Bluesky, ophyd, pseudo motors, detectors

Maksim Rakitin

January 17th 2019

U.S. DEPARTMENT OF ENERGY | Office of Science

BROOKHAVEN NATIONAL LABORATORY

Outline

- NSLS-II deployment
- Ophyd hardware abstraction library
- Bluesky data collection framework
- Event-based architecture
- Databroker library for retrieving stored data
- Examples

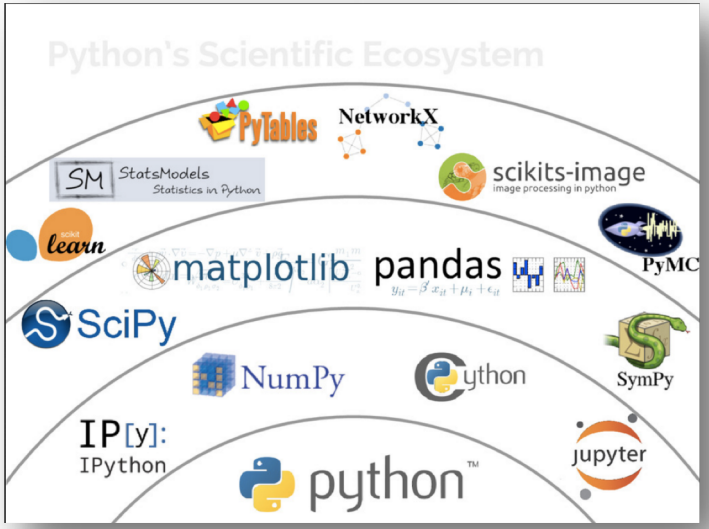
National Synchrotron Light Source II

NSLS-II is a “User Facility”

- Scientific Staff spend 20% time on experiments, the rest on user support
- Users mail samples or visit for ~1–10 days
- 28 active instruments (“beamlines”), 60-70 planned

Deployment on the experimental floor

- Conda environments for each cycle (3 times/year)
- Python 3.6 with full scientific stack: NumPy, SciPy, Matplotlib, Pandas, …
- Facility-independent packages: bluesky, ophyd, caproto, databroker
- Facility-specific package: nslsii
- Our conda channel: lightsource2-tag



- IPython collection/analysis profiles per beamline, **https://github.com/NSLS-II-<BL>/profile_collection** (BL: SIX, HXN, XFM, SRX, QAS, ISS, CHX, CMS, SMI, LIX, AMX, FMX, XFP, ESM, PDF, XPD, and more)

Example: https://github.com/NSLS-II-CHX/profile_collection

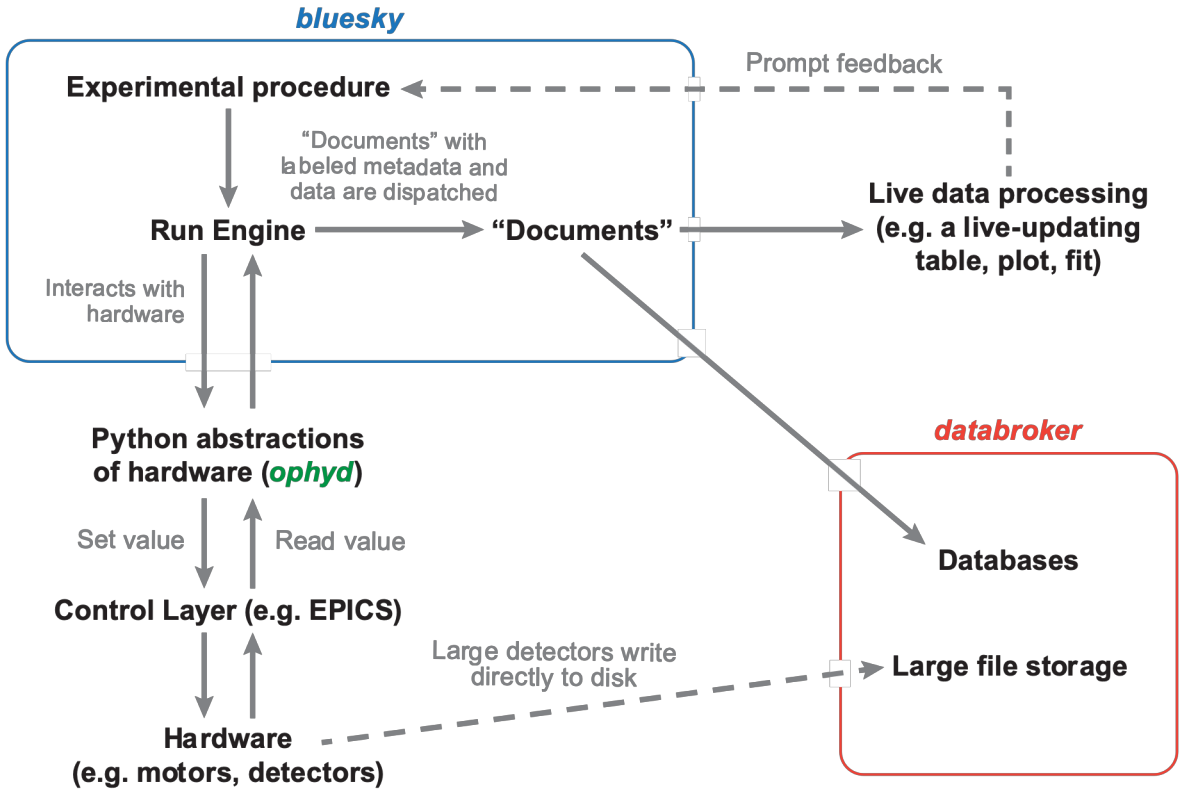
Figure Credit: "State of the Stack" by Jake VanderPlas, SciPy Conference 2015

