# kikuchipy

***Release 0.10.dev0***

**kikuchipy developers**

**Nov 03, 2023**

# CONTENTS

kikuchipy is a library for processing, simulating and analyzing electron backscatter diffraction (EBSD) patterns in Python, built on the tools for multi-dimensional data analysis provided by the HyperSpy library.

CHAPTER

ONE

# USER GUIDE

## 1.1 Installation

kikuchipy can be installed with pip, conda, the HyperSpy Bundle, or from source, and supports Python >= 3.7. All alternatives are available on Windows, macOS and Linux.

### 1.1.1 With pip

kikuchipy is availabe from the Python Package Index (PyPI), and can therefore be installed with pip. To install, run the following:

```
pip install kikuchipy
```

To update kikuchipy to the latest release:

```
pip install --upgrade kikuchipy
```

To install a specific version of kikuchipy (say version 0.8.5):

```
pip install kikuchipy==0.8.5
```

#### Optional dependencies

Some functionality is optional and requires extra dependencies which must be installed manually:

- pyebsdindex: Hough indexing. We recommend to install with optional GPU support via pyopencl with `pip install pyebsdindex[gpu]` or `conda install pyebsdindex`.
- nlopt: Extra optimization algorithms used in EBSD orientation and/or projection center refinement.
- pyvista: 3D plotting of master patterns.

Install all optional dependencies:

```
pip install kikuchipy[all]
```

Note that this command will not install `pyopencl`, which is required for GPU support in `pyebsdindex`. If the above command failed for some reason, one can try to install each optional dependency individually.

### 1.1.2 With Anaconda

To install with Anaconda, we recommend you install it in a conda environment with the Miniconda distribution. To create an environment and activate it, run the following:

```
conda create --name kp-env python=3.11
conda activate kp-env
```

If you prefer a graphical interface to manage packages and environments, you can install the Anaconda distribution instead.

To install:

```
conda install kikuchipy --channel conda-forge
```

To update kikuchipy to the latest release:

```
conda update kikuchipy
```

To install a specific version of kikuchipy (say version 0.8.5):

```
conda install kikuchipy==0.8.5
```

### 1.1.3 With the HyperSpy Bundle

kikuchipy is available in the HyperSpy Bundle. See HyperSpy Bundle for instructions.

**Warning:** kikuchipy is updated more frequently than the HyperSpy Bundle, thus the installed version of kikuchipy will most likely not be the latest version available. See the HyperSpy Bundle repository for how to update packages in the bundle.

### 1.1.4 From source

To install kikuchipy from source, clone the repository from GitHub, and install with `pip`:

```
git clone https://github.com/pyxem/kikuchipy.git
cd kikuchipy
pip install --editable .
```

See the contributing guide for *Setting up a development installation* and keeping it up to date.

### 1.1.5 Dependencies

kikuchipy builds on the great work and effort of many people. This is a list of explicit package dependencies (some are *Optional dependencies*):

| Package | Purpose |
|---|---|
| dask | Out-of-memory processing of data larger than RAM |
| diffpy.structure | Handling of crystal structures |
| diffsims | Handling of reciprocal lattice vectors and structure factors |
| hyperspy | Multi-dimensional data handling (EBSD class etc.) |
| h5py | Read/write of HDF5 files |
| imageio | Read image formats |
| matplotlib | Visualization |
| numba | CPU acceleration |
| numpy | Handling of N-dimensional arrays |
| orix | Handling of rotations and vectors using crystal symmetry |
| pooch | Downloading and caching of datasets |
| pyyaml | Parcing of YAML files |
| tqdm | Progressbars |
| scikit-image | Image processing like adaptive histogram equalization |
| scikit-learn | Multivariate analysis |
| scipy | Optimization algorithms, filtering and more |

## 1.2 Tutorials

This page contains in-depth guides for using kikuchipy. It is broken up into sections covering specific topics.

For shorter examples, see our *Examples*. For descriptions of the functions, modules, and objects in kikuchipy, see the *API reference*.

The tutorials are live and available on MyBinder: launch binder

### 1.2.1 Fundamentals and usage

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

#### Load and save data

In this tutorial, we will load and save electron backscatter diffraction (EBSD) patterns, Kikuchi master patterns generated with *EMsoft* and virtual backscatter electron (VBSE) images from and to various formats supported by kikuchipy.

### Load patterns

#### From a file

kikuchipy can read and write experimental EBSD patterns and Kikuchi master patterns from or to many formats (see *supported formats*). To load patterns from a file use, the *load()* function. Let's import the necessary libraries and read the nickel EBSD example dataset directly from file (not via *kikuchipy.data.nickel_ebsd_small()*)

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import os
from pathlib import Path
import tempfile

import dask.array as da
import imageio.v3 as iio
import numpy as np
import matplotlib.pyplot as plt

import hyperspy.api as hs
import kikuchipy as kp
```

[2]:
```
datadir = Path("../../kikuchipy/data")
nordif_ebsd = "nordif/Pattern.dat"
s = kp.load(datadir / nordif_ebsd)
s
```

[2]:
```
<EBSD, title: Pattern, dimensions: (3, 3|60, 60)>
```

Or, load the stereographic projection of the upper hemisphere of an EBSD master pattern for a 20 keV beam energy from a modified version of *EMsoft*'s master pattern file, returned from their `EMEBSDmaster.f90` program

[3]:
```
emsoft_master_pattern = (
    "emsoft_ebsd_master_pattern/ni_mc_mp_20kv_uint8_gzip_opts9.h5"
)
s_mp = kp.load(datadir / emsoft_master_pattern)
s_mp
```

[3]:
```
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (|401, 401)>
```

Both the stereographic and the square Lambert projections of this master pattern can be loaded with *kikuchipy.data.nickel_ebsd_master_pattern_small()*.

All file readers support accessing the data's metadata without loading it into memory (with the Dask library), which can be useful when processing large data sets, one data chunk at a time, to avoid memory errors

[4]:
```
s_lazy = kp.load(datadir / nordif_ebsd, lazy=True)
print(s_lazy)
s_lazy.data
```

```
<LazyEBSD, title: Pattern, dimensions: (3, 3|60, 60)>
```

[4]:
```
dask.array<array, shape=(3, 3, 60, 60), dtype=uint8, chunksize=(3, 3, 60, 60),␣
↪chunktype=numpy.ndarray>
```

Parts or all of the data can be read into memory by calling compute()

[5]:
```
s_lazy_copy = s_lazy.inav[:2, :].deepcopy()
s_lazy_copy.compute()
s_lazy_copy
```

```
[########################################] | 100% Completed | 102.25 ms
```

[5]:
```
<EBSD, title: Pattern, dimensions: (2, 3|60, 60)>
```

[6]:
```
s_lazy.compute()
s_lazy
```

```
[########################################] | 100% Completed | 101.13 ms
```

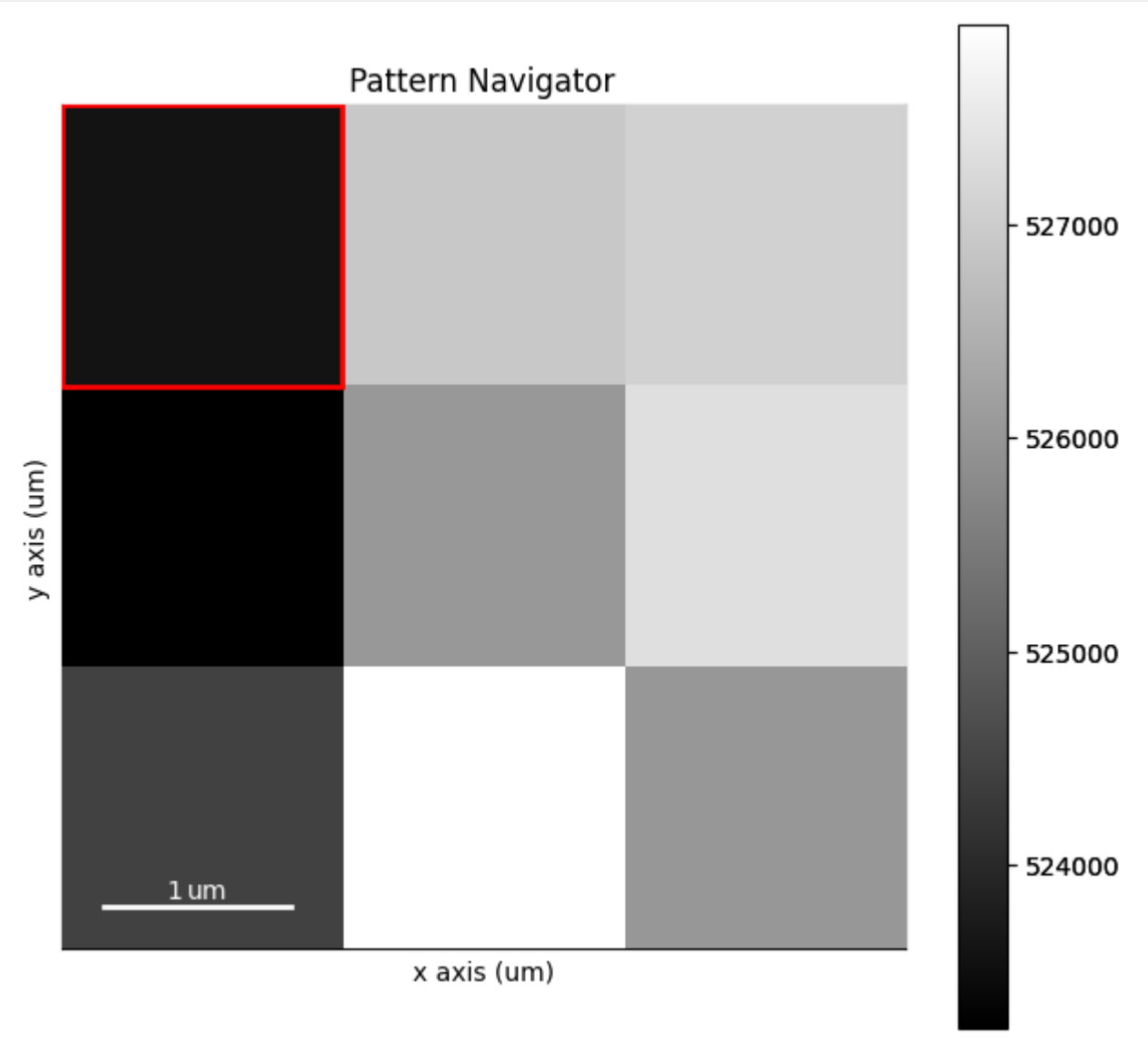
[6]:
```
<EBSD, title: Pattern, dimensions: (3, 3|60, 60)>
```

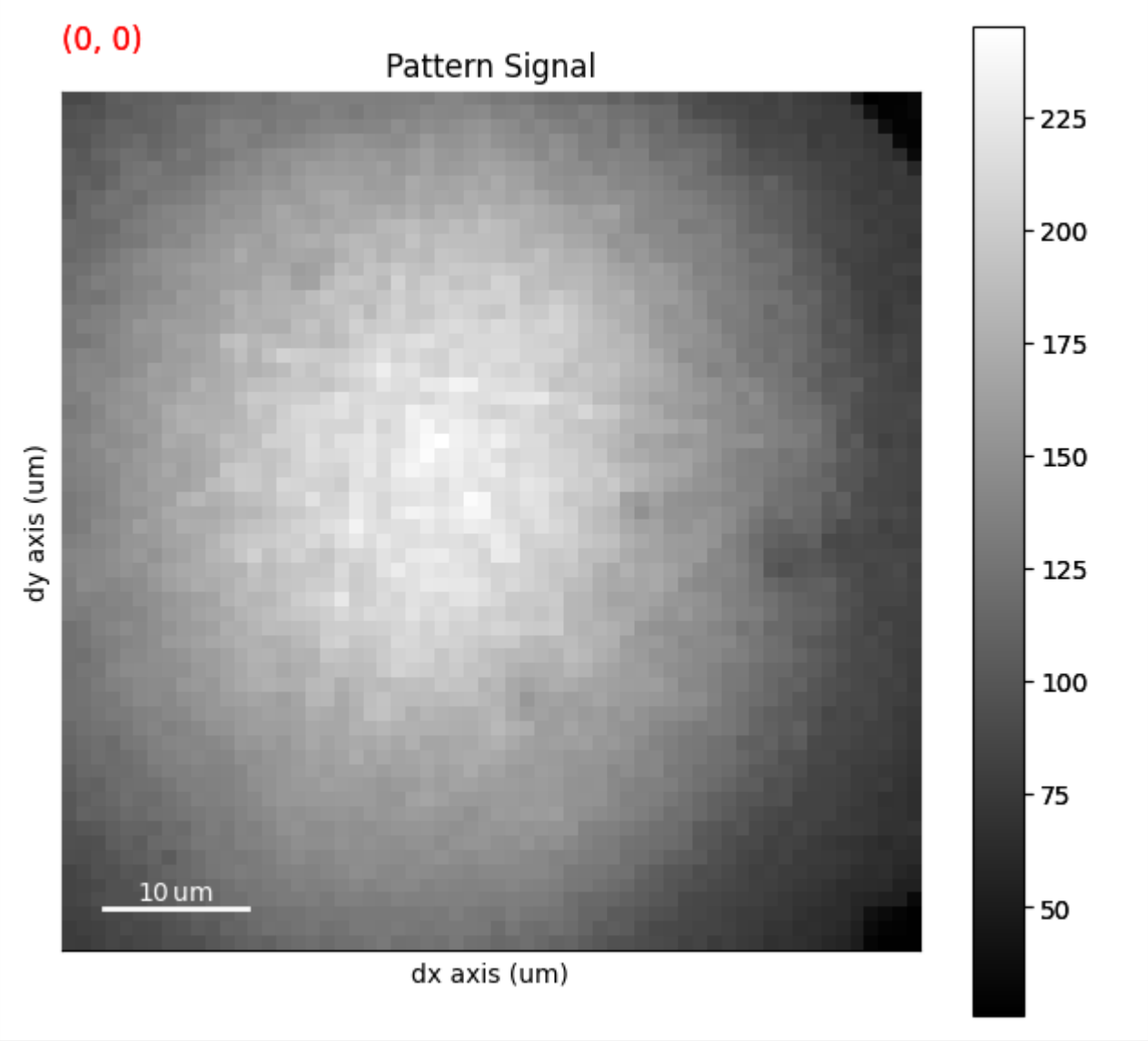
**Note**

When lazily loaded EBSD patterns are processed, they are processed chunk by chunk, which can lead to longer processing times, so processing lazy data sets should be done with some care. See the relevant HyperSpy user guide for some tips.

Patterns can be visualized by navigating the navigation space

[7]:
```
s.plot()
```


Pattern Navigator
y axis (um)
x axis (um)
1 um
527000
526000
525000
524000




Upon loading, kikuchipy reads some scan information from the file and stores it in the `original_metadata` attribute

[8]:
```
# s.original_metadata  # Long output
```

Other information is stored in a standard location in the `metadata` attribute in fields specified in the HyperSpy metadata structure; these are not used by methods in kikuchipy

[9]:
```
s.metadata
```

[9]:
```
├── Acquisition_instrument
│   └── SEM
│       ├── beam_energy = 20.0
│       ├── magnification = 200
│       ├── microscope = Hitachi SU-6600
│       └── working_distance = 24.7
├── General
│   ├── original_filename = ../../kikuchipy/data/nordif/Pattern.dat
```

(continues on next page)

(continued from previous page)

```
    └── title = Pattern
└── Signal
    └── signal_type = EBSD
```

The number of patterns in horizontal and vertical direction, pattern size in pixels, scan step size and detector pixel size are stored in the `axes_manager` attribute

[10]:
```
print(s.axes_manager)
```

```
<Axes manager, axes: (3, 3|60, 60)>
            Name |   size |  index |  offset |   scale |  units
================ | ====== | ====== | ======= | ======= | ======
               x |      3 |      0 |       0 |     1.5 |     um
               y |      3 |      0 |       0 |     1.5 |     um
---------------- | ------ | ------ | ------- | ------- | ------
              dx |     60 |      0 |       0 |       1 |     um
              dy |     60 |      0 |       0 |       1 |     um
```

This information can be modified directly, and information in `metadata` and `axes_manager` can be modified by the *EBSD* class methods *set_scan_calibration()* and *set_detector_calibration()*.

In addition to the `metadata`, `original_metadata` and `axes_manager` properties, kikuchipy tries to read the following from the file if available:

- a CrystalMap with indexing results into the *xmap* attribute
- an *EBSDDetector* describing the projection/pattern center (PC) values into the *detector* attribute
- a static background into the *static_background* attribute.

The `xmap` attribute is `None` if it is not read, while some readers return an "empty" crystal map with points containing the identity rotation and the same phase ID of an undefined phase with no point group set.

[11]:
```
s.xmap
```

[11]:
```
Phase  Orientations  Name  Space group  Point group  Proper point group     Color
    0    9 (100.0%)  None         None         None                None  tab:blue
Properties:
Scan unit: px
```

[12]:
```
s.xmap.rotations
```

[12]:
```
Rotation (9,)
[[1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]]
```

[13]:
```
s.xmap.phase_id
```

[13]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
[14]: s.detector
```

```
[14]: EBSDDetector (60, 60), px_size 1 um, binning 1, tilt 0.0, azimuthal 0.0, pc (0.5, 0.5, 0.
      ↪5)
```

```
[15]: s.detector.pc
```

```
[15]: array([[0.5, 0.5, 0.5]])
```

```
[16]: s.static_background
```

```
[16]: array([[84, 87, 90, ..., 27, 29, 30],
             [87, 90, 93, ..., 27, 28, 30],
             [92, 94, 97, ..., 39, 28, 29],
             ...,
             [80, 82, 84, ..., 36, 30, 26],
             [79, 80, 82, ..., 28, 26, 26],
             [76, 78, 80, ..., 26, 26, 25]], dtype=uint8)
```

**Note**

*New in version 0.8.*

The `EBSD` class inherits all methods from HyperSpy's Signal2D class. An effort is made to handle the mentioned attributes of `xmap`, `detector` and `static_background` when the navigation and/or signal dimensions are cropped, like when calling `EBSD.inav[]` or `EBSD.isig[]`. This handling is new, and so needs time and more tests to mature. It is considered experimental functionality.

### From a NumPy array

An `EBSD`, *EBSDMasterPattern* or *ECPMasterPattern* (electron channeling pattern) signal can be created directly from a NumPy array. To create a data set of (60 x 60) pixel patterns in a (10 x 20) grid, i.e. 10 and 20 patterns in the horizontal and vertical scan directions respectively, of random intensities

```
[17]: s_np = kp.signals.EBSD(np.random.random((20, 10, 60, 60)))
      s_np
```

```
[17]: <EBSD, title: , dimensions: (10, 20|60, 60)>
```

### From a Dask array

When processing large data sets, it is useful to load data lazily with the Dask library. This can be done upon reading patterns *from a file* by setting `lazy=True` when using the `load()` function, or directly from a `dask.array.Array`

```
[18]: s_da = kp.signals.LazyEBSD(
          da.random.random((20, 10, 60, 60), chunks=(2, 10, 60, 60))
      )
      print(s_da)
      s_da.data
```

```
<LazyEBSD, title: , dimensions: (10, 20|60, 60)>
```

[18]:
```
dask.array<random_sample, shape=(20, 10, 60, 60), dtype=float64, chunksize=(2, 10, 60,␣
↪60), chunktype=numpy.ndarray>
```

### From a HyperSpy signal

HyperSpy provides the method set_signal_type() to change between BaseSignal subclasses, of which `EBSD`, `EBSDMasterPattern`, `ECPMasterPattern` and *VirtualBSEImage* are four. To get one of these signals from a HyperSpy Signal2D signal

[19]:
```
s_hs = hs.signals.Signal2D(np.random.random((20, 10, 60, 60)))
s_hs
```

[19]:
```
<Signal2D, title: , dimensions: (10, 20|60, 60)>
```

[20]:
```
for signal_type in [
    "EBSD",
    "VirtualBSEImage",
    "ECPMasterPattern",
    "EBSDMasterPattern",
]:
    s_hs.set_signal_type(signal_type)
    print(s_hs)
```

```
<EBSD, title: , dimensions: (10, 20|60, 60)>
<VirtualBSEImage, title: , dimensions: (10, 20|60, 60)>
<ECPMasterPattern, title: , dimensions: (10, 20|60, 60)>
<EBSDMasterPattern, title: , dimensions: (10, 20|60, 60)>
```

### Save patterns

To save signals to file, use the *save()* method. For example, to save an `EBSD` signal in an HDF5 file in our default *h5ebsd format*, with file name `"patterns.h5"`

[21]:
```
temp_dir = Path(tempfile.mkdtemp())
s.save(temp_dir / "patterns")
```

**Warning**

If we want to overwrite an existing file:

```
s.save("patterns.h5", overwrite=True)
```

If we want to save patterns in *NORDIF's binary format* instead

[22]:
```
s.save(temp_dir / "patterns.dat")
```

```
WARNING:hyperspy.signal:The default iterpath will change in HyperSpy 2.0.
```

To save an `EBSDMasterPattern` to an HDF5 file, we use the save method inherited from HyperSpy to write to their HDF5 specification or zarr specification

[23]:
```
s_hs.save(temp_dir / "master_pattern.hspy")
s_hs
```

[23]:
```
<EBSDMasterPattern, title: , dimensions: (10, 20|60, 60)>
```

These master patterns can then be read into an `EBSDMasterPattern` signal again via HyperSpy's load()

[24]:
```
s_mp2 = hs.load(
    temp_dir / "master_pattern.hspy", signal_type="EBSDMasterPattern"
)
s_mp2
```

[24]:
```
<EBSDMasterPattern, title: , dimensions: (10, 20|60, 60)>
```

**Note**

To save results from statistical decomposition (machine learning) of patterns to file see the section Saving and loading results in HyperSpy's user guide. Note that the file extension `.hspy` or `.zspy` must be used upon saving, `s.save('patterns.hspy')`, as the default extension in kikuchipy, `.h5`, yields a kikuchipy h5ebsd file where the decomposition results aren't saved. The saved patterns can then be reloaded using HyperSpy's `load()` function passing the `signal_type="EBSD"` parameter *as explained above*.

### Supported file formats

Currently, kikuchipy has readers and writers for the following formats:

| Format | Read | Write |
|---|---|---|
| *EMsoft simulated EBSD HDF5* | Yes | No |
| *EMsoft EBSD master pattern HDF5* | Yes | No |
| *EMsoft ECP master pattern HDF5* | Yes | No |
| *EMsoft TKD master pattern HDF5* | Yes | No |
| *kikuchipy h5ebsd* | Yes | Yes |
| *Bruker h5ebsd* | Yes | No |
| *EDAX h5ebsd* | Yes | No |
| *EDAX binary* | Yes | No |
| *NORDIF binary* | Yes | Yes |
| *NORDIF calibration patterns* | Yes | No |
| *Oxford Instruments binary* | Yes | No |
| *Oxford Instruments h5ebsd (H5OINA)* | Yes | No |
| *Directory of EBSD patterns* | Yes | No |

**Note**

If you want to process your patterns with kikuchipy, but use an unsupported EBSD vendor software, or if you want to write your processed patterns to a vendor format that does not support writing, please request this feature in our issue tracker.

### EMsoft simulated EBSD HDF5

Dynamically simulated EBSD patterns returned by EMsoft's `EMEBSD.f90` program as HDF5 files can be read as an `EBSD` signal

[25]:
```
emsoft_ebsd = "emsoft_ebsd/simulated_ebsd.h5"  # Dummy data set
s_sim = kp.load(datadir / emsoft_ebsd)
s_sim
```

[25]:
```
<EBSD, title: simulated_ebsd, dimensions: (10|10, 10)>
```

Here, the EMsoft simulated EBSD *file_reader()* is called, which takes the optional argument `scan_size`. Passing `scan_size=(2, 5)` will reshape the pattern data shape from `(10, 10, 10)` to `(2, 5, 10, 10)`

[26]:
```
s_sim2 = kp.load(datadir / emsoft_ebsd, scan_size=(2, 5))
print(s_sim2)
print(s_sim2.data.shape)
```

```
<EBSD, title: simulated_ebsd, dimensions: (5, 2|10, 10)>
(2, 5, 10, 10)
```

Simulated EBSD patterns can be written to the *kikuchipy h5ebsd format*, the *NORDIF binary format*, or to HDF5/zarr files using HyperSpy's HDF5/zarr specifications *as explained above*.

### EMsoft EBSD master pattern HDF5

Master patterns returned by EMsoft's `EMEBSDmaster.f90` program as HDF5 files can be read as an *EBSDMasterPattern* signal

[27]:
```
s_mp = kp.load(datadir / emsoft_master_pattern)

print(s_mp)
print(s_mp.projection)
print(s_mp.hemisphere)
print(s_mp.phase)
```

```
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (|401, 401)>
stereographic
upper
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
```

Here, the EMsoft EBSD master pattern *file_reader()* is called, which takes the optional arguments `projection`, `hemisphere` and `energy`. The stereographic projection is read by default. Passing `projection="lambert"` will read the square Lambert projection instead. The upper hemisphere is read by default. Passing `hemisphere="lower"` or `hemisphere="both"` will read the lower hemisphere projection or both, respectively. Master patterns for all beam energies are read by default. Passing `energy=(10, 20)` or `energy=15` will read the master pattern(s) with beam energies from 10 to 20 keV, or just 15 keV, respectively

[28]:
```
s_mp = kp.load(
    datadir / emsoft_master_pattern,
    projection="lambert",
    hemisphere="both",
    energy=20,
```

(continues on next page)

(continued from previous page)

```
)

print(s_mp)
print(s_mp.projection)
print(s_mp.hemisphere)
```

```
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (2|401, 401)>
lambert
both
```

Master patterns can be written to HDF5/zarr files using HyperSpy's HDF5/zarr specification *as explained above*.

See [Jackson *et al.*, 2019] for a hands-on tutorial explaining how to simulate these patterns with EMsoft, and [Callahan and De Graef, 2013] for details of the underlying theory.

### EMsoft ECP master pattern HDF5

Master patterns returned by EMsoft's `EMECPmaster.f90` program as HDF5 files can be read as an *ECPMasterPattern* signal

[29]:
```
# Load a dummy pattern
s_mp = kp.load(datadir / "emsoft_ecp_master_pattern/ecp_master_pattern.h5")

print(s_mp)
print(s_mp.projection)
print(s_mp.hemisphere)
print(s_mp.phase)
```

```
<ECPMasterPattern, title: ecp_master_pattern, dimensions: (11|13, 13)>
stereographic
upper
<name: None. space group: I4/mcm. point group: 4/mmm. proper point group: 422. color:␣
↪tab:blue>
```

Here, the EMsoft ECP master pattern *file_reader()* is called, which supports the same parameters as the *EMsoft EBSD master pattern reader*.

### EMsoft TKD master pattern HDF5

Master patterns returned by EMsoft's `EMTKDmaster.f90` program as HDF5 files can be read as an *EBSDMasterPattern* signal

[30]:
```
# Load a dummy pattern
s_mp = kp.load(datadir / "emsoft_tkd_master_pattern/tkd_master_pattern.h5")

print(s_mp)
print(s_mp.projection)
print(s_mp.hemisphere)
print(s_mp.phase)
```

```
<EBSDMasterPattern, title: tkd_master_pattern, dimensions: (11|13, 13)>
stereographic
```

(continues on next page)

(continued from previous page)

```
upper
<name: None. space group: I4/mcm. point group: 4/mmm. proper point group: 422. color:␣
↪tab:blue>
```

Here, the EMsoft TKD master pattern *file_reader()* is called, which supports the same parameters as the *EMsoft EBSD master pattern reader*.

### kikuchipy h5ebsd

The h5ebsd format [Jackson *et al.*, 2014] is based on the HDF5 open standard (Hierarchical Data Format version 5). HDF5 files can be read and edited using e.g. the HDF Group's reader HDFView or the Python package h5py. Upon loading an HDF5 file with extension .h5, .hdf5, or .h5ebsd, the correct reader is determined from the file. Other supported h5ebsd formats are listed in the *table above*.

If an h5ebsd file contains multiple scans, as many scans as desirable can be read from the file. For example, if the file contains two scans with names `"Scan 1"` and `"Scan 2"`

[31]:
```
kikuchipy_ebsd = "kikuchipy_h5ebsd/patterns.h5"
s1, s2 = kp.load(
    datadir / kikuchipy_ebsd, scan_group_names=["Scan 1", "Scan 2"]
)
print(s1)
print(s2)
```

```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
<EBSD, title: patterns Scan 2, dimensions: (3, 3|60, 60)>
```

Here, the h5ebsd *file_reader()* is called. If only `"Scan 1"` is to be read, `scan_group_names="Scan 1"` can be passed

[32]:
```
s1 = kp.load(datadir / kikuchipy_ebsd, scan_group_names="Scan 1")
s1
```

[32]:
```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
```

The `scan_group_names` parameter is unnecessary if only the first scan in the file is to be read, since reading only the first scan in the file is the default behaviour.

So far, only *saving patterns* to kikuchipy's own h5ebsd format is supported. It is possible to write a new scan with a scan name `"Scan x"`, where `x` is an integer, to an existing, but closed, h5ebsd file in the kikuchipy format, e.g. one containing only `Scan 1`, by passing

[33]:
```
new_file = "patterns_new.h5"
s1.save(temp_dir / new_file, scan_number=1)
s2.save(temp_dir / new_file, add_scan=True, scan_number=2)

s1_new, s2_new = kp.load(
    temp_dir / new_file, scan_group_names=["Scan 1", "Scan 2"]
)
print(s1_new)
print(s2_new)
```

```
<EBSD, title: patterns_new Scan 1, dimensions: (3, 3|60, 60)>
<EBSD, title: patterns_new Scan 2, dimensions: (3, 3|60, 60)>
```

Here, the h5ebsd *file_writer()* is called. The EBSD class attributes `xmap` (crystal map), `detector` (EBSD detector) and `static_background` are written to and read from file if available.

### Bruker h5ebsd

Bruker Nano's h5ebsd files with extension .h5, .hdf5, or .h5ebsd can be read. Available parameters when calling `kikuchipy.load("bruker_patterns.h5")` are described in the *file_reader()*. As with the kikuchipy h5ebsd reader, multiple scans can be read at once.

The projection center (PC) arrays are read into the `detector` attribute, and the static background pattern is read into the `static_background` attribute. The orientation and phase data are so far not read, but note that this can be read using orix.

### EDAX h5ebsd

EDAX's h5ebsd files with extension .h5, .hdf5, or .h5ebsd can be read. Available parameters when calling `kikuchipy.load("edax_patterns.h5")` are described in the *file_reader()*. As with the kikuchipy h5ebsd reader, multiple scans can be read at once.

The projection center (PC) arrays are read into the `detector` attribute. The orientation and phase data are so far not read.

### EDAX binary

Patterns stored in the EDAX TSL's binary UP1/2 file format, with intensities as 8-bit or 16-bit unsigned integer, can be read. File version 1 and >= 3 are supported.

[34]:
```
edax_binary_path = datadir / "edax_binary/"  # Directory with dummy signals
s_edax = kp.load(edax_binary_path / "edax_binary.up1")
s_edax
```

[34]:
```
<EBSD, title: edax_binary, dimensions: (9|60, 60)>
```

Here, the EDAX binary *file_reader()* is called.

Files with version 1 has no information on the navigation (map) shape, so we can pass this to the reader in the `nav_shape` parameter

[35]:
```
s_edax2 = kp.load(edax_binary_path / "edax_binary.up1", nav_shape=(3, 3))
s_edax2
```

[35]:
```
<EBSD, title: edax_binary, dimensions: (3, 3|60, 60)>
```

Patterns acquired in an hexagonal grid *can* be read, but the returned signal will have only one navigation dimension

[36]:
```
s_edax3 = kp.load(edax_binary_path / "edax_binary.up2")
s_edax3
```

```
/home/docs/checkouts/readthedocs.org/user_builds/kikuchipy/conda/latest/lib/python3.11/
↪site-packages/kikuchipy/io/plugins/edax_binary.py:152: UserWarning: Returned signal␣
↪has one navigation dimension since an hexagonal grid is not supported
  warnings.warn(
```

[36]:
```
<EBSD, title: edax_binary, dimensions: (10|60, 60)>
```

### NORDIF binary

Patterns acquired using NORDIF's acquisition software are stored in a binary file usually named "Pattern.dat". Scan information is stored in a separate text file usually named "Setting.txt", and both files usually reside in the same directory. If this is the case, the patterns can be loaded by passing the file name as the only parameter. If this is not the case, the setting file can be passed upon loading

[37]:
```
s_nordif = kp.load(
    datadir / nordif_ebsd, setting_file=datadir / "nordif/Setting.txt"
)
s_nordif
```

[37]:
```
<EBSD, title: Pattern, dimensions: (3, 3|60, 60)>
```

Here, the NORDIF *file_reader()* is called. If the scan information, i.e. scan and pattern size, in the setting file is incorrect or the setting file is not available, patterns can be loaded by passing

[38]:
```
s_nordif = kp.load(
    datadir / nordif_ebsd, scan_size=(1, 9), pattern_size=(60, 60)
)
s_nordif
```

[38]:
```
<EBSD, title: Pattern, dimensions: (9|60, 60)>
```

If a static background pattern named "Background acquisition pattern.bmp" is stored in the same directory as the pattern file, this is stored in `EBSD.static_background` upon loading

[39]:
```
s_nordif.static_background
```

[39]:
```
array([[84, 87, 90, ..., 27, 29, 30],
       [87, 90, 93, ..., 27, 28, 30],
       [92, 94, 97, ..., 39, 28, 29],
       ...,
       [80, 82, 84, ..., 36, 30, 26],
       [79, 80, 82, ..., 28, 26, 26],
       [76, 78, 80, ..., 26, 26, 25]], dtype=uint8)
```

Patterns can also be *saved to a NORDIF binary file*, upon which the NORDIF *file_writer()* is called. Note, however, that so far no new setting file, background pattern, or calibration patterns are created upon saving.

### NORDIF calibration patterns

NORDIF calibration patterns in bitmap format named "Calibration (x,y).bmp", where "x" and "y" correspond to coordinates listed in the NORDIF setting file, usually named "Setting.txt", can be loaded

[40]:
```
s_nordif_cal = kp.load(datadir / "nordif/Setting.txt")
s_nordif_cal
```

[40]:
```
<EBSD, title: Calibration patterns, dimensions: (2|60, 60)>
```

Here, the NORDIF calibration patterns *file_reader()* is called. Lazy loading is not supported for this reader, thus the `lazy` parameter is not used.

If a static background pattern named "Background calibration pattern.bmp" is stored in the same directory as the pattern file, this is stored in `EBSD.static_background` upon loading.

The pixel indices of the calibration patterns into the full electron image (named "Area.bmp" by the NORDIF software) are read into the `original_metadata`. The shapes of this image and the region of interest (ROI) of the scanned area within this image are also attempted to be read. All this information can be useful when obtaining a plane of projection centers (PCs) for all patterns in the full dataset based on these calibration patterns (see other user guide tutorials for this).

### Oxford Instruments binary

Uncompressed patterns stored in the Oxford Instruments binary .ebsp file format, with intensities as 8-bit or 16-bit unsigned integer, can be read

[41]:
```
oxford_binary_path = datadir / "oxford_binary"
s_oxford = kp.load(oxford_binary_path / "patterns.ebsp")
s_oxford
```

[41]:
```
<EBSD, title: patterns, dimensions: (3, 3|60, 60)>
```

Here, the Oxford Instruments binary *file_reader()* is called.

Every pattern's flattened index into the 2D navigation map, as well as their entry in the file (map order isn't always the same as file order) can be retrieved from `s_oxford.original_metadata.map1d_id` and `s_oxford.original_metadata.file_order`, respectively. If available in the file, every pattern's row and column beam position in microns can be retrieved from `s_oxford.original_metadata.beam_y` and `s_oxford.original_metadata.beam_x`, respectively. All these are 1D arrays.

[42]:
```
s_oxford.original_metadata
```

[42]:
```
├── beam_x = array([0. , 1.5, 3. , 0. , 1.5, 3. , 0. , 1.5, 3. ])
├── beam_y = array([0. , 0. , 0. , 1.5, 1.5, 1.5, 3. , 3. , 3. ])
├── file_order = array([8, 0, 1, 2, 3, 4, 5, 6, 7])
└── map1d_id = array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

If the beam positions aren't present in the file, the returned signal will have a single navigation dimension the size of the number of patterns.

Files with only the non-indexed patterns can also be read. The returned signal will then have a single navigation dimension the size of the number of patterns. The flattened index into the 2D navigation map mentioned above can be useful to determine the location of each non-indexed pattern.

### Oxford Instruments h5ebsd (H5OINA)

Oxford Instruments' h5ebsd files (H5OINA) with extension .h5oina can be read. Available parameters when calling `kikuchipy.load("patterns.h5oina")` are described in the *file_reader()*. As with the kikuchipy h5ebsd reader, multiple scans can be read at once.

The projection center (PC) arrays are read into the `detector` attribute and the static background pattern is read into the `static_background` attribute. The orientation and phase data are so far not read.

### Directory of EBSD patterns

Many EBSD patterns in image files in a directory can be read as an `EBSD` signal, assuming all images with that file extension have the same shape and data type. Valid formats are those supported by imageio.

To demonstrate how the images can be read, we will first write nine patterns to a temporary with their filenames formatted a certain way

[43]:
```
temp_dir2 = Path(tempfile.mkdtemp())
s = kp.data.nickel_ebsd_small()
y, x = np.indices(s.axes_manager.navigation_shape[::-1])
y = y.ravel()
x = x.ravel()
s.unfold_navigation_space()
for i in range(s.axes_manager.navigation_size):
    iio.imwrite(temp_dir2 / f"pattern_x{x[i]}y{y[i]}.tif", s.data[i])

# Print file contents
os.listdir(temp_dir2)
```

[43]:
```
['pattern_x1y1.tif',
 'pattern_x2y2.tif',
 'pattern_x0y0.tif',
 'pattern_x2y1.tif',
 'pattern_x0y2.tif',
 'pattern_x1y2.tif',
 'pattern_x0y1.tif',
 'pattern_x1y0.tif',
 'pattern_x2y0.tif']
```

If filenames are formatted like this `_x0y0.tif` or `-0-0.png`, we do not have to specify this file name pattern when reading the images

[44]:
```
s1 = kp.load(temp_dir2 / "*.tif")
s1
```

[44]:
```
<EBSD, title: , dimensions: (3, 3|60, 60)>
```

Here *file_reader()* of this plugin was called. If filenames are formatted some other way, we have to pass this as a regular expression to `xy_pattern`

[45]:
```
s2 = kp.load(
    temp_dir2 / "*.tif",
    xy_pattern=r"_x(\d+)y(\d+).tif",
    show_progressbar=True,
)
s2
```

```
[########################################] | 100% Completed | 104.25 ms
```

[45]:
```
<EBSD, title: , dimensions: (3, 3|60, 60)>
```

Here we also printed a progressbar when reading the patterns from file, which can be useful when we want to read a large number of images.

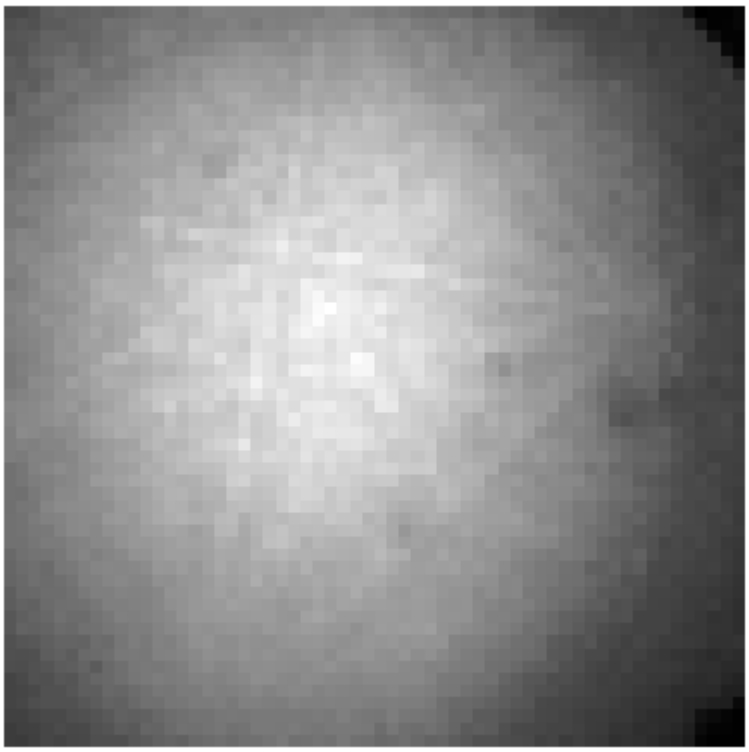
### From kikuchipy into other software

Patterns saved in the *h5ebsd format* can be read by the dictionary indexing and related routines in EMsoft using the `EMEBSD` reader. Those routines in EMsoft also have a `NORDIF` reader.

Patterns saved in the *h5ebsd format* can of course be read in Python like any other HDF5 data set

[46]:
```
import h5py

with h5py.File(datadir / kikuchipy_ebsd, mode="r") as f:
    dset = f["Scan 2/EBSD/Data/patterns"]
    print(dset)
    patterns = dset[()]
    print(patterns.shape)
    plt.figure()
    plt.imshow(patterns[0], cmap="gray")
    plt.axis("off")
```

```
<HDF5 dataset "patterns": shape (9, 60, 60), type "|u1">
(9, 60, 60)
```



### Load and save virtual BSE images

One or more virtual backscatter electron (BSE) images in a *VirtualBSEImage* signal can be read and written to file using one of HyperSpy's many readers and writers. If they are only to be used internally in HyperSpy, they can be written to and read back from HyperSpy's HDF5/zarr specification *as explained above for EBSD master patterns*.

If we want to write the images to image files, HyperSpy also provides a series of image readers/writers, as explained in their IO user guide. If we wanted to write them as a stack of TIFF images

[47]:
```
# Get virtual image from image generator
vbse_imager = kp.imaging.VirtualBSEImager(s)
print(vbse_imager)

print(vbse_imager.grid_shape)
vbse = vbse_imager.get_images_from_grid()
print(vbse)
```

```
VirtualBSEImager for <EBSD, title: patterns Scan 1, unfolded dimensions: (9|60, 60)>
(5, 5)
<VirtualBSEImage, title: , dimensions: (5|9, 5)>
```

[48]:
```
vbse.rescale_intensity()
vbse.unfold_navigation_space()  # 1D navigation space required for TIFF
vbse
```

```
WARNING:hyperspy.signal:The function you applied does not take into account the␣
↪difference of units and of scales in-between axes.
```

```
[########################################] | 100% Completed | 101.14 ms
```

[48]:
```
<VirtualBSEImage, title: , dimensions: (5|9, 5)>
```

[49]:
```
vbse_fname = "vbse.tif"
vbse.save(temp_dir / vbse_fname)  # Easily read into e.g. ImageJ
```

We can also write them to e.g. `png` or `bmp` files with `Matplotlib`

[50]:
```
nav_size = vbse.axes_manager.navigation_size
for i in range(nav_size):
    plt.imsave(temp_dir / f"vbse{i}.png", vbse.inav[i].data)
```

Read the TIFF stack back into a `VirtualBSEImage` signal

[51]:
```
vbse2 = hs.load(temp_dir / vbse_fname, signal_type="VirtualBSEImage")
vbse2
```

[51]:
```
<VirtualBSEImage, title: , dimensions: (5|9, 5)>
```

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### Visualizing patterns

The *EBSD* and *EBSDMasterPattern* signals have a powerful and versatile plot() method provided by HyperSpy. The method's uses are greatly detailed in HyperSpy's visualization user guide. This section details example uses specific to EBSD and EBSDMasterPattern signals.

Let's import the necessary libraries and a nickel EBSD test data set [Ånes *et al.*, 2019]

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import pyvista as pv
import skimage.exposure as ske
import skimage.transform as skt

import hyperspy.api as hs
import kikuchipy as kp
from orix import io, plot, quaternion, vector


# See https://docs.pyvista.org/user-guide/jupyter/index.html
pv.set_jupyter_backend("static")
```

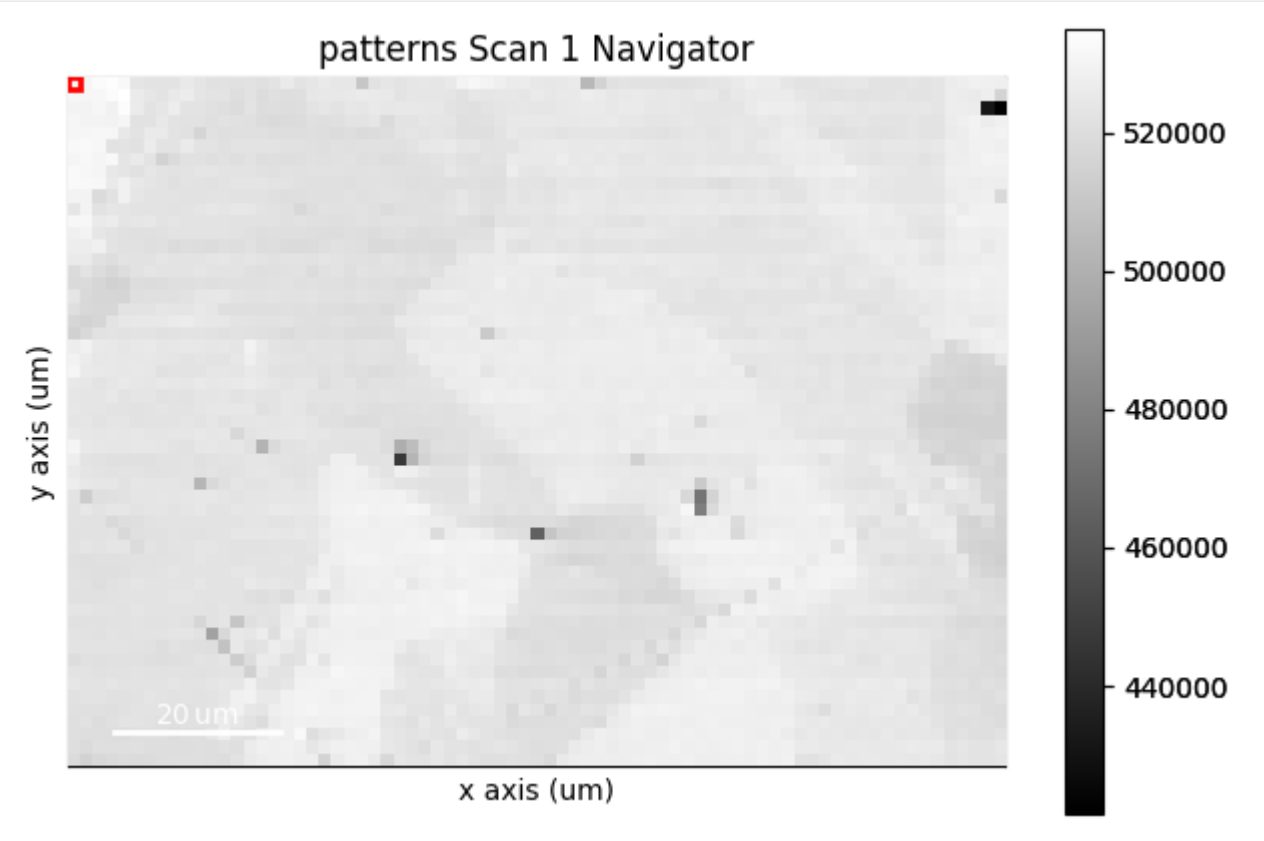
[2]:
```
# Use kp.load("data.h5") to load your own data
s = kp.data.nickel_ebsd_large(allow_download=True)  # External download
s
```

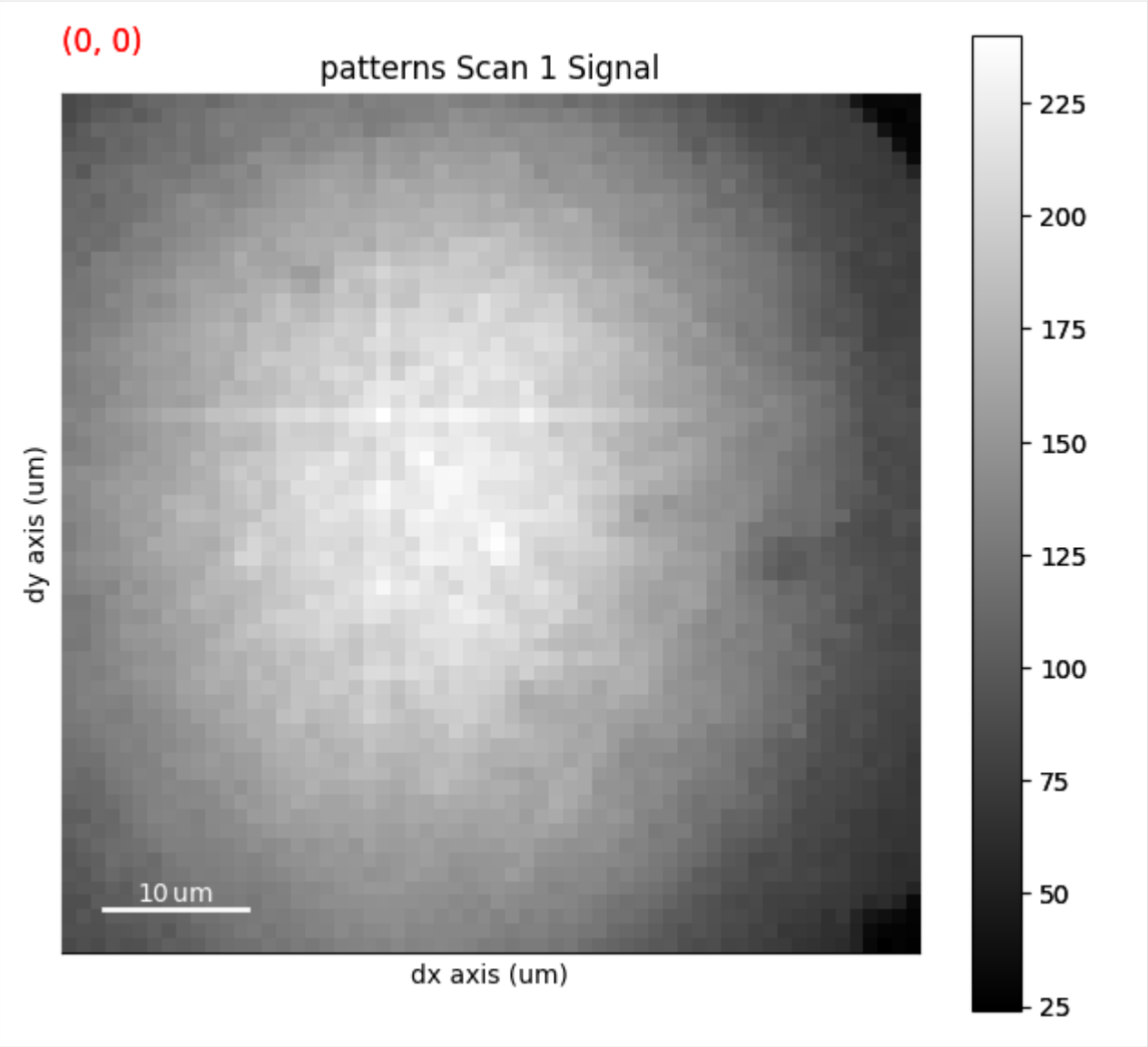
[2]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

### Navigate in custom map

Correlating results from e.g. crystal and phase structure determination, i.e. indexing, with experimental patterns is important when validating the indexing results. When calling `plot()` without any input parameters, the navigator map is a grey scale image with pixel values corresponding to the sum of all detector intensities within that pattern

[3]:
```
s.plot()
```


patterns Scan 1 Navigator
y axis (um)
x axis (um)
20 um
520000
500000
480000
460000
440000




The upper panel shows the navigation axes, in this case 2D. The current navigation position is highlighted in the upper left corner as a red square the size of one pixel. We can change the size of the square with +/-. The square can be moved either by the keyboard arrows or the mouse. The lower panel shows the pattern on the detector in the current navigation position.

Any BaseSignal signal with a 2D `signal_shape` corresponding to the scan `navigation_shape` can be passed in to the `navgiator` parameter in plot(). This includes a virtual image showing diffraction contrast, any quality metric map, or an inverse pole figure (IPF) or phase map.

### Virtual image

A virtual backscatter electron (VBSE) image created from any detector region of interest with the *get_virtual_bse_intensity()* method or *get_rgb_image()* explained in the *virtual backscatter electron imaging* tutorial, can be used as a navigator for a scan `s`

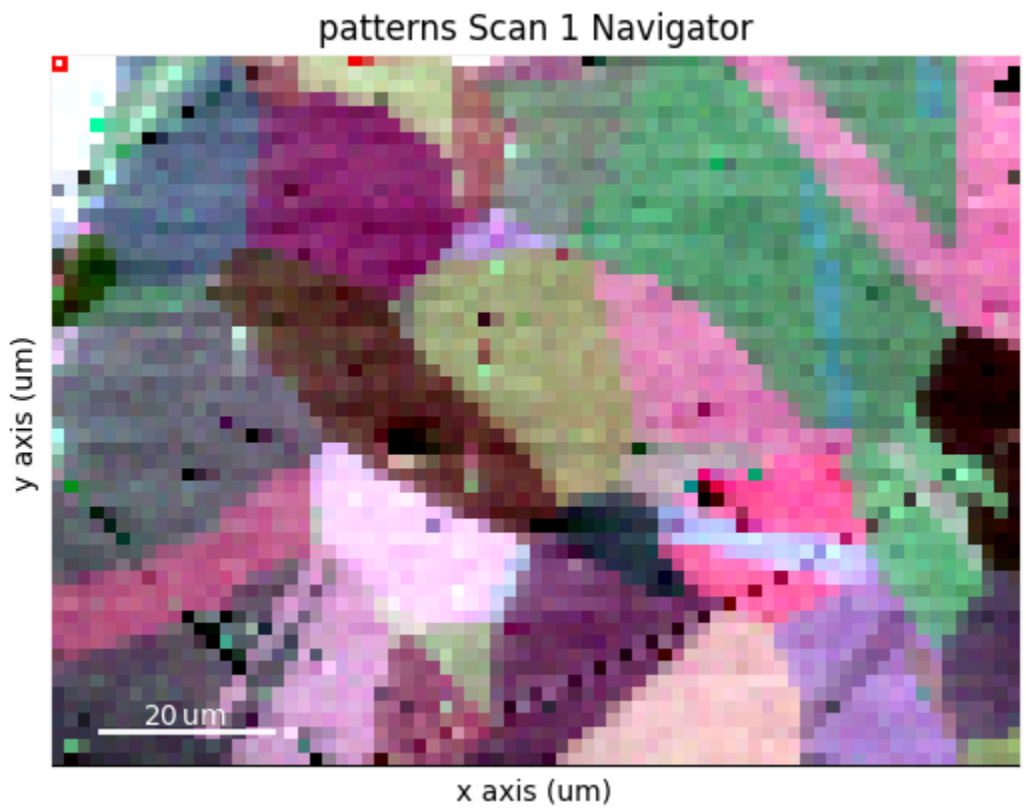
[4]:
```
vbse_imager = kp.imaging.VirtualBSEImager(s)
print(vbse_imager)
print(vbse_imager.grid_shape)
```

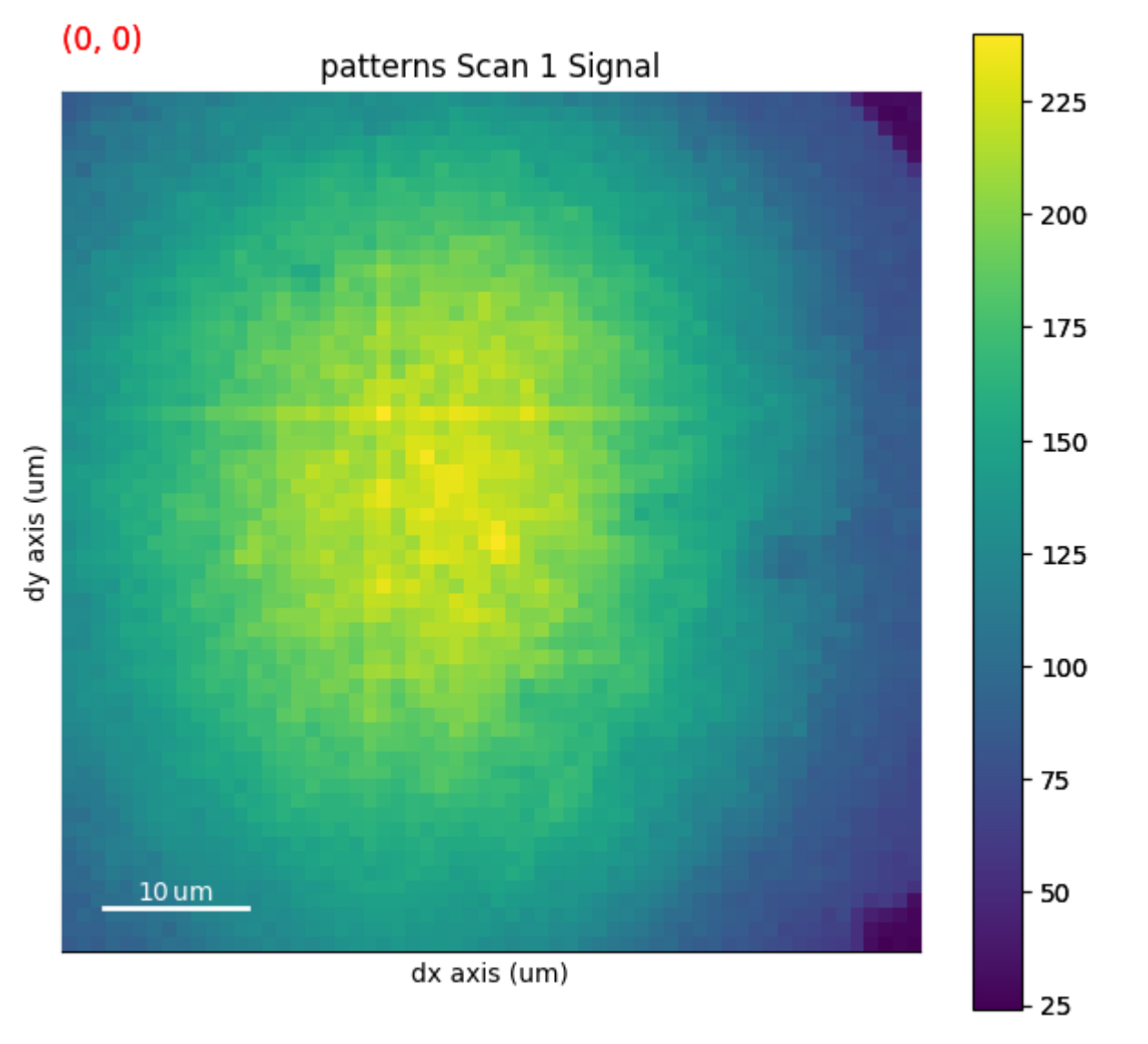
```
VirtualBSEImager for <EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
(5, 5)
```

[5]:
```
maps_vbse_rgb = vbse_imager.get_rgb_image(r=(3, 1), b=(3, 2), g=(3, 3))
maps_vbse_rgb
```

[5]:
```
<VirtualBSEImage, title: , dimensions: (|75, 55)>
```

[6]:
```
s.plot(navigator=maps_vbse_rgb, cmap="viridis")
```






### Any image

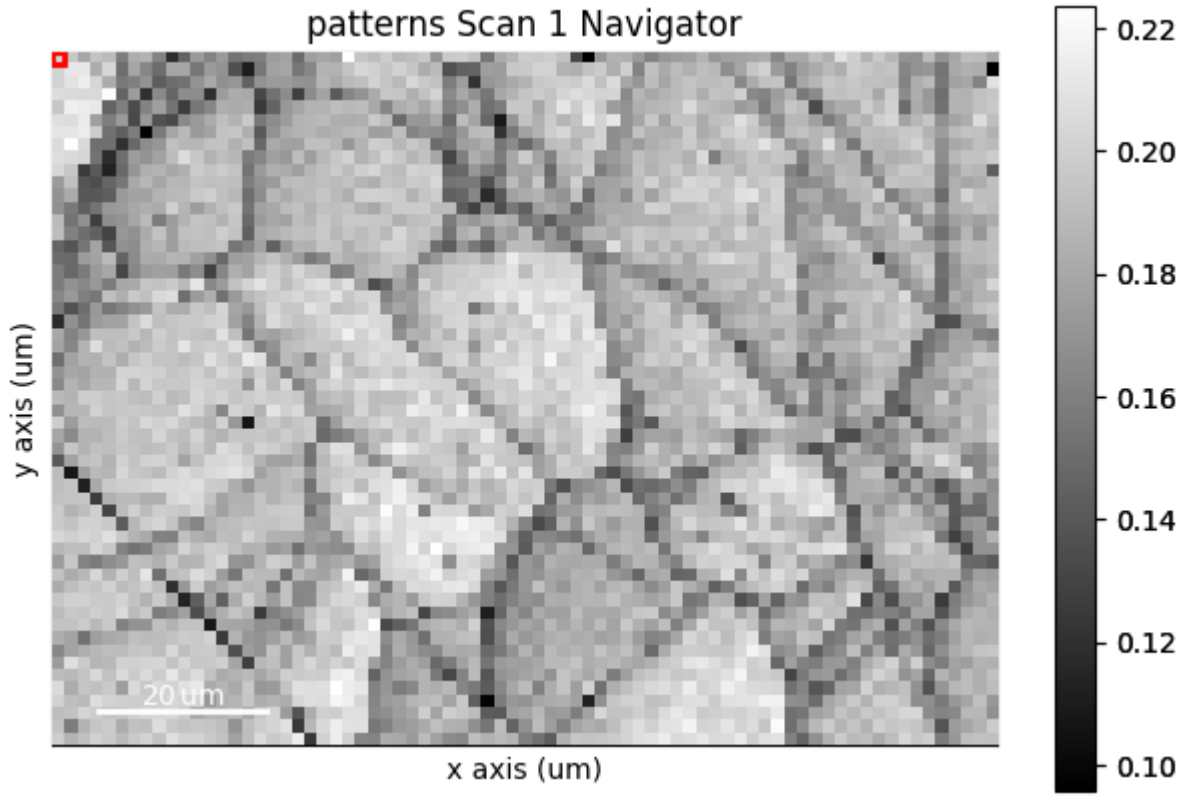
An image made into a Signal2D can be used as navigators. This includes quality metric maps such as the *image quality map*, calculated using *get_image_quality()*

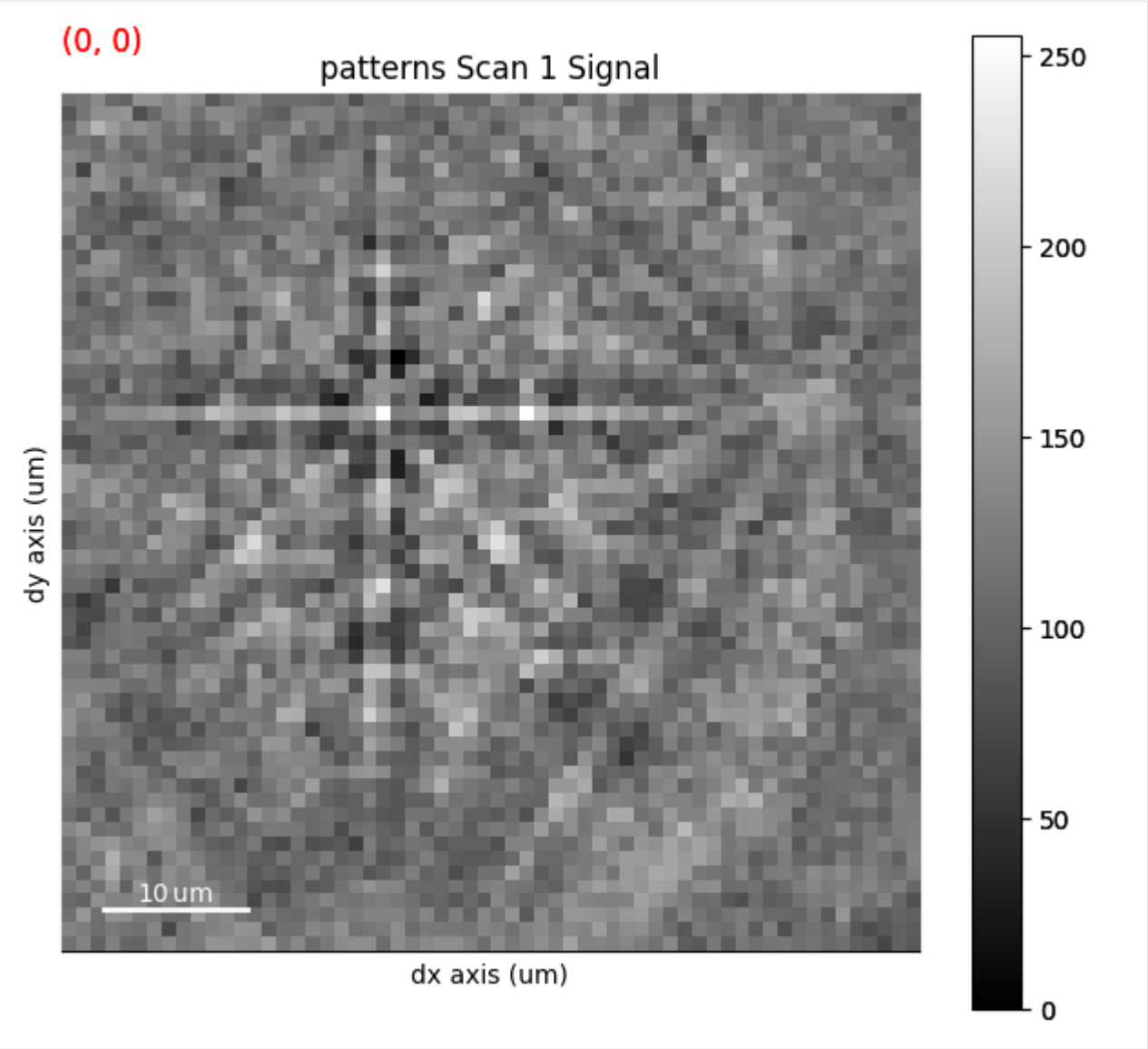
[7]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 103.06 ms
[########################################] | 100% Completed | 709.66 ms
```

[8]:
```
maps_iq = s.get_image_quality()
s_iq = hs.signals.Signal2D(maps_iq)
s.plot(navigator=s_iq)
```

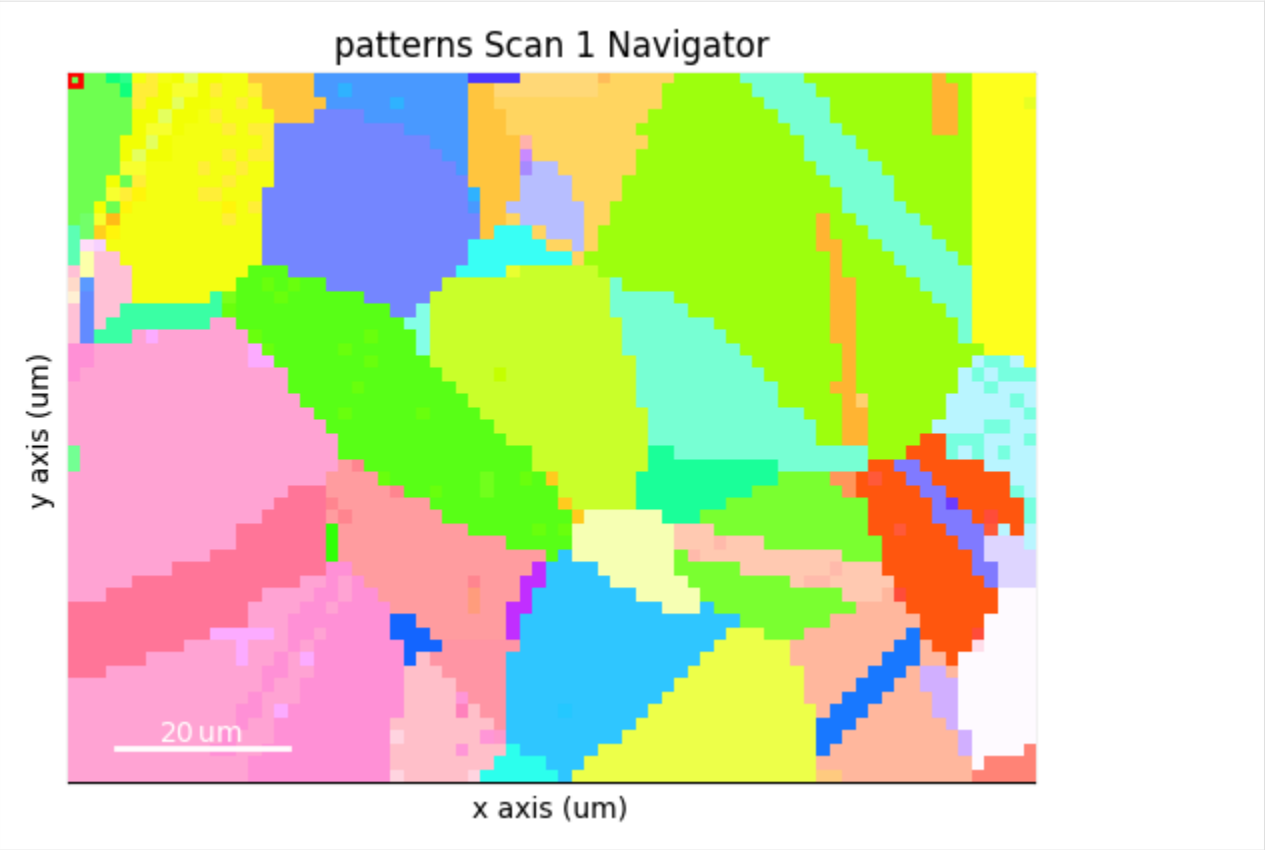
```
[########################################] | 100% Completed | 406.18 ms
```

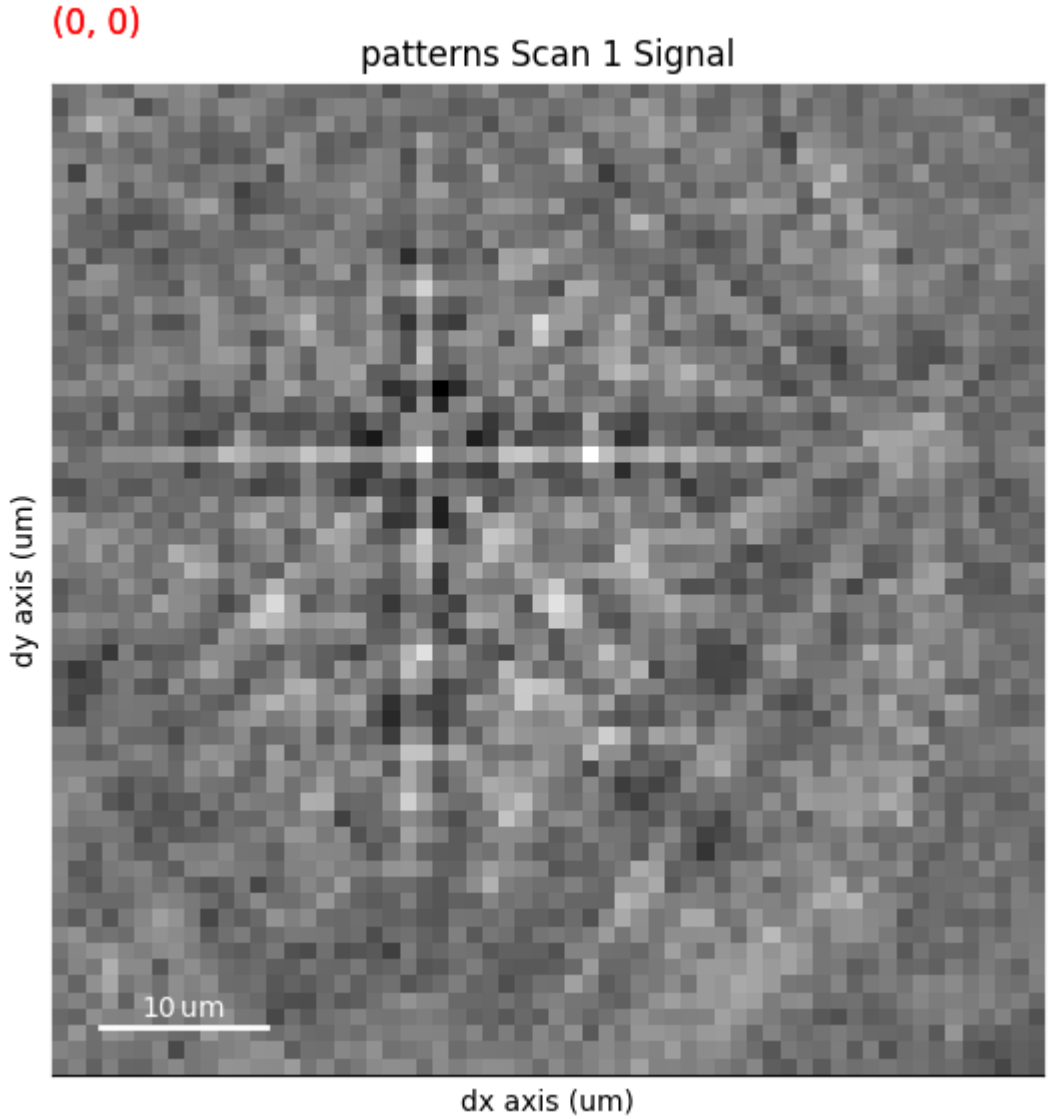





We can obtain an RGB signal from an RGB image using *get_rgb_navigator()*. Let's load an IPF-Z map representing orientations obtained from dictionary indexing in the *pattern matching* tutorial

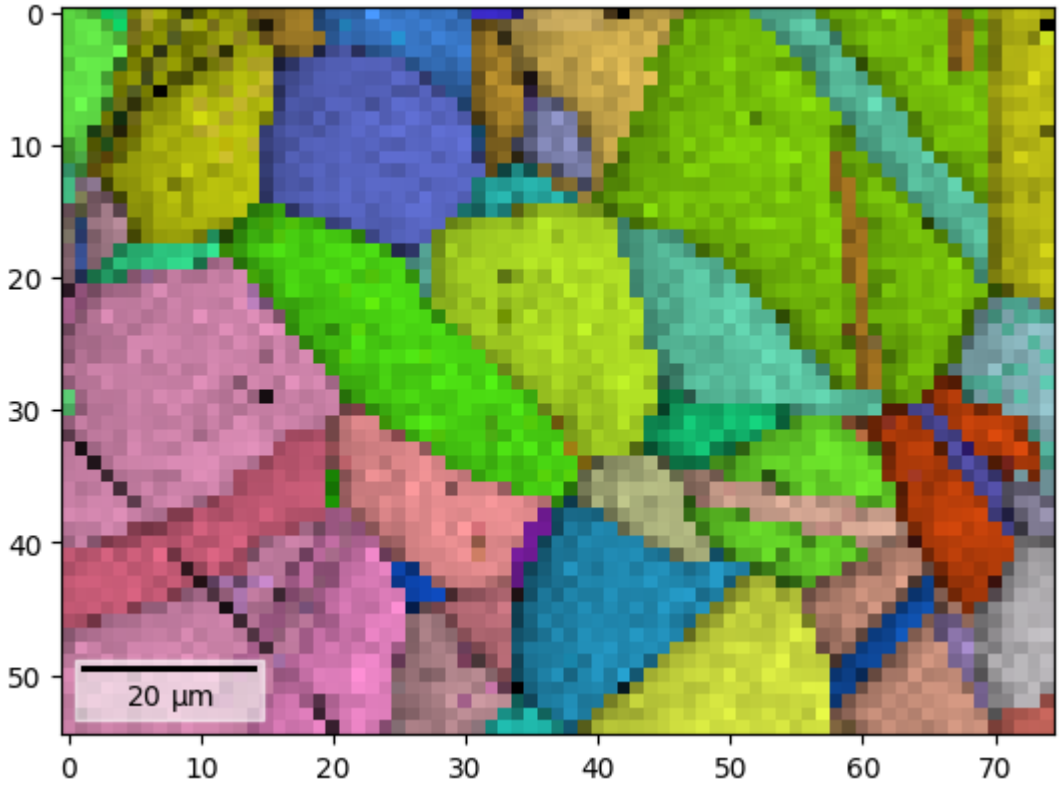
[9]:
```
maps_ipfz = plt.imread("../_static/image/visualizing_patterns/ni_large_rgb_z.png")
maps_ipfz = maps_ipfz[..., :3]  # Drop the alpha channel
s_ipfz = kp.draw.get_rgb_navigator(maps_ipfz)

s.plot(navigator=s_ipfz, colorbar=False)
```


patterns Scan 1 Navigator
y axis (um)
20 um
x axis (um)



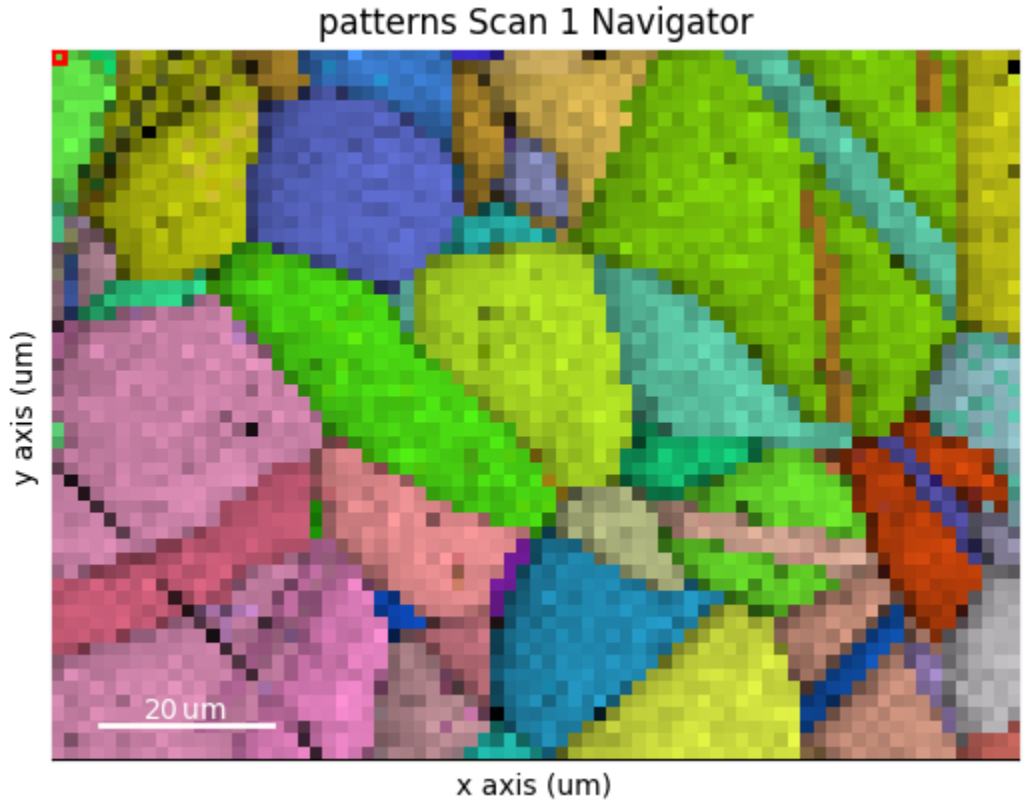

By overlaying the image quality map on the RGB image, we can visualize crystal directions within grains and the grain morphology in the same image

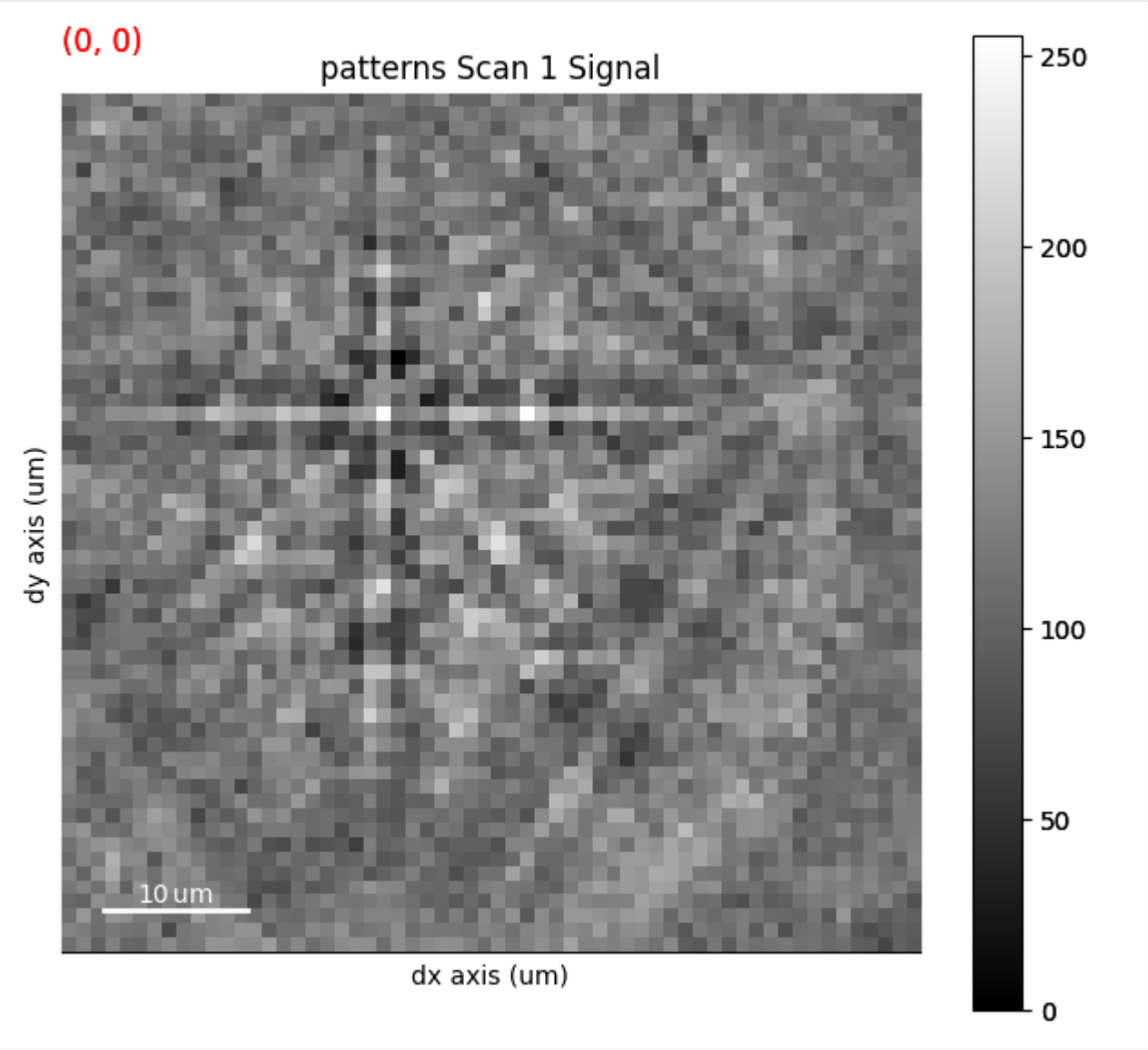
[10]:
```
maps_iq_1d = maps_iq.ravel()  # Flat array required by orix
maps_ipfz_1d = maps_ipfz.reshape(-1, 3)
fig = s.xmap.plot(maps_ipfz_1d, overlay=maps_iq_1d, return_figure=True)
```




By extracting the image array, we can use this map to navigate patterns in

[11]:
```
maps_ipfz_iq = fig.axes[0].images[0].get_array()
s_ipfz_iq = kp.draw.get_rgb_navigator(maps_ipfz_iq)
s.plot(s_ipfz_iq)
```


patterns Scan 1 Navigator
y axis (um)
20 um
x axis (um)




### Plot multiple signals

HyperSpy provides the function plot_signals() to plot multiple signals with the same navigator (detailed in their documentation). Among other uses, this function enables plotting of the experimental and best matching simulated patterns side by side. This can be a powerful visual validation of indexing results. See the *pattern matching tutorial* for a demonstration.

### Plot master patterns

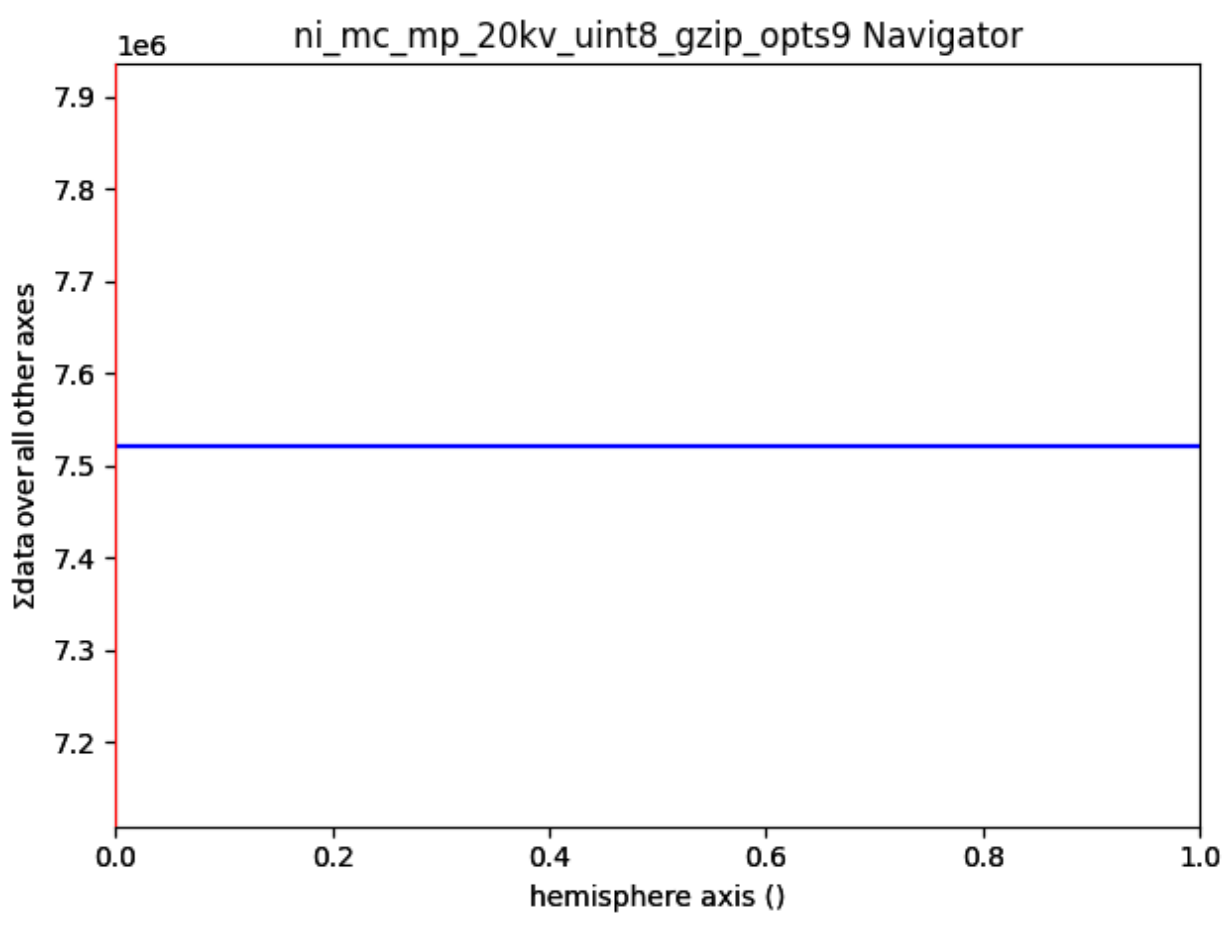
*EBSDMasterPattern* signals can be navigated along their energy axis and/or their upper/lower hemispheres. Let's reload the nickel master pattern used in the previous section, but this time in the stereographic projection.

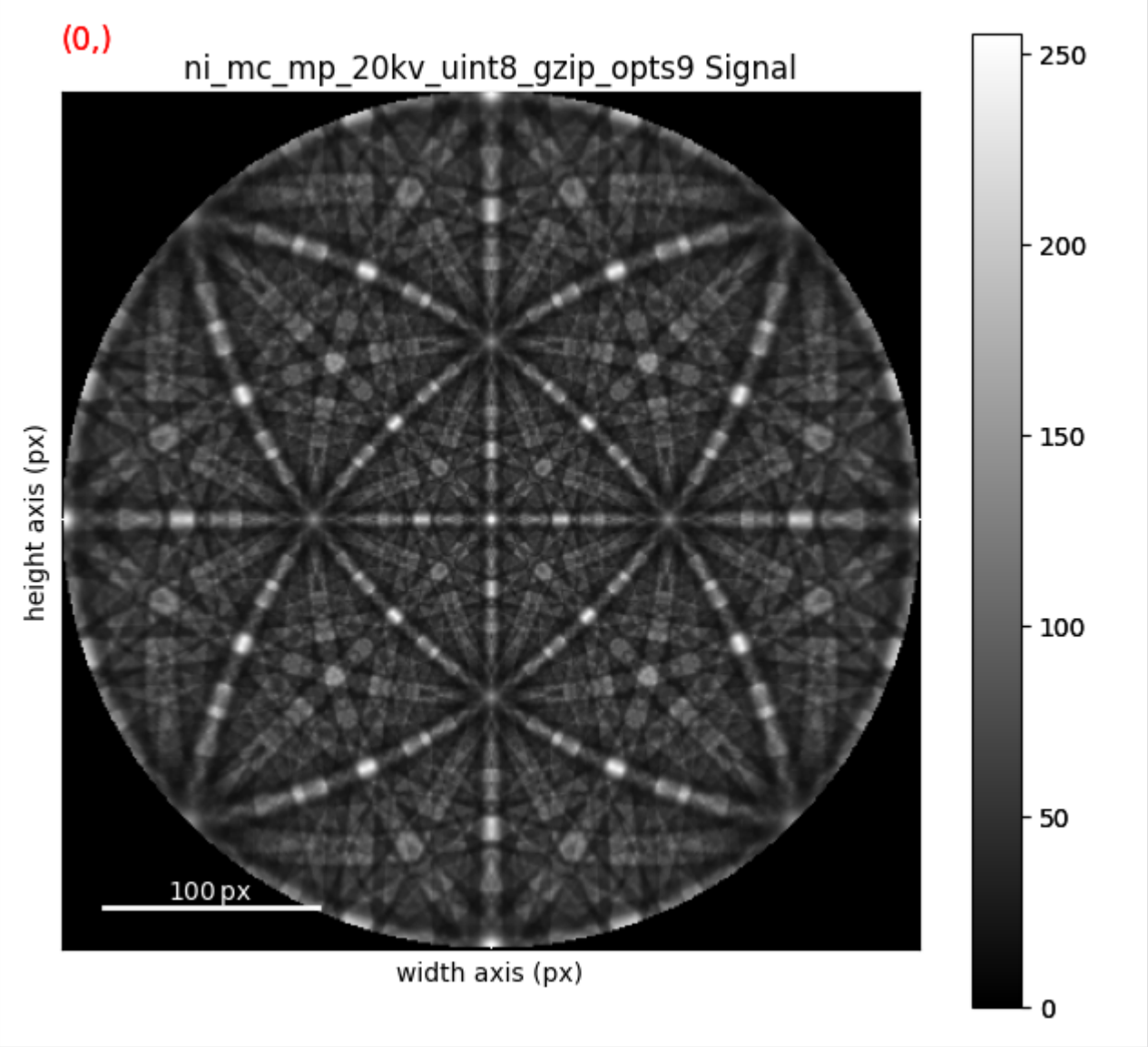
[12]:
```
# Only a single energy, 20 keV
mp_stereo = kp.data.nickel_ebsd_master_pattern_small(
    projection="stereographic", hemisphere="both"
)
print(mp_stereo.axes_manager)
```

```
<Axes manager, axes: (2|401, 401)>
            Name |   size |  index |  offset |   scale |  units
================ | ====== | ====== | ======= | ======= | ======
      hemisphere |      2 |      0 |       0 |       1 |
---------------- | ------ | ------ | ------- | ------- | ------
           width |    401 |      0 |  -2e+02 |       1 |     px
          height |    401 |      0 |  -2e+02 |       1 |     px
```

As can be seen from the axes manager, the master pattern has two navigation axes, the upper and lower hemispheres. When plotting, we therefore get a navigation slider

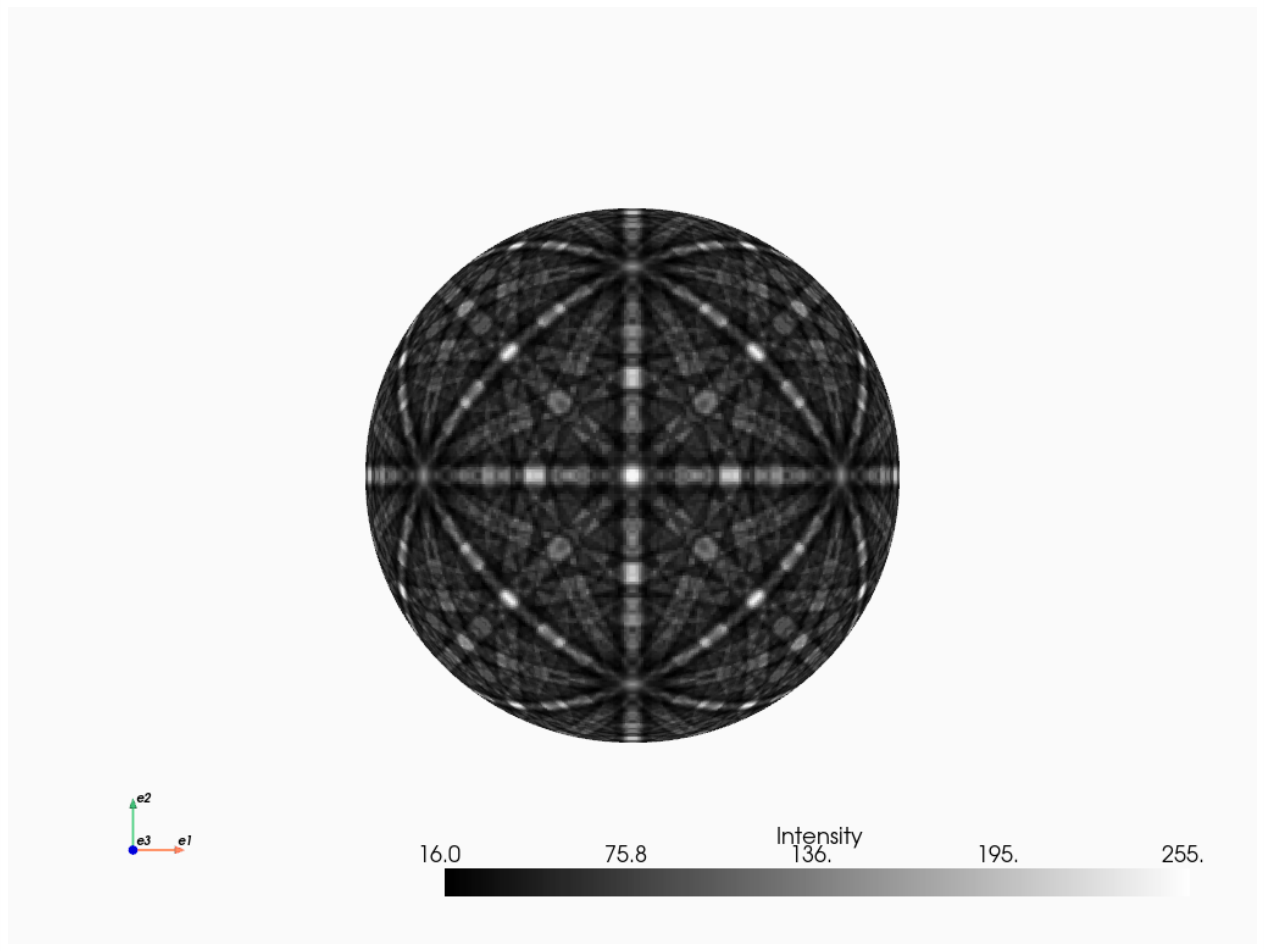
[13]:
```
mp_stereo.plot()
```






We can plot the master pattern on the sphere with *EBSDMasterPattern.plot_spherical()*. This visualization requires the master pattern to be in the stereographic projection. If the corresponding phase is centrosymmetry, the upper and lower hemispheres are identical, so we only need one of them to cover the sphere. If the phase is non-centrosymmetric, however, both hemispheres must be loaded, as they are unequal. The initial orientation of the sphere corresponds to the orientation of the stereographic and Lambert projections.

[14]:
```
mp_stereo.plot_spherical(style="points")
```




`PyVista`, required for this plot, is an optional dependency of kikuchipy (see *the installation guide* for details). Here, the plot uses the static Jupyter backend supported by PyVista. The backend was set in the first notebook cell. When running the notebook locally, we can make the plot interactive setting the backend to `"trame"`. We can pass `plotter_kwargs={"notebook": False}` to `plot_spherical()` if we want to plot the master pattern in a separate window.

---

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

---

## Pattern processing

The raw EBSD signal can be characterized as a superposition of a Kikuchi diffraction pattern and a smooth background intensity. For pattern indexing, the latter intensity is undesirable, while for *virtual backscatter electron VBSE) imaging*, this intensity can reveal topographical, compositional or diffraction contrast.

This tutorial details methods to enhance the Kikuchi diffraction pattern and manipulate detector intensities in patterns in an *EBSD* signal.

Most of the methods operating on EBSD objects use functions that operate on the individual patterns (`numpy.ndarray`). Some of these single pattern functions are available in the *kikuchipy.pattern* module.

Let's import the necessary libraries and read the Nickel EBSD test data set

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

import hyperspy.api as hs
import kikuchipy as kp
```

[2]:
```
# Use kp.load("data.h5") to load your own data
s = kp.data.nickel_ebsd_small()
```

Most methods operate inplace (indicated in their docstrings), meaning they overwrite the patterns in the EBSD signal. If we instead want to keep the original signal and operate on a new signal, we can either create a *deepcopy()* of the original signal…

[3]:
```
s2 = s.deepcopy()
np.may_share_memory(s.data, s2.data)
```

[3]:
```
False
```

… or pass `inplace=False` to return a new signal and keep the original signal unaffected (new in version 0.8).

Let's make a convenience function to plot a pattern before and after processing with the intensity distributions below. We'll also globally silence progressbars

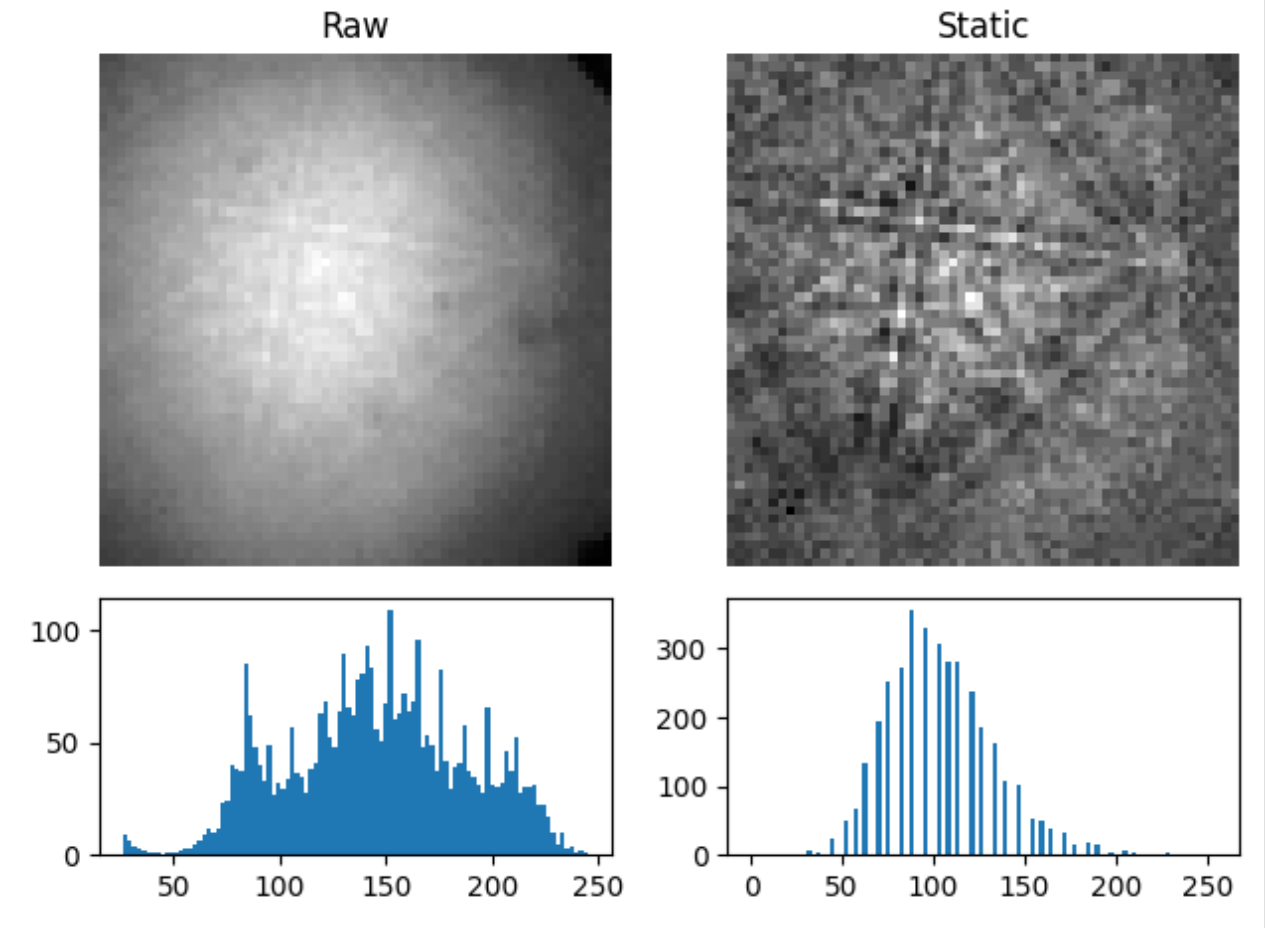
[4]:
```
def plot_pattern_processing(patterns, titles):
    """Plot two patterns side by side with intensity histograms below.

    Parameters
    ----------
    patterns : list of numpy.ndarray
    titles : list of str
    """
    fig, axes = plt.subplots(2, 2, height_ratios=[3, 1.5])
    for ax, pattern, title in zip(axes[0], patterns, titles):
        ax.imshow(pattern, cmap="gray")
        ax.set_title(title)
        ax.axis("off")
    for ax, pattern in zip(axes[1], patterns):
        ax.hist(pattern.ravel(), bins=100)
    fig.tight_layout()


hs.preferences.General.show_progressbar = False
```

### Background correction

### Remove the static background

Effects which are constant, like hot pixels or dirt on the detector, can be removed by subtracting or dividing by a static background wit *remove_static_background()*

[5]:
```
s2 = s.remove_static_background(inplace=False)

plot_pattern_processing(
    [s.inav[0, 0].data, s2.inav[0, 0].data], ["Raw", "Static"]
)
```



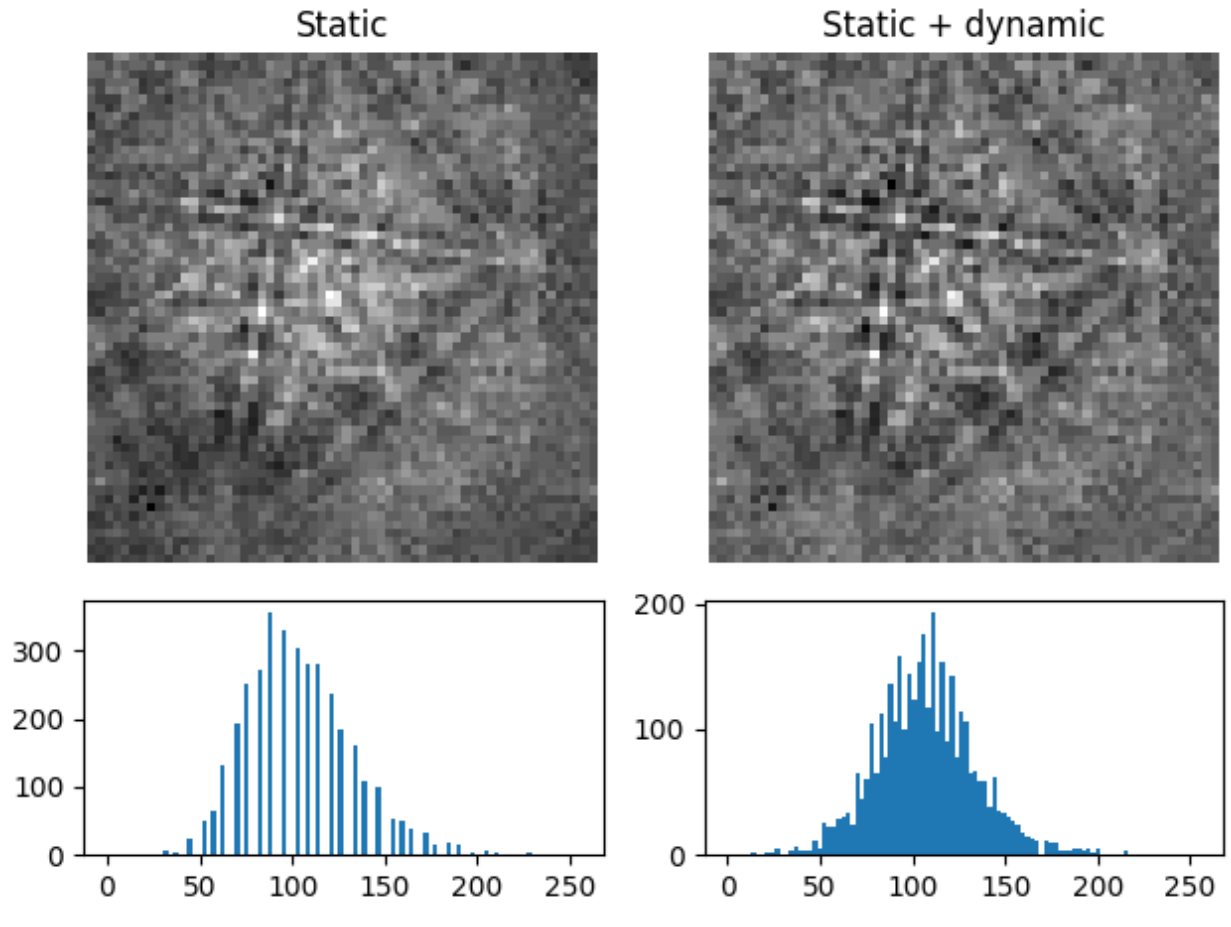

We didn't have to pass a background pattern since it is stored with the currenct signal in the *static_background* attribute. We could instead pass the background pattern in the `static_bg` parameter.

The static background pattern intensity range can be scaled to each individual pattern's range prior to removal with `scale_bg=True`.

### Remove the dynamic background

Uneven intensity in a static background subtracted pattern can be corrected by subtraction or division of a dynamic background pattern obtained by Gaussian blurring with *remove_dynamic_background()*. A Gaussian window with a standard deviation set by `std` is used to blur each pattern individually (dynamic) either in the spatial or frequency domain, set by `filter_domain`. Blurring in the frequency domain uses a low-pass *Fast Fourier Transform (FFT) filter*. Each pattern is then subtracted or divided by the individual dynamic background pattern depending on the `operation`

[6]:
```
s3 = s2.remove_dynamic_background(
    operation="subtract",  # Default
    filter_domain="frequency",  # Default
    std=8,  # Default is 1/8 of the pattern width
    truncate=4,  # Default
    inplace=False,
)

plot_pattern_processing(
    [s2.inav[0, 0].data, s3.inav[0, 0].data], ["Static", "Static + dynamic"]
)
```

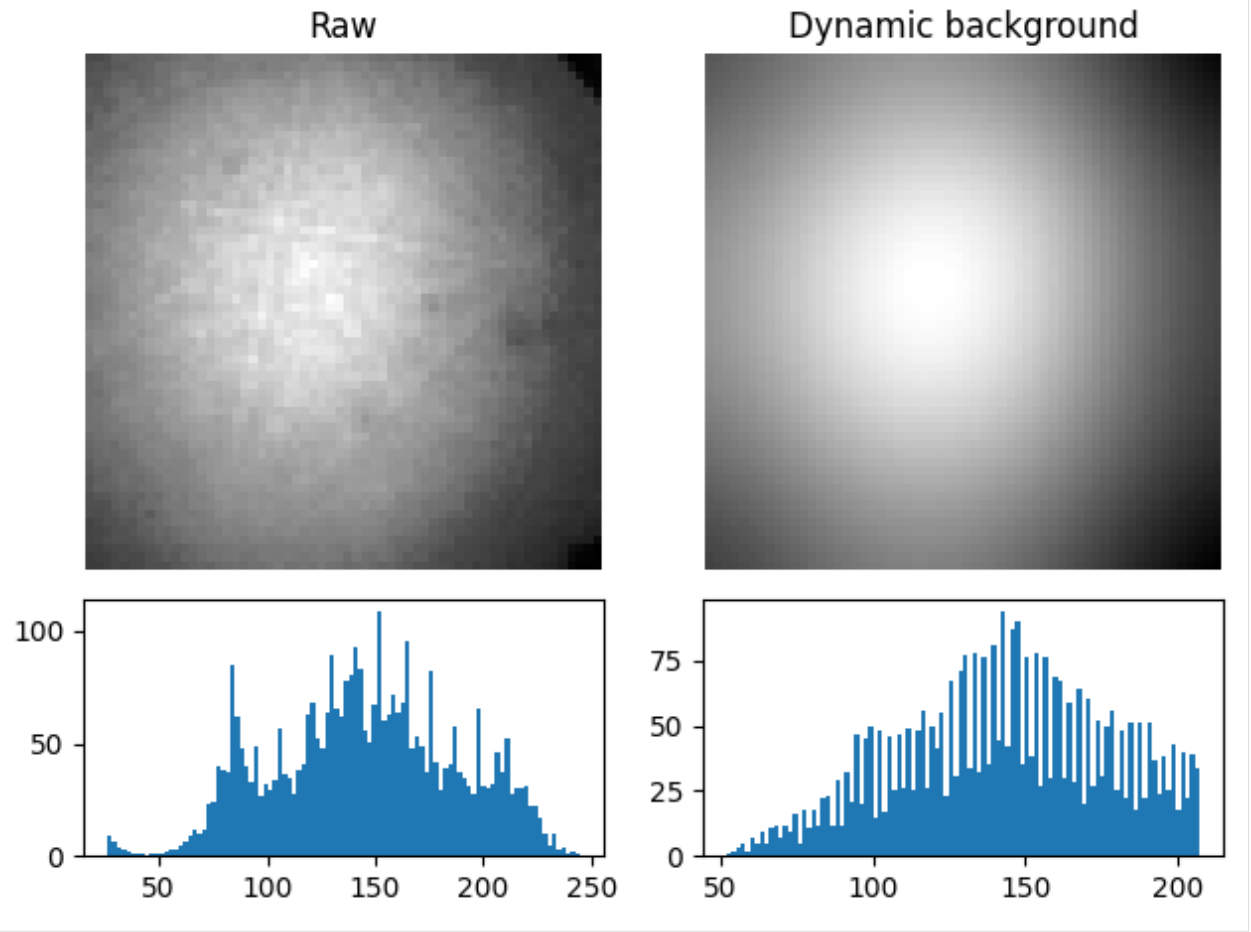



The width of the Gaussian window is truncated at the `truncated` number of standard deviations. Output patterns are rescaled to fill the input patterns' data type range.

### Get the dynamic background

The Gaussian blurred pattern removed during dynamic background correction can be obtained as an EBSD signal by calling *get_dynamic_background()*

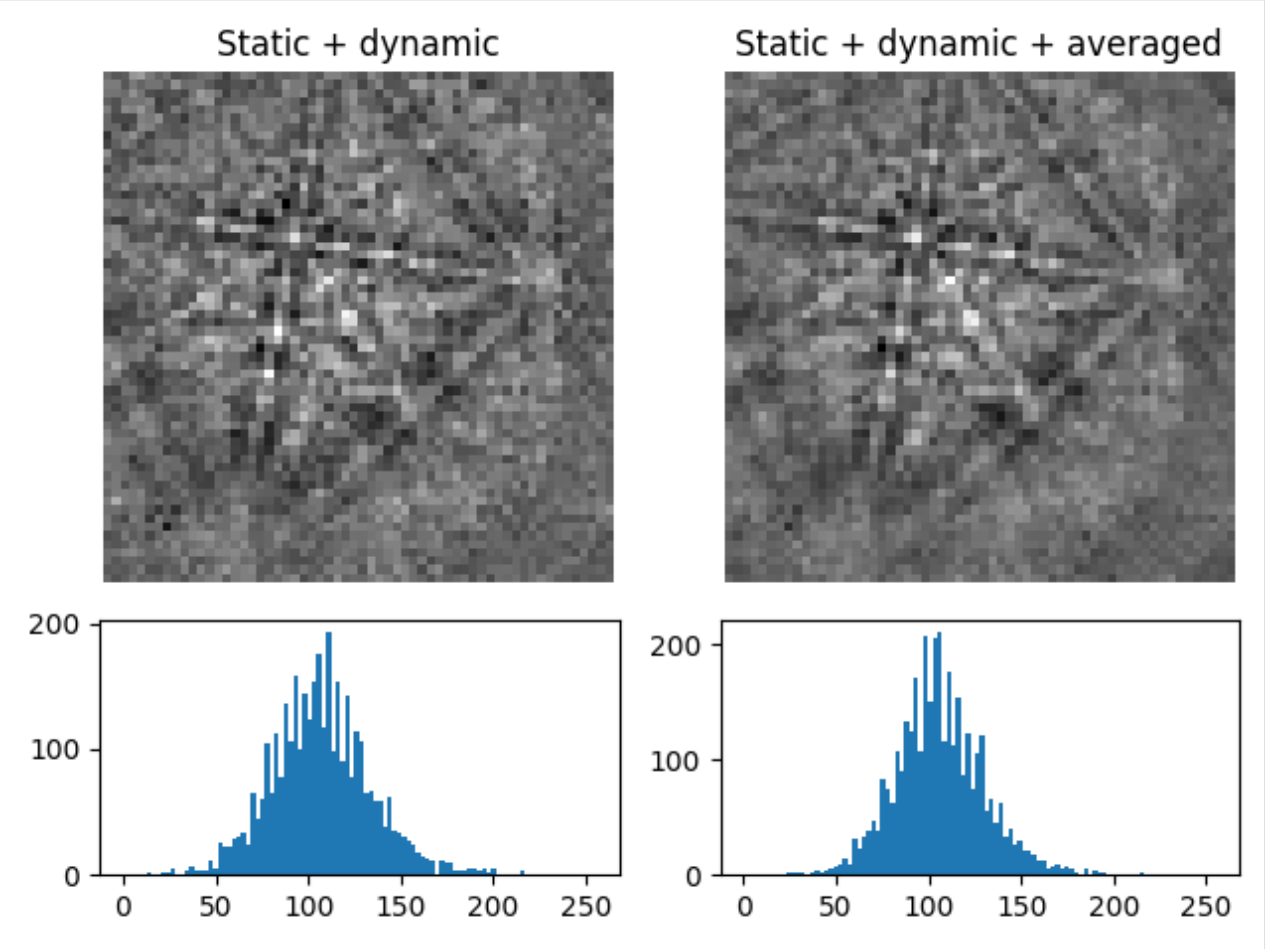
[7]:
```
bg = s.get_dynamic_background(filter_domain="frequency", std=8, truncate=4)

plot_pattern_processing(
    [s.inav[0, 0].data, bg.inav[0, 0].data], ["Raw", "Dynamic background"]
)
```




### Average neighbour patterns

The signal-to-noise ratio in patterns in a scan can be improved by averaging patterns with their closest neighbours within a window/kernel/mask using *average_neighbour_patterns()*

[8]:
```
s4 = s3.average_neighbour_patterns(window="gaussian", std=1, inplace=False)

plot_pattern_processing(
    [s3.inav[0, 0].data, s4.inav[0, 0].data],
    ["Static + dynamic", "Static + dynamic + averaged"],
)
```

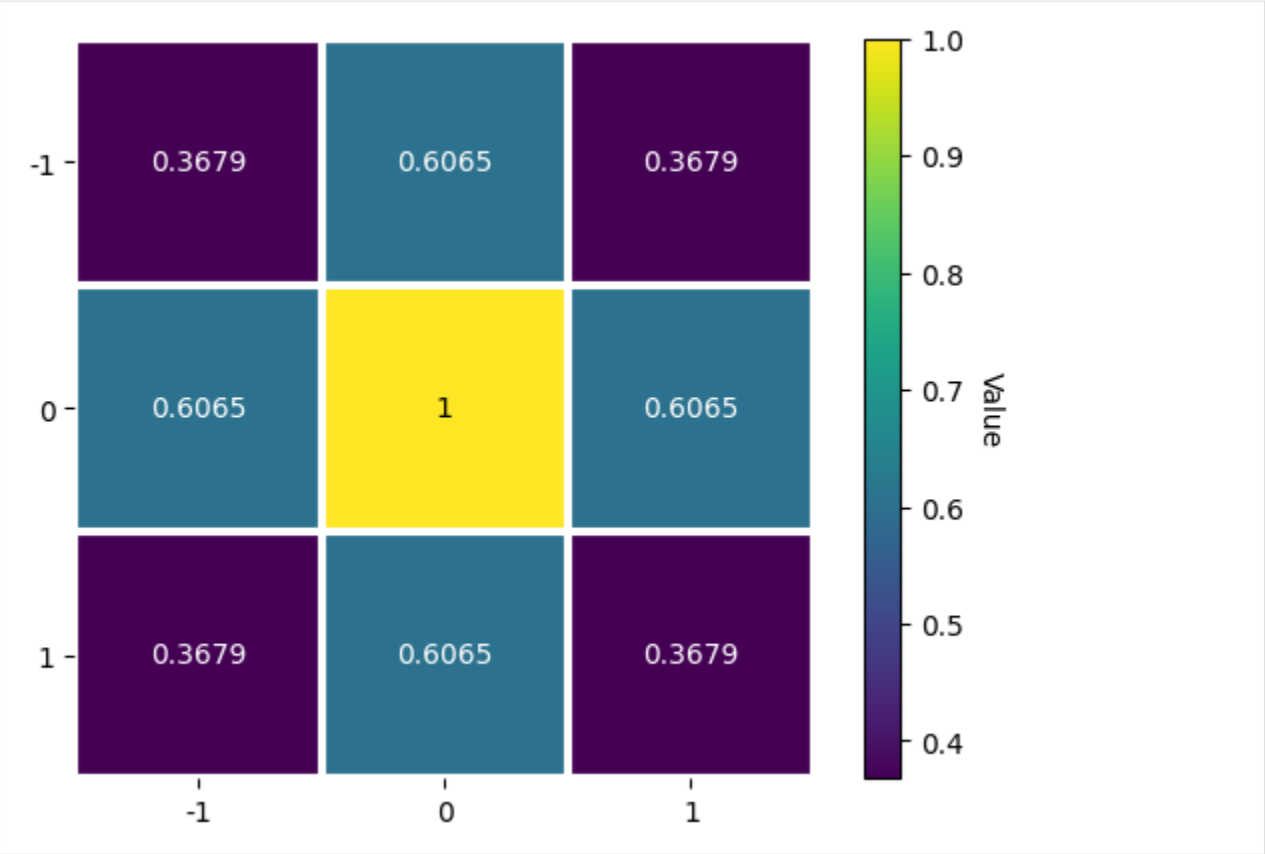



The array of averaged patterns $g(n_\mathrm{x}, n_\mathrm{y})$ is obtained by spatially correlating a window $w(s,t)$ with the array of patterns $f(n_\mathrm{x}, n_\mathrm{y})$, here 4D, which is padded with zeros at the edges. As coordinates $n_\mathrm{x}$ and $n_\mathrm{y}$ are varied, the window origin moves from pattern to pattern, computing the sum of products of the window coefficients with the neighbour pattern intensities, defined by the window shape, followed by normalizing by the sum of the window coefficients. For a symmetrical window of shape $m \times n$, this becomes [Gonzalez and Woods, 2017]

$$g(n_\mathrm{x}, n_\mathrm{y}) = \frac{\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(n_\mathrm{x}+s, n_\mathrm{y}+t)}{\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)}, \tag{1.1}$$

where $a = (m-1)/2$ and $b = (n-1)/2$. The window $w$, a *Window* object, can be plotted

[9]:
```
w = kp.filters.Window(window="gaussian", shape=(3, 3), std=1)
w.plot()
```

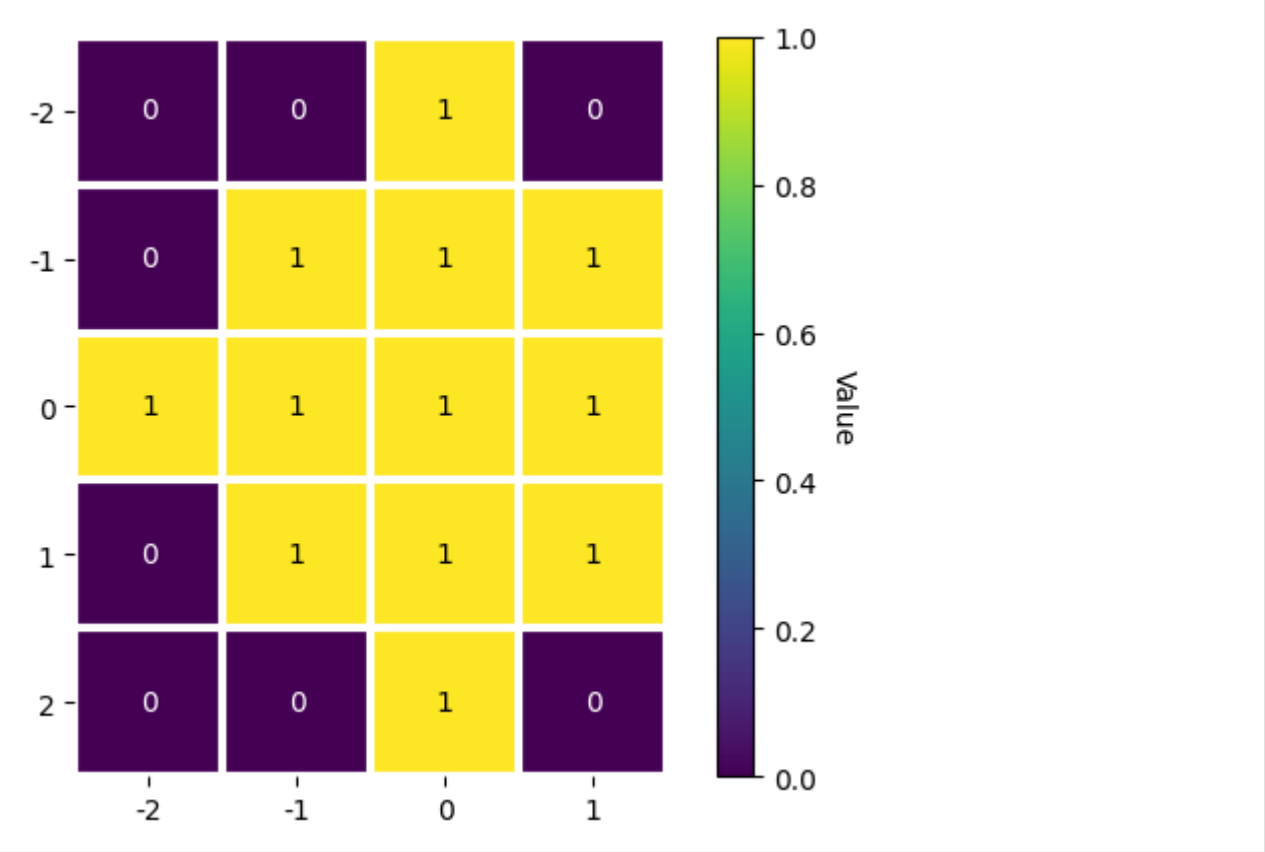



Any 1D or 2D window with desired coefficients can be used. This custom window can be passed to the `window` parameter in *average_neighbour_patterns()* or *Window* as a `numpy.ndarray` or a `dask.array.Array`. Additionally, any window in scipy.signal.windows.get_window() passed as a string via `window` with the necessary parameters as keyword arguments (like `std=1` for `window="gaussian"`) can be used. To demonstrate the creation and use of an asymmetrical circular window (and the use of *make_circular()*, although we could create a circular window directly by calling `window="circular"` upon window initialization)

[10]:
```
w = kp.filters.Window(window="rectangular", shape=(5, 4))
w
```

[10]:
```
Window (5, 4) rectangular
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[11]:
```
w.make_circular()
w.plot()
```




But this `(5, 4)` averaging window cannot be used with our `(3, 3)` navigation shape signal.

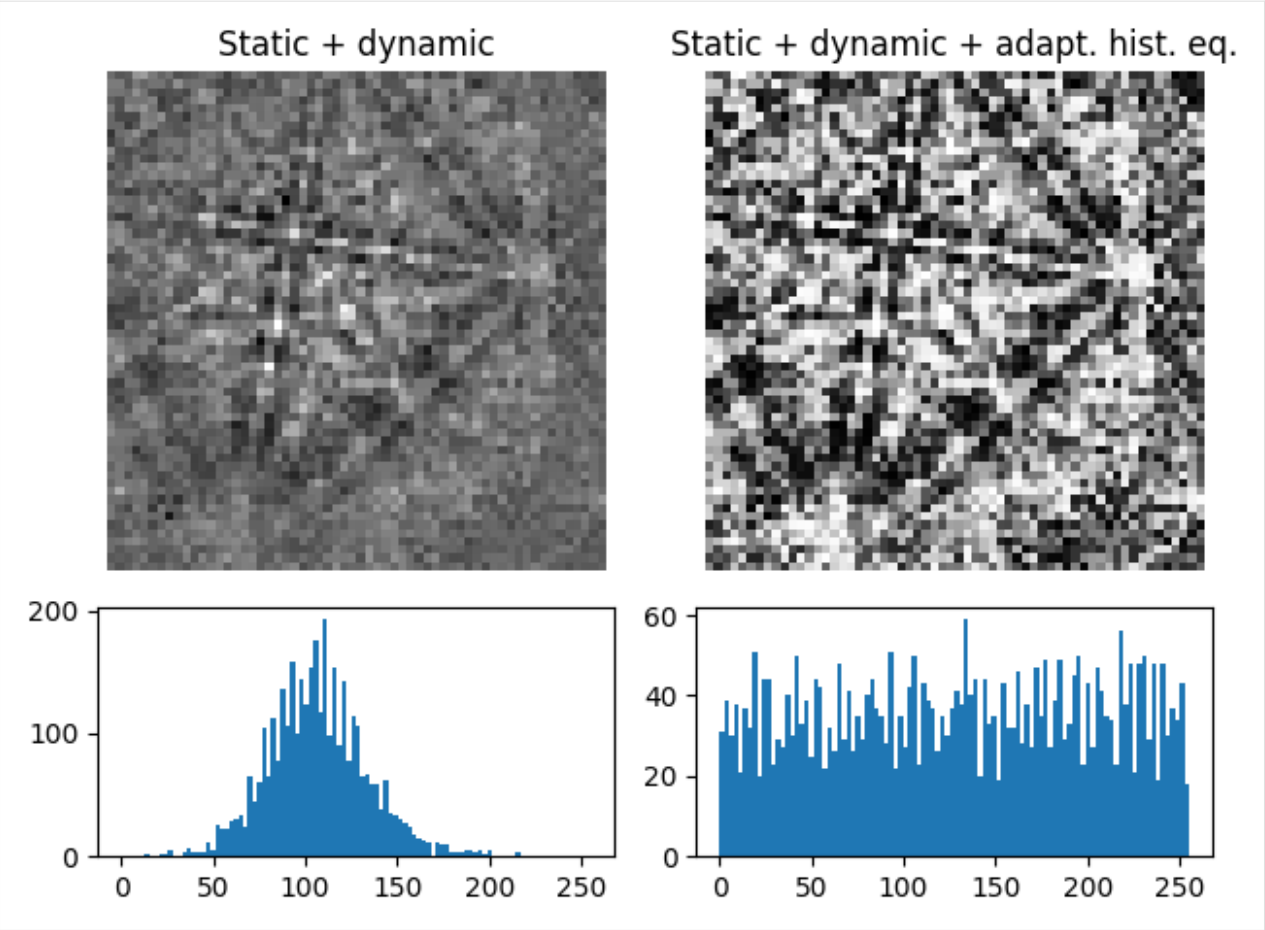
**Note**

Neighbour pattern averaging increases the virtual interaction volume of the electron beam with the sample, leading to a potential loss in spatial resolution. Averaging may in some cases, like on grain boundaries, mix two or more different diffraction patterns, which might be unwanted. See [Wright *et al.*, 2015] for a discussion of this concern.

### Adaptive histogram equalization

Enhancing the pattern contrast with adaptive histogram equalization has been found useful when comparing patterns for dictionary indexing [Marquardt *et al.*, 2017]. With *adaptive_histogram_equalization()*, the intensities in the pattern histogram are spread to cover the available range, e.g. [0, 255] for patterns of `uint8` data type

[12]:
```
s6 = s3.adaptive_histogram_equalization(kernel_size=(15, 15), inplace=False)

plot_pattern_processing(
    [s3.inav[0, 0].data, s6.inav[0, 0].data],
    ["Static + dynamic", "Static + dynamic + adapt. hist. eq."],
)
```




The `kernel_size` parameter determines the size of the contextual regions. See e.g. Fig. 5 in [Jackson *et al.*, 2019], also available via EMsoft's GitHub repository wiki, for the effect of varying `kernel_size`.

### Filtering in the frequency domain

Filtering of patterns in the frequency domain can be done with *fft_filter()*. This method takes a spatial kernel defined in the spatial domain, or a transfer function defined in the frequency domain, in the `transfer_function` argument as a `numpy.ndarray` or a *Window*. Which domain the transfer function is defined in must be passed to the `function_domain` argument. Whether to shift zero-frequency components to the center of the FFT can also be controlled via `shift`, but note that this is only used when `function_domain="frequency"`.

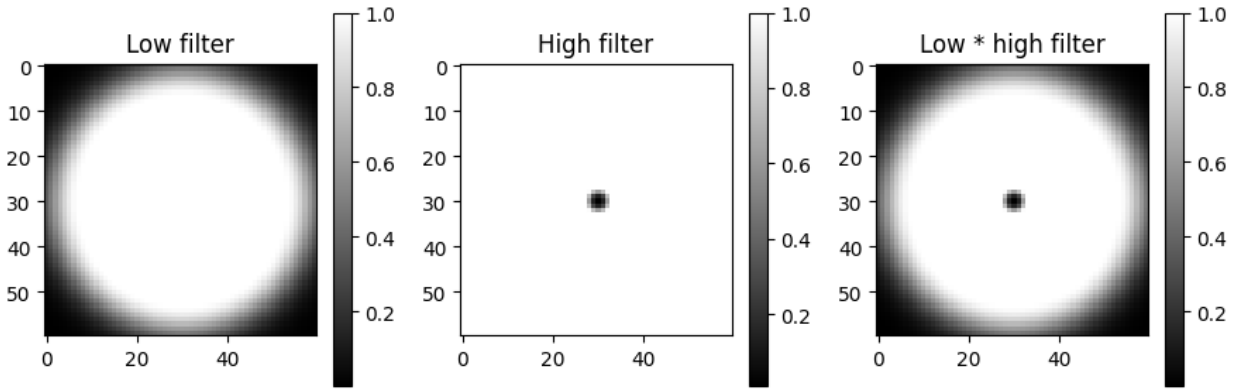
Popular uses of filtering of EBSD patterns in the frequency domain include removing large scale variations across the detector with a Gaussian high pass filter, or removing high frequency noise with a Gaussian low pass filter. These particular functions are readily available via `Window`

[13]:
```
pattern_shape = s.axes_manager.signal_shape[::-1]
w_low = kp.filters.Window(
    window="lowpass", cutoff=23, cutoff_width=10, shape=pattern_shape
)
w_high = kp.filters.Window(
    window="highpass", cutoff=3, cutoff_width=2, shape=pattern_shape
)
```

(continues on next page)

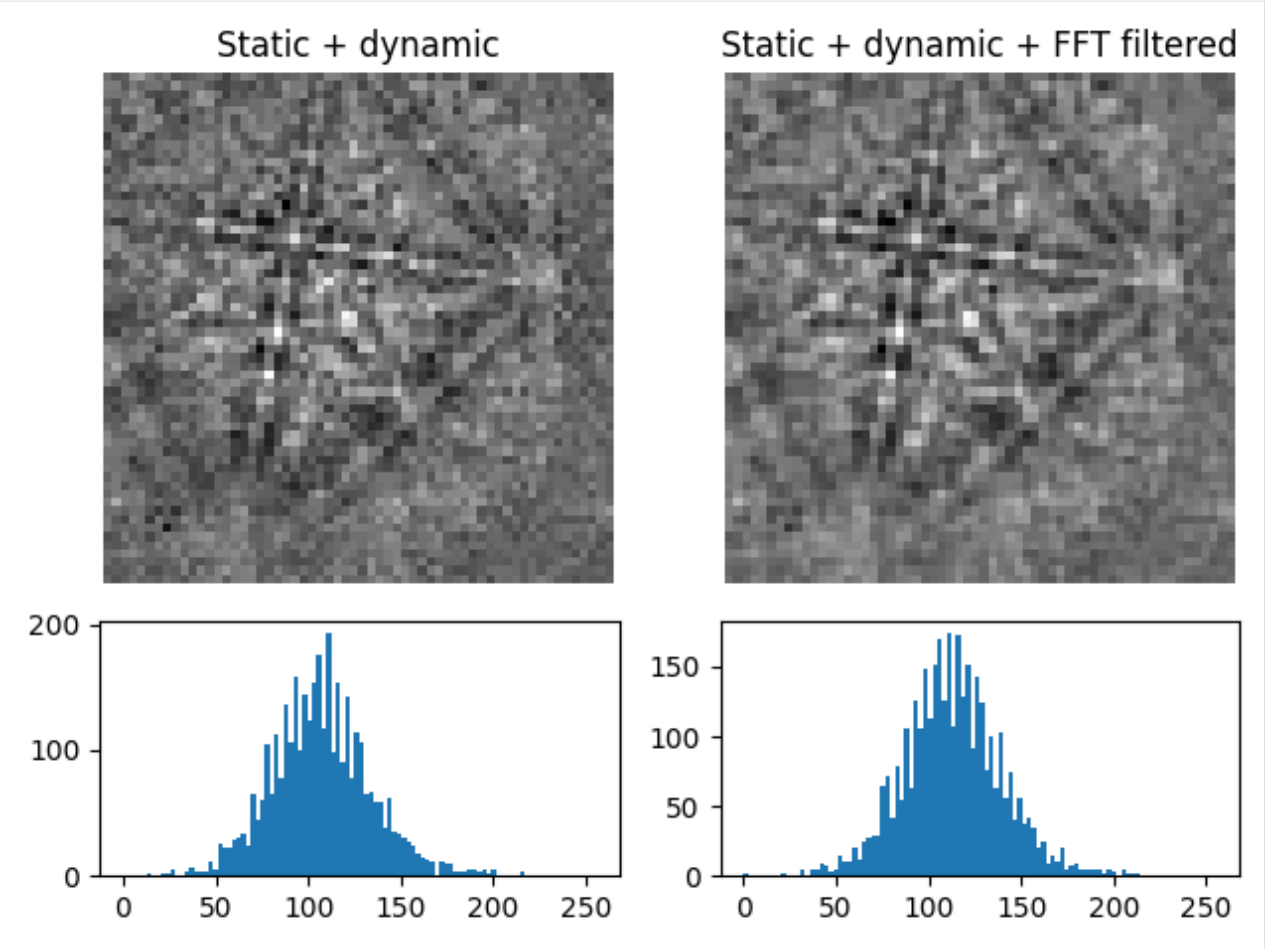
(continued from previous page)

```
fig, axes = plt.subplots(figsize=(9, 3), ncols=3)
for ax, data, title in zip(
    axes, [w_low, w_high, w_low * w_high], ["Low", "High", "Low * high"]
):
    im = ax.imshow(data, cmap="gray")
    ax.set_title(title + " filter")
    fig.colorbar(im, ax=ax)
fig.tight_layout()
```




Then, to multiply the FFT of each pattern with this transfer function, and subsequently computing the inverse FFT (IFFT), we use `fft_filter()`, and remember to shift the zero-frequency components to the centre of the FFT

[14]:
```
s7 = s3.fft_filter(
    transfer_function=w_low * w_high,
    function_domain="frequency",
    shift=True,
    inplace=False,
)

plot_pattern_processing(
    [s3.inav[0, 0].data, s7.inav[0, 0].data],
    ["Static + dynamic", "Static + dynamic + FFT filtered"],
)
```




Note that filtering with a spatial kernel in the frequency domain, after creating the kernel's transfer function via FFT, and computing the inverse FFT (IFFT), is, in this case, the same as spatially correlating the kernel with the pattern.

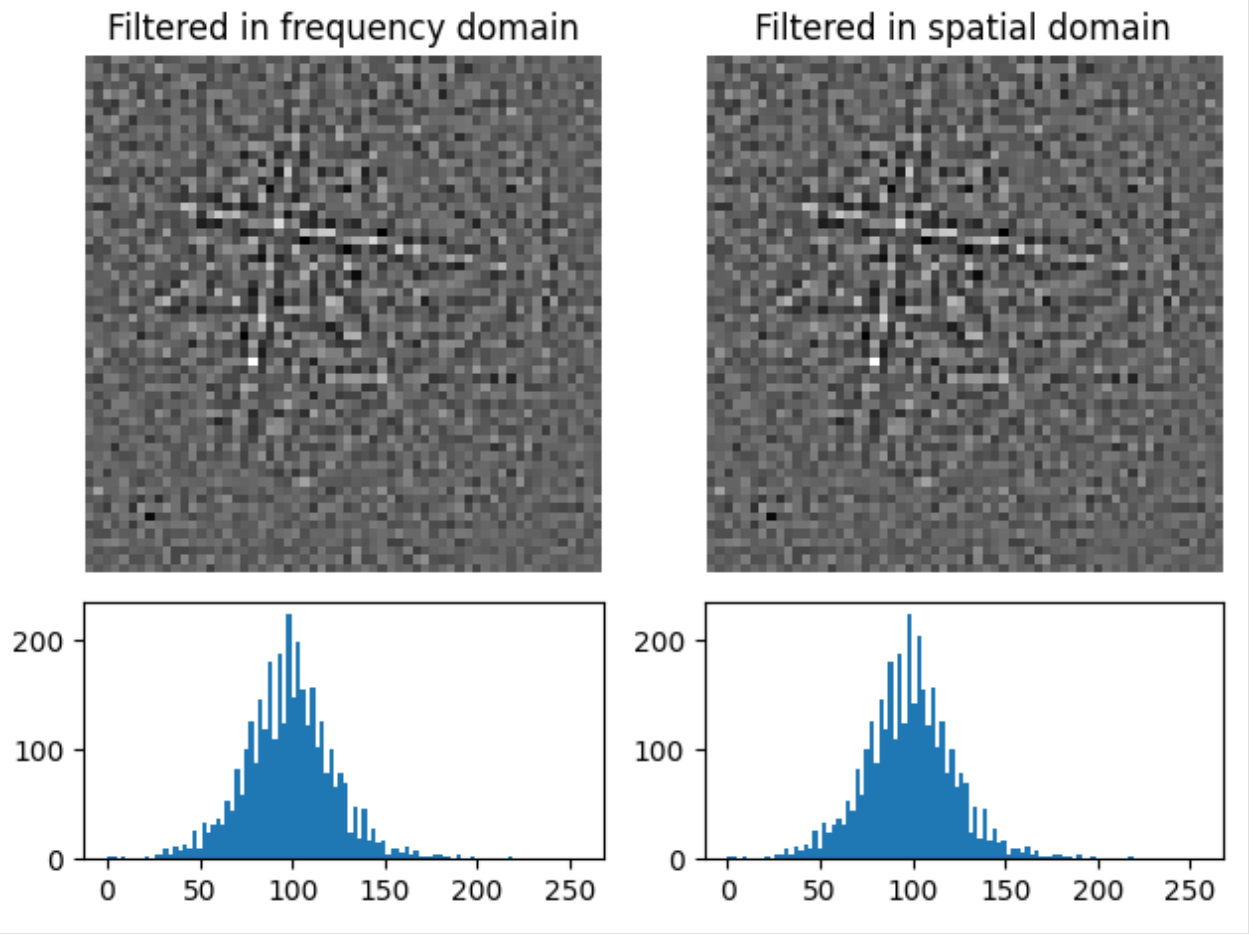
Let's demonstrate this by attempting to sharpen a pattern with a Laplacian kernel in both the spatial and frequency domains and comparing the results (this is a purely illustrative example, and perhaps not that practically useful)

[15]:
```
from scipy.ndimage import correlate


# fmt: off
w_laplacian = np.array([
    [-1, -1, -1],
    [-1,  8, -1],
    [-1, -1, -1]
])
# fmt: on
s8 = s3.fft_filter(
    transfer_function=w_laplacian, function_domain="spatial", inplace=False
)


p_filt = correlate(
    s3.inav[0, 0].data.astype(np.float32), weights=w_laplacian
```

(continues on next page)

(continued from previous page)

```
)
p_filt_resc = kp.pattern.rescale_intensity(p_filt, dtype_out=np.uint8)

plot_pattern_processing(
    [s8.inav[0, 0].data, p_filt_resc],
    ["Filtered in frequency domain", "Filtered in spatial domain"],
)
```




[16]:
```
np.sum(s8.inav[0, 0].data - p_filt_resc)  # Which is zero
```

[16]:
```
1275
```

Note also that `fft_filter()` performs the filtering on the patterns with data type `np.float32`, and therefore have to rescale back to the pattern's original data type if necessary.

### Rescale intensity

Vendors usually write patterns to file with 8 (`uint8`) or 16 (`uint16`) bit integer depth, holding $[0, 2^8]$ or $[0, 2^{16}]$ gray levels, respectively. To avoid losing intensity information when processing, we often change data types to e.g. 32 bit floating point (`float32`). However, only changing the data type with change_dtype() does not rescale pattern intensities, leading to patterns not using the full available data type range

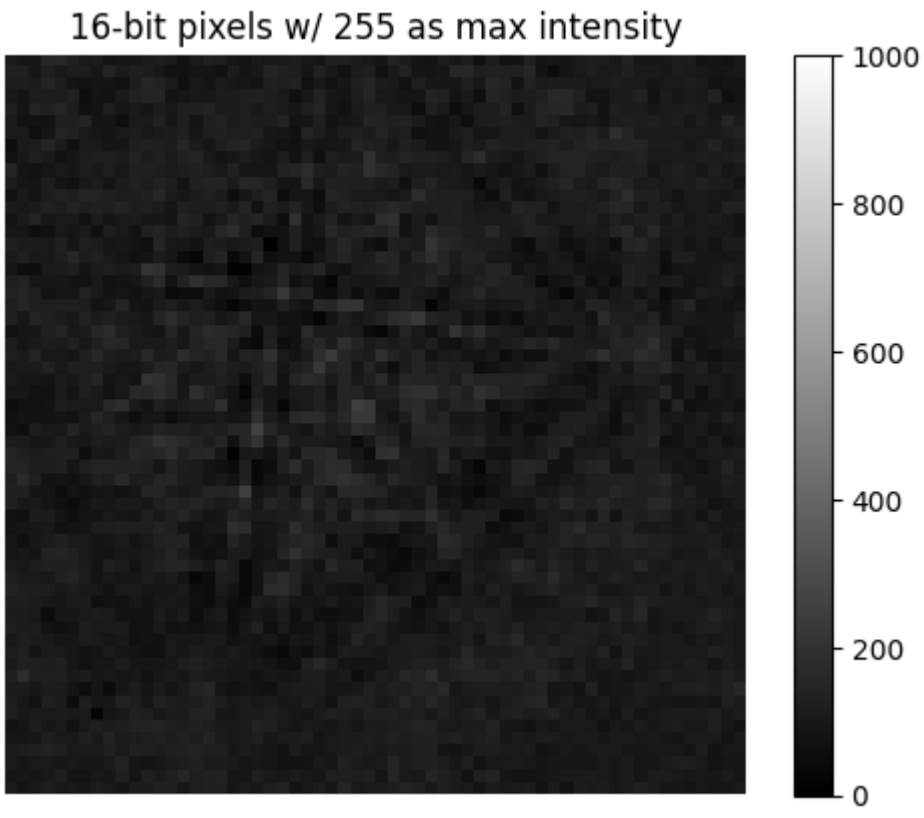
[17]:
```
s9 = s3.deepcopy()
print(s9.data.dtype, s9.data.max())
```

```
uint8 255
```

[18]:
```
s9.change_dtype(np.uint16)
print(s9.data.dtype, s9.data.max())
```

```
uint16 255
```

[19]:
```
plt.figure()
plt.imshow(s9.inav[0, 0].data, vmax=1000, cmap="gray")
plt.title("16-bit pixels w/ 255 as max intensity")
plt.axis("off")
_ = plt.colorbar()
```

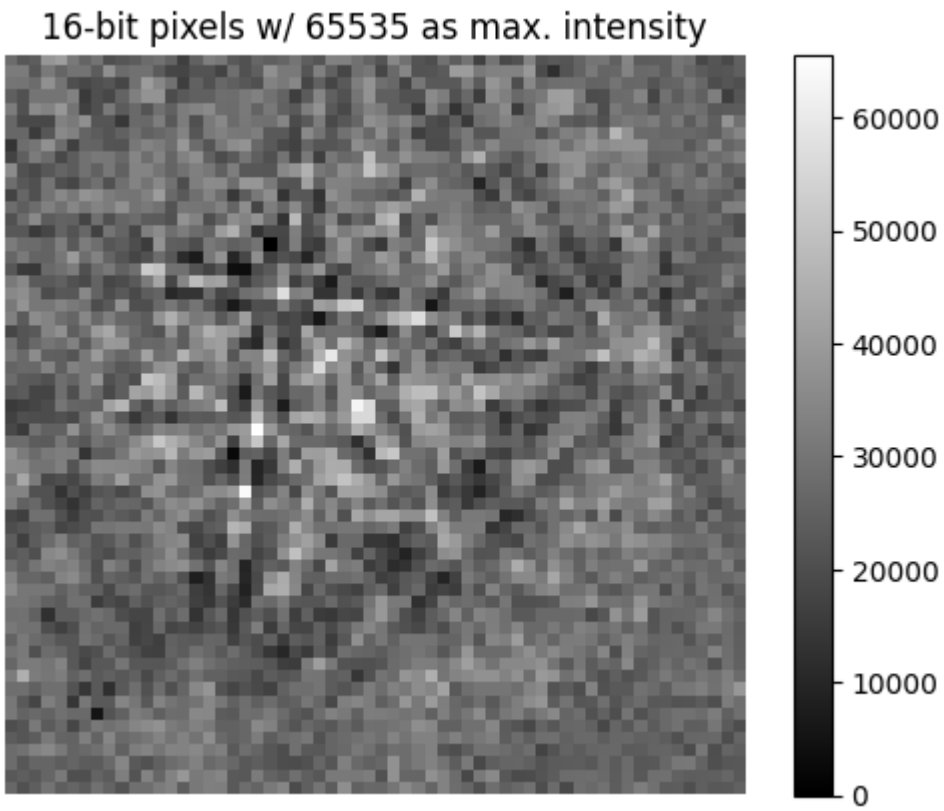



In these cases it is convenient to rescale intensities to a desired data type range, either keeping relative intensities between patterns in a scan or not. We can do this for all patterns in an EBSD signal with *kikuchipy.signals.EBSD.rescale_intensity()*

[20]:
```
s9.rescale_intensity(relative=True)
print(s9.data.dtype, s9.data.max())
```

```
uint16 65535
```

[21]:
```
plt.figure()
plt.imshow(s9.inav[0, 0].data, cmap="gray")
plt.title("16-bit pixels w/ 65535 as max. intensity")
plt.axis("off")
_ = plt.colorbar()
```




Or, we can do it for a single pattern (`numpy.ndarray`) with *kikuchipy.pattern.rescale_intensity()*

[22]:
```
p = s3.inav[0, 0].data
print(p.min(), p.max())
```

```
0 255
```

[23]:
```
p2 = kp.pattern.rescale_intensity(p, dtype_out=np.uint16)
print(p2.min(), p2.max())
```

```
0 65535
```

We can also stretch the pattern contrast by removing intensities outside a range passed to `in_range` or at certain percentiles by passing percentages to `percentiles`

[24]:
```
print(s3.inav[0, 0].data.min(), s3.inav[0, 0].data.max())
```

```
0 255
```

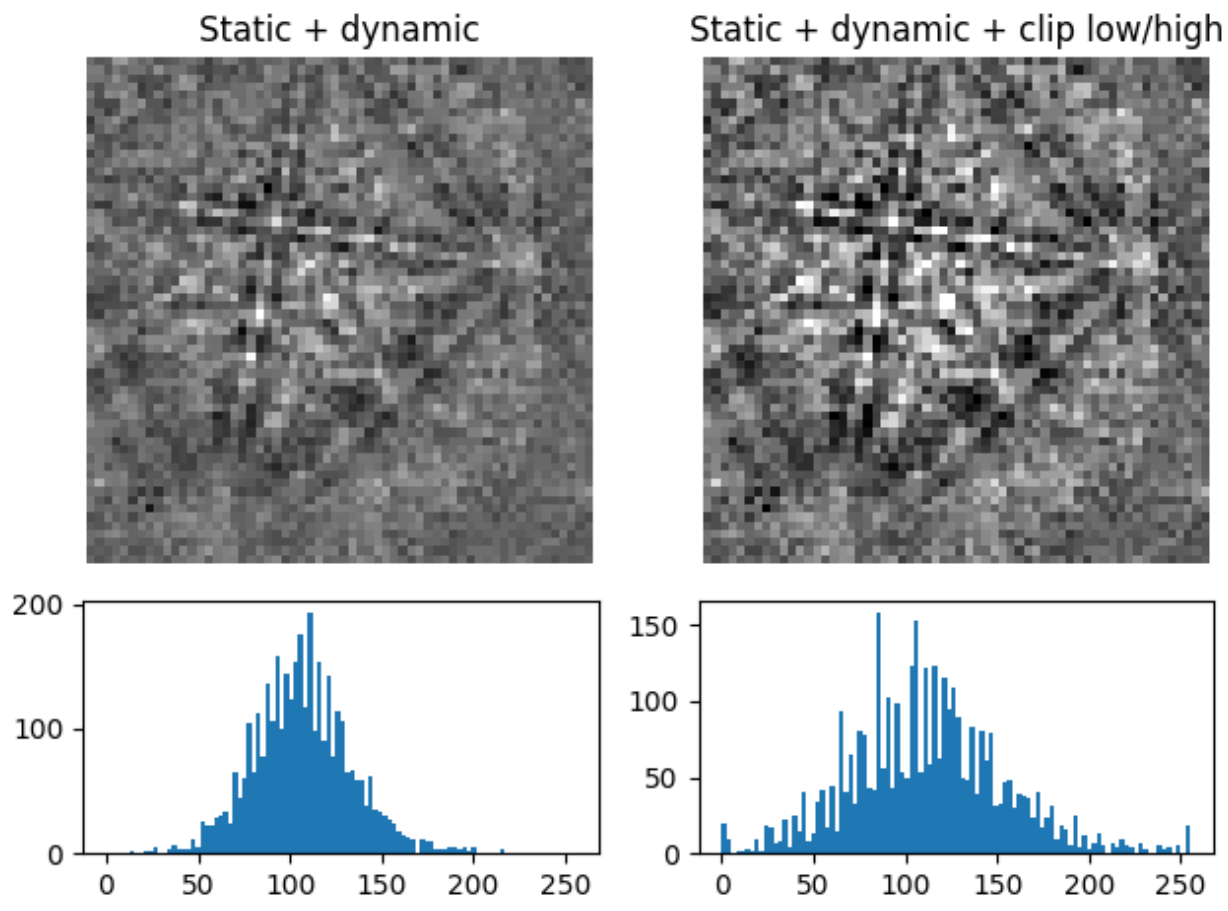
[25]:
```
s10 = s3.inav[0, 0].rescale_intensity(out_range=(10, 245), inplace=False)
print(s10.data.min(), s10.data.max())
```

```
10 245
```

[26]:
```
s10.rescale_intensity(percentiles=(0.5, 99.5))
print(s10.data.min(), s3.data.max())
```

```
0 255
```

[27]:
```
plot_pattern_processing(
    [s3.inav[0, 0].data, s10.data],
    ["Static + dynamic", "Static + dynamic + clip low/high"],
)
```




This can reduce the influence of outliers with exceptionally high or low intensities, like hot or dead pixels.

### Normalize intensity

It can be useful to normalize pattern intensities to a mean value of $\mu = 0.0$ and a standard deviation of e.g. $\sigma = 1.0$ when e.g. comparing patterns or calculating the *image quality*. Patterns can be normalized with *normalize_intensity()*

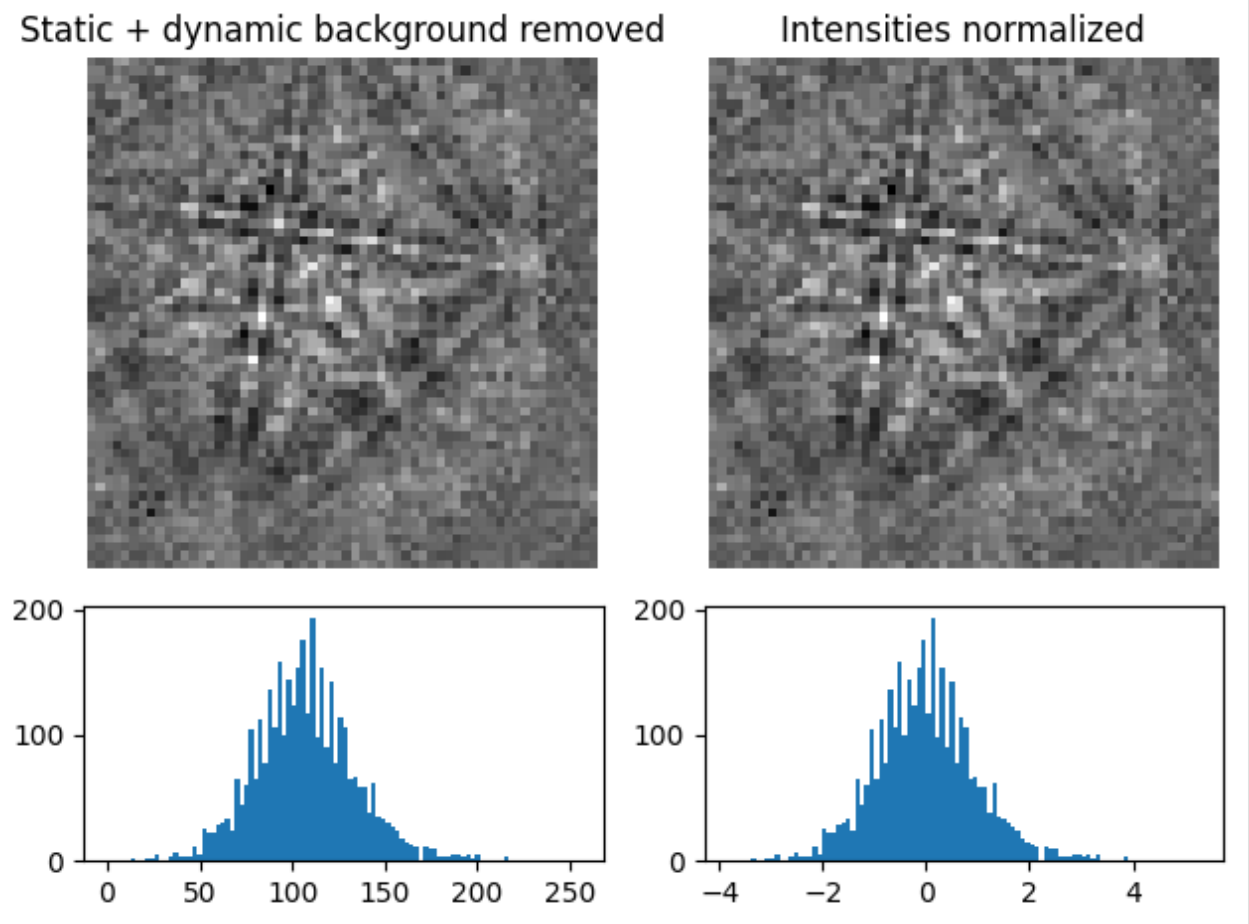
[28]:
```
np.mean(s3.inav[0, 0].data)
```

[28]:
```
107.2775
```

[29]:
```
s11 = s3.inav[0, 0].normalize_intensity(num_std=1, dtype_out=np.float32, inplace=False)
np.mean(s11.data)
```

[29]:
```
0.0
```

[30]:
```
plot_pattern_processing(
    [s3.inav[0, 0].data, s11.data],
    ["Static + dynamic background removed", "Intensities normalized"],
)
```




**Live notebook**

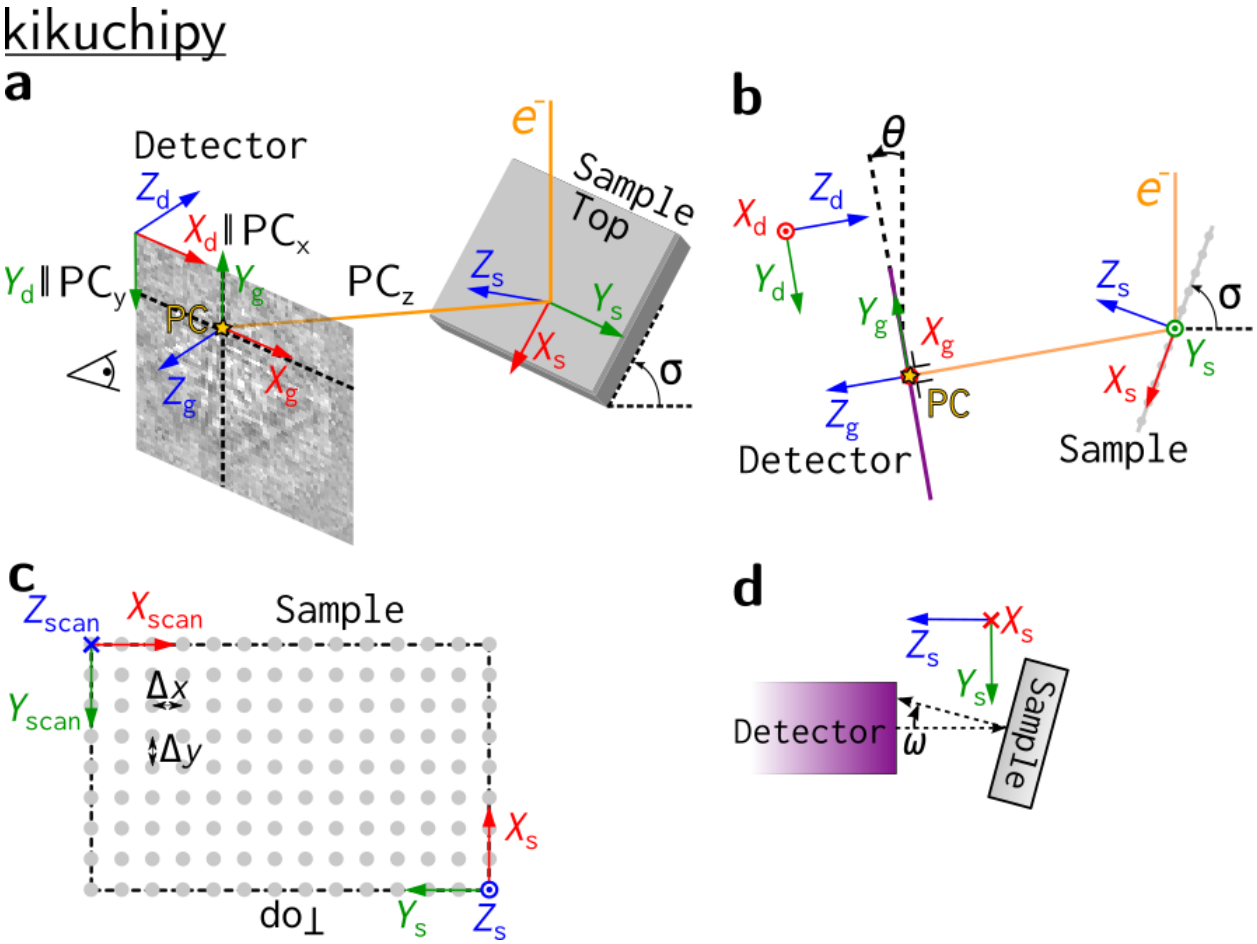
You can run this notebook in a live session launch binder or view it on Github.

### Reference frames

In this tutorial, we cover the reference frames most important to EBSD, chosen conventions in kikuchipy, and how they relate to conventions in the other softwares by Bruker Nano, EDAX TSL, EMsoft, and Oxford Instruments. We also test conversions between the conventions by indexing simulated patterns using Hough indexing (HI) from PyEBSDIndex.

### Detector-sample geometry

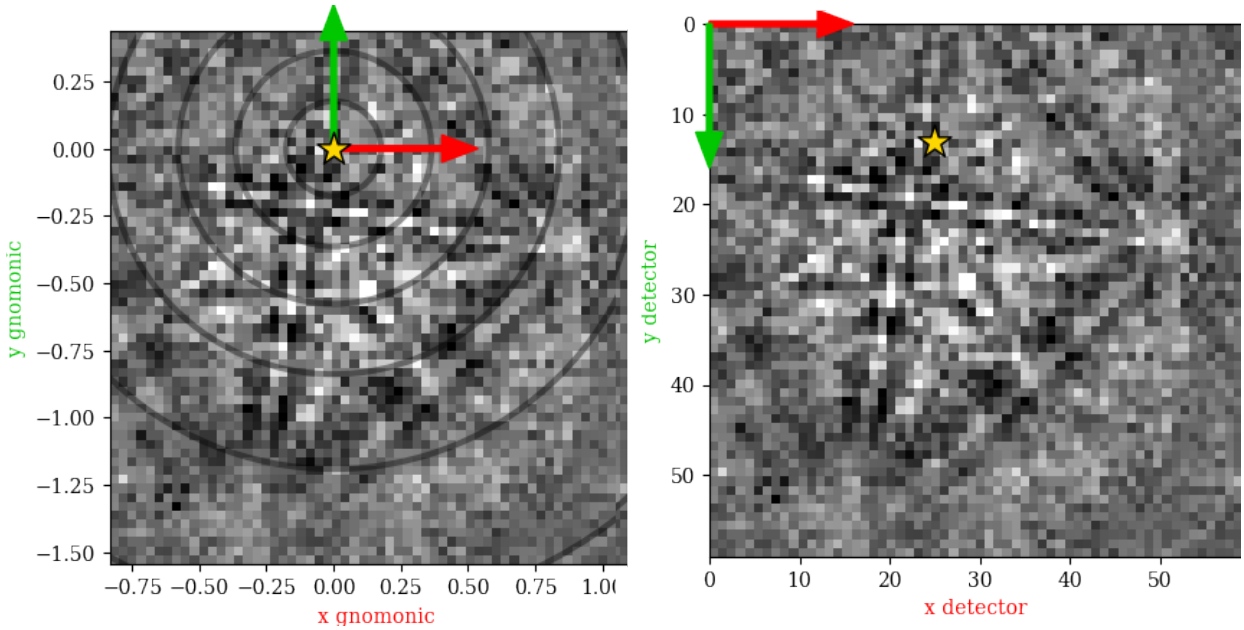
The figure below shows the *sample reference frame* and the detector reference frame used in kikuchipy, all of which are right handed. In short, the sample reference frame ($X_s$, $Y_s$, $Z_s$) is the one used by EDAX TSL, (RD, TD, ND), while the pattern center ($PC_x$, $PC_y$, $PC_z$) is the one used by Bruker Nano, ($PC_x$, $PC_y$, $DD$).




In **a**, the electron beam interacts with the sample in the source point. The shortest distance from this point to the detector is called the projection or pattern center (PC). A part of the Kikuchi sphere, resulting from the beam-sample interaction, is projected onto the flat detector in the gnomonic projection, constituting the EBSD pattern (EBSP). A gnomonic coordinate system $CS_g$, ($X_g$, $Y_g$, $Z_g$) with (0, 0, 0) in the PC is defined for the detector plane. We also define a detector coordinate system $CS_d$, ($X_d$, $Y_d$, $Z_d$) for the detector plane with (0, 0, 0) in the upper left corner. The projection center coordinates ($PC_x$, $PC_y$, $PC_z$) are defined in this detector coordinate system:

- $PC_x$ is measured from the left border of the detector in fractions of detector width.
- $PC_y$ is measured from the top border of the detector in fractions of detector height.
- $PC_z$ is the distance from the detector scintillator to the sample divided by pattern height.

Orientations are defined in the Bunge convention with respect to the sample coordinate system $CS_s$, ($X_s$, $Y_s$, $Z_s$). The detector and sample viewed along the microscope $X$ axis are shown in **b**, with the three coordinate systems and the PC also shown. The scanned map is shown in **c**. Note the orientation of $CS_s$ and the sample "Top": the map is scanned from the bottom of the sample and upwards. Three tilt angles are defined: the sample tilt $\sigma$ shown in **a** and **b**; the detector tilt $\theta$ shown in **b**; the azimuthal angle $\omega$ which is defined as the sample tilt angle around the $X_s$ axis, shown in a top view of the detector and sample along the microscope $Z$ axis in **d**.




The above figure shows the EBSP in the *sample reference frame figure* **a** as viewed from behind the screen towards the sample in (left) the gnomonic coordinate system with its origin (0, 0) in the PC, and in (right) the detector coodinate system with (0, 0) in the upper left pixel. The circles indicate the angular distance from the PC in steps of 10°.

### The EBSD detector

All relevant parameters for the detector-sample geometry are stored in an *EBSDDetector* instance. Let's first import necessary libraries and a small Nickel EBSD test data set

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

import kikuchipy as kp
from orix.quaternion import Orientation, Rotation
from orix.vector import Vector3d


plt.rcParams["figure.figsize"] = (15, 5)
```

[2]:
```
# Use kp.load("data.h5") to load your own data
s = kp.data.nickel_ebsd_small()
s
```

[2]:
```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
```

Then we can define a detector with the same parameters as the one used to acquire the small nickel data set

[3]:
```
det = kp.detectors.EBSDDetector(
    shape=s.axes_manager.signal_shape[::-1],
    pc=[0.4221, 0.2179, 0.4954],
    px_size=70,  # Microns
    binning=8,
    tilt=0,
    sample_tilt=70,
)
det
```

[3]:
```
EBSDDetector (60, 60), px_size 70 um, binning 8, tilt 0, azimuthal 0, pc (0.422, 0.218,␣
↪0.495)
```

[4]:
```
det.pc_tsl()
```

[4]:
```
array([[0.4221, 0.7821, 0.4954]])
```

Above, the PC was passed in the Bruker convention. Passing the PC in the EDAX TSL, Oxford, or EMsoft convention is also supported. The definitions of the conventions are given in the *EBSDDetector* API reference, together with the conversion from PC coordinates in the EDAX TSL, Oxford, or EMsoft conventions to PC coordinates in the Bruker convention.

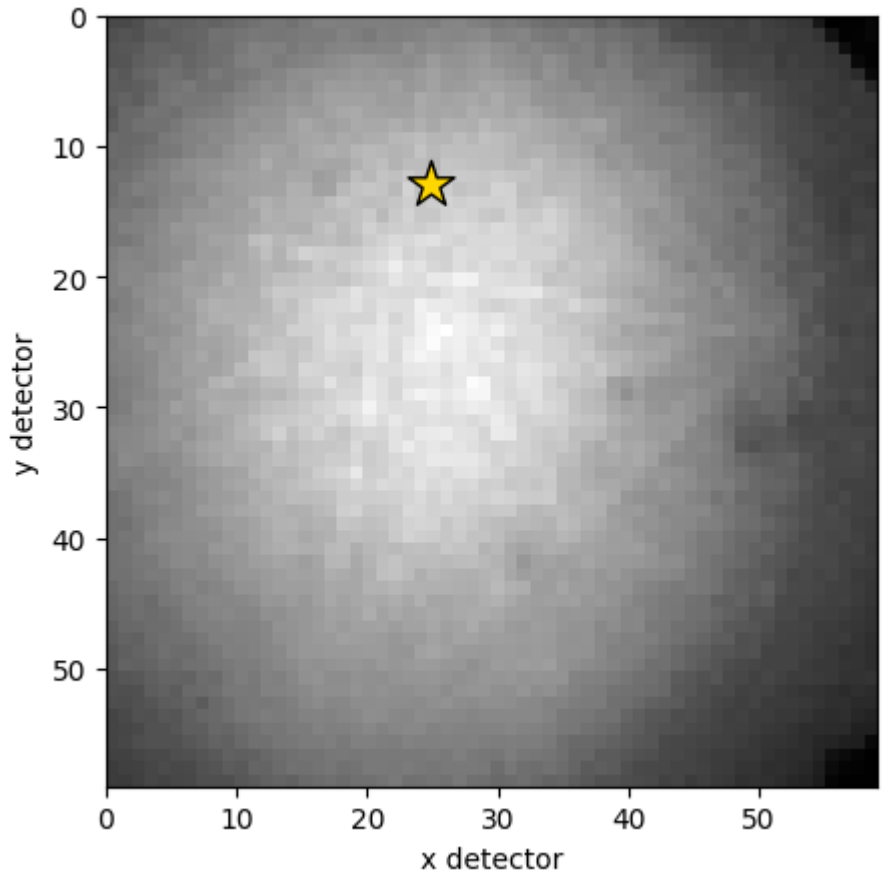
The PC coordinates in the EDAX TSL, Oxford, or EMsoft conventions can be retreived via *EBSDDetector.pc_tsl()*, *EBSDDetector.pc_oxford()*, and *EBSDDetector.pc_emsoft()*, respectively. The latter requires the unbinned detector pixel size in microns and the detector binning to be given upon initialization.

[5]:
```
det.pc_emsoft()
```

[5]:
```
array([[   37.392,   135.408, 16645.44 ]])
```

The detector can be plotted to show whether the average PC is placed as expected using *EBSDDetector.plot()* (see its docstring for a complete explanation of its parameters)

[6]:
```
det.plot(pattern=s.inav[0, 0].data)
```




This will produce a figure similar to the right panel in the detector coordinates figure above, without the arrows and colored labels.

Multiple PCs with a 1D or 2D navigation shape can be passed to the `pc` parameter upon initialization, or they can be set directly. This gives the detector a navigation shape (not to be confused with the detector shape) and a navigation dimension (maximum of two)

[7]:
```
det.pc = np.ones([3, 4, 3]) * det.pc
det.navigation_shape
```

[7]:
```
(3, 4)
```

[8]:
```
det.navigation_dimension
```

[8]:
```
2
```

[9]:
```
det.navigation_size
```

[9]:
```
12
```

[10]:
```
det.pc = det.pc[0, 0]
det.navigation_shape
```

[10]:
```
(1,)
```

**Note**

The offset and scale of HyperSpy's `axes_manager` is fixed for a signal. This restricts us from letting the PC vary with scan position if we want to calibrate the EBSD detector via the `axes_manager`. The need for a varying PC was the main motivation for the `EBSDDetector` class.

The left panel in the detector coordinates figure above shows the detector plotted in the gnomonic projection using *EBSDDetector.plot()*. The 2D gnomonic coordinates $(x_g, y_g)$ in $CS_g$ are defined in $CS_d$ are

$$x_g = \frac{x_d}{z_d}, \qquad y_g = \frac{y_d}{z_d}.$$

The detector bounds and pixel scale in this projection, per navigation point, are stored with the detector

[11]:
```
det.bounds
```
[11]:
```
array([ 0, 59,  0, 59])
```

[12]:
```
det.gnomonic_bounds
```
[12]:
```
array([[-0.85203876,  1.1665321 , -1.57872426,  0.43984659]])
```

[13]:
```
det.x_range
```
[13]:
```
array([[-0.85203876,  1.1665321 ]])
```

[14]:
```
det.r_max  # Greatest radial distance to PC
```
[14]:
```
array([[1.96294866]])
```

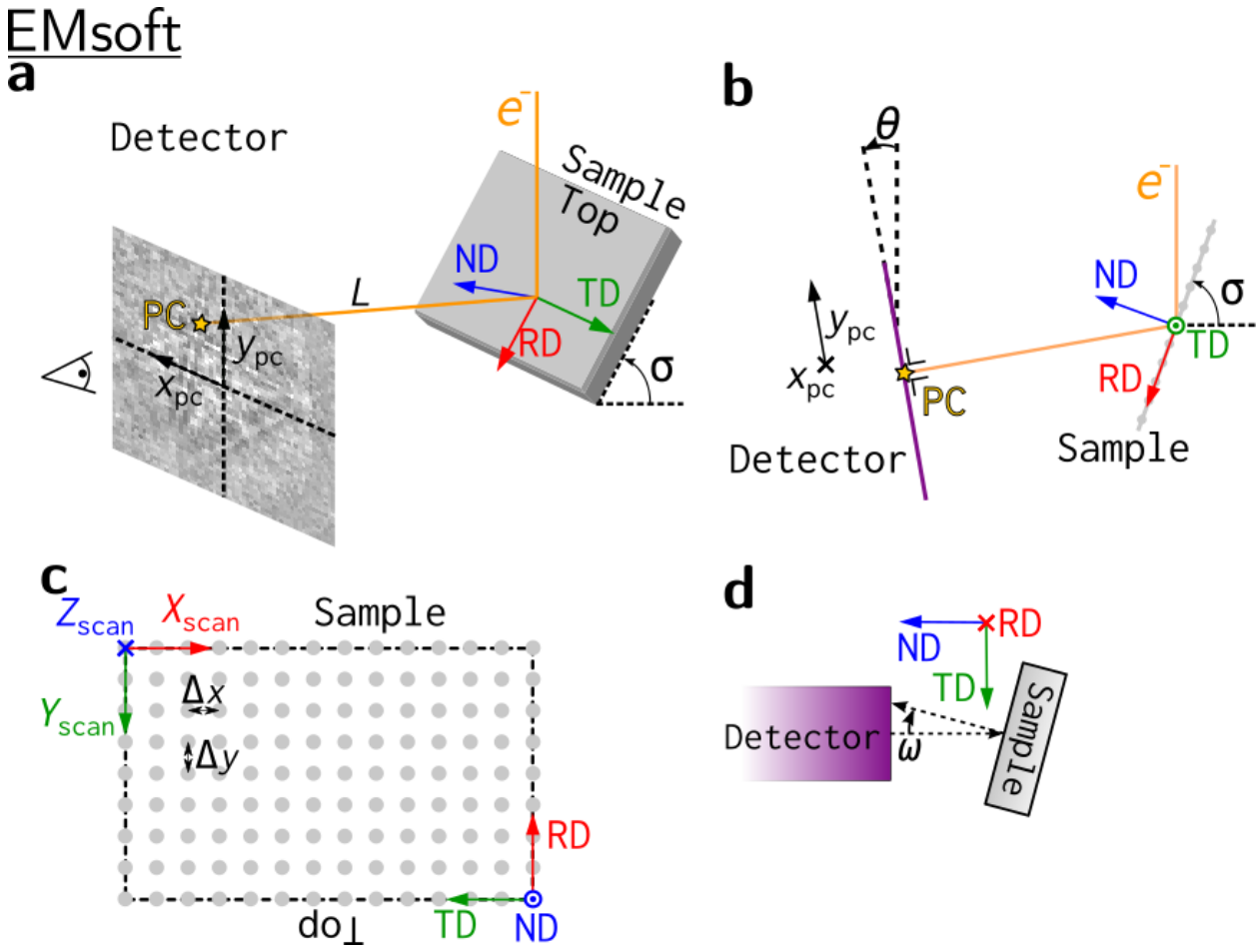
### Other software's reference frames

Other software use other reference frames. To aid in the conversion of orientations between softwares, the reference frames used in other softwares are also shown here. They represented to the best of the contributors understanding.

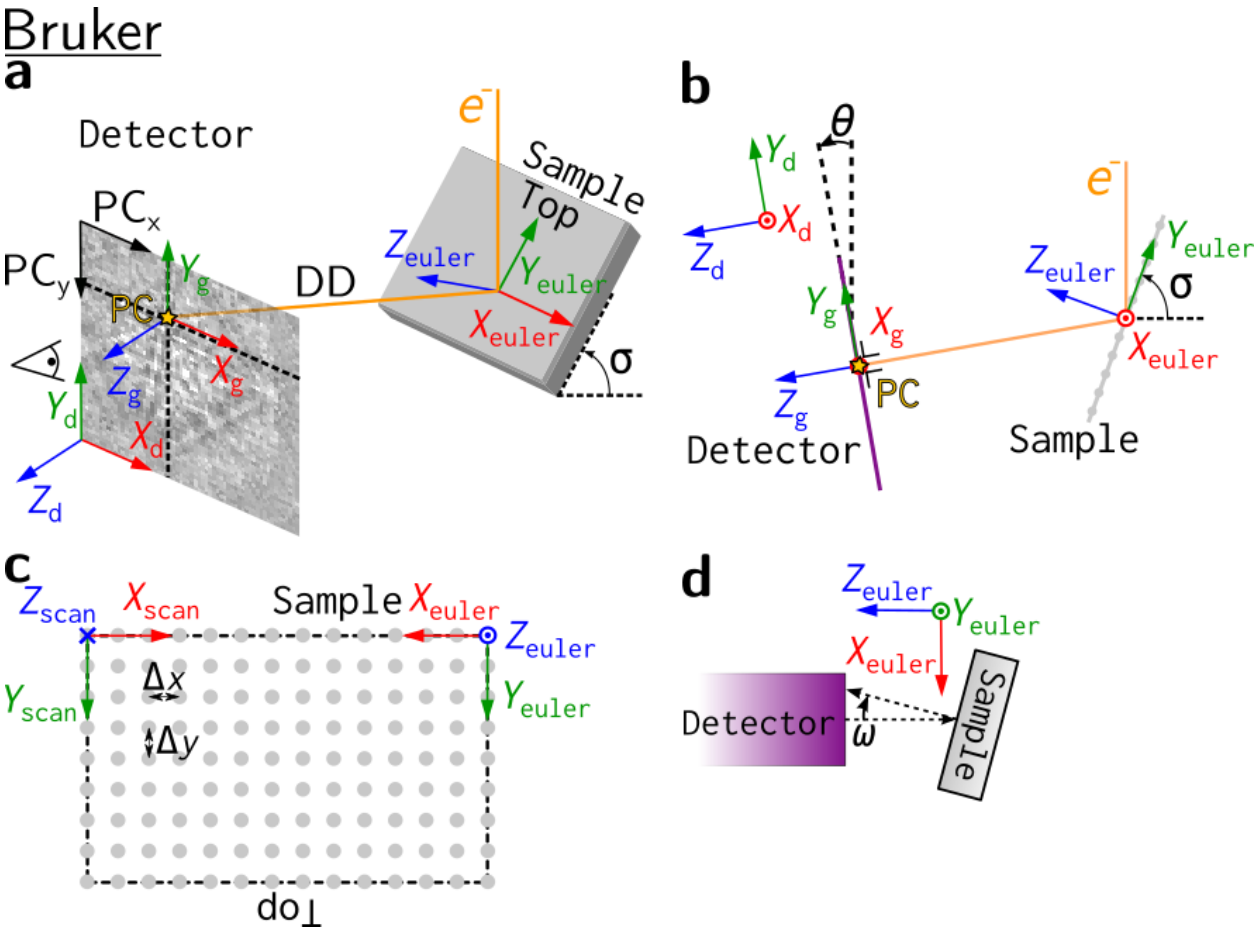
**Warning**

Reference frames used in other softwares given here are based on instruction manuals from the internet. Use with care, and double check whenever possible.

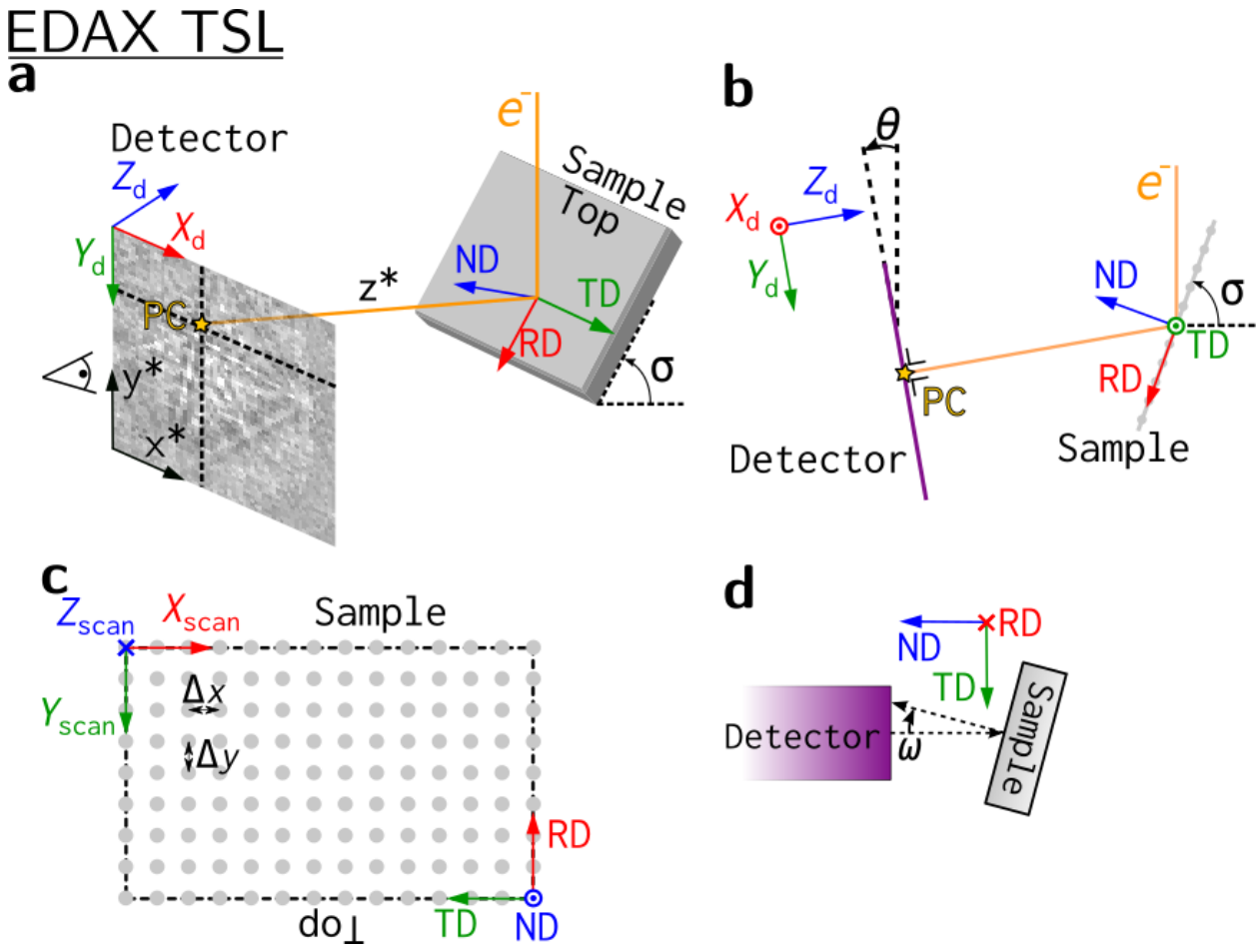
### EMsoft


EMsoft
a
Detector
e⁻
Sample Top
ND
TD
RD
L
PC
ypc
xpc
σ
b
θ
e⁻
ND
σ
TD
RD
ypc
xpc
PC
Detector
Sample
c
Zscan
Xscan
Sample
Yscan
Δx
Δy
RD
do⊥
TD
ND
d
RD
ND
TD
Detector
ω
Sample

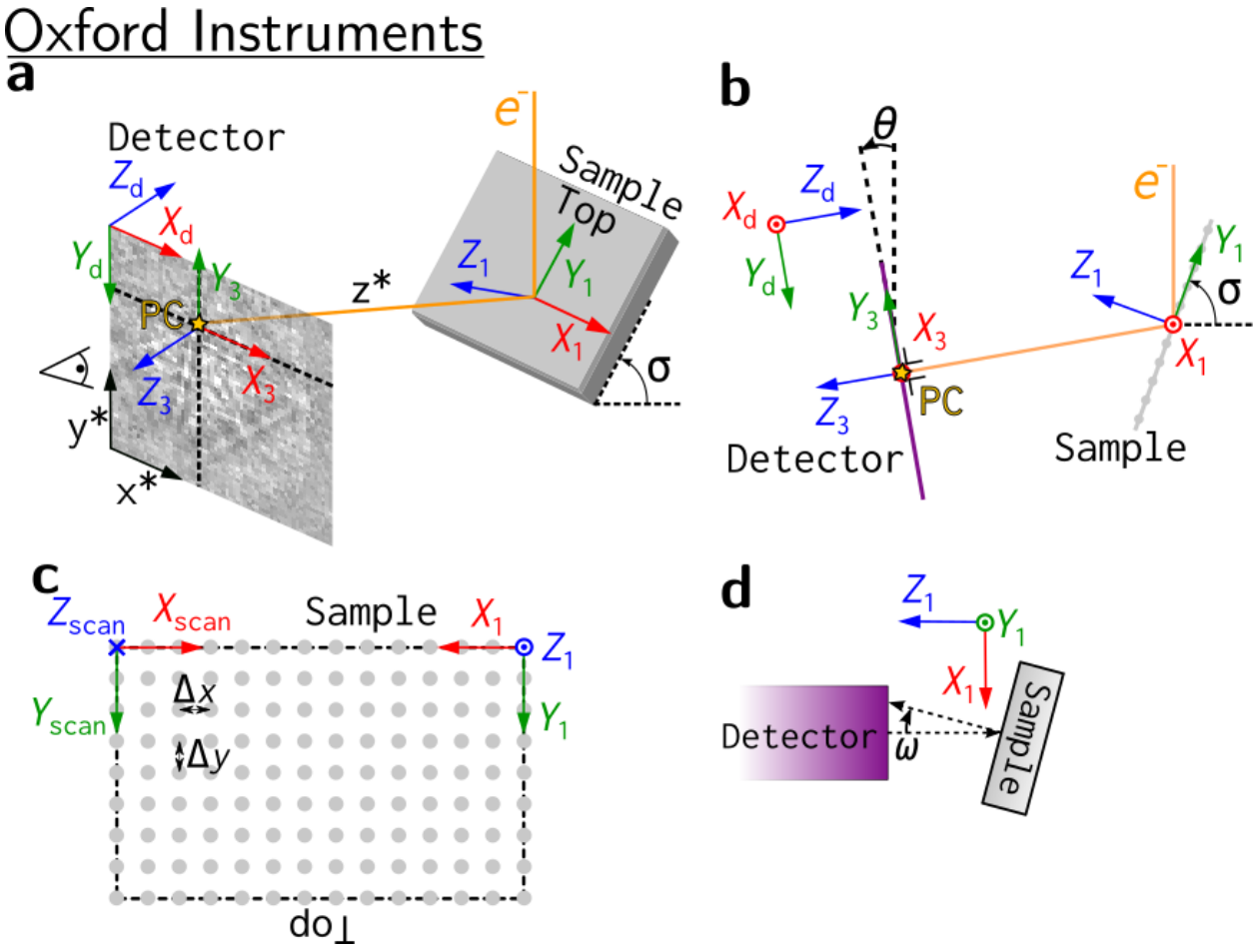
### Bruker


Bruker
a
Detector
PC$_x$
PC$_y$
PC
DD
$Y_g$
$Z_g$
$X_g$
$Y_d$
$X_d$
$Z_d$
$e^-$
Sample Top
$Z_{euler}$
$Y_{euler}$
$X_{euler}$
σ
b
$Y_d$
$X_d$
$Z_d$
θ
$Y_g$
$X_g$
$Z_g$
PC
Detector
$e^-$
$Z_{euler}$
$Y_{euler}$
σ
$X_{euler}$
Sample
c
$Z_{scan}$
$X_{scan}$
Sample
$X_{euler}$
$Z_{euler}$
$Y_{scan}$
Δx
Δy
$Y_{euler}$
do⊥
d
$Z_{euler}$
$Y_{euler}$
$X_{euler}$
Detector
ω
Sample

### EDAX TSL


EDAX TSL
a
Detector
$Z_d$
$X_d$
$Y_d$
PC
z*
y*
x*
$e^-$
Sample Top
ND
TD
RD
σ
b
θ
$Z_d$
$X_d$
$Y_d$
$e^-$
ND
σ
TD
RD
PC
Detector
Sample
c
$Z_{scan}$
$X_{scan}$
Sample
$Y_{scan}$
Δx
Δy
RD
$do_\perp$
TD
ND
d
RD
ND
TD
Detector
ω
Sample

### Oxford Instruments




### Test PC conventions with PyEBSDIndex

We can test the PC conventions using Hough indexing in PyEBSDIndex. We will use ten dynamically simulated nickel patterns with a fixed PC and random orientations. We check for consistency by passing the PC in all the conventions described above when indexing, and making sure that the indexed orientations are rotated so that they are defined with respect to the same sample reference frame (the one used in kikuchipy, EDAX TSL and EMsoft).

