow was available within that time.

3.11 Checking data files

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

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

Or a single data file:

```
extra-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 have the same number of entries as train IDs.
- 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.

3.12 Command line tools

3.12.1 `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
```

(continues on next page)

(continued from previous page)

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

--detail <source-pattern>
: Show more detail on the keys and data of the sources selected by a pattern like `*/XGM/*`. Only applies when inspecting a single run or file. Can be used several times to select different patterns.

  This option can make `lsxfel` considerably slower.

3.12.2 extra-data-validate

Check the structure of an EuXFEL run or HDF5 file:

```
extra-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.

3.12.3 karabo-bridge-serve-files

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

```
karabo-bridge-serve-files /gpfs/exfel/exp/XMPL/201750/p700000/proc/r0005 4321
```

--source <source>
: Only sources matching the string <source> will be streamed. Default is '*', serving as a global wildcard for all sources.

--key <key>
: Only data sets keyed by the string <key> will be streamed. Default is '*', serving as a global wildcard for all keys.

--append-detector-modules
: If the file data contains multiple detector modules as separate sources, i. e. for big area detectors (AGIPD, LPD and DSSC), append these into one single source.

--dummy-timestamps
: Add mock timestamps if missing in the original meta-data.

These two options above - appended module sources and dummy timestamps - are required if streamed data shall be provided to OnDA.

-z <type>, **--socket-type** <type>
: The ZMQ socket type to use, one of `PUB`, `PUSH` or `REP`. Default: `REP`.

--use-infiniband
: Use the infiniband network interface (`ib0`) if it's present.

3.12.4 extra-data-make-virtual-cxi

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

```
extra-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).

--fill-value <dataset> <value>
Set the fill value for dataset (one of `data`, `gain` or `mask`). The defaults are different in different cases:

- data (raw, uint16): 0
- data (proc, float32): NaN
- gain: 0
- mask: 0xffffffff

3.12.5 extra-data-locality

Check how the files are stored:

```
extra-data-locality /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002
```

The file reading may hang for a long time if files are unavailable or require staging in dCache from the tape. The program helps finding problem files.

If it finds problems with the data locality, the program will produce a list of files located on tape, lost or at unknown locality and exit with the non-zero status.

3.13 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.

3.13.1 HDF5 file structure

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.

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:

```
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.

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?)

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.

3.14 Performance notes

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

3.14.1 Load data into memory

Where the data you need can fit into memory, it's more efficient to load it in one go using *`get_array()`*, *`get_series()`* or *`get_dataframe()`*, and then work with it using xarray, numpy or pandas. *Reading data to analyse in memory* has some examples of this. The methods to get data by trains—*`trains()`*, *`train_from_id()`* and *`train_from_index()`*—only load the data for one train at once, which saves memory for big data but is slower to process.

Machines in the Maxwell cluster have hundreds of gigabytes of RAM, so it's practical to load many kinds of data completely into memory. However, data for a full run from megahertz detectors such as AGIPD, LPD or DSSC can easily be too much.

The command `free -h` will show the amount of memory on any Linux machine.

3.14.2 Select sources before getting trains

If you do need to use *trains()*, *train_from_id()* or *train_from_index()* to get data for one train at a time, first pick the sources and keys you need with *select()*. Otherwise, you will load the data for every source in the run, which could be very slow.

```
run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0004")