Software

- Bluesky http://nsls-ii.github.io/bluesky: data collection framework
- Ophyd http://nsls-ii.github.io/ophyd: hardware abstraction library
- Caproto http://nsls-ii.github.io/caproto: pure-Python Channel Access protocol library
- Databroker http://nsls-ii.github.io/databroker: user-friendly interface for retrieving stored data and metadata from multiple sources

Contributors:

- NSLS-II
- APS
- LCLS-II
- ALS



Ophyd: a hardware abstraction layer

- Puts the control layer behind a **high-level interface** with methods like trigger(), read(), and set(...).
- **Group** individual signals into logical “Devices” to be configured and used as one unit.
- Assign signals and devices **human-friendly names** that propagate into metadata.
- **Categorize** signals by “kind” (primary reading, configuration, engineering/debugging).

Ophyd: a hardware abstraction layer

- **Signal:** represents an atomic ‘process variable’. This is nominally a ‘scalar’ value and cannot be decomposed any further by layers above ophyd. In this context an array (waveform) or string would be a scalar because there is no ophyd API to read only part of it.
- **Device:** hierarchy composed of Signals and other Devices. The components of a Device can be introspected by layers above ophyd and may be decomposed to, ultimately, the underlying Signals.
- **Uniform High-level Interface:** all ophyd objects implemented a small set of methods which are used by bluesky plans. It is the responsibility of the *ophyd* objects to correctly implement these methods in terms of the underlying control system.
- **Read-able Interface:** the minimum set of methods an object must implement is

`trigger` ()	Trigger the device and return status object
`read` ()	Read data from the device
`describe` ()	Provide schema and meta-data for `read()`

Optional:

`stage` ()	Stage the device for data collection.
`unstage` ()	Unstage the device.

Ophyd: a hardware abstraction layer

- **Set-able Interface:** the high-level API has the **set** method and a corresponding **stop** method to halt motion before it is complete.

  The **set** method which returns *Status* that can be used to tell when motion is done. It is the responsibility of the *ophyd* objects to implement this functionality in terms of the underlying control system. Thus, from the perspective of the bluesky, a motor, a temperature controller, a gate valve, and software pseudo-positioner can all be treated the same.

- **Configuration:** in addition to values we will want to read, as 'data', or set, as a 'position', there tend to be many values associated with the configuration of hardware, like the velocity of a motor or the chip temperature of a detector. In general these are measurements that are not directly related to the measurement of interest, but maybe needed for understanding the measured data.

`configure` (d, typing.Any])	Configure the device for something during a run
`read_configuration` ()	Dictionary mapping names to value dicts with keys: value, timestamp
`describe_configuration` ()	Provide schema & meta-data for `read_configuration()`

Ophyd: a hardware abstraction layer

- **Fly-able Interface:** there is some hardware where instead of the fine-grained control provided by set, trigger, and read we just want to tell it “Go!” and check back later when it is done. This is typically done when there are needs to coordinated motion or triggering at rates beyond what can reasonably done in via EPICS/Python and tend to be called ‘fly scans’.
- The flyable interface provides four methods

`kickoff` ()	Start a flyer
`complete` ()	Wait for flying to be complete.
`describe_collect` ()	Provide schema & meta-data from `collect()`
`collect` ()	Retrieve data from the flyer as proto-events

Ophyd: a hardware abstraction layer

- **Device and Component:** the core class of ophyd is "Device" which encodes the nodes of the hierarchical structure of the device and provides much of core API.
- The base "Device" is not particularly useful on it's own, it must be sub-classed to provide it with components to do something with.
- Creating a custom device is as simple as

```
from ophyd import Device, EpicsMotor
from ophyd import Component as Cpt

class StageXY(Device):
    x = Cpt(EpicsMotor, ':X')
    y = Cpt(EpicsMotor, ':Y')

stage = StageXY('STAGE_PV', name='stage')
```