---

**Note**

PyEBSDIndex is an optional dependency of kikuchipy, and can be installed with both `pip` and `conda` (from `conda-forge`). To install PyEBSDIndex, see their installation instructions.

---

Load master pattern in the square Lambert projection

[15]:
```
mp = kp.data.nickel_ebsd_master_pattern_small(
    projection="lambert", hemisphere="upper"
)
```

Define a rectangular EBSD detector to project simulated patterns onto

[16]:
```
det2 = kp.detectors.EBSDDetector(
    (100, 120), pc=(0.4, 0.2, 0.5), sample_tilt=70
)
det2
```

[16]:
```
EBSDDetector (100, 120), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.4, 0.2, 0.5)
```

Create ten random orientations

[17]:
```
Gr = Rotation.random(10)
G = Orientation(Gr, mp.phase.point_group)
```

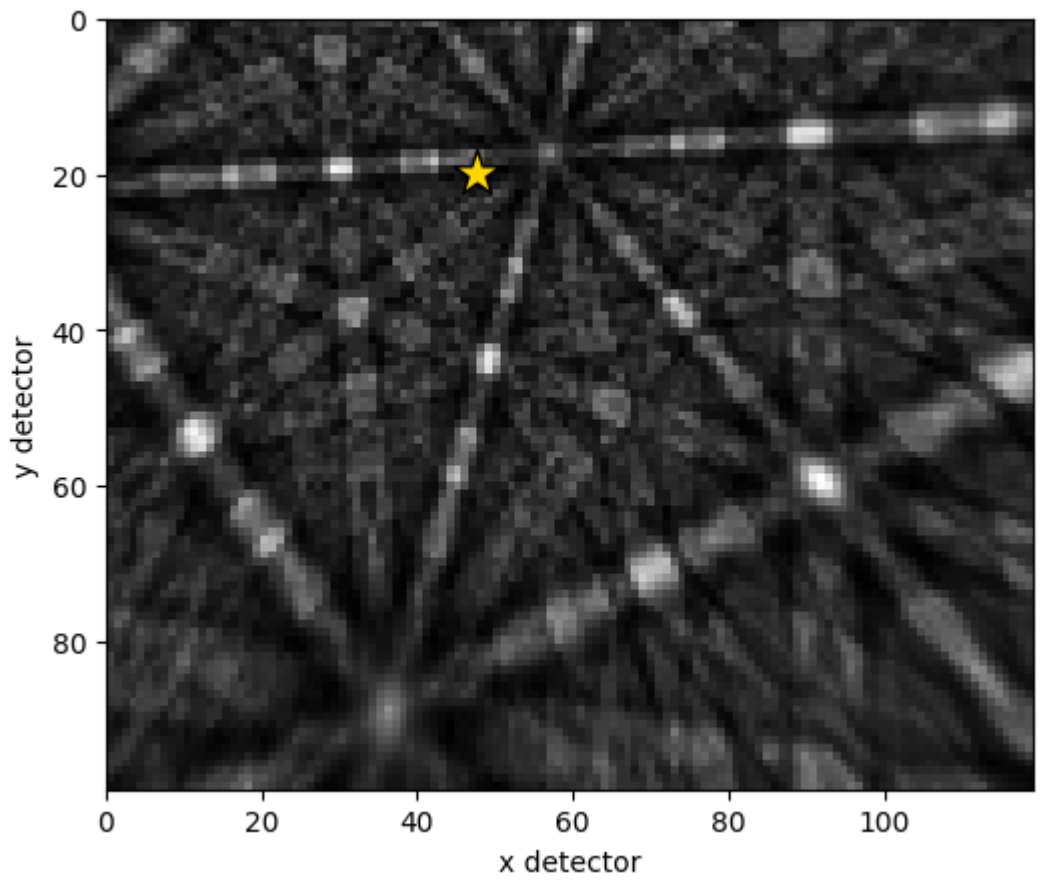
Project patterns onto detector

[18]:
```
s2 = mp.get_patterns(Gr, det2, energy=20, compute=True)
```

```
[########################################] | 100% Completed | 102.70 ms
```

Plot the first simulated pattern and the PC

[19]:
```
det2.plot(pattern=s2.inav[0].data)
```




Load necessary PyEBSDIndex modules for Hough indexing

[20]:
```
from pyebsdindex import ebsd_index, pcopt
```

For the various softwares, define the PCs and the transformations bringing the indexed orientations returned from PyEBSDIndex back to the sample reference frame used in kikuchipy, EDAX TSL, and EMsoft

[21]:
```
pcs = {
    "KIKUCHIPY": det2.pc,
    "BRUKER": det2.pc,
    "EDAX": det2.pc_tsl(),
    "OXFORD": det2.pc_oxford(),
    "EMSOFT": det2.pc_emsoft(),
}
Gr_sample = {
    "KIKUCHIPY": Rotation.identity(),
    "BRUKER": Rotation.from_axes_angles([0, 0, 1], -np.pi / 2),
    "EDAX": Rotation.identity(),
    "OXFORD": Rotation.from_axes_angles([0, 0, 1], -np.pi / 2),
    "EMSOFT": Rotation.identity(),
}

# Some wrangling to display a nice table
for softw, pc_i in pcs.items():
    print(
        f"{softw:<9} {pc_i[0, 0]:>6.3f} {pc_i[0, 1]:>6.3f} {pc_i[0, 2]:>6.3f}"
    )
```

```
KIKUCHIPY  0.400  0.200  0.500
BRUKER     0.400  0.200  0.500
EDAX       0.400  0.800  0.500
OXFORD     0.400  0.667  0.417
EMSOFT    12.000 30.000 50.000
```

Then we do the following in a loop:

1. Initialize a PyEBSDIndex indexer, specifying the vendor and vendor specific PC
2. Index the ten patterns
3. Apply vendor specific conversion to the returned orientations
4. Print misorientation angle between indexed orientations and ground truth orientations
5. Plot the indexed orientations and the ground truth in inverse pole figures (IPFs)

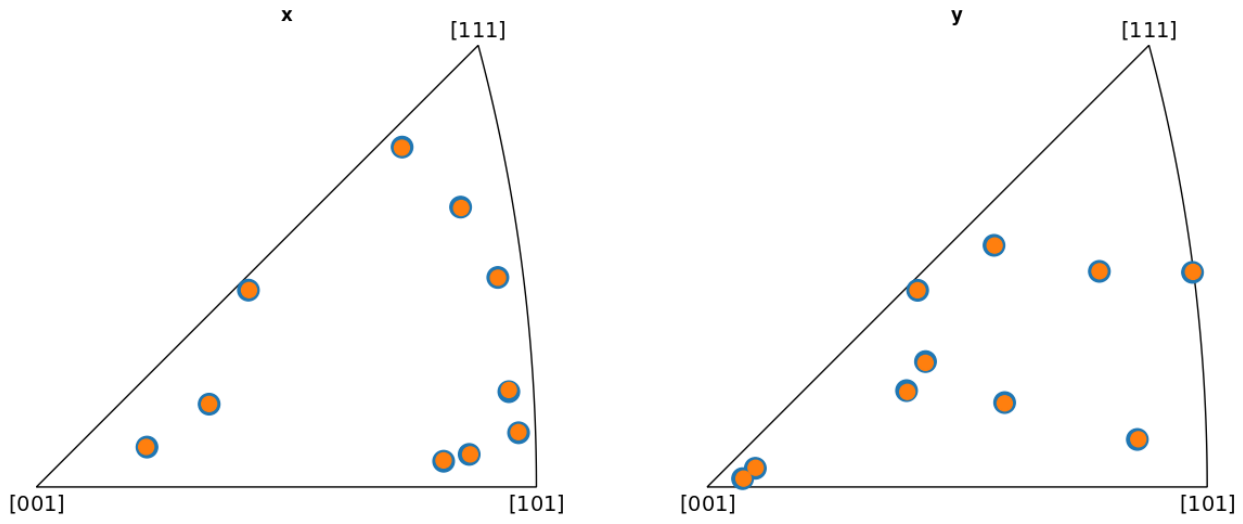
[22]:
```
v_sample = Vector3d([(1, 0, 0), (0, 1, 0)])
for vendor, pc in pcs.items():
    print(vendor)

    if vendor == "EMSOFT":
        # PyEBSDIndex requires the pixel size to be passed as the forth PC
        # value in order to correctly scale the L (PCz) parameter to obtain the
        # PC used internally in PyEBSDIndex
        pc = np.append(pc, [1])

    indexer = ebsd_index.EBSDIndexer(
        vendor=vendor,
        PC=pc,
        sampleTilt=det2.sample_tilt,
        camElev=det2.tilt,
```

(continues on next page)

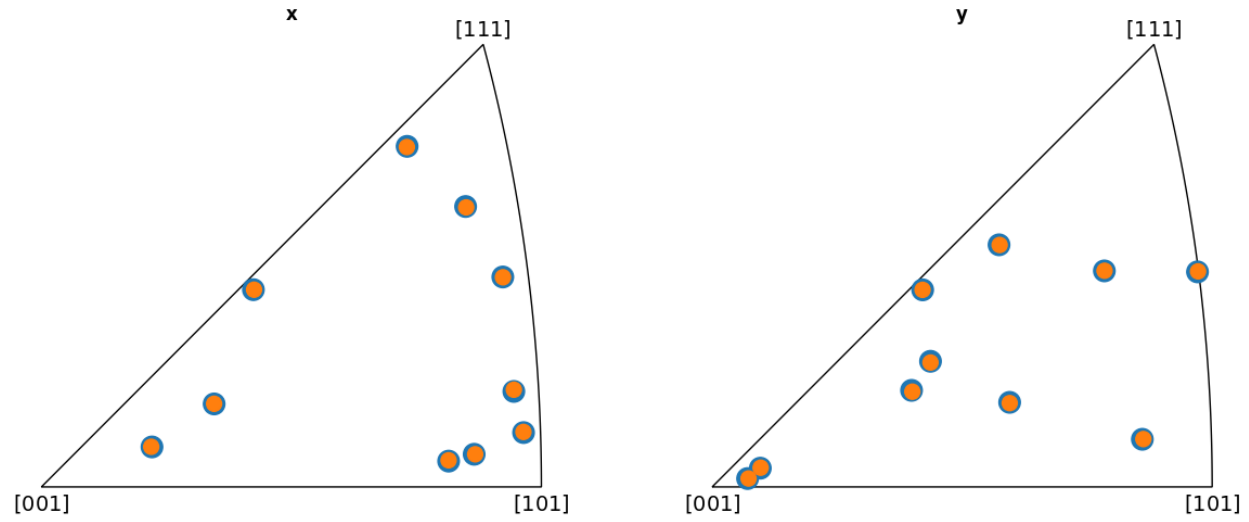
(continued from previous page)

```
        patDim=det2.shape,
    )
    data, *_ = indexer.index_pats(s2.data)
    Gr_hi = Rotation(data[0]["quat"]) * Gr_sample[vendor]
    G_hi = Orientation(Gr_hi, mp.phase.point_group)

    print(f"Average misorientation angle to ground truth: {G_hi.angle_with(G,␣
↪degrees=True).mean():.4f}")
    fig = G.scatter(
        "ipf",
        direction=v_sample,
        c="C0",
        s=200,
        return_figure=True,
    )
    G_hi.scatter("ipf", figure=fig, c="C1", s=100)
    plt.pause(0.5)  # Show IPFs before continuing with next vendor
```

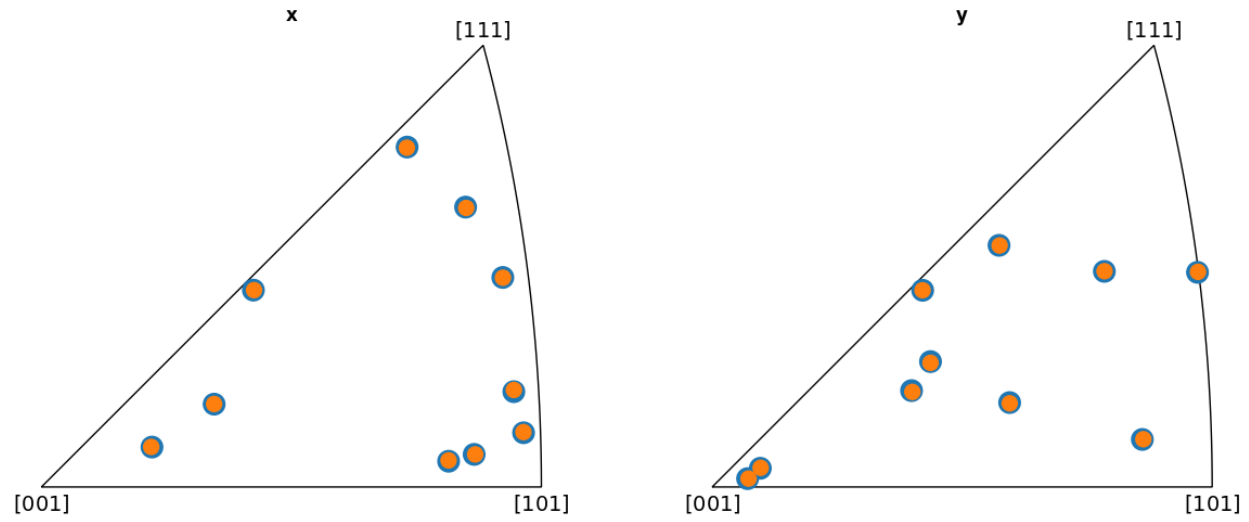
```
KIKUCHIPY
Average misorientation angle to ground truth: 0.0738
```



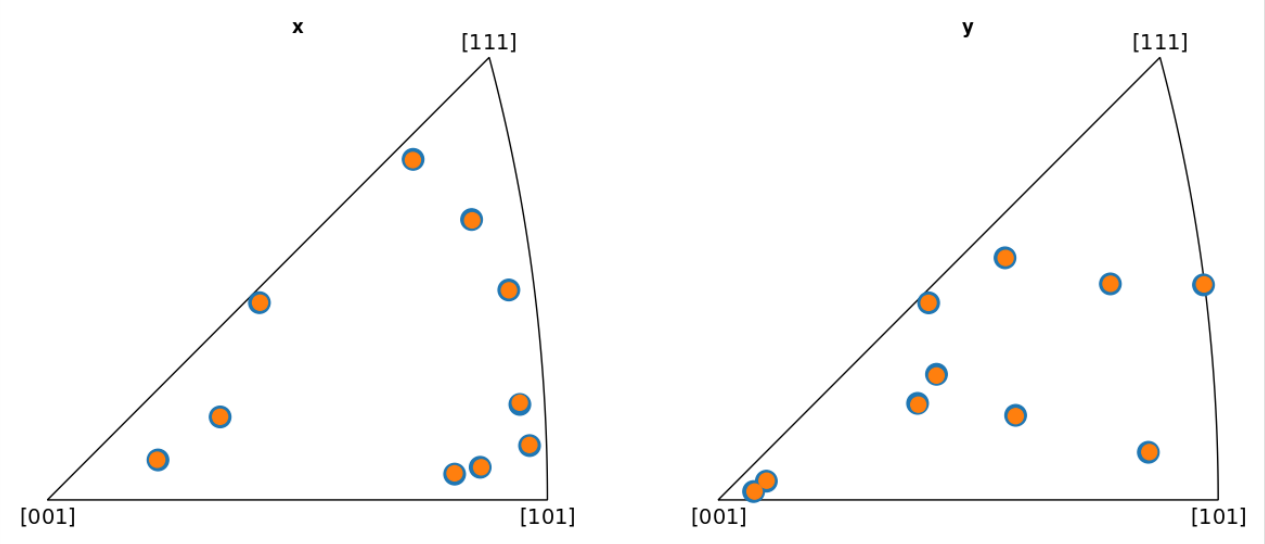

```
BRUKER
Average misorientation angle to ground truth: 0.0738
```



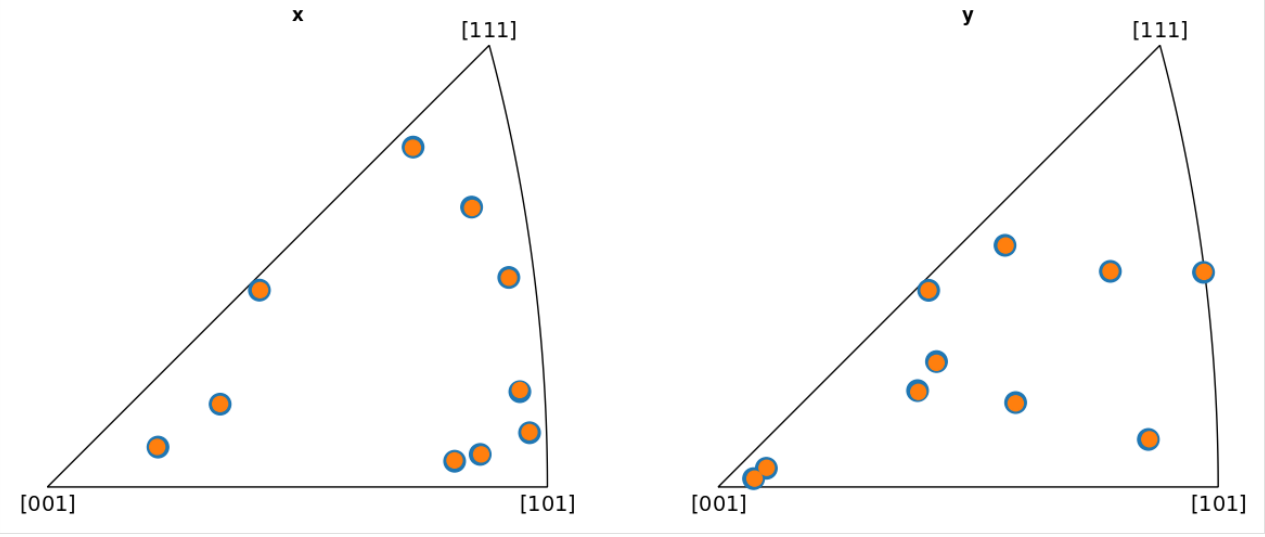

```
EDAX
Average misorientation angle to ground truth: 0.0738
```




```
OXFORD
Average misorientation angle to ground truth: 0.0738
```




```
EMSOFT
Average misorientation angle to ground truth: 0.0738
```




From the IPFs, we see that all indexed orientations for all vendors are close to the ground truth orientations, with an average misorientation angle below 0.5°. This confirms that the PC conventions for the various vendors are consistent and that PyEBSDIndex is consistent with kikuchipy.

### 1.2.2 Feature maps

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### Feature maps

In this tutorial we will extract qualitative information from pattern intensities, called feature maps (for lack of a better description).

These maps be useful when interpreting indexing results, as they are indexing independent, and also to assert the pattern quality and similarity.

Let us import the necessary libraries and a small nickel EBSD test dataset of 75 x 55 patterns

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

import hyperspy.api as hs
import kikuchipy as kp


# Use kp.load("data.h5") to load your own data
s = kp.data.nickel_ebsd_large(allow_download=True)  # External download
s
```

[1]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

### Image quality

The image quality metric $Q$ presented by Krieger Lassen [Lassen, 1994] can be calculated for an *EBSD* signal with *get_image_quality()*, or, for a single pattern (`numpy.ndarray`), with *get_image_quality()*. Following the notation in [Marquardt *et al.*, 2017], it is given by

$$Q = 1 - \frac{J}{J_{\text{res}} w_{\text{tot}}}, \tag{1.2}$$

$$J = \sum_{h=-N/2}^{N/2} \sum_{k=-N/2}^{N/2} w(h,k) \, |\mathbf{q}|^2, \tag{1.3}$$

$$J_{\text{res}} = \frac{1}{N^2} \sum_{h=-N/2}^{N/2} \sum_{k=-N/2}^{N/2} |\mathbf{q}|^2, \tag{1.4}$$

$$w_{\text{tot}} = \sum_{h=-N/2}^{N/2} \sum_{k=-N/2}^{N/2} w(h,k), \tag{1.5}$$

The function $w(h,k)$ is the Fast Fourier Transform (FFT) power spectrum of the EBSD pattern, and the vectors $\mathbf{q}$ are the frequency vectors with components $(h,k)$. The sharper the Kikuchi bands, the greater the high frequency content of the power spectrum, and thus the closer $Q$ will be to unity.

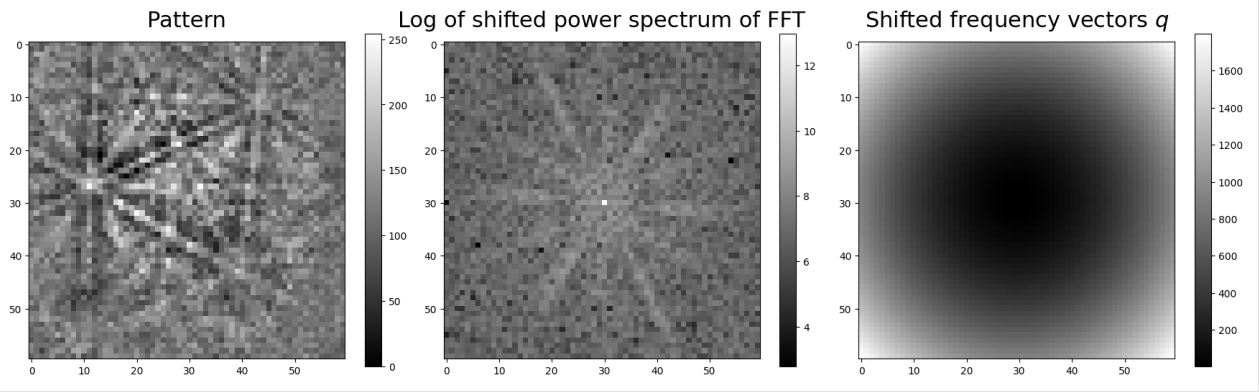
Since we want to use the image quality metric to determine how pronounced the Kikuchi bands in our patterns are, we first remove the static and dynamic background

[2]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 102.85 ms
[########################################] | 100% Completed | 709.96 ms
```

To visualize parts of the computation, we compute the power spectrum of a pattern in the Nickel EBSD data set and the frequency vectors, shift the zero-frequency components to the centre, and plot them

[3]:
```
p = s.inav[20, 11].data
p_fft = kp.pattern.fft(p, shift=True)
q = kp.pattern.fft_frequency_vectors(shape=p.shape)

fig, ax = plt.subplots(figsize=(22, 6), ncols=3)
title_kwargs = dict(fontsize=22, pad=15)
fig.subplots_adjust(wspace=0.05)

im0 = ax[0].imshow(p, cmap="gray")
ax[0].set_title("Pattern", **title_kwargs)
fig.colorbar(im0, ax=ax[0])

im1 = ax[1].imshow(np.log(kp.pattern.fft_spectrum(p_fft)), cmap="gray")
ax[1].set_title("Log of shifted power spectrum of FFT", **title_kwargs)
fig.colorbar(im1, ax=ax[1])

im2 = ax[2].imshow(np.fft.fftshift(q), cmap="gray")
ax[2].set_title(r"Shifted frequency vectors $q$", **title_kwargs)
_ = fig.colorbar(im2, ax=ax[2])
```




If we don't want the EBSD patterns to be *zero-mean normalized* before computing $Q$, we must pass `get_image_quality(normalize=False)`.

Let's compute the image quality $Q$ and plot it for the entire data set (using the CrystalMap.plot() method of the *EBSD.xmap* attribute)

[4]:
```
iq = s.get_image_quality(normalize=True)  # Default
```

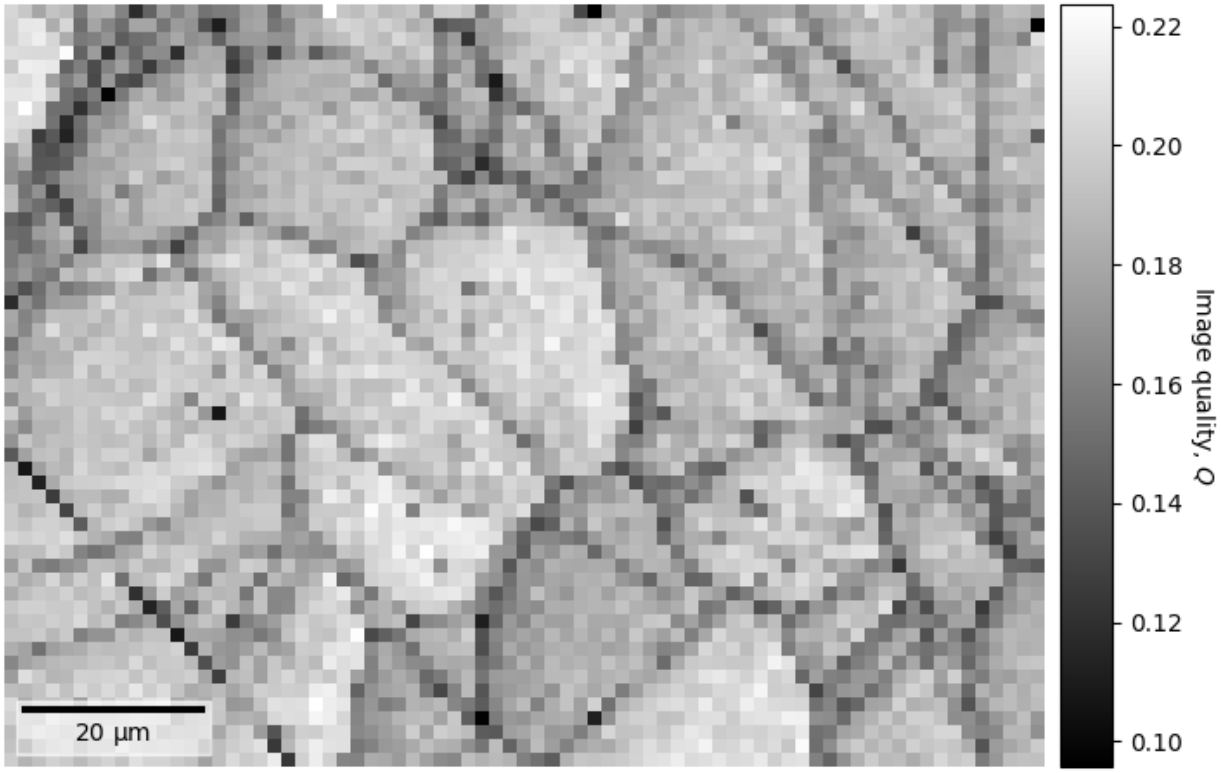
```
[########################################] | 100% Completed | 404.28 ms
```

[5]:
```
s.xmap.plot(
    iq.ravel(),
    cmap="gray",
    colorbar=True,
    colorbar_label="Image quality, $Q$",
```

(continues on next page)

(continued from previous page)

```
    remove_padding=True,
)
```




If we want to use this map to navigate around in when plotting patterns, we can easily do that as explained in the *visualizing patterns* tutorial.

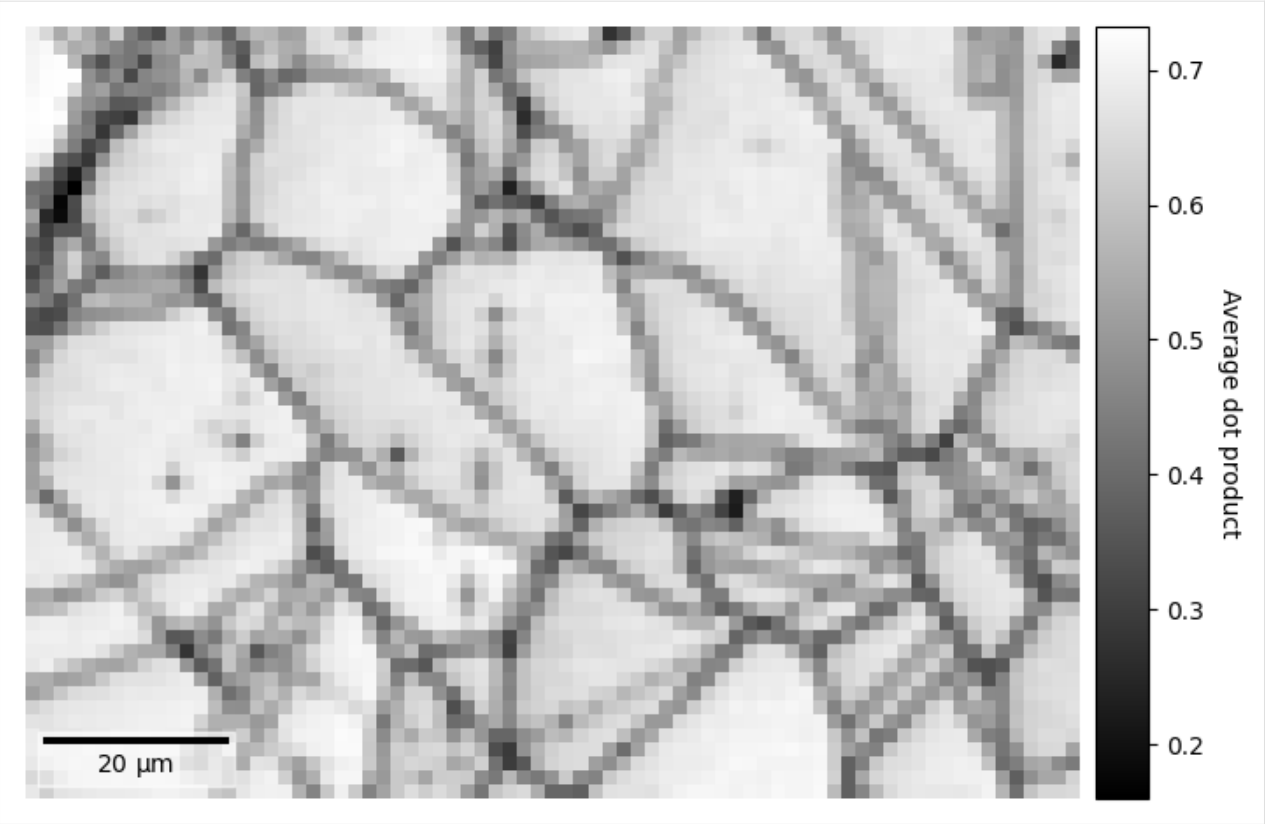
### Average dot product

The average dot product, or normalized cross-correlation when centering each pattern's intensity about zero and normalizing the intensities to a standard deviation $\sigma$ of 1 (which is the default behaviour), between each pattern and their four nearest neighbours, can be obtained for an *EBSD* signal with *get_average_neighbour_dot_product_map()*

[6]:
```
adp = s.get_average_neighbour_dot_product_map()
```

```
[########################################] | 100% Completed | 1.16 s
```

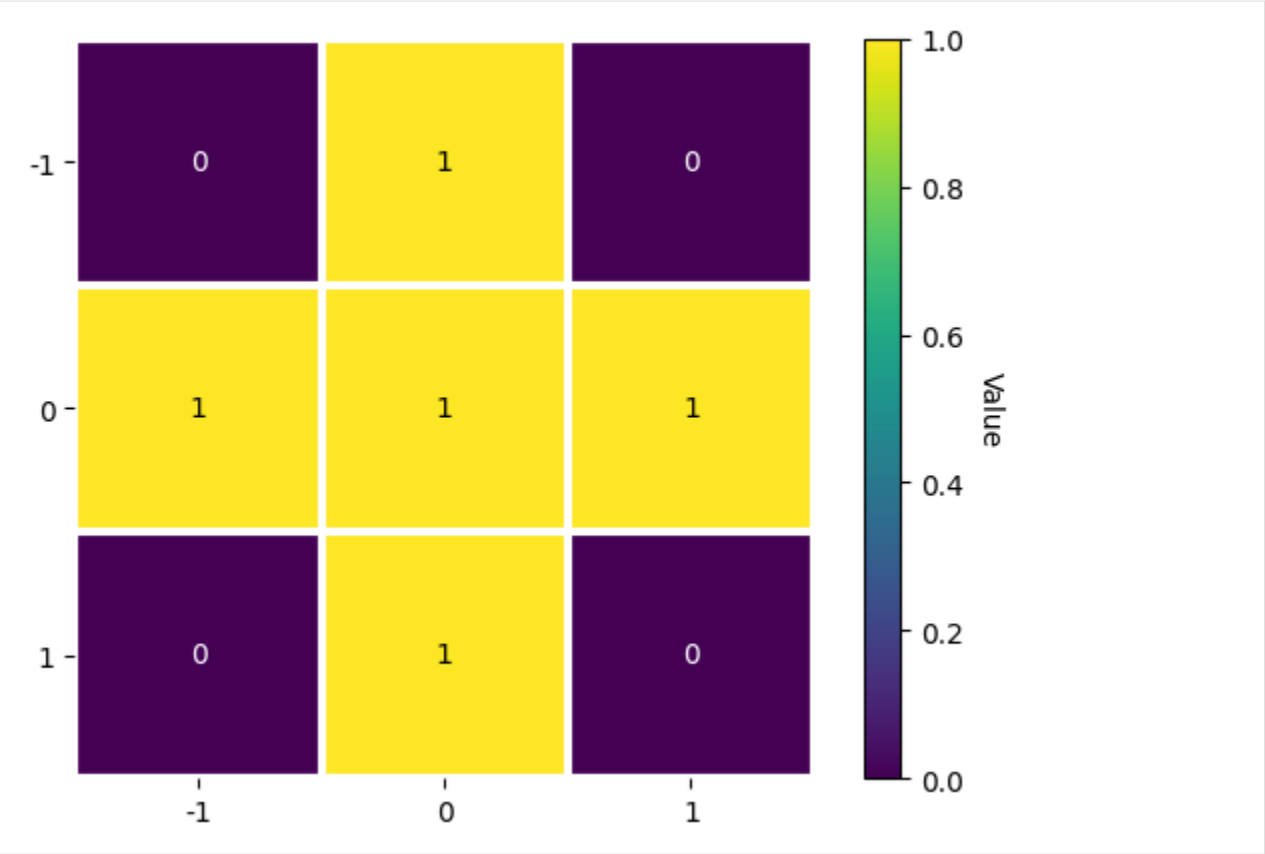
[7]:
```
s.xmap.plot(
    adp.ravel(),
    cmap="gray",
    colorbar=True,
    colorbar_label="Average dot product",
    remove_padding=True,
)
```




The map displays how similar each pattern is to its neighbours. Grain boundaries, and some scratches on the sample, can be clearly seen as pixels with a lower value, signifying that they are more dissimilar to their neighbouring pixels, as the ones within grains where the neighbour pixel similarity is high.

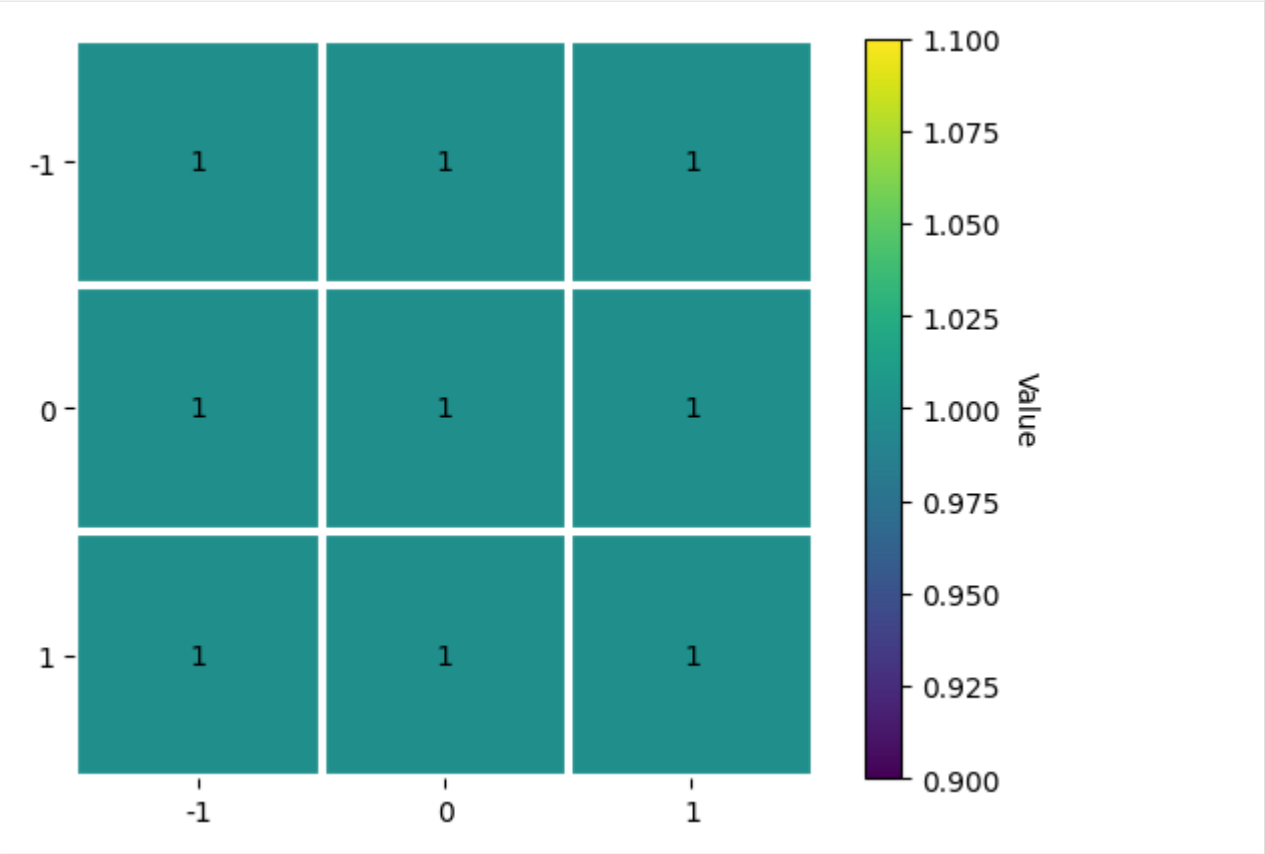
The map above was created by averaging the dot product matrix per map point, created by calculating the dot product between each pattern and their four nearest neighbours, which can be seen in the black spots (uneven sample surface) in the left grains

[8]:
```
w1 = kp.filters.Window()
w1.plot()
```



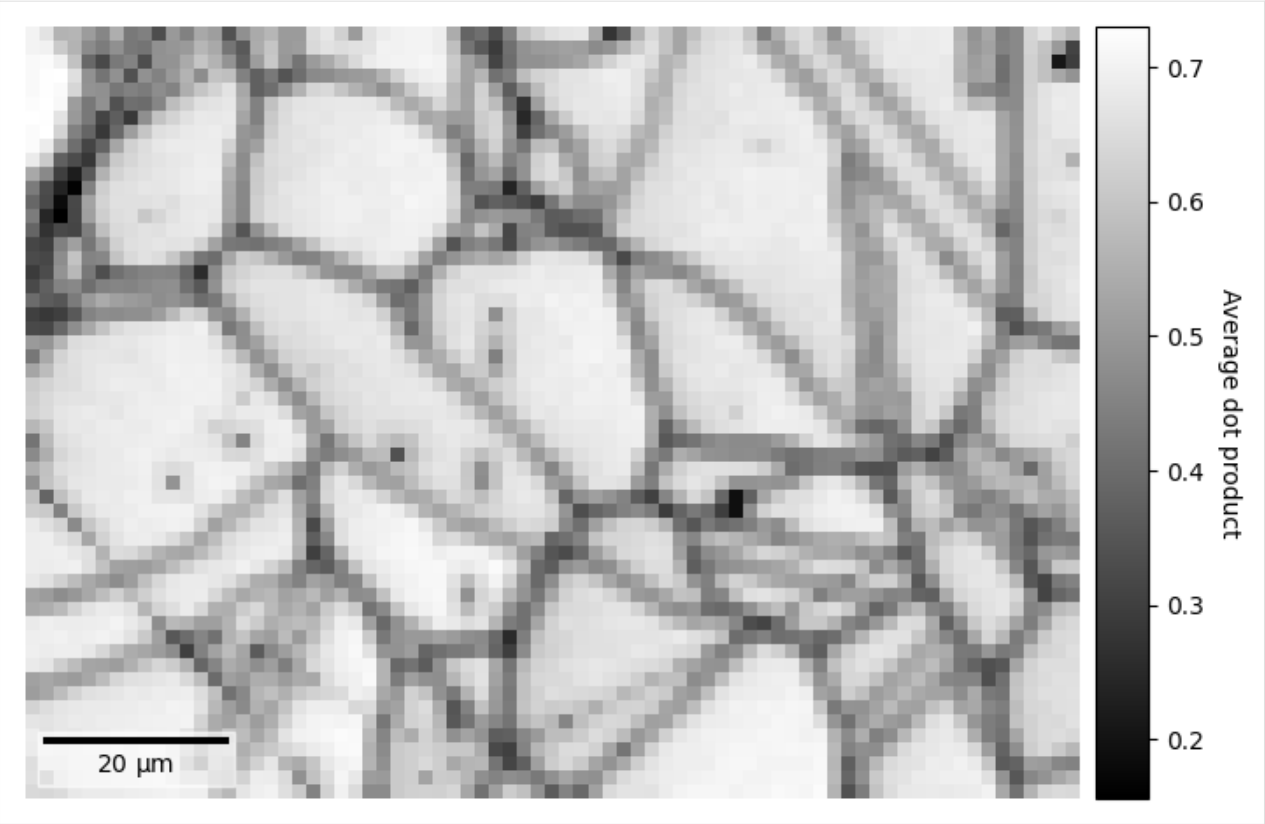

We could instead average with e.g. the eight nearest neighbours

[9]:
```
w2 = kp.filters.Window(window="rectangular", shape=(3, 3))
w2.plot()
```




[10]:
```
adp2 = s.get_average_neighbour_dot_product_map(window=w2)

s.xmap.plot(
    adp2.ravel(),
    cmap="gray",
    colorbar=True,
    colorbar_label="Average dot product",
    remove_padding=True,
)
```

```
[########################################] | 100% Completed | 2.35 s
```




Note that the window coefficients must be integers.

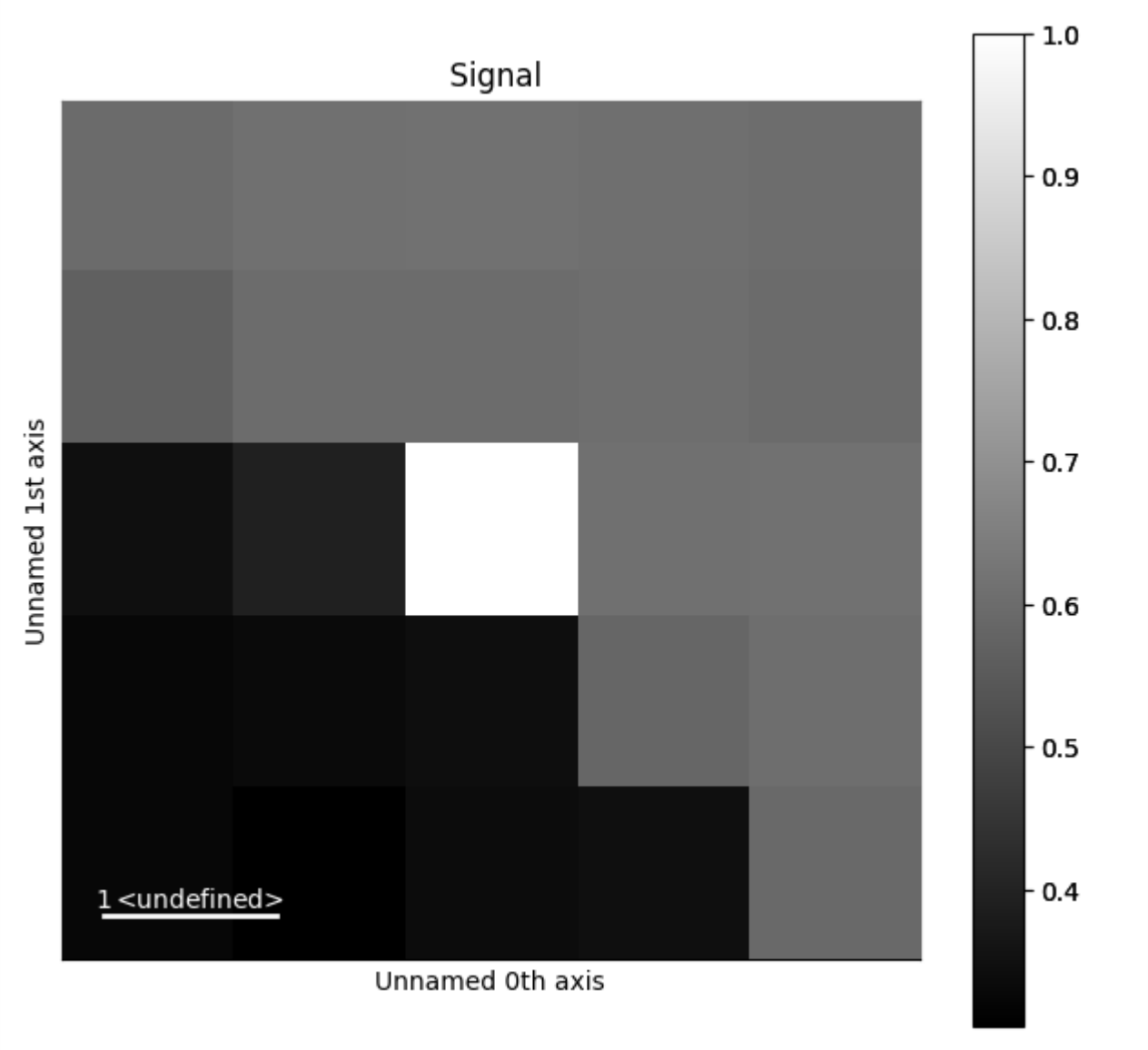
We can also control whether pattern intensities should be centered about zero and/or whether they should be normalized prior to calculating the dot products by passing `zero_mean=False` and/or `normalize=False`. These are `True` by default. The data type of the output map, 32-bit floating point by default, can be set by passing e.g. `dtype_out=np.float64`.

We can obtain the dot product matrices per map point, that is the matrices before they are averaged, with *get_neighbour_dot_product_matrices()*. Let's see similar a pattern on a grain boundary in map point (x, y) = (50, 19) is to all its nearest neighbour in a (5, 5) window centered on that point

[11]:
```
w3 = kp.filters.Window("rectangular", shape=(5, 5))
dp_matrices = s.get_neighbour_dot_product_matrices(window=w3)
```

```
[########################################] | 100% Completed | 4.08 s
```

[12]:
```
x, y = (50, 19)
s_dp_matrices = hs.signals.Signal2D(dp_matrices)
s_dp_matrices.inav[x, y].plot()
```



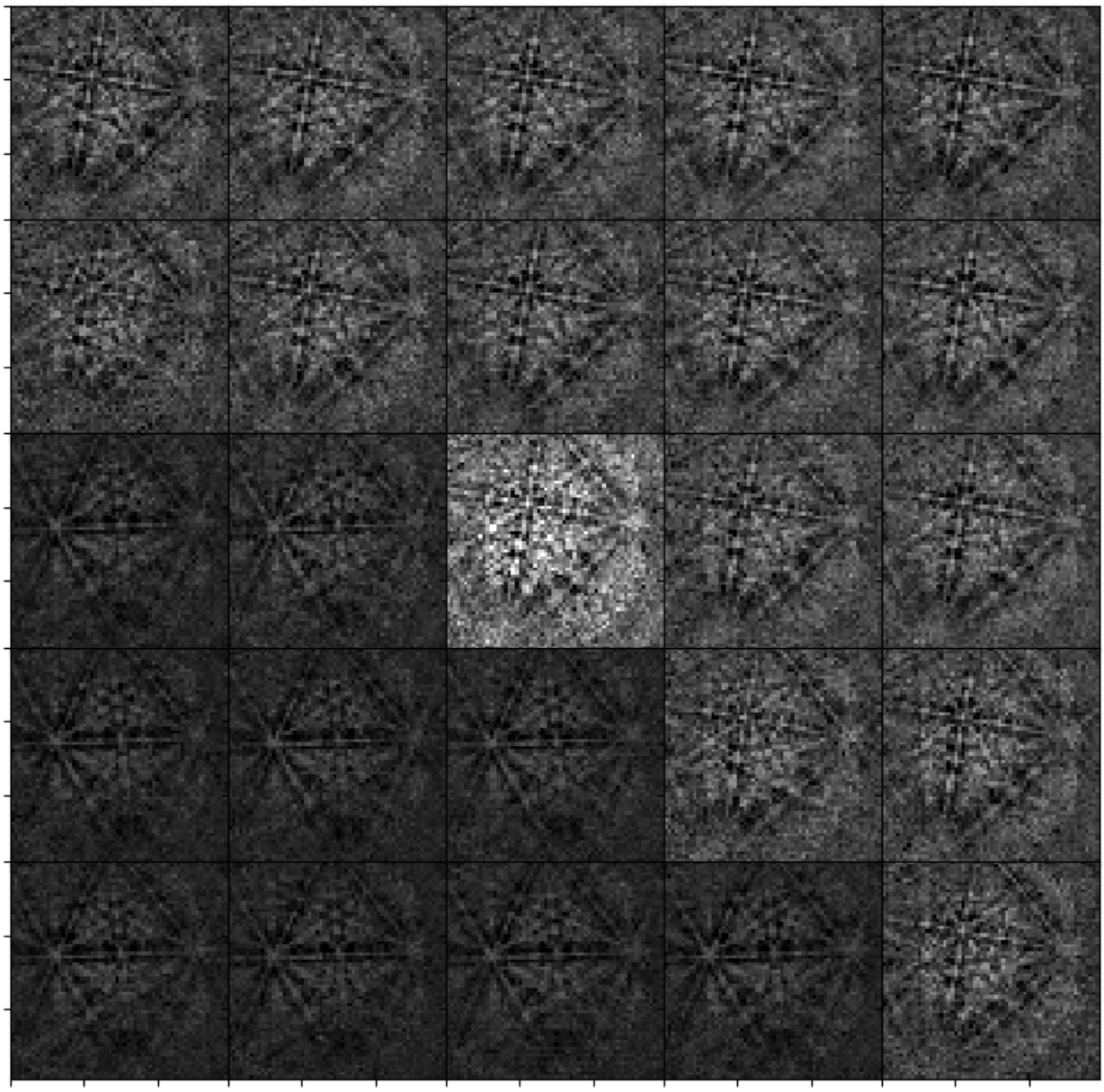

We can see that the pattern is more similar to the patterns up to the right, while it is quite dissimilar to the patterns to the lower left. Let's visualize this more clearly, as is done e.g. in Fig. 1 by [Brewick *et al.*, 2019]

[13]:
```
y_n, x_n = w3.n_neighbours

s2 = s.inav[x - x_n : x + x_n + 1, y - y_n : y + y_n + 1].deepcopy()
s2.rescale_intensity(percentiles=(0.5, 99.5))  # Stretch the contrast a bit

# Signals must have same navigation shape (warning can be ignored)
s3 = s2 * s_dp_matrices.inav[x, y].T
```

```
[########################################] | 100% Completed | 102.30 ms
```

```
WARNING:hyperspy.misc.signal_tools:Axis calibration mismatch detected along axis 0. The␣
↪calibration of signal 0 along this axis will be applied to all signals after stacking.
WARNING:hyperspy.misc.signal_tools:Axis calibration mismatch detected along axis 1. The␣
↪calibration of signal 0 along this axis will be applied to all signals after stacking.
```

[14]:
```
_ = hs.plot.plot_images(
    images=s3,
    per_row=5,
    label=None,
    suptitle=None,
    axes_decor=None,
    colorbar=None,
    vmin=int(s3.data.min()),
    vmax=int(s3.data.max()),
    padding=dict(wspace=0, hspace=-0.05),
    fig=plt.figure(figsize=(10, 10)),
)
```



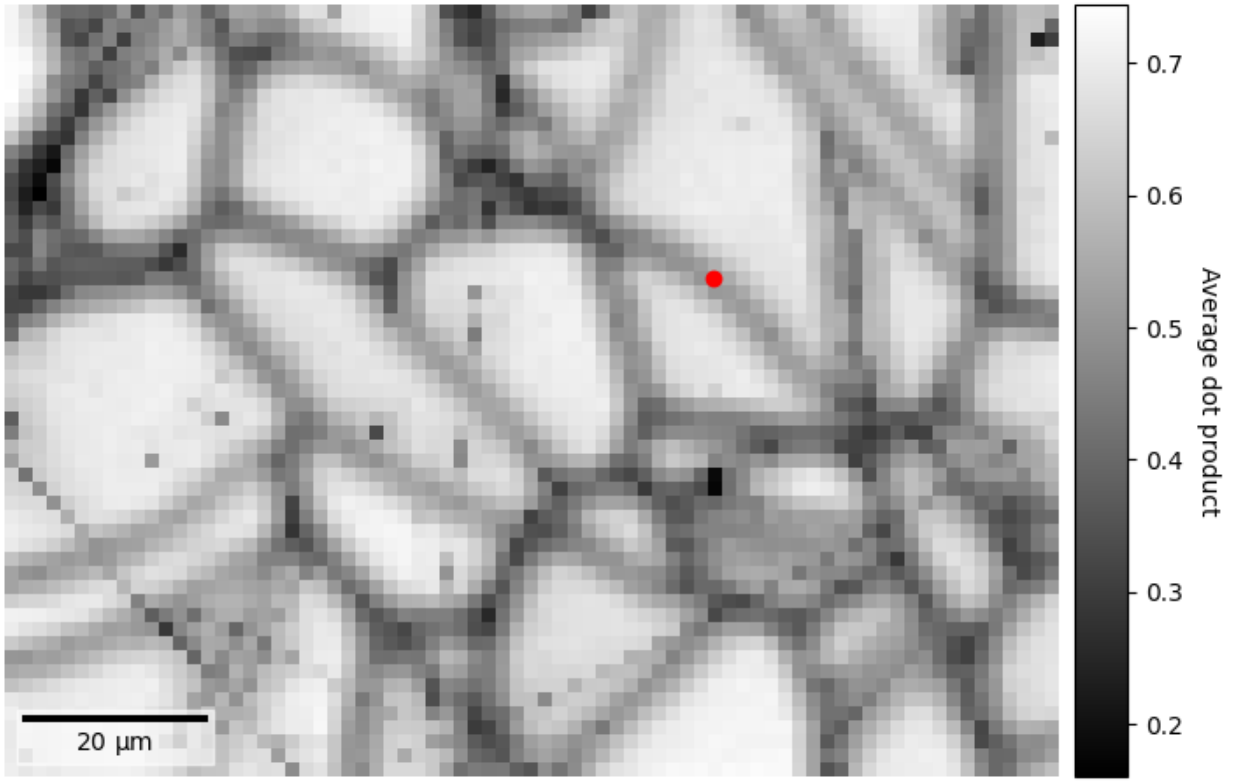
Finally, we can pass this dot product matrix directly to *get_average_neighbour_dot_product_map()* via the

`dp_matrices` parameter to obtain the average dot product map from these matrices

[15]:
```
adp3 = s.get_average_neighbour_dot_product_map(dp_matrices=dp_matrices)
```

Let's plot this and highlight the location of the pattern on the grain boundary above with a red circle

[16]:
```
fig = s.xmap.plot(
    adp3.ravel(),
    cmap="gray",
    colorbar=True,
    colorbar_label="Average dot product",
    remove_padding=True,
    return_figure=True
)
fig.axes[0].scatter(x, y, c="r");
```




**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### Virtual backscatter electron imaging

In this tutorial, we will perform virtual imaging on the EBSD detector to generate maps, known as virtual backscatter electron (VBSE) imaging.

This is useful for getting a qualitative overview of the sample and pattern quality prior to indexing, and can also be useful when interpreting indexing resuls, as they are indexing independent.

### Interactive plotting

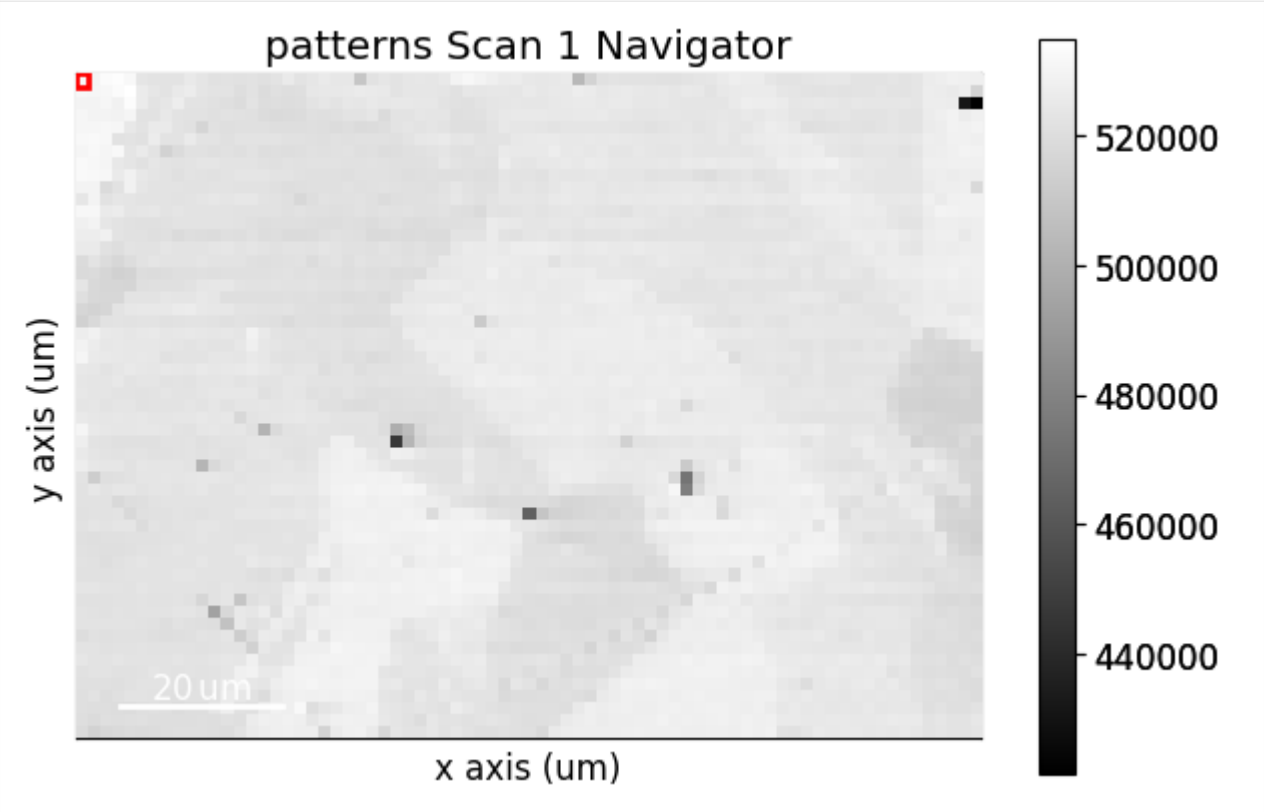
Angle resolved backscatter electron (BSE) imaging can be performed interactively with the method *plot_virtual_bse_intensity()*, adopted from pyxem, by integrating the intensities within a part, e.g. a (10 x 10) pixel rectangular region of interest (ROI), of the stack of EBSD patterns. Let's first import necessary libraries and a 13 MB Nickel EBSD data set

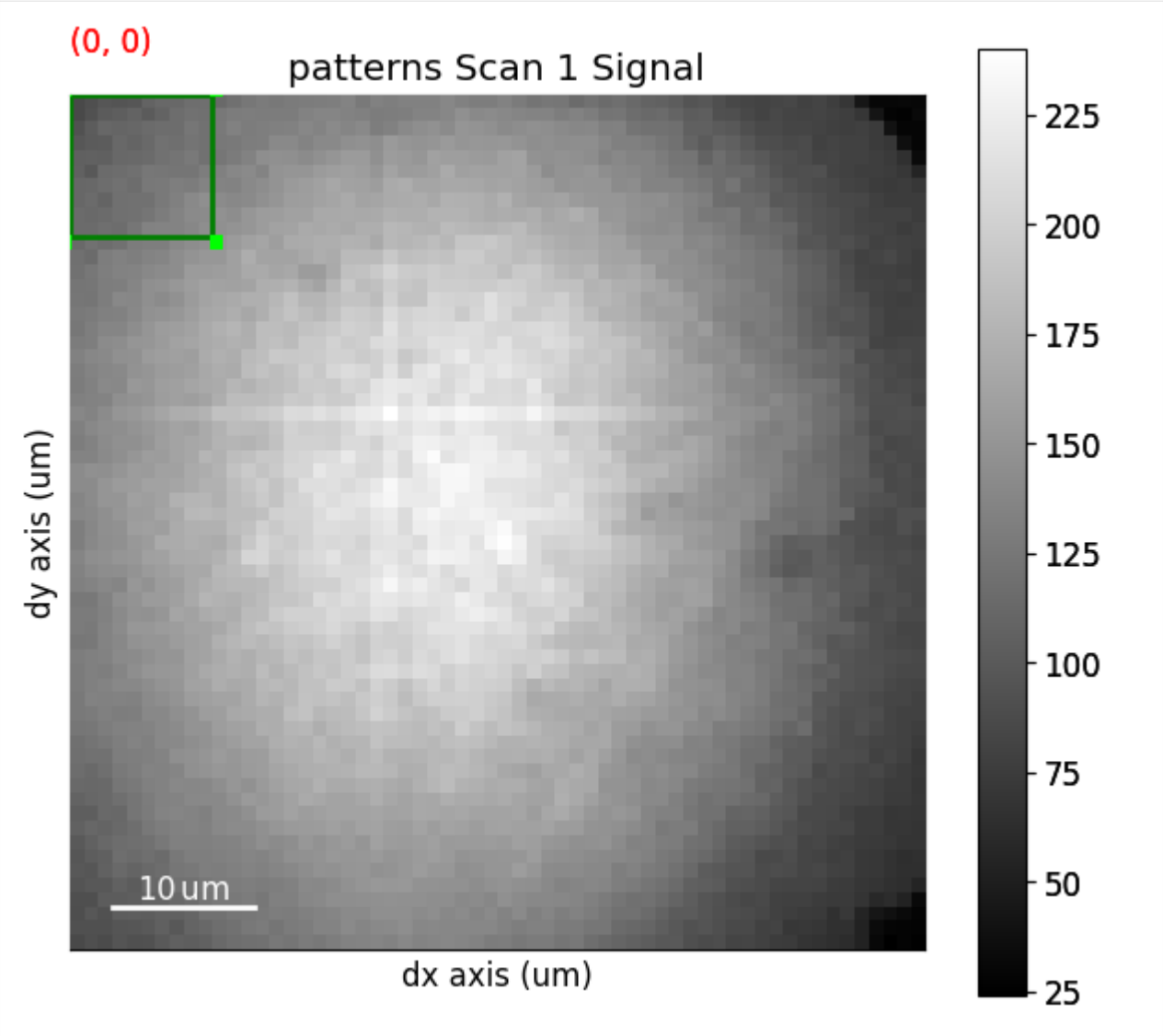
[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

from pathlib import Path
import tempfile

import matplotlib.pyplot as plt
import numpy as np

import hyperspy.api as hs
import kikuchipy as kp


plt.rcParams["font.size"] = 12


s = kp.data.nickel_ebsd_large(allow_download=True)  # External download
s
```

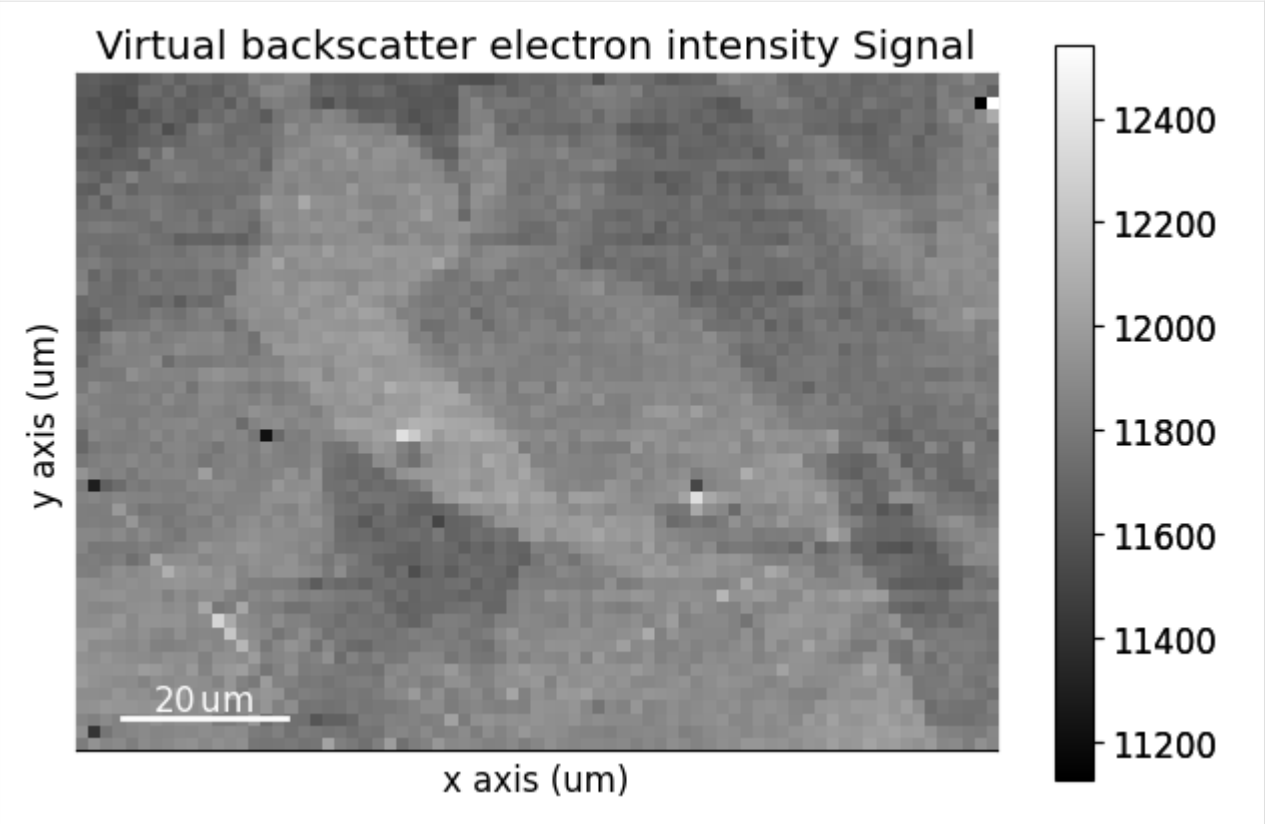
[1]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

We create a rectangular ROI by specifying the upper left and lower right coordinates of the rectangle in units of the detector pixel size (scale of dx and dy in the signal axes manager)

[2]:
```
roi = hs.roi.RectangularROI(left=0, top=0, right=10, bottom=10)
s.plot_virtual_bse_intensity(roi)
```


patterns Scan 1 Navigator
y axis (um)
x axis (um)
20 um
520000
500000
480000
460000
440000


(0, 0)
patterns Scan 1 Signal
dy axis (um)
dx axis (um)
10 um
225
200
175
150
125
100
75
50
25




Note that the position of the ROI on the detector is updated during interactive plotting if it is moved around by hand. See HyperSpy's ROI user guide for more detailed use of ROIs.

The virtual image, created from integrating the intensities within the ROI, can then be written to an image file using *get_virtual_bse_intensity()*

[3]:
```
vbse = s.get_virtual_bse_intensity(roi)
vbse
```

[3]:
```
<VirtualBSEImage, title: Virtual backscatter electron image, dimensions: (|75, 55)>
```

[4]:
```
temp_dir = Path(tempfile.mkdtemp())
plt.imsave(temp_dir / "vbse1.png", arr=vbse.data)
```

A *VirtualBSEImage* instance is returned.

### Generate many virtual images

Sometimes we want to get many images from parts of the detector, e.g. like what is demonstrated in the xcdskd project with the angle resolved virtual backscatter electron array (arbse/vbse array). Instead of keeping track of multiple hyperspy.roi.BaseInteractiveROI objects, we can create a detector grid of a certain shape, e.g. (5, 5), and obtain gray scale images, or combine multiple grid tiles in red, green and channels to obtain RGB images.

First, we initialize a virtual BSE image generator, *kikuchipy.imaging.VirtualBSEImager*, with an *EBSD* signal, in case the raw EBSD patterns without any background correction or other processing

[5]:
```
vbse_imager = kp.imaging.VirtualBSEImager(s)
vbse_imager
```

[5]:
```
VirtualBSEImager for <EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

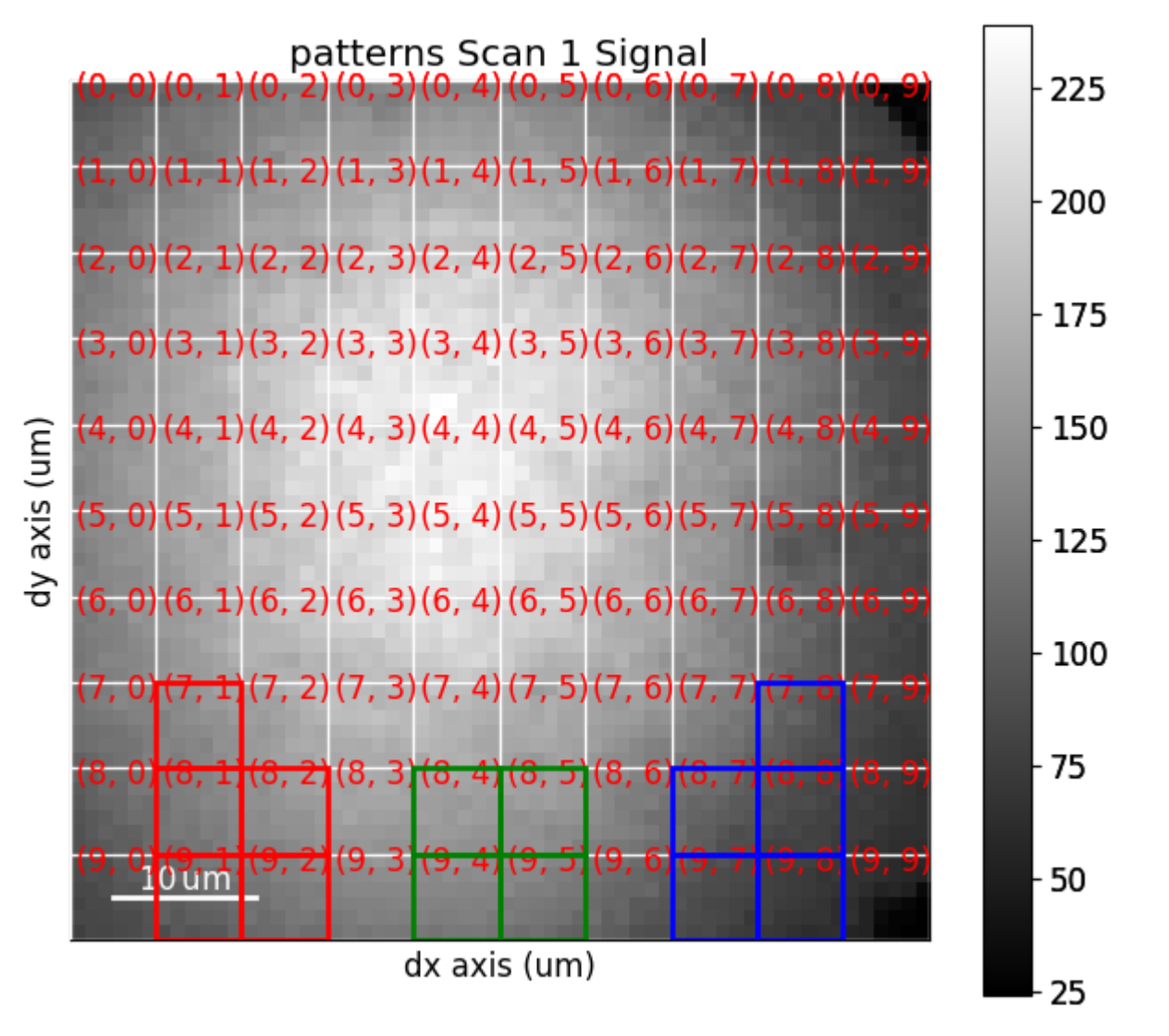
We can set and plot the detector grid on one of the EBSD patterns, also coloring one or more of the grid tiles red, green and blue, as is done in [Nolze *et al.*, 2017], by calling *VirtualBSEImager.plot_grid()*

[6]:
```
vbse_imager.grid_shape
```

[6]:
```
(5, 5)
```

[7]:
```
vbse_imager.grid_shape = (10, 10)
red = [(7, 1), (8, 1), (8, 2), (9, 1), (9, 2)]
green = [(8, 4), (8, 5), (9, 4), (9, 5)]
blue = [(7, 8), (8, 7), (8, 8), (9, 7), (9, 8)]
p = vbse_imager.plot_grid(
    rgb_channels=[red, green, blue],
    visible_indices=True,  # Default
    pattern_idx=(10, 20),  # Default is (0, 0)
)
p
```

[7]:
```
<EBSD, title: patterns Scan 1, dimensions: (|60, 60)>
```



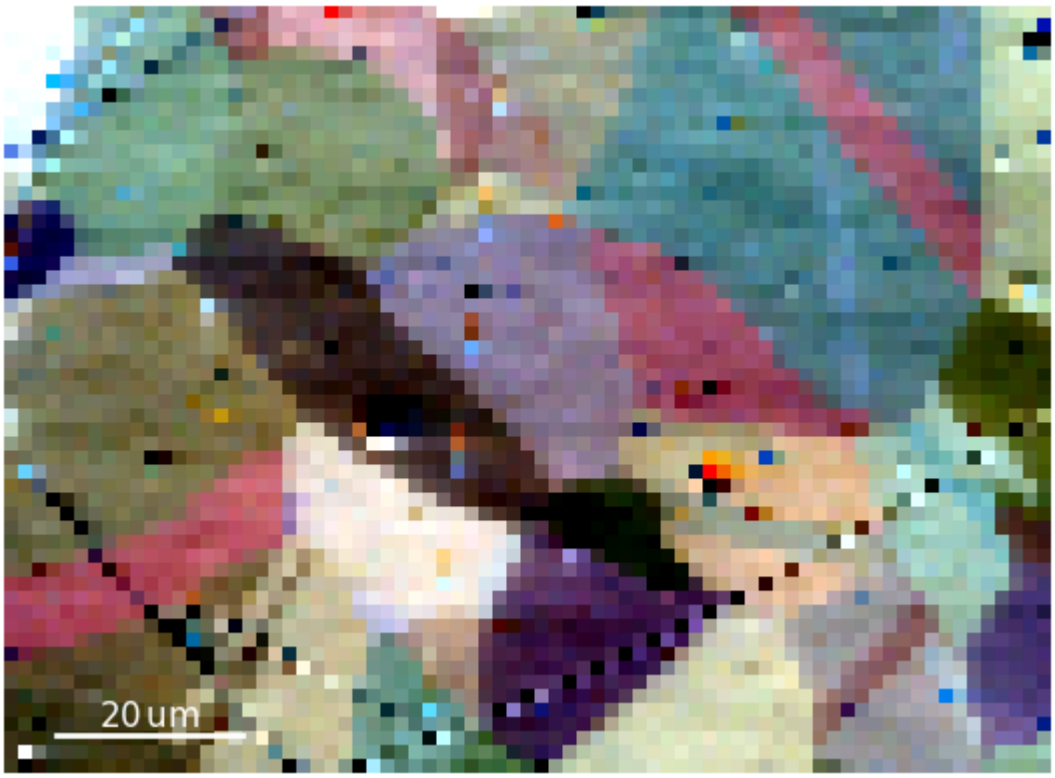

As shown above, whether to show the grid tile indices or not is controlled with the `visible_indices` argument, and which signal pattern to superimpose the grid upon is controlled with the `pattern_idx` parameter.

To obtain an RGB image from the detector grid tiles shown above, we use *get_rgb_image()* (see the docstring for all available parameters)

[8]:
```
vbse_rgb_img = vbse_imager.get_rgb_image(r=red, g=green, b=blue)
vbse_rgb_img
```

[8]:
```
<VirtualBSEImage, title: , dimensions: (|75, 55)>
```

[9]:
```
vbse_rgb_img.plot(title="", axes_off=True)
```



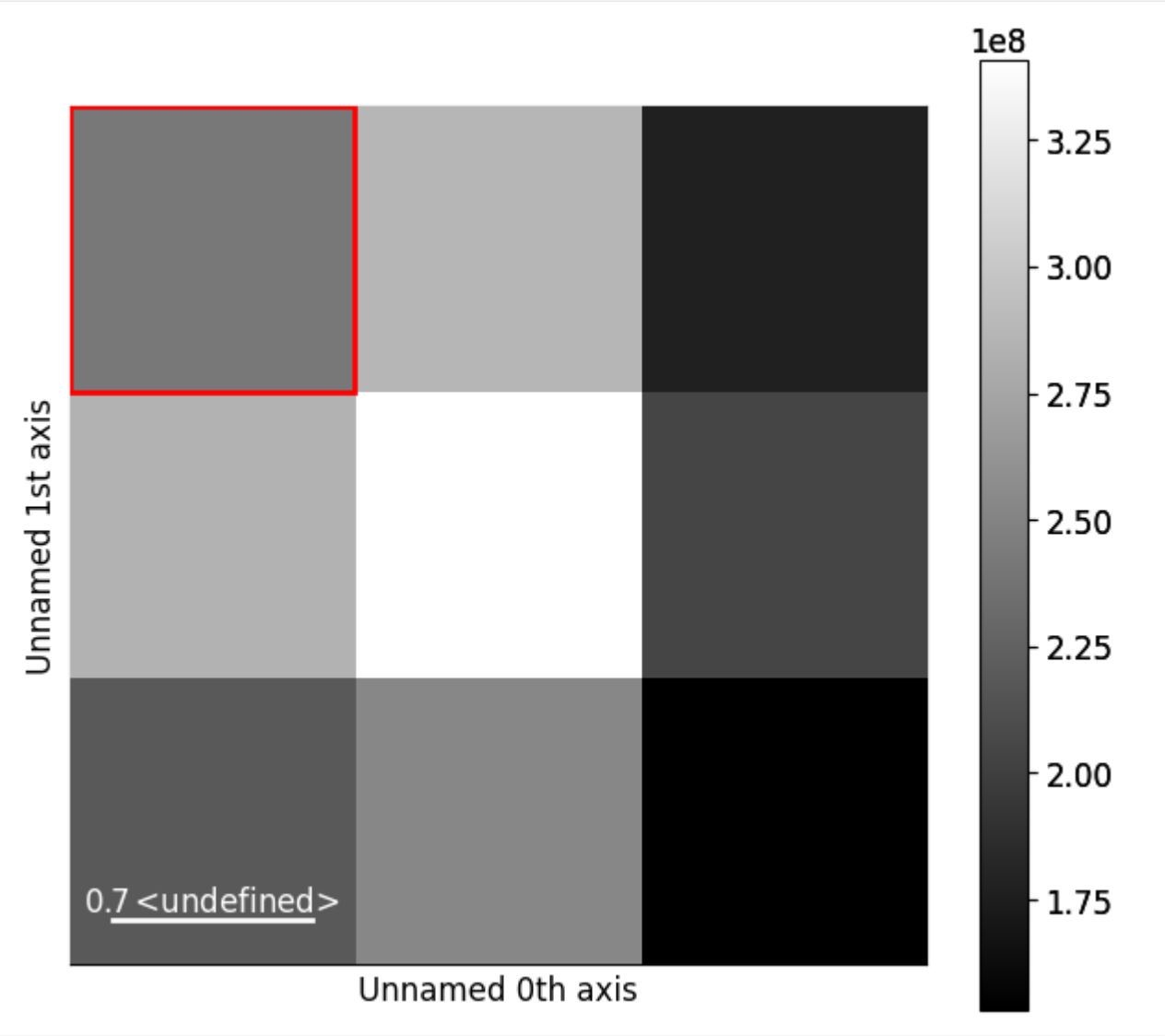

An RGB image formed from coloring three grey scale virtual BSE images red, green and blue.

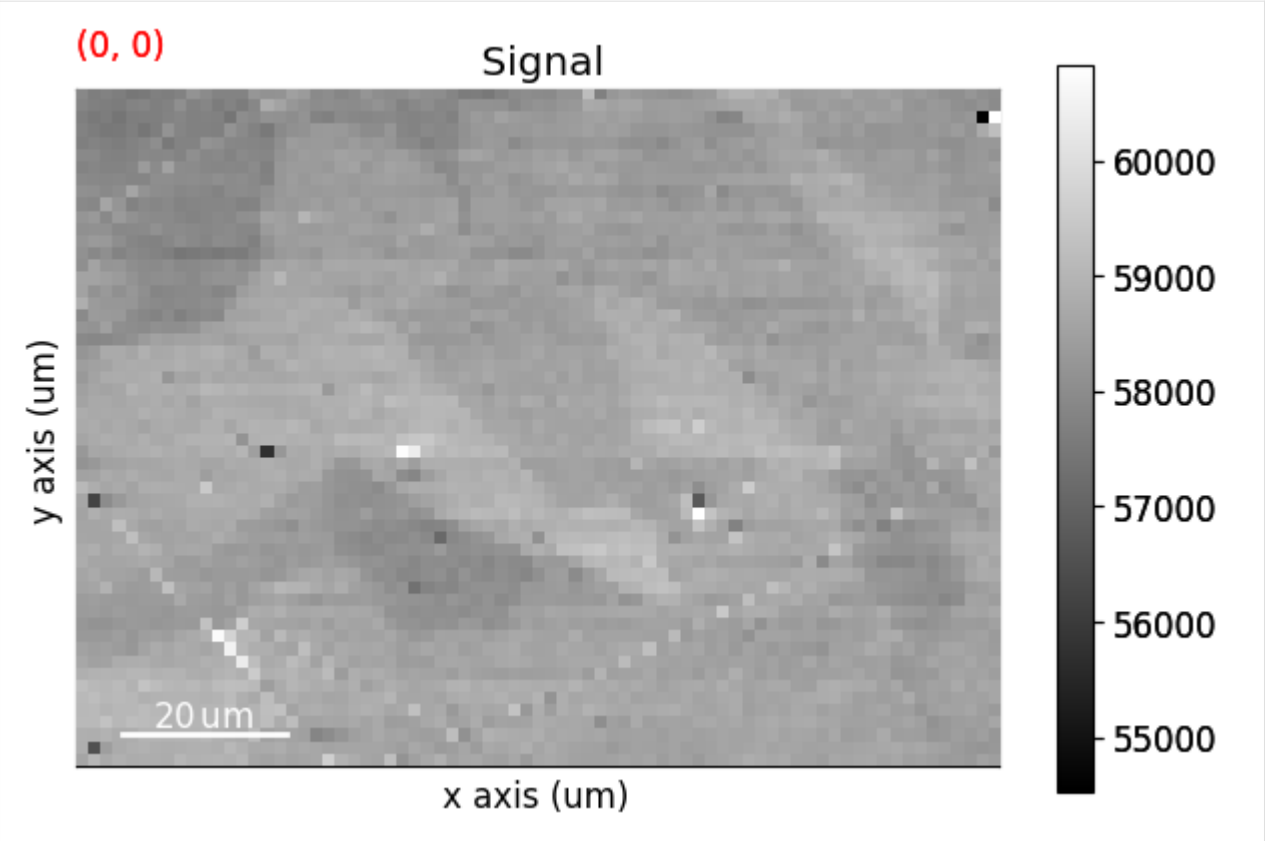
To obtain one grey scale virtual BSE image from each grid tile, we use *get_images_from_grid()*

[10]:
```
vbse_imager.grid_shape = (3, 3)
vbse_imgs = vbse_imager.get_images_from_grid()
vbse_imgs
```

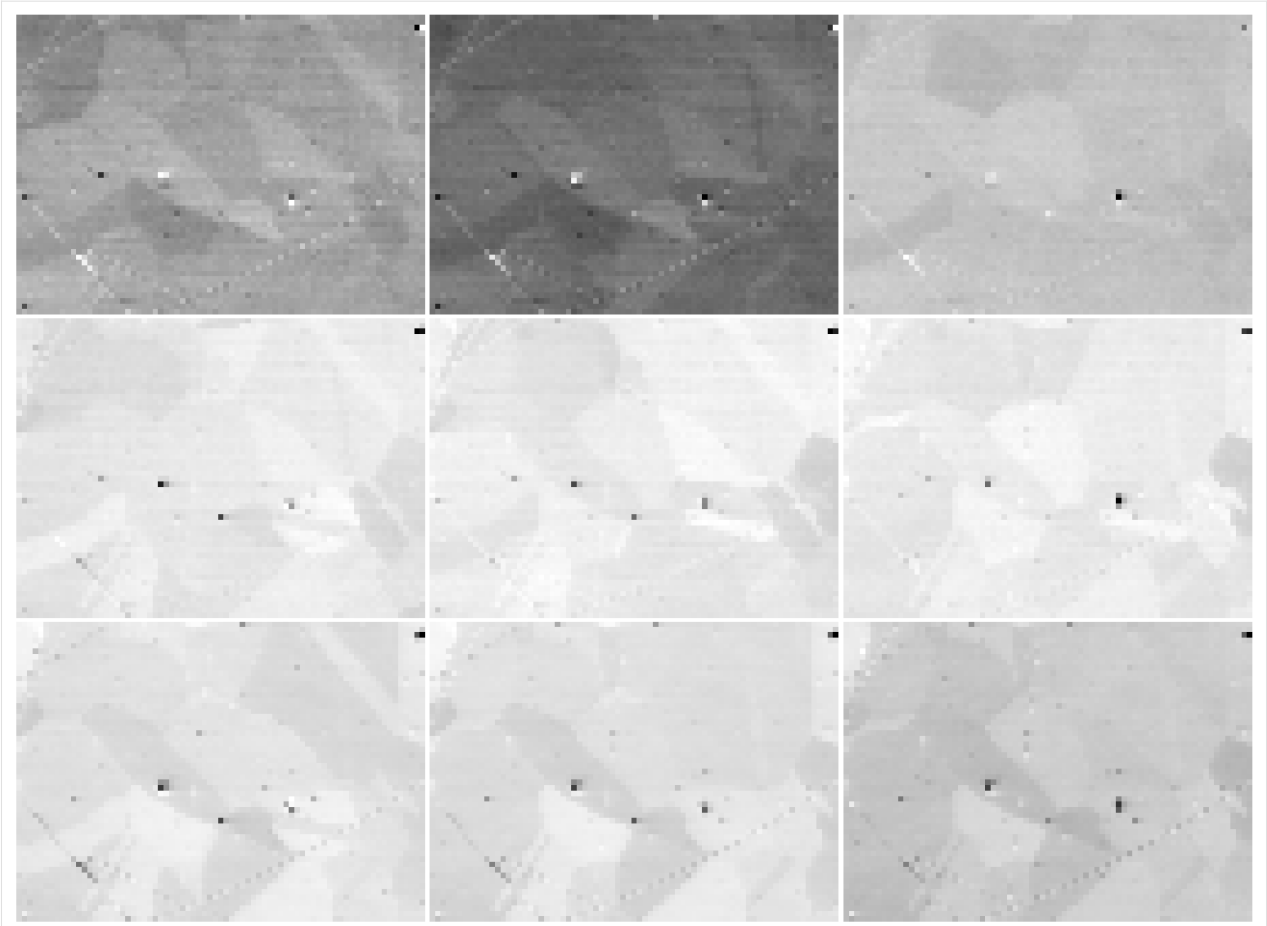
[10]:
```
<VirtualBSEImage, title: , dimensions: (3, 3|75, 55)>
```

[11]:
```
vbse_imgs.plot()
```


1e8
3.25
3.00
2.75
2.50
2.25
2.00
1.75
Unnamed 1st axis
Unnamed 0th axis
0.7 <undefined>




[12]:
```
fig, ax = plt.subplots(nrows=3, ncols=3, figsize=(20, 20))
for idx in np.ndindex(vbse_imgs.axes_manager.navigation_shape[::-1]):
    # HyperSpy uses (col, row) instead of NumPy's (row, col)
    hs_idx = idx[::-1]
    ax[idx].imshow(vbse_imgs.inav[hs_idx].data, cmap="gray")
    ax[idx].axis("off")
fig.tight_layout(w_pad=0.5, h_pad=-24)
```



It might be desirable to normalize, rescale or stretch the intensities in the images, as shown e.g. in Fig. 9 in [Wright *et al.*, 2015]. This can be done with *VirtualBSEImage.normalize_intensity()* or *VirtualBSEImage.rescale_intensity()*. Let's rescale the intensities in each image to the range [0, 1], while also excluding the intensities outside the lower and upper 0.5% percentile, per image

[13]:
```
vbse_imgs.data.dtype
```

[13]:
```
dtype('float32')
```

[14]:
```
vbse_imgs2 = vbse_imgs.deepcopy()
vbse_imgs2.rescale_intensity(out_range=(0, 1), percentiles=(0.5, 99.5))
```

```
[########################################] | 100% Completed | 114.89 ms
```

[15]:
```
print(vbse_imgs.data.min(), vbse_imgs.data.max())
print(vbse_imgs2.data.min(), vbse_imgs2.data.max())
```

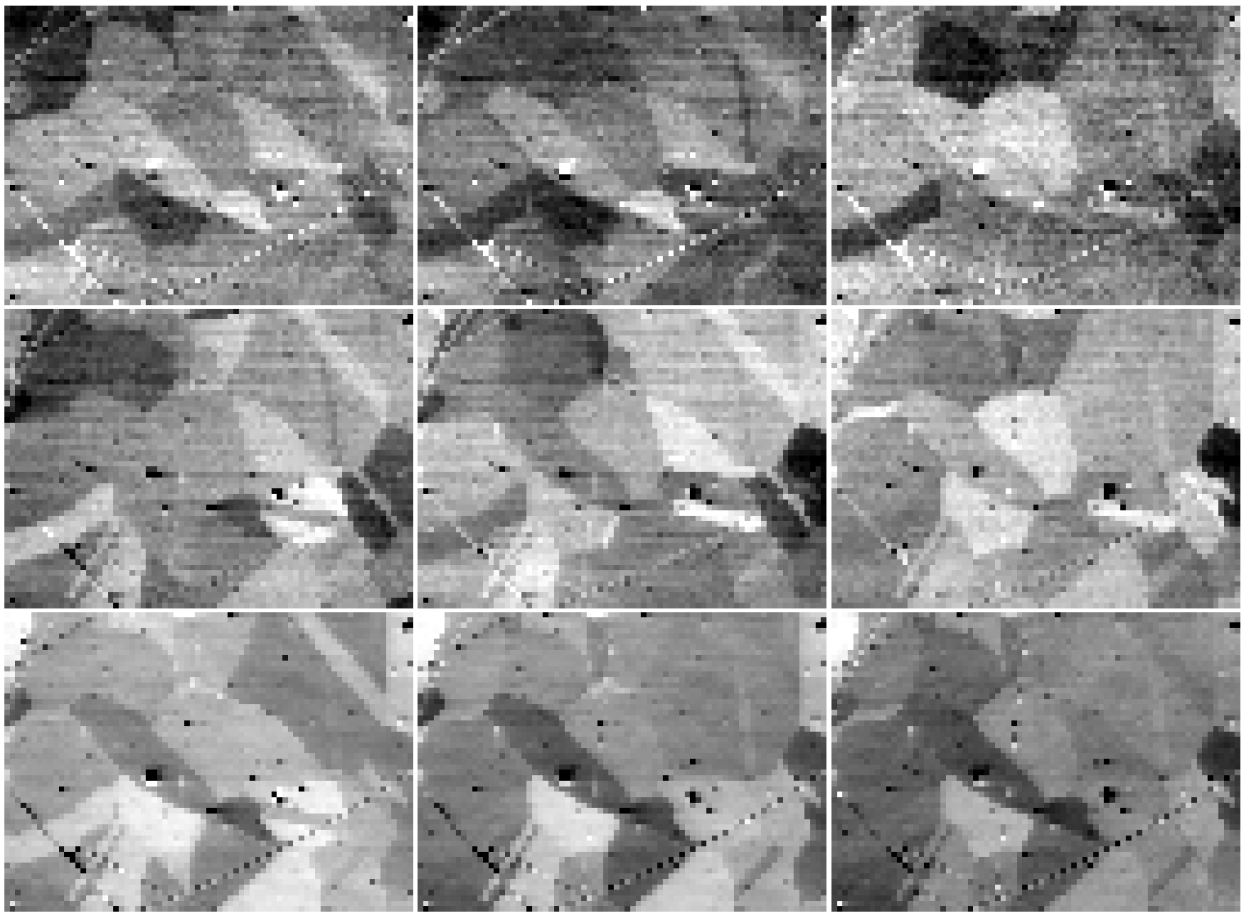
```
21321.0 84168.0
-7.962798e-07 1.0000005
```

[16]:
```
fig, ax = plt.subplots(nrows=3, ncols=3, figsize=(20, 20))
for idx in np.ndindex(vbse_imgs2.axes_manager.navigation_shape[::-1]):
    hs_idx = idx[::-1]
    ax[idx].imshow(vbse_imgs2.inav[hs_idx].data, cmap="gray")
```

(continues on next page)

(continued from previous page)

```
    ax[idx].axis("off")
fig.tight_layout(w_pad=0.5, h_pad=-24)
```



To obtain a rectangular ROI from the grid, we can use *VirtualBSEGenerator.roi_from_grid()*

[17]:
```
roi2 = vbse_imager.roi_from_grid((3, 3))  # (Row, column)
roi2
```

[17]:
```
RectangularROI(left=60, top=60, right=80, bottom=80)
```

### 1.2.3 Indexing

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

#### Hough indexing

In this tutorial, we will perform Hough/Radon indexing (HI) with PyEBSDIndex. We'll use a tiny dataset of recrystallized, polycrystalline nickel available with kikuchipy.

**Note**

PyEBSDIndex is an optional dependency of kikuchipy. It can be installed with both `pip` and `conda` (from `conda-forge`). See their installation instructions for how to install PyEBSDIndex.

Let's import necessary libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

from diffpy.structure import Atom, Lattice, Structure
from diffsims.crystallography import ReciprocalLatticeVector
import kikuchipy as kp
from orix import io, plot
from orix.crystal_map import Phase, PhaseList
from orix.vector import Vector3d


plt.rcParams.update({"font.size": 15, "lines.markersize": 15})
```

Load the dataset of (75, 55) nickel EBSD patterns of (60, 60) pixels with a step size of 1.5 um

[2]:
```
s = kp.data.nickel_ebsd_large(allow_download=True)
s
```

[2]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

#### Pre-indexing maps

We start by inspecting two indexing-independent maps showing microstructural features: a *virtual backscatter electron (VBSE) image* and an *image quality (IQ) map*. The VBSE image gives a qualitative orientation contrast and is created using the BSE yield on the detector. We should use the BSE yield of the raw unprocessed patterns. The IQ map correlates a higher image quality with sharpness of Kikuchi bands. We should thus use processed patterns here.

[3]:
```
vbse_imager = kp.imaging.VirtualBSEImager(s)
print(vbse_imager.grid_shape)
```

```
(5, 5)
```

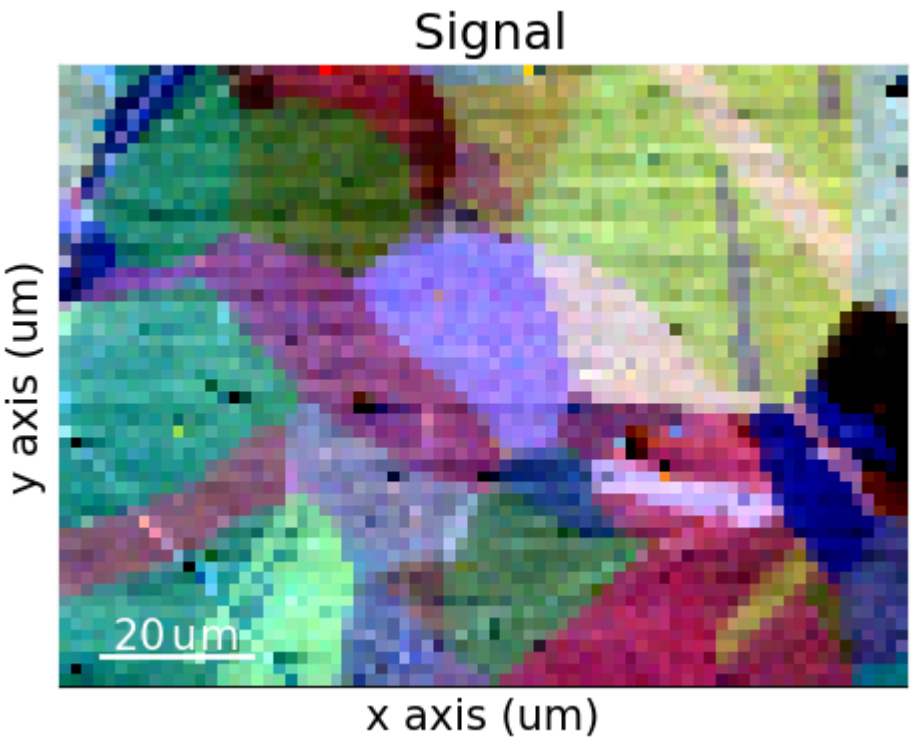
Get the VBSE image by coloring the three grid tiles in the center of the detector red, green, and blue

[4]:
```
maps_vbse_rgb = vbse_imager.get_rgb_image(r=(2, 1), g=(2, 2), b=(2, 3))
maps_vbse_rgb
```

[4]:
```
<VirtualBSEImage, title: , dimensions: (|75, 55)>
```

Plot the VBSE image

[5]:
```
maps_vbse_rgb.plot()
```




The orientation contrast shows that the region of interest covers 20-30 grains. Several of the grains seem to contain annealing twins.

Enhance the Kikuchi bands by removing the static and dynamic background (see the *pattern processing tutorial* for details)

[6]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 102.39 ms
[########################################] | 100% Completed | 709.20 ms
```

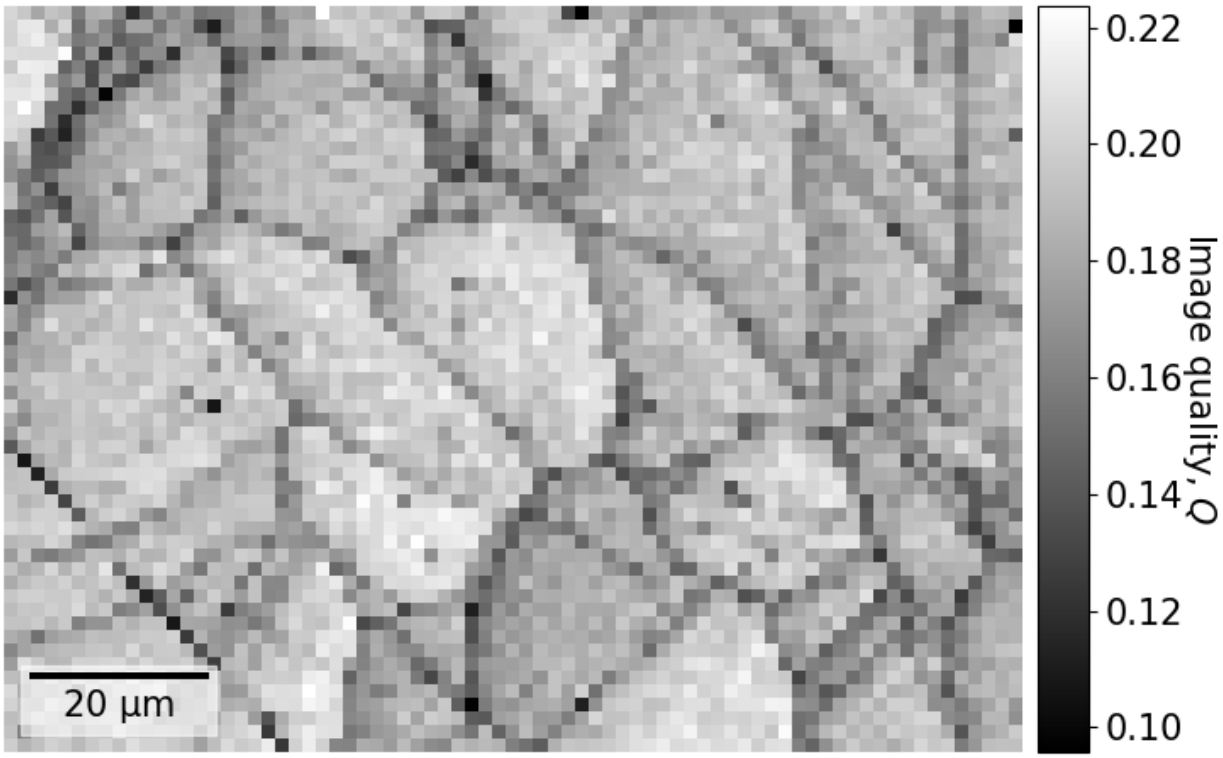
Get the IQ map

[7]:
```
maps_iq = s.get_image_quality()
```

```
[########################################] | 100% Completed | 404.04 ms
```

Plot the IQ map (using the CrystalMap.plot() method of the *EBSD.xmap* attribute)

[8]:
```
s.xmap.plot(
    maps_iq.ravel(),  # Array must be 1D
    cmap="gray",
    colorbar=True,
    colorbar_label="Image quality, $Q$",
    remove_padding=True,
)
```




We recognize the boundaries of grains and (presumably) the annealing twins seen in the VBSE image. There are some dark lines, e.g. to the lower and upper left, which look like scratches on the sample surface.

### Calibrate detector-sample geometry

Indexing requires knowledge of the position of the sample with respect to the detector. The detector-sample geometry is described by the projection or pattern center (PC) and the tilts of the detector and the sample (see the *reference frames tutorial* for all conventions). We assume the tilts are known and are thus required input. We will estimate the PC. We do so by optimizing an initial guess of the PC (obtained from similar experiments on the same microscope) using a few selected patterns.

All detector-sample geometry parameters are conveniently stored in an *EBSDDetector*

[9]:
```
sig_shape = s.axes_manager.signal_shape[::-1]  # Make (rows, columns)
det = kp.detectors.EBSDDetector(sig_shape, sample_tilt=70)
det
```

[9]:
```
EBSDDetector (60, 60), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.5, 0.5, 0.5)
```

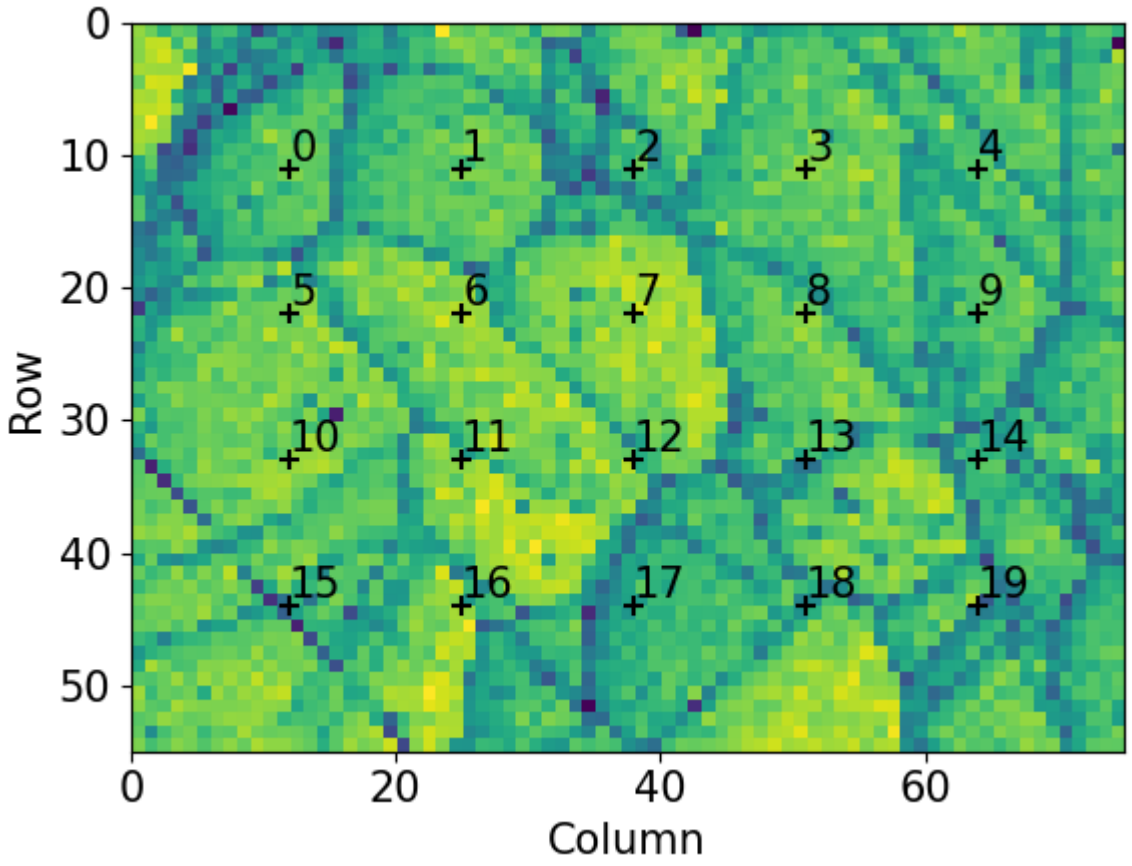
Extract selected patterns from the full dataset. The patterns should be spread out evenly in a map grid to prevent the estimation being biased by diffraction from particular grains or areas of the sample

[10]:
```
grid_shape = (5, 4)
s_grid, idx = s.extract_grid(grid_shape, return_indices=True)
s_grid
```

[10]:
```
<EBSD, title: patterns Scan 1, dimensions: (5, 4|60, 60)>
```

Plot the grid from where the patterns are extracted on the IQ map

[11]:
```
nav_shape = s.axes_manager.navigation_shape[::-1]

kp.draw.plot_pattern_positions_in_map(
    rc=idx.reshape(2, -1).T,  # Shape (n patterns, 2)
    roi_shape=nav_shape,  # Or maps_iq.shape
    roi_image=maps_iq,
)
```




We optimize one PC per pattern in this grid using *EBSD.hough_indexing_optimize_pc()*. The method calls the `PyEBSDIndex` function pcopt.optimize() internally. Hough indexing with `PyEBSDIndex` requires the use of an EBSDIndexer. The indexer stores the list of candidate phases, detector information, and indexing parameters such as the resolution of the Hough transform and the number of bands to use for orientation estimation. We could create this indexer from scratch. Another approach is to get it from an `EBSDDetector` via *EBSDDetector.get_indexer()*. This method requires a PhaseList.

We can optionally pass in a list of reflectors per phase (either directly the {hkl} or ReciprocalLatticeVector). The strongest reflectors (bands) for a phase are most likely to be detected in the Radon transform for indexing. Our reflector list should ideally contain these bands. We also need to make sure that our reflector list has enough bands for

consistent indexing. This is especially important for multi-phase indexing. We can build up a suitable reflector list with `ReciprocalLatticeVector`; see e.g. the tutorial on *kinematical simulations* for how to do this for nickel (point group m-3m), the tetragonal sigma phase in steels (4/mmm), and an hexagonal silicon carbide phase (6mm).

[12]:
```
phase_list = PhaseList(
    Phase(
        name="ni",
        space_group=225,
        structure=Structure(
            lattice=Lattice(3.5236, 3.5236, 3.5236, 90, 90, 90),
            atoms=[Atom("Ni", [0, 0, 0])],
        ),
    ),
)
phase_list
```

[12]:
```
Id  Name  Space group  Point group  Proper point group     Color
 0    ni        Fm-3m         m-3m                 432  tab:blue
```

[13]:
```
indexer = det.get_indexer(
    phase_list,
    [[1, 1, 1], [2, 0, 0], [2, 2, 0], [3, 1, 1]],
    nBands=10,
    tSigma=2,
    rSigma=2
)

print(indexer.vendor)
print(indexer.sampleTilt)
print(indexer.camElev)
print(indexer.PC)

print(indexer.phaselist[0].latticeparameter)
print(indexer.phaselist[0].polefamilies)
```

```
KIKUCHIPY
70
0
[0.5 0.5 0.5]
[ 3.5236  3.5236  3.5236 90.     90.     90.    ]
[[2 0 0]
 [2 2 0]
 [1 1 1]
 [3 1 1]]
```

We overwrote the defaults of some of the Hough indexing parameters to use values suggested in PyEBSDIndex' Radon indexing tutorial:

- tSigma: size of the Gaussian kernel in the $\theta$ direction in the Radon transform $(\rho, \theta)$
- rSigma: size of the Gaussian kernel in the $\rho$ direction

We also set the number of bands to search for in the Radon transform to 10. This is because testing has shown that the default number of 9 can be too few bands for some of the patterns in this dataset.

Now, we can optimize the PCs for each pattern in the extracted grid. (We will "overwrite" the existing detector variable.) We use the particle swarm optimization (PSO) algorithm implemented in `PyEBSDIndex` The search limit range is set to

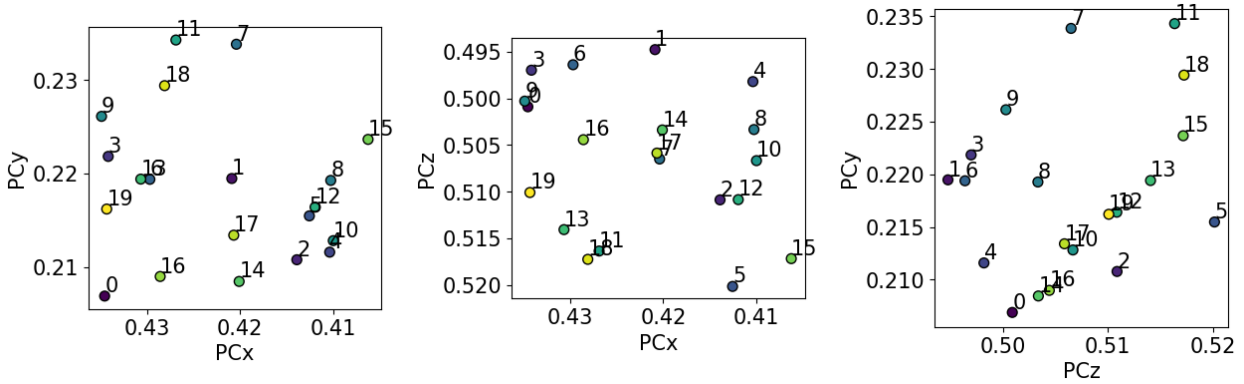
+/- 0.05 (default is 0.2) since we are sufficiently confident that all PCs are within this range; if we were not, we should increase the search limit.

[14]:
```
det = s_grid.hough_indexing_optimize_pc(
    pc0=[0.42, 0.22, 0.50],  # Initial guess based on previous experiments
    indexer=indexer,
    batch=True,
    method="PSO",
    search_limit=0.05,
)

# Print mean and standard deviation
print(det.pc_flattened.mean(axis=0))
print(det.pc_flattened.std(0))
```

```
PC found: [**********] 20/20  global best:1.15  PC opt:[0.4343 0.2162 0.5101]
[0.42194278 0.21838479 0.50671947]
[0.00946586 0.00779942 0.00746256]
```

Plot the PCs

[15]:
```
det.plot_pc("scatter", s=50, annotate=True)
```

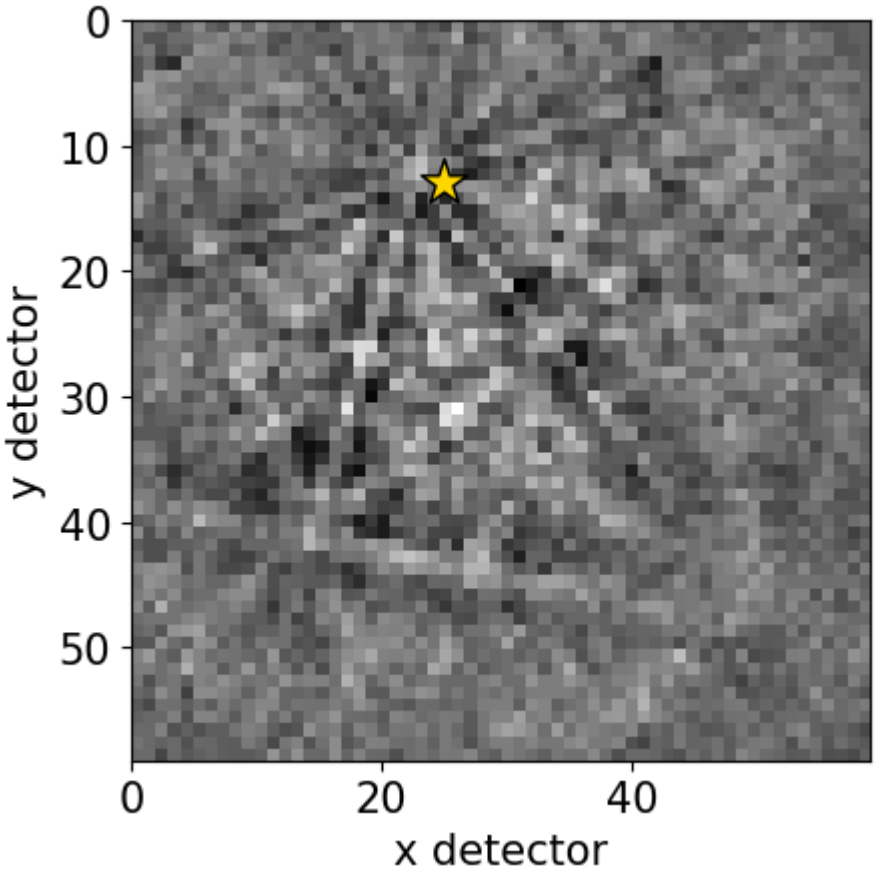



The values do not order nicely in the grid they were extracted from… This is not surprising since they are only (60, 60) pixels wide! Fortunately, the spread is small, and, since the region of interest covers such as small area, we can use the mean PC for indexing.

[16]:
```
det.pc = det.pc_average
```

We can check the position of the mean PC on the detector before using it

[17]:
```
det.plot(pattern=s_grid.inav[0, 0].data)
```




### Perform indexing

We will index all patterns with this PC calibration. We get a new indexer from the detector with the average PC as determined from the optimization above

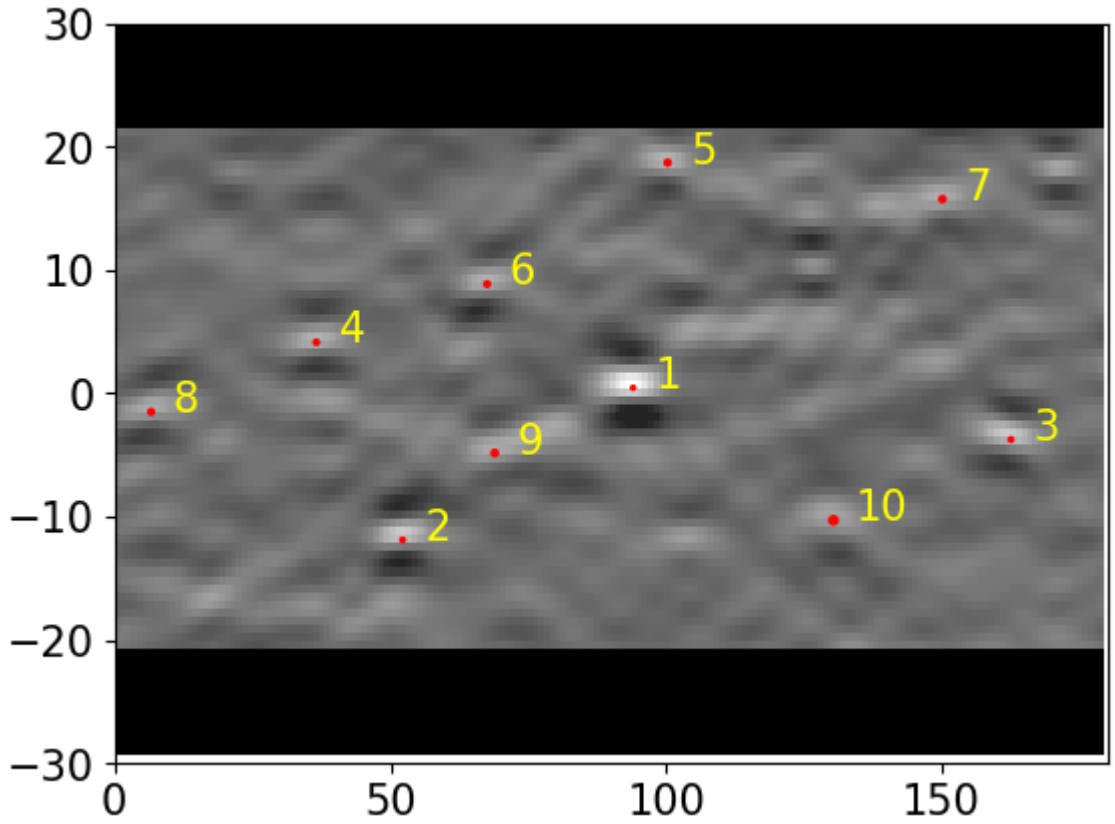
[18]:
```
indexer = det.get_indexer(
    phase_list,
    [[1, 1, 1], [2, 0, 0], [2, 2, 0], [3, 1, 1]],
    nBands=10,
    tSigma=2,
    rSigma=2
)
indexer.PC
```

[18]:
```
array([0.42194278, 0.21838479, 0.50671947])
```

Hough indexing is done with *EBSD.hough_indexing()*. Since we ask `PyEBSDIndex` to be "verbose" when reporting on the indexing progress, a figure with the last pattern and its Radon transform is shown. We need to pass the phase list again to `EBSD.hough_indexing()` (the indexer does not keep all phase information stored in the list [atoms are lost]) for the phases to be described correctly in the returned CrystalMap.

[19]:
```
xmap = s.hough_indexing(phase_list=phase_list, indexer=indexer, verbose=2)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: False
  Projection center (Bruker): (0.4219, 0.2184, 0.5067)
  Indexing 4125 pattern(s) in 8 chunk(s)
Radon Time: 6.364346714000931
Convolution Time: 4.434683096998924
Peak ID Time: 3.1442413160020806
Band Label Time: 1.0708521209999162
Total Band Find Time: 15.015167161998761
Band Vote Time:  3.675401554999553
  Indexing speed: 220.35604 patterns/s
```




[20]:
```
xmap
```

[20]:
```
Phase   Orientations  Name  Space group  Point group  Proper point group     Color
    0  4125 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: fit, cm, pq, nmatch
Scan unit: um
```

We see that all points were indexed as nickel.

Using orix, the indexing results can be exported to an HDF5 file or a text file (.ang) importable by software such as MTEX or other commercial software

[21]:
```
#io.save("xmap_ni.h5", xmap)
#io.save(
```

(continues on next page)

(continued from previous page)

```
#     "xmap_ni.ang",
#     xmap,
#     image_quality_prop="pq",
#     confidence_index_prop="cm",
#     extra_prop="nmatch",
#)
```

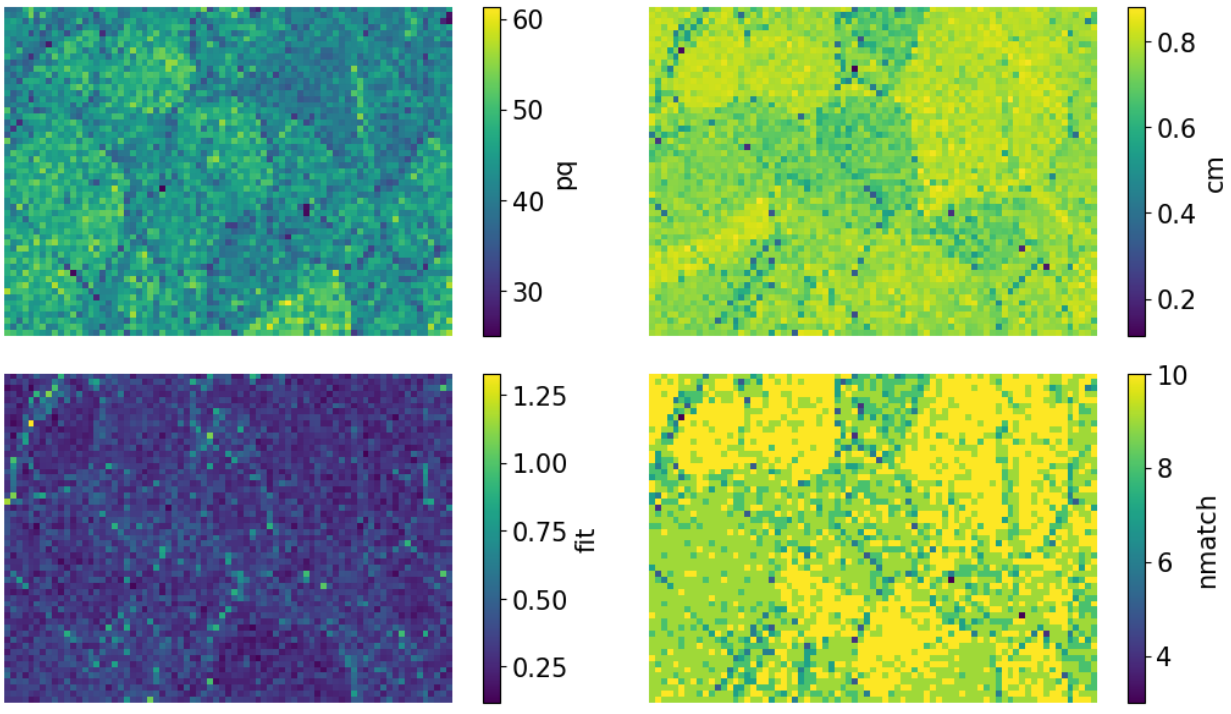
Before analyzing the returned orientations, however, we should validate our results.

### Validate indexing results

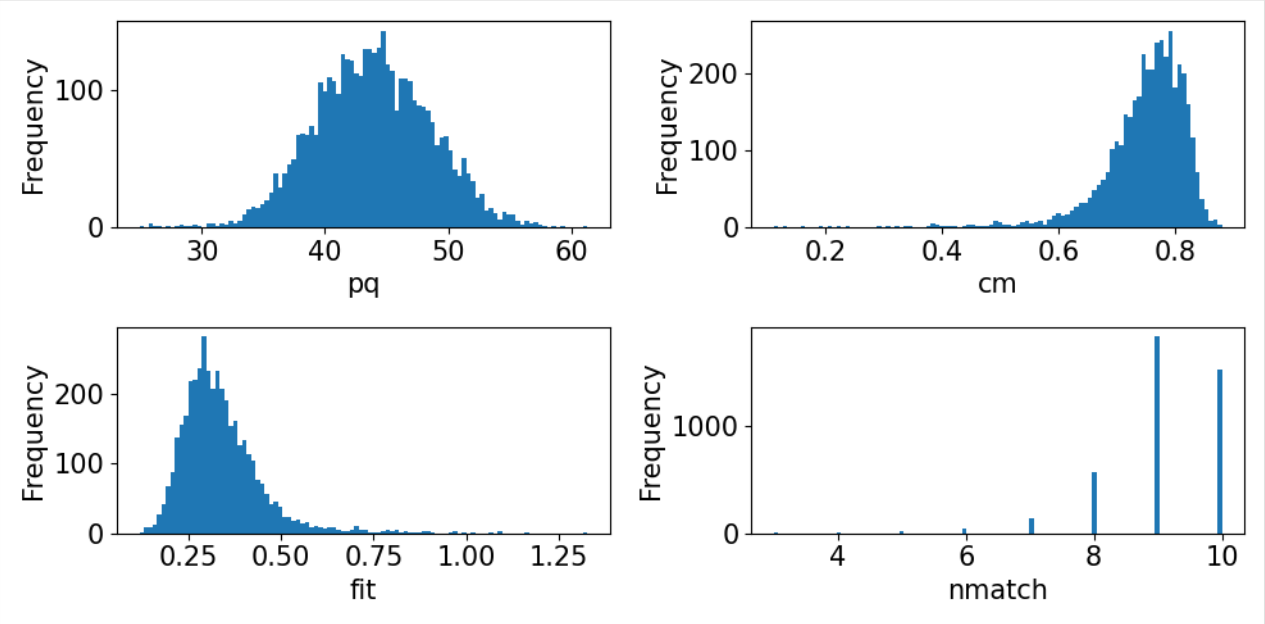
We validate our results by inspecting indexing quality metrics, inverse pole figure (IPF) maps, and comparing geometrical simulations to the experimental patterns.

Plot quality metrics

[22]:
```
aspect_ratio = xmap.shape[1] / xmap.shape[0]
figsize = (8 * aspect_ratio, 4.5 * aspect_ratio)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=figsize, layout="tight")
for ax, to_plot in zip(axes.ravel(), ["pq", "cm", "fit", "nmatch"]):
    im = ax.imshow(xmap.get_map_data(to_plot))
    fig.colorbar(im, ax=ax, label=to_plot)
    ax.axis("off")
```




[23]:
```
fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(10, 5), layout="tight")
for ax, to_plot in zip(axes.ravel(), ["pq", "cm", "fit", "nmatch"]):
    ax.hist(xmap.prop[to_plot], bins=100)
    ax.set(xlabel=to_plot, ylabel="Frequency");
```




The pattern quality (PQ) and confidence metric (CM) maps show little variation across the sample. The most important map here is that of the pattern fit, also known as the mean angular error/deviation (MAE/MAD): it shows the average angular deviation between the positions of each detected band to the closest "theoretical" band in the indexed solution. The pattern fit is below an OK fit of 1.4° for all patterns. We see that the highest (worst) fit is found in the upper left corner where we recognized some scratches in our IQ map. The final map (*nmatch*) shows that most detected bands were matched inside most of the grains, with as few as four on some grain boundaries and triple junctions. See `PyEBSDIndex`' Radon indexing tutorial for a complete explanation of all the indexing quality metrics.

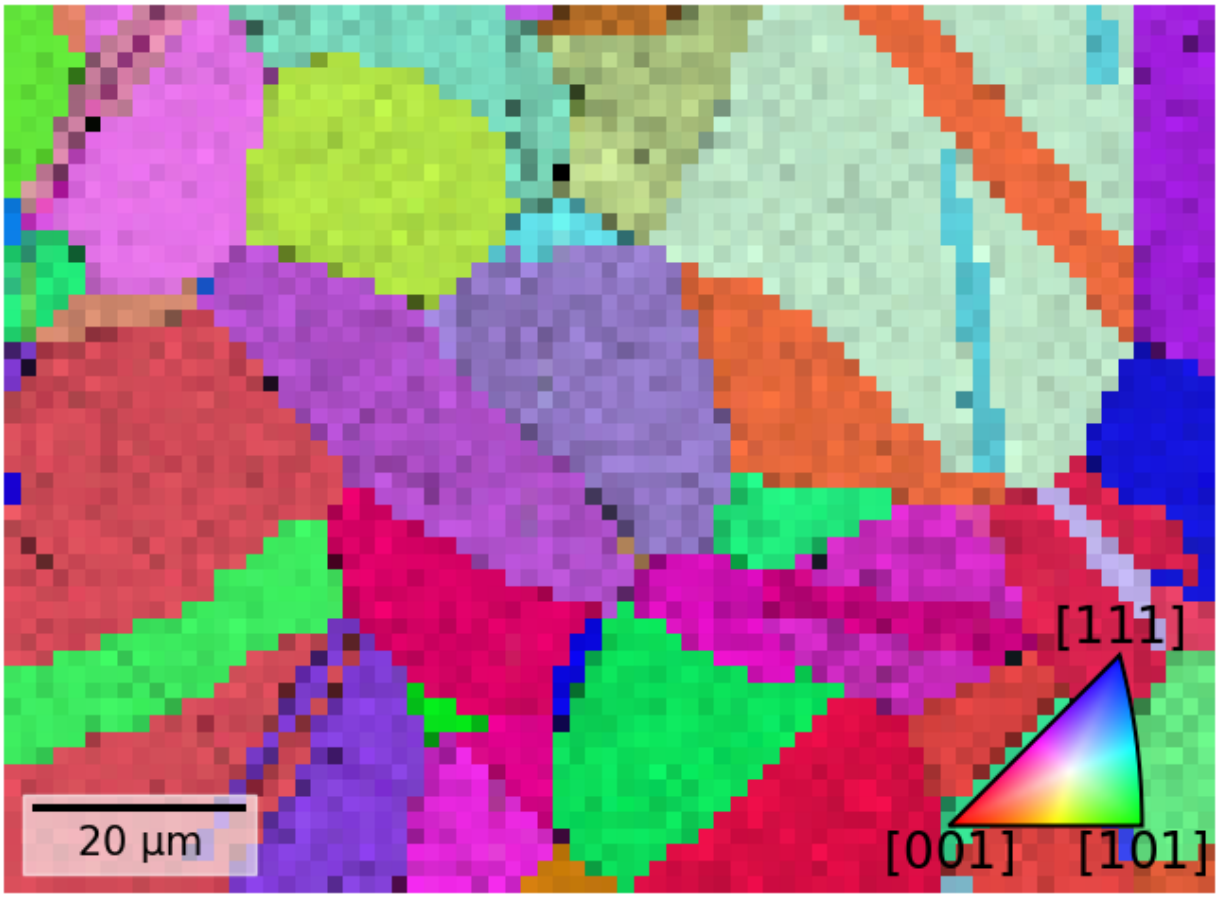
Create a color key to get IPF colors with

[24]:
```
v_ipf = Vector3d.xvector()
sym = xmap.phases[0].point_group

ckey = plot.IPFColorKeyTSL(sym, v_ipf)
ckey
```

[24]:
```
IPFColorKeyTSL, symmetry: m-3m, direction: [1 0 0]
```

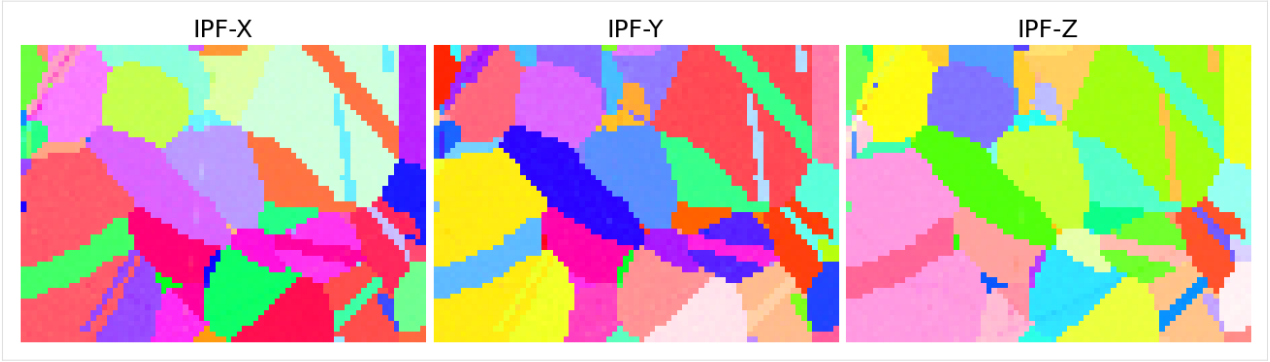
Each point is assigned a color based on which crystal direction $\langle uvw \rangle$ points in a certain sample direction. Let's plot the IPF-X map with the confidence metric map overlayed

[25]:
```
rgb_x = ckey.orientation2color(xmap.rotations)
fig = xmap.plot(rgb_x, overlay="cm", remove_padding=True, return_figure=True)

# Place color key in bottom right corner, coordinates are [left, bottom, width, height]
ax_ckey = fig.add_axes(
    [0.76, 0.08, 0.2, 0.2], projection="ipf", symmetry=sym
)
ax_ckey.plot_ipf_color_key(show_title=False)
ax_ckey.patch.set_facecolor("None")
```




Let's also plot the three IPF maps for sample **X**, **Y**, and **Z** side by side

[26]:
```
directions = Vector3d(((1, 0, 0), (0, 1, 0), (0, 0, 1)))
n = directions.size

figsize = (4 * n * aspect_ratio, n * aspect_ratio)
fig, ax = plt.subplots(ncols=n, figsize=figsize)
for i, title in zip(range(n), ["X", "Y", "Z"]):
    ckey.direction = directions[i]
    rgb = ckey.orientation2color(xmap.rotations)
    ax[i].imshow(rgb.reshape(xmap.shape + (3,)))
    ax[i].set_title(f"IPF-{title}")
    ax[i].axis("off")
fig.subplots_adjust(wspace=0.02)
```




The IPF maps show grains and twins as we expected from the VBSE image and IQ map obtained before indexing.

Plot geometrical simulations on top of the experimental patterns (see the *geometrical simulations tutorial* for details)

[27]:
```
ref = ReciprocalLatticeVector(
    phase=xmap.phases[0], hkl=[[1, 1, 1], [2, 0, 0], [2, 2, 0], [3, 1, 1]]
)
ref = ref.symmetrise()
simulator = kp.simulations.KikuchiPatternSimulator(ref)
sim = simulator.on_detector(det, xmap.rotations.reshape(*xmap.shape))
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.36 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.94 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.83 ms
```

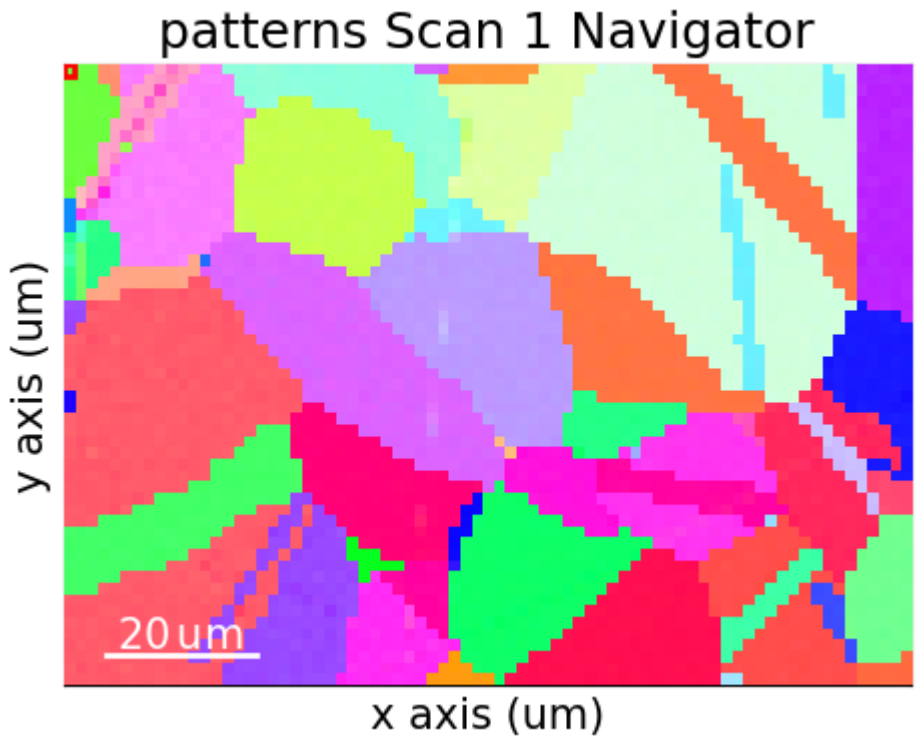
Add markers to EBSD signal

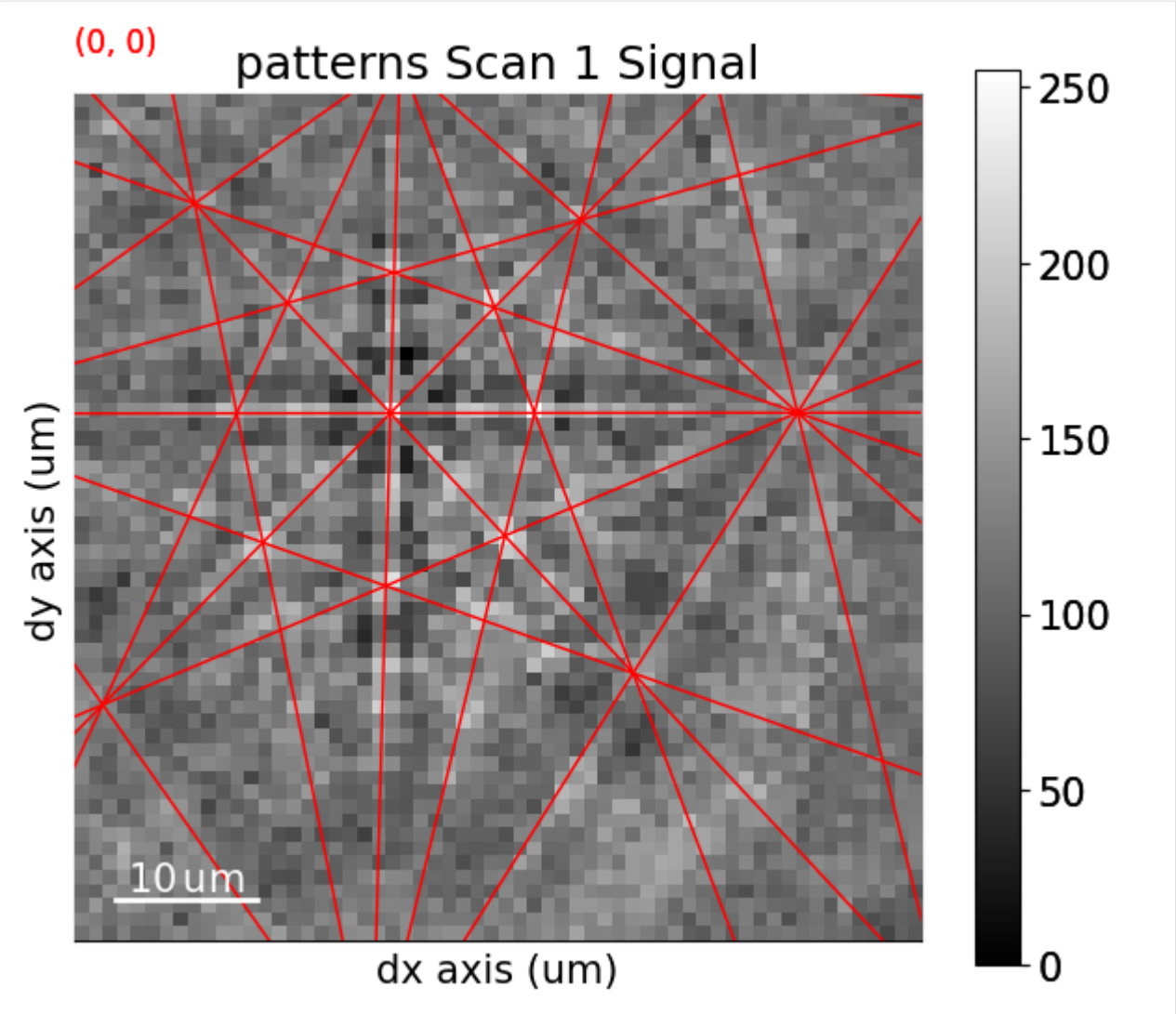
[28]:
```
# To remove existing markers
# del s.metadata.Markers

markers = sim.as_markers()
s.add_marker(markers, plot_marker=False, permanent=True)
```

Navigate patterns with simulations in the IPF-X map (see the *visualization tutorial* for details)

[29]:
```
maps_nav_rgb = kp.draw.get_rgb_navigator(rgb_x.reshape(xmap.shape + (3,)))
```

[30]:
```
s.plot(maps_nav_rgb)
```


patterns Scan 1 Navigator
y axis (um)
20 um
x axis (um)




### What's next?

If we require a high angular resolution, we might achieve this via refining returned orientations using dynamical simulations. See the *refinement section* of the pattern matching tutorial for how to do this.

If our validation shows unexpectedly bad results, there are a few key parameters we should look into:

- Phase information: lattice parameters or space group symmetry might be incorrect.
- Detector-sample geometry: sample and detector tilts, but most importantly, the projection center!
- Hough indexing parameters:
  - Reflector list: perhaps the default list or the list passed in by us contain too few reflectors, or they are not the brightest bands for a particular phase?
  - The Gaussian kernel widths for parameters $(\rho, \theta)$ in the Radon transform can affect which bands are detected (wider or narrower). A parameter search might be benefitial.

If we want to speed up Hough indexing, we could try to use `PyEBSDIndex`' multi-threading functionality ([see the Radon indexing demo for details).

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

## Pattern matching

Crystal orientations can be determined from experimental EBSD patterns by matching them to simulated patterns of known phases and orientations, see e.g. [Chen *et al.*, 2015], [Nolze *et al.*, 2016], [Foden *et al.*, 2019].

In this tutorial, we perform *dictionary indexing* (DI) of a small nickel EBSD dataset. We use a dynamically simulated nickel master pattern from EMsoft found in the *kikuchipy.data* module. The master pattern has relatively low pixel resolution. We generate the pattern dictionary from a uniform grid of orientations with a fixed projection center (PC) followng [Singh and De Graef, 2016]. The true orientation is likely to fall in between grid points. This means that the lowest angular accuracy obtainable with DI has is bounded by the grid sampling resolution. However, we can improve upon each orientation solution by searching the space in between the grid points. We do this by maximizing the similarity between experimental and simulated patterns using numerical optimization algorithms. This is here called *orientation refinement*. We could instead keep the orientations fixed and let the PC parameters deviate from their fixed values used in the dictionary, here called *projection center refinement*. Finally, we can also refine both at the same time, here called *orientation and projection center refinement*. The need for orientation or orientation/PC refinement is discussed by e.g. [Singh *et al.*, 2017], [Winkelmann *et al.*, 2020], and [Pang *et al.*, 2020].

The term *pattern matching* is here used for the combined approach of DI followed by refinement.

Before we can generate a dictionary of simulated patterns, we need a master pattern containing all possible scattering vectors for a candidate phase. This can be simulated using EMsoft ([Callahan and De Graef, 2013] and [Jackson *et al.*, 2014]) and then read into kikuchipy.

First, we import libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import tempfile

import matplotlib.pyplot as plt
import numpy as np

import hyperspy.api as hs
import kikuchipy as kp
from orix import sampling, plot, io
from orix.vector import Vector3d


plt.rcParams.update({"figure.facecolor": "w", "font.size": 15})
```

Load the small experimental nickel test data

[2]:
```
# Use kp.load("data.h5") to load your own data
s = kp.data.nickel_ebsd_large(allow_download=True)  # External download
s
```

[2]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

To obtain a good match, we must increase the signal-to-noise ratio. In this pattern matching analysis, the Kikuchi bands are considered the signal. The angle-dependent backscatter intensity, along with unwanted detector effects, are considered to be noise. See the *pattern processing tutorial* for further details.

[3]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 100.76 ms
[########################################] | 100% Completed | 605.37 ms
```

**Note**

DI is computationally intensive and takes in general a long time to run due to all the pattern comparisons being done. To maintain an acceptable memory usage and be done within reasonable time, it is recommended to write processed patterns to an HDF5 file for quick reading during DI.

[4]:
```
# s.save("pattern_static_dynamic.h5")
```

### Dictionary indexing

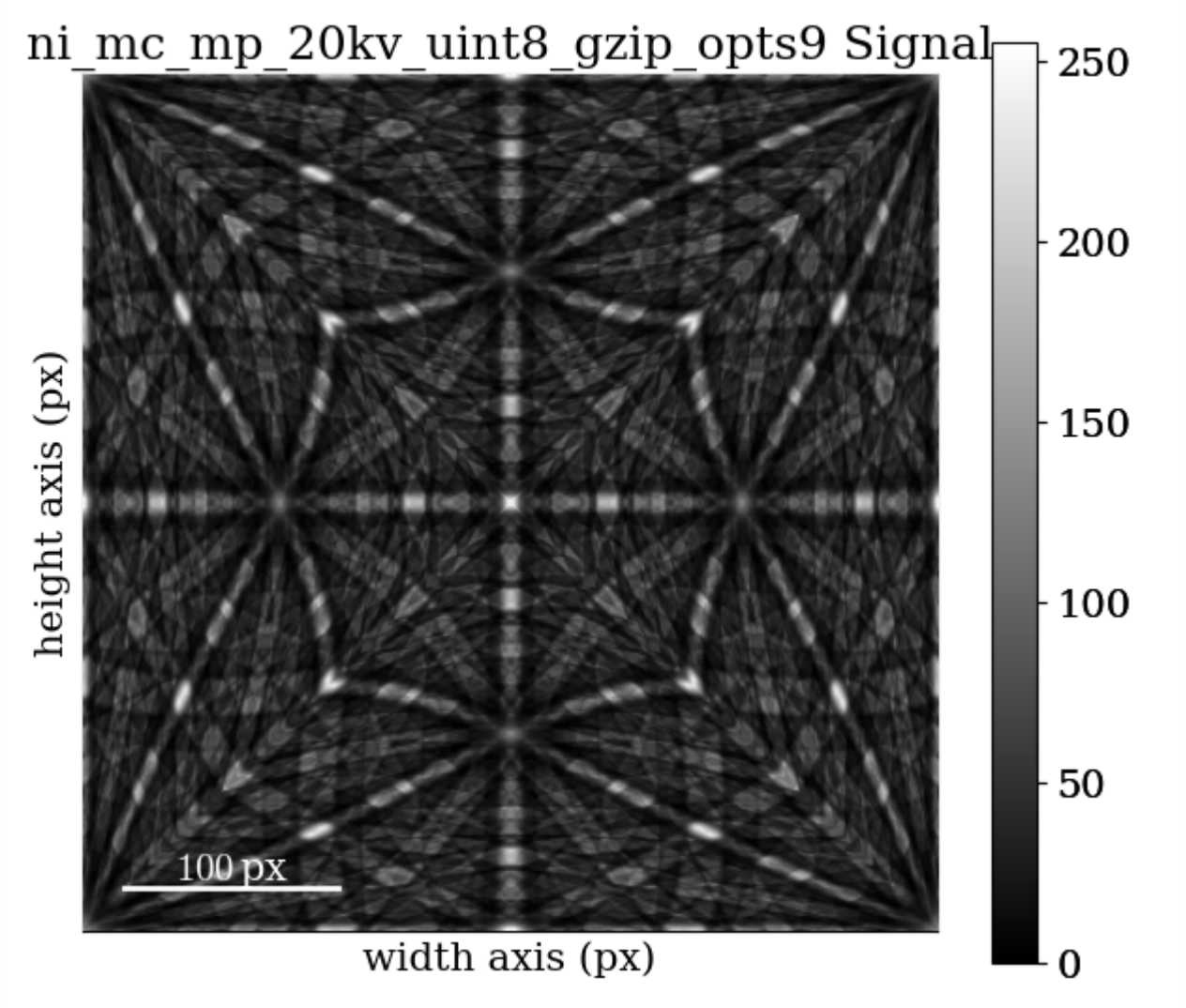
#### Load a master pattern

Next, we load a dynamically simulated nickel master pattern for an accelerating voltage of 20 keV, generated with EMsoft. The master pattern is in the upper hemisphere projection of the square Lambert projection.

[5]:
```
energy = 20
mp = kp.data.nickel_ebsd_master_pattern_small(
    projection="lambert", energy=energy
)
mp
```

```
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (|401, 401)>
```

[6]:
```
mp.plot()
```




The nickel phase information, specifically the crystal symmetry, asymmetric atom positions, and crystal lattice, is conveniently stored in an orix.crystal_map.Phase

[7]:
```
ni = mp.phase

print(ni)
print(ni.structure)
```

```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
lattice=Lattice(a=0.35236, b=0.35236, c=0.35236, alpha=90, beta=90, gamma=90)
28   0.000000 0.000000 0.000000 1.0000
```

### Sample orientation space

If we don't know anything about the possible crystal orientations in our sample, the safest thing to do is to generate a dictionary of orientations uniformly distributed in a candidate phase's orientation space. To achieve this, we sample the Rodrigues Fundamental Zone (RFZ) of the proper point group *432* using cubochoric sampling [Singh and De Graef, 2016] available in orix.sampling.get_sample_fundamental(). We can choose the average disorientation between orientations, the "resolution", of this sampling. Here, we will use a rather low resolution of 3°.

[8]:
```
Gr = sampling.get_sample_fundamental(
    method="cubochoric", resolution=3, point_group=ni.point_group
)
Gr
```

[8]:
```
Rotation (30443,)
[[ 0.8562 -0.3474 -0.3474 -0.1595]
 [ 0.8562 -0.3511 -0.3511 -0.1425]
 [ 0.8562 -0.3544 -0.3544 -0.1252]
 ...
 [ 0.8562  0.3544  0.3544  0.1252]
 [ 0.8562  0.3511  0.3511  0.1425]
 [ 0.8562  0.3474  0.3474  0.1595]]
```

This sampling resulted in about 30 000 crystal orientations. See the orix documentation on orientation sampling for further details and options for orientation sampling.

---

**Note**

An average disorientation of 3° results in a course sampling of orientation space; a lower average disorientation should be used for experimental work.

---

### Define the detector-sample geometry

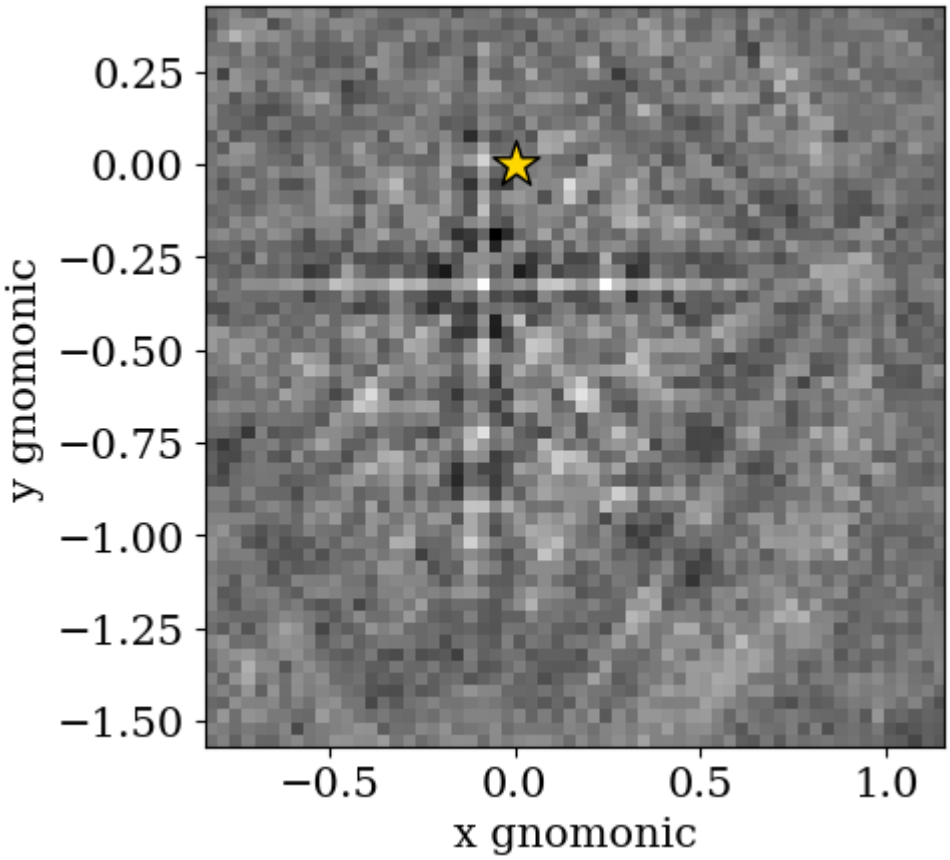
Now that we have our master pattern and crystal orientations, we need to describe the EBSD detector's position with respect to the sample (interaction volume). This ensures that projecting parts of the master pattern onto our detector yields dynamically simulated patterns resembling our experimental ones. See the *reference frames* tutorial and the *EBSDDetector* class for further details.

[9]:
```
det = kp.detectors.EBSDDetector(
    shape=s.axes_manager.signal_shape[::-1],
    pc=[0.4198, 0.2136, 0.5015],
    sample_tilt=70,
)
det
```

[9]:
```
EBSDDetector (60, 60), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.42, 0.214, 0.
↪501)
```

Let's double check the projection/pattern center (PC) position on the detector using *plot()*

[10]:
```
det.plot(coordinates="gnomonic", pattern=s.inav[0, 0].data)
```




### Generate dictionary

Now we're ready to generate our dictionary of simulated patterns. We do so by projecting parts of the master pattern onto our detector for all sampled orientations using the *get_patterns()* method. The method assumes the crystal orientations are defined with respect to the EDAX TSL sample reference frame RD-TD-ND.

---

**Note**

It is in general adviced to not compute the dictionary immediately, but let the dictionary indexing method handle this, by passing `compute=False`. This will return a `LazyEBSD` signal, with the dictionary patterns as a Dask array.

**Tip:** If `compute=False`, it is recommended to control the number of patterns in each chunk by passing `chunk_shape=1000` if we want 1 000 patterns per chunk. Alternatively, `chunk_shape=rot.size // 30` gives 30-31 chunks. If we at the same time do *not* pass `n_per_iteration` to `dictionary_indexing()`, our here specified chunk size will be the number of simulated patterns matched per iteration, which should give a faster and less memory intensive indexing!

---

[11]:
```
sim = mp.get_patterns(
    rotations=Gr,
    detector=det,
    energy=energy,
    dtype_out=np.float32,
```

(continues on next page)

(continued from previous page)

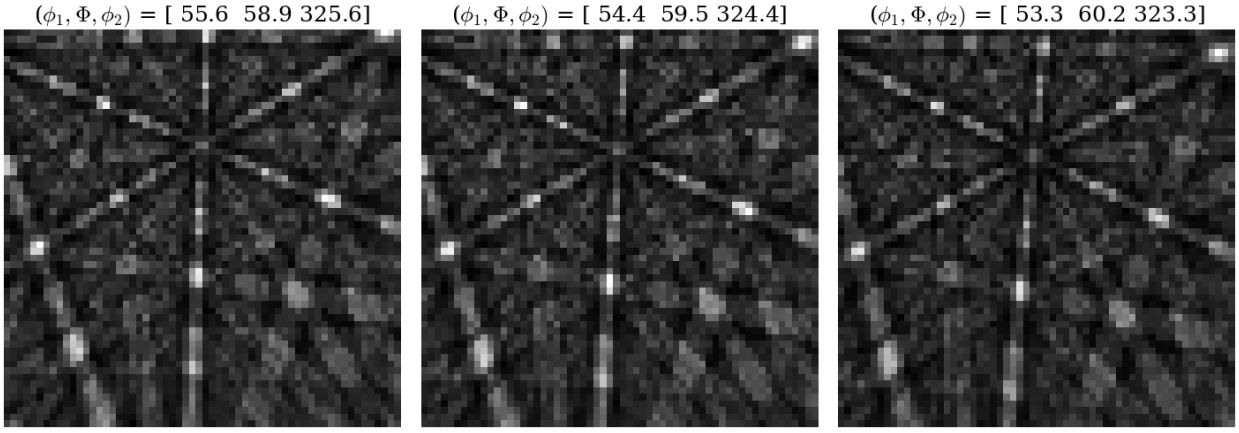
```
    compute=True,
)
sim
```

```
[########################################] | 100% Completed | 3.07 ss
```

[11]:
```
<EBSD, title: , dimensions: (30443|60, 60)>
```

Let's inspect the three first of the simulated patterns

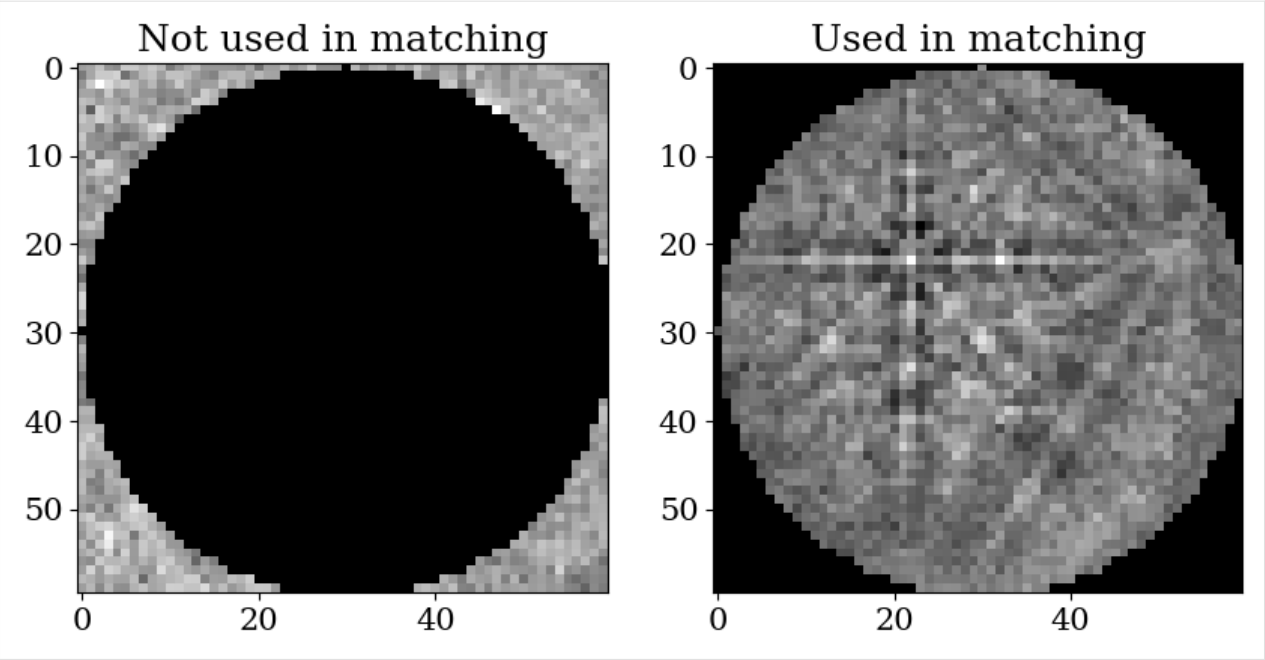
[12]:
```
# sim.plot()  # Plot the patterns with a navigator for easy inspection
fig, ax = plt.subplots(ncols=3, figsize=(18, 6))
for i in range(3):
    ax[i].imshow(sim.inav[i].data, cmap="gray")
    euler = sim.xmap[i].rotations.to_euler(degrees=True)[0]
    ax[i].set_title(
        f"($\phi_1, \Phi, \phi_2)$ = {np.array_str(euler, precision=1)}"
    )
    ax[i].axis("off")
fig.subplots_adjust(wspace=0.05)
```




### Perform indexing

The Kikuchi pattern signal is often weak towards the corners of the detector. In such cases, we can pass a signal mask to exclude the pixels where the mask values are `True`. This convention is in line with how NumPy, Dask, scikit-image etc. defines a mask. The mask can take whatever shape we want, as long as it is a boolean array of the detector shape.

[13]:
```
signal_mask = ~kp.filters.Window("circular", det.shape).astype(bool)

p = s.inav[0, 0].data
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))
ax[0].imshow(p * signal_mask, cmap="gray")
ax[0].set_title("Not used in matching")
ax[1].imshow(p * ~signal_mask, cmap="gray")
_ = ax[1].set_title("Used in matching");
```




Finally, let's use the *dictionary_indexing()* method to match the simulated patterns to our experimental patterns. We use the *zero-mean normalized cross correlation (NCC)* coefficient $r$ [Gonzalez and Woods, 2017] as our similarity metric (it is the default). Let's keep the 20 best matching orientations. A number of 4125 * 30 443 comparisons is quite small, which we can do in memory all at once. In cases where the number of comparisons is too big for our memory to handle, we should iterate over the dictionary of simulated patterns by passing the number of patterns per iteration. To demonstrate this, we do at least 10 iterations here. The results are returned as a orix.crystal_map.CrystalMap.

[14]:
```
xmap = s.dictionary_indexing(
    sim,
    metric="ncc",
    keep_n=20,
    n_per_iteration=sim.axes_manager.navigation_size // 10,
    signal_mask=signal_mask,
)
xmap
```

```
Dictionary indexing information:
  Phase name: ni
  Matching 4125 experimental pattern(s) to 30443 dictionary pattern(s)
  NormalizedCrossCorrelationMetric: float32, greater is better, rechunk: False,␣
↪navigation mask: False, signal mask: True
```

```
100%|----------------------------------| 11/11 [00:08<00:00,  1.25it/s]
```

```
  Indexing speed: 467.55826 patterns/s, 14233876.20971 comparisons/s
```

[14]:
```
Phase   Orientations  Name  Space group  Point group  Proper point group     Color
    0  4125 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: scores, simulation_indices
Scan unit: um
```

**Note**

Dictionary indexing of real world data sets takes a long time because the matching is computationally intensive. The

*dictionary_indexing()* method accepts parameters *n_per_iteration*, *rechunk*, and *dtype* that lets us control this behaviour to a certain extent. Be sure to take a look at the method's docstring for further details.

Check the average of the best matching score per pattern

[15]:
```
print(xmap.scores[:, 0].mean())
```

```
0.3227371
```

The *normalized dot product* can be used instead of the NCC by passing `metric="ndp"`. A custom metric can be used instead, by creating a class which inherits from the abstract class *SimilarityMetric*.

The results can be exported to an HDF5 file re-readable by orix, or an .ang file readable by MTEX and some commercial packages

[16]:
```
temp_dir = tempfile.mkdtemp() + "/"
io.save(temp_dir + "ni.h5", xmap)
io.save(temp_dir + "ni.ang", xmap)
```

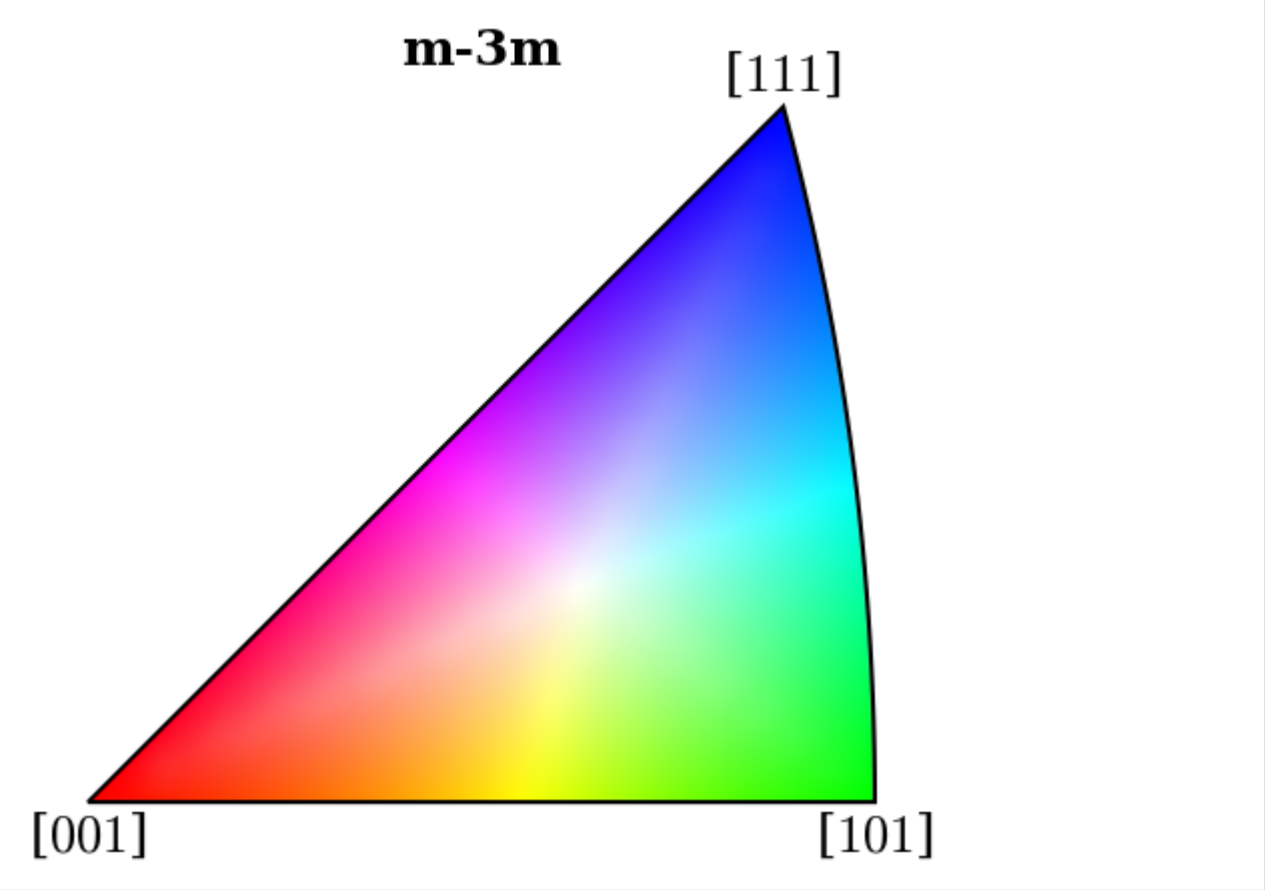
### Validate indexing results

With the orix library we can plot inverse pole figures (IPFs), color crystal directions to produce inverse pole figure (IPF) maps, and more. This is useful for quick validation of our results before quantitative orientation analysis. At the moment, if we want to analyze the results in terms of reconstructed grains, textures from orientation density functions, or similar, we have to use other software, such as MTEX or DREAM.3D.

Let's generate an IPF color key and plot it

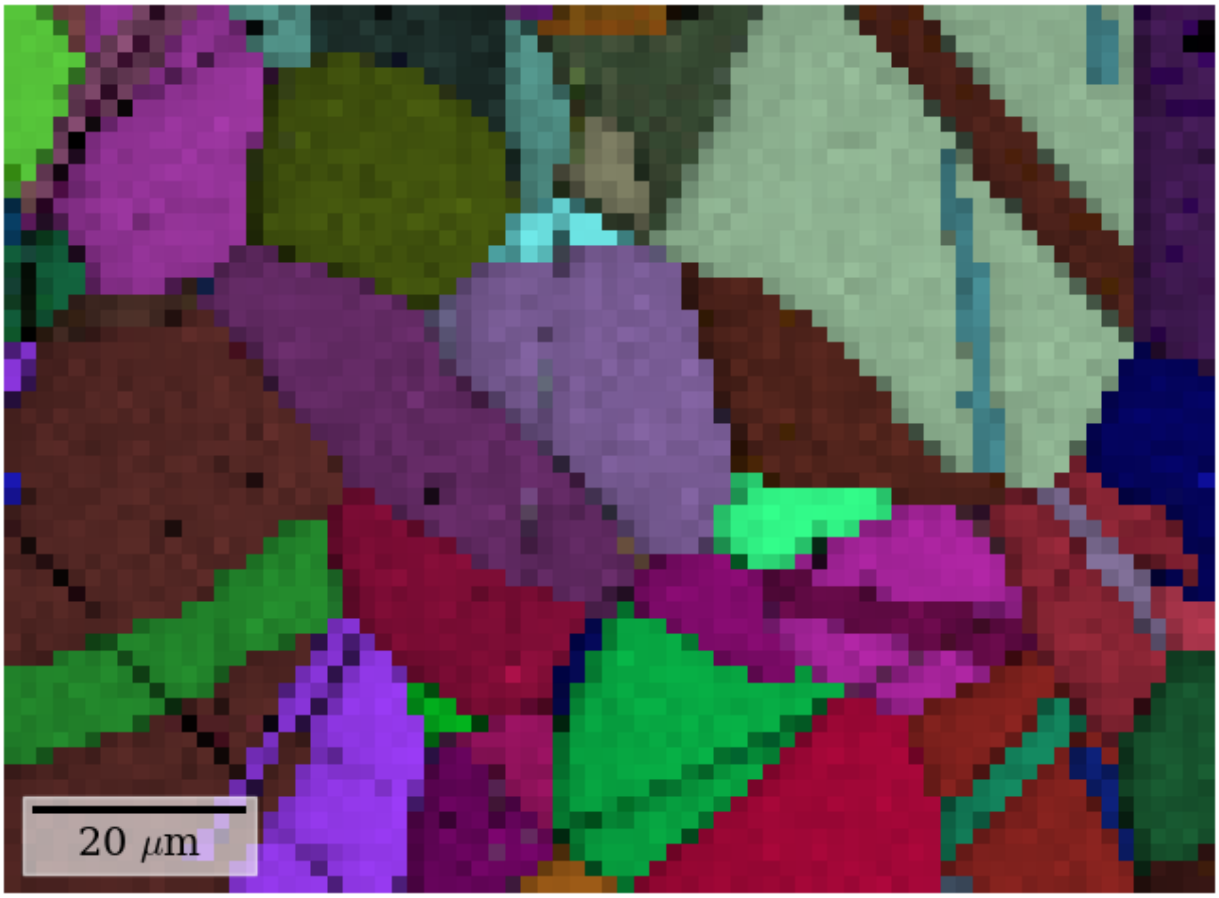
[17]:
```
pg = ni.point_group

ckey_m3m = plot.IPFColorKeyTSL(pg, direction=Vector3d.xvector())
print(ckey_m3m)
ckey_m3m.plot()
```

```
IPFColorKeyTSL, symmetry: m-3m, direction: [1 0 0]
```



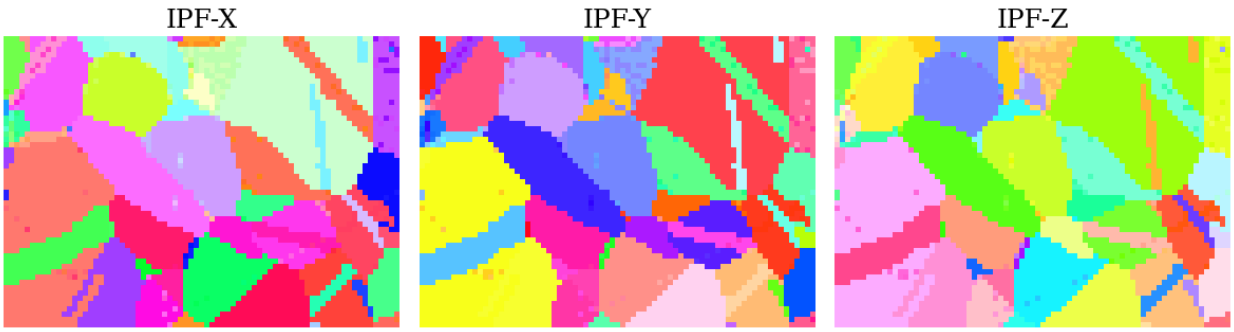

With this color key, we can color orientations according to which crystal directions the sample direction $X$ points in in every map pixel (overlaying the NCC scores of the best match)

[18]:
```
rgb_x = ckey_m3m.orientation2color(xmap.orientations)

xmap.plot(rgb_x, overlay=xmap.scores[:, 0], remove_padding=True)
```




With a few more lines, we can plot IPF maps for X, Y, and Z

[19]:
```
G = xmap.orientations
v_sample = Vector3d(((1, 0, 0), (0, 1, 0), (0, 0, 1)))

fig, axes = plt.subplots(figsize=(15, 5), ncols=3)
for ax, v, title in zip(axes, v_sample, ("X", "Y", "Z")):
    ckey_m3m.direction = v
    rgb = ckey_m3m.orientation2color(G).reshape(xmap.shape + (3,))
    ax.imshow(rgb)
    ax.axis("off")
    ax.set(title=f"IPF-{title}")
fig.subplots_adjust(wspace=0.05)
```



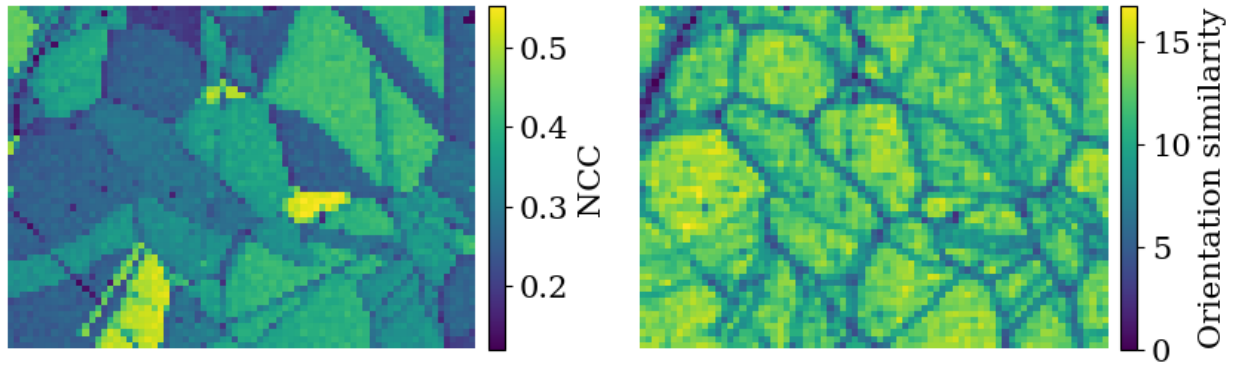

The sample is recrystallized nickel, so we expect a continuous color within grains, except for twinning grains. The orientation maps are mostly in line with our expectation. Some grains have a scatter of similar colors, which is most likely due to the discrete nature of our dictionary. To improve upon this result we can reduce the orientation sampling characteristic distance. This will increase our dictionary size and thus our indexing time. An alternative is to perform orientation refinement, which we will do below.

We can get further confirmation of our initial analysis by inspecting some indexing quality maps, like the best matching scores and so-called orientation similarity (OS) map, which compares the best matching orientations for each pattern to it's nearest neighbours. Let's get the NCC map in the correct shape from the CrystalMap's scores property and the OS map with *orientation_similarity_map()* using the full list of best matches per point

[20]:
```
ncc_map = xmap.scores[:, 0].reshape(*xmap.shape)
os_map = kp.indexing.orientation_similarity_map(xmap)
```

And plot the maps

[21]:
```
fig, axes = plt.subplots(ncols=2, figsize=(11, 3))
im0 = axes[0].imshow(ncc_map)
im1 = axes[1].imshow(os_map)
fig.colorbar(im0, ax=axes[0], label="NCC", pad=0.02)
fig.colorbar(im1, ax=axes[1], label="Orientation similarity", pad=0.02)
for ax in axes:
    ax.axis("off")
fig.subplots_adjust(wspace=0)
```




From the NCC map we see that some grains match better than others. Due to the discrete nature of our dictionary, it is safe to assume that the best matching grains have patterns more similar to those in the dictionary than the others, i.e. the different coefficients does not reflect anything physical in the microstructure. The OS map doesn't tell us much more than the NCC map in this case.

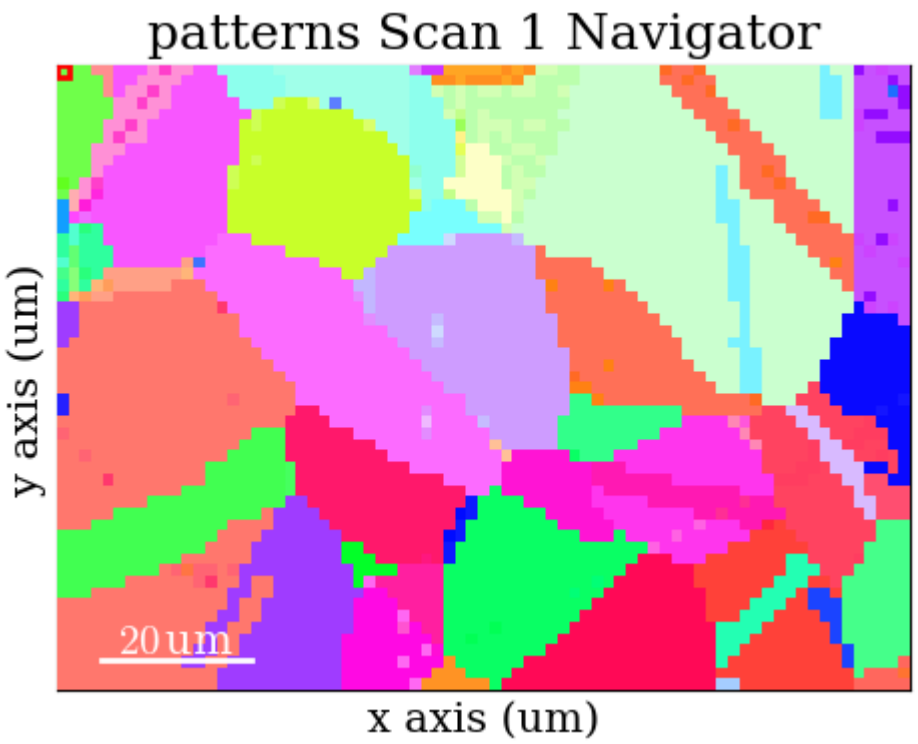
We can use the crystal map property `simulation_indices` to get the best matching simulated patterns from the dictionary of simulated patterns

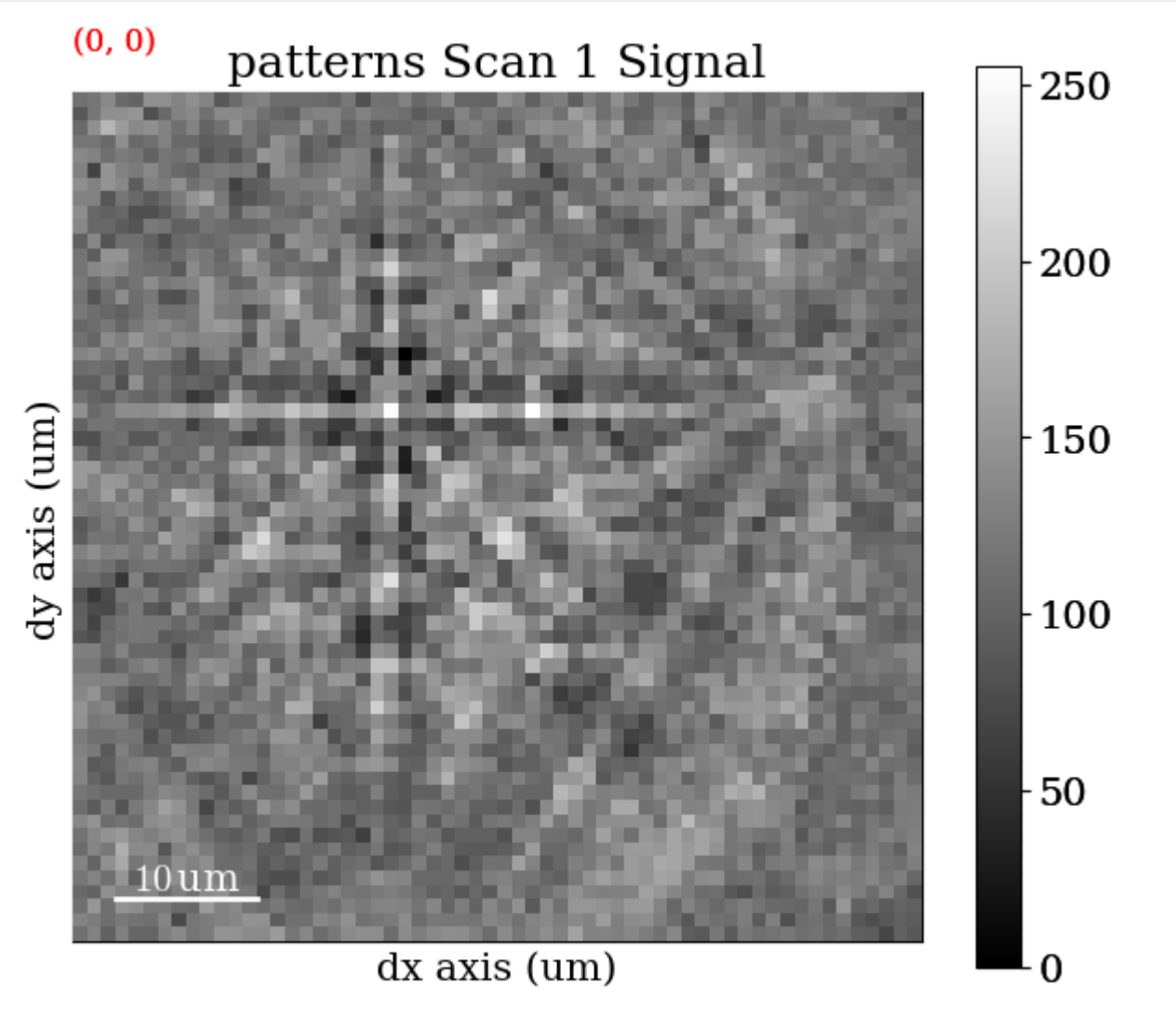
[22]:
```
best_patterns = sim.data[xmap.simulation_indices[:, 0]].reshape(s.data.shape)
s_best = kp.signals.EBSD(best_patterns)
s_best
```

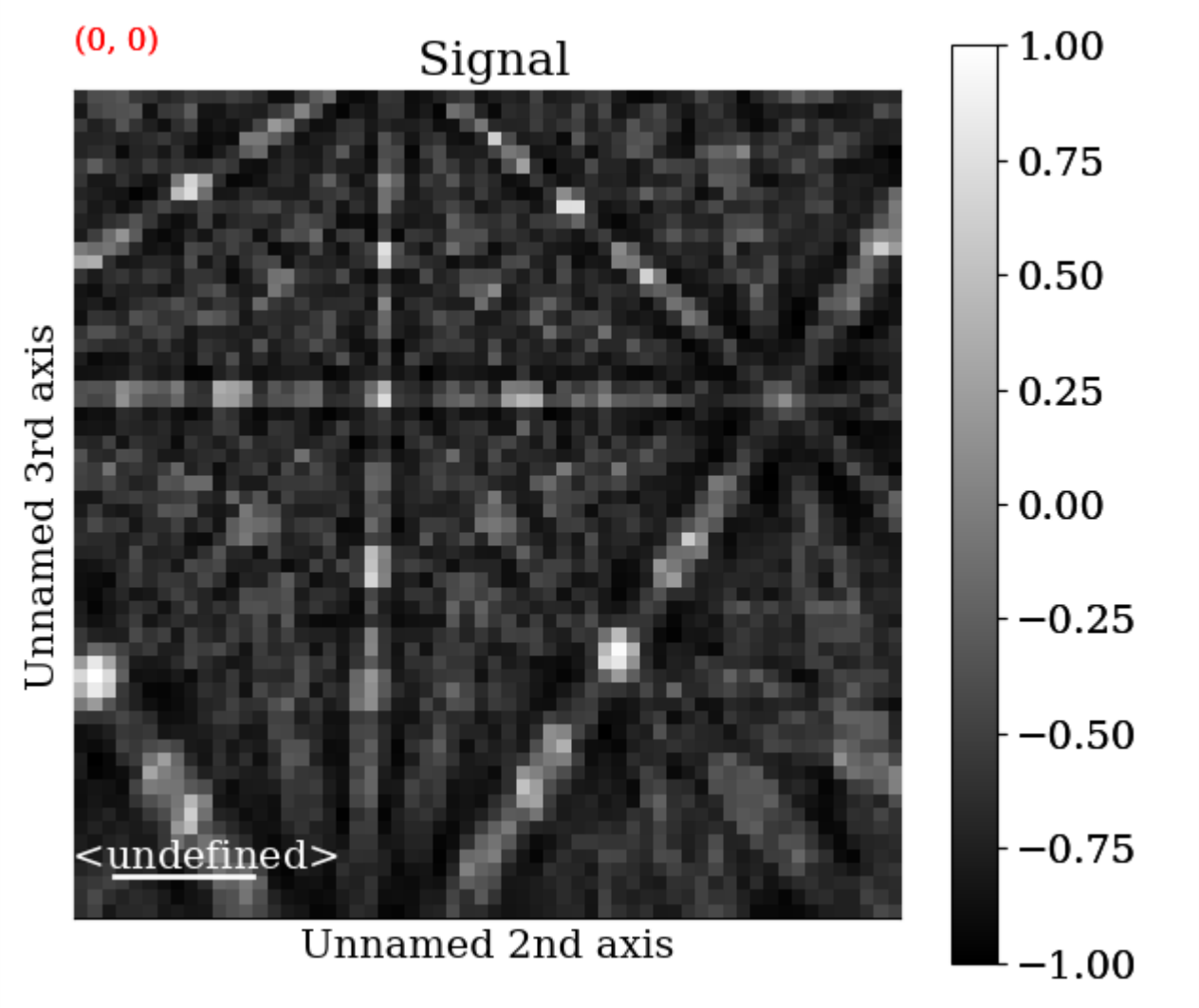
[22]:
```
<EBSD, title: , dimensions: (75, 55|60, 60)>
```

The simplest way to visually compare the experimental and best matching simulated patterns is to plot them with the same navigator. Let's create an RGB navigator signal from the IPF-X orientation map with *get_rgb_navigator()*. When using an interactive backend like `Qt5Agg`, we can then move the red square around to look at the patterns in each point.

