EXPERIMENTAL TESTS OF PYTAC & BLUESKY @ HZB & ESRF

Teresia Olsson & Simone Liuzzo,

Accelerator Middlelayer Workshop, 19-21 June 2024

Software Testing

INTRODUCTION

PURPOSE

- Get hands-on experience of using some already existing python tools (pytac & bluesky).
- Test both at EPICS (HZB) and Tango (ESRF) facility.
- Evaluate **user experience** as input for further discussions about the python middlelayer.
- Put the functionality of these tools in the context of what MML does.
- Test case: orbit response matrix → change steerers and measure the orbit change.
- Note: this is personal experience from learners of the tools.

WHAT DOES MML DO?

MML

- Hide control system names
- Group elements
- Convert between different element name conventions
- Store machine configuration (save/restore)
- Switch operational modes
- Convert between raw and calibrated hardware data
- Convert between hardware and physics units
- Quick measurement scripts
- Standardized scripts for common tasks

  Orbit bumps, tune, chromaticity, response matrix etc.
- High level applications

  Orbit correction, BBA, LOCO etc.
- Handle wait time when set/read devices
- Machine independent
- Collaboration and sharing of tools
- Matlab environment

  Debugger, variable explorer, graphical tools, etc.
- Switch between live and simulator
- Data management

Some points borrowed from Greg Portmann, pyML meeting Feb 12, 2024

WHO ARE THE MML USERS?

- Distributed all over the world.
- Wide range of experience (students → experienced accelerator physicists).
- Different type of machines (storage rings, ramped machines, transfer lines).

Important to consider:

Greg Portmann, "Matlab Middlelayer at Spear3, ALS, Soleil and other Light Sources", pyML meeting, Feb 12 2024

- MML target audience: non-professional programmers
- Who will maintain middlelayer? → at many labs this will be accelerator physicists

PYTAC + ATIP

- Python toolkit for accelerator controls
- Developed at Diamond Light Source (EPICS facility)
- Influenced by MML and APHLA (NSLS-II)
- ATIP – simulator using pyAT

https://github.com/DiamondLightSource/pytac

https://github.com/dls-controls/atip

BLUESKY + OPHYD

- Python toolbox for experiment control and scientific data acquisition
- Collaboration with contributors from many labs
- So far focused on beamlines
- Also so far focused on EPICS but development ongoing for Tango
- Ophyd – layer for hardware abstraction, e.g. devices to use with bluesky

https://blueskyproject.io/

TEST RESULTS

ORM IN MATLAB MIDDLELAYER

- Standardized measurement script: measbpmresp.m
- Functionality:
  - User can use default settings but also customize measurement
  - Automatically handles devices set to bad status
  - Different options for how modulation should be done
  - Different options for how to handle wait time for devices to be ready
  - Handles analysis and data management
  - Either hardware/physics units & live/simulator
  - Pause measurement if current too low and prompt for injection
- Script already quite complex → actual measurement hidden among a lot of setup and data management.
- Did not attempt to implement all of this functionality.

REQUIRED STEPS

- Setup
  - Setup for your machine (import data for elements, channels etc.)
  - Setup the measurement (actuators, monitors, change magnitude etc.)
- Run measurement
  - Set/read devices
  - Wait/sleep to make sure you get correct data
- Data analysis & storage
  - Post-process the measurement results
  - Save the data for the future

PYTAC

- Machine data in CSV files (elements, families, channel names, conversion factors).

Example for BESSY II

Name
- elements.csv
- epics_devices.csv
- families.csv
- simple_devices.csv
- uc_pchip_data.csv
- uc_poly_data.csv
- unitconv.csv

elements.csv

	A	B	C
1	name	type	length
2	MRING_START	Marker	0
3	DU_MSEPEXIT	Drift	0.56
4	MSEPEXIT	Marker	0
5	DU_FOMZ2D1R	Drift	0.0555
6	FOMZ2D1R	Marker	0
7	DU_KIK3D1R	Drift	0.245
8	KIK3D1R	Drift	0.595
9	DU_KIK4D1R	Drift	0.456