Ophyd: a hardware abstraction layer

- **PseudoPositioner:** relates one or more pseudo (virtual) axes to one or more real (physical) axes via forward and inverse calculations. To define such a PseudoPositioner, one must subclass from PseudoPositioner

- Usage:

```
pseudo.forward(px=1, py=2, pz=3)
```

```
from ophyd import (PseudoPositioner, PseudoSingle, EpicsMotor)
from ophyd import (Component as Cpt)
from ophyd.pseudopos import (pseudo_position_argument,
                             real_position_argument)


class Pseudo3x3(PseudoPositioner):
    # The pseudo positioner axes:
    px = Cpt(PseudoSingle, limits=(-10, 10))
    py = Cpt(PseudoSingle, limits=(-10, 10))
    pz = Cpt(PseudoSingle)

    # The real (or physical) positioners:
    rx = Cpt(EpicsMotor, 'XF:31IDA-OP{Tbl-Ax:X1}Mtr')
    ry = Cpt(EpicsMotor, 'XF:31IDA-OP{Tbl-Ax:X2}Mtr')
    rz = Cpt(EpicsMotor, 'XF:31IDA-OP{Tbl-Ax:X3}Mtr')

    @pseudo_position_argument
    def forward(self, pseudo_pos):
        '''Run a forward (pseudo -> real) calculation'''
        return self.RealPosition(rx=-pseudo_pos.px,
                                 ry=-pseudo_pos.py,
                                 rz=-pseudo_pos.pz)

    @real_position_argument
    def inverse(self, real_pos):
        '''Run an inverse (real -> pseudo) calculation'''
        return self.PseudoPosition(px=-real_pos.rx,
                                   py=-real_pos.ry,
                                   pz=-real_pos.rz)
```

