Journey to a Digital Twin
Developing a Python based Digital Shadow at BESSY II

Waheedullah Sulaiman Khail, Pierre Schnizer,

Helmholtz-Zentrum Berlin (HZB), Germany

12. Dezember 2023

Outline

Motivation

Experience @ BESSY II

- Tracy II, thor-scsi & self built / IOC
- experimenting field
- in parallel: measurement scripts on Bluesky

Accelerator community and beyond

- FRIB online model flame [1]
- LUME project of LCLS II
- Functional Mockup Interface
- Open simulation platform [2, 3]

MML Success

- MML success: *one* tool → many machines
- eng↔phys mapping: 1↔1

MML to next level

- eng↔phys mapping: many↔many aka FMI, OSP?
- modelling device(s) response time: aka Bluesky/ophyd UX ↑
- repo pattern: handling configuration data abstraction

Not all answers today: rather a proposal for journey to come

BESSY II twin: current developments

User requests

- *set-values*: magnets, cavities, . . .
- *beam parameters*: e.g. δP

Calculation response

- orbit $\rightarrow$ beam position monitor readings
- optics: twiss function

Current realisation

e.g. magnet setpoints

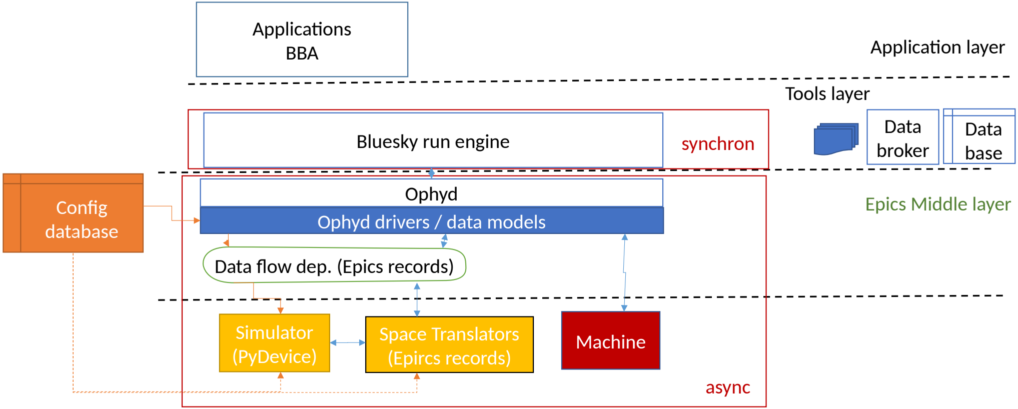
EPICS records

- virtual power converter per magnet
- connected to virtual power converter of “main supply”
- recalculation to main multipole $\leftarrow$ separate

PyDevice based engine

- engine: thor-scsi $\rightarrow$ PyAT
- current refactor: simplified design

BESSY II: overview



Handling configuration data

- Repository pattern
  - access of data
  - independent of storage: e.g. csv files, database, ...
- Identifier (pattern)
  - sole requirements: Hashable
  - PyAT → `typing.SupportsIndex`
  - dataclass: more readable?
- Datamodel?
  - express data structure to others
  - support by IDE's, etc
  - simplifies: storage, display (e.g. mongodb, RestAPI)

Loading parameters: repository pattern

```
class ParameterRepository(AbstractRepository):
    def get(self, reference: AbstractIdentifier) -> Any:
        return self.data[reference]

    def __init__(self, data):
        self.data = data
```

Inspiration

Basis identifier

```
class AbstractIdentifier(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __eq__(self, other: AbstractIdentifier) -> bool:
        raise NotImplementedError

@dataclass(frozen=True, eq=True, kw_only=True)
class MMLIdentifier(AbstractIdentifier):
    family_name: str
    sector: int
    number: int
```

Communication handling: command pattern I/II

Motivation: abstract complex handling

- many tasks:
  - change devices
  - take data
  - analyse
- command interface:
  - e.g. CAD command line interface
  - e.g. Bluesky's RunEngine messages
  - device changes → command → record
  - twinning:
    - test on twin
    - replay to machine
  - optimisation applications, middle layer: → in different languages ← communication?

```
record(ao, "$(PREFIX):$(ELEMENT):Cm:set"){
    field(DTYP, "pydev")
    field(OUT,
      "@update(element_id='$(ELEMENT)',
              property_name='K', value=%VAL%)")
}

def update(*, element_id, property_name, value=None):
    with UpdateContext(...):
        elem_proxy = acc.get_element(element_id)
        elem_proxy.update(property_name, value)
class AcceleratorProxy(AcceleratorInterface):
    def get_element(self, element_id):
        return ElementProxy(self.acc[element_id])

class ElementProxy(ElementInterface):
    def __init__(self, obj):
        self._obj = obj
    def update(self, property_id: str, value):
        """complete the job"""
```