# SLOW: Don't do this!
for tid, train_data in run.trains():
    ...

# Better option: select image data from all detector modules first.
for tid, train_data in run.select('*/DET/*', 'image.data').trains():
    ...
```

The `devices=` parameter for all three train methods does the same thing as using *select()* like this.

3.14.3 Reduce before assembling

Assembling detector images (see EXtra-geom) 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.

3.15 Release Notes

3.15.1 1.3

New features:

- A new interface for data from a single source & key: use `run[source, key]` to get a `KeyData` object, which can inspect and load the data from several sequence files (PR #70).
- Methods which took a `by_index` object now accept slices (e.g. `numpy.s_[:10]`) or indices directly (PR #68, PR #79). This includes *select_trains()*, *get_array()* and various methods for multi-module detectors, described in *AGIPD, LPD & DSSC data*.
- `extra-data-make-virtual-cxi --fill-value` now accepts numbers in hexadecimal, octal & binary formats, e.g. `0xfe` (PR #73).
- Added an `unstack` parameter to the *get_array()* method for multi-module detectors, making it possible to retrieve an array as the data is stored, without separating the train & pulse axes (PR #72).
- Added a `require_all` parameter to the *trains()* method for multi-module detectors, to allow iterating with incomplete frames included (PR #77).
- New *identify_multimod_detectors()* function to find multi-module detectors in the data (PR #61).

Fixes and improvements:

- Fix writing selected detector frames with *write_frames()* for corrected data (PR #82).
- Fix compatibility with pandas 1.1 (PR #83).
- The *trains()* iterator no longer includes zero-length arrays when a source has no data for that train (PR #75).

- Fix a test which failed when run as root (PR #67).

3.15.2 1.2

New features:

- New `karabo-bridge-serve-files --append-detector-modules` option to combine data from multiple detector modules. This makes streaming large detector data more similar to the live data streams (PR #40 and PR #51).
- *karabo-bridge-serve-files* has new options to control the ZMQ socket and the use of an infiniband network interface (PR #50). It also works with newer versions of the `karabo_bridge` Python package.
- New options to filter files from dCache which are unavailable or need to be read from tape when opening a run (PR #35). This also comes with a new command *extra-data-locality* to inspect this information.
- New `lsxfel --detail` option to show more detail on selected sources (PR #38).
- New `extra-data-make-virtual-cxi --fill-value` option to control the fill value for missing data (PR #59)
- New method `write_frames()` to save a subset of detector frames to a new file in EuXFEL HDF5 format (PR #47).
- `DataCollection.select()` can take arbitrary iterables of patterns, rather than just lists (PR #43).

Fixes and improvements:

- EXtra-data now tries to manage how many HDF5 files it has open at one time, to avoid hitting a limit on the total number of open files in a process (PR #25 and PR #48). Importing EXtra-data will now raise this limit as far as it can (to 4096 on Maxwell), and try to keep the files it handles to no more than half of this. Files should be silently closed and reopened as needed, so this shouldn't affect how you use it.
- A better way of creating Dask arrays to avoid problems with Dask's local schedulers, and with arrays comprising very large numbers of files (PR #63).
- The classes for accessing multi-module detector data (see *AGIPD, LPD & DSSC data*) and writing virtual CXI files no longer assume that the same number of frames are recorded in every train (PR #44).
- Fix validation where a file has no trains at all (PR #42).
- More testing of EuXFEL file format version 1.0 (PR #56).
- Test coverage measurement fixed with multiprocessing (PR #37).
- Tests switched from `mock` module to `unittest.mock` (PR #52).

3.15.3 1.1

- Opening and validating run directories now handles files in parallel, which should make it substantially faster (PR #30).
- Various data access operations no longer require finding all the keys for a given data source, which saves time in certain situations (PR #24).
- `open_run()` now accepts numpy integers for proposal and run numbers, as well as standard Python integers (PR #34).
- *Run map cache files* can be saved on the EuXFEL online cluster, which speeds up reopening runs there (PR #36).
- Added tests with simulated bad files for the validation code (PR #23).

3.15.4 1.0

- New *get_dask_array()* method for accessing detector data with Dask (PR #18).
- Fix `extra-data-validate` with a run directory without a *cached data map* (PR #12).
- Add `.squeeze()` method for virtual stacks of detector data from *stack_detector_data()* (PR #16).
- Close each file after reading its metadata, to avoid hitting the limit of open files when opening a large run (PR #8). This is a mitigation: you will still hit the limit if you access data from enough files. The default limit on Maxwell is 1024 files, but you can raise this to 4096 using the Python resource module.
- Display progress information while validating a run directory (PR #19).
- Display run duration to only one decimal place (PR #5).
- Documentation reorganised to emphasise tutorials and examples (PR #10).

This version requires Python 3.6 or above.

3.15.5 0.8

First separated version. No functional changes from karabo_data 0.7.

3.15.6 Earlier history

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

See also:

Data Analysis at European XFEL

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

A

C

D

E

F

G