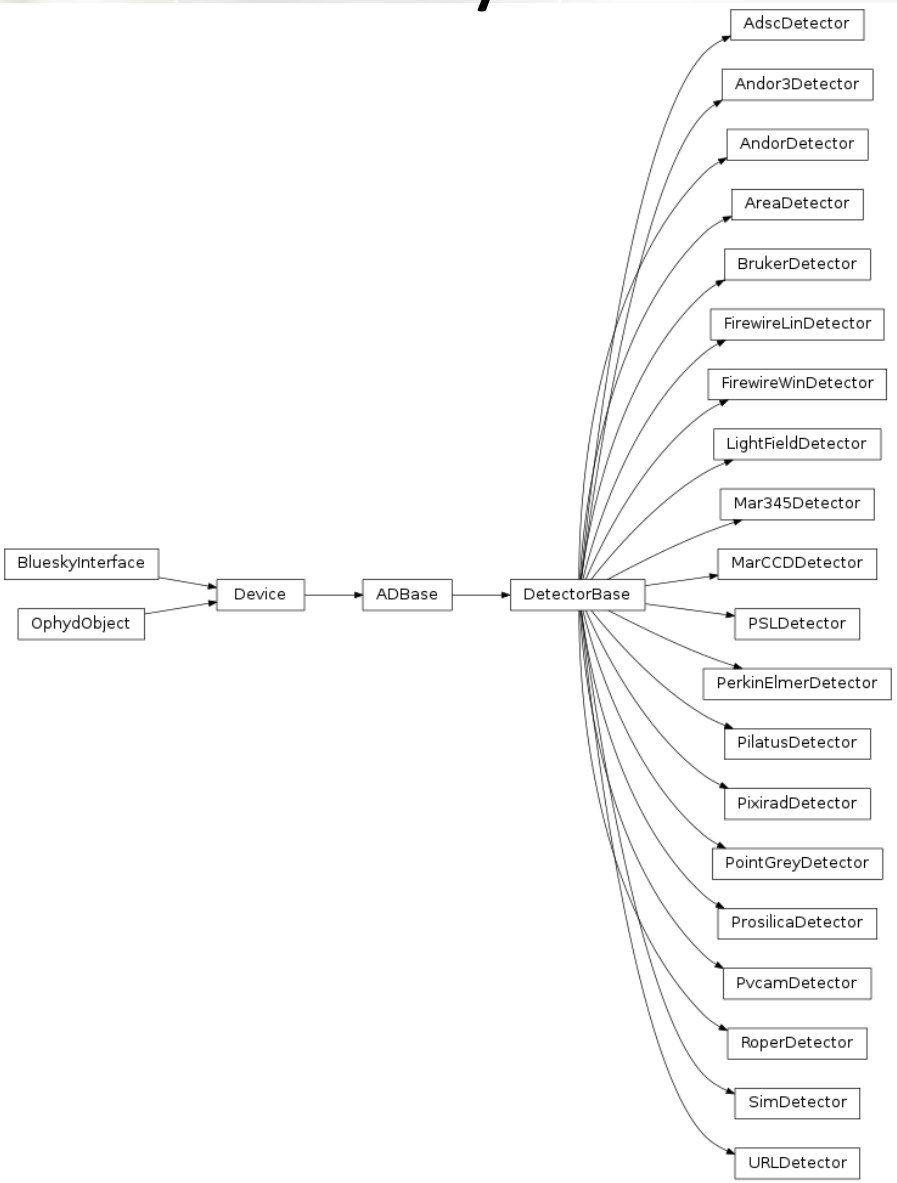
Ophyd: a hardware abstraction layer

- **Area Detectors:** such devices require some customization to use. Here is the simplest possible configuration:

```
from ophyd import AreaDetector, SingleTrigger

class MyDetector(SingleTrigger, AreaDetector):
    pass

prefix = 'XF:23ID1-ES{Tst-Cam:1}'
det = MyDetector(prefix)
```



Bluesky Data Collection Framework

Bluesky is a library for experiment control and collection of scientific data and metadata.

- **Live, Streaming Data:** Available for inline visualization and processing.
- **Rich Metadata:** Captured and organized to facilitate reproducibility and searchability.
- **Experiment Generality:** Seamlessly reuse a procedure on completely different hardware.
- **Interruption Recovery:** Experiments are “rewindable,” recovering cleanly from interruptions.
- **Automated Suspend/Resume:** Experiments can be run unattended, automatically suspending and resuming if needed.
- **Pluggable I/O:** Export data (live) into any desired format or database.
- **Customizability:** Integrate custom experimental procedures and commands, and get the I/O and interruption features for free.
- **Integration with Scientific Python:** Interface naturally with numpy and Python scientific stack.

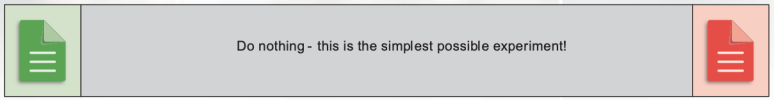
Bluesky Data Collection Framework

- **Plans:** a *plan* is bluesky's concept of an experimental procedure. A plan may be any iterable object (list, tuple, custom iterable class, …) but most commonly it is implemented as a Python *generator*.

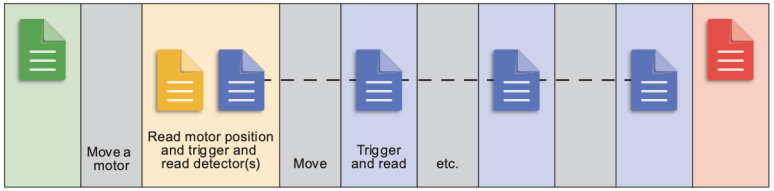
`count`	Take one or more readings from detectors.
`scan`	Scan over one multi-motor trajectory.
`rel_scan`	Scan over one multi-motor trajectory relative to current position.
`list_scan`	Scan over one variable in steps.
`rel_list_scan`	Scan over one variable in steps relative to current position.
`log_scan`	Scan over one variable in log-spaced steps.
`rel_log_scan`	Scan over one variable in log-spaced steps relative to current position.
`grid_scan`	Scan over a mesh; each motor is on an independent trajectory.
`rel_grid_scan`	Scan over a mesh relative to current position.
`scan_nd`	Scan over an arbitrary N-dimensional trajectory.
`spiral`	Spiral scan, centered around (x_start, y_start)
`spiral_fermat`	Absolute fermat spiral scan, centered around (x_start, y_start)
`spiral_square`	Absolute square spiral scan, centered around (x_center, y_center)
`rel_spiral`	Relative spiral scan
`rel_spiral_fermat`	Relative fermat spiral scan
`rel_spiral_square`	Relative square spiral scan, centered around current (x, y) position.
`adaptive_scan`	Scan over one variable with adaptively tuned step size.
`rel_adaptive_scan`	Relative scan over one variable with adaptively tuned step size.
`tune_centroid`	plan: tune a motor to the centroid of signal(motor)
`tweak`	Move and motor and read a detector with an interactive prompt.
`ramp_plan`	Take data while ramping one or more positioners.
`fly`	Perform a fly scan with one or more 'flyers'.

