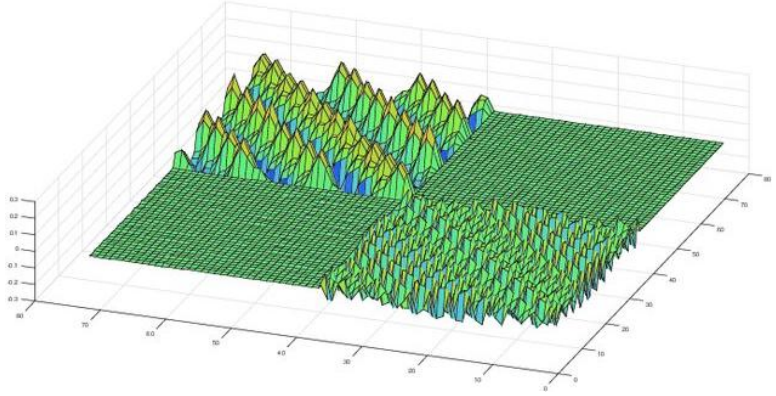
PyAML software architecture

python accelerator middle layer

Joint technology platform for design, commissioning and operation of particle accelerators.



https://github.com/python-accelerator-middle-layer/pyaml

Jean Luc Pons

ESRF | The European Synchrotron

SOFTWARE REQUIREMENTS

PyAML is designed to be 100% control system agnostic

- No control system related code in PyAML
- Tango external bindings (**tango-pyaml**)
- OphydAsync external bindings for Tango and Epics (**pyaml-cs-oa**)

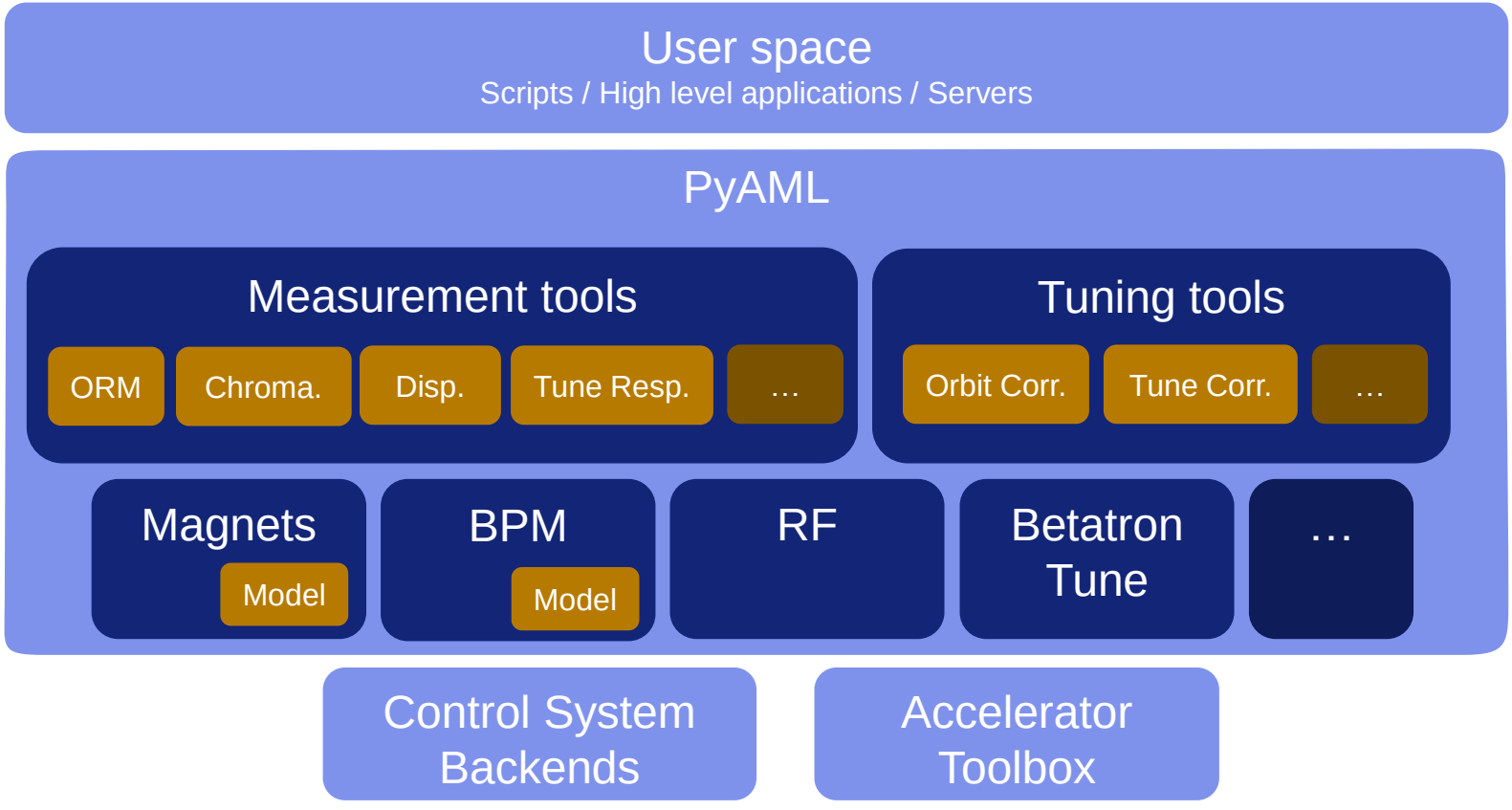
PyAML is designed to allow external plugins