epics_devices.csv

	A	B	C	D	E	F
1	el_id	name	field	get_pv	set_pv	
2	0	DCCT	beam_current	MDIZ3T5G:current		
3	11	BPMZ5D1R	x	BPMZ5D1R:rdX		
4	11	BPMZ5D1R	y	BPMZ5D1R:rdY		
5	13	S4PD1R	b2	S4PD1R:rdbk	S4PD1R:set	
6	13	HS4P2D1R	x_kick	HS4P2D1R:rdbk	HS4P2D1R:set	
7	15	Q4PD1R	b1	Q4PD1R:rdbk	Q4PD1R:set	
8	17	S3PD1R	b2	S3PD1R:rdbk	S3PD1R:set	
9	17	VS3P2D1R	y_kick	VS3P2D1R:rdbk	VS3P2D1R:set	
10	19	BPMZ6D1R	x	BPMZ6D1R:rdX		
11	19	BPMZ6D1R	y	BPMZ6D1R:rdY		

unitconv.csv

	A	B	C	D	E	F	G	H
1	el_id	field	uc_type	uc_id	phys_units	eng_units	lower_lim	upper_lim
2	0	s_position	null	0	m	m		
3	0	beta	null	0	m	m		
4	0	dispersion	null	0	m	m		
5	0	beam_current	null	0	A	A		
6	983	f	null	0	Hz	Hz		
7	985	f	null	0	Hz	Hz		
8	989	f	null	0	Hz	Hz		
9	991	f	null	0	Hz	Hz		
10	0	energy	poly	1	GeV	MeV		
11	23	b0	poly	2	m^-1	A	-inf	inf
12	52	b0	poly	2	m^-1	A	-inf	inf
13	97	b0	poly	2	m^-1	A	-inf	inf
14	128	b0	poly	2	m^-1	A	-inf	inf
15	158	b0	poly	2	m^-1	A	-inf	inf
16	187	b0	poly	2	m^-1	A	-inf	inf
17	228	b0	poly	2	m^-1	A	-inf	inf
18	260	b0	poly	2	m^-1	A	-inf	inf
19	290	b0	poly	2	m^-1	A	-inf	inf

- ESRF: conversion factors handled by separate package connected to Tango devices → not tested together with pytac.

HZB BESSY II Light Source

PYTAC

Courtesy of Jean-Luc Pons, ESRF

- Easy to modify to use with Tango.

```
pytac
  .devcontainer
  .github
  .vscode
  build
  docs
  src
    pytac
      data
      __init__.py
      __main__.py
      _version.py
      cothread_cs.py
      cs.py
      data_source.py
      device.py
      element.py
      exceptions.py
      lattice.py
      load_csv.py
      tango_cs.py
      units.py
      utils.py
```

```python
from pytac.load_csv import load
from pytac.tango_cs import TangoControlSystem
```

```python
lattice = load('EBS-vec',
               directory=Path('/machfs/liuzzo/EBS/beamdyn/MDT/2024/2024_06_01/pytac/src/pytac/data'),
               control_system=TangoControlSystem(wait=True, timeout=2.0))
```

pyTAC already ready for other control systems

set_single and read_single methods are the only control system dependent methods

HZB BESSY II Light Source

PYTAC

- Very easy and quick to write a script to get the measurement done.

```
horStrs = lattice.get_elements("HCM")
verStrs = lattice.get_elements("VCM")
```

```
def get_orbit():
    return lattice.get_value('orbit_h'), lattice.get_value("orbit_v")
```

- But need to implement most functionality yourself → code will not handle wait time, current limits etc.
- For example, no status attribute to quickly disable broken devices.
- No data management → code will not save metadata, results etc automatically.