Event-Based Architecture

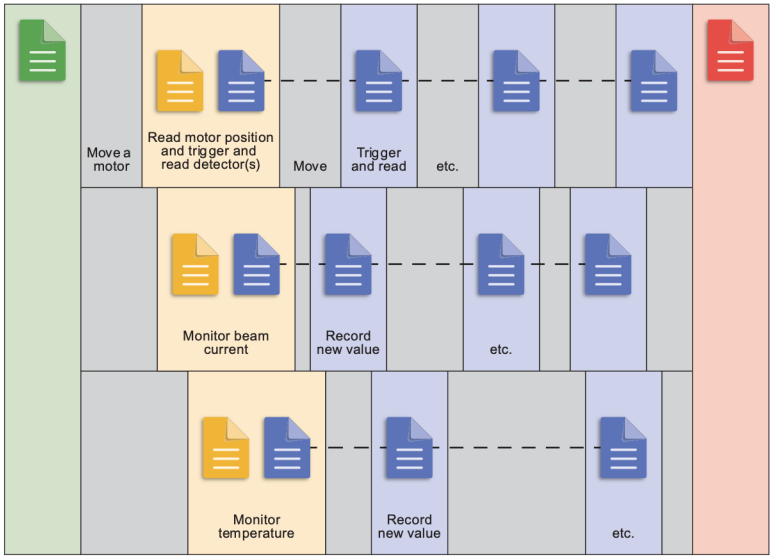
Example 1: Simplest Possible Run



Example 2: A Simple Scan



Example 3: Asynchronously Monitor During a Scan



Run Start: Metadata about this run, including everything we know in advance: time, type of experiment, sample info., etc.

Event: Readings and timestamps

Event Descriptor: Metadata about the readings in the event (units, precision, etc.) and the relevant hardware

Run Stop: Additional metadata known at the end: what time it completed and its exit status (success, aborted, failed)

Ophyd-bluesky protocol

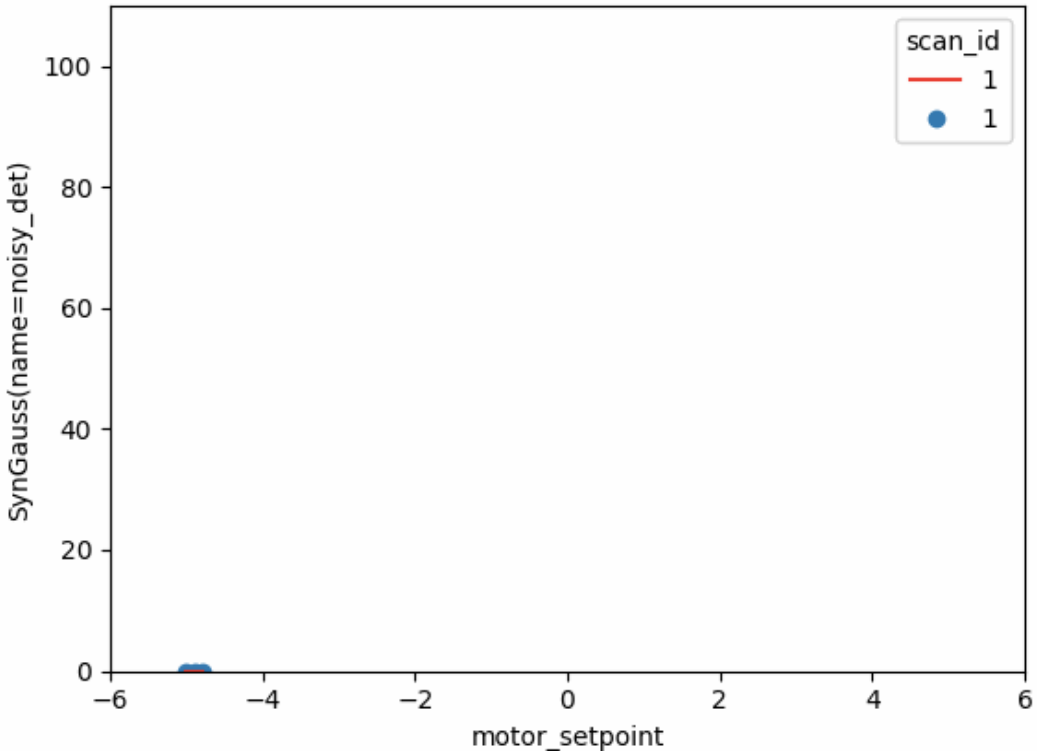
- Bluesky knows nothing about EPICS or any hardware communication.
- Bluesky expects objects with certain methods, comprising this ophyd–bluesky protocol, and only ever interacts with hardware through them.
- This makes it easy to drop in simulated hardware for real hardware, or to implement alternatives to ophyd that work the same with bluesky.