[23]:
```
rgb_navigator = kp.draw.get_rgb_navigator(rgb_x.reshape(xmap.shape + (3,)))
hs.plot.plot_signals([s, s_best], navigator=rgb_navigator)
```

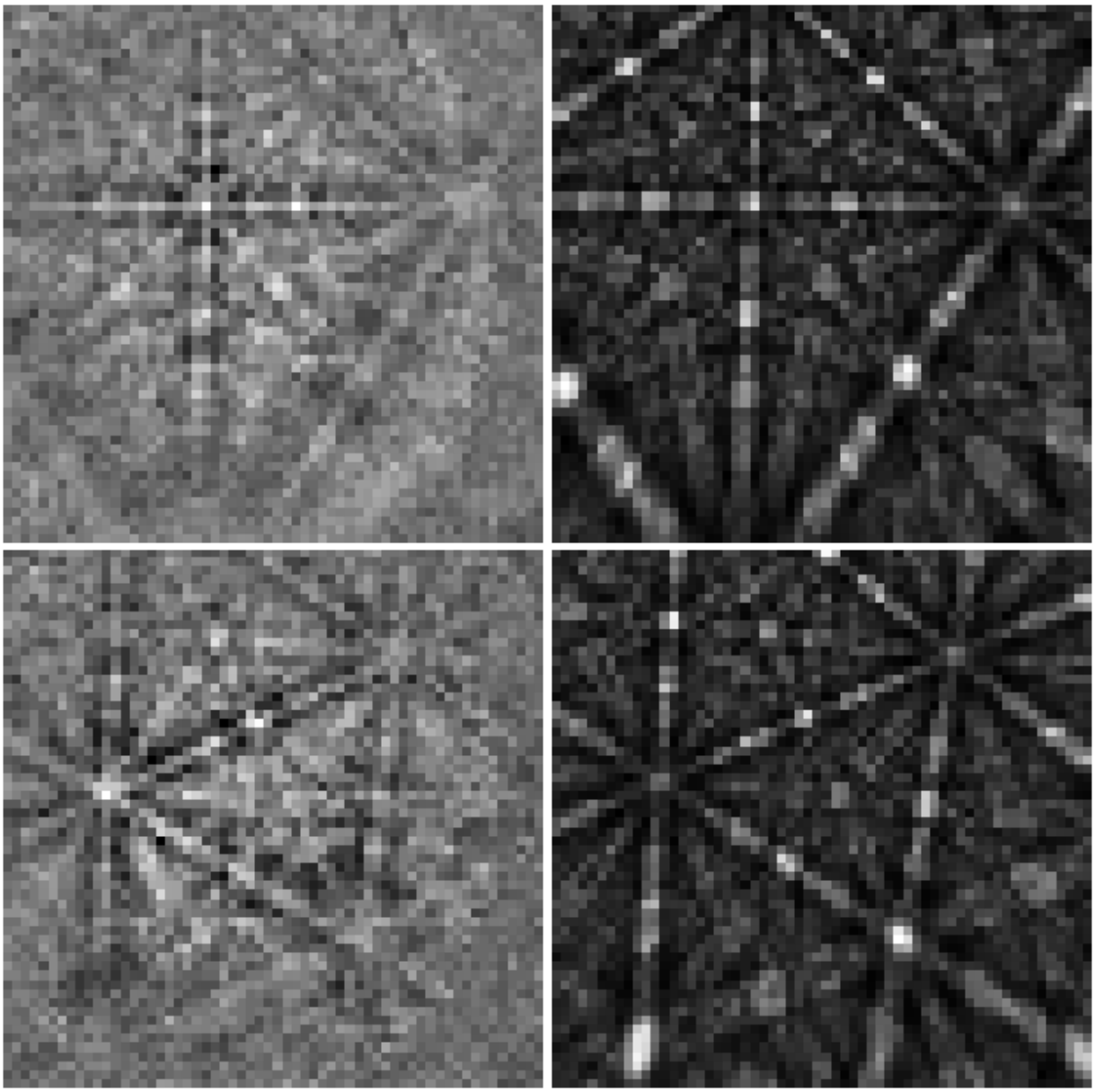



(0, 0)
patterns Scan 1 Signal
dy axis (um)
dx axis (um)
10 um
250
200
150
100
50
0




Let's also plot the best matches for patterns from two grains

[24]:
```
grain1 = (0, 0)
grain2 = (30, 10)

fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(8, 8))
axes[0, 0].imshow(s.inav[grain1].data, cmap="gray")
axes[0, 1].imshow(s_best.inav[grain1].data, cmap="gray")
axes[1, 0].imshow(s.inav[grain2].data, cmap="gray")
axes[1, 1].imshow(s_best.inav[grain2].data, cmap="gray")
for ax in axes.ravel():
    ax.axis("off")
fig.subplots_adjust(wspace=0.01, hspace=0.01)
```



### Refinement

Let's look at the effect of three refinement routes, all implemented as `EBSD` methods:

1. Refine orientations while keeping the PCs fixed: *refine_orientation()*
2. Refine PCs while keeping the orientations fixed: *refine_projection_center()*
3. Refine orientations and PCs at the same time: *refine_orientation_projection_center()*

For each route we will compare the maps and histograms of NCC scores before and after refinement, and also the PC parameters if appropriate.

Optimization is performed using an algorithm from the SciPy library or the NLopt library. Note that NLopt is an

optional dependency (see *the installation guide* for details). For every orientation and/or PC, we want to iteratively increase the similarity (NCC score) between our experimental pattern and a new simulated pattern projected onto our EBSD detector for every iteration, until the increase in similarity (gain) from one iteration to the next is smaller a certain threshold, by default 0.0001 for most algorithms. The orientation and/or PC is changed slightly in a controlled manner by the optimization algorithm for every iteration. The number of optimization evaluations (iterations) is returned after each refinement, either as a property in the crystal map (in route 1. and 3.) or as an array (in route 2.). We have access to both local and global optimization algorithms. Consult the kikuchipy docstring methods and the documentation of SciPy and NLopt, all linked above, for all available parameters and options.

Note that while we here refine orientations obtained from DI, any orientation results, e.g. from Hough indexing, can be refined with this approach, as long as an appropriate master pattern, *EBSDDetector* with PCs and a CrystalMap with orientations are provided.

---

**Note**

When using the Nelder-Mead optimization algorithm, EBSD refinement is generally faster with NLopt than with SciPy. Therefore, it is recommended to use NLopt. NLopt is however not as available on various machine architectures and across Python versions as SciPy is. This is why NLopt is an optional dependency and why SciPy's Nelder-Mead algorithm is the default in all refinement methods.

---

### Refine orientations

We use *refine_orientation()* to refine orientations while keeping the PCs fixed, using the default local Nelder-Mead simplex method from SciPy

[25]:
```
xmap_ref = s.refine_orientation(
    xmap=xmap,
    detector=det,
    master_pattern=mp,
    energy=energy,
    signal_mask=signal_mask,
    # The following are default values
    method="minimize",
    method_kwargs=dict(method="Nelder-Mead", tol=1e-4),
    compute=True,
)
```

```
Refinement information:
  Method: Nelder-Mead (local) from SciPy
  Trust region (+/-): None
  Keyword arguments passed to method: {'method': 'Nelder-Mead', 'tol': 0.0001}
Refining 4125 orientation(s):
[########################################] | 100% Completed | 59.03 ss
Refinement speed: 69.87429 patterns/s
```

Check the mean of the best matching score per pattern and the average number of optimization evaluations (iterations)

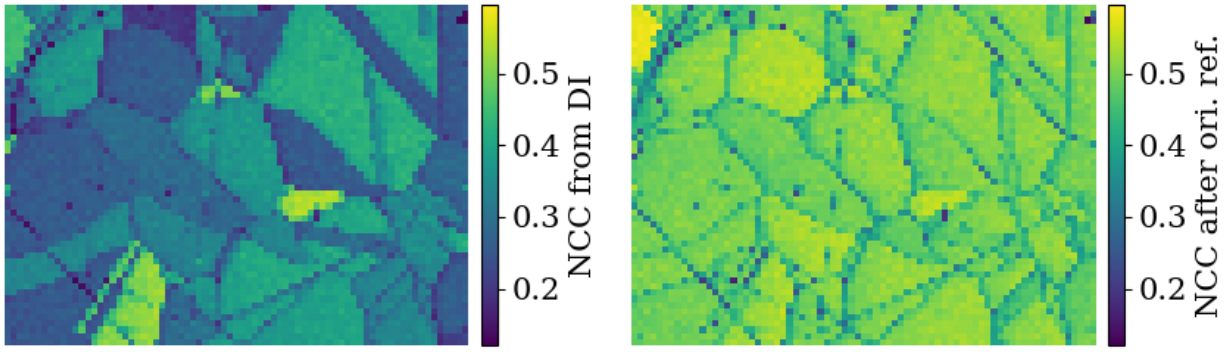
[26]:
```
print(xmap_ref.scores.mean())
print(xmap_ref.num_evals.mean())
```

```
0.48399137603875364
93.78060606060606
```

Compare the NCC score maps. We use the same colorbar bounds for both maps

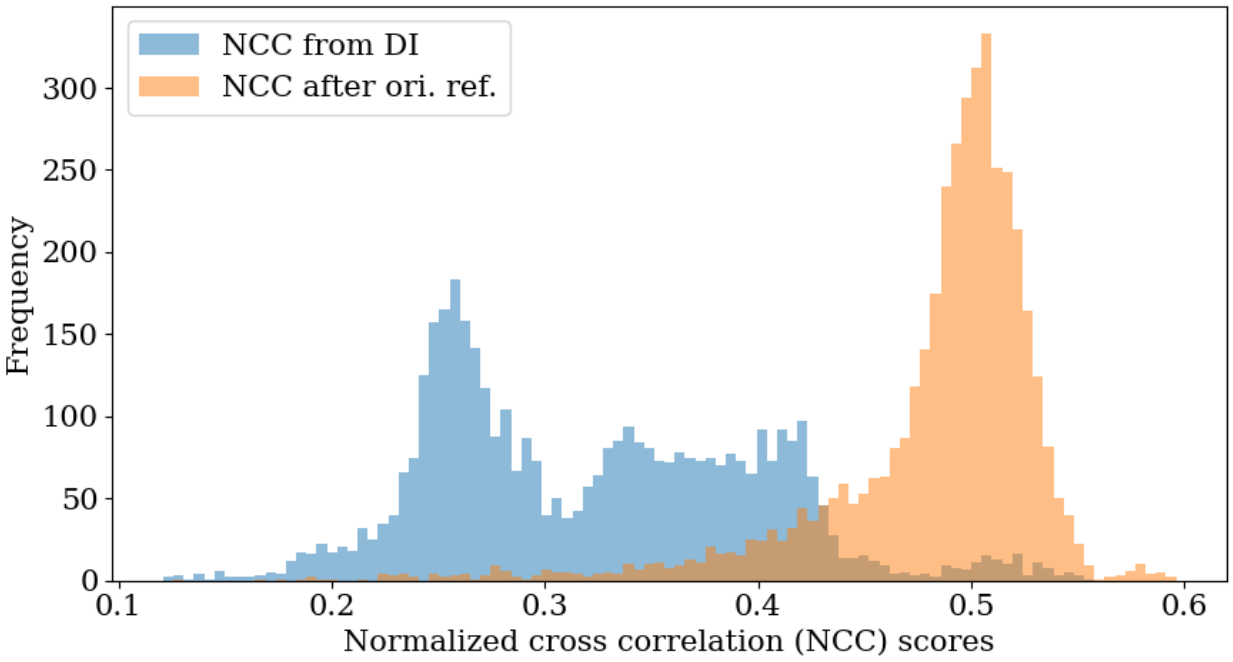
[27]:
```
ncc_after_ori_ref = xmap_ref.get_map_data("scores")

ncc_di_min = np.min(ncc_map)
ncc_di_max = np.max(ncc_map)
ncc_ori_ref_min = np.min(ncc_after_ori_ref)
ncc_ori_ref_max = np.max(ncc_after_ori_ref)

vmin = min([ncc_di_min, ncc_ori_ref_min])
vmax = max([ncc_di_max, ncc_ori_ref_max])
```

[28]:
```
ncc_after_ori_ref_label = "NCC after ori. ref."

fig, axes = plt.subplots(ncols=2, figsize=(11, 3))
im0 = axes[0].imshow(ncc_map, vmin=vmin, vmax=vmax)
im1 = axes[1].imshow(ncc_after_ori_ref, vmin=vmin, vmax=vmax)
fig.colorbar(im0, ax=axes[0], label="NCC from DI", pad=0.02)
fig.colorbar(im1, ax=axes[1], label=ncc_after_ori_ref_label, pad=0.02)
for ax in axes:
    ax.axis("off")
fig.subplots_adjust(wspace=0)
```




Compare the histograms

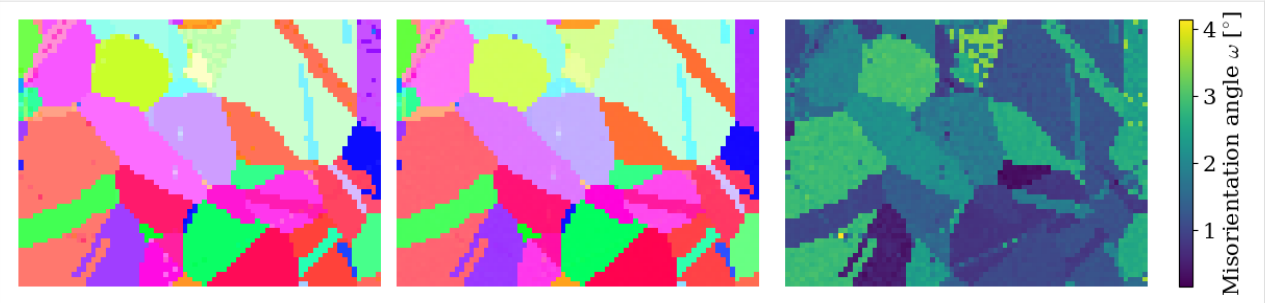
[29]:
```
bins = np.linspace(vmin, vmax, 100)
fig, ax = plt.subplots(figsize=(9, 5))
_ = ax.hist(ncc_map.ravel(), bins, alpha=0.5, label="NCC from DI")
_ = ax.hist(
    ncc_after_ori_ref.ravel(),
    bins,
    alpha=0.5,
    label=ncc_after_ori_ref_label,
)
ax.set(xlabel="Normalized cross correlation (NCC) scores", ylabel="Frequency")
ax.legend()
fig.tight_layout();
```




We see that DI found the best orientation (with a fixed PC) for most of the patterns, which the refinement was able to improve further. However, there are a few patterns with a very low NCC score (0.1-0.2) which refinement couldn't improve upon, which tells us that these were most likely misindexed during DI. If there are Kikuchi bands in these patterns, a larger dictionary with a finer orientation sampling could improve indexing of them.

Let's also plot the IPF-X orientation maps before and after refinement, and also the disorientation angle (smallest misorientation angle found after accounting for symmetry) difference between the two maps

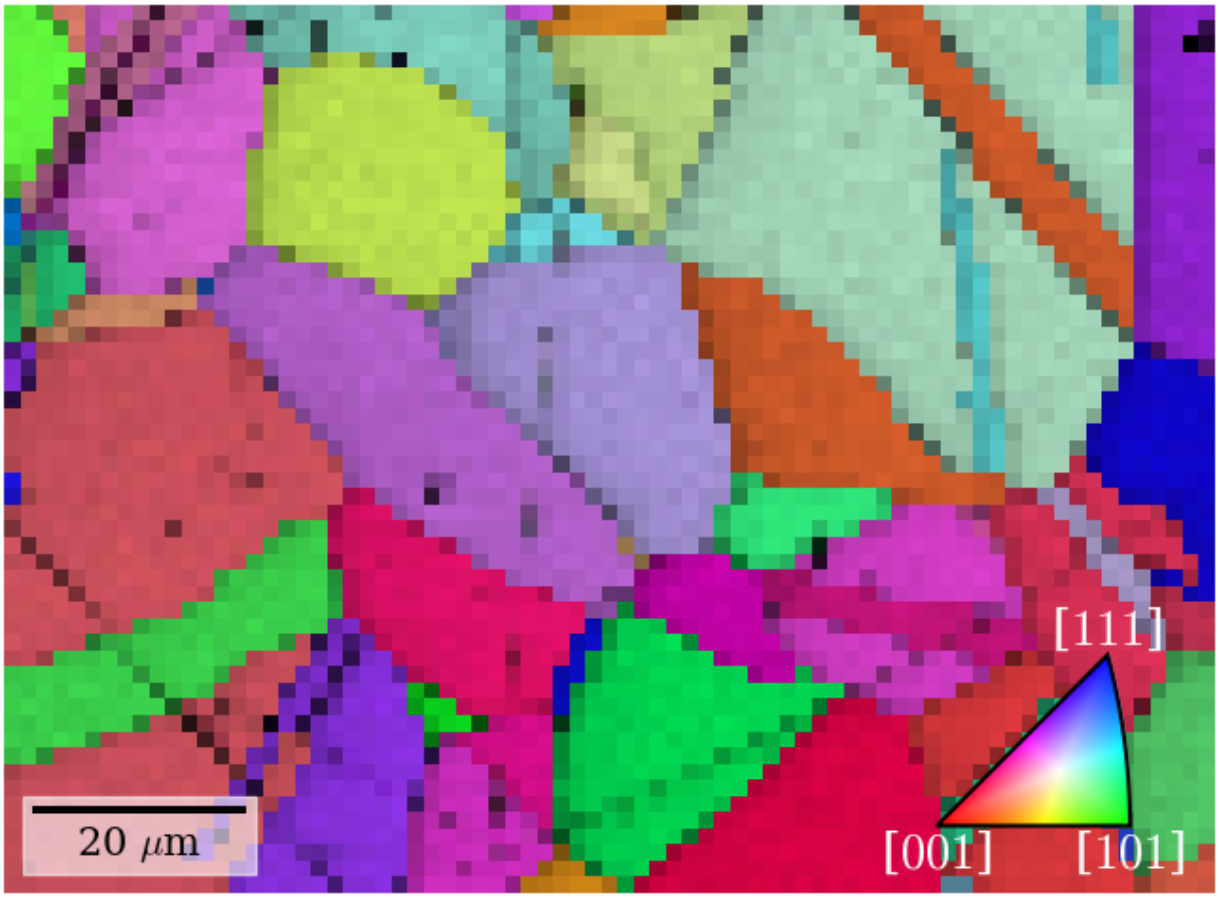
[30]:
```
G_ref = xmap_ref.orientations

ckey_m3m.direction = Vector3d.xvector()
rgb_x = ckey_m3m.orientation2color(G)
rgb_x_ref = ckey_m3m.orientation2color(G_ref)

angles = G.angle_with(G_ref, degrees=True)

fig, axes = plt.subplots(ncols=3, figsize=(14, 3))
axes[0].imshow(rgb_x.reshape(xmap.shape + (3,)))
axes[1].imshow(rgb_x_ref.reshape(xmap.shape + (3,)))
im2 = axes[2].imshow(angles.reshape(xmap.shape))
fig.colorbar(
    im2, ax=axes[2], label=r"Misorientation angle $\omega$ [$^{\circ}$]"
)
for ax in axes:
    ax.axis("off")
fig.tight_layout(w_pad=-12)
```




We see that refinement was able to remove the scatter of similar colors in the grains.

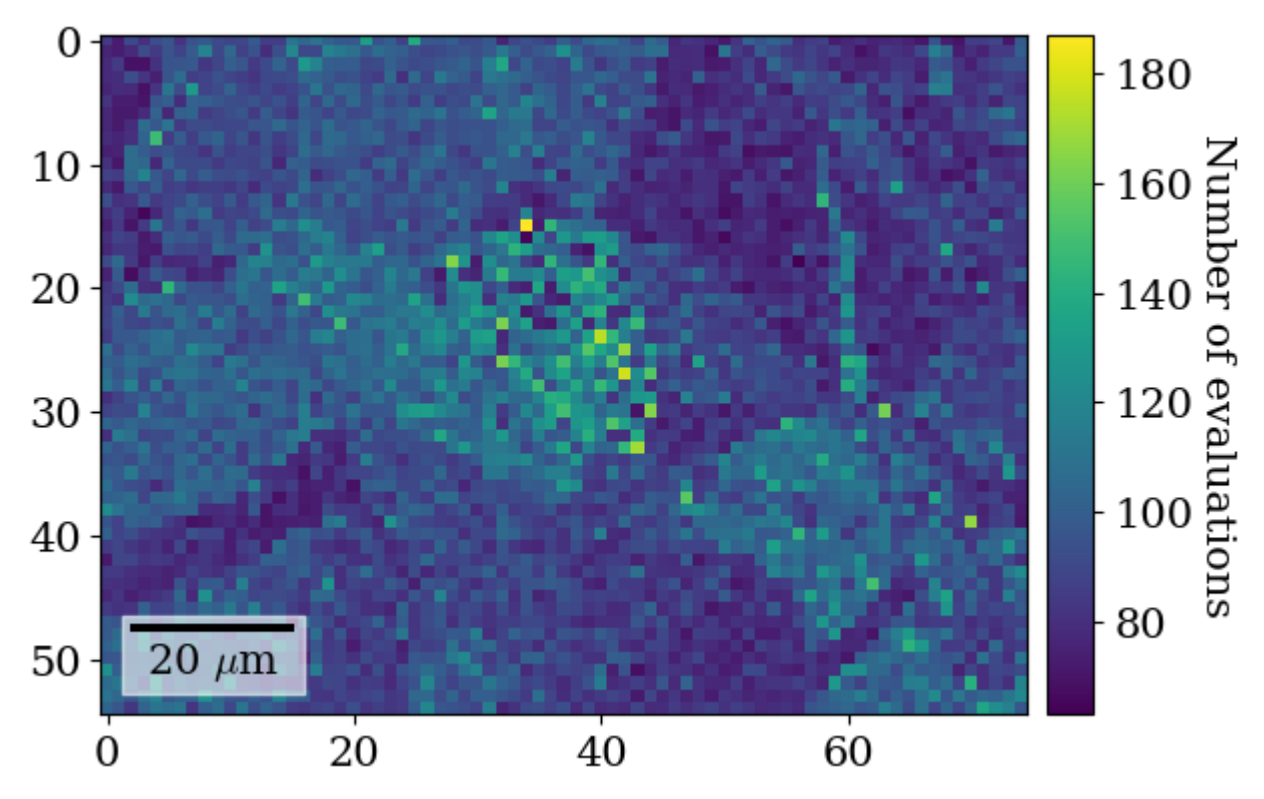
Finally, let's plot the IPF-X map with correlation scores overlayed. We will also plot the IPF color key in the bottom right corner.

[31]:
```
fig = xmap_ref.plot(
    rgb_x_ref, overlay="scores", remove_padding=True, return_figure=True
)

# Place color key in bottom right corner,
# coordinates are [left, bottom, width, height]
ax_ckey = fig.add_axes(
    [0.75, 0.08, 0.2, 0.2], projection="ipf", symmetry=pg
)
ax_ckey.plot_ipf_color_key(show_title=False)
ax_ckey.patch.set_facecolor("None")
_ = [text.set_color("w") for text in ax_ckey.texts]  # White [uvw]
```




Finally, let's plot the number of optimization evaluations (iterations) necessary for each pattern

[32]:
```
xmap_ref.plot(
    "num_evals", colorbar=True, colorbar_label="Number of evaluations"
)
```




### Refine projection centers

We use *refine_projection_center()* to refine PCs while keeping the orientations fixed. We'll demonstrate this using the local modified Powell method from SciPy. This method is also known as Bound Optimization BY Quadratic Approximation (BOBYQA), and is used in EMsoft and discussed by [Singh *et al.*, 2017]. We will pass a `trust_region` of +/- 2% for the PC parameters to use as upper and lower bounds during refinement.

We can also pass `compute=False`, to do the refinement later. The results will then be a dask.array.Array. We can pass this array to *kikuchipy.indexing.compute_refine_projection_center_results()* and perform the refinement to retrieve the results

[33]:
```
result_arr = s.refine_projection_center(
    xmap=xmap,
    detector=det,
    master_pattern=mp,
    energy=energy,
    signal_mask=signal_mask,
    method="minimize",
    method_kwargs=dict(method="Powell", tol=1e-3),
    trust_region=[0.02, 0.02, 0.02],
    compute=False,
)
```

```
Refinement information:
  Method: Powell (local) from SciPy
  Trust region (+/-): [0.02 0.02 0.02]
  Keyword arguments passed to method: {'method': 'Powell', 'tol': 0.001}
```

[34]:
```
(
    ncc_after_pc_ref,
    det_ref,
    num_evals_ref,
) = kp.indexing.compute_refine_projection_center_results(
    results=result_arr, detector=det, xmap=xmap
)
```

```
Refining 4125 projection center(s):
[########################################] | 100% Completed | 54.42 ss
Refinement speed: 75.74699 patterns/s
```

Check the mean of the best matching score per pattern and the average number of optimization evaluations (iterations)

[35]:
```
print(ncc_after_pc_ref.mean())
print(num_evals_ref.mean())
```

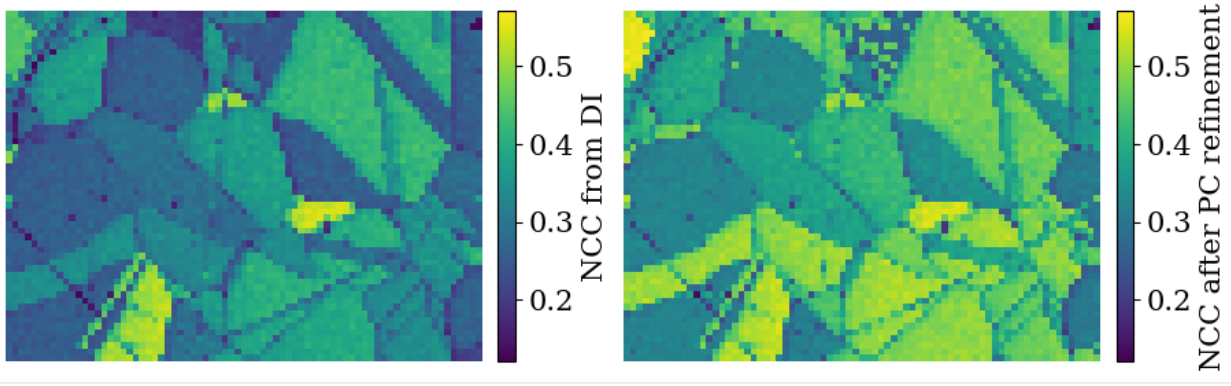
```
0.40068437741380747
65.6339393939394
```

Note that `refine_orientation()` and `refine_orientation_projection_center()` also takes the `compute` parameter, and there are similar functions to compute the refinement results: *kikuchipy.indexing.compute_refine_orientation_results()* and *kikuchipy.indexing.compute_refine_orientation_projection_center_results()*.

Let's plot the refined scores and PCs

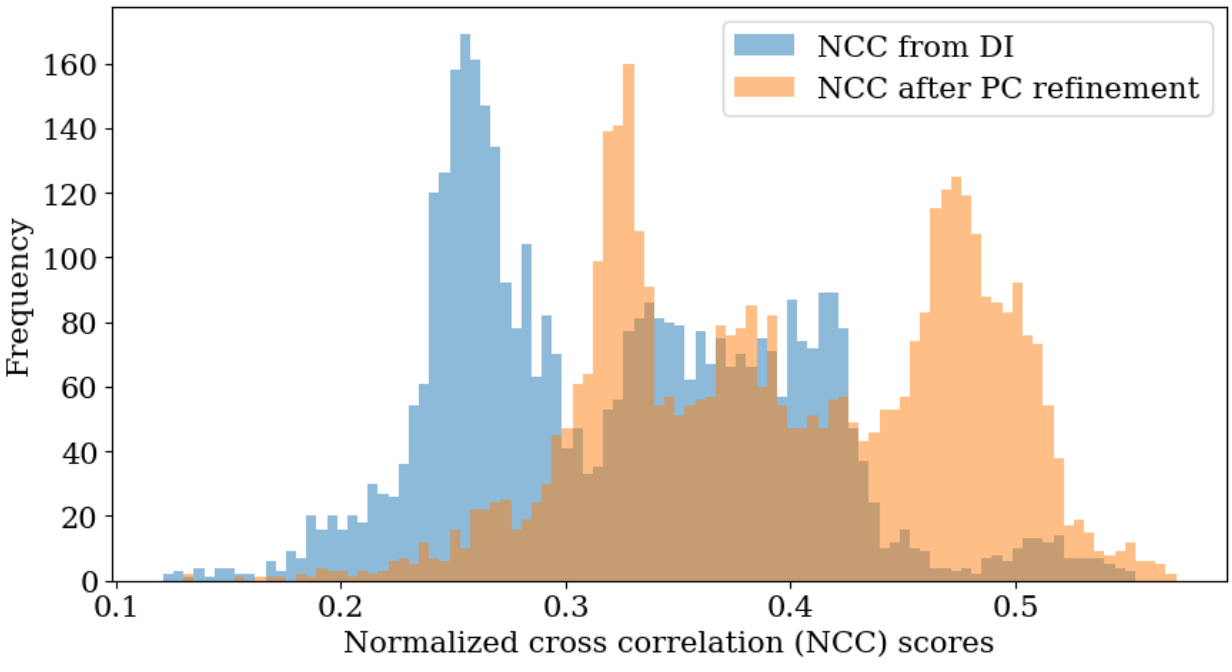
[36]:
```
ncc_pc_ref_min = np.min(ncc_after_pc_ref)
ncc_pc_ref_max = np.max(ncc_after_pc_ref)

vmin2 = min([ncc_di_min, ncc_pc_ref_min])
vmax2 = max([ncc_di_max, ncc_pc_ref_max])
```

[37]:
```
ncc_after_pc_ref_label = "NCC after PC refinement"

fig, axes = plt.subplots(ncols=2, figsize=(11, 3))
im0 = axes[0].imshow(ncc_map, vmin=vmin2, vmax=vmax2)
im1 = axes[1].imshow(ncc_after_pc_ref, vmin=vmin2, vmax=vmax2)
fig.colorbar(im0, ax=axes[0], label="NCC from DI", pad=0.02)
fig.colorbar(im1, ax=axes[1], label=ncc_after_pc_ref_label, pad=0.02)
for ax in axes:
    ax.axis("off")
fig.tight_layout(w_pad=-7)
```




Compare the NCC score histograms

[38]:
```
bins = np.linspace(vmin2, vmax2, 100)
fig, ax = plt.subplots(figsize=(9, 5))
_ = ax.hist(ncc_map.ravel(), bins, alpha=0.5, label="NCC from DI")
_ = ax.hist(
    ncc_after_pc_ref.ravel(),
    bins,
    alpha=0.5,
    label=ncc_after_pc_ref_label,
)
ax.set(xlabel="Normalized cross correlation (NCC) scores", ylabel="Frequency")
ax.legend()
fig.tight_layout();
```



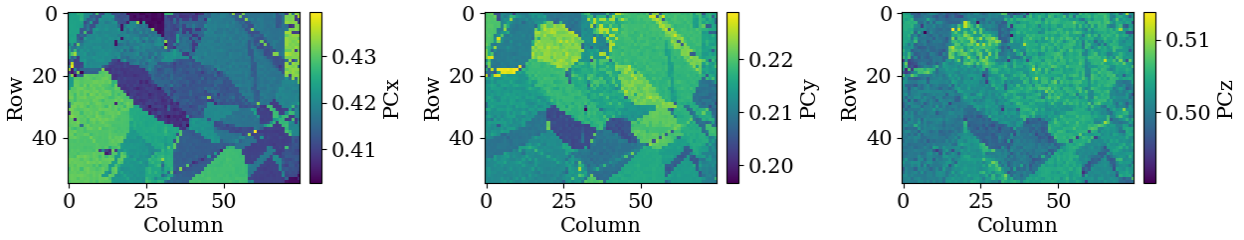

[39]:
```
print(
    f"PC used in DI:           {det.pc_average}\n"
    f"PC after PC refinement: {det_ref.pc_average.round(4)}"
```

(continues on next page)

(continued from previous page)

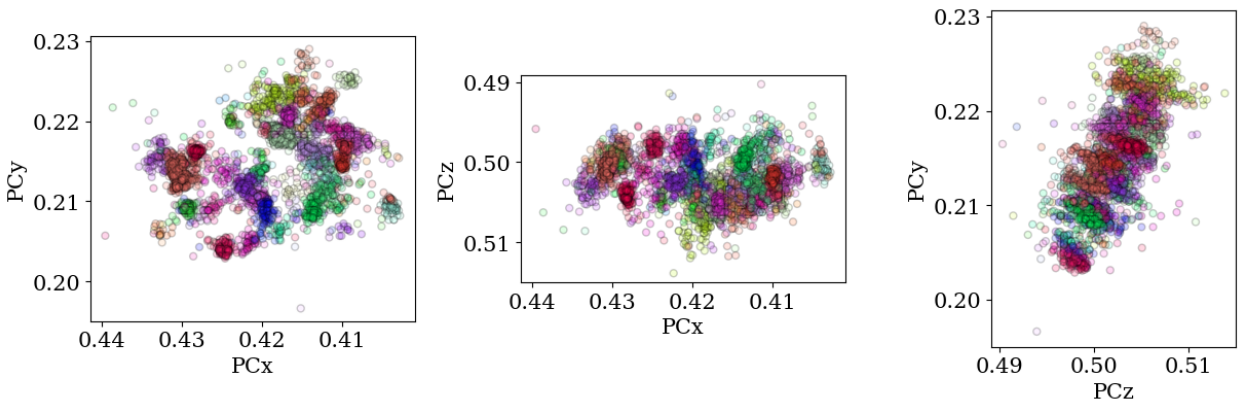
```
)

det_ref.plot_pc()
```

```
PC used in DI:          [0.4198 0.2136 0.5015]
PC after PC refinement: [0.4188 0.215  0.5023]
```




The maps indicate an orientation dependence of the PC, meaning that changes in PC can be accomodated by crystal rotations, to a certain extent. We can plot the PCs as functions of each other and color them according to the IPF-X map above to see this more clearly

[40]:
```
det_ref.plot_pc("scatter", c=rgb_x, alpha=0.2)
```




This orientation dependent "slop" is nicely demonstrated by [Pang *et al.*, 2020]. A part of their analysis is shown in the *Orientation dependence of the projection center tutorial*.

### Refine orientations and projection centers

We use *refine_orientation_projection_center()* to refine orientations and PCs at the same time. To do this we'll use the implementation of Nelder-Mead in NLopt, an optional dependency (see *the installation guide* for details).

[41]:
```
xmap_ref2, det_ref2 = s.refine_orientation_projection_center(
    xmap=xmap,
    detector=det,
    master_pattern=mp,
    energy=energy,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    trust_region=[2, 2, 2, 0.05, 0.05, 0.05],
```

(continues on next page)

(continued from previous page)

```
    rtol=1e-3,
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [2.   2.   2.   0.05 0.05 0.05]
  Relative tolerance: 0.001
Refining 4125 orientation(s) and projection center(s):
[########################################] | 100% Completed | 75.38 ss
Refinement speed: 54.71418 patterns/s
```

Check the mean of the best matching score per pattern and the average number of optimization evaluations (iterations)

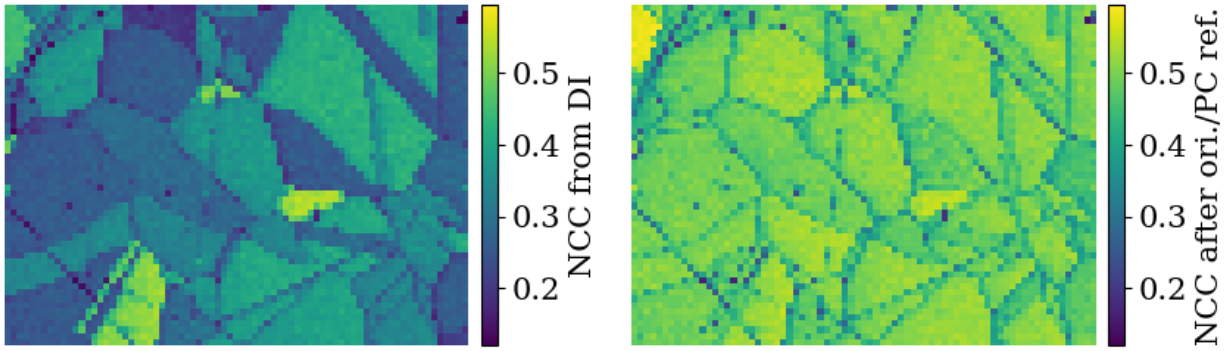
[42]:
```
print(xmap_ref2.scores.mean())
print(xmap_ref2.num_evals.mean())
```

```
0.4793854339050524
116.48145454545454
```

Compare the NCC score maps. We use the same colorbar bounds for both maps

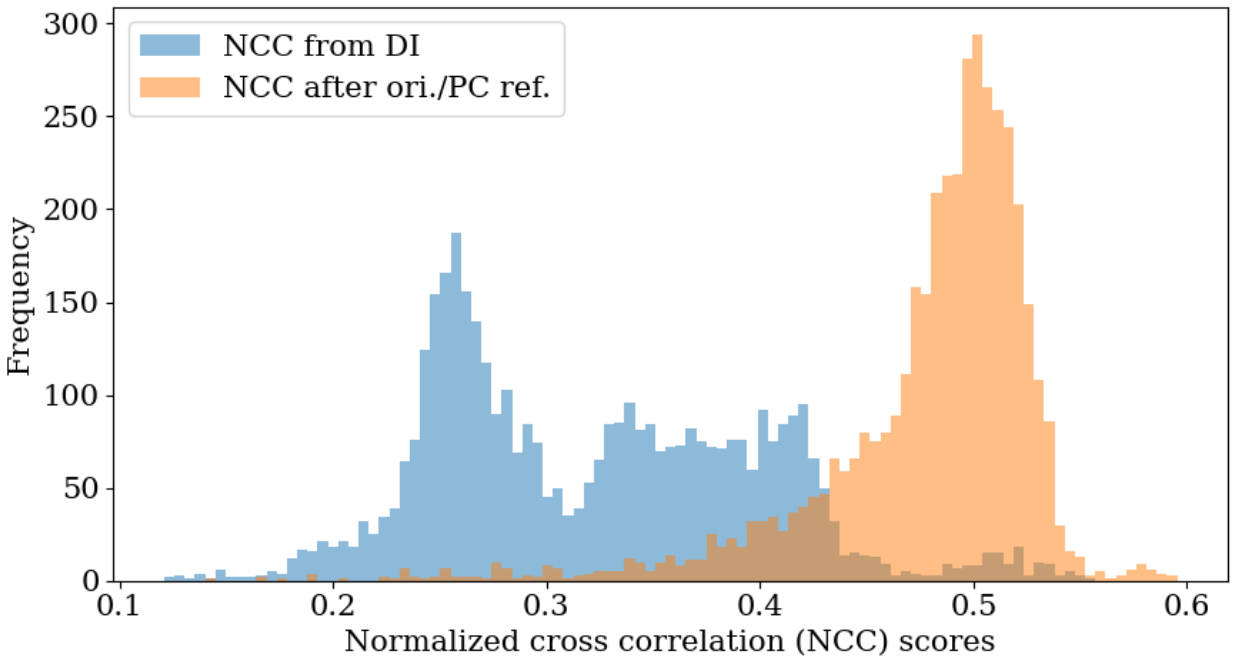
[43]:
```
ncc_after_ori_pc_ref = xmap_ref2.get_map_data("scores")

ncc_ori_pc_ref_min = np.min(ncc_after_ori_pc_ref)
ncc_ori_pc_ref_max = np.max(ncc_after_ori_pc_ref)

vmin3 = min([ncc_di_min, ncc_ori_pc_ref_min])
vmax3 = max([ncc_di_max, ncc_ori_pc_ref_max])
```

[44]:
```
ncc_after_ori_pc_ref_label = "NCC after ori./PC ref."

fig, axes = plt.subplots(ncols=2, figsize=(11, 3))
im0 = axes[0].imshow(ncc_map, vmin=vmin3, vmax=vmax3)
im1 = axes[1].imshow(ncc_after_ori_pc_ref, vmin=vmin3, vmax=vmax3)
fig.colorbar(im0, ax=axes[0], label="NCC from DI", pad=0.02)
fig.colorbar(im1, ax=axes[1], label=ncc_after_ori_pc_ref_label, pad=0.02)
for ax in axes:
    ax.axis("off")
fig.subplots_adjust(wspace=0)
```




Compare the histograms

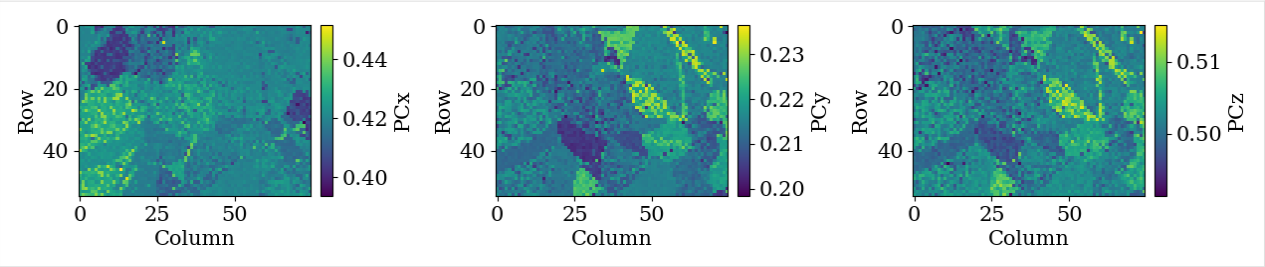
[45]:
```
bins = np.linspace(vmin3, vmax3, 100)
fig, ax = plt.subplots(figsize=(9, 5))
_ = ax.hist(ncc_map.ravel(), bins, alpha=0.5, label="NCC from DI")
_ = ax.hist(
    ncc_after_ori_pc_ref.ravel(),
    bins,
    alpha=0.5,
    label=ncc_after_ori_pc_ref_label,
)
ax.set(xlabel="Normalized cross correlation (NCC) scores", ylabel="Frequency")
ax.legend()
fig.tight_layout();
```



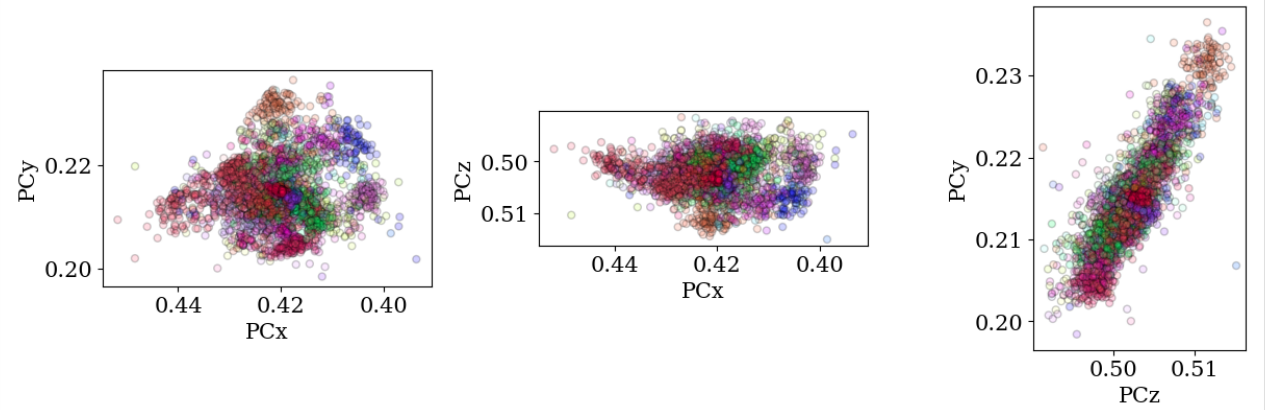

Let's also inspect the refined PC parameters

[46]:
```
print(
    f"PC used in DI:          {det.pc_average}\n"
    f"PC after PC refinement: {det_ref2.pc_average.round(4)}"
)

det_ref2.plot_pc()
```

```
PC used in DI:          [0.4198 0.2136 0.5015]
PC after PC refinement: [0.4204 0.2148 0.5022]
```




[47]:
```
rgbx_ref2 = ckey_m3m.orientation2color(xmap_ref2.orientations)
det_ref2.plot_pc("scatter", c=rgbx_ref2, alpha=0.2)
```




We see that the ranges of refined PCs have narrowed compared to when only the PC was optimized (as seen above). But, the unexpected inverse relation between PCz and PCy remains.

It is generally advisable to refine orientations and PCs simultaneously only when estimating PCs in a grid across the sample in order to fit a plane of PCs to this grid. Indexing can then be performed with a PC from this plane for each pattern, only refining the orientation.

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

## Hybrid indexing

In this tutorial, we combine Hough indexing (HI), dictionary indexing (DI), and refinement in a hybrid indexing approach. HI is generally much faster than DI, but less robust towards noisy patterns. To make good use of both indexing approaches, we can index all patterns with HI, identify badly indexed map points, and re-index these with DI. Before combining orientations obtained from HI and DI into a single map, we must refine them. As always, it is important to validate intermediate results after each step by inspecting quality metrics, geometrical simulations etc.

We demonstrate this workflow with an EBSD dataset from a single-phase recrystallized nickel sample. The dataset is available in a repository on Zenodo at [Ånes *et al.*, 2019]. The dataset is number ten (24 dB) out of a series of ten datasets in the repostiroy, taken with increasing gain (0-24 dB). It is *very* noisy, so we will average each pattern with its nearest neighbours before indexing.

The complete workflow is:

1. Load, process, and inspect the full dataset

2. Calibrate geometry to obtain a plane of projection centers (one for each map point)
3. Hough indexing of all patterns
4. Identify (bad) points for re-indexing
5. Re-indexing with dictionary indexing
6. Refine Hough indexed and dictionary indexed points
7. Merge results
8. Validate results

Let's start by important the necessary libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

from diffsims.crystallography import ReciprocalLatticeVector
import hyperspy.api as hs
import kikuchipy as kp
from orix import io, plot, sampling
from orix.crystal_map import PhaseList
from orix.vector import Vector3d


plt.rcParams.update({
    "figure.facecolor": "w",
    "font.size": 15,
    "figure.dpi": 75,
})
```

### Load, process and inspect data

[2]:
```
s = kp.data.ni_gain(10, allow_download=True)  # ~100 MB into memory
s
```

[2]:
```
<EBSD, title: Pattern, dimensions: (200, 149|60, 60)>
```

Enhance the Kikuchi pattern with background correction

[3]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 706.39 ms
[########################################] | 100% Completed | 3.13 ss
```

Average each pattern to its eight nearest neighbors in a Gaussian kernel with a standard deviation of 1

[4]:
```
window = kp.filters.Window("gaussian", std=1)
s.average_neighbour_patterns(window)
```

```
[########################################] | 100% Completed | 1.23 sms
```

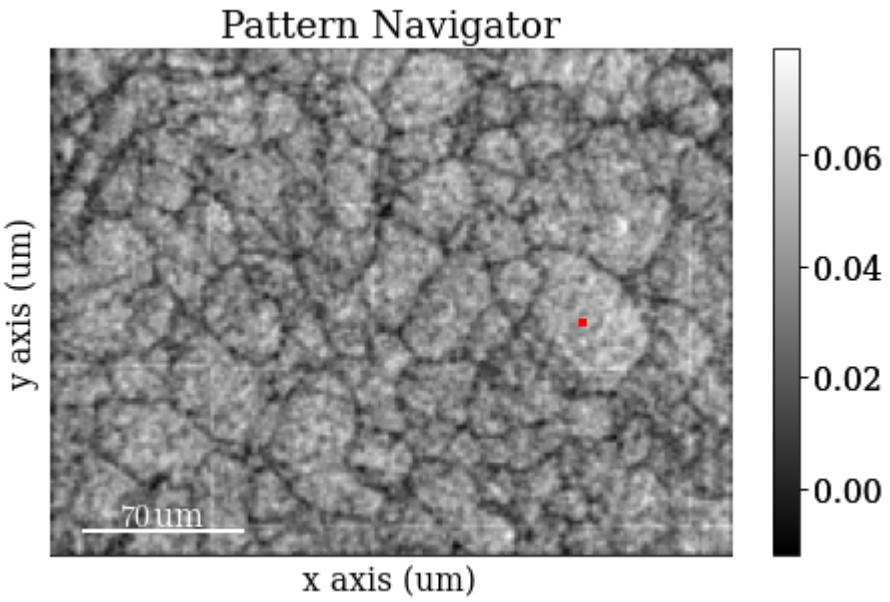
Inspect an image quality map (pattern sharpness, not to be confused with image/pattern quality determined from the height of peaks in the Radon transform)

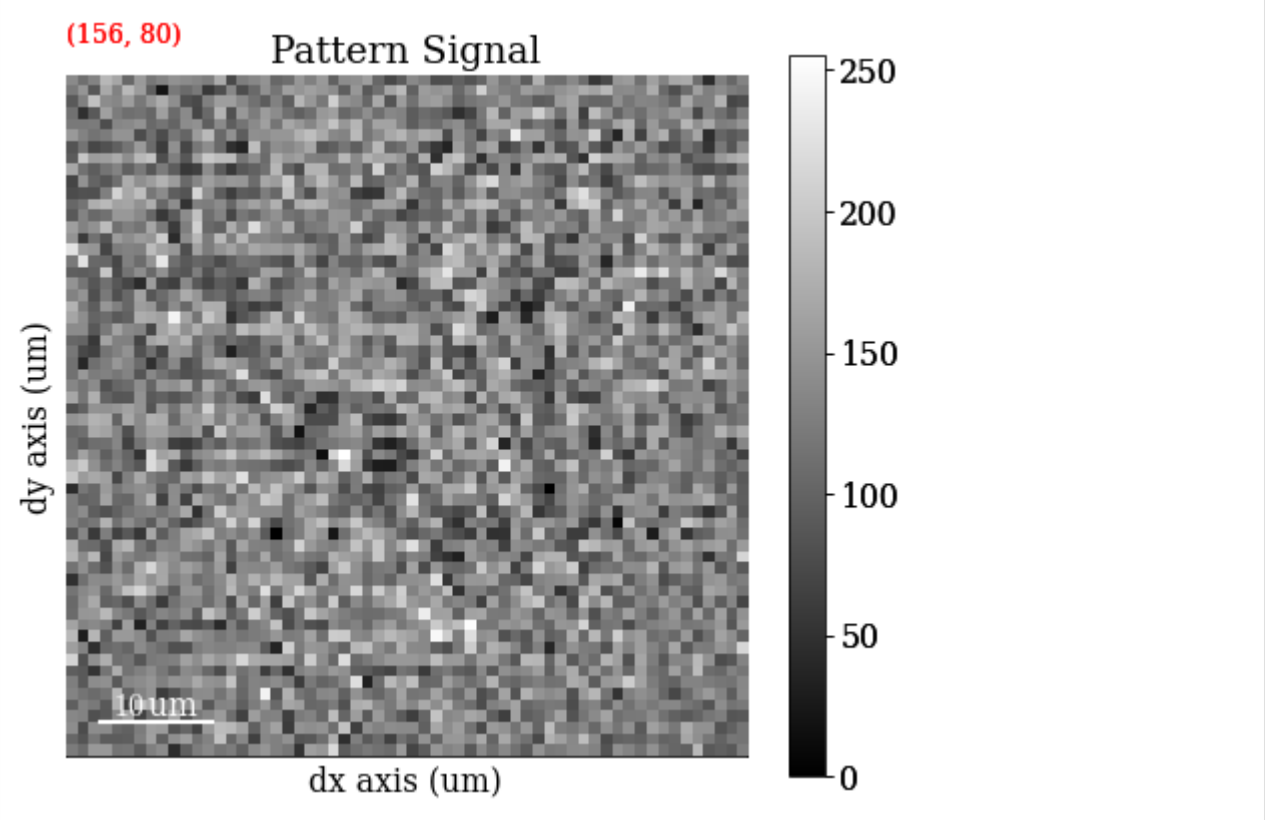
[5]:
```
maps_iq = s.get_image_quality()
```

```
[########################################] | 100% Completed | 1.61 ss
```

Inspect patterns in the image quality map

[6]:
```
# Move the pointer programmatically to the center of a large grain
s.axes_manager.indices = (156, 80)

s.plot(hs.signals.Signal2D(maps_iq))
```






The image quality map is *very* noisy. However, we might be able to convince ourselves that the darker lines are grain boundaries. The map is noisy because the patterns are noisy. The pattern shown is from the center of a large grain on the right side of the map from the small red square (the pointer). Even though the material is well recrystallized with appreciably large grains, the pattern is *very* noisy. But again, we might be able to convince ourselves that the pattern shows "correlated noise", e.g. a couple of zones axes (darker regions in the pattern) and some bands delineated by darker lines on each side of the band.

### Calibrate geometry

Seven calibration patterns of high quality was acquired prior to acquiring the full dataset above. These were acquired to calibrate the sample-detector geometry. The detector was mounted with the screen normal at 0° to the horizontal. The sample was tilted to 70° from the horizontal. We assume these tilts to be correct. What remains is to determine a plane of projection centers (PCs), one for each map point. Since we know the detector pixel size (~70 um on the NORDIF UF-1100 detector), we can extrapolate this plane of PCs from a mean PC. The workflow is as follows:

1. Estimate PCs from an initial guess using Hough indexing
2. Hough indexing of calibration patterns using estimated PCs
3. Refine Hough indexed orientations and estimated PCs using pattern matching
4. Extrapolate plane of PCs from mean of refined PCs

We validate the results after each step.

Load calibration patterns

[7]:
```
s_cal = kp.data.ni_gain_calibration(10)
s_cal
```

[7]:
```
<EBSD, title: Calibration patterns, dimensions: (7|480, 480)>
```

Remove static and dynamic background

[8]:
```
s_cal.remove_static_background()
s_cal.remove_dynamic_background()
```

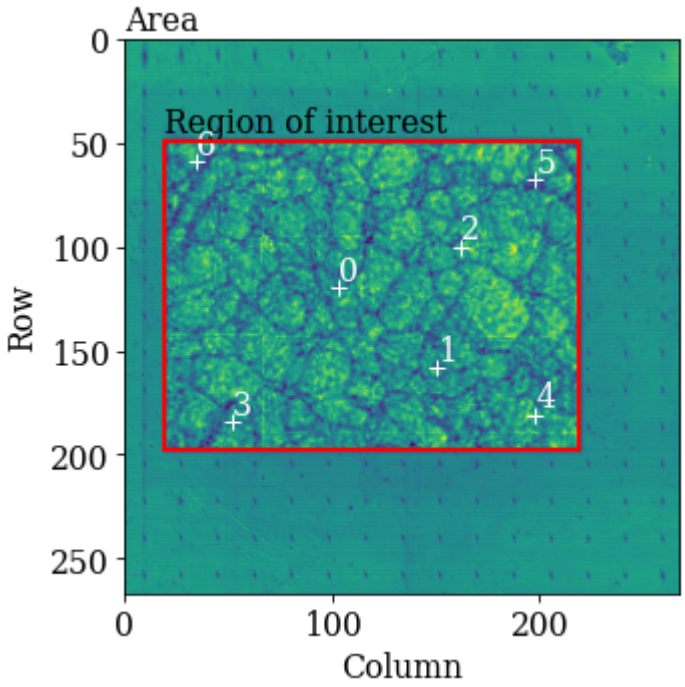
```
[########################################] | 100% Completed | 101.75 ms
[########################################] | 100% Completed | 101.68 ms
```

Extract the positions of the calibration patterns (possibly outside the region of interest [ROI]) and the shape and position of the ROI relative to the area imaged in an overview secondary electron image. This information is read with the calibration patterns from the NORDIF settings file.

[9]:
```
omd = s_cal.original_metadata
```

Plot calibration pattern map locations

[10]:
```
kp.draw.plot_pattern_positions_in_map(
    rc=omd.calibration_patterns.indices_scaled,
    roi_shape=omd.roi.shape_scaled,
    roi_origin=omd.roi.origin_scaled,
    roi_image=maps_iq,
    area_shape=omd.area.shape_scaled,
    area_image=omd.area_image,
    color="w",
)
```




Hough indexing requires a phase list in order to make a look-up table of interplanar angles to compare the detected angles to (from combinations of bands). See the *Hough indexing tutorial* for more details. Since we later on need a dynamically simulated master pattern of nickel (simulated with EMsoft), we will load this here and use the phase description of the master pattern in the phase list.

**Note**

PyEBSDIndex is an optional dependency of kikuchipy, and can be installed with both `pip` and `conda` (from `conda-forge`). To install PyEBSDIndex, see their installation instructions.

[11]:
```
# kikuchipy.data.nickel_ebsd_master_pattern_small() is a lower-resolution alternative
mp = kp.data.ebsd_master_pattern(
    "ni", allow_download=True, projection="lambert", energy=20
)
mp
```

[11]:
```
<EBSDMasterPattern, title: ni_mc_mp_20kv, dimensions: (|1001, 1001)>
```

[12]:
```
phase = mp.phase
phase
```

[12]:
```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
```

Create a PyEBSDIndex indexer to use with Hough indexing. We get this from our EBSD detector instance attached to the calibration pattern signal

[13]:
```
det_cal = s_cal.detector
phase_list = PhaseList(phase)
indexer = det_cal.get_indexer(phase_list, rSigma=2, tSigma=2)
```

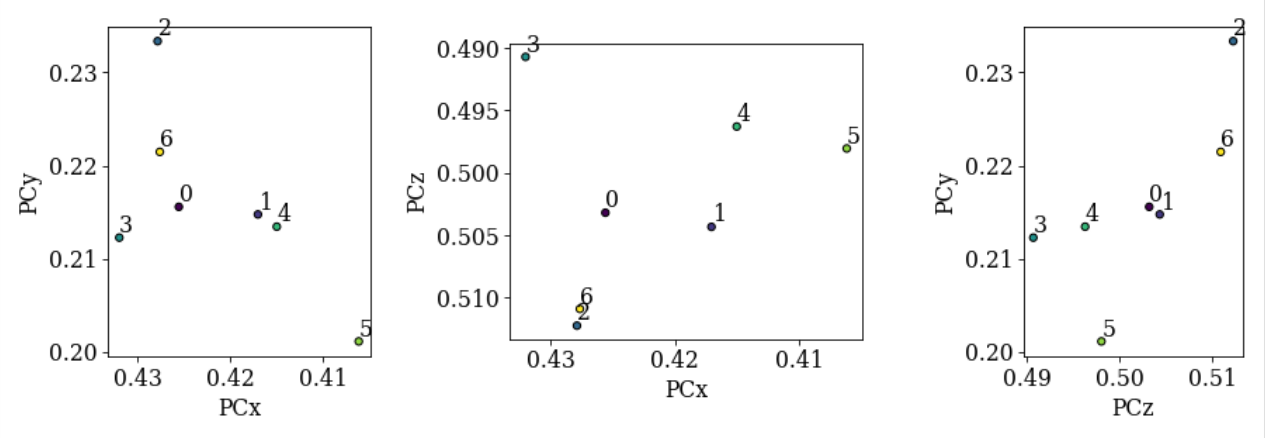
Estimate PCs from an initial guess (based on previous experiments) and print the mean and standard deviation

[14]:
```
det_cal = s_cal.hough_indexing_optimize_pc(
    pc0=[0.42, 0.22, 0.50],
    indexer=indexer,
    batch=True,
    method="PSO",
    search_limit=0.05,
)

print(det_cal.pc_flattened.mean(axis=0))
print(det_cal.pc_flattened.std(0))
```

```
PC found: [********* ] 7/7  global best:0.125  PC opt:[0.4276 0.2215 0.5109]]
[0.42161222 0.2159899  0.50226706]
[0.00845582 0.00906874 0.00723199]
```

Compare the distribution of PCs to the above plotted map locations (especially PCx vs. PCy)

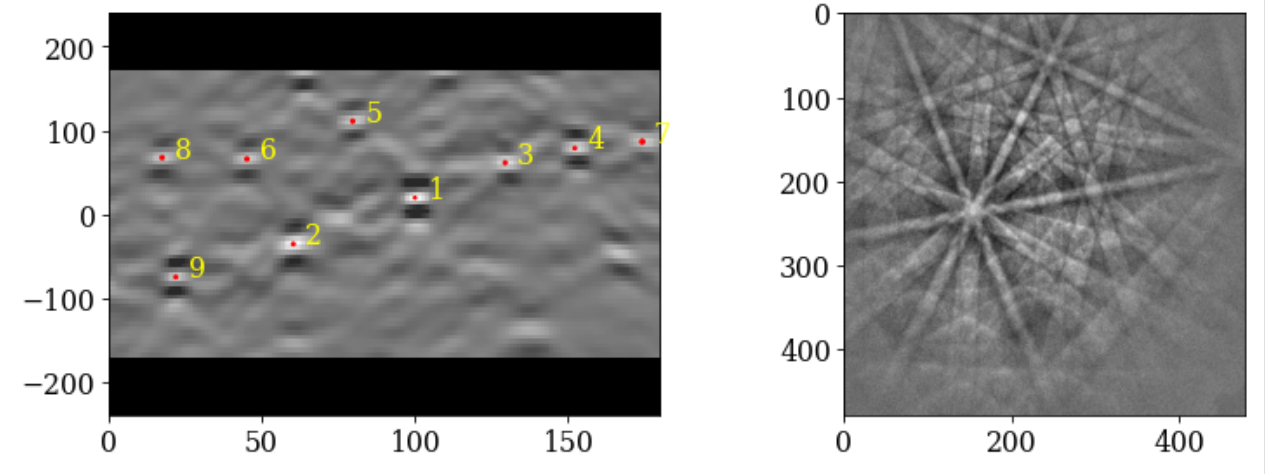
[15]:
```
det_cal.plot_pc("scatter", annotate=True)
```




We see no direct correlation between the sample positions and the PCs. Let's index the calibration patterns using these PCs and compare the solutions' band positions to the actual bands. We update our indexer instance with the estimated PCs.

[16]:
```
indexer.PC = det_cal.pc
xmap_cal = s_cal.hough_indexing(
    phase_list=phase_list, indexer=indexer, verbose=2
)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: True
  Projection center (Bruker, mean): (0.4216, 0.216, 0.5023)
  Indexing 7 pattern(s) in 1 chunk(s)
Radon Time: 0.04019342199899256
Convolution Time: 0.0054605890036327764
Peak ID Time: 0.0027515019974089228
Band Label Time: 0.04485934600234032
Total Band Find Time: 0.09329523499764036
Band Vote Time:  0.011680042996886186
  Indexing speed: 54.47269 patterns/s
```




Check indexed orientations by plotting geometrical simulations on top of the patterns. See the *tutorial on geometrical EBSD simulations* for details.

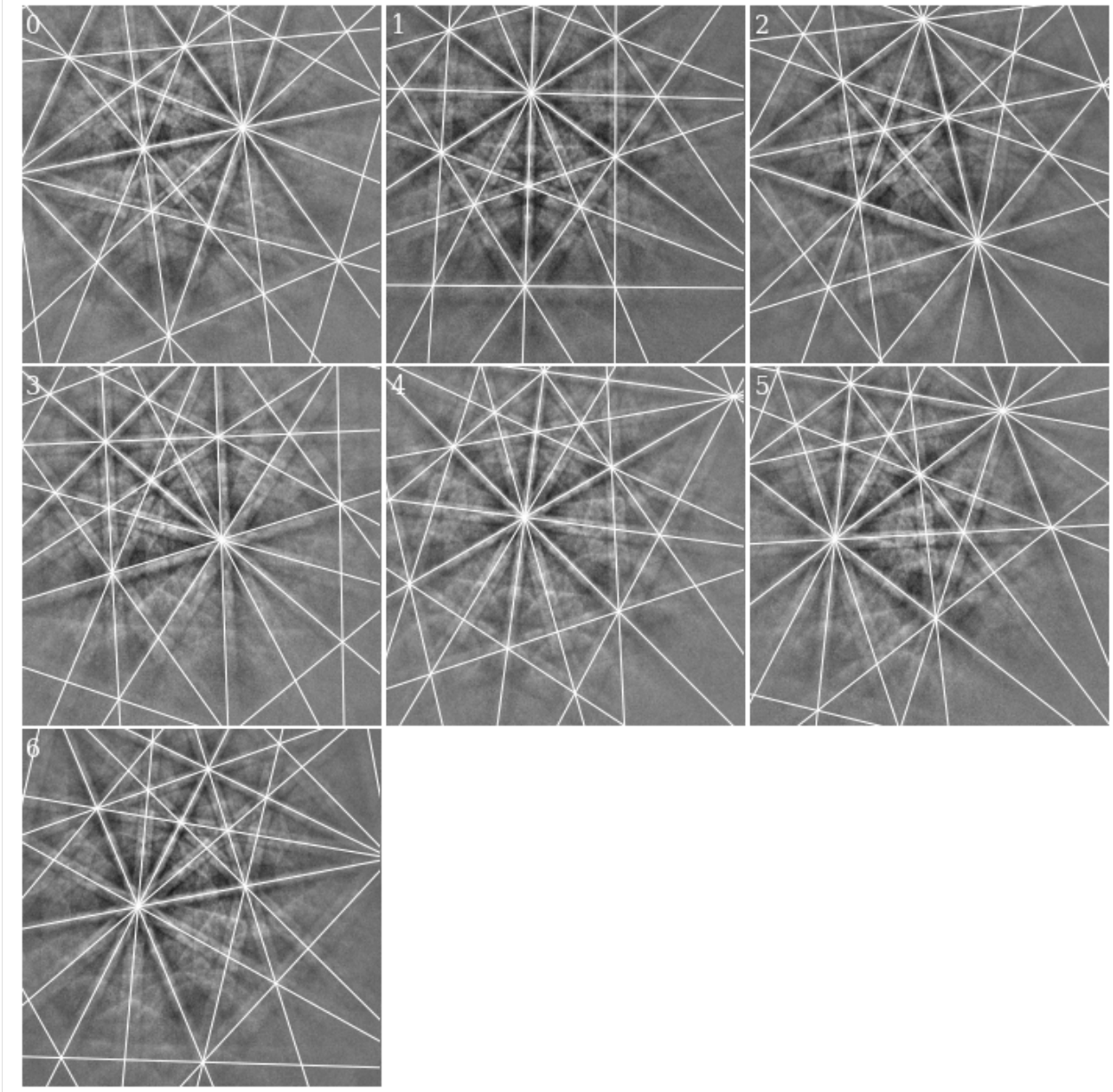
[17]:
```
ref = ReciprocalLatticeVector.from_min_dspacing(phase.deepcopy(), 0.07)
ref.sanitise_phase()  # "Fill atoms in unit cell", required for structure factor
ref.calculate_structure_factor()
structure_factor = abs(ref.structure_factor)
ref = ref[structure_factor > 0.12 * structure_factor.max()]
ref.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    0.203     0.4      0.2      100.0      8
 2 0 0    0.176     0.3      0.1       60.9      6
 2 2 0    0.125     0.1      0.0        8.7      12
 3 1 1    0.106     0.1      0.0        2.0      24
```

[18]:
```
simulator = kp.simulations.KikuchiPatternSimulator(ref)
sim_cal = simulator.on_detector(det_cal, xmap_cal.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 109.65 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.32 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 108.70 ms
```

[19]:
```
fig, axes = plt.subplots(ncols=3, nrows=3, figsize=(12, 12))
axes = axes.ravel()
for i in range(xmap_cal.size):
    axes[i].imshow(s_cal.inav[i].data, cmap="gray")
    lines = sim_cal.as_collections(i, lines_kwargs=dict(color="w"))[0]
    axes[i].add_collection(lines)
    axes[i].text(5, 10, i, c="w", va="top", ha="left")
_ = [ax.axis("off") for ax in axes]
fig.subplots_adjust(wspace=0.01, hspace=0.01)
```




Most lines align quite well with the bands. Some of them, especially in the lower part of the patterns and on wide bands, do not follow the band center, though (see e.g. pattern 4). Let's refine these solutions using dynamical simulations

[20]:
```
xmap_cal_ref, det_cal_ref = s_cal.refine_orientation_projection_center(
    xmap=xmap_cal,
    detector=det_cal,
    master_pattern=mp,
    energy=20,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5, 0.1, 0.1, 0.1],  # Sufficiently wide
    rtol=1e-5,
    # One pattern per iteration to utilize all CPUs
    chunk_kwargs=dict(chunk_shape=1),
```

(continues on next page)

(continued from previous page)

```
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5.  5.  5.  0.1 0.1 0.1]
  Relative tolerance: 1e-05
Refining 7 orientation(s) and projection center(s):
[########################################] | 100% Completed | 32.81 ss
Refinement speed: 0.21331 patterns/s
```

Check quality metrics

[21]:
```
print(xmap_cal_ref.scores.mean())
print(xmap_cal_ref.num_evals.mean())
```

```
0.5028477609157562
283.2857142857143
```

Check deviations from Hough indexed solutions (it is important that these deviations are well within our trust region above)

[22]:
```
angles_cal = xmap_cal.orientations.angle_with(
    xmap_cal_ref.orientations, degrees=True
)
pc_dev_cal = det_cal.pc_flattened - det_cal_ref.pc_flattened

print(angles_cal)
print(abs(pc_dev_cal).max(0))
```

```
[0.85406578 0.64784226 3.07477977 1.37188787 0.52298959 1.32954753
 1.1562491 ]
[0.00991018 0.01902905 0.0121604 ]
```

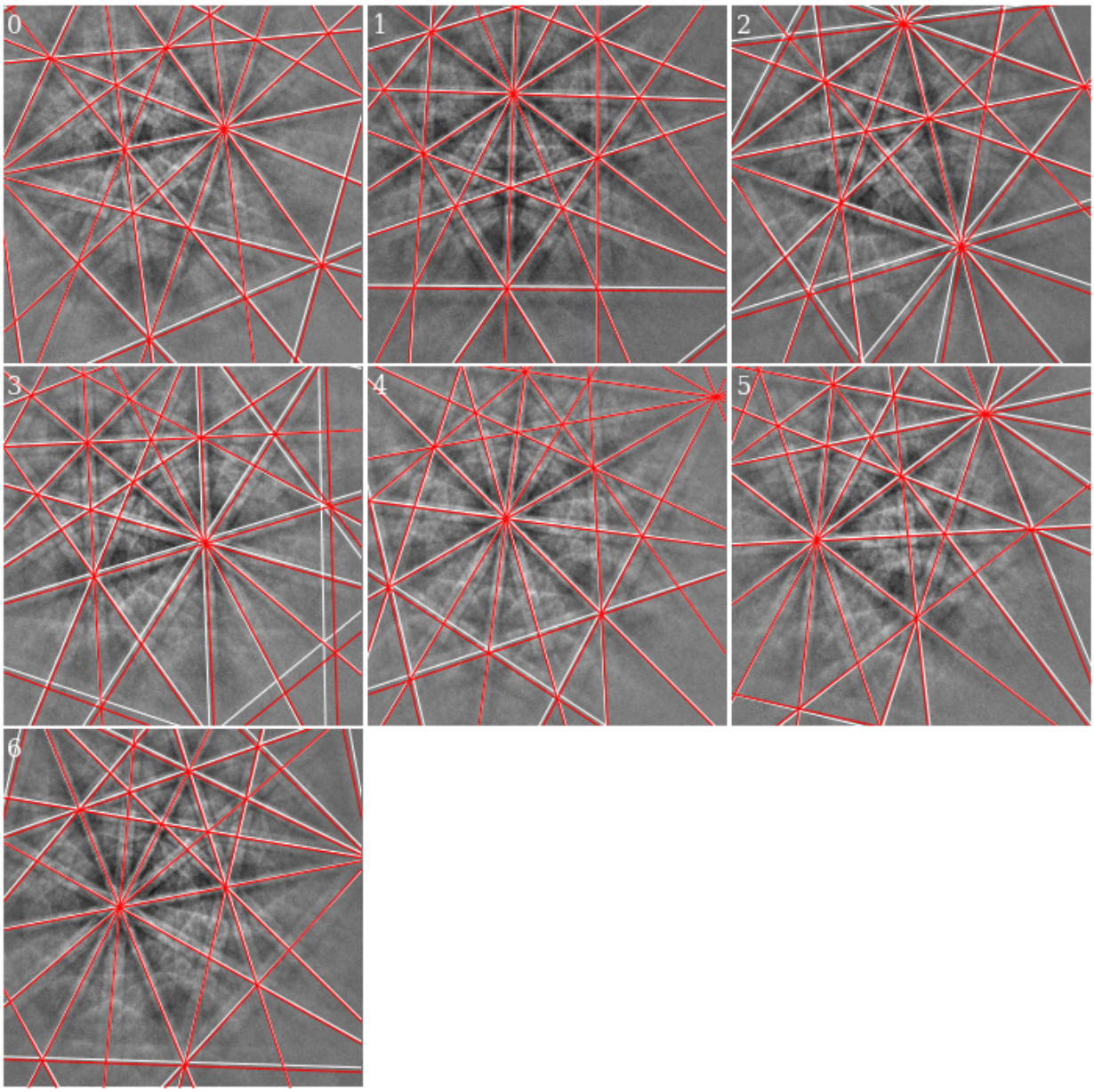
Get geometrical simulations from refined orientations and PCs and add lines from these simulations (in red) to the existing figure

[23]:
```
sim_cal_ref = simulator.on_detector(det_cal_ref, xmap_cal_ref.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.77 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.31 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.42 ms
```

[24]:
```
for i in range(xmap_cal_ref.size):
    lines = sim_cal_ref.as_collections(i)[0]
    axes[i].add_collection(lines)
fig
```

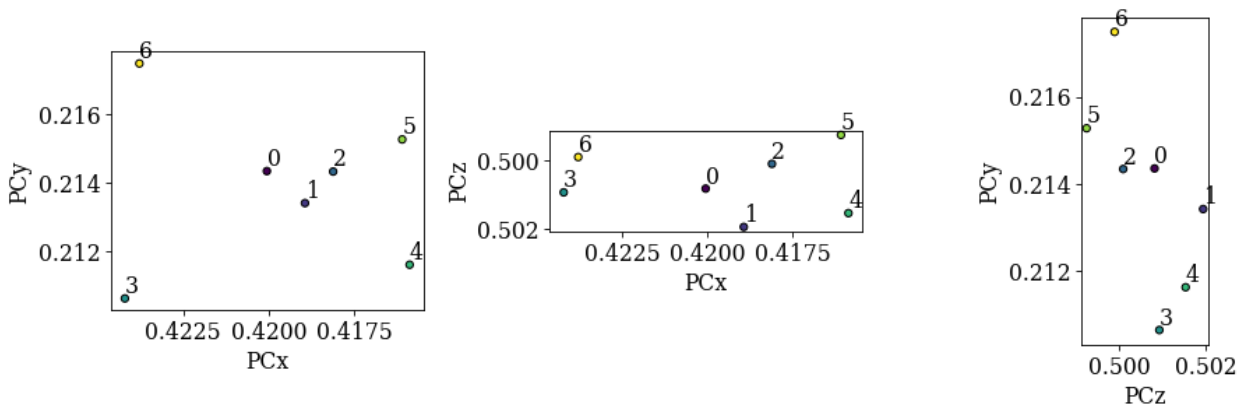
[24]:




We see that the red lines align better with wider bands and bands in the lower part of the patterns (where the deviations are greater).

Check the refined PCs

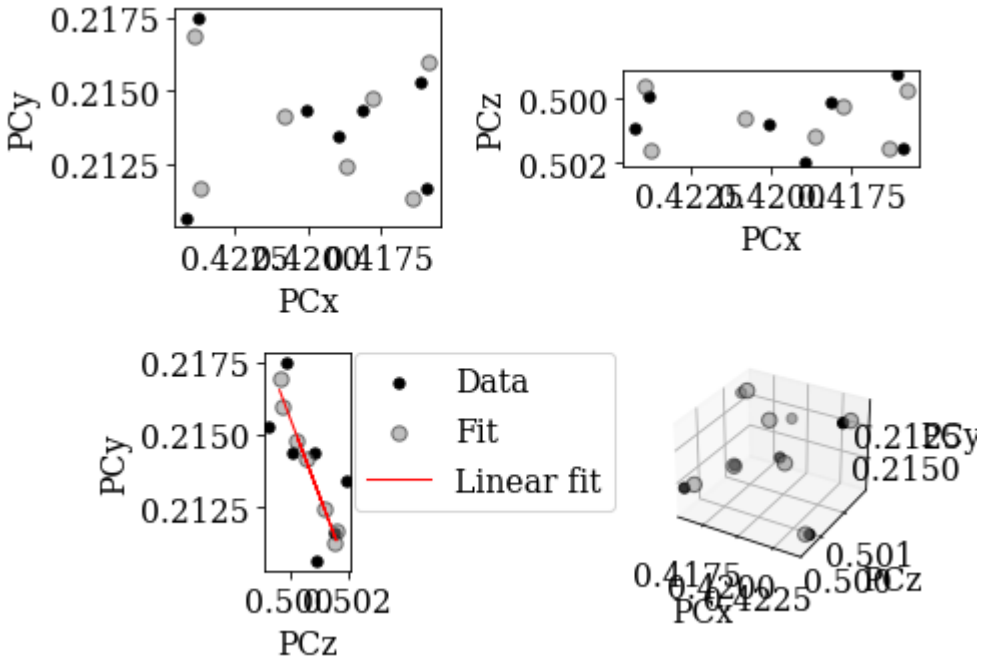
[25]:
```
det_cal_ref.plot_pc("scatter", annotate=True)
```




We now see that the patterns align quite well compared to the sample positions. We will therefore attempt to fit a plane to these PCs using an affine transformation (see the tutorial on *PC plane fitting* for more details). Note that if the PCs hadn't aligned as nicely as here, we should instead extrapolate a plane of PCs from an average; this procedure is detailed in *another tutorial*.

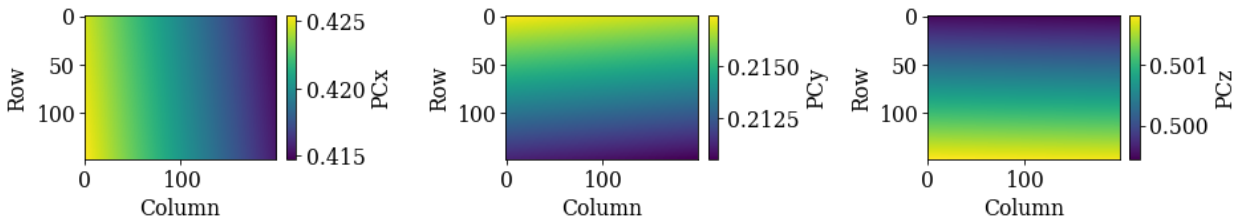
[26]:
```
pc_indices = omd.calibration_patterns.indices_scaled.copy()
pc_indices -= omd.roi.origin_scaled
pc_indices = pc_indices.T

det_cal_fit = det_cal_ref.fit_pc(
    pc_indices,
    map_indices=np.indices(s.axes_manager.navigation_shape[::-1]),
    transformation="affine",
)
print(det_cal_fit.pc_average)

# Sample tilt
print(det_cal_fit.sample_tilt)

# Max. deviation between experimental and fitted PC
pc_diff_fit = det_cal_ref.pc - det_cal_fit.pc[tuple(pc_indices)]
print(abs(pc_diff_fit.reshape(-1, 3)).mean(axis=0))
```

```
[0.42006255 0.21396137 0.50062654]
69.29360786283358
[0.00040465 0.00061294 0.00037163]
```




Check the plane of PCs

[27]:
```
det_cal_fit.plot_pc()
```




As a final validation of this plane of PCs, we will refine the (refined) orientations using a fixed PC for each pattern, taken from the plane of PCs

[28]:
```
det_cal_fit2 = det_cal_fit.deepcopy()
det_cal_fit2.pc = det_cal_fit2.pc[tuple(pc_indices)]
```

[29]:
```
xmap_cal_ref2 = s_cal.refine_orientation(
    xmap=xmap_cal_ref,
    detector=det_cal_fit2,
    master_pattern=mp,
    energy=20,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5],
    chunk_kwargs=dict(chunk_shape=1),
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
```

(continues on next page)

(continued from previous page)

```
  Relative tolerance: 0.0001
Refining 7 orientation(s):
[########################################] | 100% Completed | 4.90 sms
Refinement speed: 1.42526 patterns/s
```

[30]:
```
print(xmap_cal_ref2.scores.mean())
print(xmap_cal_ref2.num_evals.mean())
```

```
0.5027158260345459
59.0
```

[31]:
```
angles_cal2 = xmap_cal_ref.orientations.angle_with(
    xmap_cal_ref2.orientations, degrees=True
)
print(angles_cal2)
```

```
[0.73723683 0.82276047 0.66917407 0.60292755 0.73111167 0.62122844
 0.76997895]
```

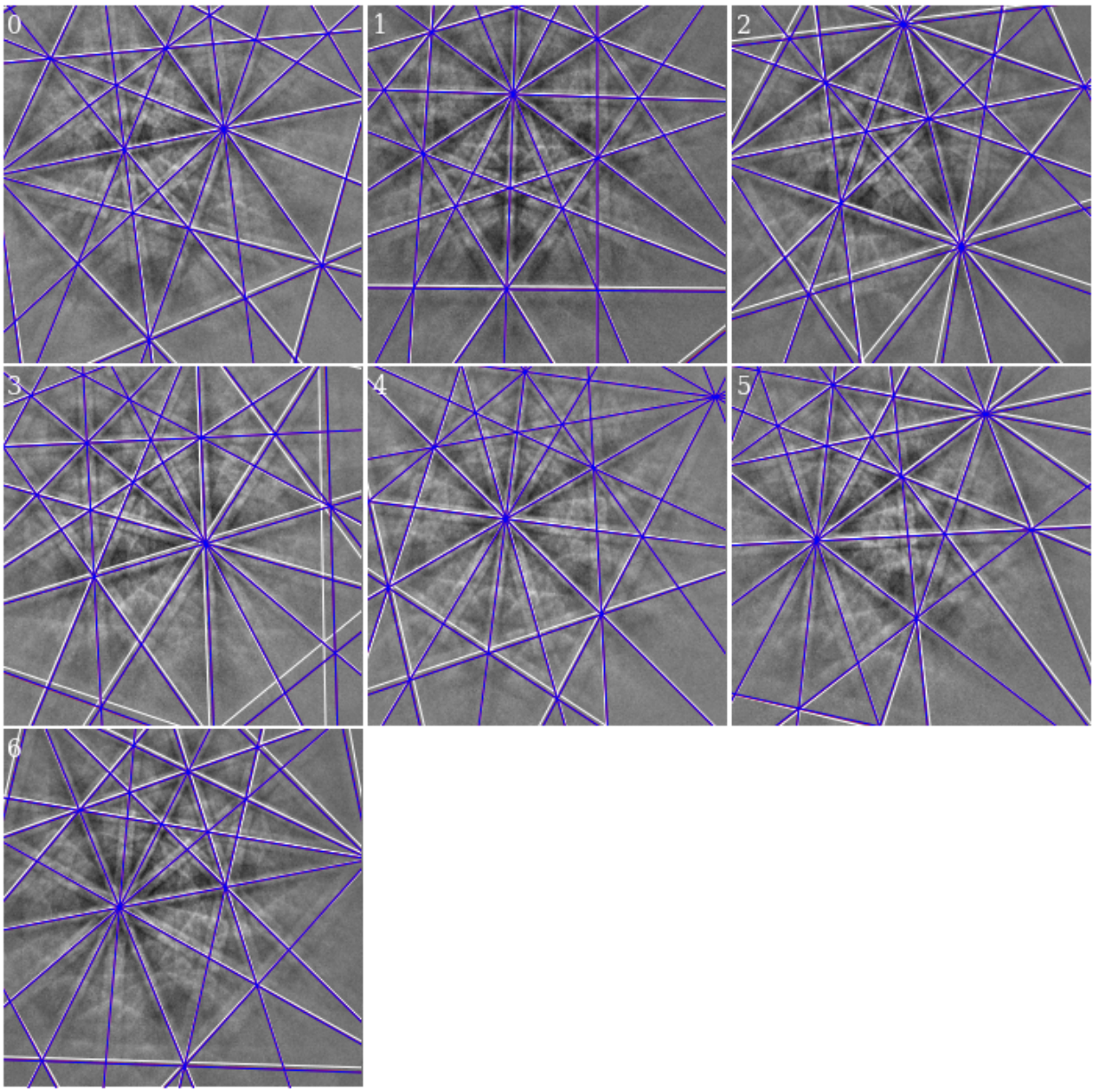
Get geometrical simulations and add a third set of lines (in blue) to the existing figure

[32]:
```
sim_cal_ref2 = simulator.on_detector(det_cal_fit2, xmap_cal_ref2.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.02 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.63 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.93 ms
```

[33]:
```
for i in range(xmap_cal_ref2.size):
    lines = sim_cal_ref2.as_collections(i, lines_kwargs=dict(color="b"))[0]
    axes[i].add_collection(lines)
fig
```

[33]:




### Hough indexing of all patterns

Now that we are confident of our geometry calibration, we can index all patterns in our noisy experimental dataset.

Copy the detector with the calibrated PCs and update the detector shape to match our experimental patterns

[34]:
```
det = det_cal_fit.deepcopy()
det.shape = s.detector.shape
det
```

[34]:
```
EBSDDetector (60, 60), px_size 1 um, binning 1, tilt 0.0, azimuthal 0, pc (0.42, 0.214,␣
↪0.501)
```

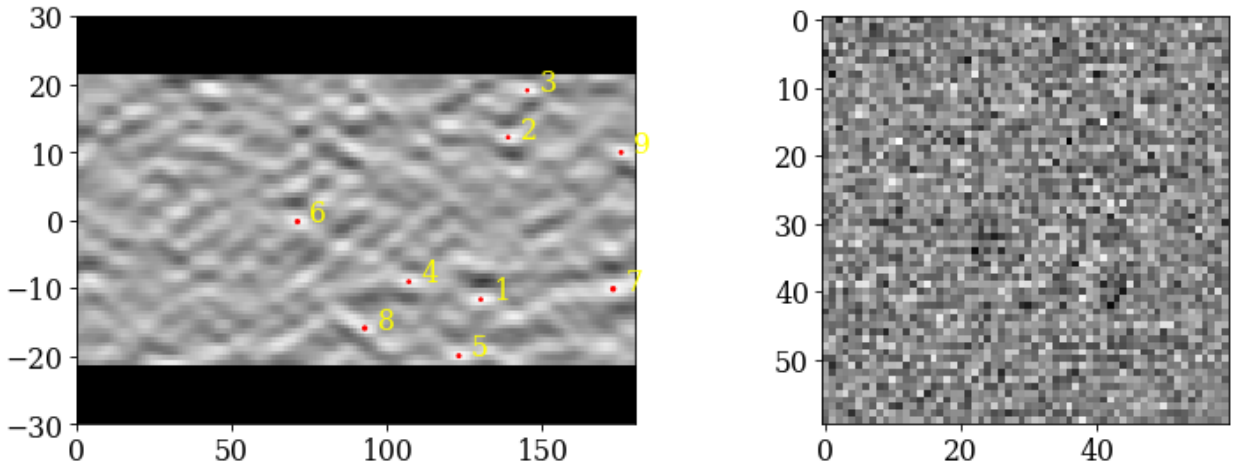
Get a new indexer

[35]:
```
indexer = det.get_indexer(phase_list, rSigma=2, tSigma=2)
```

Perform Hough indexing with PyEBSDIndex (using the GPU via PyOpenCL, but only a single CPU)

[36]:
```
xmap_hi = s.hough_indexing(phase_list=phase_list, indexer=indexer, verbose=2)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: True
  Projection center (Bruker, mean): (0.4201, 0.214, 0.5006)
  Indexing 29800 pattern(s) in 57 chunk(s)
Radon Time: 2.1906664220732637
Convolution Time: 3.5428661539772293
Peak ID Time: 2.6689383829798317
Band Label Time: 4.550726840039715
Total Band Find Time: 12.953891986995586
Band Vote Time:  17.05054735599697
  Indexing speed: 989.48539 patterns/s
```




Our use of PyEBSDIndex here gave indexing of about 900 - 1 000 patterns/s.

**Note**

Note that PyEBSDIndex can index a lot faster than this by using more CPUs and by passing a file directly (not via NumPy or Dask arrays, as done here) to its indexing functions. See its documentation for details: https://pyebsdindex.readthedocs.io/en/latest.

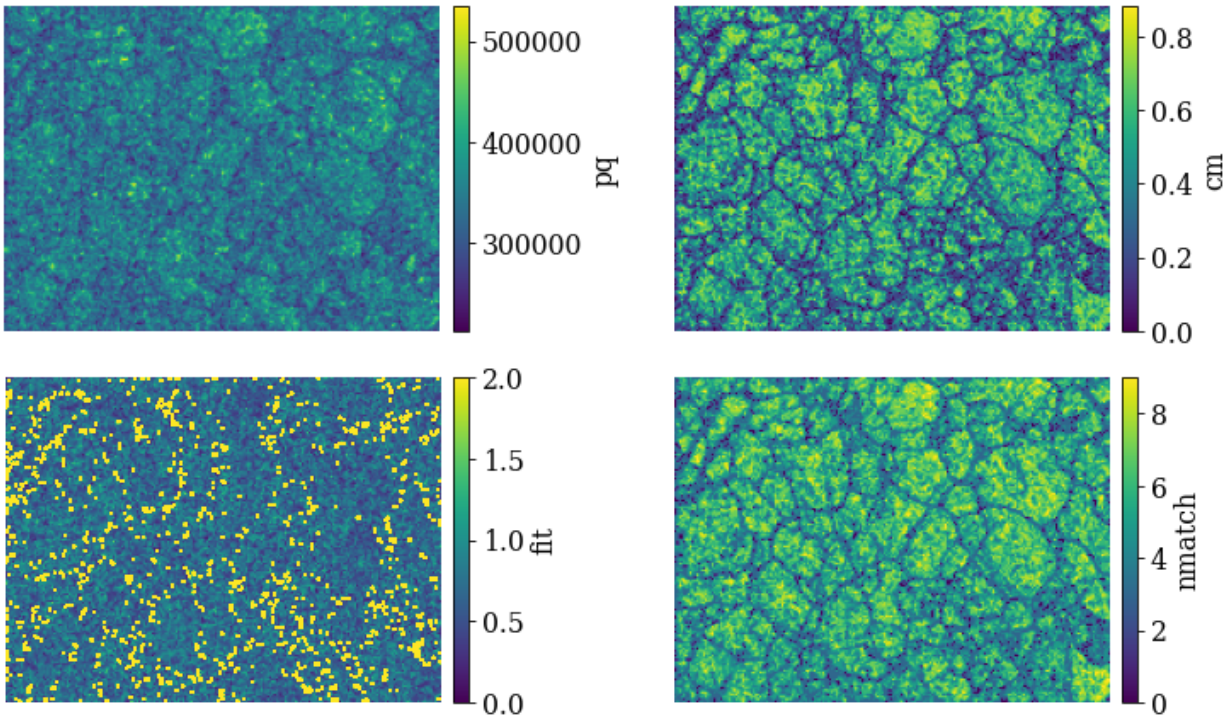
[37]:
```
xmap_hi
```

[37]:
```
Phase    Orientations         Name  Space group  Point group  Proper point group    ␣
↪Color
   -1     1096 (3.7%)  not_indexed         None         None                None        ␣
↪W
    0   28704 (96.3%)           ni        Fm-3m        m-3m                 432  tab:
↪blue
Properties: fit, cm, pq, nmatch
Scan unit: um
```

[38]:
```
# Save HI map
# io.save("xmap_hi.ang", xmap_hi)
# io.save("xmap_hi.h5", xmap_hi)
```

PyEBSDIndex could not index some 4% of patterns (too high pattern fit). Let's check the quality metrics (pattern fit, confidence metric, pattern quality, and the number of detected bands that were assigned an index ["indexed"])

[39]:
```
aspect_ratio = xmap_hi.shape[1] / xmap_hi.shape[0]
figsize = (8 * aspect_ratio, 4.5 * aspect_ratio)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=figsize, layout="tight")
for ax, to_plot in zip(axes.ravel(), ["pq", "cm", "fit", "nmatch"]):
    if to_plot == "fit":
        im = ax.imshow(xmap_hi.get_map_data(to_plot), vmin=0, vmax=2)
    else:
        im = ax.imshow(xmap_hi.get_map_data(to_plot))
    fig.colorbar(im, ax=ax, label=to_plot, pad=0.02)
    ax.axis("off")
```



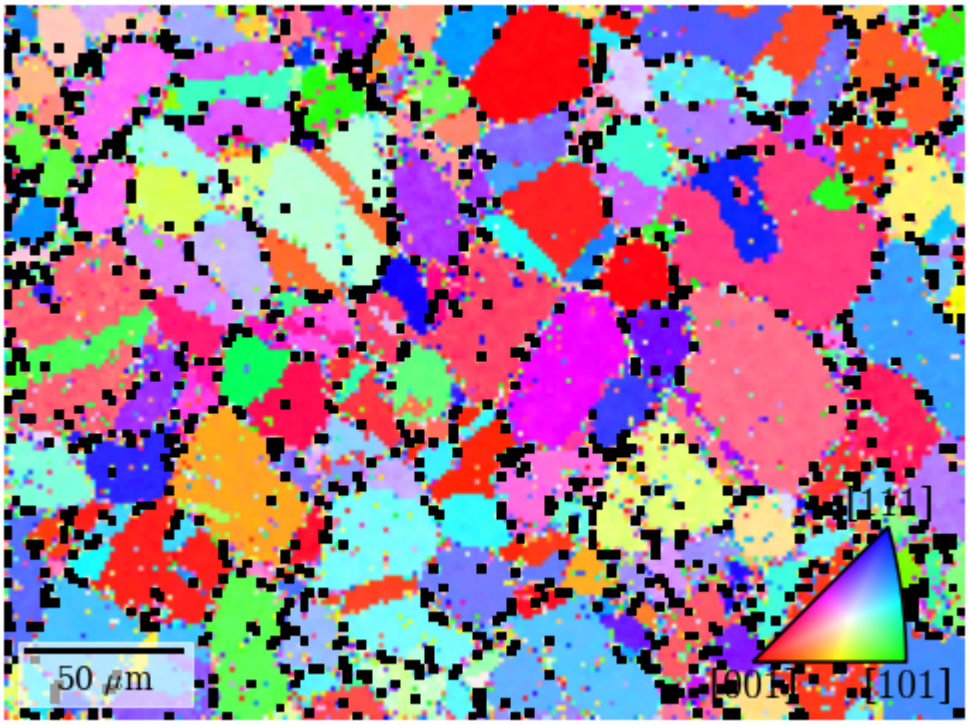

The bright points in the lower left pattern fit map are the points considered not indexed. The confidence metric and number of successfully labeled bands (out of nine) seem to be highest within grains and lowest at grain boundaries. Let's inspect the spatial variation of "successfully" indexed orientations in an inverse pole figure map (IPF-X)

[40]:
```
pg = xmap_hi.phases[0].point_group
ckey = plot.IPFColorKeyTSL(pg, Vector3d([1, 0, 0]))
ckey
```

[40]:
```
IPFColorKeyTSL, symmetry: m-3m, direction: [1 0 0]
```

[41]:
```
rgb_hi = ckey.orientation2color(xmap_hi["indexed"].rotations)
fig = xmap_hi["indexed"].plot(
    rgb_hi, remove_padding=True, return_figure=True
)

# Place color key in bottom right corner, coordinates are
# [left, bottom, width, height]
ax_ckey = fig.add_axes(
    [0.76, 0.08, 0.2, 0.2], projection="ipf", symmetry=pg
)
ax_ckey.plot_ipf_color_key(show_title=False)
ax_ckey.patch.set_facecolor("None")
```

```
/home/hakon/miniconda3/envs/kp-dev/lib/python3.11/site-packages/matplotlib/cm.py:478:␣
↪RuntimeWarning: invalid value encountered in cast
  xx = (xx * 255).astype(np.uint8)
```

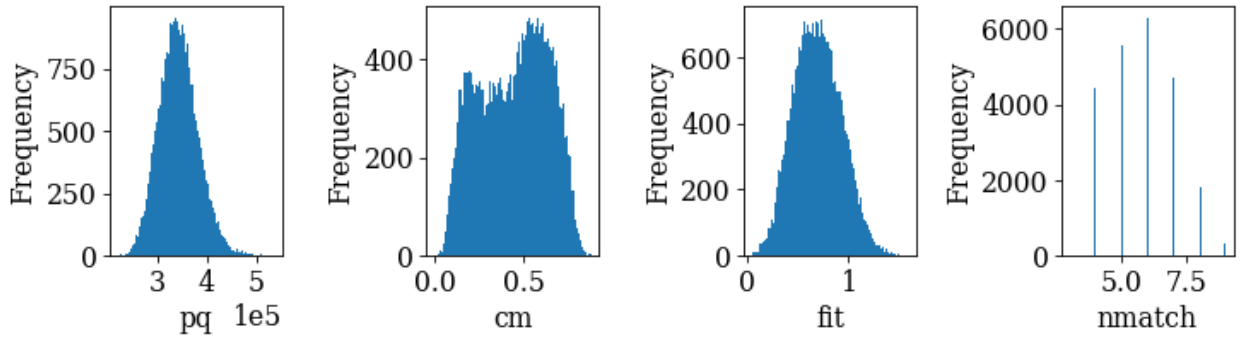



Many points seen as single color deviations from otherwise smooth colors within recrystallized grains are located mostly at grain boundaries.

### Identify points for re-indexing

Let's see if we can easily separate the good from bad points using any of the quality metrics

[42]:
```
fig, axes = plt.subplots(ncols=4, figsize=(10, 3), layout="tight")
for ax, to_plot in zip(axes.ravel(), ["pq", "cm", "fit", "nmatch"]):
    _ = ax.hist(xmap_hi["indexed"].prop[to_plot], bins=100)
    ax.set(xlabel=to_plot, ylabel="Frequency")
    if to_plot == "pq":
        ax.ticklabel_format(axis="x", style="sci", scilimits=(0, 0))
```




… hm, there is no clear bimodal distribution in any of the histograms. In such cases, one solution is to find the "good" and "bad" points by trial-and-error until a desired separation is achieved. Here, we have combined metrics by trial-and-error to get a plausible separation. Note that other combinations might be better for other datasets.

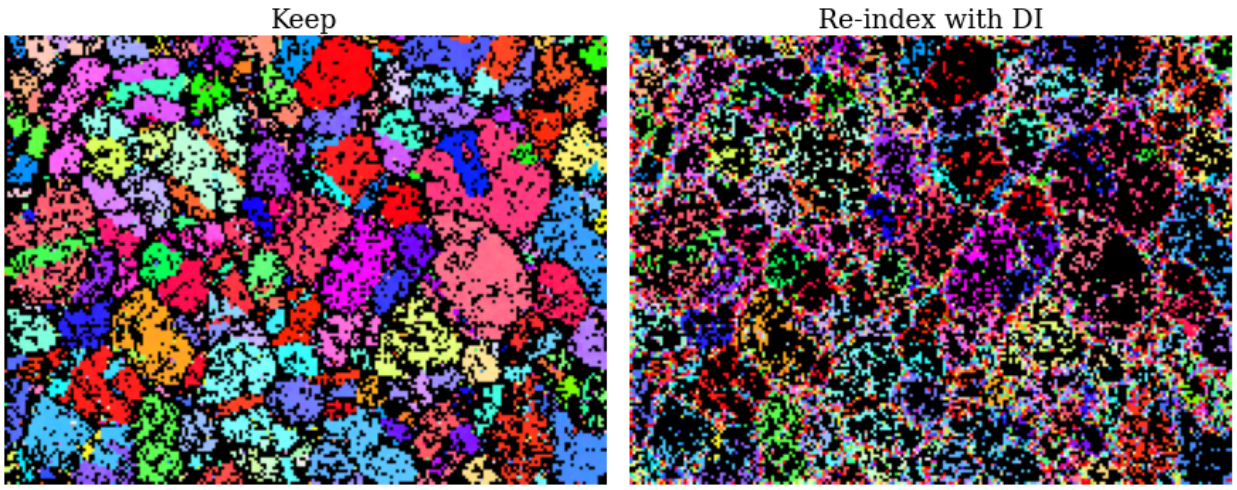
[43]:
```
mask_reindex = np.logical_or.reduce(
    (
        ~xmap_hi.is_indexed,
        xmap_hi.fit > 0.9,
        xmap_hi.nmatch < 4,
        xmap_hi.cm < 0.25,
    )
)
frac_reindex = mask_reindex.sum() / mask_reindex.size
print(f"Fraction to re-index: {100 * frac_reindex:.2f}%")

# Get colors for all points, even the ones considered not-indexed
rgb_hi_all = ckey.orientation2color(xmap_hi.rotations)

# Get separate arrays for points to keep and to re-index, with in the other
# array as black
rgb_hi_reindex = np.zeros((xmap_hi.size, 3))
rgb_hi_keep = np.zeros_like(rgb_hi_reindex)
rgb_hi_reindex[mask_reindex] = rgb_hi_all[mask_reindex]
rgb_hi_keep[~mask_reindex] = rgb_hi_all[~mask_reindex]

nav_shape = xmap_hi.shape + (3,)

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(12, 5), layout="tight")
ax0.imshow(rgb_hi_keep.reshape(nav_shape))
ax1.imshow(rgb_hi_reindex.reshape(nav_shape))
for ax, title in zip([ax0, ax1], ["Keep", "Re-index with DI"]):
```

(continues on next page)

(continued from previous page)

```
    ax.axis("off")
    ax.set(title=title)
```

```
Fraction to re-index: 43.35%
```




There are some spurious points still left among the points to keep. Otherwise, the inverse pole figure map looks fairly convincing.

Make a 2D navigation mask where points to re-index are set to `False`.

[44]:
```
nav_mask = np.ones(xmap_hi.size, dtype=bool)
nav_mask[mask_reindex] = False
nav_mask = nav_mask.reshape(xmap_hi.shape)
```

### Re-indexing with dictionary indexing

To generate the dictionary of nickel patterns, we need to sample orientation space at a sufficiently high resolution (here 2°) with a fixed calibration geometry (PC). See the *pattern matching tutorial* for details.

[45]:
```
rot = sampling.get_sample_fundamental(resolution=2, point_group=pg)
rot
```

[45]:
```
Rotation (100347,)
[[ 0.8541 -0.3536 -0.3536 -0.1435]
 [ 0.8541 -0.3536 -0.3536  0.1435]
 [ 0.8541 -0.3536 -0.1435 -0.3536]
 ...
 [ 0.8541  0.3536  0.1435  0.3536]
 [ 0.8541  0.3536  0.3536 -0.1435]
 [ 0.8541  0.3536  0.3536  0.1435]]
```

[46]:
```
det_pc1 = det.deepcopy()
det_pc1.pc = det_pc1.pc_average

det_pc1.pc
```

[46]:
```
array([[0.42006255, 0.21396137, 0.50062654]])
```

[47]:
```
sim = mp.get_patterns(
    rotations=rot,
    detector=det_pc1,
    energy=20,
    chunk_shape=rot.size // 20,
)
sim
```

[47]:
```
<LazyEBSD, title: , dimensions: (100347|60, 60)>
```

We only match the intensities within a circular mask (note the inversion!)

[48]:
```
signal_mask = kp.filters.Window("circular", det.shape).astype(bool)
signal_mask = ~signal_mask
```

Perform dictionary indexing of the patterns and intensities marked as `False` in the navigation and signal masks

[49]:
```
xmap_di = s.dictionary_indexing(
    sim,
    keep_n=1,
    navigation_mask=nav_mask,
    signal_mask=signal_mask,
)
```

```
Dictionary indexing information:
  Phase name: ni
  Matching 12918/29800 experimental pattern(s) to 100347 dictionary pattern(s)
  NormalizedCrossCorrelationMetric: float32, greater is better, rechunk: False,␣
↪navigation mask: True, signal mask: True
```

```
100%|----------------------| 21/21 [01:32<00:00,  4.40s/it]
```

```
  Indexing speed: 139.90687 patterns/s, 14039234.31546 comparisons/s
```

[50]:
```
# Save DI map
# io.save("xmap_di.h5", xmap_di)
```

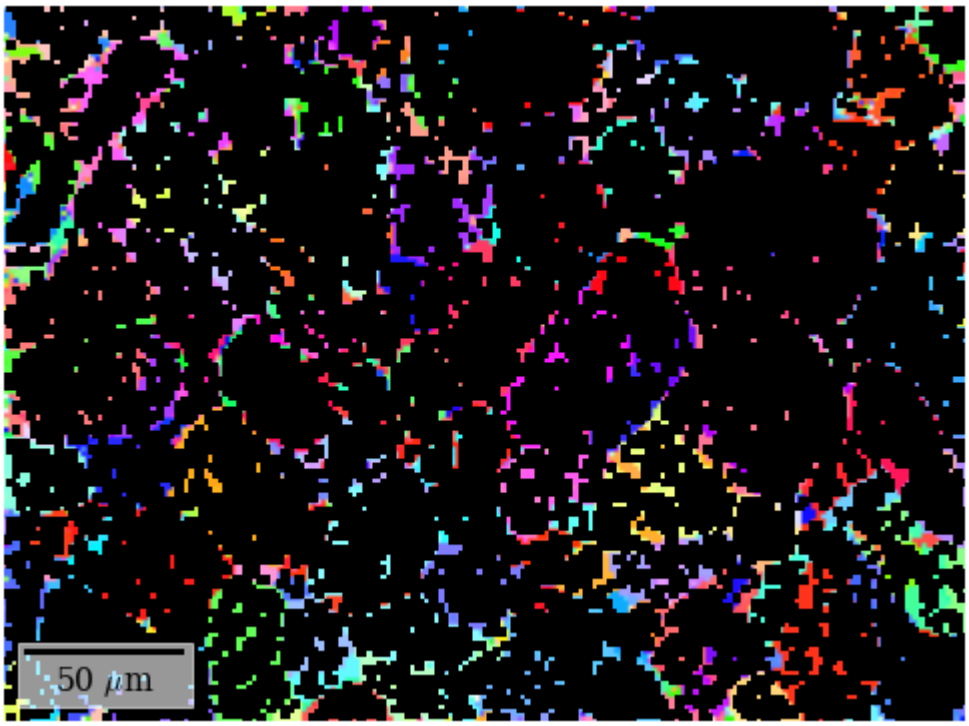
We see that HI is about 6-8x faster than DI.

[51]:
```
xmap_di.scores.mean()
```

[51]:
```
0.16163187
```

[52]:
```
rgb_di = ckey.orientation2color(xmap_di.rotations)
xmap_di.plot(rgb_di, remove_padding=True)
```

```
/home/hakon/miniconda3/envs/kp-dev/lib/python3.11/site-packages/matplotlib/cm.py:478:␣
↪RuntimeWarning: invalid value encountered in cast
  xx = (xx * 255).astype(np.uint8)
```




An average correlation score of about 0.15 is low but OK, since we can refine the solutions and would expect a higher score from this. The IPF-Z map looks plausible, though, which it did not from these patterns after HI.

### Refine Hough indexed and dictionary indexed points

First we specify common refinement parameters, so that the scores obtain can be compared. This is *very* important!

[53]:
```
ref_kw = dict(
    detector=det,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5],
)
```

Of the Hough indexed solutions, we only want to refine those that are not re-indexed using dictionary indexing. We therefore pass the navigation mask, but have to take care to set those points that we want to index to `False`

[54]:
```
xmap_hi_ref = s.refine_orientation(
    xmap=xmap_hi, navigation_mask=~nav_mask, **ref_kw
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
  Relative tolerance: 0.0001
Refining 16882 orientation(s):
```

(continues on next page)

(continued from previous page)

```
[########################################] | 100% Completed | 122.58 s
Refinement speed: 137.63313 patterns/s
```

[55]:
```
print(xmap_hi_ref.scores.mean())
print(xmap_hi_ref.num_evals.max())
```

```
0.24417378626987354
144
```

An average correlation score of about 0.24 is OK.

We now refine the re-indexed points. No navigation mask is necessary, since the crystal map returned from DI has a mask keeping track of which points are "in the data" via CrystalMap.is_in_data.

[56]:
```
xmap_di_ref = s.refine_orientation(xmap=xmap_di, **ref_kw)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
  Relative tolerance: 0.0001
Refining 12918 orientation(s):
[########################################] | 100% Completed | 94.58 ss
Refinement speed: 136.53398 patterns/s
```

[57]:
```
print(xmap_di_ref.scores.mean())
print(xmap_di_ref.num_evals.max())
```

```
0.20418422812439374
190
```

An average correlation score of about 0.20 is still OK.

### Merge results

We can now merge the results, taking care to pass the navigation mask where each refined map should be considered. Since only the points not in the refined HI map were indexed with DI, the same mask can be used in both cases.

[58]:
```
xmap_ref = kp.indexing.merge_crystal_maps(
    [xmap_hi_ref, xmap_di_ref], navigation_masks=[~nav_mask, nav_mask]
)
```

[59]:
```
xmap_ref
```

[59]:
```
Phase    Orientations  Name  Space group  Point group  Proper point group     Color
    0  29800 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: scores, merged_scores
Scan unit: um
```

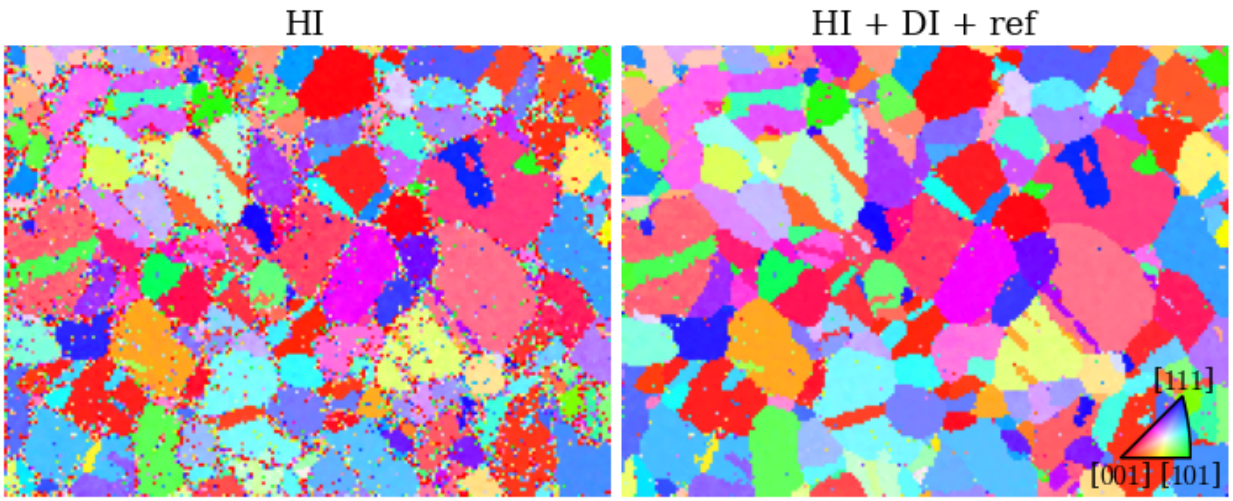
We see that we have a complete map for all our points!

[60]:
```
# Save final refined combined map
# io.save("xmap_ref.ang", xmap_ref)
# io.save("xmap_ref.h5", xmap_ref)
```

### Validate indexing results

Finally, we can compare the IPF-X maps with HI only and after re-indexing, refinement and combination

[61]:
```
rgb_ref = ckey.orientation2color(xmap_ref.orientations)
rgb_shape = xmap_ref.shape + (3,)

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(12, 5))
ax0.imshow(rgb_hi_all.reshape(rgb_shape))
ax1.imshow(rgb_ref.reshape(rgb_shape))
for ax, title in zip([ax0, ax1], ["HI", "HI + DI + ref"]):
    ax.axis("off")
    ax.set(title=title)

ax_ckey = fig.add_axes(
    [0.805, 0.21, 0.1, 0.1], projection="ipf", symmetry=pg
)
ax_ckey.plot_ipf_color_key(show_title=False)
ax_ckey.patch.set_facecolor("None")
_ = [t.set_fontsize(15) for t in ax_ckey.texts]

fig.subplots_adjust(wspace=0.02)
```

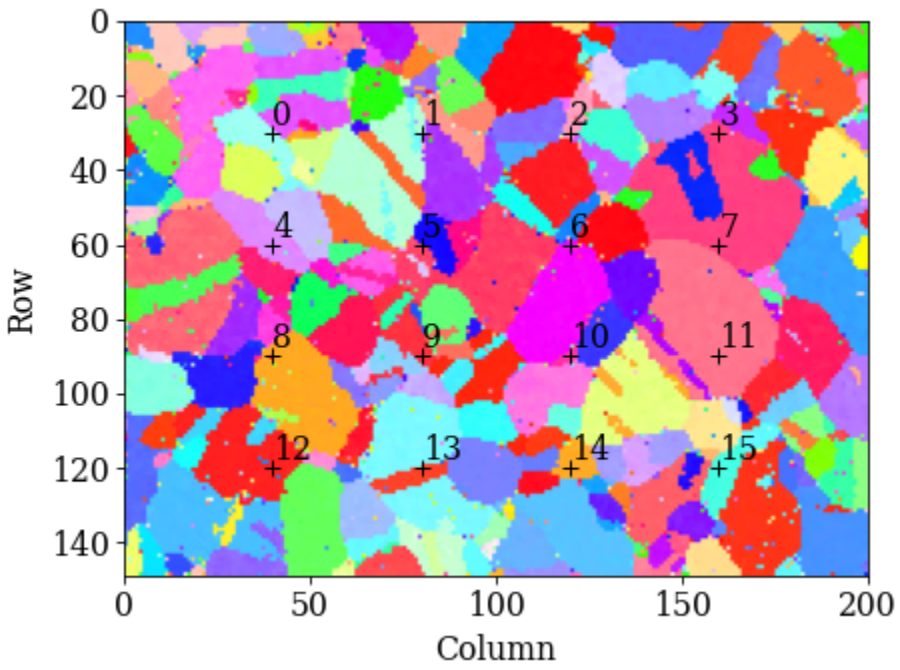



We extract a grid of patterns and plot the geometrical simulations on top of these patterns

[62]:
```
s.xmap = xmap_ref
s.detector = det
```

[63]:
```
grid_shape = (4, 4)
s_grid, idx = s.extract_grid(grid_shape, return_indices=True)
```

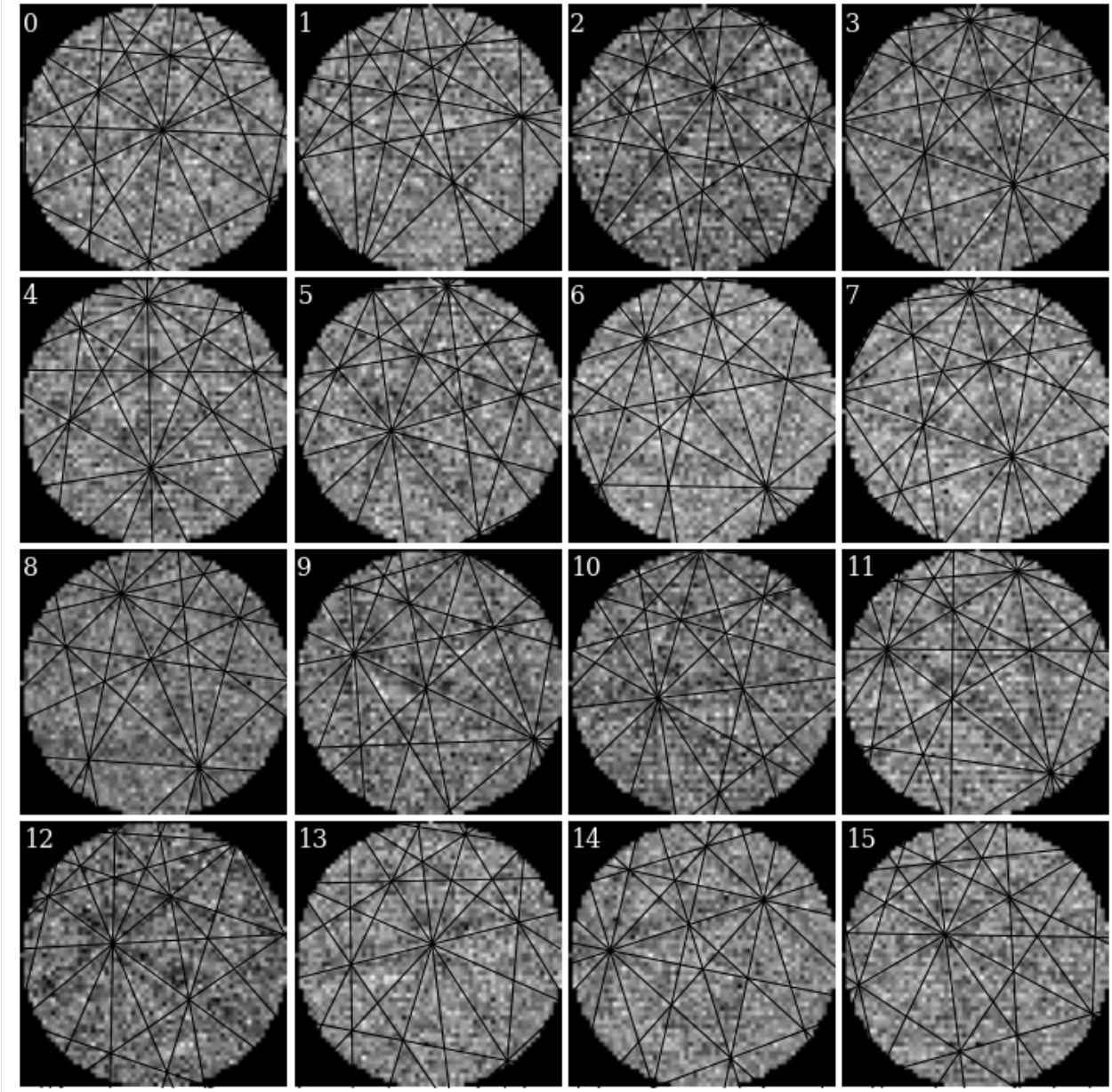
[64]:
```
kp.draw.plot_pattern_positions_in_map(
    rc=idx.reshape((2, -1)).T,
    roi_shape=xmap_ref.shape,
    roi_image=rgb_ref.reshape(rgb_shape),
)
```




[65]:
```
sim_grid = simulator.on_detector(
    s_grid.detector, s_grid.xmap.rotations.reshape(*grid_shape)
)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.64 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 102.45 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.67 ms
```

[66]:
```
fig, axes = plt.subplots(
    nrows=grid_shape[0], ncols=grid_shape[1], figsize=(12, 12)
)
for idx in np.ndindex(grid_shape):
    ax = axes[idx]
    ax.imshow(s_grid.data[idx] * ~signal_mask, cmap="gray")
    lines = sim_grid.as_collections(idx, lines_kwargs=dict(color="k"))[0]
    ax.add_collection(lines)
    ax.axis("off")
    ax.text(
        0,
        1,
        np.ravel_multi_index(idx, grid_shape),
        va="top",
        ha="left",
        c="w",
    )
fig.subplots_adjust(wspace=0.02, hspace=0.02)
```




It's difficult to see any bands in these *very* noisy patterns… There at least seems to be a correlation between darker regions in the patterns (not the corners) and zone axes, which is expected.

In conclusion, by combining the speed of Hough indexing with the robustness towards noise of dictionary indexing, a dataset can be indexed in a shorter time and achieve about the same results as with DI (and refinement) only.

### What's next?

Can we improve indexing results by improving further the signal-to-noise ratio in our very noisy EBSD patterns? Instead of the "naive" Gaussian kernel used in neighbour pattern averaging, we could try out a more sophisticated kernel with non-local pattern averaging (NLPAR) from `PyEBSDIndex`. More details are found in their NLPAR tutorial.

---

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

---

### Orientation-dependence of the projection center

In this tutorial, we will see that the error in the projection center (PC) estimated from pattern matching can be orientation-dependent. When finding an average PC to use for indexing, it is therefore important to average PCs from not only many patterns, but from many patterns from different grains as well, if possible.

The orientation-dependence of the PC error is nicely demonstrated by [Pang *et al.*, 2020]. They simulateneously optimize the orientation and PCs of experimental nickel patterns from an openly available dataset, released by [Jackson *et al.*, 2019]. To test their optimization routine, they compare optimized PCs to those expected from geometrical considerations.

When the dataset was acquired, the sample was tilted $\sigma = 75.7°$ towards the detector, while the detector was tilted $\theta = 10°$ away from the sample. These tilts give a combined $\alpha = 90° - \sigma + \theta$ tilt about the detector $X_d$ axis, which brings the sample normal parallel to the detector normal. A scan of (n rows, m columns) = (151, 181) patterns with a nominal step size of 1.5 um was acquired in a nominally regular grid on the sample. The sample $y$-direction increases "up the sample". Given these geometrical considerations, the PC is expected to change following the following equations:

$$\frac{PC_x}{\Delta y} = 1, \tag{1.6}$$

$$\frac{PC_y}{\Delta y} = \cos\alpha \cdot \frac{1}{\delta}, \tag{1.7}$$

$$\frac{PC_z}{\Delta y} = \sin\alpha. \tag{1.8}$$

Here, Bruker's PC convention is used (see the *reference frame tutorial*). $\delta$ is the detector pixel size. The detector used in this experiment has a pixel size of $\delta = 59.2$ um.

Pang and co-workers optimize the orientation solutions and PCs saved with the experimental data as determined from Hough indexing with EDAX OIM. In this tutorial, we do the following:

1. Obtain a good starting PC for the refinement:
    1. Optimize the PC of 49 patterns extracted in a grid from the full dataset using Hough indexing. We will use the EDAX OIM PC as the initial guess.
    2. Index the grid patterns using Hough indexing.
    3. Refine the orientations and PCs using pattern matching.
    4. Calculate an average PC using the reliably refined PCs.
2. Index all (151, 181) patterns using Hough indexing with the average PC.
3. Refine Hough indexed orientations and average PC using pattern matching. This is only be done for a vertical slice of the full dataset (the same slice used by Pang and co-workers). The slice has shape (151, 10).

To validate our results, we average the refined PCs along the horizontal (giving one PC per 151 vertical position) and compare them to the ones expected from the equations above.

Pang and co-workers use the global optimization algorithm SNOBFIT to optimize orientations and PCs simultaneously. Here, we will use the local optimization algorithm Nelder-Mead, as implemented in NLopt, and see that we obtain comparable results.

---

**Note**

To run this tutorial, the experimental nickel patterns must be downloaded from the supplementary material to [Jackson *et al.*, 2019].

Additionally, a high resolution [typically of (1001, 1001) pixels] nickel EBSD master pattern in the (square) Lambert projection is required. This can be simulated with e.g. EMsoft. Or, it can be downloaded via the *kikuchipy.data* module.

---

Let's import necessary libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit

from diffsims.crystallography import ReciprocalLatticeVector
import hyperspy.api as hs
import kikuchipy as kp
from orix import plot
from orix.crystal_map import PhaseList


plt.rcParams.update(
    {
        "figure.facecolor": "w",
        "figure.dpi": 75,
        "figure.figsize": (8, 8),
        "font.size": 15,
    }
)
```

### Load and inspect data

Load (lazily) the experimental nickel data from [Jackson *et al.*, 2019]

[2]:
```
s = kp.load("../../../../phd/data/ni/edax/ni/EDAX-Ni.h5", lazy=True)
s
```

[2]:
```
<LazyEBSD, title: EDAX-Ni Scan 1, dimensions: (186, 151|60, 60)>
```

These are already static background corrected. But here, we remove the dynamic background (lazily) as well

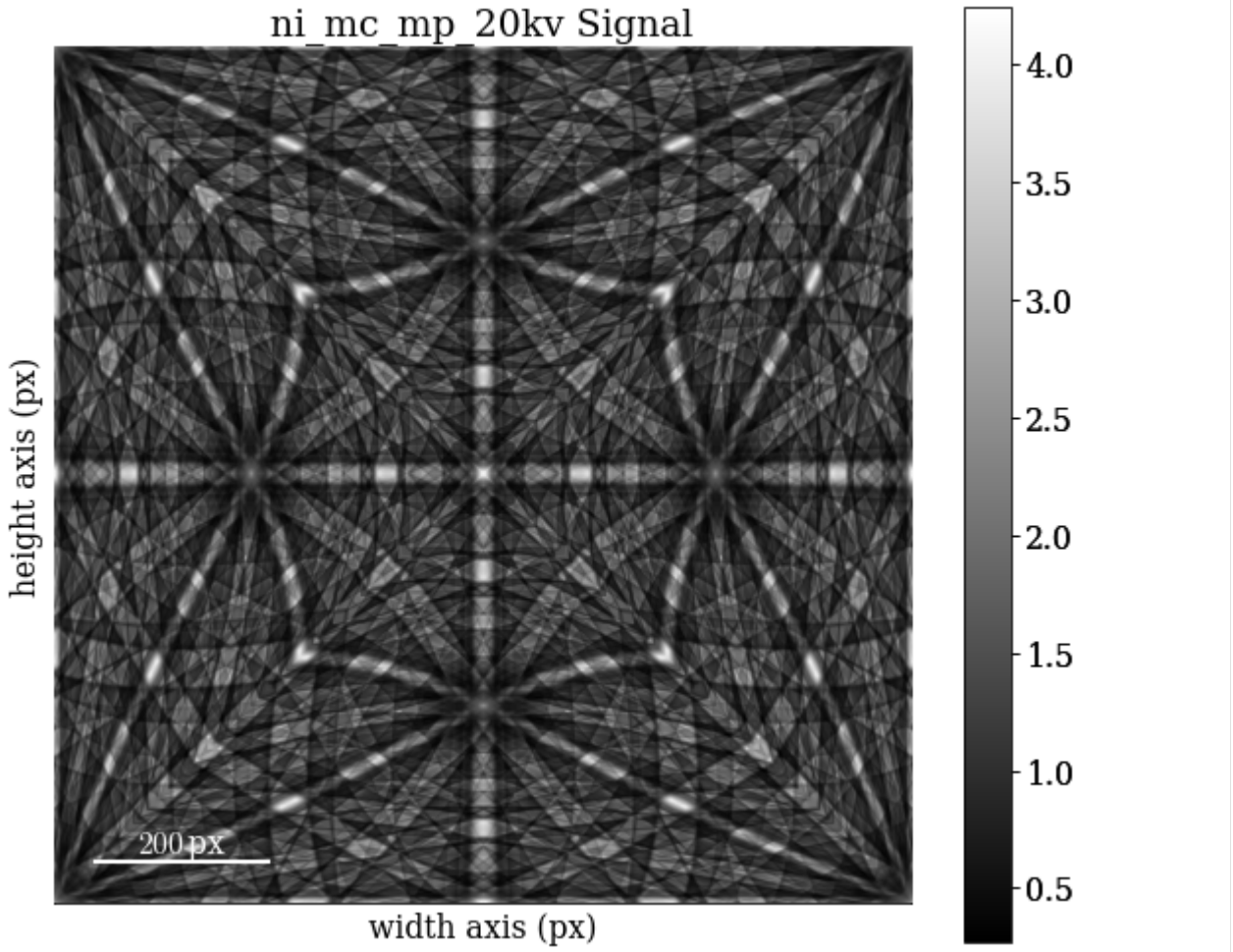
[3]:
```
s.remove_dynamic_background()
```

Load the Ni EBSD master pattern simulated with EMsoft (upper hemisphere only)

[4]:
```
mp = kp.data.ebsd_master_pattern(
    "ni", allow_download=True, projection="lambert", energy=20
)
mp
```

[4]:
```
<EBSDMasterPattern, title: ni_mc_mp_20kv, dimensions: (|1001, 1001)>
```

[5]:
```
mp.plot(navigator=None)
```




Extract the phase and change the lattice parameters from nm to Ångström

[6]:
```
phase = mp.phase
lat = phase.structure.lattice
lat.setLatPar(lat.a * 10, lat.b * 10, lat.c * 10)

print(phase)
print(phase.structure)
```

```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
lattice=Lattice(a=3.5236, b=3.5236, c=3.5236, alpha=90, beta=90, gamma=90)
```

(continues on next page)

(continued from previous page)

```
28   0.000000 0.000000 0.000000 1.0000
```

Extract a (4, 4) grid of patterns using *EBSD.extract_grid()*

[7]:
```
grid_shape = (4, 4)
s_grid, idx = s.extract_grid(grid_shape, return_indices=True)

s_grid.compute()

s_grid
```

```
[########################################] | 100% Completed | 204.95 ms
```

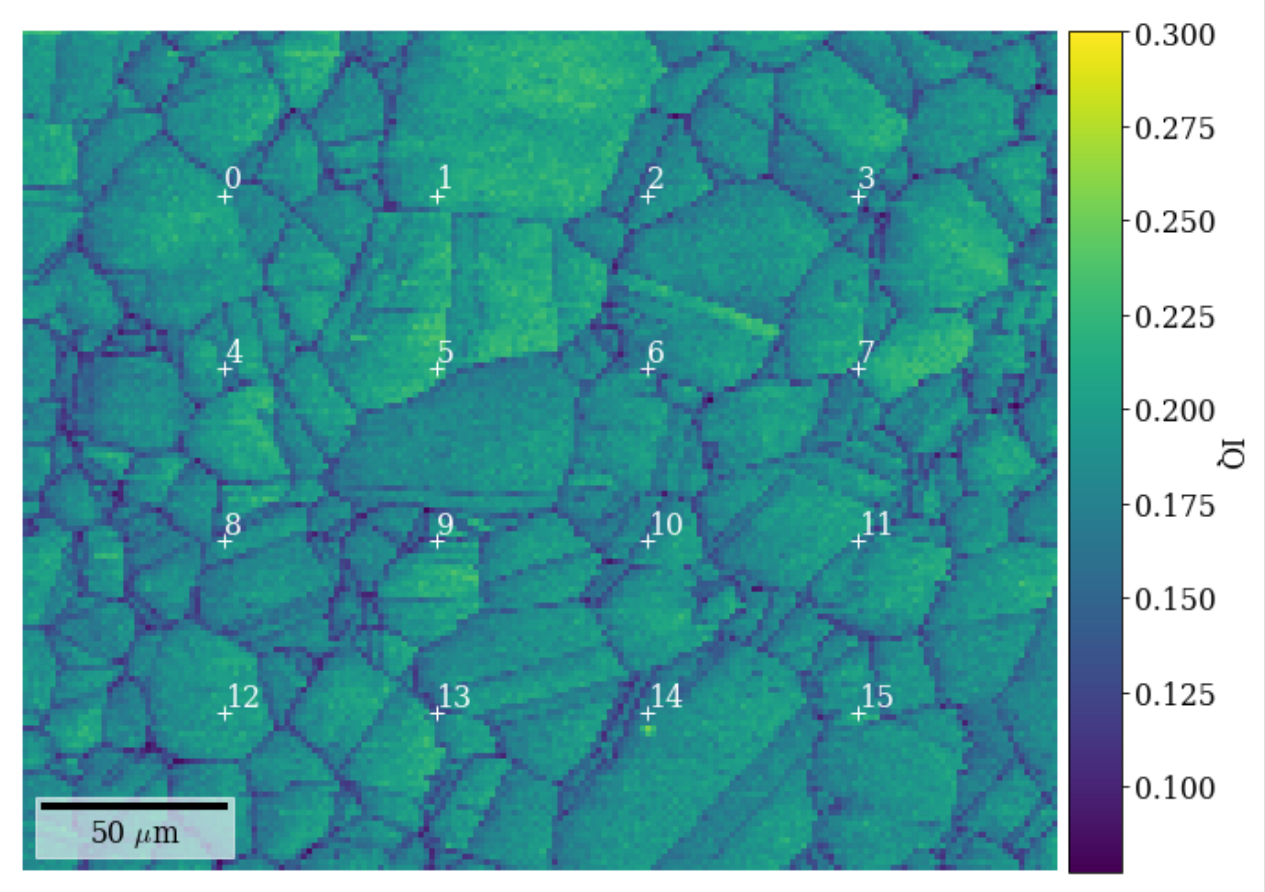
[7]:
```
<EBSD, title: EDAX-Ni Scan 1, dimensions: (4, 4|60, 60)>
```

Inspect an image quality map (we have to compute the returned Dask array since we loaded data lazily above)

[8]:
```
iq = s.get_image_quality()
iq = iq.compute()
```

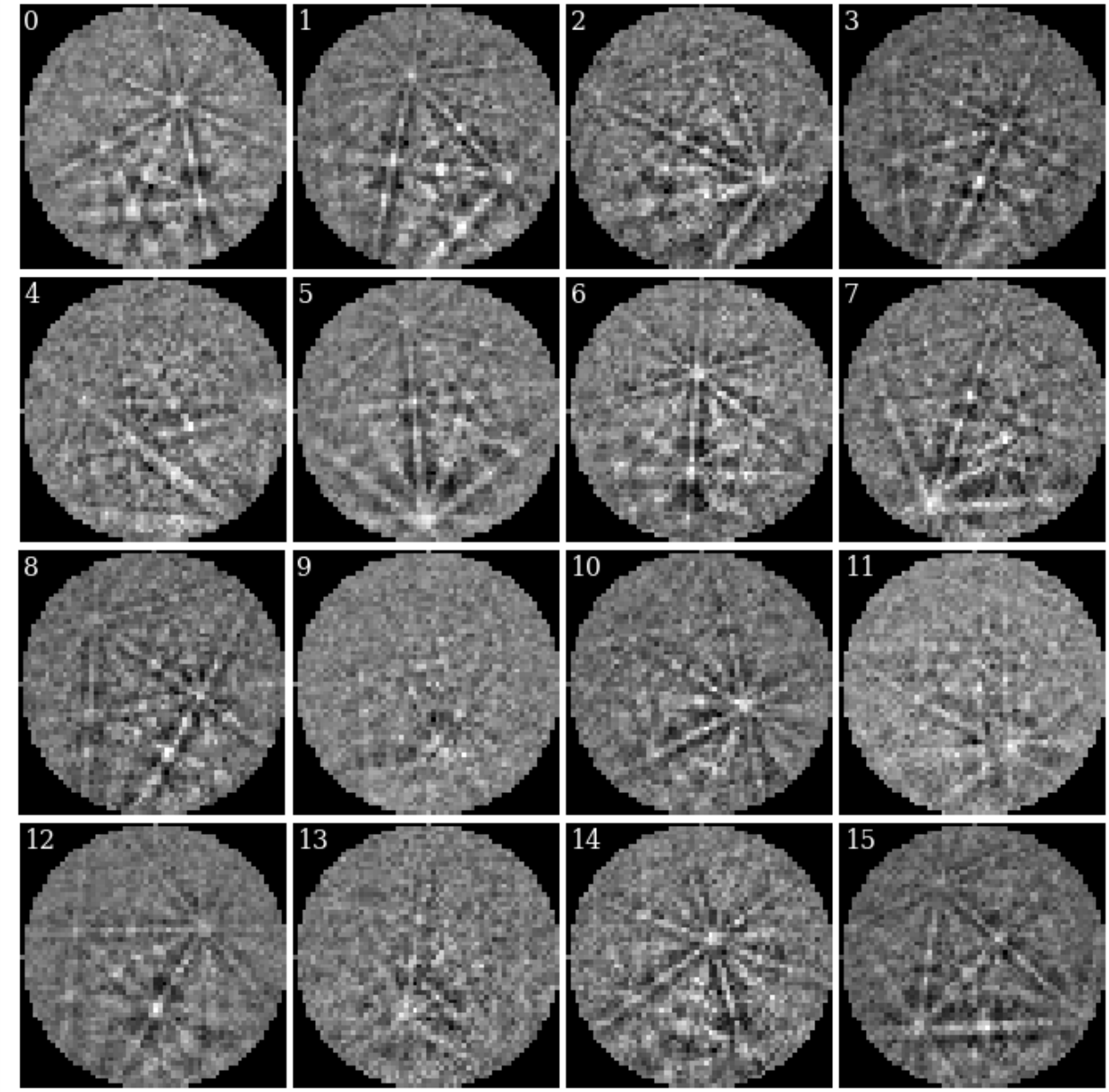
Plot the positions of the extracted patterns on the grid

[9]:
```
# For convenience, use the plot method of the crystal map attached to the EBSD
# signal to plot the IQ map. The array must be 1D.
s.xmap.scan_unit = "um"
fig = s.xmap.plot(
    iq.ravel(),
    vmax=0.3,
    remove_padding=True,
    colorbar=True,
    colorbar_label="IQ",
    return_figure=True,
)

kp.draw.plot_pattern_positions_in_map(
    rc=idx.reshape(2, -1).T,
    roi_shape=s.xmap.shape,
    axis=fig.axes[0],
    color="w",
)
```




Plot the grid patterns, using a radial mask to remove intensities without Kikuchi diffraction

[10]:
```
signal_mask = kp.filters.Window("circular", s_grid.detector.shape)
signal_mask = ~signal_mask.astype(bool)

fig = plt.figure(figsize=(9.5, 9.5))
_ = hs.plot.plot_images(
    s_grid * ~signal_mask,
    fig=fig,
    axes_decor=None,
    label=None,
    colorbar=False,
    per_row=grid_shape[0],
    padding=dict(wspace=0.03, hspace=0.03),
    tight_layout=True,
)
for i, ax in enumerate(fig.axes):
    ax.text(1, 1, i, va="top", ha="left", c="w")
    ax.axis("off")
```




### Estimate average PC from grid of patterns

We estimate an average PC and do Hough indexing of the grid patterns using PyEBSDIndex. See the *Hough indexing tutorial* for more details on the Hough indexing-related commands below.

**Note**

PyEBSDIndex is an optional dependency of kikuchipy, and can be installed with both `pip` and `conda` (from `conda-forge`). To install PyEBSDIndex, see their installation instructions.

We use relevant sample-detector geometry values read from the EDAX file, stored in the

*kikuchipy.signals.EBSD.detector* attribute. Note that the PC is in the Bruker convention (used internally); see the *reference frames tutorial* for details.

[11]:
```
det = s_grid.detector
det
```

[11]:
```
EBSDDetector (60, 60), px_size 1.0 um, binning 1, tilt 10.0, azimuthal 0.0, pc (0.507, 0.
↪262, 0.558)
```

We need an EBSDIndexer to use `PyEBSDIndex`. We can obtain an indexer by passing a PhaseList to *EBSDDetector.get_indexer()*

[12]:
```
phase_list = PhaseList(phase)
phase_list
```

[12]:
```
Id  Name  Space group  Point group  Proper point group     Color
 0    ni        Fm-3m         m-3m                 432  tab:blue
```

[13]:
```
indexer = det.get_indexer(phase_list, rSigma=2, tSigma=2)
```

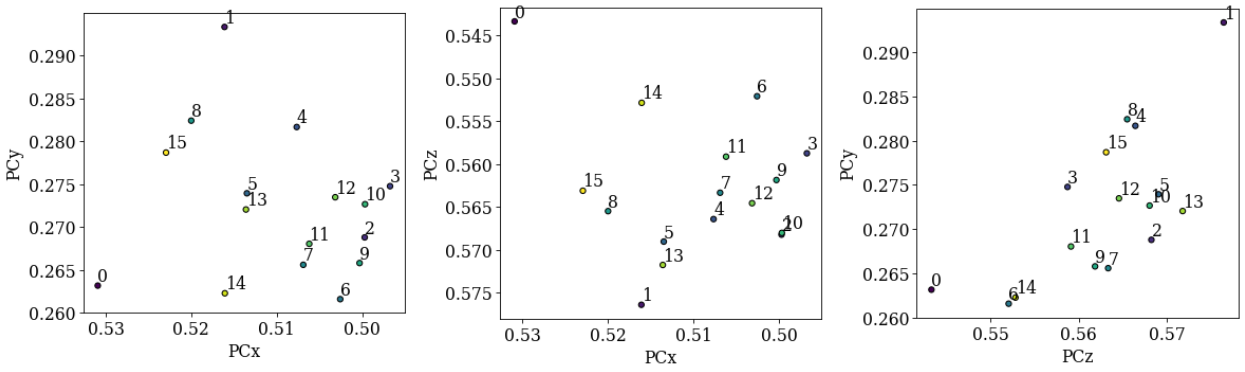
Estimate the PC for the grid patterns using particle swarm optimization (PSO), also printing the mean and standard deviation

[14]:
```
det_grid = s_grid.hough_indexing_optimize_pc(
    pc0=det.pc,
    indexer=indexer,
    batch=True,
    method="PSO",
    search_limit=0.05,
)

print(det_grid.pc_flattened.mean(axis=0))
print(det_grid.pc_flattened.std(0))
```

```
PC found: [********* ] 16/16  global best:0.182  PC opt:[0.5229 0.2787 0.5631]
[0.50974634 0.27237936 0.56279319]
[0.00942024 0.00830489 0.00795026]
```

Plot the PCs

[15]:
```
det_grid.plot_pc("scatter", annotate=True)
```



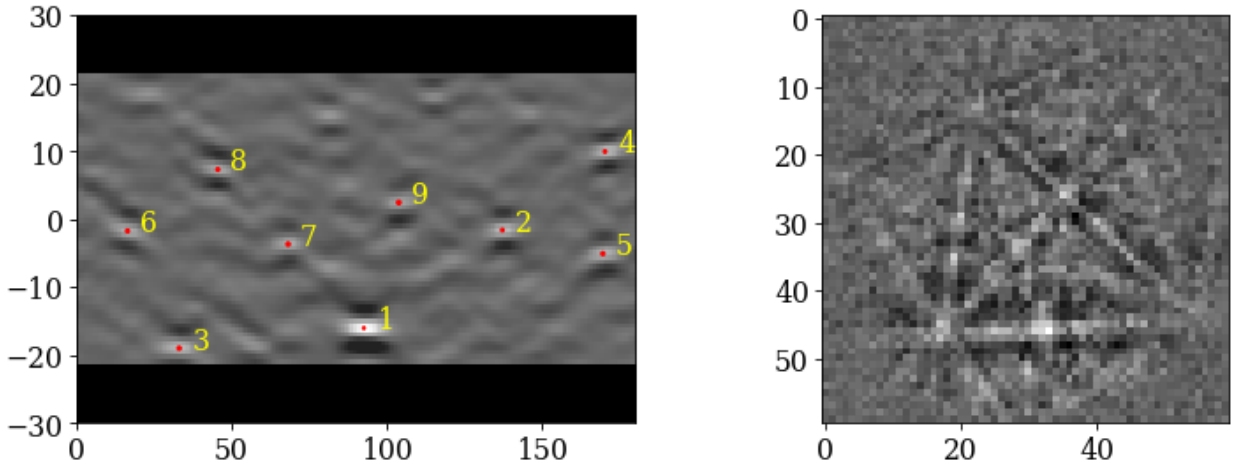
The PCs are quite dispersed, but most seem to cluster around an average PC. We should only use the average PC from these initial estimates and not try to fit a plane to them, as they do not vary in the grid they were extracted from.

Index the grid patterns using Hough indexing with the initially estimated PCs (to be averaged later!)

[16]:
```
indexer = det_grid.get_indexer(phase_list, rSigma=2, tSigma=2)
```

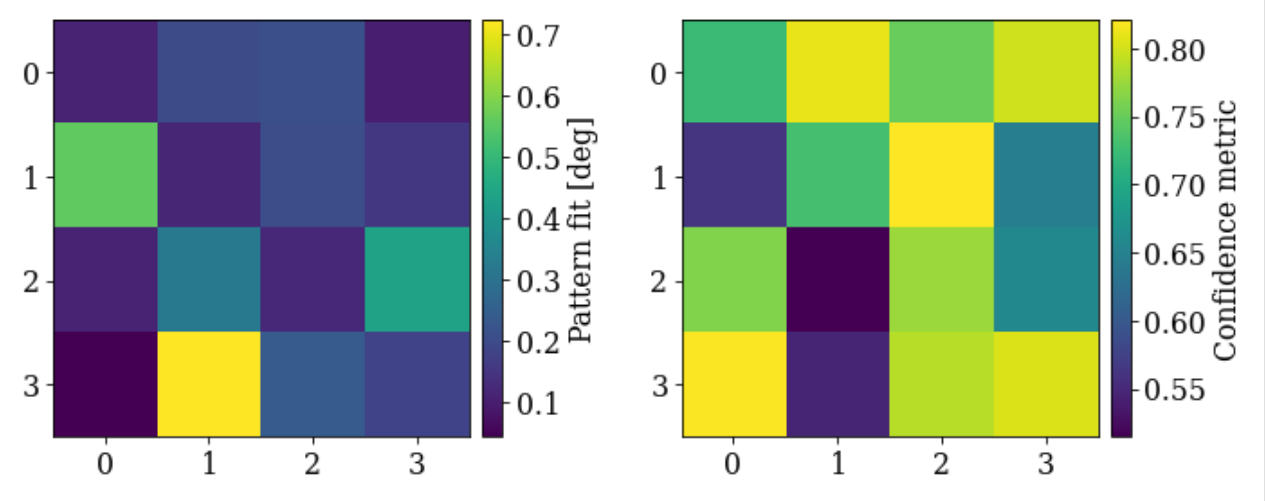
[17]:
```
xmap_grid = s_grid.hough_indexing(
    phase_list=phase_list, indexer=indexer, verbose=2
)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: True
  Projection center (Bruker, mean): (0.5097, 0.2724, 0.5628)
  Indexing 16 pattern(s) in 1 chunk(s)
Radon Time: 0.008487391998642124
Convolution Time: 0.006044162000762299
Peak ID Time: 0.0011351369903422892
Band Label Time: 0.016328211990185082
Total Band Find Time: 0.032020863000070676
Band Vote Time:  0.01946234800561797
  Indexing speed: 204.60269 patterns/s
```




Plot the pattern fit (values in range [0, 3]) and confidence metric (values in the range [0, 1])

[18]:
```
fig, axes = plt.subplots(ncols=2, figsize=(12, 4))
for ax, to_plot, label in zip(
    axes, ["fit", "cm"], ["Pattern fit [deg]", "Confidence metric"]
):
    im = ax.imshow(xmap_grid.get_map_data(to_plot))
    fig.colorbar(im, ax=ax, label=label, pad=0.02)
fig.subplots_adjust(wspace=0)
```




We see that PyEBSDIndex is fairly confident on the indexed solution for most patterns. Patterns with a low confidence seem to be located on boundaries in the IQ map above (4, 9, 11, and 13). These patterns have the highest pattern (mis)fit as well.

Let's refine both the PCs and orientations using pattern matching (see the *pattern matching tutorial* for details). We use the Nelder-Mead implementation from the NLopt package, which is an optional dependency of kikuchipy (see *the installation guide* for details).

[19]:
```
xmap_grid_ref, det_grid_ref = s_grid.refine_orientation_projection_center(
    xmap=xmap_grid,
    detector=det_grid,
    master_pattern=mp,
    energy=20,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5, 0.05, 0.05, 0.05],  # Wide trust region (!)
    rtol=1e-7,
    signal_mask=signal_mask,
    # Recommended when refining few patterns
    chunk_kwargs=dict(chunk_shape=1),
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5.   5.   5.   0.05 0.05 0.05]
  Relative tolerance: 1e-07
Refining 16 orientation(s) and projection center(s):
[########################################] | 100% Completed | 1.83 sms
Refinement speed: 8.69913 patterns/s
```

Print normalized cross-correlation (NCC) scores, number of evaluations (iterations) and the mean and standard deviation of the refined (PCx, PCy, PCz)

[20]:
```
print(xmap_grid_ref.scores.mean())
print(xmap_grid_ref.num_evals.mean())
print(det_grid_ref.pc_average)
print(det_grid_ref.pc_flattened.std(axis=0))
```

```
0.4922779928892851
```

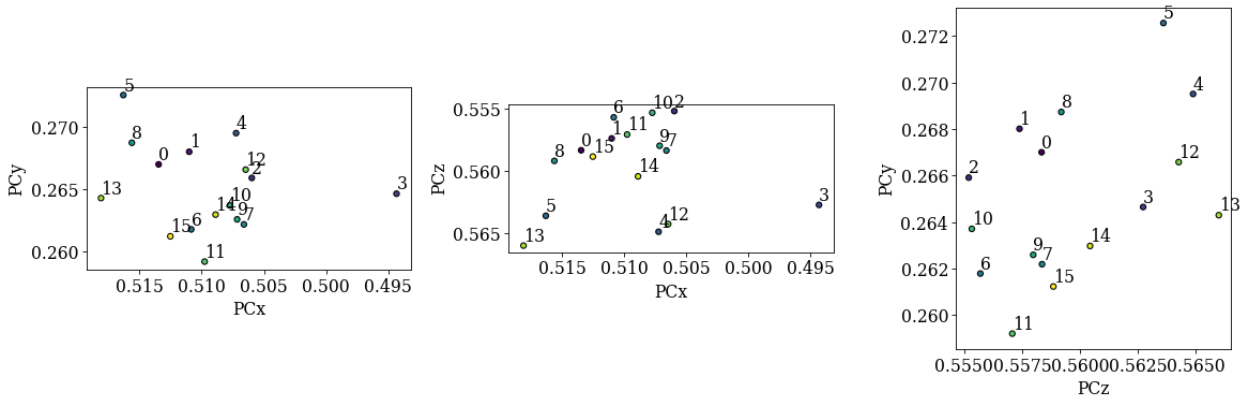
(continues on next page)

(continued from previous page)

```
607.125
[0.50950009 0.26506428 0.55970274]
[0.00536648 0.00340901 0.00341814]
```

Plot the refined PCs as we did above

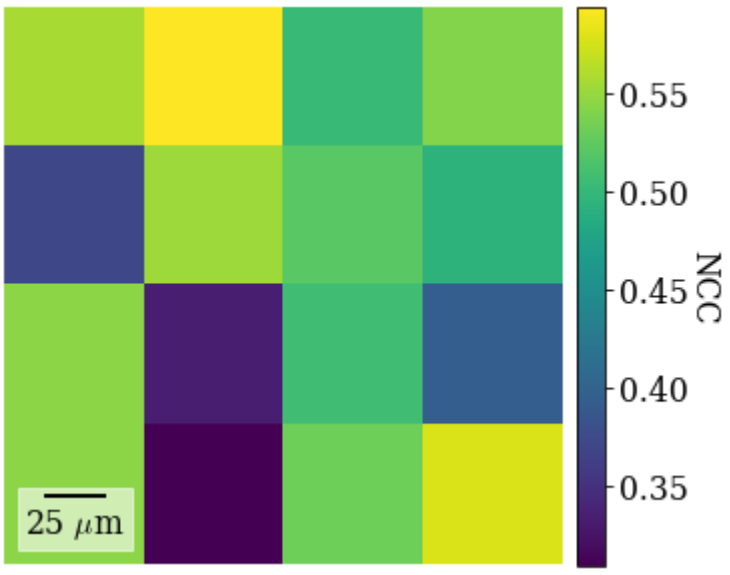
[21]:
```
det_grid_ref.plot_pc("scatter", annotate=True)
```




The PCs are still quite spread out, but the deviations from the mean are much smaller compared to the PCs from Hough indexing alone.

Let's plot the NCC score

[22]:
```
xmap_grid_ref.plot(
    "scores",
    remove_padding=True,
    colorbar=True,
    colorbar_label="NCC",
    figure_kwargs=dict(figsize=(4, 4)),
)
```



Let's inspect the refined orientations by plotting center lines of the Kikuchi bands on top of patterns (see the *geometrical EBSD simulations tutorial* for details)

[23]:
```
ref = ReciprocalLatticeVector.from_min_dspacing(phase, 1)
ref.sanitise_phase()  # Expand unit cell
ref.calculate_structure_factor()
F = abs(ref.structure_factor)
ref = ref[F > 0.6 * F.max()]
ref.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    2.034    11.8     140.0     100.0      8
 2 0 0    1.762    10.4     108.2      77.3      6
 2 2 0    1.246     7.4      55.0      39.3      12
```

[24]:
```
simulator = kp.simulations.KikuchiPatternSimulator(ref)
```

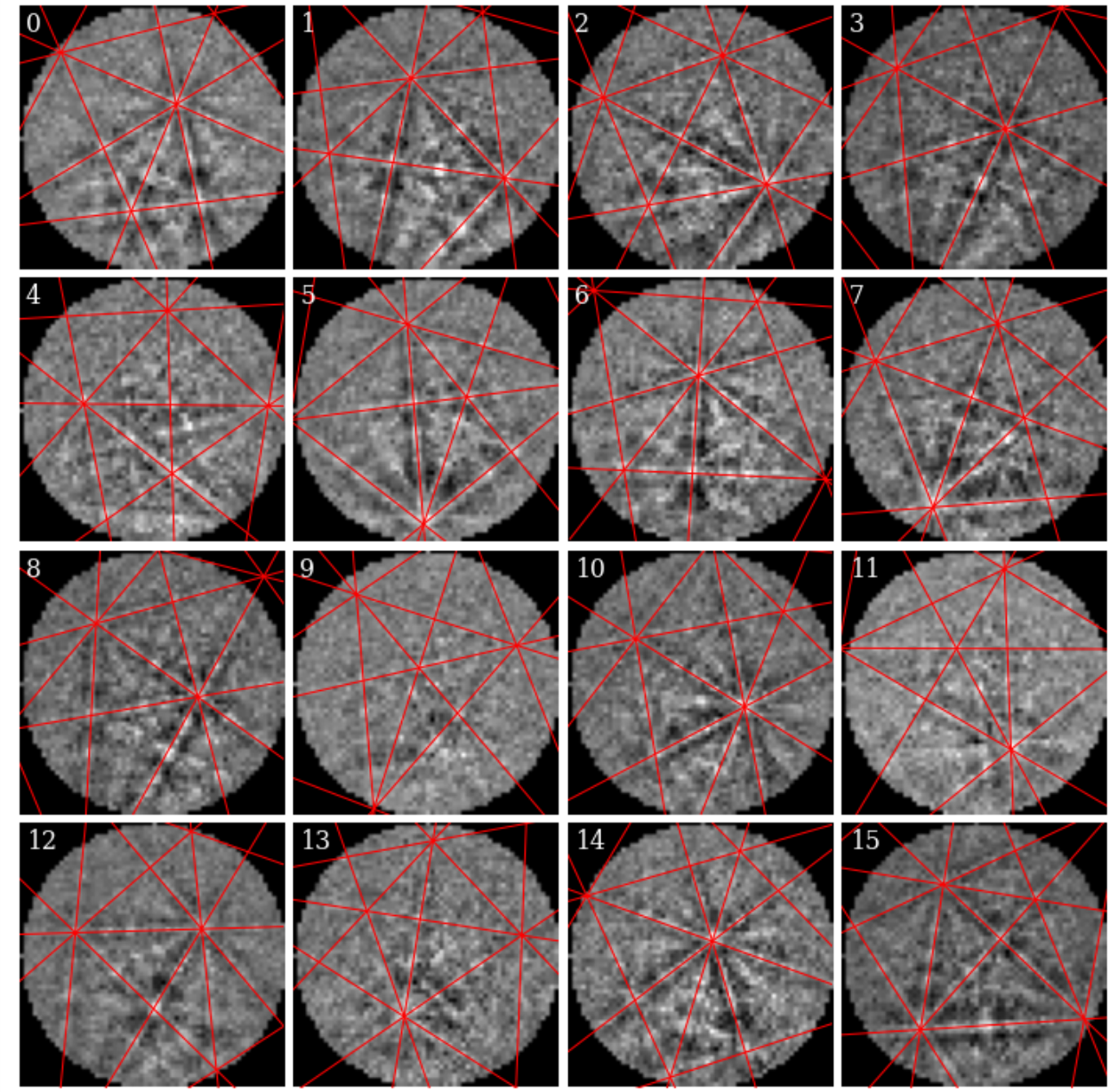
Get simulations

[25]:
```
sim = simulator.on_detector(
    det_grid_ref, xmap_grid_ref.rotations.reshape(*xmap_grid_ref.shape)
)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 105.04 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.92 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.79 ms
```

Plot geometrical simulations on top of patterns

[26]:
```
fig, axes = plt.subplots(
    nrows=grid_shape[0], ncols=grid_shape[1], figsize=(12, 12)
)
for i, rc in enumerate(np.ndindex(grid_shape)):
    axes[rc].imshow(s_grid.data[rc] * ~signal_mask, cmap="gray")
    axes[rc].axis("off")
    lines = sim.as_collections(rc)[0]
    axes[rc].add_collection(lines)
    axes[rc].text(1, 1, i, va="top", ha="left", c="w")
fig.subplots_adjust(wspace=0.03, hspace=0.03)
```




Most simulated lines seem lie on top of experimental bands, as expected.

Let's *now* compute the average PC weighted by correlation scores

[27]:
```
det_ref = det_grid_ref.deepcopy()
det_ref.pc = np.average(
    det_grid_ref.pc.reshape((-1, 3)), weights=xmap_grid_ref.scores, axis=0
)

print("Estimated PC:", det_ref.pc)
print("EDAX OIM PC: ", det.pc)
print("Difference:  ", det_ref.pc - det.pc)
```

```
Estimated PC: [[0.50943553 0.265213   0.55954418]]
```

(continues on next page)

(continued from previous page)

```
EDAX OIM PC:  [[0.50726193 0.26207614 0.55848867]]
Difference:   [[0.00217359 0.00313686 0.00105551]]
```

We see that the PC we obtained with our routine above deviates little from the EDAX OIM PC. This is not surprising since we used the EDAX OIM PC as an initial guess. Based on the geometrical simulations above, these PC values seem OK.

### Index all patterns with Hough indexing

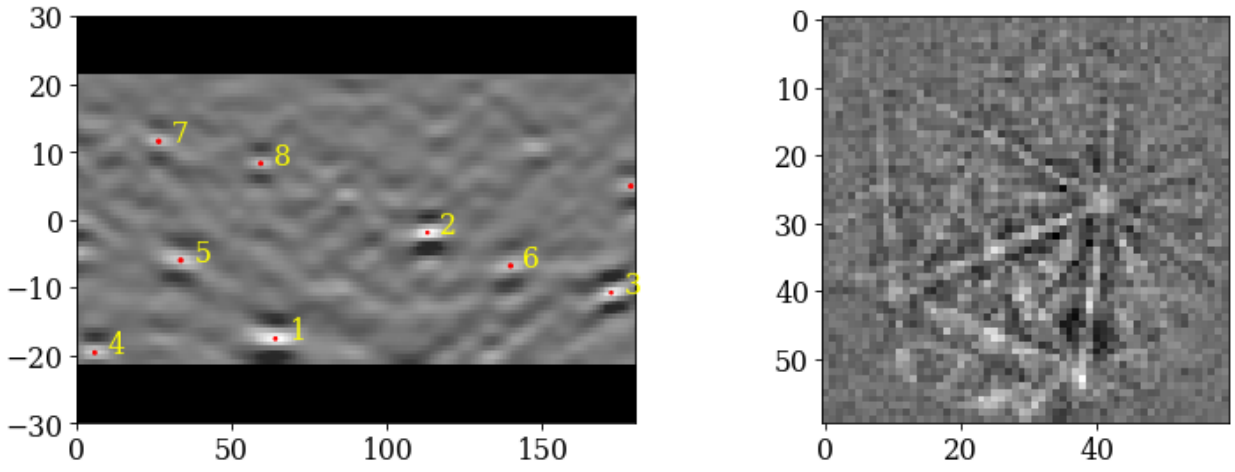
Let's index the entire map of patterns with PyEBSDIndex, using our indexer from before but with the new estimated PC

[28]:
```
indexer = det_ref.get_indexer(phase_list, rSigma=2, tSigma=2)
indexer.PC
```

[28]:
```
array([0.50943553, 0.265213  , 0.55954418])
```

[29]:
```
xmap = s.hough_indexing(phase_list=phase_list, indexer=indexer, verbose=2)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: True
  Projection center (Bruker): (0.5094, 0.2652, 0.5595)
  Indexing 28086 pattern(s) in 54 chunk(s)
Radon Time: 8.110585576971062
Convolution Time: 6.115091640967876
Peak ID Time: 1.9820071019930765
Band Label Time: 4.58851349100587
Total Band Find Time: 20.79672850400675
Band Vote Time:  24.79082443600055
  Indexing speed: 614.25158 patterns/s
```




[30]:
```
xmap
```

[30]:
```
Phase    Orientations          Name  Space group  Point group  Proper point group    ␣
↪Color
   -1        2 (0.0%)  not_indexed         None         None                None       ␣
```

(continues on next page)

(continued from previous page)

```
↪w
    0  28084 (100.0%)           ni        Fm-3m         m-3m                 432  tab:
↪blue
Properties: fit, cm, pq, nmatch
Scan unit: um
```

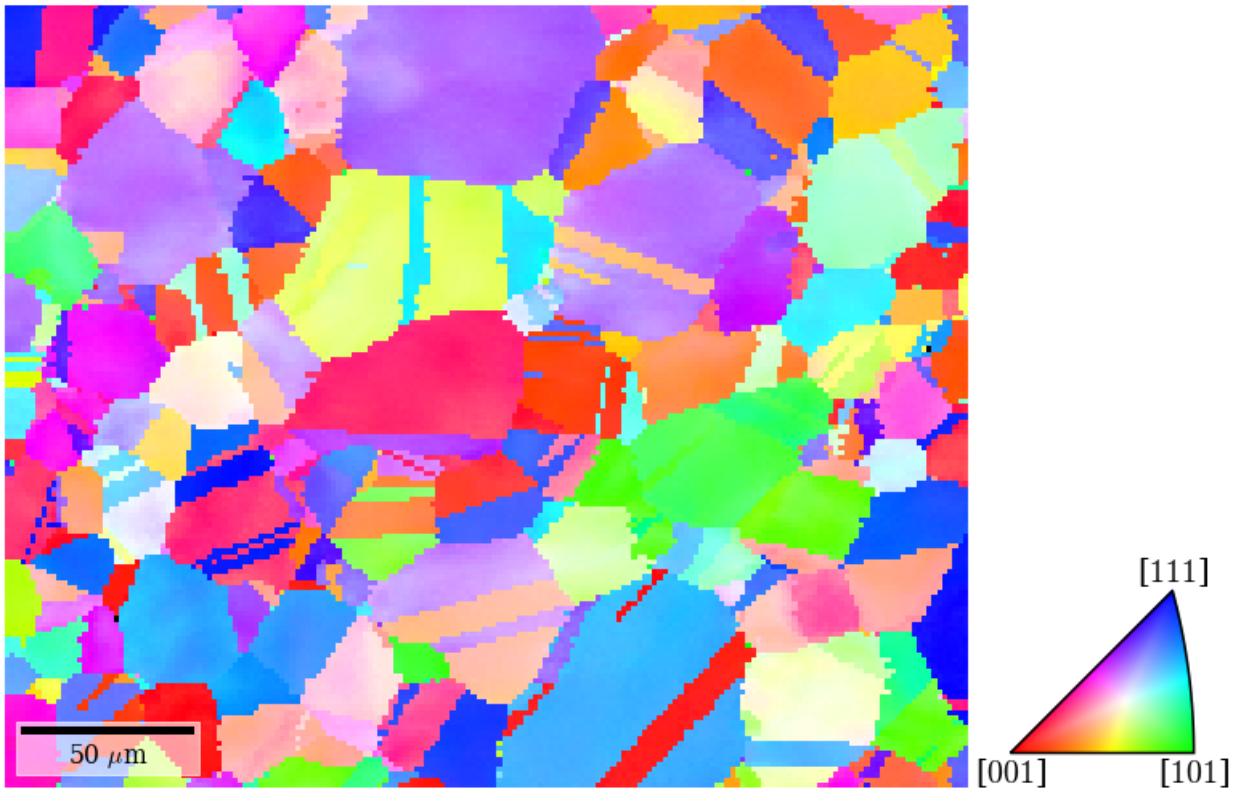
Let's plot the inverse pole figure (IPF)-Z color map. First, we must get a color key, potentially setting the sample direction if we want another IPF than IPF-Z

[31]:
```
pg = xmap.phases[0].point_group

ckey_m3m = plot.IPFColorKeyTSL(pg)
print(ckey_m3m)
```

```
IPFColorKeyTSL, symmetry: m-3m, direction: [0 0 1]
```

Plot the map with the IPF color key next to the map

[32]:
```
# Make an array of full map size of zeros (black), and fill colors from
# successfully indexed nickel in the correct points
rgb = np.zeros((xmap.size, 3))
rgb[xmap.is_indexed] = ckey_m3m.orientation2color(xmap["indexed"].orientations)
fig = xmap.plot(rgb, remove_padding=True, return_figure=True)

# Place color key next to map
rect = [1.04, 0.115, 0.2, 0.2]  # [Left, bottom, width, height]
ax_ckey = fig.add_axes(rect, projection="ipf", symmetry=pg)
ax_ckey.plot_ipf_color_key(show_title=False)
ax_ckey.patch.set_facecolor("None")
```



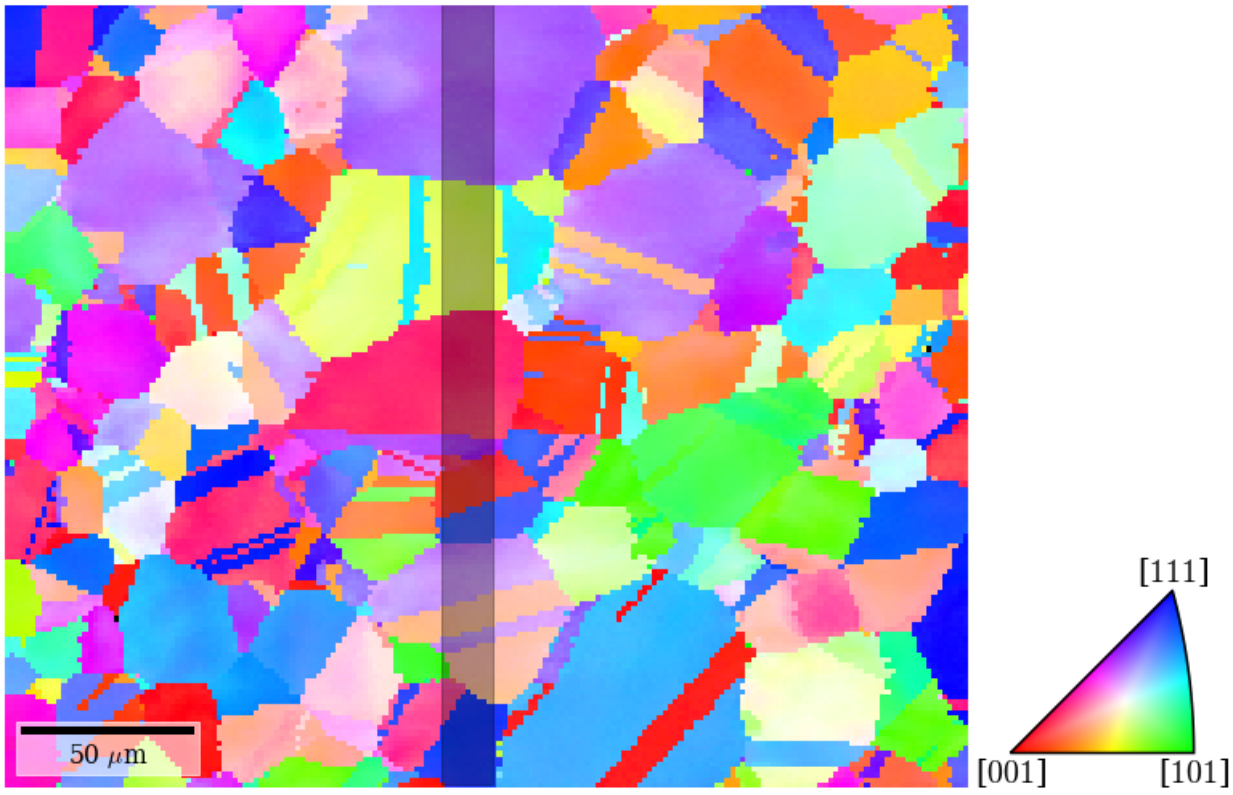

We see that Hough indexing with PyEBSDIndex is able to index the patterns quite convincingly.

### Fit PC in vertical direction

Finally, we'll refine orientations and PCs in a vertical slice, average the PCs in the horizontal direction, and inspect the PCs and PC errors as a function of the vertical position. We'll use the same vertical slice as [Pang *et al.*, 2020] use.

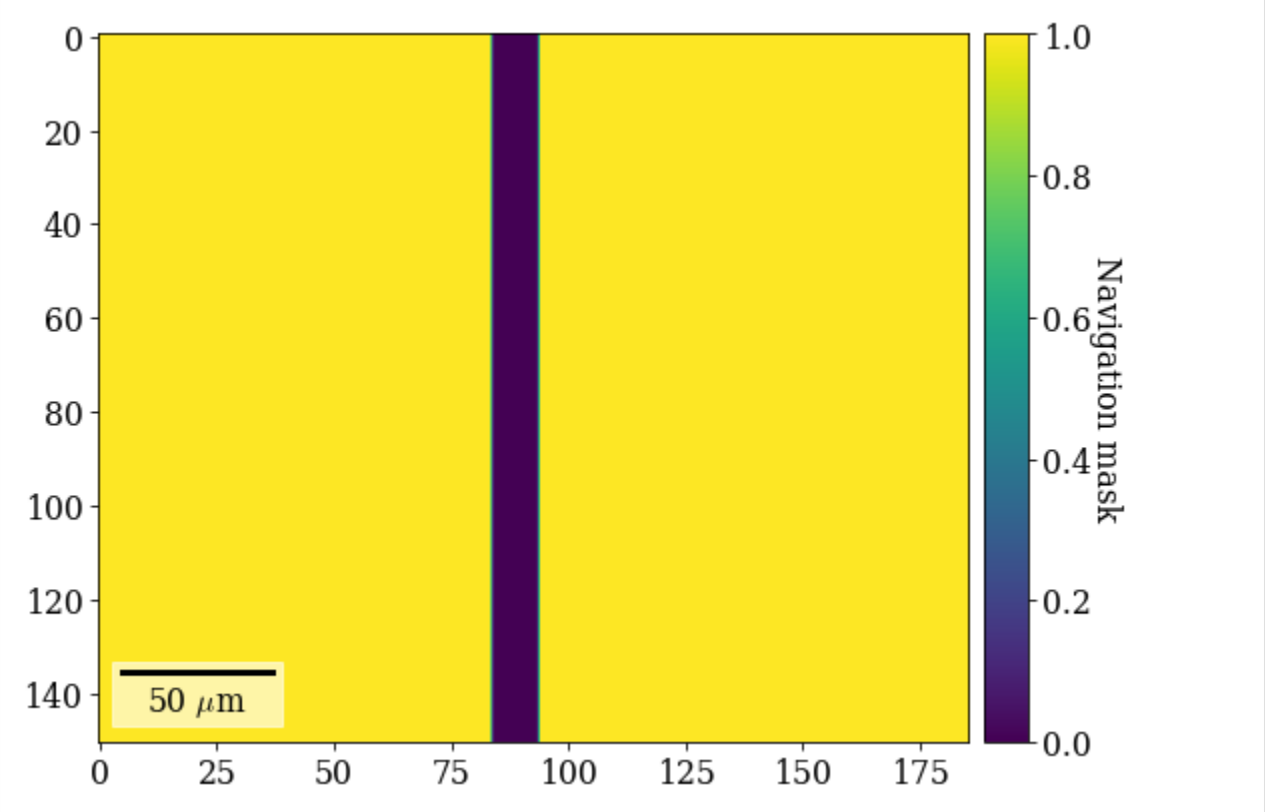
[33]:
```
# Get tuple of slices to extract data of interest
x0, x1 = 84, 84 + 10
y0, y1 = 0, xmap.shape[0]
slices = (slice(y0, y1), slice(x0, x1))

# Rectangle to highlight the slice on top of IPF-Z map
rect_slice = mpatches.Rectangle(
    xy=(x0, y0 - 1),
    width=x1 - x0,
    height=y1 - y0,
    color="k",
    alpha=0.3,
)
fig.axes[0].add_artist(rect_slice)
fig  # Show the figure again
```

[33]:




Create a navigation mask to only index the patterns within this slice. Navigation and signal masks in kikuchipy follow the convention in other packages like NumPy, where points to *mask out* are set to `True`.

[34]:
```
nav_mask = np.ones(xmap.shape, dtype=bool)
nav_mask[:, x0:x1] = False

xmap.plot(nav_mask.ravel(), colorbar=True, colorbar_label="Navigation mask")
```




Before going further, it is important to set the binning factor and (unbinned) detector pixel size

[35]:
```
det_ref.binning = 8
det_ref.px_size = 59.2
```

We'll use the exact same trust region (+/- bounds) on the orientations (1°) and PCs in EMsoft's convention (5 px for $x_{pc}$ and $y_{pc}$ and 500 $\mu$m for $L$) in the refinement as used by [Pang *et al.*, 2020].

[36]:
```
trust_region = [
    1,
    1,
    1,
    5 / (det_ref.ncols * det_ref.binning),
    5 / (det_ref.nrows * det_ref.binning),
    500 / (det_ref.nrows * det_ref.px_size * det_ref.binning),
]
```

Refine orientations and PCs of patterns in the slice, using the Hough indexed orientations and the estimated average PC as initial guesses

[37]:
```
xmap_slice_ref, det_slice_ref = s.refine_orientation_projection_center(
    xmap=xmap,
    detector=det_ref,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    navigation_mask=nav_mask,
```

(continues on next page)

(continued from previous page)

```
    method="LN_NELDERMEAD",
    trust_region=trust_region,
    rtol=1e-7,
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [1.      1.      1.      0.01042 0.01042 0.0176 ]
  Relative tolerance: 1e-07
Refining 1510 orientation(s) and projection center(s):
[########################################] | 100% Completed | 154.00 s
Refinement speed: 9.79877 patterns/s
```

Inspect the average NCC score, number of evaluations, and PC and the standard deviation of the average PC

[38]:
```
print(xmap_slice_ref.scores.mean())
print(xmap_slice_ref.num_evals.mean())
print(det_slice_ref.pc_average)
print(det_slice_ref.pc_flattened.std(axis=0))
```

```
0.548018371521045
533.548344370861
[0.50976996 0.26621384 0.55939217]
[0.00342416 0.0032401  0.00272805]
```

To compare our results to those of [Pang *et al.*, 2020], we will plot the PCs in EMsoft's version 4 definition (see *EBSDDetector.pc_emsoft()* for details on the conversion from Bruker's definition)

[39]:
```
pc_slice = det_slice_ref.pc_emsoft(version=4)

# Average horizontally
pc_slice_mean = pc_slice.mean(axis=1)

# Reshape for easy plotting
pcx, pcy, pcz = pc_slice_mean.reshape((-1, 3)).T
```

Get curves for the expected changes in PCs based on the equations given at the start

[40]:
```
y_pos_um = np.arange(y1) * xmap.dy
y_pos_px = y_pos_um / det_slice_ref.px_size

alpha = np.deg2rad(90 - det_slice_ref.sample_tilt + det_slice_ref.tilt)

# Find best intercept value for PCy and PCz by curve fitting
pcy_inter, _ = curve_fit(lambda y, a: a + np.cos(alpha) * y, y_pos_px, pcy)
pcz_inter, _ = curve_fit(lambda y, a: a + np.sin(alpha) * y, y_pos_um, pcz)

pcx_fit = np.ones(y_pos_um.size) * pcx.mean()
pcy_fit = pcy_inter[0] + np.cos(alpha) * y_pos_px
pcz_fit = pcz_inter[0] + np.sin(alpha) * y_pos_um
```

Get the distance (error) between measured and expected changes, and the cumulative fraction of these distances

[41]:
```
# Deviations
pcx_err = abs(pcx_fit - pcx)
pcy_err = abs(pcy_fit - pcy)
pcz_err = abs(pcz_fit - pcz)

# Cumulative fraction
pcx_err_sorted = np.sort(pcx_err)
pcy_err_sorted = np.sort(pcy_err)
pcz_err_sorted = np.sort(pcz_err)
y_cum = np.arange(pcx.size) / pcx.size
```

Let's find out the 90th percentile error in each PC coefficient

[42]:
```
idx_pct = np.where(y_cum > 0.9)[0][0]
pcx_err_pct = pcx_err_sorted[idx_pct]
pcy_err_pct = pcy_err_sorted[idx_pct]
pcz_err_pct = pcz_err_sorted[idx_pct]

print(
    pcx_err_pct / det_slice_ref.binning,
    pcy_err_pct / det_slice_ref.binning,
    pcz_err_pct / (det_slice_ref.px_size * det_slice_ref.binning),
)
```

```
0.1514950425682673 0.12398719115084944 0.12285224277355544
```

As Pang and co-workers found, for $x_{pc}$ and $y_{pc}$, 90% of points have an error less than 0.2% of the detector width, while for $L$, 90% of points have an error less than 0.15%.

Let's plot the experimental and expected changes of the PCs along the vertical direction, along with the cumulative fractions. The experimental changes are colored according to the grain orientations within the vertical slice in the IPF-Z above (last column).

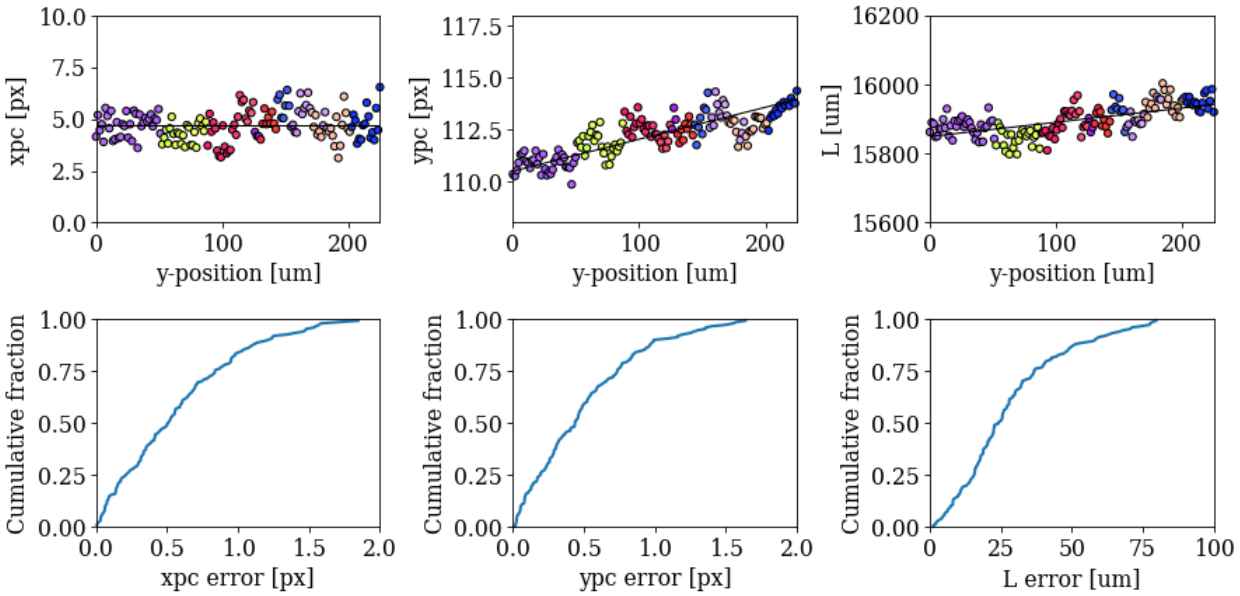
[43]:
```
rgb_slice = rgb.reshape(xmap.shape + (3,))[slices]
rgb_slice = rgb_slice[:, -1, :]
```

[44]:
```
# Common keyword arguments
scatter_kw = {"c": rgb_slice, "ec": "k", "clip_on": False}
scatter_set_kw = {"xlabel": "y-position [um]", "xlim": (0, y_pos_um[-1])}
plot_kw = {"lw": 2, "clip_on": False}
plot_set_kw = {"ylabel": "Cumulative fraction", "ylim": (0, 1)}

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 6))
ax0, ax1, ax2, ax3, ax4, ax5 = axes.ravel()

# PC vs. y-position w/ best linear fit
ax0.scatter(y_pos_um, pcx, **scatter_kw)
ax1.scatter(y_pos_um, pcy, **scatter_kw)
ax2.scatter(y_pos_um, pcz, **scatter_kw)
ax0.plot(y_pos_um, pcx_fit, c="k")
ax1.plot(y_pos_um, pcy_fit.ravel(), c="k")
ax2.plot(y_pos_um, pcz_fit.ravel(), c="k")
ax0.set(ylabel="xpc [px]", ylim=(0, 10), **scatter_set_kw)
ax1.set(ylabel="ypc [px]", ylim=(108, 118), **scatter_set_kw)
```

(continues on next page)

(continued from previous page)

```
ax2.set(ylabel="L [um]", ylim=(15600, 16200), **scatter_set_kw)

# Cumulative fraction of deviations
ax3.plot(pcx_err_sorted, y_cum, **plot_kw)
ax4.plot(pcy_err_sorted, y_cum, **plot_kw)
ax5.plot(pcz_err_sorted, y_cum, **plot_kw)
ax3.set(xlabel="xpc error [px]", xlim=(0, 2), **plot_set_kw)
ax4.set(xlabel="ypc error [px]", xlim=(0, 2), **plot_set_kw)
ax5.set(xlabel="L error [um]", xlim=(0, 100), **plot_set_kw)

fig.tight_layout()
```




There are clear systematic deviations in the PC away from the expected PC caused by grain orientations. See the discussion by [Pang *et al.*, 2020] for possible causes of this and more details on the above analysis.

In conclusion, coming back to the point made in the beginning: whenever possible, we should estimate a PC from not only many patterns, but also from many patterns from different grains.

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### Fit a plane to selected projection centers

In this tutorial, we will fit a plane to projection centers (PCs). The PCs are determined from a (5, 5) grid of EBSD patterns from a single crystal silicon wafer. The resulting fitted plane contains one PC per pattern in the dataset from which the grid is obtained. This plane of PCs can be used in subsequent indexing of the dataset. To relate the sample positions of the grid to the PCs, we will test both an affine transformation and a projective transformation. These transformations are based on the work by [Winkelmann *et al.*, 2020].

We'll start by importing necessary libraries

[1]:
```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import skimage.filters as skf

from diffsims.crystallography import ReciprocalLatticeVector
import kikuchipy as kp
from orix.crystal_map import PhaseList


plt.rcParams.update(
    {
        "figure.facecolor": "w",
        "figure.dpi": 75,
        "figure.figsize": (8, 8),
        "font.size": 15,
    }
)
```

### Load and inspect data

Load data lazily (~500 MB) and inspect its shape and step size

[2]:
```
s = kp.data.si_wafer(allow_download=True, lazy=True)
s
```

[2]:
```
<LazyEBSD, title: Pattern, dimensions: (50, 50|480, 480)>
```

We see that we have (50, 50) patterns of shape (480, 480). This is the full pattern resolution of the NORDIF UF-420 detector the patterns are acquired on. In the axes manager…

[3]:
```
print(s.axes_manager)
```