BLUESKY

- Need Bluesky compatible devices (e.g. ophyd) → ready-made devices exist only for beamline components.
- HZB: easy to make simple devices but complex ones require a lot of work.
- ESRF: interface required to Tango, e.g. ophyd-tango (tests ongoing at HZB).

```
class Steerer(Device):
    readback = Component(EpicsSignalRO, ":rdbk")
    setpoint = Component(EpicsSignal,":set")

    def set(self, setpoint):
        """
        Set the setpoint and return a Status object that monitors the readback.
        """

        # Set the new value
        self.setpoint.put(setpoint)

        # Return the Status object
        status = Status(self,success=True, done=True)
        return status
```

A device has components which can be used to connect the device to the correct PVs.

- Devices can handle a lot of functionality (status, wait time etc.) but you need to add it.

BLUESKY

- Difficulty: grouping of devices → sometimes you end up with a chain of devices that you never really wanted.

```python
class AllSteerers(Device):

    # Need to create a dictionary of all the steerers and their PVs to feed into the DynamicDeviceComponent

    def generate_definition(**kwargs):

        steerer_def = {}
        hor_steerer_names = ['HS4M2D1R','HS1MT1R','HS4M1T1R','HS4M2T1R','HS1MD2R','HS4M1D2R','HS4M2D2R','HS1MT2R','HS4
        ver_steerer_names = ['VS3M2D1R','VS2M2D1R','VS2M1T1R','VS3M1T1R','VS3M2T1R','VS2M2T1R','VS2M1D2R','VS3M1D2R','

        for name in hor_steerer_names:
            # PVs named based on power supply name rather than magnet name
            ps_name = name.replace("M", "P")
            steerer_def[name] = (Steerer, ps_name, kwargs)

        for name in ver_steerer_names:
            # PVs named based on power supply name rather than magnet name
            ps_name = name.replace("M", "P")
            steerer_def[name] = (Steerer, ps_name, kwargs)

        return steerer_def

    steerers = DDC(generate_definition())
    selected = Component(SelectedSteerer, name="selected") # This is required to be able to change the value of the se

    # To only read the value of the selected steerer
    _default_read_attrs = ("selected",)

    # Set which steerer that is selected
    def set(self, name):

        # Set the steerer
        self.selected.set_selected(name)

        # Return the Status object
        status = Status(self,success=True, done=True)

        return status
```

DynamicDeviceComponent

```python
class SelectedSteerer(Device):

    steererName = Component(Signal, name="name", value='')

    def set_selected(self, name):
        self.steerer = getattr(self.parent.steerers, name)
        self.steererName.put(name)

    def set(self, value):
        self.steerer.set(value)
        status = Status(self,success=True, done=True)
        return status

    def read(self):
        method = getattr(self.steerer, "read")
        r = method()

        # Need to rename the keys here otherwise key error
        return r
```


HZB BESSY II Light Source

BLUESKY

- Specific workflow for how to run a measurement.
    1. Write a plan
    2. Set up callbacks (live output, live plotting etc.)
    3. Send the plan to the Bluesky run engine.
    4. The run engine will not return the data but a number (uid) which can be used to extract the data from storage.

```
RE = RunEngine()
```

```
bec = BestEffortCallback()
RE.subscribe(bec)
```

```
def scan_steerers(used_steerers, orbit, rel_change):

    for s in used_steerers:
        steerers.set(s)
        yield from rel_scan([], getattr(steerers.steerers,s), -rel_change, rel_change, 2)
```

```
uid = RE( scan_steerers(used_steerers, orbit, rel_change) )
```

- Ready-made plans exist but not entirely adapted for our user cases → we often want to change hundreds of devices after each other instead of a few together.
- But also starting blocks (stubs) exist that can be used to put together your own plan.

BLUESKY

- Bluesky has a whole framework for how to handle data (Documents) → includes both metadata and measurement data.
- A run has three documents (start, event & stop) which are generated by the run engine.
- Can customize how and which data is stored.

- You get data out from a run by using Databroker → e.g. by the uid generated by the run engine.
- Works best when saving to database.
- It should also be possible to write directly to file, but we were not successful.