Communication handling: command pattern II/II

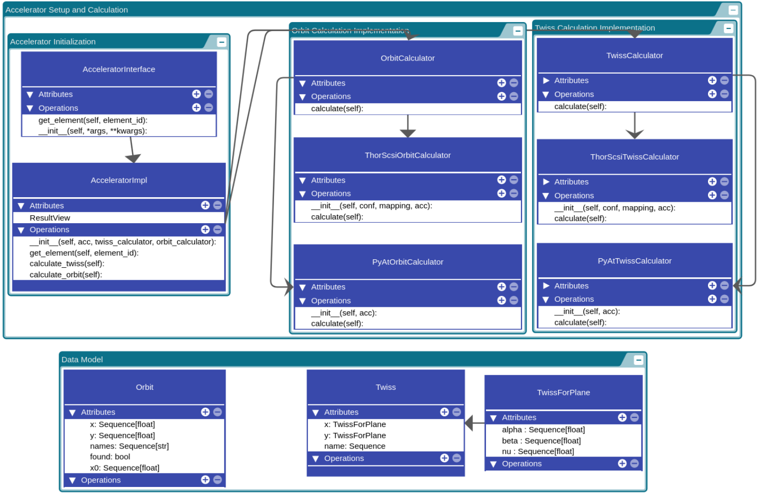
Motivation: abstract complex handling

Command interface

- *single entry point*
- $\rightarrow$ shadow
  - forward command arguments to shadow
  - callers $\rightarrow$ eavesdrop data they receive
    e.g. EPICS: register callbacks
  - slim interface: communication between languages?