- Custom tuning tools
- Custom measurement tools
- Custom magnet models
- etc…

PyAML is a young project

- Tune response measurement / Tune correction
- Orbit response measurement / Orbit correction (using **pySC**)
- Dispersion measurement (using **pySC**)
- Chromaticity measurement

MAIN SOFTWARE OVERVIEW



PyAML provides a unified interface for accessing a control system or a simulator

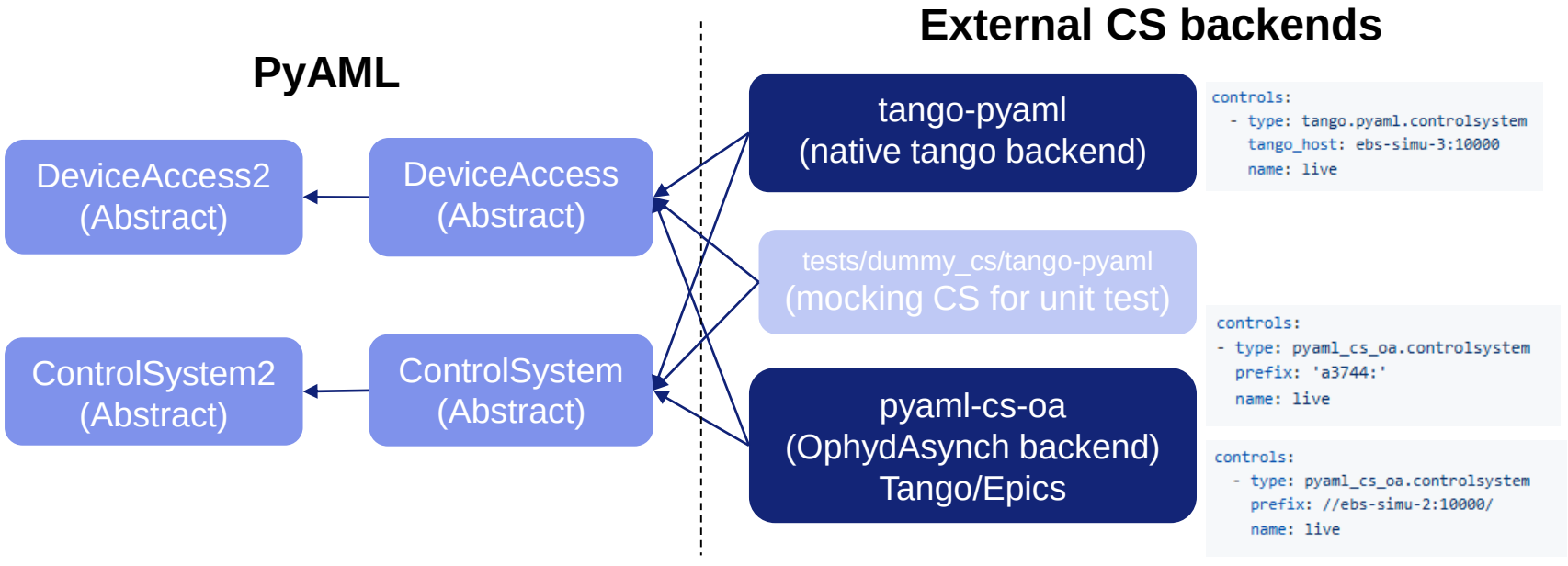
PyAML defines 2 abstract classes to be implemented in CS backends

- The attach methods are not connection method. The CS backend is in charge of returning ***DeviceAccess*** objects attached to a given CS instance (i.e. different TANGO_HOST, different PV prefix, etc…).
- The pydantic configuration models associated to a ***ControlSystem*** or a ***DeviceAccess*** are not a part of PyAML but of the backend

```python
class DeviceAccess(metaclass=ABCMeta):

    # Returns the name of the variable
    def name(self) -> str:

    # Returns the name of the readback variable
    def measure_name(self) -> str:

    # Write control system device variable(s) (i.e. a power supply current)
    def set(self, value):

    # Returns the setpoint(s) of a control system device variable(s)
    def get(self):

    # Returns the measured variable(s)
    def readback(self):

    # Returns the variable unit
    def unit(self) -> str:

    # Returns the range
    def get_range(self) -> list[float]:

    # Check device availabily (ping)
    def check_device_availability(self) -> bool:
```

```python
class ControlSystem(ElementHolder, metaclass=ABCMeta):

    # Return new instances of DeviceAccess objects
    def attach(self, dev: list[DeviceAccess]) -> list[DeviceAccess]:

    # Return new instances of DeviceAccess objects
    def attach_array(self, dev: list[DeviceAccess]) -> list[DeviceAccess]:

    # Return control system name (i.e. live)
    def name(self) -> str:

    # Returns the module name used for handling aggregator of DeviceAccess
    def scalar_aggregator(self) -> str | None:
        return None

    # Returns the module name used for handling aggregator of DeviceAccess
    def vector_aggregator(self) -> str | None:
```

CONTROL SYSTEM ABSTRACTION



- OphydAsynch needs the explicit type of the CS variable (not the native tango backend). This kind of discrepancy should be handled in the CS backend implementation
- Change of PyAML abstract interfaces can be handled by overriding ***DeviceAccess*** and ***ControlSystem*** classes to avoid breaking backward compatibility. CS backends can be updated later to get new features
- Implementation of new CS backend has to be clearly documented