```
<Axes manager, axes: (50, 50|480, 480)>
            Name |   size |  index |  offset |   scale |  units
================ | ====== | ====== | ======= | ======= | ======
               x |     50 |      0 |       0 |      40 |     um
               y |     50 |      0 |       0 |      40 |     um
---------------- | ------ | ------ | ------- | ------- | ------
              dx |    480 |      0 |       0 |       1 |     um
              dy |    480 |      0 |       0 |       1 |     um
```

… we see that the nominal step sizes are $(\Delta x, \Delta y) = (40, 40)$ $\mu$m. The scan region of interest (ROI) therefore covers an area of $(2 \times 2)$ $\mu$m$^2$. The NORDIF UF-420 (unbinned) detector pixel size is about 90 $\mu$m. This means that we

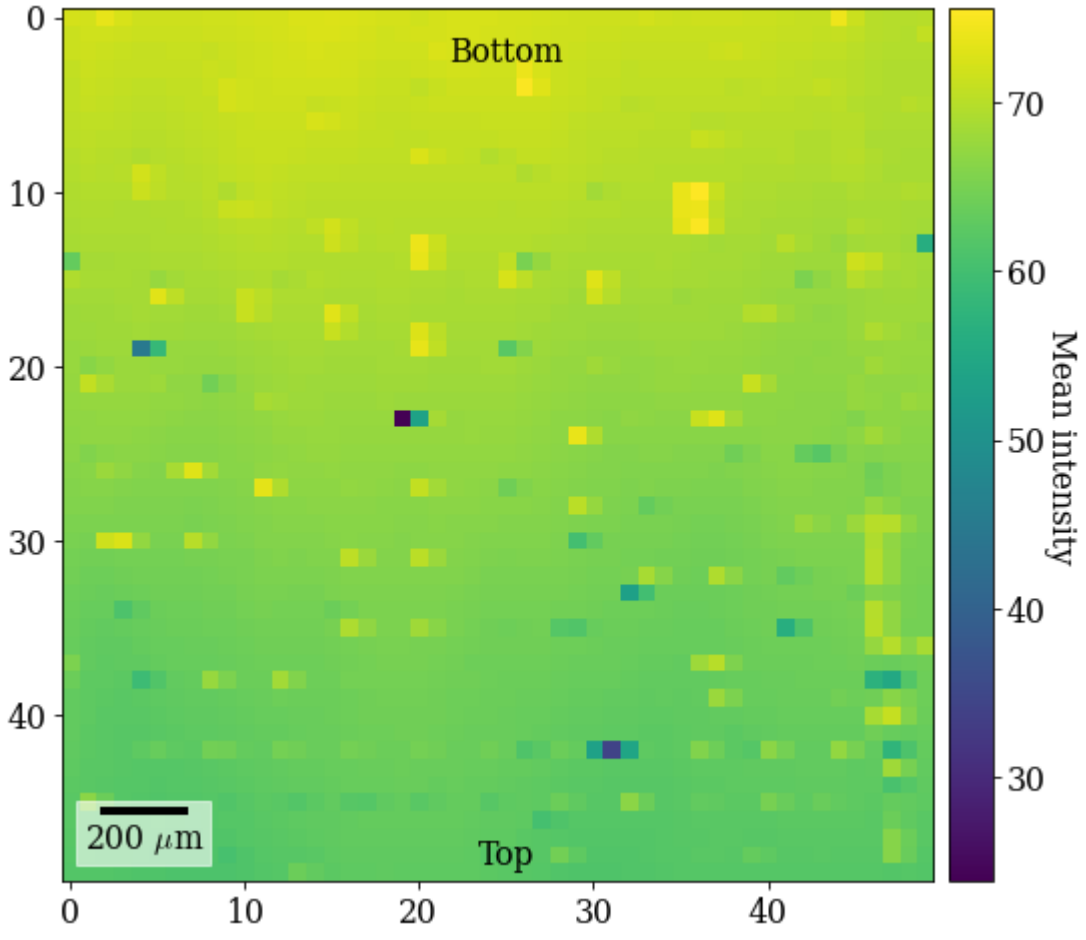
expect the PC to shift about 2000 / 90 ≈ 22 detector pixels when moving horizontally in the scan. To get a better understanding of the expected changes of the PC in the scan, we'll look at the mean intensity map of the sample.

[4]:
```
s_mean_nav = s.mean(axis=(2, 3))
s_mean_nav.compute()  # Need this call because data was loaded lazily
```

```
[########################################] | 100% Completed | 315.08 ms
```

Plot it, also annotating the orientation of the scan relative to the sample position in the chamber. We use the plot() method of the *EBSD.xmap* attribute for this.

[5]:
```
s.xmap.scan_unit = "um"
fig = s.xmap.plot(
    s_mean_nav.data.ravel(),
    colorbar=True,
    colorbar_label="Mean intensity",
    return_figure=True,
)

# Annotate
ax = fig.axes[0]
ax.text(25, 1, "Bottom", va="top", ha="center")
_ = ax.text(25, 49, "Top", va="bottom", ha="center");
```



The brighter and darker spots come from patterns acquired close to fiducial markers on the Si wafer. The vertical intensity gradient is a result of the intensity distribution on the detector shifting across the sample. Note the orientation of the ROI on the sample: to avoid depositing carbon ahead of the scanning beam, the sample is scanned from down the sample upwards, annotated on the map ("Bottom", "Top").

The Bruker PC convention is used internally in kikuchipy. The detector is viewed from the detector towards the sample. The convention measures (PCx, PCy, PCz) in the following manner:

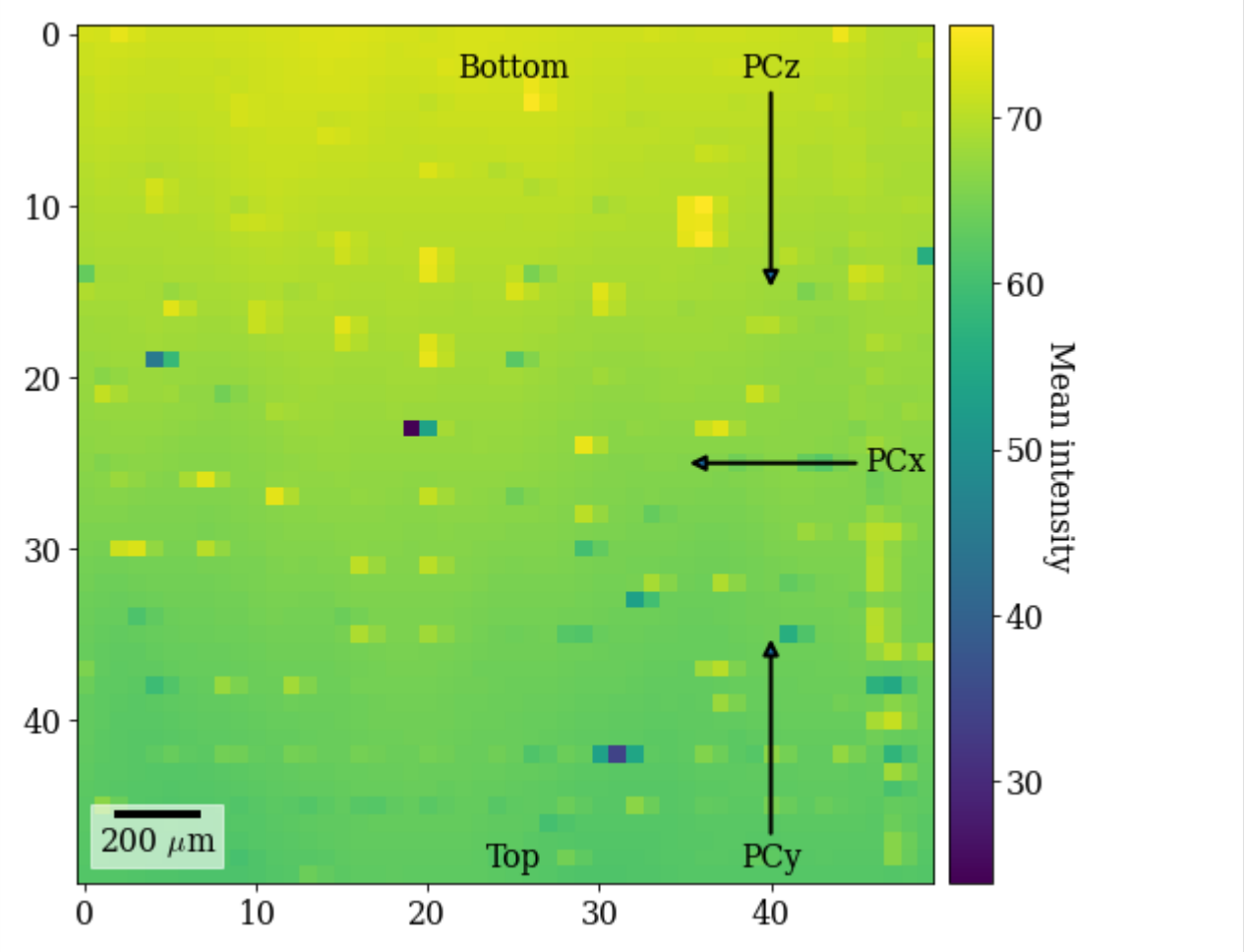
- PCx from left towards the right
- PCy downwards from the top
- PCz as the shortest distance from the beam-sample interaction position to the detector in fractions of the detector width.

See the *reference frames tutorial* for more details and conversions between conventions.

Given this definition, let's add annotations of the expected changes to the PC (increasing in the direction of the arrow) to the map

[6]:
```
annotate_kw = {
    "arrowprops": {"arrowstyle": "-|>", "lw": 2, "mutation_scale": 15}
}
xys = [(35, 25), (40, 35), (40, 15)]
xy_texts = [(49, 25), (40, 49), (40, 1)]
has = ["right", "center", "center"]
vas = ["center", "bottom", "top"]
labels = ["PCx", "PCy", "PCz"]
for xy, xytext, ha, va, label in zip(xys, xy_texts, has, vas, labels):
    ax.annotate(label, xy=xy, xytext=xytext, ha=ha, va=va, **annotate_kw)

fig  # Show figure again with new annotations
```

[6]:




Our goal is to fit a plane of (PCx, PCy, PCz) to this (50, 50) map with these variations. To this end, we perform the following actions:

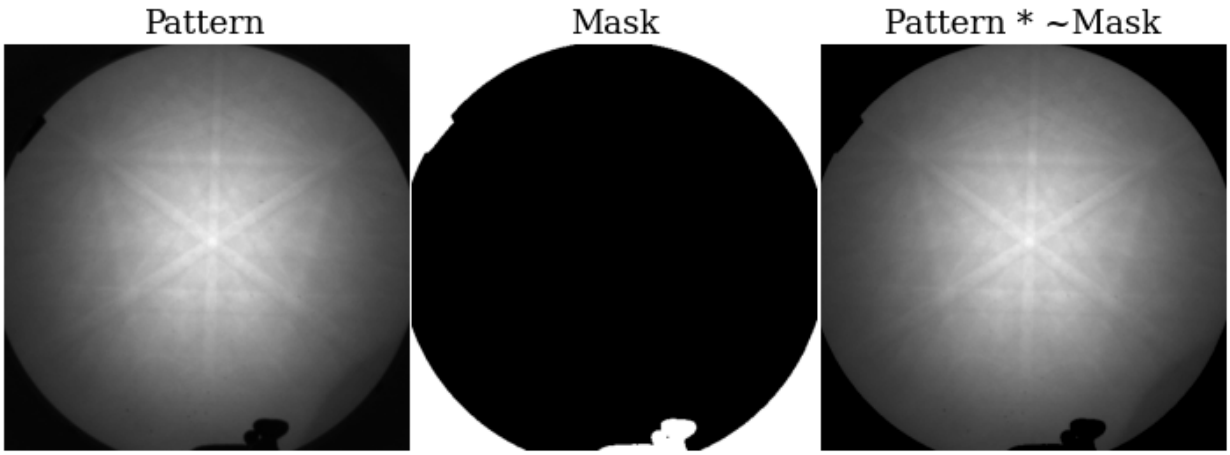
1. Extract an evenly spaced grid of patterns from the full dataset
2. Get initial guesses of the PC for each grid pattern with Hough indexing (from PyEBSDIndex)
3. Get initial guesses of the orientation for each grid pattern using the PCs with Hough indexing
4. Refine the orientations and PCs simultaneously by matching experimental to dynamically simulated patterns (simulated with EMsoft)
5. Fit a plane to the refined PCs

### Extract a grid of patterns

Before we extract the grid of patterns and prepare these for indexing by background correction, we obtain a binary mask to remove parts of the pattern without Kikuchi diffraction (typically the corners). For this dataset, we get the mask via thresholding. The threshold value is found by using the minimum threshold algorithm implemented in scikit-image (note that they have other thresholding algorithms that might work better for other datasets).

[7]:
```
s_mean = s.mean((0, 1))  # (480, 480)
mean_data = s_mean.data.compute()
```

[8]:
```
# Threshold
threshold = skf.threshold_minimum(mean_data)
signal_mask = mean_data < threshold  # Exclude pixels set to True

# Extract center pattern for plotting
p0 = s.inav[24, 24].data

fig, axes = plt.subplots(ncols=3, figsize=(12, 4))
for ax, arr, title in zip(
    axes,
    [p0, signal_mask, p0 * ~signal_mask],
    ["Pattern", "Mask", "Pattern * ~Mask"],
):
    ax.imshow(arr, cmap="gray")
    ax.axis("off")
    ax.set(title=title)
fig.subplots_adjust(wspace=0)
```




The mask efficiently removes the parts of the pattern which contains no information of interest.

Extract the grid using *EBSD.extract_grid()*

[9]:
```
grid_shape = (5, 5)
s_grid, idx = s.extract_grid(grid_shape, return_indices=True)

nav_shape_grid = s_grid.axes_manager.navigation_shape[::-1]
sig_shape_grid = s_grid.axes_manager.signal_shape[::-1]

s_grid.compute()

s_grid
```

```
[########################################] | 100% Completed | 103.18 ms
```

[9]:
```
<EBSD, title: Pattern, dimensions: (5, 5|480, 480)>
```

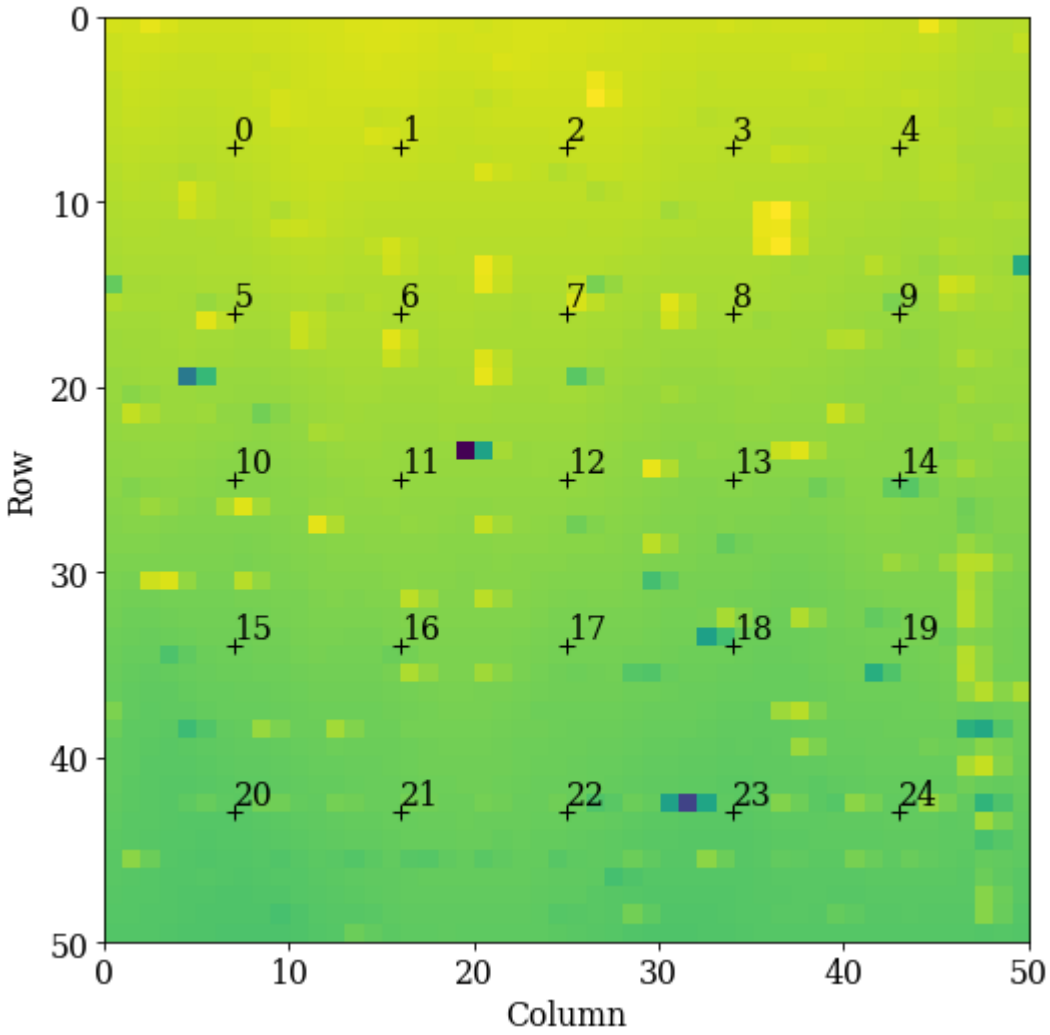
Let's plot the grid positions on the navigation map

[10]:
```
kp.draw.plot_pattern_positions_in_map(
```

(continues on next page)

(continued from previous page)

```
    rc=idx.reshape(2, -1).T,
    roi_shape=(50, 50),
    roi_image=s_mean_nav.data,
)
```




We know that all patterns are of silicon. To get a description of silicon, we could create a Phase manually. However, we will later on use a dynamically simulated EBSD master pattern of silicon (created with EMsoft), which is loaded with a `Phase`. We will use this in the remaining analysis.

[11]:
```
mp = kp.data.ebsd_master_pattern(
    "si", allow_download=True, projection="lambert", energy=20
)
mp
```

[11]:
```
<EBSDMasterPattern, title: si_mc_mp_20kv, dimensions: (|1001, 1001)>
```

Extract the phase and change lattice constant unit from nm to Ångström

[12]:
```
phase = mp.phase
```

(continues on next page)

(continued from previous page)

```
lat = phase.structure.lattice
lat.setLatPar(lat.a * 10, lat.b * 10, lat.c * 10)

print(phase)
print(phase.structure)
```

```
<name: si. space group: Fd-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
lattice=Lattice(a=5.4307, b=5.4307, c=5.4307, alpha=90, beta=90, gamma=90)
14   0.000000 0.000000 0.000000 1.0000
```

### Initial guess of PC using Hough indexing

We will estimate PCs and do Hough indexing of the grid of patterns using PyEBSDIndex. See the *Hough indexing tutorial* for more details on the Hough indexing related commands below.

---

**Note**

PyEBSDIndex is an optional dependency of kikuchipy. It can be installed with `pip` or `conda` (from `conda-forge`). To install PyEBSDIndex, see their installation instructions.

---

Prepare the patterns for indexing by background removal and setting the masked out values to 0

[13]:
```
s_grid.remove_static_background()
s_grid.remove_dynamic_background()
s_grid = s_grid * ~signal_mask
```

```
[########################################] | 100% Completed | 102.00 ms
[########################################] | 100% Completed | 202.01 ms
```

Get an initial EBSD detector to store PC values in

[14]:
```
det_grid = s_grid.detector.deepcopy()

print(det_grid)
print(det_grid.sample_tilt)
```

```
EBSDDetector (480, 480), px_size 1 um, binning 1, tilt 0.0, azimuthal 0.0, pc (0.5, 0.5,␣
↪0.5)
70.0
```

We need an EBSDIndexer to use `PyEBSDIndex`. We can obtain an indexer by passing a PhaseList to *EBSDDetector.get_indexer()*

[15]:
```
phase_list = PhaseList(phase)
phase_list
```

[15]:
```
Id  Name  Space group  Point group  Proper point group     Color
 0    si        Fd-3m         m-3m                 432  tab:blue
```

Get the indexer

[16]:
```
indexer = det_grid.get_indexer(phase_list, nBands=6)
```

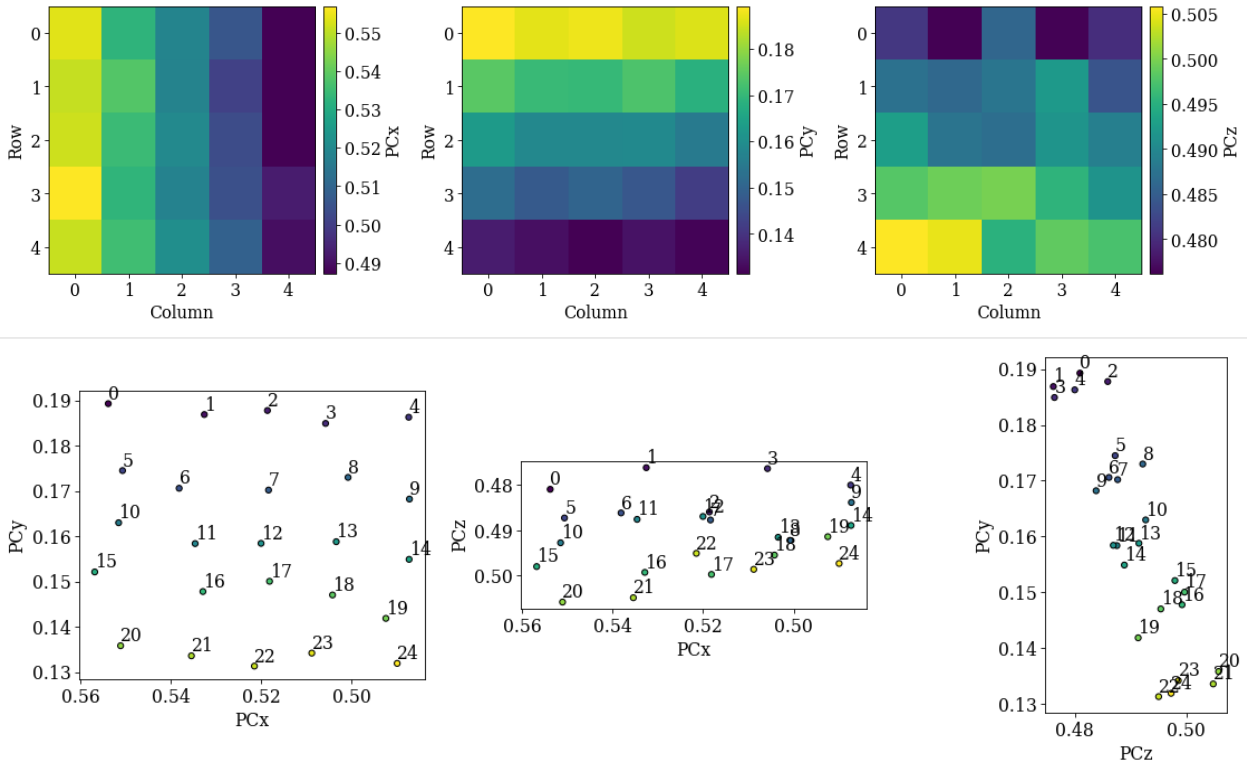
We estimate the PC of each pattern with Hough indexing using *EBSD.hough_indexing_optimize_pc()*, and plot both the mean and standard deviation of the resulting PCs. (We will "overwrite" the existing detector variable in memory.)

[17]:
```
det_grid = s_grid.hough_indexing_optimize_pc(
    pc0=[0.52, 0.16, 0.49],  # Initial guess based on previous experiments
    indexer=indexer,
    batch=True,
    method="PSO",
    search_limit=0.1,
)

print(det_grid.pc_flattened.mean(axis=0))
print(det_grid.pc_flattened.std(0))
```

```
PC found: [**********] 25/25  global best:0.274  PC opt:[0.4899 0.1319 0.4973]
[0.52006439 0.1596286  0.4907931 ]
[0.02247494 0.01867778 0.00789391]
```

Plot the PCs

[18]:
```
det_grid.plot_pc("map")
det_grid.plot_pc("scatter", annotate=True)
```




Looking at the maps, as expected:

- PCx increases towards the left
- PCy increases upwards

- PCz increases downwards

Looking at the scatter plots:

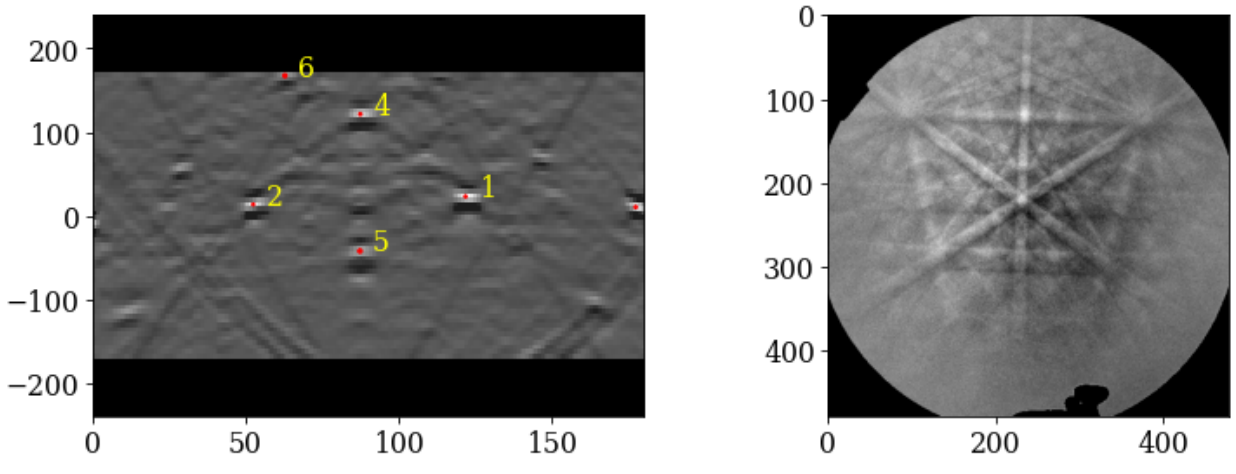
- We recognize the regular (5, 5) grid in the (PCx, PCy) plot.
- There is an inverse relation between PCy and PCz: as PCy decreases, PCz increases. This is as expected. However, there is a significant spread in each row of points (5-9, 10-14, etc.).

These initial guesses seem OK and we could perhaps fit a plane to these values already. But, we can get a better fit by refining the PC values using pattern matching. To that end, we obtain initial guesses for the orientations via Hough indexing with *EBSD.hough_indexing()*. We get a new indexer with the optimized PCs and then index

[19]:
```
indexer = det_grid.get_indexer(phase_list, nBands=6)
```

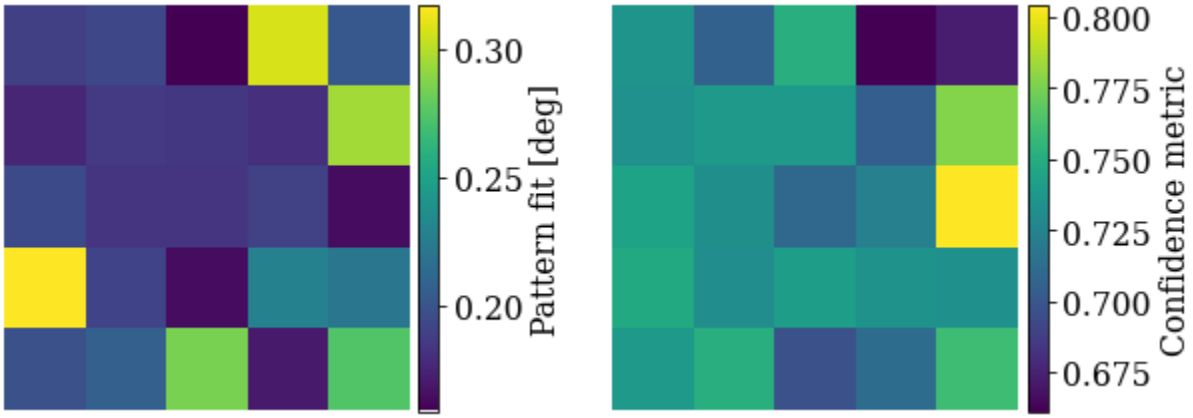
[20]:
```
xmap_grid = s_grid.hough_indexing(
    phase_list=phase_list, indexer=indexer, verbose=2
)

print(xmap_grid)
print(xmap_grid.fit.mean())
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: True
  Projection center (Bruker, mean): (0.5201, 0.1596, 0.4908)
  Indexing 25 pattern(s) in 1 chunk(s)
Radon Time: 0.04952027699619066
Convolution Time: 0.005532572002266534
Peak ID Time: 0.0023458810028387234
Band Label Time: 0.06669590099772904
Total Band Find Time: 0.12412348898942582
Band Vote Time:  0.020521485988865606
  Indexing speed: 147.07505 patterns/s
Phase  Orientations  Name  Space group  Point group  Proper point group     Color
    0   25 (100.0%)    si        Fd-3m         m-3m                 432  tab:blue
Properties: fit, cm, pq, nmatch
Scan unit: um
0.20949268
```




Check the average pattern fit and confidence metric

[21]:
```
fig, axes = plt.subplots(ncols=2, figsize=(10, 3.5))
for ax, to_plot, label in zip(
    axes, ["fit", "cm"], ["Pattern fit [deg]", "Confidence metric"]
):
    im = ax.imshow(xmap_grid.get_map_data(to_plot))
    fig.colorbar(im, ax=ax, label=label, pad=0.02)
    ax.axis("off")
fig.subplots_adjust(wspace=0.1)
```



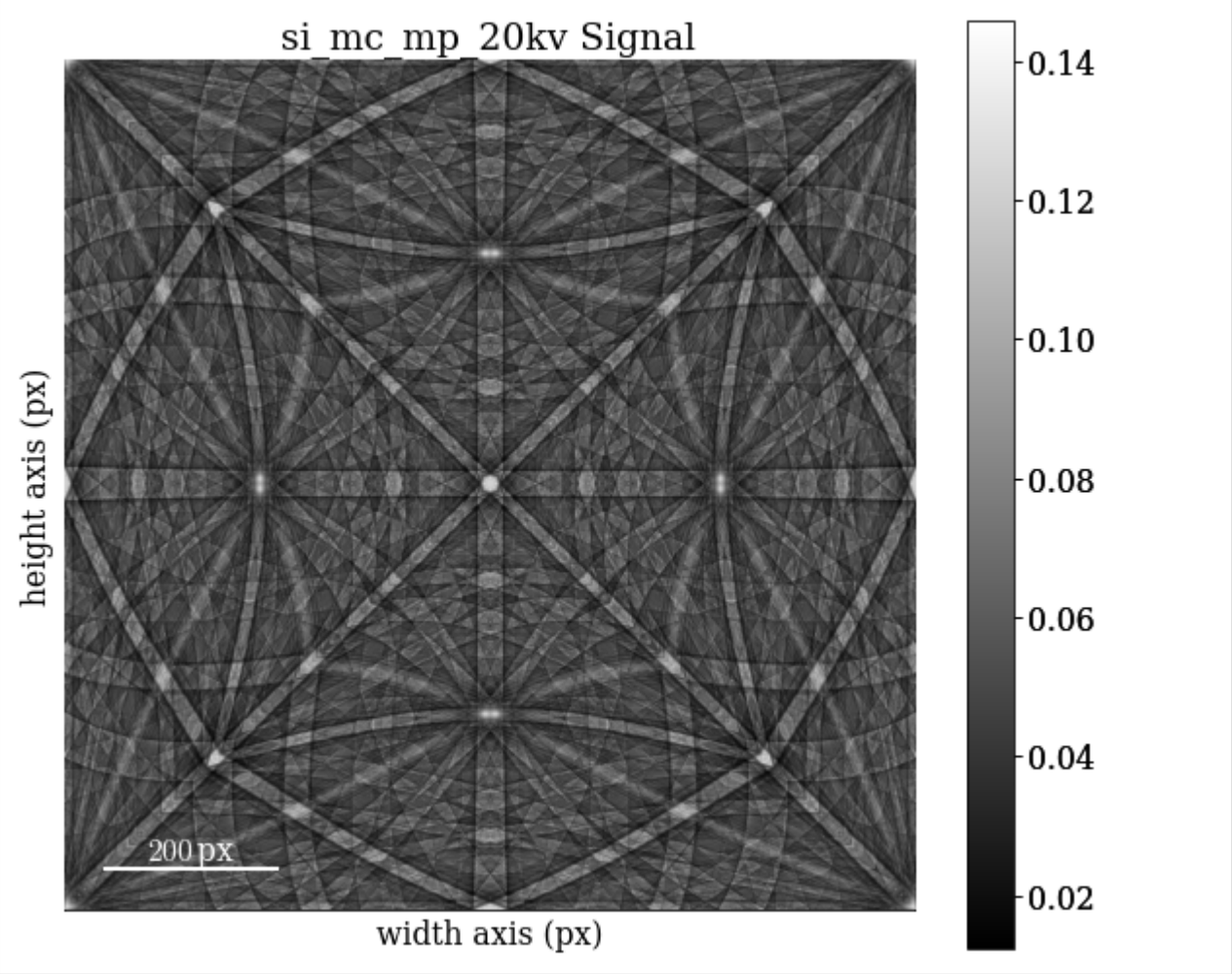

Most patterns have a low pattern fit and high confidence metric.

Let's refine these initial guesses of orientations and PCs using the dynamically simulated master pattern.

### Refine PCs using pattern matching

Let's look at the dynamical simulation (which is in the square Lambert projection, required for use in dictionary indexing and refinement in kikuchipy [and EMsoft])

[22]:
```
mp.plot()
```




Refine orientations and PCs using the Nelder-Mead optimization algorithm as implemented in NLopt. During optimization, orientations are represented as three Euler angles $(\phi_1, \Phi, \phi_2)$. The angles are varied within a trust region in degrees of the orientation obtained from Hough indexing above. PCs are varied within a trust region in percent of the detector width (for PCx and PCz) or height (for PCy). Optimization of each pattern stops when the increase in normalized cross-correlation (NCC) score between iterations, obtained by comparing experimental to simulated pattern, is lower than $10^{-5}$.

[23]:
```
xmap_grid_ref, det_grid_ref = s_grid.refine_orientation_projection_center(
    xmap=xmap_grid,
    detector=det_grid,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5, 0.05, 0.05, 0.05],
    rtol=1e-5,
    chunk_kwargs=dict(chunk_shape=1),
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
```

(continues on next page)

(continued from previous page)

```
  Trust region (+/-): [5.   5.   5.   0.05 0.05 0.05]
  Relative tolerance: 1e-05
Refining 25 orientation(s) and projection center(s):
[########################################] | 100% Completed | 114.76 s
Refinement speed: 0.21783 patterns/s
```

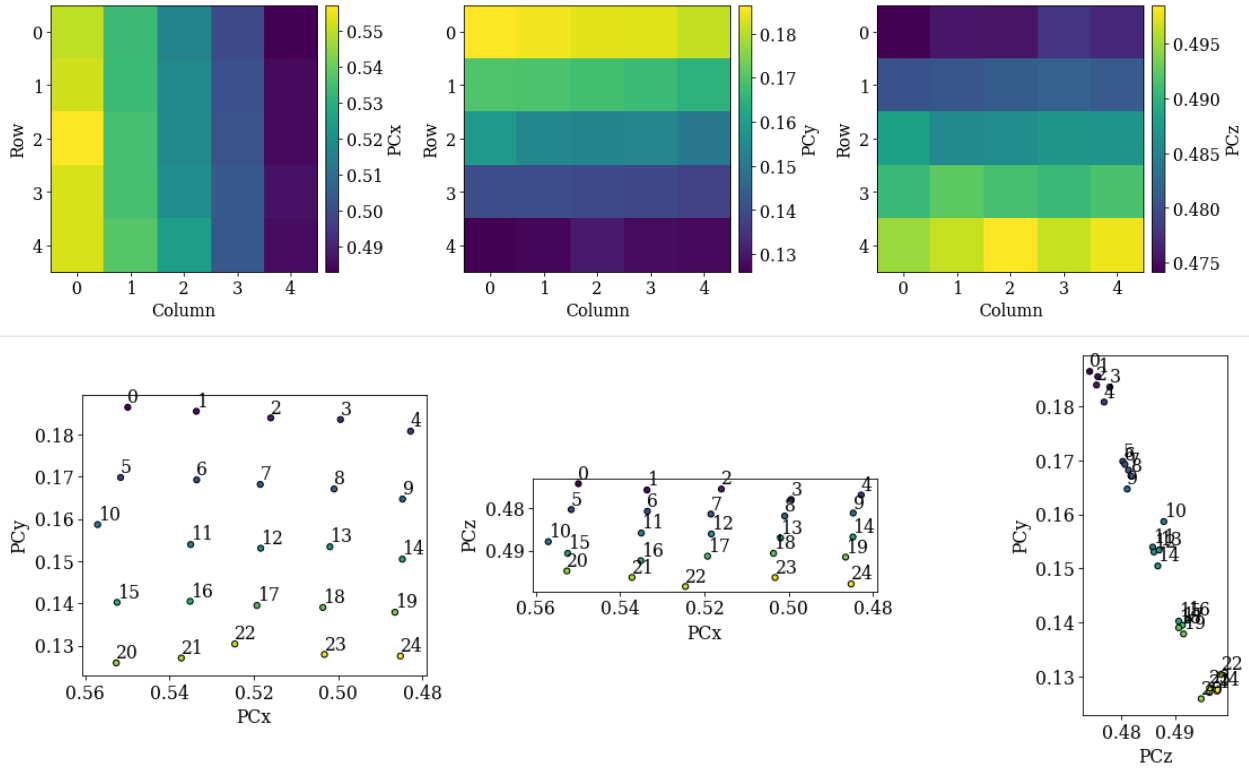
Check the average NCC score, number of evaluations, and PC and the standard deviation of the average PC

[24]:
```
print(xmap_grid_ref.scores.mean())
print(xmap_grid_ref.num_evals.mean())
print(det_grid_ref.pc_average)
print(det_grid_ref.pc_flattened.std(0))
```

```
0.33365421533584594
291.44
[0.51870718 0.15458022 0.48636834]
[0.02396429 0.02004163 0.00739694]
```

Check the distribution of refined PCs

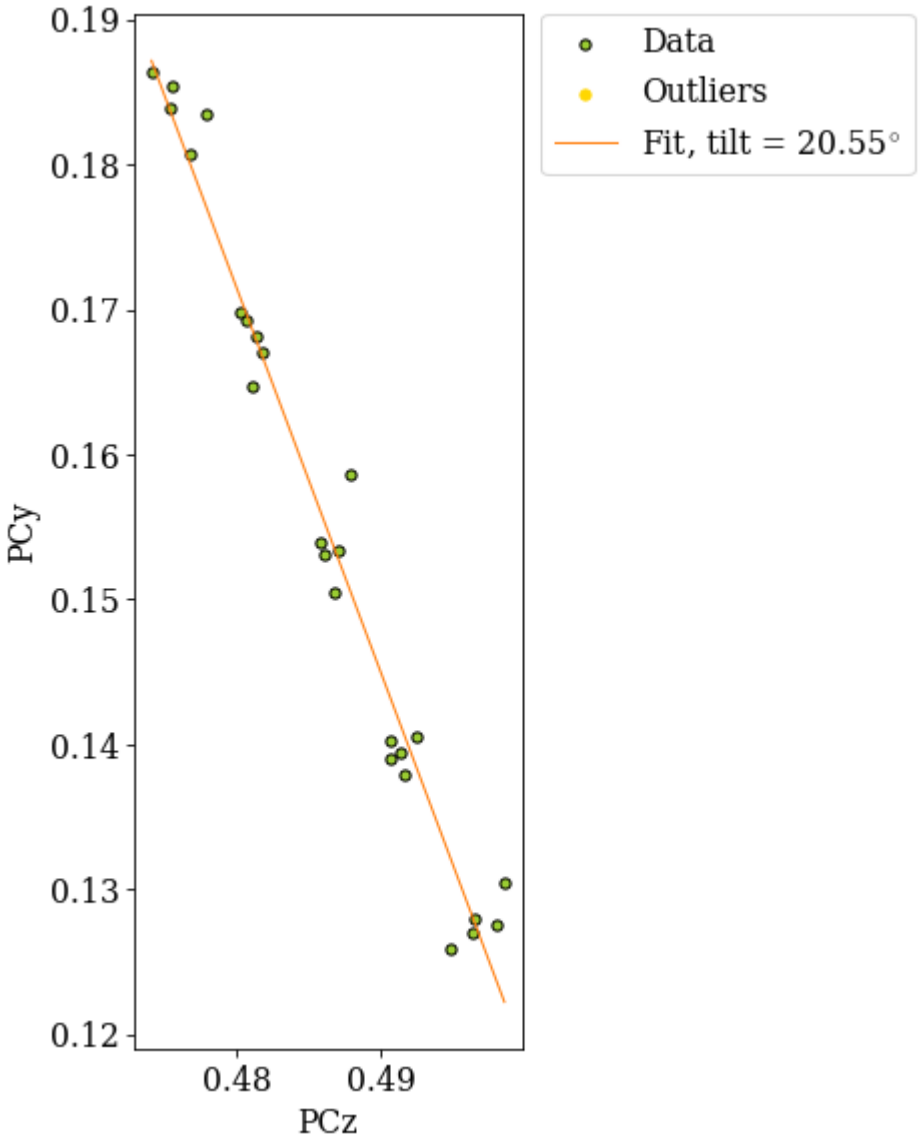
[25]:
```
det_grid_ref.plot_pc("map")
det_grid_ref.plot_pc("scatter", annotate=True)
```




The grid is easily recognizable in all three scatter plots now. If there were any outliers, we could exclude them by passing a boolean array when fitting a plane. This array can either be created manually, or we can try to find outliers by robust fitting of a line to the (PCz, PCy) plot. Assuming the sample is perfectly tilted about the detector $X_d$, the slope of the fitted line is the estimated tilt angle.

[26]:
```
xtilt, is_outlier = det_grid_ref.estimate_xtilt(
    detect_outliers=True, degrees=True, return_outliers=True
)
print(xtilt)
```

```
20.550490605557012
```




The sample was tilted to 70° from the horizontal, so we would expect an angle of about 20° from the vertical.

We can now fit a plane to the remaining PCs.

### Fit a plane to the refined PCs

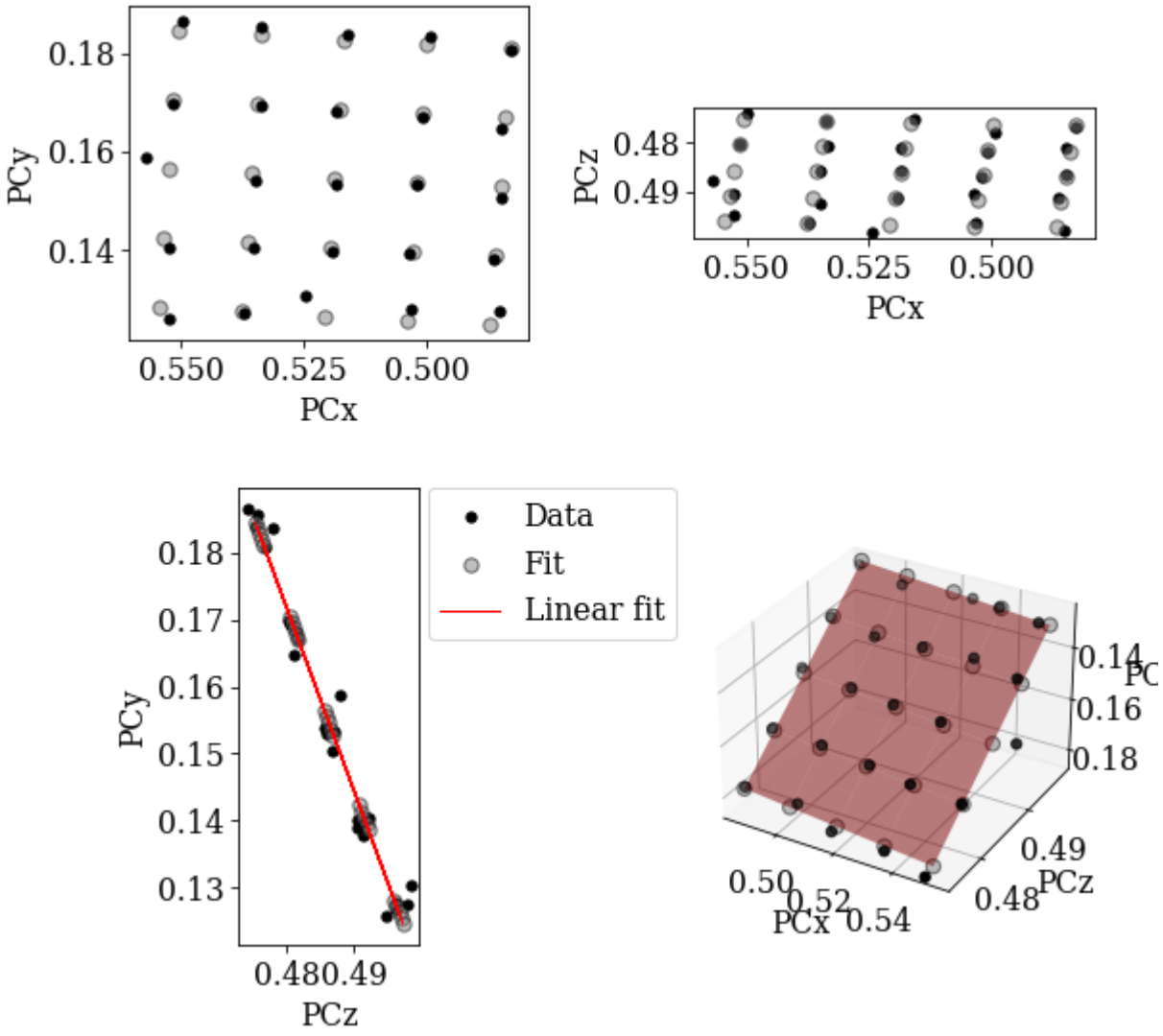
To fit a plane to the PCs, we must pass an array of all indices in the full map and the indices of the patterns which the PCs were estimated from. In the case of the grid patterns we've used here, the map indices were returned from the `EBSD.extract_grid()`.

[27]:
```
nav_shape = s.axes_manager.navigation_shape[::-1]
map_indices = np.indices(nav_shape)
```

Using *EBSDDetector.fit_pc()*, we will fit a plane to the PCs using both an affine transformation and a projective transformation. The fit method automatically plots the three scatter plots above and a forth 3D plot. The experimental PC values are black circles and the fitted PCs are larger gray circles.

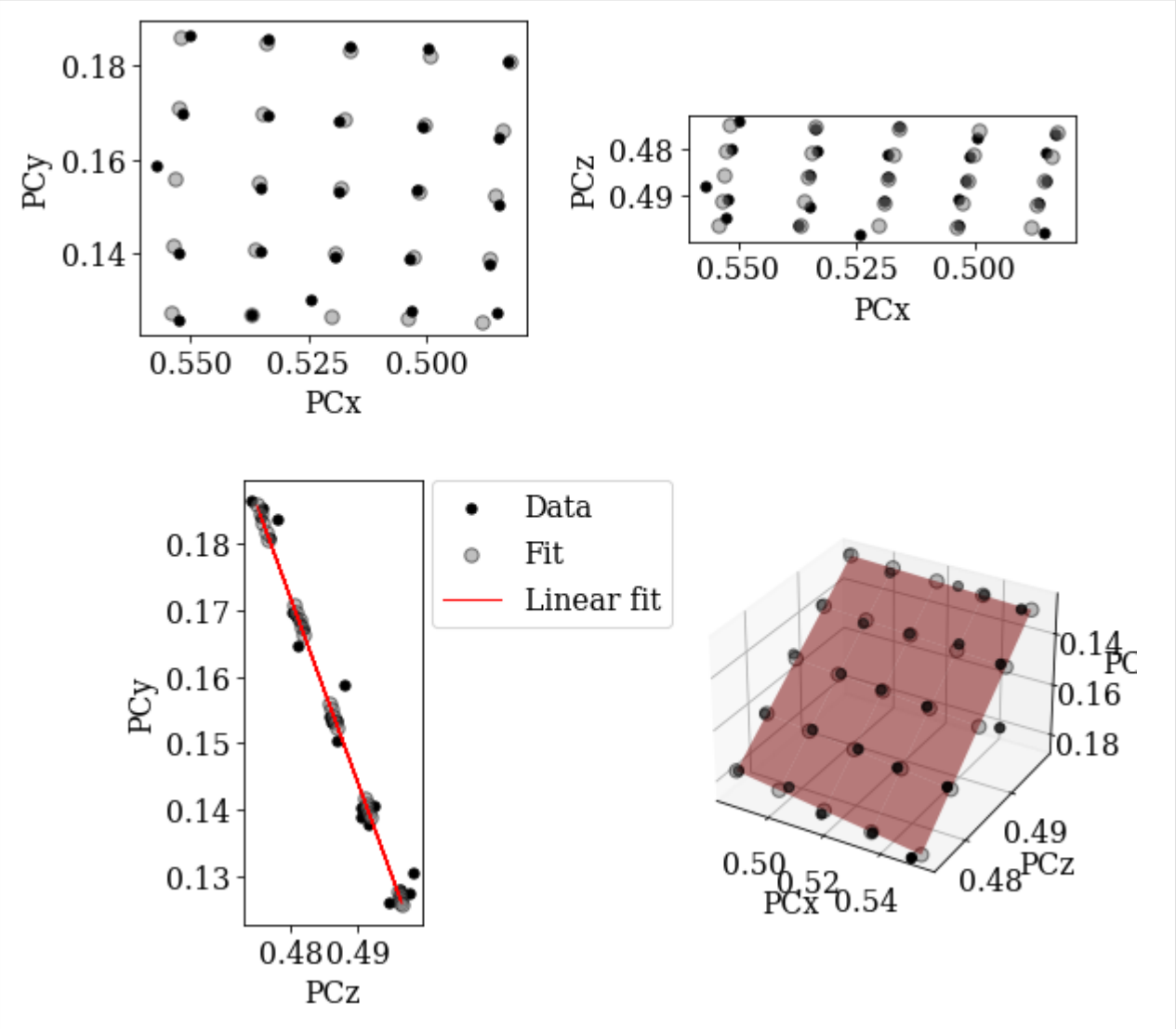
[28]:
```
det_ref_aff = det_grid_ref.fit_pc(
    idx,
    map_indices=map_indices,
    transformation="affine",
)
print(det_ref_aff)

# Sample tilt
print(det_ref_aff.sample_tilt)

# Max. deviation between experimental and fitted PC
pc_diff_aff = det_grid_ref.pc - det_ref_aff.pc[tuple(idx)]
print(abs(pc_diff_aff.reshape(-1, 3)).mean(axis=0))
```

```
EBSDDetector (480, 480), px_size 1 um, binning 1, tilt 0.0, azimuthal 0, pc (0.52, 0.155,
↪ 0.486)
69.8010018609663
[0.00092969 0.0014497  0.00065325]
```




[29]:
```
det_ref_proj = det_grid_ref.fit_pc(
    idx,
    map_indices=map_indices,
    transformation="projective",
)
print(det_ref_proj)

# Sample tilt
print(det_ref_proj.sample_tilt)

# Max. deviation between experimental and fitted PC
pc_diff_proj = det_grid_ref.pc - det_ref_proj.pc[tuple(idx)]
print(abs(pc_diff_proj.reshape(-1, 3)).mean(axis=0))
```

```
EBSDDetector (480, 480), px_size 1 um, binning 1, tilt 0.0, azimuthal 0, pc (0.52, 0.155,
↪ 0.486)
69.96587791952271
[0.00105687 0.00110206 0.00063361]
```




Both planes fit the experimental PCs well.

### Validate fitted PCs

Finally, we refine the (already refined) orientations of the grid patterns using the above fitted PCs. We validate our results by comparing geometrical simulations to the patterns.

[30]:
```
det_ref_proj_grid = det_ref_proj.deepcopy()
det_ref_proj_grid.pc = det_ref_proj_grid.pc[tuple(idx)]
```

[31]:
```
xmap_grid_ref2 = s_grid.refine_orientation(
    xmap=xmap_grid_ref,
    detector=det_ref_proj_grid,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
```

(continues on next page)

(continued from previous page)

```
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5],
    rtol=1e-5,
    chunk_kwargs=dict(chunk_shape=1),
)

print(xmap_grid_ref2.scores.mean())
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
  Relative tolerance: 1e-05
Refining 25 orientation(s):
[########################################] | 100% Completed | 20.83 ss
Refinement speed: 1.19951 patterns/s
0.3322068232297897
```

To plot geometrical simulations on top of our experimental patterns, we create a *KikuchiPatternSimulator* using the silicon Phase from the master pattern above (see the *geometrical EBSD simulations tutorial* for more details)

[32]:
```
ref = ReciprocalLatticeVector.from_min_dspacing(phase)

# Ensure a complete unit cell (potentially changes the number of atoms)
ref.sanitise_phase()

ref.calculate_structure_factor()
F = abs(ref.structure_factor)
ref = ref[F > 0.4 * F.max()]

ref.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    3.135    18.4     338.2     100.0      8
 2 2 0    1.920    15.0     224.0      66.2      12
 4 0 0    1.358     9.2     83.7       24.8      6
 3 1 1    1.637     8.5     72.4       21.4      24
```

[33]:
```
simulator = kp.simulations.KikuchiPatternSimulator(ref)
```

Get one simulation per pattern

[34]:
```
sim = simulator.on_detector(
    det_ref_proj_grid,
    xmap_grid_ref2.rotations.reshape(*xmap_grid_ref2.shape),
)
```

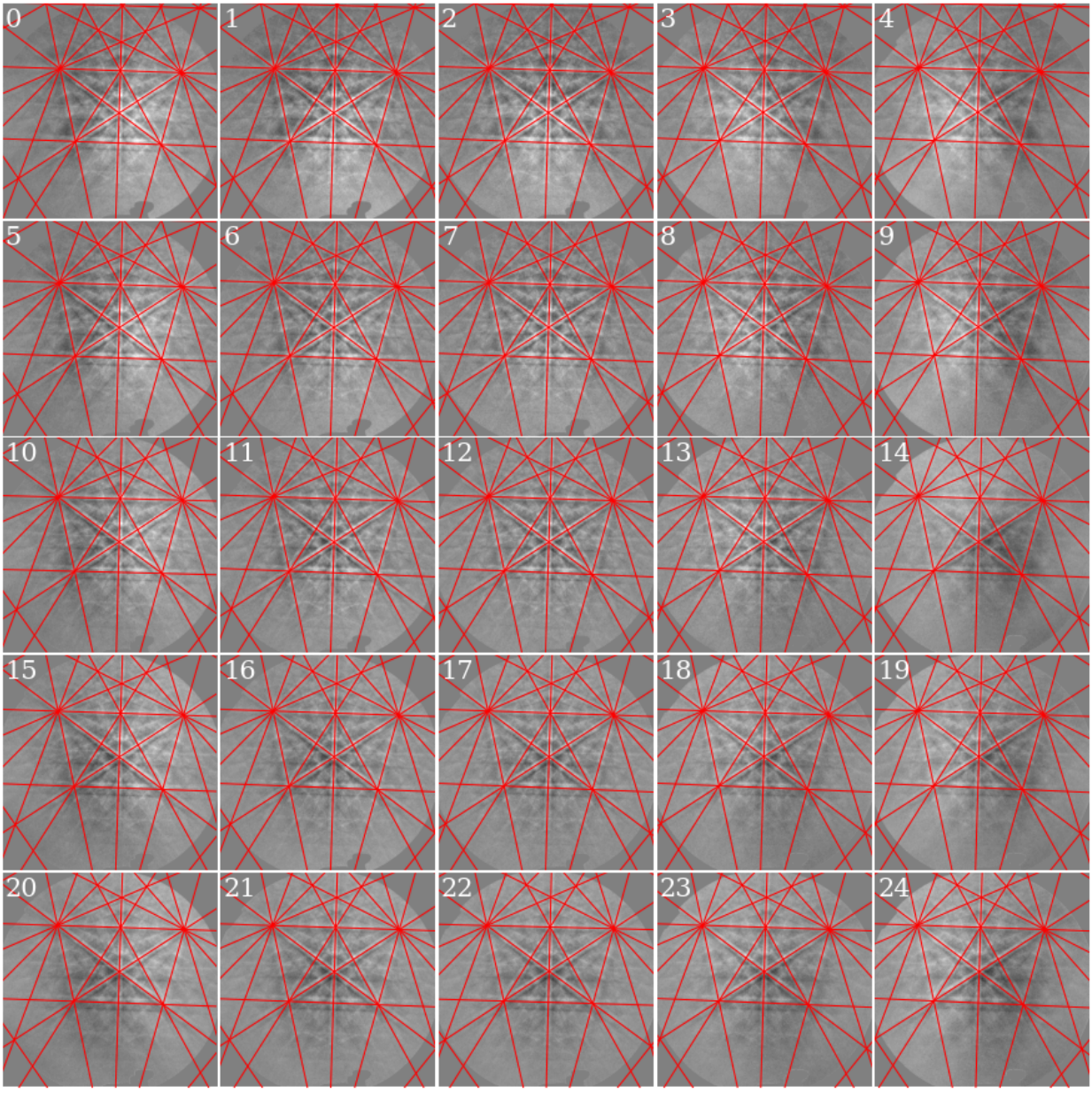
```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 103.73 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.46 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 100.83 ms
```

Plot the geometrical simulations on top of the patterns (after normalization to a mean of 0 and standard deviation of 1)

[35]:
```
s_grid2 = s_grid.normalize_intensity(dtype_out="float32", inplace=False)
```

```
[########################################] | 100% Completed | 100.92 ms
```

[36]:
```
fig, axes = plt.subplots(*grid_shape, figsize=(15, 15))
for i in np.ndindex(grid_shape):
    axes[i].imshow((s_grid2.data[i] * ~signal_mask), cmap="gray", vmin=-3, vmax=3)
    axes[i].axis("off")

    lines = sim.as_collections(i)[0]
    axes[i].add_collection(lines)

    idx1d = np.ravel_multi_index(i, grid_shape)
    axes[i].text(5, 10, idx1d, c="w", va="top", ha="left", fontsize=20)

fig.subplots_adjust(wspace=0.01, hspace=0.01)
```

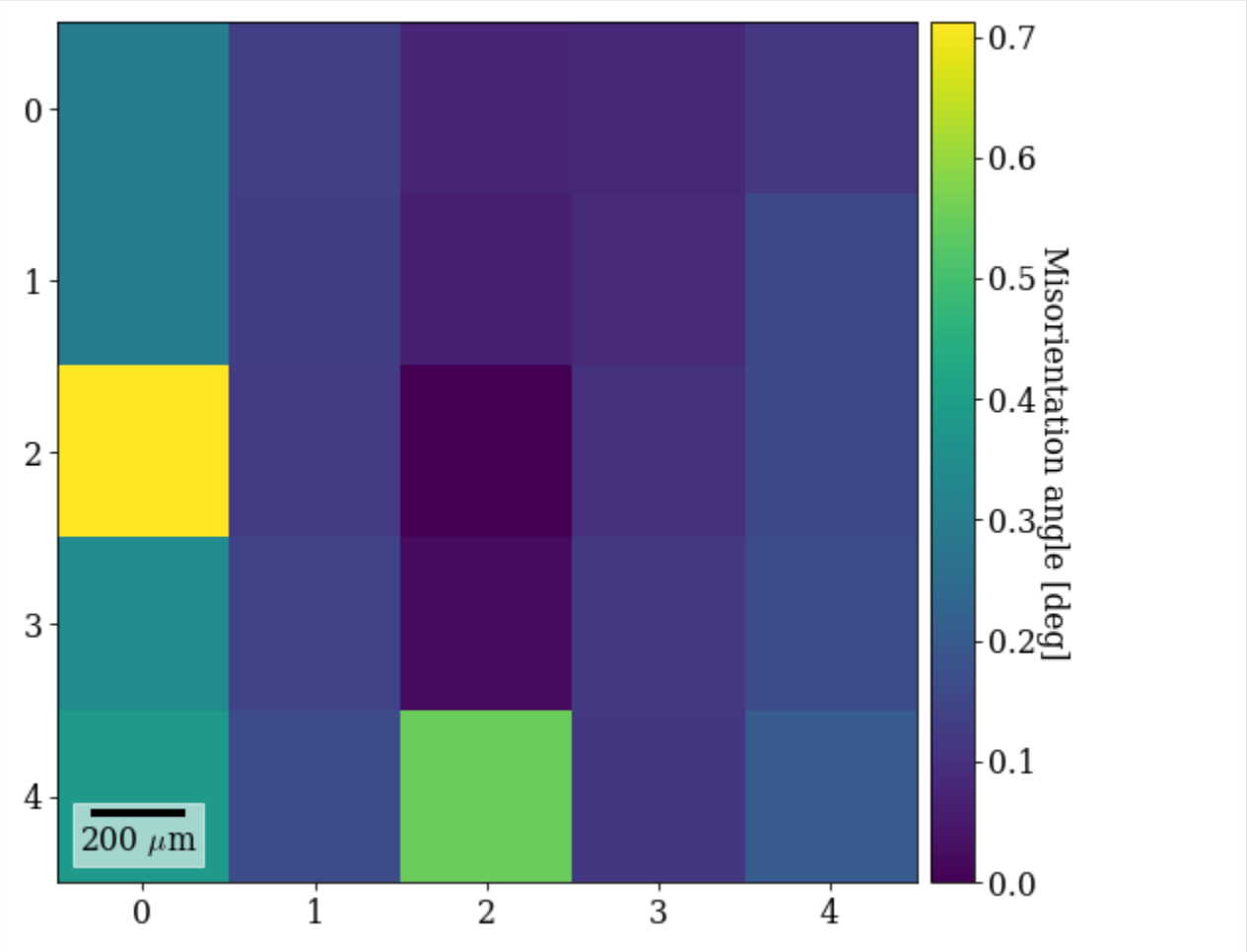



As expected, if we look at the first pattern (0), we see that the patterns shift upwards and to the left as we move down (see how the upper-most near-horizontal Kikuchi line disappears) and to the right (see how the left-most zone axis disappears) in the grid.

The patterns come from a single crystal. How misoriented is the estimated orientation for the center pattern to the other orientations?

[37]:
```
angles = xmap_grid_ref2.orientations[12].angle_with(
    xmap_grid_ref2.orientations, degrees=True
)
print(angles.max())
xmap_grid_ref2.plot(angles, colorbar=True, colorbar_label="Misorientation angle [deg]")
```

```
0.7114465443203103
```




We see that it is the least misoriented to the closest patterns, while the misorientation increases outwards radially. All misorientation angles are < 1°.

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### Extrapolate projection centers from a mean

In this tutorial, we will extrapolate a plane of projection centers (PCs) from a mean PC. The PCs are determined from patterns spread out across the sample region of interest (ROI). This is an alternative to *fitting* a plane to PCs, as is demonstrated in the tutorial *Fit a plane to selected projection centers*. As a validation of the extrapolated PCs, we will compare them to the PCs obtained from fitting a plane to the PCs.

We'll start by importing the necessary libraries

[1]:
```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from diffsims.crystallography import ReciprocalLatticeVector
import hyperspy.api as hs
import kikuchipy as kp
from orix.crystal_map import PhaseList


plt.rcParams.update(
    {
        "figure.facecolor": "w",
        "figure.dpi": 75,
        "figure.figsize": (8, 8),
        "font.size": 15,
    }
)
```

### Load and inspect data

We will use nine calibration patterns from recrystallized nickel. The patterns are acquired with a NORDIF UF-1100 EBSD detector with the full (480, 480) $px^2$ resolution. These patterns should always be acquired from spread out sample positions across the ROI in order to calibrate the PCs prior to indexing the full dataset.

Read the calibration patterns

[2]:
```
s_cal = kp.data.ni_gain_calibration(1, allow_download=True)
s_cal
```

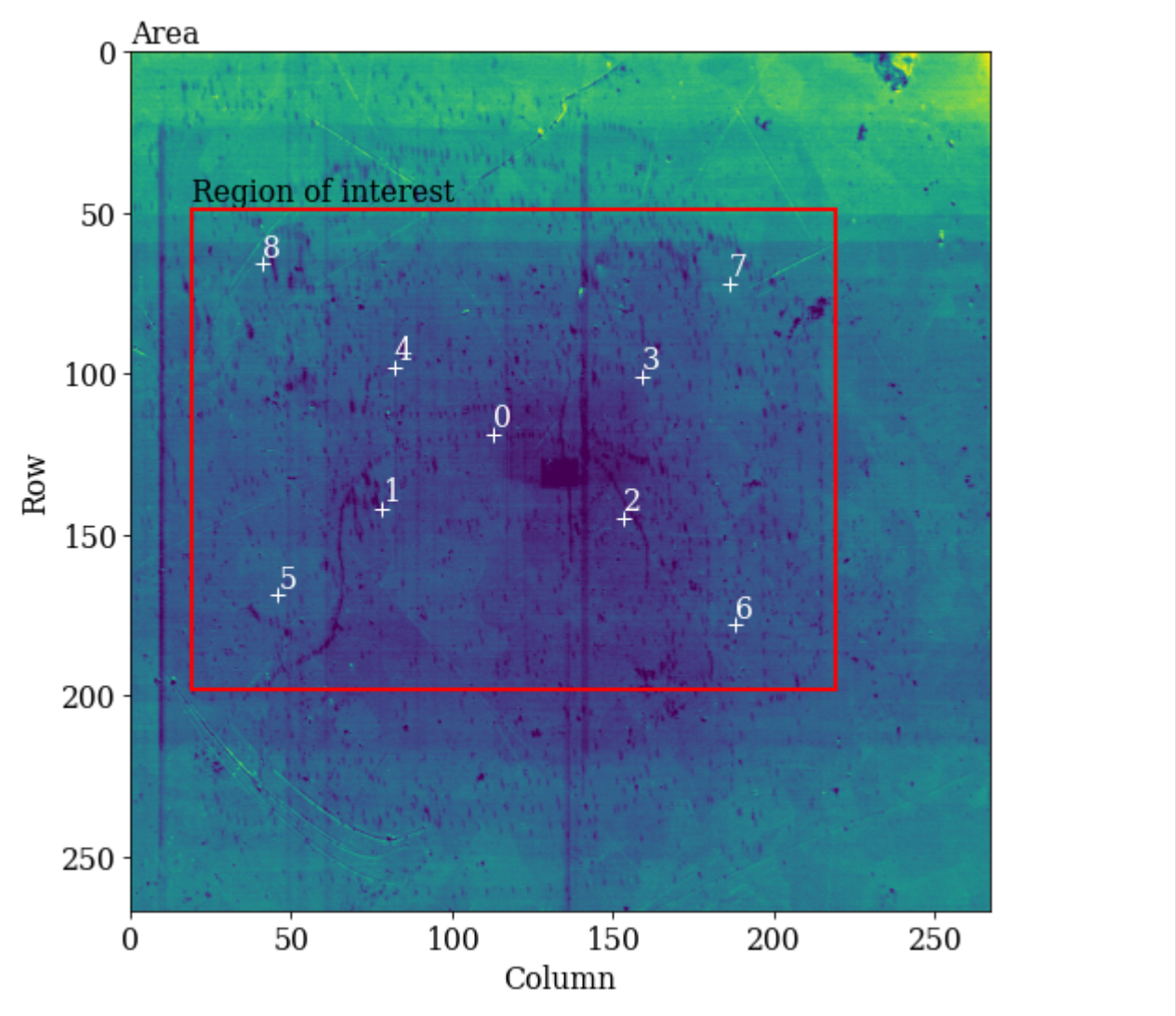
[2]:
```
<EBSD, title: Calibration patterns, dimensions: (9|480, 480)>
```

Get information read from the NORDIF settings file

[3]:
```
omd = s_cal.original_metadata.as_dictionary()
```

Plot coordinates of calibration patterns on the secondary electron area overview image (part of the dataset), highlighting the ROI

[4]:
```
kp.draw.plot_pattern_positions_in_map(
    rc=omd["calibration_patterns"]["indices_scaled"],
    roi_shape=omd["roi"]["shape_scaled"],
    roi_origin=omd["roi"]["origin_scaled"],
    area_shape=omd["area"]["shape_scaled"],
    area_image=omd["area_image"],
    color="w",
)
```




Improve signal-to-noise ratio by removing the static and dynamic background

[5]:
```
s_cal.remove_static_background("divide")
s_cal.remove_dynamic_background("divide")
```

```
[########################################] | 100% Completed | 101.53 ms
[########################################] | 100% Completed | 101.24 ms
```

Let's plot the nine background-corrected calibration patterns.

Before we do that, though, we find a suitable signal mask. The mask should exclude parts of the pattern without Kikuchi diffraction. Previous EBSD experiments on the microscope showed that the maximum intensity on the detector is a little to the left of the detector center. We therefore use a circular mask to exclude intensities in the upper and lower right corners of the detector

[6]:
```
r_pattern = kp.filters.distance_to_origin(s_cal.axes_manager.signal_shape[::-1],␣
↪origin=(230, 220))
signal_mask = r_pattern > 310  # Exclude pixels set to True
```

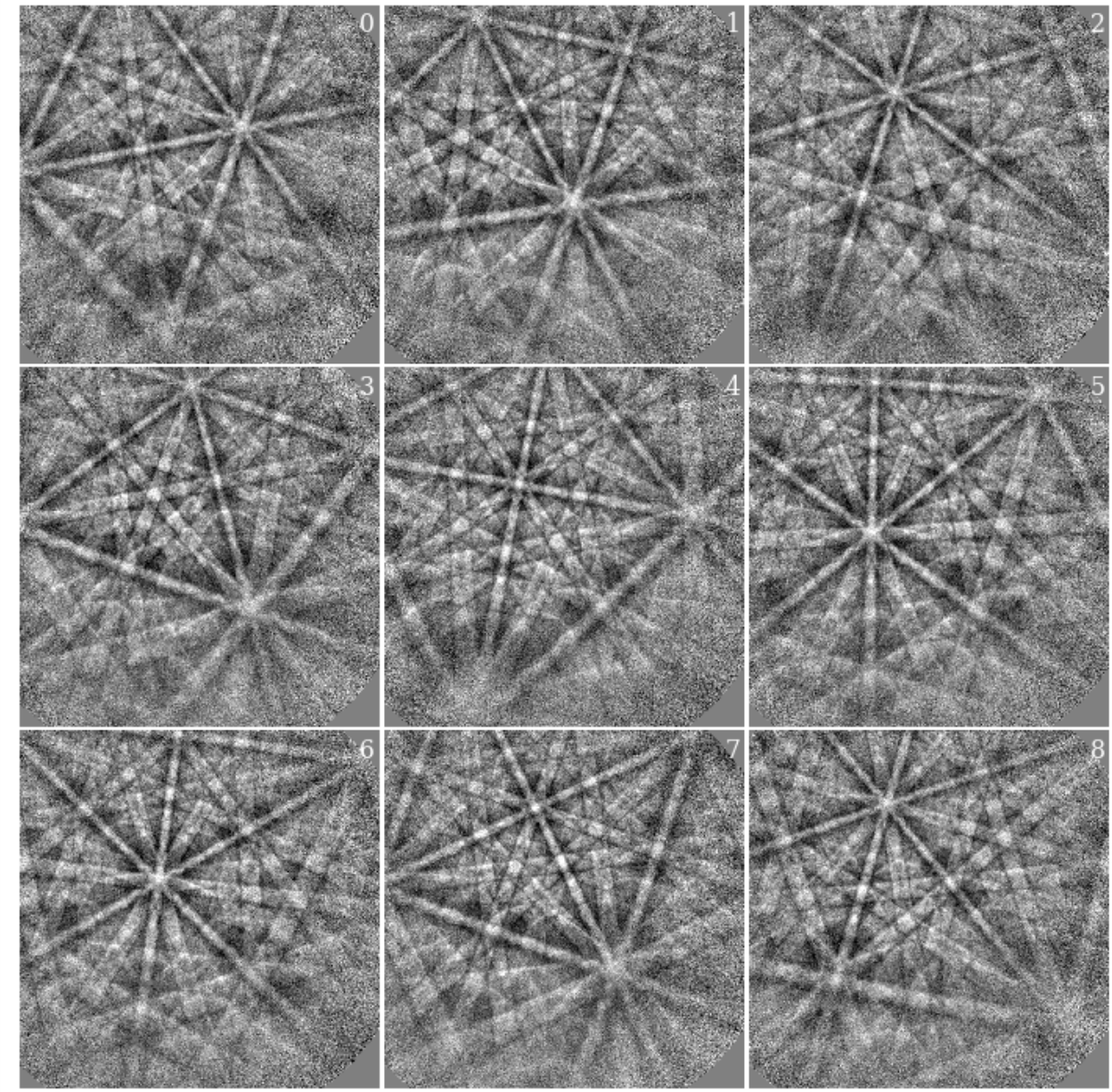
For visual display only, we normalize intensities to a mean of 0 and a standard deviation of 1. We also exclude extreme

intensities outside the range [-3, 3].

[7]:
```
s_cal2 = s_cal.normalize_intensity(dtype_out="float32", inplace=False)
```

```
[########################################] | 100% Completed | 101.56 ms
```

[8]:
```
fig = plt.figure(figsize=(12, 12))
_ = hs.plot.plot_images(
    s_cal2 * ~signal_mask,
    per_row=3,
    axes_decor=None,
    colorbar=False,
    label=None,
    fig=fig,
    vmin=-3,
    vmax=3,
)
for i, ax in enumerate(fig.axes):
    ax.axis("off")
    ax.text(475, 10, str(i), va="top", ha="right", c="w")
fig.subplots_adjust(wspace=0.01, hspace=0.01)
```




We know that all patterns are of nickel. To get a description of nickel, we could create a Phase manually. However, we will later on use a dynamically simulated EBSD master pattern of nickel (created with EMsoft), which is loaded with a `Phase`. We will use this in the remaining analysis.

[9]:
```
mp = kp.data.ebsd_master_pattern(
    "ni", allow_download=True, projection="lambert", energy=20
)
mp
```

[9]:
```
<EBSDMasterPattern, title: ni_mc_mp_20kv, dimensions: (|1001, 1001)>
```

Extract the phase, and change the lattice parameter unit from nm to Ångström

[10]:
```
phase = mp.phase

lat = phase.structure.lattice
lat.setLatPar(lat.a * 10, lat.b * 10, lat.c * 10)
```

[11]:
```
print(phase)
print(phase.structure)
```

```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
lattice=Lattice(a=3.5236, b=3.5236, c=3.5236, alpha=90, beta=90, gamma=90)
28   0.000000 0.000000 0.000000 1.0000
```

### Estimate PCs with Hough indexing

We will estimate PCs for the nine calibration patterns using PyEBSDIndex. See the *Hough indexing tutorial* for more details.

**Note**

PyEBSDIndex is an optional dependency of kikuchipy, and can be installed with both `pip` and `conda` (from `conda-forge`). To install PyEBSDIndex, see their installation instructions.

We need an EBSDIndexer to use `PyEBSDIndex`. We can obtain an indexer by passing a PhaseList to *EBSDDetector.get_indexer()*. Therefore, we need an initial EBSD detector

[12]:
```
det_cal = s_cal.detector

print(det_cal)
print(det_cal.sample_tilt)
```

```
EBSDDetector (480, 480), px_size 1 um, binning 1, tilt 0.0, azimuthal 0.0, pc (0.5, 0.5,␣
↪0.5)
70.0
```

[13]:
```
phase_list = PhaseList(phase)
phase_list
```

[13]:
```
Id  Name  Space group  Point group  Proper point group     Color
 0    ni        Fm-3m         m-3m                 432  tab:blue
```

[14]:
```
indexer = det_cal.get_indexer(phase_list, rhoMaskFrac=0.05)

print(indexer.phaselist[0].phasename)
print(indexer.bandDetectPlan.rhoMaskFrac)
```

```
ni
0.05
```

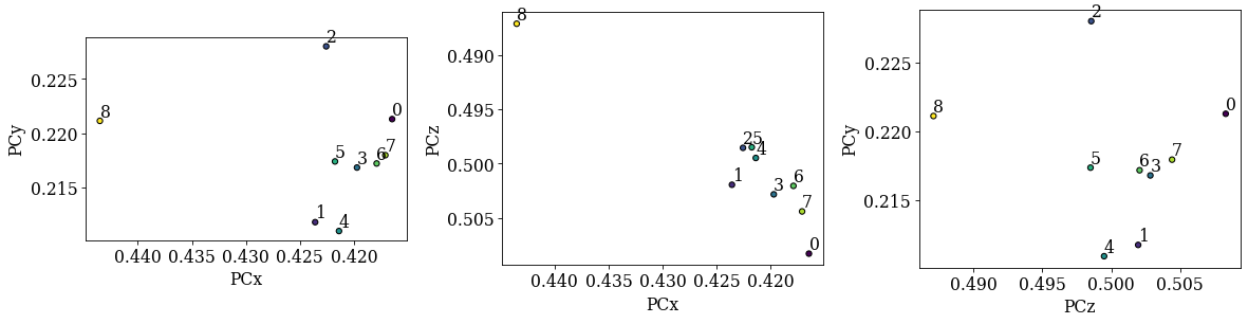
We estimate the PC of each pattern with Hough indexing, and plot both the mean and standard deviation of the resulting PCs. Note that Bruker's PC convention is used in kikuchipy. (We will "overwrite" the existing detector variable.)

[15]:
```
det_cal = s_cal.hough_indexing_optimize_pc(
    pc0=[0.42, 0.21, 0.50],  # Initial guess based on previous experiments
    indexer=indexer,
    batch=True,
    method="PSO",
    search_limit=0.05,
)

print(det_cal.pc.mean(axis=0))
print(det_cal.pc.std(0))
```

```
PC found: [********* ] 9/9  global best:0.161  PC opt:[0.4435 0.2211 0.4871]
[0.42267129 0.21805602 0.50035695]
[0.00773081 0.00484972 0.00552956]
```

Plot the PCs

[16]:
```
det_cal.plot_pc("scatter", annotate=True)
```



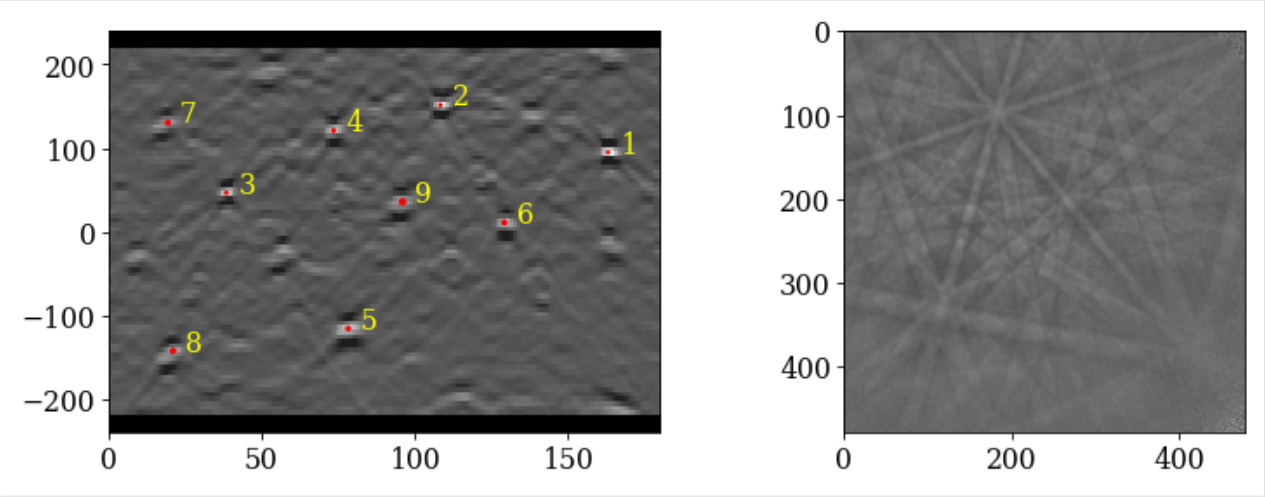

Unfortunately, we do not recognize the spatial distribution from the overview image above. The expected inverse relation between (PCz, PCy) is not present either. We can try to improve indexing by refining the PCs using dynamical simulations. These simulations are created with EMsoft.

First, we need an initial guess of the orientations, which we get using Hough indexing via *EBSD.hough_indexing()*. We will use the mean PC for all patterns

[17]:
```
indexer.PC = det_cal.pc_average
```

[18]:
```
xmap_hi = s_cal.hough_indexing(
    phase_list=phase_list, indexer=indexer, verbose=2
)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: True
  Projection center (Bruker): (0.4227, 0.2181, 0.5004)
  Indexing 9 pattern(s) in 1 chunk(s)
Radon Time: 0.037286512000719085
Convolution Time: 0.005005180006264709
Peak ID Time: 0.002896414996939711 3
Band Label Time: 0.0441566069930559
Total Band Find Time: 0.08939686599478591
Band Vote Time:  0.014271903011831455
  Indexing speed: 66.11647 patterns/s
```




[19]:
```
print(xmap_hi)
print(xmap_hi.fit.mean())
```

```
Phase  Orientations  Name  Space group  Point group  Proper point group     Color
    0    9 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: fit, cm, pq, nmatch
Scan unit: um
0.25232032
```

### Refine PCs with pattern matching

Refine the PCs (and orientations) using the Nelder-Mead implementation from NLopt

[20]:
```
xmap_ref, det_ref = s_cal.refine_orientation_projection_center(
    xmap=xmap_hi,
    detector=det_cal,
    master_pattern=mp,
    signal_mask=signal_mask,
    energy=20,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5, 0.05, 0.05, 0.05],
    rtol=1e-6,
    # A pattern per iteration to use all CPUs
    chunk_kwargs=dict(chunk_shape=1),
)
```

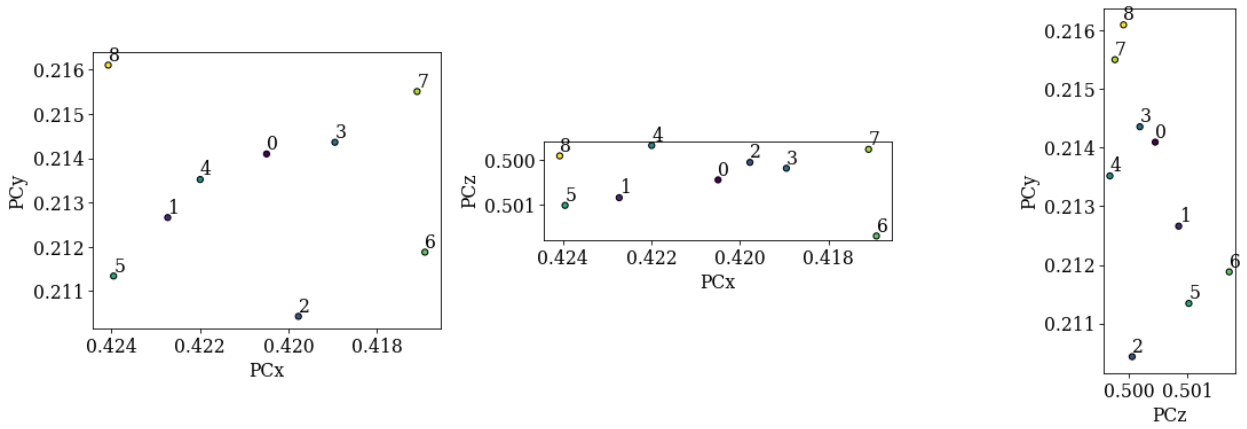
```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5.   5.   5.   0.05 0.05 0.05]
  Relative tolerance: 1e-06
Refining 9 orientation(s) and projection center(s):
[########################################] | 100% Completed | 53.70 ss
Refinement speed: 0.16759 patterns/s
```

Inspect some refinement statistics

[21]:
```
print("Score mean:                ", xmap_ref.scores.mean())
print("Mean number of evaluations:", xmap_ref.num_evals.mean())
print("PC mean:     ", det_ref.pc.mean(0))
print("PC std:      ", det_ref.pc.std(0))
print("PC max. diff:", abs(det_cal.pc - det_ref.pc).max(0))

angles = xmap_hi.orientations.angle_with(xmap_ref.orientations, degrees=True)
print("Ori. max. diff [deg]:", angles.max())
```

```
Score mean:                0.5227384467919668
Mean number of evaluations: 339.55555555555554
PC mean:      [0.42066694 0.21332143 0.50039889]
PC std:       [0.00256103 0.00179425 0.00063848]
PC max. diff: [0.0194249  0.0175794  0.01281866]
Ori. max. diff [deg]: 1.3225756229285888
```

[22]:
```
det_ref.plot_pc("scatter", annotate=True)
```




The diagonals 5-1-0-3-7 and 8-4-0-2-6 seen in the overview image above should be replicated in the (PCx, PCy) scatter plot. We see that 5-1-0-3-7 and 8-0-6 align as expected, but the PC values from the 2nd and 4th patterns do not lie in the expected range. Fitting a plane to all these values might not work to our satisfaction, so we will exclude the 2nd and 4th PC values when fitting a plane to the seven remaining PC values. The plane will have PC values for all points in the ROI in the overview image above.

[23]:
```
is_outlier = np.zeros(det_ref.navigation_size, dtype=bool)
is_outlier[[2, 4]] = True
```

### Get PC plane by fitting

The fitting is done by finding a transformation function which takes 2D sample coordinates and gives PC values for those coordinates. Both an affine and a projective transformation function is supported, following [Winkelmann *et al.*, 2020]. By passing 2D indices of all ROI map points and of the points where the nine calibration patterns were obtained, *EBSDDetector.fit_pc()* returns a new detector with PC values for all map points. We will use the maximum difference between the above refined PC values and the corresponding fitted PC values as a measure of how good the fitted PC values are.

[24]:
```
pc_indices = omd["calibration_patterns"]["indices_scaled"]
pc_indices -= omd["roi"]["origin_scaled"]
```

(continues on next page)

(continued from previous page)

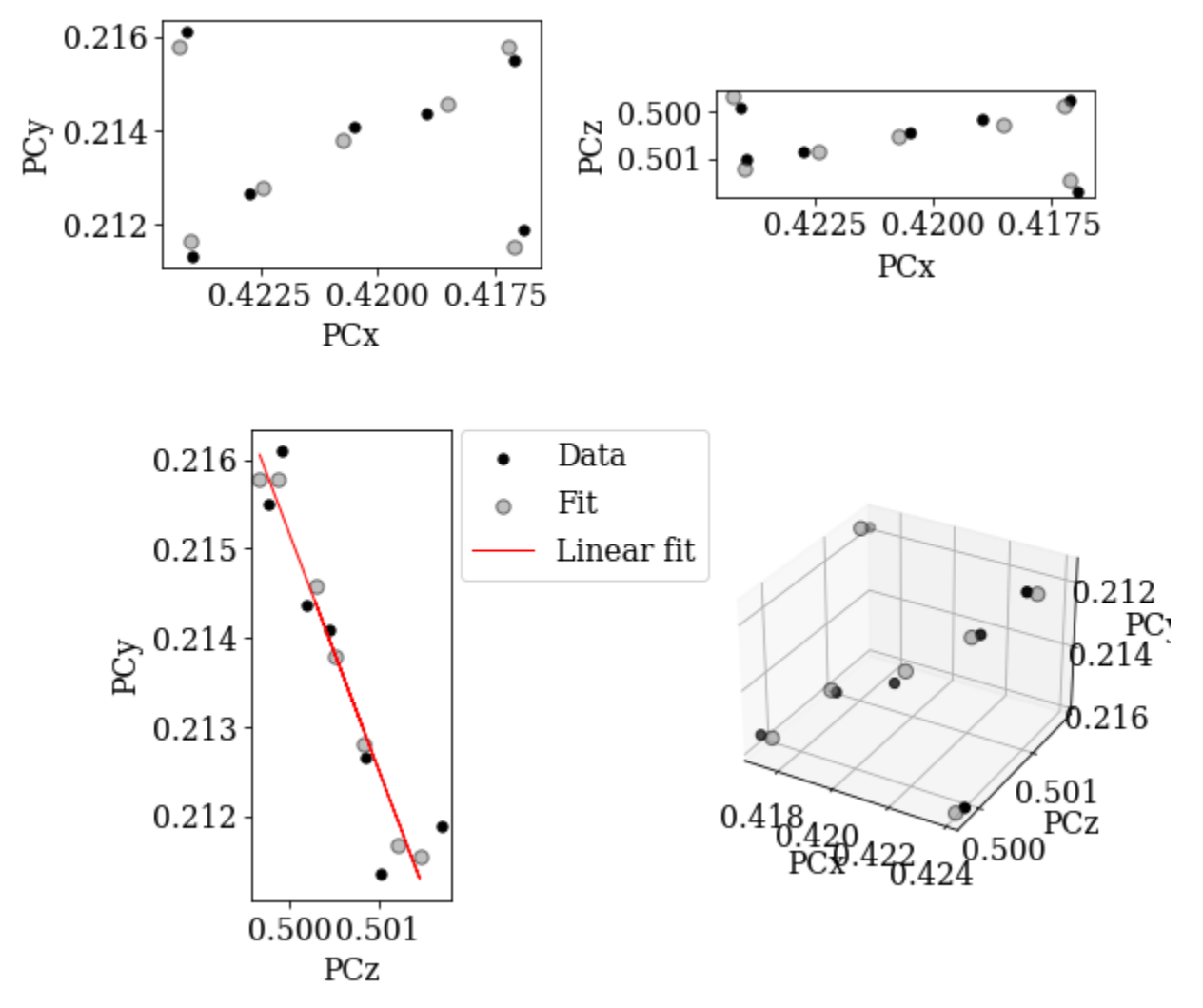
```
pc_indices = pc_indices.T

map_indices = np.indices(omd["roi"]["shape_scaled"])
print("Full map shape (n rows, n columns):", omd["roi"]["shape_scaled"])
```

```
Full map shape (n rows, n columns): (149, 200)
```

Fit PC values using the affine transformation function

[25]:
```
det_fit_aff = det_ref.fit_pc(
    pc_indices=pc_indices,
    map_indices=map_indices,
    transformation="affine",
    is_outlier=is_outlier,
)

print(det_fit_aff.pc_average)
print(det_fit_aff.sample_tilt)
```

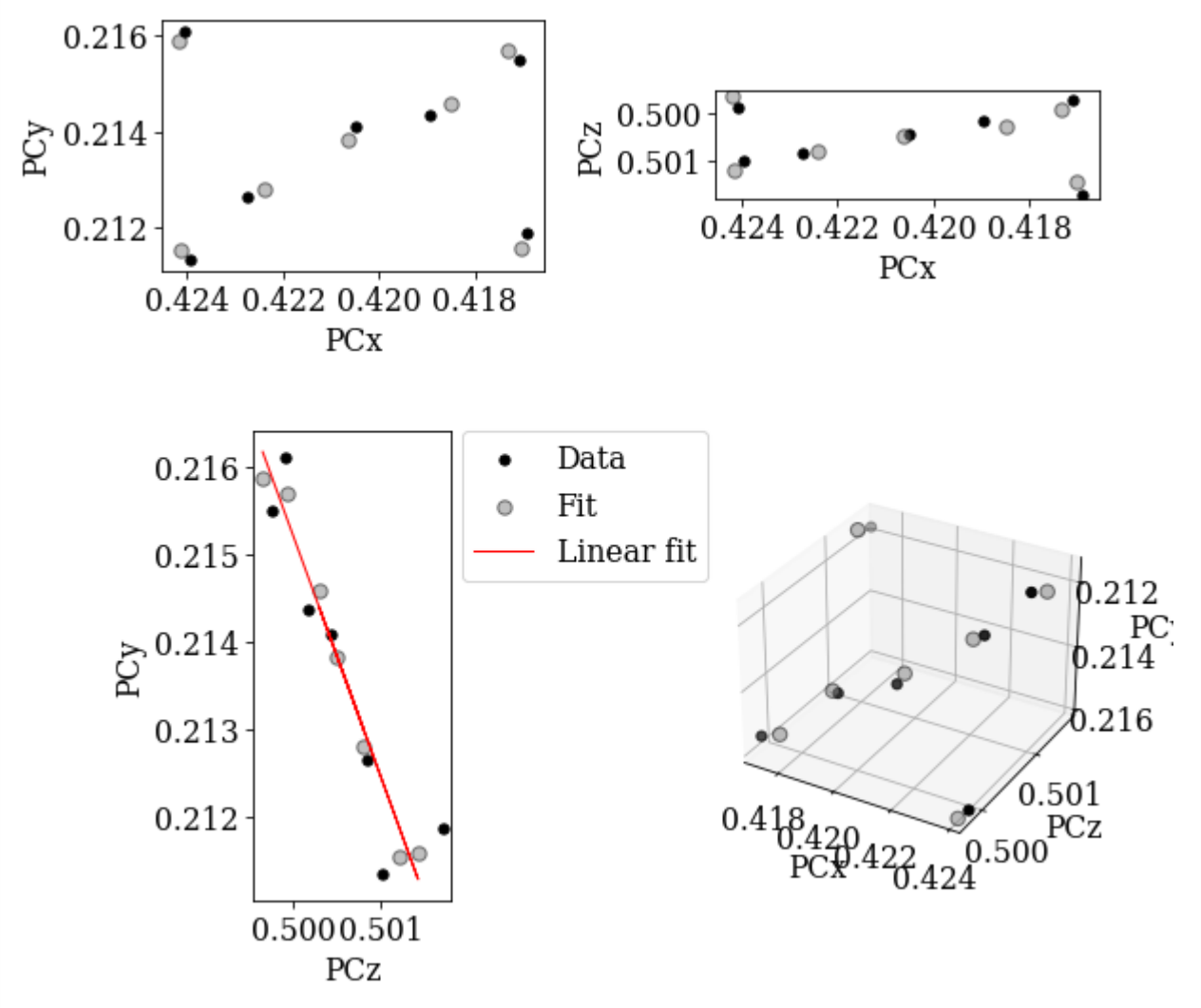
```
[0.42047592 0.21363019 0.50058492]
69.28188534139477
```




Fit PC values using the projective transformation function

[26]:
```
det_fit_proj = det_ref.fit_pc(
    pc_indices=pc_indices,
    map_indices=map_indices,
    transformation="projective",
    is_outlier=is_outlier,
)

print(det_fit_proj.pc_average)
print(det_fit_proj.sample_tilt)
```

```
[0.42047208 0.21362613 0.50058565]
70.01869762605253
```




Compare PC differences of all but the PC values from the 2nd and 4th patterns

[27]:
```
pc_indices2 = pc_indices.T[~is_outlier].T

# Refined PC values as a reference (ground truth)
pc_ref = det_ref.pc[~is_outlier]

# Difference in PC values from the affine transformation function
pc_diff_aff = det_fit_aff.pc[tuple(pc_indices2)] - pc_ref
pc_diff_aff_max = abs(pc_diff_aff).max(axis=0)
print(pc_diff_aff_max)

# Difference in PC values from the projective transformation function
pc_diff_proj = det_fit_proj.pc[tuple(pc_indices2)] - pc_ref
pc_diff_proj_max = abs(pc_diff_proj).max(axis=0)
print(pc_diff_proj_max)

# Which fitted PCs are more different from refined PCs, the projective (True)
# or the affine (False)?
```

(continues on next page)

(continued from previous page)

```
print(pc_diff_proj_max > pc_diff_aff_max)
```

```
[0.00043651 0.00033697 0.00024425]
[0.00046658 0.00029737 0.0002805 ]
[ True False  True]
```

### Get PC plane by extrapolation

Instead of fitting a plane to several PCs, we can extrapolate an average PC using *EBSDDetector.extrapolate_pc()*. To do this we need to know the detector pixel size and map step sizes, both given in the same unit.

[28]:
```
det_ext = det_ref.extrapolate_pc(
    pc_indices=pc_indices,
    navigation_shape=omd["roi"]["shape_scaled"],
    step_sizes=(1.5, 1.5),  # um
    shape=det_cal.shape,
    px_size=70,  # In um. This is unique for every detector model!
    is_outlier=is_outlier,
)

print(det_ext.pc_average)
print(det_ext.sample_tilt)
```

```
[0.41955486 0.21404149 0.50043058]
70.0
```

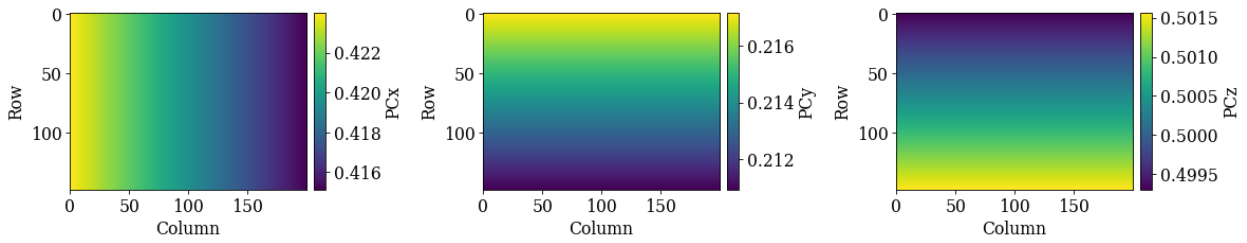
Difference in PC values between the refined PCs and the corresponding extrapolated PCs

[29]:
```
pc_diff_ext = det_ext.pc[tuple(pc_indices2)] - pc_ref
pc_diff_ext_max = abs(pc_diff_ext).max(axis=0)
print(pc_diff_ext_max)
print(pc_diff_ext_max > pc_diff_proj_max)
```

```
[0.00136875 0.00077    0.00044116]
[ True  True  True]
```

The extrapolated PCs deviate more from the refined PCs than the PCs obtained from fitting, although the difference is small.

[30]:
```
det_ext.plot_pc()
```



### Validate extrapolated PCs

As a final check of the difference in PCs, we can plot the geometrical simulations on top of the patterns using the refined orientation but the three different PCs.

[31]:
```
ref = ReciprocalLatticeVector.from_min_dspacing(phase.deepcopy())

ref.sanitise_phase()  # "Fill atoms in the unit cell"

ref.calculate_structure_factor()

F = abs(ref.structure_factor)
ref = ref[F > 0.5 * F.max()]

ref.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    2.034    11.8     140.0     100.0      8
 2 0 0    1.762    10.4     108.2      77.3      6
 2 2 0    1.246     7.4      55.0      39.3      12
 3 1 1    1.062     6.2      38.6      27.6      24
```

[32]:
```
simulator = kp.simulations.KikuchiPatternSimulator(ref)
```

Using PCs from the affine transformation function

[33]:
```
det_aff_cal = det_fit_aff.deepcopy()
det_aff_cal.pc = det_aff_cal.pc[tuple(pc_indices)]
```

[34]:
```
xmap_aff = s_cal.refine_orientation(
    xmap=xmap_hi,
    detector=det_aff_cal,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5],
    rtol=1e-6,
    chunk_kwargs=dict(chunk_shape=1),
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
  Relative tolerance: 1e-06
Refining 9 orientation(s):
[########################################] | 100% Completed | 10.39 ss
Refinement speed: 0.86597 patterns/s
```

[35]:
```
sim_aff = simulator.on_detector(det_aff_cal, xmap_aff.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.38 ms
Finding zone axes that are in some pattern:
```

(continues on next page)

(continued from previous page)

```
[########################################] | 100% Completed | 101.89 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.84 ms
```

Using PCs from the projective transformation function

[36]:
```
det_proj_cal = det_fit_proj.deepcopy()
det_proj_cal.pc = det_proj_cal.pc[tuple(pc_indices)]
```

[37]:
```
xmap_proj = s_cal.refine_orientation(
    xmap=xmap_hi,
    detector=det_proj_cal,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5],
    rtol=1e-6,
    chunk_kwargs=dict(chunk_shape=1),
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
  Relative tolerance: 1e-06
Refining 9 orientation(s):
[########################################] | 100% Completed | 10.47 ss
Refinement speed: 0.85929 patterns/s
```

[38]:
```
sim_proj = simulator.on_detector(det_proj_cal, xmap_proj.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.36 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 103.01 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.56 ms
```

Using the extrapolated PCs

[39]:
```
det_ext_cal = det_ext.deepcopy()
det_ext_cal.pc = det_ext_cal.pc[tuple(pc_indices)]
```

[40]:
```
xmap_ext = s_cal.refine_orientation(
    xmap=xmap_hi,
    detector=det_ext_cal,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    trust_region=[5, 5, 5],
    rtol=1e-6,
```

(continues on next page)

(continued from previous page)

```
    chunk_kwargs=dict(chunk_shape=1),
)
```

```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): [5 5 5]
  Relative tolerance: 1e-06
Refining 9 orientation(s):
[########################################] | 100% Completed | 9.71 sms
Refinement speed: 0.92637 patterns/s
```

[41]:
```
sim_ext = simulator.on_detector(det_ext_cal, xmap_ext.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.19 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.27 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.37 ms
```

Compare normalized cross-correlation scores and number of evaluations (iterations)

[42]:
```
print("Scores\n------")
print(f"Affine:       {xmap_aff.scores.mean():.7f}")
print(f"Projective:   {xmap_proj.scores.mean():.7f}")
print(f"Extrapolated: {xmap_ext.scores.mean():.7f}\n")

print("Number of evaluations\n---------------------")
print(f"Affine:       {xmap_aff.num_evals.mean():.1f}")
print(f"Projective:   {xmap_proj.num_evals.mean():.1f}")
print(f"Extrapolated: {xmap_ext.num_evals.mean():.1f}")
```

```
Scores
------
Affine:       0.5225310
Projective:   0.5225224
Extrapolated: 0.5221936

Number of evaluations
---------------------
Affine:       94.3
Projective:   95.2
Extrapolated: 88.6
```

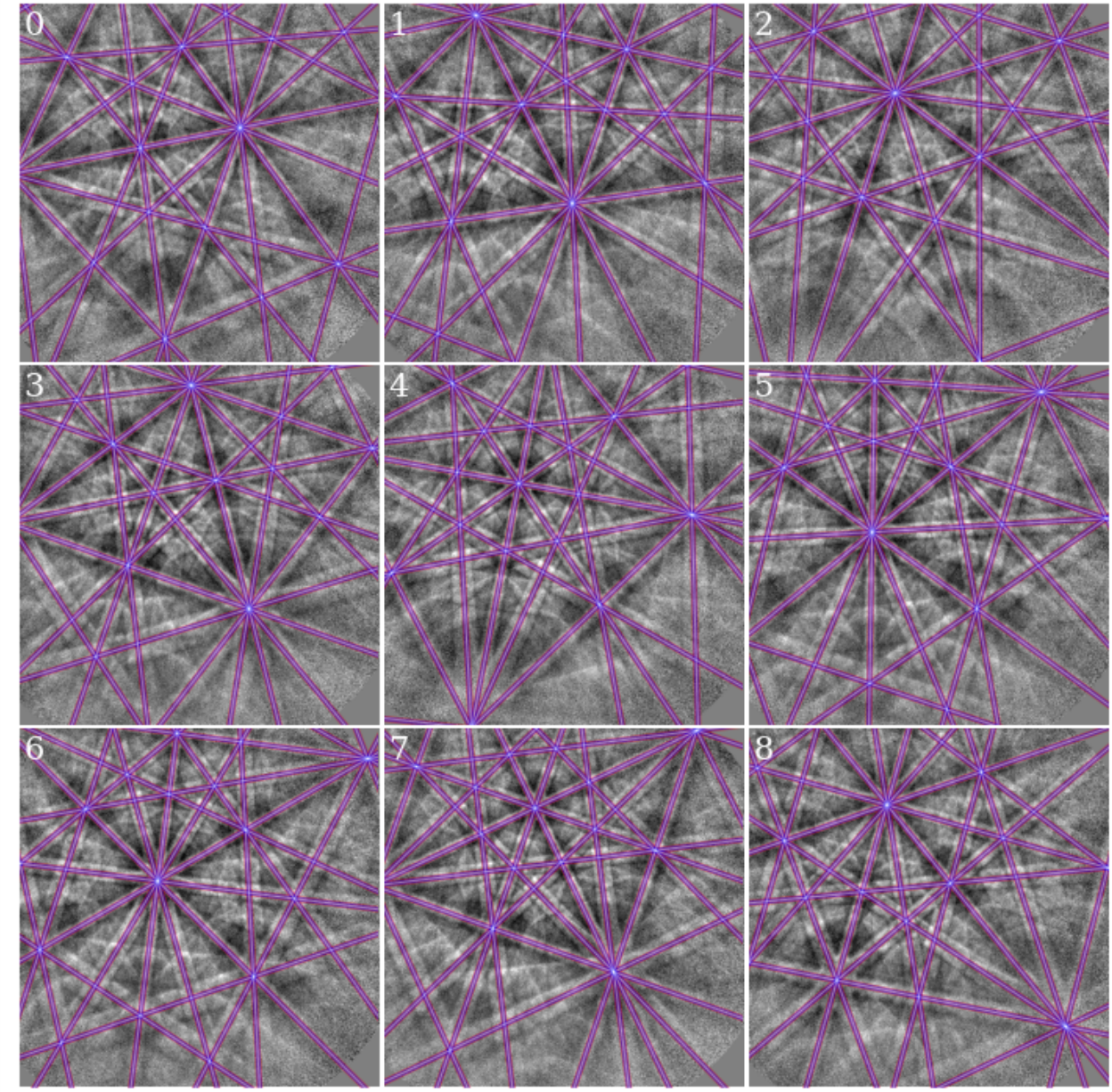
Plot Kikuchi bands on top of patterns for the solutions using the affine transformed PCs (red), projective transformed PCs (blue), and extrapolated PCs (white)

[43]:
```
fig, axes = plt.subplots(ncols=3, nrows=3, figsize=(12, 12))
for i, ax in enumerate(axes.ravel()):
    ax.imshow(s_cal2.inav[i].data * ~signal_mask, cmap="gray", vmin=-3, vmax=3)
    ax.axis("off")

    # Affine
    lines = sim_aff.as_collections(
```

(continues on next page)

(continued from previous page)

```
        i, lines_kwargs={"linewidth": 4, "alpha": 0.4}
    )[0]
    ax.add_collection(lines)

    # Projective
    lines = sim_proj.as_collections(
        i, lines_kwargs={"color": "b", "linewidth": 3, "alpha": 0.4}
    )[0]
    ax.add_collection(lines)

    # Extrapolated
    lines = sim_ext.as_collections(
        i, lines_kwargs={"color": "w", "linewidth": 1, "alpha": 0.4}
    )[0]
    ax.add_collection(lines)

    ax.text(5, 10, i, c="w", va="top", ha="left", fontsize=20)
fig.subplots_adjust(wspace=0.01, hspace=0.01)
```




As expected from the intermediate results above (similar average PC and NCC score), all PCs produce visually identical geometrical simulations. However, the orientations may be slightly different

[44]:
```
xmap_proj.orientations.angle_with(xmap_ext.orientations, degrees=True).mean()
```

[44]:
```
0.08375248119170249
```

The estimated orientations using PCs from plane fitting are on average misoriented by less than 1° from the estimated orientations using extrapolated PCs. The misorientation is most likely a result of the difference in applied sample tilts

[45]:
```
abs(det_proj_cal.sample_tilt - det_ext_cal.sample_tilt)
```

[45]:
```
0.01869762605252845
```

Since these are experimental data, it's difficult to say which sample tilt is more correct, although the nominal sample

tilt from the microscope was 70°.

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### PC calibration: "moving-screen" technique

The projection center (PC) describing the position of the EBSD detector relative to the beam-sample interaction volume can be estimated by the "moving-screen" technique [Hjelen *et al.*, 1991]. In this tutorial, we test this technique to get a rough estimate of the PC.

The technique assumes that the PC vector from the detector to the sample, shown in the *top figure in the reference frames tutorial*, is normal to the detector screen as well as the incoming electron beam. It will therefore intersect the screen at a position independent of the detector distance (DD). To find this position, we need two EBSD patterns acquired with a stationary beam but a known difference $\Delta z$ in DD, say 5 mm.

First, the goal is to find the pattern position that does not shift between the two camera positions, ($PC_x$, $PC_y$). This point can be estimated in fractions of screen width and height, respectively, by selecting the same pattern features in both patterns. The two points of each pattern feature can then be used to form a straight line, and two or more such lines should intersect at ($PC_x$, $PC_y$).

Second, the DD ($PC_z$) can be estimated from the same points. After finding the distances $L_{in}$ and $L_{out}$ between two points (features) in both patterns (in = operating position, out = 5 mm from operating position), the DD can be found from the relation

$$\mathrm{DD} = \frac{\Delta z}{L_{out}/L_{in} - 1},$$

where DD is given in the same unit as $\Delta z$. If also the detector pixel size $\delta$ is known (e.g. 46 mm / 508 px), $PC_z$ can be given in the fraction of the detector screen height

$$PC_z = \frac{\mathrm{DD}}{N_r \delta b},$$

where $N_r$ is the number of detector rows and $b$ is the binning factor.

Let's first import necessary libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

from diffsims.crystallography import ReciprocalLatticeVector
from orix.crystal_map import PhaseList
import kikuchipy as kp
```

We will find an estimate of the PC from two single crystal silicon patterns. These are included in the *kikuchipy.data* module

[2]:
```
s_in = kp.data.si_ebsd_moving_screen(0, allow_download=True)
s_in.remove_static_background()
s_in.remove_dynamic_background()
```

(continues on next page)

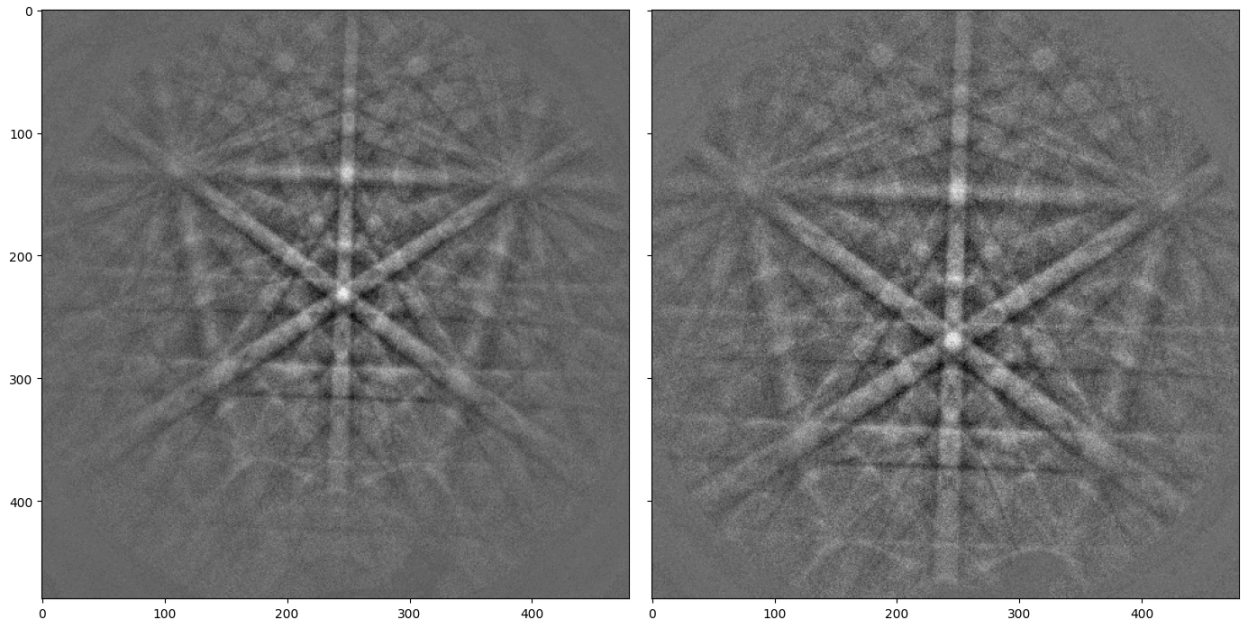
(continued from previous page)

```
s_out5mm = kp.data.si_ebsd_moving_screen(5, allow_download=True)
s_out5mm.remove_static_background()
s_out5mm.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 104.87 ms
[########################################] | 100% Completed | 101.54 ms
[########################################] | 100% Completed | 104.02 ms
[########################################] | 100% Completed | 102.72 ms
```

As a first approximation, we can find the detector pixel positions of the same features in both patterns manually with Matplotlib. Cursor pixel coordinates are displayed in the upper right part of the Matplotlib window when plotting with an interactive backend (e.g. qt5 or notebook).

[3]:
```
fig, axes = plt.subplots(ncols=2, sharex=True, sharey=True, figsize=(14, 7), layout=
↪"tight")
for ax, data in zip(axes, [s_in.data, s_out5mm.data]):
    ax.imshow(data, cmap="gray")
```




In this example, we choose the positions of three zone axes. The PC calibration is performed by creating an *PCCalibrationMovingScreen* instance

[4]:
```
cal = kp.detectors.PCCalibrationMovingScreen(
    pattern_in=s_in.data,
    pattern_out=s_out5mm.data,
    points_in=[(109, 131), (390, 139), (246, 232)],
    points_out=[(77, 146), (424, 156), (246, 269)],
    delta_z=5,
    px_size=None,  # Default
    convention="tsl",  # Default
)
cal
```

[4]:
```
PCCalibrationMovingScreen: (PCx, PCy, PCz) = (0.5123, 0.8606, 21.6518)
3 points:
[[[109 131]
  [390 139]
  [246 232]]

 [[ 77 146]
  [424 156]
  [246 269]]]
```

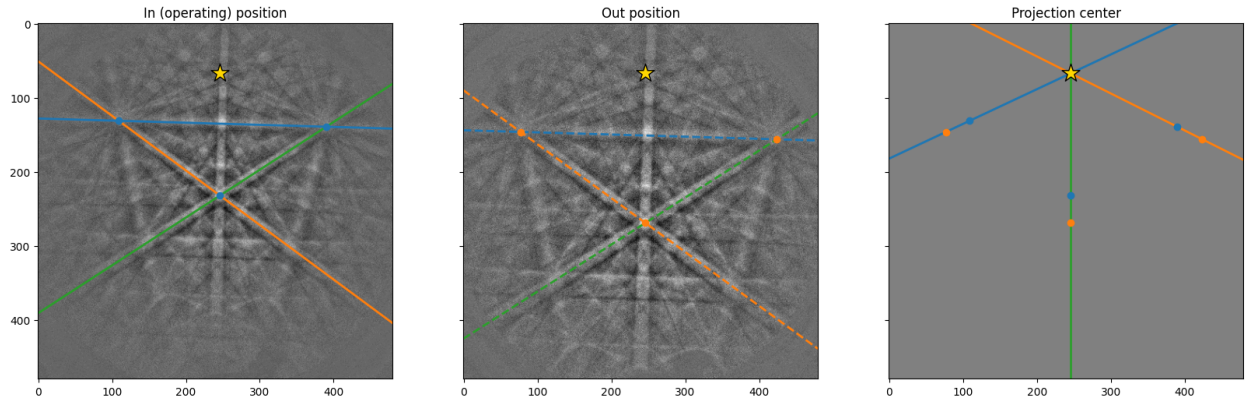
We see that ($PC_x$, $PC_y$) = (0.5123, 0.8606), while DD = 21.7 mm. To get $PC_z$ in fractions of detector height, we have to provide the detector pixel size $\delta$ upon initialization, or set it directly and recalculate the PC

[5]:
```
cal.px_size = 90e-3  # mm/px
cal
```

[5]:
```
PCCalibrationMovingScreen: (PCx, PCy, PCz) = (0.5123, 0.8606, 0.5012)
3 points:
[[[109 131]
  [390 139]
  [246 232]]

 [[ 77 146]
  [424 156]
  [246 269]]]
```

We can visualize the estimation by using the convenience method *PCCalibrationMovingScreen.plot()*

[6]:
```
cal.plot()
```



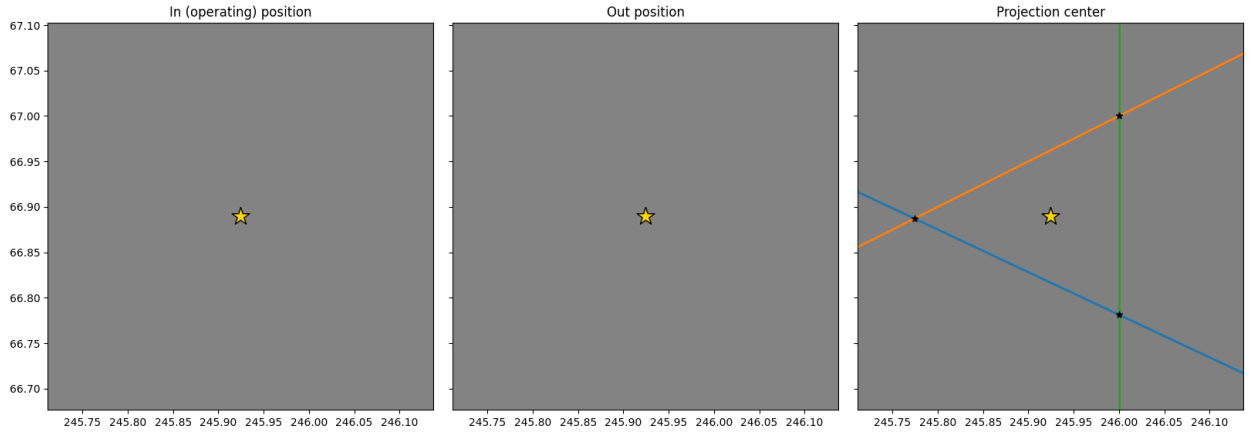

As expected, the three lines in the right figure meet at approimately the same point. We can replot the three images and zoom in on the PC to see how close they are to each other

[7]:
```
# PCy defined from top to bottom, otherwise "tsl", defined from bottom to top
cal.convention = "bruker"
pcx, pcy, _ = cal.pc

# Use two standard deviations of all $PC_x$ estimates as the axis limits
# (scaled with pattern shape)
two_std = 2 * np.std(cal.pcx_all, axis=0)
```

(continues on next page)

(continued from previous page)

```
fig = cal.plot(return_figure=True)
ax2 = fig.axes[2]
ax2.set_xlim([cal.ncols * (pcx - two_std), cal.ncols * (pcx + two_std)])
ax2.set_ylim([cal.nrows * (pcy - two_std), cal.nrows * (pcy + two_std)])
fig.subplots_adjust(wspace=0.05)
```




We can use this PC estimate as an initial guess when refining the PC using Hough indexing available from PyEBSDIndex. See the *Hough indexing tutorial* for details.

**Note**

PyEBSDIndex is an optional dependency of kikuchipy, and can be installed with both `pip` and `conda` (from `conda-forge`). To install PyEBSDIndex, see their installation instructions.

Create a detector with the correct shape and sample tilt, adding the PC estimate

[8]:
```
det = kp.detectors.EBSDDetector(cal.shape, sample_tilt=70, pc=cal.pc)
det
```

[8]:
```
EBSDDetector (480, 480), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.512, 0.139,␣
↪0.501)
```

Create an EBSDIndexer for use with PyEBSDIndex

[9]:
```
phase_list = PhaseList(names="si", space_groups=227)
print(phase_list)

indexer = det.get_indexer(phase_list)
```

```
Id  Name  Space group  Point group  Proper point group     Color
 0    si        Fd-3m         m-3m                 432  tab:blue
```

Optimize the PC via Hough indexing and plot the difference between the estimated and optimized PCs

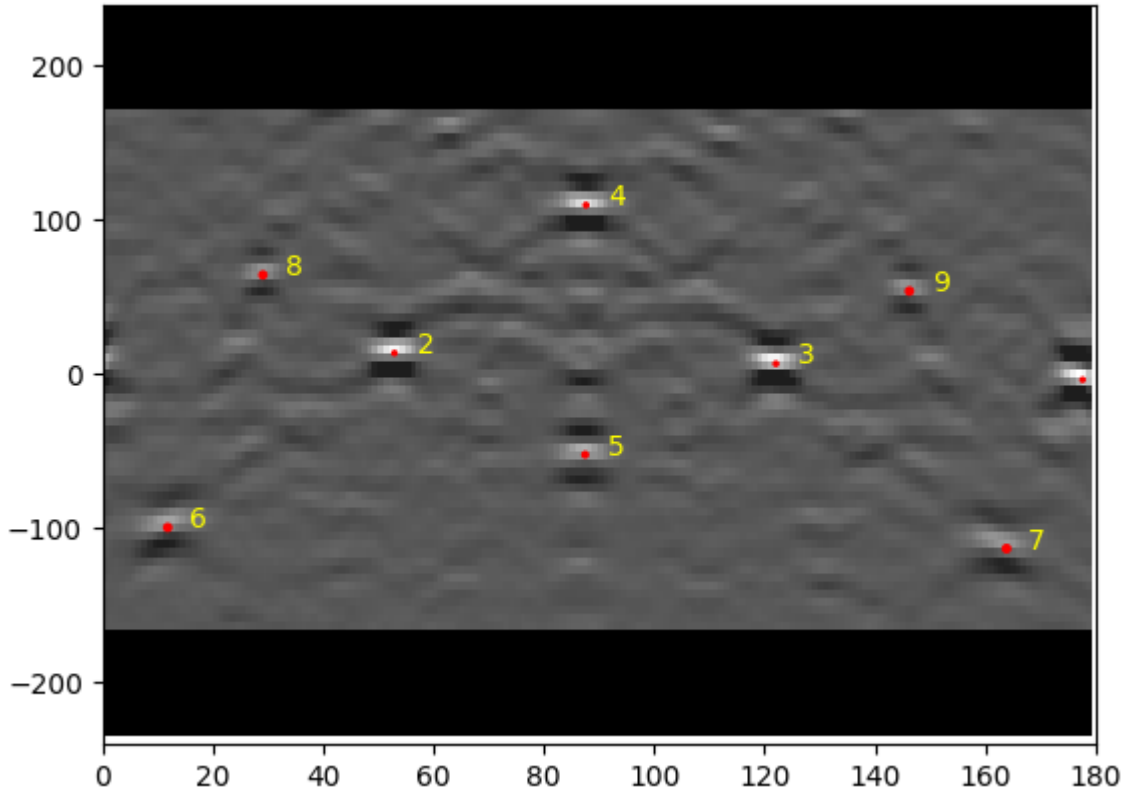
[10]:
```
det_ref = s_in.hough_indexing_optimize_pc(pc0=det.pc, indexer=indexer, method="PSO")

print(det.pc - det_ref.pc)
```

```
n_particles: 30 c1: 3.5 c2: 3.5 w: 0.8
Progress [**********] 50/50  global best:0.175  best loc:[0.5212 0.1589 0.4878]
Optimization finished | best cost: 0.17461253702640533, best pos: [0.52123837 0.15886974␣
↪0.48775927]

[[-0.00889518 -0.01951672  0.01343962]]
```

Index the pattern via Hough indexing

[11]:
```
indexer.PC = det_ref.pc

xmap = s_in.hough_indexing(phase_list, indexer=indexer, verbose=2)
```

```
Hough indexing with PyEBSDIndex information:
  PyOpenCL: False
  Projection center (Bruker): (0.5212, 0.1589, 0.4878)
  Indexing 1 pattern(s) in 1 chunk(s)
Radon Time: 0.01303924399871903
Convolution Time: 0.001581546999659622
Peak ID Time: 0.0008980009988590609
Band Label Time: 0.0001462939999282277
Total Band Find Time: 0.015692842000134988
Band Vote Time:  0.0012379230010992615
  Indexing speed: 31.41918 patterns/s
```




[12]:
```
print(xmap)
print(xmap.fit)
```

```
Phase  Orientations  Name  Space group  Point group  Proper point group     Color
    0    1 (100.0%)    si        Fd-3m         m-3m                 432  tab:blue
Properties: fit, cm, pq, nmatch
Scan unit: px
[0.17461254]
```

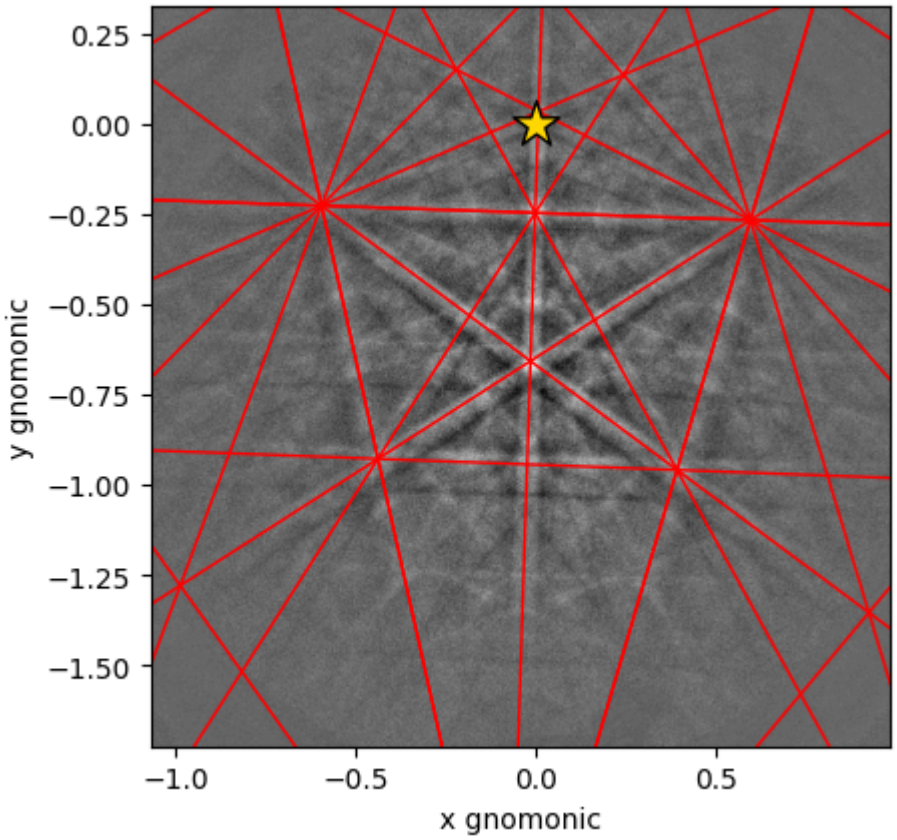
Create a simulator with the five $\{hkl\}$ families (reflectors) $\{111\}$, $\{200\}$, $\{220\}$, $\{222\}$, and $\{311\}$

[13]:
```
ref = ReciprocalLatticeVector(
    phase=xmap.phases[0],
    hkl=[[1, 1, 1], [2, 0, 0], [2, 2, 0], [2, 2, 2], [3, 1, 1]],
)
ref = ref.symmetrise()

simulator = kp.simulations.KikuchiPatternSimulator(ref)

sim = simulator.on_detector(det_ref, xmap.rotations)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.85 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 102.00 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.39 ms
```

Plot a *geometrical simulation* on top of the pattern using Kikuchi band centers

[14]:
```
sim.plot(
    coordinates="gnomonic",
    pattern=s_in.data,
    zone_axes_labels=False,
    zone_axes=False,
)
```




### 1.2.4 Simulations

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

#### Geometrical EBSD simulations

In this tutorial, we will inspect and visualize the results from EBSD indexing by plotting Kikuchi lines and zone axes onto an EBSD signal. We consider this a *geometrical* EBSD simulation, since it is only positions of Kikuchi lines and zone axes that are computed. These simulations are based on the work by Aimo Winkelmann in the supplementary material to [Britton *et al.*, 2016].

These simulations can be helpful when checking whether indexing results are correct and for interpreting them.

Let's import the necessary libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from diffpy.structure import Atom, Lattice, Structure
from diffsims.crystallography import ReciprocalLatticeVector
import hyperspy.api as hs
import kikuchipy as kp
from orix.crystal_map import Phase
from orix.quaternion import Rotation


# Plotting parameters
plt.rcParams.update(
    {"figure.figsize": (10, 10), "font.size": 20, "lines.markersize": 10}
)
```

We'll inspect the indexing results of a small nickel EBSD dataset of (3, 3) patterns

[2]:
```
# Use kp.load("data.h5") to load your own data
s = kp.data.nickel_ebsd_small()
s
```

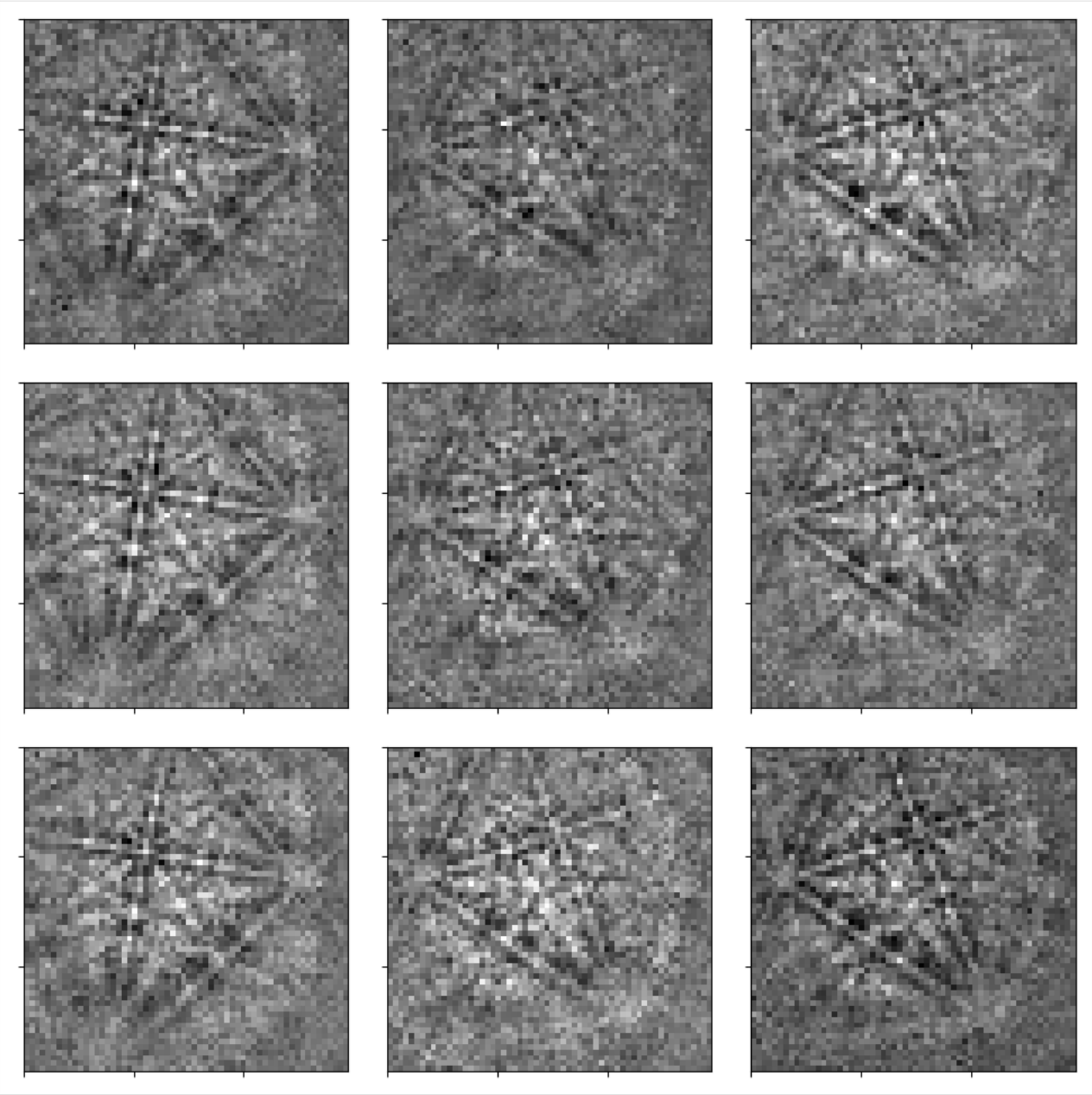
[2]:
```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
```

Let's enhance the Kikuchi bands by removing the static and dynamic backgrounds

[3]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 105.55 ms
[########################################] | 100% Completed | 101.53 ms
```

[4]:
```
_ = hs.plot.plot_images(
    s, axes_decor=None, label=None, colorbar=False, tight_layout=True
)
```



To project Kikuchi lines and zone axes onto our detector, we need

1. a description of the crystal phase
2. the set of Kikuchi bands to consider, e.g. the sets of planes {111}, {200}, {220}, and {311}
3. the crystal orientations with respect to the reference frame
4. the position of the detector with respect to the sample, in the form of a sample-detector model which includes the sample and detector tilt and the projection center (shortes distance from the source point on the sample to the detector), given here as ($PC_x$, $PC_y$, $PC_z$)

### Prepare phase and reflector list

We'll store the crystal phase information in an orix.crystal_map.Phase instance

[5]:
```
phase = Phase(
    space_group=225,
    structure=Structure(
        atoms=[Atom("Ni", [0, 0, 0])],
        lattice=Lattice(3.52, 3.52, 3.52, 90, 90, 90),
    ),
)

print(phase)
print(phase.structure)
```

```
<name: . space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:blue>
lattice=Lattice(a=3.52, b=3.52, c=3.52, alpha=90, beta=90, gamma=90)
Ni   0.000000 0.000000 0.000000 1.0000
```

We'll build up the reflector list using diffsims.crystallography.ReciprocalLatticeVector

[6]:
```
rlv = ReciprocalLatticeVector(
    phase=phase, hkl=[[1, 1, 1], [2, 0, 0], [2, 2, 0], [3, 1, 1]]
)
rlv
```

[6]:
```
ReciprocalLatticeVector (4,),  (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]
 [2. 2. 0.]
 [3. 1. 1.]]
```

We'll obtain the symmetrically equivalent vectors and plot each family of vectors in a distinct colour in the stereographic projection

[7]:
```
rlv = rlv.symmetrise().unique()
rlv.size
```

[7]:
```
50
```

[8]:
```
rlv.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 3 1 1    1.061     nan      nan       nan       24
 1 1 1    2.032     nan      nan       nan       8
 2 2 0    1.245     nan      nan       nan       12
 2 0 0    1.760     nan      nan       nan       6
```

[9]:
```
# Dictionary with {hkl} as key and indices into ReciprocalLatticeVector as values
hkl_sets = rlv.get_hkl_sets()
hkl_sets
```

[9]:
```
defaultdict(tuple,
            {(2.0, 0.0, 0.0): (array([ 8,  9, 10, 11, 12, 13]),),
             (2.0,
```

(continues on next page)

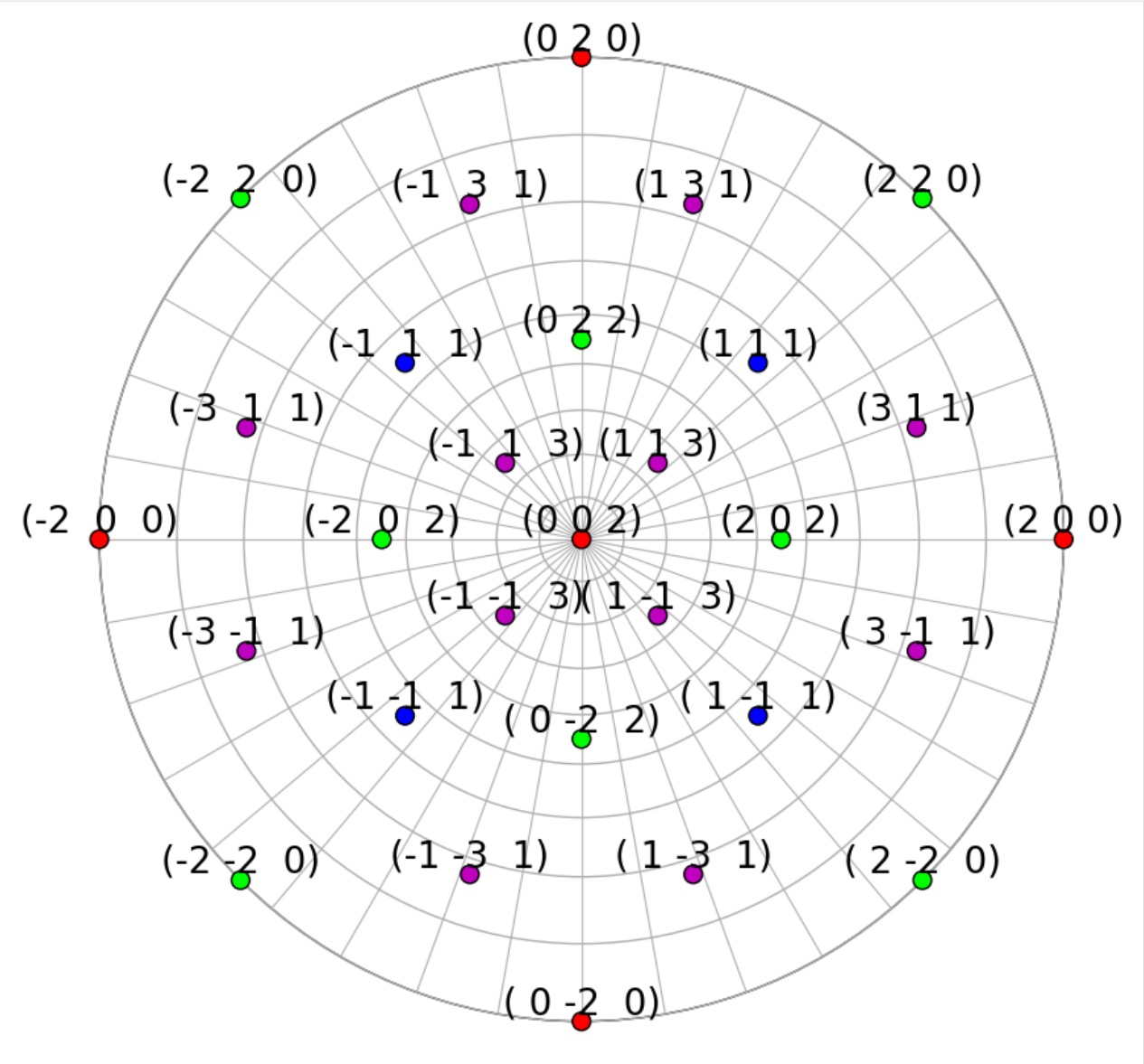
(continued from previous page)

```
             2.0,
             0.0): (array([14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]),),
            (1.0, 1.0, 1.0): (array([0, 1, 2, 3, 4, 5, 6, 7]),),
            (3.0,
             1.0,
             1.0): (array([26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,␣
↪41, 42,
                    43, 44, 45, 46, 47, 48, 49]),)})
```

[10]:
```
hkl_colors = np.zeros((rlv.size, 3))
for idx, color in zip(
    hkl_sets.values(),
    [
        [1, 0, 0],
        [0, 1, 0],
        [0, 0, 1],
        [0.75, 0, 0.75],
    ],  # Red, green, blue, magenta
):
    hkl_colors[idx] = color
```

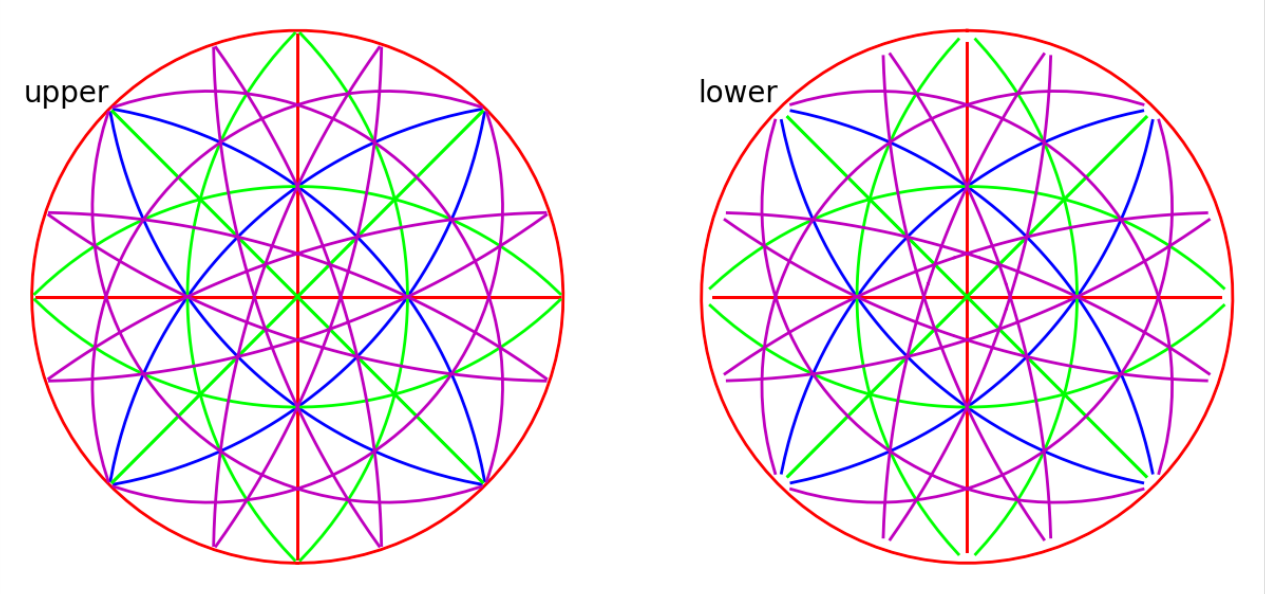
[11]:
```
hkl_labels = []
for hkl in rlv.hkl.round(0).astype(int):
    hkl_labels.append(str(hkl).replace("[", "(").replace("]", ")"))
```

[12]:
```
rlv.scatter(c=hkl_colors, grid=True, ec="k", vector_labels=hkl_labels)
```




We can also plot the plane traces, i.e. the Kikuchi lines, in both hemispheres (they are identical for Ni)

[13]:
```
rlv.draw_circle(
    color=hkl_colors, hemisphere="both", figure_kwargs=dict(figsize=(15, 10))
)
```




### Specify rotations and detector-sample geometry

Rotations and the detector-sample geometry for the nine nickel EBSD patterns are stored in the kikuchipy h5ebsd file. These were found by Hough indexing using *PyEBSDIndex* followed by orientation and PC refinement using a nickel EBSD master pattern simulated with *EMsoft*. See the *Hough indexing* and *Pattern matching* tutorials for details on indexing and the *reference frame* tutorial for details on the definition of the detector-sample geometry.

[14]:
```
rot = s.xmap.rotations
rot = rot.reshape(*s.xmap.shape)
rot  # Quaternions
```

[14]:
```
Rotation (3, 3)
[[[ 0.8743  0.0555  0.475   0.083 ]
  [ 0.4745  0.2915  0.4221 -0.7153]
  [ 0.4761  0.2915  0.4232 -0.7136]]

 [[ 0.2686 -0.1295  0.6879 -0.6618]
  [ 0.4755  0.2923  0.4244 -0.713 ]
  [ 0.4773  0.2925  0.4251 -0.7113]]

 [[ 0.5608 -0.2951  0.3731  0.6776]
  [ 0.7103 -0.4248  0.294   0.4781]
  [ 0.2999  0.0357  0.7124  0.6335]]]
```

We describe the sample-detector model in an *kikuchipy.detectors.EBSDDetector* instance. The sample was tilted 70° about the microscope X direction towards the detector, and the detector normal was orthogonal to the optical axis (beam direction). Using this information, the projection center (PC) was found using *PyEBSDIndex* as described in the Hough indexing tutorial.

[15]:
```
s.detector
```

[15]:
```
EBSDDetector (60, 60), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.425, 0.213, 0.
↪501)
```

[16]:
```
s.detector.pc
```

[16]:
```
array([[[0.4214844 , 0.21500351, 0.50201974],
        [0.42414583, 0.21014019, 0.50104439],
        [0.42637843, 0.21145593, 0.5004183 ]],

       [[0.42088203, 0.2165417 , 0.50079336],
        [0.42725023, 0.21450546, 0.49996293],
        [0.43082231, 0.21369458, 0.50123367]],

       [[0.42194418, 0.21202927, 0.4997446 ],
        [0.43085722, 0.21436106, 0.50068903],
        [0.42248503, 0.21257126, 0.50045621]]])
```

### Create simulations on detector

Now we're ready to create geometrical simulations. We create simulations using the *kikuchipy.simulations.KikuchiPatternSimulator*, which takes the reflectors as input

[17]:
```
simulator = kp.simulations.KikuchiPatternSimulator(rlv)
```

[18]:
```
sim = simulator.on_detector(s.detector, rot)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.25 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.95 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.38 ms
```

By passing the detector and crystal orientations to *KikuchiPatternSimulator.on_detector()*, we've obtained a *kikuchipy.simulations.GeometricalKikuchiPatternSimulation*, which stores the detector and gnomonic coordinates of the Kikuchi lines and zone axes for each crystal orientation

[19]:
```
sim
```

[19]:
```
GeometricalKikuchiPatternSimulation (3, 3):
ReciprocalLatticeVector (49,),  (m-3m)
[[ 1.  1.  1.]
 [-1.  1.  1.]
 [-1. -1.  1.]
 [ 1. -1.  1.]
 [ 1. -1. -1.]
 [ 1.  1. -1.]
 [-1.  1. -1.]
 [-1. -1. -1.]
 [ 2.  0.  0.]
 [ 0.  2.  0.]
 [-2.  0.  0.]
 [ 0. -2.  0.]
 [ 0.  0.  2.]
 [ 0.  0. -2.]
 [ 2.  2.  0.]
```

(continues on next page)

(continued from previous page)

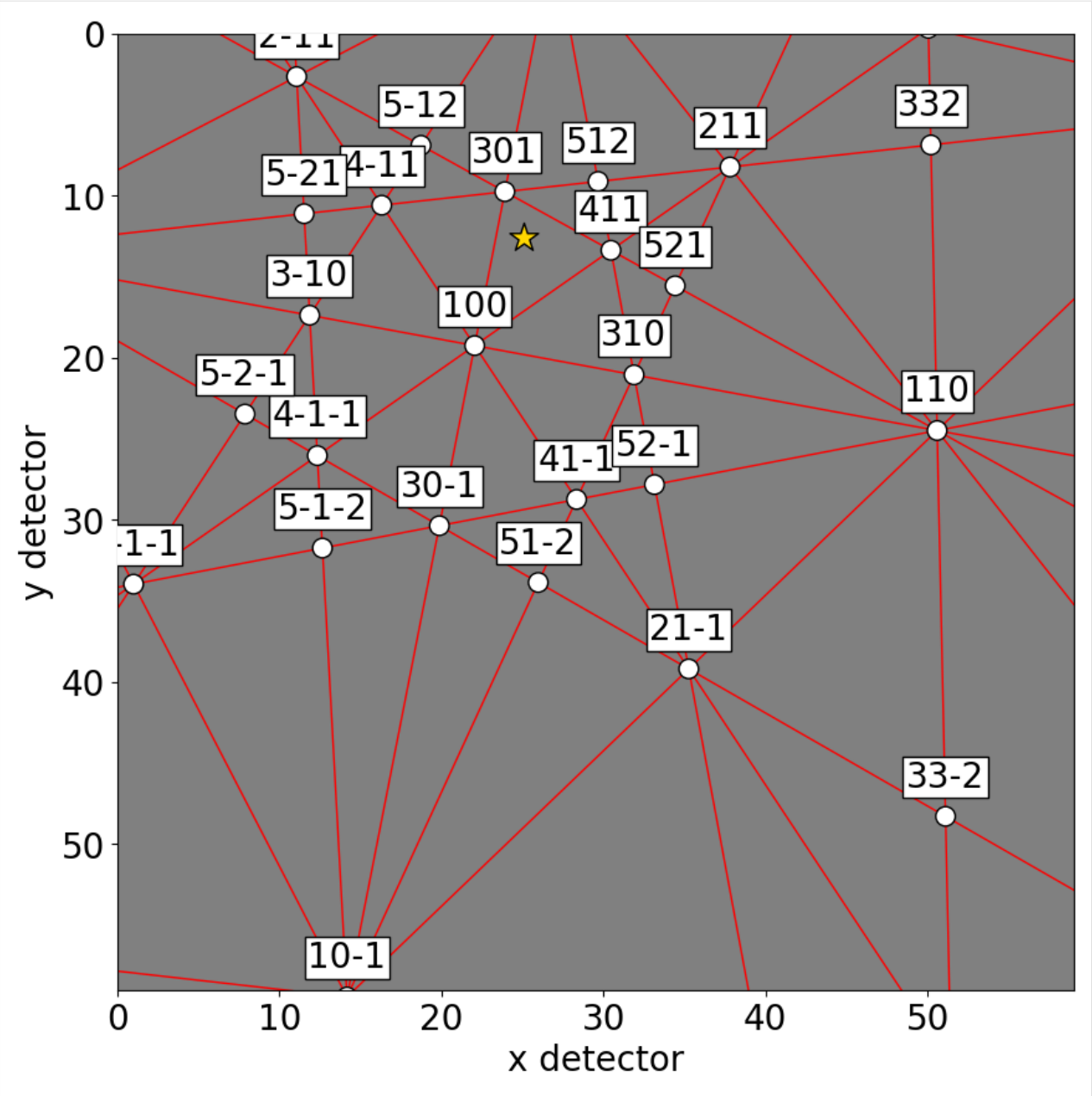
```
 [-2.  2.  0.]
 [-2. -2.  0.]
 [ 2. -2.  0.]
 [ 0.  2.  2.]
 [-2.  0.  2.]
 [ 0. -2.  2.]
 [ 2.  0.  2.]
 [ 0.  2. -2.]
 [ 0. -2. -2.]
 [ 2.  0. -2.]
 [ 3.  1.  1.]
 [-1.  3.  1.]
 [-3. -1.  1.]
 [ 1. -3.  1.]
 [ 1.  3.  1.]
 [-3.  1.  1.]
 [-1. -3.  1.]
 [ 3. -1.  1.]
 [ 3. -1. -1.]
 [ 1.  3. -1.]
 [-3.  1. -1.]
 [-1. -3. -1.]
 [-1.  3. -1.]
 [-3. -1. -1.]
 [ 1. -3. -1.]
 [ 3.  1. -1.]
 [ 1. -1.  3.]
 [ 1.  1.  3.]
 [-1.  1.  3.]
 [-1. -1.  3.]
 [-1. -1. -3.]
 [ 1. -1. -3.]
 [ 1.  1. -3.]
 [-1.  1. -3.]]
```

We see that not all 50 of the reflectors in the reflector list are present in some pattern.

### Plot simulations

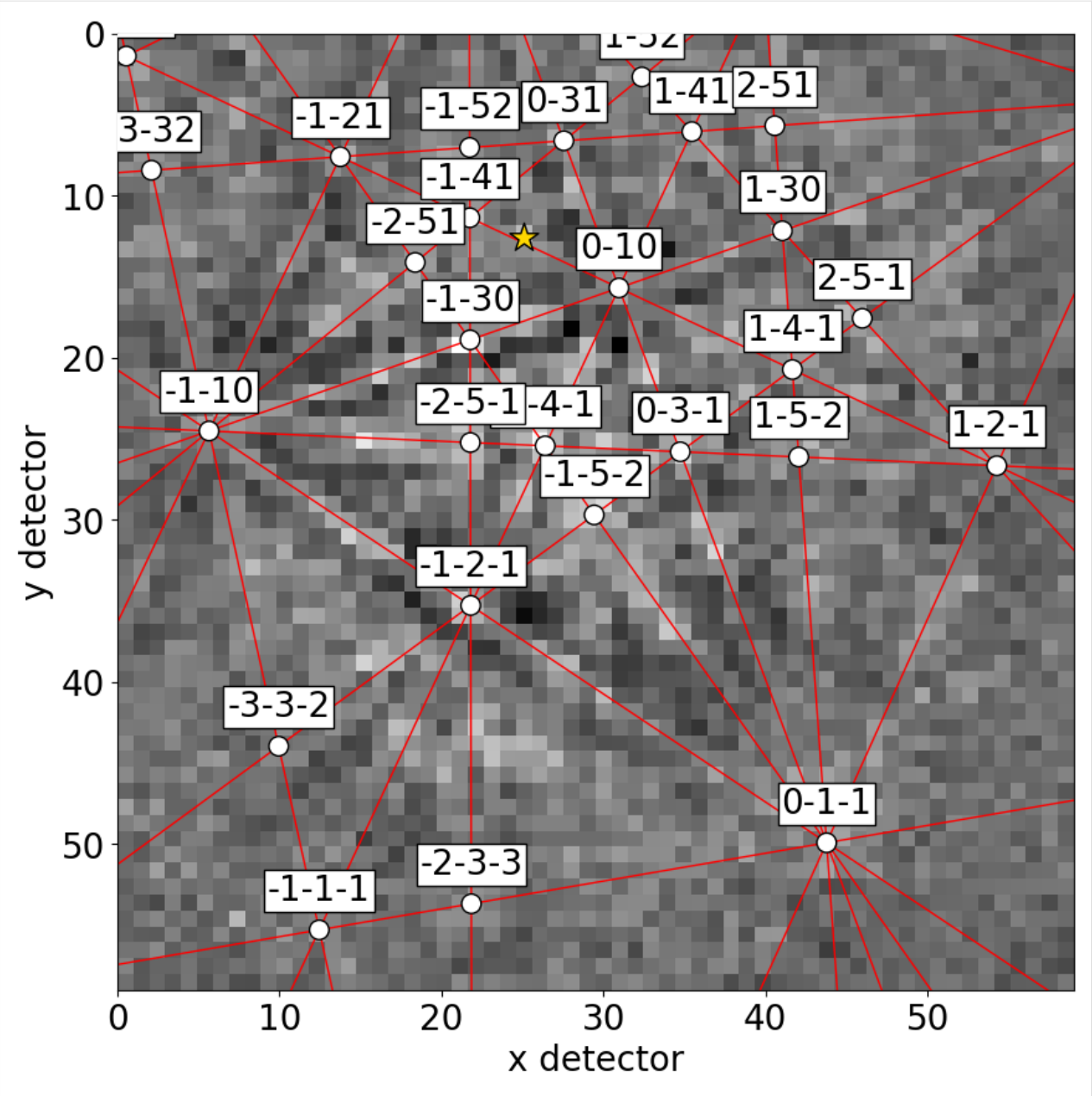
These geometrical simulations can be plotted one-by-one by themselves

[20]:
```
sim.plot()
```




Or, they can be plotted on top of patterns in three ways: passing a pattern to *GeometricalKikuchiPatternSimulation.plot()*

[21]:
```
sim.plot(index=(1, 2), pattern=s.inav[2, 1].data)
```



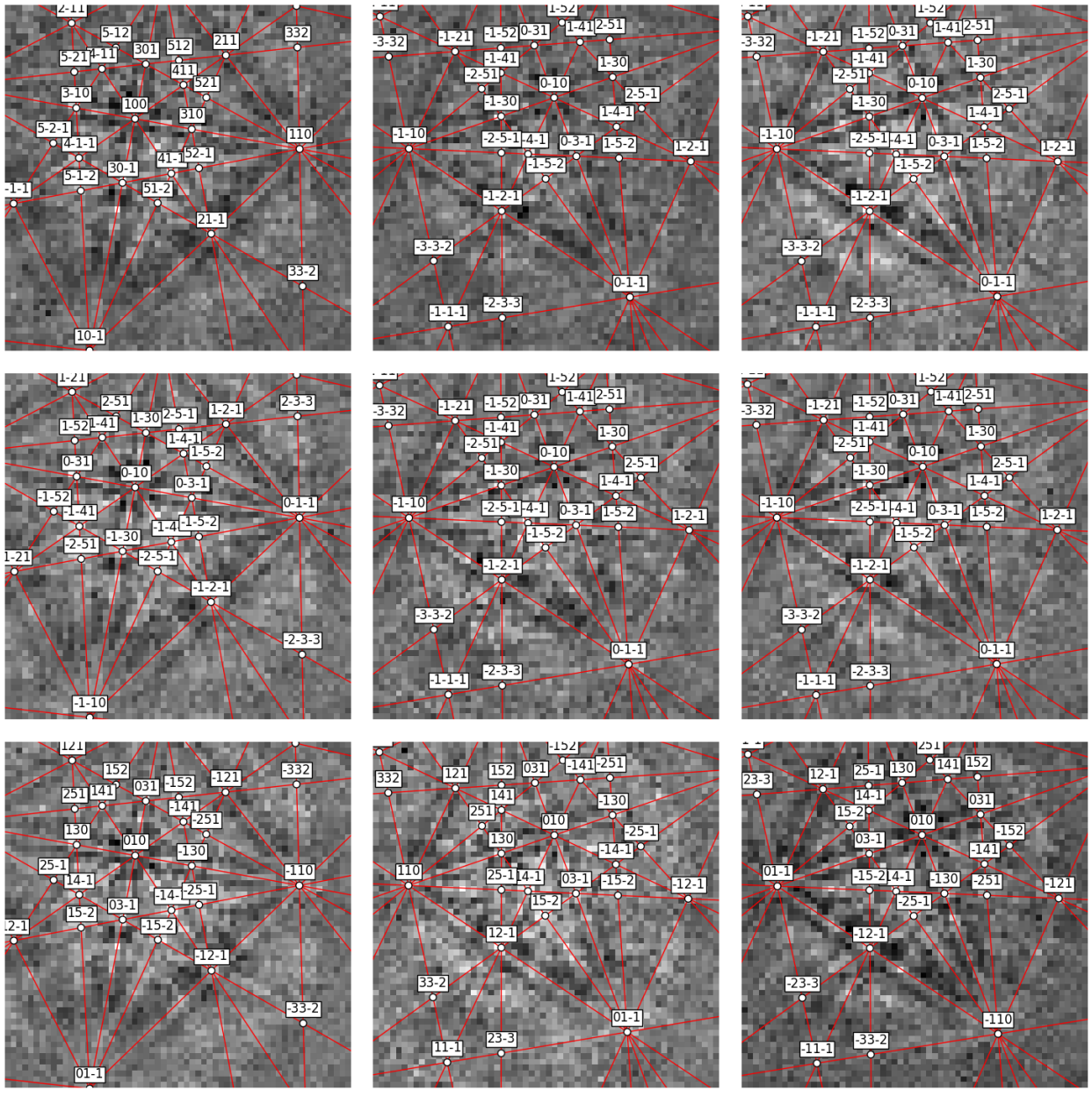

Or, we can obtain collections of lines, zone axes and zone axes labels as *Matplotlib* objects via *GeometricalKikuchiPatternSimulation.as_collections()* and add them to an existing Matplotlib axis

[22]:
```
fig, ax = plt.subplots(ncols=3, nrows=3, figsize=(15, 15))

for idx in np.ndindex(s.axes_manager.navigation_shape[::-1]):
    ax[idx].imshow(s.data[idx], cmap="gray")
    ax[idx].axis("off")

    lines, zone_axes, zone_axes_labels = sim.as_collections(
        idx,
        zone_axes=True,
```

(continues on next page)

(continued from previous page)

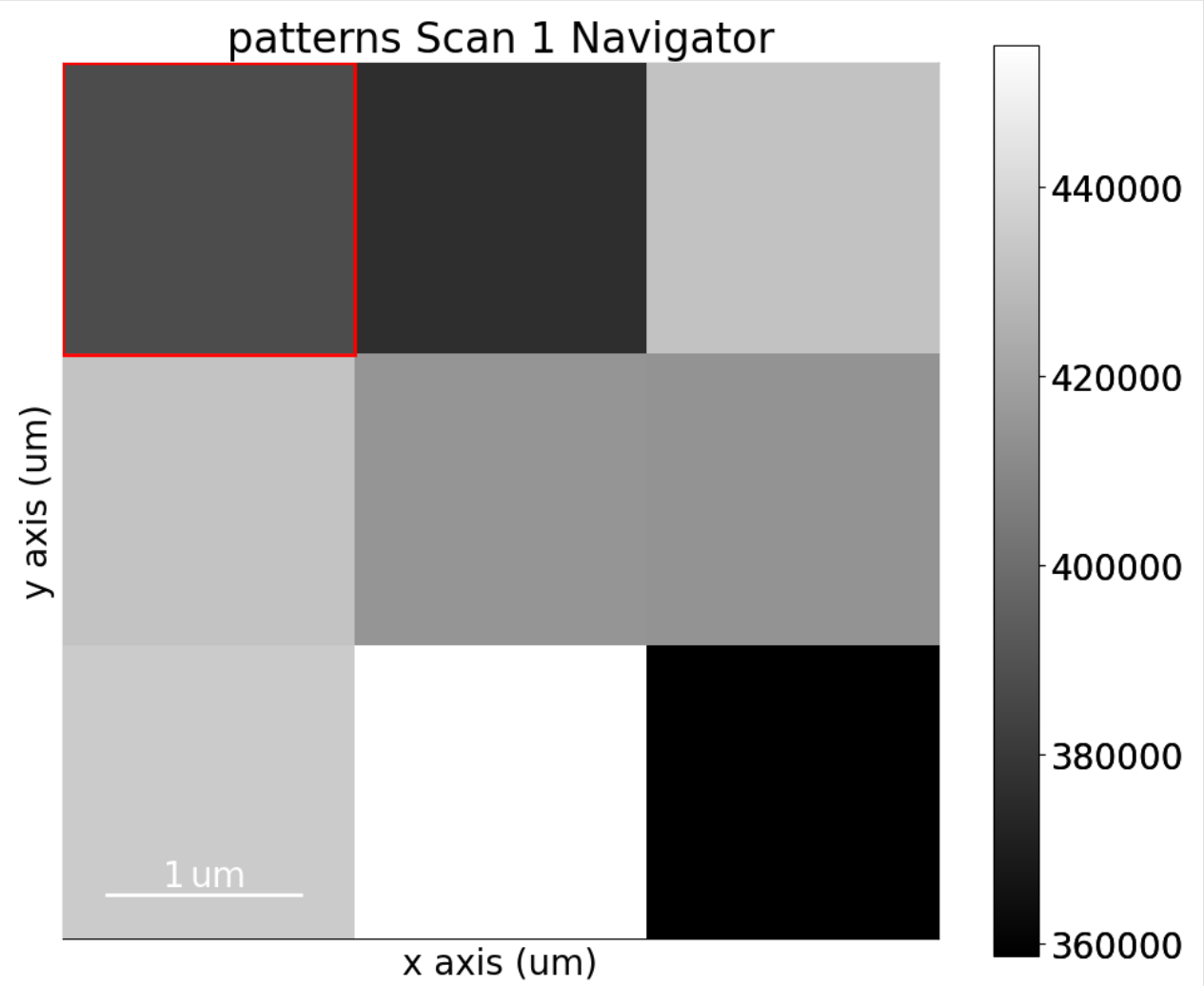
```
        zone_axes_labels=True,
        zone_axes_labels_kwargs=dict(fontsize=12),
    )
    ax[idx].add_collection(lines)
    ax[idx].add_collection(zone_axes)
    for label in zone_axes_labels:
        ax[idx].add_artist(label)

fig.tight_layout()
```

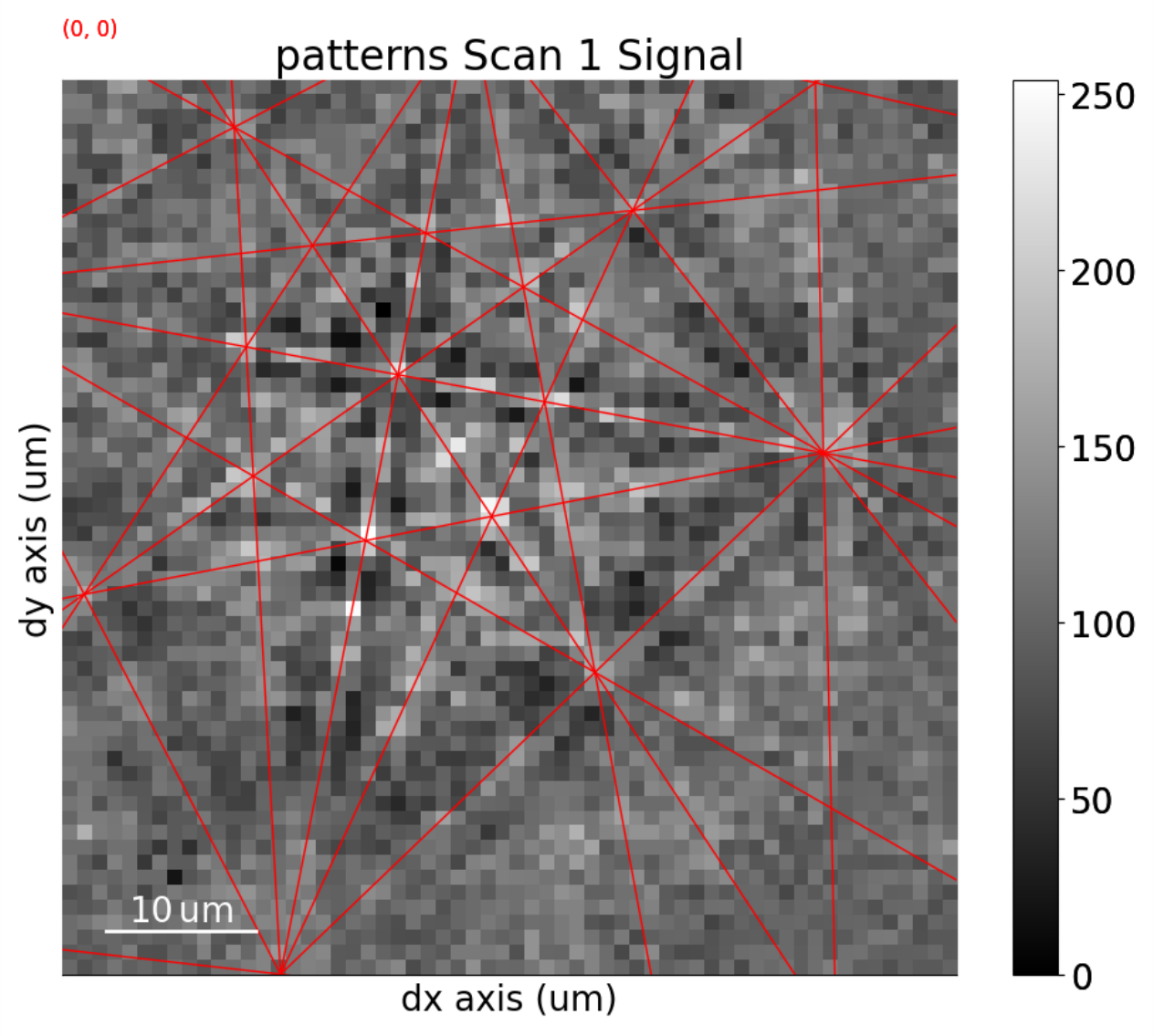


Or, we can obtain the lines, zone axes, zone axes labels and PCs as *HyperSpy* markers via *GeometricalKikuchiPatternSimulation.as_markers()* and add them to a signal of the same navigation shape as the simulation instance. This enables navigating the patterns *with* the geometrical simulations

[23]:
```
markers = sim.as_markers()

# To delete previously added permanent markers, do
# del s.metadata.Markers

s.add_marker(markers, plot_marker=False, permanent=True)
```

[24]:
```
s.plot()
```






**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

## Kinematical EBSD simulations

In this tutorial, we will perform kinematical Kikuchi pattern simulations of nickel, a variant of the $\sigma$-phase (Fe, Cr) in steels, and silicon carbide 6H.

We can generate kinematical master patterns using *KikuchiPatternSimulator.calculate_master_pattern()*. The simulator must be created from a ReciprocalLatticeVector instance that satisfy the following conditions:

1. All atom positions are filled in the unit cell, i.e. the `structure` used to create the `phase` used in `ReciprocalLatticeVector`. This can be achieved by creating a `Phase` instance with all asymmetric atom positions listed, creating a reflector list, and then calling ReciprocalLatticeVector.sanitise_phase(). The phase can be created manually or imported from a valid CIF file with Phase.from_cif().

2. The atoms in the `structure` have their elements described by the symbol (Ni), not by the atomic number (28).
3. The lattice parameters $(a, b, c)$ are given in Ångström.
4. Kinematical structure factors $F_{\mathrm{hkl}}$ have been calculated with ReciprocalLatticeVector.calculate_structure_factor().
5. Bragg angles $\theta_{\mathrm{B}}$ have been calculated with ReciprocalLatticeVector.calculate_theta().