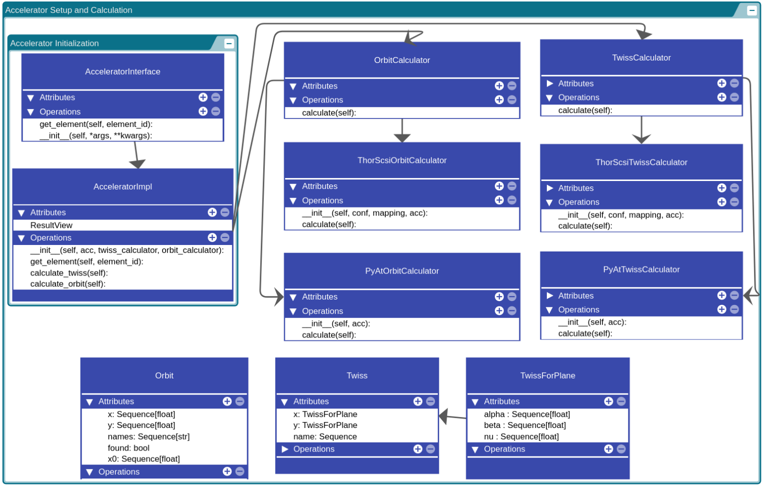
Different back-ends I/II

Proxy / dependency injection



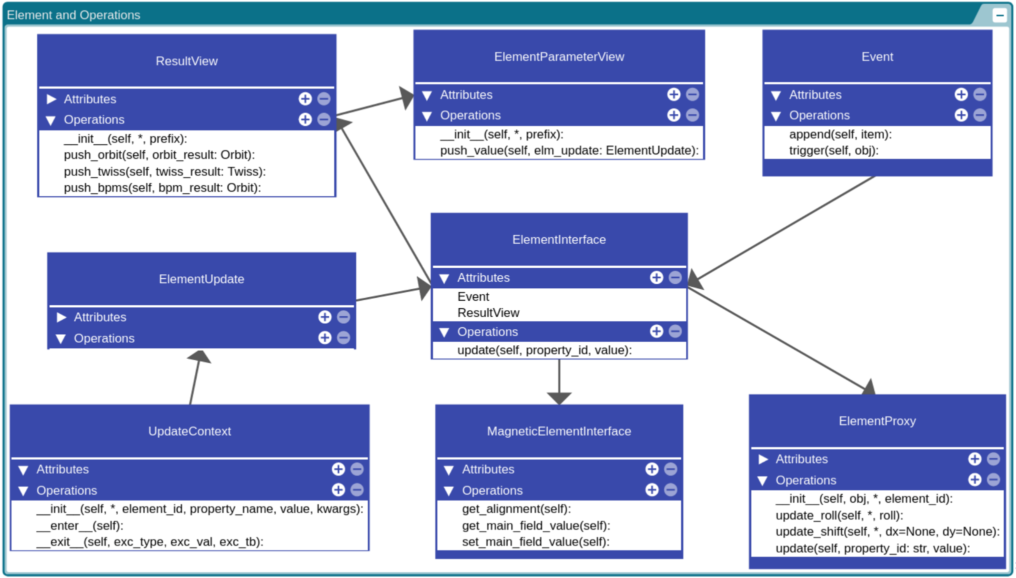
Different back-ends II/III

Proxy / calculators



Different back-ends II/II

Viewer



Waiting for results

Futures, promises and beer garden buzzers

- waiting for new data
- chaining asynchronous actions: e.g.
  - parallel setup
    - prepare injection automaton, injector chain
    - change optics
  - data taking
    - data arrived in device A
    - device B: only accept data after arrival in A
- alternative concepts
  - select
  - poll
  - threads: .run(), .join()

Ophyd: status

```python
class BPM(Device):
    """ensure to return only new data"""
    packed_data = Cpt(EpicsSignalRO, ":bdata")
    count = Cpt(EpicsSignalRO, ":count")
    timeout = Cpt(Signal, name="timeout",
                  value=3, kind=Kind.config)

    def trigger(self):
        def cb(value, old_value, **kwargs):
            # called when new packed data arrive
            # could check for value
            return True

        return SubscriptionStatus(
            self.packed_data, cb, run=False,
            timeout=self.timeout.get())
```

Timely device response: is it an issue?

Devices & time

- higher abstraction levels → automatation of more complex tasks: occasional failure → frequent occurance
- Modelling device response: Ophyd: timeout (settle time)
- → track response: report on failure → UX ↑

Real world example

dynamic aperture measurement / amplitude dependent tune shift

- diagnostic kicker
- turn by turn measurement: occasional failure
- time response superision: program failure when device failure → easier to track culprit

show what was don Python: dropping GIL: get prepared

Middle layer: layering, components

Middle layer: task

- conversion between spaces: eng $\leftrightarrow$ physics : OSP[1] *information layer*
- communication to: machine / simulation: OSP *communication layer*

Layers: advantage

- separation of responsibility / tasks $\rightarrow$ shadow as eavesdropper
- increase flexibility: e.g. communication layer (partly) control system dependent

Current design: review?

- MML engineering $\leftrightarrow$ physics: one to one
- FMU[2] many to many, units cross check

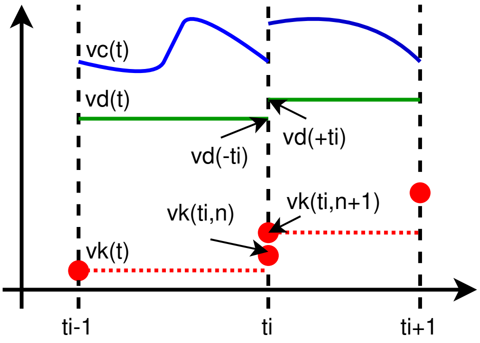
[1] **O**pen **S**imulation **P**latform [2, 3]
[2] **F**unctional **M**ockup **U**nit see https://fmi-standard.org

Functional mockup interface

- split up: different components: standardised
- standard interface:
  - calculation / simulation engine → library
  - interface definiton → XML-file
- simulation engine:
  - differential equations *model exchange*
  - *co simulation*
  - simulation components *scheduled execution*
- concept of time: (including handling that states can switch)

split up simulation components in a standardised fashion

Concept of time



advancing time

Conclusion

- MML: prove many machines ← optimised by single toolkit
- python reimplementation: chance
  - revisit: OSP FMI, LUME,...
  - FMU: components, self describing, unit standard,...
  - OSP: layering: networking, information
  - software patterns: repository, dependency injection
- *target* keep flexibility, machine independence
- use patterns: simplify different use case implementations
- first points?
  - repo pattern ← configuration data management
  - time response: detect early when communication fails
  - shadow by eavesdropping

Z. He, J. Bengtsson, M. Davidsaver, K. Fukushima, G. Shen, and M. Ikegami.
The fast linear accelerator modeling engine for FRIB online model service, 2016.