Two options exist for how to setup the databroker and they return the data in different formats

```
db = Broker.named('temp')
RE.subscribe(db.insert)

catalog = temp()
RE.subscribe(catalog.v1.insert)
```

Old option: pandas dataframe

New option: xarray

BLUESKY

- Issue: you need to make sure that you have collected the data with useful headers.

```
>>> d = pickle.load(open('/machfs/liuzzo/EBS/beamdyn/MDT/2024/2024_06_01/ORMdata_srmag hst-sf2 c10-a
>>> d['table']
                              time                                       positions  strength
seq_num
1       2024-05-30 12:18:32.410099506  [[5.781650598015143e-05, -2.7994524738073546e-...  -0.00001
2       2024-05-30 12:18:38.433176517  [[-5.79179119039193e-05, 2.8087759775409007e-0...   0.00001
```

Very difficult to analyze

Which steerer was changed?
Which position is horizontal/vertical?

seq_num	time	steerers_steerers_HS4M2D1R_readback	steerers_steerers_HS4M2D1R_setpoint
1	2024-06-06 15:05:15.164999962	-0.1	-0.1
2	2024-06-06 15:05:15.167530298	0.1	0.1

Much easier to analyze because you can filter the data using the headers

- Headers are however automatically generated if devices are setup in a good way → data can be understood for years to come.

PYTAC + BLUESKY

- Idea: use the devices already setup by pytac to run with Bluesky.
- Problem: Pytac devices are not Bluesky compatible so needed wrappers around them.

- ESRF: worked but data management complicated → issues with headers in the data.
- HZB: did not work due to conflicts between cothread and Bluesky.

- For this to work well it is better to make pytac compatible with Bluesky.

```python
class Steerer(BlueskyInterface):

    def __init__(self, name, pytacsteerer, field, **kwargs):

        super().__init__( **kwargs)

        self.name = name
        self.parent = None
        self.pytacsteerer=pytacsteerer  # pytac element, used to set/get strengths
        self.field = field

    def trigger(self):
        return Status(success=True, done=True)

    def read(self):
        """Return an OrderedDict mapping string field name(s) to dictionaries
        of values and timestamps and optional per-point metadata.
        """
        print(self.name)
        val = self.pytacsteerer.get_value(self.field)

        return dict(strength=dict(value=val, timestamp=time.time()))

    def describe(self):
        """Return an OrderedDict with exactly the same keys as the ``read``
        method, here mapped to per-scan metadata about each field.
        """
        return dict(strength={'source': 'steerer', 'dtype': 'number', 'shape': []})

    def set(self, val):
        self.pytacsteerer.set_value(self.field, val)
        time.sleep(6) # hard coded for test

    def stage(self, **kwargs):
        super().__init__(**kwargs)
        pass

    def unstage(self):
        pass
```

CONCLUSIONS

WHAT DOES MML DO?

MML

- Hide control system names
- Group elements
- Convert between different element name conventions
- Store machine configuration (save/restore)
- Switch operational modes
- Convert between raw and calibrated hardware data
- Convert between hardware and physics units
- Quick measurement scripts
- Standardized scripts for common tasks — Orbit bumps, tune, chromaticity, response matrix etc.
- High level applications — Orbit correction, BBA, LOCO etc.
- Handle wait time when set/read devices
- Machine independent
- Collaboration and sharing of tools
- Matlab environment — Debugger, variable explorer, graphical tools, etc.
- Switch between live and simulator
- Data management

Pytac = red
Bluesky = purple
Both = orange

WHAT ASPECTS MADE A TOOL EASY TO USE?

- An existing device layer → a lot of work if you have to write devices from scratch and figure out how to make good ones.
- Easy way to group devices together and iterate which one to use.
- Way to run a measurement step-by-step in debugging mode.
- “Someone“ has already setup the data management for you → this is very important and requires careful consideration.

CONCLUSIONS

- Middlelayer has to come with ready-made devices which easily can be grouped together → no “normal” user should have to write a device.
- Conversion between hardware and physics units should be handled in a separate layer.
- We need option to make quick scripts (set/read) but also more complex applications using a full data management framework.
- Input from software engineers is crucial for getting good devices and data management.
- Training has to be fundamental part of the project → both software skills and better data management.
- How to balance choice of software solutions vs users/developers software skills?