Top level configuration

A simple example with 2 quads

```
from pyaml.accelerator import Accelerator
sr = Accelerator.load("EBSQuads.yaml",use_fast_loader=True)
strAll = sr.live.get_magnets("quads").strengths.get()
strQD2 = sr.live.get_magnet("QD2E-C04").strength.get()
strQF1 = sr.live.get_magnet("QF1E-C04").strength.get()

print(strAll)
print(strQD2)
print(strQF1)
```

outputs

```
[-0.68124609  0.95541258]
-0.6812460852633381
0.9554125829520602
```

Python attributes can be automatically added (if magnet naming style is good in the config) to allow:

`sr.live.quads.strengths.get()`

In order to closely **fulfill** the **specification document**.

```
type: pyaml.accelerator
facility: ESRF
machine: sr
energy: 6e9
simulators:
  - type: pyaml.lattice.simulator
    lattice: config/sr/lattices/ebs.mat
    name: design
controls:
  - type: tango.pyaml.controlsystem
    tango_host: ebs-simu-3:10000
    name: live
data_folder: /data/store
arrays:
  - type: pyaml.arrays.magnet
    name: quads
    elements:
      - QD2E-C04
      - QF1E-C04
devices:
- type: pyaml.magnet.quadrupole
  name: QF1E-C04
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: 1.00054
    crosstalk: 1.0
    curve:
      type: pyaml.configuration.csvcurve
      file: config/sr/magnet_models/QF1_strength.csv
    unit: 1/m
    powerconverter:
      type: tango.pyaml.attribute
      attribute: srmag/vps-qf1/c04-e/current
      unit: A
- type: pyaml.magnet.quadrupole
  name: QD2E-C04
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: 0.999305341
    crosstalk: 0.99912
    curve:
      type: pyaml.configuration.csvcurve
      file: config/sr/magnet_models/QD2_strength.csv
    unit: 1/m
    powerconverter:
      type: tango.pyaml.attribute
      attribute: srmag/vps-qd2/c04-e/current
      unit: A
```

Supported magnets

Normal and skew multipole components are separated to allow the syntax: `mag.strength.set(value)`

- HCorrector / VCorrector
- Quadrupole / SkewQuad
- Sextupole / SkewSext
- Octupole / SkewOctu

Complex combined function magnets and "*virtual*" magnets

When multipoles are not separated at the hardware level, the underlying magnet model needs all strength setpoints to compute coil currents. Virtual magnets act exactly as simple magnets and can be added in arrays.
PyAML is in charge of sending PS setpoints without overlap.

Simple magnet configuration

```yaml
- type: pyaml.magnet.quadrupole
  name: Q3M2T6R # Name in pyaml and in lattice
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: -0.015735
    unit: 1/m
    powerconverter:
      type: pyaml_cs_oa.epicsRW
      write_pvname: Q3P2T6R:set
      read_pvname: Q3P2T6R:rdbk
      unit: A
```

```yaml
- type: pyaml.magnet.quadrupole
  name: Q3MT7R # Name in pyaml
  lattice_names: list(Q3M1T7R,Q3M2T7R) # Serialized or split magnet
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: -0.03147
    unit: 1/m
    powerconverter:
      type: pyaml_cs_oa.epicsRW
      write_pvname: Q3PT7R:set
      read_pvname: Q3PT7R:rdbk
      unit: A
```

```yaml
- type: pyaml.magnet.quadrupole
  name: QF1E-C04
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: 1.00054
    crosstalk: 1.0
    curve:
      type: pyaml.configuration.csvcurve
      file: sr/magnet_models/QF1_strength.csv
    unit: 1/m
    powerconverter:
      type: tango.pyaml.attribute
      attribute: srmag/vps-qf1/c04-e/current
      unit: A
```

Fig 2: Single function magnet

The **type** field refers to a python module. It is dynamically imported when the configuration is loaded.

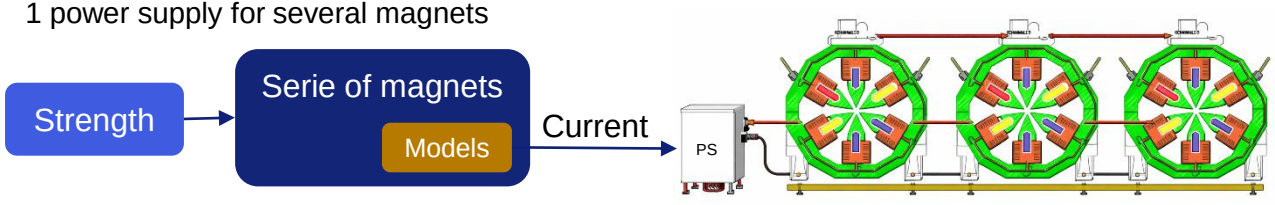
lattice_names : str or None, optional

The name(s) of the associated element(s) in the lattice. By default, the PyAML element name is used. lattice_names accept the following syntax:
- list(name,[name]) : Element names
- [name]@idx[,idx] : Element indices in the subset formed by name.
- [name]#start_idx..end_idx : Element range in the subset formed by name.

In the above syntax, if the name is not specified, the whole set of lattice element is used for indexing.