OSP Interface Specification OSP-IS, Febrary 2022.

Florian Perabo, Daeseong Park, Mehdi Karbalaye Zadeh, Øyvind Smogeli, and Levi Jamt.
Digital twin modelling of ship power and propulsion systems: Application of the open simulation platform (OSP).
In *2020 IEEE 29th International Symposium on Industrial Electronics (ISIE)*, pages 1265–1270, 2020.

Harry Percival and Bob Gregory.
Architecture Patterns with Python.
O'Reilly Media, Inc., 2020.

G. Portmann, J. Corbett, and A. Terbilo.
An accelerator control middle layer using MATLAB.

In *Proceedings of 2005 Particle Accelerator Conference, Knoxville, Tennessee*, pages 4009–11, 2005.

A. Terbilo.
Accelerator toolbox for MATLAB.
Technical report, Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94309 USA, 2001.

M. T. Heron et al.
The DIAMOND light source control system.
In *Proceedings of EPAC 2006, Edinburgh, Scotland*, pages 3068–70, 2006.

Guobao Shen, Lingyun Yang, and Kunal Shroff.
NSLS-II high level application infrastructure and client API design.
In *Proceedings of 2011 Particle Accelerator Conference, New York, NY, USA*, pages 584–4, 2011.
MOP250.

M. Böge and J. Chrin.
A CORBA based client-server model for beam dynamics applications at the SLS.

In *International Conference on Accelerator and Large Experimental Physics Control Systems, 1999, Trieste, Italy*, pages 555–557, 1999.

P. Goryl, A. I. Wawrzyniak, T. Szymocha, and M. Sjöström.
An implementation of the virtual accelerator in the TANGO control system.
In *Proceedings of ICAP2012, Rostock-Warnemünde, Germany*, pages 23–25, 2012.
ISBN 978-3-95450-116-8.

Werner Kritzinger, Matthias Karner, Georg Traar, Jand Henjes, and Wilfried Sihn.
Digital twin in manufacturing: A categorical literature review and classification.

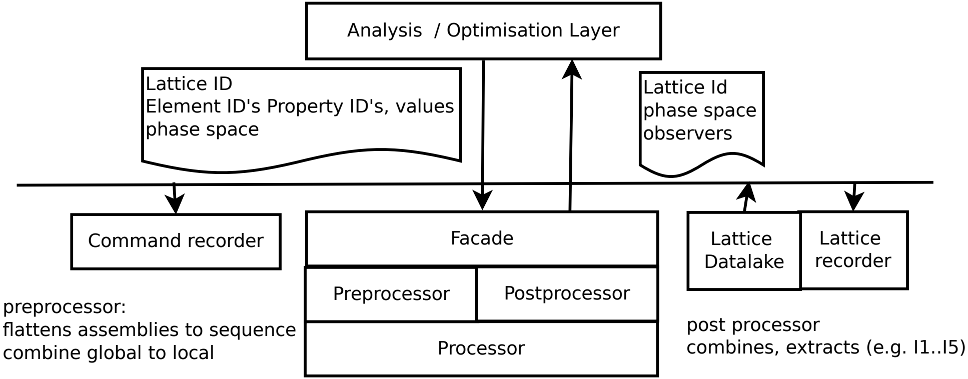
IFAC-PapersOnLine, 51(11):1016–1022, 2018.
16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018.

Automation systems and integration — digital twin framework for manufacturing — part 1: Overview and general principles.
ISO standard.

backup slides

Single particle dynamics: an architecture?

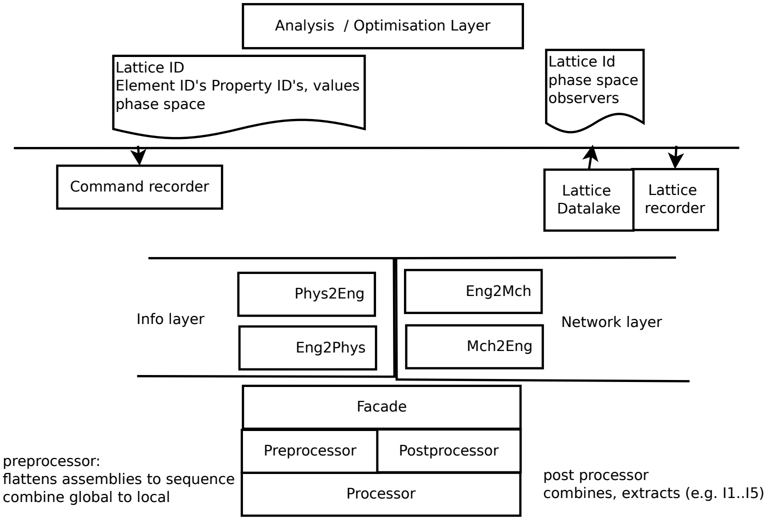
Proposal: overview



Details explained below, influenced by python architecture patterns [4]