Databroker: rich search and access to saved data

- An API on top of a database (e.g. MongoDB)
- Search on arbitrary user-provided or automatically-captured metadata.
- Streaming-friendly (lazy)
- Exactly the same layout originally emitted by Bluesky, so consumer code does not distinguish between “online” and saved data

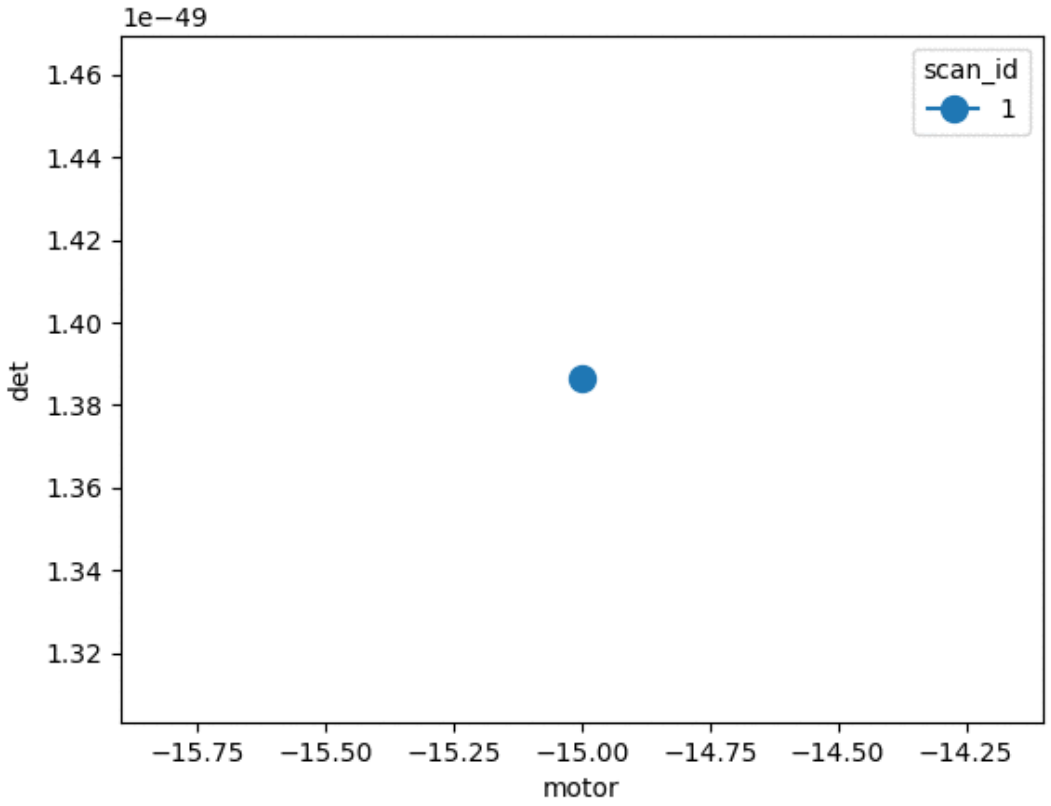
http://nsls-ii.github.io/databroker/tutorial.html

Examples: a Gaussian live-fit to a stream of measured data using the Python library *lmfit* (from U. Chicago / APS)



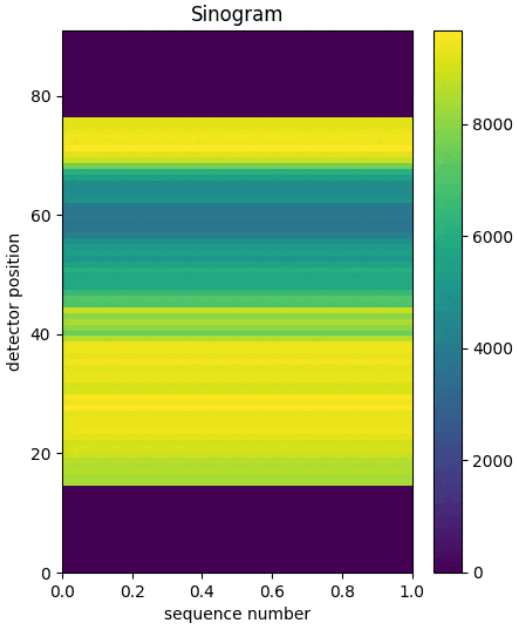
http://nsls-ii.github.io/cookbook/adaptive_fitting.html