Magnets in series configuration

1 power supply for several magnets



All magnet can have the same calibration curve, or a specific curve can be applied to each magnet.

```yaml
- type: pyaml.magnet.serialized_magnet
  name: mySeriesOfMagnets
  function: B1
  elements:
    - QF8B-C04
    - QF8D-C04
    - QD5D-C04
    - QF6D-C04
    - QF4D-C04
  model:
    type: pyaml.magnet.linear_serialized_model
    calibration_factors: 1.00504
    calibration_offsets: 0.0
    unit: m-1
    curves: sr/magnet_models/quadcurve.yaml
    powerconverter:
      type: tango.pyaml.attribute
      attribute: srmag/ps-corr-sh1/c01-a-ch1/current
      unit: A
```

```yaml
-   type: pyaml.magnet.serialized_magnet
    name: mySeriesOfMagnets
    function: B1
    elements:
      - QF8B-C04
      - QF8D-C04
      - QD5D-C04
      - QF6D-C04
      - QF4D-C04
    model:
      type: pyaml.magnet.linear_serialized_model
      calibration_factors: [1.00504, 1.00504, 1.00504, 1.00504,
1.00504]
      calibration_offsets: [0.0, 0.0, 0.0, 0.0, 0.0]
      unit: m-1
      curves:
        - sr/magnet_models/quadcurve1.yaml
        - sr/magnet_models/quadcurve2.yaml
        - sr/magnet_models/quadcurve3.yaml
        - sr/magnet_models/quadcurve4.yaml
        - sr/magnet_models/quadcurve5.yaml
      powerconverter:
        type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sh1/c01-a-ch1/current
        unit: A
```

Setting the strength on one magnet computes the required current for that magnet and then applies this current to all magnets.

```python
sr: Accelerator = Accelerator.load(sr_file)
magnets = [
    sr.design.get_element("QF8B-C04"),
    sr.design.get_element("QF8B-C04"),
    sr.design.get_element("QD5D-C04"),
    sr.design.get_element("QF6D-C04"),
    sr.design.get_element("QF4D-C04"),
]

magnets[3].strength.set(0.6)
# Strengths may not all be equal, depending on the magnet calibration
parameters
strengths = [magnet.strength.get() for magnet in magnets]

# All currents must be equal; this is what binds the magnets together.

currents = [magnet.hardware.get() for magnet in magnets]
```

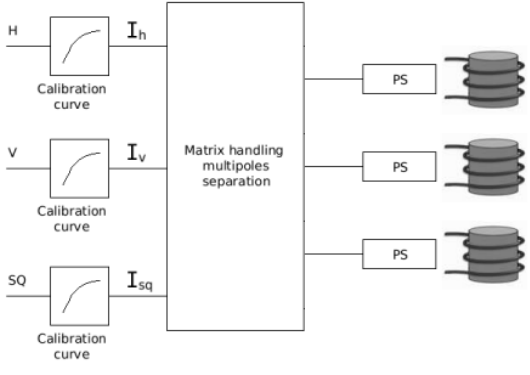
Combined function magnet configuration

```yaml
# Horizontal corrector on sextupole
- type: pyaml.magnet.hcorrector
  name: HS4M2D1R # pyaml name
  lattice_names: S4M2D1R@0
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: -0.0044502
    unit: rad
    powerconverter:
      type: pyaml_cs_oa.epicsRW
      read_pvname: HS4P2D1R:rdbk
      write_pvname: HS4P2D1R:set
      unit: A
# Sextupole
- type: pyaml.magnet.sextupole
  name: S4M2D1R # pyaml name
  model:
    type: pyaml.magnet.linear_model
    calibration_factor: -0.015735
    unit: 1/m^2
    powerconverter:
      type: pyaml_cs_oa.epicsRW
      read_pvname: S4P2D1R:rdbk
      write_pvname: S4P2D1R:set
      unit: A
```

Simple HCorrector + Sextupole with 2 independent PS that control each multipole

Complex HCorrector + VCorrector + SkewQuad with 3 PS and a multipole separation matrix. On the side of the PyAML development, hcorr is a `virtual` magnet.

```
hcorr = sr.live.get_magnet("SH2B-C04-H")
# The underlying model will recompute all 3 currents
# to get the desired H strength without affecting the others
hcorr.strength.set(value)
```



@ESRF, we call I_h, I_v and I_{sq} pseudo currents.

```yaml
- type: pyaml.magnet.cfm_magnet
  name: SH2B-C04
  mapping:  # map each multipole to virtual magnets
    - [B0, SH2B-C04-H]
    - [A0, SH2B-C04-V]
    - [A1, SH2B-C04-SQ]
  model:
    type: pyaml.magnet.linear_cfm_model
    multipoles: [B0,A0,A1]
    units: [rad,rad,m-1]
    pseudo_factors: [1.0,-1.0,-1.0]
    curves:
      - type: pyaml.configuration.csvcurve
        file: sr/magnet_models/SH2_h_strength.csv
      - type: pyaml.configuration.csvcurve
        file: sr/magnet_models/SH2_v_strength.csv
      - type: pyaml.configuration.csvcurve
        file: sr/magnet_models/SH2_sq_strength.csv
    matrix:
      type: pyaml.configuration.csvmatrix
      file: sr/magnet_models/SH_matrix.csv
    powerconverters:
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sh2/c04-b-ch1/current
        unit: A
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sh2/c04-b-ch2/current
        unit: A
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sh2/c04-b-ch3/current
        unit: A
```

External magnet model (example EBS Sextupole model)

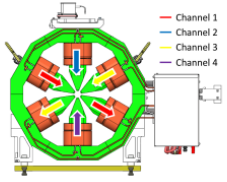


Figure 1: Design view of a combined sextupole-corrector magnet. Each pole is equipped with one sextupole coil and one correction coil. All sextupole coils are connected in series, while correction coils are connected according to the arrows.