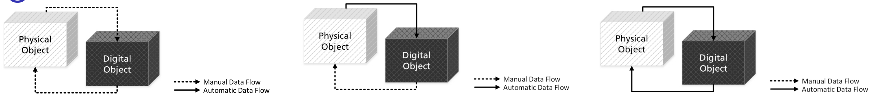
Single particle dynamics: an architecture?

Twinning aspects



Influenced by FMI[2], / OSP[3]

Digital twin: nomen clature



model shadow twin

Status

- different beam dynamics models available ← interaction
- **M**atlab **M**iddle **L**ayer [5] + **A**ccelerator **T**oolbox: [6] similar simulation many different ring light sources
- further online model implementations for ring light sources: DIAMOND [7], NSLS II [8], SLS [9] Solaris [10],
- FLAME: FRIB online model [1]

Need to go further?

- 20 years → experience gained
- software industry → Futures ("beer garden" buzzers), async, μ-services
- compare: iso standard , functional mock-up interface

split up [11], iso standard [12]

Excurs: Bluesky dynamic aperture check

Kicker delay

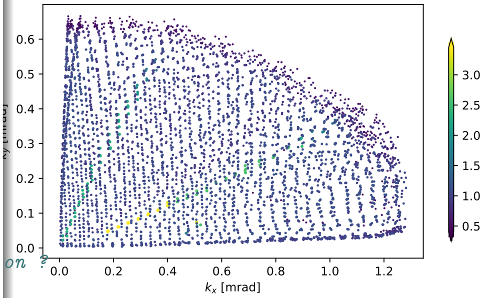
```python
class Delay(Device):
    '''Delay of kicker pulse to trigger'''
    #: delay after the trigger
    offset = Cpt(EpicsSignal, ':offset')

    #: switch the kicker on or off
    switch = Cpt(EpicsSignal, ':switch')

    def stage(self):
        assert(self.switch.get()) # is kicker on ?
        return super().stage()

    def unstage(self):
        return super().unstage()

    def set(self, value):
        return self.offset.set(value)
```



Devices

- kicker: delay,

Excurs: Bluesky dynamic aperture check

Kicker power converter

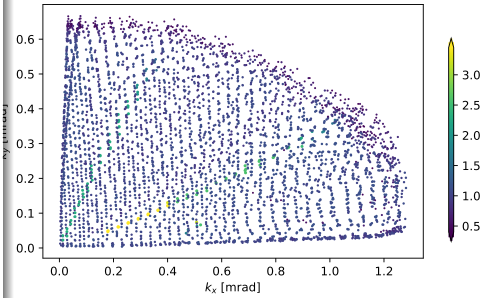
```python
class KickerPS(PVPositioner):
    '''Power converter of the kicker'''
    setpoint = Cpt(EpicsSignal, ':set')
    readback = Cpt(EpicsSignalRO, ':rdbk')
    done = Cpt(EpicsSignalRO, ':done')

    powered = Cpt(EpicsSignalRO, ':stat1')
    hv_on = Cpt(EpicsSignalRO, ':stat2')

    def stage(self):
        assert(self.powered.get())
        assert(self.hv_on.get())
        return super().stage()

    def unstage(self):
        return super().unstage()

    def stop(self, success=False):
        stat = self.setpoint.set(0)
```



Devices

- kicker: delay, pc

Excurs: Bluesky dynamic aperture check

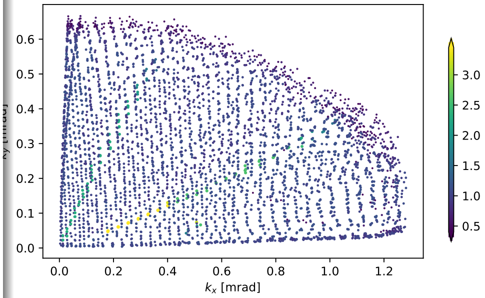
Hor & vert kicker

```
class HKicker(Device):
    '''Horizontal kicker supply and delay
   '''
    delay = Cpt(Delay, 'KDHKR', name='delay')
    ps = Cpt(KickerPS, 'PKDHKR', name='ps',
             timeout=30) # NB: limit

    def set(self, value):
        return self.ps.set(value)

class VKicker(Device):
    '''Vertical kicker supply and delay
    '''
    delay = Cpt(Delay, 'KDVKR', name='delay')
    ps = Cpt(KickerPS, 'PKDVKR', name='ps',
             timeout=30)

    def set(self, value):
        return self.ps.set(value)
```