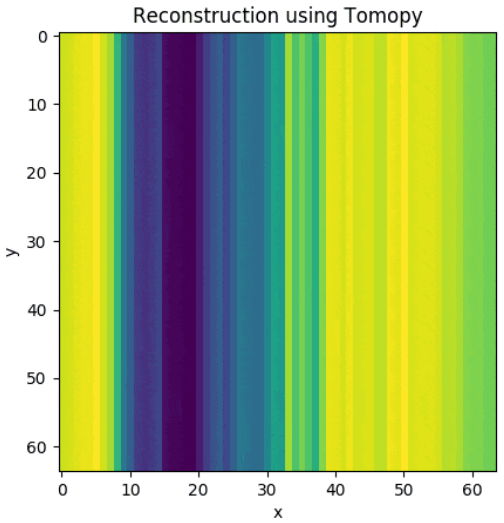
Examples: adaptive scan – each step determined adaptively in response to local slope



http://nsls-ii.github.io/cookbook/adaptive-steps.html

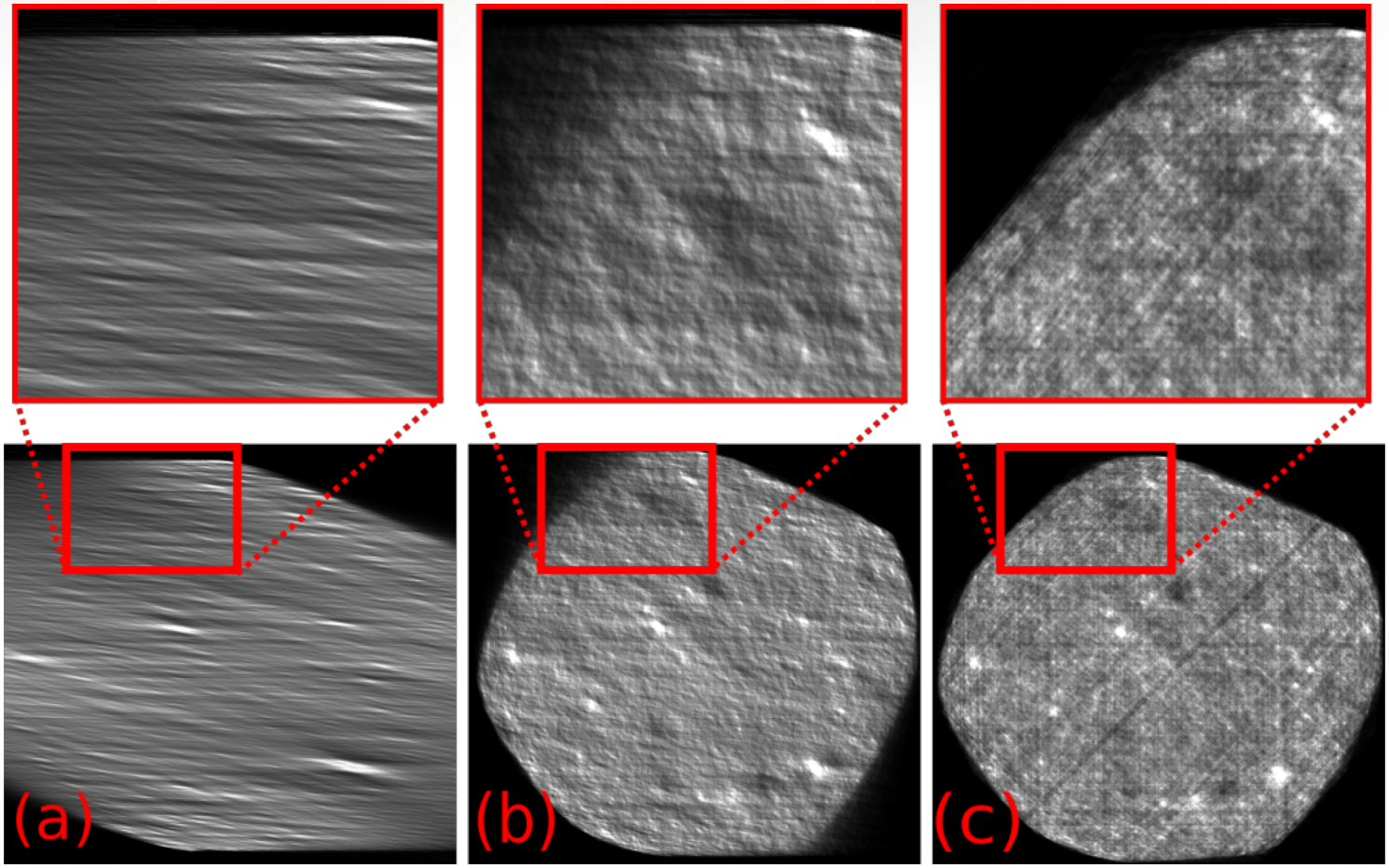
Examples: a stream of slices is tomographically reconstructed using *tomopy* (APS)