Design and C++ code by G. Le Bec,
https://proceedings.jacow.org/ipac2021/papers/tupab369.pdf
https://github.com/python-accelerator-middle-layer/ebs_sextupole

External module

```yaml
- type: pyaml.magnet.cfm_magnet
  name: SJ2A-C04
  mapping:
    - [B2, SJ2A-C04-S]
    - [B0, SJ2A-C04-H]
    - [A0, SJ2A-C04-V]
    - [A1, SJ2A-C04-SQ]
  model:
    type: ebs_sextupole.sextu_model
    strength_data: /operation/control/infra/equipment/magnets/MagnetModel/Parameters/SF2_meas_strengths
    param_file: /operation/control/infra/equipment/magnets/MagnetModel/Parameters/SF2_params.csv
    serial_number: SF2-16246
    powerconverters:
      - type: tango.pyaml.attribute
        attribute: srmag/vps-sf2/c04-a/current
        unit: A
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sf2/c04-a-ch01/current
        unit: A
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sf2/c04-a-ch02/current
        unit: A
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sf2/c04-a-ch03/current
        unit: A
      - type: tango.pyaml.attribute
        attribute: srmag/ps-corr-sf2/c04-a-ch04/current
        unit: A
```

```cpp
namespace py = pybind11;

void *mag_create(std::string strength_file_name, std::string param_file_name, std::string mag_s_n) {

    MagnetModel::Sextupole *sextu = new MagnetModel::Sextupole();
    sextu->init(strength_file_name,param_file_name,mag_s_n);
    return sextu;

}

std::vector<double> compute_strengths(void *ptr,double magnet_rigidity_inv,std::vector<double>& in_currents) {

  MagnetModel::Sextupole *s = (MagnetModel::Sextupole *)ptr;
  std::vector<double> out_strengths;
  s->compute_strengths(magnet_rigidity_inv,in_currents,out_strengths);
  return out_strengths;

}

std::vector<double> compute_currents(void *ptr,double magnet_rigidity,std::vector<double>& in_strengths) {

  MagnetModel::Sextupole *s = (MagnetModel::Sextupole *)ptr;
  std::vector<double> out_currents;
  s->compute_currents(magnet_rigidity,in_strengths,out_currents);
  return out_currents;

}

PYBIND11_MODULE(ebs_sextupole_bind, m) {
    m.doc() = "EBS Sextupole magnet model for PyAML";
    m.def("mag_init", &mag_create,py::return_value_policy::reference);
    m.def("compute_strengths",&compute_strengths);
    m.def("compute_currents",&compute_currents);
}
```

BPM configuration

A simple BPM with 2 distinct Tango attributes for H & V positions.

```
- type: pyaml.bpm.bpm
  name: BPM_C32-02
  model:
    type: pyaml.bpm.bpm_simple_model
    x_pos:
      type: tango.pyaml.attribute_read_only
      attribute: srdiag/bpm/c32-02/SA_HPosition
      unit: m
    y_pos:
      type: tango.pyaml.attribute_read_only
      attribute: srdiag/bpm/c32-02/SA_VPosition
      unit: m
```

A simple indexed BPM with 1 Epics PV for the whole H & V orbit

```
- type: pyaml.bpm.bpm
  name: BPMZ7D1R
  model:
    type: pyaml.bpm.bpm_simple_model
    x_pos_index: 4
    y_pos_index: 5
    x_pos:
      type: pyaml_cs_oa.epicsR
      read_pvname: ORBITCC:rdPos
      unit: nm
    y_pos:
      type: pyaml_cs_oa.epicsR
      read_pvname: ORBITCC:rdPos
      unit: nm
```

BPM offsets and tilt are handled in the same way

```
bpm = sr.design.get_bpm("BPM_C32-02")
pos = bpm.positions.get()
```

RF configuration

```yaml
- type: pyaml.rf.rf_plant
  name: RF
  masterclock:
    type: tango.pyaml.attribute
    attribute: sy/ms/1/Frequency
    unit: Hz
  transmitters:
    - type: pyaml.rf.rf_transmitter
      name: RFTRA1
      cavities: [CAV_C05_01,CAV_C05_02,CAV_C05_03,...,CAV_C07_01,CAV_C07_02,CAV_C07_03,CAV_C07_04,CAV_C07_05]
      harmonic: 1
      distribution: 0.83333333333
      voltage:
        type: tango.pyaml.attribute
        attribute: sr/tra/1/RfVoltage
        unit: V
    - type: pyaml.rf.rf_transmitter
      name: RFTRA2
      cavities: [CAV_C25_01,CAV_C25_02]
      harmonic: 1
      distribution: 0.16666666666
      voltage:
        type: tango.pyaml.attribute
        attribute: sr/tra/2/RfVoltage
        unit: V
    - type: pyaml.rf.rf_transmitter
      name: RFTRA_HARMONIC
      cavities: [CAV_C25_03]
      harmonic: 4
      voltage:
        type: tango.pyaml.attribute
        attribute: sr/tra/harm/RfVoltage
        unit: V
```

RF transmitters definition is optional.

When RF transmitters are not defined, PyAML default to AT `set_rf_frequency()` and `set_rf_voltage()` methods for the design, setting of RF voltage on the live is no longer possible.

```python
sr = Accelerator.load("tests/config/EBS_rf.yaml")
RF = sr.design.get_rf_plant("RF")
RF.frequency.set(3.523e8)
RF.voltage.set(10.0e6)
```

Tune monitor configuration

A simple transverse tune monitor with 2 distinct Tango attributes for H & V tunes.