Let's import the necessary libraries

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
import pyvista as pv

from diffpy.structure import Atom, Lattice, Structure
from diffsims.crystallography import ReciprocalLatticeVector
import hyperspy.api as hs
import kikuchipy as kp
from orix.crystal_map import Phase


# Plotting parameters
plt.rcParams.update(
    {"figure.figsize": (10, 10), "font.size": 20, "lines.markersize": 10}
)
# See https://docs.pyvista.org/user-guide/jupyter/index.html
pv.set_jupyter_backend("static")
```

### Nickel

We'll compare our kinematical simulations to dynamical simulations performed with EMsoft (see [Callahan and De Graef, 2013]), since we have a nickel master pattern available in the *kikuchipy.data* module

[2]:
```
mp_ni_dyn = kp.data.nickel_ebsd_master_pattern_small(
    projection="stereographic"
)
```

Inspect the phase

[3]:
```
phase_ni = mp_ni_dyn.phase.deepcopy()

print(phase_ni)
print(phase_ni.structure.lattice)
```

```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
Lattice(a=0.35236, b=0.35236, c=0.35236, alpha=90, beta=90, gamma=90)
```

Change lattice parameters from nm to Ångström

[4]:
```
lat_ni = phase_ni.structure.lattice  # Shallow copy
lat_ni.setLatPar(lat_ni.a * 10, lat_ni.b * 10, lat_ni.c * 10)

print(phase_ni.structure.lattice)
```

```
Lattice(a=3.5236, b=3.5236, c=3.5236, alpha=90, beta=90, gamma=90)
```

We'll build up the reflector list by:

1. Finding all reflectors with a minimal interplanar spacing $d$
2. Keeping those that have a structure factor above a certain $|F_{\text{min}}|$ of the reflector with the highest structure factor $|F_{\text{max}}|$

[5]:
```
ref_ni = ReciprocalLatticeVector.from_min_dspacing(phase_ni, 0.5)

# Exclude non-allowed reflectors (not available for hexagonal or trigonal
# phases!)
ref_ni = ref_ni[ref_ni.allowed]
ref_ni = ref_ni.unique(use_symmetry=True).symmetrise()
```

Sanitise the `phase` by completing the unit cell

[6]:
```
ref_ni.phase.structure
```

[6]:
```
[28   0.000000 0.000000 0.000000 1.0000]
```

[7]:
```
ref_ni.sanitise_phase()
ref_ni.phase.structure
```

[7]:
```
[Ni   0.000000 0.000000 0.000000 1.0000,
 Ni   0.000000 0.500000 0.500000 1.0000,
 Ni   0.500000 0.000000 0.500000 1.0000,
 Ni   0.500000 0.500000 0.000000 1.0000]
```

We can now calculate the structure factors $F$. Two parametrizations are available, from [Kirkland, 1998] (`"xtables"`, the default) and [Lobato and Van Dyck, 2014] (`"lobato"`)

[8]:
```
ref_ni.calculate_structure_factor()
```

[9]:
```
F_ni = abs(ref_ni.structure_factor)
ref_ni = ref_ni[F_ni > 0.05 * F_ni.max()]

ref_ni.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    2.034    11.8     140.0     100.0      8
 2 0 0    1.762    10.4     108.2      77.3      6
 2 2 0    1.246     7.4      55.0       39.3      12
 3 1 1    1.062     6.2      38.6       27.6      24
 2 2 2    1.017     5.9      34.7       24.8      8
 4 0 0    0.881     4.9      23.9       17.1      6
 3 3 1    0.808     4.3      18.8       13.4      24
 4 2 0    0.788     4.2      17.4       12.5      24
 4 2 2    0.719     3.6      13.3       9.5       24
 5 1 1    0.678     3.3      11.1       8.0       24
```

(continues on next page)

(continued from previous page)

```
3 3 3    0.678     3.3     11.1       8.0       8
4 4 0    0.623     2.9      8.6       6.1       12
5 3 1    0.596     2.7      7.4       5.3       48
6 0 0    0.587     2.7      7.1       5.1       6
4 4 2    0.587     2.7      7.1       5.1       24
6 2 0    0.557     2.4      6.0       4.3       24
5 3 3    0.537     2.3      5.3       3.8       24
6 2 2    0.531     2.3      5.1       3.7       24
4 4 4    0.509     2.1      4.4       3.2       8
```

Calculate the Bragg angle $\theta_{\mathrm{B}}$

[10]:
```
ref_ni.calculate_theta(20e3)
```

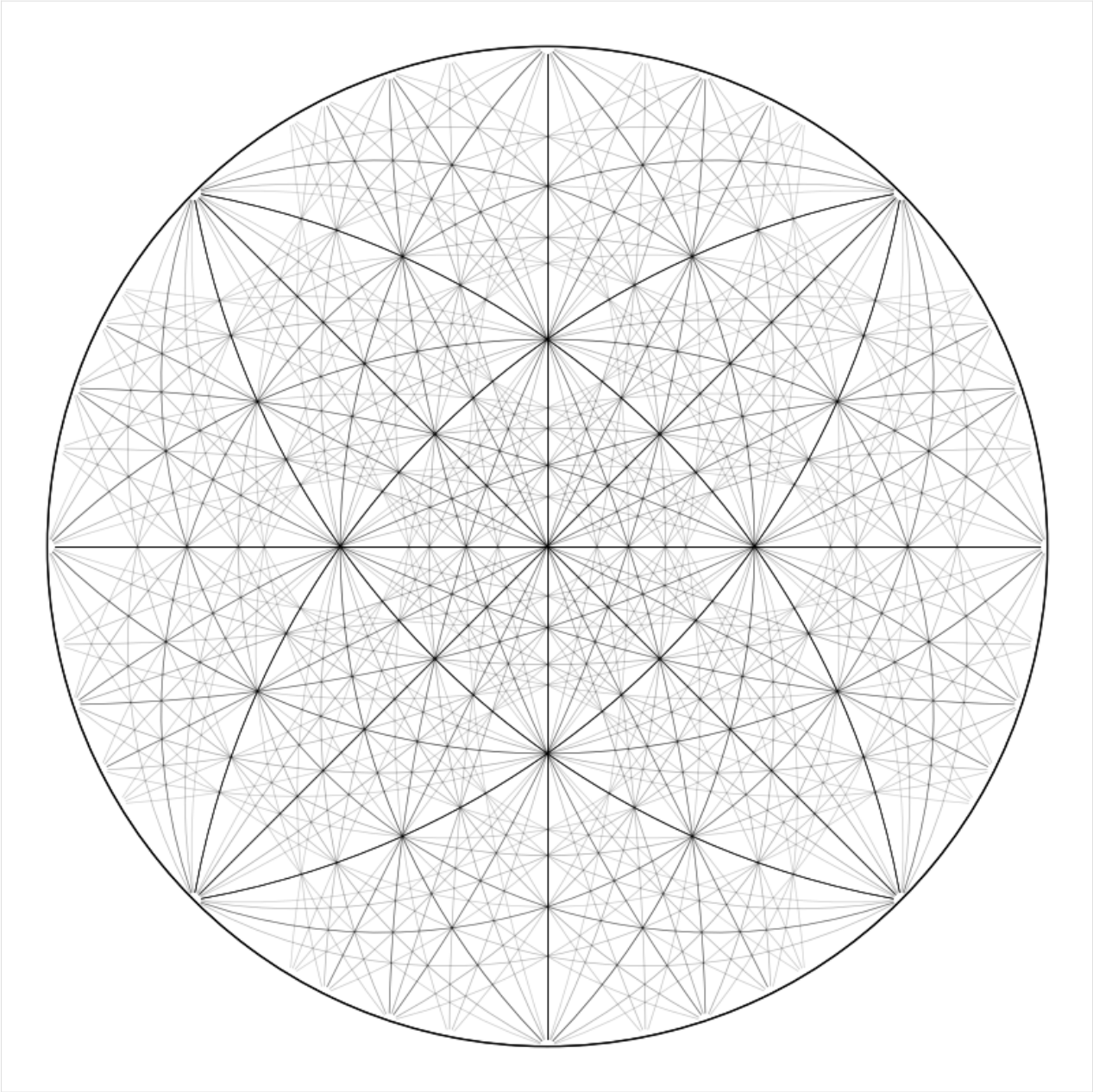
We can now create our simulator and plot the simulation

[11]:
```
simulator_ni = kp.simulations.KikuchiPatternSimulator(ref_ni)
simulator_ni.reflectors.size
```

[11]:
```
338
```

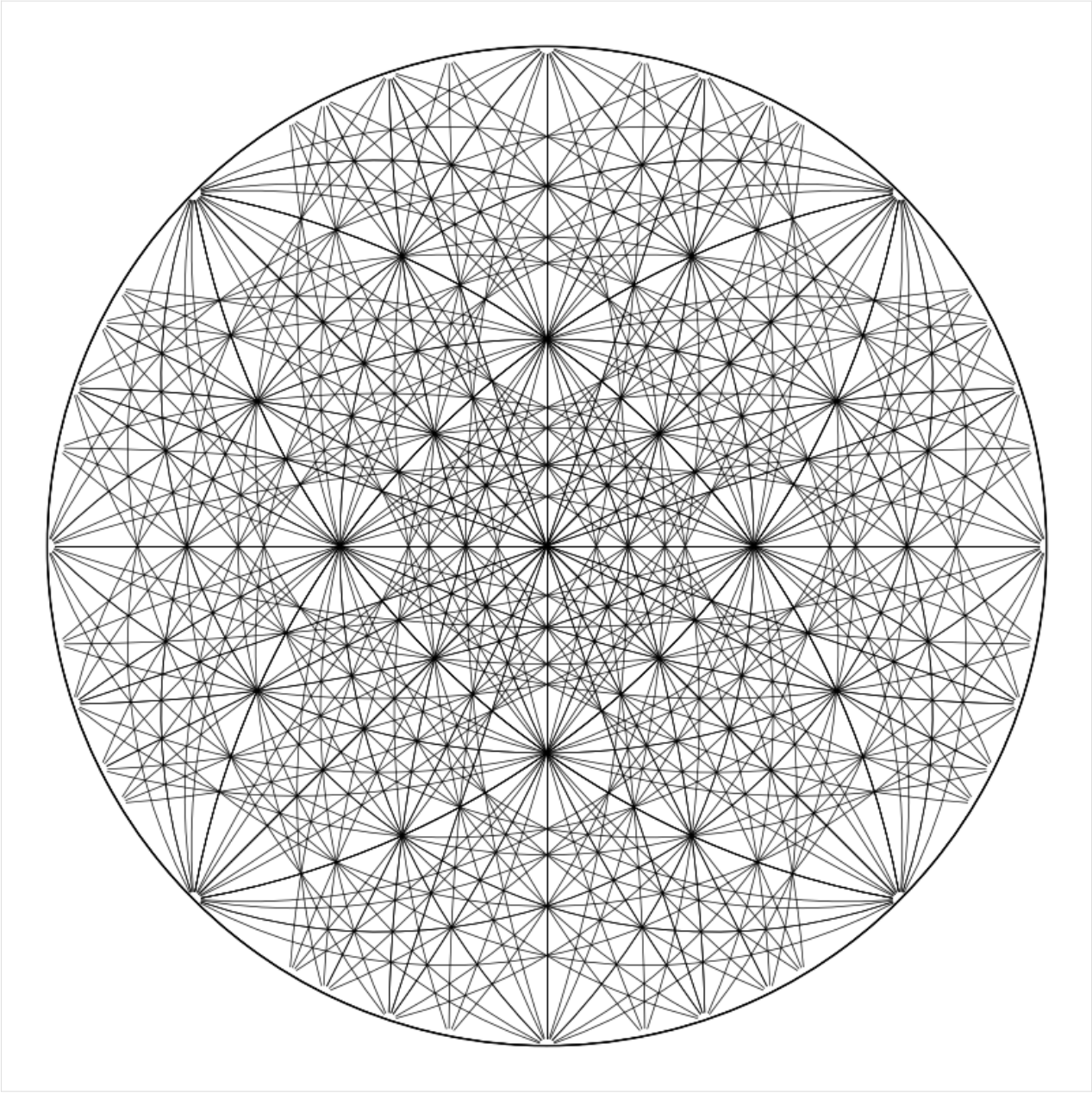
Plotting the band centers with intensities scaled by the structure factor $|F|$

[12]:
```
simulator_ni.plot()
```



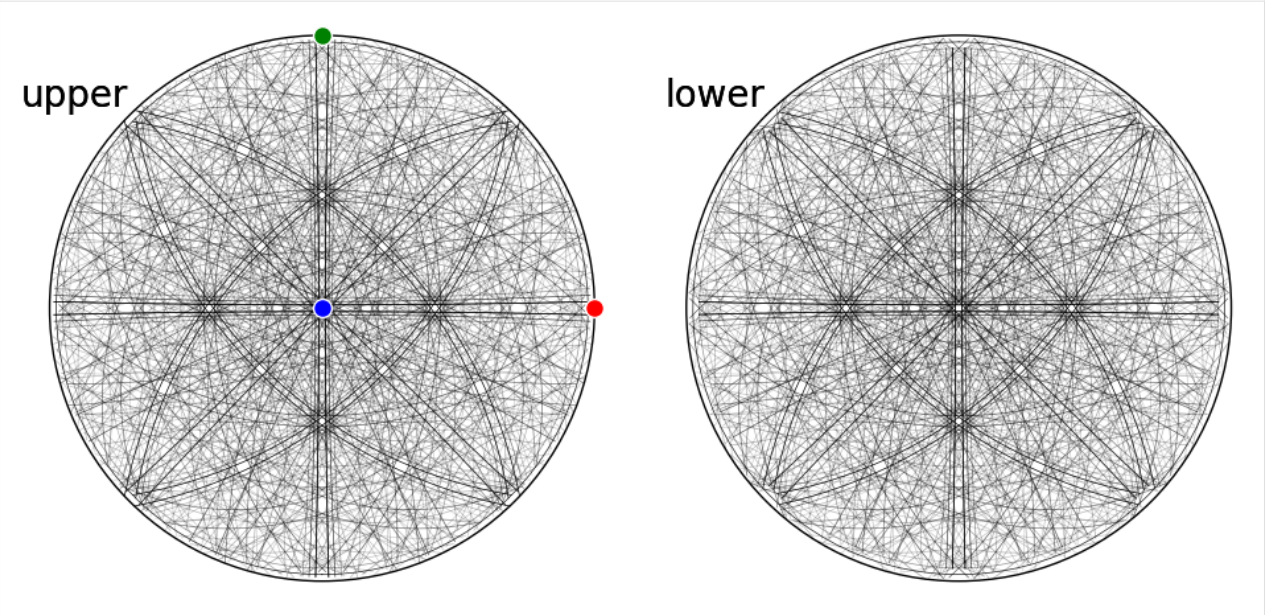
Or no scaling, $|F| = 1$ (`scaling="square"` for the structure factor squared $|F|^2$)

[13]:
```
simulator_ni.plot(scaling=None)
```



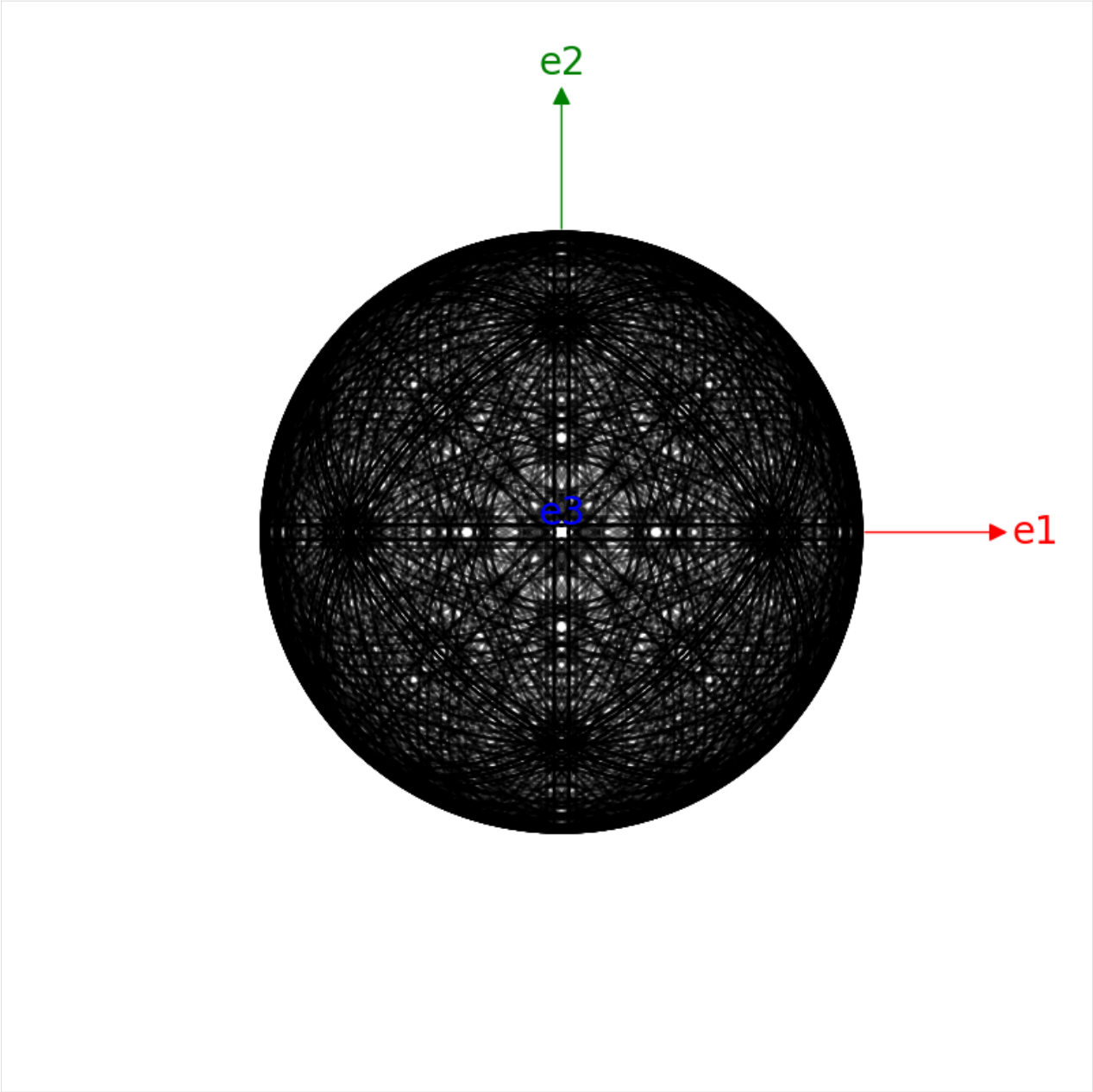
We can also plot the Kikuchi bands, showing both hemispheres, also adding the crystal axes alignment

[14]:
```
fig = simulator_ni.plot(hemisphere="both", mode="bands", return_figure=True)

ax = fig.axes[0]
ax.scatter(simulator_ni.phase.a_axis, c="r", ec="w")
ax.scatter(simulator_ni.phase.b_axis, c="g", ec="w")
ax.scatter(simulator_ni.phase.c_axis, c="b", ec="w")
fig.tight_layout()
```

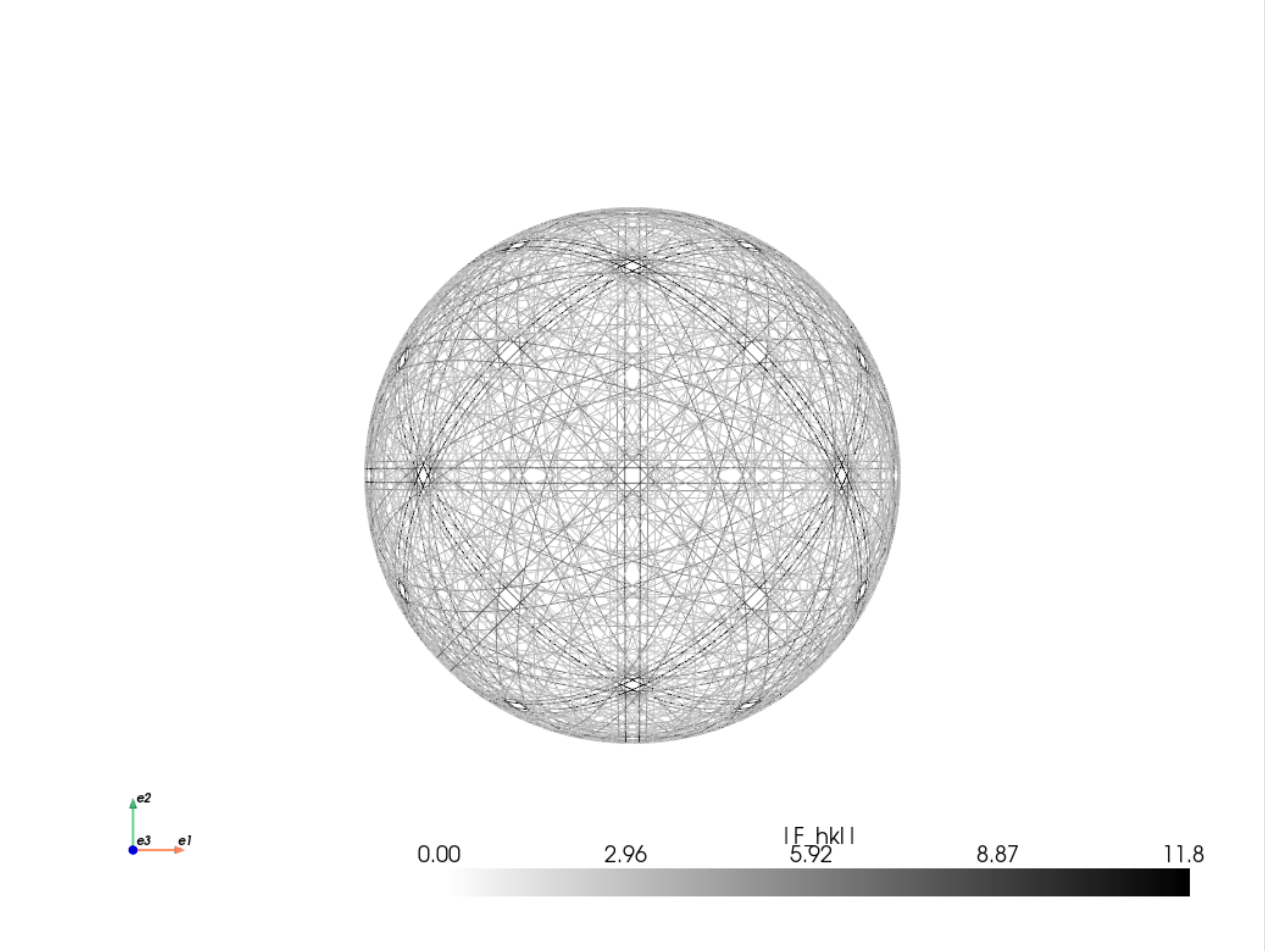



The simulation can be plotted in the spherical projection as well using *Matplotlib* or *PyVista*, provided that it is *installed*

[15]:
```
simulator_ni.plot("spherical", mode="bands")
```




[16]:
```
simulator_ni.plot("spherical", mode="bands", backend="pyvista")
```




When we're happy with the reflector list in the simulator, we can generate our kinematical master pattern

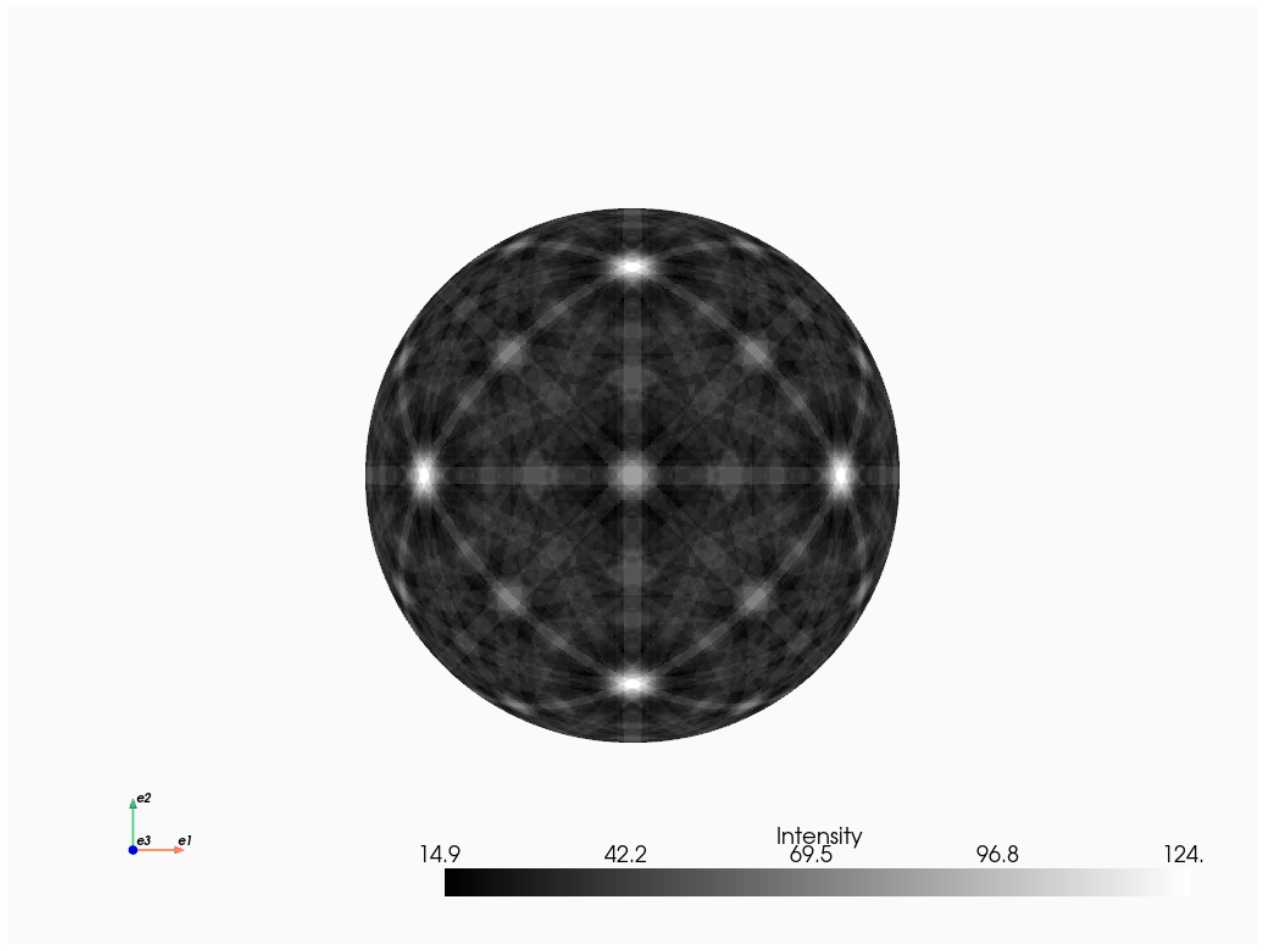
[17]:
```
mp_ni_kin = simulator_ni.calculate_master_pattern(half_size=200)
```
```
[########################################] | 100% Completed | 2.80 s
```

The returned master pattern is an instance of *EBSDMasterPattern* in the stereographic projection

[18]:
```
mp_ni_kin
```
[18]:
```
<EBSDMasterPattern, title: , dimensions: (|401, 401)>
```

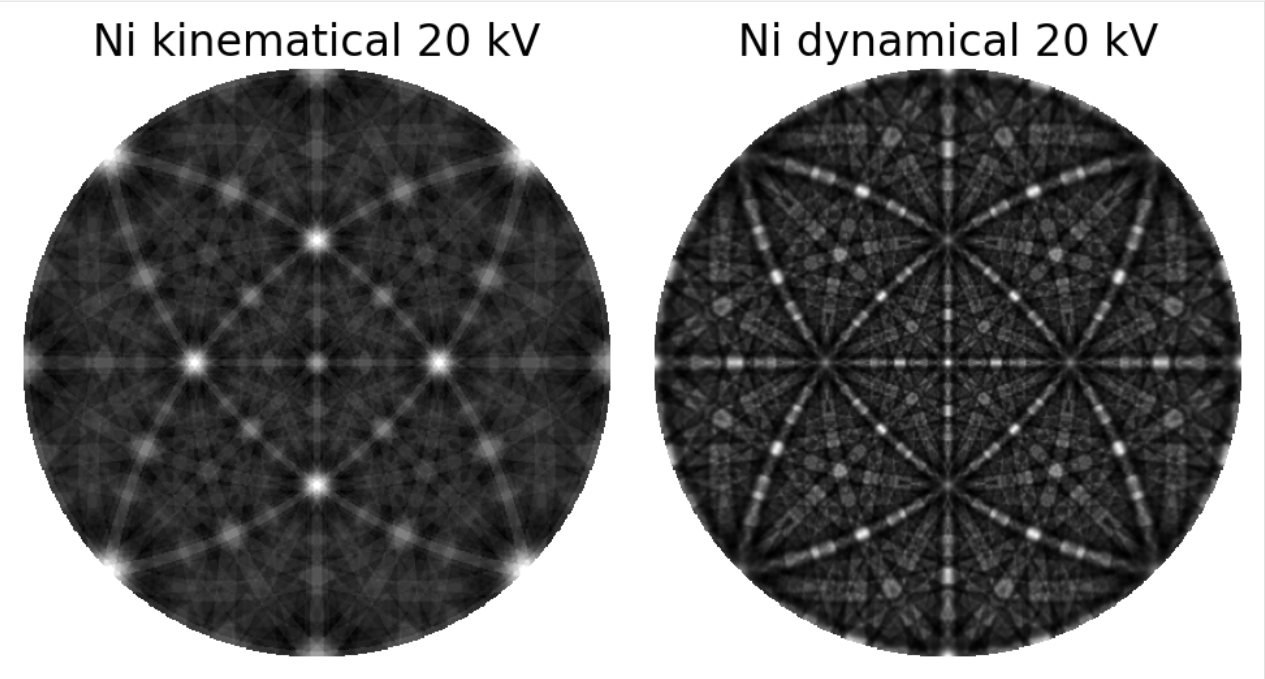
A spherical plot (requires that *PyVista* is installed)

[19]:
```
mp_ni_kin.plot_spherical(style="points")
```




Comparing kinematical and dynamical simulations

[20]:
```
# Exclude outside equator
ni_dyn_data = mp_ni_dyn.data.astype("float32")
ni_kin_data = mp_ni_kin.data.astype("float32")
mask = ni_dyn_data == 0
ni_dyn_data[mask] = np.nan
ni_kin_data[mask] = np.nan

fig, axes = plt.subplots(ncols=2, layout="tight")
for ax, data, title in zip(axes, [ni_kin_data, ni_dyn_data], ["kinematical", "dynamical
↪"]):
    ax.imshow(data, cmap="gray")
    ax.axis("off")
    ax.set(title=f"Ni {title} 20 kV")
```



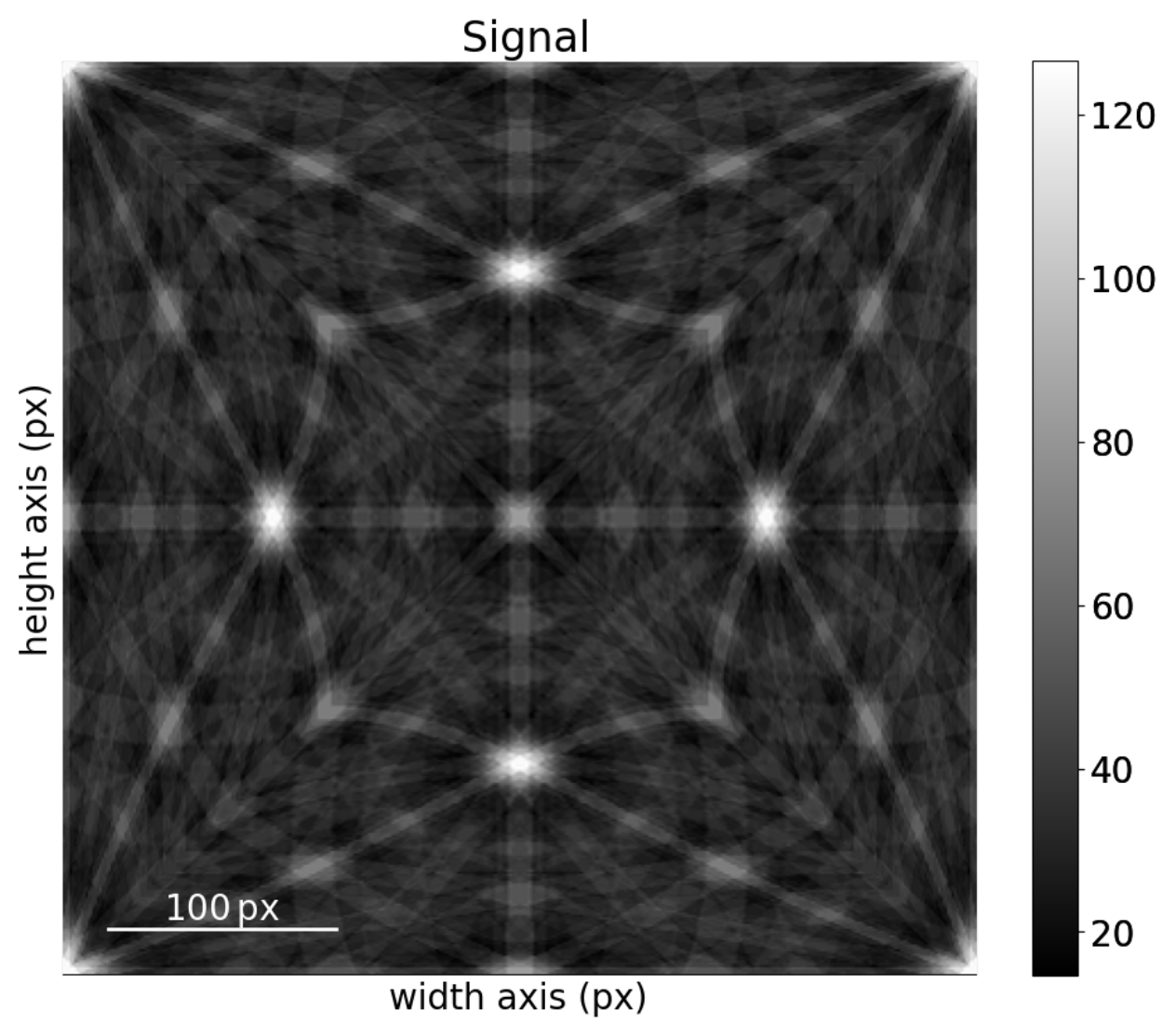

**Warning**

Use dynamical simulations when performing pattern matching, not kinematical simulations. The latter intensities are not realistic, as demonstrated in the above comparison.

Finally, we can transform the master pattern in the stereographic projection to one in the Lambert projection

[21]:
```
mp_ni_kin_lp = mp_ni_kin.as_lambert()
```
```
100%|----| 1/1 [00:00<00:00,  6.23it/s]
```

[22]:
```
mp_ni_kin_lp.plot()
```




We can then project parts of this pattern onto our EBSD detector using *get_patterns()*. Let's do this for the (3, 3) patterns used to demonstrate geometrical simulations in the *geometrical EBSD simulations tutorial*. These patterns are stored with the indexed solutions and an optimized detector-sample geometry (both found using *PyEBSDIndex*, see the *Hough indexing* for details)

[23]:
```
s = kp.data.nickel_ebsd_small(lazy=True)  # Don't load the patterns

Gr = s.xmap.rotations
Gr = Gr.reshape(*s.xmap.shape)
print(Gr)

print(s.detector)
```

```
Rotation (3, 3)
[[[ 0.8743  0.0555  0.475   0.083 ]
  [ 0.4745  0.2915  0.4221 -0.7153]
  [ 0.4761  0.2915  0.4232 -0.7136]]
```

(continues on next page)

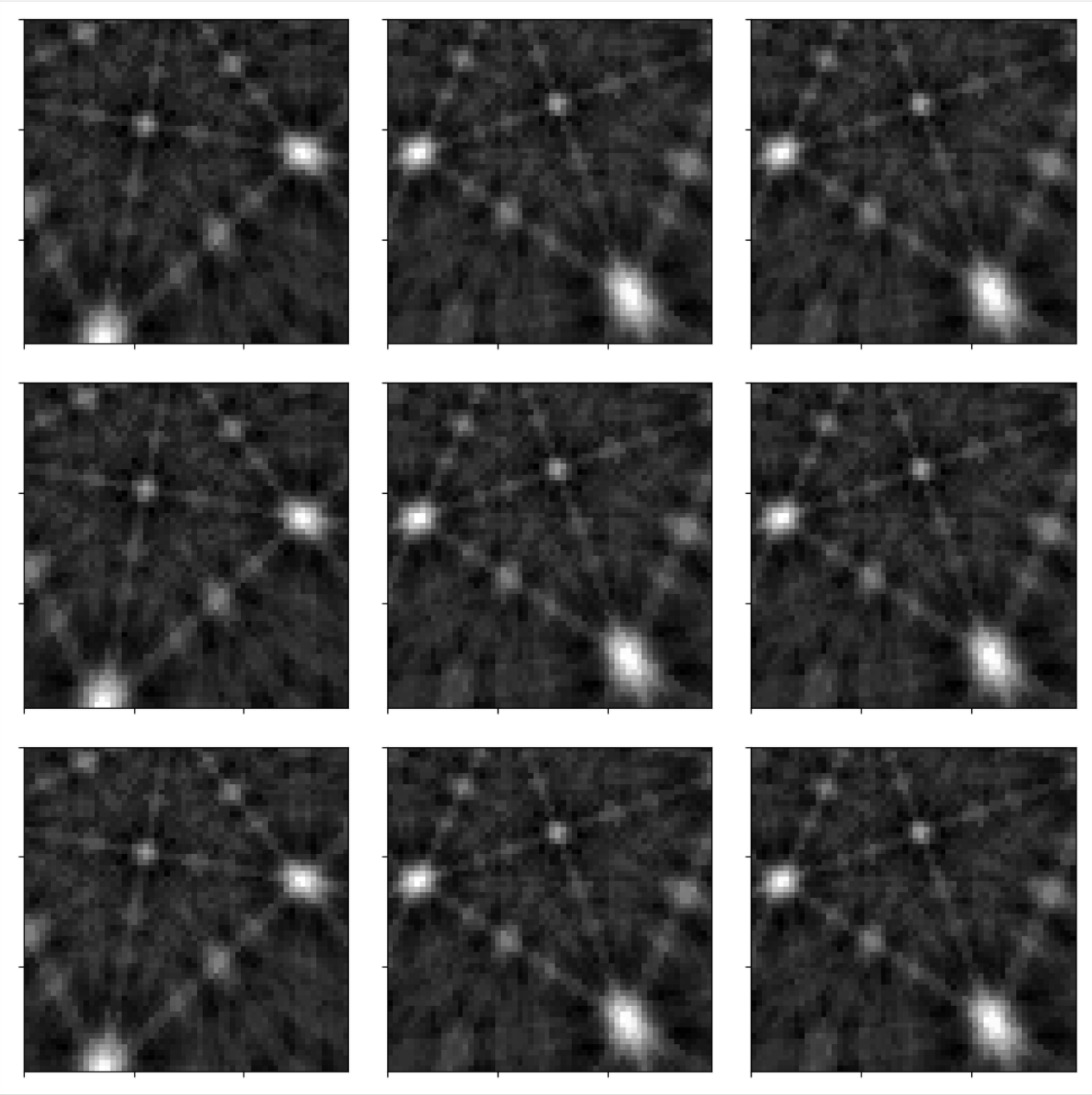
(continued from previous page)

```
 [[ 0.2686 -0.1295  0.6879 -0.6618]
  [ 0.4755  0.2923  0.4244 -0.713 ]
  [ 0.4773  0.2925  0.4251 -0.7113]]

 [[ 0.5608 -0.2951  0.3731  0.6776]
  [ 0.7103 -0.4248  0.294   0.4781]
  [ 0.2999  0.0357  0.7124  0.6335]]]
EBSDDetector (60, 60), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.425, 0.213, 0.
↪501)
```

[24]:
```
s_kin = mp_ni_kin_lp.get_patterns(Gr, s.detector, energy=20, compute=True)
```

```
[########################################] | 100% Completed | 102.32 ms
```

[25]:
```
_ = hs.plot.plot_images(
    s_kin, axes_decor=None, label=None, colorbar=False, tight_layout=True
)
```



Feel free to compare these patterns to the experimental patterns in the *geometrical EBSD simulations tutorial*!

### Sigma phase

[26]:
```
phase_sigma = Phase(
    name="sigma",
    space_group=136,
    structure=Structure(
        atoms=[
            Atom("Cr", [0, 0, 0], 0.5),
            Atom("Fe", [0, 0, 0], 0.5),
```

(continues on next page)

(continued from previous page)

```
            Atom("Cr", [0.31773, 0.31773, 0], 0.5),
            Atom("Fe", [0.31773, 0.31773, 0], 0.5),
            Atom("Cr", [0.06609, 0.26067, 0], 0.5),
            Atom("Fe", [0.06609, 0.26067, 0], 0.5),
            Atom("Cr", [0.13122, 0.53651, 0], 0.5),
            Atom("Fe", [0.13122, 0.53651, 0], 0.5),
        ],
        lattice=Lattice(8.802, 8.802, 4.548, 90, 90, 90),
    ),
)
phase_sigma
```

[26]:
```
<name: sigma. space group: P42/mnm. point group: 4/mmm. proper point group: 422. color:␣
↪tab:blue>
```

[27]:
```
ref_sigma = ReciprocalLatticeVector.from_min_dspacing(phase_sigma, 1)

ref_sigma.sanitise_phase()

ref_sigma.calculate_structure_factor("lobato")

F_sigma = abs(ref_sigma.structure_factor)
ref_sigma = ref_sigma[F_sigma > 0.05 * F_sigma.max()]

ref_sigma.calculate_theta(20e3)

ref_sigma.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 0 0 2    2.274    143.1   20470.3    100.0      2
 0 0 4    1.137    70.5    4970.7      24.3      2
 3 3 0    2.075    66.2    4381.5      21.4      4
 4 1 0    2.135    64.6    4170.3      20.4      8
 4 1 1    1.932    63.5    4027.8      19.7      16
 3 3 1    1.888    56.4    3185.6      15.6      8
 3 3 2    1.533    49.6    2461.7      12.0      8
 1 1 1    3.672    48.6    2361.2      11.5      8
 4 1 2    1.556    48.0    2299.3      11.2      16
 5 5 1    1.201    47.3    2232.9      10.9      8
 8 2 0    1.067    42.1    1775.5      8.7       8
 5 0 1    1.642    41.1    1688.5      8.2       8
 4 1 3    1.236    40.0    1599.1      7.8       16
 1 1 0    6.224    37.3    1391.3      6.8       4
 7 2 0    1.209    36.6    1337.2      6.5       8
 2 2 1    2.568    36.0    1297.3      6.3       8
 3 3 3    1.224    35.9    1290.3      6.3       8
 7 2 1    1.168    32.4    1048.0      5.1       16
 7 2 2    1.068    31.3     977.7      4.8       16
 6 6 0    1.037    31.3     977.4      4.8       4
 5 5 0    1.245    28.6     816.9      4.0       4
 4 1 4    1.004    28.5     814.4      4.0       16
 5 0 3    1.149    27.7     764.7      3.7       8
 3 2 0    2.441    27.4     748.8      3.7       8
```

(continues on next page)

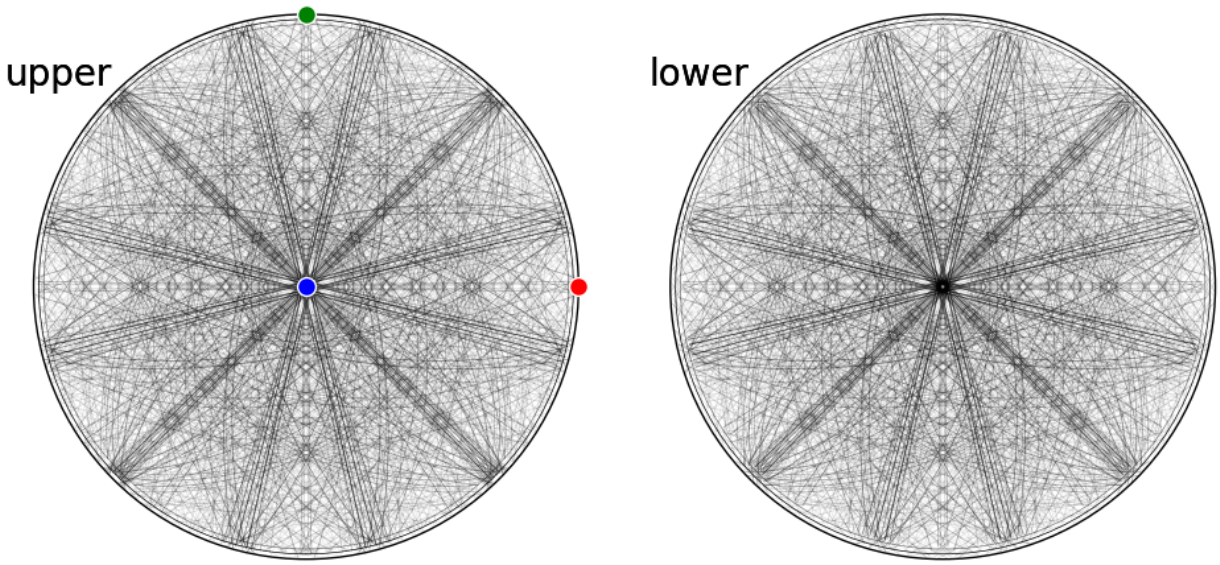
(continued from previous page)

```
1 0 1    4.041    25.4     642.9      3.1       8
5 5 2    1.092    24.3     590.9      2.9       8
3 2 1    2.151    23.7     561.8      2.7       16
6 4 1    1.179    22.9     522.5      2.6       16
6 6 1    1.011    22.8     520.7      2.5       8
1 1 3    1.473    22.5     504.7      2.5       8
4 0 0    2.201    21.8     476.3      2.3       4
4 4 1    1.472    21.8     475.6      2.3       8
7 3 1    1.120    21.0     439.9      2.1       16
2 2 3    1.363    19.9     394.7      1.9       8
3 2 2    1.664    19.4     375.4      1.8       16
8 2 1    1.039    19.3     374.0      1.8       16
1 1 2    2.136    19.1     364.2      1.8       8
4 3 1    1.642    18.3     336.3      1.6       16
2 2 0    3.112    18.1     327.6      1.6       4
5 1 0    1.726    17.1     292.8      1.4       8
2 1 1    2.976    16.3     265.8      1.3       16
4 0 2    1.581    16.0     257.3      1.3       8
4 4 0    1.556    15.7     245.6      1.2       4
3 1 0    2.783    15.5     241.3      1.2       8
5 2 1    1.538    15.4     236.3      1.2       16
4 4 3    1.086    15.3     233.2      1.1       8
6 1 0    1.447    15.0     223.5      1.1       8
7 0 1    1.212    14.5     211.4      1.0       8
3 2 3    1.288    14.2     202.8      1.0       16
2 0 0    4.401    14.2     202.6      1.0       4
5 1 2    1.375    13.5     183.3      0.9       16
8 3 0    1.030    12.8     164.7      0.8       8
4 4 2    1.284    12.7     161.9      0.8       8
4 3 3    1.149    12.3     152.3      0.7       16
6 1 2    1.221    12.3     152.2      0.7       16
6 1 1    1.379    11.7     136.8      0.7       16
3 0 1    2.465    11.7     136.0      0.7       8
2 2 2    1.836    11.7     135.9      0.7       8
1 0 3    1.494    11.2     126.0      0.6       8
3 2 4    1.031    11.2     125.0      0.6       16
5 2 3    1.112    10.6     112.3      0.5       16
3 1 2    1.761    10.5     109.4      0.5       16
1 1 4    1.118     9.7      94.6      0.5       8
8 3 1    1.005     9.6      91.2      0.4       16
4 0 4    1.010     9.5      89.8      0.4       8
6 2 1    1.331     9.2      84.6      0.4       16
4 2 0    1.968     9.0      81.0      0.4       8
4 3 0    1.760     8.7      75.7      0.4       8
3 1 1    2.374     8.7      75.0      0.4       16
6 1 3    1.047     8.4      70.2      0.3       16
2 1 3    1.415     8.4      69.8      0.3       16
2 0 2    2.020     8.0      64.7      0.3       8
8 1 1    1.062     7.6      57.7      0.3       16
6 5 0    1.127     7.5      56.6      0.3       8
6 3 1    1.261     7.4      55.4      0.3       16
```

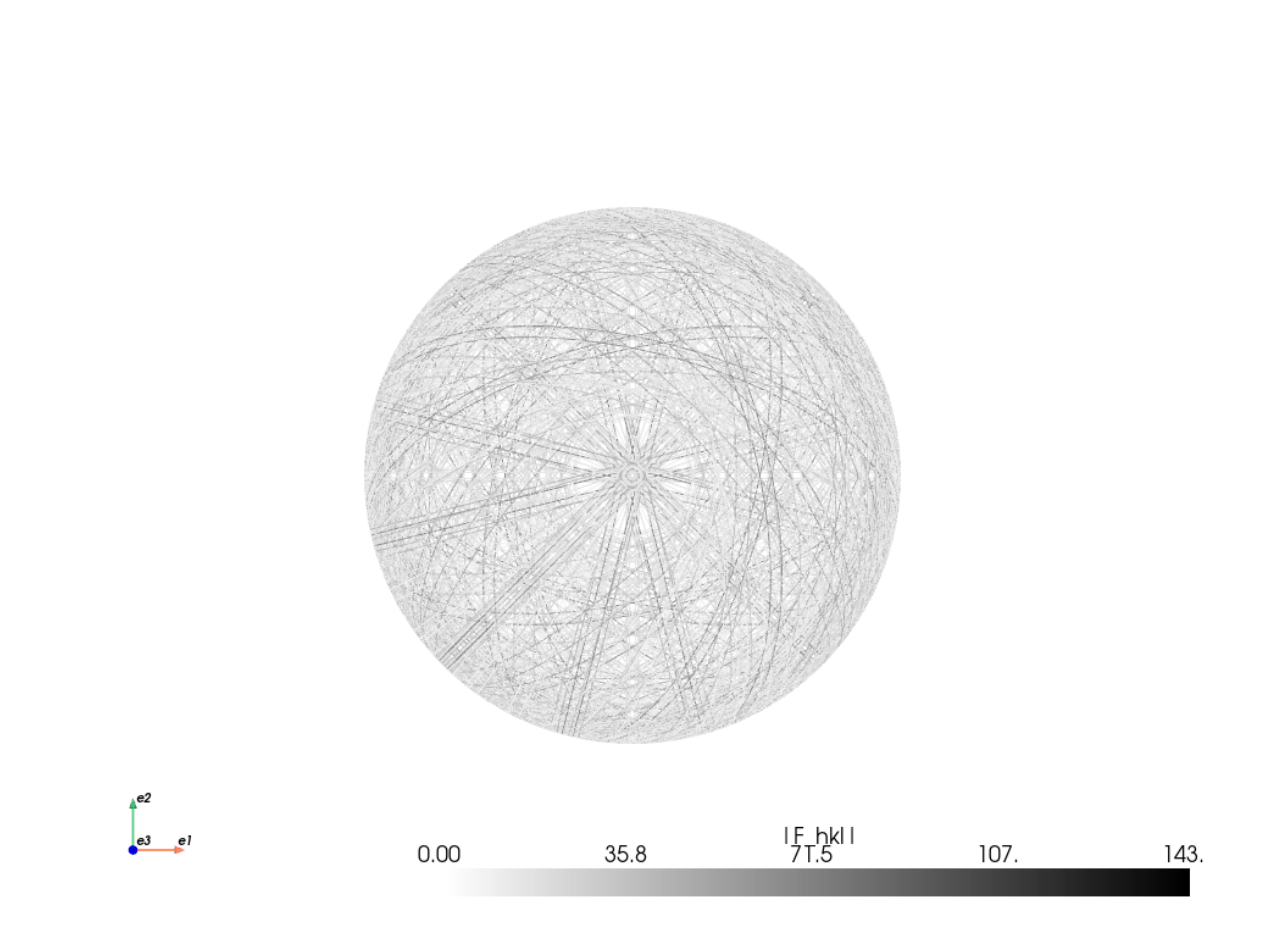
[28]:
```
simulator_sigma = kp.simulations.KikuchiPatternSimulator(ref_sigma)
simulator_sigma
```

[28]:
```
KikuchiPatternSimulator:
ReciprocalLatticeVector (788,), sigma (4/mmm)
[[ 8.  3.  1.]
 [ 8.  3.  0.]
 [ 8.  3. -1.]
 ...
 [-8. -3.  1.]
 [-8. -3.  0.]
 [-8. -3. -1.]]
```

[29]:
```
fig = simulator_sigma.plot(
    hemisphere="both", mode="bands", return_figure=True
)

ax = fig.axes[0]
ax.scatter(simulator_sigma.phase.a_axis, c="r", ec="w")
ax.scatter(simulator_sigma.phase.b_axis, c="g", ec="w")
ax.scatter(simulator_sigma.phase.c_axis, c="b", ec="w")
fig.tight_layout()
```

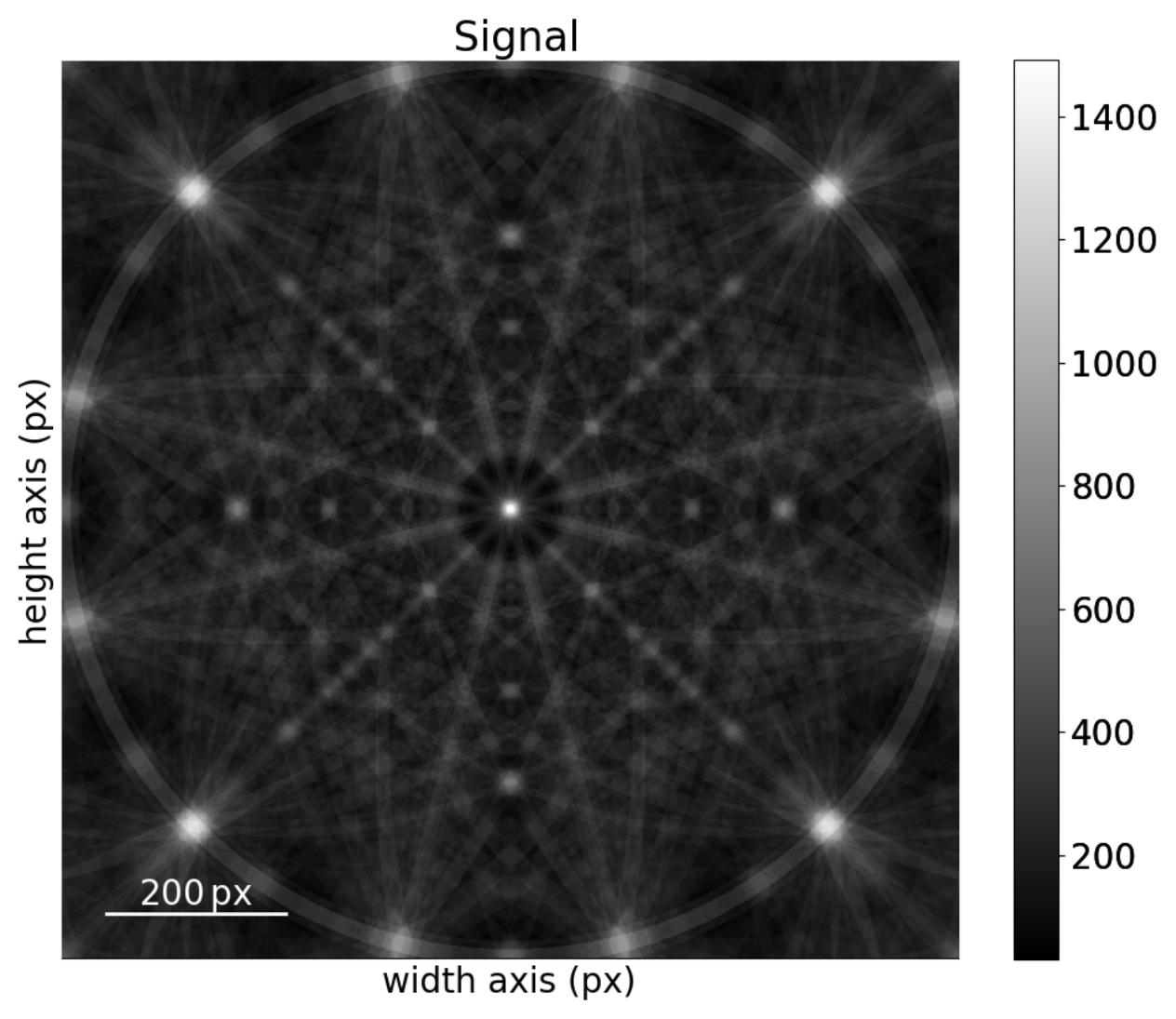



[30]:
```
simulator_sigma.plot("spherical", mode="bands", backend="pyvista")
```

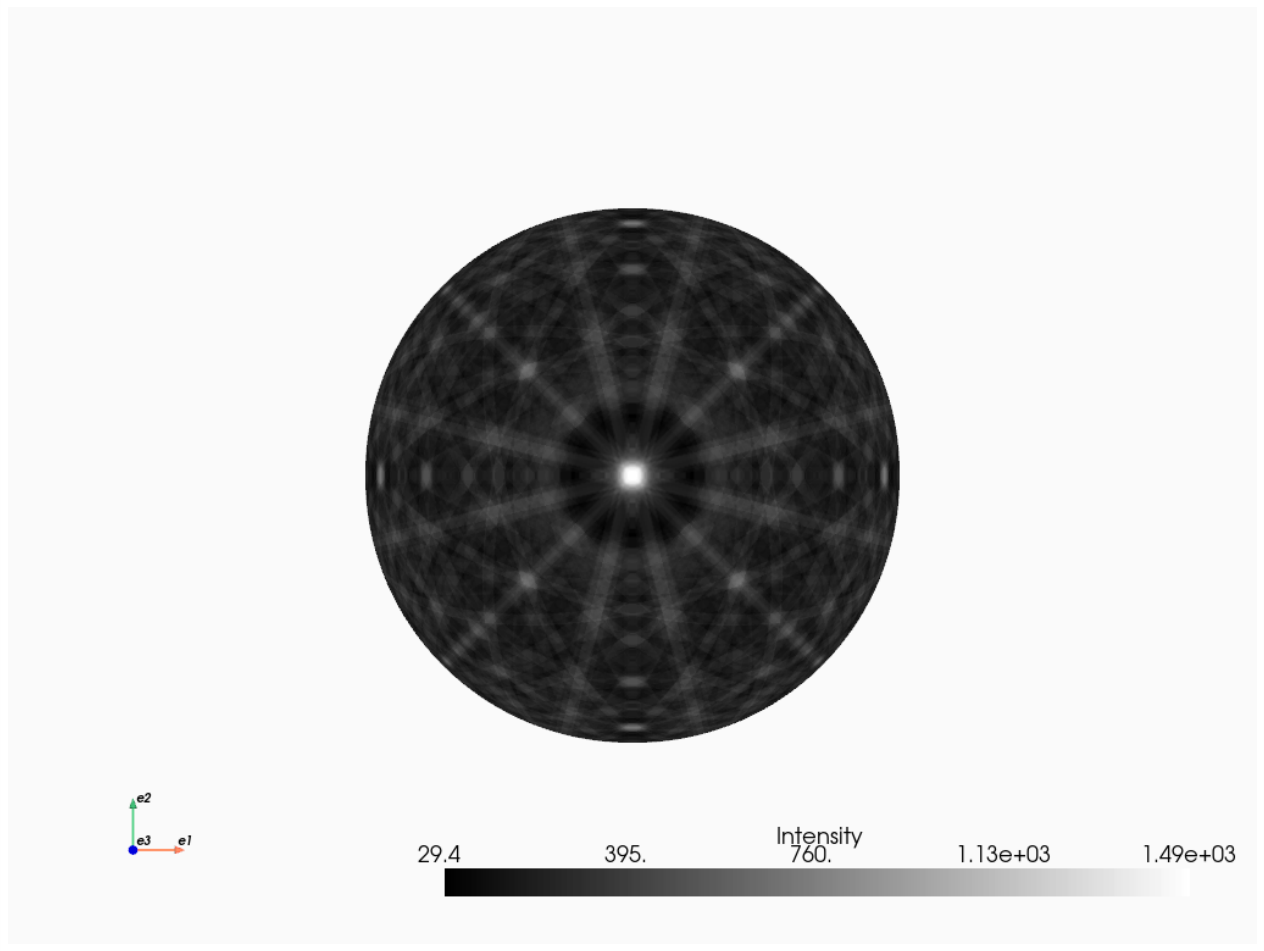



[31]:
```
mp_sigma = simulator_sigma.calculate_master_pattern()
```
```
[########################################] | 100% Completed | 16.51 s
```

[32]:
```
mp_sigma.plot()
```




[33]:
```
mp_sigma.plot_spherical(style="points")
```




### Silicon carbide 6H

[34]:
```
phase_sic = Phase(
    name="sic_6h",
    space_group=186,
    structure=Structure(
        atoms=[
            Atom("Si", [1 / 3, 2 / 3, 0.20778]),
            Atom("C", [1 / 3, 2 / 3, 0.33298]),
            Atom("Si", [1 / 3, 2 / 3, 0.54134]),
            Atom("C", [1 / 3, 2 / 3, 0.66647]),
            Atom("C", [0, 0, 0]),
            Atom("Si", [0, 0, 0.37461]),
        ],
        lattice=Lattice(3.081, 3.081, 15.2101, 90, 90, 120),
    ),
)
phase_sic
```

[34]:
```
<name: sic_6h. space group: P63mc. point group: 6mm. proper point group: 6. color: tab:
↪blue>
```

[35]:
```
ref_sic = ReciprocalLatticeVector.from_min_dspacing(phase_sic)
ref_sic.sanitise_phase()

ref_sic.calculate_structure_factor()

F_sic = abs(ref_sic.structure_factor)
ref_sic = ref_sic[F_sic > 0.05 * F_sic.max()]

ref_sic.calculate_theta(20e3)

ref_sic.print_table()
```

| | h | k | l | d | \|F\|_hkl | \|F\|^2 | \|F\|^2_rel | Mult |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 6 | 2.535 | 18.1 | 328.7 | 100.0 | 1 |
| 0 | 0 | 0 | -6 | 2.535 | 18.1 | 328.7 | 100.0 | 1 |
| 1 | 0 | 0 | 0 | 2.668 | 10.4 | 108.9 | 33.1 | 6 |
| 2 | 0 | 0 | 0 | 1.334 | 9.6 | 92.7 | 28.2 | 6 |
| 1 | 0 | 0 | 1 | 2.628 | 7.7 | 58.8 | 17.9 | 6 |
| 1 | 0 | 0 | -1 | 2.628 | 7.7 | 58.8 | 17.9 | 6 |
| 1 | 0 | 0 | -9 | 1.428 | 7.1 | 50.7 | 15.4 | 6 |
| 1 | 0 | 0 | 9 | 1.428 | 7.1 | 50.7 | 15.4 | 6 |
| 1 | 0 | 0 | -3 | 2.361 | 6.7 | 45.5 | 13.8 | 6 |
| 1 | 0 | 0 | 3 | 2.361 | 6.7 | 45.5 | 13.8 | 6 |
| 2 | -1 | 0 | -2 | 1.510 | 5.9 | 34.7 | 10.6 | 6 |
| 2 | -1 | 0 | 2 | 1.510 | 5.9 | 34.7 | 10.6 | 6 |
| 2 | 0 | 0 | 6 | 1.181 | 5.9 | 34.4 | 10.4 | 6 |
| 2 | 0 | 0 | -6 | 1.181 | 5.9 | 34.3 | 10.4 | 6 |
| 1 | 0 | 0 | 2 | 2.518 | 5.8 | 33.5 | 10.2 | 6 |
| 1 | 0 | 0 | -2 | 2.518 | 5.8 | 33.5 | 10.2 | 6 |
| 2 | -1 | 0 | 8 | 1.197 | 5.7 | 32.9 | 10.0 | 6 |
| 2 | -1 | 0 | -8 | 1.197 | 5.7 | 32.9 | 10.0 | 6 |
| 1 | 0 | 0 | -6 | 1.838 | 5.0 | 25.1 | 7.6 | 6 |
| 1 | 0 | 0 | 6 | 1.838 | 5.0 | 25.0 | 7.6 | 6 |
| 1 | 0 | 0 | -7 | 1.685 | 4.5 | 20.2 | 6.2 | 6 |
| 1 | 0 | 0 | 7 | 1.685 | 4.5 | 20.2 | 6.2 | 6 |
| 1 | 0 | 0 | 8 | 1.548 | 4.3 | 18.8 | 5.7 | 6 |
| 1 | 0 | 0 | -8 | 1.548 | 4.3 | 18.7 | 5.7 | 6 |
| 3 | 0 | 0 | 0 | 0.889 | 4.1 | 17.1 | 5.2 | 12 |
| 0 | 0 | 0 | 18 | 0.845 | 4.0 | 15.7 | 4.8 | 1 |
| 0 | 0 | 0 | -18 | 0.845 | 4.0 | 15.7 | 4.8 | 1 |
| 1 | 0 | 0 | 15 | 0.948 | 3.9 | 14.9 | 4.5 | 6 |
| 1 | 0 | 0 | -15 | 0.948 | 3.9 | 14.9 | 4.5 | 6 |
| 2 | -1 | 0 | 10 | 1.082 | 3.5 | 12.5 | 3.8 | 6 |
| 2 | -1 | 0 | -10 | 1.082 | 3.5 | 12.5 | 3.8 | 6 |
| 3 | -1 | 0 | 0 | 1.008 | 3.4 | 11.7 | 3.6 | 12 |
| 2 | -1 | 0 | 0 | 1.541 | 3.4 | 11.6 | 3.5 | 6 |
| 2 | 2 | 0 | 0 | 0.770 | 3.3 | 11.0 | 3.3 | 10 |
| 2 | -1 | 0 | -16 | 0.809 | 3.1 | 9.9 | 3.0 | 6 |
| 2 | -1 | 0 | 16 | 0.809 | 3.1 | 9.9 | 3.0 | 6 |
| 3 | 0 | 0 | -6 | 0.839 | 2.8 | 8.0 | 2.4 | 12 |
| 3 | 0 | 0 | 6 | 0.839 | 2.8 | 8.0 | 2.4 | 12 |
| 1 | 0 | 0 | -5 | 2.006 | 2.8 | 7.7 | 2.4 | 6 |
| 1 | 0 | 0 | 5 | 2.006 | 2.8 | 7.7 | 2.4 | 6 |

(continues on next page)

(continued from previous page)

```
2   0  18  0.714     2.8      7.7       2.3       6
2   0 -18  0.714     2.8      7.6       2.3       6
2  -1  14  0.888     2.7      7.5       2.3       6
2  -1 -14  0.888     2.7      7.5       2.3       6
3  -1  -8  0.891     2.6      6.7       2.1       12
3  -1   9  0.866     2.6      6.7       2.0       12
3  -1  -9  0.866     2.6      6.7       2.0       12
3  -1   8  0.891     2.6      6.7       2.0       12
0   0 -12  1.268     2.5      6.2       1.9       1
0   0  12  1.268     2.5      6.2       1.9       1
1   0  10  1.321     2.5      6.1       1.9       6
1   0 -10  1.321     2.5      6.1       1.8       6
2  -1  -9  1.138     2.4      5.9       1.8       6
2  -1   9  1.138     2.4      5.9       1.8       6
2   2   6  0.737     2.3      5.4       1.6       10
2   2  -6  0.737     2.3      5.4       1.6       10
3  -1   6  0.937     2.3      5.2       1.6       12
3  -1  -6  0.937     2.3      5.1       1.6       12
3  -1  -2  1.000     2.3      5.1       1.5       12
3  -1   2  1.000     2.3      5.1       1.5       12
3   0  -9  0.787     2.2      5.0       1.5       12
3   0   9  0.787     2.2      5.0       1.5       12
2   0   9  1.047     2.1      4.6       1.4       6
2   0  -9  1.047     2.1      4.6       1.4       6
2  -1   6  1.316     2.0      3.8       1.2       6
2  -1  -6  1.316     2.0      3.8       1.2       6
3  -1 -15  0.715     1.9      3.7       1.1       12
3  -1  15  0.715     1.9      3.7       1.1       12
1   0  16  0.895     1.9      3.6       1.1       6
2  -1   1  1.533     1.9      3.6       1.1       6
2  -1  -1  1.533     1.9      3.6       1.1       6
1   0 -16  0.895     1.9      3.6       1.1       6
2  -1  -4  1.428     1.8      3.3       1.0       6
2  -1   4  1.428     1.8      3.3       1.0       6
1   0  -4  2.184     1.8      3.2       1.0       6
3   1  -2  0.737     1.8      3.2       1.0       12
3   1   2  0.737     1.8      3.2       1.0       12
1   0   4  2.184     1.8      3.2       1.0       6
2   2  -8  0.714     1.8      3.1       0.9       10
3  -1 -10  0.841     1.7      3.0       0.9       12
2   2   8  0.714     1.7      3.0       0.9       10
3  -1  10  0.841     1.7      3.0       0.9       12
2  -1  -3  1.474     1.7      3.0       0.9       6
2  -1   3  1.474     1.7      3.0       0.9       6
1   0  14  1.006     1.7      2.9       0.9       6
1   0 -14  1.006     1.7      2.8       0.9       6
1   0 -17  0.848     1.7      2.8       0.8       6
1   0  17  0.848     1.7      2.8       0.8       6
2   0 -12  0.919     1.7      2.7       0.8       6
2   0  12  0.919     1.7      2.7       0.8       6
3  -1  -1  1.006     1.6      2.6       0.8       12
3  -1   1  1.006     1.6      2.6       0.8       12
```

(continues on next page)

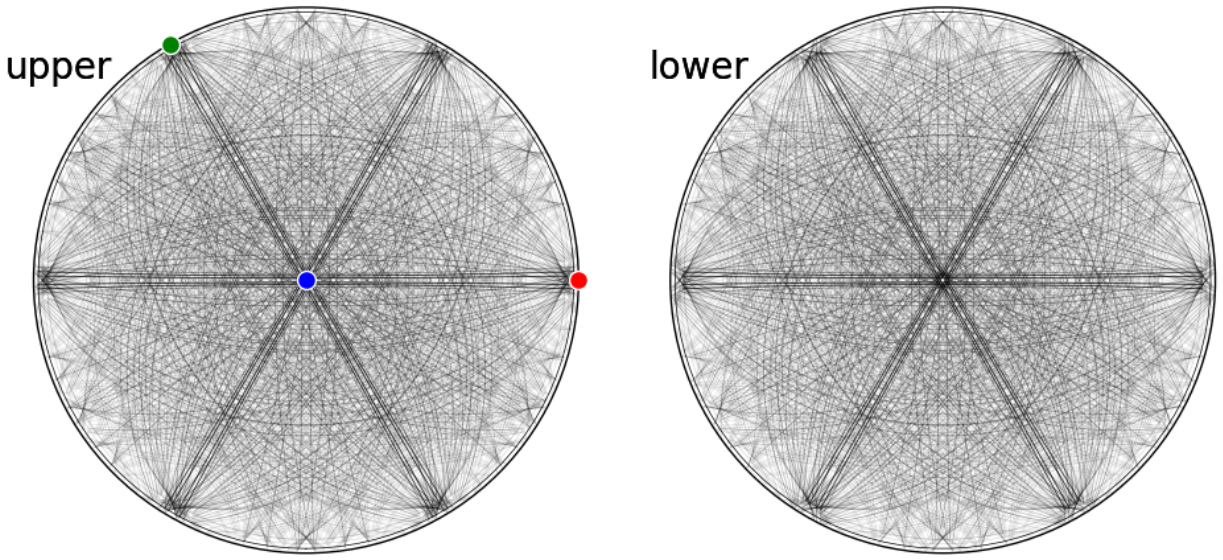
(continued from previous page)

```
2  -1  15  0.847     1.6      2.5       0.8       6
2  -1 -15  0.847     1.6      2.5       0.8       6
3  -1   3  0.989     1.6      2.5       0.8       12
3  -1  -3  0.989     1.6      2.5       0.8       12
2   0  -1  1.329     1.5      2.4       0.7       6
2   0   1  1.329     1.5      2.4       0.7       6
1   0 -18  0.806     1.5      2.2       0.7       6
3  -1 -14  0.739     1.5      2.2       0.7       12
1   0  18  0.806     1.5      2.2       0.7       6
3  -1  14  0.739     1.5      2.2       0.7       12
2   0 -15  0.807     1.5      2.1       0.7       6
2   0  15  0.807     1.5      2.1       0.7       6
2   2  -2  0.766     1.5      2.1       0.6       10
2   2   2  0.766     1.5      2.1       0.6       10
2   0   3  1.290     1.4      2.0       0.6       6
2   0  -3  1.290     1.4      2.0       0.6       6
2  -1  -7  1.257     1.4      2.0       0.6       6
2  -1   7  1.257     1.4      2.0       0.6       6
3  -1   7  0.915     1.4      1.9       0.6       12
3  -1  -7  0.915     1.4      1.9       0.6       12
1   0  11  1.228     1.4      1.9       0.6       6
1   0 -11  1.228     1.4      1.9       0.6       6
3   0  -3  0.876     1.4      1.8       0.6       12
3   0   3  0.876     1.4      1.8       0.6       12
3   0   1  0.888     1.3      1.8       0.5       12
3   0  -1  0.888     1.3      1.8       0.5       12
3   0   8  0.806     1.3      1.6       0.5       12
2   0  -8  1.092     1.2      1.6       0.5       6
3   0  -8  0.806     1.2      1.5       0.5       12
2   0   8  1.092     1.2      1.5       0.4       6
2   0   7  1.137     1.2      1.5       0.4       6
2   0  -7  1.137     1.2      1.5       0.4       6
3   0  -7  0.823     1.2      1.4       0.4       12
3   0   7  0.823     1.2      1.4       0.4       12
2   0  -2  1.314     1.2      1.4       0.4       6
2   0   2  1.314     1.2      1.4       0.4       6
2   2   9  0.701     1.2      1.4       0.4       10
2   2  -9  0.701     1.2      1.4       0.4       10
1   0  13  1.072     1.1      1.3       0.4       6
1   0 -13  1.072     1.1      1.3       0.4       6
3   0  12  0.728     1.1      1.2       0.4       12
3   1   3  0.732     1.1      1.2       0.4       12
3   1  -3  0.732     1.1      1.2       0.4       12
3   0 -12  0.728     1.1      1.2       0.4       12
3   0   2  0.883     1.1      1.1       0.3       12
3   0  -2  0.883     1.1      1.1       0.3       12
3   1  -1  0.739     1.0      1.0       0.3       12
3   1   1  0.739     1.0      1.0       0.3       12
3   1   7  0.701     0.9      0.9       0.3       12
3   1  -7  0.701     0.9      0.9       0.3       12
```

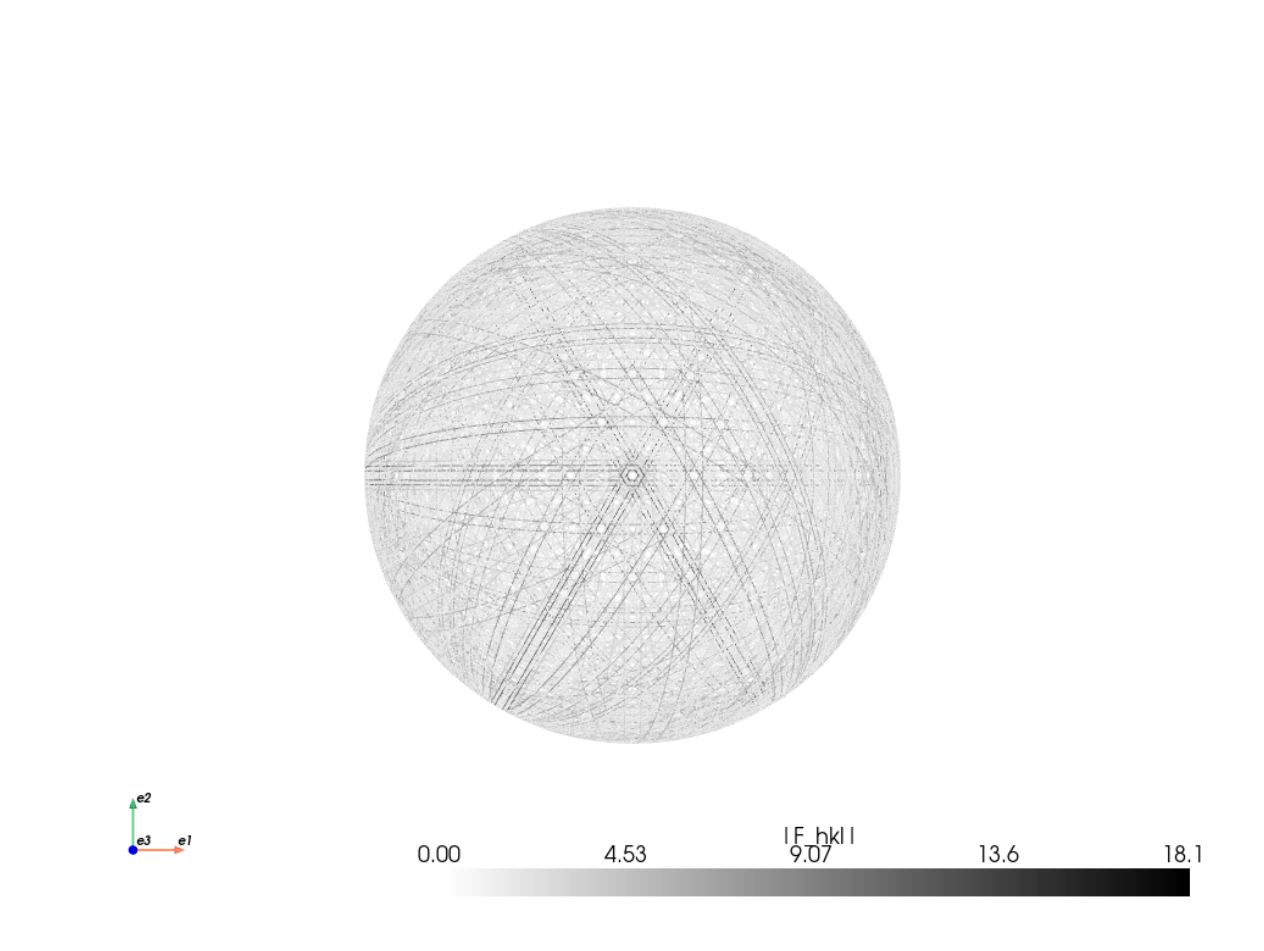
[36]:
```
simulator_sic = kp.simulations.KikuchiPatternSimulator(ref_sic)
simulator_sic
```

[36]:
```
KikuchiPatternSimulator:
ReciprocalLatticeVector (896,), sic_6h (6mm)
[[ 3.  1.  7.]
 [ 3.  1.  3.]
 [ 3.  1.  2.]
 ...
 [-3. -1. -2.]
 [-3. -1. -3.]
 [-3. -1. -7.]]
```

[37]:
```
fig = simulator_sic.plot(hemisphere="both", mode="bands", return_figure=True)

ax = fig.axes[0]
ax.scatter(simulator_sic.phase.a_axis, c="r", ec="w")
ax.scatter(simulator_sic.phase.b_axis, c="g", ec="w")
ax.scatter(simulator_sic.phase.c_axis, c="b", ec="w")
fig.tight_layout()
```




[38]:
```
simulator_sic.plot("spherical", mode="bands", backend="pyvista")
```



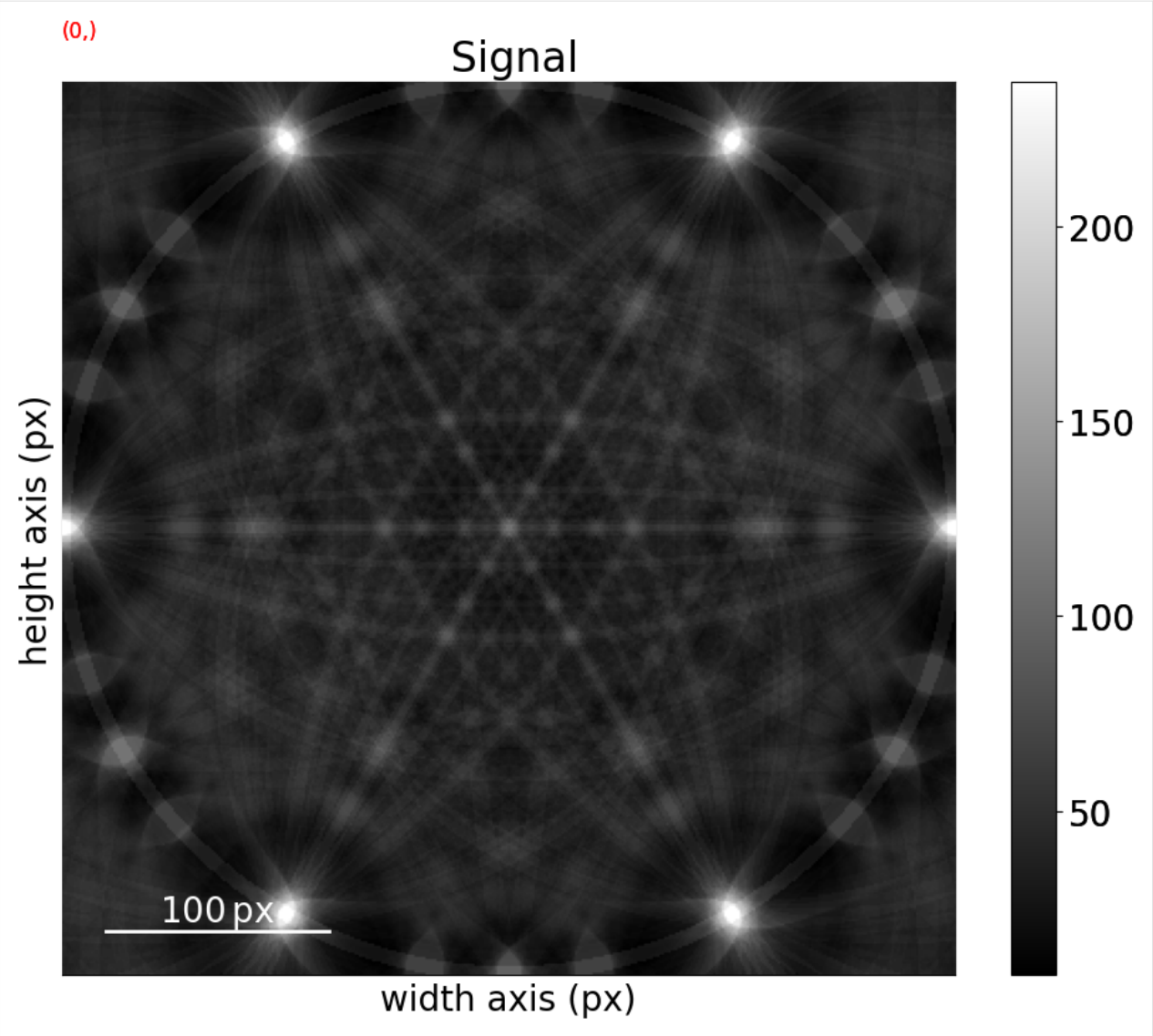

[39]:
```
mp_sic = simulator_sic.calculate_master_pattern(
    hemisphere="both", half_size=200
)
```

```
[########################################] | 100% Completed | 8.03 s
```

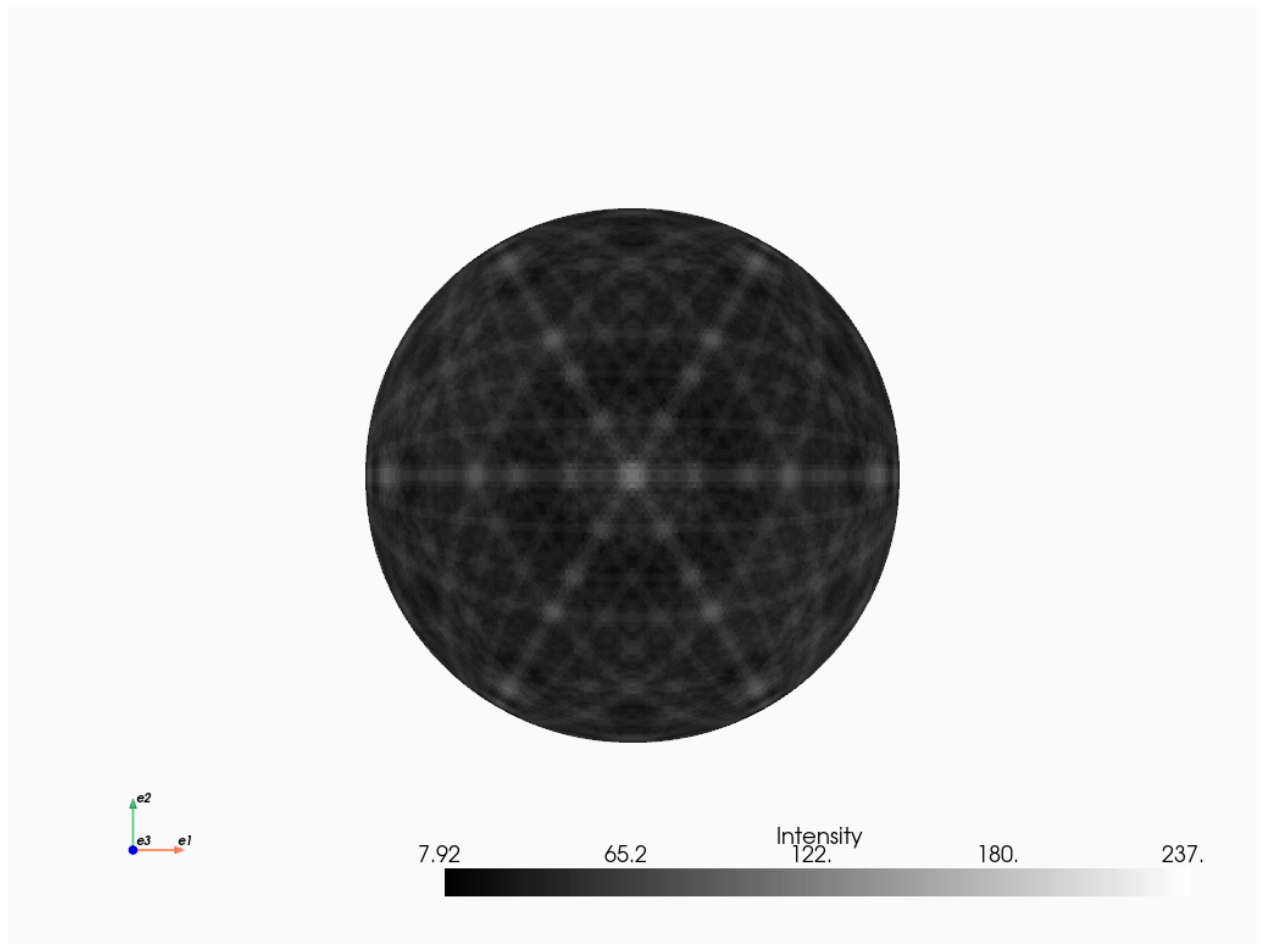
[40]:
```
mp_sic
```

[40]:
```
<EBSDMasterPattern, title: , dimensions: (2|401, 401)>
```

[41]:
```
mp_sic.plot(navigator=None)
```




[42]:
```
mp_sic.plot_spherical(style="points")
```




### 1.2.5 Advanced usage

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### Multivariate analysis

See HyperSpy's user guide for explanations on available multivariate statistical analysis ("machine learning") methods and more examples of their use.

### Denoising EBSD patterns with dimensionality reduction

Let's use principal component analysis (PCA) followed by dimensionality reduction to increase the signal-to-noise ratio $S/N$ in a small Nickel EBSD data set, here called denoising. This denoising is explained further in [Ånes *et al.*, 2020].

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

import hyperspy.api as hs
import kikuchipy as kp


s = kp.data.nickel_ebsd_large(allow_download=True)  # External download
s
```

[1]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

Let's first increase $S/N$ by removing the undesired static and dynamic backgrounds

[2]:
```
s.remove_static_background()
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 101.32 ms
[########################################] | 100% Completed | 709.30 ms
```

Followed by averaging each pattern with the eight nearest patterns using a Gaussian kernel of $\sigma = 2$ centered on the pattern being averaged

[3]:
```
s.average_neighbour_patterns(window="gaussian", std=2)
```

```
[########################################] | 100% Completed | 409.57 ms
```

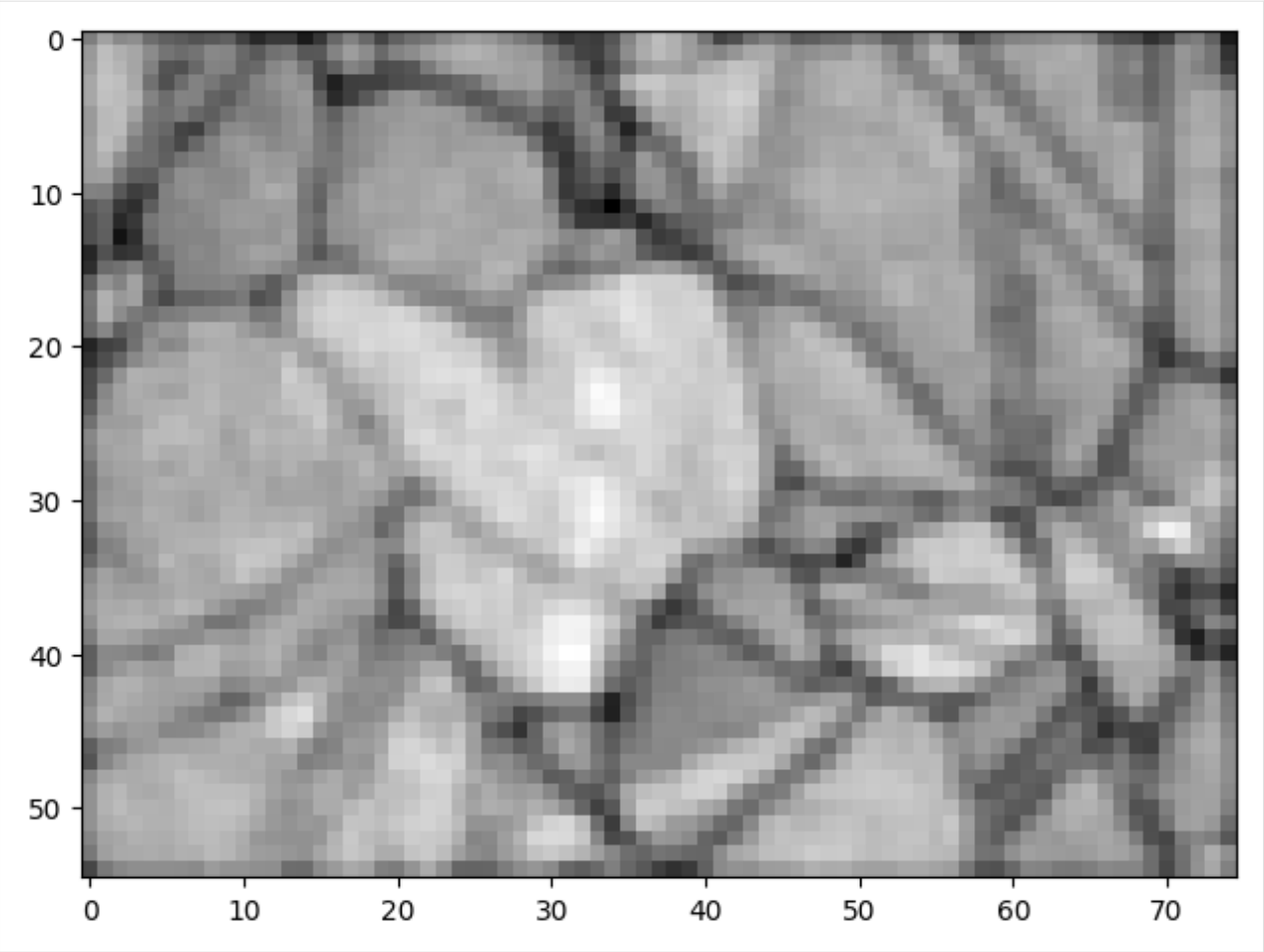
We use the average image quality (IQ) and the IQ map to assess how successful our denoising was. Let's inspect these before denoising

[4]:
```
iq1 = s.get_image_quality()
```

```
[########################################] | 100% Completed | 404.33 ms
```

[5]:
```
print(iq1.mean())

plt.imshow(iq1, cmap="gray")
plt.tight_layout()
```

```
0.30131632
```




The basic idea of PCA is to decompose the data to a set of values of linearly uncorrelated, orthogonal variables called principal components, or component factors in HyperSpy, while retaining as much as possible of the variation in the data. The factors are ordered by variance. For each component factor, we obtain a component loading, showing the variation of the factor's strength from one observation point to the next.

Ideally, the first component corresponds to the crystallographic feature most prominent in the data, for example the largest grain, the next corresponds to the second largest feature, and so on, until the later components at some point contain only noise. If this is the case, we can increase $S/N$ by reconstructing our EBSD signal from the first $n$ components only, discarding the later components.

PCA decomposition in HyperSpy is done via singular value decomposition (SVD) as implemented in scikit-learn. To prevent number overflow during the decomposition, our detector pixels data type must be of the float or complex type

[6]:
```
dtype_orig = s.data.dtype
s.change_dtype("float32")
```

To reduce the effect of the mean intensity per pattern on the overall variance in the entire dataset, we center the patterns by subtracting their mean intensity before decomposing. This is done by passing `centre="signal"`. Considering the expected number of components in our small Nickel data set, let's keep only 100 of the ranked components

[7]:
```
n_components = 100
s.decomposition(
    algorithm="SVD",
    output_dimension=n_components,
```

(continues on next page)

(continued from previous page)

```
    centre="signal",
)
```

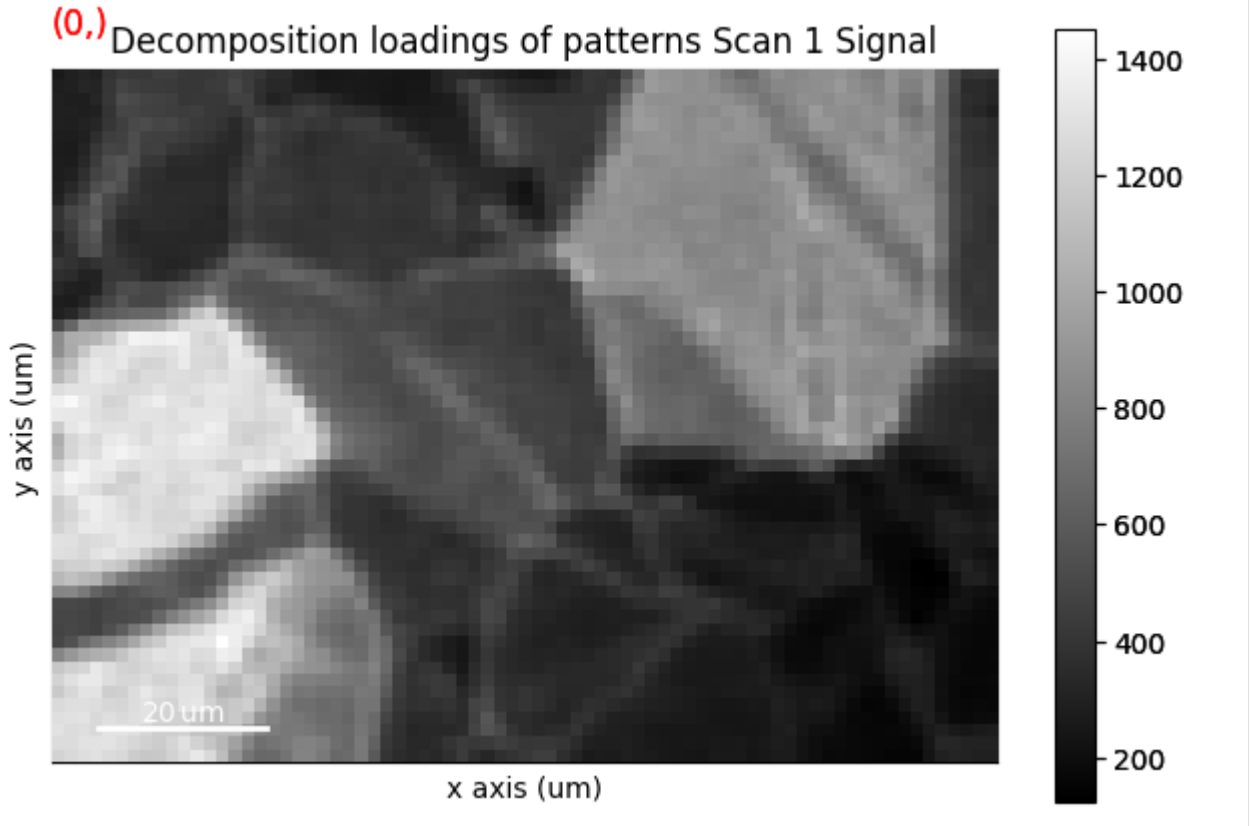
```
Decomposition info:
  normalize_poissonian_noise=False
  algorithm=SVD
  output_dimension=100
  centre=signal
```

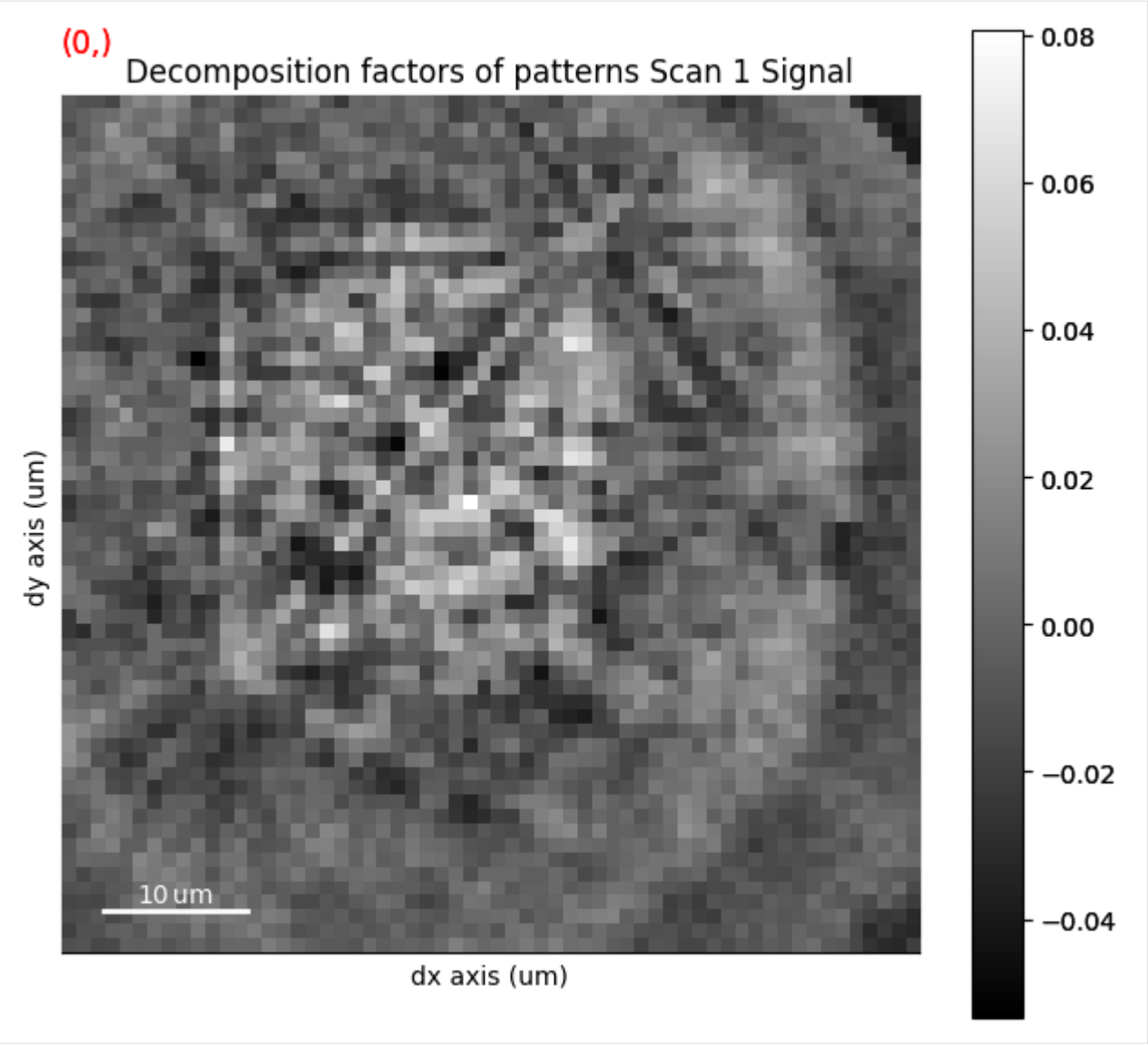
[8]:
```
s.change_dtype(dtype_orig)
```

We can inspect our decomposition results by clicking through the ranked component factors and their corresponding loading

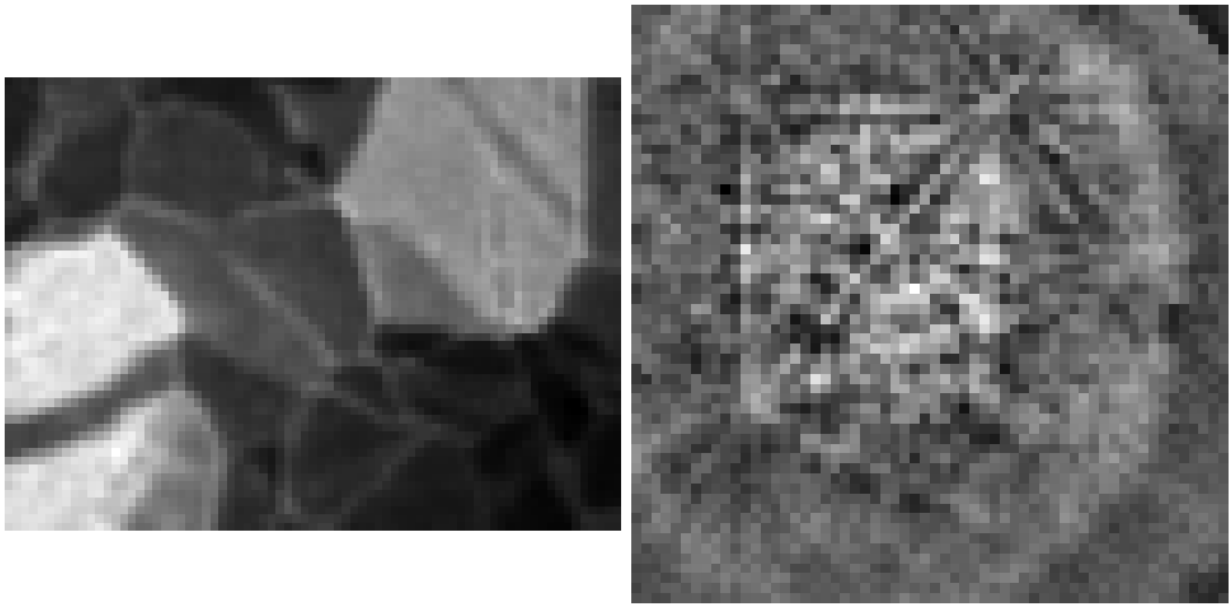
[9]:
```
s.plot_decomposition_results()
```

```
WARNING:hyperspy.drawing.mpl_he:Navigation sliders not available. No toolkit registered.␣
↪Install hyperspy_gui_ipywidgets or hyperspy_gui_traitsui GUI elements.
```



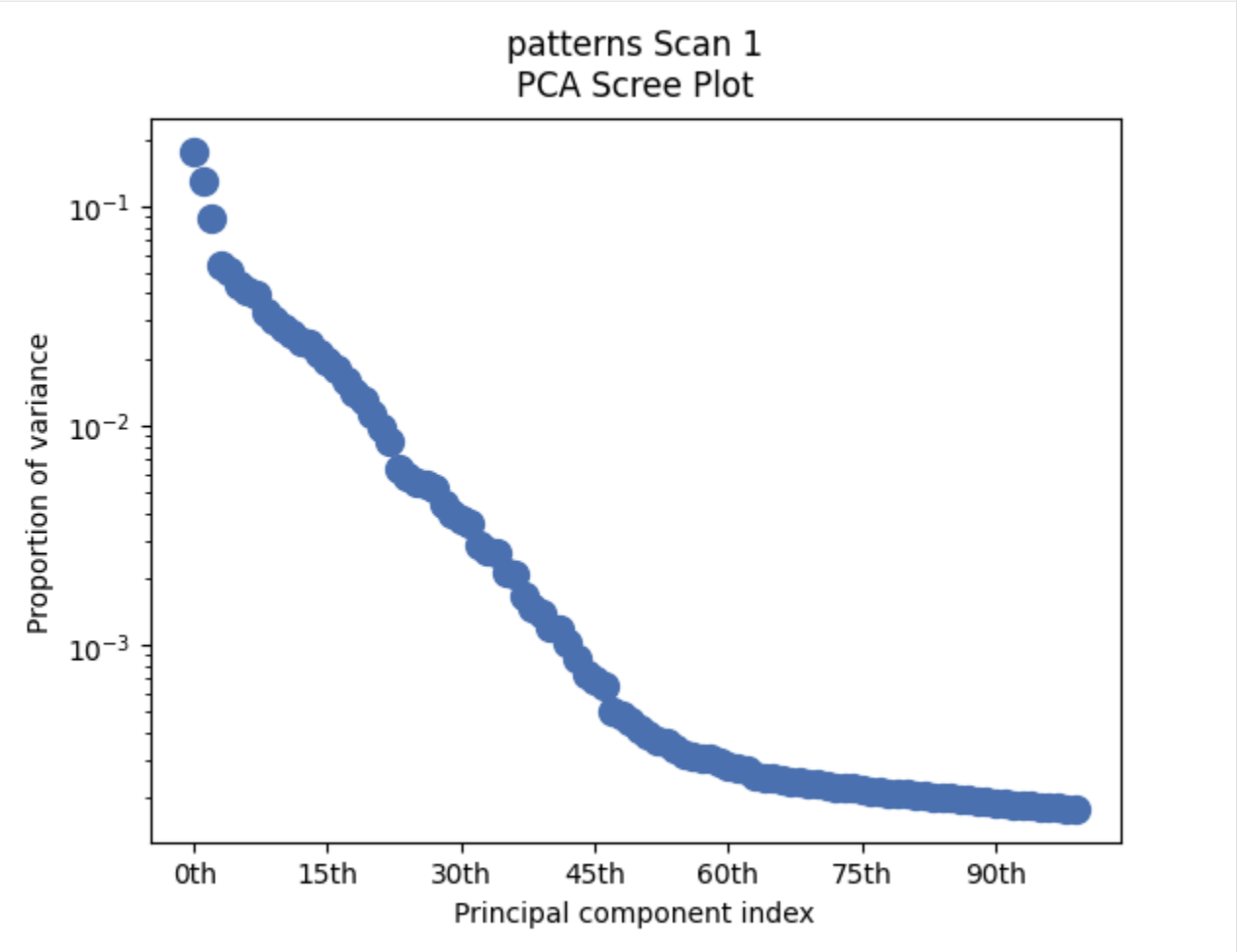



[10]:
```
factors = s.learning_results.factors  # (n detector pixels, m components)
sig_shape = s.axes_manager.signal_shape[::-1]
loadings = s.learning_results.loadings  # (n patterns, m components)
nav_shape = s.axes_manager.navigation_shape[::-1]

fig, ax = plt.subplots(ncols=2, figsize=(10, 5))
ax[0].imshow(loadings[:, 0].reshape(nav_shape), cmap="gray")
ax[0].axis("off")
ax[1].imshow(factors[:, 0].reshape(sig_shape), cmap="gray")
ax[1].axis("off")
fig.tight_layout(w_pad=0)
```



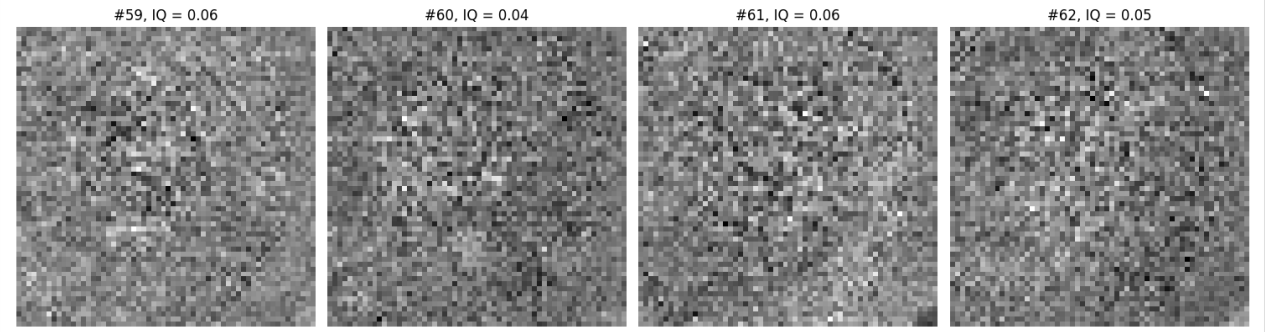
We can also inspect the so-called scree plot of the proportion of variance as a function of the ranked components

[11]:
```
_ = s.plot_explained_variance_ratio(n=n_components)
```




The slope of the proportion of variance seems to fall after about 50-60 components. Let's inspect the components 60-64 for any useful signal

[12]:
```
fig, ax = plt.subplots(ncols=4, figsize=(15, 5))
for i in range(4):
    factor_idx = i + 59
    factor = factors[:, factor_idx].reshape(sig_shape)
    factor_iq = kp.pattern.get_image_quality(factor)
    ax[i].imshow(factor, cmap="gray")
    ax[i].set_title(f"#{factor_idx}, IQ = {np.around(factor_iq, 2)}")
    ax[i].axis("off")
fig.tight_layout()
```




It seems reasonable to discard these components. Note, however, that the selection of a suitable number of components is in general difficult.

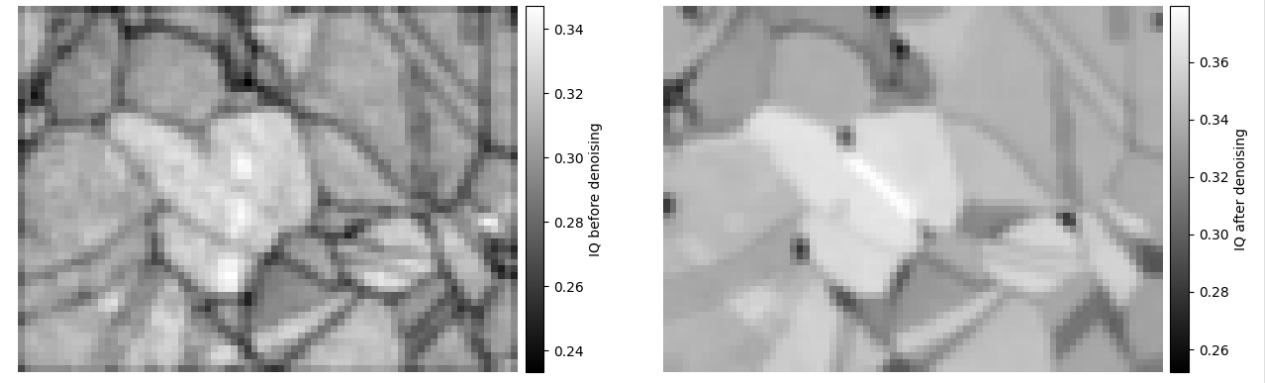
[13]:
```
s2 = s.get_decomposition_model(components=59)
```

[14]:
```
iq2 = s2.get_image_quality()
iq2.mean()
```

```
[########################################] | 100% Completed | 404.15 ms
```

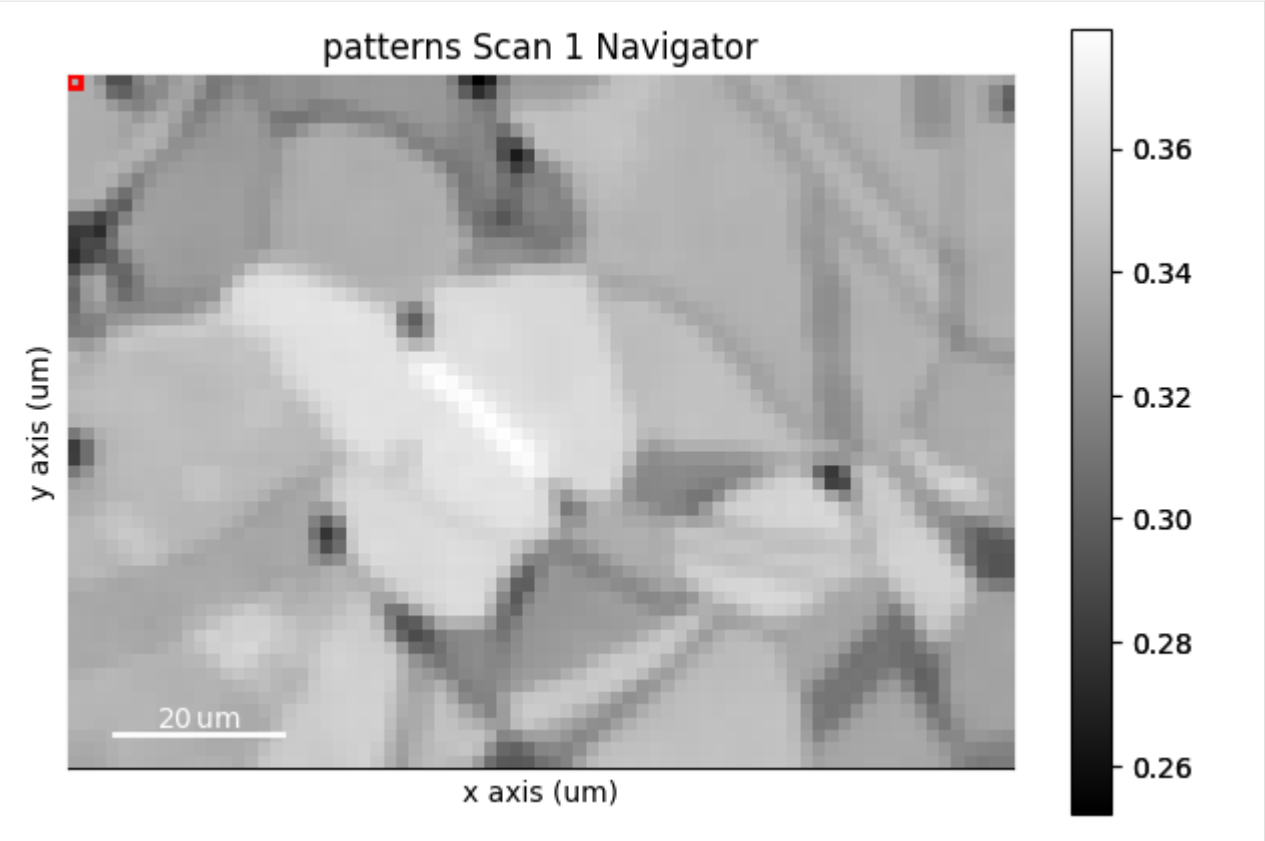
[14]:
```
0.33968845
```

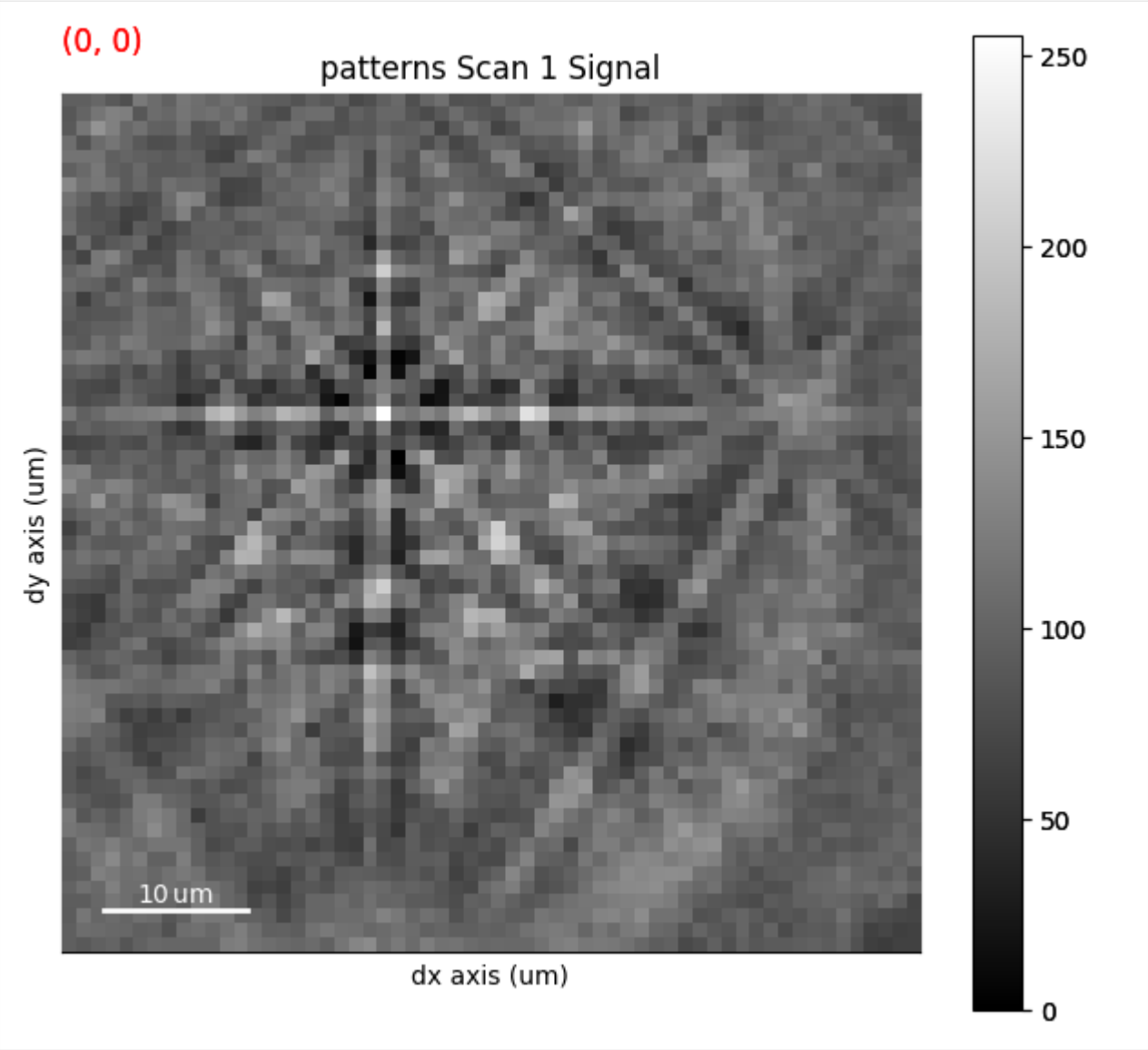
[15]:
```
fig, ax = plt.subplots(ncols=2, figsize=(15, 4))
im0 = ax[0].imshow(iq1, cmap="gray")
ax[0].axis("off")
fig.colorbar(im0, ax=ax[0], pad=0.01, label="IQ before denoising")
im1 = ax[1].imshow(iq2, cmap="gray")
ax[1].axis("off")
fig.colorbar(im1, ax=ax[1], pad=0.01, label="IQ after denoising")
fig.tight_layout(w_pad=-10)
```

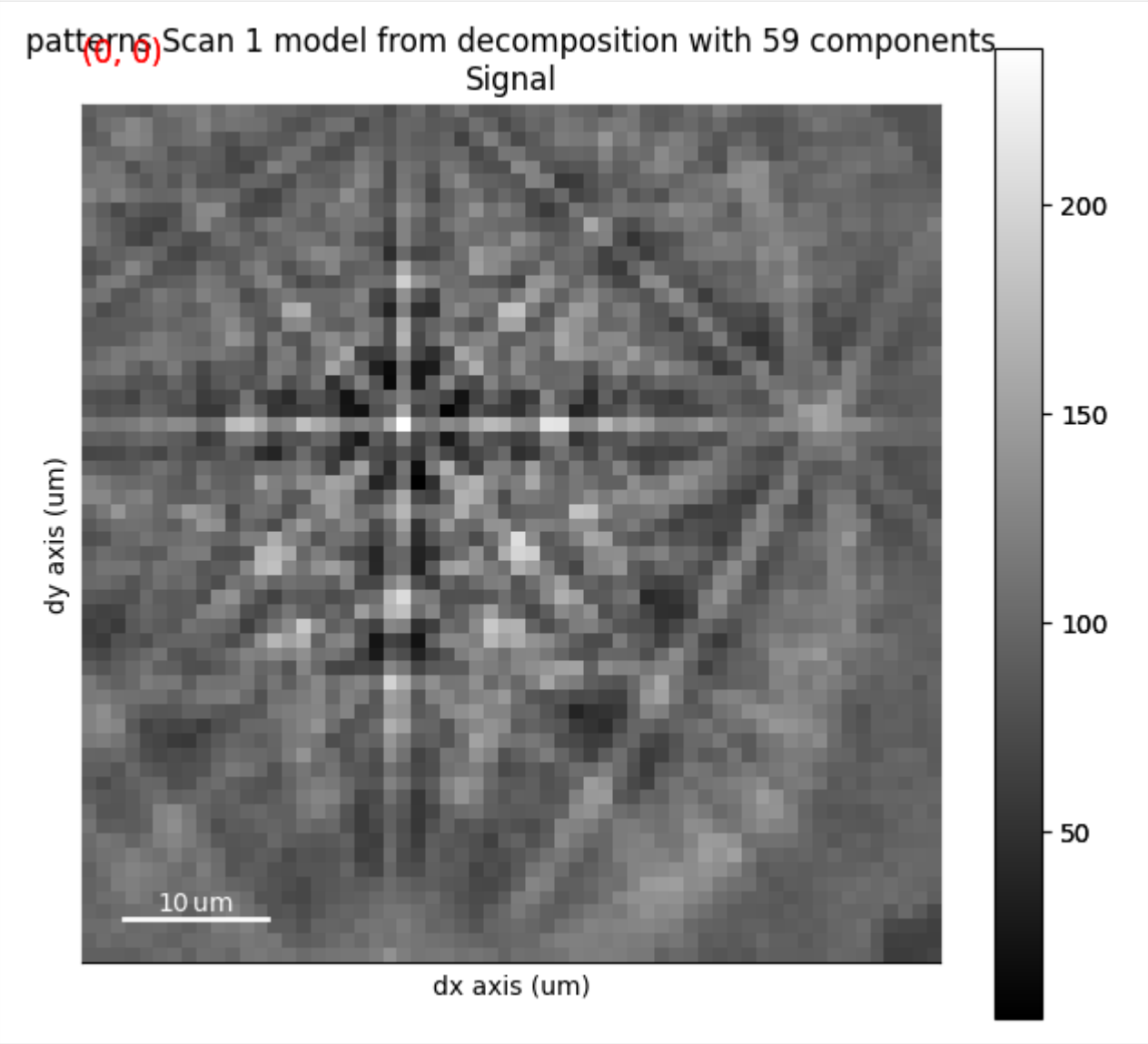



We see that the average IQ increased from 0.30 to 0.34. We can inspect the results per pattern by plotting the signal before and after denoising in the same navigator

[16]:
```
hs.plot.plot_signals([s, s2], navigator=hs.signals.Signal2D(iq2))
```


patterns Scan 1 Navigator
y axis (um)
x axis (um)
20 um
0.36
0.34
0.32
0.30
0.28
0.26


(0, 0)
patterns Scan 1 Signal
dy axis (um)
dx axis (um)
10 um
250
200
150
100
50
0




### 1.2.6 Tutorials given at workshops

**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

### M&M 2021 Sunday Short Course

**Note**

This notebook was used to demonstrate EBSD analysis with kikuchipy as part of the HyperSpy workshop during the Microscopy & Microanalysis Virtual Meeting in 2021. The offical name of the workshop is M&M 2021 Sunday Short Course *X-15 Data Analysis in Materials Science*.

This notebook has been updated to work with the current release of kikuchipy and to fit our documentation format. The original notebook, as presented during the workshop, is available via this GitHub repository hosted by the National Institute of Standards and Technology (NIST).

### EBSD analysis of polycrystalline nickel

The goal of EBSD analysis is often to determine the crystal orientation from each EBSD pattern, typically called *indexing*. One approach is dictionary indexing, first described in [Chen *et al.*, 2015]. Here we'll demonstrate how to do this in kikuchipy. The implementation is based on the one in EMsoft, as described in [Jackson *et al.*, 2019].

Dictionary indexing is not as tried and tested as the commonly used Hough indexing. To aid the evaluation of dictionary indexing results, we therefore first obtain several maps to get an overview of the quality of the EBSD patterns and the features in the region of interest before indexing, independent of any bias introdued in indexing. After indexing, we'll also inspect the results visually using dynamical and geometrical EBSD simulations.

Set Matplotlib plotting backend and import packages

[1]:
```
# Exchange inline for notebook or qt5 (from pyqt) for interactive plotting
%matplotlib inline

import hyperspy.api as hs
import matplotlib.pyplot as plt

from diffsims.crystallography import ReciprocalLatticeVector
import kikuchipy as kp
from orix import io, quaternion, sampling, vector


plt.rcParams.update(
    {"figure.facecolor": "w", "font.size": 15, "figure.dpi": 75}
)
```

Load (and download) an EBSD dataset of polycrystalline, recrystallized Nickel which is part of the `kikuchipy.data` module ("large" = 13 MB, compared to "small" < 1 MB).

[2]:
```
s = kp.data.nickel_ebsd_large(allow_download=True)
s
```

[2]:
```
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

Inspect the navigation and signal dimensions (more closely) in the `axes_manager`

[3]:
```
print(s.axes_manager)
```

```
<Axes manager, axes: (75, 55|60, 60)>
            Name |   size |  index |  offset |   scale |  units
```

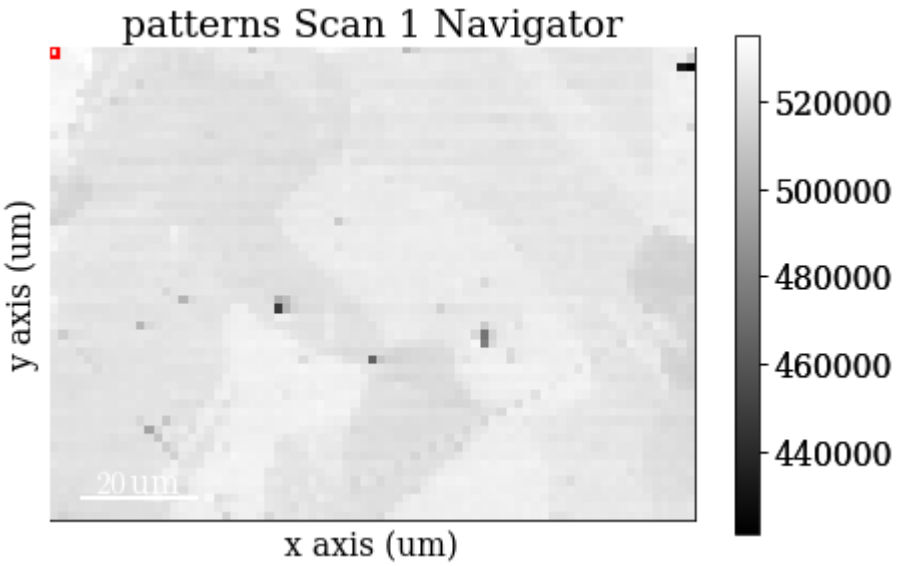
(continues on next page)

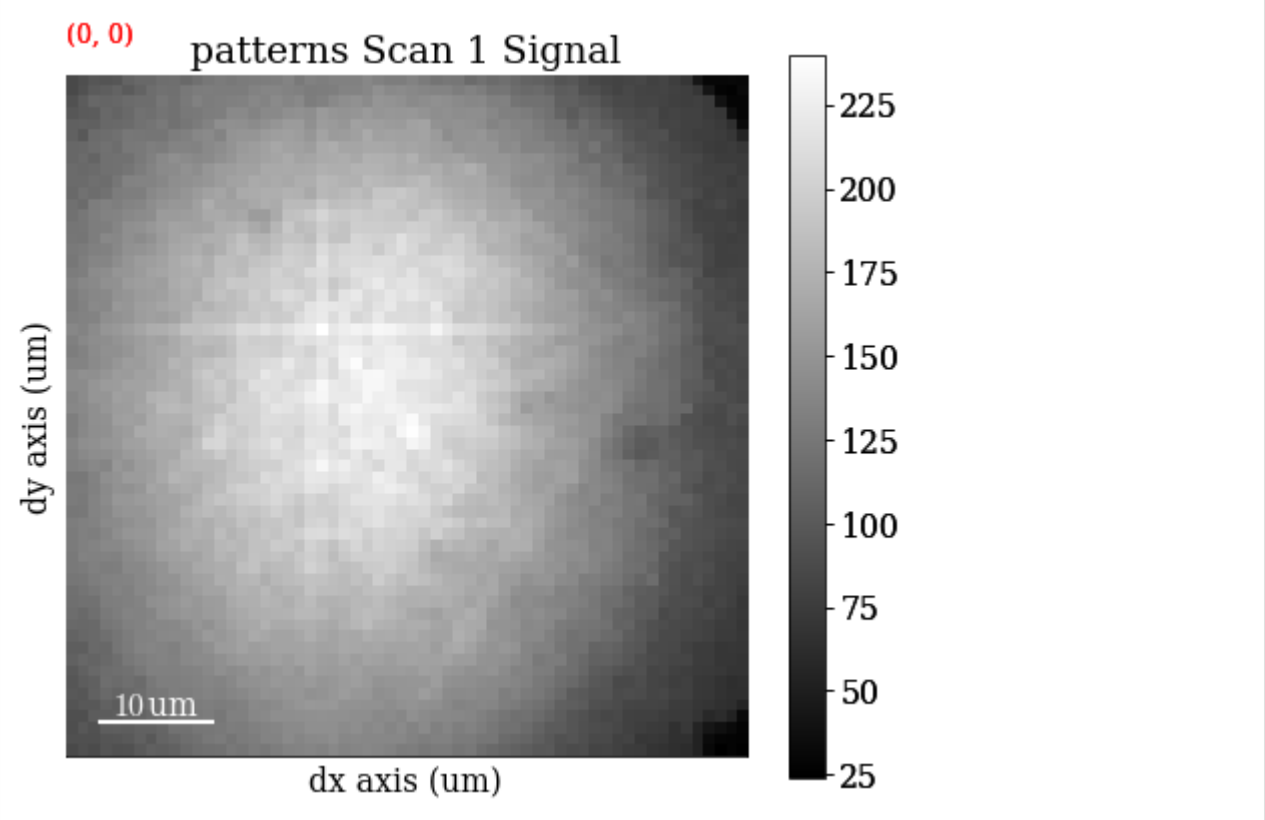
(continued from previous page)

```
================ | ====== | ====== | ======= | ======= | ======
               x |     75 |      0 |       0 |     1.5 |     um
               y |     55 |      0 |       0 |     1.5 |     um
---------------- | ------ | ------ | ------- | ------- | ------
              dx |     60 |      0 |       0 |       1 |     um
              dy |     60 |      0 |       0 |       1 |     um
```

Plot the data (by navigating the patterns in a mean intensity map) with `plot()`

[4]:
```
s.plot()
```






Note that `kikuchipy` has a `kikuchipy.load()` function almost identical to `hyperspy.api.load()`, which can read several commercial EBSD formats. See the *IO tutorial* for more information.

### Pre-pattern-processing maps

### Mean intensity in each pattern

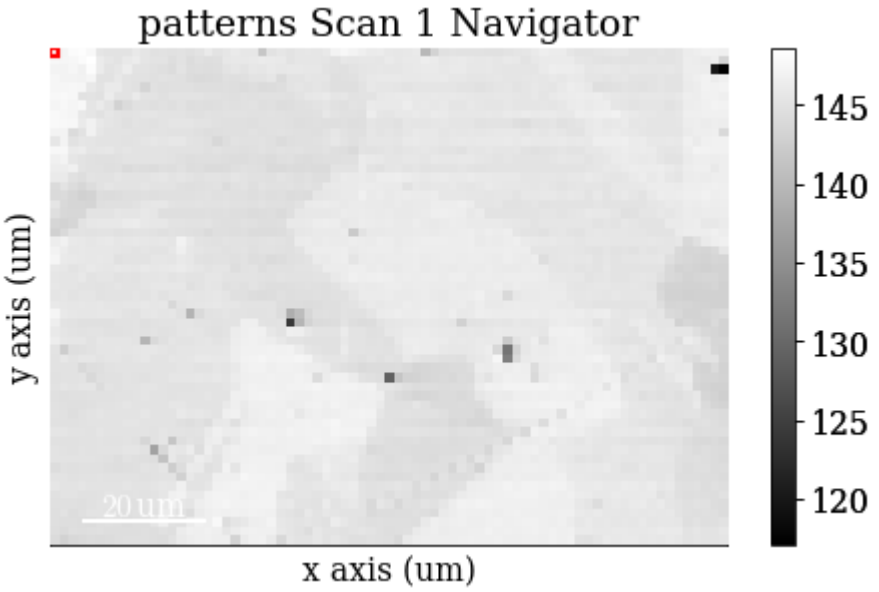
Get the map of the mean intensity in each pattern with `mean()`

[5]:
```
mean_intensity = s.mean(axis=(2, 3))
mean_intensity
```

[5]:
```
<BaseSignal, title: patterns Scan 1, dimensions: (75, 55|)>
```

Plot the mean intensity map

[6]:
```
mean_intensity.plot()
```




### Virtual backscatter electron images

Inspect angle resolved backscatter electron (BSE) images, typically called VBSE/vBSE/virtual diode imaging.

Create a `VirtualBSEImager`

[7]:
```
vbse_imager = kp.imaging.VirtualBSEImager(s)
vbse_imager
```

[7]:
```
VirtualBSEImager for <EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
```

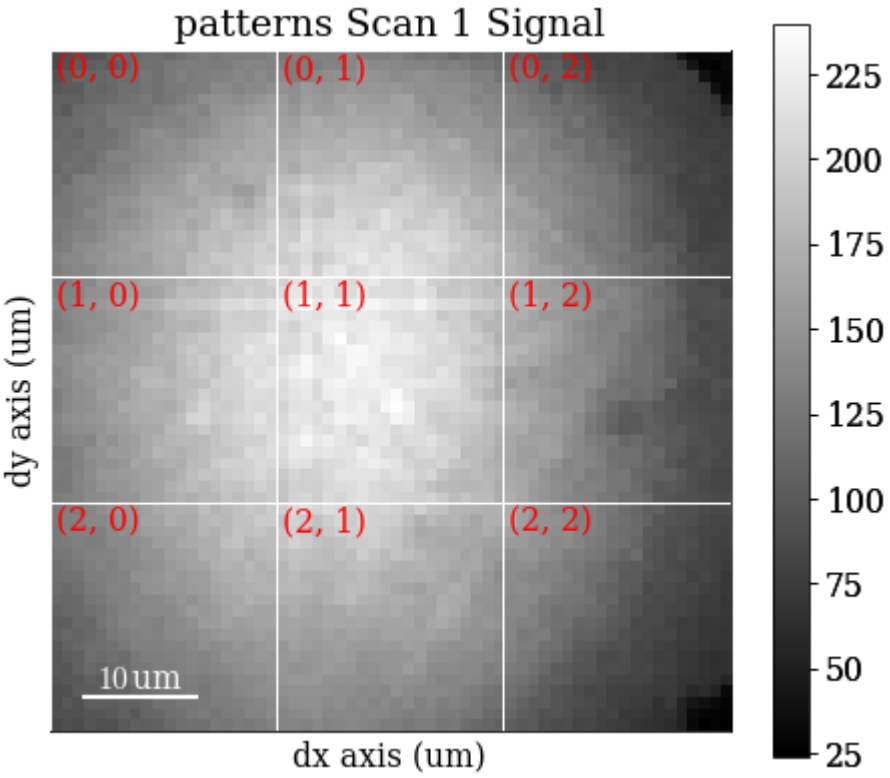
### One image per VBSE grid tile

Separate the EBSD detector (signal dimensions) into a (3 x 3) grid by setting `grid_shape`

[8]:
```
vbse_imager.grid_shape = (3, 3)
```

Plot the grid with `plot_grid()` (not keeping the output signal)

[9]:
```
_ = vbse_imager.plot_grid();
```



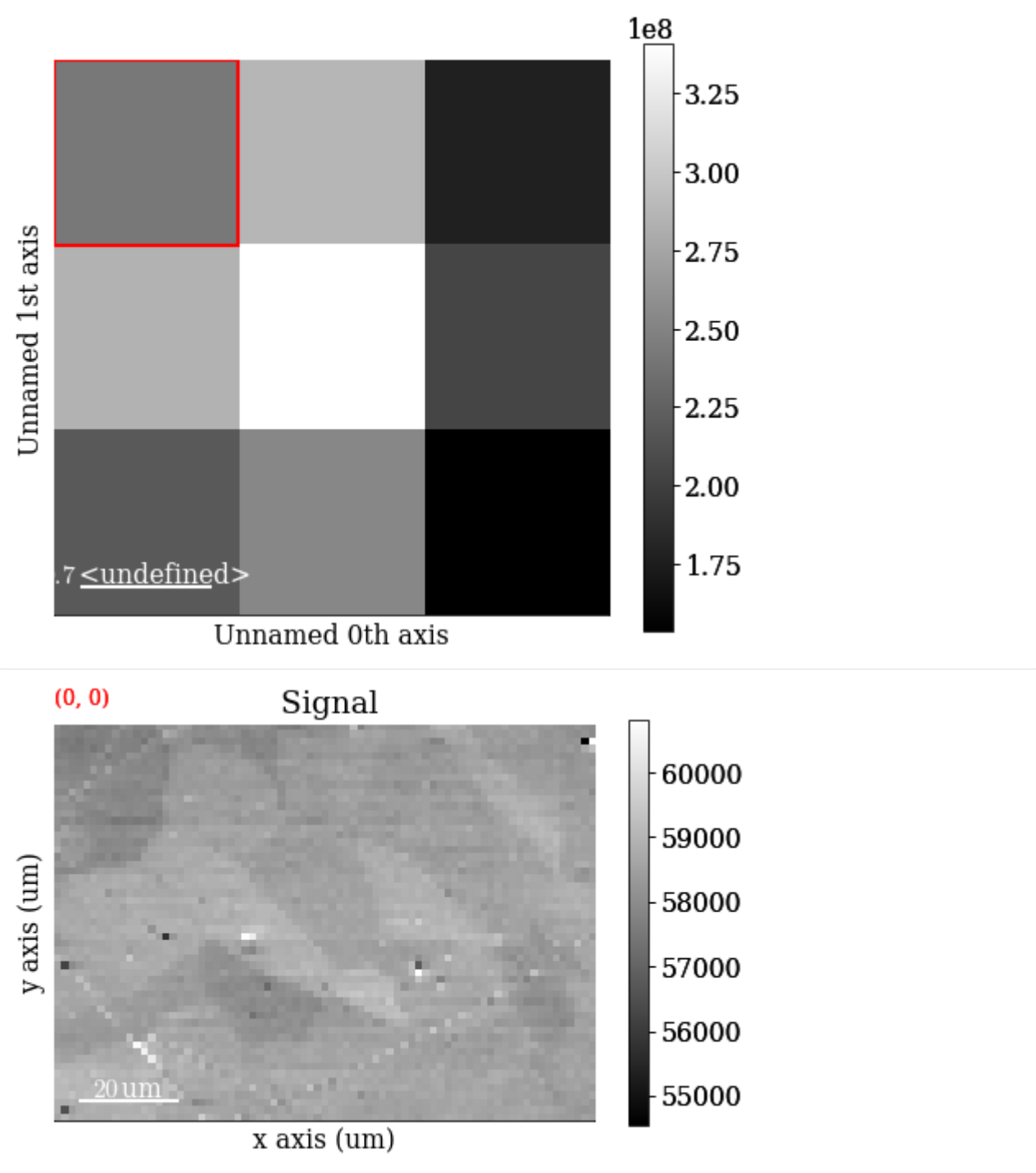

Get one VBSE image from the intensity within each grid tile with `get_images_from_grid()`

[10]:
```
vbse_imgs = vbse_imager.get_images_from_grid()
vbse_imgs
```

[10]:
```
<VirtualBSEImage, title: , dimensions: (3, 3|75, 55)>
```

Plot the images (one by one)

[11]:
```
vbse_imgs.plot()
```



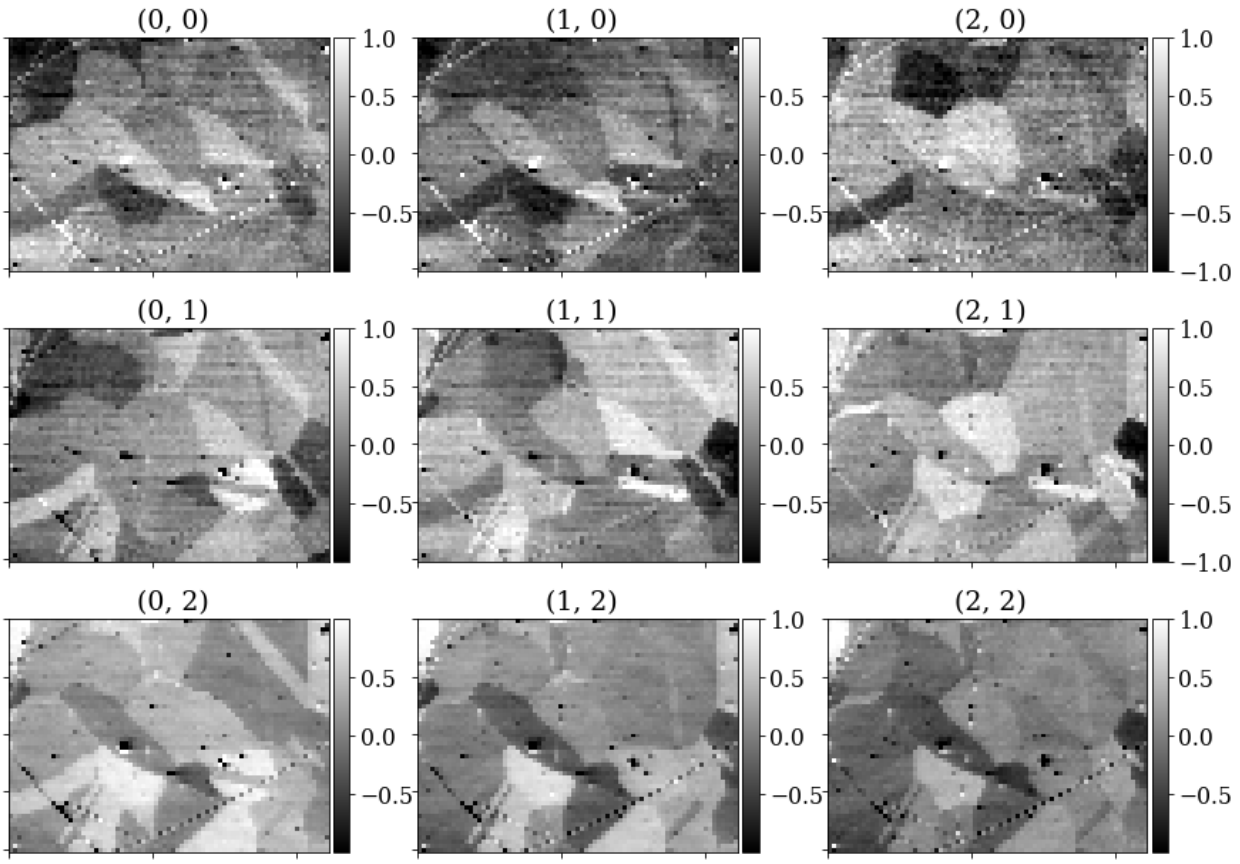

Strech the image contrast in each VBSE image by setting the darkest intensities to 0 and the highest intensities to 255 within the 0.5% percentiles, using `rescale_intensity()`

[12]:
```
vbse_imgs.rescale_intensity(percentiles=(0.5, 99.5))
```

```
[########################################] | 100% Completed | 101.17 ms
```

Replot the images after intensity rescaling in a nice image grid using HyperSpy's `plot_images()`

[13]:
```
fig = plt.figure(figsize=(14, 10))
_ = hs.plot.plot_images(vbse_imgs, fig=fig, axes_decor=None);
```




### RGB image

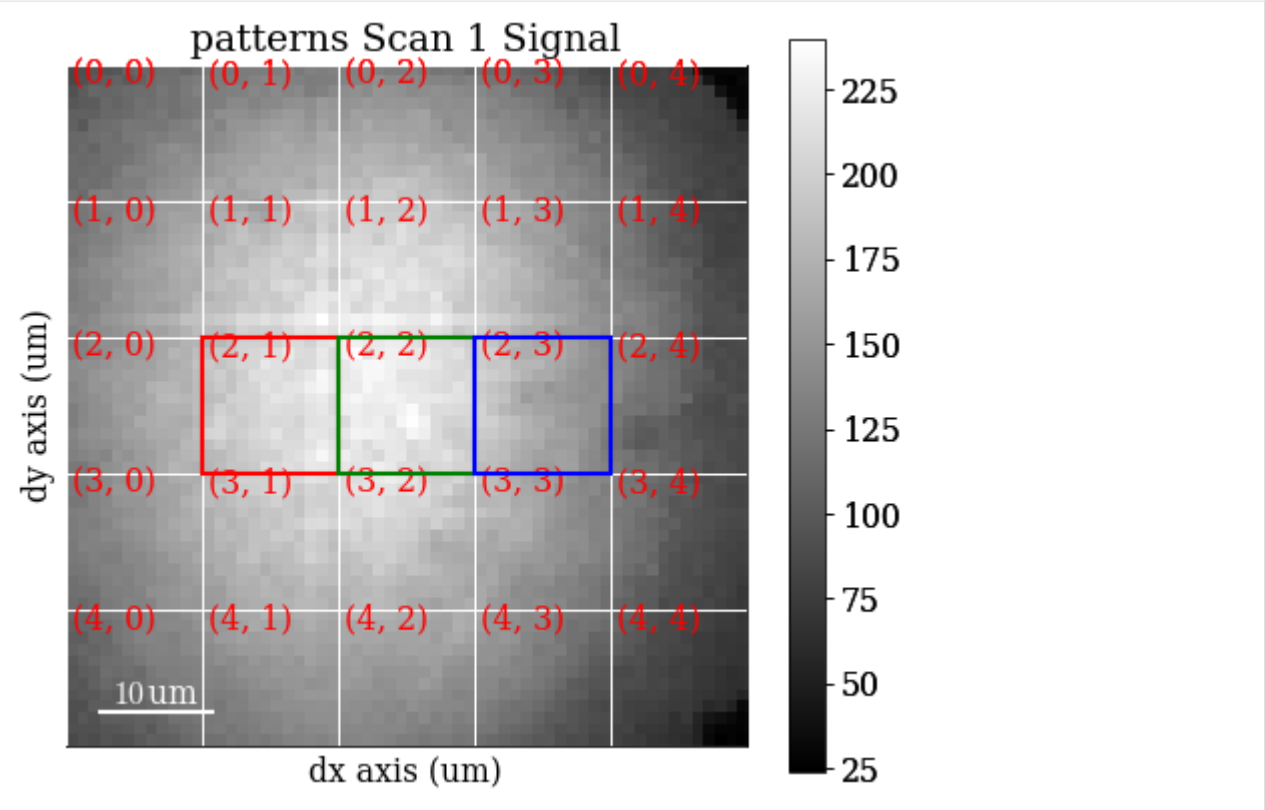
Separate the EBSD detector into a (5 x 5) grid

[14]:
```
vbse_imager.grid_shape = (5, 5)
```

Set some (can be more than one) of the grid tiles to be coloured red, green, or blue, and plot the color key

[15]:
```
rgb = [(2, 1), (2, 2), (2, 3)]
vbse_imager.plot_grid(rgb_channels=rgb)
```

[15]:
```
<EBSD, title: patterns Scan 1, dimensions: (|60, 60)>
```



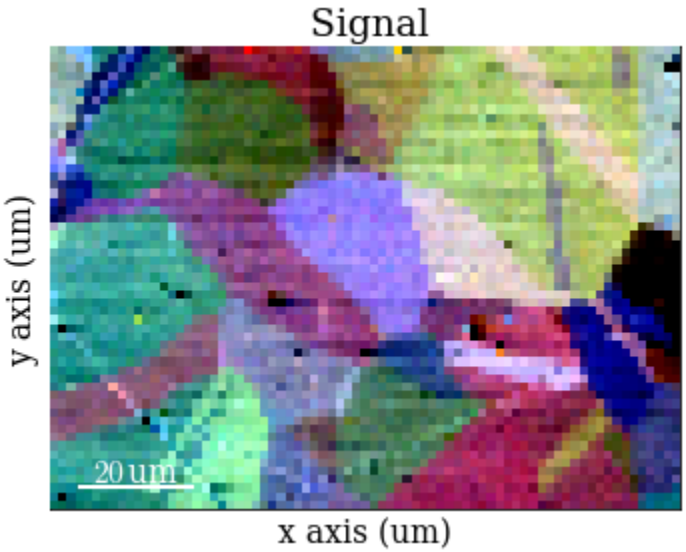

Create an RGB image from the specified grid tiles with `get_rgb_image()`

[16]:
```
vbse_rgb_img = vbse_imager.get_rgb_image(*rgb)
vbse_rgb_img
```

[16]:
```
<VirtualBSEImage, title: , dimensions: (|75, 55)>
```

Plot the resulting image

[17]:
```
vbse_rgb_img.plot()
```




### Process pattern intensities

The raw EBSD signal can be empirically evaluated as a superposition of a Kikuchi diffraction pattern and a smooth background intensity. For pattern indexing, the latter intensity is usually undesirable, while for VBSE imaging, as we saw above, this intensity can reveal topographical, compositional or diffraction contrast.

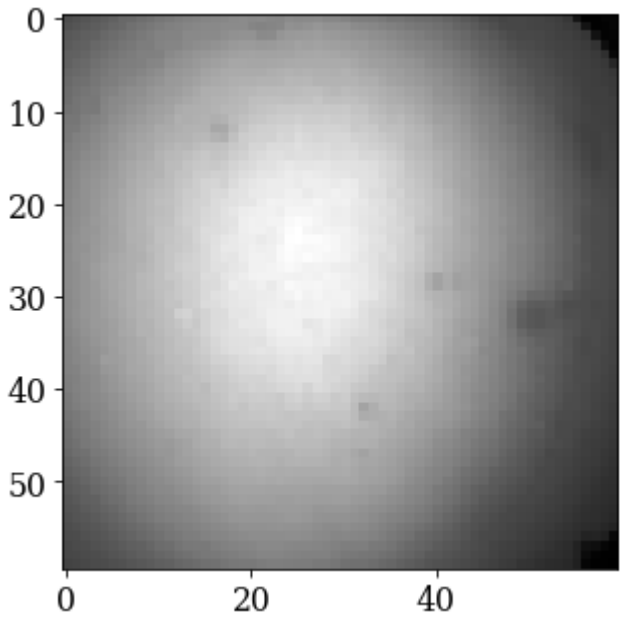
### Remove the static background

Effects which are constant, like hot pixels or dirt on the detector, can be removed by either subtracting or dividing by a static background. Ideally, this background pattern has no signal of interest.

A static background pattern was acquired with the Nickel EBSD data set, which was loaded with the data set into the signal metadata.

Retrieve this background from the `metadata` and plot it

[18]:
```
bg = s.static_background

fig, ax = plt.subplots()
_ = ax.imshow(bg, cmap="gray");
```

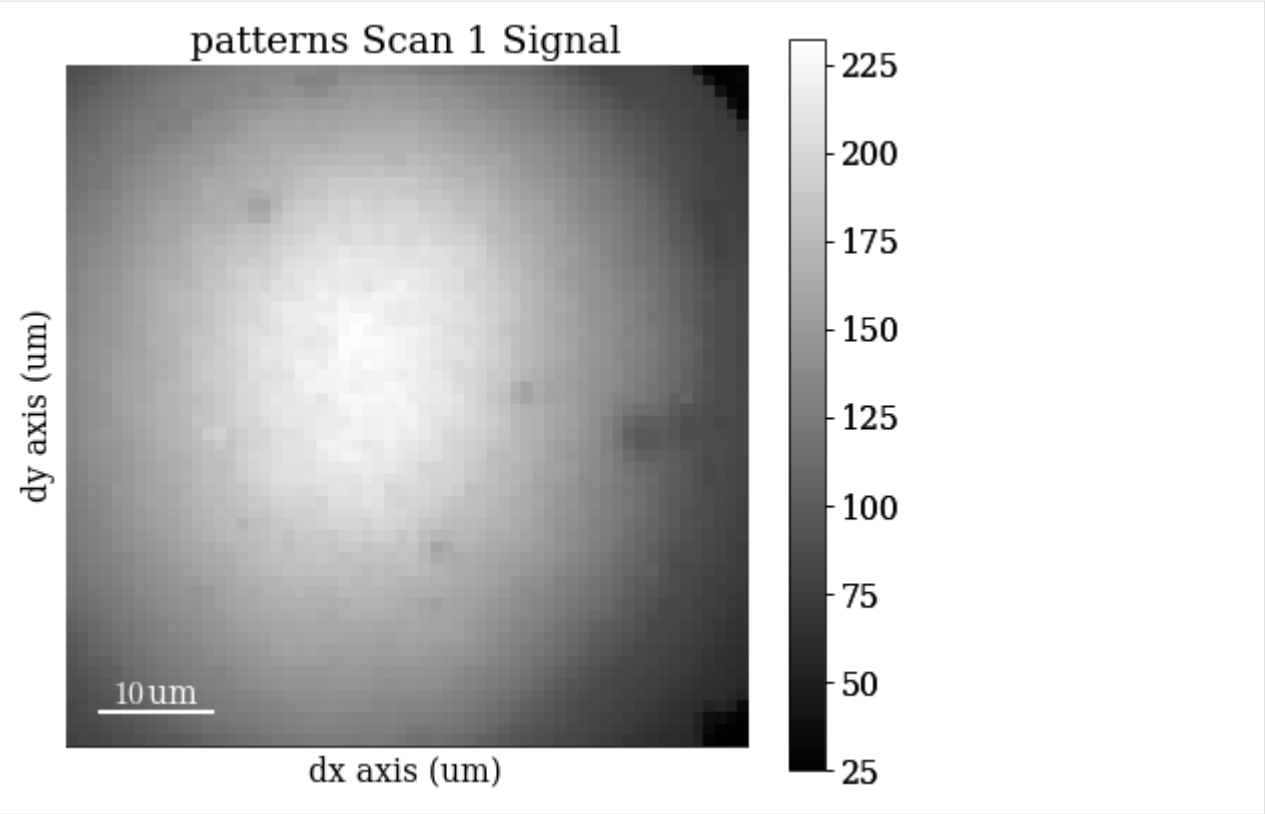



If one is not available, we can try to generate a suitable static background by averaging all patterns (and reverting the data type to 8-bit unsigned integers)

[19]:
```
bg2 = s.mean(axis=(0, 1))
bg2.change_dtype(s.data.dtype)
```

Compare it to the background from the metadata by plotting it

[20]:
```
bg2.plot()
```




Remove the static background with `remove_static_background()`

[21]:
```
s.remove_static_background()
```

```
[########################################] | 100% Completed | 101.29 ms
```

### Remove the dynamic background

Uneven intensity in a static background subtracted pattern can be corrected by subtracting or dividing by a dynamic background obtained by Gaussian blurring. A Gaussian window with a standard deviation set by `std` is used to blur each pattern individually (dynamic) either in the spatial or frequency domain. Blurring in the frequency domain is effectively accomplished by a low-pass Fast Fourier Transform (FFT) filter. The individual Gaussian blurred dynamic backgrounds are then subtracted or divided from the respective patterns.

Remove the dynamic background with `remove_dynamic_background()`

[22]:
```
s.remove_dynamic_background()
```

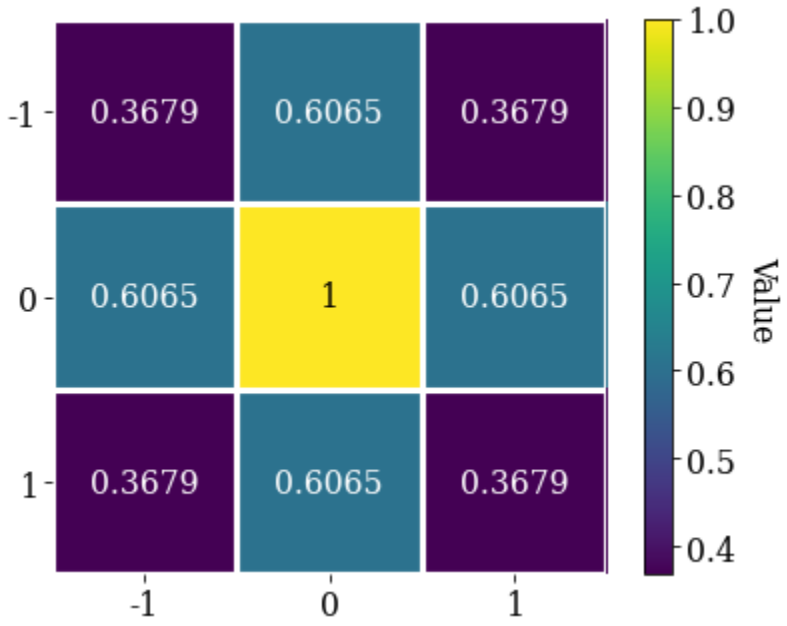
```
[########################################] | 100% Completed | 605.35 ms
```

### Average neighbour patterns

The signal-to-noise ratio in patterns can be improved by averaging patterns with their closest neighbours within a window/kernel/mask.

Let's average with all eight nearest neighbours, but use Gaussian weights with a standard deviation of 1. Create the Gaussian `filters.Window` and plot it

[23]:
```
w = kp.filters.Window(window="gaussian", std=1)
w.plot()
```




Average all patterns with their neighbour patterns using the Gaussian window with `average_neighbour_patterns()`

[24]:
```
s.average_neighbour_patterns(window=w)
```

```
[########################################] | 100% Completed | 303.36 ms
```

We can subsequently save these patterns to `kikuchipy`'s own h5ebsd specification [Jackson *et al.*, 2014] for the general format). This format can be read back into kikuchipy, or as a file in the `EMEBSD` format in the powerful suite of EMsoft command line programs.

[25]:
```
# s.save("pattern_static_dynamic_averaged.h5")
```

Note that neighbour pattern averaging increases the virtual interaction volume of the electron beam with the sample, leading to a potential loss in spatial resolution. Averaging may in some cases, like on grain boundaries, mix two or more different diffraction patterns, which might be unwanted. See [Wright *et al.*, 2015] for a discussion of this concern.

### Pre-indexing maps

The image quality metric $Q$ presented by [Lassen, 1994] relies on the assumption that the sharper the Kikuchi bands, the greater the high frequency content of the FFT power spectrum, and thus the closer $Q$ will be to unity. It can from this be expected that grain interiors will show a high $Q$, while grain boundaries will show a lower $Q$.

Get the image quality map with `get_image_quality()`

[26]:
```
maps_iq = s.get_image_quality()
```

```
[########################################] | 100% Completed | 303.16 ms
```

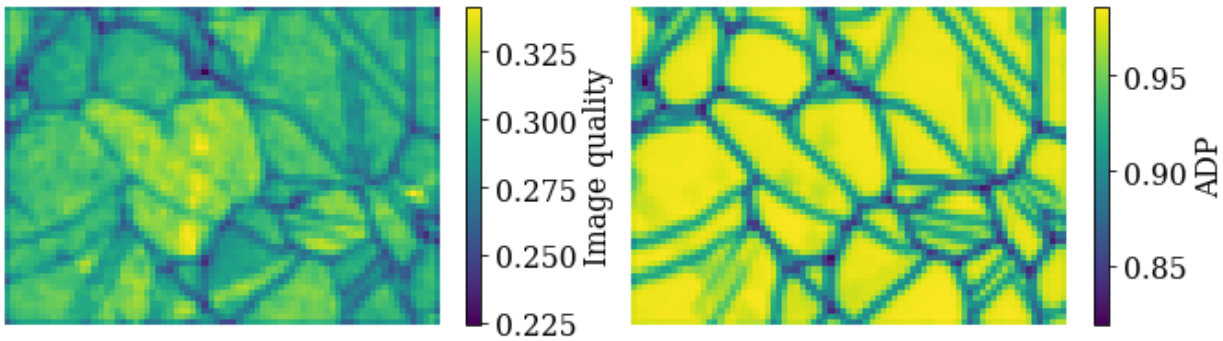
We can also produce a map showing how similar each pattern is to their four nearest neighbour (or any other number of neighbours specified by a binary mask). Get the average neighbour dot product map with `get_average_neighbour_dot_product_map()`

[27]:
```
maps_adp = s.get_average_neighbour_dot_product_map()
```

```
[########################################] | 100% Completed | 1.01 ss
```

Let's plot them side by side with colorbars using `Matplotlib`

[28]:
```
fig, axes = plt.subplots(ncols=2, figsize=(11, 3))

im0 = axes[0].imshow(maps_iq)
im1 = axes[1].imshow(maps_adp)
fig.colorbar(im0, ax=axes[0], label="Image quality")
fig.colorbar(im1, ax=axes[1], label="ADP")
for ax in axes:
    ax.axis("off")

fig.subplots_adjust(wspace=0.15)
```




### Dictionary indexing

Now we're ready to set up and run dictionary indexing of the background corrected and averaged patterns.

### Load master pattern

Before we can generate a dictionary of simulated patterns, we need a dynamically simulated master pattern containing all possible scattering vectors for a candidate phase. This can be simulated using `EMsoft` ([Callahan and De Graef, 2013]) and subsequently imported into kikuchipy using `kikuchipy.load()`.

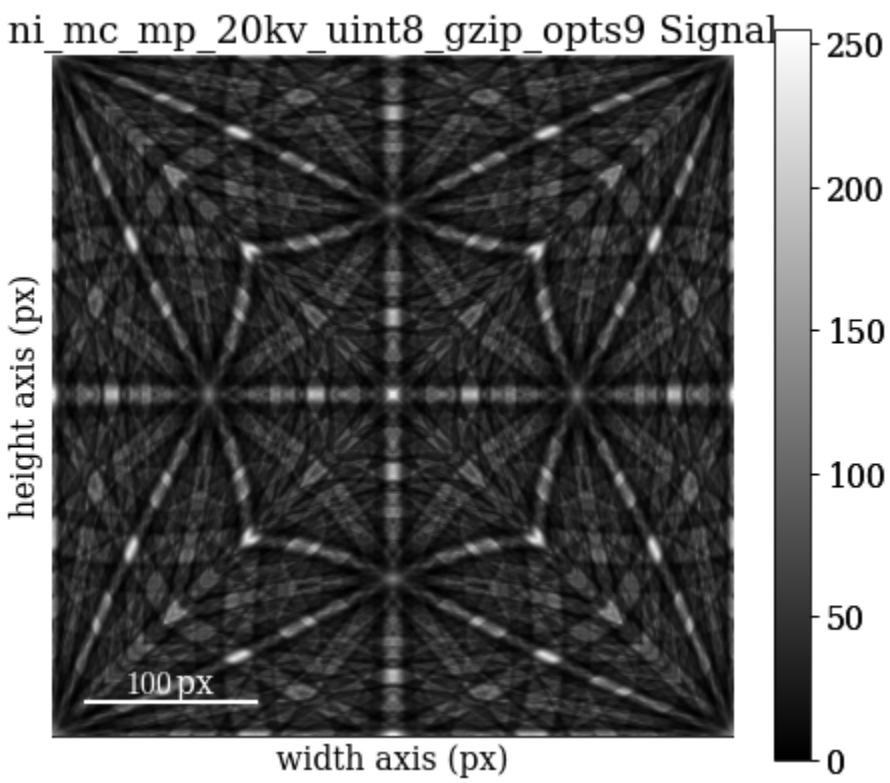
For demonstration purposes, we've included small (401 x 401) master patterns of Nickel in the stereographic and Lambert (square) projections as part of the `kikuchipy.data` module. Load the 20 keV master pattern from the northern hemisphere in the Lambert projection

[29]:
```
mp = kp.data.nickel_ebsd_master_pattern_small(
    projection="lambert", energy=20
)
mp
```

[29]:
```
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (|401, 401)>
```

Plot the master pattern

[30]:
```
mp.plot()
```




Extract `phase` information loaded with the master pattern

[31]:
```
phase = mp.phase
phase
```

[31]:
```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
```

Inspect it's crystal `structure` (list of asymmetric atom positions and a `structure.lattice`)

[32]:
```
phase.structure
```

[32]:
```
[28   0.000000 0.000000 0.000000 1.0000]
```

[33]:
```
phase.structure.lattice
```

[33]:
```
Lattice(a=0.35236, b=0.35236, c=0.35236, alpha=90, beta=90, gamma=90)
```

### Sample orientation space

Here we produce a sampling of the Rodriguez Fundamental Zone (RFZ) of point group $m\bar{3}m$ using a "characteristic distance" or "resolution" of 4°, as implemented in orix. This resolution is quite coarse, and used here because of time and memory constraints. The creators of EMsoft (see the aforementioned tutorial article by Jackson et al.) suggest using a smaller resolution of about 1.5° for experimental work.

Sample the RFZ for the Ni phase space group with a resolution of 4° using `orix.sampling.get_sample_fundamental()` and inspect the results

[34]:
```
Gr = sampling.get_sample_fundamental(
    method="cubochoric", resolution=4, point_group=phase.point_group
)
Gr
```

[34]:
```
Rotation (11935,)
[[ 0.8606 -0.3337 -0.3337 -0.1912]
 [ 0.8606 -0.3397 -0.3397 -0.1687]
 [ 0.8606 -0.345  -0.345  -0.1456]
 ...
 [ 0.8606  0.345   0.345   0.1456]
 [ 0.8606  0.3397  0.3397  0.1687]
 [ 0.8606  0.3337  0.3337  0.1912]]
```

### Define the sample-detector geometry

Now that we have our master pattern and crystal orientations, we need to describe the EBSD detector's position with respect to the sample. This ensures that projecting parts of the master pattern onto our detector yields dynamically simulated patterns presumably resembling our experimental ones. The average projection/pattern center (PC) for this experiment was determined by indexing five calibration patterns using the EDAX TSL Data Collection v7 software, and is ($x^*$, $y^*$, $z^*$) = (0.4210, 0.7794, 0.5049).

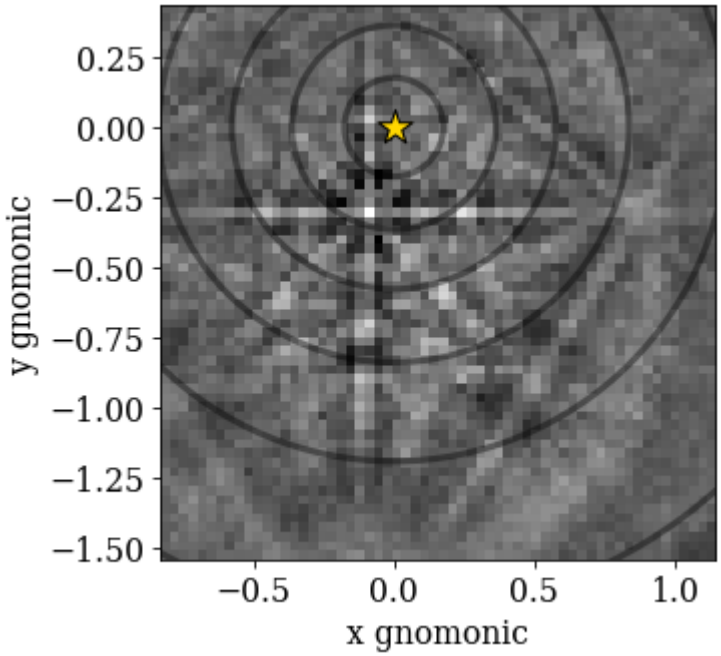
Create the `detector.EBSDDetector` and inspect it

[35]:
```
det = kp.detectors.EBSDDetector(
    shape=s.axes_manager.signal_shape[::-1],
    sample_tilt=70,
    pc=(0.421, 0.2206, 0.5049),
)
det
```

[35]:
```
EBSDDetector (60, 60), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.421, 0.221, 0.
↪505)
```

Let's double check the projection/pattern center (PC) position on the detector by plotting it in gnomonic coordinates and showing the gnomonic circles at 10° steps

[36]:
```
det.plot(
    coordinates="gnomonic",
    pattern=s.inav[0, 0].data,
    draw_gnomonic_circles=True,
)
```




### Generate dictionary of simulated patterns

Now we're ready to generate our dictionary of simulated patterns by projecting parts of the master pattern onto our detector for all sampled orientations. The method assumes the crystal orientations are represented with respect to the EDAX TSL sample reference frame RD-TD-ND. For more details, see the *reference frame tutorial*.

So, generate a dictionary of simulated patterns using `MasterPattern.get_patterns()`

[37]:
```
dynsim = mp.get_patterns(
    rotations=Gr,
    detector=det,
    energy=20,
    dtype_out=s.data.dtype,
    compute=True,
)
dynsim
```

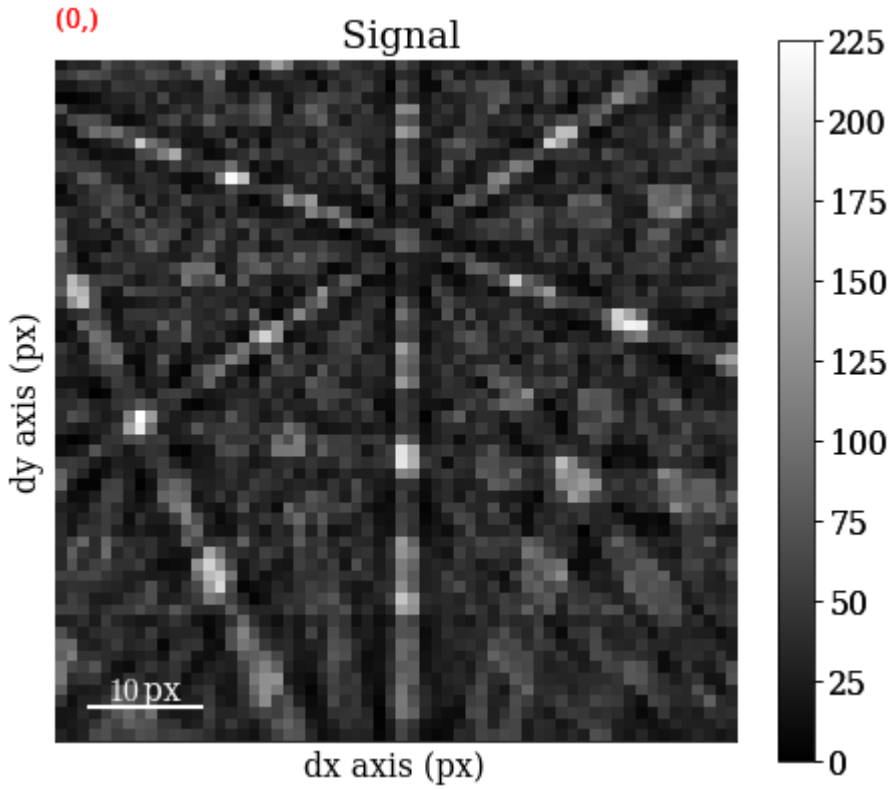
```
[########################################] | 100% Completed | 2.01 ss
```

[37]:
```
<EBSD, title: , dimensions: (11935|60, 60)>
```

We've now generated the dictionary and read it into memory. We could instead have passed `compute=False`, which would have returned a `LazyEBSD` to be computed during the indexing run. This can sometimes be desirable.

Let's inspect a few of the simulated patterns to ensure they look alright by plotting them

[38]:
```
dynsim.plot(navigator=None)
```




### Perform dictionary indexing

Finally, let's match the simulated patterns to our experimental patterns, using the zero-mean normalized cross correlation (NCC) coefficient, which is the default similarity metric. We'll keep the 10 best matching orientations. A number of about 4125 * 12000 comparisons is quite small, which we can do in memory all at once. However, in cases where the number of comparisons are too big for our memory to handle, we can iterate over the dictionary to match only parts at a time. We'll use at least 20 iterations here.

Let's perform `dictionary_indexing()`

[39]:
```
xmap = s.dictionary_indexing(
    dictionary=dynsim,
    metric="ncc",
    keep_n=10,
    n_per_iteration=dynsim.axes_manager.navigation_size // 20,
)
```

```
Dictionary indexing information:
  Phase name: ni
  Matching 4125 experimental pattern(s) to 11935 dictionary pattern(s)
  NormalizedCrossCorrelationMetric: float32, greater is better, rechunk: False,␣
↪navigation mask: False, signal mask: False
```

```
100%|----------------------------------| 21/21 [00:05<00:00,  3.63it/s]
```

```
  Indexing speed: 712.22792 patterns/s, 8500440.21492 comparisons/s
```

Inspect the returned CrystalMap

[40]:
```
xmap
```

[40]:
```
Phase   Orientations  Name  Space group  Point group  Proper point group     Color
    0  4125 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: scores, simulation_indices
Scan unit: um
```

We can write the indexing results to file using one of orix' writers. orix' own HDF5 file format stores all results to in HDF5 file, while the .ang file writer only stores the best matching orientation

[41]:
```
# io.save("di_results_ni1.h5", xmap)
# io.save("di_results_ni1.ang", xmap)
```

### Validate indexing results

### Indexing maps

See e.g. the *ESTEEM3 workshop tutorial* for more details on analyzing indexing results. orix cannot reconstruct grains and analyze textures, so this has to be done in other software, like MTEX.

Here, we'll inspect the map of best matching scores and a so-called orientation similarity map, which compares the best matching orientations for each pattern to it's nearest neighbours.

Get the NCC map in the correct shape from the CrystalMap's `scores` property

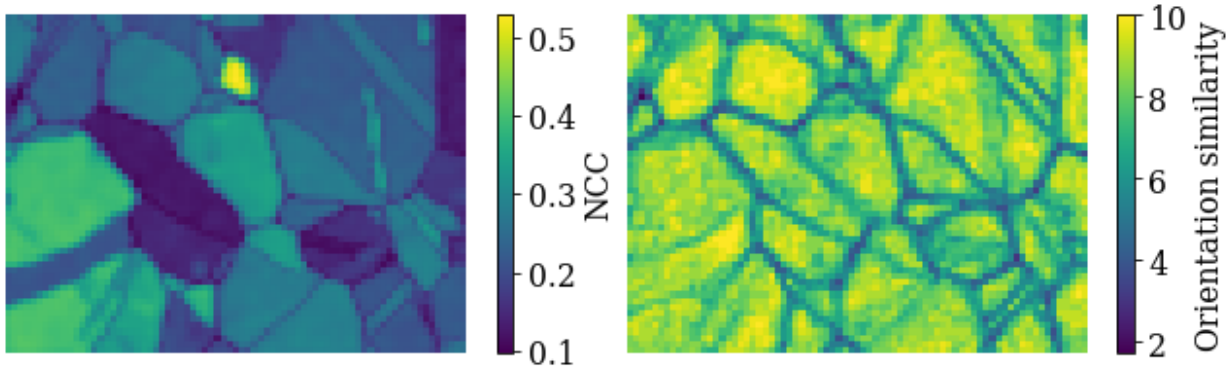
[42]:
```
maps_ncc = xmap.scores[:, 0].reshape(*xmap.shape)
```

Get the `indexing.orientation_similarity_map()` using the full list of 10 best matches per pattern/point

[43]:
```
maps_os = kp.indexing.orientation_similarity_map(xmap)
```

Plot the maps using `Matplotlib`

[44]:
```
fig, axes = plt.subplots(ncols=2, figsize=(11, 3))
im0 = axes[0].imshow(maps_ncc)
im1 = axes[1].imshow(maps_os)
fig.colorbar(im0, ax=axes[0], label="NCC")
fig.colorbar(im1, ax=axes[1], label="Orientation similarity")
for ax in axes:
    ax.axis("off")
fig.subplots_adjust(wspace=0)
```




### Compare to dynamical simulations

We can visually compare the experimental and best matching pattern side by side. First, we extract the best matching indices into the dictionary from the CrystalMap property `simulation_indices`

[45]:
```
best_sim_idx = xmap.simulation_indices[:, 0]
```

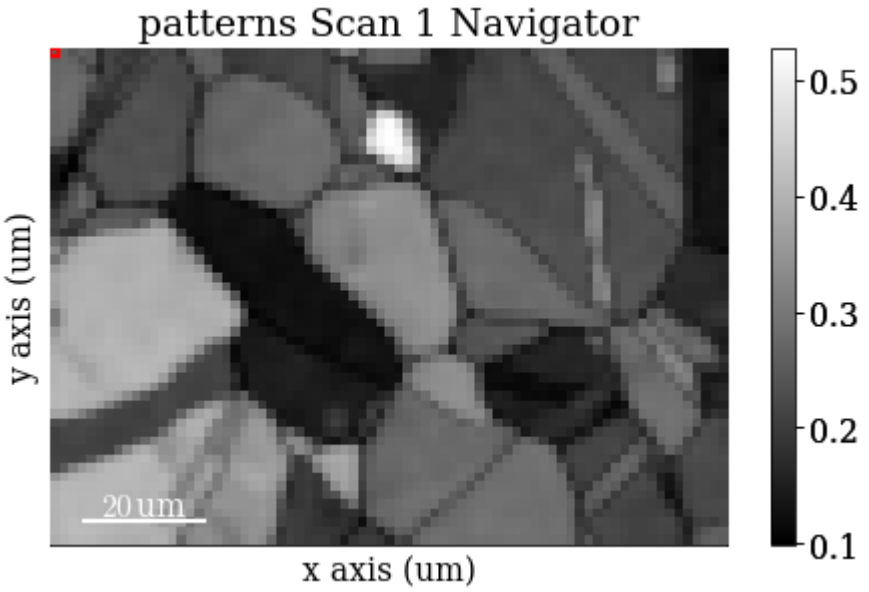
Then we extract the simulated patterns corresponding to the indices, reshape the array to the same shape as the experimental data, and create an EBSD signal from it

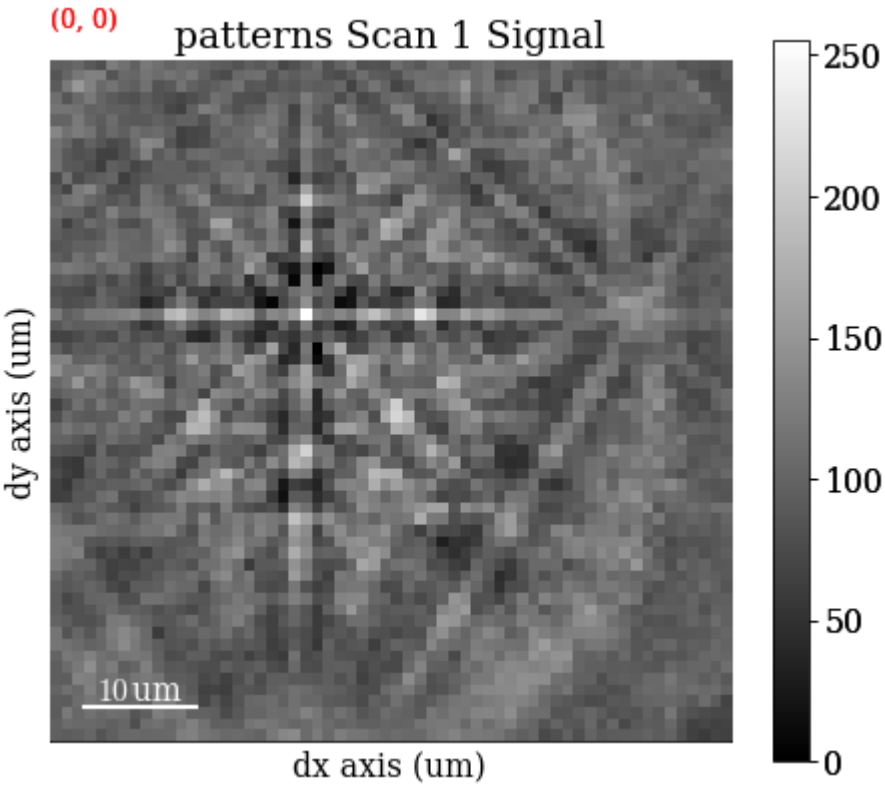
[46]:
```
best_patterns = dynsim.data[best_sim_idx].reshape(s.data.shape)
s_best = kp.signals.EBSD(best_patterns)
s_best
```

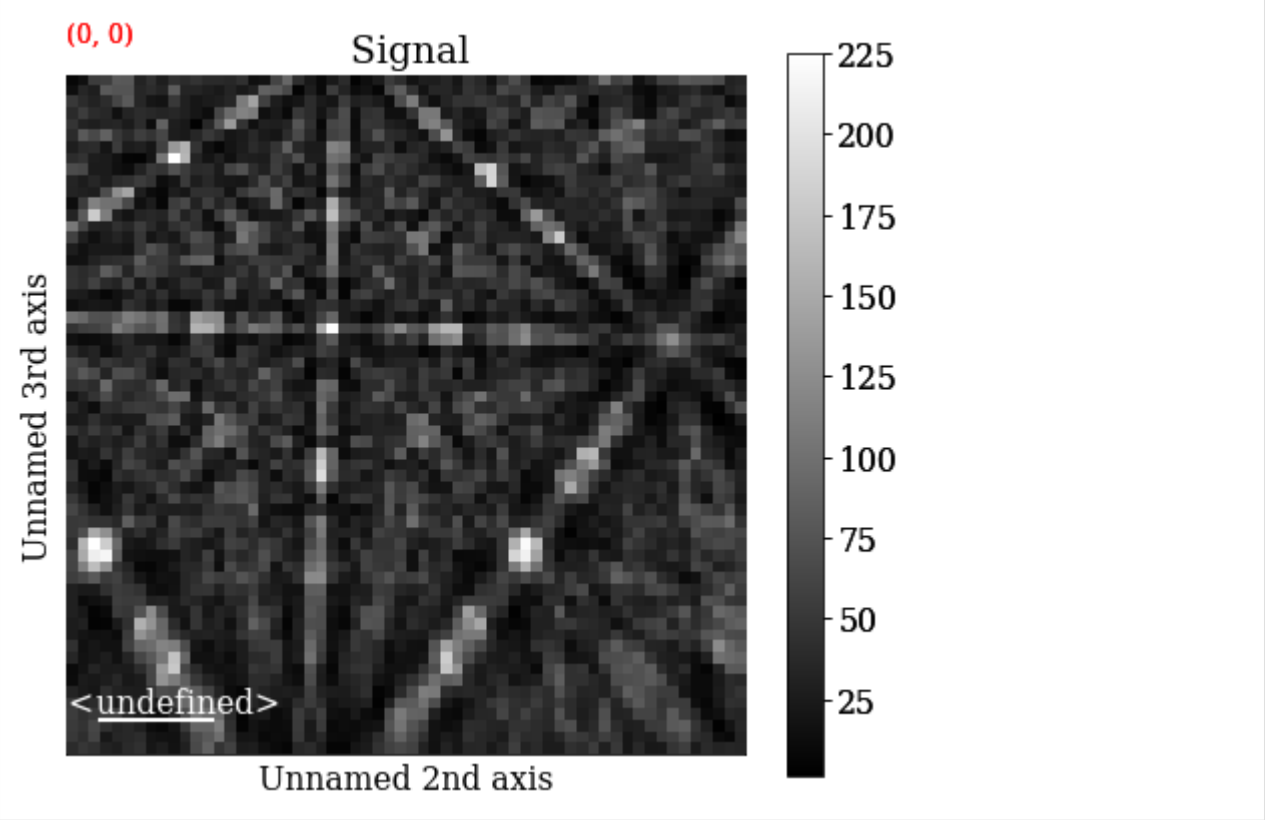
[46]:
```
<EBSD, title: , dimensions: (75, 55|60, 60)>
```

Plot the experimental and simulated patterns (this is not easily done via Binder…) side by side using HyperSpy's `plot_signals()`, navigating in the NCC map

[47]:
```
ncc_navigator = hs.signals.Signal2D(maps_ncc)
hs.plot.plot_signals([s, s_best], navigator=ncc_navigator)
```


patterns Scan 1 Navigator
y axis (um)
20 um
x axis (um)
0.5
0.4
0.3
0.2
0.1


(0, 0)
patterns Scan 1 Signal
dy axis (um)
10 um
dx axis (um)
250
200
150
100
50
0




### Compare to geometrical simulations

We can can also add bands and zone axes from the best matching orientations as markers to the experimental EBSD data. The simulations are based on the work by Aimo Winkelmann in the supplementary material to the excellent tutorial paper by [Britton *et al.*, 2016]. See also the *geometrical EBSD simulations tutorial* for more information than is given here.

First, we set up the relevant reflectors (the zone axes follows from these), namely $(hkl)$ = (111), (200), (220), and (311), using diffsims' `ReciprocalLatticeVector`

[48]:
```
ref = ReciprocalLatticeVector(
    phase=phase, hkl=[[1, 1, 1], [2, 0, 0], [2, 2, 0], [3, 1, 1]]
)
ref
```

[48]:
```
ReciprocalLatticeVector (4,), ni (m-3m)
[[1. 1. 1.]
 [2. 0. 0.]
 [2. 2. 0.]
 [3. 1. 1.]]
```

Get the symmetrically equivalent bands using `symmetrise()`

[49]:
```
ref2 = ref.symmetrise()
```

Create a `KikuchiPatternSimulator` (remember to reshape the best matching rotations array to the experimental data shape!)

[50]:
```
simulator = kp.simulations.KikuchiPatternSimulator(ref2)
simulator.reflectors.size
```

[50]:
```
50
```

Generate bands and zone axes visible on the detector for the best matching orientations using `on_detector()`

[51]:
```
sim = simulator.on_detector(det, xmap.rotations[:, 0].reshape(*xmap.shape))
sim
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 102.35 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 101.32 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 102.49 ms
```

[51]:
```
GeometricalKikuchiPatternSimulation (55, 75):
ReciprocalLatticeVector (44,), ni (m-3m)
[[ 1.  1.  1.]
 [-1.  1.  1.]
 [-1. -1.  1.]
 [ 1. -1.  1.]
 [ 1. -1. -1.]
 [ 1.  1. -1.]
 [-1. -1. -1.]
 [ 2.  0.  0.]
 [ 0.  2.  0.]
 [-2.  0.  0.]
 [ 0. -2.  0.]
 [ 0.  0.  2.]
 [ 2.  2.  0.]
 [-2.  2.  0.]
 [-2. -2.  0.]
 [ 2. -2.  0.]
 [ 0.  2.  2.]
 [-2.  0.  2.]
 [ 0. -2.  2.]
 [ 2.  0.  2.]
 [ 0.  2. -2.]
 [ 0. -2. -2.]
 [ 2.  0. -2.]
 [ 3.  1.  1.]
 [-1.  3.  1.]
 [-3. -1.  1.]
 [ 1. -3.  1.]
 [ 1.  3.  1.]
 [-3.  1.  1.]
 [-1. -3.  1.]
 [ 3. -1.  1.]
 [ 3. -1. -1.]
 [ 1.  3. -1.]
 [-3.  1. -1.]
 [-1. -3. -1.]
```

(continues on next page)

(continued from previous page)

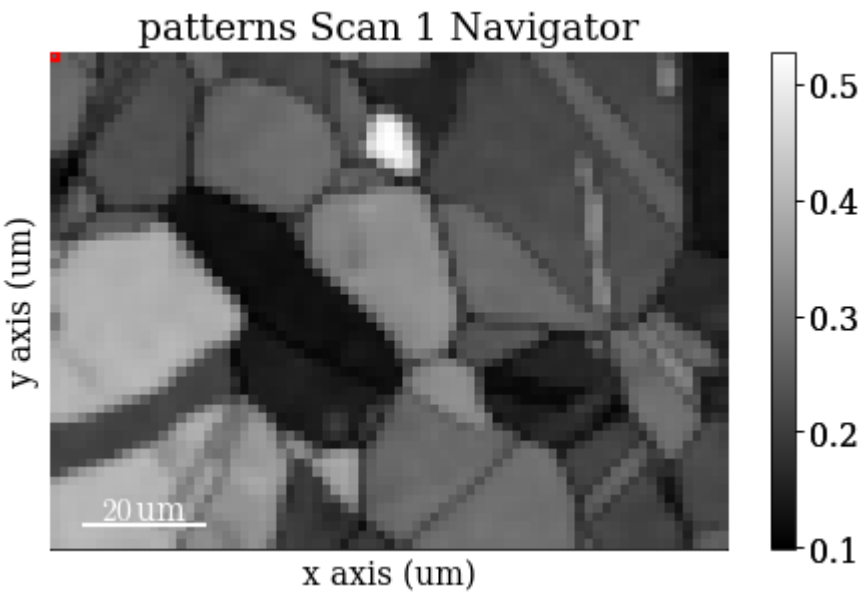
```
[-1.  3. -1.]
[-3. -1. -1.]
[ 1. -3. -1.]
[ 3.  1. -1.]
[ 1. -1.  3.]
[ 1.  1.  3.]
[-1.  1.  3.]
[-1. -1.  3.]
[ 1.  1. -3.]]
```

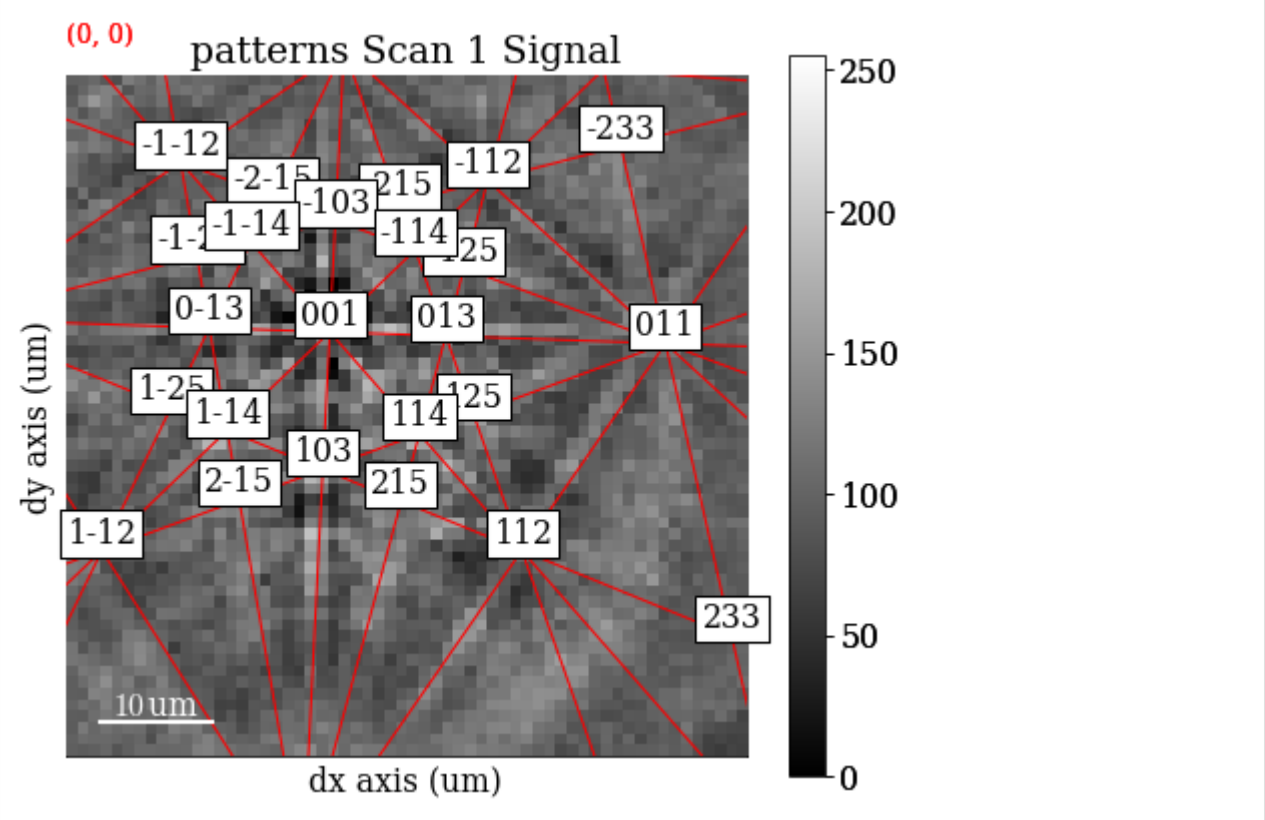
Use `as_markers()` to make HyperSpy markers from the geometrical simulations and add them to the experimental patterns with `add_marker()`

[52]:
```
markers = sim.as_markers(zone_axes_labels=True)
# del s.metadata.Markers
s.add_marker(markers, plot_marker=False, permanent=True)
```

Plot the experimental data, navigating in the NCC map

[53]:
```
s.plot(navigator=ncc_navigator)
```






**Live notebook**

You can run this notebook in a live session launch binder or view it on Github.

## ESTEEM3 workshop

**Note**

This tutorial was given at the ESTEEM3 2022 workshop entitled Electron diffraction for solving engineering problems, held at NTNU in Trondheim, Norway in June 2022.

The tutorial has been updated to work with the current release of kikuchipy and to fit our documentation format. The dataset is available from Zenodo at [Ånes *et al.*, 2019], and is scan number 7 (17 dB) out of the series of 10 (22 dB) scans taken with increasing gain (0-22 dB).

In this tutorial we will inspect a small (100 MB) EBSD data of polycrystalline recrystallized nickel by (Hough and dictionary) indexing and inspect the results using geometrical EBSD simulations.

Steps:

1. Data overview (virtual backscatter electron imaging)
2. Enhance Kikuchi pattern (background correction)
3. Data overview ("feature maps")
4. Hough indexing

5. Verification of results (geometrical simulations)
6. Dictionary indexing
7. Orientation refinement
8. Summary

Tools:

- kikuchipy: https://kikuchipy.org
- HyperSpy: https://hyperspy.org
- PyEBSDIndex: https://pyebsdindex.readthedocs.io
- orix: https://orix.readthedocs.io
- diffsims: https://diffsims.readthedocs.io
- EMsoft (indirectly): https://github.com/EMsoft-org/EMsoft

Import libraries (replace `inline` with `qt5` plotting backend for interactive plots in separate windows)

[1]:
```
%matplotlib inline

from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np

from diffsims.crystallography import ReciprocalLatticeVector
import hyperspy.api as hs
import kikuchipy as kp
from orix.crystal_map import CrystalMap, Phase, PhaseList
from orix.quaternion import Orientation, symmetry
from orix import io, plot, sampling
from orix.vector import Vector3d
from pyebsdindex import ebsd_index, pcopt


plt.rcParams.update(
    {"figure.facecolor": "w", "font.size": 15, "figure.dpi": 75}
)

data_path = Path("../../../kikuchipy_test/esteem")
```

Load data

[2]:
```
s = kp.data.ni_gain(7, allow_download=True)  # External download
```

[3]:
```
s
```

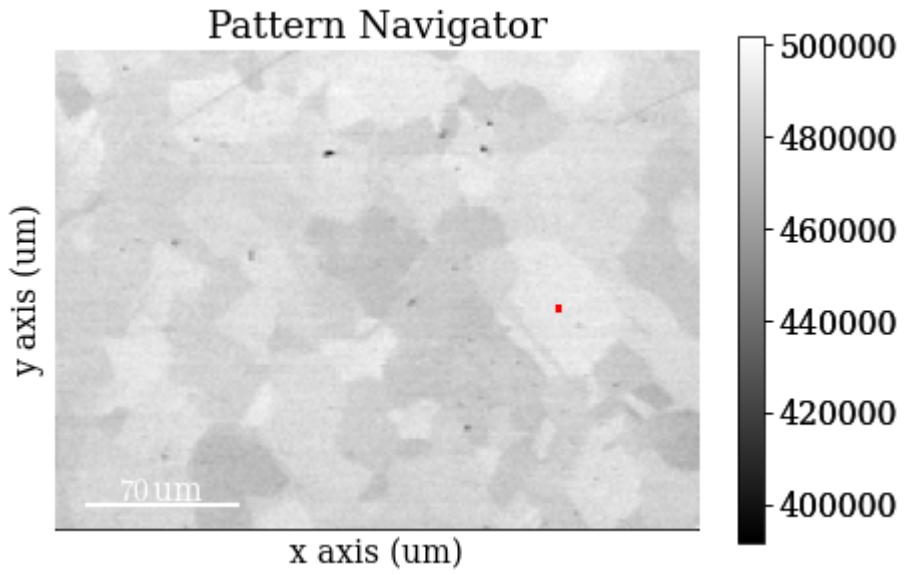
[3]:
```
<EBSD, title: Pattern, dimensions: (200, 149|60, 60)>
```

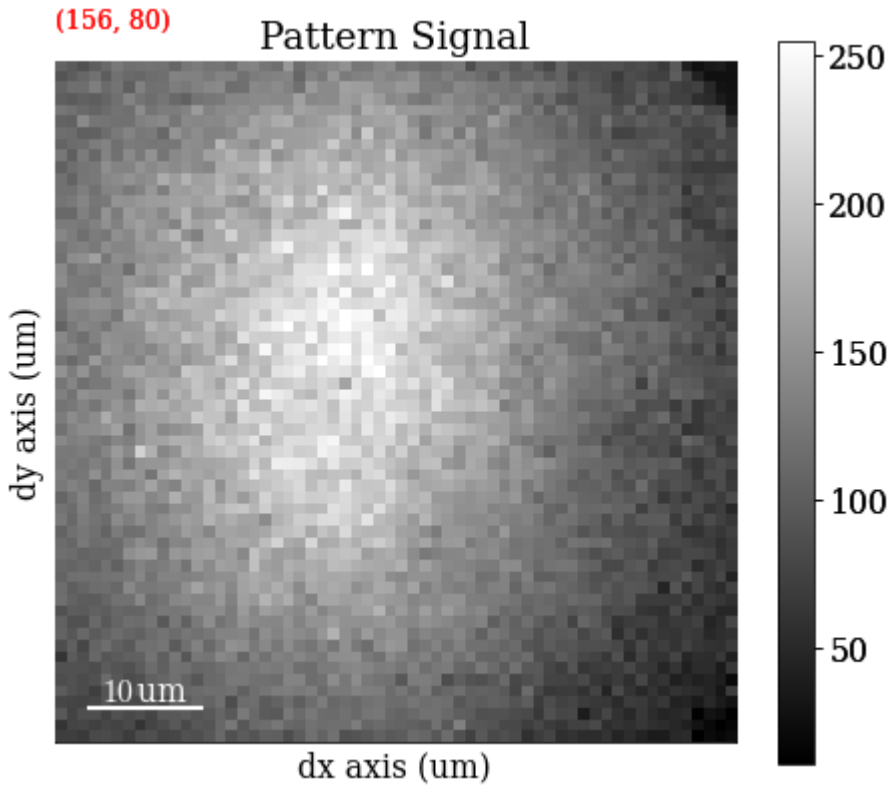
Plot a pattern from a particular grain to show improvement in signal-to-noise ratio

[4]:
```
s.axes_manager.indices = (156, 80)
```

Plot data

[5]:
```
s.plot()
```


Pattern Navigator
y axis (um)
70 um
x axis (um)
500000
480000
460000
440000
420000
400000


(156, 80)
Pattern Signal
dy axis (um)
10 um
dx axis (um)
250
200
150
100
50

### Data overview - virtual imaging

Mean intensity in each pattern

[6]:
```
s_mean = s.mean(axis=(2, 3))
s_mean
```

[6]:
```
<BaseSignal, title: Pattern, dimensions: (200, 149|)>
```

Save map

[7]:
```
# plt.imsave(data_path /  "maps_mean.png", s_mean.data, cmap="gray")
```

Set up VBSE image generator

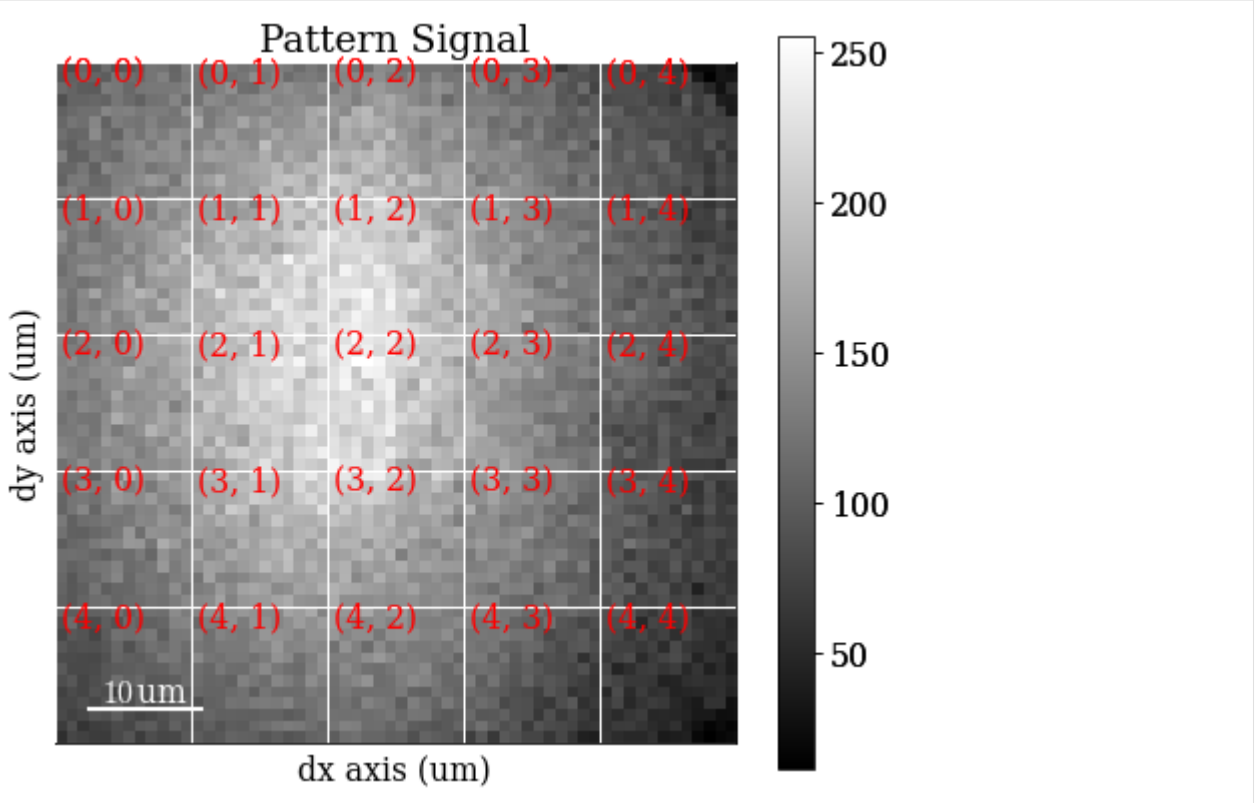
[8]:
```
vbse_imager = kp.imaging.VirtualBSEImager(s)
vbse_imager
```

[8]:
```
VirtualBSEImager for <EBSD, title: Pattern, dimensions: (200, 149|60, 60)>
```

Plot VBSE grid

[9]:
```
vbse_imager.plot_grid()
```

[9]:
```
<EBSD, title: Pattern, dimensions: (|60, 60)>
```



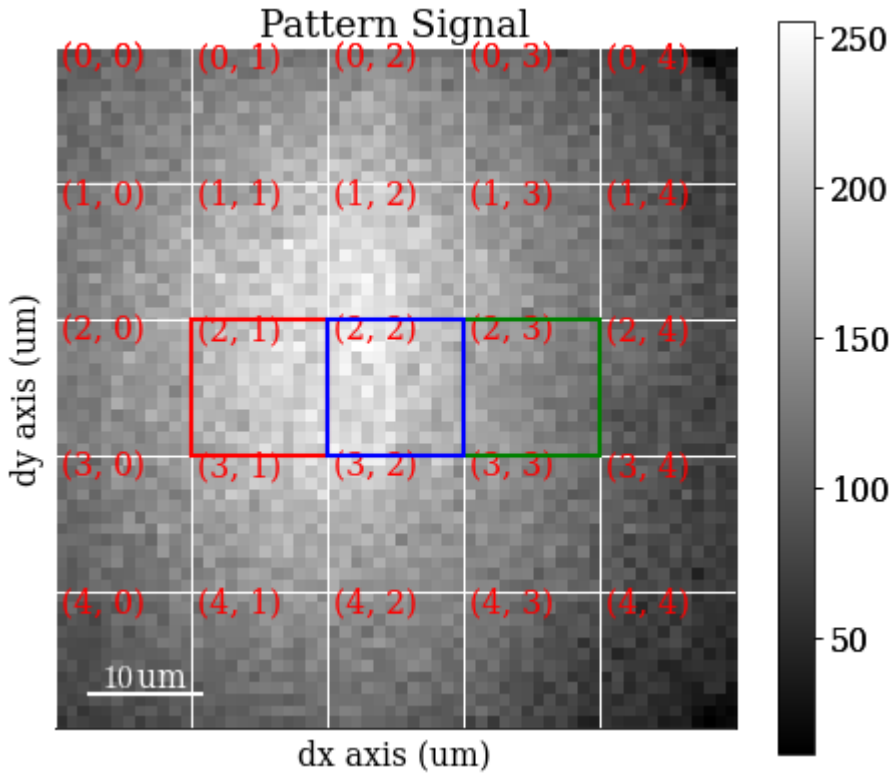

Specify RGB colour channels

[10]:
```
r = (2, 1)
b = (2, 2)
g = (2, 3)
```

Plot coloured grid tiles

[11]:
```
vbse_imager.plot_grid(rgb_channels=[r, g, b])
```

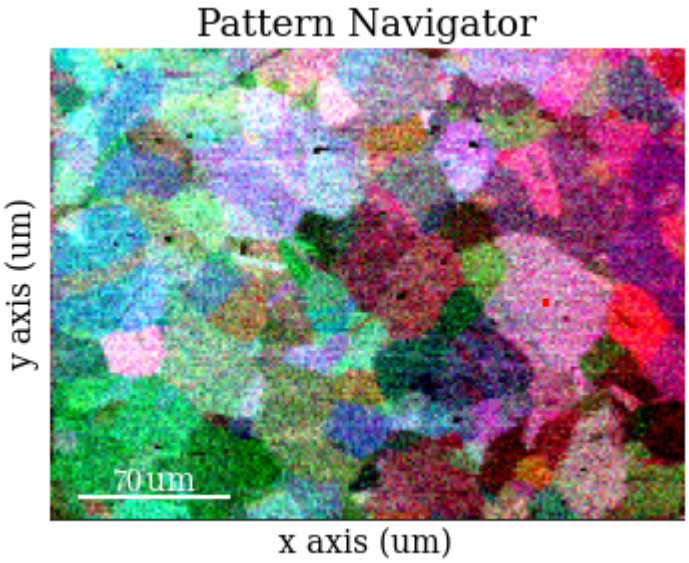
[11]:
```
<EBSD, title: Pattern, dimensions: (|60, 60)>
```

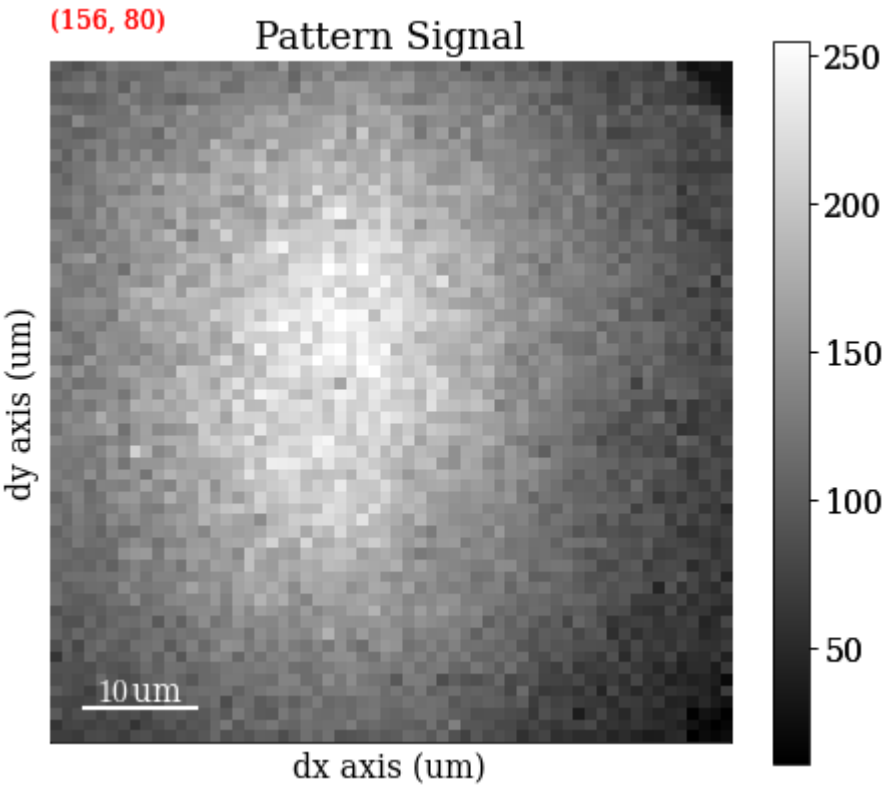



Get VBSE RGB image

[12]:
```
vbse_rgb = vbse_imager.get_rgb_image(r, g, b)
```

[13]:
```
s.plot(vbse_rgb)
```




Save RGB image

[14]:
```
# vbse_rgb.save(data_path / "maps_vbse_rgb.png")
```

### Enhance Kikuchi pattern

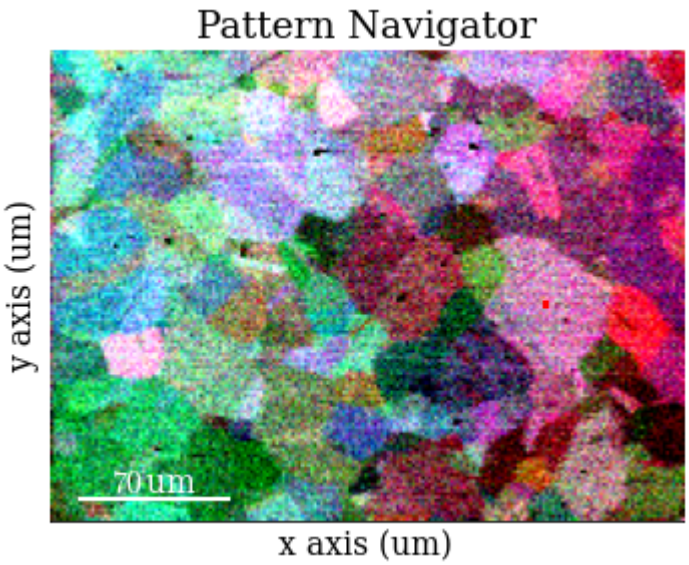
Remove static (constant) background

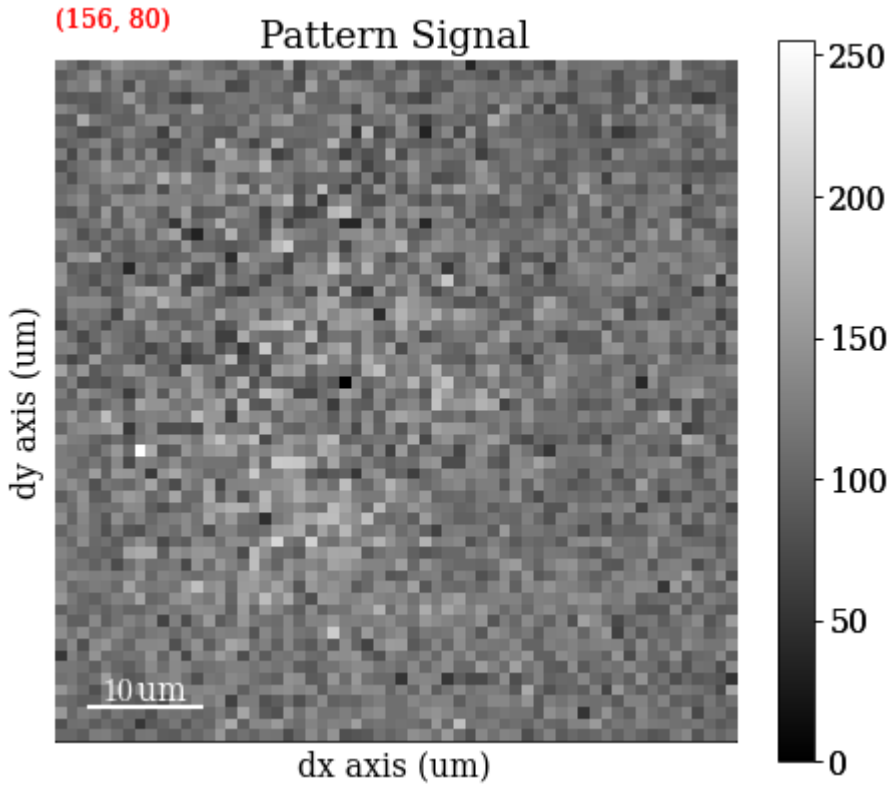
[15]:
```
s.remove_static_background()
```