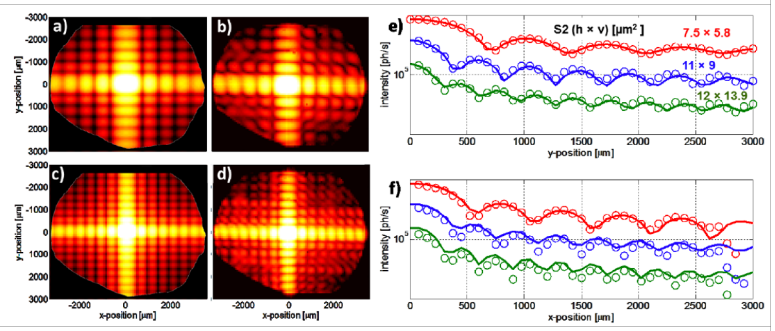
http://nsls-ii.github.io/cookbook/live_recon.html

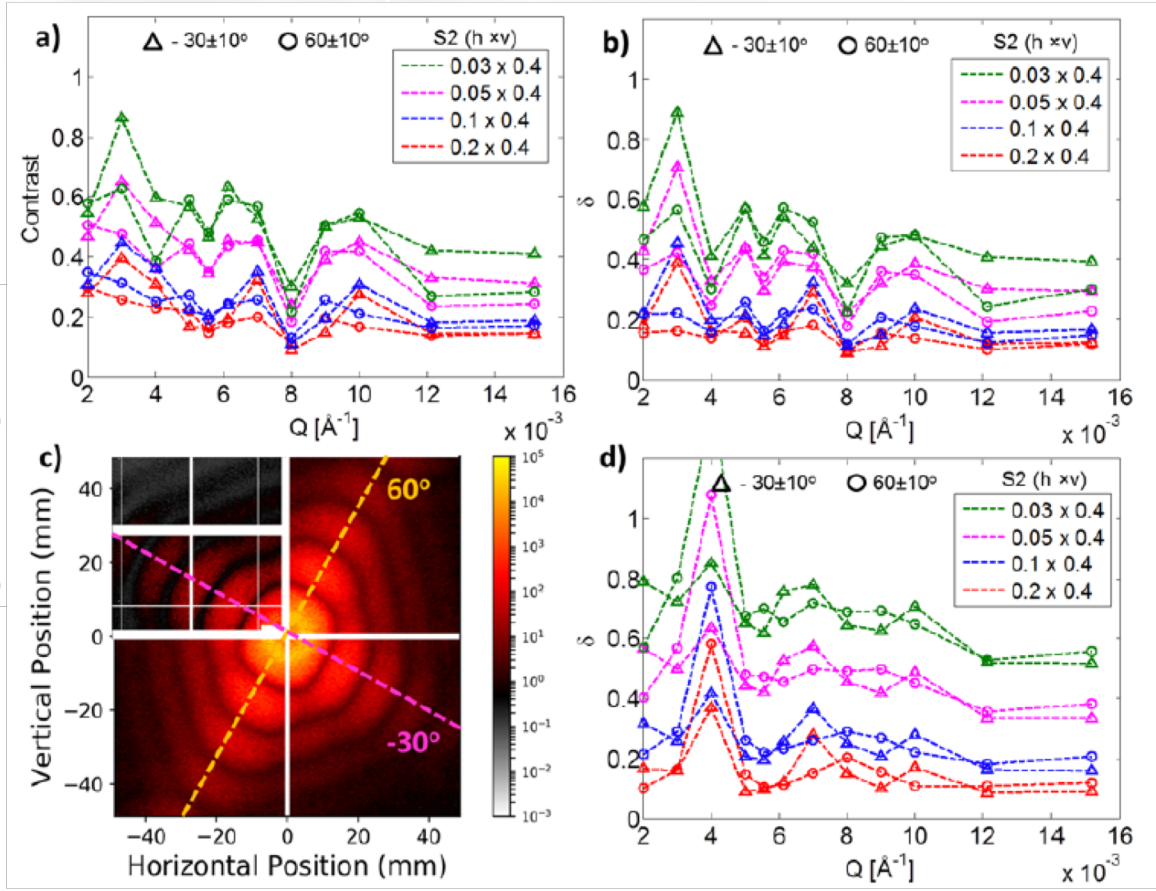
Examples: Real-time Data Analysis at APS



Data is streamed from APS to Argonne Leadership Compute Facility.
Results are immediately visualized at APS.

Examples: Comparison of data from experiments and simulations made easier with DataBroker





Same analysis pipelines were used to compare the experimental and simulation data at NSLS-II CHX beamline for diffraction and scattering (SRI-2018 proceedings).
Simulations were performed with Sirepo/SRW: https://sirepo.com

Live demo

https://github.com/mrakitin/Berlin-2019-tutorials/blob/master/LaptopCam.ipynb

Links

- Gallery of examples: http://nsls-ii.github.io/cookbook
- Bluesky tutorial materials: https://github.com/mrakitin/Berlin-2019-bluesky-tutorial
- Documentation: nsls-ii.github.io
- Code: github.com/NSLS-II