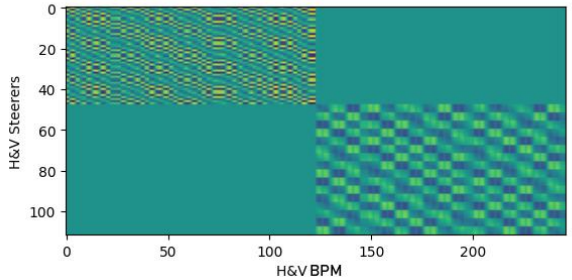
```yaml
- type: pyaml.diagnostics.tune_monitor
  name: BETATRON_TUNE
  tune_h:
    type: tango.pyaml.attribute_read_only
    attribute: srdiag/beam-tune/main/Qh
    unit: "1"
  tune_v:
    type: tango.pyaml.attribute_read_only
    attribute: srdiag/beam-tune/main/Qv
    unit: "1"
```

```python
tm = sr.design.get_betatron_tune_monitor("BETATRON_TUNE")
tune = tm.tune.get()
```

ORM measurement

Configuration

```yaml
- type: pyaml.tuning_tools.orbit_response_matrix
  bpm_array_name: BPM
  hcorr_array_name: HCorr
  vcorr_array_name: VCorr
  corrector_delta: 1e-6
  name: DEFAULT_ORBIT_RESPONSE_MATRIX
```



Callback workflow

```
for corr in correctors:
   set k0 - delta    -------> ACTION_APPLY
   get orbit         -------> ACTION_MEASURE
   set k0 + delta    -------> ACTION_APPLY
   get orbit         -------> ACTION_MEASURE
   set k0            -------> ACTION_RESTORE
```

Callback is optional

```python
from pyaml.common.constants import ACTION_MEASURE,ACTION_RESTORE
from pyaml.accelerator import Accelerator
import numpy as np
import matplotlib.pyplot as plt
import logging

# disable printing during ORM measurement to illustrate callback.
logger = logging.getLogger("pyaml.tuning_tools.orbit_response_matrix").setLevel(
    logging.WARNING
)

# Orbit callback
def orbit_callback(action: int, cdata):
    if action == ACTION_RESTORE:
        # cdata contains measurement data for a given step
        # It can be used by a high level application or a server to
        # be updated at each step of the scan
        i = cdata.last_number
        n_bpms = cdata.raw_up.shape[0] // 2
        n_correctors = cdata.raw_up.shape[1]
        corrector = cdata.last_input
        response = (cdata.raw_up[:, i] - cdata.raw_down[:, i]) / cdata.inputs_delta[i]
        std_x_resp = np.std(response[:n_bpms])
        std_y_resp = np.std(response[n_bpms:])
        print(
            f"[{i}/{n_correctors}], Measured response of {corrector}: "
            f"r.m.s H.: {std_x_resp:.2f} um/mrad, r.m.s. V: {std_y_resp:.2f} um/mrad"
        )
    return True

# Load the accelrator
sr = Accelerator.load("BESSY2Orbit.yaml")

# Mesure the ORM
sr.design.orm.measure(callback=orbit_callback)

# Plot
orm = sr.design.orm.get()
mat = np.array(orm["matrix"])
plt.imshow(mat.T)
plt.ylabel("H&V Steerers")
plt.xlabel("H&V BPM")
plt.show()
```

```
29 Jan% 2026, 13:59:32 | WARNING | PyAML OA control system binding (0.1.1) initialized with name 'live'
[0/112], Measured response of HS4M2D1R: r.m.s H.: 8.65 um/mrad, r.m.s. V: 0.00 um/mrad
[1/112], Measured response of HS1MT1R: r.m.s H.: 8.63 um/mrad, r.m.s. V: 0.00 um/mrad
[2/112], Measured response of HS4M1T1R: r.m.s H.: 7.52 um/mrad, r.m.s. V: 0.00 um/mrad
...
```

Run measurement tool in a Tango Server

- pyTango server interface is rather simple
- Allow to plug PyAML based servers to existing (ATK or Taurus) applications

Initialization

(SkewQuad error has been added to add a bit of coupling)

```
def init_device(self):
    """Initializes the attributes and properties of the ORM."""
    global DEVICE
    Device.init_device(self)
    DEVICE = self

    self.SR = Accelerator.load(self.ConfigFileName)
    self.orm_data = None
    nb_hsteer = len(self.SR.design.get_magnets("HCorr"))
    nb_vsteer = len(self.SR.design.get_magnets("VCorr"))
    nb_skew = len(self.SR.design.get_magnets("Skews"))
    skewErr = 1e-3 * np.random.normal(size=nb_skew)
    self.SR.design.get_magnets("Skews").strengths.set(skewErr)
    self.progress_data = [0] * 2 * (nb_hsteer + nb_vsteer)
    self.set_status(f"Ready to scan: {self.ConfigFileName}")
    self.set_state(DevState.ON)
```

Start command

```
@command()
@DebugIt()
def Start(self):
    if self.get_state() == DevState.ON:
        t1 = threading.Thread(target=self.orm_run)
        t1.start()
        self.set_state(DevState.MOVING)
    else:
        raise ValueError("A scan is in already progress")
```