```
[########################################] | 100% Completed | 705.70 ms
```

Inspect statically corrected patterns

[16]:
```
s.plot(vbse_rgb)
```





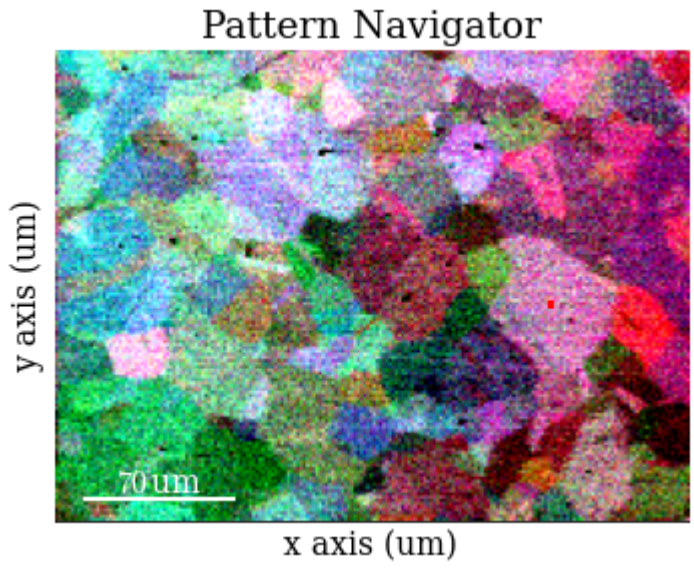

Remove dynamic (per pattern) background

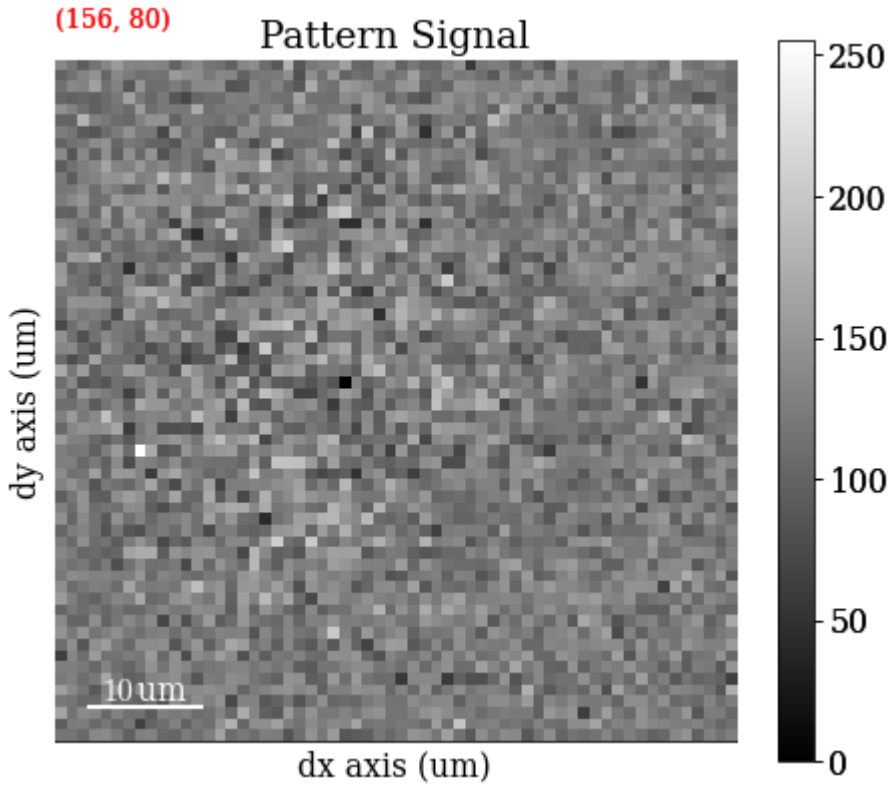
[17]:
```
s.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 2.92 ss
```

Inspect dynamically corrected patterns

[18]:
```
s.plot(vbse_rgb)
```





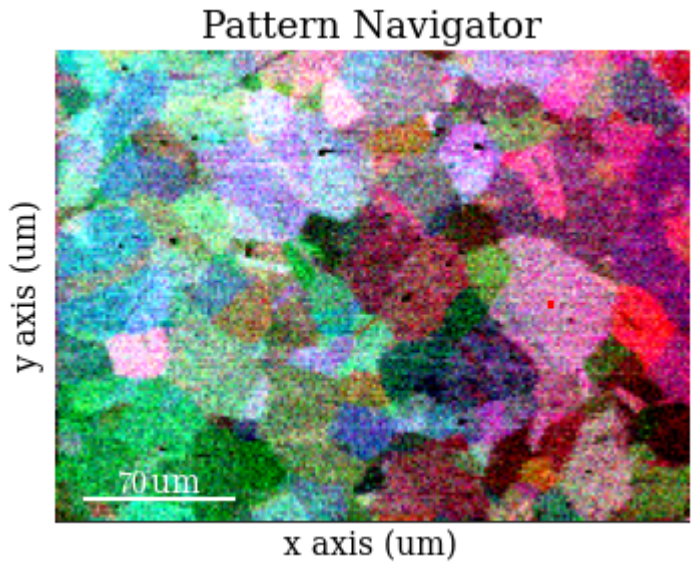

Average each patterns with the four nearest neighbours

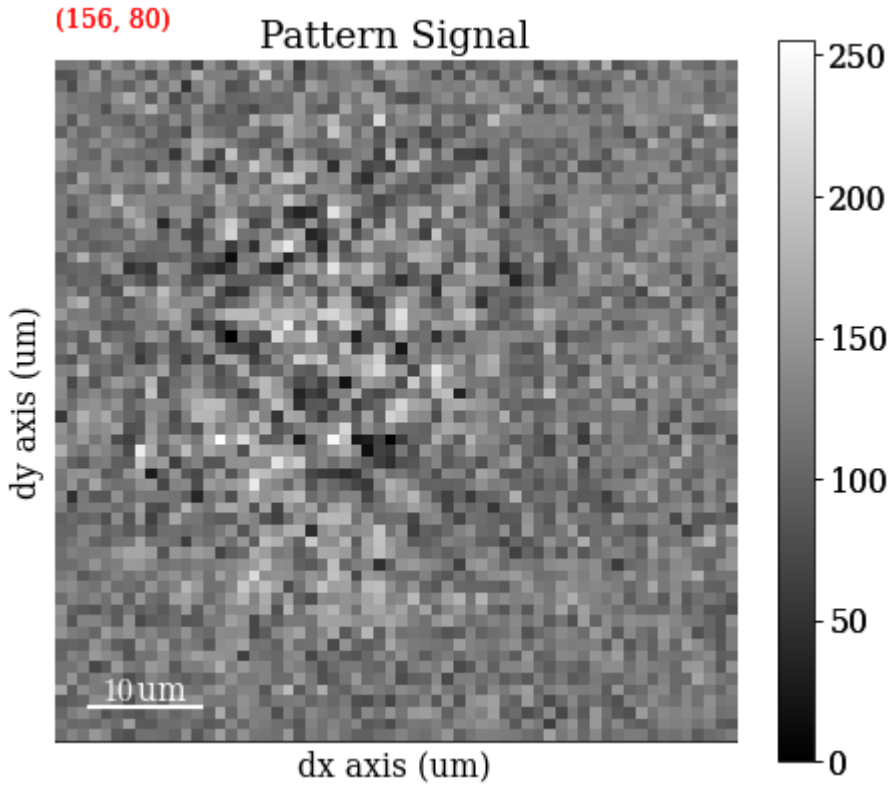
[19]:
```
s.average_neighbour_patterns()
```

```
[########################################] | 100% Completed | 1.11 sms
```

Inspect average patterns

[20]:
```
s.plot(vbse_rgb)
```






Save corrected patterns

[21]:
```
# s.save(data_path / "patterns_sda.h5")
```

### Data overview - feature maps

Get image quality $Q$ map (not image quality IQ from Hough indexing!)

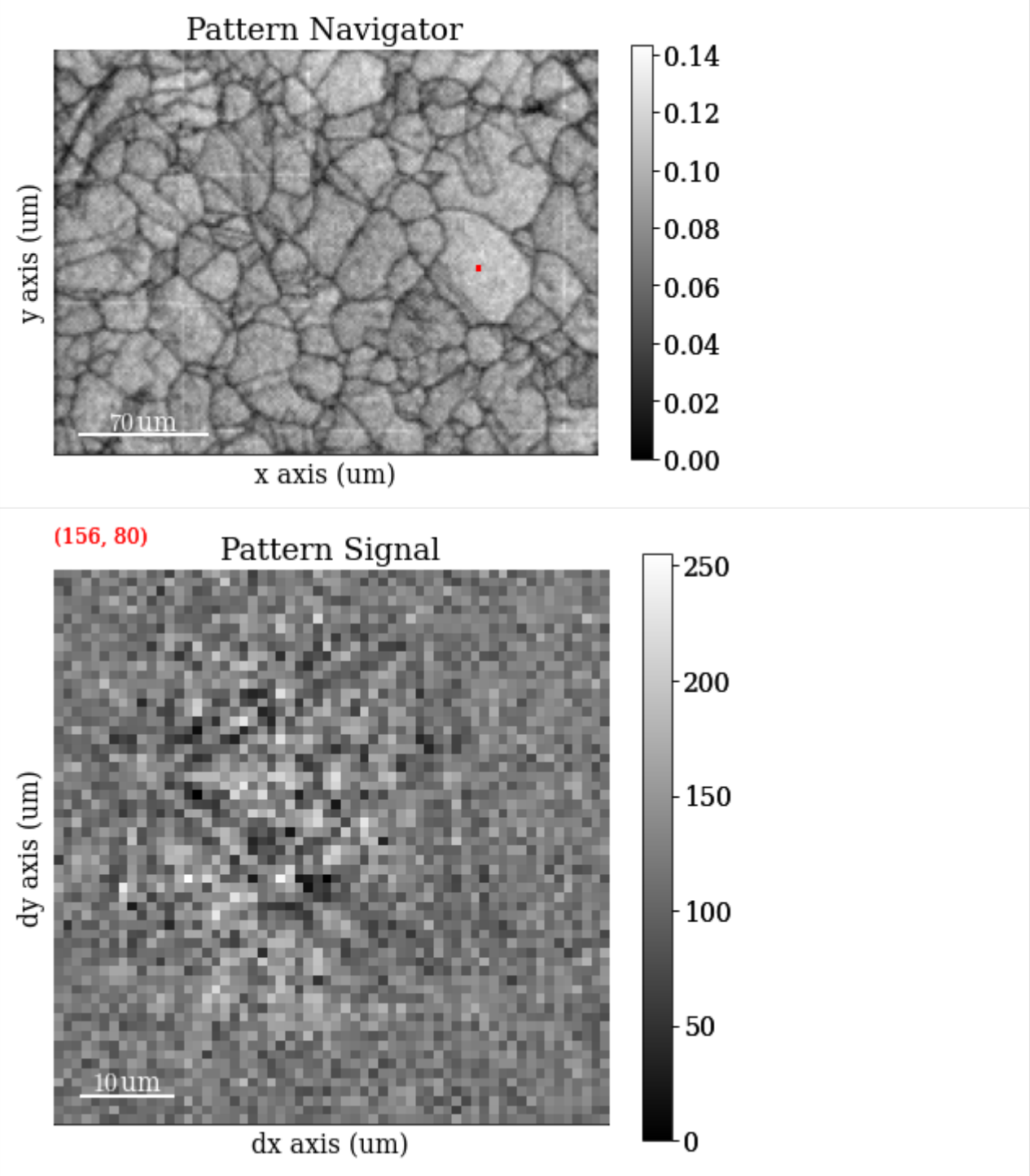
[22]:
```
maps_iq = s.get_image_quality()
```

```
[########################################] | 100% Completed | 1.51 ss
```

Navigate patterns in $Q$ map

[23]:
```
s_iq = hs.signals.Signal2D(maps_iq)
```

[24]:
```
s.plot(s_iq)
```




Save $Q$ map to file

[25]:
```
# plt.imsave(data_path / "maps_iq.png", maps_iq, cmap="gray")
```

Get average neighbour dot product map (ADP)

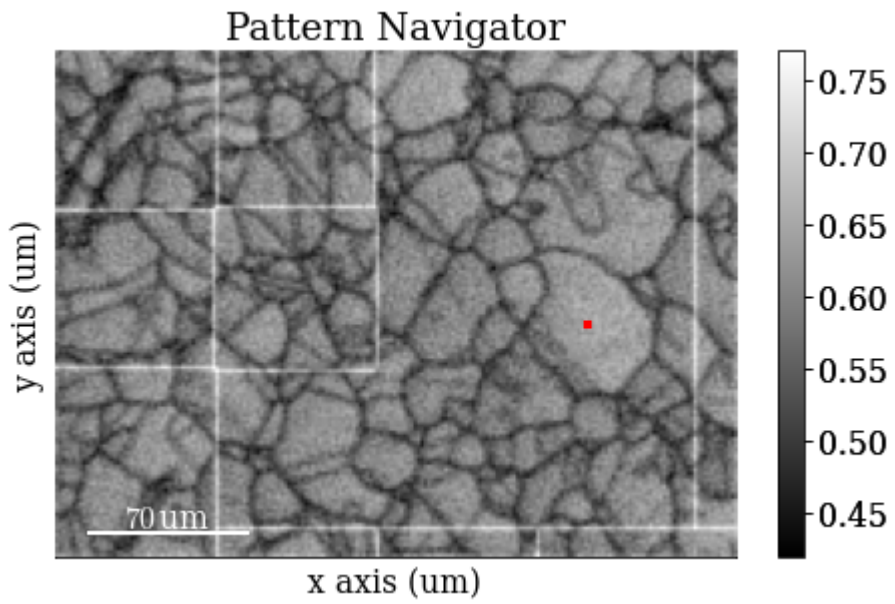
[26]:
```
maps_adp = s.get_average_neighbour_dot_product_map()
```

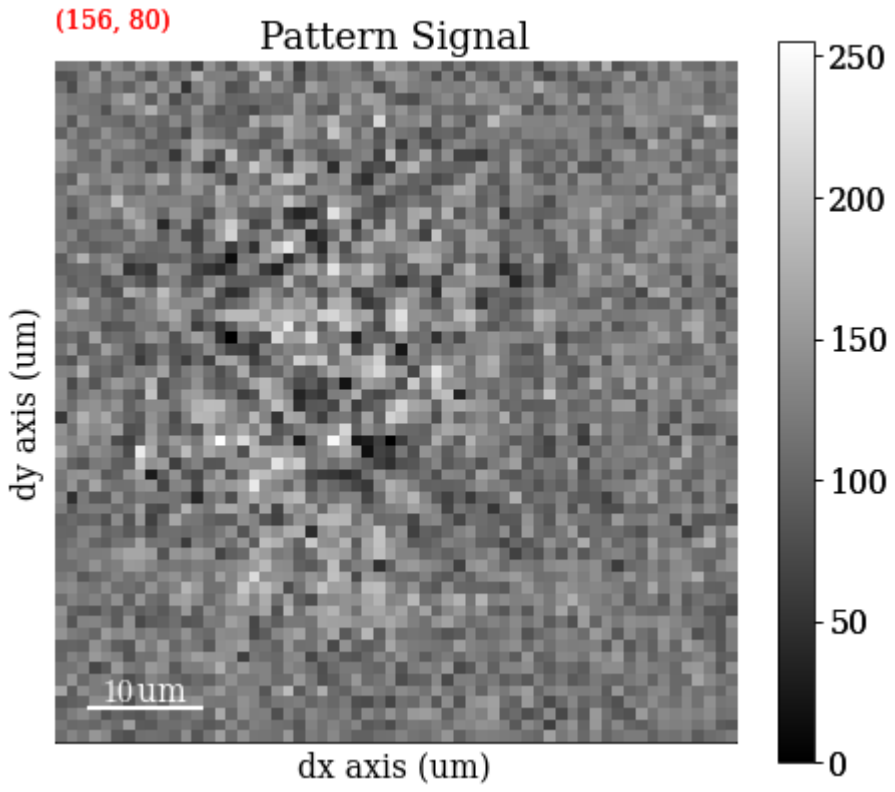
```
[########################################] | 100% Completed | 6.65 ss
```

Navigate patterns in the ADP map

[27]:
```
s_adp = hs.signals.Signal2D(maps_adp)
```

[28]:
```
s.plot(s_adp)
```



Save ADP map to file

[29]:
```
# plt.imsave(data_path / "maps_adp.png", maps_adp, cmap="gray")
```

### Hough indexing of calibration patterns

Orientation of detector with respect to the sample: * Known: * Sample tilt (about microscope X) * Camera tilt (about microscope X) * Unknown: * Projection/pattern centre (PCx, PCy, PCz): Shortest distance from source point to detector

Load calibration patterns to get a mean PC for the data

[30]:
```
s_cal = kp.data.ni_gain_calibration(7)
```

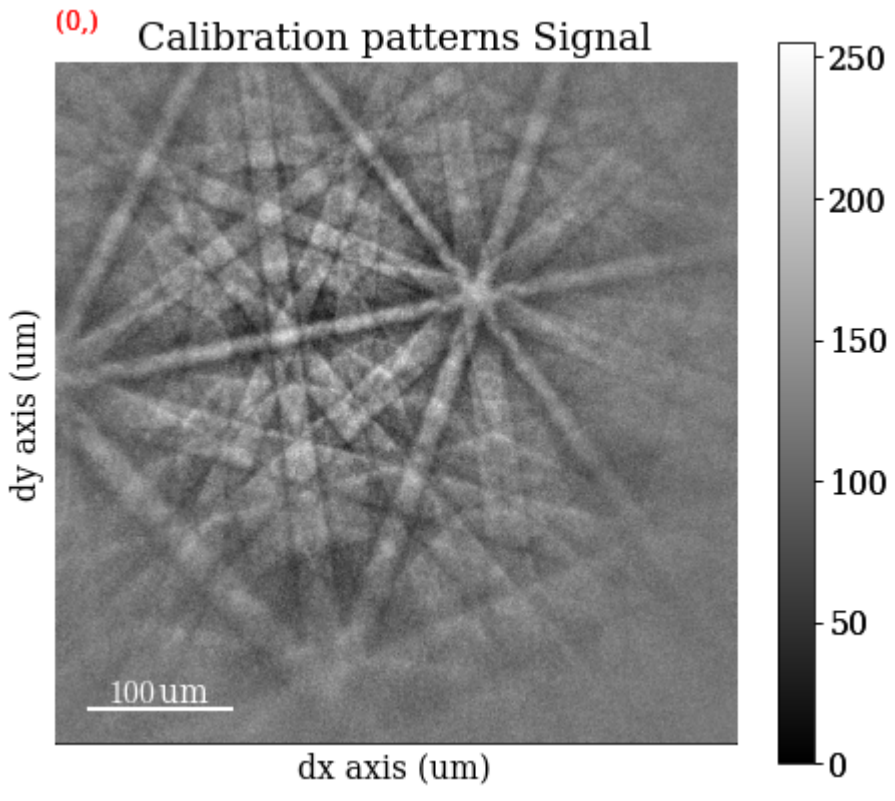
[31]:
```
s_cal
```

[31]:
```
<EBSD, title: Calibration patterns, dimensions: (9|480, 480)>
```

[32]:
```
s_cal.remove_static_background()
s_cal.remove_dynamic_background()
```

```
[########################################] | 100% Completed | 101.02 ms
[########################################] | 100% Completed | 102.32 ms
```

[33]:
```
s_cal.plot(navigator="none")
```




Generate an indexer instance used to optimize the PC and to perform Hough indexing

[34]:
```
sig_shape_cal = s_cal.axes_manager.signal_shape[::-1]
indexer_cal = ebsd_index.EBSDIndexer(
    phaselist=["FCC"],
    vendor="KIKUCHIPY",
    sampleTilt=70,
    camElev=0,
    patDim=sig_shape_cal,
)
```

Given an initial guess of the PC and optimize for the first calibration pattern, looking for convergence by updating `pc0` manually with the printed PC a couple of times (the first run of `optimize()` takes longer since PyEBSDIndex has to compile some code before running; consecutive runs are much quicker)

[35]:
```
pc0 = [0.42, 0.21, 0.50]
pc = pcopt.optimize(s_cal.inav[0].data, indexer=indexer_cal, PC0=pc0)
print(pc)
```

```
[0.40690636 0.22685186 0.50142137]
```

Optimize for all calibration patterns

[36]:
```
pc_all = pcopt.optimize(s_cal.data, indexer_cal, pc, batch=True)
print(pc_all)
```

```
[[0.40690636 0.22685186 0.50142137]
 [0.4080457  0.23103236 0.50566783]
 [0.42190216 0.23017428 0.50047604]
 [0.41001409 0.2248648  0.51609248]
 [0.42017003 0.22805524 0.48807502]
 [0.42541351 0.21285912 0.50680637]
 [0.40812411 0.24206145 0.49607673]
 [0.41075829 0.22226264 0.4940148 ]
 [0.42612194 0.22455536 0.50103979]]
```

Calculate the mean PC

[37]:
```
pc_mean = pc_all.mean(axis=0)

print(pc_mean)
print(pc_all.std(0))
```

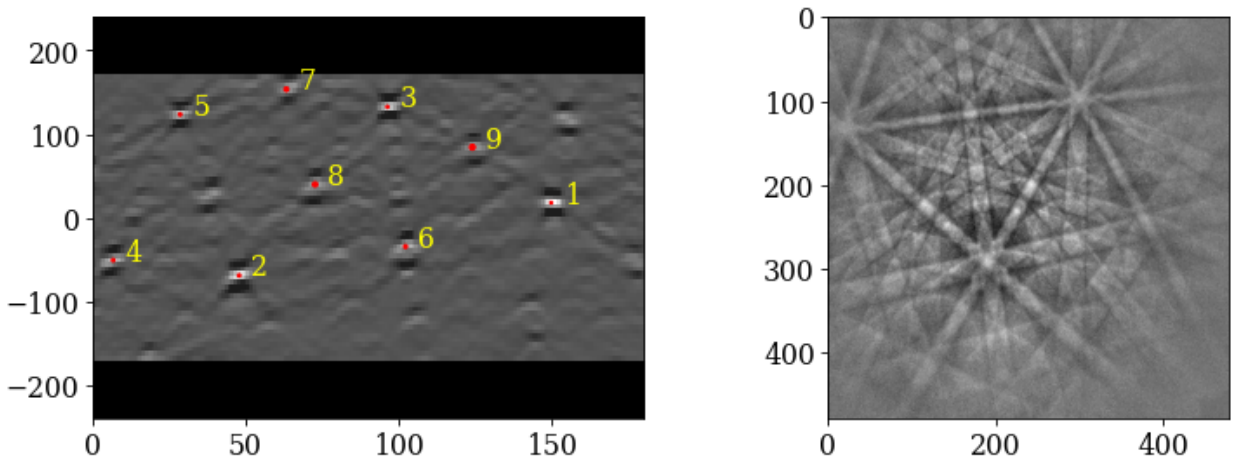
```
[0.41527291 0.22696857 0.50107449]
[0.00752652 0.00735793 0.00762872]
```

Index the calibration patterns

[38]:
```
plt.figure()
hi_res, *_ = indexer_cal.index_pats(s_cal.data, PC=pc_mean, verbose=2)
```

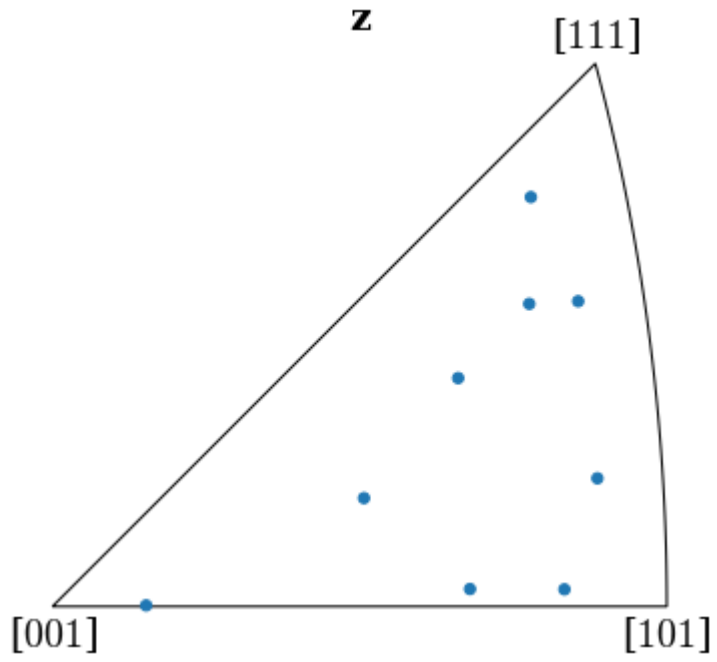
```
Radon Time: 0.032236748000002535
Convolution Time: 0.00548116299998469
Peak ID Time: 0.0021082410000019536
Band Label Time: 0.04232218699999635
Total Band Find Time: 0.08219512799999507
Band Vote Time:  0.008918966000010187
```

```
<Figure size 480x360 with 0 Axes>
```




Plot the returned orientations in the inverse pole figure (IPF), showing the crystal direction $[uvw]$ parallel to the out-of-plane direction (Z)

[39]:
```
G_hi = Orientation(hi_res[-1]["quat"], symmetry.Oh)
G_hi.scatter("ipf")
```




We can determine whether the indexed orientations are correct by projecting geometrical simulations (bands and zone axes) onto the patterns

### Geometrical simulations

Load a phase description from a CIF file

[40]:
```
phase = Phase.from_cif(str(data_path / "ni.cif"))
```

[41]:
```
phase
```

[41]:
```
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color: tab:
↪blue>
```

[42]:
```
phase.structure
```

[42]:
```
[Ni   0.000000 0.000000 0.000000 1.0000,
 Ni   0.000000 0.500000 0.500000 1.0000,
 Ni   0.500000 0.000000 0.500000 1.0000,
 Ni   0.500000 0.500000 0.000000 1.0000]
```

[43]:
```
phase.structure.lattice
```

[43]:
```
Lattice(a=3.52387, b=3.52387, c=3.52387, alpha=90, beta=90, gamma=90)
```

Generate reflectors and filter the list on on minimum structure factor

[44]:
```
ref = ReciprocalLatticeVector.from_min_dspacing(phase, 1)

ref.calculate_structure_factor()

F = abs(ref.structure_factor)
ref = ref[F > 0.5 * F.max()]

ref.print_table()
```

```
 h k l      d     |F|_hkl   |F|^2   |F|^2_rel   Mult
 1 1 1    2.035    11.8     140.0     100.0      8
 2 0 0    1.762    10.4     108.2      77.3      6
 2 2 0    1.246     7.4      55.0      39.3      12
 3 1 1    1.062     6.2      38.6      27.6      24
```

Give each distinct set of $\{hkl\}$ a colour

[45]:
```
hkl_sets = ref.get_hkl_sets()
```

[46]:
```
hkl_sets
```

[46]:
```
defaultdict(tuple,
            {(2.0, 0.0, 0.0): (array([ 6, 22, 24, 25, 27, 43]),),
             (2.0,
              2.0,
              0.0): (array([ 4,  5,  7,  8, 21, 23, 26, 28, 41, 42, 44, 45]),),
             (1.0, 1.0, 1.0): (array([12, 13, 16, 17, 32, 33, 36, 37]),),
             (3.0,
              1.0,
              1.0): (array([ 0,  1,  2,  3,  9, 10, 11, 14, 15, 18, 19, 20, 29, 30, 31,␣
```

(continues on next page)

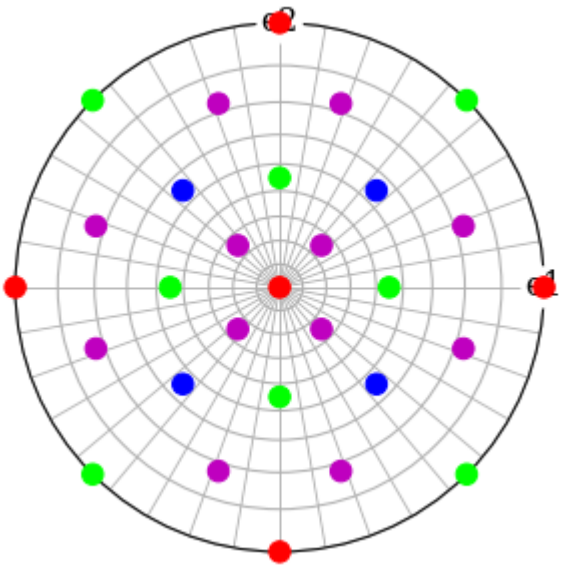
(continued from previous page)

```
↪34, 35,
                      38, 39, 40, 46, 47, 48, 49]),)})
```

[47]:
```
hkl_rgb = np.zeros((ref.size, 3))
rgb = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [0.75, 0, 0.75]]
for i, idx in enumerate(hkl_sets.values()):
    hkl_rgb[idx] = rgb[i]
```

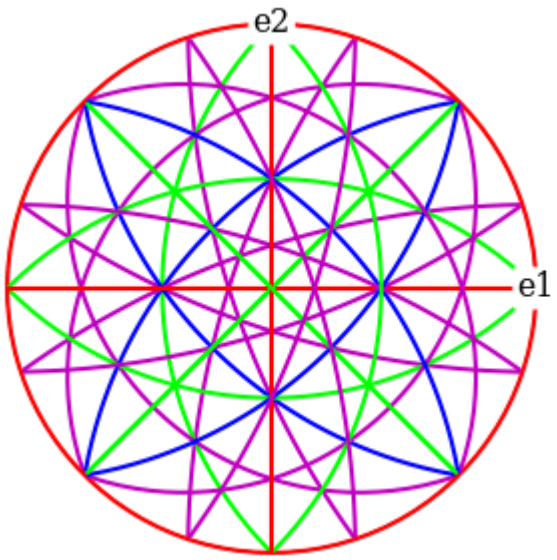
Plot each $(hkl)$ in the stereographic projection

[48]:
```
ref.scatter(c=hkl_rgb, grid=True, s=100, axes_labels=["e1", "e2"])
```




Plot the plane trace of each $(hkl)$ in the stereographic projection

[49]:
```
ref.draw_circle(color=hkl_rgb, axes_labels=["e1", "e2"])
```



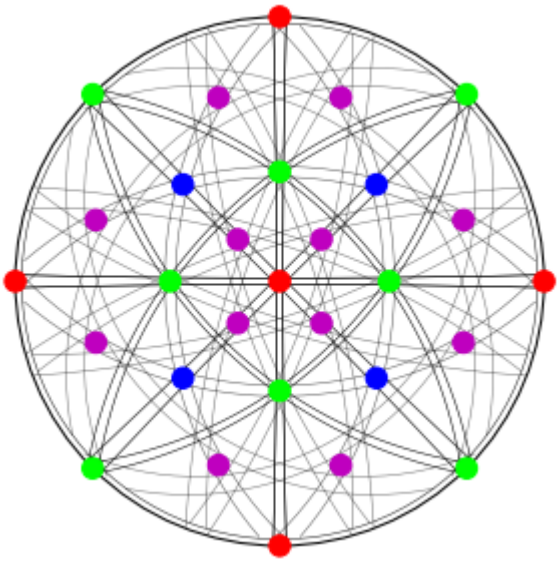

Set up geometrical simulations

[50]:
```
simulator = kp.simulations.KikuchiPatternSimulator(ref)
```

Calculate Bragg angles and plot bands in the stereographic projection

[51]:
```
simulator.reflectors.calculate_theta(20e3)
```

[52]:
```
fig = simulator.plot(hemisphere="upper", mode="bands", return_figure=True)
fig.axes[0].scatter(ref, c=hkl_rgb, s=100)
```



Specify the detector-sample geometry and perform geometrical simulations on the detector for each orientation found from Hough indexing

[53]:
```
det_cal = kp.detectors.EBSDDetector(
    shape=sig_shape_cal, pc=pc_mean, sample_tilt=70
)
det_cal
```

[53]:
```
EBSDDetector (480, 480), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.415, 0.227,␣
↪0.501)
```

[54]:
```
sim = simulator.on_detector(det_cal, G_hi)
```

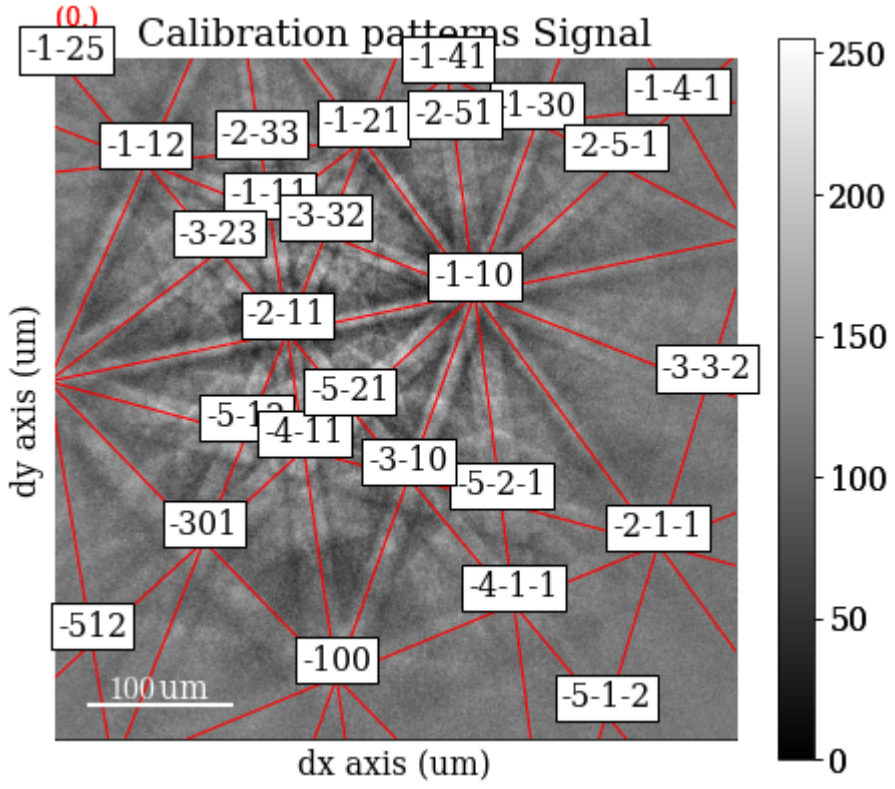
```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.39 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 103.17 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 101.66 ms
```

Plot the simulations as markers

[55]:
```
sim_markers = sim.as_markers(zone_axes_labels=True)
```

[56]:
```
s_cal.add_marker(sim_markers, permanent=True, plot_signal=False)
```

[57]:
```
s_cal.plot(navigator="none")
```




[58]:
```
# To delete previously added permanent markers, do
del s_cal.metadata.Markers
```

### Hough indexing of all patterns

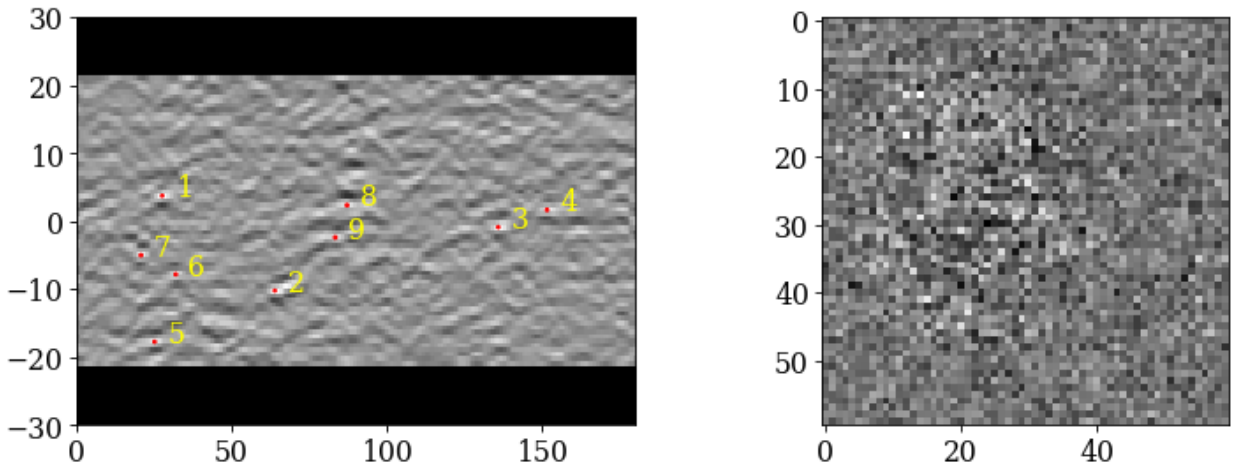
Create a new indexer and Hough index all patterns

[59]:
```
sig_shape = s.axes_manager.signal_shape[::-1]
indexer = ebsd_index.EBSDIndexer(
    phaselist=["FCC"],
    vendor="KIKUCHIPY",
    sampleTilt=det_cal.sample_tilt,
    camElev=det_cal.tilt,
    patDim=sig_shape,
)
```

[60]:
```
plt.figure()
hi_res_all, *_ = indexer.index_pats(
    s.data.reshape((-1,) + sig_shape), PC=det_cal.pc, verbose=2
)
```

```
Radon Time: 2.3059811180000622
Convolution Time: 3.065812205000043
Peak ID Time: 2.4583082690002698
Band Label Time: 4.4460913610001
Total Band Find Time: 12.276845957000006
Band Vote Time:  18.648167346999998
```

```
<Figure size 480x360 with 0 Axes>
```




Inspect the first output object (the only one kept)

[61]:
```
hi_res_all.dtype
```

[61]:
```
dtype([('quat', '<f8', (4,)), ('iq', '<f4'), ('pq', '<f4'), ('cm', '<f4'), ('phase', '<i4
↪'), ('fit', '<f4'), ('nmatch', '<i4'), ('matchattempts', '<i4', (4,)), ('totvotes', '
↪<i4')])
```

Create map coordinate arrays

[62]:
```
step_size = s.axes_manager["x"].scale
nav_shape = s.axes_manager.navigation_shape[::-1]
```

(continues on next page)

(continued from previous page)

```
i, j = np.indices(nav_shape) * step_size
```

Contain indexing results in a crystal map for easy plotting and saving

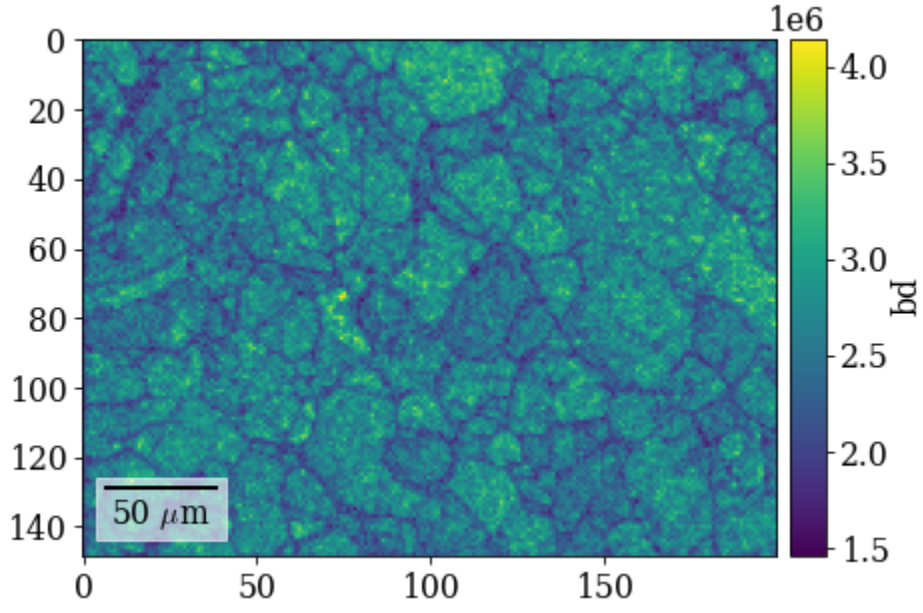
[63]:
```
xmap_hi = CrystalMap(
    rotations=Orientation(hi_res_all[-1]["quat"]),
    phase_list=PhaseList(phase),
    x=j.ravel(),
    y=i.ravel(),
    prop={
        "pq": hi_res_all[-1]["pq"],
        "cm": hi_res_all[-1]["cm"],
        "fit": hi_res_all[-1]["fit"],
    },
    scan_unit="um",
)
```

[64]:
```
xmap_hi
```

[64]:
```
Phase    Orientations  Name  Space group  Point group  Proper point group     Color
    0  29800 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: pq, cm, fit
Scan unit: um
```

Plot pattern quality (PC) and confidence metric (CM) maps

[65]:
```
fig = xmap_hi.plot(
    "pq", colorbar=True, colorbar_label="pq", return_figure=True
)
fig.savefig(data_path / "maps_hi_pq.png")
```

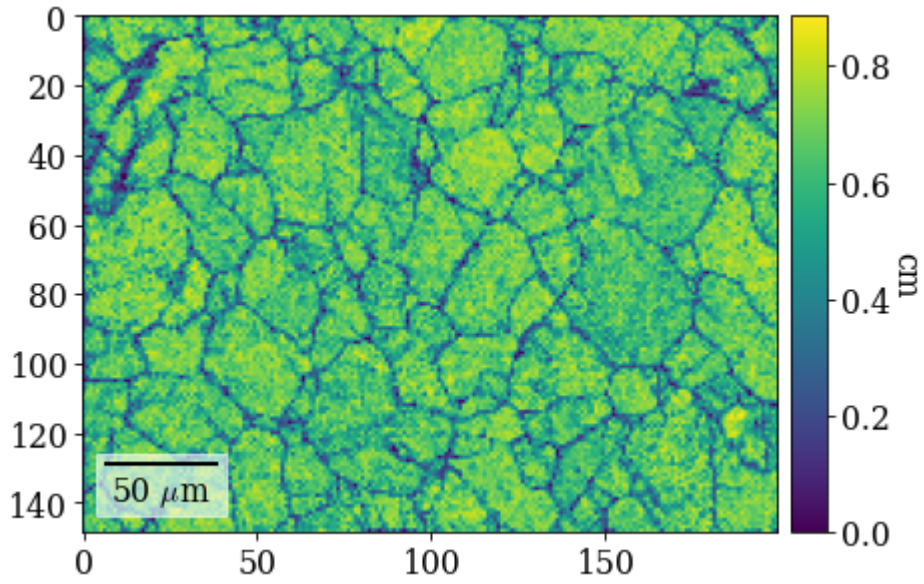



[66]:
```
fig = xmap_hi.plot(
```

(continues on next page)

(continued from previous page)

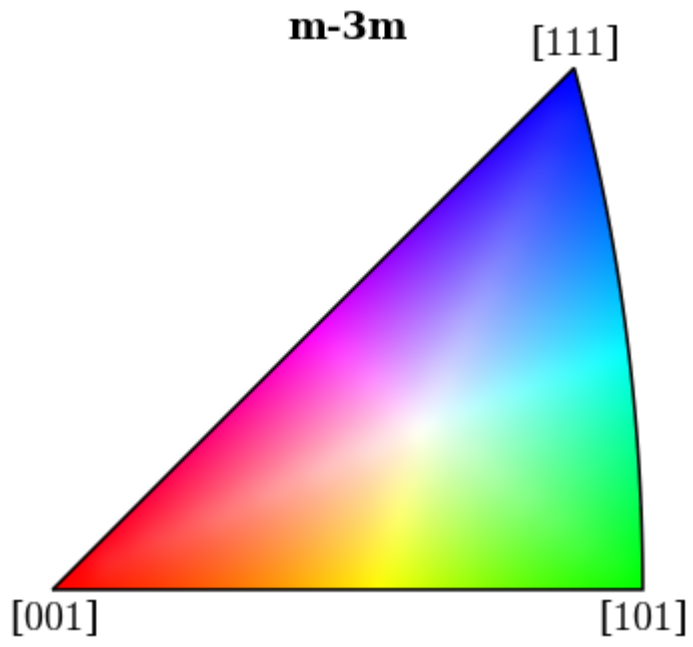
```
    "cm", colorbar=True, colorbar_label="cm", return_figure=True
)
fig.savefig(data_path / "maps_hi_cm.png")
```




Plot (IPF-X) orientation map

[67]:
```
ipfkey = plot.IPFColorKeyTSL(
    xmap_hi.phases[0].point_group, direction=Vector3d.xvector()
)
```

[68]:
```
fig = ipfkey.plot(return_figure=True)
```

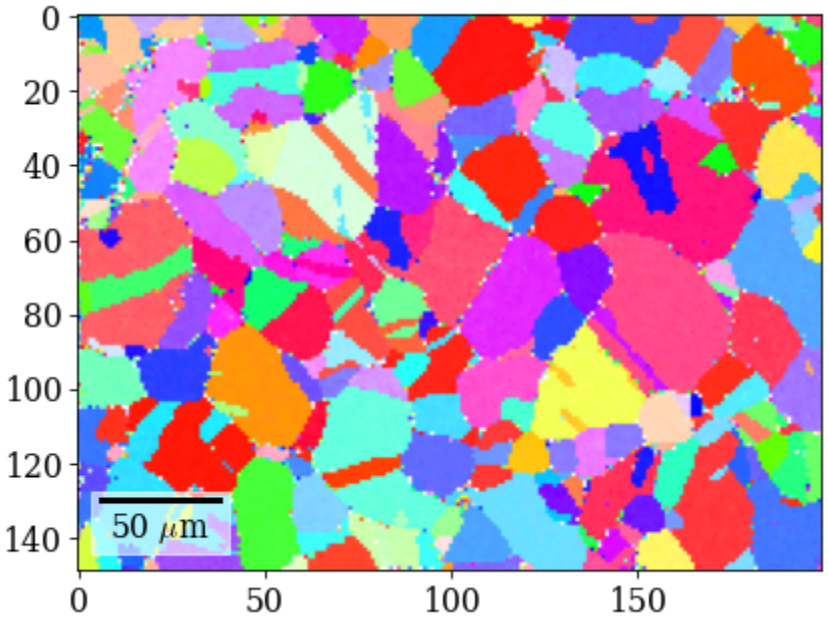


[69]:
```
# fig.savefig(data_path / "ipfkey.png")
```

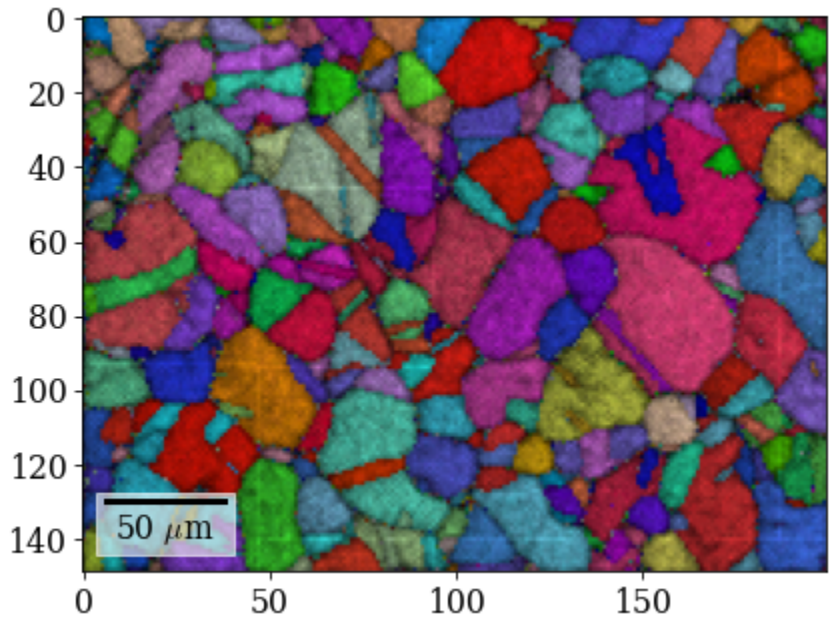
[70]:
```
G_hi = xmap_hi.orientations
```

[71]:
```
rgb_hi = ipfkey.orientation2color(G_hi)
```

[72]:
```
fig = xmap_hi.plot(rgb_hi, return_figure=True)
# fig.savefig(data_path / "maps_hi_ipfz.png")
```




[73]:
```
fig = xmap_hi.plot(rgb_hi, overlay=maps_iq.ravel(), return_figure=True)
# fig.savefig(data_path / "maps_hi_ipfz_iq.png")
```




Save Hough indexing results to file (.ang file readable by MTEX, EDAX TSL OIM Analysis etc., HDF5 file can be read back in into Python)

[74]:
```
# io.save(data_path / "xmap_hi.ang", xmap_hi)
```

[75]:
```
# io.save(data_path / "xmap_hi.h5", xmap_hi)
```

Create a new detector with a shape corresponding to the experimental patterns

[76]:
```
det = det_cal.deepcopy()
det.shape = sig_shape
```

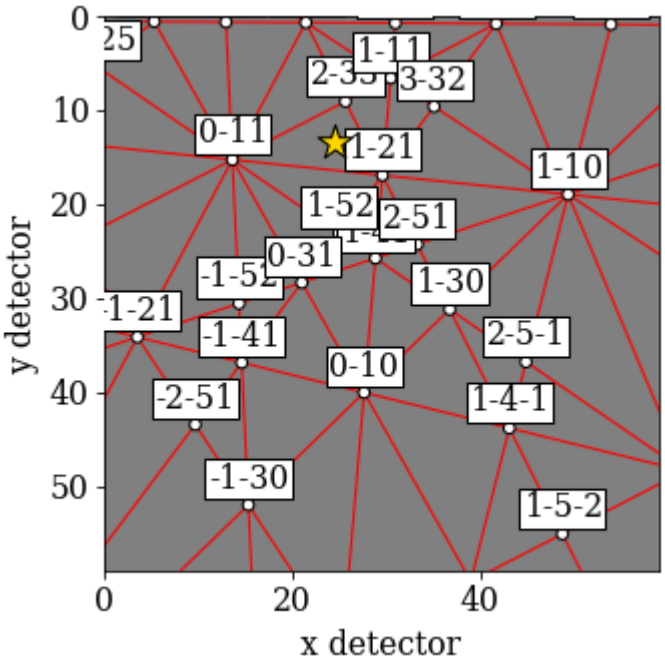
Get geometrical simulations on this detector for all orientations from the map

[77]:
```
G_hi_2d = G_hi.reshape(*xmap_hi.shape)
sim_hi = simulator.on_detector(det, G_hi_2d)
```

```
Finding bands that are in some pattern:
[########################################] | 100% Completed | 101.53 ms
Finding zone axes that are in some pattern:
[########################################] | 100% Completed | 203.94 ms
Calculating detector coordinates for bands and zone axes:
[########################################] | 100% Completed | 304.15 ms
```

Plot the first simulated pattern

[78]:
```
sim_hi.plot()
```

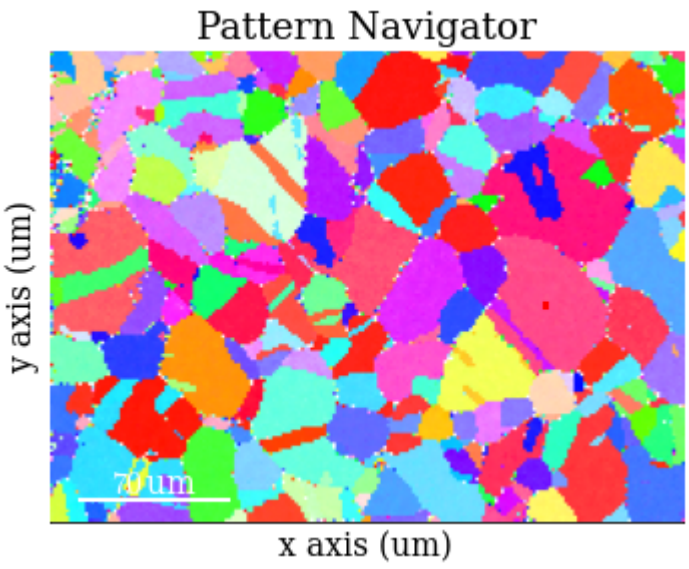


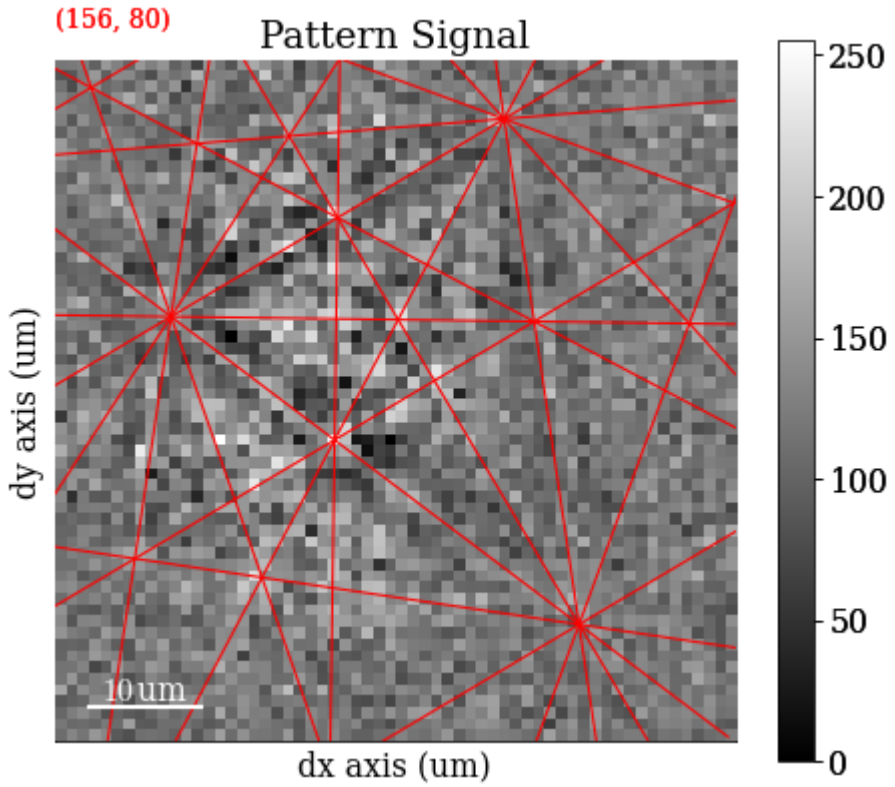

[79]:
```
rgb_hi_2d = rgb_hi.reshape(xmap_hi.shape + (3,))
s_rgb_hi = kp.draw.get_rgb_navigator(rgb_hi_2d)
```

Add the geometrical simulations

[80]:
```
s.add_marker(sim_hi.as_markers(), permanent=True, plot_signal=False)
```

[81]:
```
s.plot(s_rgb_hi)
```





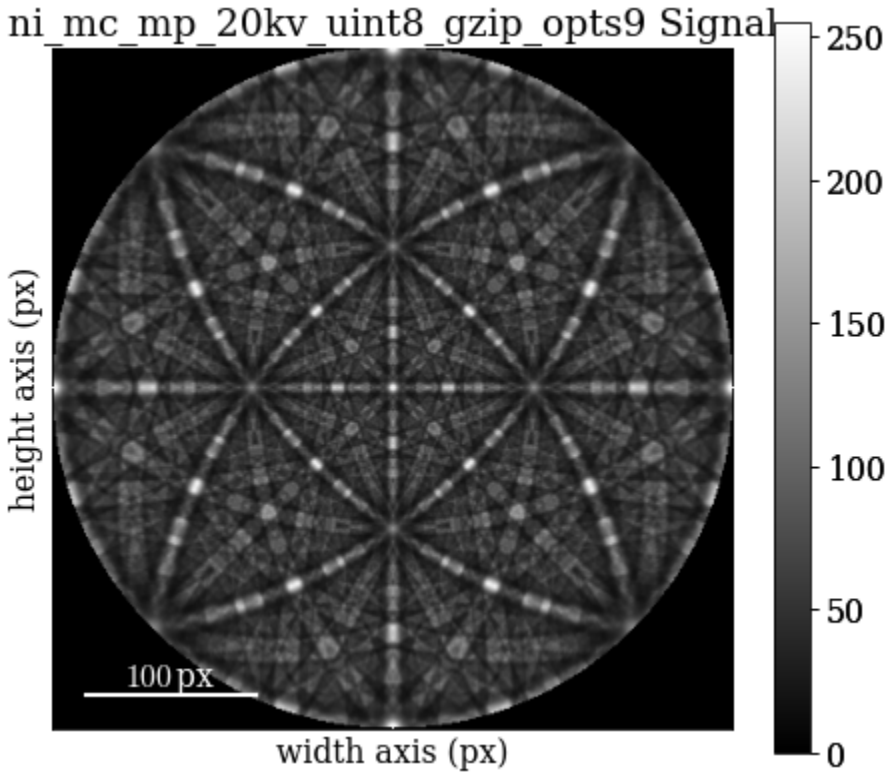

[82]:
```
# To delete previously added permanent markers, do
del s.metadata.Markers
```

### Dictionary indexing

Improve indexing quality by using dynamical simulations. Start by inspecting the dynamically simulated master pattern in the familiar (?) stereographic projection

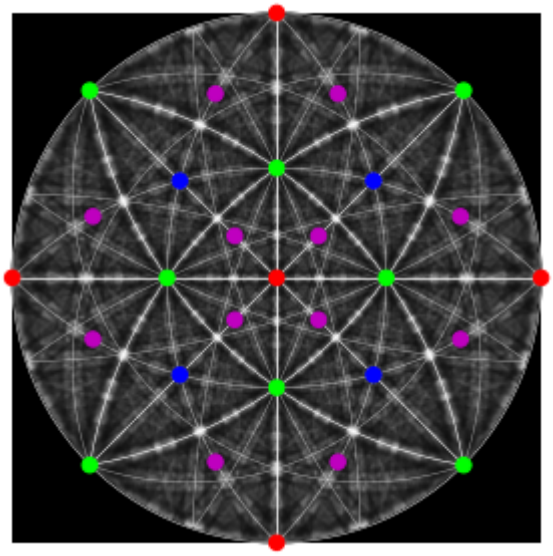
[83]:
```
mp_sp = kp.data.nickel_ebsd_master_pattern_small(projection="stereographic")
```

[84]:
```
mp_sp.plot()
```




Plot our geometrical simulation on top of the dynamical simulation

[85]:
```
fig = simulator.plot(hemisphere="upper", mode="lines", return_figure=True, color="w")
fig.axes[0].scatter(ref, c=hkl_rgb, s=50)

fig.axes[0].imshow(mp_sp.data, cmap="gray", extent=(-1, 1, -1, 1));
```

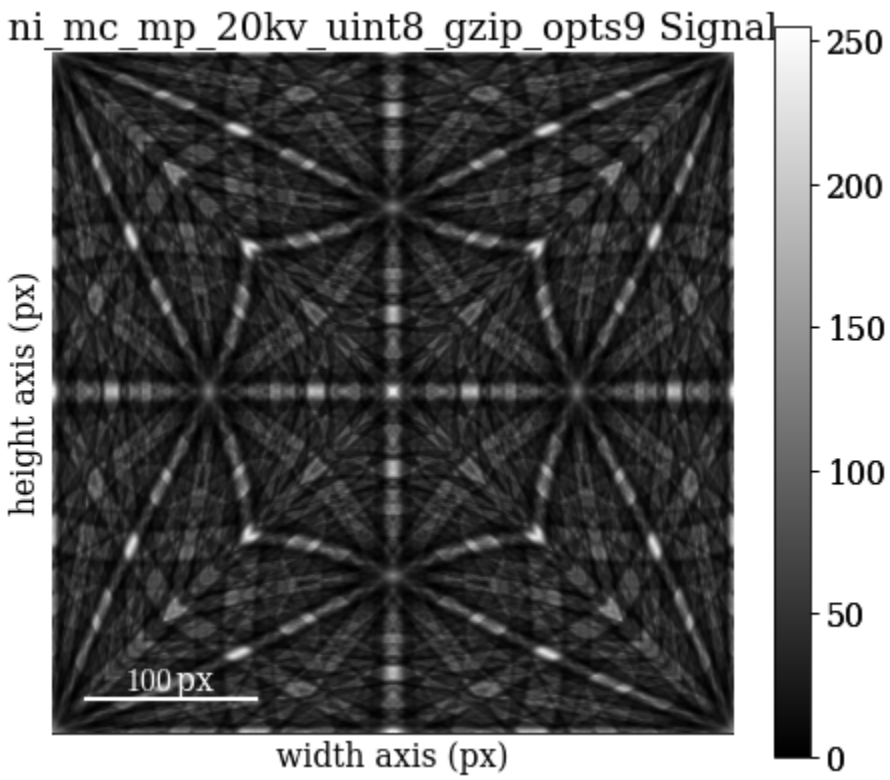


Re-load the master pattern but in the Lambert projection, used to generate the dictionary of dynamically simulated patterns

[86]:
```
mp = kp.data.nickel_ebsd_master_pattern_small(projection="lambert")
```

Quickly inspect the Lambert master pattern

[87]:
```
mp.plot()
```




Discretely sample the complete orientation space of point group $m\bar{3}m$ (*Oh*) with an average misorientation of about 2.4° between points

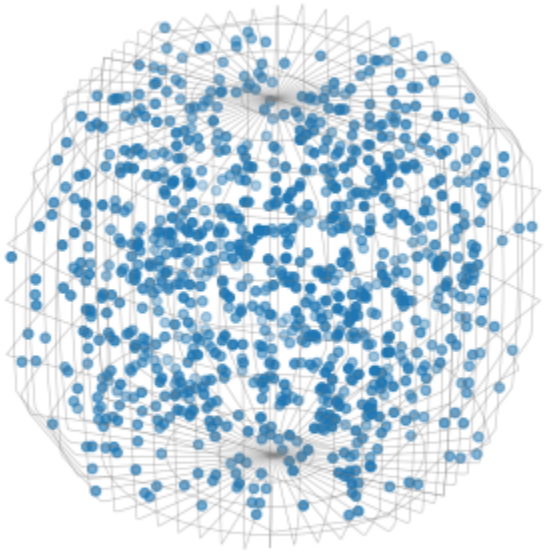
[88]:
```
Gr_sample = sampling.get_sample_fundamental(
    resolution=2.4, point_group=mp.phase.point_group
)
G_sample = Orientation(Gr_sample, symmetry=mp.phase.point_group)
```

[89]:
```
G_sample
```

[89]:
```
Orientation (58453,) m-3m
[[ 0.8614 -0.3311 -0.3311 -0.1968]
 [ 0.8614 -0.3349 -0.3349 -0.1836]
 [ 0.8614 -0.3384 -0.3384 -0.1703]
 ...
 [ 0.8614  0.3384  0.3384  0.1703]
 [ 0.8614  0.3349  0.3349  0.1836]
 [ 0.8614  0.3311  0.3311  0.1968]]
```

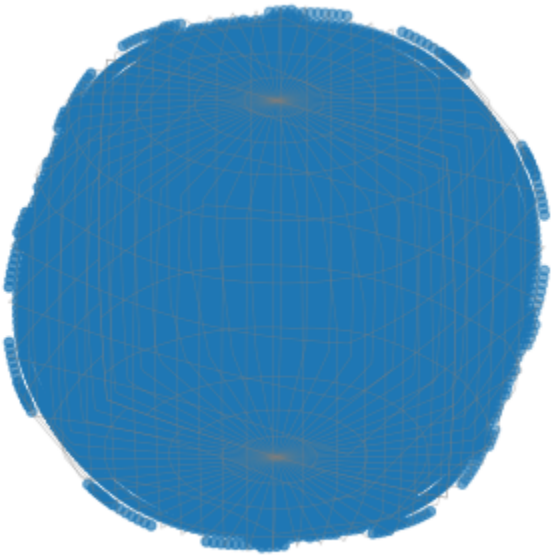
Plot 1000 sampled orientations in axis-angle space

[90]:
```
G_sample.get_random_sample(1000).scatter()
```



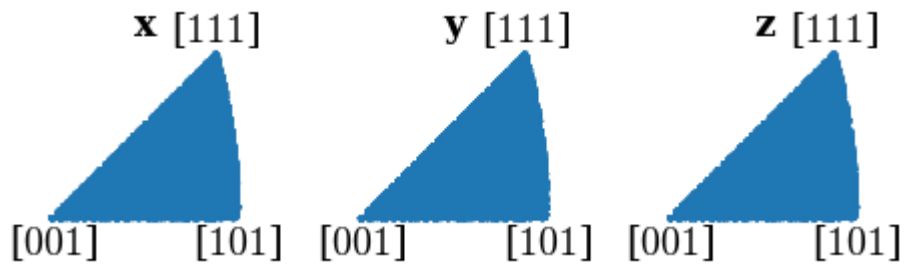
Plot all sampled orientations

[91]:
```
G_sample.scatter()
```



Plot all sampled orientations in the IPFs X, Y, and Z directions

[92]:
```
directions = Vector3d(((1, 0, 0), (0, 1, 0), (0, 0, 1)))
G_sample.scatter("ipf", direction=directions, c="C0", s=5)
```

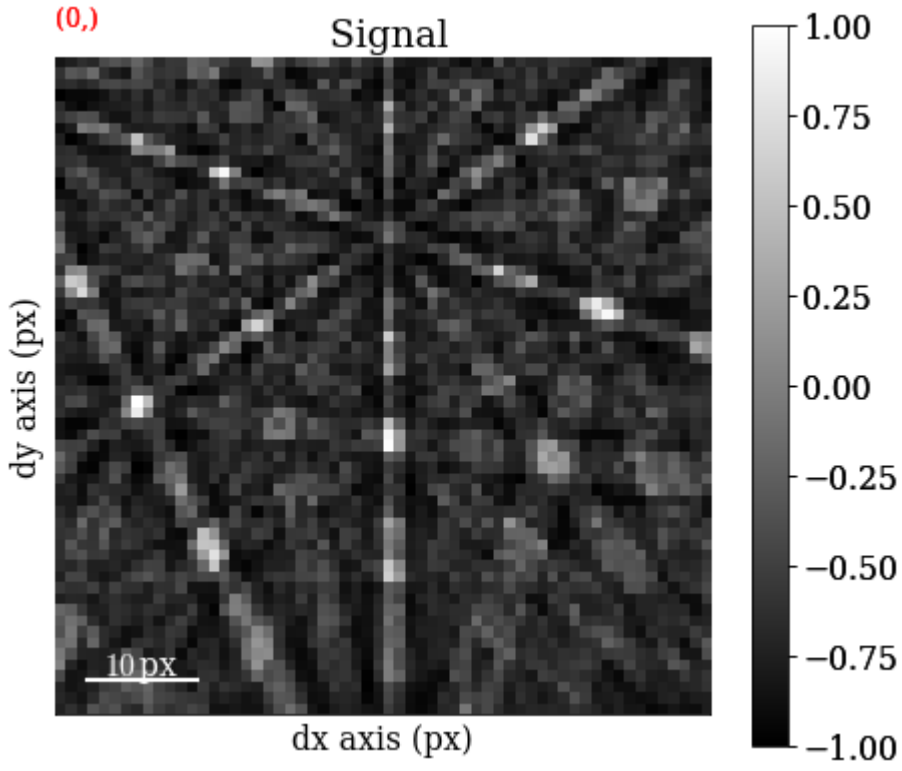



Set up generation of the dictionary of dynamically simulated patterns (with 2 000 patterns per chunk)

[93]:
```
s_dict = mp.get_patterns(G_sample, det, energy=20, chunk_shape=2000)
```

Generate the five first patterns and plot them

[94]:
```
s_dict.inav[:5].plot(navigator="none")
```



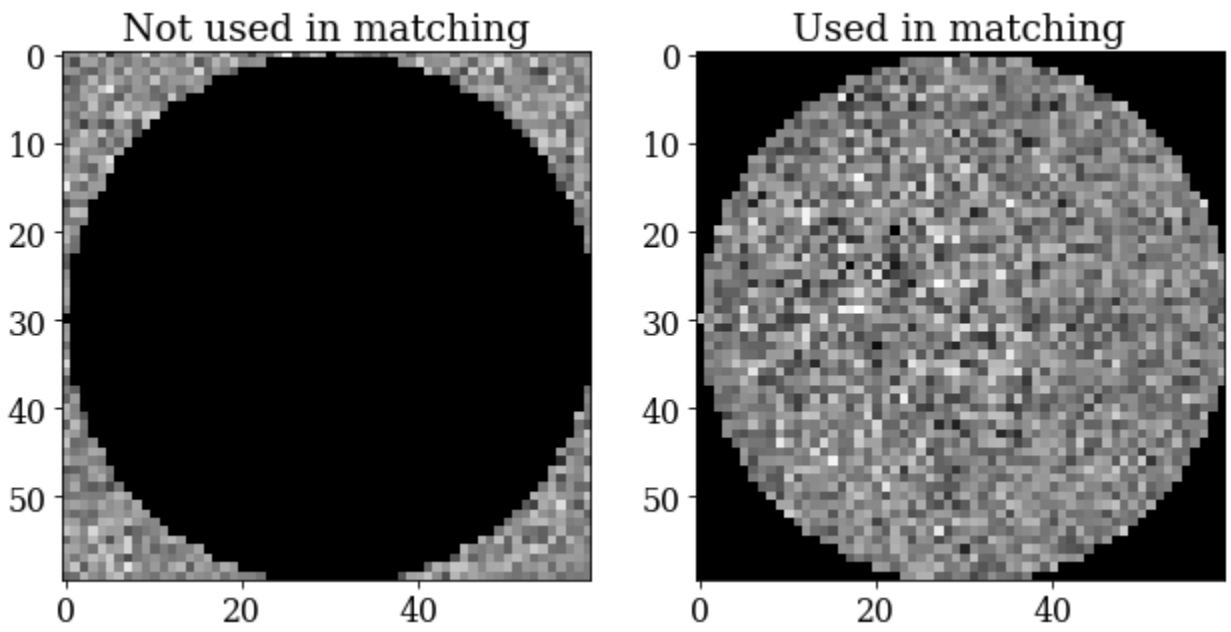

We will use a signal mask so that only pixels with the strongest signal are used during dictionary indexing and refinement

[95]:
```
signal_mask = ~kp.filters.Window("circular", det.shape).astype(bool)

p = s.inav[0, 0].data
fig, ax = plt.subplots(ncols=2, figsize=(10, 5))
```

(continues on next page)

(continued from previous page)

```
ax[0].imshow(p * signal_mask, cmap="gray")
ax[0].set_title("Not used in matching")
ax[1].imshow(p * ~signal_mask, cmap="gray")
ax[1].set_title("Used in matching");
```




Perform dictionary indexing by generating 2 000 simulated patterns at a time and compare them to all the experimental patterns

[96]:
```
xmap_di = s.dictionary_indexing(s_dict, signal_mask=signal_mask)
```

```
Dictionary indexing information:
  Phase name: ni
  Matching 29800 experimental pattern(s) to 58453 dictionary pattern(s)
  NormalizedCrossCorrelationMetric: float32, greater is better, rechunk: False,␣
↪navigation mask: False, signal mask: True
```

```
100%|----------------------------------| 30/30 [02:17<00:00,  4.60s/it]
```

```
  Indexing speed: 216.13963 patterns/s, 12634009.67826 comparisons/s
```

[97]:
```
xmap_di.scan_unit = "um"
```

[98]:
```
xmap_di
```

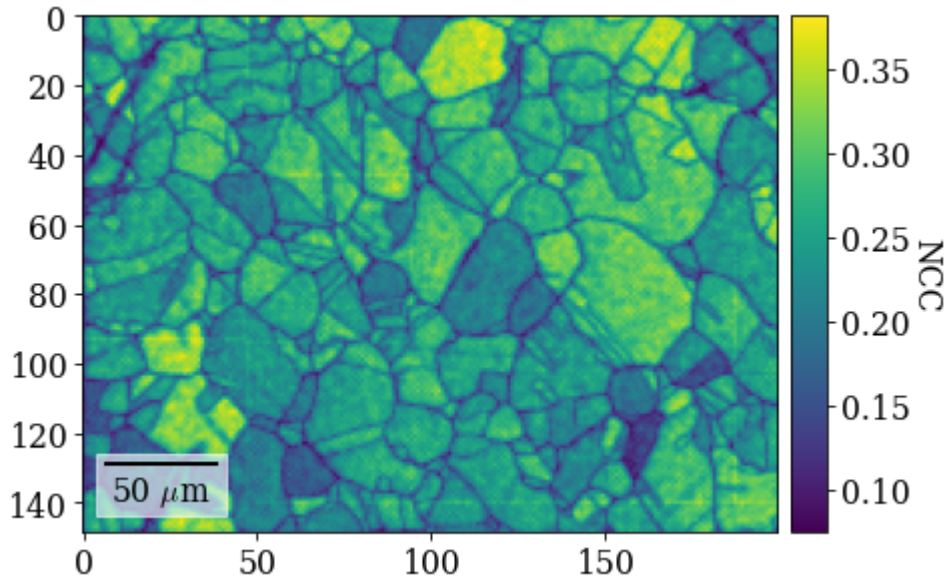
[98]:
```
Phase    Orientations  Name  Space group  Point group  Proper point group     Color
    0  29800 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: scores, simulation_indices
Scan unit: um
```

[99]:
```
xmap_di.scores.shape
```

[99]:
```
(29800, 20)
```

Plot similarity scores (normalized cross-correlation, NCC) between best matching experimental and simulated patterns

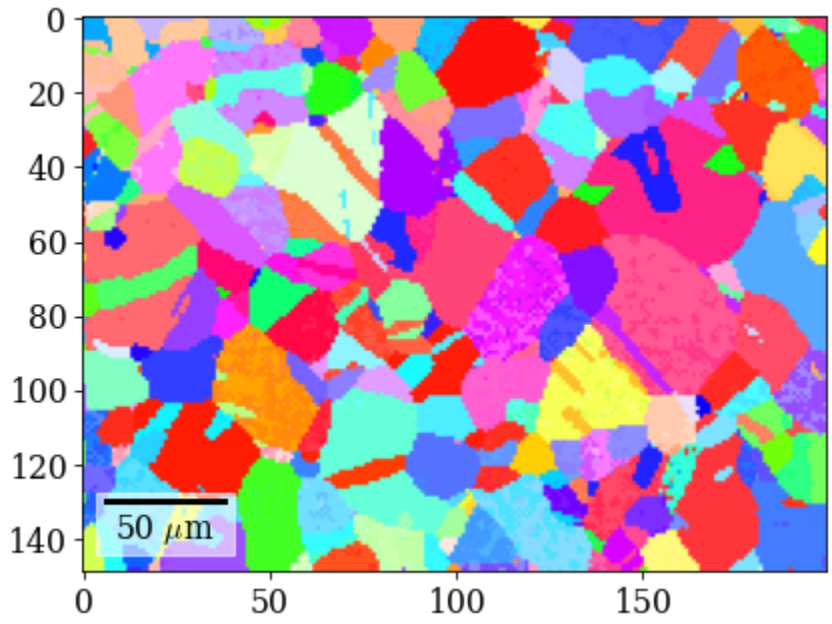
[100]:
```
fig = xmap_di.plot(
    xmap_di.scores[:, 0],
    colorbar=True,
    colorbar_label="NCC",
    return_figure=True,
)
# fig.savefig(data_path / "maps_di_ncc.png")
```




Plot IPF-X orientation map

[101]:
```
rgb_di = ipfkey.orientation2color(xmap_di.orientations)
```

[102]:
```
fig = xmap_di.plot(rgb_di, return_figure=True)
# fig.savefig(data_path / "maps_di_ipfz.png")
```




Save dictionary indexing results to file

[103]:
```
# io.save(data_path / "xmap_di.ang", xmap_di)
# io.save(data_path / "xmap_di.h5", xmap_di)
```

### Orientation refinement

Refine dictionary indexing results by re-generating dynamically simulated patterns iteratively by letting the discretly sampled orientations vary to find the best match

[104]:
```
xmap_ref = s.refine_orientation(
    xmap=xmap_di,
    detector=det,
    master_pattern=mp,
    energy=20,
    signal_mask=signal_mask,
    method="LN_NELDERMEAD",
    rtol=1e-3,
)
```

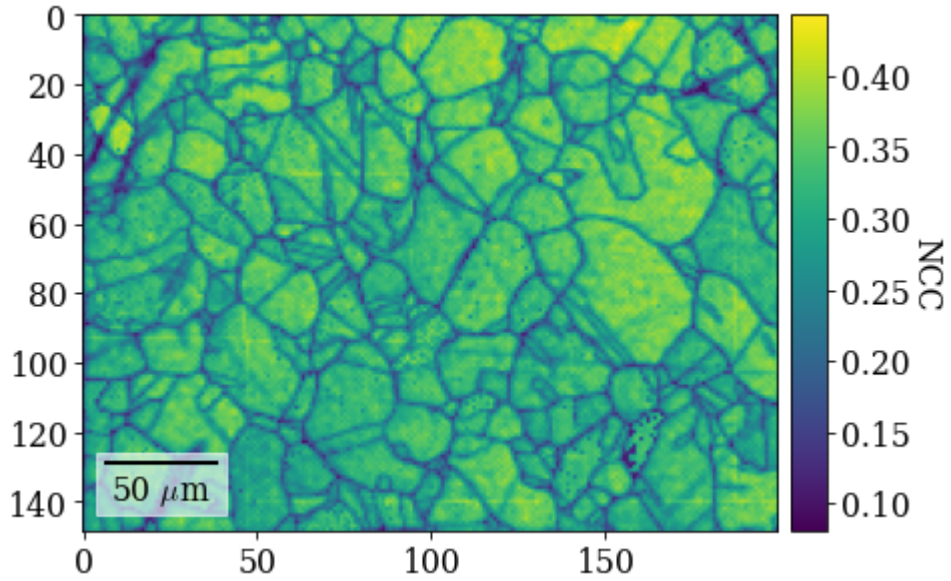
```
Refinement information:
  Method: LN_NELDERMEAD (local) from NLopt
  Trust region (+/-): None
  Relative tolerance: 0.001
Refining 29800 orientation(s):
[########################################] | 100% Completed | 225.72 s
Refinement speed: 131.99694 patterns/s
```

[105]:
```
xmap_ref
```

[105]:
```
Phase    Orientations  Name  Space group  Point group  Proper point group     Color
    0  29800 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: scores, num_evals
Scan unit: um
```

Plot refined NCC scores

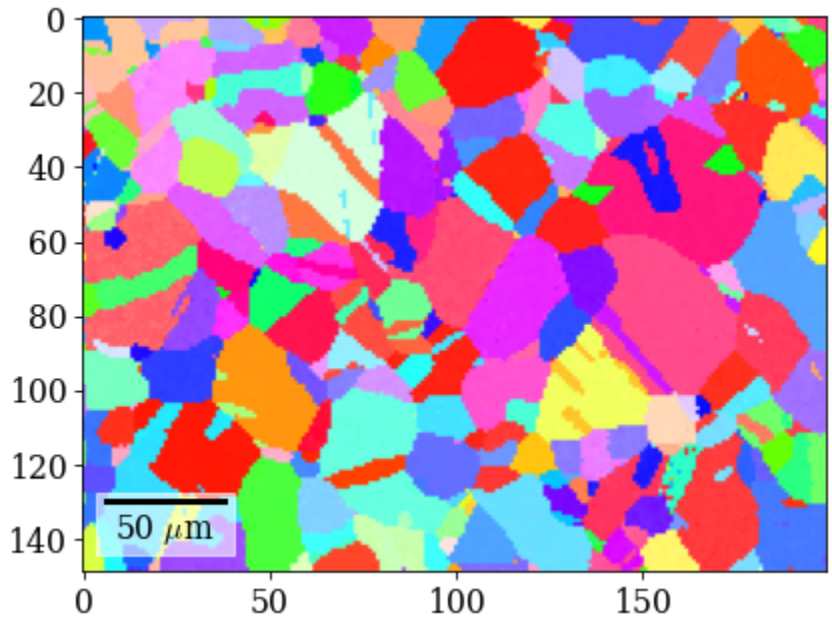
[106]:
```
fig = xmap_ref.plot(
    xmap_ref.scores, colorbar=True, colorbar_label="NCC", return_figure=True
)
# fig.savefig(data_path / "maps_di_ref_ncc.png")
```



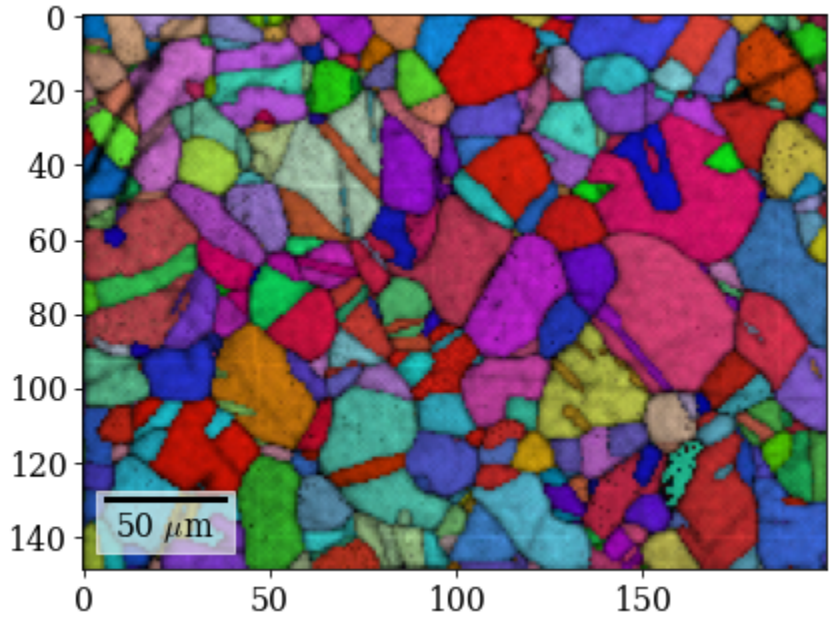

Plot refined IPF-X orientation map

[107]:
```
rgb_ref = ipfkey.orientation2color(xmap_ref.orientations)
```

[108]:
```
fig = xmap_ref.plot(rgb_ref, return_figure=True)
# fig.savefig(data_path / "maps_di_ref_ipfz.png")
```




[109]:
```
fig = xmap_ref.plot(rgb_ref, return_figure=True, overlay=xmap_ref.scores)
# fig.savefig(data_path / "maps_di_ref_ipfz_ncc.png")
```




Save refined results to file

[110]:
```
# io.save(data_path / "xmap_di_ref.ang", xmap_ref)
# io.save(data_path / "xmap_di_ref.h5", xmap_ref)
```

## 1.3 Examples

This page contains short examples of common tasks using kikuchipy. It is broken up into sections covering specific topics.

For longer in-depth guides, see our *Tutorials*. For descriptions of all the functions, modules, and objects in kikuchipy, see the *API reference*.

---

**Note:** Making examples is a work in progress. The topics and examples, and thus the URLs to examples, might change between releases. The *Tutorials* represent a complete showcase of the existing functionality in kikuchipy.

---

### 1.3.1 Pattern processing

These examples cover processing of pattern intensities like binning and background subtraction.

### 1.3.2 Reference frames

These examples cover use of tools like the *EBSDDetector* for understanding reference frames relevant to EBSD.

### 1.3.3 Selecting data

These examples cover selection of data via extraction a subset of the navigation and/or signal axes.

### 1.3.4 Visualization

These examples cover visualization of Kikuchi patterns and derived maps.

#### Pattern processing

These examples cover processing of pattern intensities like binning and background subtraction.

#### Pattern binning

This example shows how to bin *EBSD* patterns using *downsample()* or HyperSpy's *rebin()* (see Rebinning for details).

---

**Note:** In general, better contrast is obtained by removing the static (and dynamic) background prior to binning instead of after it.

---

```
import hyperspy.api as hs
import kikuchipy as kp


s = kp.data.si_ebsd_moving_screen(allow_download=True, show_progressbar=False)
s.remove_static_background(show_progressbar=False)
```

(continues on next page)

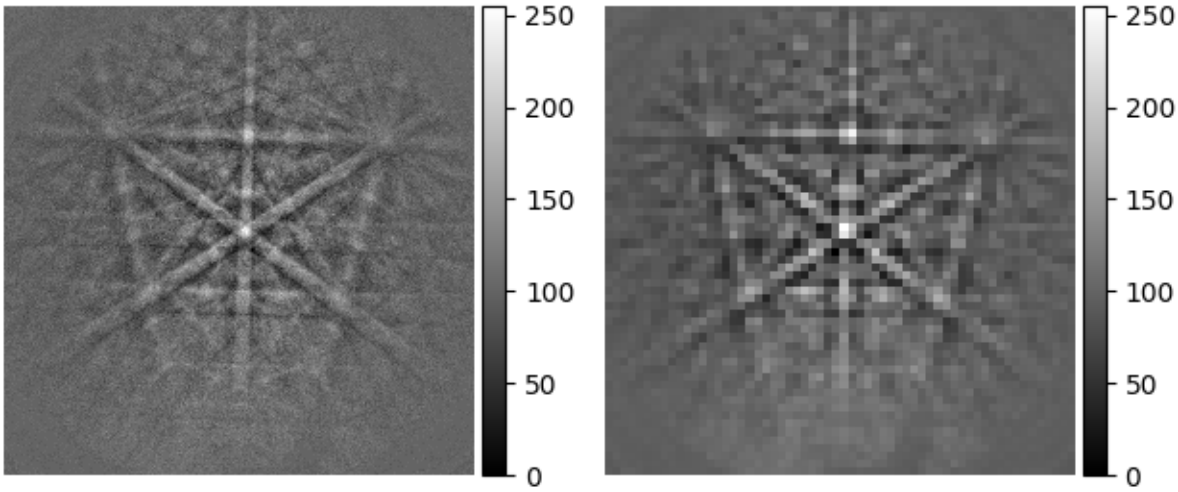
(continued from previous page)

```
print(s)
print(s.static_background.shape)
print(s.detector)
```

```
<EBSD, title: si_in Scan 1, dimensions: (|480, 480)>
(480, 480)
EBSDDetector (480, 480), px_size 1.0 um, binning 1, tilt 0.0, azimuthal 0.0, pc (0.5, 0.
↪5, 0.5)
```

Downsample by a factor of 8 while maintaining the data type (achieved by rescaling the pattern intensity). Note how the *static_background* and *detector* attributes are updated.

```
s2 = s.downsample(8, inplace=False)
_ = hs.plot.plot_images([s, s2], axes_decor="off", tight_layout=True, label=None)
print(s2.static_background.shape)
print(s2.detector)
```




```
[                                        ] | 0% Completed | 452.71 us
[########################################] | 100% Completed | 102.30 ms
(60, 60)
EBSDDetector (60, 60), px_size 1.0 um, binning 8, tilt 0.0, azimuthal 0.0, pc (0.5, 0.5,␣
↪0.5)
```

Rebin by passing the new shape (use `(1, 1, 60, 60)` if binning a 2D map). Note how the pattern is not rescaled and the data type is cast to either `int64` or `float64` depending on the initial data type.

```
s3 = s.rebin(new_shape=(60, 60))
print(s3.data.dtype)
print(s3.data.min(), s3.data.max())
```

```
uint64
3152 11998
```

The latter method is more flexible in that it allows for different binning factors in each axis, the factors are not restricted to being integers and the factors do not have to be divisors of the initial signal shape.

```
s4 = s.rebin(scale=(8, 9))
print(s4.data.shape)
```

```
(53, 60)
```

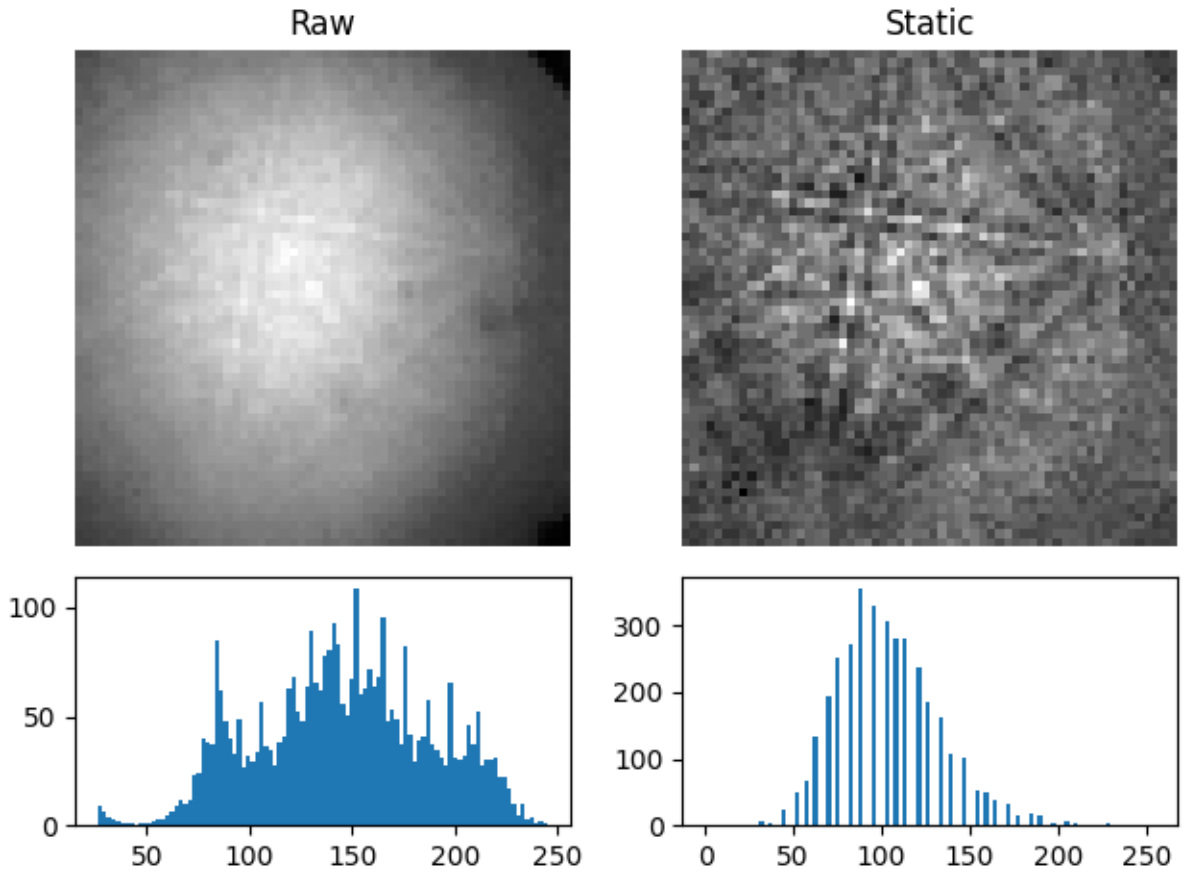
**Total running time of the script:** (0 minutes 4.773 seconds)

**Estimated memory usage:** 35 MB

### Static background correction

This example shows how to remove the static background of an EBSD pattern using *remove_static_background()*.

More details are given in the *pattern processing tutorial*.




```
[[84 87 90 ... 27 29 30]
 [87 90 93 ... 27 28 30]
```

(continues on next page)

(continued from previous page)

```
 [92 94 97 ... 39 28 29]
 ...
 [80 82 84 ... 36 30 26]
 [79 80 82 ... 28 26 26]
 [76 78 80 ... 26 26 25]]

[                                        ] | 0% Completed | 452.14 us
[########################################] | 100% Completed | 100.92 ms
```

```
import matplotlib.pyplot as plt
import kikuchipy as kp


# Load low resolution Ni patterns and check that the background pattern
# is stored with the signal
s = kp.data.nickel_ebsd_small()
print(s.static_background)

s2 = s.remove_static_background(inplace=False)

# Plot pattern before and after correction and the intensity histograms
patterns = [s.inav[0, 0].data, s2.inav[0, 0].data]
fig, axes = plt.subplots(2, 2, height_ratios=[3, 1.5])
for ax, pattern, title in zip(axes[0], patterns, ["Raw", "Static"]):
    ax.imshow(pattern, cmap="gray")
    ax.set_title(title)
    ax.axis("off")
for ax, pattern in zip(axes[1], patterns):
    ax.hist(pattern.ravel(), bins=100)
fig.tight_layout()
```

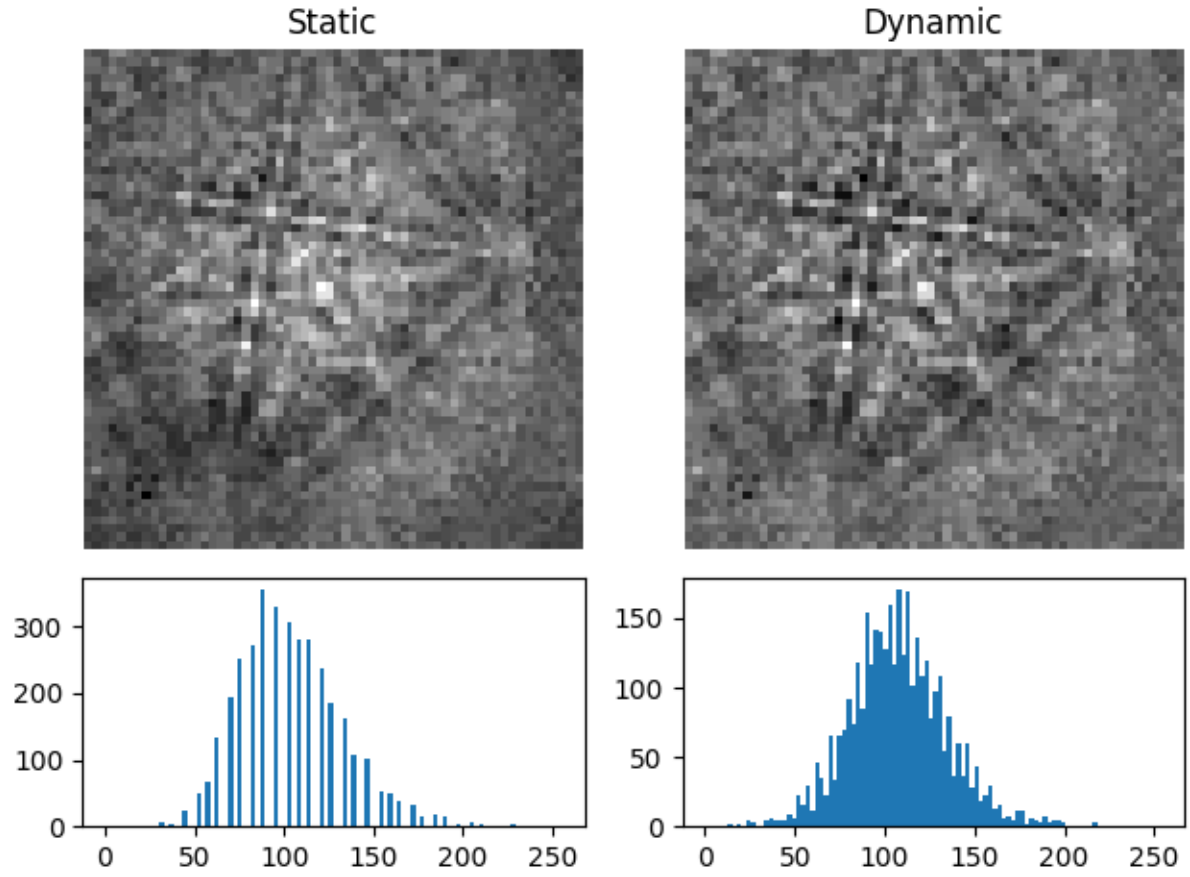
**Total running time of the script:** (0 minutes 0.859 seconds)

**Estimated memory usage:** 9 MB

### Dynamic background correction

This example shows how to remove the dynamic background of an EBSD pattern using *remove_dynamic_background()*.

More details are given in the *pattern processing tutorial*.




```
[[84 87 90 ... 27 29 30]
 [87 90 93 ... 27 28 30]
 [92 94 97 ... 39 28 29]
 ...
 [80 82 84 ... 36 30 26]
 [79 80 82 ... 28 26 26]
 [76 78 80 ... 26 26 25]]

[                                        ] | 0% Completed | 467.60 us
[########################################] | 100% Completed | 100.90 ms

[                                        ] | 0% Completed | 162.68 us
[########################################] | 100% Completed | 100.41 ms
```

```
import matplotlib.pyplot as plt
import kikuchipy as kp
```

(continues on next page)

(continued from previous page)

```
# Load low resolution Ni patterns and check that the *static* background
# pattern is stored with the signal
s = kp.data.nickel_ebsd_small()
print(s.static_background)

s.remove_static_background()
s2 = s.remove_dynamic_background(inplace=False)

# Plot pattern before and after correction and the intensity histograms
patterns = [s.inav[0, 0].data, s2.inav[0, 0].data]
fig, axes = plt.subplots(2, 2, height_ratios=[3, 1.5])
for ax, pattern, title in zip(axes[0], patterns, ["Static", "Dynamic"]):
    ax.imshow(pattern, cmap="gray")
    ax.set_title(title)
    ax.axis("off")
for ax, pattern in zip(axes[1], patterns):
    ax.hist(pattern.ravel(), bins=100)
fig.tight_layout()
```

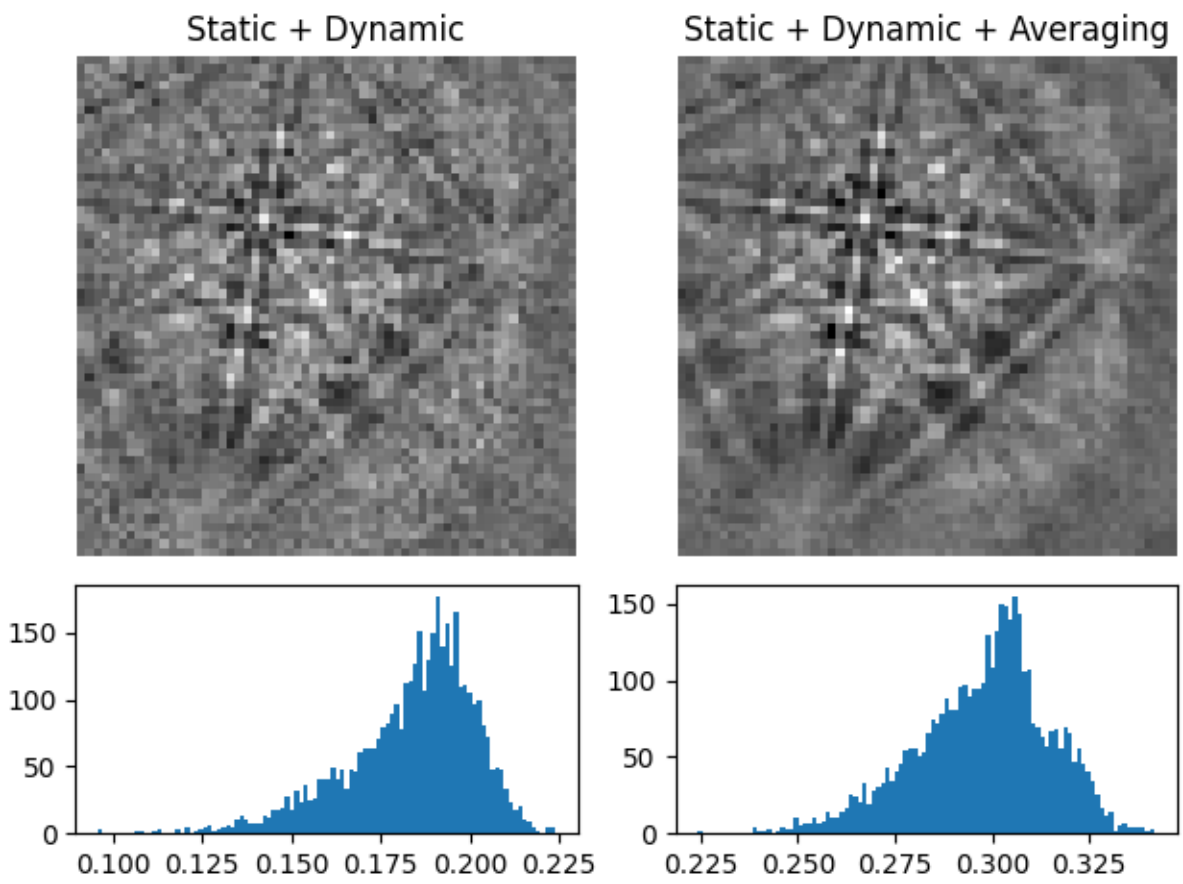
**Total running time of the script:** (0 minutes 2.495 seconds)

**Estimated memory usage:** 13 MB

### Neighbour pattern averaging

This example shows how to average each pattern in a scan with its nearest neighbours using *average_neighbour_patterns()*.

More details are given in the *pattern processing tutorial* and the *feature maps tutorial*.




```
import hyperspy.api as hs
import kikuchipy as kp
import matplotlib.pyplot as plt


# Silence progressbars
hs.preferences.General.show_progressbar = False

# Load Ni patterns and subtract static and dynamic background
s = kp.data.nickel_ebsd_large()
s.remove_static_background()
s.remove_dynamic_background()

# Get image quality before averaging
iq0 = s.get_image_quality()

# Keep one pattern for comparison
x, y = (50, 8)
pattern0 = s.inav[x, y].deepcopy()

# Average in a (3, 3) window with a Gaussian kernel with a standard
# deviation of 1
s.average_neighbour_patterns(window="gaussian", std=1)
```

(continues on next page)

(continued from previous page)

```
iq1 = s.get_image_quality()
pattern1 = s.inav[x, y]

# Plot pattern and histograms of image qualities before and after
# averaging
fig, axes = plt.subplots(2, 2, height_ratios=[3, 1.5])
for ax, pattern, title in zip(
    axes[0],
    [pattern0, pattern1],
    ["Static + Dynamic", "Static + Dynamic + Averaging"],
):
    ax.imshow(pattern, cmap="gray")
    ax.set_title(title)
    ax.axis("off")
for ax, iq in zip(axes[1], [iq0, iq1]):
    ax.hist(iq.ravel(), bins=100)
fig.tight_layout()
```

**Total running time of the script:** (0 minutes 5.038 seconds)

**Estimated memory usage:** 155 MB

### Adaptive histogram equalization

This example shows how to perform AHE on a simulated pattern and a master pattern. Two identical simulated patterns, but one projected from the master pattern *after* AHE has been applied to it, are compared. We'll use *kikuchipy.signals.EBSDMasterPattern.adaptive_histogram_equalization()* and the equivalent method for the EBSD class.

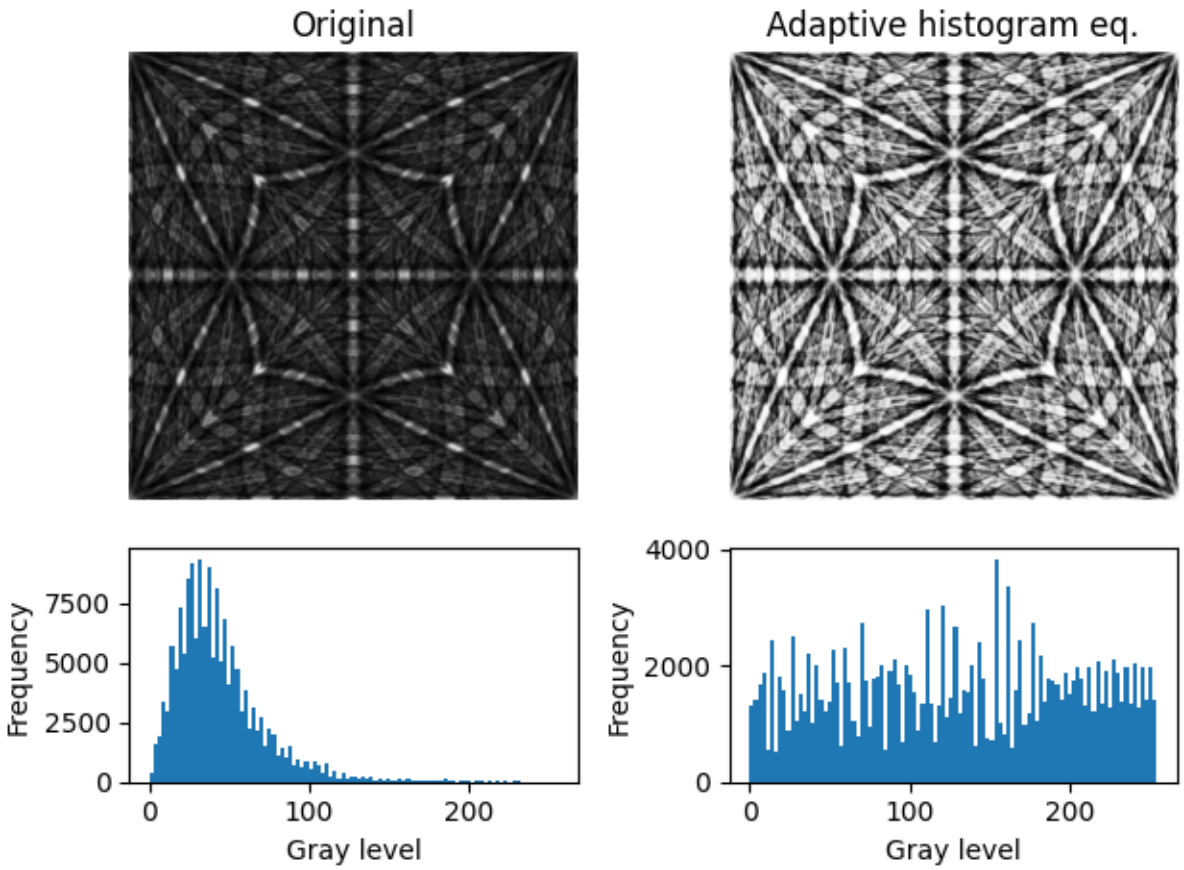
Adaptive histogram equalization (AHE) has been used to enhance pattern contrast and improve the efficacy of the normalized dot product (NDP) metric when comparing experimental and simulated patterns, e.g. in dictionary indexing [Marquardt *et al.*, 2017]. Before performing AHE, it might be worth considering using the normalized cross-correlation (NCC) metric instead.

```
import hyperspy.api as hs
import kikuchipy as kp
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from orix.quaternion import Rotation


hs.preferences.General.show_progressbar = False

# Master pattern in square Lambert projection, of integer data type
mp = kp.data.nickel_ebsd_master_pattern_small(projection="lambert")

mp2 = mp.adaptive_histogram_equalization(inplace=False)

# Plot master pattern before and after correction and the intensity histograms
mps_data = [mp.data, mp2.data]
fig, axes = plt.subplots(2, 2, height_ratios=[3, 1.5])
```

(continues on next page)

(continued from previous page)

```
for ax, pattern, title in zip(
    axes[0], mps_data, ["Original", "Adaptive histogram eq."]
):
    ax.imshow(pattern, cmap="gray")
    ax.set_title(title)
    ax.axis("off")
for ax, pattern in zip(axes[1], mps_data):
    ax.hist(pattern.ravel(), bins=100)
    ax.set(xlabel="Gray level", ylabel="Frequency")
fig.tight_layout()
```



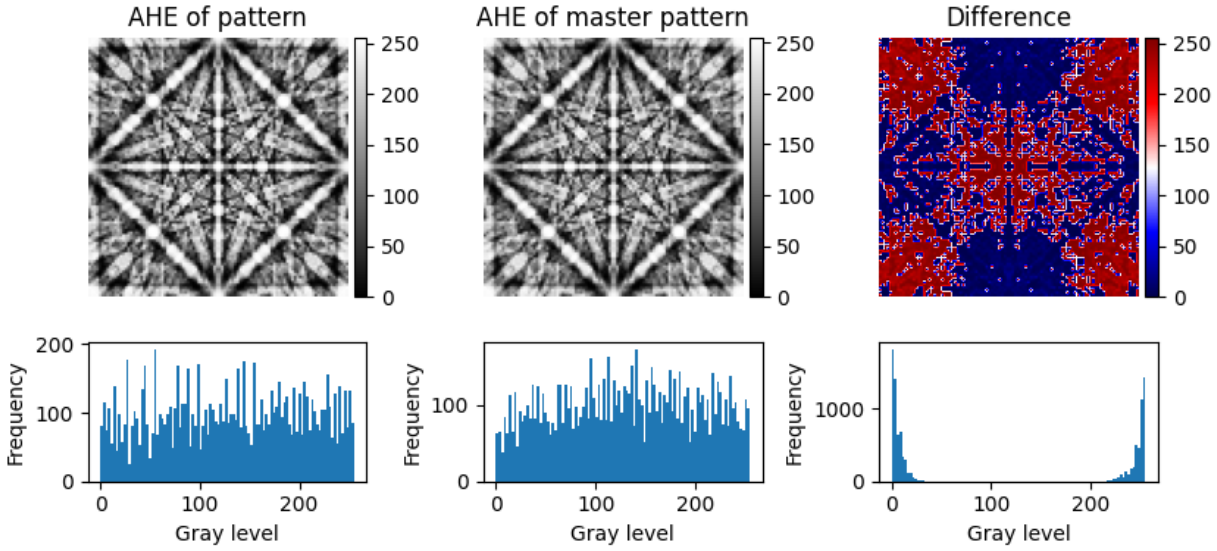

Let's show that intensities are approximately the same in patterns where one is equalized while the other is projected from a master pattern which itself is equalized.

```
# Project experimental patterns from each master pattern
det = kp.detectors.EBSDDetector((100, 100), sample_tilt=0)
r = Rotation.identity()
s1 = mp.get_patterns(r, det, energy=20, dtype_out="uint8", compute=True)
s2 = mp2.get_patterns(r, det, energy=20, dtype_out="uint8", compute=True)

# Adaptive histogram equalization of the first pattern
s1.adaptive_histogram_equalization()
```

(continues on next page)

(continued from previous page)

```
# Plot the patterns, their difference and their intensity histograms
patterns = [s1.data, s2.data, (s1 - s2).data]
fig, axes = plt.subplots(2, 3, figsize=(8.5, 4), height_ratios=[3, 1.5])
for ax, pattern, title, cmap in zip(
    axes[0],
    patterns,
    ["AHE of pattern", "AHE of master pattern", "Difference"],
    ["gray", "gray", "seismic"],
):
    im = ax.imshow(pattern.squeeze(), cmap=cmap)
    ax.set_title(title)
    ax.axis("off")
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.05)
    plt.colorbar(im, cax=cax)
for ax, pattern in zip(axes[1], patterns):
    ax.hist(pattern.ravel(), bins=100)
    ax.set(xlabel="Gray level", ylabel="Frequency")
fig.tight_layout()
```




**Total running time of the script:** (0 minutes 5.482 seconds)

**Estimated memory usage:** 12 MB

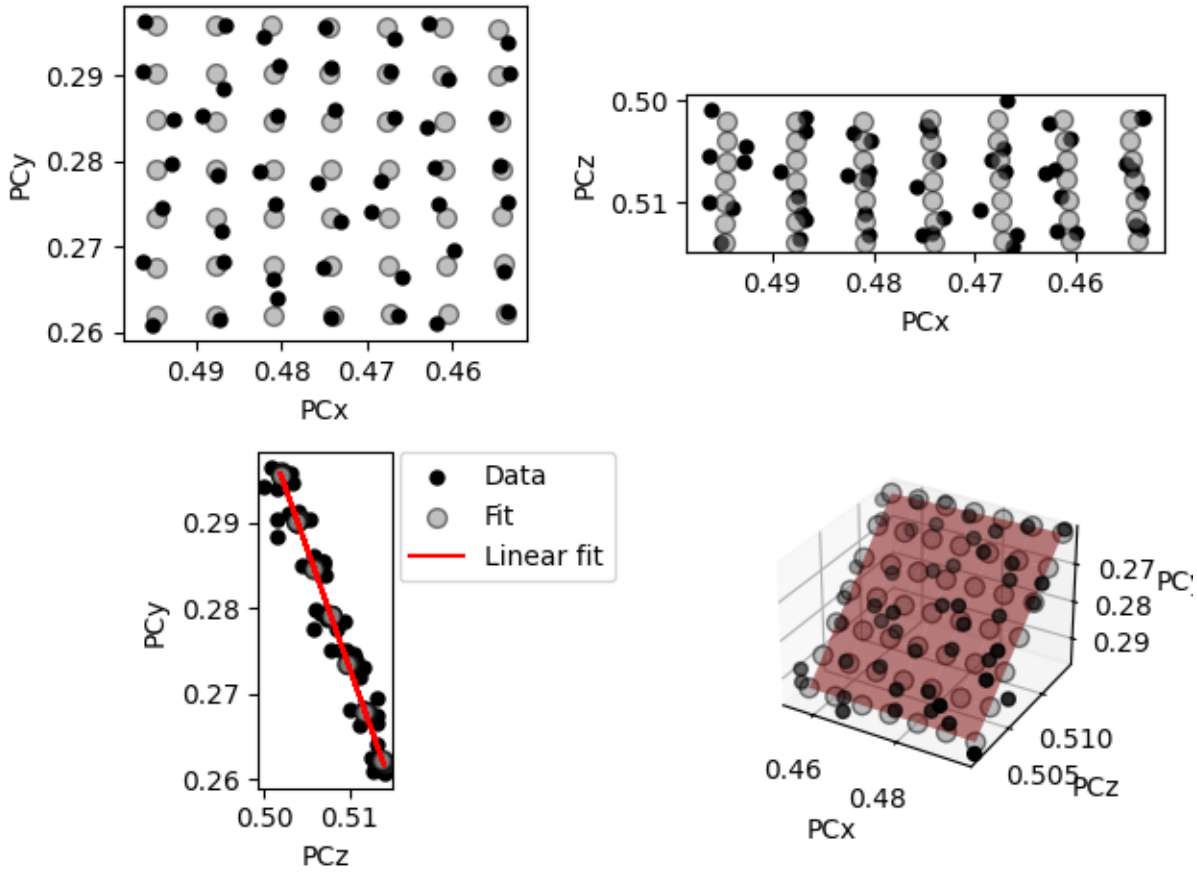
### Reference frames

These examples cover use of tools like the *EBSDDetector* for understanding reference frames relevant to EBSD.

### Fit a plane to selected projection centers

This example shows how to fit a plane to selected projection centers (PCs) using a projective transformation, following [Winkelmann *et al.*, 2020].

To test the fit, we add some noise to realistic projection center (PC) values (PCx, PCy, PCz) and fit a plane to a few of the PCs. The realistic PCs are extrapolated from a PC in the upper left corner of a map, assuming a nominal sample tilt of 70 degrees, a detector tilt of 0 degrees, a detector pixel size of 70 microns and a sample step size of 50 microns.




```
Max error in (PCx, PCy, PCz): [0.00233311 0.00244245 0.00243886]
Estimated sample tilt [deg]: 70.39
```

```
import kikuchipy as kp
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import numpy as np


plt.rcParams["font.size"] = 10

# Create an initial detector with one PC assumed to be for the upper
# left corner of a map
det0 = kp.detectors.EBSDDetector(
    shape=(480, 640),
    pc=(0.5, 0.3, 0.5),
    sample_tilt=70,
    tilt=0,
    px_size=70,
)

# Extrapolate a map of PCs
nav_shape = (30, 45)
det1 = det0.extrapolate_pc(
    pc_indices=[0, 0],
    navigation_shape=nav_shape,
    step_sizes=(50, 50),
)

# Add random noise
rng = np.random.default_rng()
dev = 0.002
det1.pcx += rng.uniform(-dev, dev, det1.navigation_size).reshape(nav_shape)
det1.pcy += rng.uniform(-dev, dev, det1.navigation_size).reshape(nav_shape)
det1.pcz += rng.uniform(-dev, dev, det1.navigation_size).reshape(nav_shape)

# Extract a (7, 7) grid of PCs
grid_shape = (7, 7)
pc_indices = kp.signals.util.grid_indices(grid_shape, nav_shape=nav_shape)
det2 = det1.deepcopy()
det2.pc = det2.pc[tuple(pc_indices)].reshape(grid_shape + (3,))

# Get a plane of PCs and plot the match at the same time
map_indices = np.stack(np.indices(nav_shape))
det_fit = det2.fit_pc(pc_indices=pc_indices, map_indices=map_indices)

# Inspect the max. error and sample tilt
max_err = abs(det_fit.pc_flattened - det1.pc_flattened).max(axis=0)
print("Max error in (PCx, PCy, PCz):", max_err)
print(f"Estimated sample tilt [deg]: {det_fit.sample_tilt:.2f}")
```

**Total running time of the script:** (0 minutes 0.719 seconds)

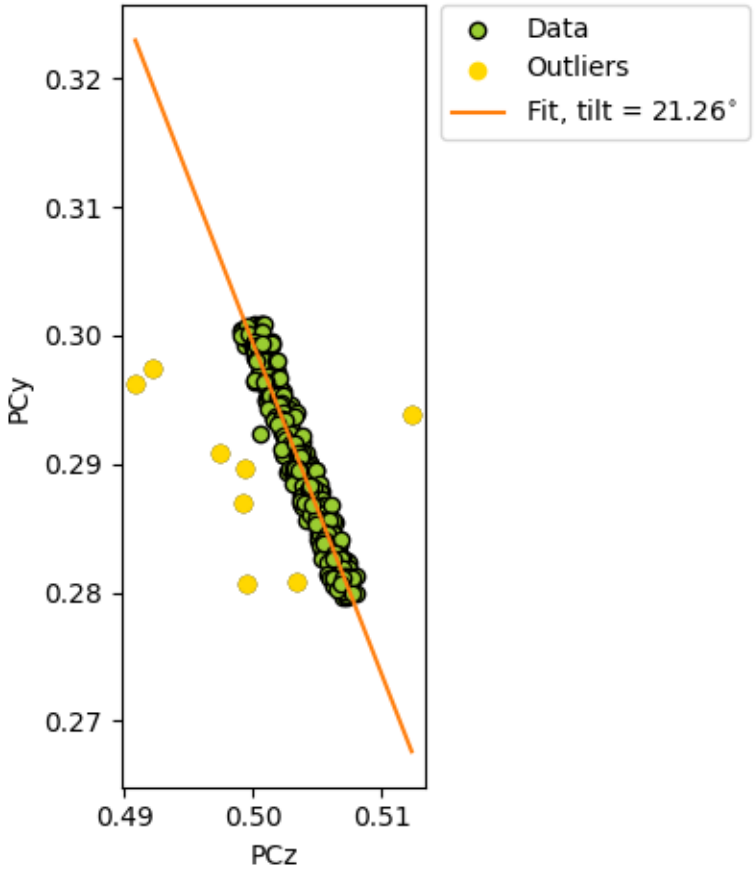
**Estimated memory usage:** 10 MB

### Estimate tilt about the detector x axis

This example shows how to (robustly) estimate the tilt about the detector $X_d$ axis which brings the sample plane normal into coincidence with the detector plane normal (but in the opposite direction) [Winkelmann *et al.*, 2020].

The estimate is found using `estimate_xtilt()` which performs linear regression of `pcz` vs. `pcy`.

To test the estimation, we add some noise to realistic projection center (PC) values (PCx, PCy, PCz). The realistic PCs are extrapolated from a PC in the upper left corner of a map, assuming a nominal sample tilt of 70 degrees, a detector tilt of 0 degrees, a detector pixel size of 70 microns and a sample step size of 50 microns.




```
True/estimated tilt about detector x [deg]: 20.00/21.26
8/10 of added outliers detected
```

```
import kikuchipy as kp
import numpy as np


# Create an initial detector with one PC assumed to be for the upper
# left corner of a map
```

(continues on next page)

(continued from previous page)

```
det0 = kp.detectors.EBSDDetector(
    shape=(480, 480),
    pc=(0.5, 0.3, 0.5),
    sample_tilt=70,
    tilt=0,
    px_size=70,
)

# Extrapolate a map of PCs
nav_shape = (15, 20)
det = det0.extrapolate_pc(
    pc_indices=[0, 0],
    navigation_shape=nav_shape,
    step_sizes=(50, 50),
)

# Add +/- 0.001 as random noise to PCy and PCz
rng = np.random.default_rng()
det.pcy += rng.uniform(-0.001, 0.001, det.navigation_size).reshape(nav_shape)
det.pcz += rng.uniform(-0.001, 0.001, det.navigation_size).reshape(nav_shape)

# Add outliers by adding more noise to PCz
outlier_idx1d = rng.choice(det.navigation_size, 10, replace=False)
is_outlier = np.zeros(det.navigation_size, dtype=bool)
is_outlier[outlier_idx1d] = True
noise_outlier = rng.uniform(-0.01, 0.01, outlier_idx1d.size)
outlier_idx2d = np.unravel_index(outlier_idx1d, shape=det.navigation_shape)
det.pcz[outlier_idx2d] += noise_outlier

# Robust estimation by detecting outliers
xtilt, outlier_detected_2d = det.estimate_xtilt(
    detect_outliers=True, degrees=True, return_outliers=True
)

# Print true tilt and estimated tilt
true_tilt = 90 - det.sample_tilt + det.tilt
print(f"True/estimated tilt about detector x [deg]: {true_tilt:.2f}/{xtilt:.2f}")

outlier_idx2d_detected = np.where(outlier_detected_2d)
outlier_idx1d_detected = np.ravel_multi_index(
    outlier_idx2d_detected, det.navigation_shape
)
correct_outliers = np.isin(outlier_idx1d, outlier_idx1d_detected)
print(f"{correct_outliers.sum()}/{outlier_idx1d.size} of added outliers detected")
```

**Total running time of the script:** (0 minutes 0.692 seconds)

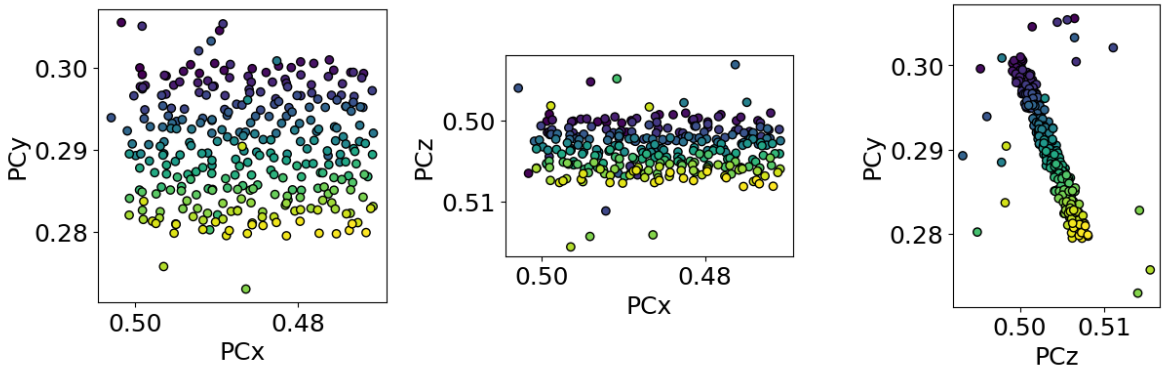
**Estimated memory usage:** 10 MB

### Estimate tilts about the detector x and z axis

This example shows how to (robustly) estimate the tilts about the detector $X_d$ and $Z_d$ axes, which bring the sample plane normal into coincidence with the detector plane normal (but in the opposite direction) [Winkelmann *et al.*, 2020].

Estimates are found using *estimate_xtilt_ztilt()*, which fits a hyperplane to *pc* using singular value decomposition.

To test the estimations, we add some noise to realistic projection center (PC) values (PCx, PCy, PCz). The realistic PCs are extrapolated from a PC in the upper left corner of a map, assuming a nominal sample tilt of 70 degrees, a detector tilt of 0 degrees, a detector pixel size of 70 microns and a sample step size of 50 microns.




```
True/estimated tilt about detector x [deg]: 20.00/19.21
True/estimated tilt about detector z [deg]: 0/-1.98
```

```
import kikuchipy as kp
import matplotlib.pyplot as plt
import numpy as np


plt.rcParams["font.size"] = 18

# Create an initial detector with one PC assumed to be for the upper
# left corner of a map
det0 = kp.detectors.EBSDDetector(
    shape=(480, 480),
    pc=(0.5, 0.3, 0.5),
    sample_tilt=70,
    tilt=0,
    px_size=70,
)

# Extrapolate a map of PCs
nav_shape = (15, 20)
nav_size = np.prod(nav_shape)
```

(continues on next page)

(continued from previous page)

```
det = det0.extrapolate_pc(
    pc_indices=[0, 0],
    navigation_shape=nav_shape,
    step_sizes=(50, 50),
)

# Add +/- 0.0025 as random noise
dev_noise = 0.001
rng = np.random.default_rng()
det.pcx += rng.uniform(-dev_noise, dev_noise, nav_size).reshape(nav_shape)
det.pcy += rng.uniform(-dev_noise, dev_noise, nav_size).reshape(nav_shape)
det.pcz += rng.uniform(-dev_noise, dev_noise, nav_size).reshape(nav_shape)

# Add outliers by adding more noise
dev_outlier = 0.01
n_outliers = 20
outlier_idx1d = rng.choice(nav_size, n_outliers, replace=False)
is_outlier = np.zeros(nav_size, dtype=bool)
is_outlier[outlier_idx1d] = True
outlier_idx2d = np.unravel_index(outlier_idx1d, shape=det.navigation_shape)
det.pcx[outlier_idx2d] += rng.uniform(-dev_outlier, dev_outlier, n_outliers)
det.pcy[outlier_idx2d] += rng.uniform(-dev_outlier, dev_outlier, n_outliers)
det.pcz[outlier_idx2d] += rng.uniform(-dev_outlier, dev_outlier, n_outliers)

# Plot PC values
det.plot_pc("scatter")

# Robust estimation by detecting outliers
xtilt, ztilt = det.estimate_xtilt_ztilt(degrees=True)

# Print true tilt and estimated tilt
true_xtilt = 90 - det.sample_tilt + det.tilt
print(f"True/estimated tilt about detector x [deg]: {true_xtilt:.2f}/{xtilt:.2f}")
print(f"True/estimated tilt about detector z [deg]: {0}/{ztilt:.2f}")
```

**Total running time of the script:** (0 minutes 0.589 seconds)

**Estimated memory usage:** 9 MB

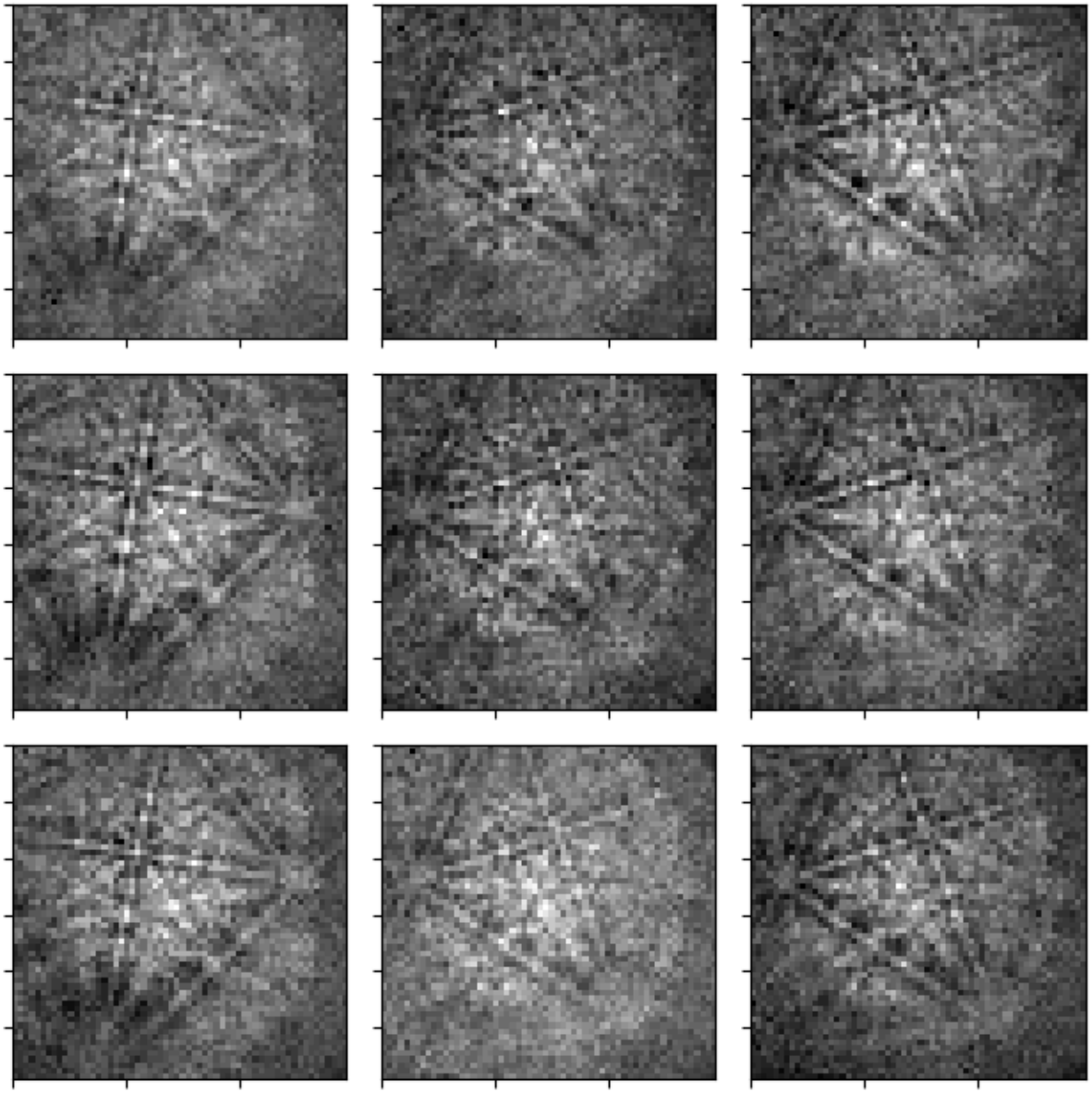
### Selecting data

These examples cover selection of data via extraction a subset of the navigation and/or signal axes.

### Crop navigation axes

This example shows various ways to crop the navigation axes of an *EBSD* signal using HyperSpy's `inav` slicer and *crop()* method (see Indexing for details).

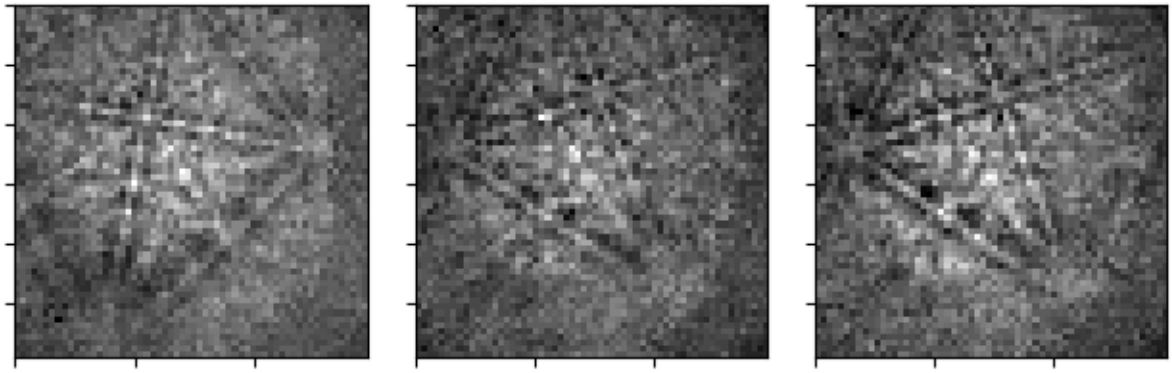
```
import hyperspy.api as hs
import kikuchipy as kp


# Import data
s = kp.data.nickel_ebsd_small()
s.remove_static_background(show_progressbar=False)

# Inspect data and attributes
plot_kwds = dict(axes_decor=None, label=None, colorbar=None, tight_layout=True)
_ = hs.plot.plot_images(s, **plot_kwds)
print(s)
print(s.xmap.shape)
print(s.detector.navigation_shape)
```



```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
(3, 3)
(3, 3)
```

Get a new signal with the patterns in the first row using `inav`. Note how the *xmap* and *detector* attributes are updated.

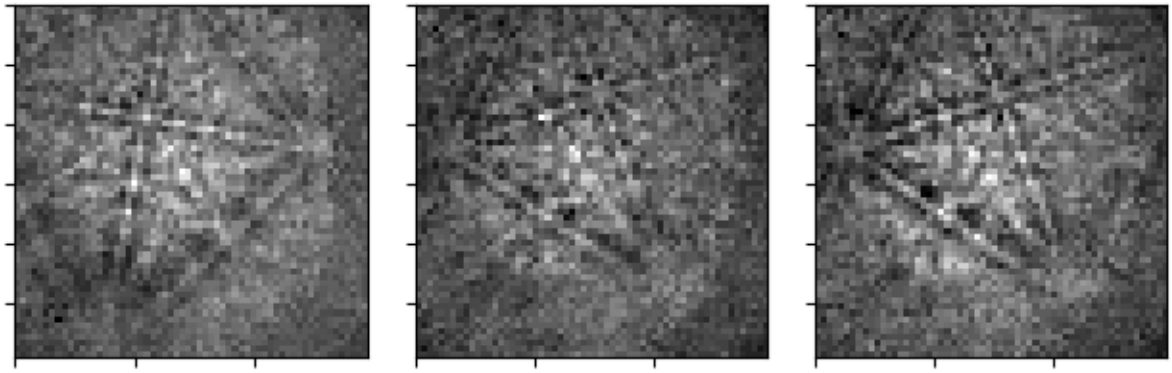
```
s2 = s.inav[:, 0]

_ = hs.plot.plot_images(s2, **plot_kwds)
print(s2)
print(s2.xmap.shape)
print(s2.detector.navigation_shape)
```



```
<EBSD, title: patterns Scan 1, dimensions: (3|60, 60)>
(3,)
(3,)
```

Get the first column using `crop()`, which overwrites the signal inplace

```
s3 = s.deepcopy()
s3.crop(1, start=0, end=1)

_ = hs.plot.plot_images(s3, **plot_kwds)
print(s3)
print(s3.xmap.shape)
print(s3.detector.navigation_shape)
```



```
<EBSD, title: patterns Scan 1, dimensions: (3, 1|60, 60)>
(1, 3)
(1, 3)
```

While `inav` returned a signal with only one navigation dimension, `crop()` left a single row. We can remove this `(1,)` dimension using `squeeze()`, but note that the custom `EBSD` attributes are not cropped accordingly

```
s4 = s3.squeeze()

print(s4)
print(s4.xmap.shape)
print(s4.detector.navigation_shape)
```

```
<EBSD, title: patterns Scan 1, dimensions: (3|60, 60)>
(1, 3)
(1, 3)
```

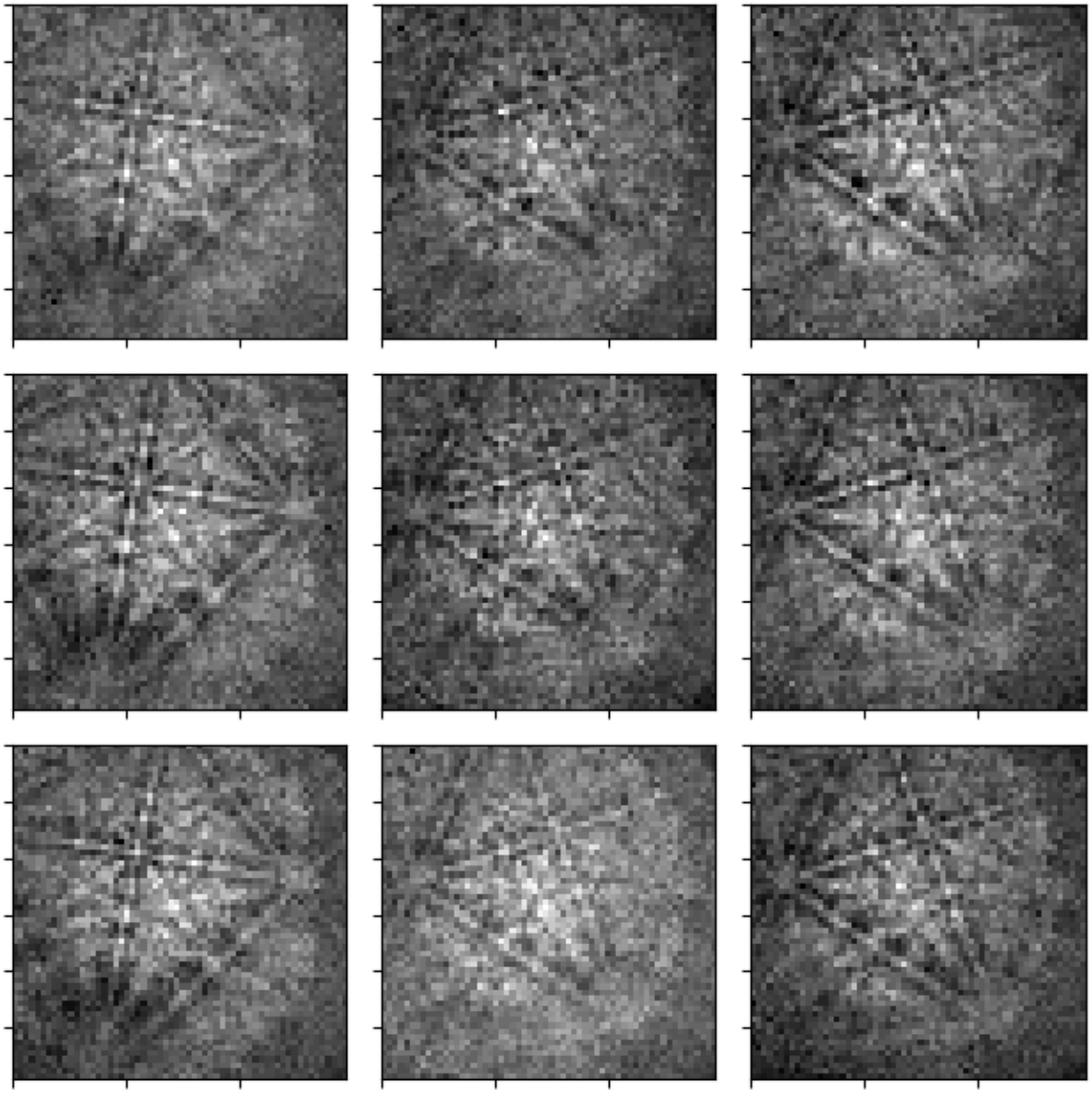
**Total running time of the script:** (0 minutes 2.946 seconds)

**Estimated memory usage:** 9 MB

### Crop signal axes

This example shows various ways to crop the signal axes of an *EBSD* signal using HyperSpy's `isig` slicer and the *crop()* and `crop_image()` methods (see Indexing for details).

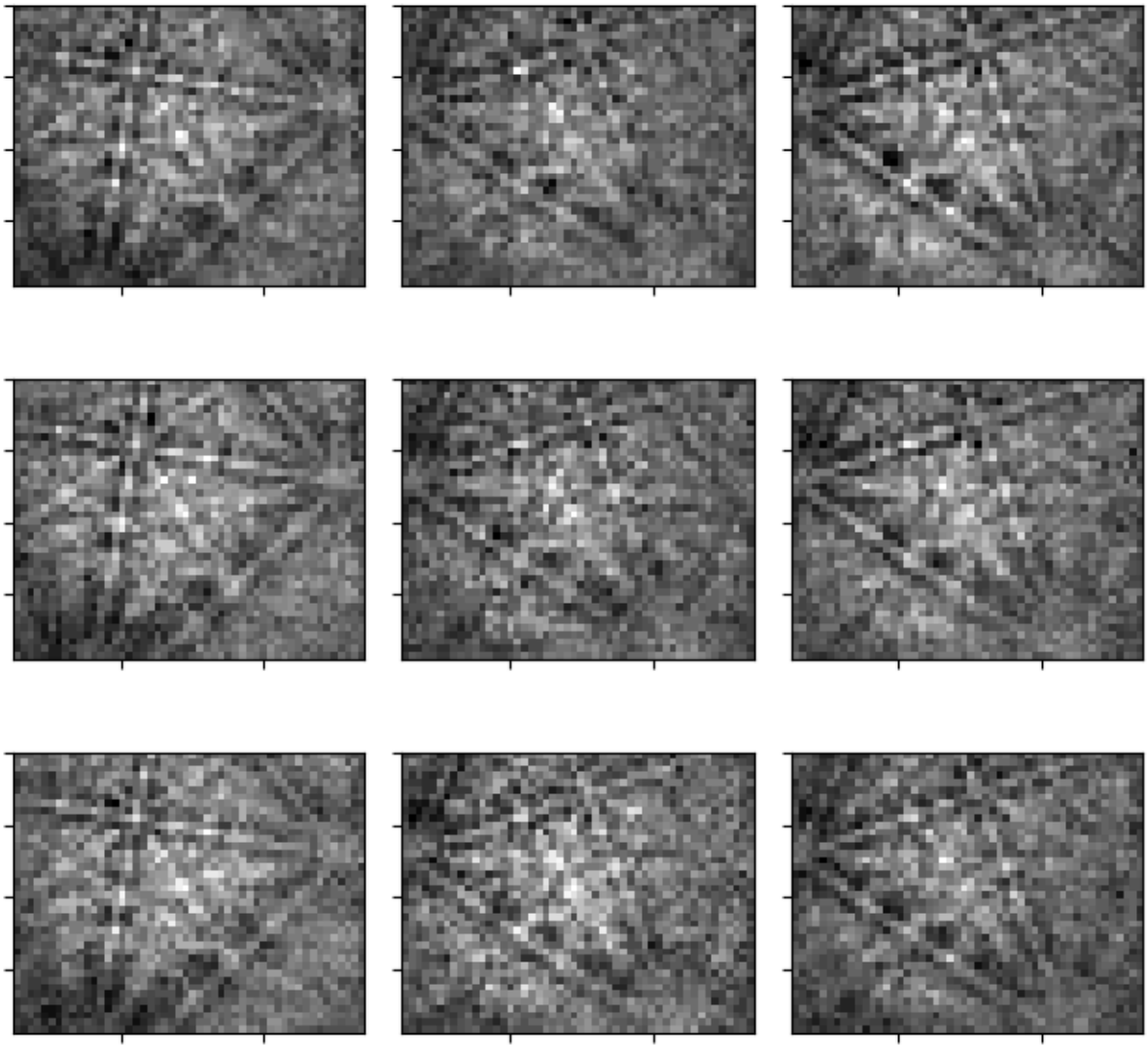
```
import hyperspy.api as hs
import kikuchipy as kp


# Import data
s = kp.data.nickel_ebsd_small()
s.remove_static_background(show_progressbar=False)

# Inspect data and attributes
plot_kwds = dict(axes_decor=None, label=None, colorbar=None, tight_layout=True)
_ = hs.plot.plot_images(s, **plot_kwds)
print(s)
print(s.static_background.shape)
print(s.detector)
```



```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
(60, 60)
EBSDDetector (60, 60), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.425, 0.213, 0.
↪501)
```

Get a new signal, removing the first and last ten rows of pixels and first and last five columns of pixels. Note how the *static_background* and *detector* attributes are updated.

```
s2 = s.isig[5:55, 10:50]

_ = hs.plot.plot_images(s2, **plot_kwds)
print(s2)
print(s2.static_background.shape)
print(s2.detector)
```



```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|50, 40)>
(40, 50)
EBSDDetector (40, 50), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.41, 0.07, 0.
↪751)
```

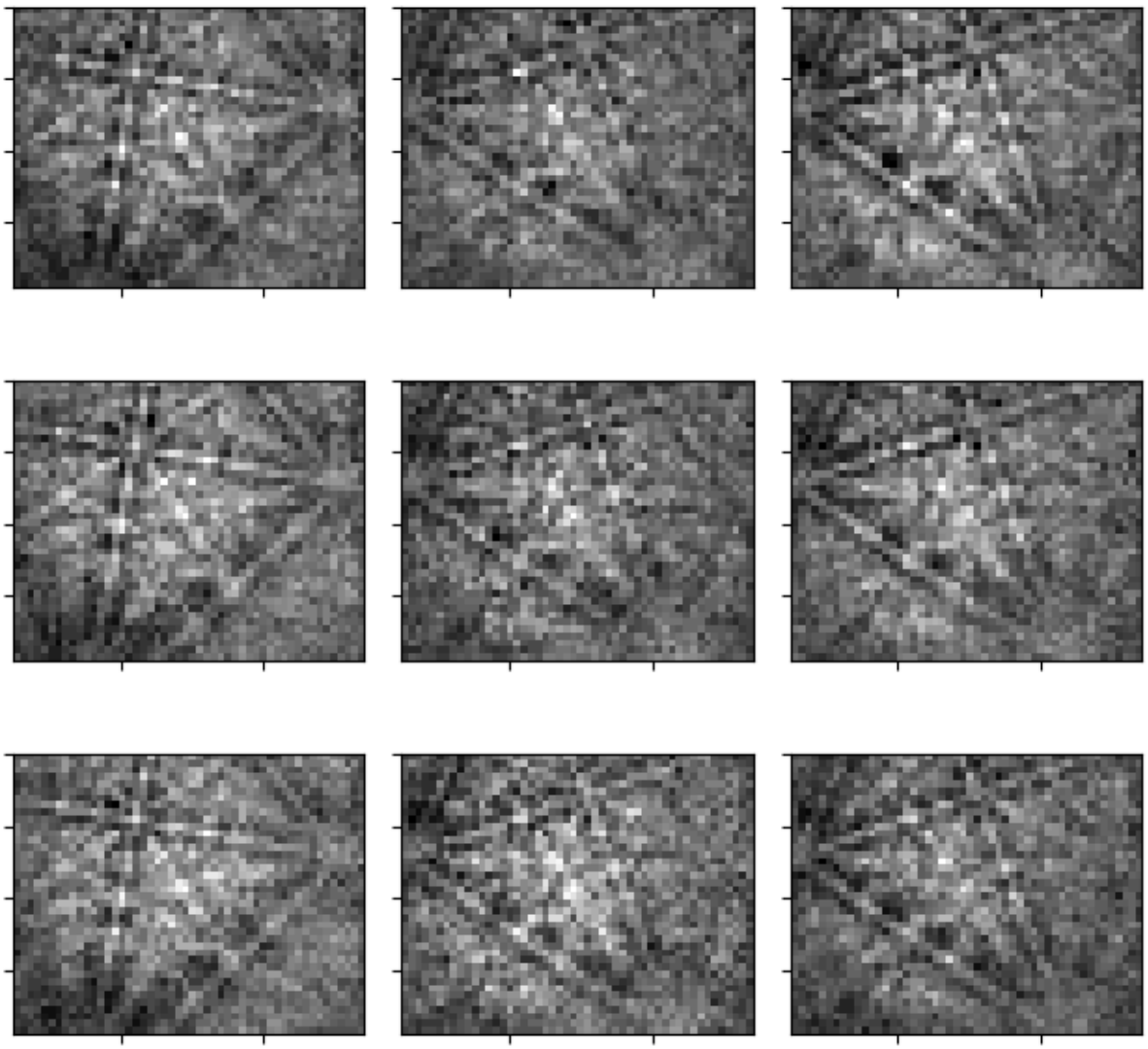
Do the same inplace using *crop()*

```
s3 = s.deepcopy()
s3.crop(2, start=5, end=55)
s3.crop("dy", start=10, end=50)

_ = hs.plot.plot_images(s3, **plot_kwds)
print(s3)
print(s3.static_background.shape)
```

(continues on next page)

(continued from previous page)

```
print(s3.detector)
```



```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|50, 40)>
(40, 50)
EBSDDetector (40, 50), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.41, 0.07, 0.
↪751)
```

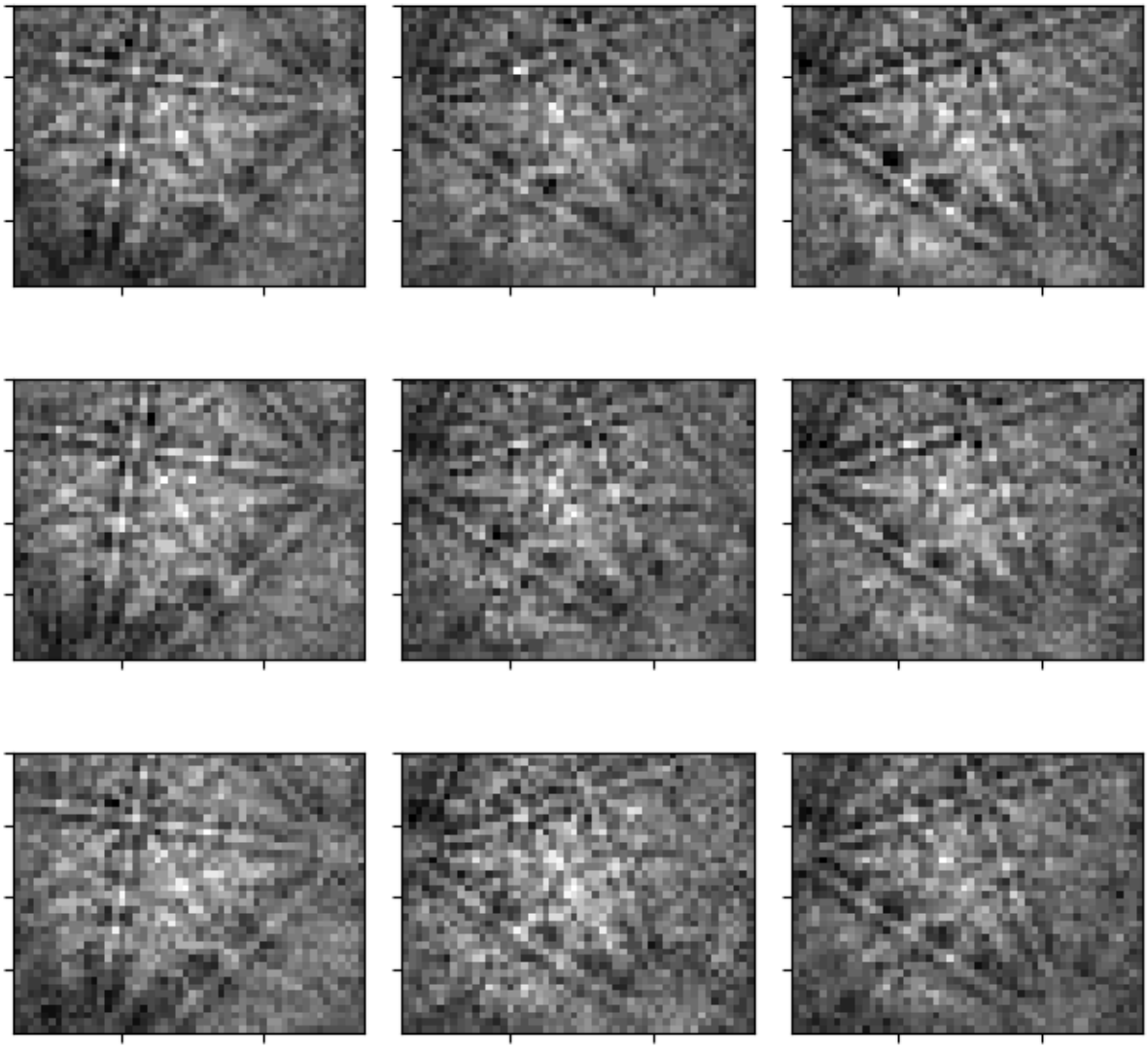
Do the same inplace using `crop_image()`

```
s4 = s.deepcopy()
s4.crop_image(top=10, bottom=50, left=5, right=55)

_ = hs.plot.plot_images(s4, **plot_kwds)
```

(continues on next page)

(continued from previous page)

```
print(s4)
print(s4.static_background.shape)
print(s4.detector)
```



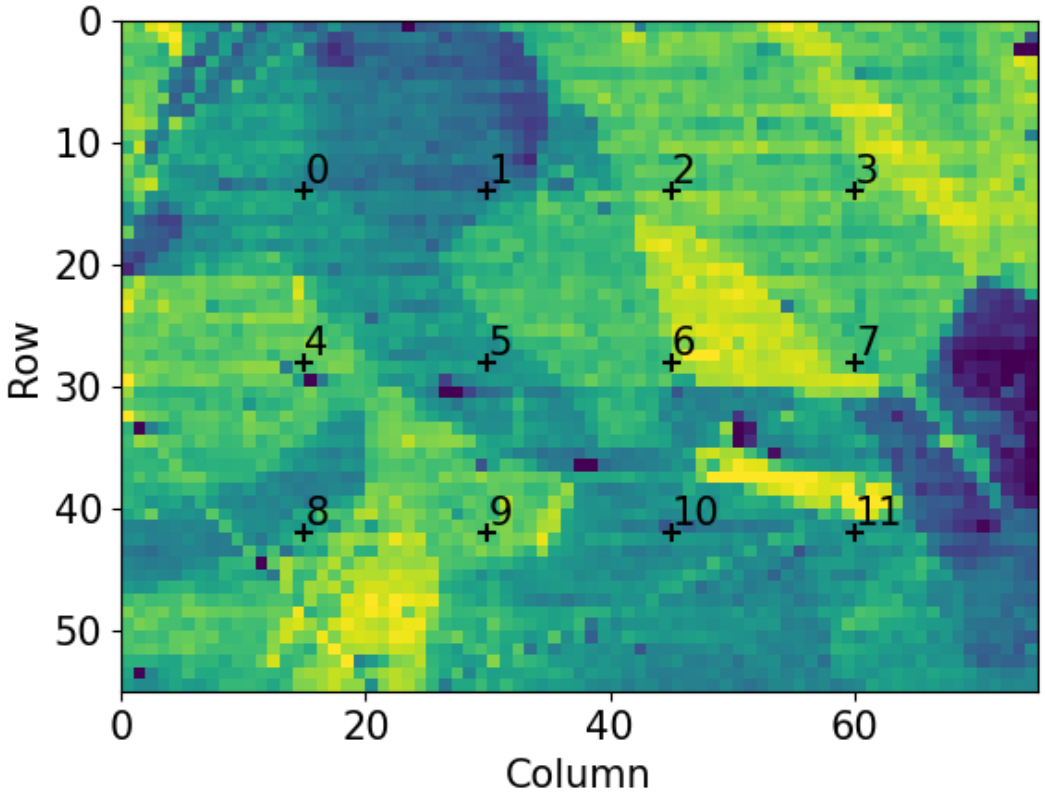
```
<EBSD, title: patterns Scan 1, dimensions: (3, 3|50, 40)>
(40, 50)
EBSDDetector (40, 50), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.41, 0.07, 0.
↪751)
```

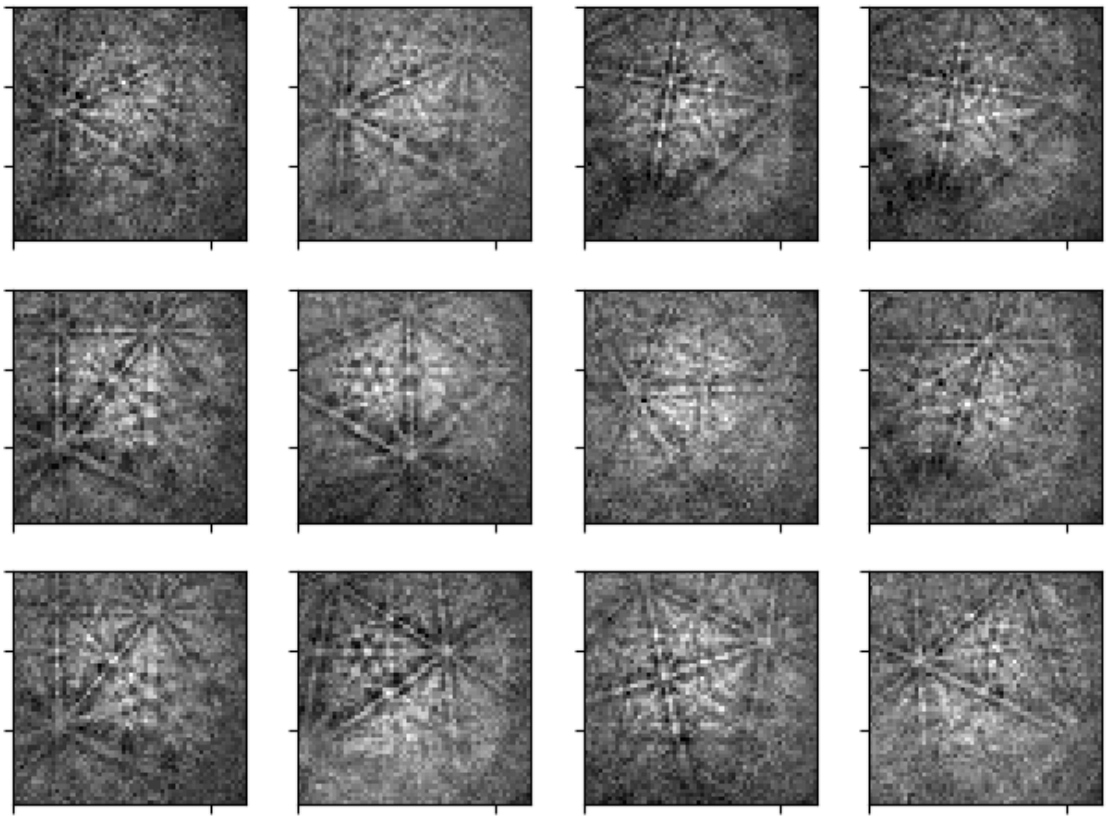
**Total running time of the script:** (0 minutes 2.958 seconds)

**Estimated memory usage:** 12 MB

### Extract patterns from a grid

This example shows how to extract *EBSD* patterns from positions in a grid evenly spaced in navigation space.





```
<LazyEBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
<LazyEBSD, title: patterns Scan 1, dimensions: (4, 3|60, 60)>
```

```
import hyperspy.api as hs
import kikuchipy as kp
import matplotlib.pyplot as plt


plt.rcParams["font.size"] = 15

# Silence progressbars
hs.preferences.General.show_progressbar = False

# Import data (lazily)
s = kp.data.nickel_ebsd_large(lazy=True)
print(s)

# Extract data, also getting the grid positions
s2, idx = s.extract_grid((4, 3), return_indices=True)
print(s2)
```

(continues on next page)

(continued from previous page)

```
# Get virtual backscatter electron (VBSE) image from the intensities from the
# center of the detector, also slightly stretching the contrast
roi = hs.roi.RectangularROI(left=20, top=20, right=40, bottom=40)
vbse_img = s.get_virtual_bse_intensity(roi)
vbse_img.compute()  # Drop if data was not loaded lazily
vbse_img.rescale_intensity(dtype_out="float32", percentiles=(0.5, 99.5))

# Plot grid of extracted patterns
kp.draw.plot_pattern_positions_in_map(
    idx.reshape(2, -1).T,
    roi_shape=s.axes_manager.navigation_shape[::-1],
    roi_image=vbse_img.data,
)

# Plot extracted patterns
s2.remove_static_background()
_ = hs.plot.plot_images(
    s2, per_row=4, axes_decor=None, label=None, colorbar=None, tight_layout=True
)
```

**Total running time of the script:** (0 minutes 1.343 seconds)

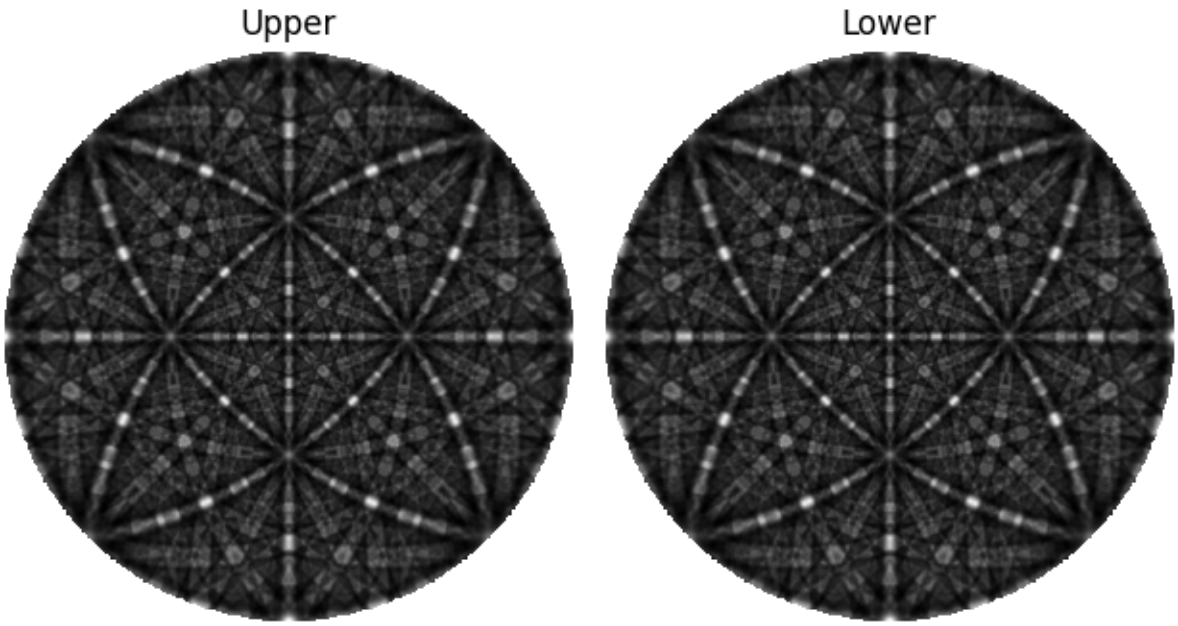
**Estimated memory usage:** 9 MB

### Visualization

These examples cover visualization of Kikuchi patterns and derived maps.

### Plot nice master pattern image

This example shows you how to plot a nice and clean image of an EBSD master pattern. More details are given in the *visualization tutorial*.




```
import kikuchipy as kp
import matplotlib.pyplot as plt
import numpy as np


# Load both hemispheres of master pattern in stereographic projection
mp = kp.data.nickel_ebsd_master_pattern_small(hemisphere="both")

# Extract the underlying data of both hemipsheres and mask out the
# surrounding black pixels
data = mp.data.astype("float32")
mask = data[0] == 0
data[:, mask] = np.nan

# Plot both hemispheres with labels
fig, (ax0, ax1) = plt.subplots(ncols=2)
ax0.imshow(data[0], cmap="gray")
ax1.imshow(data[1], cmap="gray")
ax0.axis("off")
ax1.axis("off")
ax0.set_title("Upper")
ax1.set_title("Lower")
fig.tight_layout()
```

**Total running time of the script:** (0 minutes 0.625 seconds)

**Estimated memory usage:** 9 MB

## 1.4 Bibliography

This is a list of works refered to in the *User guide* or *API reference*:

## 1.5 Applications

If you are using kikuchipy in your scientific research, please help our scientific visibility by citing the Zenodo DOI: https://doi.org/10.5281/zenodo.3597646.

This is a tentative list of works using kikuchipy. If you think your work should be listed here, please raise an issue on GitHub or contact the developers. Most of these works are also listed when searching for `"kikuchipy"` on Google Scholar.

### 1.5.1 2023

- A. V. Bugten, L. Michels, R. B. Brurok, C. Hartung, E. Ott, L. Vines, Y. Li, L. Arnberg and M. Di Sabatino, "The Role of Boron in Low Copper Spheroidal Graphite Irons," *Metallurgical and Materials Transactions A* **54** (2023). https://doi.org/10.1007/s11661-023-07014-y.
- O. W. Sandvik, A. M. Müller, H. W. Ånes, M. Zahn, J. He, M. Fiebig, T. Lottermoser, Th. Rojac, D. Meier and J. Schultheiß, "Pressure Control of Nonferroelastic Ferroelectric Domains in ErMnO3," *Nano Letters* (2023). https://doi.org/10.1021/acs.nanolett.3c01638 (arXiv).
- H. W. Ånes, A. T. J. van Helvoort and K. Marthinsen, "Orientation dependent pinning of (sub)grains by dispersoids during recovery and recrystallization in an Al-Mn alloy," *Acta Materialia* **248** (2023). https://doi.org/10.1016/j.actamat.2023.118761 (arXiv).
- T. Bergh, H. W. Ånes, R. Aune, S. Wenner, R. Holmestad, X. Ren and P. E. Vullum, "Intermetallic Phase Layers in Cold Metal Transfer Aluminium-Steel Welds with an Al-Si–Mn Filler Alloy," *Materials Transactions* **64(2)** (2023). https://doi.org/10.2320/matertrans.MT-LA2022046.

### 1.5.2 2022

- O. M. Akselsen, R. Bjørge, H. W. Ånes, X. Ren, and B. Nyhus, "Microstructure and Properties of Wire Arc Additive Manufacturing of Inconel 625," *Metals* **12(11)** (2022). https://doi.org/10.3390/met12111867
- H. W. Ånes, A. T. J. van Helvoort and K. Marthinsen, "Correlated subgrain and particle analysis of a recovered Al-Mn alloy by directly combining EBSD and backscatter electron imaging," *Materials Characterization* **193** (2022). https://doi.org/10.1016/j.matchar.2022.112228 (arXiv).
- J. Schultheiß, F. Xue, E. Roede, H. W. Ånes, F. H. Danmo, S. M. Selbach, L.-Q. Chen and D. Meier, "Confinement-driven inverse domain scaling in polycrystalline ErMnO3," *Advanced Materials*, **34** (2022). https://doi.org/10.1002/adma.202203449 (arXiv).

### 1.5.3 2021

- O. M. Akselsen, R. Bjørge, H. W. Ånes, X. Ren and B. Nyhus, "Effect of Sigma Phase in Wire Arc Additive Manufacturing of Superduplex Stainless Steel," *Metals* **11(12)** (2021). https://doi.org/10.3390/met11122045.

### 1.5.4 2020

- B. E. Sørensen, J. Hjelen, H. W. Ånes and T. Breivik, "Recent features in EBSD, including new trapezoidal correction for multi-mapping," In *IOP Conference Series: Materials Science and Engineering*, volume **891** IOP Publishing (2020). https://doi.org/10.1088/1757-899X/891/1/012021.
- H. W. Ånes, J. Hjelen, B. E. Sørensen, A. T. J. van Helvoort and K. Marthinsen, "Processing and indexing of electron backscatter patterns using open-source software," In *IOP Conference Series: Materials Science and Engineering*, volume **891** IOP Publishing (2020). https://doi.org/10.1088/1757-899X/891/1/012002.

## 1.6 Open datasets

**Note:** See the `data` module for data sets used in the tests or documentation of kikuchipy.

This is a non-exhaustive list of EBSD datasets openly available on the internet which can be read by kikuchipy:

- [Jackson *et al.*, 2019]
- [Shi, 2021]
- [Shi, 2022]
- [Wilkinson and Collins, 2018]
- [Ånes *et al.*, 2019]
- [Ånes *et al.*, 2022]
- [Ånes *et al.*, 2022]
- [Ånes *et al.*, 2022]

## 1.7 Related projects

This is a non-exhaustive list of related, open-source projects for analysis of EBSD data that users of kikuchipy might find useful:

- HyperSpy: Python library with tools for multi-dimensional data analysis. kikuchipy extends this library for EBSD analysis.
- EMsoft: Series of Fortran programs which, among numerous other tasks, can dynamically simulate EBSD, Electron Channeling Pattern (ECP) and Transmission Kikuchi Diffraction (TKD) master patterns.
- orix: Python library for handling crystal orientation mapping data. kikuchipy depends on this library for all operations with vectors, rotations and crystal symmetry.
- diffsims: Python library for simulating diffraction. kikuchipy depends on this library for handling of reciprocal lattice vectors.
- pyxem: Python library for multi-dimensional diffraction microscopy.

- PyEBSDIndex: Python library for Hough/Radon based EBSD indexing. kikuchipy depends on this library for Hough indexing.
- OpenECCI: GUI-based software for controlled Electron Channelling Contrast Imaging (ECCI) analysis of crystal defects in an SEM.
- MTEX: MATLAB toolbox for analyzing and modeling crystallographic textures by means of EBSD or pole figure data.
- DREAM.3D: C++ library to reconstruct, instatiate, quantify, mesh, handle, and visualize multidimensional (3D), multimodal data (mainly EBSD orientation data).
- AstroEBSD: MATLAB package with indexing tools for EBSD patterns.
- xcdskd: Python library with tools for Kikuchi Diffraction in the SEM, with example Jupyter Notebooks.
- OpenXY: MATLAB tool for cross-correlation analysis of EBSD patterns.
- DefDAP: Python package for correlating EBSD and high-resolution digital image correlation data.
- pycotem: Python package for working with crystal orientations in transmission electron microscopy.

CHAPTER

TWO

# API REFERENCE

**Release**: 0.10.dev0

**Date**: Nov 03, 2023

This reference manual describes the public functions, modules, and objects in kikuchipy. Many of the descriptions include brief examples. For learning how to use kikuchipy, see the *Examples* or *Tutorials*.

**Caution:** kikuchipy is in continuous development, meaning that some breaking changes and changes to this reference are likely with each release.

kikuchipy's import structure is designed to feel familiar to HyperSpy users. It is recommended to import functionality from the below list of functions and modules like this:

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
```

### Functions

| | |
|---|---|
| *load*(filename[, lazy]) | Load an *EBSD*, *EBSDMasterPattern* or *ECPMasterPattern* signal from one of the *Supported file formats*. |
| *set_log_level*(level) | Set level of kikuchipy logging messages. |

## 2.1 load

kikuchipy.**load**(*filename: str | Path, lazy: bool = False, **kwargs*) → *EBSD* | *EBSDMasterPattern* | *ECPMasterPattern* | List[*EBSD*] | List[*EBSDMasterPattern*] | List[*ECPMasterPattern*]

Load an *EBSD*, *EBSDMasterPattern* or *ECPMasterPattern* signal from one of the *Supported file formats*.

This function is a modified version of `hyperspy.io.load()`.

**Parameters**

**filename**
Name of file to load.

**lazy**
Open the data lazily without actually reading the data from disk until required. Allows opening arbitrary sized datasets. Default is `False`.

****kwargs**
Keyword arguments passed to the corresponding kikuchipy reader. See their individual documentation for available options.

**Returns**

**out**
Signal or a list of signals.

**Raises**

`IOError`
If the file was not found or could not be read.

### Examples

Import nine patterns from an HDF5 file in a directory `DATA_DIR`

```
>>> import kikuchipy as kp
>>> s = kp.load(DATA_DIR + "/patterns.h5")
>>> s
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
```

## 2.2 set_log_level

kikuchipy.**set_log_level**(*level: int | str*)

Set level of kikuchipy logging messages.

**Parameters**

**level**
Any value accepted by `logging.Logger.setLevel()`. Levels are `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"` and `"CRITICAL"`.

### Notes

See https://docs.python.org/3/howto/logging.html.

### Examples

Note that you might have to set the logging level of the root stream handler to display kikuchipy's debug messages, as this handler might have been initialized by another package

```
>>> import logging
>>> logging.root.handlers[0]
<StreamHandler <stderr> (INFO)>
>>> logging.root.handlers[0].setLevel("DEBUG")
```

```
>>> import kikuchipy as kp
>>> kp.set_log_level("DEBUG")
>>> s = kp.data.nickel_ebsd_master_pattern_small()
>>> s.set_signal_type("EBSD")
DEBUG:kikuchipy.signals._kikuchi_master_pattern:Delete custom attributes when␣
↪setting signal type
```

### Modules

| | |
|---|---|
| *data* | Example datasets for use when testing functionality. |
| *detectors* | Tools for handling the EBSD detector-sample geometry. |
| *draw* | Tools for use in plotting of signals. |
| *filters* | Pattern filters used on signals, e.g. for pattern averaging. |
| *imaging* | Imaging using the EBSD detector. |
| *indexing* | Tools for indexing of EBSD patterns by matching to a dictionary of simulated patterns. |
| *io* | Read and write signals from and to file. |
| *pattern* | Single pattern processing (used by signals). |
| *signals* | Experimental and simulated diffraction patterns and virtual backscatter electron images. |
| *simulations* | Simulations returned by a generator and handling of Kikuchi bands and zone axes. |

## 2.3 data

Example datasets for use when testing functionality.

Some datasets are packaged with the source code while others must be downloaded from the web. For more test datasets, see *Open datasets*.

Datasets are placed in a local cache, in the location returned from `pooch.os_cache("kikuchipy")` by default. The location can be overwritten with a global `KIKUCHIPY_DATA_DIR` environment variable.

With every new version of kikuchipy, a new directory of datasets with the version name is added to the cache directory. Any old directories are not deleted automatically, and should then be deleted manually if desired.

### Functions

| | |
|---|---|
| *ebsd_master_pattern*(phase[, allow_download, ...]) | EBSD master pattern of an available phase of (1001, 1001) pixel resolution in both the square Lambert or stereographic projection. |
| *ni_gain*([number, allow_download, ...]) | EBSD dataset of (149, 200) patterns of (60, 60) pixels from polycrystalline recrystallized Ni, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019]. |
| *ni_gain_calibration*([number, ...]) | A few EBSD calibration patterns of (480, 480) pixels from polycrystalline recrystallized Ni, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019]. |
| *nickel_ebsd_large*([allow_download, ...]) | 4125 EBSD patterns in a (55, 75) navigation shape of (60, 60) pixels from nickel, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019]. |
| *nickel_ebsd_master_pattern_small*(**kwargs) | (401, 401) `uint8` square Lambert or stereographic projection of the northern and southern hemisphere of a nickel master pattern at 20 keV accelerating voltage. |
| *nickel_ebsd_small*(**kwargs) | Ni EBSD patterns in a (3, 3) navigation shape of (60, 60) pixels from nickel, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019]. |
| *si_wafer*([allow_download, show_progressbar]) | EBSD dataset of (50, 50) patterns of (480, 480) pixels from a single crystal silicon wafer, acquired on a NORDIF UF-420 detector [Ånes *et al.*, 2022]. |
| *si_ebsd_moving_screen*([distance, ...]) | One EBSD pattern of (480, 480) pixels from a single crystal silicon sample, acquired on a NORDIF UF-420 detector [Ånes *et al.*, 2022]. |

## 2.3.1 ebsd_master_pattern

kikuchipy.data.**ebsd_master_pattern**(*phase: str*, *allow_download: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSDMasterPattern*

EBSD master pattern of an available phase of (1001, 1001) pixel resolution in both the square Lambert or stereographic projection.