Devices

- kicker: delay, pc $\rightarrow$ combined

Excurs: Bluesky dynamic aperture check

Turn by turn data

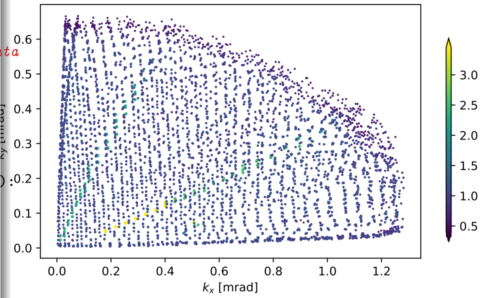
```
class Plane(Device):
    '''waveform and stats for one plane of the turn by turn data
    '''
    #: waveform
    wf = FC(EpicsSignalRO, '{self.prefix}:WF{self._coor}')
    #: peak to peak
    pp = FC(EpicsSignalRO, '{self.prefix}:PP{self._coor}')

    def __init__(self, prefix, coordinate_name=None, **kwargs):
        self._coor = coordinate_name
        super().__init__(prefix, **kwargs)

class SmallBuffer(Device):
    '''On the graphical user intrface: `Free run`
    '''
    x = Cpt(Plane, ':FR', coordinate_name='X')
    y = Cpt(Plane, ':FR', coordinate_name='Y')
    charge = Cpt(EpicsSignalRO, ':FR:WFS')

    def trigger(self):
        def cb(*, value, **kwargs): return True
        x_ready = SubscriptionStatus(self.x.wf, cb, run=False, ti
        y_ready = SubscriptionStatus(self.y.wf, cb, run=False, ti
        stat = AndStatus(x_ready, y_ready)
        return stat

class LiberaBox(Device):
    sht_buf = Cpt(SmallBuffer, prefix='bpmlbz2g', name='small_buffer')
    def trigger(self):  return self.sht_buf.trigger()
```



Devices

- kicker: delay, pc $\rightarrow$ combined
- turn by turn data: readout

Excurs: Bluesky dynamic aperture check

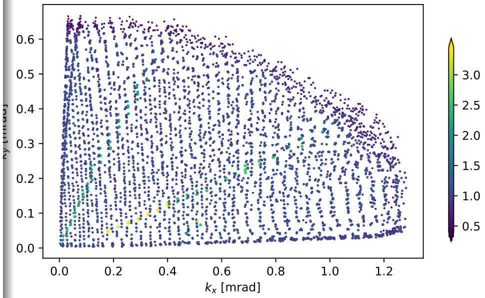
Plan (stub)

```
hk = HKicker(name='hk')
vk = VKicker(name='vk')
lb = LiberaBox(name='lb')

# excute on ray after the other
for ray_num in rays.coords['ray'].values:
    ray_data = rays.sel(dict(ray=ray_num))
    x = ray_data.sel(
        dict(coor='x', num=items)).values
    y = ray_data.sel(
        dict(coor='y', num=items)).values

    x_steps = cycler(hk, x)
    y_steps = cycler(vk, y)

    RE(
        bp.scan_nd(
            [lb, hk, vk], x_steps + y_steps
        )
    )
```



Devices

- kicker: delay, pc $\rightarrow$ combined
- turn by turn data: readout
- "rays from file" $\rightarrow$ plan $\rightarrow$ execute

Excurs bluesky: Lessons learned

Learning curve

- Device programmers
  - python experience: object oriented programming
  - event / call back patterns
  - personal recommendation: implement stop method (first)
- Plan programmers
  - python experience: generators
- Users
  - python experience: coding scripts

A little obstacle

- "epics.PV": for simple measurements → follow "manual work flow"
- Bluesky → "event" based → requires adaption

"Selling" Arguments

- Document structure → automatic storage
- Replay → life plot development
- Device drivers → make available
- Pay back:
  - Complex devices → details abstracted by standard interface
  - Large number of variables → device tree
  - Sophistcated plan stubs → reuse in different scripts