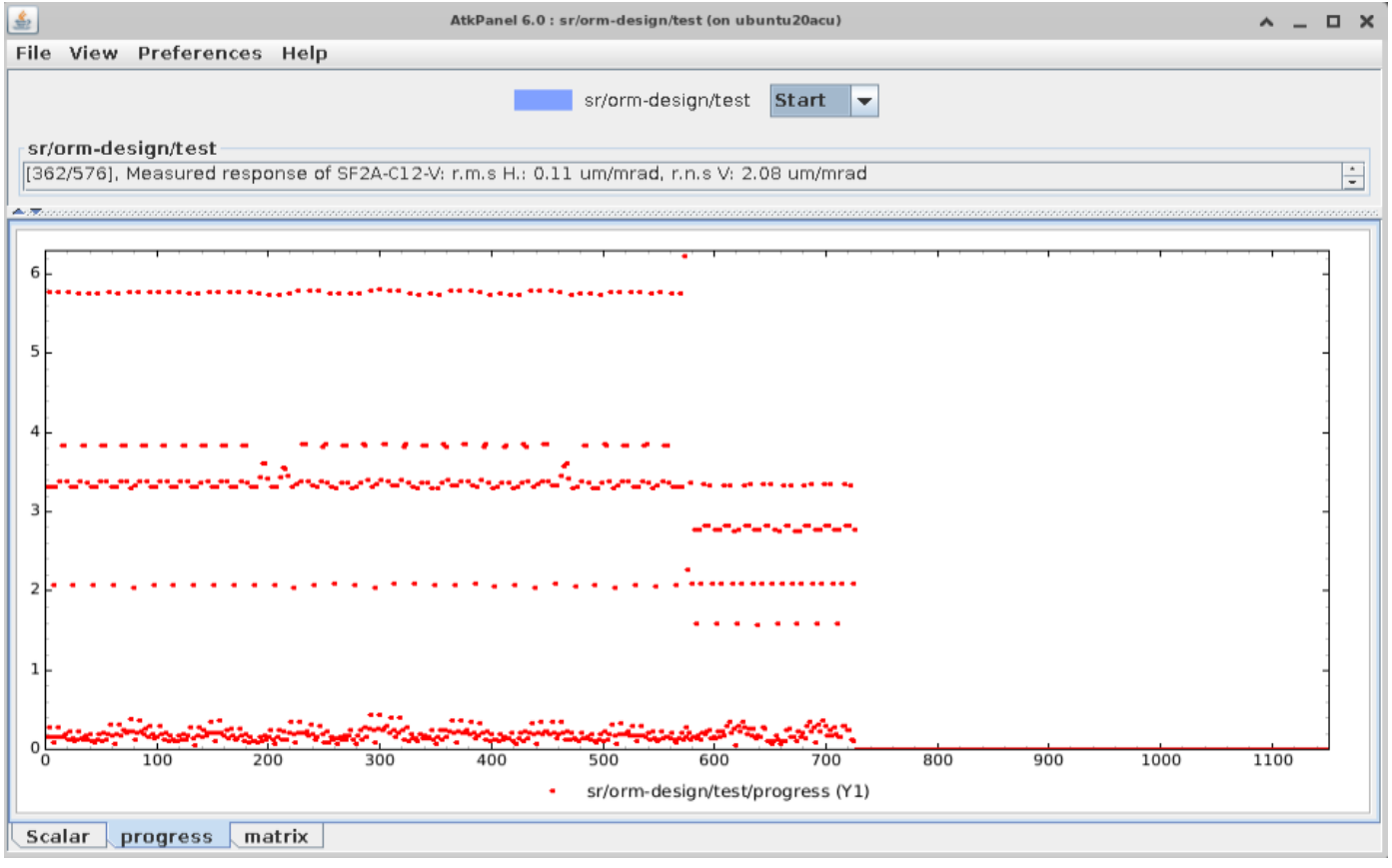
ORM thread

```
def orm_run(self):
    # On design sleep 10ms to allow thread scheduling
    self.SR.design.orm.measure(callback=orbit_callback, set_wait_time=0.01)
    self.orm_data = self.SR.design.orm.get()
    self.set_status(f"Ready to scan: {self.ConfigFileName}")
    self.set_state(DevState.ON)
```