Master patterns were simulated with *EMsoft* [Callahan and De Graef, 2013].

**Parameters**

**phase**
Name of available phase. Options are (see *Notes* for details): ni, al, si, austenite, ferrite, steel_chi, steel_sigma.

**allow_download**
Whether to allow downloading the dataset from the internet to the local cache with the pooch Python package. Default is `False`.

**show_progressbar**
Whether to show a progressbar when downloading. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *load()*.

**Returns**

**ebsd_master_pattern_signal**
EBSD master pattern signal.

**See also:**

*nickel_ebsd_master_pattern_small*, *ebsd_master_pattern*

### Notes

The master patterns are downloaded in HDF5 files (carrying a CC BY 4.0 license) from Zenodo.

| phase | Name | Symbol | Energy [keV] | File size [GB] | Zenodo DOI |
|---|---|---|---|---|---|
| ni | nickel | Ni | 5-20 | 0.3 | 10.5281/zenodo.7628443 |
| al | aluminium | Al | 10-20 | 0.2 | 10.5281/zenodo.7628365 |
| si | silicon | Si | 5-20 | 0.3 | 10.5281/zenodo.7498729 |
| austenite | austenite | $\gamma$-Fe | 10-20 | 0.3 | 10.5281/zenodo.7628387 |
| ferrite | ferrite | $\alpha$-Fe | 5-20 | 0.3 | 10.5281/zenodo.7628394 |
| steel_chi | chi | Fe36Cr15Mo7 | 10-20 | 0.6 | 10.5281/zenodo.7628417 |
| steel_sigma | sigma | FeCr | 5-20 | 1.5 | 10.5281/zenodo.7628443 |

### Examples

Import master pattern in the stereographic projection

```
>>> import kikuchipy as kp
>>> s = kp.data.ebsd_master_pattern("ni", hemisphere="both")
>>> s
<EBSDMasterPattern, title: ni_mc_mp_20kv, dimensions: (16, 2|1001, 1001)>
>>> s.projection
'stereographic'
```

## 2.3.2 ni_gain

kikuchipy.data.**ni_gain**(*number: int = 1*, *allow_download: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSD*

EBSD dataset of (149, 200) patterns of (60, 60) pixels from polycrystalline recrystallized Ni, acquired on a NORDIF UF-1100 detector [Åness *et al.*, 2019].

Ten datasets are available from the same region of interest, acquired with increasing gain on the detector, from no gain to maximum gain.

**Parameters**

**number**
Dataset number 1-10. The camera gains in dB are 0, 3, 6, 9, 12, 15, 17, 20, 22 and 24. Default is dataset number 1, acquired without detector gain.

**allow_download**
Whether to allow downloading the dataset from the internet to the local cache with the pooch Python package. Default is `False`.

**show_progressbar**
Whether to show a progressbar when downloading. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *load()*.

**Returns**

**ebsd_signal**
EBSD signal.

**See also:**

*ni_gain_calibration*, *nickel_ebsd_small*, *nickel_ebsd_large*

### Notes

The datasets are hosted in the Zenodo repository https://doi.org/10.5281/zenodo.7497682 and comprise about 100 MB each as zipped files and about 116 MB when unzipped. Each zipped file is deleted after it is unzipped.

The datasets carry a CC BY 4.0 license.

### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.ni_gain(allow_download=True, lazy=True)
>>> s
<LazyEBSD, title: Pattern, dimensions: (200, 149|60, 60)>
```

## 2.3.3 ni_gain_calibration

kikuchipy.data.**ni_gain_calibration**(*number: int = 1*, *allow_download: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSD*

A few EBSD calibration patterns of (480, 480) pixels from polycrystalline recrystallized Ni, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019].

The patterns are used to calibrate the detector-sample geometry of the datasets in *ni_gain()*. The calibration patterns were acquired with no gain on the detector.

**Parameters**

**number**
Dataset number 1-10. Default is dataset number 1, i.e. the calibration patterns used to calibrate the detector-sample geometry for the dataset acquired without detector gain.

**allow_download**
Whether to allow downloading the dataset from the internet to the local cache with the pooch Python package. Default is `False`.

**show_progressbar**
Whether to show a progressbar when downloading. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *load()*.

**Returns**

**ebsd_signal**
EBSD signal.

**See also:**

*ni_gain*, *nickel_ebsd_small*, *nickel_ebsd_large*

### Notes

The datasets are hosted in the Zenodo repository https://doi.org/10.5281/zenodo.7497682 and comprise about 100 MB each as zipped files and about 116 MB when unzipped. Each zipped file is deleted after it is unzipped.

The datasets carry a CC BY 4.0 license.

### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.ni_gain_calibration(allow_download=True, lazy=True)
>>> s
<LazyEBSD, title: Calibration patterns, dimensions: (9|480, 480)>
```

## 2.3.4 nickel_ebsd_large

kikuchipy.data.**nickel_ebsd_large**(*allow_download: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSD*

4125 EBSD patterns in a (55, 75) navigation shape of (60, 60) pixels from nickel, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019].

**Parameters**

**allow_download**
Whether to allow downloading the dataset from the internet to the local cache with the pooch Python package. Default is `False`.

**show_progressbar**
Whether to show a progressbar when downloading. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *load()*.

**Returns**

**ebsd_signal**
EBSD signal.

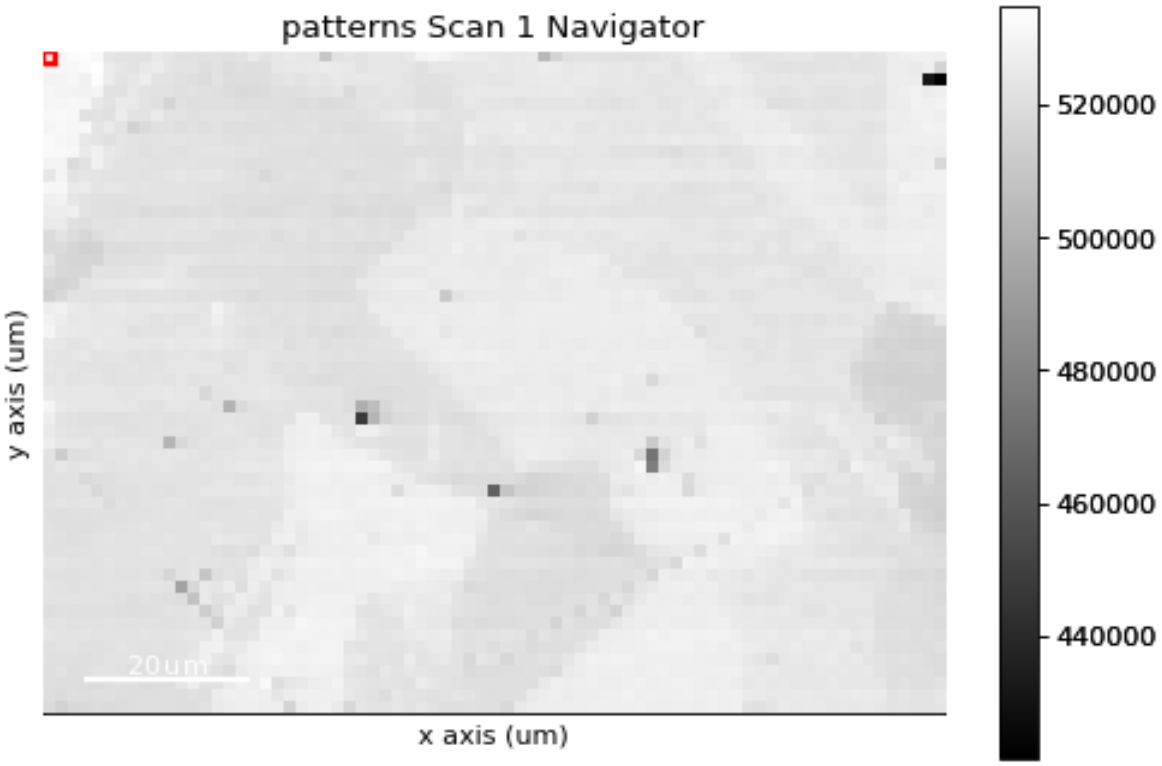
### Notes

The dataset is hosted in the GitHub repository https://github.com/pyxem/kikuchipy-data.

The dataset carries a CC BY 4.0 license.

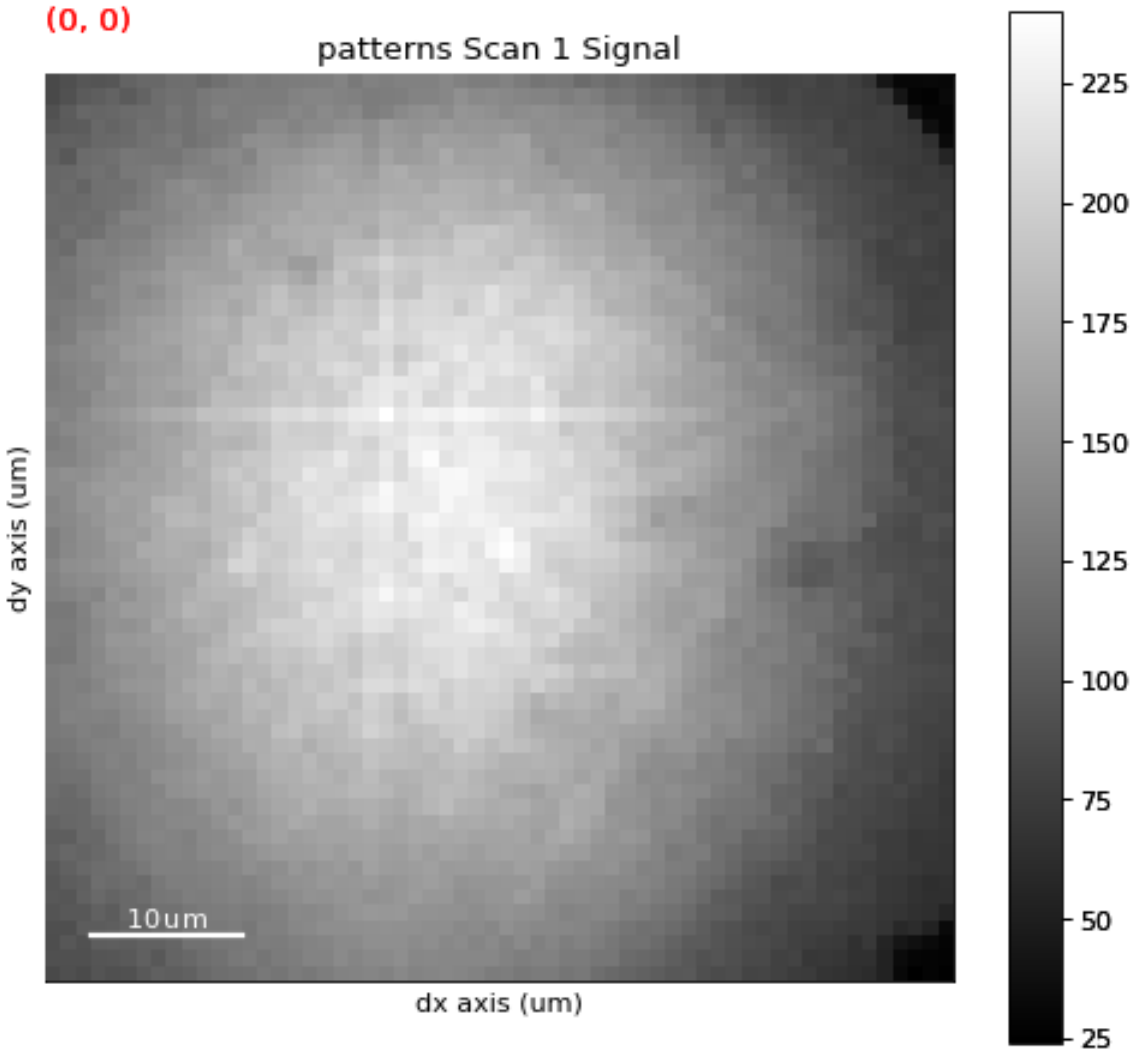
### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_large(allow_download=True)
>>> s
<EBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
>>> s.plot()
```




### Examples using `nickel_ebsd_large`

- *Neighbour pattern averaging*
- *Extract patterns from a grid*


(0, 0)
patterns Scan 1 Signal
dy axis (um)
dx axis (um)
10um
225
200
175
150
125
100
75
50
25

### 2.3.5 nickel_ebsd_master_pattern_small

kikuchipy.data.**nickel_ebsd_master_pattern_small**(***kwargs*) → *EBSDMasterPattern*

(401, 401) `uint8` square Lambert or stereographic projection of the northern and southern hemisphere of a nickel master pattern at 20 keV accelerating voltage.

The master pattern was simulated with *EMsoft* [Callahan and De Graef, 2013].

**Parameters**

****kwargs**
Keyword arguments passed to *load()*.

**Returns**

**ebsd_master_pattern_signal**
EBSD master pattern signal.

**See also:**

*ebsd_master_pattern*

#### Notes

The dataset carries a CC BY 4.0 license.

Initially generated using the EMsoft EMMCOpenCL and EMEBSDMaster programs. The included file was rewritten to disk with `h5py`, where the master patterns' data type is converted from `float32` to `uint8` with *rescale_intensity()*, all datasets were written with `dict2h5ebsdgroup()` with keyword arguments `compression="gzip"` and `compression_opts=9`. All other HDF5 groups and datasets are the same as in the original file.

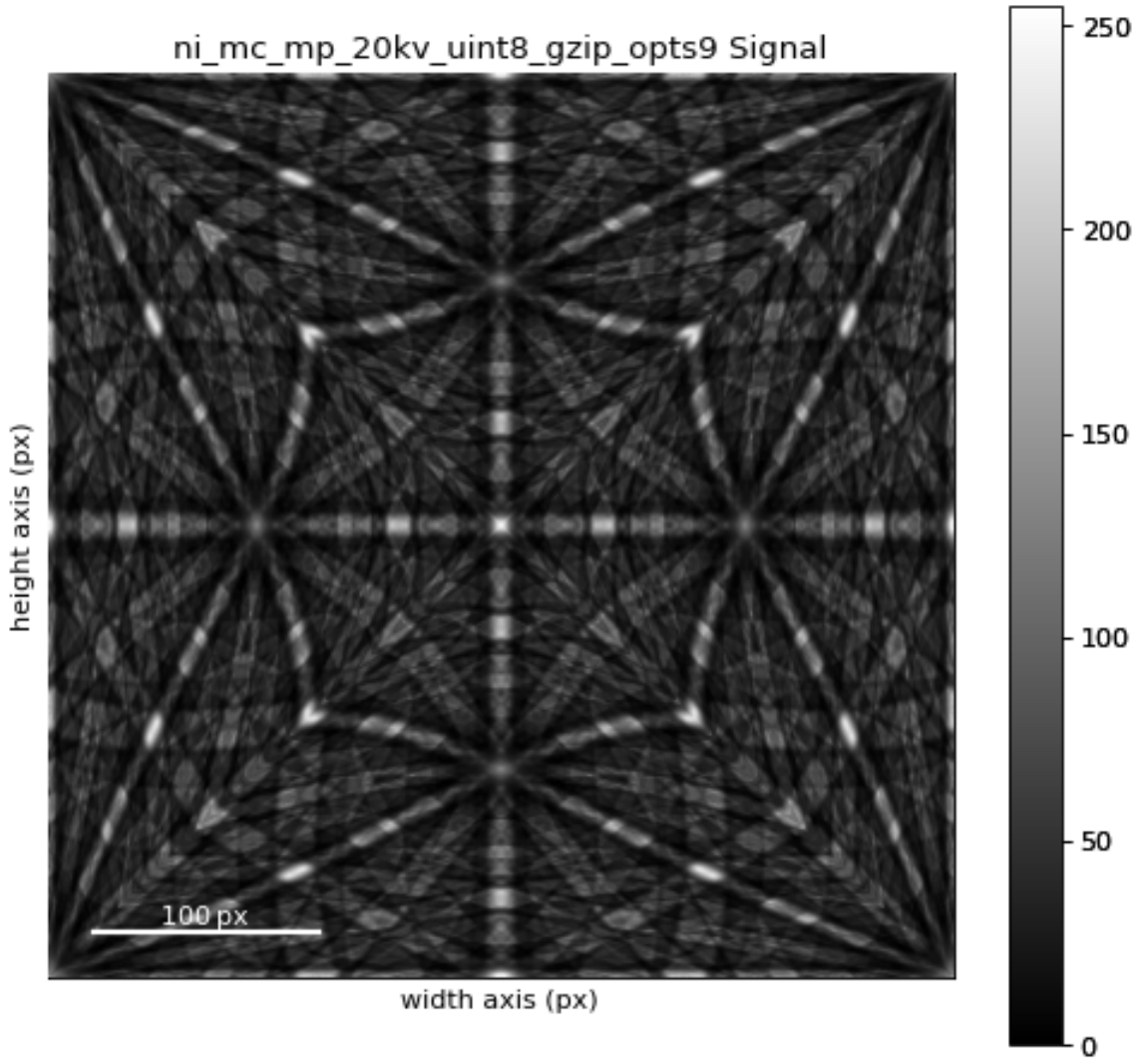
#### Examples

Import master pattern in the stereographic projection

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_master_pattern_small()
>>> s
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (|401, 401)>
>>> s.projection
'stereographic'
```

Import master pattern in the square Lambert projection and plot it

```
>>> s2 = kp.data.nickel_ebsd_master_pattern_small(projection="lambert")
>>> s2.projection
'lambert'
>>> s2.plot()
```


ni_mc_mp_20kv_uint8_gzip_opts9 Signal
height axis (px)
width axis (px)
100 px
250
200
150
100
50
0

### Examples using `nickel_ebsd_master_pattern_small`

- *Adaptive histogram equalization*
- *Plot nice master pattern image*

## 2.3.6 nickel_ebsd_small

kikuchipy.data.**nickel_ebsd_small**(***kwargs*) → *EBSD*

Ni EBSD patterns in a (3, 3) navigation shape of (60, 60) pixels from nickel, acquired on a NORDIF UF-1100 detector [Ånes *et al.*, 2019].

**Parameters**

****kwargs**
Keyword arguments passed to `load()`.

**Returns**

**ebsd_signal**
EBSD signal.

### Notes

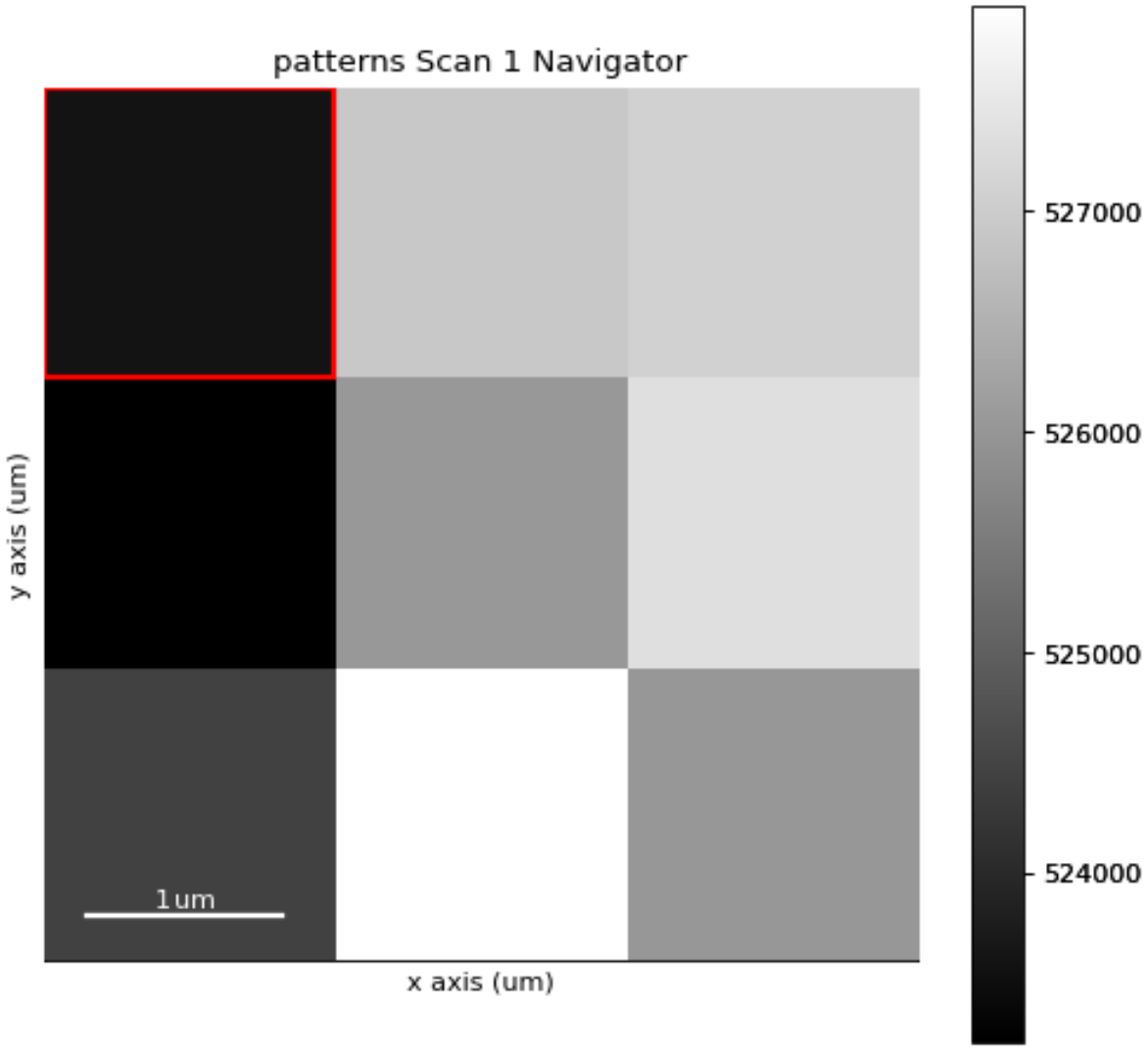
The dataset carries a CC BY 4.0 license.

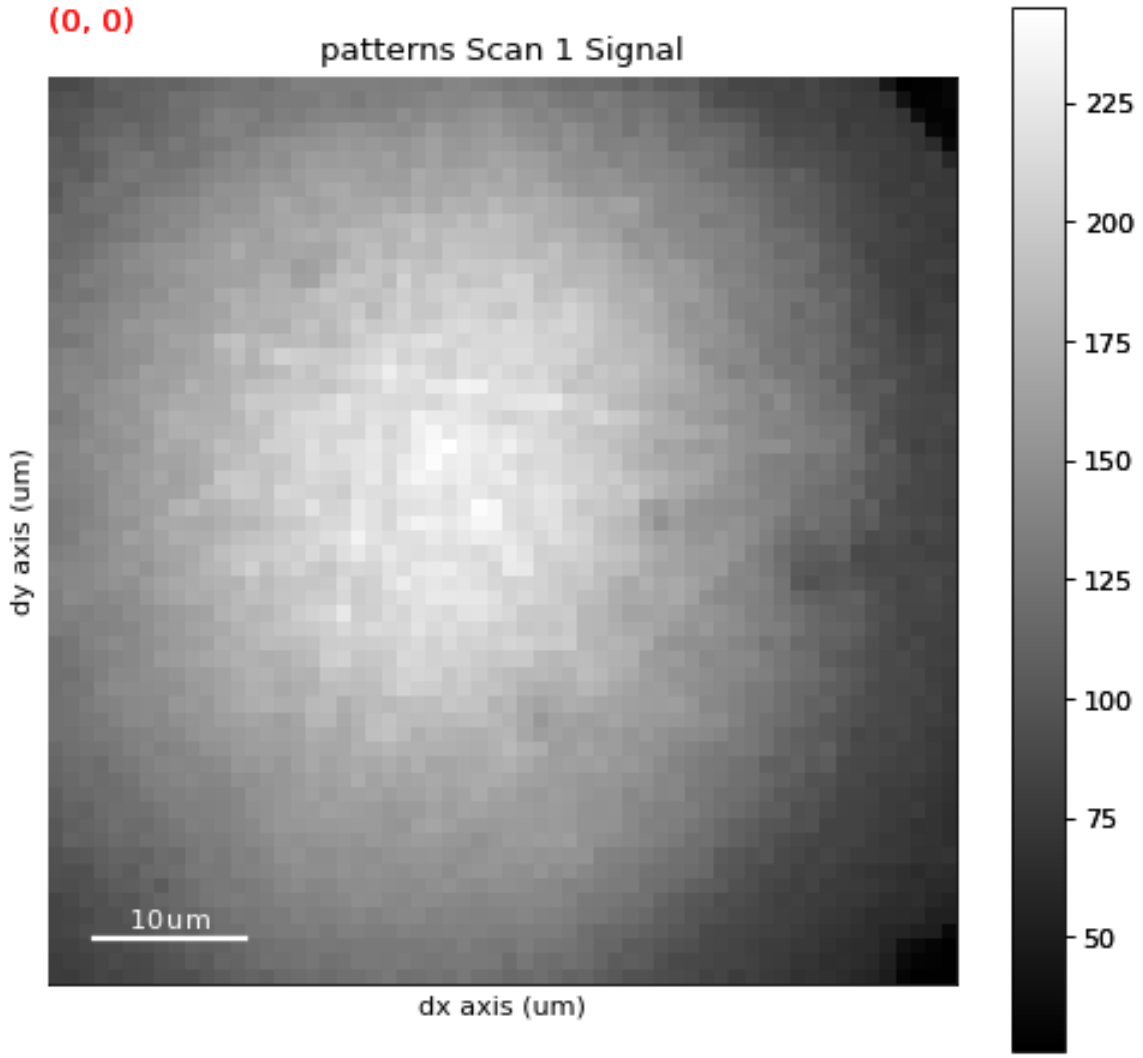
### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
>>> s.plot()
```

### Examples using `nickel_ebsd_small`

- *Static background correction*
- *Dynamic background correction*
- *Crop navigation axes*
- *Crop signal axes*


patterns Scan 1 Navigator
y axis (um)
x axis (um)
1um
527000
526000
525000
524000


(0, 0)
patterns Scan 1 Signal
dy axis (um)
dx axis (um)
10um
225
200
175
150
125
100
75
50

### 2.3.7 si_wafer

kikuchipy.data.**si_wafer**(*allow_download: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSD*

EBSD dataset of (50, 50) patterns of (480, 480) pixels from a single crystal silicon wafer, acquired on a NORDIF UF-420 detector [Ånes *et al.*, 2022].

The dataset was acquired in order to test various ways to calibrate projection centers (PCs), e.g. the moving-screen PC estimation technique [Hjelen *et al.*, 1991]. The EBSD pattern in `silicon_ebsd_moving_screen_in()` is from this dataset.

**Parameters**

**allow_download**
Whether to allow downloading the dataset from the internet to the local cache with the pooch Python package. Default is `False`.

**show_progressbar**
Whether to show a progressbar when downloading. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *`load()`*.

**Returns**

**ebsd_signal**
EBSD signal.

**See also:**

**silicon_ebsd_moving_screen_in**, **silicon_ebsd_moving_screen_out5mm**
**silicon_ebsd_moving_screen_out10mm**

#### Notes

The dataset is hosted in the Zenodo repository https://doi.org/10.5281/zenodo.7491388 and comprises 311 MB as a zipped file and about 581 MB when unzipped. The zipped file is deleted after it is unzipped.

The dataset carries a CC BY 4.0 license.

#### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.si_wafer(allow_download=True, lazy=True)
>>> s
<EBSD, title: Pattern, dimensions: (50, 50|480, 480)>
```

### 2.3.8 si_ebsd_moving_screen

kikuchipy.data.**si_ebsd_moving_screen**(*distance: int = 0*, *allow_download: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSD*

One EBSD pattern of (480, 480) pixels from a single crystal silicon sample, acquired on a NORDIF UF-420 detector [Ånes *et al.*, 2022].

Three patterns are available from the same sample position but with different sample-screen distances: normal position (`0`) and 5 mm and 10 mm greater than the normal position (`5` and `10`).

They were acquired to test the moving-screen projection center estimation technique [Hjelen *et al.*, 1991].

**Parameters**

**distance**
Sample-screen distance away from the normal position, either 0, 5 or 10.

**allow_download**
Whether to allow downloading the dataset from the internet to the local cache with the pooch Python package. Default is `False`.

**show_progressbar**
Whether to show a progressbar when downloading. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *`load()`*.

**Returns**

**ebsd_signal**
EBSD signal.

**See also:**

*`si_wafer`*

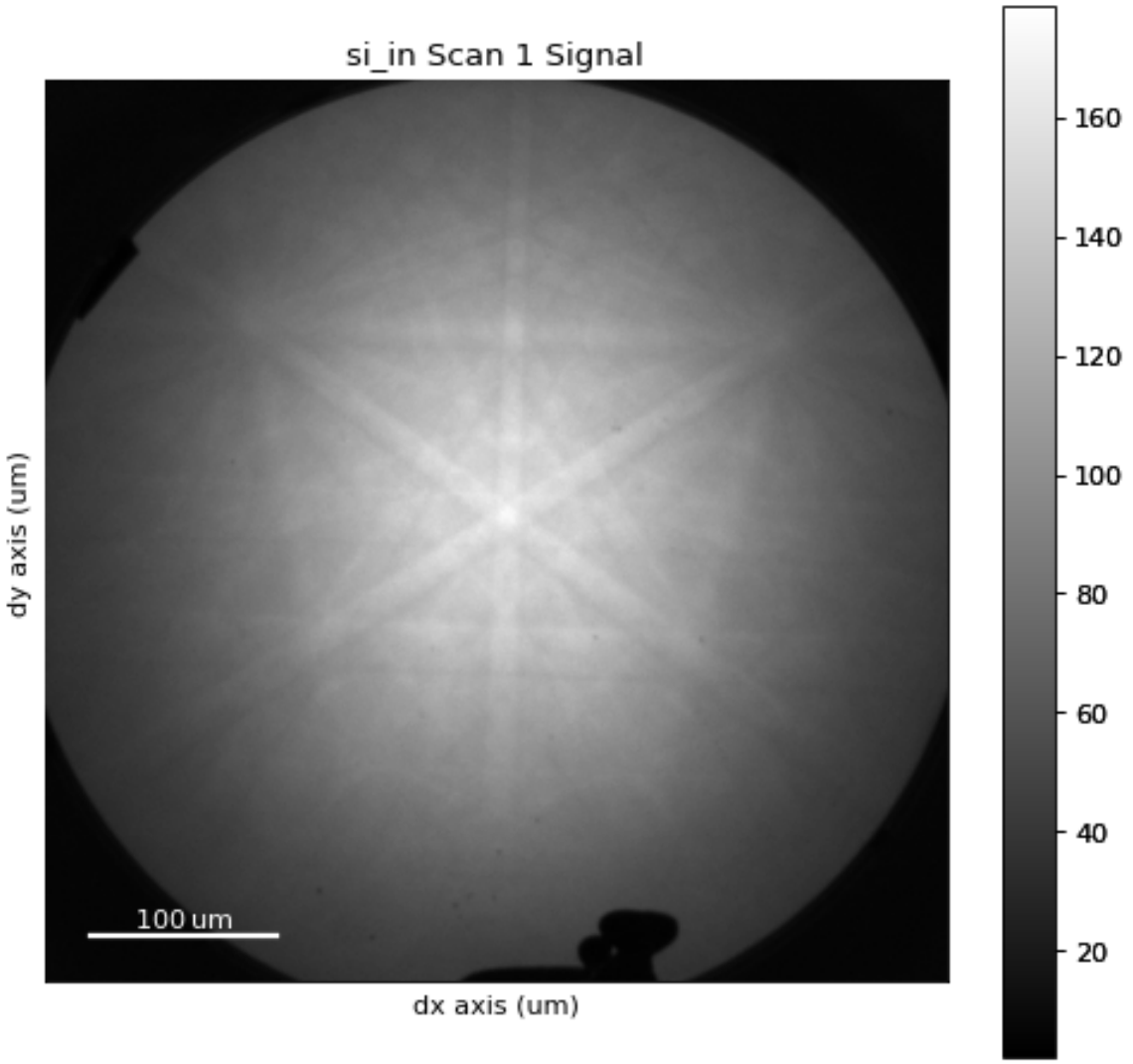
#### Notes

The datasets are hosted in the GitHub repository https://github.com/pyxem/kikuchipy-data.

The datasets carry a CC BY 4.0 license.

#### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.si_ebsd_moving_screen(allow_download=True)
>>> s
<EBSD, title: si_in Scan 1, dimensions: (|480, 480)>
>>> s.plot()
```


si_in Scan 1 Signal
dy axis (um)
dx axis (um)
100 um
160
140
120
100
80
60
40
20

### Examples using `si_ebsd_moving_screen`

- *Pattern binning*

## 2.4 detectors

Tools for handling the EBSD detector-sample geometry.

### Classes

| | |
|---|---|
| *EBSDDetector*([shape, px_size, binning, ...]) | An EBSD detector class storing its shape, pixel size, binning factor, detector tilt, sample tilt and projection center (PC) per pattern. |
| *PCCalibrationMovingScreen*(pattern_in, ...[, ...]) | A class to perform and inspect the calibration of the EBSD projection center (PC) using the "moving screen" technique from [Hjelen *et al.*, 1991]. |

### 2.4.1 EBSDDetector

**class** kikuchipy.detectors.**EBSDDetector**(*shape: Tuple[int, int] = (1, 1)*, *px_size: float = 1*, *binning: int = 1*, *tilt: float = 0*, *azimuthal: float = 0*, *sample_tilt: float = 70*, *pc: ndarray | list | tuple = (0.5, 0.5, 0.5)*, *convention: str | None = None*)

Bases: `object`

An EBSD detector class storing its shape, pixel size, binning factor, detector tilt, sample tilt and projection center (PC) per pattern. Given one or multiple PCs, the detector's gnomonic coordinates are calculated. Uses of these include projecting Kikuchi bands, given a unit cell, unit cell orientation and family of planes, onto the detector.

Calculation of gnomonic coordinates is based on the work by Aimo Winkelmann in the supplementary material to [Britton *et al.*, 2016].

**Parameters**

**shape**
Number of detector rows and columns in pixels. Default is (1, 1).

**px_size**
Size of unbinned detector pixel in um, assuming a square pixel shape. Default is 1.

**binning**
Detector binning, i.e. how many pixels are binned into one. Default is 1, i.e. no binning.

**tilt**
Detector tilt from horizontal in degrees. Default is 0.

**azimuthal**
Sample tilt about the sample RD (downwards) axis. A positive angle means the sample normal moves towards the right looking from the sample to the detector. Default is 0.

**sample_tilt**
Sample tilt from horizontal in degrees. Default is 70.

**pc**
: X, Y and Z coordinates of the projection/pattern centers (PCs), describing the location of the beam on the sample measured relative to the detection screen. PCs are stored and used in Bruker's convention. See *Notes* for the definition and conversions between conventions. If multiple PCs are passed, they are assumed to be on the form [[x0, y0, z0], [x1, y1, z1], ...]. Default is [0.5, 0.5, 0.5].

**convention**
: Convention of input PC, to determine which conversion to Bruker's definition to use. If not given, Bruker's convention is assumed. Options are "tsl"/"edax"/"amatek", "oxford"/"aztec", "bruker", "emsoft", "emsoft4", and "emsoft5". "emsoft" and "emsoft5" is the same convention. See *Notes* for conversions between conventions.

## Notes

The pattern on the detector is always viewed *from* the detector *towards* the sample. Pattern width and height is here given as $N_x$ and $N_y$ (possibly binned). PCs are stored and used in Bruker's convention.

The Bruker PC coordinates $(x_B^*, y_B^*, z_B^*)$ are defined in fractions of $N_x$, $N_y$, and $N_y$, respectively, with $x_B^*$ and $y_B^*$ defined with respect to the upper left corner of the detector. These coordinates are used internally, called $(PC_x, PC_y, PC_z)$ in the rest of the documentation when there is no reference to Bruker specifically.

The EDAX TSL PC coordinates $(x_T^*, y_T^*, z_T^*)$ are defined in fractions of $(N_x, N_y, min(N_x, N_y))$ with respect to the lower left corner of the detector.

The Oxford Instruments PC coordinates $(x_O^*, y_O^*, z_O^*)$ are defined in fractions of $N_x$ with respect to the lower left corner of the detector.

The EMsoft PC coordinates $(x_{pc}, y_{pc})$ are defined as number of pixels (subpixel accuracy) with respect to the center of the detector, with $x_{pc}$ towards the right and $y_{pc}$ upwards. The final PC coordinate $L$ is the detector distance in microns. Note that prior to EMsoft v5.0, $x_{pc}$ was defined towards the left.

Given these definitions, the following is the conversion from EDAX TSL to Bruker

$$
\begin{aligned}
x_B^* &= x_T^*, \\
y_B^* &= 1 - y_T^*, \\
z_B^* &= \frac{min(N_x, N_y)}{N_y} z_T^*.
\end{aligned}
$$

The conversion from Oxford Instruments to Bruker is given as

$$
\begin{aligned}
x_B^* &= x_O^*, \\
y_B^* &= 1 - y_O^* \frac{N_x}{N_y}, \\
z_B^* &= \frac{N_x}{N_y} z_O^*.
\end{aligned}
$$

The conversion from EMsoft to Bruker is given as

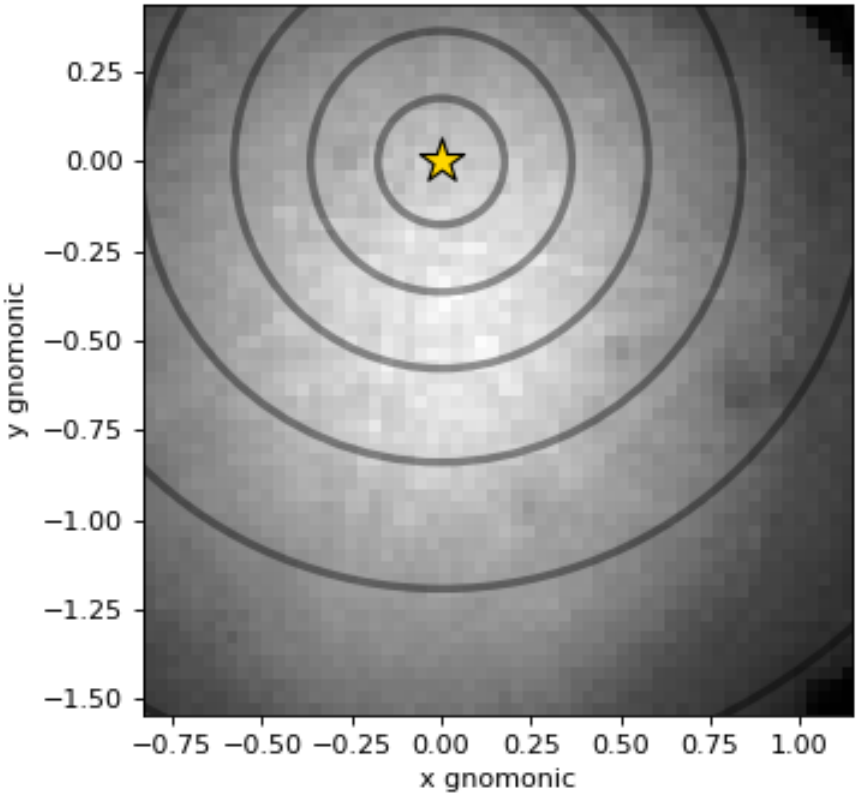
$$
\begin{aligned}
x_B^* &= \frac{1}{2} - \frac{x_{pc}}{N_x b}, \\
y_B^* &= \frac{1}{2} - \frac{y_{pc}}{N_y b}, \\
z_B^* &= \frac{L}{N_y b \delta},
\end{aligned}
$$

where $\delta$ is the unbinned detector pixel size in microns, and $b$ is the binning factor.

### Examples

Create an EBSD detector and plot the PC on top of a pattern

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> det = kp.detectors.EBSDDetector(
...     shape=(60, 60),
...     pc=np.ones((10, 20, 3)) * (0.421, 0.779, 0.505),
...     convention="edax",
...     px_size=70,
...     binning=8,
...     tilt=5,
...     sample_tilt=70,
... )
>>> det
EBSDDetector (60, 60), px_size 70 um, binning 8, tilt 5, azimuthal 0, pc (0.421, 0.
↪221, 0.505)
>>> det.navigation_shape
(10, 20)
>>> det.bounds
array([ 0, 59,  0, 59])
>>> det.gnomonic_bounds[0, 0]
array([-0.83366337,  1.14653465, -1.54257426,  0.43762376])
>>> s = kp.data.nickel_ebsd_small()
>>> det.plot(
...     pattern=s.inav[0, 0].data,
...     coordinates="gnomonic",
...     draw_gnomonic_circles=True
... )
```


0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
y gnomonic
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x gnomonic

### Attributes

| | |
|---|---|
| *EBSDDetector.aspect_ratio* | Return the number of detector columns divided by rows. |
| *EBSDDetector.bounds* | Return the detector bounds [x0, x1, y0, y1] in pixel coordinates. |
| *EBSDDetector.gnomonic_bounds* | Return the detector bounds [x0, x1, y0, y1] in gnomonic coordinates. |
| *EBSDDetector.height* | Return the detector height in microns. |
| *EBSDDetector.navigation_dimension* | Return the number of navigation dimensions of the projection center array (a maximum of 2). |
| *EBSDDetector.navigation_shape* | Return or set the navigation shape of the projection center array. |
| *EBSDDetector.navigation_size* | Return the number of projection centers. |
| *EBSDDetector.ncols* | Return the number of detector pixel columns. |
| *EBSDDetector.nrows* | Return the number of detector pixel rows. |
| *EBSDDetector.pc* | Return or set all projection center coordinates. |
| *EBSDDetector.pc_average* | Return the overall average projection center. |
| *EBSDDetector.pc_flattened* | Return flattened array of projection center coordinates of shape (*navigation_size*, 3). |
| *EBSDDetector.pcx* | Return or set the projection center x coordinates. |
| *EBSDDetector.pcy* | Return or set the projection center y coordinates. |
| *EBSDDetector.pcz* | Return or set the projection center z coordinates. |
| *EBSDDetector.px_size_binned* | Return the binned pixel size in microns. |
| *EBSDDetector.r_max* | Return the maximum distance from PC to detector edge in gnomonic coordinates. |
| *EBSDDetector.size* | Return the number of detector pixels. |
| *EBSDDetector.specimen_scintillator_distan* | Return the specimen to scintillator distance, known in EMsoft as $L$. |
| *EBSDDetector.unbinned_shape* | Return the unbinned detector shape in pixels. |
| *EBSDDetector.width* | Return the detector width in microns. |
| *EBSDDetector.x_max* | Return the right bound of detector in gnomonic coordinates. |
| *EBSDDetector.x_min* | Return the left bound of detector in gnomonic coordinates. |
| *EBSDDetector.x_range* | Return the x detector limits in gnomonic coordinates. |
| *EBSDDetector.x_scale* | Return the width of a pixel in gnomonic coordinates. |
| *EBSDDetector.y_max* | Return the bottom bound of detector in gnomonic coordinates. |
| *EBSDDetector.y_min* | Return the top bound of detector in gnomonic coordinates. |
| *EBSDDetector.y_range* | Return the y detector limits in gnomonic coordinates. |
| *EBSDDetector.y_scale* | Return the height of a pixel in gnomonic coordinates. |

### aspect_ratio

property EBSDDetector.**aspect_ratio**: float

Return the number of detector columns divided by rows.

### bounds

property EBSDDetector.**bounds**: ndarray

Return the detector bounds [x0, x1, y0, y1] in pixel coordinates.

### gnomonic_bounds

property EBSDDetector.**gnomonic_bounds**: ndarray

Return the detector bounds [x0, x1, y0, y1] in gnomonic coordinates.

### height

property EBSDDetector.**height**: float

Return the detector height in microns.

### navigation_dimension

property EBSDDetector.**navigation_dimension**: int

Return the number of navigation dimensions of the projection center array (a maximum of 2).

### navigation_shape

property EBSDDetector.**navigation_shape**: tuple

Return or set the navigation shape of the projection center array.

**Parameters**

**value**
[tuple] Navigation shape, with a maximum dimension of 2.

### Examples using EBSDDetector.navigation_shape

- *Estimate tilt about the detector x axis*
- *Estimate tilts about the detector x and z axis*

### navigation_size

property EBSDDetector.**navigation_size**: int

Return the number of projection centers.

### Examples using EBSDDetector.navigation_size

- *Fit a plane to selected projection centers*
- *Estimate tilt about the detector x axis*

### ncols

property EBSDDetector.**ncols**: int

Return the number of detector pixel columns.

### nrows

property EBSDDetector.**nrows**: int

Return the number of detector pixel rows.

### pc

property EBSDDetector.**pc**: ndarray

Return or set all projection center coordinates.

**Parameters**

**value**
[numpy.ndarray, list or tuple] Projection center coordinates. If multiple PCs are passed, they are assumed to be on the form [[x0, y0, z0], [x1, y1, z1], ...]. Default is [[0.5, 0.5, 0.5]].

### Examples using EBSDDetector.pc

- *Fit a plane to selected projection centers*
- *Estimate tilts about the detector x and z axis*

### pc_average

property EBSDDetector.**pc_average**: ndarray

Return the overall average projection center.

### pc_flattened

property EBSDDetector.**pc_flattened**: ndarray

Return flattened array of projection center coordinates of shape (*navigation_size*, 3).

### Examples using EBSDDetector.pc_flattened

- *Fit a plane to selected projection centers*

### pcx

property EBSDDetector.**pcx**: ndarray

Return or set the projection center x coordinates.

**Parameters**

**value**
[numpy.ndarray, list, tuple or float] Projection center x coordinates in Bruker's convention. If multiple x coordinates are passed, they are assumed to be on the form [x0, x1,...].

### Examples using EBSDDetector.pcx

- *Fit a plane to selected projection centers*
- *Estimate tilts about the detector x and z axis*

### pcy

property EBSDDetector.**pcy**: ndarray

Return or set the projection center y coordinates.

**Parameters**

**value**
[numpy.ndarray, list, tuple or float] Projection center y coordinates in Bruker's convention. If multiple y coordinates are passed, they are assumed to be on the form [y0, y1,...].

### Examples using EBSDDetector.pcy

- *Fit a plane to selected projection centers*
- *Estimate tilt about the detector x axis*
- *Estimate tilts about the detector x and z axis*

### pcz

property EBSDDetector.**pcz**: ndarray

Return or set the projection center z coordinates.

**Parameters**

**value**
[numpy.ndarray, list, tuple or float] Projection center z coordinates in Bruker's convention. If multiple z coordinates are passed, they are assumed to be on the form [z0, z1,...].

### Examples using EBSDDetector.pcz

- *Fit a plane to selected projection centers*
- *Estimate tilt about the detector x axis*
- *Estimate tilts about the detector x and z axis*

### px_size_binned

property EBSDDetector.**px_size_binned**: float

Return the binned pixel size in microns.

### r_max

property EBSDDetector.**r_max**: ndarray

Return the maximum distance from PC to detector edge in gnomonic coordinates.

### size

property EBSDDetector.**size**: int

Return the number of detector pixels.

### specimen_scintillator_distance

property EBSDDetector.**specimen_scintillator_distance**: float

Return the specimen to scintillator distance, known in EMsoft as $L$.

#### unbinned_shape

property EBSDDetector.**unbinned_shape**: Tuple[int, int]

Return the unbinned detector shape in pixels.

#### width

property EBSDDetector.**width**: float

Return the detector width in microns.

#### x_max

property EBSDDetector.**x_max**: ndarray | float

Return the right bound of detector in gnomonic coordinates.

#### x_min

property EBSDDetector.**x_min**: ndarray | float

Return the left bound of detector in gnomonic coordinates.

#### x_range

property EBSDDetector.**x_range**: ndarray

Return the x detector limits in gnomonic coordinates.

#### x_scale

property EBSDDetector.**x_scale**: ndarray

Return the width of a pixel in gnomonic coordinates.

#### y_max

property EBSDDetector.**y_max**: ndarray | float

Return the bottom bound of detector in gnomonic coordinates.

#### y_min

property EBSDDetector.**y_min**: ndarray | float

Return the top bound of detector in gnomonic coordinates.

### y_range

**property** EBSDDetector.**y_range: ndarray**

Return the y detector limits in gnomonic coordinates.

### y_scale

**property** EBSDDetector.**y_scale: ndarray**

Return the height of a pixel in gnomonic coordinates.

### Methods

| | |
|---|---|
| *EBSDDetector.crop*(extent) | Return a new detector with its shape cropped and *pc* values updated accordingly. |
| *EBSDDetector.deepcopy*() | Return a deep copy using `copy.deepcopy()`. |
| *EBSDDetector.estimate_xtilt*([...]) | Estimate the tilt about the detector $X_d$ axis. |
| *EBSDDetector.estimate_xtilt_ztilt*([degrees, ...]) | Estimate the tilts about the detector $X_d$ and $Z_d$ axes. |
| *EBSDDetector.extrapolate_pc*(pc_indices, ...) | Return a new detector with projection centers (PCs) in a 2D map extrapolated from an average PC. |
| *EBSDDetector.fit_pc*(pc_indices, map_indices) | Return a new detector with interpolated projection centers (PCs) for all points in a map by fitting a plane to *pc* [Winkelmann *et al.*, 2020]. |
| *EBSDDetector.get_indexer*(phase_list[, ...]) | Return a PyEBSDIndex EBSD indexer. |
| *EBSDDetector.load*(fname) | Return an EBSD detector loaded from a text file saved with *save()*. |
| *EBSDDetector.pc_bruker*() | Return PC in the Bruker convention, given in the class description. |
| *EBSDDetector.pc_emsoft*([version]) | Return PC in the EMsoft convention. |
| *EBSDDetector.pc_oxford*() | Return PC in the Oxford convention. |
| *EBSDDetector.pc_tsl*() | Return PC in the EDAX TSL convention. |
| *EBSDDetector.plot*([coordinates, show_pc, ...]) | Plot the detector screen viewed from the detector towards the sample. |
| *EBSDDetector.plot_pc*([mode, return_figure, ...]) | Plot all projection centers (PCs). |
| *EBSDDetector.save*(filename[, convention]) | Save detector in a text file with projection centers (PCs) in the given convention. |

### crop

EBSDDetector.**crop**(*extent: Tuple[int, int, int, int] | List[int]*) → *EBSDDetector*

Return a new detector with its shape cropped and *pc* values updated accordingly.

**Parameters**

**extent**
Tuple with four integers: (top, bottom, left, right).

**Returns**

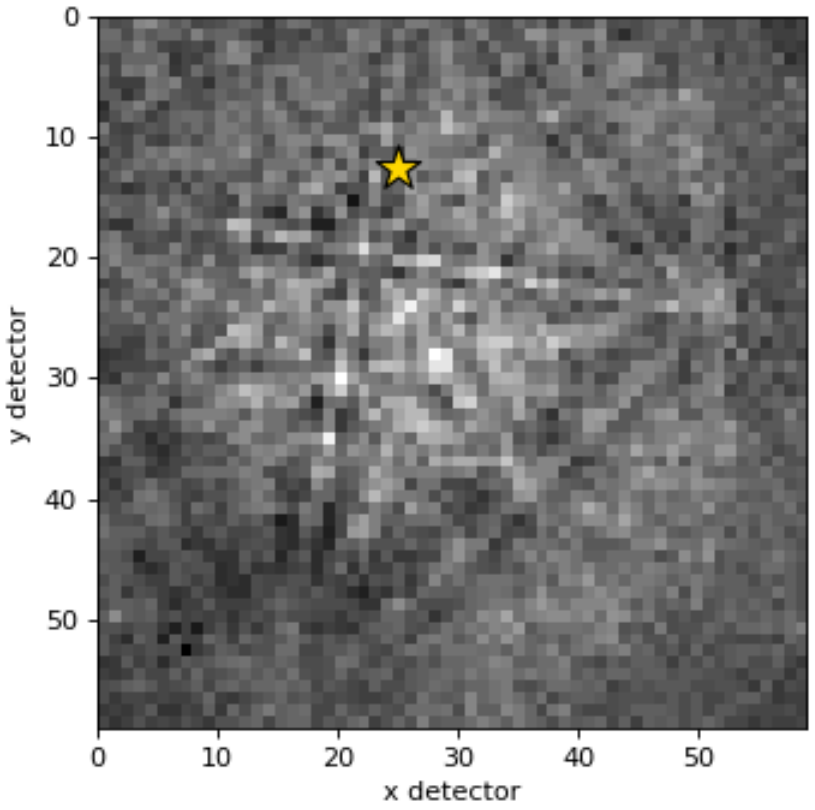
**new_detector**
A new detector with a new shape and PC values.

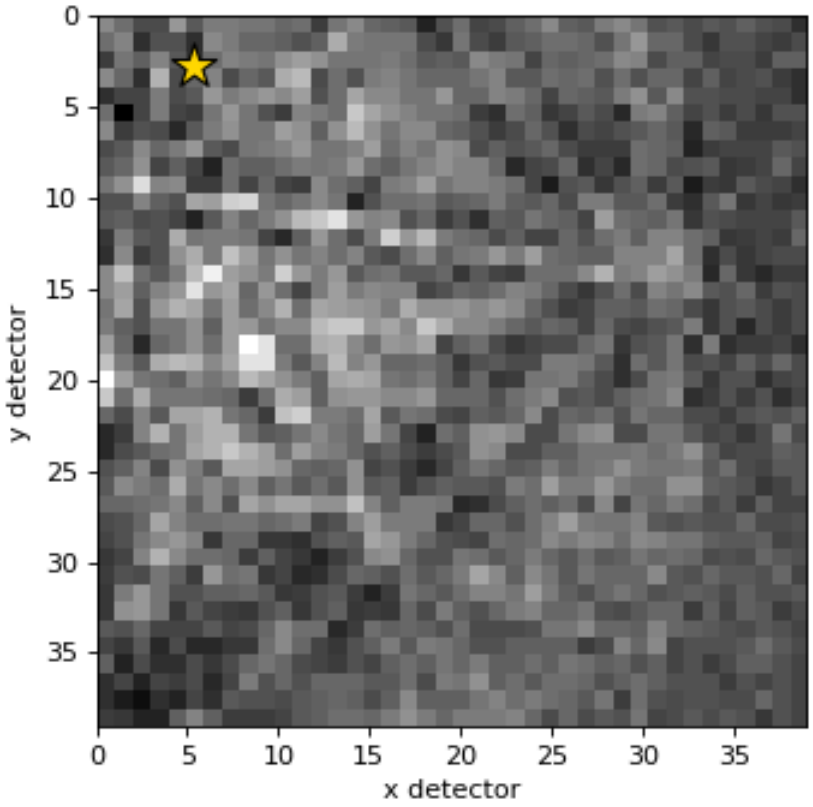
### Examples

```
>>> import kikuchipy as kp
>>> det = kp.detectors.EBSDDetector((6, 6), pc=[3 / 6, 2 / 6, 0.5])
>>> det
EBSDDetector (6, 6), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.5, 0.
↪333, 0.5)
>>> det.crop((1, 5, 2, 6))
EBSDDetector (4, 4), px_size 1 um, binning 1, tilt 0, azimuthal 0, pc (0.25, 0.
↪25, 0.75)
```

Plot a cropped detector with the PC on a cropped pattern

```
>>> s = kp.data.nickel_ebsd_small()
>>> s.remove_static_background(show_progressbar=False)
>>> det2 = s.detector
>>> det2.plot(pattern=s.inav[0, 0].data)
>>> det3 = det2.crop((10, 50, 20, 60))
>>> det3.plot(pattern=s.inav[0, 0].data[10:50, 20:60])
```




0
5
10
15
20
25
30
35
y detector
0
5
10
15
20
25
30
35
x detector

### deepcopy

EBSDDetector.**deepcopy**() → *EBSDDetector*

Return a deep copy using `copy.deepcopy()`.

**Returns**

**detector**
Identical detector without shared memory.

### Examples using `EBSDDetector.deepcopy`

- *Fit a plane to selected projection centers*

### estimate_xtilt

EBSDDetector.**estimate_xtilt**(*detect_outliers: bool = False*, *plot: bool = True*, *degrees: bool = False*, *return_figure: bool = False*, *return_outliers: bool = False*, *figure_kwargs: dict | None = None*) → float | Tuple[float, ndarray] | Tuple[float, Figure] | Tuple[float, ndarray, Figure]

Estimate the tilt about the detector $X_d$ axis.

This tilt is assumed to bring the sample plane normal into coincidence with the detector plane normal (but in the opposite direction) [Winkelmann *et al.*, 2020].

See the *reference frame tutorial* for details on the detector sample geometry.

An estimate is found by linear regression of `pcz` vs. `pcy`.

**Parameters**

**detect_outliers**
Whether to attempt to detect outliers. If `False` (default), a linear fit to all points is performed. If `True`, a robust fit using the RANSAC algorithm is performed instead, which also detects outliers.

**plot**
Whether to plot data points and the estimated line. Default is `True`.

**degrees**
Whether to return the estimated tilt in radians (`False`, default) or degrees (`True`).

**return_figure**
Whether to return the plotted figure. Default is `False`.

**return_outliers**
Whether to return a mask with `True` for PC values considered outliers. Default is `False`. If `True`, `detect_outliers` is assumed to be `True` and the value passed is not considered.

**figure_kwargs**
Keyword arguments passed to `matplotlib.pyplot.Figure()` if `plot=True`.

**Returns**

**x_tilt**
Estimated tilt about detector $X_d$ in radians (`degrees=False`) or degrees (`degrees=True`).

**outliers**
Returned if `return_outliers=True`, in the shape of `navigation_shape`.

**fig**
Returned if `plot=True` and `return_figure=True`.

**See also:**

`sklearn.linear_model.LinearRegression`
`sklearn.linear_model.RANSACRegressor`, *`estimate_xtilt_ztilt`*
*`fit_pc`*

### Notes

This method is adapted from Aimo Winkelmann's function `fit_xtilt()` in the *xcdskd* Python package. See [Winkelmann *et al.*, 2020] for their use of related functions.

### Examples using `EBSDDetector.estimate_xtilt`

- *Estimate tilt about the detector x axis*

### estimate_xtilt_ztilt

`EBSDDetector.`**`estimate_xtilt_ztilt`**(*degrees: bool = False*, *is_outlier: list | tuple | ndarray | None = None*) → float | Tuple[float, float]

Estimate the tilts about the detector $X_d$ and $Z_d$ axes.

These tilts bring the sample plane normal into coincidence with the detector plane normal (but in the opposite direction) [Winkelmann *et al.*, 2020].

See the *reference frame tutorial* for details on the detector sample geometry.

Estimates are found by fitting a hyperplane to *`pc`* using singular value decomposition.

**Parameters**

**degrees**
Whether to return the estimated tilts in radians (`False`, default) or degrees (`True`).

**is_outlier**
Boolean array with `True` for PCs to not include in the fit. If not given, all PCs are used. Must be of *`navigation_shape`*.

**Returns**

**x_tilt**
Estimated tilt about detector $X_d$ in radians (`degrees=False`) or degrees (`degrees=True`).

**z_tilt**
Estimated tilt about detector $Z_d$ in radians (`degrees=False`) or degrees (`degrees=True`).

**See also:**

*`estimate_xtilt`*, *`fit_pc`*

### Notes

This method is adapted from Aimo Winkelmann's function `fit_plane()` in the *xcdskd* Python package. Its use is described in [Winkelmann *et al.*, 2020].

Winkelmann refers to Gander & Hrebicek, "Solving Problems in Scientific Computing", 3rd Ed., Chapter 6, p. 97 for the implementation of the hyperplane fitting.

### Examples using `EBSDDetector.estimate_xtilt_ztilt`

- *Estimate tilts about the detector x and z axis*

## extrapolate_pc

`EBSDDetector.`**`extrapolate_pc`**(*pc_indices: tuple | list | ndarray*, *navigation_shape: tuple*, *step_sizes: tuple*, *shape: tuple | None = None*, *px_size: float = None*, *binning: int = None*, *is_outlier: list | tuple | ndarray | None = None*)

Return a new detector with projection centers (PCs) in a 2D map extrapolated from an average PC.

The average PC $\bar{PC}$ is calculated from `pc`, possibly excluding some PCs based on the `is_outlier` mask. The sample position having this PC, $(\bar{x}, \bar{y})$, is assumed to be the one obtained by averaging `pc_indices`. All other PCs $(PC_x, PC_y, PC_z)$ in positions $(x, y)$ are then extrapolated based on the following equations given in appendix A in [Singh *et al.*, 2017]:

$$
\begin{aligned}
PC_x &= \bar{PC}_x + (\bar{x} - x) \cdot \Delta x / (\delta \cdot N_x \cdot b), \\
PC_y &= \bar{PC}_y + (\bar{y} - y) \cdot \Delta y \cdot \cos\alpha / (\delta \cdot N_y \cdot b), \\
PC_z &= \bar{PC}_z - (\bar{y} - y) \cdot \Delta y \cdot \sin\alpha / (\delta \cdot N_y \cdot b),
\end{aligned}
$$

where $(\Delta y, \Delta x)$ are the vertical and horizontal step sizes, respectively, $(N_y, N_x)$ are the number of binned detector rows and columns, respectively, the angle $\alpha = 90° - \sigma + \theta$, where $\sigma$ is the sample tilt and $\theta$ is the detector tilt, $\delta$ is the unbinned detector pixel size and $b$ is the binning factor.

**Parameters**

**pc_indices**
2D map pixel coordinates (row, column) of each `pc`, possibly outside `navigation_shape`. Must be a flattened array of shape (2,) + `navigation_size`.

**navigation_shape**
Shape of the output PC array (n rows, n columns).

**step_sizes**
Vertical and horizontal step sizes (dy, dx).

**shape**
Detector (signal) shape (n rows, n columns). If not given, (`nrows`, `ncols`) is used.

**px_size**
Unbinned detector pixel size. If not given, `px_size` is used.

**binning**
Detector binning factor. If not given, `binning` is used.

**is_outlier**
Boolean array with `True` for PCs to not include in the fit. If not given, all PCs are used. Must be of `navigation_shape`.

**Returns**

**new_detector**
Detector with *navigation_shape* given by input navigation_shape.

### Examples using EBSDDetector.extrapolate_pc

- *Fit a plane to selected projection centers*
- *Estimate tilt about the detector x axis*
- *Estimate tilts about the detector x and z axis*

### fit_pc

EBSDDetector.**fit_pc**(*pc_indices: list | tuple | ndarray*, *map_indices: list | tuple | ndarray*, *transformation: str = 'projective'*, *is_outlier: ndarray | None = None*, *plot: bool = True*, *return_figure: bool = False*, *figure_kwargs: dict | None = None*) → *EBSDDetector* | Tuple[*EBSDDetector*, Figure]

Return a new detector with interpolated projection centers (PCs) for all points in a map by fitting a plane to *pc* [Winkelmann *et al.*, 2020].

**Parameters**

**pc_indices**
2D coordinates (row, column) of each *pc* in map_coordinates. Must be a flattened array of shape (2,) + *navigation_shape*.

**map_indices**
2D coordinates (row, column) of all map points in a regular grid to interpolate PCs for. Must be a flattened array of shape (2,) + map_shape.

**transformation**
Which transformation function to use when fitting PCs, either "projective" (default) or "affine". Both transformations perserve co-planarity of map points, while the projective transformation allows parallel lines in the map point grid to become non-parallel within the sample plane.

**is_outlier**
Boolean array with True for PCs to not include in the fit. If not given, all PCs are used. Must be of *navigation_shape*.

**plot**
Whether to plot the experimental and estimated PCs (default is True).

**return_figure**
Whether to return the figure if plot=True (default is False).

**figure_kwargs**
Keyword arguments passed to matplotlib.pyplot.Figure() if plot=True.

**Returns**

**new_detector**
New detector with as many interpolated PCs as indices given in map_indices and an estimated sample tilt. The detector tilt is assumed to be constant.

**fig**
Figure of experimental and estimated PCs, returned if `plot=True` and `return_figure=True`.

**Raises**

`ValueError`
If *navigation_size* is 1 or if the `pc_indices` or `map_indices` arrays have the incorrect shape.

**See also:**

*estimate_xtilt*, *estimate_xtilt_ztilt*, *extrapolate_pc*

### Notes

This method is adapted from Aimo Winkelmann's functions `fit_affine()` and `fit_projective()` in the *xcdskd* Python package. Their uses are described in [Winkelmann *et al.*, 2020]. Winkelmann refers to a code example from StackOverflow (https://stackoverflow.com/a/20555267/3228100) for the affine transformation.

### Examples using `EBSDDetector.fit_pc`

- *Fit a plane to selected projection centers*

## get_indexer

`EBSDDetector.get_indexer`(*phase_list: PhaseList*, *reflectors: List['ReciprocalLatticeVector' | np.ndarray | list | tuple | None] | None = None*, ***kwargs*) → EBSDIndexer

Return a PyEBSDIndex EBSD indexer.

**Parameters**

**phase_list**
List of phases. `EBSDIndexer` only supports a list containing one face-centered cubic (FCC) phase, one body-centered cubic (BCC) phase or both.

**reflectors**
List of reflectors or pole families $\{hkl\}$ to use in indexing for each phase. If not passed, the default in `pyebsdindex.tripletvote.addphase()` is used. For each phase, the reflectors can either be a NumPy array, a list, a tuple, a `ReciprocalLatticeVector`, or None.

****kwargs**
Keyword arguments passed to `EBSDIndexer`, except for the following arguments which cannot be passed since they are determined from the detector or `phase_list`: `phaselist` (not to be confused with `phase_list`), `vendor`, `PC`, `sampleTilt`, `camElev` and `patDim`.

**Returns**

`pyebsdindex.ebsd_index.EBSDIndexer`
Indexer instance for use with PyEBSDIndex or in *hough_indexing()*. `indexer.PC` is set equal to *pc_flattened*.

**See also:**

`pyebsdindex.tripletvote.addphase`

### Notes

Requires that PyEBSDIndex is installed, which is an optional dependency of kikuchipy. See *Optional dependencies* for details.

## load

**classmethod** EBSDDetector.**load**(*fname: Path | str*) → *EBSDDetector*

Return an EBSD detector loaded from a text file saved with `save()`.

**Parameters**

**fname**
Full path to file.

**Returns**

**detector**
Loaded EBSD detector.

## pc_bruker

EBSDDetector.**pc_bruker**() → ndarray

Return PC in the Bruker convention, given in the class description.

**Returns**

**new_pc**
PC in the Bruker convention.

## pc_emsoft

EBSDDetector.**pc_emsoft**(*version: int = 5*) → ndarray

Return PC in the EMsoft convention.

**Parameters**

**version**
Which EMsoft PC convention to use. The direction of the x PC coordinate, $x_{pc}$, flipped in version 5.

**Returns**

**new_pc**
PC in the EMsoft convention.

### Notes

The PC coordinate conventions of Bruker, EDAX TSL, Oxford Instruments and EMsoft are given in the class description. The PC is stored in the Bruker convention internally, so the conversion is

$$x_{pc} = N_x b \left( \frac{1}{2} - x_B^* \right),$$
$$y_{pc} = N_y b \left( \frac{1}{2} - y_B^* \right),$$
$$L = N_y b \delta z_B^*,$$

where $N_x$ and $N_y$ are number of detector columns and rows, $b$ is binning, $\delta$ is the unbinned pixel size, $(x_B^*, y_B^*, z_B^*)$ are the Bruker PC coordinates, and $(x_{pc}, y_{pc}, L)$ are the returned EMsoft PC coordinates.

### Examples

```
>>> import kikuchipy as kp
>>> det = kp.detectors.EBSDDetector(
...     shape=(60, 80),
...     pc=(0.4, 0.2, 0.6),
...     convention="bruker",
...     px_size=59.2,
...     binning=8,
... )
>>> det.pc_emsoft()
array([[   64. ,   144. , 17049.6]])
>>> det.pc_emsoft(4)
array([[  -64. ,   144. , 17049.6]])
```

## pc_oxford

EBSDDetector.**pc_oxford**() → ndarray

Return PC in the Oxford convention.

**Returns**

**new_pc**
PC in the Oxford convention.

## pc_tsl

EBSDDetector.**pc_tsl**() → ndarray

Return PC in the EDAX TSL convention.

**Returns**

**new_pc**
PC in the EDAX TSL convention.

### Notes

The PC coordinate conventions of Bruker, EDAX TSL, Oxford Instruments and EMsoft are given in the class description. The PC is stored in the Bruker convention internally, so the conversion is

$$
\begin{aligned}
x_T^* &= x_B^*, \\
y_T^* &= \frac{N_y}{N_x}(1 - y_B^*), \\
z_T^* &= \frac{N_y}{N_x} z_B^*,
\end{aligned}
$$

where $N_x$ and $N_y$ are number of detector columns and rows, $(x_B^*, y_B^*, z_B^*)$ are the Bruker PC coordinates, and $(x_T^*, y_T^*, z_T^*)$ are the returned EDAX TSL PC coordinates.

### Examples

```
>>> import kikuchipy as kp
>>> det = kp.detectors.EBSDDetector(
...     shape=(60, 80),
...     pc=(0.4, 0.2, 0.6),
...     convention="bruker",
... )
>>> det.pc_tsl()
array([[0.4 , 0.6 , 0.45]])
```

## plot

EBSDDetector.**plot**(*coordinates: str = 'detector', show_pc: bool = True, pc_kwargs: dict | None = None, pattern: ndarray | None = None, pattern_kwargs: dict | None = None, draw_gnomonic_circles: bool = False, gnomonic_angles: None | list | ndarray = None, gnomonic_circles_kwargs: dict | None = None, zoom: float = 1, return_figure: bool = False*) → None | Figure

Plot the detector screen viewed from the detector towards the sample.

The plotting of gnomonic circles and general style is adapted from the supplementary material to [Britton *et al.*, 2016] by Aimo Winkelmann.

**Parameters**

**coordinates**
Which coordinates to use, `"detector"` (default) or `"gnomonic"`.

**show_pc**
Show the average projection center in the Bruker convention. Default is `True`.

**pc_kwargs**
A dictionary of keyword arguments passed to `matplotlib.axes.Axes.scatter()`.

**pattern**
A pattern to put on the detector. If not given, no pattern is displayed. The pattern array must have the same shape as the detector.

**pattern_kwargs**
A dictionary of keyword arguments passed to `matplotlib.axes.Axes.imshow()`.

**draw_gnomonic_circles**
Draw circles for angular distances from pattern. Default is `False`. Circle positions are only correct when `coordinates="gnomonic"`.

**gnomonic_angles**
Which angular distances to plot if `draw_gnomonic_circles=True`. Default is from 10 to 80 in steps of 10.

**gnomonic_circles_kwargs**
A dictionary of keyword arguments passed to `matplotlib.patches.Circle()`.

**zoom**
Whether to zoom in/out from the detector, e.g. to show the extent of the gnomonic projection circles. A zoom > 1 zooms out. Default is 1, i.e. no zoom.

**return_figure**
Whether to return the figure. Default is False.

**Returns**

**fig**
Matplotlib figure instance, if *return_figure* is True.

### Examples

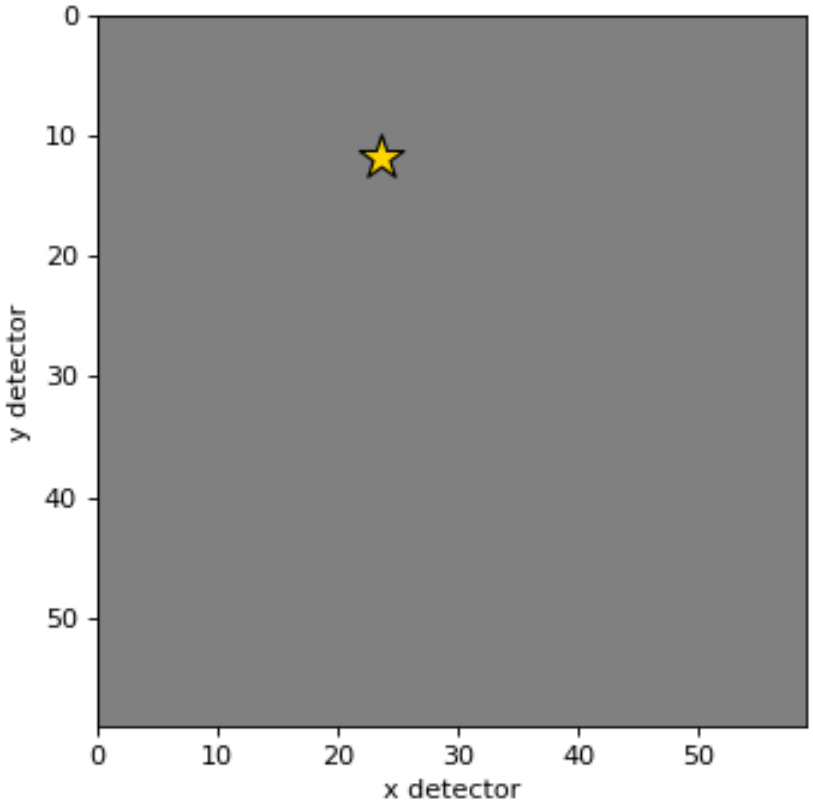
```
>>> import matplotlib.pyplot as plt
>>> import kikuchipy as kp
>>> det = kp.detectors.EBSDDetector(
...     shape=(60, 60),
...     pc=(0.4, 0.8, 0.5),
...     convention="tsl",
...     sample_tilt=70,
... )
>>> det.plot()
>>> plt.show()
```

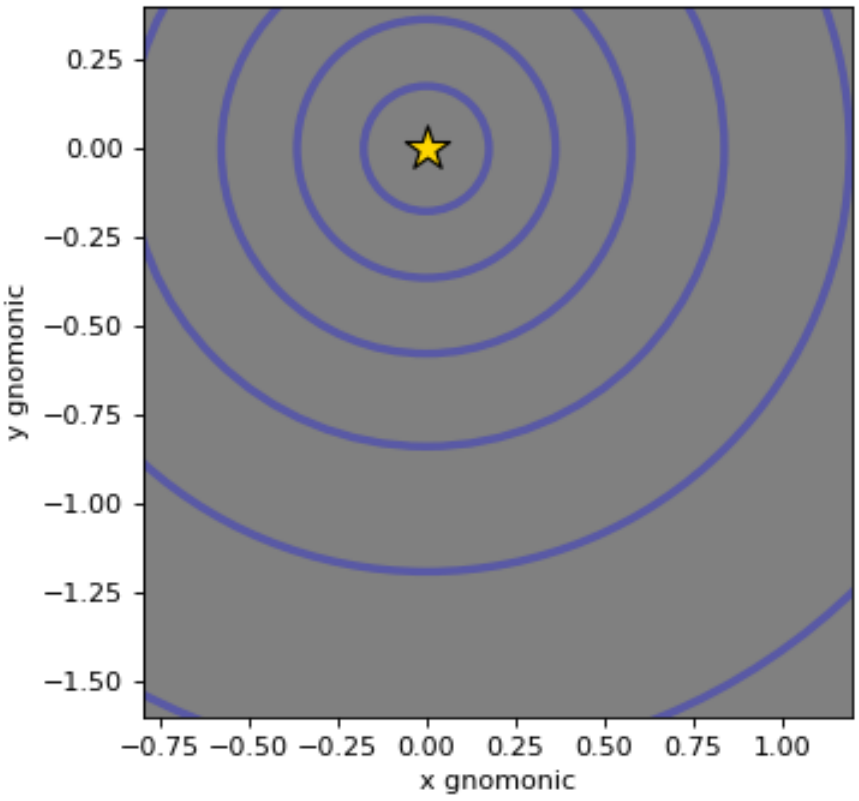
Plot with gnomonic coordinates and circles

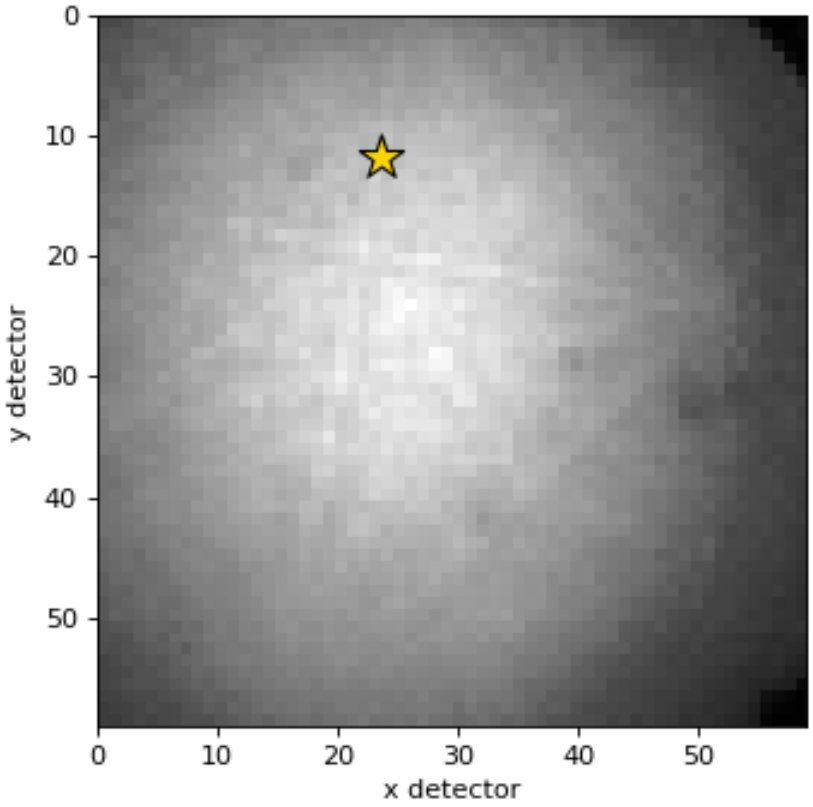
```
>>> det.plot(
...     coordinates="gnomonic",
...     draw_gnomonic_circles=True,
...     gnomonic_circles_kwargs={"edgecolor": "b", "alpha": 0.3}
... )
>>> plt.show()
```

Plot a pattern on the detector and return it for saving etc.

```
>>> s = kp.data.nickel_ebsd_small()
>>> fig = det.plot(pattern=s.inav[0, 0].data, return_figure=True)
```


0
10
20
30
40
50
y detector
0
10
20
30
40
50
x detector


0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
y gnomonic
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x gnomonic


0
10
20
30
40
50
y detector
0
10
20
30
40
50
x detector

### plot_pc

EBSDDetector.**plot_pc**(*mode: str = 'map'*, *return_figure: bool = False*, *orientation: str = 'horizontal'*, *annotate: bool = False*, *figure_kwargs: dict | None = None*, ***kwargs*) → None | Figure

Plot all projection centers (PCs).

**Parameters**

**mode**
String describing how to plot PCs. Options are `"map"` (default), `"scatter"` and `"3d"`. If `mode="map"`, *navigation_dimension* must be 2.

**return_figure**
Whether to return the figure (default is `False`).

**orientation**
Whether to align the plots in a `"horizontal"` (default) or `"vertical"` orientation.

**annotate**
Whether to label each pattern with its 1D index into *pc_flattened* when `mode="scatter"`. Default is `False`.

**figure_kwargs**
Keyword arguments to pass to `matplotlib.pyplot.figure()` upon figure creation. Note that `layout="tight"` is used by default unless another layout is passed.

****kwargs**
Keyword arguments passed to the plotting function, which is `imshow()` if `mode="map"`, `scatter()` if `mode="scatter"` and `scatter()` if `mode="3d"`.

**Returns**

**fig**
Figure is returned if `return_figure=True`.

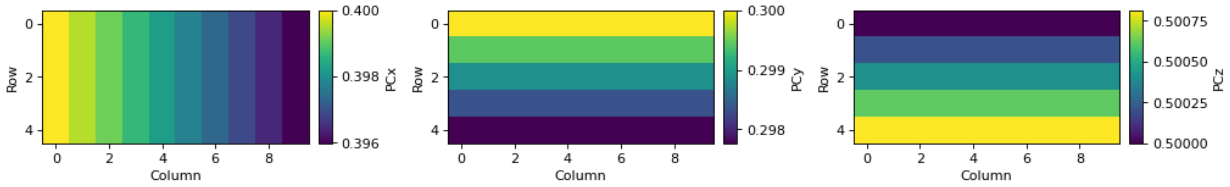
#### Examples

Create a detector with smoothly changing PC values, extrapolated from a single PC (assumed to be in the upper left corner of a map)

```
>>> import matplotlib.pyplot as plt
>>> import kikuchipy as kp
>>> det0 = kp.detectors.EBSDDetector(
... shape=(480, 640), pc=(0.4, 0.3, 0.5), px_size=70, sample_tilt=70
... )
>>> det0
EBSDDetector (480, 640), px_size 70 um, binning 1, tilt 0, azimuthal 0, pc (0.4,
↪ 0.3, 0.5)
>>> det = det0.extrapolate_pc(
... pc_indices=[0, 0], navigation_shape=(5, 10), step_sizes=(20, 20)
... )
>>> det
EBSDDetector (480, 640), px_size 70 um, binning 1, tilt 0, azimuthal 0, pc (0.
↪398, 0.299, 0.5)
```

Plot PC values in maps

```
>>> det.plot_pc()
>>> plt.show()
```




Plot in scatter plots in vertical orientation

```
>>> det.plot_pc("scatter", orientation="vertical", annotate=True)
>>> plt.show()
```

Plot in a 3D scatter plot, returning the figure for saving etc.

```
>>> fig = det.plot_pc("3d", return_figure=True)
```

### Examples using `EBSDDetector.plot_pc`

- *Estimate tilts about the detector x and z axis*

### save

`EBSDDetector.save`(*filename: str*, *convention: str = 'Bruker'*, ***kwargs*) → None

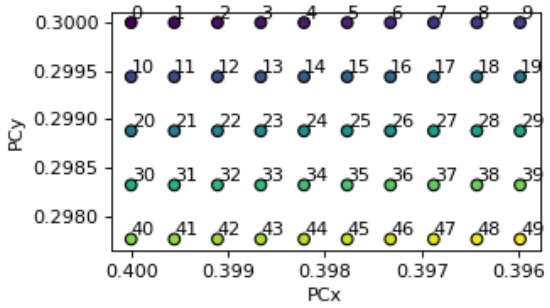
Save detector in a text file with projection centers (PCs) in the given convention.

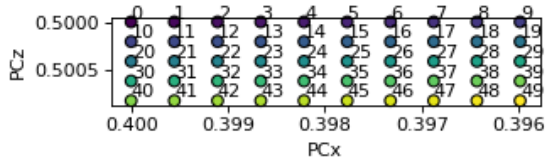
**Parameters**

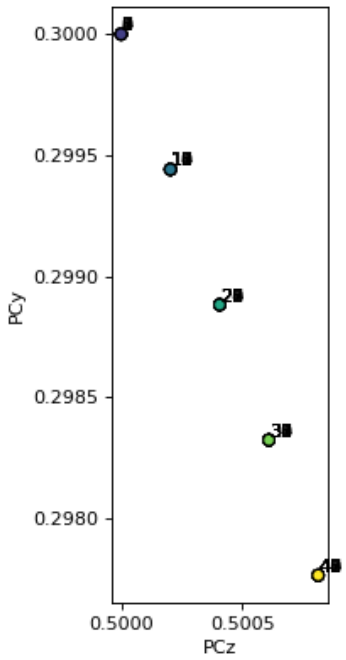
**filename**
Name of text file to write to. See `savetxt()` for supported file formats.

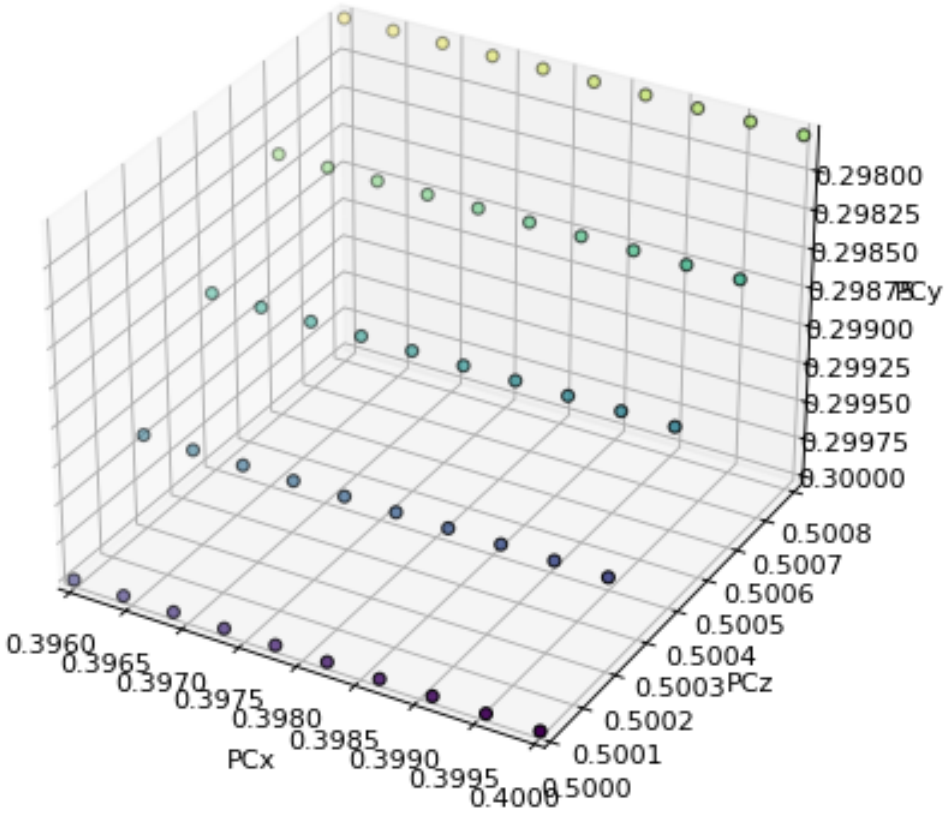
**convention**
PC convention. Default is Bruker's convention. Options are "tsl"/"edax", "oxford", "bruker", "emsoft", "emsoft4", and "emsoft5". "emsoft" and "emsoft5" is the same convention. See *Notes* in *`EBSDDetector`* for conversions between conventions.

****kwargs**
Keyword arguments passed to `savetxt()`, e.g. `fmt="%.4f"` to reduce the number of PC decimals from the default 7 to 4.


0.3000
0.2995
0.2990
0.2985
0.2980
PCy
0.400
0.399
0.398
0.397
0.396
PCx


0.5000
0.5005
PCz
0.400
0.399
0.398
0.397
0.396
PCx


0.3000
0.2995
0.2990
0.2985
0.2980
PCy
0.5000
0.5005
PCz


0.29800
0.29825
0.29850
0.29875
PCy
0.29900
0.29925
0.29950
0.29975
0.30000
0.5008
0.5007
0.5006
0.5005
0.5004
0.5003
PCz
0.5002
0.5001
0.5000
0.3960
0.3965
0.3970
0.3975
0.3980
0.3985
PCx
0.3990
0.3995
0.4000

### Examples using `EBSDDetector`

- *Adaptive histogram equalization*
- *Fit a plane to selected projection centers*
- *Estimate tilt about the detector x axis*
- *Estimate tilts about the detector x and z axis*

## 2.4.2 PCCalibrationMovingScreen

**class** kikuchipy.detectors.**PCCalibrationMovingScreen**(*pattern_in: ndarray*, *pattern_out: ndarray*, *points_in: ndarray | List[Tuple[float]]*, *points_out: ndarray | List[Tuple[float]]*, *delta_z: float = 1.0*, *px_size: float | None = None*, *binning: int = 1*, *convention: str = 'tsl'*)

Bases: `object`

A class to perform and inspect the calibration of the EBSD projection center (PC) using the "moving screen" technique from [Hjelen *et al.*, 1991].

The technique requires two patterns acquired with a stationary beam but with different specimen detector distances (SDDs) where the difference is known. First, the goal is to find the pattern region which does not shift between the two camera positions, (PCx, PCy). This point can be estimated by selecting the same pattern features in both patterns. Second, the DD (PCz) can be estimated in the same unit as the known camera distance difference. If also the detector pixel size is known, PCz can be given in the fraction of the detector screen height.

**Parameters**

**pattern_in**
Pattern acquired with the shortest detector distance (DD) in the "in" position.

**pattern_out**
Pattern acquired with the longer DD in the "out" position, with the camera a known distance `delta_z` from the "in" position.

**points_in**
Set of $n$ coordinates `[(x1, y1), (x2, y2), ...]` of pattern features in `pattern_in`.

**points_out**
Set of $n$ coordinates `[(x1, y1), (x2, y2), ...]` of pattern features, the same as in `points_in`, in `pattern_out`. They must be in the same order as in `points_in`.

**delta_z**
Known distance between the "in" and "out" camera positions in which the `pattern_in` and `pattern_out` were acquired, respectively. Default is `1.0`. The output PCz value will be in the same unit as this value, unless `px_size` is provided.

**px_size**
Known size of the detector pixels, in the same unit as `delta_z`. If not given (default), the PCz will not be scaled to fractions of detector height.

**binning**
Detector pixel binning. Default is 1, meaning no binning. This is used together with `px_size` to scale PCz.

**convention**
Whether to present PCy as the value from bottom to top (TSL), or top to bottom (Bruker). Default is `"tsl"`.

### Attributes

| | |
|---|---|
| *PCCalibrationMovingScreen.line_lengths* | Return the length of lines within the patterns in pixels. |
| *PCCalibrationMovingScreen.lines* | Return the start and end points of all possible lines between all points per pattern, of shape `(2, n_lines, 4)`, where the last axis is `(x1, y1, x2, y2)`. |
| *PCCalibrationMovingScreen.lines_end* | Return the end points of lines within both patterns, of shape `(2, n_lines, 2)`. |
| *PCCalibrationMovingScreen.lines_out_in* | Return the start (out) and end (in) points of the lines between corresponding points in the patterns, of shape `(n_points, 4)`. |
| *PCCalibrationMovingScreen.lines_out_in_end* | Return the end points of the lines between corresponding points in the patterns, of shape `(n_points, 2)`. |
| *PCCalibrationMovingScreen.lines_out_in_start* | Return the starting points of the lines between corresponding points in the patterns, of shape `(n_points, 2)`. |
| *PCCalibrationMovingScreen.lines_start* | Return the starting points of lines within the patterns, of shape `(2, n_lines, 2)`. |
| *PCCalibrationMovingScreen.n_lines* | Return the number of lines in each pattern. |
| *PCCalibrationMovingScreen.n_points* | Return the number of points of pattern features in each pattern. |
| *PCCalibrationMovingScreen.ncols* | Return the number of detector columns. |
| *PCCalibrationMovingScreen.nrows* | Return the number of detector rows. |
| *PCCalibrationMovingScreen.pc* | Return the average PC calculated from all estimates. |
| *PCCalibrationMovingScreen.pc_all* | Return all estimates of PC. |
| *PCCalibrationMovingScreen.pcx_all* | Return all estimates of PCx. |
| *PCCalibrationMovingScreen.pcy_all* | Return all estimates of PCy. |
| *PCCalibrationMovingScreen.pcz_all* | Return all estimates of PCz, scaled to fraction of detector height if `px_size` is not `None`. |
| *PCCalibrationMovingScreen.pxy* | Return the average of intersections of the lines between corresponding points in the patterns. |
| *PCCalibrationMovingScreen.pxy_all* | Return the intersections of the lines between the corresponding points in the patterns, i.e. estimates of (PCx, PCy), of shape `(n_points, 2)`. |
| *PCCalibrationMovingScreen.pxy_within_detector* | Return the boolean array stating whether each intersection of lines between corresponding points in the patterns are inside the detector (`True`), or outside (`False`). |
| *PCCalibrationMovingScreen.shape* | Return the detector shape, (nrows, ncols). |

### line_lengths

**property** PCCalibrationMovingScreen.**line_lengths: ndarray**

Return the length of lines within the patterns in pixels.

### lines

**property** PCCalibrationMovingScreen.**lines: ndarray**

Return the start and end points of all possible lines between all points per pattern, of shape `(2, n_lines, 4)`, where the last axis is `(x1, y1, x2, y2)`.

### lines_end

**property** PCCalibrationMovingScreen.**lines_end: ndarray**

Return the end points of lines within both patterns, of shape `(2, n_lines, 2)`.

### lines_out_in

**property** PCCalibrationMovingScreen.**lines_out_in: ndarray**

Return the start (out) and end (in) points of the lines between corresponding points in the patterns, of shape `(n_points, 4)`.

### lines_out_in_end

**property** PCCalibrationMovingScreen.**lines_out_in_end: ndarray**

Return the end points of the lines between corresponding points in the patterns, of shape `(n_points, 2)`.

### lines_out_in_start

**property** PCCalibrationMovingScreen.**lines_out_in_start: ndarray**

Return the starting points of the lines between corresponding points in the patterns, of shape `(n_points, 2)`.

### lines_start

**property** PCCalibrationMovingScreen.**lines_start: ndarray**

Return the starting points of lines within the patterns, of shape `(2, n_lines, 2)`.

### n_lines

**property** PCCalibrationMovingScreen.**n_lines: int**

Return the number of lines in each pattern.

#### n_points

property PCCalibrationMovingScreen.**n_points: int**

Return the number of points of pattern features in each pattern.

#### ncols

property PCCalibrationMovingScreen.**ncols: int**

Return the number of detector columns.

#### nrows

property PCCalibrationMovingScreen.**nrows: int**

Return the number of detector rows.

#### pc

property PCCalibrationMovingScreen.**pc: ndarray**

Return the average PC calculated from all estimates.

#### pc_all

property PCCalibrationMovingScreen.**pc_all: ndarray**

Return all estimates of PC.

#### pcx_all

property PCCalibrationMovingScreen.**pcx_all: ndarray**

Return all estimates of PCx.

#### pcy_all

property PCCalibrationMovingScreen.**pcy_all: ndarray**

Return all estimates of PCy.

#### pcz_all

property PCCalibrationMovingScreen.**pcz_all: ndarray**

Return all estimates of PCz, scaled to fraction of detector height if `px_size` is not `None`.

### pxy

**property** PCCalibrationMovingScreen.**pxy: ndarray**

Return the average of intersections of the lines between corresponding points in the patterns.

### pxy_all

**property** PCCalibrationMovingScreen.**pxy_all: ndarray**

Return the intersections of the lines between the corresponding points in the patterns, i.e. estimates of (PCx, PCy), of shape `(n_points, 2)`.

### pxy_within_detector

**property** PCCalibrationMovingScreen.**pxy_within_detector: ndarray**

Return the boolean array stating whether each intersection of lines between corresponding points in the patterns are inside the detector (`True`), or outside (`False`).

### shape

**property** PCCalibrationMovingScreen.**shape: Tuple[int, int]**

Return the detector shape, (nrows, ncols).

### Methods

| | |
|---|---|
| *PCCalibrationMovingScreen.make_lines*() | Draw lines between all points within a pattern and populate *self.lines*. |
| *PCCalibrationMovingScreen.plot*([...]) | A convenience method of three images, the first two with the patterns with points and lines annotated, and the third with the calibration results. |

### make_lines

PCCalibrationMovingScreen.**make_lines**()

Draw lines between all points within a pattern and populate *self.lines*. Is first run upon initialization.

### plot

PCCalibrationMovingScreen.**plot**(*pattern_kwargs: dict = {'cmap': 'gray'}*, *line_kwargs: dict = {'linewidth': 2, 'zorder': 1}*, *scatter_kwargs: dict = {'zorder': 2}*, *pc_kwargs: dict = {'edgecolor': 'k', 'facecolor': 'gold', 'marker': '*', 's': 300}*, *return_figure: bool = False*, ***kwargs: dict*) → None | Tuple[Figure, List[Axes]]

A convenience method of three images, the first two with the patterns with points and lines annotated, and the third with the calibration results.

**Parameters**

**pattern_kwargs**
Keyword arguments passed to `matplotlib.axes.Axes.imshow()`.

**line_kwargs**
Keyword arguments passed to `matplotlib.axes.Axes.axline()`.

**scatter_kwargs**
Keyword arguments passed to `matplotlib.axes.Axes.scatter()`.

**pc_kwargs**
Keyword arguments, along with *scatter_kwargs*, passed to `matplotlib.axes.Axes.scatter()` when plotting the PCs.

**return_figure**
Whether to return the figure and axes, default is `False`.

****kwargs**
Keyword arguments passed to `matplotlib.pyplot.subplots()`.

**Returns**

**fig**
Figure, returned if `return_figure=True`.

## 2.5 draw

Tools for use in plotting of signals.

### Functions

| | |
|---|---|
| *get_rgb_navigator*(image[, dtype]) | Create an RGB navigator signal which is suitable to pass to `plot()` as the `navigator` parameter. |
| *plot_pattern_positions_in_map*(rc, roi_shape) | Plot pattern positions in a 2D map within a region of interest (ROI), the ROI potentially within a larger area. |

### 2.5.1 get_rgb_navigator

kikuchipy.draw.**get_rgb_navigator**(*image: ndarray*, *dtype: str | dtype | type = 'uint16'*) → Signal2D

Create an RGB navigator signal which is suitable to pass to `plot()` as the `navigator` parameter.

**Parameters**

**image**
RGB color image of shape `(n rows, n columns, 3)`.

**dtype**
Which data type to cast the signal data to, either `"uint16"` (default) or `"uint8"`. Must be a valid `numpy.dtype` identifier.

**Returns**

**s**
Signal with an (n columns, n rows) signal shape and no navigation shape, of data type either `rgb8` or `rgb16`.

### 2.5.2 plot_pattern_positions_in_map

kikuchipy.draw.**plot_pattern_positions_in_map**(*rc: ndarray*, *roi_shape: tuple*, *roi_origin: tuple = (0, 0)*, *area_shape: tuple | None = None*, *roi_image: ndarray | None = None*, *area_image: ndarray | None = None*, *axis: Axes | None = None*, *return_figure: bool = False*, *color: str | None = 'k'*) → Figure | None

Plot pattern positions in a 2D map within a region of interest (ROI), the ROI potentially within a larger area.

**Parameters**

**rc**
Position coordinates (row, column) in an array of shape (n, 2). If `area_shape` is passed, coordinates are assumed to be given with respect to the area origin, and so if `roi_origin` is passed, the origin is subtracted from the coordinates.

**roi_shape**
Shape of the ROI as (n rows, n columns).

**roi_origin**
Origin (row, column) of the ROI with respect to the area. If this and `area_shape` is passed, the origin is subtracted from the `rc` coordinates.

**area_shape**
Shape of the area including the ROI as (n rows, n columns). If this and `roi_origin` is passed, the origin is subtracted from the `rc` coordinates.

**roi_image**
Image to plot within the ROI, of the same aspect ratio.

**area_image**
Image to plot within the area, of the same aspect ratio.

**axis**
Existing Matplotlib axis to add the positions to. If not passed, a new figure will be created. If passed, only the coordinate markers and labels are added to the axis. E.g. `roi_image` or `area_image` will not be used.

**return_figure**
Whether to return the created figure. Default is `False`.

**color**
Color of position markers and labels. Default is `k`. Must be a valid Matplotlib color identifier.

**Returns**

**fig**
Created figure, returned if `return_figure=True`.

### Examples using `plot_pattern_positions_in_map`

- *Extract patterns from a grid*

## 2.6 filters

Pattern filters used on signals, e.g. for pattern averaging.

### Functions

| | |
|---|---|
| *distance_to_origin*(shape[, origin]) | Return the distance to the window origin in pixels. |
| *highpass_fft_filter*(shape, cutoff[, ...]) | Return a frequency domain high-pass filter transfer function in 2D. |
| *lowpass_fft_filter*(shape, cutoff[, cutoff_width]) | Return a frequency domain low-pass filter transfer function in 2D. |
| *modified_hann*(Nx) | Return a 1D modified Hann window with the maximum value normalized to 1. |

### 2.6.1 distance_to_origin

kikuchipy.filters.**distance_to_origin**(*shape: Tuple[int] | Tuple[int, int]*, *origin: None | Tuple[int] | Tuple[int, int] = None*) → ndarray

Return the distance to the window origin in pixels.

**Parameters**

**shape**
Window shape.

**origin**
Window origin. If not given, half the shape is used as origin for each axis.

**Returns**

**distance**
Distance to the window origin in pixels.

### 2.6.2 highpass_fft_filter

kikuchipy.filters.**highpass_fft_filter**(*shape: Tuple[int, int]*, *cutoff: int | float*, *cutoff_width: None | int | float = None*) → ndarray

Return a frequency domain high-pass filter transfer function in 2D.

Used in [Wilkinson *et al.*, 2006].

**Parameters**

**shape**
Shape of function.

**cutoff**
Cut-off frequency.

**cutoff_width**
Width of cut-off region. If not given (default), it is set to half of the cutoff frequency.

**Returns**

**window**
2D transfer function.

### Notes

The high-pass filter transfer function is defined as

$$w(r) = e^{-\left((c-r)/(\sqrt{2}w_c/2)\right)^2}, w(r) = \begin{cases} 0, & r < c - 2w_c \\ 1, & r > c, \end{cases}$$

where $r$ is the radial distance to the window centre, $c$ is the cut-off frequency, and $w_c$ is the width of the cut-off region.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> w1 = kp.filters.Window(
...     "highpass", cutoff=1, cutoff_width=0.5, shape=(96, 96)
... )
>>> w2 = kp.filters.highpass_fft_filter(
...     shape=(96, 96), cutoff=1, cutoff_width=0.5
... )
>>> np.allclose(w1, w2)
True
```

## 2.6.3 lowpass_fft_filter

kikuchipy.filters.**lowpass_fft_filter**(*shape: Tuple[int, int]*, *cutoff: int | float*, *cutoff_width: None | int | float = None*) → ndarray

Return a frequency domain low-pass filter transfer function in 2D.

Used in [Wilkinson *et al.*, 2006].

**Parameters**

**shape**
Shape of function.

**cutoff**
Cut-off frequency.

**cutoff_width**
Width of cut-off region. If None (default), it is set to half of the cutoff frequency.

**Returns**

**window**
2D transfer function.

### Notes

The low-pass filter transfer function is defined as

$$w(r) = e^{-\left((r-c)/(\sqrt{2}w_c/2)\right)^2}, w(r) = \begin{cases} 0, & r > c + 2w_c \\ 1, & r < c, \end{cases}$$

where $r$ is the radial distance to the window centre, $c$ is the cut-off frequency, and $w_c$ is the width of the cut-off region.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> w1 = kp.filters.Window(
...     "lowpass", cutoff=30, cutoff_width=15, shape=(96, 96)
... )
>>> w2 = kp.filters.lowpass_fft_filter(
...     shape=(96, 96), cutoff=30, cutoff_width=15
... )
>>> np.allclose(w1, w2)
True
```

## 2.6.4 modified_hann

kikuchipy.filters.**modified_hann**(*Nx: int*) → ndarray

Return a 1D modified Hann window with the maximum value normalized to 1.

Used in [Wilkinson *et al.*, 2006].

**Parameters**

**Nx**
Number of points in the window.

**Returns**

**window**
1D Hann window.

### Notes

The modified Hann window is defined as

$$w(x) = \cos\left(\frac{\pi x}{N_x}\right),$$

with $x$ relative to the window centre.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> w1 = kp.filters.modified_hann(Nx=30)
>>> w2 = kp.filters.Window("modified_hann", shape=(30,))
>>> np.allclose(w1, w2)
True
```

### Classes

| | |
|---|---|
| *Window*([window, shape]) | A window/kernel/mask/filter of a given shape with some values. |

## 2.6.5 Window

**class** kikuchipy.filters.**Window**(*window: None | str | ndarray | Array = None*, *shape: Sequence[int] | None = None*, ***kwargs*)

Bases: ndarray

A window/kernel/mask/filter of a given shape with some values.

This class is a subclass of numpy.ndarray with some additional convenience methods.

It can be used to create a transfer function for filtering in the frequency domain, create an averaging window for averaging patterns with their nearest neighbours, and so on.

**Parameters**

**window**
: Window type to create. Available types are listed in scipy.signal.windows.get_window() and includes "rectangular" and "gaussian", in addition to a "circular" window (default) filled with ones in which corner data are set to zero, a "modified_hann" window and "lowpass" and "highpass" FFT windows. A window element is considered to be in a corner if its radial distance to the origin (window center) is shorter or equal to the half width of the windows's longest axis. A 1D or 2D numpy.ndarray or dask.array.Array can also be passed.

**shape**
: Shape of the window. Not used if a custom window is passed to window. This can be either 1D or 2D, and can be asymmetrical. Default is (3, 3).

****kwargs**
: Required keyword arguments passed to the window type.

**See also:**

scipy.signal.windows.get_window

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
```

The following passed parameters are the default

```
>>> w = kp.filters.Window(window="circular", shape=(3, 3))
>>> w
Window (3, 3) circular
[[0. 1. 0.]
 [1. 1. 1.]
 [0. 1. 0.]]
```

A window can be made circular

```
>>> w = kp.filters.Window(window="rectangular")
>>> w
Window (3, 3) rectangular
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
>>> w.make_circular()
>>> w
Window (3, 3) circular
[[0. 1. 0.]
 [1. 1. 1.]
 [0. 1. 0.]]
```

A custom window can be created

```
>>> w = kp.filters.Window(np.arange(6).reshape(3, 2))
>>> w
Window (3, 2) custom
[[0 1]
 [2 3]
 [4 5]]
```

To create a Gaussian window with a standard deviation of 2, obtained from `scipy.signal.windows.gaussian()`

```
>>> w = kp.filters.Window(window="gaussian", std=2)
>>> w
Window (3, 3) gaussian
[[0.7788 0.8825 0.7788]
 [0.8825 1.     0.8825]
 [0.7788 0.8825 0.7788]]
```

### Attributes

| | |
|---|---|
| *Window.circular* | Return whether the window is circular. |
| *Window.distance_to_origin* | Return the radial distance for each pixel to the window origin. |
| *Window.is_valid* | Return whether the window is in a valid state. |
| *Window.n_neighbours* | Return the maximum number of nearest neighbours in each navigation axis to the origin. |
| *Window.name* | Return the name of the window. |
| *Window.origin* | Return the window origin. |

#### circular

**property** Window.**circular: bool**

Return whether the window is circular.

#### distance_to_origin

**property** Window.**distance_to_origin: ndarray**

Return the radial distance for each pixel to the window origin.

#### is_valid

**property** Window.**is_valid: bool**

Return whether the window is in a valid state.

#### n_neighbours

**property** Window.**n_neighbours: tuple**

Return the maximum number of nearest neighbours in each navigation axis to the origin.

#### name

**property** Window.**name: str**

Return the name of the window.

#### origin

**property** Window.**origin: tuple**

Return the window origin.

### Methods

| | |
|---|---|
| *Window.make_circular*() | Make the window circular. |
| *Window.plot*([grid, show_values, textcolors, ...]) | Plot window values with indices relative to the origin. |
| *Window.shape_compatible*(shape) | Return whether the window shape is compatible with another shape. |

### make_circular

Window.**make_circular**()

Make the window circular.

The data of window elements who's radial distance to the window origin is shorter or equal to the half width of the window's longest axis are set to zero. This has no effect if the window has only one axis.

### plot

Window.**plot**(*grid: bool = True*, *show_values: bool = True*, *textcolors: List[str] | None = None*, *cmap: str = 'viridis'*, *cmap_label: str = 'Value'*, *colorbar: bool = True*, *return_figure: bool = False*) → Figure

Plot window values with indices relative to the origin.

**Parameters**

**grid**
Whether to separate each value with a white spacing in a grid. Default is `True`.

**show_values**
Whether to show values as text in centre of element. Default is `True`.

**textcolors**
A list of two color specifications. The first is used for values below a threshold, the second for those above. If not given (default), this is set to `["white", "black"]`.

**cmap**
A colormap to color data with, available in `matplotlib.colors.ListedColormap`. Default is `"viridis"`.

**cmap_label**
Colormap label. Default is `"Value"`.

**colorbar**
Whether to show the colorbar. Default is `True`.

**return_figure**
Whether to return the figure. Default is `False`.

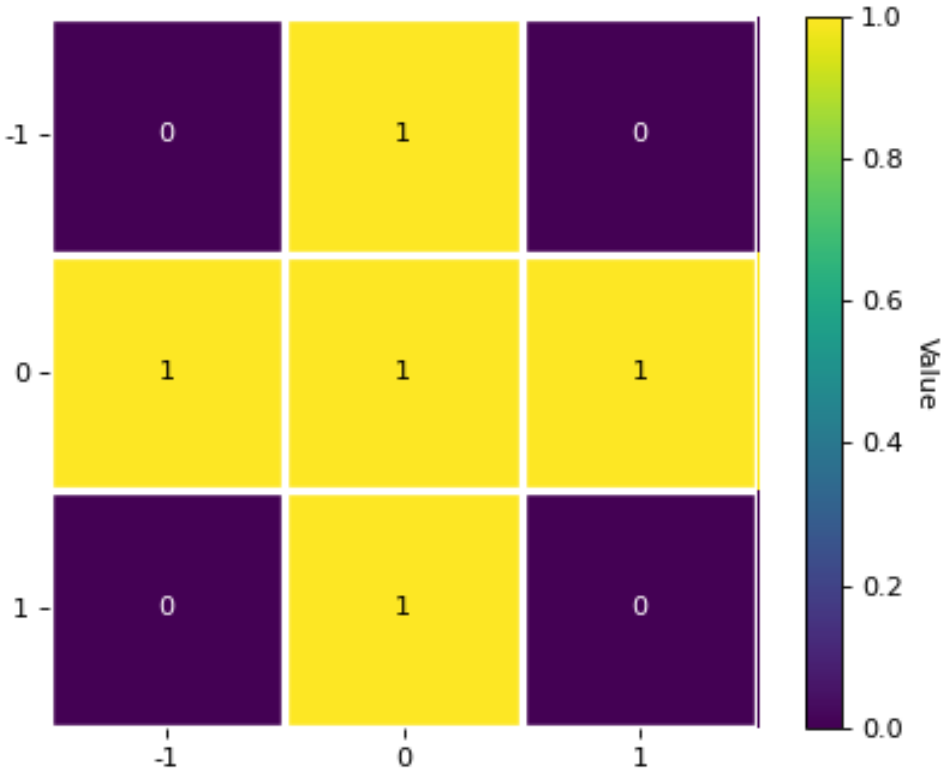
**Returns**

**fig**
Figure returned if `return_figure=True`.

### Examples

A plot of window data with indices relative to the origin, showing element values and x/y ticks, can be produced and written to file

```
>>> import kikuchipy as kp
>>> w = kp.filters.Window()
>>> fig = w.plot(return_figure=True)
>>> fig.savefig('my_kernel.png')
```




### shape_compatible

Window.**shape_compatible**(*shape: Tuple[int]*) → bool

Return whether the window shape is compatible with another shape.

**Parameters**

**shape**
Shape of data to apply window to.

**Returns**

**is_compatible**
Whether the window shape is compatible with another shape.

## 2.7 imaging

Imaging using the EBSD detector.

### Classes

| | |
|---|---|
| *VirtualBSEImager*(signal) | Generate virtual backscatter electron (BSE) images for an electron backscatter diffraction (EBSD) signal and a set of EBSD detector areas in a convenient manner. |

### 2.7.1 VirtualBSEImager

**class** kikuchipy.imaging.**VirtualBSEImager**(*signal:* EBSD | LazyEBSD)

Bases: `object`

Generate virtual backscatter electron (BSE) images for an electron backscatter diffraction (EBSD) signal and a set of EBSD detector areas in a convenient manner.

**Parameters**

**signal**
EBSD signal.

**See also:**

***kikuchipy.signals.EBSD.plot_virtual_bse_intensity***
***kikuchipy.signals.EBSD.get_virtual_bse_intensity***

#### Attributes

| | |
|---|---|
| *VirtualBSEImager.grid_cols* | Return the detector grid columns, defined by *grid_shape*. |
| *VirtualBSEImager.grid_rows* | Return the detector grid rows, defined by *grid_shape*. |
| *VirtualBSEImager.grid_shape* | Return or set the generator grid shape. |

#### grid_cols

**property** VirtualBSEImager.**grid_cols: ndarray**

Return the detector grid columns, defined by *grid_shape*.

### grid_rows

**property** VirtualBSEImager.**grid_rows: ndarray**

Return the detector grid rows, defined by *grid_shape*.

### grid_shape

**property** VirtualBSEImager.**grid_shape: tuple**

Return or set the generator grid shape.

**Parameters**

**shape**
[tuple or list of int] Generator grid shape.

### Methods

| | |
|---|---|
| *VirtualBSEImager.get_images_from_grid*([...]) | Return an in-memory signal with a stack of virtual backscatter electron (BSE) images by integrating the intensities within regions of interest (ROI) defined by the image generator *grid_shape*. |
| *VirtualBSEImager.get_rgb_image*(r, g, b[, ...]) | Return an in-memory RGB virtual BSE image from three regions of interest (ROIs) on the EBSD detector, with a potential "alpha channel" in which all three arrays are multiplied by a fourth. |
| *VirtualBSEImager.plot_grid*([pattern_idx, ...]) | Plot a pattern with the detector grid superimposed, potentially coloring the edges of three grid tiles red, green and blue. |
| *VirtualBSEImager.roi_from_grid*(index) | Return a rectangular region of interest (ROI) on the EBSD detector from one or multiple grid tile indices as row(s) and column(s). |

### get_images_from_grid

VirtualBSEImager.**get_images_from_grid**(*dtype_out: str | dtype | type = 'float32'*) → *VirtualBSEImage*

Return an in-memory signal with a stack of virtual backscatter electron (BSE) images by integrating the intensities within regions of interest (ROI) defined by the image generator *grid_shape*.

**Parameters**

**dtype_out**
Output data type, default is "float32".

**Returns**

**vbse_images**
In-memory signal with virtual BSE images.

### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
>>> vbse_imager = kp.imaging.VirtualBSEImager(s)
>>> vbse_imager.grid_shape = (5, 5)
>>> vbse = vbse_imager.get_images_from_grid()
>>> vbse
<VirtualBSEImage, title: , dimensions: (5, 5|3, 3)>
```

## get_rgb_image

VirtualBSEImager.**get_rgb_image**(*r: BaseInteractiveROI | Tuple | List[BaseInteractiveROI] | List[Tuple], g: BaseInteractiveROI | Tuple | List[BaseInteractiveROI] | List[Tuple], b: BaseInteractiveROI | Tuple | List[BaseInteractiveROI] | List[Tuple], percentiles: Tuple | None = None, normalize: bool = True, alpha: None | ndarray | VirtualBSEImage = None, dtype_out: str | dtype | type = 'uint8', add_bright: int = 0, contrast: float = 1.0*) → *VirtualBSEImage*

Return an in-memory RGB virtual BSE image from three regions of interest (ROIs) on the EBSD detector, with a potential "alpha channel" in which all three arrays are multiplied by a fourth.

**Parameters**

**r**
One ROI or a list of ROIs, or one tuple or a list of tuples with detector grid indices specifying one or more ROI(s). Intensities within the specified ROI(s) are summed up to form the red color channel.

**g**
One ROI or a list of ROIs, or one tuple or a list of tuples with detector grid indices specifying one or more ROI(s). Intensities within the specified ROI(s) are summed up to form the green color channel.

**b**
One ROI or a list of ROIs, or one tuple or a list of tuples with detector grid indices specifying one or more ROI(s). Intensities within the specified ROI(s) are summed up to form the blue color channel.

**percentiles**
Whether to apply contrast stretching with a given percentile tuple with percentages, e.g. (0.5, 99.5), after creating the RGB image. If not given (default), no contrast stretching is performed.

**normalize**
Whether to normalize the individual images (channels) before RGB image creation.

**alpha**
"Alpha channel". If not given (default), no "alpha channel" is added to the image.

**dtype_out**
Output data type, either `"uint8"` (default) or `"uint16"`.

**add_bright**
Brightness offset to for each array. Default is `0`.

**contrast**
Contrast factor for each array. Default is `1.0`.

**Returns**

**vbse_rgb_image**
Virtual RGB image in memory.

### Notes

HyperSpy only allows for RGB signal dimensions with data types unsigned 8 or 16 bit.

## plot_grid

`VirtualBSEImager.plot_grid`(*pattern_idx: Tuple[int, ...] | None = None*, *rgb_channels: None | List[Tuple] | List[List[Tuple]] = None*, *visible_indices: bool = True*, ***kwargs*) → *EBSD*

Plot a pattern with the detector grid superimposed, potentially coloring the edges of three grid tiles red, green and blue.

**Parameters**

**pattern_idx**
A tuple of integers defining the pattern to superimpose the grid on. If not given (default), the first pattern is used.

**rgb_channels**
A list of tuple indices defining three or more detector grid tiles which edges to color red, green and blue. If not given (default), no tiles' edges are colored.

**visible_indices**
Whether to show grid indices. Default is `True`.

****kwargs**
Keyword arguments passed to `matplotlib.pyplot.axhline()` and `axvline`, used by HyperSpy to draw lines.

**Returns**

**pattern**
A signal with a single pattern with the markers added.

## roi_from_grid

`VirtualBSEImager.roi_from_grid`(*index: Tuple | List[Tuple]*) → RectangularROI

Return a rectangular region of interest (ROI) on the EBSD detector from one or multiple grid tile indices as row(s) and column(s).

**Parameters**

**index**
Row and column of one or multiple grid tiles as a tuple or a list of tuples.

**Returns**

**roi**
ROI defined by the grid indices.

## 2.8 indexing

Tools for indexing of EBSD patterns by matching to a dictionary of simulated patterns.

Some of these tools are used in *dictionary_indexing()*.

### Functions

| | |
|---|---|
| *compute_refine_orientation_projection_center* | Compute the results from *refine_orientation_projection_center()* and return the CrystalMap and *EBSDDetector*. |
| *compute_refine_orientation_results*(results, ...) | Compute the results from *refine_orientation()* and return the CrystalMap. |
| *compute_refine_projection_center_results*(...) | Compute the results from *refine_projection_center()* and return the score array, *EBSDDetector* and number of function evaluations per pattern. |
| *merge_crystal_maps*(crystal_maps[, ...]) | Return a multi phase CrystalMap by merging maps of 1D or 2D navigation shape based on scores. |
| *orientation_similarity_map*(xmap[, n_best, ...]) | Compute an orientation similarity map (OSM) where the ranked list of the dictionary indices of the best matching simulated patterns in one point is compared to the corresponding lists in the nearest neighbour points [Marquardt *et al.*, 2017]. |
| *xmap_from_hough_indexing_data*(data, phase_list) | Convert Hough indexing result array from pyebsdindex to a CrystalMap. |

### 2.8.1 compute_refine_orientation_projection_center_results

kikuchipy.indexing.**compute_refine_orientation_projection_center_results**(*results: Array*, *detector:* EBSDDetector, *xmap: CrystalMap*, *master_pattern:* EBSDMasterPattern, *navigation_mask: ndarray* | *None* = *None*, *pseudo_symmetry_checked: bool* = *False*) → Tuple[CrystalMap, *EBSDDetector*]

Compute the results from *refine_orientation_projection_center()* and return the CrystalMap and *EBSDDetector*.

**Parameters**

**results**
Dask array returned from `refine_orientation_projection_center()`.

**detector**
Detector passed to `refine_orientation_projection_center()` to obtain `results`.

**xmap**
Crystal map passed to `refine_orientation_projection_center()` to obtain `results`.

**master_pattern**
Master pattern passed to `refine_orientation_projection_center()` to obtain `results`.

**navigation_mask**
Navigation mask passed to `refine_orientation_projection_center()` to obtain `results`. If not given, it is assumed that it was not given to `refine_orientation_projection_center()` either.

**pseudo_symmetry_checked**
Whether pseudo-symmetry operators were passed to `refine_orientation_projection_center()`. Default is `False`.

**Returns**

**xmap_refined**
Crystal map with refined orientations, scores, the number of function evaluations and the pseudo-symmetry index if `pseudo_symmetry_checked=True`. See the docstring of `refine_orientation_projection_center()` for details.

**new_detector**
EBSD detector with refined projection center parameters.

**See also:**

*kikuchipy.signals.EBSD.refine_orientation_projection_center*

### 2.8.2 compute_refine_orientation_results

kikuchipy.indexing.**compute_refine_orientation_results**(*results: Array*, *xmap: CrystalMap*, *master_pattern:* EBSDMasterPattern, *navigation_mask: ndarray | None = None*, *pseudo_symmetry_checked: bool = False*) → CrystalMap

Compute the results from *refine_orientation()* and return the `CrystalMap`.

**Parameters**

**results**
Dask array returned from `refine_orientation()`.

**xmap**
Crystal map passed to `refine_orientation()` to obtain `results`.

**master_pattern**
Master pattern passed to `refine_orientation()` to obtain `results`.

**navigation_mask**
Navigation mask passed to `refine_orientation()` to obtain `results`. If not given, it is assumed that it was not given to `refine_orientation()` either.

**pseudo_symmetry_checked**
Whether pseudo-symmetry operators were passed to `refine_orientation()`. Default is `False`.

**Returns**

**xmap_refined**
Crystal map with refined orientations, scores, the number of function evaluations and the pseudo-symmetry index if `pseudo_symmetry_checked=True`. See the docstring of `refine_orientation()` for details.

### 2.8.3 compute_refine_projection_center_results

kikuchipy.indexing.**compute_refine_projection_center_results**(*results: Array*, *detector:* EBSDDetector, *xmap: CrystalMap*, *navigation_mask: ndarray | None = None*) → Tuple[ndarray, *EBSDDetector*, ndarray]

Compute the results from *refine_projection_center()* and return the score array, *EBSDDetector* and number of function evaluations per pattern.

**Parameters**

**results**
Dask array returned from `refine_projection_center()`.

**detector**
Detector passed to `refine_projection_center()` to obtain `results`.

**xmap**
Crystal map passed to `refine_projection_center()` to obtain `results`.

**navigation_mask**
Navigation mask passed to `refine_projection_center()` to obtain `results`. If not given, it is assumed that it was not given to `refine_projection_center()` either.

**Returns**

**new_scores**
Score array.

**new_detector**
EBSD detector with refined projection center parameters.

**num_evals**
Number of function evaluations per pattern.

### 2.8.4 merge_crystal_maps

kikuchipy.indexing.**merge_crystal_maps**(*crystal_maps: List[CrystalMap]*, *mean_n_best: int = 1*, *greater_is_better: int | None = None*, *scores_prop: str = 'scores'*, *simulation_indices_prop: str | None = None*, *navigation_masks: List[None | ndarray] | None = None*) → CrystalMap

Return a multi phase `CrystalMap` by merging maps of 1D or 2D navigation shape based on scores.

It is required that all maps have the same number of rotations and scores (and simulation indices if applicable) per point.

**Parameters**

**crystal_maps**
A list of at least two crystal maps with simulated indices and scores among their properties. The maps must have the same shape, unless navigation masks are passed (see `navigation_masks`). Identical phases are considered as one phase in the returned map.

**mean_n_best**
Number of best metric results to take the mean of before comparing. Default is 1. If given with a negative sign and `greater_is_better` is not given, the n lowest valued metric results are chosen.

**greater_is_better**
`True` if a higher score means a better match. If not given, the sign of `mean_n_best` is used, with a positive sign meaning `True`.

**scores_prop**
Name of scores array in the crystal maps' properties. Default is `"scores"`.

**simulation_indices_prop**
Name of simulated indices array in the crystal maps' properties. If not given (default), the merged crystal map will not contain an array of merged simulation indices from the input crystal maps' properties. If a string, there must be as many simulation indices per point as there are scores.

**navigation_masks**
A list of boolean masks of shapes equal to the full 1D or 2D navigation (map) shape, where only points equal to `False` are considered when comparing scores. The number of `False` entries in a mask must be equal to the number of points in a crystal map (`size`). The order corresponds to the order in `crystal_maps`. If not given, all points are used. If all points in one or more of the maps should be used, this map's entry can be `None`.

**Returns**

**merged_xmap**
A crystal map where the rotation of the phase with the best matching score(s) is assigned to each point. The best matching scores, merge sorted, are added to its properties with a name equal to whatever passed to `scores_prop` with "merged" as a suffix. If `simulation_indices_prop` is passed, the best matching simulation indices are added in the same way as the scores.

### Notes

The initial motivation behind this function was to merge single phase maps produced by dictionary indexing.

## 2.8.5 orientation_similarity_map

kikuchipy.indexing.**orientation_similarity_map**(*xmap: CrystalMap*, *n_best: int | None = None*, *simulation_indices_prop: str = 'simulation_indices'*, *normalize: bool = False*, *from_n_best: int | None = None*, *footprint: ndarray | None = None*, *center_index: int = 2*) → ndarray

Compute an orientation similarity map (OSM) where the ranked list of the dictionary indices of the best matching simulated patterns in one point is compared to the corresponding lists in the nearest neighbour points [Marquardt *et al.*, 2017].

**Parameters**

**xmap**
A crystal map with a ranked list of the array indices of the best matching simulated patterns among its properties.

**n_best**
Number of ranked indices to compare. If not given (default), all indices are compared.

**simulation_indices_prop**
Name of simulated indices array in the crystal maps' properties. Default is `"simulation_indices"`.

**normalize**
Whether to normalize the number of equal indices to the range [0, 1], by default `False`.

**from_n_best**
Return an OSM for each n in the range `[from_n_best, n_best]`. If not given (default), the OSM for `n_best` indices is returned.

**footprint**
Boolean 2D array specifying which neighbouring points to compare lists with, by default the four nearest neighbours.

**center_index**
Flat index of central navigation point in the truthy values of footprint, by default `2`.

**Returns**

**osm**
Orientation similarity map(s). If `from_n_best` is given, the returned array has three dimensions, where `n_best` is at `osm[:, :, 0]` and `from_n_best` at `osm[:, :, -1]`.

### Notes

If the set $S_{r,c}$ is the ranked list of best matching indices for a given point $(r, c)$, then the orientation similarity index $\eta_{r,c}$ is the average value of the cardinalities (#) of the intersections with the neighbouring sets

$$\eta_{r,c} = \frac{1}{4}\left(\#(S_{r,c} \cap S_{r-1,c}) + \#(S_{r,c} \cap S_{r+1,c}) + \#(S_{r,c} \cap S_{r,c-1}) + \#(S_{r,c} \cap S_{r,c+1})\right).$$

Changed in version 0.5: Default value of `normalize` changed to `False`.

## 2.8.6 xmap_from_hough_indexing_data

kikuchipy.indexing.**xmap_from_hough_indexing_data**(*data: ndarray*, *phase_list: PhaseList*, *data_index: int = -1*, *navigation_shape: tuple | None = None*, *step_sizes: tuple | None = None*, *scan_unit: str = 'px'*) → CrystalMap

Convert Hough indexing result array from `pyebsdindex` to a `CrystalMap`.

**Parameters**

**data**
Array with the following data type field names: `"quat"`, `"phase"`, `"fit"`, `"cm"`, `"pq"` and `"nmatch"`.

**phase_list**
List of phases. If `data_index=-1`, the phase IDs in the list must match the phase IDs in `data[-1]["phase"]`. If `data_index` is another ID, it must be in the phase list.

**data_index**
Index into `data` of which to return a crystal map from. Default is `-1`, which returns the most probable (best) solutions in each map point. Other options depend on the number of phases used in indexing, and starts with `0`.

**navigation_shape**
Navigation shape of resulting map. If not given, a 1D crystal map is returned. Maximum of two dimensions.

**step_sizes**
Step sizes in each navigation direction. If not given, a step size of 1 is used in each direction.

**scan_unit**
Scan unit of map. If not given, it is not set.

**Returns**

**xmap**
Crystal map.

### Classes

| | |
|---|---|
| *NormalizedCrossCorrelationMetric*([...]) | Similarity metric implementing the normalized cross-correlation, or Pearson Correlation Coefficient [Gonzalez and Woods, 2017]. |
| *NormalizedDotProductMetric*([...]) | Similarity metric implementing the normalized dot product [Chen *et al.*, 2015]. |
| *SimilarityMetric*([n_experimental_patterns, ...]) | Abstract class implementing a similarity metric to match experimental and simulated EBSD patterns in a dictionary. |

## 2.8.7 NormalizedCrossCorrelationMetric

**class** kikuchipy.indexing.**NormalizedCrossCorrelationMetric**(*n_experimental_patterns: int | None = None*, *n_dictionary_patterns: int | None = None*, *navigation_mask: ndarray | None = None*, *signal_mask: ndarray | None = None*, *dtype: str | dtype | type = 'float32'*, *rechunk: bool = False*)

Bases: *SimilarityMetric*

Similarity metric implementing the normalized cross-correlation, or Pearson Correlation Coefficient [Gonzalez and Woods, 2017].

The metric is defined as

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}},$$

where experimental patterns $x$ and simulated patterns $y$ are centered by subtracting out the mean of each pattern, and the sum of cross-products of the centered patterns is accumulated. The denominator adjusts the scales of the patterns to have equal units.

Equivalent results are obtained with `dask.array.tensordot()` with `axes=([2, 3], [1, 2]))` for 4D and 3D experimental and simulated data sets, respectively.

See *SimilarityMetric* for the description of the initialization parameters and the list of attributes.

### Attributes

### Methods

| | |
|---|---|
| *NormalizedCrossCorrelationMetric.match*(...) | Match all experimental patterns to all dictionary patterns and return their similarities. |
| *NormalizedCrossCorrelationMetric.prepare_dictionary*(...) | Prepare dictionary patterns before matching to experimental patterns in *match()*. |
| *NormalizedCrossCorrelationMetric.prepare_experimental*(...) | Prepare experimental patterns before matching to dictionary patterns in *match()*. |

### match

NormalizedCrossCorrelationMetric.**match**(*experimental:* *ndarray* | *Array*, *dictionary:* *ndarray* | *Array*) → Array

Match all experimental patterns to all dictionary patterns and return their similarities.

**Parameters**

**experimental**
Experimental patterns.

**dictionary**
Dictionary patterns.

**Returns**

**scores**
Normalized cross-correlation scores.

### prepare_dictionary

NormalizedCrossCorrelationMetric.**prepare_dictionary**(*patterns:* *ndarray* | *Array*) → ndarray | Array

Prepare dictionary patterns before matching to experimental patterns in *match()*.

**Patterns are prepared by:**

1. Setting the data type to `dtype`.
2. Applying a signal mask if `signal_mask` is set.
3. Normalizing to a mean of 0 and a standard deviation of 1.

**Parameters**

**patterns**
Dictionary patterns.

**Returns**

**prepared_patterns**
Prepared dictionary patterns.

#### prepare_experimental

NormalizedCrossCorrelationMetric.**prepare_experimental**(*patterns: ndarray | Array*) → ndarray | Array

Prepare experimental patterns before matching to dictionary patterns in *match()*.

**Patterns are prepared by:**

1. Setting the data type to `dtype`.
2. Excluding the experimental patterns where `navigation_mask` is `False` if the mask is set.
3. Reshaping to shape `(n_experimental_patterns, -1)`
4. Applying a signal mask if `signal_mask` is set.
5. Rechunking if `rechunk` is `True`.
6. Normalizing to a mean of 0 and a standard deviation of 1.

**Parameters**

**patterns**
Experimental patterns.

**Returns**

**prepared_patterns**
Prepared experimental patterns.

### 2.8.8 NormalizedDotProductMetric

**class** kikuchipy.indexing.**NormalizedDotProductMetric**(*n_experimental_patterns: int | None = None*, *n_dictionary_patterns: int | None = None*, *navigation_mask: ndarray | None = None*, *signal_mask: ndarray | None = None*, *dtype: str | dtype | type = 'float32'*, *rechunk: bool = False*)

Bases: *SimilarityMetric*

Similarity metric implementing the normalized dot product [Chen *et al.*, 2015].

The metric is defined as

$$\rho = \frac{\langle \mathbf{X}, \mathbf{Y} \rangle}{||\mathbf{X}|| \cdot ||\mathbf{Y}||},$$

where $\langle \mathbf{X}, \mathbf{Y} \rangle$ is the dot (inner) product of the pattern vectors $\mathbf{X}$ and $\mathbf{Y}$.

See *SimilarityMetric* for the description of the initialization parameters and the list of attributes.

### Attributes

### Methods

| | |
|---|---|
| *NormalizedDotProductMetric.match*(...) | Match all experimental patterns to all dictionary patterns and return their similarities. |
| *NormalizedDotProductMetric.prepare_dictionary*(...) | Prepare dictionary patterns before matching to experimental patterns in *match()*. |
| *NormalizedDotProductMetric.prepare_experimental*(...) | Prepare experimental patterns before matching to dictionary patterns in *match()*. |

### match

NormalizedDotProductMetric.**match**(*experimental: ndarray | Array*, *dictionary: ndarray | Array*) → Array

Match all experimental patterns to all dictionary patterns and return their similarities.

**Parameters**

**experimental**
Experimental patterns.

**dictionary**
Dictionary patterns.

**Returns**

**dot_products**
Normalized dot products.

### prepare_dictionary

NormalizedDotProductMetric.**prepare_dictionary**(*patterns: ndarray | Array*) → ndarray | Array

Prepare dictionary patterns before matching to experimental patterns in *match()*.

**Patterns are prepared by:**

1. Setting the data type to `dtype`.
2. Applying a signal mask if `signal_mask` is set.
3. Normalizing to a mean of 0.

**Parameters**

**patterns**
Dictionary patterns.

**Returns**

**prepared_patterns**
Prepared dictionary patterns.

#### prepare_experimental

NormalizedDotProductMetric.**prepare_experimental**(*patterns: ndarray | Array*) → ndarray | Array

Prepare experimental patterns before matching to dictionary patterns in *match()*.

**Patterns are prepared by:**

1. Setting the data type to `dtype`.
2. Excluding the experimental patterns where `navigation_mask` is `False` if the mask is set.
3. Reshaping to shape `(n_experimental_patterns, -1)`.
4. Applying a signal mask if `signal_mask` is set.
5. Rechunking if `rechunk` is `True`.
6. Normalizing to a mean of 0.

**Parameters**

**patterns**
Experimental patterns.

**Returns**

**prepared_patterns**
Prepared experimental patterns.

### 2.8.9 SimilarityMetric

class kikuchipy.indexing.**SimilarityMetric**(*n_experimental_patterns: int | None = None*, *n_dictionary_patterns: int | None = None*, *navigation_mask: ndarray | None = None*, *signal_mask: ndarray | None = None*, *dtype: str | dtype | type = 'float32'*, *rechunk: bool = False*)

Bases: ABC

Abstract class implementing a similarity metric to match experimental and simulated EBSD patterns in a dictionary.

For use in *dictionary_indexing()* or directly on pattern arrays if a `__call__()` method is implemented. Note that *dictionary_indexing()* will always reshape the dictionary pattern array to 2D (1 navigation dimension, 1 signal dimension) before calling *prepare_dictionary()* and *match()*.

Take a look at the implementation of *NormalizedCrossCorrelationMetric* for how to write a concrete custom metric.

When writing a custom similarity metric class, the methods listed as *abstract* below must be implemented. Any number of custom parameters can be passed. Also listed are the attributes available to the methods if set properly during initialization or after.

**Parameters**

**n_experimental_patterns**
Number of experimental patterns. If not given, this is set to `None` and must be set later. Must be at least 1.

**n_dictionary_patterns**
Number of dictionary patterns. If not given, this is set to `None` and must be set later. Must be at least 1.

**navigation_mask**
A boolean mask equal to the experimental patterns' navigation (map) shape, where only patterns equal to `False` are matched. If not given, all patterns are used.

**signal_mask**
A boolean mask equal to the experimental patterns' detector shape `(n rows, n columns)`, where only pixels equal to `False` are matched. If not given, all pixels are used.

**dtype**
Which data type to cast the patterns to before matching to.

**rechunk**
Whether to allow rechunking of arrays before matching. Default is `False`.

### Attributes

| | |
|---|---|
| *SimilarityMetric.allowed_dtypes* | Return the list of allowed array data types used during matching. |
| *SimilarityMetric.dtype* | Return or set which data type to cast the patterns to before matching. |
| *SimilarityMetric.n_dictionary_patterns* | Return or set the number of dictionary patterns to match. |
| *SimilarityMetric.n_experimental_patterns* | Return or set the number of experimental patterns to match. |
| *SimilarityMetric.navigation_mask* | Return or set the boolean mask of patterns to match, equal to the navigation (map) shape. |
| *SimilarityMetric.rechunk* | Return or set whether to allow rechunking of arrays before matching. |
| *SimilarityMetric.sign* | Return the sign signifying whether a greater match is better, either +1 (greater is better) or -1 (lower is better). |
| *SimilarityMetric.signal_mask* | Return or set the boolean mask equal to the experimental patterns' detector shape `(s rows, s columns)`. |

### allowed_dtypes

**property** SimilarityMetric.**allowed_dtypes: List[type]**

Return the list of allowed array data types used during matching.

### dtype

**property** SimilarityMetric.**dtype: dtype**

Return or set which data type to cast the patterns to before matching.

**Parameters**

**value**
Data type listed in *allowed_dtypes*.

### n_dictionary_patterns

property SimilarityMetric.**n_dictionary_patterns: int**

Return or set the number of dictionary patterns to match.

This information might be necessary when reshaping the dictionary array in *prepare_dictionary()*.

**Parameters**

**value**
Number of dictionary patterns to match.

### n_experimental_patterns

property SimilarityMetric.**n_experimental_patterns: int**

Return or set the number of experimental patterns to match.

This information might be necessary when reshaping the experimental array in *prepare_experimental()*.

**Parameters**

**value**
Number of experimental patterns to match.

### navigation_mask

property SimilarityMetric.**navigation_mask: ndarray**

Return or set the boolean mask of patterns to match, equal to the navigation (map) shape.

**Parameters**

**value**
Navigation mask where points set to `False` are matched.

### rechunk

property SimilarityMetric.**rechunk: bool**

Return or set whether to allow rechunking of arrays before matching.

**Parameters**

**value**
Whether to allow rechunking of arrays before matching.

### sign

property SimilarityMetric.**sign: int**

Return the sign signifying whether a greater match is better, either +1 (greater is better) or -1 (lower is better).

### signal_mask

**property** SimilarityMetric.**signal_mask: ndarray**

Return or set the boolean mask equal to the experimental patterns' detector shape `(s rows, s columns)`.

**Parameters**

**value**
Signal mask where pixels set to `False` are matched.

### Methods

| | |
|---|---|
| *SimilarityMetric.match*(*args, **kwargs) | Match all experimental patterns to all dictionary patterns and return their similarities. |
| *SimilarityMetric.prepare_dictionary*(*args, ...) | Prepare dictionary patterns before matching to experimental patterns in *match()*. |
| *SimilarityMetric.prepare_experimental*(*args, ...) | Prepare experimental patterns before matching to dictionary patterns in *match()*. |
| *SimilarityMetric.raise_error_if_invalid*() | Raise a ValueError if *dtype* is not among *allowed_dtypes* and the latter is not an empty list. |

### match

**abstract** SimilarityMetric.**match**(**args*, ***kwargs*)

Match all experimental patterns to all dictionary patterns and return their similarities.

### prepare_dictionary

**abstract** SimilarityMetric.**prepare_dictionary**(**args*, ***kwargs*)

Prepare dictionary patterns before matching to experimental patterns in *match()*.

### prepare_experimental

**abstract** SimilarityMetric.**prepare_experimental**(**args*, ***kwargs*)

Prepare experimental patterns before matching to dictionary patterns in *match()*.

### raise_error_if_invalid

SimilarityMetric.**raise_error_if_invalid**()

Raise a ValueError if *dtype* is not among *allowed_dtypes* and the latter is not an empty list.

## 2.9 io

Read and write signals from and to file.

### Modules

| | |
|---|---|
| *plugins* | Input/output plugins. |

### 2.9.1 plugins

Input/output plugins.

### Modules

| | |
|---|---|
| *bruker_h5ebsd* | Reader of EBSD data from a Bruker Nano h5ebsd file. |
| *ebsd_directory* | Reader of EBSD patterns from a dictionary of images. |
| *edax_binary* | Reader of EBSD data from EDAX TSL UP1/2 files. |
| *edax_h5ebsd* | Reader of EBSD data from an EDAX TSL h5ebsd file. |
| *emsoft_ebsd* | Reader of simulated EBSD patterns from an EMsoft HDF5 file. |
| *emsoft_ebsd_master_pattern* | Reader of simulated EBSD master patterns from an EMsoft HDF5 file. |
| *emsoft_ecp_master_pattern* | Reader of simulated ECP master patterns from an EMsoft HDF5 file. |
| *emsoft_tkd_master_pattern* | Reader of simulated TKD master patterns from an EMsoft HDF5 file. |
| *kikuchipy_h5ebsd* | Reader and writer of EBSD data from a kikuchipy h5ebsd file. |
| *nordif* | Reader and writer of EBSD patterns from a NORDIF binary file. |
| *nordif_calibration_patterns* | Reader of EBSD calibration patterns from NORDIF files. |
| *oxford_binary* | Reader of uncompressed EBSD patterns from a Oxford Instruments binary .ebsp file. |
| *oxford_h5ebsd* | Reader of EBSD data from an Oxford Instruments h5ebsd (H5OINA) file. |

#### bruker_h5ebsd

Reader of EBSD data from a Bruker Nano h5ebsd file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, scan_group_names, lazy]) | Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from a Bruker h5ebsd file [Jackson *et al.*, 2014]. |

### file_reader

kikuchipy.io.plugins.bruker_h5ebsd.**file_reader**(*filename: str | Path*, *scan_group_names: None | str | List[str] = None*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from a Bruker h5ebsd file [Jackson *et al.*, 2014].

Not ment to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF5 file.

**scan_group_names**
Name or a list of names of HDF5 top group(s) containing the scan(s) to return. If not given (default), the first scan in the file is returned.

**lazy**
Open the data lazily without actually reading the data from disk until required. Allows opening arbitrary sized datasets. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**scan_dict_list**
List of one or more dictionaries with the keys `"axes"`, `"data"`, `"metadata"`, `"original_metadata"`, `"detector"`, `"static_background"`, and `"xmap"`. This dictionary can be passed as keyword arguments to create an *EBSD* signal.

### ebsd_directory

Reader of EBSD patterns from a dictionary of images.

### Functions

| | |
|---|---|
| *file_reader*(filename[, xy_pattern, ...]) | Read all images in a directory, assuming they are electron backscatter diffraction (EBSD) patterns of equal shape and data type. |

### file_reader

kikuchipy.io.plugins.ebsd_directory.**file_reader**(*filename: str | Path*, *xy_pattern: str | None = None*, *show_progressbar: bool | None = None*, *lazy: bool = False*) → List[Dict]

Read all images in a directory, assuming they are electron backscatter diffraction (EBSD) patterns of equal shape and data type.

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Name of directory with patterns.

**xy_pattern**
Regular expression to extract map coordinates from the filenames. If not given, two regular expressions will be tried: assuming (x, y) = (5, 10), "_x5y10.tif" or "-5-10.bmp". Valid `xy_pattern` equal to these are `r"_x(\d+)y(\d+).tif"` and `r"-(\d+)-(\d+).bmp"`, respectively. If none of these expressions match the first file's name in the directory, a warning is printed and the returned signal will have only one navigation dimension.

**show_progressbar**
Whether to show a progressbar when reading the signal into memory when `lazy=False`.

**lazy**
Read the patterns lazily without actually reading them from disk until required. Default is `False`.

**Returns**

**scan**
Data, axes, metadata and original metadata.

**Warns**

`UserWarning`
If navigation coordinates can not be read from the filenames.

`UserWarning`
If there are more detected patterns in the directory than the navigation shape determined from the filenames suggest.

#### Notes

Adapted from https://blog.dask.org/2019/06/20/load-image-data.

### edax_binary

Reader of EBSD data from EDAX TSL UP1/2 files.

The reader is adapted from the EDAX UP1/2 reader in PyEBSDIndex.

### Functions

| | |
|---|---|
| *file_reader*(filename[, nav_shape, lazy]) | Read EBSD patterns from an EDAX binary UP1/2 file. |

### file_reader

kikuchipy.io.plugins.edax_binary.**file_reader**(*filename: str | Path*, *nav_shape: Tuple[int, int] | None = None*, *lazy: bool = False*) → List[dict]

Read EBSD patterns from an EDAX binary UP1/2 file.

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
File path to UP1/2 file with "up1" or "up2" extension.

**nav_shape**
Navigation shape, as (n map rows, n map columns), of the returned *EBSD* signal, matching the number of patterns in the file. If not given, this shape will be attempted to be determined from the file. If it could not be, the returned signal will have only one navigation dimension. If patterns were acquired in an hexagonal grid, the returned signal will have only one navigation dimension irrespective of this parameter's value.

**lazy**
Read the data lazily without actually reading the data from disk until required. Default is False.

**Returns**

**scan**
Data, axes, metadata and original metadata.

**Raises**

**ValueError**
If file version is 2, since only version 1 or >= 3 is supported.

**ValueError**
If nav_shape does not match the number of patterns in the file.

**Warns**

**UserWarning**
If patterns were acquired in an hexagonal grid, since then the returned signal will have only one navigation dimension, even though nav_shape is given.

### Notes

Reader adapted from the EDAX UP1/2 reader in PyEBSDIndex.

## edax_h5ebsd

Reader of EBSD data from an EDAX TSL h5ebsd file.

## Functions

| | |
|---|---|
| *file_reader*(filename[, scan_group_names, lazy]) | Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from an EDAX h5ebsd file [Jackson *et al.*, 2014]. |

## file_reader

kikuchipy.io.plugins.edax_h5ebsd.**file_reader**(*filename: str | Path*, *scan_group_names: None | str | List[str] = None*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from an EDAX h5ebsd file [Jackson *et al.*, 2014].

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF5 file.

**scan_group_names**
Name or a list of names of HDF5 top group(s) containing the scan(s) to return. If not given (default), the first scan in the file is returned.

**lazy**
Open the data lazily without actually reading the data from disk until required. Allows opening arbitrary sized datasets. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**scan_dict_list**
List of one or more dictionaries with the keys `"axes"`, `"data"`, `"metadata"`, `"original_metadata"`, `"detector"`, and `"xmap"`. This dictionary can be passed as keyword arguments to create an *EBSD* signal.

### emsoft_ebsd

Reader of simulated EBSD patterns from an EMsoft HDF5 file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, scan_size, lazy]) | Read dynamically simulated electron backscatter diffraction patterns from EMsoft's format produced by their EMEBSD.f90 program. |

### file_reader

kikuchipy.io.plugins.emsoft_ebsd.**file_reader**(*filename: str | Path*, *scan_size: None | int | Tuple[int, ...] = None*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read dynamically simulated electron backscatter diffraction patterns from EMsoft's format produced by their EMEBSD.f90 program.

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF file.

**scan_size**
Scan size in number of patterns in width and height.

**lazy**
Open the data lazily without actually reading the data from disk until requested. Allows opening datasets larger than available memory. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**signal_dict_list**
Data, axes, metadata and original metadata.

### emsoft_ebsd_master_pattern

Reader of simulated EBSD master patterns from an EMsoft HDF5 file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, energy, projection, ...]) | Read simulated electron backscatter diffraction master patterns from EMsoft's HDF5 file format [Callahan and De Graef, 2013]. |

### file_reader

kikuchipy.io.plugins.emsoft_ebsd_master_pattern.**file_reader**(*filename: str | Path*, *energy: range | None = None*, *projection: str = 'stereographic'*, *hemisphere: str = 'upper'*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read simulated electron backscatter diffraction master patterns from EMsoft's HDF5 file format [Callahan and De Graef, 2013].

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF file.

**energy**
Desired beam energy or energy range. If not given (default), all available energies are read.

**projection**
Projection(s) to read. Options are `"stereographic"` (default) or `"lambert"`.

**hemisphere**
Projection hemisphere(s) to read. Options are `"upper"` (default), `"lower"` or `"both"`. If `"both"`, these will be stacked in the vertical navigation axis.

**lazy**
Open the data lazily without actually reading the data from disk until requested. Allows opening datasets larger than available memory. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**signal_dict_list**
Data, axes, metadata and original metadata.

### emsoft_ecp_master_pattern

Reader of simulated ECP master patterns from an EMsoft HDF5 file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, energy, projection, ...]) | Read simulated electron channeling pattern (ECP) master patterns from EMsoft's HDF5 file format [Callahan and De Graef, 2013]. |

### file_reader

kikuchipy.io.plugins.emsoft_ecp_master_pattern.**file_reader**(*filename: str | Path*, *energy: range | None = None*, *projection: str = 'stereographic'*, *hemisphere: str = 'upper'*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read simulated electron channeling pattern (ECP) master patterns from EMsoft's HDF5 file format [Callahan and De Graef, 2013].

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF file.

**energy**
Desired beam energy or energy range. If not given (default), all available energies are read.

**projection**
Projection(s) to read. Options are `"stereographic"` (default) or `"lambert"`.

**hemisphere**
Projection hemisphere(s) to read. Options are `"upper"` (default), `"lower"` or `"both"`. If `"both"`, these will be stacked in the vertical navigation axis.

**lazy**
Open the data lazily without actually reading the data from disk until requested. Allows opening datasets larger than available memory. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**signal_dict_list**
Data, axes, metadata and original metadata.

### emsoft_tkd_master_pattern

Reader of simulated TKD master patterns from an EMsoft HDF5 file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, energy, projection, ...]) | Read simulated transmission kikuchi diffraction master patterns from EMsoft's HDF5 file format [Callahan and De Graef, 2013]. |

### file_reader

kikuchipy.io.plugins.emsoft_tkd_master_pattern.**file_reader**(*filename: str | Path*, *energy: range | None = None*, *projection: str = 'stereographic'*, *hemisphere: str = 'upper'*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read simulated transmission kikuchi diffraction master patterns from EMsoft's HDF5 file format [Callahan and De Graef, 2013].

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF file.

**energy**
Desired beam energy or energy range. If not given (default), all available energies are read.

**projection**
Projection(s) to read. Options are `"stereographic"` (default) or `"lambert"`.

**hemisphere**
Projection hemisphere(s) to read. Options are `"upper"` (default), `"lower"` or `"both"`. If `"both"`, these will be stacked in the vertical navigation axis.

**lazy**
Open the data lazily without actually reading the data from disk until requested. Allows opening datasets larger than available memory. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**signal_dict_list**
Data, axes, metadata and original metadata.

### kikuchipy_h5ebsd

Reader and writer of EBSD data from a kikuchipy h5ebsd file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, scan_group_names, lazy]) | Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from a kikuchipy h5ebsd file [Jackson *et al.*, 2014]. |
| *file_writer*(filename, signal[, add_scan, ...]) | Write an *EBSD* or *LazyEBSD* signal to an existing but not open or new h5ebsd file. |

### file_reader

kikuchipy.io.plugins.kikuchipy_h5ebsd.**file_reader**(*filename: str | Path*, *scan_group_names: None | str | List[str] = None*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from a kikuchipy h5ebsd file [Jackson *et al.*, 2014].

Not ment to be used directly; use *load()*.

The file is closed after reading if `lazy=False`.

**Parameters**

**filename**
Full file path of the HDF5 file.

**scan_group_names**
Name or a list of names of HDF5 top group(s) containing the scan(s) to return. If not given (default), the first scan in the file is returned.

**lazy**
Open the data lazily without actually reading the data from disk until required. Allows opening arbitrary sized datasets. Default is `False`.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**scan_dict_list**
List of one or more dictionaries with the keys `"axes"`, `"data"`, `"metadata"`, `"original_metadata"`, `"detector"`, `"static_background"`, and `"xmap"`. This dictionary can be passed as keyword arguments to create an *EBSD* signal.

### file_writer

kikuchipy.io.plugins.kikuchipy_h5ebsd.**file_writer**(*filename: str*, *signal*, *add_scan: bool | None = None*, *scan_number: int = 1*, ***kwargs*)

Write an *EBSD* or *LazyEBSD* signal to an existing but not open or new h5ebsd file.

Not meant to be used directly; use *save()*.

The file is closed after writing.

**Parameters**

**filename**
Full path of HDF5 file.

**signal**
[*kikuchipy.signals.EBSD* or *kikuchipy.signals.LazyEBSD*] Signal instance.

**add_scan**
Add signal to an existing, but not open, h5ebsd file. If it does not exist it is created and the signal is written to it.

**scan_number**
Scan number in name of HDF dataset when writing to an existing, but not open, h5ebsd file.

****kwargs**
Keyword arguments passed to `h5py.Group.require_dataset()`.

### nordif

Reader and writer of EBSD patterns from a NORDIF binary file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, mmap_mode, ...]) | Read electron backscatter patterns from a NORDIF data file. |
| *file_writer*(filename, signal) | Write an *EBSD* or *LazyEBSD* instance to a NORDIF binary file. |

### file_reader

kikuchipy.io.plugins.nordif.**file_reader**(*filename: str | Path*, *mmap_mode: str | None = None*, *scan_size: None | int | Tuple[int, ...] = None*, *pattern_size: Tuple[int, ...] | None = None*, *setting_file: str | None = None*, *lazy: bool = False*) → List[Dict]

Read electron backscatter patterns from a NORDIF data file.

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
File path to NORDIF data file.

**mmap_mode**
Memory map mode. If not given, `"r"` is used unless `lazy=True`, in which case `"c"` is used.

**scan_size**
Scan size in number of patterns in width and height.

**pattern_size**
Pattern size in detector pixels in width and height.

**setting_file**
File path to NORDIF setting file (default is *Setting.txt* in same directory as `filename`).

**lazy**
Open the data lazily without actually reading the data from disk until required. Allows opening arbitrary sized datasets. Default is `False`.

**Returns**

**scan**
Data, axes, metadata and original metadata.

### file_writer

kikuchipy.io.plugins.nordif.**file_writer**(*filename: str*, *signal:* EBSD | LazyEBSD)

Write an *EBSD* or *LazyEBSD* instance to a NORDIF binary file.

**Parameters**

**filename**
Full path of HDF file.

**signal**
Signal instance.

## nordif_calibration_patterns

Reader of EBSD calibration patterns from NORDIF files.

## Functions

| | |
|---|---|
| *file_reader*(filename[, lazy]) | Reader electron backscatter patterns from .bmp files stored in a NORDIF project directory, their filenames listed in a text file. |

### file_reader

kikuchipy.io.plugins.nordif_calibration_patterns.**file_reader**(*filename: str | Path*, *lazy: bool = False*) → List[dict]

Reader electron backscatter patterns from .bmp files stored in a NORDIF project directory, their filenames listed in a text file.

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
File path to the NORDIF settings text file.

**lazy**
This parameter is not used in this reader.

**Returns**

**scan**
Data, axes, metadata and original metadata.

### oxford_binary

Reader of uncompressed EBSD patterns from a Oxford Instruments binary .ebsp file.

Information about the file format was generously provided by Oxford Instruments.

### Functions

| | |
|---|---|
| *file_reader*(filename[, lazy]) | Read EBSD patterns from an Oxford Instruments' binary .ebsp file. |

### file_reader

kikuchipy.io.plugins.oxford_binary.**file_reader**(*filename: str | Path*, *lazy: bool = False*) → List[dict]

Read EBSD patterns from an Oxford Instruments' binary .ebsp file.

Only uncompressed patterns can be read. If only non-indexed patterns are stored in the file, the navigation shape will be 1D.

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
File path to .ebsp file.

**lazy**
Read the data lazily without actually reading the data from disk until required. Default is `False`.

**Returns**

**scan**
Data, axes, metadata and original metadata.

#### Notes

Information about the .ebsp file format was generously provided by Oxford Instruments.

### oxford_h5ebsd

Reader of EBSD data from an Oxford Instruments h5ebsd (H5OINA) file.

### Functions

| | |
|---|---|
| *file_reader*(filename[, scan_group_names, lazy]) | Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from an Oxford Instruments h5ebsd (H5OINA) file [Jackson *et al.*, 2014]. |

### file_reader

kikuchipy.io.plugins.oxford_h5ebsd.**file_reader**(*filename: str | Path*, *scan_group_names: None | str | List[str] = None*, *lazy: bool = False*, ***kwargs*) → List[dict]

Read electron backscatter diffraction patterns, a crystal map, and an EBSD detector from an Oxford Instruments h5ebsd (H5OINA) file [Jackson *et al.*, 2014].

Not meant to be used directly; use *load()*.

**Parameters**

**filename**
Full file path of the HDF5 file.

**scan_group_names**
Name or a list of names of HDF5 top group(s) containing the scan(s) to return. If not given (default), the first scan in the file is returned.

**lazy**
Open the data lazily without actually reading the data from disk until required. Allows opening arbitrary sized datasets. Default is False.

****kwargs**
Keyword arguments passed to `h5py.File`.

**Returns**

**scan_dict_list**
List of one or more dictionaries with the keys "axes", "data", "metadata", "original_metadata", "detector", and "xmap". This dictionary can be passed as keyword arguments to create an *EBSD* signal.

## 2.10 pattern

Single pattern processing (used by signals).

### Functions

| | |
|---|---|
| *fft*(pattern[, apodization_window, shift, ...]) | Compute the discrete Fast Fourier Transform (FFT) of an EBSD pattern. |
| *fft_filter*(pattern, transfer_function[, ...]) | Filter an EBSD patterns in the frequency domain. |
| *fft_frequency_vectors*(shape) | Get the frequency vectors in a Fourier Transform spectrum. |
| *fft_spectrum*(fft_pattern) | Compute the FFT spectrum of a Fourier transformed EBSD pattern. |
| *get_dynamic_background*(pattern[, ...]) | Get the dynamic background in an EBSD pattern. |
| *get_image_quality*(pattern[, normalize, ...]) | Return the image quality of an EBSD pattern. |
| *ifft*(fft_pattern[, shift, real_fft_only]) | Compute the inverse Fast Fourier Transform (IFFT) of an FFT of an EBSD pattern. |
| *normalize_intensity*(pattern[, num_std, ...]) | Normalize image intensities to a mean of zero and a given standard deviation. |
| *remove_dynamic_background*(pattern[, ...]) | Remove the dynamic background in an EBSD pattern. |
| *rescale_intensity*(pattern[, in_range, ...]) | Rescale intensities in an EBSD pattern. |

## 2.10.1 fft

kikuchipy.pattern.**fft**(*pattern: ndarray*, *apodization_window: None | ndarray | Window = None*, *shift: bool = False*, *real_fft_only: bool = False*, ***kwargs*) → ndarray

Compute the discrete Fast Fourier Transform (FFT) of an EBSD pattern.

Very light wrapper around routines in `scipy.fft`. The routines are wrapped instead of used directly to accommodate easy setting of `shift` and `real_fft_only`.

**Parameters**

**pattern**
EBSD pattern.

**apodization_window**
An apodization window to apply before the FFT in order to suppress streaks.

**shift**
Whether to shift the zero-frequency component to the centre of the spectrum (default is `False`).

**real_fft_only**
If `True`, the discrete FFT is computed for real input using `scipy.fft.rfft2()`. If `False` (default), it is computed using `scipy.fft.fft2()`.

****kwargs**
Keyword arguments pass to `scipy.fft.fft2()` or `scipy.fft.rfft2()`.

**Returns**

**out**
The result of the 2D FFT.

### 2.10.2 fft_filter

kikuchipy.pattern.**fft_filter**(*pattern: ndarray*, *transfer_function: ndarray | Window*, *apodization_window: None | ndarray | Window = None*, *shift: bool = False*) → ndarray

Filter an EBSD patterns in the frequency domain.

**Parameters**

**pattern**
EBSD pattern.

**transfer_function**
Filter transfer function in the frequency domain.

**apodization_window**
An apodization window to apply before the FFT in order to suppress streaks.

**shift**
Whether to shift the zero-frequency component to the centre of the spectrum. Default is `False`.

**Returns**

**filtered_pattern**
Filtered EBSD pattern.

### 2.10.3 fft_frequency_vectors

kikuchipy.pattern.**fft_frequency_vectors**(*shape: Tuple[int, int]*) → ndarray

Get the frequency vectors in a Fourier Transform spectrum.

**Parameters**

**shape**
Fourier transform shape.

**Returns**

**frequency_vectors**
Frequency vectors.

### 2.10.4 fft_spectrum

kikuchipy.pattern.**fft_spectrum**(*fft_pattern: ndarray*) → ndarray

Compute the FFT spectrum of a Fourier transformed EBSD pattern.

**Parameters**

**fft_pattern**
Fourier transformed EBSD pattern.

**Returns**

**spectrum**
2D FFT spectrum of the EBSD pattern.

### 2.10.5 get_dynamic_background

kikuchipy.pattern.**get_dynamic_background**(*pattern: ndarray*, *filter_domain: str = 'frequency'*, *std: None | int | float = None*, *truncate: int | float = 4.0*) → ndarray

Get the dynamic background in an EBSD pattern.

The background is obtained either in the frequency domain, by a low pass Fast Fourier Transform (FFT) Gaussian filter, or in the spatial domain by a Gaussian filter.

Data type is preserved.

**Parameters**

**pattern**
EBSD pattern.

**filter_domain**
Whether to obtain the dynamic background by applying a Gaussian convolution filter in the `"frequency"` (default) or `"spatial"` domain.

**std**
Standard deviation of the Gaussian window. If not given, a deviation of pattern width/8 is chosen.

**truncate**
Truncate the Gaussian window at this many standard deviations. Default is `4.0`.

**Returns**

**dynamic_bg**
The dynamic background.

### 2.10.6 get_image_quality

kikuchipy.pattern.**get_image_quality**(*pattern: ndarray*, *normalize: bool = True*, *frequency_vectors: ndarray | None = None*, *inertia_max: None | int | float = None*) → float

Return the image quality of an EBSD pattern.

The image quality is calculated based on the procedure defined by Krieger Lassen [Lassen, 1994].

**Parameters**

**pattern**
EBSD pattern.

**normalize**
Whether to normalize the pattern to a mean of zero and standard deviation of 1 before calculating the image quality (default is `True`).

**frequency_vectors**
Integer 2D array assigning each FFT spectrum frequency component a weight. If not given, these are calculated from *fft_frequency_vectors()*. This only depends on the pattern shape.

**inertia_max**
Maximum possible inertia of the FFT power spectrum of the image. If not given, this is calculated from the `frequency_vectors`, which in this case *must* be passed. This only depends on the pattern shape.

**Returns**

**image_quality**
Image quality of the pattern.

### 2.10.7 ifft

kikuchipy.pattern.**ifft**(*fft_pattern: ndarray*, *shift: bool = False*, *real_fft_only: bool = False*, ***kwargs*) → ndarray

Compute the inverse Fast Fourier Transform (IFFT) of an FFT of an EBSD pattern.

Very light wrapper around routines in `scipy.fft`. The routines are wrapped instead of used directly to accommodate easy setting of `shift` and `real_fft_only`.

**Parameters**

**fft_pattern**
FFT of EBSD pattern.

**shift**
Whether to shift the zero-frequency component back to the corners of the spectrum (default is `False`).

**real_fft_only**
If `True`, the discrete IFFT is computed for real input using `scipy.fft.irfft2()`. If `False` (default), it is computed using `scipy.fft.ifft2()`.

****kwargs**
Keyword arguments pass to `scipy.fft.ifft()`.

**Returns**

**pattern**
Real part of the IFFT of the EBSD pattern.

### 2.10.8 normalize_intensity

kikuchipy.pattern.**normalize_intensity**(*pattern: ndarray*, *num_std: int = 1*, *divide_by_square_root: bool = False*, *dtype_out: type | None = None*) → ndarray

Normalize image intensities to a mean of zero and a given standard deviation.

Data type is preserved.

**Parameters**

**pattern**
EBSD pattern.

**num_std**
Number of standard deviations of the output intensities (default is 1).

**divide_by_square_root**
Whether to divide output intensities by the square root of the image size (default is `False`).

**dtype_out**
Data type of the normalized pattern. If not given, it is set to the same data type as the input pattern.

**Returns**

**normalized_pattern**
Normalized pattern.

### Notes

Data type should always be changed to floating point, e.g. `float32` with `numpy.ndarray.astype()`, before normalizing the intensities.

## 2.10.9 remove_dynamic_background

kikuchipy.pattern.**remove_dynamic_background**(*pattern: ndarray*, *operation: str = 'subtract'*, *filter_domain: str = 'frequency'*, *std: None | int | float = None*, *truncate: int | float = 4.0*, *dtype_out: str | dtype | type | Tuple[int, int] | Tuple[float, float] | None = None*) → ndarray

Remove the dynamic background in an EBSD pattern.

The removal is performed by subtracting or dividing by a Gaussian blurred version of the pattern. The blurred version is obtained either in the frequency domain, by a low pass Fast Fourier Transform (FFT) Gaussian filter, or in the spatial domain by a Gaussian filter. Returned pattern intensities are rescaled to fill the input data type range.

**Parameters**

**pattern**
EBSD pattern.

**operation**
Whether to `"subtract"` (default) or `"divide"` by the dynamic background pattern.

**filter_domain**
Whether to obtain the dynamic background by applying a Gaussian convolution filter in the `"frequency"` (default) or `"spatial"` domain.

**std**
Standard deviation of the Gaussian window. If not given, it is set to width/8.

**truncate**
Truncate the Gaussian window at this many standard deviations. Default is `4.0`.

**dtype_out**
Data type of corrected pattern. If not given, it is set to input patterns' data type.

**Returns**

**corrected_pattern**
Pattern with the dynamic background removed.

**See also:**

*kikuchipy.signals.EBSD.remove_dynamic_background*
*kikuchipy.pattern.remove_dynamic_background*

### 2.10.10 rescale_intensity

kikuchipy.pattern.**rescale_intensity**(*pattern: ndarray*, *in_range: Tuple[int | float, ...] | None = None*, *out_range: Tuple[int | float, ...] | None = None*, *dtype_out: str | dtype | type | None = None*, *percentiles: None | Tuple[int, int] | Tuple[float, float] = None*) → ndarray

Rescale intensities in an EBSD pattern.

Pattern max./min. intensity is determined from *out_range* or the data type range of `numpy.dtype` passed to *dtype_out*.

This method is based on `skimage.exposure.rescale_intensity()`.

**Parameters**

**pattern**
EBSD pattern.

**in_range**
Min./max. intensity values of the input pattern. If not given, it is set to the pattern's min./max intensity.

**out_range**
Min./max. intensity values of the rescaled pattern. If not given, it is set to `dtype_out` min./max according to `skimage.util.dtype.dtype_range`.

**dtype_out**
Data type of the rescaled pattern. If not given, it is set to the same data type as the input pattern.

**percentiles**
Disregard intensities outside these percentiles. Calculated per pattern. Will overwrite `in_range` if given.

**Returns**

**rescaled_pattern**
Rescaled pattern.

## 2.11 signals

Experimental and simulated diffraction patterns and virtual backscatter electron images.

#### Modules

| | |
|---|---|
| *util* | Signal utilities for controlling chunking of lazy signal data in `Array` and other array tools. |

### 2.11.1 util

Signal utilities for controlling chunking of lazy signal data in `Array` and other array tools.

#### Functions

| | |
|---|---|
| *get_chunking*([signal, data_shape, nav_dim, ...]) | Get a chunk tuple based on the shape of the signal data. |
| *get_dask_array*(signal[, dtype]) | Return dask array of patterns with appropriate chunking. |
| *grid_indices*(grid_shape, nav_shape[, ...]) | Return indices of a grid evenly spaced in a larger grid of max. |

#### get_chunking

kikuchipy.signals.util.**get_chunking**(*signal:* EBSD | LazyEBSD | *None* = *None*, *data_shape:* *tuple* | *None* = *None*, *nav_dim:* *int* | *None* = *None*, *sig_dim:* *int* | *None* = *None*, *chunk_shape:* *int* | *None* = *None*, *chunk_bytes:* *int* | *float* | *str* | *None* = *30000000.0*, *dtype:* *str* | *dtype* | *type* | *None* = *None*) → tuple

Get a chunk tuple based on the shape of the signal data.

The signal dimensions will not be chunked, and the navigation dimensions will be chunked based on either `chunk_shape`, or be optimized based on the `chunk_bytes`.

This function is inspired by a similar function in `pyxem`.

**Parameters**

**signal**
If not given, the following must be: data shape to be chunked, `data_shape`, the number of navigation dimensions, `nav_dim`, the number of signal dimensions, `sig_dim`, and the data array data type `dtype`.

**data_shape**
Data shape, must be given if `signal` is not.

**nav_dim**
Number of navigation dimensions, must be given if `signal` is not.

**sig_dim**
Number of signal dimensions, must be given if `signal` is not.

**chunk_shape**
Shape of navigation chunks. If not given, this size is set automatically based on `chunk_bytes`. This is a rectangle if `signal` has two navigation dimensions.

**chunk_bytes**
Number of bytes in each chunk. Default is 30e6, i.e. 30 MB. Only used if freedom is given to choose, i.e. if `chunk_shape` is not given. Various parameter types are allowed, e.g. `30000000`, `"30 MB"`, `"30MiB"`, or the default `30e6`, all resulting in approximately 30 MB chunks.

**dtype**
Data type of the array to chunk. Will take precedence over the signal data type if `signal` is given. Must be given if `signal` is not.

**Returns**

**chunks**
Chunk tuple.

### get_dask_array

kikuchipy.signals.util.**get_dask_array**(*signal:* EBSD | LazyEBSD, *dtype:* *str* | *dtype* | *type* | *None* = *None*, ***kwargs*) → Array

Return dask array of patterns with appropriate chunking.

**Parameters**

**signal**
Signal with data to return dask array from.

**dtype**
Data type of returned dask array. This is also passed on to *get_chunking()*.

****kwargs**
Keyword arguments passed to *get_chunking()* to control the number of chunks the output data array is split into. Only `chunk_shape`, `chunk_bytes` and `dtype` are passed on.

**Returns**

**dask_array**
Dask array with signal data with appropriate chunking and data type.

### grid_indices

kikuchipy.signals.util.**grid_indices**(*grid_shape:* *Tuple[int, int]* | *int*, *nav_shape:* *Tuple[int, int]* | *int*, *return_spacing:* *bool* = *False*) → ndarray | Tuple[ndarray, ndarray]

Return indices of a grid evenly spaced in a larger grid of max. two dimensions.

**Parameters**

**grid_shape**
Tuple of integers or just an integer signifying the number of grid indices in each dimension. If 2D, the shape is (n rows, n columns).

**nav_shape**
Tuple of integers or just an integer of giving the shape of the larger grid. If 2D, the shape is (n rows, n columns).

**return_spacing**
Whether to return the spacing in each dimension. Default is `False`.

**Returns**

**indices**
Array of indices of shape `(2,) + grid_shape` or `(1,) + grid_shape` into the larger grid spanned by `nav_shape`.

**spacing**
The spacing in each dimension. Only returned if `return_spacing=True`.

### Examples

```
>>> import kikuchipy as kp
>>> kp.signals.util.grid_indices((4, 5), (55, 75))
array([[[11, 11, 11, 11, 11],
        [22, 22, 22, 22, 22],
        [33, 33, 33, 33, 33],
        [44, 44, 44, 44, 44]],
       [[12, 25, 38, 51, 64],
        [12, 25, 38, 51, 64],
        [12, 25, 38, 51, 64],
        [12, 25, 38, 51, 64]]])
>>> idx, spacing = kp.signals.util.grid_indices(10, 105, return_spacing=True)
>>> idx
array([[ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98]])
>>> spacing
array([10])
```

### Examples using `grid_indices`

- *Fit a plane to selected projection centers*

### Classes

| | |
|---|---|
| *EBSD*(*args, **kwargs) | Scan of Electron Backscatter Diffraction (EBSD) patterns. |
| *EBSDMasterPattern*(*args, **kwargs) | Simulated Electron Backscatter Diffraction (EBSD) master pattern. |
| *ECPMasterPattern*(*args, **kwargs) | Simulated Electron Channeling Pattern (ECP) master pattern. |
| *LazyEBSD*(*args, **kwargs) | Lazy implementation of *EBSD*. |
| *LazyEBSDMasterPattern*(*args, **kwargs) | Lazy implementation of *EBSDMasterPattern*. |
| *LazyECPMasterPattern*(*args, **kwargs) | Lazy implementation of *ECPMasterPattern*. |
| *LazyVirtualBSEImage*(*args, **kwargs) | Lazy implementation of *VirtualBSEImage*. |
| *VirtualBSEImage*(*args, **kwargs) | Virtual backscatter electron (BSE) image(s). |

## 2.11.2 EBSD

**class** kikuchipy.signals.**EBSD**(**args*, ***kwargs*)

Bases: `KikuchipySignal2D`

Scan of Electron Backscatter Diffraction (EBSD) patterns.

This class extends HyperSpy's Signal2D class for EBSD patterns. Some of the docstrings are obtained from HyperSpy. See the docstring of `Signal2D` for the list of inherited attributes and methods.

**Parameters**

***args**
See `Signal2D`.

**detector**
[EBSDDetector, optional] Detector describing the EBSD detector-sample geometry. If not given, this is a default detector (see EBSDDetector).

**static_background**
[ndarray or Array, optional] Static background pattern. If not given, this is None.

**xmap**
[CrystalMap] Crystal map containing the phases, unit cell rotations and auxiliary properties of the EBSD dataset. If not given, this is None.

****kwargs**
See Signal2D.

**See also:**

*kikuchipy.data.nickel_ebsd_small*
An EBSD signal with (3, 3) experimental nickel patterns.

*kikuchipy.data.nickel_ebsd_large*
An EBSD signal with (55, 75) experimental nickel patterns.

**kikuchipy.data.silicon_ebsd_moving_screen_in**
An EBSD signal with one experimental silicon pattern.

**kikuchipy.data.silicon_ebsd_moving_screen_out5mm**
An EBSD signal with one experimental silicon pattern.

**kikuchipy.data.silicon_ebsd_moving_screen_out10mm**
An EBSD signal with one experimental silicon pattern.

### Examples

Load one of the example datasets and inspect some properties

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
>>> s.detector
EBSDDetector (60, 60), px_size 1 um, binning 8, tilt 0, azimuthal 0, pc (0.425, 0.
↪213, 0.501)
>>> s.static_background
array([[84, 87, 90, ..., 27, 29, 30],
       [87, 90, 93, ..., 27, 28, 30],
       [92, 94, 97, ..., 39, 28, 29],
       ...,
       [80, 82, 84, ..., 36, 30, 26],
       [79, 80, 82, ..., 28, 26, 26],
       [76, 78, 80, ..., 26, 26, 25]], dtype=uint8)
>>> s.xmap
Phase  Orientations  Name  Space group  Point group  Proper point group     Color
    0    9 (100.0%)    ni        Fm-3m         m-3m                 432  tab:blue
Properties: scores
Scan unit: px
```

### Attributes

| | |
|---|---|
| *EBSD.detector* | Return or set the detector describing the EBSD detector-sample geometry. |
| *EBSD.static_background* | Return or set the static background pattern. |
| *EBSD.xmap* | Return or set the crystal map containing the phases, unit cell rotations and auxiliary properties of the EBSD dataset. |

### detector

**property** EBSD.**detector: *EBSDDetector***

Return or set the detector describing the EBSD detector-sample geometry.

**Parameters**

**value**
[EBSDDetector] EBSD detector.

### Examples using `EBSD.detector`

- *Pattern binning*
- *Crop navigation axes*
- *Crop signal axes*

### static_background

**property** EBSD.**static_background: ndarray | Array | None**

Return or set the static background pattern.

**Parameters**

**value**
[ndarray or Array] Static background pattern with the same (signal) shape and data type as the EBSD signal.

### Examples using `EBSD.static_background`

- *Pattern binning*
- *Static background correction*
- *Dynamic background correction*
- *Crop signal axes*

### xmap

property EBSD.xmap: CrystalMap | None

Return or set the crystal map containing the phases, unit cell rotations and auxiliary properties of the EBSD dataset.

**Parameters**

**value**
[CrystalMap] Crystal map with the same shape as the signal navigation shape.

### Examples using EBSD.xmap

- *Crop navigation axes*

### Methods

| | |
|---|---|
| *EBSD.adaptive_histogram_equalization*([...]) | Enhance the local contrast using adaptive histogram equalization. |
| *EBSD.as_lazy*(*args, **kwargs) | Create a copy of the given Signal as a `LazySignal`. |
| *EBSD.average_neighbour_patterns*([window, ...]) | Average patterns with its neighbours within a window. |
| *EBSD.change_dtype*(*args, **kwargs) | Change the data type of a Signal. |
| *EBSD.crop*(*args, **kwargs) | Crops the data in a given axis. |
| *EBSD.deepcopy*() | Return a "deep copy" of this Signal using the standard library's `deepcopy()` function. |
| *EBSD.dictionary_indexing*(dictionary[, ...]) | Index patterns by matching each pattern to a dictionary of simulated patterns of known orientations [Chen *et al.*, 2015, Jackson *et al.*, 2019]. |
| *EBSD.downsample*(factor[, dtype_out, ...]) | Downsample the pattern shape by an integer factor and rescale intensities to fill the data type range. |
| *EBSD.extract_grid*(grid_shape[, return_indices]) | Return a new signal with patterns from positions in a grid of shape `grid_shape` evenly spaced in navigation space. |
| *EBSD.fft_filter*(transfer_function, ...[, ...]) | Filter patterns in the frequency domain. |
| *EBSD.get_average_neighbour_dot_product_ma* | Get a map of the average dot product between patterns and their neighbours within an averaging window. |
| *EBSD.get_decomposition_model*([components, ...]) | Get the model signal generated with the selected number of principal components from a decomposition. |
| *EBSD.get_dynamic_background*([filter_domain, ...]) | Return the dynamic background per pattern in a new signal. |
| *EBSD.get_image_quality*([normalize, ...]) | Compute the image quality map of patterns in an EBSD scan. |
| *EBSD.get_neighbour_dot_product_matrices*([.. | Get an array with dot products of a pattern and its neighbours within a window. |
| *EBSD.get_virtual_bse_intensity*(roi[, ...]) | Get a virtual backscatter electron (VBSE) image formed from intensities within a region of interest (ROI) on the detector. |
| *EBSD.hough_indexing*(phase_list, indexer[, ...]) | Index patterns by Hough indexing using `pyebsdindex`. |
| *EBSD.hough_indexing_optimize_pc*(pc0, indexer) | Return a detector with one projection center (PC) per pattern optimized using Hough indexing from `pyebsdindex`. |
| *EBSD.normalize_intensity*([num_std, ...]) | Normalize image intensities to a mean of zero with a given standard deviation. |
| *EBSD.plot_virtual_bse_intensity*(roi[, ...]) | Plot an interactive virtual backscatter electron (VBSE) image formed from intensities within a specified and adjustable region of interest (ROI) on the detector. |
| *EBSD.rebin*(*args, **kwargs) | Rebin the signal into a smaller or larger shape, based on linear interpolation. |
| *EBSD.refine_orientation*(xmap, detector, ...) | Refine orientations by searching orientation space around the best indexed solution using fixed projection centers. |
| *EBSD.refine_orientation_projection_center* | Refine orientations and projection centers simultaneously by searching the orientation and PC parameter space. |
| *EBSD.refine_projection_center*(xmap, ...[, ...]) | Refine projection centers by searching the parameter space using fixed orientations. |
| *EBSD.remove_dynamic_background*([operation, ...]) | Remove the dynamic background. |
| *EBSD.remove_static_background*([operation, ...]) | Remove the static background. |

### adaptive_histogram_equalization

EBSD.**adaptive_histogram_equalization**(*kernel_size: Tuple[int, int] | List[int] | None = None*, *clip_limit: int | float = 0*, *nbins: int = 128*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSD* | *LazyEBSD*

Enhance the local contrast using adaptive histogram equalization.

This method uses `skimage.exposure.equalize_adapthist()`.

**Parameters**

**kernel_size**
Shape of contextual regions for adaptive histogram equalization, default is 1/4 of image height and 1/4 of image width.

**clip_limit**
Clipping limit, normalized between 0 and 1 (higher values give more contrast). Default is `0`.

**nbins**
Number of gray bins for histogram ("data range"), default is `128`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

`s_out`
Equalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*rescale_intensity*
*normalize_intensity*

#### Notes

It is recommended to perform adaptive histogram equalization only *after* static and dynamic background corrections of EBSD patterns, otherwise some unwanted darkening towards the edges might occur.

The default window size might not fit all pattern sizes, so it may be necessary to search for the optimal window size.

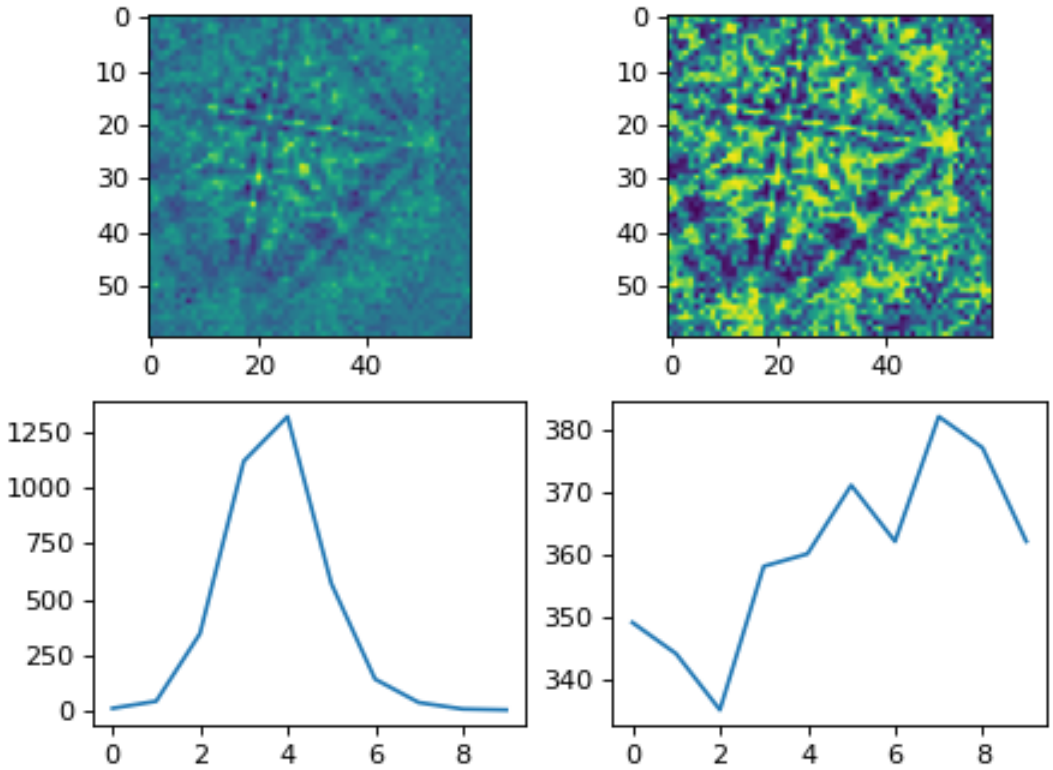
### Examples

Load one pattern from the small nickel dataset, remove the background and perform adaptive histogram equalization. A copy without equalization is kept for comparison.

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small().inav[0, 0]
>>> s.remove_static_background()
>>> s.remove_dynamic_background()
>>> s2 = s.deepcopy()
>>> s2.adaptive_histogram_equalization()
```

Compute the intensity histograms and plot the patterns and histograms

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> hist, _ = np.histogram(s.data, range=(0, 255))
>>> hist2, _ = np.histogram(s2.data, range=(0, 255))
>>> _, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
>>> _ = ax0.imshow(s.data)
>>> _ = ax1.imshow(s2.data)
>>> _ = ax2.plot(hist)
>>> _ = ax3.plot(hist2)
```



### Examples using `EBSD.adaptive_histogram_equalization`

- *Adaptive histogram equalization*

### as_lazy

`EBSD.as_lazy`(**args*, ***kwargs*) → *LazyEBSD*

Create a copy of the given Signal as a `LazySignal`.

This docstring was copied from HyperSpy's Signal2D.as_lazy. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**copy_variance**
[bool] Whether or not to copy the variance from the original Signal to the new lazy version. Default is True.

**copy_navigator**
[bool] Whether or not to copy the navigator from the original Signal to the new lazy version. Default is True.

**copy_learning_results**
[bool] Whether to copy the learning_results from the original signal to the new lazy version. Default is True.

**Returns**

**res**
[`LazySignal`] The same signal, converted to be lazy

### average_neighbour_patterns

`EBSD.average_neighbour_patterns`(*window: str | ndarray | Array | Window = 'circular'*, *window_shape: Tuple[int, ...] = (3, 3)*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*, ***kwargs*) → None | *EBSD* | *LazyEBSD*

Average patterns with its neighbours within a window.

The amount of averaging is specified by the window coefficients. All patterns are averaged with the same window. Map borders are extended with zeros. Resulting pattern intensities are rescaled to fill the input patterns' data type range individually.

Averaging is accomplished by correlating the window with the extended array of patterns using `scipy.ndimage.correlate()`.

**Parameters**

**window**
Name of averaging window or an array. Available types are listed in `scipy.signal.windows.get_window()`, in addition to a `"circular"` window (default) filled with ones in which corner coefficients are set to zero. A window element is considered to be in a corner if its radial distance to the origin (window center) is shorter or equal to the half width of the window's longest axis. A 1D or 2D `ndarray`, `Array` or *`Window`* can also be passed.

**window_shape**
Shape of averaging window. Not used if a custom window or *Window* is passed to `window`. This can be either 1D or 2D, and can be asymmetrical. Default is `(3, 3)`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

****kwargs**
Keyword arguments passed to the available window type listed in `get_window()`. If not given, the default values of that particular window are used.

**Returns**

**s_out**
Averaged signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*kikuchipy.filters.Window*, `scipy.signal.windows.get_window`
`scipy.ndimage.correlate`

### Examples using `EBSD.average_neighbour_patterns`

- *Neighbour pattern averaging*

### change_dtype

EBSD.**change_dtype**(**args*, ***kwargs*) → None

Change the data type of a Signal.

This docstring was copied from HyperSpy's Signal2D.change_dtype. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**dtype**
[`str` or `numpy.dtype`] Typecode string or data-type to which the Signal's data array is cast. In addition to all the standard numpy Data type objects (dtype), HyperSpy supports four extra dtypes for RGB images: `'rgb8'`, `'rgba8'`, `'rgb16'`, and `'rgba16'`. Changing from and to any `rgb(a)` *dtype* is more constrained than most other *dtype* conversions. To change to an `rgb(a)` *dtype*, the *signal_dimension* must be 1, and its size should be 3 (for `rgb`) or 4 (for `rgba`) *dtypes*. The original *dtype* should be `uint8` or `uint16` if converting to `rgb(a)8` or `rgb(a))16`, and the *navigation_dimension* should be at least 2. After conversion, the *signal_dimension* becomes 2. The *dtype* of images with original *dtype* `rgb(a)8` or `rgb(a)16` can only be changed to `uint8` or `uint16`, and the *signal_dimension* becomes 1.

**rechunk: bool**
Only has effect when operating on lazy signal. If `True` (default), the data may be automatically rechunked before performing this operation.

### Examples

```
>>> s = hs.signals.Signal1D([1,2,3,4,5])
>>> s.data
array([1, 2, 3, 4, 5])
>>> s.change_dtype('float')
>>> s.data
array([ 1.,  2.,  3.,  4.,  5.])
```

## crop

`EBSD.crop`(**args*, ***kwargs*)

Crops the data in a given axis. The range is given in pixels.

This docstring was copied from HyperSpy's Signal2D.crop. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**axis**
[`int` or `str`] Specify the data axis in which to perform the cropping operation. The axis can be specified using the index of the axis in *axes_manager* or the axis name.

**start**
[`int`, `float`, or `None`] The beginning of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start*/*end* is `None` the method crops from/to the low/high end of the axis.

**end**
[`int`, `float`, or `None`] The end of the cropping interval. If type is `int`, the value is taken as the axis index. If type is `float` the index is calculated using the axis calibration. If *start*/*end* is `None` the method crops from/to the low/high end of the axis.

**convert_units**
[bool] Default is `False`. If `True`, convert the units using the `convert_units()` method of the `AxesManager`. If `False`, does nothing.

### Examples using `EBSD.crop`

- *Crop navigation axes*
- *Crop signal axes*

### deepcopy

EBSD.**deepcopy**() → *EBSD*

Return a "deep copy" of this Signal using the standard library's `deepcopy()` function. Note: this means the underlying data structure will be duplicated in memory.

This docstring was copied from HyperSpy's Signal2D.deepcopy. Some inconsistencies with the kikuchipy version may exist.

**See also:**

`copy()`

### Examples using `EBSD.deepcopy`

- *Crop navigation axes*
- *Crop signal axes*

### dictionary_indexing

EBSD.**dictionary_indexing**(*dictionary:* EBSD, *metric:* SimilarityMetric | *str* = *'ncc'*, *keep_n:* *int* = *20*, *n_per_iteration:* *int* | *None* = *None*, *navigation_mask:* *ndarray* | *None* = *None*, *signal_mask:* *ndarray* | *None* = *None*, *rechunk:* *bool* = *False*, *dtype:* *str* | *dtype* | *type* | *None* = *None*) → CrystalMap

Index patterns by matching each pattern to a dictionary of simulated patterns of known orientations [Chen *et al.*, 2015, Jackson *et al.*, 2019].

**Parameters**

**dictionary**
One EBSD signal with dictionary patterns. The signal must have a 1D navigation axis, an *xmap* property with crystal orientations set, and equal detector shape.

**metric**
Similarity metric, by default `"ncc"` (normalized cross-correlation). `"ndp"` (normalized dot product) is also available. A valid user-defined similarity metric may be used instead. The metric must be a class implementing the *SimilarityMetric* abstract class methods. See *NormalizedCrossCorrelationMetric* and *NormalizedDotProductMetric* for examples.

**keep_n**
Number of best matches to keep, by default 20 or the number of dictionary patterns if fewer than 20 are available.

**n_per_iteration**
Number of dictionary patterns to compare to all experimental patterns in each indexing iteration. If not given, and the dictionary is a `LazyEBSD` signal, it is equal to the chunk size of the first pattern array axis, while if if is an `EBSD` signal, it is set equal to the number of dictionary patterns, yielding only one iteration. This parameter can be increased to use less memory during indexing, but this will increase the computation time.

**navigation_mask**
A boolean mask equal to the signal's navigation (map) shape, where only patterns equal to `False` are indexed. This can be used by `metric` in *prepare_experimental()*. If not given, all patterns are indexed.

**signal_mask**
A boolean mask equal to the experimental patterns' detector shape, where only pixels equal to `False` are matched. This can be used by `metric` in *prepare_experimental()*. If not given, all pixels are used.

**rechunk**
Whether `metric` is allowed to rechunk experimental and dictionary patterns before matching. Default is `False`. Rechunking usually makes indexing faster, but uses more memory. If a custom `metric` is passed, whatever *rechunk* is set to will be used.

**dtype**
Which data type `metric` shall cast the patterns to before matching. If not given, `"float32"` will be used unless a custom `metric` is passed and it has set the *dtype*, which will then be used instead. `"float32"` and `"float64"` are allowed for the available `"ncc"` and `"ndp"` metrics.

**Returns**

***xmap***
A crystal map with `keep_n` rotations per point with the sorted best matching orientations in the dictionary. The corresponding best scores and indices into the dictionary are stored in the `xmap.prop` dictionary as `"scores"` and `"simulation_indices"`.

**See also:**

***refine_orientation***
***refine_projection_center***
***refine_orientation_projection_center***
***kikuchipy.indexing.SimilarityMetric***
***kikuchipy.indexing.NormalizedCrossCorrelationMetric***
***kikuchipy.indexing.NormalizedDotProductMetric***
***kikuchipy.indexing.merge_crystal_maps***
Merge multiple single phase crystal maps into one multi phase map.

***kikuchipy.indexing.orientation_similarity_map***
Calculate an orientation similarity map.

### downsample

EBSD.**downsample**(*factor: int*, *dtype_out: str | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSD* | *LazyEBSD*

Downsample the pattern shape by an integer factor and rescale intensities to fill the data type range.

**Parameters**

**factor**
Integer binning factor to downsample by. Must be a divisor of the initial pattern shape $(s_y, s_x)$. The new pattern shape given by the `factor` $k$ is $(s_y/k, s_x/k)$.

**dtype_out**
Name of the data type of the new patterns overwriting `data`. Contrast between patterns is lost. If not given, patterns maintain their data type and. Patterns are rescaled to fill the data type range.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Downsampled signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*rebin*, *crop*

### Notes

This method differs from *rebin()* in that intensities are rescaled after binning in order to maintain the data type. If rescaling is undesirable, use *rebin()* instead.

### Examples using `EBSD.downsample`

- *Pattern binning*

## extract_grid

EBSD.**extract_grid**(*grid_shape: Tuple[int, int] | int*, *return_indices: bool = False*) → *EBSD* | *LazyEBSD* | Tuple[*EBSD* | *LazyEBSD*, ndarray]

Return a new signal with patterns from positions in a grid of shape `grid_shape` evenly spaced in navigation space.

**Parameters**

**grid_shape**
Tuple of integers or just an integer signifying the number of grid indices in each dimension. If 2D, the shape is (n columns, n rows).

**return_indices**
Whether to return the indices of the extracted patterns into `data` as an array of shape `(2,) + grid_shape`. Default is `False`.

**Returns**

**new**
New signal with patterns from indices in a grid corresponding to `grid_shape`. Attributes *xmap*, *static_background* and *detector* are deep copied.

**indices**
Indices of the extracted patterns into `data`, returned if `return_indices=True`.

### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_large(lazy=True)
>>> s
<LazyEBSD, title: patterns Scan 1, dimensions: (75, 55|60, 60)>
>>> s2 = s.extract_grid((5, 4))
>>> s2
<LazyEBSD, title: patterns Scan 1, dimensions: (5, 4|60, 60)>
```

### Examples using `EBSD.extract_grid`

- *Extract patterns from a grid*

### fft_filter

EBSD.**fft_filter**(*transfer_function: ndarray | Window*, *function_domain: str*, *shift: bool = False*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSD* | *LazyEBSD*

Filter patterns in the frequency domain.

Patterns are transformed via the Fast Fourier Transform (FFT) to the frequency domain, where their spectrum is multiplied by the `transfer_function`, and the filtered spectrum is subsequently transformed to the spatial domain via the inverse FFT (IFFT). Filtered patterns are rescaled to input data type range.

Note that if `function_domain` is `"spatial"`, only real valued FFT and IFFT is used.

**Parameters**

**transfer_function**
Filter to apply to patterns. This can either be a transfer function in the frequency domain of pattern shape or a kernel in the spatial domain. What is passed is determined from `function_domain`.

**function_domain**
Options are `"frequency"` and `"spatial"`, indicating, respectively, whether the filter function passed to `filter_function` is a transfer function in the frequency domain or a kernel in the spatial domain.

**shift**
Whether to shift the zero-frequency component to the center. Default is `False`. This is only used when `function_domain="frequency"`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Filtered signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*kikuchipy.filters.Window*

### Examples

Applying a Gaussian low pass filter with a cutoff frequency of 20:

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> pattern_shape = s.axes_manager.signal_shape[::-1]
>>> w = kp.filters.Window(
...     "lowpass", cutoff=20, shape=pattern_shape
... )
>>> s.fft_filter(
...     transfer_function=w,
...     function_domain="frequency",
...     shift=True,
... )
```

## get_average_neighbour_dot_product_map

EBSD.**get_average_neighbour_dot_product_map**(*window: Window | None = None*, *zero_mean: bool = True*, *normalize: bool = True*, *dtype_out: str | dtype | type = 'float32'*, *dp_matrices: ndarray | None = None*, *show_progressbar: bool | None = None*) → ndarray | Array

Get a map of the average dot product between patterns and their neighbours within an averaging window.

**Parameters**

**window**
Window with integer coefficients defining the neighbours to calculate the average with. If not given, the four nearest neighbours are used. Must have the same number of dimensions as signal navigation dimensions.

**zero_mean**
Whether to subtract the mean of each pattern individually to center the intensities about zero before calculating the dot products. Default is `True`.

**normalize**
Whether to normalize the pattern intensities to a standard deviation of 1 before calculating the dot products. This operation is performed after centering the intensities if `zero_mean=True`. Default is `True`.

**dtype_out**
Data type of the output map. Default is `"float32"`.

**dp_matrices**
Optional pre-calculated dot product matrices, by default `None`. If an array is passed, the average dot product map is calculated from this array. The `dp_matrices` array can be

obtained from *get_neighbour_dot_product_matrices()*. Its shape must correspond to the signal's navigation shape and the window's shape.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**Returns**

**adp**
Average dot product map.

### get_decomposition_model

EBSD.**get_decomposition_model**(*components: int | List[int] | None = None*, *dtype_out: str | dtype | type = 'float32'*) → *EBSD | LazyEBSD*

Get the model signal generated with the selected number of principal components from a decomposition.

Calls HyperSpy's `get_decomposition_model()`. Learning results are preconditioned before this call, doing the following:

1. Set `numpy.dtype` to desired `dtype_out`.
2. Remove unwanted components.
3. Rechunk to suitable chunks if `Array`.

**Parameters**

**components**
If not given, rebuilds the signal from all components. If `int`, rebuilds signal from `components` in range 0-given `int`. If list of `ints`, rebuilds signal from only `components` in given list.

**dtype_out**
Data type to cast learning results to (default is `"float32"`). Note that HyperSpy casts to `"float64"`.

**Returns**

**s_model**
Model signal.

### get_dynamic_background

EBSD.**get_dynamic_background**(*filter_domain: str = 'frequency'*, *std: int | float | None = None*, *truncate: int | float = 4.0*, *dtype_out: str | dtype | type | None = None*, *show_progressbar: bool | None = None*, *lazy_output: bool | None = None*, ***kwargs*) → *EBSD | LazyEBSD*

Return the dynamic background per pattern in a new signal.

**Parameters**

**filter_domain**
Whether to apply a Gaussian convolution filter in the `"frequency"` (default) or `"spatial"` domain.

**std**
Standard deviation of the Gaussian window. If not given, it is set to width/8.

**truncate**
Truncate the Gaussian filter at this many standard deviations. Default is `4.0`.

**dtype_out**
Data type of the background patterns. If not given, it is set to the same data type as the input pattern.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal.

****kwargs**
Keyword arguments passed to the Gaussian blurring function determined from `filter_domain`.

**Returns**

**s_out**
Signal with the large scale variations across the detector. Whether it is lazy is determined from `lazy_output`.

### get_image_quality

EBSD.**get_image_quality**(*normalize: bool = True*, *show_progressbar: bool | None = None*) → ndarray | Array

Compute the image quality map of patterns in an EBSD scan.

The image quality $Q$ is calculated based on the procedure defined by Krieger Lassen [Lassen, 1994].

**Parameters**

**normalize**
Whether to normalize patterns to a mean of zero and standard deviation of 1 before calculating the image quality. Default is `True`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**Returns**

**image_quality_map**
Image quality map of same shape as navigation axes. This is a Dask array if the signal is lazy.

**See also:**

*kikuchipy.pattern.get_image_quality*

### Examples

Load an example dataset, remove the static and dynamic background and compute $Q$

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s
<EBSD, title: patterns Scan 1, dimensions: (3, 3|60, 60)>
>>> s.remove_static_background()
>>> s.remove_dynamic_background()
>>> iq = s.get_image_quality()
>>> iq
array([[0.19935645, 0.16657268, 0.18803978],
       [0.19040637, 0.1616931 , 0.17834103],
       [0.19411428, 0.16031407, 0.18413563]], dtype=float32)
```

## Examples using `EBSD.get_image_quality`

- *Neighbour pattern averaging*

## get_neighbour_dot_product_matrices

EBSD.**get_neighbour_dot_product_matrices**(*window:* Window | *None* = *None*, *zero_mean:* *bool* = *True*, *normalize:* *bool* = *True*, *dtype_out:* *str* | *dtype* | *type* = *'float32'*, *show_progressbar:* *bool* | *None* = *None*) → ndarray | Array

Get an array with dot products of a pattern and its neighbours within a window.

**Parameters**

**window**
Window with integer coefficients defining the neighbours to calculate the dot products with. If not given, the four nearest neighbours are used. Must have the same number of dimensions as signal navigation dimensions.

**zero_mean**
Whether to subtract the mean of each pattern individually to center the intensities about zero before calculating the dot products. Default is `True`.

**normalize**
Whether to normalize the pattern intensities to a standard deviation of 1 before calculating the dot products. This operation is performed after centering the intensities if `zero_mean=True`. Default is `True`.

**dtype_out**
Data type of the output map. Default is `"float32"`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**Returns**

**dp_matrices**
Dot products between a pattern and its nearest neighbours.

### get_virtual_bse_intensity

EBSD.**get_virtual_bse_intensity**(*roi: BaseInteractiveROI*, *out_signal_axes: Iterable[int] | Iterable[str] | None = None*) → *VirtualBSEImage*

Get a virtual backscatter electron (VBSE) image formed from intensities within a region of interest (ROI) on the detector.

Adapted from `pyxem.signals.common_diffraction.CommonDiffraction.get_integrated_intensity()`.

**Parameters**

**roi**
Any interactive ROI detailed in HyperSpy.

**out_signal_axes**
Which navigation axes to use as signal axes in the virtual image. If not given, the first two navigation axes are used.

**Returns**

**virtual_image**
VBSE image formed from detector intensities within an ROI on the detector.

**See also:**

*plot_virtual_bse_intensity*

#### Examples

```
>>> import hyperspy.api as hs
>>> import kikuchipy as kp
>>> rect_roi = hs.roi.RectangularROI(
...     left=0, right=5, top=0, bottom=5
... )
>>> s = kp.data.nickel_ebsd_small()
>>> vbse_image = s.get_virtual_bse_intensity(rect_roi)
```

### Examples using `EBSD.get_virtual_bse_intensity`

- *Extract patterns from a grid*

### hough_indexing

EBSD.**hough_indexing**(*phase_list: PhaseList*, *indexer: EBSDIndexer*, *chunksize: int = 528*, *verbose: int = 1*, *return_index_data: bool = False*, *return_band_data: bool = False*) → CrystalMap | Tuple[CrystalMap, np.ndarray] | Tuple[CrystalMap, np.ndarray, np.ndarray]

Index patterns by Hough indexing using `pyebsdindex`.

See `EBSDIndexer` and `index_pats()` for details.

Currently, PyEBSDIndex only supports indexing with a single mean projection center (PC).

**Parameters**

**phase_list**
List of phases. The list must correspond to the phase list in the passed.

**indexer**
PyEBSDIndex EBSD indexer instance of which the `index_pats()` method is called. Its *phaselist* must be compatible with the given `phase_list`, and the `indexer.vendor` must be `"KIKUCHIPY"`. An indexer can be obtained with *`get_indexer()`*.

**chunksize**
Number of patterns to index at a time. Default is the minimum of 528 or the number of patterns in the signal. Increasing the chunksize may give faster indexing but increases memory use.

**verbose**
Which information to print from PyEBSDIndex. Options are 0 - no output, 1 - timings (default), 2 - timings and the Hough transform of the first pattern with detected bands highlighted.

**return_index_data**
Whether to return the index data array returned from `EBSDIndexer.index_pats()` in addition to the resulting crystal map. Default is `False`.

**return_band_data**
Whether to return the band data array returned from `EBSDIndexer.index_pats()`. Default is `False`.

**Returns**

***xmap***
Crystal map with indexing results.

**index_data**
Array returned from `EBSDIndexer.index_pats()`, returned if `return_index_data=True`.

**band_data**
Array returned from `EBSDIndexer.index_pats()`, returned if `return_band_data=True`.

### Notes

Requires `pyebsdindex` to be installed. See *Optional dependencies* for further details.

This wrapper of PyEBSDIndex is meant for convenience more than speed. It uses the GPU if `pyopencl` is installed, but only uses a single thread. If you need the fastest indexing, refer to the PyEBSDIndex documentation for multi-threading and more.

## hough_indexing_optimize_pc

`EBSD.`**`hough_indexing_optimize_pc`**(*pc0: list | tuple | np.ndarray*, *indexer: EBSDIndexer*, *batch: bool = False*, *method: str = 'Nelder-Mead'*, ***kwargs*) → *EBSDDetector*

Return a detector with one projection center (PC) per pattern optimized using Hough indexing from `pyebsdindex`.

See `EBSDIndexer` and `pcopt` for details.

**Parameters**

**pc0**
A single initial guess of PC for all patterns in Bruker's convention, (PCx, PCy, PCz).

**indexer**
PyEBSDIndex EBSD indexer instance to pass on to the optimization function. An indexer can be obtained with *get_indexer()*.

**batch**
Whether the fit for the patterns should be optimized using the cumulative fit for all patterns (`False`, default), or if an optimization is run for each pattern individually.

**method**
Which optimization method to use, either `"Nelder-Mead"` from SciPy (default) or `"PSO"` (particle swarm).

****kwargs**
Keyword arguments passed on to PyEBSDIndex' optimization method (depending on the chosen `method`).

**Returns**

**new_detector**
EBSD detector with one PC if `batch=False` or one PC per pattern if `batch=True`. The detector attributes are extracted from `indexer.sampleTilt` etc.

### Notes

Requires `pyebsdindex` to be installed. See *Optional dependencies* for further details.

## normalize_intensity

EBSD.**normalize_intensity**(*num_std: int = 1*, *divide_by_square_root: bool = False*, *dtype_out: str | dtype | type | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSD* | *LazyEBSD*

Normalize image intensities to a mean of zero with a given standard deviation.

**Parameters**

**num_std**
Number of standard deviations of the output intensities. Default is `1`.

**divide_by_square_root**
Whether to divide output intensities by the square root of the signal dimension size. Default is `False`.

**dtype_out**
Data type of normalized images. If not given, the input images' data type is used.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Normalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

### Notes

Data type should always be changed to floating point, e.g. `float32` with `change_dtype()`, before normalizing the intensities.

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> np.mean(s.data)
146.0670987654321
>>> s.normalize_intensity(dtype_out=np.float32)
>>> np.mean(s.data)
2.6373216e-08
```

## plot_virtual_bse_intensity

EBSD.**plot_virtual_bse_intensity**(*roi: BaseInteractiveROI*, *out_signal_axes: Iterable[int] | Iterable[str] | None = None*, ***kwargs*) → None

Plot an interactive virtual backscatter electron (VBSE) image formed from intensities within a specified and adjustable region of interest (ROI) on the detector.

Adapted from `pyxem.signals.common_diffraction.CommonDiffraction.plot_integrated_intensity()`.

**Parameters**

**roi**
Any interactive ROI detailed in HyperSpy.

**out_signal_axes**
Which navigation axes to use as signal axes in the virtual image. If not given, the first two navigation axes are used.

****kwargs:**
Keyword arguments passed to the `plot()` method of the virtual image.

**See also:**

*get_virtual_bse_intensity*

### Examples

```
>>> import hyperspy.api as hs
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> rect_roi = hs.roi.RectangularROI(
...     left=0, right=5, top=0, bottom=5
... )
>>> s.plot_virtual_bse_intensity(rect_roi)
```

## rebin

EBSD.**rebin**(**args*, ***kwargs*)

Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify **either** *new_shape* or *scale*. Scale of 1 means no binning and scale less than one results in up-sampling.

This docstring was copied from HyperSpy's Signal2D.rebin. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**new_shape**
[`list` (`of floats` or `integer`) `or None`] For each dimension specify the new_shape. This will internally be converted into a `scale` parameter.

**scale**
[`list` (`of floats` or `integer`) `or None`] For each dimension, specify the new:old pixel ratio, e.g. a ratio of 1 is no binning and a ratio of 2 means that each pixel in the new spectrum is twice the size of the pixels in the old spectrum. The length of the list should match the dimension of the Signal's underlying data array. *Note : Only one of `scale` or `new_shape` should be specified, otherwise the function will not run*

**crop**
[bool] Whether or not to crop the resulting rebinned data (default is `True`). When binning by a non-integer number of pixels it is likely that the final row in each dimension will contain fewer than the full quota to fill one pixel. For example, a 5*5 array binned by 2.1 will produce two rows containing 2.1 pixels and one row containing only 0.8 pixels. Selection of `crop=True` or `crop=False` determines whether or not this *"black"* line is cropped from the final binned array or not. *Please note that if ``crop=False`` is used, the final row in each dimension may appear black if a fractional number of pixels are left over. It can be removed but has been left to preserve total counts before and after binning.*

**dtype**
[{`None`, `numpy.dtype`, "same"}] Specify the dtype of the output. If None, the dtype will be determined by the behaviour of `numpy.sum()`, if "same", the dtype will be kept the same. Default is None.

**out**
[`BaseSignal` (`or subclasses`) `or None`] If `None`, a new Signal is created with the result of the operation and returned (default). If a Signal is passed, it is used to receive the output of the operation, and nothing is returned.

**Returns**

**s**
[`BaseSignal` (`or subclass`)] The resulting cropped signal.

**Raises**

**NotImplementedError**
If trying to rebin over a non-uniform axis.

### Examples

```
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<EDXTEMSpectrum, title: dimensions: (4, 4|10)>
>>> print ('Sum = ', sum(sum(sum(spectrum.data))))
Sum = 164.0
```

```
>>> scale = [2, 2, 5]
>>> test = spectrum.rebin(scale)
>>> print(test)
<EDSTEMSpectrum, title: dimensions (2, 2|2)>
>>> print('Sum = ', sum(sum(sum(test.data))))
Sum =  164.0
```

```
>>> s = hs.signals.Signal1D(np.ones((2, 5, 10), dtype=np.uint8)
>>> print(s)
<Signal1D, title: , dimensions: (5, 2|10)>
>>> print(s.data.dtype)
uint8
```

Use dtype=np.unit16 to specify a dtype

```
>>> s2 = s.rebin(scale=(5, 2, 1), dtype=np.uint16)
>>> print(s2.data.dtype)
uint16
```

Use dtype="same" to keep the same dtype

```
>>> s3 = s.rebin(scale=(5, 2, 1), dtype="same")
>>> print(s3.data.dtype)
uint8
```

By default *dtype=None*, the dtype is determined by the behaviour of numpy.sum, in this case, unsigned integer of the same precision as the platform interger

```
>>> s4 = s.rebin(scale=(5, 2, 1))
>>> print(s4.data.dtype)
uint64
```

### Examples using `EBSD.rebin`

- *Pattern binning*

### refine_orientation

EBSD.**refine_orientation**(*xmap: CrystalMap*, *detector:* EBSDDetector, *master_pattern:* EBSDMasterPattern, *energy: int | float*, *navigation_mask: np.ndarray | None = None*, *signal_mask: np.ndarray | None = None*, *pseudo_symmetry_ops: Rotation | None = None*, *method: str | None = 'minimize'*, *method_kwargs: dict | None = None*, *trust_region: tuple | list | np.ndarray | None = None*, *initial_step: float = None*, *rtol: float = 0.0001*, *maxeval: int | None = None*, *compute: bool = True*, *rechunk: bool = True*, *chunk_kwargs: dict | None = None*) → CrystalMap | da.Array

Refine orientations by searching orientation space around the best indexed solution using fixed projection centers.

Refinement attempts to maximize the similarity between patterns in this signal and simulated patterns projected from a master pattern. The similarity metric used is the normalized cross-correlation (NCC). The orientation, represented by a Euler angle triplet ($\phi_1$, $\Phi$, $\phi_2$) relative to the EDAX TSL sample reference frame RD-TD-ND, is optimized during refinement, while the sample-detector geometry, represented by the three projection center (PC) parameters (PCx, PCy, PCz) in the Bruker convention, is fixed.

A subset of the optimization methods in *SciPy* and *NLopt* are available:

- **Local optimization:**
  - `minimize()` (includes Nelder-Mead, Powell etc.).
  - Nelder-Mead via nlopt.LN_NELDERMEAD
- **Global optimization:**
  - `differential_evolution()`
  - `dual_annealing()`
  - `basinhopping()`
  - `shgo()`

**Parameters**

**xmap**
Crystal map with points to refine. Only the points in the data (see `CrystalMap`) are refined. If a `navigation_mask` is given, points equal to points in the data and points equal to `False` in this mask are refined.

**detector**
Detector describing the detector-sample geometry with either one PC to be used for all map points or one for each point.

**master_pattern**
Master pattern in the square Lambert projection of the same phase as the one in the crystal map.

**energy**
Accelerating voltage of the electron beam in kV specifying which master pattern energy to use during projection of simulated patterns.

**navigation_mask**
A boolean mask of points in the crystal map to refine (equal to `False`, i.e. points to *mask out* are `True`). The mask must be of equal shape to the signal's navigation shape. If not given, all points in the crystal map data are refined.

**signal_mask**
A boolean mask of detector pixels to use in refinement (equal to `False`, i.e. pixels to *mask out* are `True`). The mask must be of equal shape to the signal's signal shape. If not given, all pixels are used.

**pseudo_symmetry_ops**
Pseudo-symmetry operators as rotations. If given, each map point will be refined using the map orientation and the orientation after applying each operator. The chosen solution is the one with the highest score. E.g. if two operators are given, each map point is refined three times. If given, the returned crystal map will have a property array with the operator index giving the best score, with 0 meaning the original map point gave the best score.

**method**
Name of the `scipy.optimize` or *NLopt* optimization method, among `"minimize"`, `"differential_evolution"`, `"dual_annealing"`, `"basinhopping"`, `"shgo"` and `"ln_neldermead"` (from *NLopt*). Default is `"minimize"`, which by default performs local optimization with the Nelder-Mead method, unless another `"minimize"` method is passed to `method_kwargs`.

**method_kwargs**
Keyword arguments passed to the `scipy.optimize` `method`. For example, to perform refinement with the modified Powell algorithm from *SciPy*, pass `method="minimize"` and `method_kwargs=dict(method="Powell")`. Not used if `method="LN_NELDERMEAD"`.

**trust_region**
List of three +/- angular deviations in degrees used to determine the bound constraints on the three Euler angles per navigation point, e.g. `[2, 2, 2]`. Not passed to *SciPy* `method` if it does not support bounds. The definition ranges of the Euler angles are $\phi_1 \in [0, 360]$, $\Phi \in [0, 180]$ and $\phi_2 \in [0, 360]$ in radians.

**initial_step**
A single initial step size for all Euler angle, in degrees. Only used if `method="LN_NELDERMEAD"`. If not given, this is not set for the *NLopt* optimizer.

**rtol**
Stop optimization of a pattern when the difference in NCC score between two iterations is below this value (relative tolerance). Default is `1e-4`. Only used if `method="LN_NELDERMEAD"`.

**maxeval**
Stop optimization of a pattern when the number of function evaluations exceeds this value, e.g. `100`. Only used if `method="LN_NELDERMEAD"`.

**compute**
Whether to refine now (`True`) or later (`False`). Default is `True`. See `compute()` for more details.

**rechunk**
If `True` (default), rechunk the dask array with patterns used in refinement (not the signal data inplace) if it is returned from *`get_dask_array()`* in a single chunk. This ensures small data sets are rechunked so as to utilize multiple CPUs.

**chunk_kwargs**
Keyword arguments passed to *`get_chunking()`* if `rechunk=True` and the dask array with

patterns used in refinement is returned from *get_dask_array()* in a single chunk.

**Returns**

**out**
If `compute=True`, a crystal map with refined orientations, NCC scores in a `"scores"` property, the number of function evaluations in a `"num_evals"` property and which pseudo-symmetry operator gave the best score if `pseudo_symmetry_ops` is given is returned.. If `compute=False`, a dask array of navigation size + (5,) (or (6,) if `pseudo_symmetry_ops` is passed) is returned, to be computed later. See *compute_refine_orientation_results()*. Each navigation point in the data has the optimized score, the number of function evaluations, the three Euler angles in radians and potentially the pseudo-symmetry operator index in element 0, 1, 2, 3, 4 and 5, respectively.

**See also:**

`scipy.optimize`, *refine_projection_center*
*refine_orientation_projection_center*

### Notes

*NLopt* is for now an optional dependency, see *Optional dependencies* for details. Be aware that *NLopt* does not fail gracefully. If continued use of *NLopt* proves stable enough, its implementation of the Nelder-Mead algorithm might become the default.

## refine_orientation_projection_center

EBSD.**refine_orientation_projection_center**(*xmap: CrystalMap*, *detector:* EBSDDetector, *master_pattern:* EBSDMasterPattern, *energy: int | float*, *navigation_mask: np.ndarray | None = None*, *signal_mask: np.ndarray | None = None*, *pseudo_symmetry_ops: Rotation | None = None*, *method: str | None = 'minimize'*, *method_kwargs: dict | None = None*, *trust_region: tuple | list | np.ndarray | None = None*, *initial_step: tuple | list | np.ndarray | None = None*, *rtol: float | None = 0.0001*, *maxeval: int | None = None*, *compute: bool = True*, *rechunk: bool = True*, *chunk_kwargs: dict | None = None*) → Tuple[CrystalMap, *EBSDDetector*] | da.Array

Refine orientations and projection centers simultaneously by searching the orientation and PC parameter space.

Refinement attempts to maximize the similarity between patterns in this signal and simulated patterns projected from a master pattern. The only supported similarity metric is the normalized cross-correlation (NCC). The orientation, represented by a Euler angle triplet ($\phi_1$, $\Phi$, $\phi_2$) relative to the EDAX TSL sample reference frame RD-TD-ND, is optimized during refinement, while the sample-detector geometry, represented by the three projection center (PC) parameters (PCx, PCy, PCz) in the Bruker convention, is fixed.

A subset of the optimization methods in *SciPy* and *NLopt* are available:

- **Local optimization:**
  - `minimize()` (includes Nelder-Mead, Powell etc.).
  - Nelder-Mead via nlopt.LN_NELDERMEAD

- **Global optimization:**
  - differential_evolution()
  - dual_annealing()
  - basinhopping()
  - shgo()

**Parameters**

**xmap**
Crystal map with points to refine. Only the points in the data (see CrystalMap) are refined. If a navigation_mask is given, points equal to points in the data and points equal to False in this mask are refined.

**detector**
Detector describing the detector-sample geometry with either one PC to be used for all map points or one for each point. Which PCs are refined depend on xmap and navigation_mask.

**master_pattern**
Master pattern in the square Lambert projection of the same phase as the one in the crystal map.

**energy**
Accelerating voltage of the electron beam in kV specifying which master pattern energy to use during projection of simulated patterns.

**navigation_mask**
A boolean mask of points in the crystal map to refine (equal to False, i.e. points to *mask out* are True). The mask must be of equal shape to the signal's navigation shape. If not given, all points in the crystal map data are refined.

**signal_mask**
A boolean mask of detector pixels to use in refinement (equal to False, i.e. pixels to *mask out* are True). The mask must be of equal shape to the signal's signal shape. If not given, all pixels are used.

**pseudo_symmetry_ops**
Pseudo-symmetry operators as rotations. If given, each map point will be refined using the map orientation and the orientation after applying each operator. The chosen solution is the one with the highest score. E.g. if two operators are given, each map point is refined three times. If given, the returned crystal map will have a property array with the operator index giving the best score, with 0 meaning the original map point gave the best score.

**method**
Name of the scipy.optimize or *NLopt* optimization method, among "minimize", "differential_evolution", "dual_annealing", "basinhopping", "shgo" and "ln_neldermead" (from *NLopt*). Default is "minimize", which by default performs local optimization with the Nelder-Mead method, unless another "minimize" method is passed to method_kwargs.

**method_kwargs**
Keyword arguments passed to the scipy.optimize method. For example, to perform refinement with the modified Powell algorithm from *SciPy*, pass method="minimize" and method_kwargs=dict(method="Powell"). Not used if method="LN_NELDERMEAD".

**trust_region**
List of three +/- angular deviations in degrees as bound constraints on the three Euler angles

and three +/- deviations in the range [0, 1] as bound constraints on the PC parameters, e.g. `[2, 2, 2, 0.05, 0.05, 0.05]`. Not passed to *SciPy* `method` if it does not support bounds. The definition ranges of the Euler angles are $\phi_1 \in [0, 360]$, $\Phi \in [0, 180]$ and $\phi_2 \in [0, 360]$ in radians, while the definition range of the PC parameters are assumed to be [-2, 2].

**initial_step**
A list of two initial step sizes to use, one in degrees for all Euler angles and one in the range [0, 1] for all PC parameters. Only used if `method="LN_NELDERMEAD"`.

**rtol**
Stop optimization of a pattern when the difference in NCC score between two iterations is below this value (relative tolerance). Only used if `method="LN_NELDERMEAD"`. If not given, this is set to `1e-4`.

**maxeval**
Stop optimization of a pattern when the number of function evaluations exceeds this value, e.g. `100`. Only used if `method="LN_NELDERMEAD"`.

**compute**
Whether to refine now (`True`) or later (`False`). Default is `True`. See `compute()` for more details.

**rechunk**
If `True` (default), rechunk the dask array with patterns used in refinement (not the signal data inplace) if it is returned from *get_dask_array()* in a single chunk. This ensures small data sets are rechunked so as to utilize multiple CPUs.

**chunk_kwargs**
Keyword arguments passed to *get_chunking()* if `rechunk=True` and the dask array with patterns used in refinement is returned from *get_dask_array()* in a single chunk.

**Returns**

**out**
If `compute=True`, a crystal map with refined orientations, NCC scores in a `"scores"` property, the number of function evaluations in a `"num_evals"` property and which pseudo-symmetry operator gave the best score if `pseudo_symmetry_ops` is given is returned, as well as a new EBSD detector with the refined PCs. If `compute=False`, a dask array of navigation size + (8,) (or (9,) if `pseudo_symmetry_ops` is passed) is returned, to be computed later. See *compute_refine_orientation_projection_center_results()*. Each navigation point in the data has the score, the number of function evaluations, the three Euler angles in radians, the three PC parameters and potentially the pseudo-symmetry operator index in element 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, respectively.

**See also:**

`scipy.optimize`, *refine_orientation*, *refine_projection_center*

### Notes

If the crystal map to refine contains points marked as not indexed, the returned detector might not have a 2D navigation shape.

The method attempts to refine the orientations and projection center at the same time for each map point. The optimization landscape is sloppy [Pang *et al.*, 2020], where the orientation and PC can make up for each other. Thus, it is possible that the parameters that yield the highest similarity are incorrect. As always, it is left to the user to ensure that the output is reasonable.

*NLopt* is for now an optional dependency, see *Optional dependencies* for details. Be aware that *NLopt* does not fail gracefully. If continued use of *NLopt* proves stable enough, its implementation of the Nelder-Mead algorithm might become the default.

## refine_projection_center

EBSD.**refine_projection_center**(*xmap: CrystalMap*, *detector:* EBSDDetector, *master_pattern:* EBSDMasterPattern, *energy: int | float*, *navigation_mask: np.ndarray | None = None*, *signal_mask: np.ndarray | None = None*, *method: str | None = 'minimize'*, *method_kwargs: dict | None = None*, *trust_region: tuple | list | np.ndarray | None = None*, *initial_step: float = None*, *rtol: float = 0.0001*, *maxeval: int | None = None*, *compute: bool = True*, *rechunk: bool = True*, *chunk_kwargs: dict | None = None*) → Tuple[np.ndarray, *EBSDDetector*, np.ndarray] | da.Array

Refine projection centers by searching the parameter space using fixed orientations.

Refinement attempts to maximize the similarity between patterns in this signal and simulated patterns projected from a master pattern. The similarity metric used is the normalized cross-correlation (NCC). The sample-detector geometry, represented by the three projection center (PC) parameters (PCx, PCy, PCz) in the Bruker convention, is updated during refinement, while the orientations, defined relative to the EDAX TSL sample reference frame RD-TD-ND, are fixed.

A subset of the optimization methods in *SciPy* and *NLopt* are available:

- **Local optimization:**
  - `minimize()` (includes Nelder-Mead, Powell etc.).
  - Nelder-Mead via nlopt.LN_NELDERMEAD
- **Global optimization:**
  - `differential_evolution()`
  - `dual_annealing()`
  - `basinhopping()`
  - `shgo()`

**Parameters**

**xmap**
Crystal map with points to use in refinement. Only the points in the data (see `CrystalMap`) are used. If a `navigation_mask` is given, points equal to points in the data and points equal to `False` in this mask are used.

**detector**
Detector describing the detector-sample geometry with either one PC to be used for

all map points or one for each point. Which PCs are refined depend on `xmap` and `navigation_mask`.

**master_pattern**
Master pattern in the square Lambert projection of the same phase as the one in the crystal map.

**energy**
Accelerating voltage of the electron beam in kV specifying which master pattern energy to use during projection of simulated patterns.

**navigation_mask**
A boolean mask of points in the crystal map to use in refinement (equal to `False`, i.e. points to *mask out* are `True`). The mask must be of equal shape to the signal's navigation shape. If not given, all points in the crystal map data are used.

**signal_mask**
A boolean mask of detector pixels to use in refinement (equal to `False`, i.e. pixels to *mask out* are `True`). The mask must be of equal shape to the signal's signal shape. If not given, all pixels are used.

**method**
Name of the `scipy.optimize` or *NLopt* optimization method, among `"minimize"`, `"differential_evolution"`, `"dual_annealing"`, `"basinhopping"`, `"shgo"` and `"ln_neldermead"` (from *NLopt*). Default is `"minimize"`, which by default performs local optimization with the Nelder-Mead method, unless another `"minimize"` method is passed to `method_kwargs`.

**method_kwargs**
Keyword arguments passed to the `scipy.optimize` `method`. For example, to perform refinement with the modified Powell algorithm from *SciPy*, pass `method="minimize"` and `method_kwargs=dict(method="Powell")`. Not used if `method="LN_NELDERMEAD"`.

**trust_region**
List of three +/- deviations in the range [0, 1] used to determine the bounds constraints on the PC parameters per navigation point, e.g. `[0.05, 0.05, 0.05]`. Not passed to *SciPy* `method` if it does not support bounds. The definition range of the PC parameters are assumed to be [-2, 2].

**initial_step**
A single initial step size for all PC parameters in the range [0, 1]. Only used if `method="LN_NELDERMEAD"`.

**rtol**
Stop optimization of a pattern when the difference in NCC score between two iterations is below this value (relative tolerance). Default is `1e-4`. Only used if `method="LN_NELDERMEAD"`.

**maxeval**
Stop optimization of a pattern when the number of function evaluations exceeds this value, e.g. `100`. Only used if `method="LN_NELDERMEAD"`.

**compute**
Whether to refine now (`True`) or later (`False`). Default is `True`. See `compute()` for more details.

**rechunk**
If `True` (default), rechunk the dask array with patterns used in refinement (not the signal data inplace) if it is returned from *`get_dask_array()`* in a single chunk. This ensures small data sets are rechunked so as to utilize multiple CPUs.

**chunk_kwargs**
Keyword arguments passed to *get_chunking()* if `rechunk=True` and the dask array with patterns used in refinement is returned from *get_dask_array()* in a single chunk.

**Returns**

**out**
New similarity metrics, a new EBSD detector instance with the refined PCs and the number of function evaluations if `compute=True`. If `compute=False`, a dask array of navigation size + (5,) is returned, to be computed later. See *compute_refine_projection_center_results()*. Each navigation point has the optimized score, the three PC parameters in the Bruker convention and the number of function evaluations in element 0, 1, 2, 3 and 4, respectively.

**See also:**

`scipy.optimize`, *refine_orientation*
*refine_orientation_projection_center*

### Notes

If the crystal map to refine contains points marked as not indexed, the returned detector might not have a 2D navigation shape.

*NLopt* is for now an optional dependency, see *Optional dependencies* for details. Be aware that *NLopt* does not fail gracefully. If continued use of *NLopt* proves stable enough, its implementation of the Nelder-Mead algorithm might become the default.

## remove_dynamic_background

EBSD.**remove_dynamic_background**(*operation: str = 'subtract'*, *filter_domain: str = 'frequency'*, *std: int | float | None = None*, *truncate: int | float = 4.0*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*, ***kwargs*) → None | *EBSD* | *LazyEBSD*

Remove the dynamic background.

The removal is performed by subtracting or dividing by a Gaussian blurred version of each pattern. Resulting pattern intensities are rescaled to fill the input patterns' data type range individually.

**Parameters**

**operation**
Whether to `"subtract"` (default) or `"divide"` by the dynamic background pattern.

**filter_domain**
Whether to obtain the dynamic background by applying a Gaussian convolution filter in the `"frequency"` (default) or `"spatial"` domain.

**std**
Standard deviation of the Gaussian window. If None (default), it is set to width/8.

**truncate**
Truncate the Gaussian window at this many standard deviations. Default is `4.0`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

****kwargs**
Keyword arguments passed to the Gaussian blurring function determined from `filter_domain`.

**Returns**

**s_out**
Background corrected signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*remove_static_background*
*get_dynamic_background*
*kikuchipy.pattern.remove_dynamic_background*
*kikuchipy.pattern.get_dynamic_background*

### Examples

Remove the static and dynamic background

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s.remove_static_background()
>>> s.remove_dynamic_background(operation="divide", std=5)
```

### Examples using `EBSD.remove_dynamic_background`

- *Dynamic background correction*
- *Neighbour pattern averaging*

### remove_static_background

EBSD.**remove_static_background**(*operation: str = 'subtract'*, *static_bg: ndarray | Array | None = None*, *scale_bg: bool = False*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSD* | *LazyEBSD*

Remove the static background.

The removal is performed by subtracting or dividing by a static background pattern. Resulting pattern intensities are rescaled loosing relative intensities and stretched to fill the available grey levels in the patterns' data type range.

**Parameters**

**operation**
Whether to `"subtract"` (default) or `"divide"` by the static background pattern.

**static_bg**
Static background pattern. If not given, the background is obtained from the `EBSD.static_background` property.

**scale_bg**
Whether to scale the static background pattern to each individual pattern's data range before removal. Default is `False`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Background corrected signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*remove_dynamic_background*

### Examples

It is assumed that a static background pattern of the same shape and data type (e.g. 8-bit unsigned integer, `uint8`) as the patterns is available in signal metadata:

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s.static_background
array([[84, 87, 90, ..., 27, 29, 30],
       [87, 90, 93, ..., 27, 28, 30],
       [92, 94, 97, ..., 39, 28, 29],
       ...,
       [80, 82, 84, ..., 36, 30, 26],
       [79, 80, 82, ..., 28, 26, 26],
       [76, 78, 80, ..., 26, 26, 25]], dtype=uint8)
```

The static background can be removed by subtracting or dividing this background from each pattern:

```
>>> s.remove_static_background(operation="divide")
```

If the `static_background` property is `None`, this must be passed in the `static_bg` parameter as a `numpy` or `dask` array.

### Examples using `EBSD.remove_static_background`

- *Pattern binning*
- *Static background correction*
- *Dynamic background correction*
- *Neighbour pattern averaging*
- *Crop navigation axes*
- *Crop signal axes*
- *Extract patterns from a grid*

## rescale_intensity

EBSD.**rescale_intensity**(*relative: bool = False*, *in_range: Tuple[int, int] | Tuple[float, float] | None = None*, *out_range: Tuple[int, int] | Tuple[float, float] | None = None*, *dtype_out: str | dtype | type | Tuple[int, int] | Tuple[float, float] | None = None*, *percentiles: Tuple[int, int] | Tuple[float, float] | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSD* | *LazyEBSD*

Rescale image intensities.

Output min./max. intensity is determined from `out_range` or the data type range of the `numpy.dtype` passed to `dtype_out` if `out_range` is `None`.

This method is based on `skimage.exposure.rescale_intensity()`.

**Parameters**

**relative**
Whether to keep relative intensities between images (default is `False`). If `True`, `in_range` must be `None`, because `in_range` is in this case set to the global min./max. intensity. Use with care, as this requires the computation of the min./max. intensity of the signal before rescaling.

**in_range**
Min./max. intensity of input images. If not given, `in_range` is set to pattern min./max intensity. Contrast stretching is performed when `in_range` is set to a narrower intensity range than the input patterns. Must be `None` if `relative=True` or `percentiles` are passed.

**out_range**
Min./max. intensity of output images. If not given, `out_range` is set to `dtype_out` min./max according to `skimage.util.dtype.dtype_range`.

**dtype_out**
Data type of rescaled images, default is input images' data type.

**percentiles**
Disregard intensities outside these percentiles. Calculated per image. Must be `None` if `in_range` or `relative` is passed. Default is `None`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Rescaled signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

`skimage.exposure.rescale_intensity()`

### Notes

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
```

Image intensities are stretched to fill the available grey levels in the input images' data type range or any data type range passed to `dtype_out`, either keeping relative intensities between images or not

```
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint8 23 246 26 245
>>> s2 = s.deepcopy()
>>> s.rescale_intensity(dtype_out=np.uint16)
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint16 0 65535 0 65535
>>> s2.rescale_intensity(relative=True)
>>> print(
...     s2.data.dtype, s2.data.min(), s2.data.max(),
...     s2.inav[0, 0].data.min(), s2.inav[0, 0].data.max()
... )
uint8 0 255 3 253
```

Contrast stretching can be performed by passing percentiles

```
>>> s.rescale_intensity(percentiles=(1, 99))
```

Here, the darkest and brightest pixels within the 1% percentile are set to the ends of the data type range, e.g. 0 and 255 respectively for images of `uint8` data type.

### save

EBSD.**save**(*filename: str | None = None*, *overwrite: bool | None = None*, *extension: str | None = None*, ***kwargs*) → None

Write the signal to file in the specified format.

The function gets the format from the extension: `h5`, `hdf5` or `h5ebsd` for kikuchipy's specification of the h5ebsd format, `dat` for the NORDIF binary format or `hspy` for HyperSpy's HDF5 specification. If no extension is provided the signal is written to a file in kikuchipy's h5ebsd format. Each format accepts a different set of parameters.

This method is a modified version of HyperSpy's function `hyperspy.signal.BaseSignal.save()`.

**Parameters**

**filename**
If not given and `tmp_parameters.filename` and `tmp_parameters.folder` in signal metadata are defined, the filename and path will be taken from there. A valid extension can be provided e.g. `"data.h5"`, see `extension`.

**overwrite**
If not given and the file exists, it will query the user. If `True` (`False`) it (does not) overwrite the file if it exists.

**extension**
Extension of the file that defines the file format. Options are `"h5"`, `"hdf5"`, `"h5ebsd"`, `"dat"`, `"hspy"`. `"h5"`, `"hdf5"`, and `"h5ebsd"` are equivalent. If not given, the extension is determined from the following list in this order: i) the filename, ii) `tmp_parameters.extension` or iii) `"h5"` (kikuchipy's h5ebsd format).

****kwargs**
Keyword arguments passed to the writer.

**See also:**

*kikuchipy.io.plugins*

### set_detector_calibration

EBSD.**set_detector_calibration**(*delta: int | float*) → None

Set detector pixel size in microns. The offset is set to the the detector center.

**Parameters**

**delta**
Detector pixel size in microns.

**See also:**

*set_scan_calibration*

### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s.axes_manager['dx'].scale  # Default value
1.0
>>> s.set_detector_calibration(delta=70.)
>>> s.axes_manager['dx'].scale
70.0
```

## set_scan_calibration

EBSD.**set_scan_calibration**(*step_x: int | float = 1.0*, *step_y: int | float = 1.0*) → None

Set the step size in microns.

**Parameters**

**step_x**
Scan step size in um per pixel in horizontal direction.

**step_y**
Scan step size in um per pixel in vertical direction.

**See also:**

*set_detector_calibration*

### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> s.axes_manager['x'].scale
1.5
>>> s.set_scan_calibration(step_x=2)  # Microns
>>> s.axes_manager['x'].scale
2.0
```

## Examples using EBSD

- *Pattern binning*
- *Static background correction*
- *Dynamic background correction*
- *Neighbour pattern averaging*
- *Adaptive histogram equalization*
- *Crop navigation axes*
- *Crop signal axes*
- *Extract patterns from a grid*

### 2.11.3 EBSDMasterPattern

**class** kikuchipy.signals.**EBSDMasterPattern**(**args*, ***kwargs*)

Bases: KikuchiMasterPattern

Simulated Electron Backscatter Diffraction (EBSD) master pattern.

This class extends HyperSpy's Signal2D class for EBSD master patterns.

See the documentation of Signal2D for the list of inherited attributes and methods.

**Parameters**

***args**
See Signal2D.

**hemisphere**
[str] Which hemisphere the data contains, either "upper", "lower", or "both".

**phase**
[Phase] The phase describing the crystal structure used in the master pattern simulation.

**projection**
[str] Which projection the pattern is in, "stereographic" or "lambert".

****kwargs**
See Signal2D.

**See also:**

*kikuchipy.data.nickel_ebsd_master_pattern_small*
A nickel EBSD master pattern dynamically simulated with *EMsoft*.

#### Examples

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_master_pattern_small()
>>> s
<EBSDMasterPattern, title: ni_mc_mp_20kv_uint8_gzip_opts9, dimensions: (|401, 401)>
>>> s.hemisphere
'upper'
>>> s.phase
<name: ni. space group: Fm-3m. point group: m-3m. proper point group: 432. color:␣
↪tab:blue>
>>> s.projection
'stereographic'
```

### Attributes

| | |
|---|---|
| *EBSDMasterPattern.hemisphere* | Return or set which hemisphere(s) the signal contains. |
| *EBSDMasterPattern.phase* | Return or set the phase describing the crystal structure used in the master pattern simulation. |
| *EBSDMasterPattern.projection* | Return or set which projection the pattern is in. |

### hemisphere

**property** EBSDMasterPattern.**hemisphere: str**

Return or set which hemisphere(s) the signal contains.

Options are `"upper"`, `"lower"` or `"both"`.

**Parameters**

**value**
[str] Which hemisphere(s) the signal contains.

### phase

**property** EBSDMasterPattern.**phase: Phase**

Return or set the phase describing the crystal structure used in the master pattern simulation.

**Parameters**

**value**
[Phase] The phase used in the master pattern simulation.

### projection

**property** EBSDMasterPattern.**projection: str**

Return or set which projection the pattern is in.

**Parameters**

**value**
[str] Which projection the pattern is in, either `"stereographic"` or `"lambert"`.

### Methods

| | |
|---|---|
| *EBSDMasterPattern.adaptive_histogram_equalization*([...]) | Enhance the local contrast using adaptive histogram equalization. |
| *EBSDMasterPattern.as_lambert*([show_progressbar]) | Return a new master pattern in the Lambert projection [Callahan and De Graef, 2013]. |
| *EBSDMasterPattern.deepcopy*() | Return a deep copy using `copy.deepcopy()`. |
| *EBSDMasterPattern.get_patterns*(rotations, ...) | Return one or more EBSD patterns projected onto a detector from a master pattern in the square Lambert projection for rotation(s) relative to the EDAX TSL sample reference frame (RD, TD, ND) [Callahan and De Graef, 2013]. |
| *EBSDMasterPattern.normalize_intensity*([...]) | Normalize image intensities to a mean of zero with a given standard deviation. |
| *EBSDMasterPattern.plot_spherical*([energy, ...]) | Plot the master pattern sphere. |
| *EBSDMasterPattern.rescale_intensity*([...]) | Rescale image intensities. |

### adaptive_histogram_equalization

EBSDMasterPattern.**adaptive_histogram_equalization**(*kernel_size: Tuple[int, int] | List[int] | None = None*, *clip_limit: int | float = 0*, *nbins: int = 128*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSDMasterPattern* | *LazyEBSDMasterPattern*

Enhance the local contrast using adaptive histogram equalization.

This method uses `skimage.exposure.equalize_adapthist()`.

**Parameters**

**kernel_size**
Shape of contextual regions for adaptive histogram equalization, default is 1/4 of image height and 1/4 of image width.

**clip_limit**
Clipping limit, normalized between 0 and 1 (higher values give more contrast). Default is `0`.

**nbins**
Number of gray bins for histogram ("data range"), default is `128`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Equalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*rescale_intensity*
*normalize_intensity*

### Notes

It is recommended to perform adaptive histogram equalization only *after* static and dynamic background corrections of EBSD patterns, otherwise some unwanted darkening towards the edges might occur.

The default window size might not fit all pattern sizes, so it may be necessary to search for the optimal window size.

### Examples

Load one pattern from the small nickel dataset, remove the background and perform adaptive histogram equalization. A copy without equalization is kept for comparison.

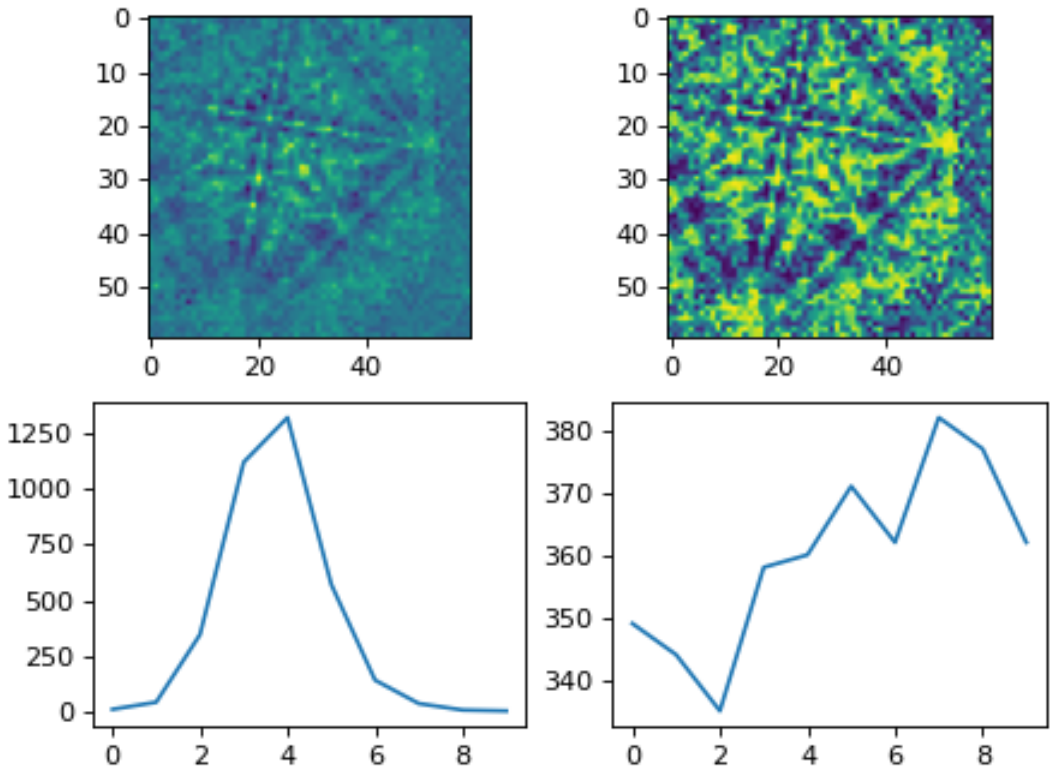
```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small().inav[0, 0]
>>> s.remove_static_background()
>>> s.remove_dynamic_background()
>>> s2 = s.deepcopy()
>>> s2.adaptive_histogram_equalization()
```

Compute the intensity histograms and plot the patterns and histograms

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> hist, _ = np.histogram(s.data, range=(0, 255))
>>> hist2, _ = np.histogram(s2.data, range=(0, 255))
>>> _, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
>>> _ = ax0.imshow(s.data)
>>> _ = ax1.imshow(s2.data)
>>> _ = ax2.plot(hist)
>>> _ = ax3.plot(hist2)
```

### Examples using `EBSDMasterPattern.adaptive_histogram_equalization`

- *Adaptive histogram equalization*


0
10
20
30
40
50
0
20
40
0
10
20
30
40
50
0
20
40
1250
1000
750
500
250
0
0
2
4
6
8
380
370
360
350
340
0
2
4
6
8

### as_lambert

EBSDMasterPattern.**as_lambert**(*show_progressbar: bool | None = None*) → *EBSDMasterPattern*

Return a new master pattern in the Lambert projection [Callahan and De Graef, 2013].

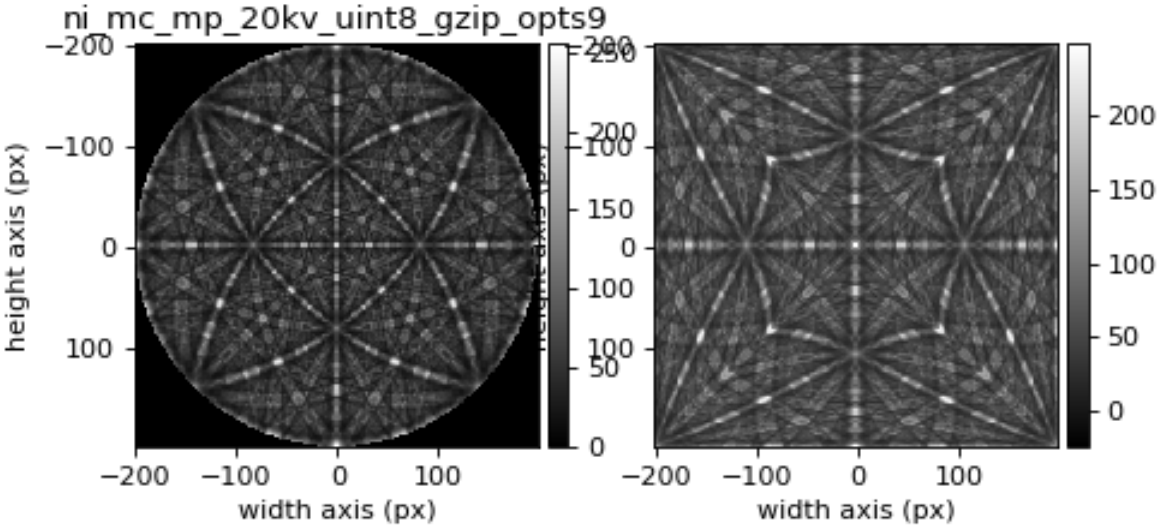
Only implemented for non-lazy signals.

**Returns**

**lambert_master_pattern**
Master pattern in the Lambert projection with the same data shape but in 32-bit floating point data dtype.

### Examples

```
>>> import hyperspy.api as hs
>>> import kikuchipy as kp
>>> mp_sp = kp.data.nickel_ebsd_master_pattern_small()
>>> mp_sp.projection
'stereographic'
>>> mp_lp = mp_sp.as_lambert()
>>> mp_lp.projection
'lambert'
>>> _ = hs.plot.plot_images([mp_sp, mp_lp], per_row=2)
```



### deepcopy

EBSDMasterPattern.**deepcopy**() → *EBSDMasterPattern*

Return a deep copy using `copy.deepcopy()`.

**Returns**

**s_new**
Identical signal without shared memory.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> mp = kp.data.nickel_ebsd_master_pattern_small()
>>> mp2 = mp.deepcopy()
>>> np.may_share_memory(mp.data, mp2.data)
False
```

### get_patterns

EBSDMasterPattern.**get_patterns**(*rotations: Rotation*, *detector:* EBSDDetector, *energy: int | float | None = None*, *dtype_out: str | dtype | type = 'float32'*, *compute: bool = False*, *show_progressbar: bool | None = None*, ***kwargs*) → *EBSD | LazyEBSD*

Return one or more EBSD patterns projected onto a detector from a master pattern in the square Lambert projection for rotation(s) relative to the EDAX TSL sample reference frame (RD, TD, ND) [Callahan and De Graef, 2013].

**Parameters**

**rotations**
Crystal rotations to get patterns from. The shape of this instance, a maximum of two dimensions, determines the navigation shape of the output signal.

**detector**
EBSD detector describing the detector dimensions and the detector-sample geometry with a fixed projection center (PC) or varying PCs, one per rotation. If the detector has multiple PCs, its `navigation_shape` must be identical to the shape of the rotations, `shape`.

**energy**
Acceleration voltage, in kV, used to simulate the desired master pattern to create a dictionary from. If only a single energy is present in the signal, this will be used no matter its energy. If not given, the highest energy will be used.

**dtype_out**
Data type of the returned patterns, by default `"float32"`.

**compute**
Whether to return a lazy result, by default `False`. For more information see `compute()`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

****kwargs**
Keyword arguments passed to *get_chunking()* to control the number of chunks the dictionary creation and the output data array is split into. Only `chunk_shape`, `chunk_bytes` and `dtype_out` (to `dtype`) are passed on.

**Returns**

**out**
Signal with navigation and signal shape equal to the rotation instance and detector shape, respectively.

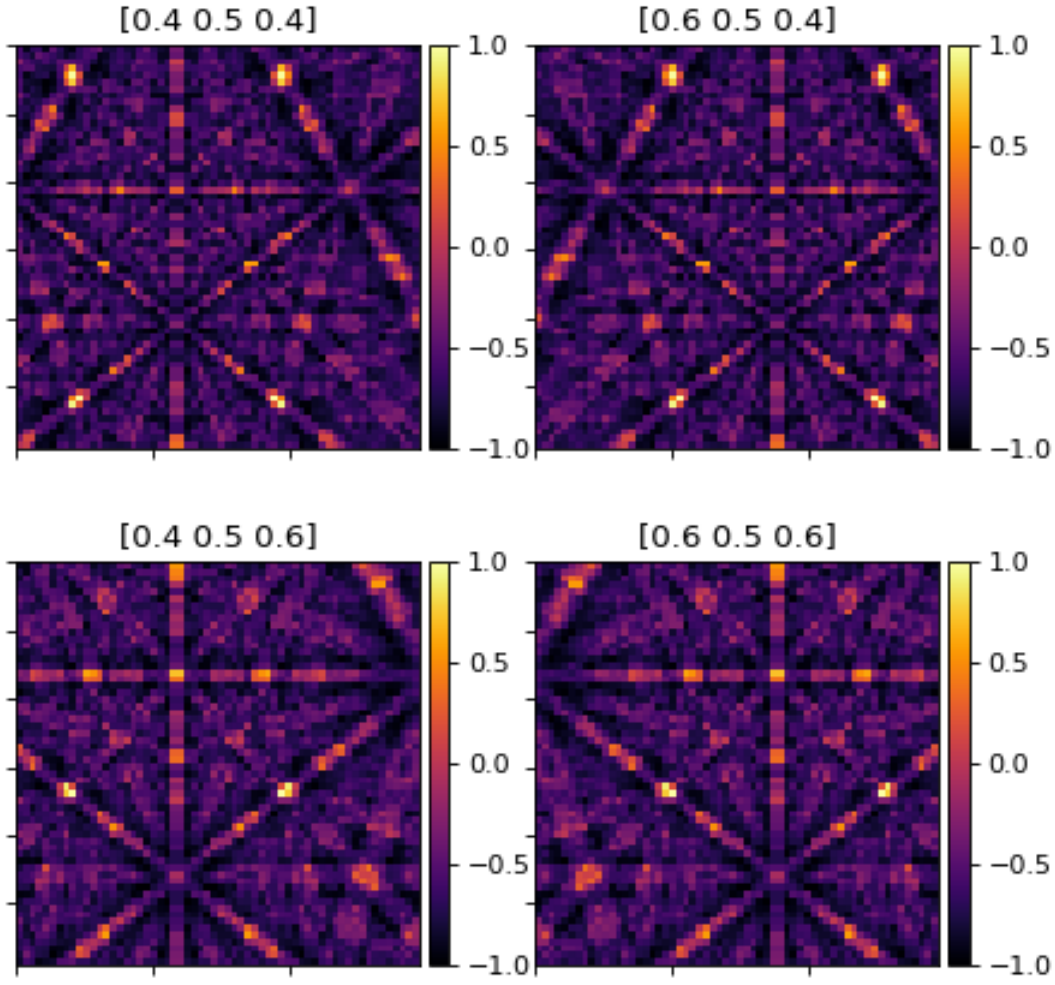
### Notes

If the master pattern *phase* has a non-centrosymmetric point group, both the upper and lower hemispheres must be provided. For more details regarding the reference frame visit the reference frame tutorial.

### Examples

Get patterns for four identity rotations with varying projection centers (PCs), the upper two with PCx increasing towards the right, the lower two with increased PCz compared to the upper two

```
>>> import numpy as np
>>> from orix.quaternion import Rotation
>>> import hyperspy.api as hs
>>> import kikuchipy as kp
>>> mp = kp.data.nickel_ebsd_master_pattern_small(projection="lambert")
>>> det = kp.detectors.EBSDDetector(
...     shape=(60, 60),
...     pc=np.array([
...         [[0.4, 0.5, 0.4], [0.6, 0.5, 0.4]],
...         [[0.4, 0.5, 0.6], [0.6, 0.5, 0.6]],
...     ])
... )
>>> rot = Rotation.identity(det.navigation_shape)
>>> s = mp.get_patterns(rot, det, compute=True, show_progressbar=False)
>>> _ = hs.plot.plot_images(
...     s,
...     per_row=2,
...     cmap="inferno",
...     label=np.array_str(det.pc.reshape((-1, 3)))[1:-1].split("\n "),
...     axes_decor=None,
... )
```


[0.4 0.5 0.4]
[0.6 0.5 0.4]
[0.4 0.5 0.6]
[0.6 0.5 0.6]
1.0
0.5
0.0
−0.5
−1.0

### Examples using `EBSDMasterPattern.get_patterns`

- *Adaptive histogram equalization*

### normalize_intensity

`EBSDMasterPattern.`**`normalize_intensity`**(*num_std: int = 1*, *divide_by_square_root: bool = False*, *dtype_out: str | dtype | type | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSDMasterPattern* | *LazyEBSDMasterPattern*

Normalize image intensities to a mean of zero with a given standard deviation.

**Parameters**

**num_std**
Number of standard deviations of the output intensities. Default is 1.

**divide_by_square_root**
Whether to divide output intensities by the square root of the signal dimension size. Default is `False`.

**dtype_out**
Data type of normalized images. If not given, the input images' data type is used.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Normalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

### Notes

Data type should always be changed to floating point, e.g. `float32` with `change_dtype()`, before normalizing the intensities.

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> np.mean(s.data)
146.0670987654321
>>> s.normalize_intensity(dtype_out=np.float32)
>>> np.mean(s.data)
2.6373216e-08
```

## plot_spherical

EBSDMasterPattern.**plot_spherical**(*energy: int | float | None = None*, *return_figure: bool = False*, *style: str = 'surface'*, *plotter_kwargs: dict = None*, *show_kwargs: dict = None*) → pyvista.Plotter

Plot the master pattern sphere.

This requires the master pattern to be in the stereographic projection and both hemispheres to be present.

**Parameters**

**energy**
Acceleration voltage in kV used to simulate the master pattern to plot. If not given, the highest energy is used.

**return_figure**
Whether to return the `pyvista.Plotter` instance for further modification and then plotting. Default is `False`. If `True`, the figure is not plotted.

**style**
Visualization style of the mesh, either `"surface"` (default), `"wireframe"` or `"points"`. In general, `"surface"` is recommended when zoomed out, while `"points"` is recommended when zoomed in. See `pyvista.Plotter.add_mesh()` for details.

**plotter_kwargs**
Dictionary of keyword arguments passed to `pyvista.Plotter`.

**show_kwargs**
Dictionary of keyword arguments passed to `pyvista.Plotter.show()` if `return_figure=False`.

**Returns**

**pl**
Only returned if `return_figure=True`.

### Notes

Requires `pyvista` (see *the installation guide*).

### Examples

```
>>> import kikuchipy as kp
>>> mp = kp.data.nickel_ebsd_master_pattern_small(projection="stereographic")
>>> mp.plot_spherical()
```

## rescale_intensity

EBSDMasterPattern.**rescale_intensity**(*relative: bool = False*, *in_range: Tuple[int, int] | Tuple[float, float] | None = None*, *out_range: Tuple[int, int] | Tuple[float, float] | None = None*, *dtype_out: str | dtype | type | Tuple[int, int] | Tuple[float, float] | None = None*, *percentiles: Tuple[int, int] | Tuple[float, float] | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *EBSDMasterPattern* | *LazyEBSDMasterPattern*

Rescale image intensities.

Output min./max. intensity is determined from `out_range` or the data type range of the `numpy.dtype` passed to `dtype_out` if `out_range` is `None`.

This method is based on `skimage.exposure.rescale_intensity()`.

**Parameters**

**relative**
Whether to keep relative intensities between images (default is `False`). If `True`, `in_range` must be `None`, because `in_range` is in this case set to the global min./max. intensity. Use with care, as this requires the computation of the min./max. intensity of the signal before rescaling.

**in_range**
Min./max. intensity of input images. If not given, `in_range` is set to pattern min./max intensity. Contrast stretching is performed when `in_range` is set to a narrower intensity range than the input patterns. Must be `None` if `relative=True` or `percentiles` are passed.

**out_range**
Min./max. intensity of output images. If not given, `out_range` is set to `dtype_out` min./max according to `skimage.util.dtype.dtype_range`.

**dtype_out**
Data type of rescaled images, default is input images' data type.

**percentiles**
Disregard intensities outside these percentiles. Calculated per image. Must be `None` if `in_range` or `relative` is passed. Default is `None`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Rescaled signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

`skimage.exposure.rescale_intensity()`

### Notes

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
```

Image intensities are stretched to fill the available grey levels in the input images' data type range or any data type range passed to `dtype_out`, either keeping relative intensities between images or not

```
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint8 23 246 26 245
>>> s2 = s.deepcopy()
>>> s.rescale_intensity(dtype_out=np.uint16)
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint16 0 65535 0 65535
>>> s2.rescale_intensity(relative=True)
>>> print(
...     s2.data.dtype, s2.data.min(), s2.data.max(),
...     s2.inav[0, 0].data.min(), s2.inav[0, 0].data.max()
... )
uint8 0 255 3 253
```

Contrast stretching can be performed by passing percentiles

```
>>> s.rescale_intensity(percentiles=(1, 99))
```

Here, the darkest and brightest pixels within the 1% percentile are set to the ends of the data type range, e.g. 0 and 255 respectively for images of `uint8` data type.

### Examples using `EBSDMasterPattern`

- *Adaptive histogram equalization*
- *Plot nice master pattern image*

## 2.11.4 ECPMasterPattern

**class** kikuchipy.signals.**ECPMasterPattern**(**args*, ***kwargs*)

Bases: `KikuchiMasterPattern`

Simulated Electron Channeling Pattern (ECP) master pattern.

This class extends HyperSpy's Signal2D class for ECP master patterns.

See the documentation of `Signal2D` for the list of inherited attributes and methods.

**Parameters**

***args**
See `Signal2D`.

**hemisphere**
[`str`] Which hemisphere the data contains, either `"upper"`, `"lower"`, or `"both"`.

**phase**
[`Phase`] The phase describing the crystal structure used in the master pattern simulation.

**projection**
[`str`] Which projection the pattern is in, `"stereographic"` or `"lambert"`.

****kwargs**
See `Signal2D`.

### Attributes

| | |
|---|---|
| *ECPMasterPattern.hemisphere* | Return or set which hemisphere(s) the signal contains. |
| *ECPMasterPattern.phase* | Return or set the phase describing the crystal structure used in the master pattern simulation. |
| *ECPMasterPattern.projection* | Return or set which projection the pattern is in. |

### hemisphere

**property** ECPMasterPattern.**hemisphere: str**

Return or set which hemisphere(s) the signal contains.

Options are `"upper"`, `"lower"` or `"both"`.

**Parameters**

**value**
[`str`] Which hemisphere(s) the signal contains.

### phase

**property** ECPMasterPattern.**phase: Phase**

Return or set the phase describing the crystal structure used in the master pattern simulation.

**Parameters**

**value**
[Phase] The phase used in the master pattern simulation.

### projection

**property** ECPMasterPattern.**projection: str**

Return or set which projection the pattern is in.

**Parameters**

**value**
[str] Which projection the pattern is in, either "stereographic" or "lambert".

### Methods

| | |
|---|---|
| *ECPMasterPattern.adaptive_histogram_equalization*([...]) | Enhance the local contrast using adaptive histogram equalization. |
| *ECPMasterPattern.as_lambert*([show_progressbar]) | Return a new master pattern in the Lambert projection [Callahan and De Graef, 2013]. |
| *ECPMasterPattern.deepcopy*() | Return a deep copy using copy.deepcopy(). |
| *ECPMasterPattern.normalize_intensity*([...]) | Normalize image intensities to a mean of zero with a given standard deviation. |
| *ECPMasterPattern.plot_spherical*([energy, ...]) | Plot the master pattern sphere. |
| *ECPMasterPattern.rescale_intensity*([...]) | Rescale image intensities. |

### adaptive_histogram_equalization

ECPMasterPattern.**adaptive_histogram_equalization**(*kernel_size: Tuple[int, int] | List[int] | None = None*, *clip_limit: int | float = 0*, *nbins: int = 128*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *ECPMasterPattern* | *LazyECPMasterPattern*

Enhance the local contrast using adaptive histogram equalization.

This method uses skimage.exposure.equalize_adapthist().

**Parameters**

**kernel_size**
Shape of contextual regions for adaptive histogram equalization, default is 1/4 of image height and 1/4 of image width.

**clip_limit**
Clipping limit, normalized between 0 and 1 (higher values give more contrast). Default is 0.

**nbins**
Number of gray bins for histogram ("data range"), default is `128`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Equalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*rescale_intensity*
*normalize_intensity*

### Notes

It is recommended to perform adaptive histogram equalization only *after* static and dynamic background corrections of EBSD patterns, otherwise some unwanted darkening towards the edges might occur.

The default window size might not fit all pattern sizes, so it may be necessary to search for the optimal window size.

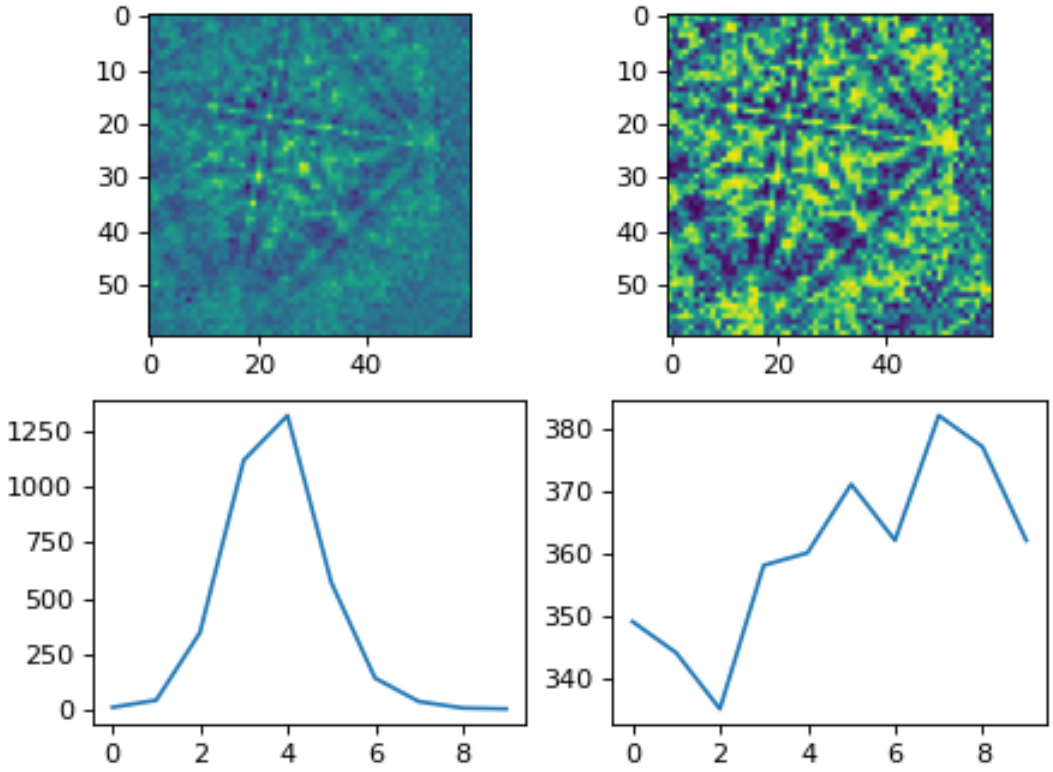
### Examples

Load one pattern from the small nickel dataset, remove the background and perform adaptive histogram equalization. A copy without equalization is kept for comparison.

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small().inav[0, 0]
>>> s.remove_static_background()
>>> s.remove_dynamic_background()
>>> s2 = s.deepcopy()
>>> s2.adaptive_histogram_equalization()
```

Compute the intensity histograms and plot the patterns and histograms

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> hist, _ = np.histogram(s.data, range=(0, 255))
>>> hist2, _ = np.histogram(s2.data, range=(0, 255))
>>> _, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
>>> _ = ax0.imshow(s.data)
>>> _ = ax1.imshow(s2.data)
>>> _ = ax2.plot(hist)
>>> _ = ax3.plot(hist2)
```


0
10
20
30
40
50
0
20
40
0
10
20
30
40
50
0
20
40
1250
1000
750
500
250
0
0
2
4
6
8
380
370
360
350
340
0
2
4
6
8

### as_lambert

ECPMasterPattern.**as_lambert**(*show_progressbar: bool | None = None*) → *ECPMasterPattern*

Return a new master pattern in the Lambert projection [Callahan and De Graef, 2013].

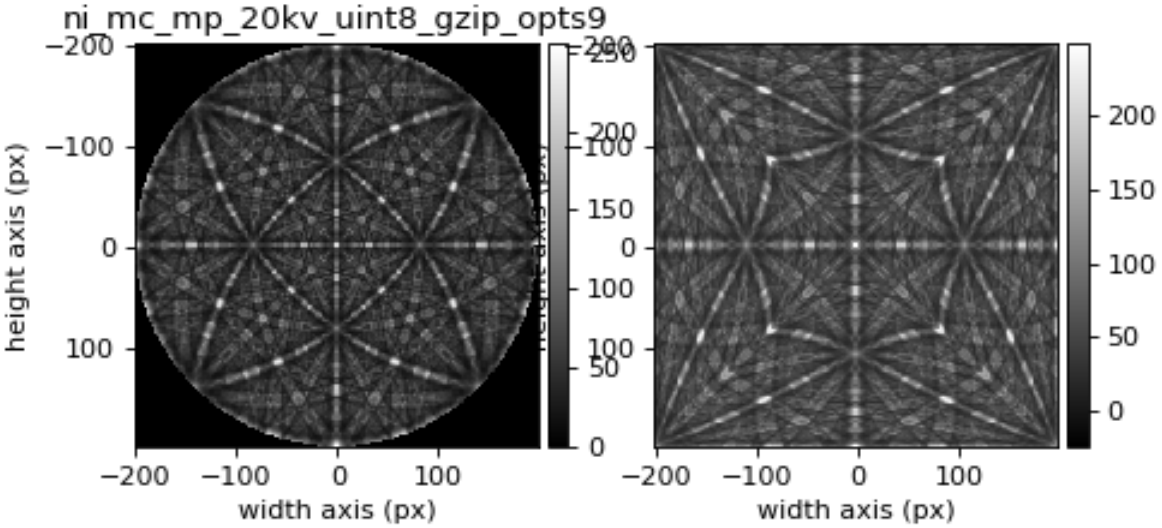
Only implemented for non-lazy signals.

**Returns**

**lambert_master_pattern**
Master pattern in the Lambert projection with the same data shape but in 32-bit floating point data dtype.

### Examples

```
>>> import hyperspy.api as hs
>>> import kikuchipy as kp
>>> mp_sp = kp.data.nickel_ebsd_master_pattern_small()
>>> mp_sp.projection
'stereographic'
>>> mp_lp = mp_sp.as_lambert()
>>> mp_lp.projection
'lambert'
>>> _ = hs.plot.plot_images([mp_sp, mp_lp], per_row=2)
```



### deepcopy

ECPMasterPattern.**deepcopy**() → *ECPMasterPattern*

Return a deep copy using `copy.deepcopy()`.

**Returns**

**s_new**
Identical signal without shared memory.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> mp = kp.data.nickel_ebsd_master_pattern_small()
>>> mp2 = mp.deepcopy()
>>> np.may_share_memory(mp.data, mp2.data)
False
```

### normalize_intensity

ECPMasterPattern.**normalize_intensity**(*num_std: int = 1*, *divide_by_square_root: bool = False*, *dtype_out: str | dtype | type | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *ECPMasterPattern* | *LazyECPMasterPattern*

Normalize image intensities to a mean of zero with a given standard deviation.

**Parameters**

**num_std**
Number of standard deviations of the output intensities. Default is `1`.

**divide_by_square_root**
Whether to divide output intensities by the square root of the signal dimension size. Default is `False`.

**dtype_out**
Data type of normalized images. If not given, the input images' data type is used.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Normalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

### Notes

Data type should always be changed to floating point, e.g. `float32` with `change_dtype()`, before normalizing the intensities.

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> np.mean(s.data)
146.0670987654321
>>> s.normalize_intensity(dtype_out=np.float32)
>>> np.mean(s.data)
2.6373216e-08
```

## plot_spherical

ECPMasterPattern.**plot_spherical**(*energy: int | float | None = None*, *return_figure: bool = False*, *style: str = 'surface'*, *plotter_kwargs: dict = None*, *show_kwargs: dict = None*) → pyvista.Plotter

Plot the master pattern sphere.

This requires the master pattern to be in the stereographic projection and both hemispheres to be present.

**Parameters**

**energy**
Acceleration voltage in kV used to simulate the master pattern to plot. If not given, the highest energy is used.

**return_figure**
Whether to return the `pyvista.Plotter` instance for further modification and then plotting. Default is `False`. If `True`, the figure is not plotted.

**style**
Visualization style of the mesh, either `"surface"` (default), `"wireframe"` or `"points"`. In general, `"surface"` is recommended when zoomed out, while `"points"` is recommended when zoomed in. See `pyvista.Plotter.add_mesh()` for details.

**plotter_kwargs**
Dictionary of keyword arguments passed to `pyvista.Plotter`.

**show_kwargs**
Dictionary of keyword arguments passed to `pyvista.Plotter.show()` if `return_figure=False`.

**Returns**

**pl**
Only returned if `return_figure=True`.

### Notes

Requires `pyvista` (see *the installation guide*).

### Examples

```
>>> import kikuchipy as kp
>>> mp = kp.data.nickel_ebsd_master_pattern_small(projection="stereographic")
>>> mp.plot_spherical()
```

## rescale_intensity

ECPMasterPattern.**rescale_intensity**(*relative: bool = False*, *in_range: Tuple[int, int] | Tuple[float, float] | None = None*, *out_range: Tuple[int, int] | Tuple[float, float] | None = None*, *dtype_out: str | dtype | type | Tuple[int, int] | Tuple[float, float] | None = None*, *percentiles: Tuple[int, int] | Tuple[float, float] | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *ECPMasterPattern* | *LazyECPMasterPattern*

Rescale image intensities.

Output min./max. intensity is determined from `out_range` or the data type range of the `numpy.dtype` passed to `dtype_out` if `out_range` is `None`.

This method is based on `skimage.exposure.rescale_intensity()`.

**Parameters**

**relative**
Whether to keep relative intensities between images (default is `False`). If `True`, `in_range` must be `None`, because `in_range` is in this case set to the global min./max. intensity. Use with care, as this requires the computation of the min./max. intensity of the signal before rescaling.

**in_range**
Min./max. intensity of input images. If not given, `in_range` is set to pattern min./max intensity. Contrast stretching is performed when `in_range` is set to a narrower intensity range than the input patterns. Must be `None` if `relative=True` or `percentiles` are passed.

**out_range**
Min./max. intensity of output images. If not given, `out_range` is set to `dtype_out` min./max according to `skimage.util.dtype.dtype_range`.

**dtype_out**
Data type of rescaled images, default is input images' data type.

**percentiles**
Disregard intensities outside these percentiles. Calculated per image. Must be `None` if `in_range` or `relative` is passed. Default is `None`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Rescaled signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

`skimage.exposure.rescale_intensity()`

### Notes

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
```

Image intensities are stretched to fill the available grey levels in the input images' data type range or any data type range passed to `dtype_out`, either keeping relative intensities between images or not

```
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint8 23 246 26 245
>>> s2 = s.deepcopy()
>>> s.rescale_intensity(dtype_out=np.uint16)
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint16 0 65535 0 65535
>>> s2.rescale_intensity(relative=True)
>>> print(
...     s2.data.dtype, s2.data.min(), s2.data.max(),
...     s2.inav[0, 0].data.min(), s2.inav[0, 0].data.max()
... )
uint8 0 255 3 253
```

Contrast stretching can be performed by passing percentiles

```
>>> s.rescale_intensity(percentiles=(1, 99))
```

Here, the darkest and brightest pixels within the 1% percentile are set to the ends of the data type range, e.g. 0 and 255 respectively for images of `uint8` data type.

### 2.11.5 LazyEBSD

**class** kikuchipy.signals.**LazyEBSD**(**args*, ***kwargs*)

Bases: LazyKikuchipySignal2D, *EBSD*

Lazy implementation of *EBSD*.

See the documentation of EBSD for attributes and methods.

This class extends HyperSpy's LazySignal2D class for EBSD patterns. See the documentation of that class for how to create this signal and the list of inherited attributes and methods.

#### Attributes

#### Methods

| | |
|---|---|
| *LazyEBSD.compute*(*args, **kwargs) | Attempt to store the full signal in memory. |
| *LazyEBSD.get_decomposition_model_write*([...] | Write the model signal generated from the selected number of principal components directly to an .hspy file. |

#### compute

LazyEBSD.**compute**(**args*, ***kwargs*) → None

Attempt to store the full signal in memory.

This docstring was copied from HyperSpy's LazySignal2D.compute. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**close_file**
[bool, default False] If True, attemp to close the file associated with the dask array data if any. Note that closing the file will make all other associated lazy signals inoperative.

**show_progressbar**
[None or bool] If True, display a progress bar. If None, the default from the preferences settings is used.

**Returns**

None

#### get_decomposition_model_write

LazyEBSD.**get_decomposition_model_write**(*components: int | List[int] | None = None*, *dtype_learn: str | dtype | type = 'float32'*, *mbytes_chunk: int = 100*, *dir_out: str | None = None*, *fname_out: str | None = None*) → None

Write the model signal generated from the selected number of principal components directly to an `.hspy` file.

The model signal intensities are rescaled to the original signals' data type range, keeping relative intensities.

**Parameters**

**components**
If not given, rebuilds the signal from all `components`. If `int`, rebuilds signal from `components` in range 0-given `int`. If list of `int`, rebuilds signal from only `components` in given list.

**dtype_learn**
Data type to set learning results to (default is `"float32"`) before multiplication.

**mbytes_chunk**
Size of learning results chunks in MB, default is 100 MB as suggested in the Dask documentation.

**dir_out**
Directory to place output signal in.

**fname_out**
Name of output signal file.

#### Notes

Multiplying the learning results' factors and loadings in memory to create the model signal cannot sometimes be done due to too large matrices. Here, instead, learning results are written to file, read into dask arrays and multiplied using `dask.array.matmul()`, out of core.

#### Examples using `LazyEBSD`

- *Extract patterns from a grid*

### 2.11.6 LazyEBSDMasterPattern

**class** kikuchipy.signals.**LazyEBSDMasterPattern**(**args*, ***kwargs*)

Bases: `LazyKikuchipySignal2D`, *EBSDMasterPattern*

Lazy implementation of *EBSDMasterPattern*.

See the documentation of `EBSDMasterPattern` for attributes and methods.

This class extends HyperSpy's `LazySignal2D` class for EBSD master patterns. See the documentation of that class for how to create this signal and the list of inherited attributes and methods.

#### Attributes

#### Methods

| | |
|---|---|
| *LazyEBSDMasterPattern.compute*(*args, **kwargs) | Attempt to store the full signal in memory. |

#### compute

LazyEBSDMasterPattern.**compute**(**args*, ***kwargs*) → None

Attempt to store the full signal in memory.

This docstring was copied from HyperSpy's LazySignal2D.compute. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**close_file**
[bool, `default` `False`] If True, attemp to close the file associated with the dask array data if any. Note that closing the file will make all other associated lazy signals inoperative.

**show_progressbar**
[`None` or bool] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

**Returns**

`None`

### 2.11.7 LazyECPMasterPattern

**class** kikuchipy.signals.**LazyECPMasterPattern**(**args*, ***kwargs*)

Bases: `LazyKikuchipySignal2D`, *ECPMasterPattern*

Lazy implementation of *ECPMasterPattern*.

See the documentation of `ECPMasterPattern` for attributes and methods.

This class extends HyperSpy's `LazySignal2D` class for ECP master patterns. See the documentation of that class for how to create this signal and the list of inherited attributes and methods.

#### Attributes

#### Methods

| | |
|---|---|
| *LazyECPMasterPattern.compute*(*args, **kwargs) | Attempt to store the full signal in memory. |

#### compute

LazyECPMasterPattern.**compute**(**args*, ***kwargs*) → None

Attempt to store the full signal in memory.

This docstring was copied from HyperSpy's LazySignal2D.compute. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**close_file**
[bool, `default False`] If True, attemp to close the file associated with the dask array data if any. Note that closing the file will make all other associated lazy signals inoperative.

**show_progressbar**
[None or bool] If `True`, display a progress bar. If `None`, the default from the preferences settings is used.

**Returns**

`None`

### 2.11.8 LazyVirtualBSEImage

**class** kikuchipy.signals.**LazyVirtualBSEImage**(**args*, ***kwargs*)

Bases: `LazyKikuchipySignal2D`, *VirtualBSEImage*

Lazy implementation of *VirtualBSEImage*.

See the documentation of `VirtualBSEImage` for attributes and methods.

This class extends HyperSpy's `LazySignal2D` class for EBSD master patterns. See the documentation of that class for how to create this signal and the list of inherited attributes and methods.

#### Attributes

#### Methods

| | |
|---|---|
| *LazyVirtualBSEImage.compute*(*args, **kwargs) | Attempt to store the full signal in memory. |

#### compute

LazyVirtualBSEImage.**compute**(**args*, ***kwargs*) → None

Attempt to store the full signal in memory.

This docstring was copied from HyperSpy's LazySignal2D.compute. Some inconsistencies with the kikuchipy version may exist.

**Parameters**

**close_file**
[bool, default False] If True, attemp to close the file associated with the dask array data if any. Note that closing the file will make all other associated lazy signals inoperative.

**show_progressbar**
[None or bool] If True, display a progress bar. If None, the default from the preferences settings is used.

**Returns**

None

### 2.11.9 VirtualBSEImage

**class** kikuchipy.signals.**VirtualBSEImage**(**args*, ***kwargs*)

Bases: KikuchipySignal2D

Virtual backscatter electron (BSE) image(s).

This class extends HyperSpy's Signal2D class for virtual BSE images.

See the docstring of Signal2D for a list of attributes and methods.

#### Attributes

#### Methods

| | |
|---|---|
| *VirtualBSEImage.adaptive_histogram_equalization*([...]) | Enhance the local contrast using adaptive histogram equalization. |
| *VirtualBSEImage.normalize_intensity*([...]) | Normalize image intensities to a mean of zero with a given standard deviation. |
| *VirtualBSEImage.rescale_intensity*([...]) | Rescale image intensities. |

### adaptive_histogram_equalization

VirtualBSEImage.**adaptive_histogram_equalization**(*kernel_size: Tuple[int, int] | List[int] | None = None*, *clip_limit: int | float = 0*, *nbins: int = 128*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *VirtualBSEImage* | *LazyVirtualBSEImage*

Enhance the local contrast using adaptive histogram equalization.

This method uses `skimage.exposure.equalize_adapthist()`.

**Parameters**

**kernel_size**
Shape of contextual regions for adaptive histogram equalization, default is 1/4 of image height and 1/4 of image width.

**clip_limit**
Clipping limit, normalized between 0 and 1 (higher values give more contrast). Default is `0`.

**nbins**
Number of gray bins for histogram ("data range"), default is `128`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Equalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

*rescale_intensity*
*normalize_intensity*

#### Notes

It is recommended to perform adaptive histogram equalization only *after* static and dynamic background corrections of EBSD patterns, otherwise some unwanted darkening towards the edges might occur.

The default window size might not fit all pattern sizes, so it may be necessary to search for the optimal window size.

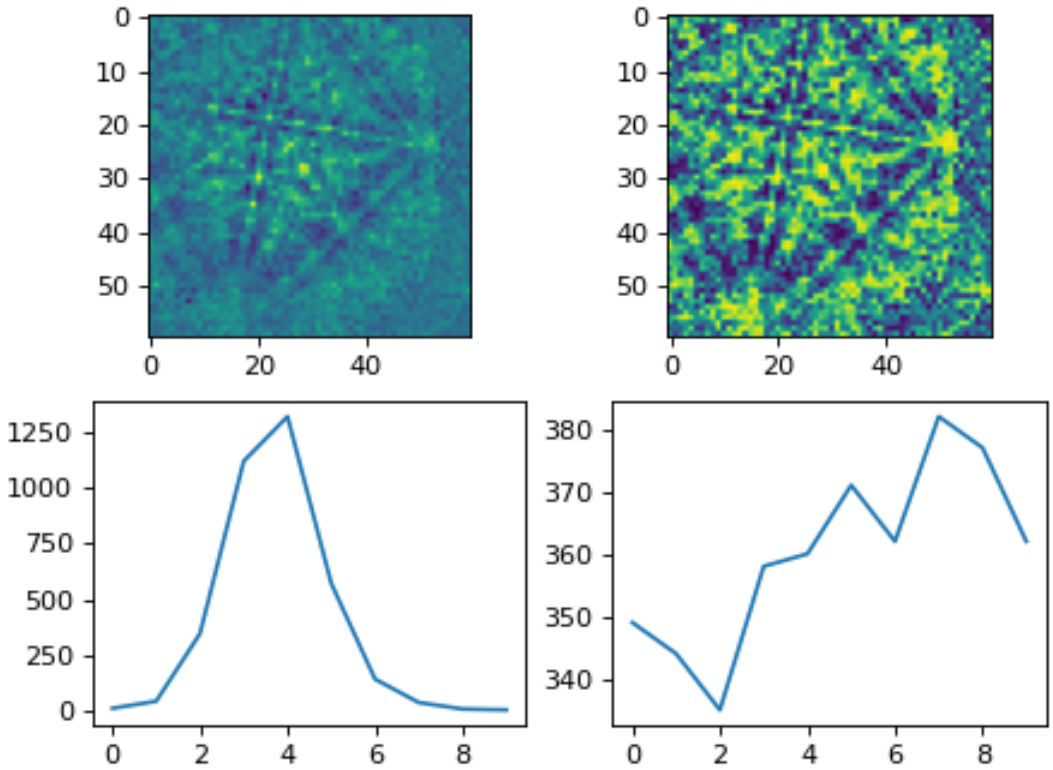
### Examples

Load one pattern from the small nickel dataset, remove the background and perform adaptive histogram equalization. A copy without equalization is kept for comparison.

```
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small().inav[0, 0]
>>> s.remove_static_background()
>>> s.remove_dynamic_background()
>>> s2 = s.deepcopy()
>>> s2.adaptive_histogram_equalization()
```

Compute the intensity histograms and plot the patterns and histograms

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> hist, _ = np.histogram(s.data, range=(0, 255))
>>> hist2, _ = np.histogram(s2.data, range=(0, 255))
>>> _, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
>>> _ = ax0.imshow(s.data)
>>> _ = ax1.imshow(s2.data)
>>> _ = ax2.plot(hist)
>>> _ = ax3.plot(hist2)
```



### normalize_intensity

VirtualBSEImage.**normalize_intensity**(*num_std: int = 1*, *divide_by_square_root: bool = False*, *dtype_out: str | dtype | type | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *VirtualBSEImage* | *LazyVirtualBSEImage*

Normalize image intensities to a mean of zero with a given standard deviation.

**Parameters**

**num_std**
Number of standard deviations of the output intensities. Default is 1.

**divide_by_square_root**
Whether to divide output intensities by the square root of the signal dimension size. Default is `False`.

**dtype_out**
Data type of normalized images. If not given, the input images' data type is used.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Normalized signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

#### Notes

Data type should always be changed to floating point, e.g. `float32` with `change_dtype()`, before normalizing the intensities.

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

#### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
>>> np.mean(s.data)
146.0670987654321
>>> s.normalize_intensity(dtype_out=np.float32)
>>> np.mean(s.data)
2.6373216e-08
```

### rescale_intensity

VirtualBSEImage.**rescale_intensity**(*relative: bool = False*, *in_range: Tuple[int, int] | Tuple[float, float] | None = None*, *out_range: Tuple[int, int] | Tuple[float, float] | None = None*, *dtype_out: str | dtype | type | Tuple[int, int] | Tuple[float, float] | None = None*, *percentiles: Tuple[int, int] | Tuple[float, float] | None = None*, *show_progressbar: bool | None = None*, *inplace: bool = True*, *lazy_output: bool | None = None*) → None | *VirtualBSEImage* | *LazyVirtualBSEImage*

Rescale image intensities.

Output min./max. intensity is determined from `out_range` or the data type range of the `numpy.dtype` passed to `dtype_out` if `out_range` is `None`.

This method is based on `skimage.exposure.rescale_intensity()`.

**Parameters**

**relative**
Whether to keep relative intensities between images (default is `False`). If `True`, `in_range` must be `None`, because `in_range` is in this case set to the global min./max. intensity. Use with care, as this requires the computation of the min./max. intensity of the signal before rescaling.

**in_range**
Min./max. intensity of input images. If not given, `in_range` is set to pattern min./max intensity. Contrast stretching is performed when `in_range` is set to a narrower intensity range than the input patterns. Must be `None` if `relative=True` or `percentiles` are passed.

**out_range**
Min./max. intensity of output images. If not given, `out_range` is set to `dtype_out` min./max according to `skimage.util.dtype.dtype_range`.

**dtype_out**
Data type of rescaled images, default is input images' data type.

**percentiles**
Disregard intensities outside these percentiles. Calculated per image. Must be `None` if `in_range` or `relative` is passed. Default is `None`.

**show_progressbar**
Whether to show a progressbar. If not given, the value of `hyperspy.api.preferences.General.show_progressbar` is used.

**inplace**
Whether to operate on the current signal or return a new one. Default is `True`.

**lazy_output**
Whether the returned signal is lazy. If not given this follows from the current signal. Can only be `True` if `inplace=False`.

**Returns**

**s_out**
Rescaled signal, returned if `inplace=False`. Whether it is lazy is determined from `lazy_output`.

**See also:**

`skimage.exposure.rescale_intensity()`

### Notes

Rescaling RGB images is not possible. Use RGB channel normalization when creating the image instead.

### Examples

```
>>> import numpy as np
>>> import kikuchipy as kp
>>> s = kp.data.nickel_ebsd_small()
```

Image intensities are stretched to fill the available grey levels in the input images' data type range or any data type range passed to `dtype_out`, either keeping relative intensities between images or not

```
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint8 23 246 26 245
>>> s2 = s.deepcopy()
>>> s.rescale_intensity(dtype_out=np.uint16)
>>> print(
...     s.data.dtype, s.data.min(), s.data.max(),
...     s.inav[0, 0].data.min(), s.inav[0, 0].data.max()
... )
uint16 0 65535 0 65535
>>> s2.rescale_intensity(relative=True)
>>> print(
...     s2.data.dtype, s2.data.min(), s2.data.max(),
...     s2.inav[0, 0].data.min(), s2.inav[0, 0].data.max()
... )
uint8 0 255 3 253
```

Contrast stretching can be performed by passing percentiles

```
>>> s.rescale_intensity(percentiles=(1, 99))
```

Here, the darkest and brightest pixels within the 1% percentile are set to the ends of the data type range, e.g. 0 and 255 respectively for images of `uint8` data type.

#### Examples using `VirtualBSEImage.rescale_intensity`

- *Extract patterns from a grid*

### Examples using `VirtualBSEImage`

- *Extract patterns from a grid*

## 2.12 simulations

Simulations returned by a generator and handling of Kikuchi bands and zone axes.

### Classes

| | |
|---|---|
| *GeometricalKikuchiPatternSimulation*(...) | Collection of coordinates of Kikuchi lines and zone axes on an EBSD detector for simple plotting or creation of HyperSpy markers to plot onto *EBSD* signals. |
| *KikuchiPatternSimulator*(reflectors) | Setup and calculation of geometrical or kinematical Kikuchi pattern simulations. |

### 2.12.1 GeometricalKikuchiPatternSimulation

**class** kikuchipy.simulations.**GeometricalKikuchiPatternSimulation**(*detector:* EBSDDetector, *rotations:* *Rotation*, *reflectors:* *ReciprocalLatticeVector*, *lines:* *KikuchiPatternLine*, *zone_axes:* *KikuchiPatternZoneAxis*)

Bases: `object`

Collection of coordinates of Kikuchi lines and zone axes on an EBSD detector for simple plotting or creation of HyperSpy markers to plot onto *EBSD* signals.

Instances of this class are returned from *kikuchipy.simulations.KikuchiPatternSimulator.on_detector()*, and *not* ment to be created directly.

**Parameters**

**detector**
EBSD detector.

**rotations**
Crystal orientations for which coordinates of Kikuchi lines and zone axes have been generated.

**reflectors**
Reciprocal lattice vectors used in the simulation.

**lines**
Collection of coordinates of Kikuchi lines on the detector.

**zone_axes**
Collection of coordinates of zone axes on the detector.

### Attributes

| | |
|---|---|
| *GeometricalKikuchiPatternSimulation.detector* | Return the EBSD detector onto which simulations were generated. |
| *GeometricalKikuchiPatternSimulation.navigation_shape* | Return the navigation shape of the simulations, equal to the shape of *rotations*. |
| *GeometricalKikuchiPatternSimulation.reflectors* | Return the reciprocal lattice vectors used in the simulations. |
| *GeometricalKikuchiPatternSimulation.rotations* | Return the crystal orientations for which simulations were generated. |

#### detector

**property** GeometricalKikuchiPatternSimulation.**detector: *EBSDDetector***

Return the EBSD detector onto which simulations were generated.

#### navigation_shape

**property** GeometricalKikuchiPatternSimulation.**navigation_shape: tuple**

Return the navigation shape of the simulations, equal to the shape of *rotations*.

#### reflectors

**property** GeometricalKikuchiPatternSimulation.**reflectors: ReciprocalLatticeVector**

Return the reciprocal lattice vectors used in the simulations.

#### rotations

**property** GeometricalKikuchiPatternSimulation.**rotations: Rotation**

Return the crystal orientations for which simulations were generated.

### Methods

| | |
|---|---|
| *GeometricalKikuchiPatternSimulation.as_collections*([...]) | Get a single simulation as a list of Matplotlib objects. |
| *GeometricalKikuchiPatternSimulation.as_markers*([...]) | Return a list of simulation markers. |
| *GeometricalKikuchiPatternSimulation.lines_coordinates*([...]) | Get Kikuchi line coordinates of a single simulation. |
| *GeometricalKikuchiPatternSimulation.plot*([...]) | Plot a single simulation on the detector. |
| *GeometricalKikuchiPatternSimulation.zone_axes_coordinates*([...]) | Get zone axis coordinates of a single simulation. |

### as_collections

GeometricalKikuchiPatternSimulation.**as_collections**(*index: int | tuple | None = None*, *coordinates: str = 'detector'*, *lines: bool = True*, *zone_axes: bool = False*, *zone_axes_labels: bool = False*, *lines_kwargs: dict = None*, *zone_axes_kwargs: dict = None*, *zone_axes_labels_kwargs: dict = None*) → list

Get a single simulation as a list of Matplotlib objects.

**Parameters**

**index**
Index of the simulation to get collections from. If not given, this is the first simulation.

**coordinates**
The coordinates of the plot axes, either `"detector"` (default) or `"gnomonic"`.

**lines**
Whether to get the collection of Kikuchi lines. Default is `True`. These are returned as `matplotlib.collections.LineCollection`.

**zone_axes**
Whether to get the collection of zone axes. Default is `False`. These are returned as `matplotlib.collections.PathCollection`.

**zone_axes_labels**
Whether to get the collection of zone axes labels. Default is `False`. These are returned as a class:*list* of `matplotlib.text.Text`.

**lines_kwargs**
Keyword arguments passed to `matplotlib.collections.LineCollection` to format Kikuchi lines if `lines=True`.

**zone_axes_kwargs**
Keyword arguments passed to `matplotlib.collections.PathCollection` to format zone axes if `zone_axes=True`.

**zone_axes_labels_kwargs**
Keyword arguments passed to `matplotlib.text.Text` to format zone axes labels if `zone_axes_labels=True`.

**Returns**

**collection_list**
List of Matplotlib collections.

**See also:**

*as_markers*, *plot*

### as_markers

GeometricalKikuchiPatternSimulation.**as_markers**(*lines: bool = True*, *zone_axes: bool = False*, *zone_axes_labels: bool = False*, *pc: bool = False*, *lines_kwargs: dict | None = None*, *zone_axes_kwargs: dict | None = None*, *zone_axes_labels_kwargs: dict | None = None*, *pc_kwargs: dict | None = None*) → List[MarkerBase]

Return a list of simulation markers.

**Parameters**

**lines**
Whether to return Kikuchi line markers. Default is `True`.

**zone_axes**
Whether to return zone axes markers. Default is `False`.

**zone_axes_labels**
Whether to return zone axes label markers. Default is `False`.

**pc**
Whether to return projection center (PC) markers. Default is `False`.

**lines_kwargs**
Keyword arguments passed to `axvline()` to format the lines.

**zone_axes_kwargs**
Keyword arguments passed to `scatter()` to format the markers.

**zone_axes_labels_kwargs**
Keyword arguments passed to `Text()` to format the labels.

**pc_kwargs**
Keyword arguments passed to `scatter()` to format the markers.

**Returns**

**markers**
List with all markers.

**See also:**

*as_collections*, *plot*

### lines_coordinates

GeometricalKikuchiPatternSimulation.**lines_coordinates**(*index: int | tuple | None = None*, *coordinates: str = 'detector'*, *exclude_nan: bool = True*) → ndarray

Get Kikuchi line coordinates of a single simulation.

**Parameters**

**index**
Index of the simulation to get line coordinates for. If not given, this is the first simulation.

**coordinates**
The type of coordinates, either `"detector"` (default) or `"gnomonic"`.

**exclude_nan**
Whether to exclude coordinates of Kikuchi lines not present in the pattern. Default is `True`. By passing `False`, all simulations (by varying `index`) returns an array of the same shape.

**Returns**

**coords**
Kikuchi line coordinates.

**See also:**

*zone_axes_coordinates*

### plot

`GeometricalKikuchiPatternSimulation.plot`(*index: int | tuple | None = None*, *coordinates: str = 'detector'*, *pattern: ndarray | None = None*, *lines: bool = True*, *zone_axes: bool = True*, *zone_axes_labels: bool = True*, *pc: bool = True*, *pattern_kwargs: dict | None = None*, *lines_kwargs: dict | None = None*, *zone_axes_kwargs: dict | None = None*, *zone_axes_labels_kwargs: dict | None = None*, *pc_kwargs: dict | None = None*, *return_figure: bool = False*) → Figure

Plot a single simulation on the detector.

**Parameters**

**index**
Index of the simulation to plot. If not given, this is the first simulation. If *navigation_shape* is 2D, and `index` is passed, it must be a 2-tuple.

**coordinates**
The coordinates of the plot axes, either `"detector"` (default) or `"gnomonic"`.

**pattern**
A pattern to plot the simulation onto. If not given, the simulation is plotted on a gray background.

**lines**
Whether to show Kikuchi lines. Default is `True`.

**zone_axes**
Whether to show zone axes. Default is `True`.

**zone_axes_labels**
Whether to show zone axes labels. Default is `True`.

**pc**
Whether to show the projection/pattern centre (PC). Default is `True`.

**pattern_kwargs**
Keyword arguments passed to `matplotlib.axes.Axes.imshow()` if `pattern` is given.

**lines_kwargs**
Keyword arguments passed to `matplotlib.collections.LineCollection` to format Kikuchi lines if `lines=True`.

**zone_axes_kwargs**
Keyword arguments passed to `matplotlib.collections.PathCollection` to format zone axes if `zone_axes=True`.

**zone_axes_labels_kwargs**
Keyword arguments passed to `matplotlib.text.Text` to format zone axes labels if `zone_axes_labels=True`.

**pc_kwargs**
Keyword arguments passed to `matplotlib.axes.Axes.scatter()` to format the PC if `pc=True`.

**return_figure**
Whether to return the figure. Default is `False`.

**Returns**

**fig**
Returned if `return_figure=True`.

**See also:**

*as_collections*, *as_markers*

### zone_axes_coordinates

GeometricalKikuchiPatternSimulation.**zone_axes_coordinates**(*index: int | tuple | None = None*, *coordinates: str = 'detector'*, *exclude_nan: bool = True*) → ndarray

Get zone axis coordinates of a single simulation.

**Parameters**

**index**
Index of the simulation to get zone axis coordinates for. If not given, this is the first simulation.

**coordinates**
The type of coordinates, either `"detector"` (default) or `"gnomonic"`.

**exclude_nan**
Whether to exclude coordinates of zone axes not present in the pattern. Default is `True`. By passing `False`, all simulations (by varying `index`) returns an array of the same shape.

**Returns**

**coords**
Zone axis coordinates.

**See also:**

*lines_coordinates*

### 2.12.2 KikuchiPatternSimulator

**class** kikuchipy.simulations.**KikuchiPatternSimulator**(*reflectors: ReciprocalLatticeVector*)

Bases: `object`

Setup and calculation of geometrical or kinematical Kikuchi pattern simulations.

**Parameters**

**reflectors**
Reflectors to use in the simulation, flattened to one navigation dimension.

#### Attributes

| | |
|---|---|
| *KikuchiPatternSimulator.phase* | Return the phase with unit cell and symmetry. |
| *KikuchiPatternSimulator.reflectors* | Return the reflectors to use in the simulation. |

#### phase

**property** KikuchiPatternSimulator.**phase: Phase**

Return the phase with unit cell and symmetry.

#### reflectors

**property** KikuchiPatternSimulator.**reflectors: ReciprocalLatticeVector**

Return the reflectors to use in the simulation.

#### Methods

| | |
|---|---|
| *KikuchiPatternSimulator.calculate_master_pattern*([...]) | Calculate a kinematical master pattern in the stereographic projection. |
| *KikuchiPatternSimulator.on_detector*(...) | Project Kikuchi lines and zone axes onto a detector, one per crystal orientation. |
| *KikuchiPatternSimulator.plot*([projection, ...]) | Plot reflectors as lines or bands in the stereographic or spherical projection. |

#### calculate_master_pattern

KikuchiPatternSimulator.**calculate_master_pattern**(*half_size: int | None = 500*, *hemisphere: str | None = 'upper'*, *scaling: str | None = 'linear'*) → *EBSDMasterPattern*

Calculate a kinematical master pattern in the stereographic projection.

Requires that the *reflectors* have structure factors (`structure_factor`) and Bragg angles (`theta`) calculated.

**Parameters**

**half_size**
Number of pixels along the x-direction of the square master pattern. Default is `500`. The full size will be `2 * half_size + 1`, given a master pattern of shape `(1001, 1001)` for the default value.

**hemisphere**
Which hemisphere(s) to calculate. Options are `"upper"` (default), `"lower"` or `"both"`.

**scaling**
Intensity scaling of the band kinematical intensities, either `"linear"` (default), $|F|$, `"square"`, $|F|^2$, or `"None"`, giving all bands an intensity of 1.

**Returns**

**master_pattern**
Kinematical master pattern in the stereographic projection.

### Notes

The algorithm for selecting which unit vector is within a Kikuchi band is derived from a similar routine in EMsoft.

## on_detector

KikuchiPatternSimulator.**on_detector**(*detector:* EBSDDetector, *rotations:* *Rotation*) → *GeometricalKikuchiPatternSimulation*

Project Kikuchi lines and zone axes onto a detector, one per crystal orientation.

**Parameters**

**detector**
EBSD detector describing the detector's view of the sample. If *navigation_shape* is anything else than `(1,)`, it must be equal to the shape of the input `rotations`.

**rotations**
Crystal orientations assumed to be expressed with respect to the default EDAX TSL sample reference frame RD-TD-ND in the Bunge convention. Rotation shape can be 1D or 2D.

**Returns**

**simulations**
Geometrical Kikuchi pattern simulation.

### Notes

This function is not optimized for large datasets, so use with care.

### plot

KikuchiPatternSimulator.**plot**(*projection: str | None = 'stereographic'*, *mode: str | None = 'lines'*, *hemisphere: str | None = 'upper'*, *scaling: str | None = 'linear'*, *figure: None | Figure | pyvista.Plotter = None*, *return_figure: bool = False*, *backend: str = 'matplotlib'*, *show_plotter: bool = True*, *color: str = 'k'*, ***kwargs*) → Figure | pyvista.Plotter

Plot reflectors as lines or bands in the stereographic or spherical projection.

**Parameters**

**projection**
Either "stereographic" (default) or "spherical".

**mode**
Either "lines" (default) or "bands". The latter option requires that *reflectors* have Bragg angles (theta) calculated.

**hemisphere**
Which hemisphere to plot when projection="stereographic". Options are "upper" (default), "lower" or "both". Ignored if figure is given.

**scaling**
Intensity scaling of the band kinematical intensities, either "linear" (default), $|F|$, "square", $|F|^2$, or None, giving all bands the same intensity. The intensity range is [0, highest intensity].

**figure**
An existing Figure or Plotter to add the reflectors to. If not given, a new figure is created.

**return_figure**
Whether to return the figure. Default is False. This is a Figure if backend=="matplotlib" or a Plotter if backend=="pyvista".

**backend**
Which plotting library to use when projection="spherical", either "matplotlib" (default) or "pyvista". The latter option requires that PyVista is installed.

**show_plotter**
Whether to show the Plotter when projection="spherical" and backend="pyvista". Default is True.

**color**
Color to give reflectors. Either a string signifying a valid Matplotlib color or "phase", which then uses the color_rgb of the *phase*. Default is black ("k"). Only used with Matplotlib.

****kwargs**
Keyword arguments passed to draw_circle().

**Returns**

**figure**
If return_figure=True, a Figure or a Plotter is returned.

CHAPTER

THREE

# CONTRIBUTING

kikuchipy is a community maintained project. We welcome contributions in the form of bug reports, documentation, code, feature requests, and more. The source code is hosted on GitHub. These guidelines provide resources on how best to contribute.

---

**Tip:** This guide can look intimidating to people who want to contribute, but have limited experience with tools like `git`, `pytest`, and `sphinx`. The shortest route to start contributing is to create a GitHub account and explain what you want to do in an issue.

That said, our contributing workflow is typical for Python projects, so reading this guide can make contributing to similar projects in the future much smoother!

---

This project follows the all-contributors specification.

kikuchipy has a *Code of conduct* that should be honoured by everyone who participates in the kikuchipy community.

Have a question, comment, suggestion for improvements, or any other inquiries regarding the project? Feel free to ask a question, open an issue or make a pull request in our GitHub repository. We also have a Gitter chat.

## 3.1 Setting up a development installation

You need a fork of the repository in order to make changes to kikuchipy.

Make a local copy of your forked repository and change directories:

```
git clone https://github.com/your-username/kikuchipy.git
cd kikuchipy
```

Set the `upstream` remote to the main kikuchipy repository:

```
git remote add upstream https://github.com/pyxem/kikuchipy.git
```

We recommend installing in a conda environment with the Miniconda distribution:

```
conda create --name kp-dev
conda activate kp-dev
```

Then, install the required dependencies while making the development version available globally (in the `conda` environment):

```
pip install --editable .[dev]
```

This installs all necessary development dependencies, including those for running tests and building documentation.

## 3.2 Code style

The code making up kikuchipy is formatted closely following the Style Guide for Python Code with The Black Code style. We use pre-commit to run `black` automatically prior to each local commit. Please install it in your environment:

```
pre-commit install
```

Next time you commit some code, your code will be formatted inplace according to `black`.

`black` can format Jupyter notebooks as well. Code lines in tutorial notebooks should be limited to 77 characters to display nicely in the documentation:

```
black -l 77 <your_nice_notebook>.ipynb
```

Note that `black` won't format docstrings. We follow the numpydoc standard (with some exceptions), and the docstrings are checked against this standard when building the documentation.

Comment and docstring lines should preferably be limited to 72 characters (including leading whitespaces).

Package imports should be structured into three blocks with blank lines between them (descending order): standard library (like `os` and `typing`), third party packages (like `numpy` and `hyperspy`) and finally kikuchipy imports.

We use type hints in the function definition without type duplication in the function docstring, for example:

```
def my_function(a: int, b: Optional[bool] = None) -> Tuple[float, np.ndarray]:
    """This is a new function.

    Parameters
    ----------
    a
        Explanation of ``a``.
    b
        Explanation of flag ``b``. Default is ``None``.

    Returns
    -------
    values
        Explanation of returned values.
    """
```

We import modules lazily using the specification in PEP 562.

## 3.3 Using Git

### 3.3.1 Making changes

If you want to add a new feature, branch off of the `develop` branch, and when you want to fix a bug, branch off of `main` instead.

To create a new feature branch that tracks the upstream development branch:

```
git checkout develop -b your-awesome-feature-name upstream/develop
```

When you've made some changes you can view them with:

```
git status
```

Add and commit your created, modified or deleted files:

```
git add my-file-or-directory
git commit -s -m "An explanatory commit message"
```

The `-s` makes sure that you sign your commit with your GitHub-registered email as the author. You can set this up following this GitHub guide.

### 3.3.2 Keeping your branch up-to-date

If you are adding a new feature, make sure to merge `develop` into your feature branch. If you are fixing a bug, merge `main` into your bug fix branch instead.

To update a feature branch, switch to the `develop` branch:

```
git checkout develop
```

Fetch changes from the upstream branch and update `develop`:

```
git pull upstream develop --tags
```

Update your feature branch:

```
git checkout your-awesome-feature-name
git merge develop
```

### 3.3.3 Sharing your changes

Update your remote branch:

```
git push -u origin your-awesome-feature-name
```

You can then make a pull request to kikuchipy's `develop` branch for new features and `main` branch for bug fixes. Good job!

## 3.4 Building and writing documentation

The documentation contains three categories of documents: `examples`, `tutorials`, and the `reference`. The documentation strategy is based on the Diátaxis Framework. New documents should fit into one of these categories.

We use Sphinx for documenting functionality. Install necessary dependencies to build the documentation:

```
pip install --editable .[doc]
```

**Note:** The tutorials and examples require some small datasets to be downloaded via the *kikuchipy.data* module upon building the documentation. See the section on the *data module* for more details.

Then, build the documentation from the doc directory:

```
cd doc
make html
```

The documentation's HTML pages are built in the doc/build/html directory from files in the reStructuredText (reST) plaintext markup language. They should be accessible in the browser by typing file:///your/absolute/path/to/kikuchipy/doc/build/html/index.html in the address bar.

We can link to other documentation in reStructuredText files using Intersphinx. Which links are available from a package's documentation can be obtained like so:

```
python -m sphinx.ext.intersphinx https://hyperspy.org/hyperspy-doc/current/objects.inv
```

We use Sphinx-Gallery to build the Projects using Sphinx. The examples are located in the top source directory examples/, and a new directory doc/examples/ is created when the docs are built.

We use nbsphinx for converting notebooks into tutorials displayed in the documentation. Code lines in notebooks should be *formatted with black*.

### 3.4.1 Writing tutorial notebooks

Here are some tips for writing tutorial notebooks:

- All notebooks should have a Markdown cell with this message at the top, "This notebook is part of the kikuchipy documentation https://kikuchipy.org. Links to the documentation won't work from the notebook.", and have "nbsphinx": "hidden" in the cell metadata so that the message is not visible when displayed in the documentation.
- Use _ = ax[0].imshow(...) to silence matplotlib output if a matplotlib command is the last line in a cell.
- Refer to our API reference with this general Markdown [fft_filter()](../reference/generated/kikuchipy.signals.EBSD.fft_filter.rst). Remember to add the parentheses () to functions and methods.
- Reference sections in other tutorial notebooks using this general Markdown [image quality](feature_maps.ipynb#image-quality).
- Reference external APIs via standard Markdown like [Signal2D](https://hyperspy.org/hyperspy-doc/current/api/hyperspy._signals.signal2d.html).
- The Sphinx gallery thumbnail used for a notebook is set by adding the nbsphinx-thumbnail tag to a code cell with an image output. The notebook must be added to the appropriate topic in doc/tutorials/index.rst to be included in the documentation pages.
- pydata_sphinx_theme displays the documentation in a light or dark theme, depending on the browser/OS setting. It is important to make sure the documentation is readable with both themes. This means explicitly printing the signal axes manager, like print(s.axes_manager), and displaying all figures with a white background for axes labels and ticks and figure titles etc. to be readable.
- Whenever the documentation is built (locally or on the Read the Docs server), nbsphinx only runs the notebooks *without* any cell output stored. It is recommended that notebooks are stored without cell output, so that functionality within them are run and tested to ensure continued compatibility with code changes. Cell output should only

be stored in notebooks which are too computationally intensive for the Read the Docs server to handle, which has a limit of 15 minutes and 3 GB of memory per documentation build.

- We also use `black` to format notebooks cells, see the page on *Code style* for details. To prevent `black` from automatically formatting regions of your code, please wrap these code blocks with the following:

```
# fmt: off
python_code_block = not_to_be_formatted
# fmt: on
```

  Please see the black documentation for more details.

- Displaying interactive 3D plots with PyVista requires a Jupyter backend. We previously used `panel`, which PyVista does not support anymore. Instead, they recommend using `trame`, but this does not work with `nbsphinx` yet. Thus, the previously interactive 3D plots are now static. The Jupyter backend used by PyVista can be set in a notebook cell at the top of the notebook via `pyvista.set_jupyter_backend("static")`.

In general, we run all notebooks every time the documentation is built with Sphinx, to ensure that all notebooks are compatible with the current API at all times. This is important! For computationally expensive notebooks however, we store the cell outputs so the documentation doesn't take too long to build, either by us locally or the Read The Docs GitHub action. To check that the notebooks with stored cell outputs are compatible with the current API, we run a scheduled GitHub Action every Monday morning which checks that the notebooks run OK and that they produce the same output now as when they were last executed. We use nbval for this.

The tutorial notebooks can be run interactively in the browser with the help of Binder. When creating a server from the kikuchipy source code, Binder installs the packages listed in the environment.yml configuration file, which must include all doc dependencies listed in setup.py necessary to run the notebooks.

### 3.4.2 Writing API reference

Inherited attributes and methods are not listed in the API reference unless they are explicitly coded in the inheriting class. To see an example of this behavior, see the source code of *EBSDMasterPattern*, which inherits attributes and methods from a private class `KikuchiMasterPattern`.

## 3.5 Handling deprecations

We attempt to adhere to semantic versioning as best we can. This means that as little, ideally no, functionality should break between minor releases. Deprecation warnings are raised whenever possible and feasible for functions/methods/properties/arguments, so that users get a heads-up one (minor) release before something is removed or changes, with a possible alternative to be used.

The decorator should be placed right above the object signature to be deprecated:

```
@deprecate(since=0.8, removal=0.9, alternative="bar")
def foo(self, n: int) -> int:
    return n + 1

@property
@deprecate(
    since=0.9, removal=0.10, alternative="another", alternative_is_function=False
)
def this_property(self) -> int:
    return 2
```

Parameters can be deprecated as well:

```
@deprecate_argument(name="n", since=0.8, removal=0.9, alternative="m")
def foo(self, n: Optional[int] = None, m: int: Optional[int] = None) -> int:
    if m is None:
        m = n
    return m + 1
```

This will raise a warning if `n` is passed.

## 3.6 Running and writing tests

All functionality in kikuchipy is tested via the pytest framework. The tests reside in a `tests/` directory within each module. Tests are short methods that call functions in kikuchipy and compare resulting output values with known answers. Install necessary dependencies to run the tests:

```
pip install --editable .[tests]
```

Some useful fixtures, like a dummy EBSD scan and the corresponding background pattern, are available in the `conftest.py` file.

---

**Note:** Some *`kikuchipy.data`* module tests check that data not part of the package distribution can be downloaded from the https://github.com/pyxem/kikuchipy-data GitHub repository, thus downloading some datasets of ~15 MB to your local cache.

This also means that running tests require an internet connection.

---

To run the tests:

```
pytest --cov --pyargs kikuchipy
```

The `--cov` flag makes coverage.py print a nice report in the terminal. For an even nicer presentation, you can use `coverage.py` directly:

```
coverage html
```

Then, you can open the created `htmlcov/index.html` in the browser and inspect the coverage in more detail.

We can run tests in parallel on four CPUs using pytest-xdist:

```
pytest -n 4
```

To run only a specific test function or class, e.g. the `TestEBSD` class:

```
pytest -k TestEBSD
```

This is useful when we only want to run a specific test and not the full test suite, e.g. when we're creating or updating a test. We have to remember to run the full test suite before pushing, though!

We can automatically rerun so-called flaky tests (tests yielding both passing and failing results without code changes) using the pytest plugin pytest-rerunfailures:

```
pytest --reruns 2
```

Docstring examples are tested with pytest as well. If you're in the top directory you can run:

```
pytest --doctest-modules --ignore-glob=kikuchipy/*/tests kikuchipy/*.py
```

### 3.6.1 Functionality using Numba

Tips for writing tests of Numba decorated functions:

- A Numba decorated function `numba_func()` is only covered if it is called in the test as `numba_func.py_func()`.
- Always test a Numba decorated function calling `numba_func()` directly, in addition to `numba_func.py_func()`, because the machine code function might give different results on different OS with the same Python code. See this issue https://github.com/pyxem/kikuchipy/issues/496 for a case where this happened.

### 3.6.2 Functionality using multiprocessing

Some functionality may run in parallel using `multiprocessing`, such as `pyebsdindex.pcopt.optimize_pso()` which is used in `hough_indexing_optimize_pc()`. A test of this functionality may hang when run in a parallel test run using `pytest-xdist`. To ensure the multiprocessing-part only runs when pytest-xdist is not used, we can ensure that the value of the `worker_id` fixture provided by pytest-xdist is `"master"`.

## 3.7 Adding data to the data module

Example datasets used in the documentation and tests are included in the *`kikuchipy.data`* module via the pooch Python library. These are listed in a file registry (`kikuchipy.data._registry.py`) with their file verification string (hash, MD5, obtain with e.g. `md5sum <file>`) and location, the latter potentially not within the package but from the kikuchipy-data repository or elsewhere, since some files are considered too large to include in the package.

If a required dataset isn't in the package, but is in the registry, it can be downloaded from the repository when the user passes `allow_download=True` to e.g. *`nickel_ebsd_large()`*. The dataset is then downloaded to a local cache, in the location returned from `pooch.os_cache("kikuchipy")`. The location can be set with a global *KIKUCHIPY_DATA_DIR* variable locally, e.g. by setting `export KIKUCHIPY_DATA_DIR=~/kikuchipy_data` in `~/.bashrc`. Pooch handles downloading, caching, version control, file verification (against hash) etc. of files not included in the package. If we have updated the file hash, pooch will re-download it. If the file is available in the cache, it can be loaded as the other files in the data module.

With every new version of kikuchipy, a new directory of datasets with the version name is added to the cache directory. Any old directories are not deleted automatically, and should then be deleted manually if desired.

## 3.8 Improving performance

When we write code, it's important that we (1) get the correct result, (2) don't fill up memory, and (3) that the computation doesn't take too long. To keep memory in check, we should use Dask wherever possible. To speed up computations, we should use Numba wherever possible.

To check whether a change is an improvement or a regression, a benchmark should be written. These are stored in the top directory `kikuchipy/benchmarks`. Benchmarks are run using pytest-benchmark:

```
pytest --benchmark-only
```

## 3.9 Continuous integration (CI)

We use GitHub Actions to ensure that kikuchipy can be installed on Windows, macOS and Linux (Ubuntu). After a successful installation of the package, the CI server runs the tests. After the tests return no errors, code coverage is reported to Coveralls. Add `"[skip ci]"` to a commit message to skip this workflow on any commit to a pull request.

## 3.10 Maintaining package credits

Whenever we get a new contributor, they should be added to the package credits. Unless they do not want to, of course. We maintain three separate sources for the list of contributors:

- `kikuchipy/release.py`: Package metadata used by PyPI and other places
- `.zenodo.json`: Zenodo entry
- All-contributors table in the README

In the package metadata and the Zenodo entry, the initial commiter is listed first, with the others sorted by line contributions.

The All-contributors table in the README is updated locally using their command line interface (see their web page for the docs).

## 3.11 kikuchipy Code of Conduct

### 3.11.1 Introduction

This code of conduct applies to all spaces managed by the kikuchipy project, including all public and private mailing lists, issue trackers, wikis, blogs, and any other communication channel used by our community. The kikuchipy project does not organise in-person events, however events related to our community should have a code of conduct similar in spirit to this one.

This code of conduct should be honoured by everyone who participates in the kikuchipy community formally or informally, or claims any affiliation with the project, in any project-related activities and especially when representing the project, in any role.

This code is not exhaustive or complete. It serves to distill our common understanding of a collaborative, shared environment and goals. Please try to follow this code in spirit as much as in letter, to create a friendly and productive environment that enriches the surrounding community.

### 3.11.2 Specific guidelines

We strive to:

1. Be open. We invite anyone to participate in our community. We prefer to use public methods of communication for project-related messages, unless discussing something sensitive. This applies to messages for help or project-related support, too; not only is a public-support request much more likely to result in an answer to a question, it also ensures that any inadvertent mistakes in answering are more easily detected and corrected.
2. Be empathetic, welcoming, friendly, and patient. We work together to resolve conflict, and assume good intentions. We may all experience some frustration from time to time, but we do not allow frustration to turn into a personal attack. A community where people feel uncomfortable or threatened is not a productive one.

3. Be collaborative. Our work will be used by other people, and in turn we will depend on the work of others. When we make something for the benefit of the project, we are willing to explain to others how it works, so that they can build on the work to make it even better. Any decision we make will affect users and colleagues, and we take those consequences seriously when making decisions.
4. Be inquisitive. Nobody knows everything! Asking questions early avoids many problems later, so we encourage questions, although we may direct them to the appropriate forum. We will try hard to be responsive and helpful.
5. Be careful in the words that we choose. We are careful and respectful in our communication and we take responsibility for our own speech. Be kind others. Do not insult or put down other participants. We will not accept harassment or other exclusionary behaviour, such as:
    - Violent threats or language directed against another person.
    - Sexist, racist, ableist, or otherwise discriminatory jokes and language.
    - Posting sexually explicit or violent material.
    - Posting (or threatening to post) other people's personally identifying information ("doxing").
    - Sharing private content, such as emails sent privately or non-publicly, or unlogged forums such as IRC channel history, without the sender's consent.
    - Personal insults, especially those using racist, sexist, or ableist terms.
    - Intentional or repeated misgendering of participants who have explicitly requested to be addressed by specific pronouns.
    - Unwelcome sexual attention.
    - Excessive profanity. Please avoid swearwords; people differ greatly in their sensitivity to swearing.
    - Repeated harassment of others. In general, if someone asks you to stop, then stop.
    - Advocating for, or encouraging, any of the above behaviour.

### 3.11.3 Diversity statement

The kikuchipy project welcomes and encourages participation by everyone. We are committed to being a community that everyone enjoys being part of. Although we may not always be able to accommodate each individual's preferences, we try our best to treat everyone kindly.

No matter how you identify yourself or how others perceive you: we welcome you. Though no list can hope to be comprehensive, we explicitly honour diversity in: age, culture, ethnicity, genotype, gender identity or expression, language, national origin, neurotype, phenotype, political beliefs, profession, race, religion, sexual orientation, socioeconomic status, subculture and technical ability, to the extent that these do not conflict with this code of conduct.

Though we welcome people fluent in all languages, kikuchipy development is conducted in English.

Standards for behaviour in the kikuchipy community are detailed in the Code of Conduct above. Participants in our community should uphold these standards in all their interactions and help others to do so as well (see next section).

### 3.11.4 Reporting guidelines

We know that it is painfully common for internet communication to start at or devolve into obvious and flagrant abuse. We also recognize that sometimes people may have had a bad day, or be unaware of some of the guidelines in this Code of Conduct. Please keep this in mind when deciding how to respond to a breach of this Code.

For clearly intentional breaches, report those to the Code of Conduct Committee (see below). For possibly unintentional breaches, you may reply to the person and point out this Code of Conduct (either in public or in private, whatever is most appropriate). If you would prefer not to do that, please feel free to report to the code of conduct committee directly, or ask the committee for advice, in confidence.

You can report issues to the kikuchipy Code of Conduct Committee, at kikuchipy-conduct@googlegroups.com. Currently, the committee consists of:

- Håkon Wiik Ånes (chair)
- Tina Bergh

### 3.11.5 Incident reporting resolution & Code of Conduct enforcement

*This section summarizes the most important points, more details can be found in* report_handling_manual.

We will investigate and respond to all complaints. The kikuchipy Code of Conduct Committee will protect the identity of the reporter, and treat the content of complaints as confidential (unless the reporter agrees otherwise).

In case of severe and obvious breaches, e.g. personal threat or violent, sexist or racist language, we will immediately disconnect the originator from kikuchipy communication channels; please see the manual for details.

In cases not involving clear severe and obvious breaches of this code of conduct, the process for acting on any received code of conduct violation report will be:

1. acknowledge report is received
2. reasonable discussion/feedback
3. mediation (if feedback didn't help, and only if both reporter and reportee agree to this)
4. enforcement via transparent decision (see resolutions) by the Code of Conduct Committee

The committee will respond to any report as soon as possible, and at most within 72 hours.

### 3.11.6 Endnotes

We are thankful to the groups behind the following documents, from which we drew content and inspiration:

- napari Code of Conduct
- NumPy Code of Conduct

CHAPTER

FOUR

# CHANGELOG

kikuchipy is a library for processing, simulating and indexing of electron backscatter diffraction (EBSD) patterns in Python, built on the tools for multi-dimensional data analysis provided by the HyperSpy library: https://kikuchipy.org.

All user facing changes to this project are documented in this file. The format is based on Keep a Changelog, and this project tries its best to adhere to Semantic Versioning.

List entries are sorted in descending chronological order. Contributors to each release were listed in alphabetical order by first name until version 0.7.0.

## 4.1 Unreleased

### 4.1.1 Added

### 4.1.2 Changed

### 4.1.3 Deprecated

### 4.1.4 Removed

### 4.1.5 Fixed

## 4.2 0.9.0 (2023-11-03)

### 4.2.1 Added

- Explicit support for Python 3.11. (#646)
- Allow Hough indexing of all Laue groups with PyEBSDIndex v0.2 (not just *m-3m*, i.e. FCC and BCC). (#652)
- Control of reflector lists in Hough indexing. One reflector list per phase in the phase list can be passed to `EBSDDetector.get_indexer()` to obtain an `EBSDIndexer` for use in `EBSD.hough_indexing()`. (#652)
- Allow passing keyword arguments to `EBSD.hough_indexing_optimize_pc()` to control the new particle swarm optimization algorithm in PyEBSDIndex v0.2. (#652)
- Allow getting one projection center (PC) per pattern when optimizing PCs using the new particle swarm optimization in PyEBSDIndex v0.2 (passing `batch=True`). (#652)

### 4.2.2 Changed

- Parameter `zone_axes_kwargs` in `GeometricalKikuchiPatternSimulation.as_collections()` does not accept `color` internally to set the default color to white anymore. It accepts `fc` (facecolor) instead. This change was necessary to improve handling of other keyword arguments. (#643)
- Increase minimal versions of diffsims, NumPy, Matplotlib, and PyEBSDIndex to 0.5.1, 1.21.6, 3.5, and 0.2, respectively. (#646, #652)
- Remove dependency on Panel for documentation, and with that the interactive 3D visualization of master patterns in the documentation. The plan is to reintroduce the interactive plots with trame later on. (#652)
- Restrict HyperSpy to below the forthcoming version 2. The plan is to remove this restriction once kikuchipy is compatible with this version. (#657)

### 4.2.3 Removed

- `generators` and `projections` modules which were deprecated in version 0.8. (#612)
- The deprecated PyPI selector `viz` is removed. (#643)
- The data module functions `silicon_ebsd_moving_screen_x()`, where "x" is "in", "out5mm" or "out10mm". They were deprecated in version 0.8. (#656)

### 4.2.4 Fixed

- Conversion from EDAX TSL projection center (PC) convention for (PCy, PCz) for rectangular detectors is corrected. (#652)
- Downloading files in the `data` module to the local cache on Windows. (#655)

## 4.3 0.8.7 (2023-07-24)

### 4.3.1 Fixed

- Passing a 3-component PC array with more than one dimension to `EBSD.hough_indexing_optimize_pc()` works. (#647)

## 4.4 0.8.6 (2023-05-29)

### 4.4.1 Changed

- Use memory mapping (`numpy.memmap()`) instead of reading into memory (`numpy.fromfile()`) for non-lazy reading of EBSD patterns from EDAX binary .up1/2 files. (#641)

### 4.4.2 Fixed

- EBSD patterns from some EDAX binary .up1/2 files were incorrectly read due to an incorrect file offset, making the patterns appear shifted horizontally. (#641)
- Reading of EBSD patterns from H5OINA files with the "Camera Binning Mode" dataset not containing the detector binning. (#641)

## 4.5 0.8.5 (2023-05-21)

### 4.5.1 Fixed

- Not-indexed points in crystal maps are handled correctly when merging. (#639)

## 4.6 0.8.4 (2023-04-07)

### 4.6.1 Fixed

- Points considered not-indexed in a crystal map are maintained after EBSD refinement. (#632)

### 4.6.2 Changed

- EBSD detector returned from combined EBSD and projection center (PC) refinement now has PC values equal to the number of indexed points, accounting for points not being in the data, navigation mask *and* points considered as not-indexed. This means that it might not have a 2D navigation shape, even though the returned crystal map has. (#632)

## 4.7 0.8.3 (2023-03-23)

### 4.7.1 Changed

- `EBSD.hough_indexing()` info message now informs that the given projection center is in Bruker's convention. (#628)

## 4.8 0.8.2 (2023-03-14)

### 4.8.1 Changed

- Set minimal version of orix to >= 0.11.1. (#623)

## 4.9 0.8.1 (2023-02-20)

### 4.9.1 Fixed

- Hough indexing with PyEBSDIndex of a lazy EBSD signal requires not only PyOpenCL to be installed, but also for PyOpenCL to be able to create a context. (#615)
- Missing progressbars for `EBSD` methods `average_neighbour_patterns()` and `fft_filter()` reintroduced. (#615)

## 4.10 0.8.0 (2023-02-11)

### 4.10.1 Added

- `kikuchipy.imaging.VirtualBSEImager` replaces the `kikuchipy.generators.VirtualBSEGenerator` class. (#608)
- Adaptive histogram equalization is available to all signals. (#606)
- Option to return a new signal (lazy or not) instead of operating inplace is added to many methods in all classes via `inplace` and `lazy_output` boolean parameters. (#605)
- Lazy version of the `VirtualBSEImage` class. (#605)
- Allow providing a color for simulator reflections when plotting with Matplotlib. (#599)
- Passing pseudo-symmetry operators to orientation and orientation/PC EBSD refinement methods in order to find the best match among pseudo-symmetric variants. (#598)
- Saving and loading of an `EBSDDetector`. (#595)
- EBSD refinement methods now return the number of function evaluations. (#593)
- Which points in a crystal map to refine can be controlled by passing a navigation mask. (#593)
- Which points to consider when merging crystal maps can be controlled by passing navigation masks. (#593)
- Which patterns to do dictionary indexing of can be controlled by passing a navigation mask. (#593)
- Downsampling of EBSD patterns which maintain the data type by also rescaling to the data type range. (#592)
- Method to get a PyEBSDIndex `EBSDIndexer` instance from an `EBSDDetector`, convenient for either indexing with PyEBSDIndex or for use with kikuchipy. (#590)
- Convenience function to get a `CrystalMap` from a PyEBSDIndex Hough indexing result array. (#590)
- PyEBSDIndex as an optional dependency. (#590)
- Two tutorials showing projection center (PC) fitting and extrapolation to obtain a plane of PCs to index a full dataset. (#588)
- Tutorial showing sloppy projection center (PC)/orientation optimization landscape of the Ni dataset from Jackson et al. (2019), replicating the results from Pang et al. (2020). (#588)
- Method `EBSDDetector.fit_pc()` to fit a plane using a projective or affine transformation to projection centers following work by Winkelmann and co-workers (2020). (#587)
- Method `EBSDDetector.extrapolate_pc()` to return a new detector with a plane of projection centers (PCs) extrapolated from a mean PC calculated from one or more PCs following work by Singh et al. (2017). (#587)

- Methods `EBSDDetector.estimate_xtilt()` and `EBSDDetector.estimate_xtilt_ztilt()` to estimate the tilts about the detector X and Z axes which bring the detector plane normal parallel to the sample plane normal, following work by Winkelmann and co-workers (2020). (#587)
- Method `EBSDDetector.plot_pc()` to plot projection centers (PCs) in maps, scatter plots or in 3D. (#587)
- Convenience function `kikuchipy.draw.plot_pattern_positions_in_map()` to plot positions of selected patterns (typically calibration patterns) in a 2D map. (#587)
- EBSD signal returned from NORDIF calibration pattern reader tries to add the following new info to the original metadata: Shapes of area and region of interest (ROI), offset of ROI, calibration pattern indices and area overview image. All shapes and coordinates are given both in units of area overview image pixels and scaled according to the pixels in the ROI (actual navigation shape). (#586)
- Method `EBSD.extract_grid()` to get a new signal from grid positions evenly spaced in navigation space. (#585)
- Utility function `grid_indices()` to extract a smaller 1D or 2D grid of indices from a larger grid. (#585)
- Seven EBSD master pattern files simulated with EMsoft are available from `ebsd_master_pattern()` via the data module for download to the local cache: aluminium, nickel, silicon, austenite, ferrite, a chi-phase in steel and a sigma-phase in steel. (#584, #607)
- Some experimental EBSD datasets are available for download to the local cache via the data module: (50, 50) patterns of (480, 480) pixels from an Si wafer via `si_wafer()`, ten full Ni datasets of (149, 200) patterns of (60, 60) pixels via `ni_gain(number)` (number 1-10) (parts of number 1 are used in `nickel_ebsd_small()/large()`) and the calibration patterns of the ten Ni datasets, `ni_gain_calibration(number)`. (#584, #593, #607)
- When using the following HyperSpy `Signal2D` methods via the `EBSD` class, the class attributes `xmap`, `static_background` and `detector` are handled correctly, which they were not before: `inav`, `isig`, `crop()`, `crop_image()`. If handling fails, the old behavior is retained. This handling is experimental. (#578)
- `EBSDDetector.crop()` method to get a new detector with its shape cropped, also updating the PC values accordingly. (#578)

### 4.10.2 Changed

- Minimal version of orix set to >= 0.11 and of Numba set to >= 0.55. (#608)
- Added warnings when trying to perform adaptive histogram equalization on a signal with data in floating type or when some of the data is NaN. (#606)
- Dask arrays returned from EBSD refinement methods has the number of function evaluations as the second element after the score. (#593)
- Stricter phase comparison in EBSD refinement. The phase in the crystal map points to refine must have the same name, space group, point group and structure (atoms and lattice) as the master pattern phase. (#593)
- Passing two crystal maps with identical phases when merging returns a map with one phase instead of two and does not raise a warning, as before. (#593)
- Exclude documentation and tests from source distribution. (#588)
- Minimal version of HyperSpy increased to >= 1.7.3. (#585)
- When binning the navigation dimension(s) with `EBSD.rebin()`, the class attributes `xmap` and `static_background` are set to `None` and `detector.pc` is set to `[0.5, 0.5, 0.5]` in the appropriate navigation shape. If the signal dimension(s) are binned, the `static_background` is binned similarly while the `detector.shape` and `detector.binning` are updated. If this handling of attributes fails, the old behavior is retained. This handling is experimental. (#578)

- EBSD signal loaded with `nickel_ebsd_small()` and `nickel_ebsd_large()` now contain crystal maps with orientations and detectors with PC values found from Hough indexing with PyEBSDIndex followed by orientation and PC refinement. (#578, #584)
- Minimal version of Matplotlib is 3.5.0 when installing optional dependencies with `pip install kikuchipy[viz]` since PyVista requires this. (#578)

### 4.10.3 Deprecated

- `kikuchipy.generators.VirtualBSEGenerator` class is deprecated and will be removed in version 0.9. Use `kikuchipy.imaging.VirtualBSEImager` instead. (#608)
- The data module functions `silicon_ebsd_moving_screen_x()`, where "x" is "in", "out5mm" or "out10mm", are deprecated and will be removed in v0.9. Use `si_ebsd_moving_screen(distance)` instead, where `distance` is `0` (in), `5` or `10`. (#607)
- The PyPI selector `viz` is replaced by `all`, which installs all optional dependencies. `viz` will be removed in version 0.9. Install optional dependencies manually or via `pip install kikuchipy[all]`. (#590)
- `projections` module with classes `GnomonicProjection`, `HesseNormalForm`, `LambertProjection` and `SphericalProjection`. These will be removed in version 0.9.0, as they are unused internally. If you depend on this module, please open an issue at https://github.com/pyxem/kikuchipy/issues. (#577)

### 4.10.4 Removed

- `mask` parameter in EBSD refinement methods; use `signal_mask` instead. (#577)
- `ebsd_projections` module. (#577)

### 4.10.5 Fixed

- Default `EBSD.detector.shape` is now correct when a detector is not passed upon initialization. (#603)
- Oxford Instruments .ebsp files of version 4 can now be read. (#602)
- When loading EBSD patterns from H5OINA files, the detector tilt and binning are available in the returned signal's `detector` attribute. (#600)
- Range of (kinematical) intensities in `KikuchiPatternSimulator.plot()` maximizes the strongest reflectors (make black) instead of minimizing the weakest reflectors (make white), which was the previous behavior. (#599)
- Inversion of `signal_mask` in the normalized cross-correlation and normalized dot product metrics is now done internally, to be in line with the docstrings (does not affect the use of this parameter and `metric="ncc"` or `metric="ndp"` in `EBSD.dictionary_indexing()`). (#593)
- `EBSDDetector.pc_average` no longer rounds the PC to three decimals. (#586)
- Microscope magnification is now read correctly from EDAX h5ebsd files. (#586)
- kikuchipy h5ebsd reader can read a signal with an EBSD detector with a PC array of different navigation shape than determined from the HDF5 file's navigation shape (e.g. `Scan 1/EBSD/Header/n_columns` and `n_rows`). (#578)

## 4.11 0.7.0 (2022-10-29)

### 4.11.1 Added

- Signal mask passed to EBSD orientation and projection center refinement methods is now applied to the experimental pattern as well. (#573)
- Dependency `imageio` needed for reading EBSD patterns in image files. (#570)
- Reader of an `EBSD` signal from all images in a directory assuming they are of the same shape and data type. (#570)
- Reader of an `EBSD` signal from EDAX TSL's binary UP1/UP2 file formats. (#569)
- Ability to project simulate patterns from a master pattern using varying projection centers (PCs) in `EBSDMasterPattern.get_patterns()`. An example is added to the method to show this. (#567)
- Allow not setting `energy` parameter in `EBSDMasterPattern.get_patterns()`, upon which the highest energy available is used. (#567)
- Improved handling of custom attributes `xmap`, `detector` and `static_background` in `EBSD` and `hemisphere`, `phase` and `projection` in EBSD/ECP master pattern classes when calling inherited HyperSpy `Signal2D` methods `as_lazy()`, `change_dtype()`, `compute()`, `deepcopy()`, `set_signal_type()` and `squeeze()`. (#564)
- Reader of an electron channelig pattern (ECP) master pattern from an EMsoft HDF5 file into an `ECPMasterPattern` signal. (#564)
- Reader of a transmission kikuchi diffraction (TKD) master pattern from an EMsoft HDF5 file into an `EBSDMasterPattern` signal. (#564)
- `ECPMasterPattern` class. (#564)
- Some internal logging which can be controlled via `kikuchipy.set_log_level()`. (#564)
- Reader of an `EBSD` signal from Oxford Instrument's h5ebsd format (H5OINA). (#562)
- Figures of reference frames of other software added to the documentation. (#552)
- Whether to show progressbars from most signal methods (except indexing and refinement) can be controlled by passing `show_progressbar` or by setting HyperSpy's `hs.preferences.General.show_progressbar` (see their docs for details). (#550)

### 4.11.2 Changed

- Documentation theme from *Furo* to *PyData*, as the growing API reference is easier to navigate with the latter. (#574)
- Use Rodrigues-Frank vector components (Rx, Ry, Rz) instead of Euler angles in EBSD orientation and projection center refinement methods. This means that if refinement is not directly but a Dask array is returned from any of these methods, the data which previously contained Euler angles now contain these vector components. This change was done to speed up refinement. (#573)
- Most of the `EBSD` metadata structure is removed, in an effort to move all relevant data to the attributes `xmap`, `static_background`, and `detector`. (#562)
- h5ebsd plugin split into one plugin for each h5ebsd format (kikuchipy, EDAX TSL, and Bruker Nano). (#562)
- `EBSDDetector.plot()` and `PCCalibrationMovingScreen.plot()` parameter `return_fig_ax` renamed to `return_figure`. (#552)
- Import modules lazily using the specification in PEP 562. (#551)

- Minimal version of HyperSpy increased to >= 1.7.1. (#550)
- `progressbar` parameter to `show_progressbar` in `kikuchipy.data` functions which accepts a `allow_download` parameter. If not given, the value is retreived from HyperSpy's preferences. (#550)

### 4.11.3 Deprecated

- `mask` parameter in EBSD orientation and projection center refinement is deprecated in favor of `signal_mask`, and will be removed in version 0.8.0. (#573)
- `projections.ebsd_projections` module. (#563)

### 4.11.4 Removed

- `EBSDSimulationGenerator` and `GeometricalEBSDSimulation` (use `KikuchiPatternSimulator` and `GeometricalKikuchiPatternSimulation` instead) and `simulations.features` module. (#563)
- `crystallography` module. (#563)
- Options `"north"` and `"south"` for property `EBSDMasterPattern.hemisphere` and in the parameter `"hemisphere"` in `kikuchipy.data.nickel_ebsd_master_pattern_small()`; use `"upper"` and `"lower"` instead. (#563)
- Functions `remove_static_background()`, `remove_dynamic_background()` and `get_image_quality()` from `chunk` module. (#563)
- Parameter `relative` in `EBSD.remove_static_background()`. (#563)
- Functions `ebsd_metadata()` and `metadata_nodes()` which have been deprecated since v0.5. (#550, #562)
- Print information emitted from `EBSD` methods like `remove_static_background()` is removed. (#550)

### 4.11.5 Fixed

- `detector` attribute of `EBSD` signal returned from the NORDIF calibration pattern reader is now an `EBSDDetector` and not just a dictionary. (#569)
- Silence dask warning about splitting large chunks in `EBSD.dictionary_indexing()`. Memory use can be controlled by rechunking the dictionary or setting the `rechunk` or `n_per_iteration` parameters. (#567)

## 4.12 0.6.1 (2022-06-17)

### 4.12.1 Contributors

- Håkon Wiik Ånes

### 4.12.2 Fixed

- Incorrect filtering of zone axes labels in geometrical simulations. (#544)

## 4.13 0.6.0 (2022-06-16)

### 4.13.1 Contributors

- Håkon Wiik Ånes

### 4.13.2 Added

- `EBSDMasterPattern.plot_spherical()` for plotting a master pattern in the stereographic projection on the 3D sphere. (#536)
- Projection of master pattern in the stereographic projection to the square Lambert projection via `EBSDMasterPattern.to_lambert()`. (#536)
- New package dependencies on `pyvista` for 3D plotting and on `pythreejs` for the docs are introduced. (#536)
- Reduce time and memory use of the following `kikuchipy.signals.EBSD` methods by using `hyperspy.signal.BaseSignal.map()`: `remove_static_background()`, `remove_dynamic_background()` and `get_image_quality()`. (#527)
- `progressbar` parameter to functions downloading external datasets in the data module. (#515)
- Support for Python 3.10. (#504)
- `EBSD.static_background` property for easier access to the background pattern. (#475)

### 4.13.3 Changed

- Valid `EBSDMasterPattern.hemisphere` values from `"north"` and `"south"` to `"upper"` and `"lower"`, respectively, to be in line with *orix*. (#537)
- Increase minimal version of diffsims to 0.5. (#537)
- Chunking of EBSD signal navigation dimensions in `EBSD.average_neighbour_patterns()` to reduce memory use. (#532)
- Remove requirement that the crystal map used for EBSD refinement has identical step size(s) to the EBSD signal's navigation axes. This raised an error previously, but now only emits a warning. (#531)
- Increase minimal version of HyperSpy to 1.7. (#527)
- Increase minimal version of SciPy to 1.7. (#504)

### 4.13.4 Deprecated

- The `kikuchipy.simulations.GeometricalEBSDSimulation` class is deprecated and will be removed in version 0.7. Obtain `kikuchipy.simulations.GeometricalKikuchiPatternSimulation` via `kikuchipy.simulations.KikuchiPatternSimulator.on_detector()` instead. The `kikuchipy.simulations.features` module is also deprecated and will be removed in version 0.7. Obtain Kikuchi line and zone axis detector/gnomonic coordinates of a simulation via `lines_coordinates()` and `zone_axes_coordinates()` instead. (#537)
- The `kikuchipy.generators.EBSDSimulationGenerator` class is deprecated and will be removed in version 0.7. Use the `kikuchipy.simulations.KikuchiPatternSimulator` class instead. (#537)
- The `kikuchipy.crystallography.matrices` module is deprecated and will be removed in version 0.7, access the matrices via `diffpy.structure.lattice.Lattice` attributes instead. (#537)
- The following functions for processing of pattern chunks in the `kikuchipy.pattern.chunk` module are deprecated and will be removed in version 0.7: `get_image_quality()`, `remove_dynamic_background()` and `remove_static_background()`. Use the `EBSD` class for processing of many patterns. (#527, #533 )

### 4.13.5 Removed

- The `relative` parameter in `kikuchipy.signals.EBSD.remove_static_background()`. The parameter is accepted but not used. Passing it after this release will result in an error. (#527)

### 4.13.6 Fixed

- Plotting of geometrical simulation markers on rectangular patterns. (#537)
- Hopefully prevent EBSD refinement tests using random data to fail on Azure. (#465)

## 4.14 0.5.8 (2022-05-16)

### 4.14.1 Contributors

- Håkon Wiik Ånes

### 4.14.2 Changed

- Minimal version of `orix` is increased to 0.9. (#520)

### 4.14.3 Fixed

- Internal use of `orix.vector.Vector3d` following `orix`' 0.9.0 release. (#520)

## 4.15 0.5.7 (2022-01-10)

### 4.15.1 Contributors

- Håkon Wiik Ånes

### 4.15.2 Fixed

- EBSD orientation refinement on Windows producing garbage results due to unpredictable behaviour in Numba function which converts Euler triplet to quaternion. (#495)

## 4.16 0.5.6 (2022-01-02)

### 4.16.1 Contributors

- Håkon Wiik Ånes

### 4.16.2 Added

- Convenience function *get_rgb_navigator()* to create an RGB signal from an RGB image. (#491)

### 4.16.3 Changed

- Pattern matching notebook to include orientation maps from orix. (#491)

## 4.17 0.5.5 (2021-12-12)

### 4.17.1 Contributors

- Håkon Wiik Ånes
- Zhou Xu

### 4.17.2 Fixed

- Not flipping rows and columns when saving non-square patterns to kikuchipy's h5ebsd format. (#486)

## 4.18 0.5.4 (2021-11-17)

### 4.18.1 Contributors

- Håkon Wiik Ånes

### 4.18.2 Added

- Optional parameters *rechunk* and *chunk_kwargs* to EBSD refinement methods to better control possible rechunking of pattern array before refinement. (#470)

### 4.18.3 Changed

- When EBSD refinement methods don't immediately compute, they return a dask array instead of a list of delayed instances. (#470)

### 4.18.4 Fixed

- Memory issue in EBSD refinement due to naive use of dask.delayed. Uses map_blocks() instead. (#470)

## 4.19 0.5.3 (2021-11-02)

### 4.19.1 Contributors

- Håkon Wiik Ånes
- Zhou Xu

### 4.19.2 Added

- Printing of speed (patterns per second) of dictionary indexing and refinement. (#461)
- Restricted newest version of hyperspy>=1.6.5 due to incompatibility with h5py>=3.5. (#461)

### 4.19.3 Fixed

- Handling of projection centers (PCs): Correct conversion from/to EMsoft's convention requires binning factor *and* detector pixel size. Conversion between TSL/Oxford and Bruker conventions correctly uses detector aspect ratio. (#455)

## 4.20 0.5.2 (2021-09-11)

### 4.20.1 Contributors

- Håkon Wiik Ånes

### 4.20.2 Changed

- Add gnomonic circles as patches in axes returned from EBSDDetector.plot(). (#445)
- Restrict lowest supported version of orix to >= 0.7. (#444)

## 4.21 0.5.1 (2021-09-01)

### 4.21.1 Contributors

- Håkon Wiik Ånes

### 4.21.2 Added

- Automatic creation of a release using GitHub Actions, which will simplify and lead to more frequent patch releases. (#433)

## 4.22 0.5.0 (2021-08-31)

### 4.22.1 Contributors

- Eric Prestat
- Håkon Wiik Ånes
- Lars Andreas Hastad Lervik

### 4.22.2 Added

- Possibility to specify whether to rechunk experimental and simulated data sets and which data type to use for dictionary indexing. (#419)
- How to use the new orientation and/or projection center refinements to the pattern matching notebook. (#405)
- Notebooks to the documentation as shorter or longer "Examples" that don't fit in the user guide. (#403)
- Refinement module for EBSD refinement. Allows for the refinement of orientations and/or projection center estimates. (#387)

### 4.22.3 Changed

- If a custom metric is to be used for dictionary indexing, it must now be a class inheriting from an abstract *SimilarityMetric* class. This replaces the previous *SimilarityMetric* class and the *make_similarity_metric()* function. (#419)
- Dictionary indexing parameter *n_slices* to *n_per_iteration*. (#419)
- *merge_crystal_maps* parameter *metric* to *greater_is_better*. (#419)
- *orientation_similarity_map* parameter *normalized* is by default False. (#419)
- Dependency versions for dask >= 2021.8.1, fixing some memory issues encountered after 2021.3.1, and HyperSpy >= 1.6.4. Remove importlib_metadata from package dependencies. (#418)
- Performance improvements to EBSD dictionary generation, giving a substantial speed-up. (#405)
- Rename projection methods from *project()*/*iproject()* to *vector2xy()*/*xy2vector()*. (#405)
- URLs of user guide topics have an extra "/user_guide/<topic>" added to them. (#403)

### 4.22.4 Deprecated

- Custom EBSD metadata, meaning the *Acquisition_instrument.SEM.EBSD.Detector* and *Sample.Phases* nodes, as well as the EBSD *set_experimental_parameters()* and *set_phase_parameters()* methods. This will be removed in v0.6 The *static_background* metadata array will become available as an EBSD property. (#428)

### 4.22.5 Removed

- *make_similarity_metric()* function is replaced by the need to create a class inheriting from a new abstract *SimilarityMetric* class, which provides more freedom over preparations of arrays before dictionary indexing. (#419)
- *EBSD.match_patterns()* is removed, use *EBSD.dictionary_indexing()* instead. (#419)
- kikuchipy.pattern.correlate module. (#419)

### 4.22.6 Fixed

- Allow static background in EBSD metadata to be a Dask array. (#413)
- Set newest supported version of Sphinx to 4.0.2 so that nbsphinx works. (#403)

## 4.23 0.4.0 (2021-07-08)

### 4.23.1 Contributors

- Håkon Wiik Ånes

### 4.23.2 Added

- Sample tilt about RD can be passed as part of an EBSDDetector. This can be used when projecting parts of master patterns onto a detector. (#381)
- Reader for uncompressed EBSD patterns stored in Oxford Instrument's binary .ebsp file format. (#371, #391)
- Unit testing of docstring examples. (#350)
- Support for Python 3.9. (#348)
- Projection/pattern center calibration via the moving screen technique in a kikuchipy.detectors.calibration module. (#322)
- Three single crystal Si EBSD patterns, from the same sample position but with varying detector distances, to the data module (via external repo). (#320)
- Reading of NORDIF calibration patterns specified in a setting file into an EBSD signal. (#317)

### 4.23.3 Changed

- Only return figure from kikuchipy.filters.Window.plot() if desired, also add a colorbar only if desired. (#375)

### 4.23.4 Deprecated

- The kikuchipy.pattern.correlate module will be removed in v0.5. Use kikuchipy.indexing.similarity_metrics instead. (#377)
- Rename the EBSD.match_patterns() method to EBSD.dictionary_indexing(). match_patterns() will be removed in v0.5. (#376)

### 4.23.5 Fixed

- Set minimal requirement of importlib_metadata to v3.6 so Binder can run user guide notebooks with HyperSpy 1.6.3. (#395)
- Row (y) coordinate array returned with the crystal map from dictionary indexing is correctly sorted. (#392)
- Deep copying EBSD and EBSDMasterPattern signals carry over, respectively, *xmap* and *detector*, and *phase*, *hemisphere* and *projection* properties (#356).
- Scaling of region of interest coordinates used in virtual backscatter electron imaging to physical coordinates. (#349)

## 4.24 0.3.4 (2021-05-26)

### 4.24.1 Contributors

- Håkon Wiik Ånes

### 4.24.2 Added

- Restricted newest version of dask<=2021.03.1 and pinned orix==0.6.0. (#360)

## 4.25 0.3.3 (2021-04-18)

### 4.25.1 Contributors

- Håkon Wiik Ånes
- Ole Natlandsmyr

### 4.25.2 Fixed

- Reading of EBSD patterns from Bruker h5ebsd with a region of interest. (#339)
- Merging of (typically refined) crystal maps, where either a simulation indices array is not present or the array contains more indices per point than scores. (#335)
- Bugs in getting plot markers from geometrical EBSD simulation. (#334)
- Passing a static background pattern to EBSD.remove_static_background() for a non-square detector dataset works. (#331)

## 4.26 0.3.2 (2021-02-01)

### 4.26.1 Contributors

- Håkon Wiik Ånes

### 4.26.2 Fixed

- Deletion of temporary files saved to temporary directories in user guide. (#312)
- Pattern matching sometimes failing to generate a crystal map due to incorrect creation of spatial arrays. (#307)

## 4.27 0.3.1 (2021-01-22)

### 4.27.1 Contributors

- Håkon Wiik Ånes

### 4.27.2 Fixed

- Version link Binder uses to make the Jupyter Notebooks run in the browser. (#301)

## 4.28 0.3.0 (2021-01-22)

Details of all development associated with this release is listed below and in this GitHub milestone.

### 4.28.1 Contributors

- Håkon Wiik Ånes
- Lars Andreas Hastad Lervik
- Ole Natlandsmyr

### 4.28.2 Added

- Calculation of an average dot product map, or just the dot product matrices. (#280)
- A nice gallery to the documentation with links to each user guide page. (#285)
- Support for writing/reading an EBSD signal with 1 or 0 navigation axes to/from a kikuchipy h5ebsd file. (#276)
- Better control over dask array chunking when processing patterns. (#275)
- User guide notebook showing basic pattern matching. (#263)
- EBSD.detector property storing an EBSDDetector. (#262)
- Link to Binder in README and in the notebooks for running them in the browser. (#257)
- Creation of dictionary of dynamically simulated EBSD patterns from a master pattern in the square Lambert projection. (#239)
- A data module with a small Nickel EBSD data set and master pattern, and a larger EBSD data set downloadable via the module. Two dependencies, pooch and tqdm, are added along with this module. (#236, #237, #243)
- Pattern matching of EBSD patterns with a dictionary of pre-computed simulated patterns with known crystal orientations, and related useful tools (#231, #233, #234): (1) A framework for creation of similarity metrics used in pattern matching, (2) computation of an orientation similarity map from indexing results, and (3) creation of a multi phase crystal map from single phase maps from pattern matching.
- EBSD.xmap property storing an orix CrystalMap. (#226)
- Dependency on the diffsims package for handling of electron scattering and diffraction. (#220)
- Square Lambert mapping, and its inverse, from points on the unit sphere to a 2D square grid, as implemented in Callahan and De Graef (2013). (#214)
- Geometrical EBSD simulations, projecting a set of Kikuchi bands and zone axes onto a detector, which can be added to an EBSD signal as markers. (#204, #219, #232)
- EBSD detector class to handle detector parameters, including detector pixels' gnomonic coordinates. EBSD reference frame documentation. (#204, #215)
- Reader for EMsoft's simulated EBSD patterns returned by their EMEBSD.f90 program. (#202)

### 4.28.3 Changed

- The feature maps notebook to include how to obtain an average dot product map and dot product matrices for an EBSD signal. (#280)
- Averaging EBSD patterns with nearest neighbours now rescales to input data type range, thus loosing relative intensities, to avoid clipping intensities. (#280)
- Dependency requirement of diffsims from >= 0.3 to >= 0.4 (#282)
- Name of hemisphere axis in EBSDMasterPattern from "y" to "hemisphere". (#275)
- Replace Travis CI with GitHub Actions. (#250)
- The EBSDMasterPattern gets phase, hemisphere and projection properties. (#246)
- EMsoft EBSD master pattern plugin can read a single energy pattern. Parameter *energy_range* changed to *energy*. (240)
- Migrate user guide from reST files to Jupyter Notebooks converted to HTML with the *nbsphinx* package. (#236, #237, #244, #245, #279, #245, #279, #281)
- Move GitHub repository to the pyxem organization. Update relevant URLs. (#198)
- Allow scikit-image >= 0.16. (#196)
- Remove language_version in pre-commit config file. (#195)

### 4.28.4 Removed

- The EBSDMasterPattern and EBSD metadata node Sample.Phases, to be replaced by class attributes. The set_phase_parameters() method is removed from both classes, and the set_simulation_parameters() is removed from the former class. (#246)

### 4.28.5 Fixed

- IndexError in neighbour pattern averaging (#280)
- Reading of square Lambert projections from EMsoft's master pattern file now sums contributions from asymmetric positions correctly. (#255)
- NumPy array creation when calculating window pixel's distance to the origin is not ragged anymore. (#221)

## 4.29 0.2.2 (2020-05-24)

This is a patch release that fixes reading of EBSD data sets from h5ebsd files with arbitrary scan group names.

### 4.29.1 Contributors

- Håkon Wiik Ånes

### 4.29.2 Fixed

- Allow reading of EBSD patterns from h5ebsd files with arbitrary scan group names, not just "Scan 1", "Scan 2", etc., like was the case before. (#188)

## 4.30 0.2.1 (2020-05-20)

This is a patch release that enables installing kikuchipy 0.2 from Anaconda and not just PyPI.

### 4.30.1 Contributors

- Håkon Wiik Ånes

### 4.30.2 Changed

- Use numpy.fft instead of scipy.fft because HyperSpy requires scipy < 1.4 on conda-forge, while scipy.fft was introduced in scipy 1.4. (#180)

### 4.30.3 Fixed

- With the change above, kikuchipy 0.2 should be installable from Anaconda and not just PyPI. (#180)

## 4.31 0.2.0 (2020-05-19)

Details of all development associated with this release are available here.

### 4.31.1 Contributors

- Håkon Wiik Ånes
- Tina Bergh

### 4.31.2 Added

- Jupyter Notebooks with tutorials and example workflows available.
- Grey scale and RGB virtual backscatter electron (BSE) images can be easily generated with the VirtualBSEGenerator class. The generator return objects of the new signal class VirtualBSEImage, which inherit functionality from HyperSpy's Signal2D class. (#170)
- EBSD master pattern class and reader of master patterns from EMsoft's EBSD master pattern file. (#159)
- Python 3.8 support. (#157)

- The public API has been restructured. The pattern processing used by the EBSD class is available in the kikuchipy.pattern subpackage, and filters/kernels used in frequency domain filtering and pattern averaging are available in the kikuchipy.filters subpackage. (#169)
- Intensity normalization of scan or single patterns. (#157)
- Fast Fourier Transform (FFT) filtering of scan or single patterns using SciPy's fft routines and Connelly Barnes' filterfft. (#157)
- Numba dependency to improve pattern rescaling and normalization. (#157)
- Computing of the dynamic background in the spatial or frequency domain for scan or single patterns. (#157)
- Image quality (IQ) computation for scan or single patterns based on N. C. K. Lassen's definition. (#157)
- Averaging of patterns with nearest neighbours with an arbitrary kernel, e.g. rectangular or Gaussian. (#134)
- Window/kernel/filter/mask class to handle such things, e.g. for pattern averaging or filtering in the frequency or spatial domain. Available in the kikuchipy.filters module. (#134, #157)

### 4.31.3 Changed

- Renamed five EBSD methods: static_background_correction to remove_static_background, dynamic_background_correction to remove_dynamic_background, rescale_intensities to rescale_intensity, virtual_backscatter_electron_imaging to plot_virtual_bse_intensity, and get_virtual_image to get_virtual_bse_intensity. (#157, #170)
- Renamed kikuchipy_metadata to ebsd_metadata. (#169)
- Source code link in the documentation should point to proper GitHub line. This *linkcode_resolve* in the *conf.py* file is taken from SciPy. (#157)
- Read the Docs CSS style. (#157)
- New logo with a gradient from experimental to simulated pattern (with EMsoft), with a color gradient from the plasma color maps. (#157)
- Dynamic background correction can be done faster due to Gaussian blurring in the frequency domain to get the dynamic background to remove. (#157)

### 4.31.4 Removed

- Explicit dependency on scikit-learn (it is imported via HyperSpy). (#168)
- Dependency on pyxem. Parts of their virtual imaging methods are adapted here—a big thank you to the pyxem/HyperSpy team! (#168)

### 4.31.5 Fixed

- RtD builds documentation with Python 3.8 (fixed problem of missing .egg leading build to fail). (#158)

## 4.32 0.1.3 (2020-05-11)

kikuchipy is an open-source Python library for processing and analysis of electron backscatter diffraction patterns: https://kikuchipy.org.

This is a patch release. It is anticipated to be the final release in the *0.1.x* series.

### 4.32.1 Added

- Package installation with Anaconda via the conda-forge channel.

### 4.32.2 Fixed

- Static and dynamic background corrections are done at float 32-bit precision, and not integer 16-bit.
- Chunking of static background pattern.
- Chunking of patterns in the h5ebsd reader.

## 4.33 0.1.2 (2020-01-09)

kikuchipy is an open-source Python library for processing and analysis of electron backscatter diffraction patterns: https://kikuchipy.org.

This is a bug-fix release that ensures, unlike the previous bug-fix release, that necessary files are downloaded when installing from PyPI.

## 4.34 0.1.1 (2020-01-04)

This is a bug fix release that ensures that necessary files are uploaded to PyPI.

## 4.35 0.1.0 (2020-01-04)

We're happy to announce the release of kikuchipy v0.1.0!

kikuchipy is an open-source Python library for processing and analysis of electron backscatter diffraction (EBSD) patterns. The library builds upon the tools for multi-dimensional data analysis provided by the HyperSpy library.

For more information, a user guide, and the full reference API documentation, please visit: https://kikuchipy.org.

This is the initial pre-release, where things start to get serious… seriously fun!

### 4.35.1 Features

- Load EBSD patterns and metadata from the NORDIF binary format (.dat), or Bruker Nano's or EDAX TSL's h5ebsd formats (.h5) into an `EBSD` object, e.g. `s`, based upon HyperSpy's *Signal2D* class, using `s = kp.load()`. This ensures easy access to patterns and metadata in the attributes `s.data` and `s.metadata`, respectively.
- Save EBSD patterns to the NORDIF binary format (.dat) and our own h5ebsd format (.h5), using `s.save()`. Both formats are readable by EMsoft's NORDIF and EMEBSD readers, respectively.
- All functionality in kikuchipy can be performed both directly and lazily (except some multivariate analysis algorithms). The latter means that all operations on a scan, including plotting, can be done by loading only necessary parts of the scan into memory at a time. Ultimately, this lets us operate on scans larger than memory using all of our cores.
- Visualize patterns easily with HyperSpy's powerful and versatile `s.plot()`. Any image of the same navigation size, e.g. a virtual backscatter electron image, quality map, phase map, or orientation map, can be used to navigate in. Multiple scans of the same size, e.g. a scan of experimental patterns and the best matching simulated patterns to that scan, can be plotted simultaneously with HyperSpy's `plot_signals()`.
- Virtual backscatter electron (VBSE) imaging is easily performed with `s.virtual_backscatter_electron_imaging()` based upon similar functionality in pyXem. Arbitrary regions of interests can be used, and the corresponding VBSE image can be inspected interactively. Finally, the VBSE image can be obtained in a new `EBSD` object with `vbse = s.get_virtual_image()`, before writing the data to an image file in your desired format with matplotlib's `imsave('filename.png', vbse.data)`.
- Change scan and pattern size, e.g. by cropping on the detector or extracting a region of interest, by using `s.isig` or `s.inav`, respectively. Patterns can be binned (upscaled or downscaled) using `s.rebin`. These methods are provided by HyperSpy.
- Perform static and dynamic background correction by subtraction or division with `s.static_background_correction()` and `s.dynamic_background_correction()`. For the former correction, relative intensities between patterns can be kept if desired.
- Perform adaptive histogram equalization by setting an appropriate contextual region (kernel size) with `s.adaptive_histogram_equalization()`.
- Rescale pattern intensities to desired data type and range using `s.rescale_intensities()`.
- Multivariate statistical analysis, like principal component analysis and many other decomposition algorithms, can be easily performed with `s.decomposition()`, provided by HyperSpy.
- Since the `EBSD` class is based upon HyperSpy's `Signal2D` class, which itself is based upon their `BaseSignal` class, all functionality available to `Signal2D` is also available to the `EBSD` class. See HyperSpy's user guide (http://hyperspy.org/hyperspy-doc/current/index.html) for details.

### 4.35.2 Contributors

- Håkon Wiik Ånes
- Tina Bergh

CHAPTER

FIVE

# INSTALLATION

kikuchipy can be installed with pip or conda:

### pip

```
pip install kikuchipy
```

### conda

```
conda install kikuchipy -c conda-forge
```

Further details are available in the *installation guide*.

CHAPTER

SIX

# LEARNING RESOURCES

Tutorials

In-depth guides for using kikuchipy.

Examples

Short recipies to common tasks using kikuchipy.

API reference

Descriptions of all functions, modules, and objects in kikuchipy.

Contributing

kikuchipy is a community project maintained for and by its users. There are many ways you can help!

CHAPTER

SEVEN

# CITING KIKUCHIPY

If you are using kikuchipy in your scientific research, please help our scientific visibility by citing the Zenodo DOI: https://doi.org/10.5281/zenodo.3597646.

# BIBLIOGRAPHY


[BAA+23] Tina Bergh, Håkon Wiik Ånes, Ragnhild Aune, Sigurd Wenner, Randi Holmestad, Xiaobo Ren, and Per Erik Vullum. Intermetallic phase layers in cold metal transfer aluminium-steel welds with an al–si–mn filler alloy. *MATERIALS TRANSACTIONS*, 64(2):352–359, 2023. doi:10.2320/matertrans.MT-LA2022046.

[BWR19] Patrick T. Brewick, Stuart I. Wright, and David J. Rowenhorst. NLPAR: Non-local smoothing for enhanced EBSD pattern indexing. *Ultramicroscopy*, 200:50–61, 2019. doi:10.1016/j.ultramic.2019.02.013.

[BJG+16] T. B. Britton, J. Jiang, Y. Guo, A. Vilalta-Clemente, D. Wallis, L. N. Hansen, Aimo Winkelmann, and Angus J. Wilkinson. Tutorial: Crystal orientations and EBSD - Or which way is up? *Materials Characterization*, 117:113–126, 2016. doi:10.1016/j.matchar.2016.04.008.

[CDeGraef13] Patrick G. Callahan and Marc De Graef. Dynamical Electron Backscatter Diffraction Patterns. Part I: Pattern Simulations. *Microscopy and Microanalysis*, pages 1255–1265, 2013. doi:10.1017/S1431927613001840.

[CPW+15] Yu H. Chen, Se Un Park, Dennis Wei, Greg Newstadt, Michael A. Jackson, Jeff P. Simmons, Marc De Graef, and Alfred O. Hero. A Dictionary Approach to Electron Backscatter Diffraction Indexing. *Microscopy and Microanalysis*, 21(3):739–752, 2015. doi:10.1017/S1431927615000756.

[FCWB19] Alexander Foden, David M. Collins, Angus J. Wilkinson, and Thomas B. Britton. Indexing electron backscatter diffraction patterns with a refined template matching approach. *Ultramicroscopy*, 207:112845, 2019. doi:10.1016/j.ultramic.2019.112845.

[GW17] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson Education Limited, 4th edition, 2017. ISBN 978-0133356724.

[Gos12] Ardeshir A. Goshtasby. *Image registration: Principles, tools and methods*. Springer Science & Business Media, 2012.

[HH91] Jarle Hjelen, Eivind Hoel, and Roar Ørsund. Electron diffraction in the SEM. *Micron and Microscopica Acta*, 22(1-2):137–138, 1991. doi:10.1016/0739-6260(91)90128-M.

[JGU+14] Michael A. Jackson, Michael A. Groeber, Michael D. Uchic, David J. Rowenhorst, and Marc De Graef. h5ebsd: an archival data format for electron back-scatter diffraction data sets. *Integrating Materials and Manufacturing Innovation*, 3(1):4, 2014. doi:10.1186/2193-9772-3-4.

[JPDeGraef19] Michael A. Jackson, Elena Pascal, and Marc De Graef. Dictionary Indexing of Electron Back-Scatter Diffraction Patterns: a Hands-On Tutorial. *Integrating Materials and Manufacturing Innovation*, pages 1–21, 2019. doi:10.1007/s40192-019-00137-4.

[Kir98] Earl J Kirkland. *Advanced computing in electron microscopy*. Volume 12. Springer, 1998. doi:10.1007/978-1-4757-4406-4.

[Las94] Niels Christian Krieger Lassen. *Automated Determanation of Crystal Orientations from Electron Backscattering Patterns*. PhD thesis, Institute of Mathematical Modelling, 1994.

[LVanDyck14] Ivan Lobato and Dirk Van Dyck. An accurate parameterization for scattering factors, electron densities and electrostatic potentials for neutral atoms that obey all physical constraints. *Acta Crystallographica Section A: Foundations and Advances*, 70(6):636–649, 2014. doi:10.1107/S205327331401643X.

[MDeGraefS+17] Katharina Marquardt, Marc De Graef, Saransh Singh, Hauke Marquardt, Anja Rosenthal, and Sanae Koizuimi. Quantitative electron backscatter diffraction (EBSD) data analyses using the dictionary indexing (DI) approach: Overcoming indexing difficulties on geological materials. *American Mineralogist*, 102(9):1843–1855, 2017. doi:10.2138/am-2017-6062.

[NHW17] Gert Nolze, Ralf Hielscher, and Aimo Winkelmann. Electron backscatter diffraction beyond the mainstream. *Crystal Research and Technology*, 52(1):1–24, 2017. doi:10.1002/crat.201600252.

[NWB16] Gert Nolze, Aimo Winkelmann, and Alan P. Boyle. Pattern matching approach to pseudosymmetry problems in electron backscatter diffraction. *Ultramicroscopy*, 160:146–154, 2016. doi:10.1016/j.ultramic.2015.10.010.

[PLS20] Edward L. Pang, Peter M. Larsen, and Christopher A. Schuh. Global optimization for accurate determination of EBSD pattern centers. *Ultramicroscopy*, 209:112876, 2020. doi:10.1016/j.ultramic.2019.112876.

[Shi21] Qiwei Shi. High-definition electron diffraction patterns and their indexation results of a single crystal Si wafer. December 2021. doi:10.5281/zenodo.5803073.

[Shi22] Qiwei Shi. High-definition electron diffraction patterns and their indexation results of a polycrystal Al-Mg sample. August 2022. doi:10.5281/zenodo.6990325.

[SRDeGraef17] Saransh Singh, Farangis Ram, and Marc De Graef. Application of forward models to crystal orientation refinement. *Journal of Applied Crystallography*, 50(6):1664–1676, 2017. doi:10.1107/S1600576717014200.

[SDeGraef16] Saransh Singh and Marc De Graef. Orientation sampling for dictionary-based diffraction pattern indexing methods. *Modelling and Simulation in Materials Science and Engineering*, 2016. doi:10.1088/0965-0393/24/8/085013.

[WC18] Angus Wilkinson and David M Collins. Small ebsd map from annealed ferritic steel. June 2018. doi:10.5281/zenodo.1288431.

[WMD06] Angus J. Wilkinson, Graham Meaden, and David J. Dingley. High-resolution elastic strain measurement from electron backscatter diffraction patterns: New levels of sensitivity. *Ultramicroscopy*, 106(4-5):307–313, 2006. doi:10.1016/j.ultramic.2005.10.001.

[WNC+20] Aimo Winkelmann, Gert Nolze, Grzegorz Cios, Tomasz Tokarski, and Piotr Bała. Refined Calibration Model for Improving the Orientation Precision of Electron Backscatter Diffraction Maps. *Materials*, 13(12):2816, 2020. doi:10.3390/ma13122816.

[WNL+15] Stuart I. Wright, Matthew M. Nowell, Scott P. Lindeman, Patrick P. Camus, Marc De Graef, and Michael A. Jackson. Introduction and comparison of new EBSD post-processing methodologies. *Ultramicroscopy*, 159:81–94, 2015. doi:10.1016/j.ultramic.2015.08.001.

[WNDeKloe+15] Stuart I. Wright, Matthew M. Nowell, René De Kloe, Patrick Camus, and Travis Rampton. Electron imaging with an EBSD detector. *Ultramicroscopy*, 148:132–145, 2015. doi:10.1016/j.ultramic.2014.10.002.

[AHS+20] Håkon W. Ånes, Jarle Hjelen, Bjørn E. Sørensen, Antonius T. J. van Helvoort, and Knut Marthinsen. Processing and indexing of electron backscatter patterns using open-source software. In *IOP Conference Series: Materials Science and Engineering*, volume 891, 012002. IOP Publishing, 2020. doi:10.1088/1757-899X/891/1/012002.

[AHvHM19] Håkon W. Ånes, Jarle Hjelen, Antonius T. J. van Helvoort, and Knut Marthinsen. Electron backscatter patterns from Nickel acquired with varying camera gain. 2019. [Data set]. doi:10.5281/zenodo.7498632.

[AHvHM22] Håkon Wiik Ånes, Jarle Hjelen, Antonius T. J. van Helvoort, and Knut Marthinsen. Electron backscatter diffraction patterns from a single crystal silicon wafer. December 2022. [Data set]. doi:10.5281/zenodo.7491388.

[AvHM22a] Håkon Wiik Ånes, Antonius T. J. van Helvoort, and Knut Marthinsen. Electron backscatter diffraction data and backscatter electron images from a cold-rolled and recovered Al-Mn alloy. May 2022. [Data set]. doi:10.5281/zenodo.6470217.

[AvHM22b] Håkon Wiik Ånes, Antonius T. J. van Helvoort, and Knut Marthinsen. Electron backscatter diffraction data and backscatter electron images from four conditions from a cold-rolled and annealed Al-Mn alloy. November 2022. [Data set]. doi:10.5281/zenodo.7383087.

# PYTHON MODULE INDEX

## k

# INDEX

## F



## G



## H

## I



## K



## L



## M

## N



## O



## P

## R

## S



## U



## V



## W



## X



## Y



## Z