Matrix attribute

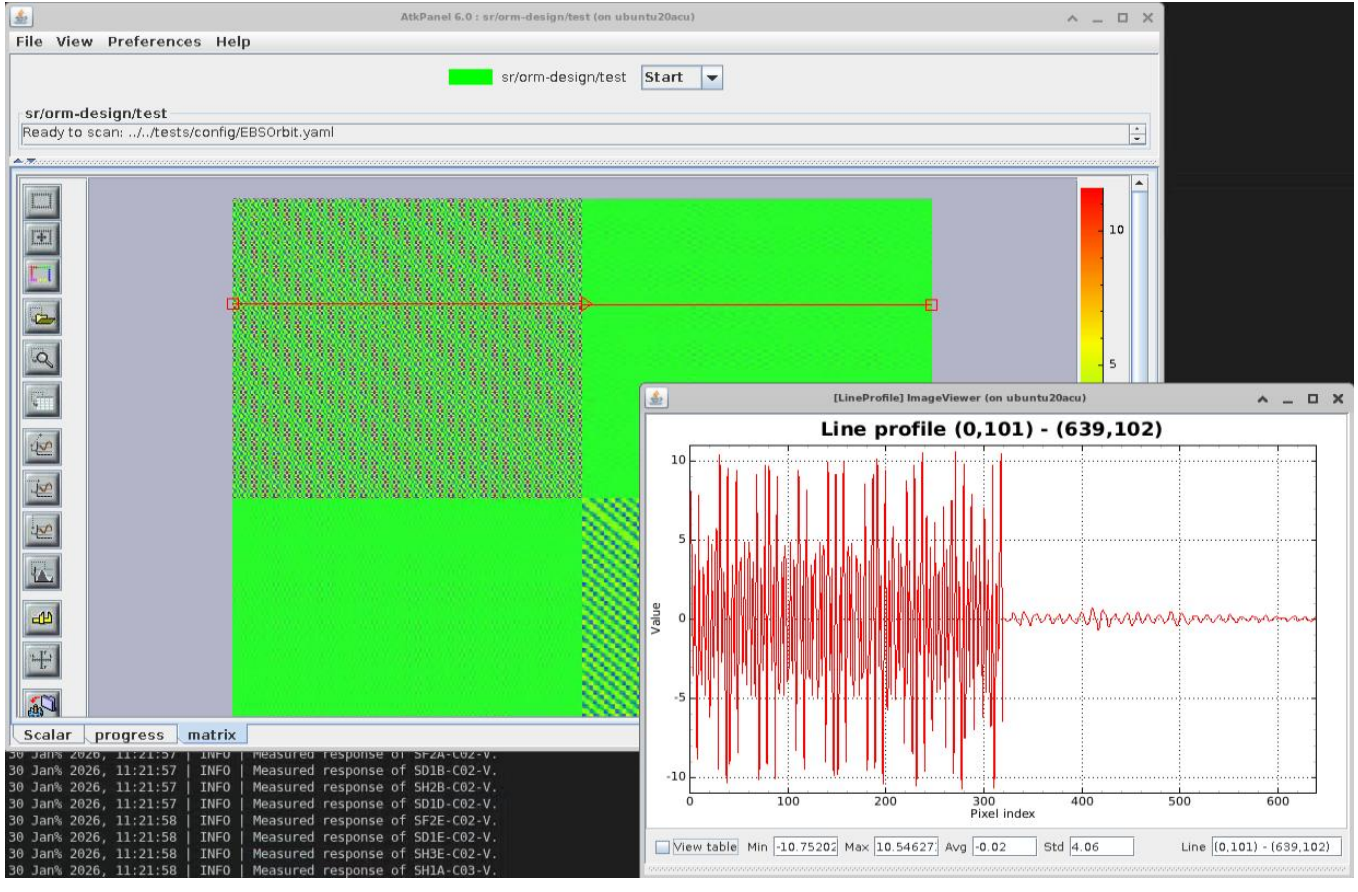
```
@attribute(
    label="Matrix",
    dtype=(("DevDouble",),),
    max_dim_x=1024,
    max_dim_y=1024,
)
def matrix(self):
    if self.orm_data:
        return np.array(self.orm_data["matrix"])
    else:
        return [[0.0]]
```

ORM MEASUREMENT TOOL IN TANGO SERVER



Generic Tango client (ATKPanel) showing the scan progress.
Orbit h,v rms is plotted sequentially for each steerer. The effect of the beta function can be seen on top and the coupling on bottom.

ORM MEASUREMENT TOOL IN TANGO SERVER



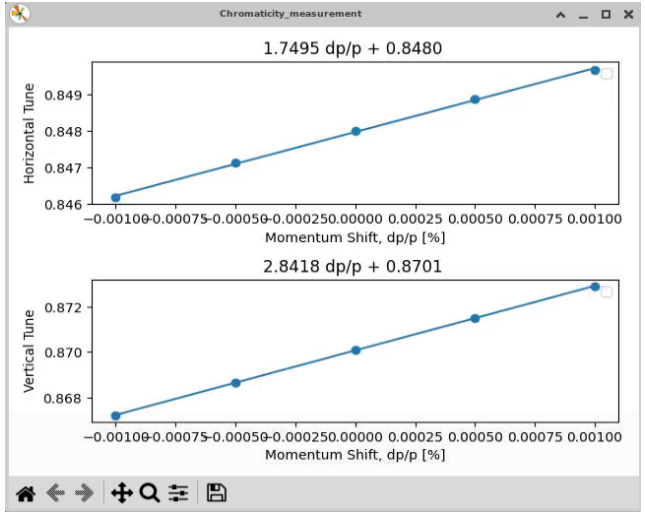
Generic Tango client (ATKPanel) showing the ORM

Chromaticity measurement configuration

Configuration

```yaml
- type: pyaml.diagnostics.chromaticity_monitor
  name: KSI
  betatron_tune: BETATRON_TUNE # PyAML object name
  RFfreq: RF # PyAML object name
  fit_order: 1
  N_tune_meas: 1
  N_step: 5
  Sleep_between_meas: 0
  Sleep_between_RFvar: 2
  E_delta: 1e-3
  Max_E_delta: 1e-3
```

```python
def chroma_callback(step: int, action: int, rf: float, tune: np.array):
    if action == ACTION_MEASURE:
        print(f"Chromaticy measurement: #{step} RF={rf} Tune={tune}")
    return True

sr = Accelerator.load("BESSY2Chroma.yaml")
sr.design.get_lattice().disable_6d()
# Retrieve MCF from the model
alphac = sr.design.get_lattice().get_mcf()
print(f"Moment compaction factor: {alphac}")
chromaticity_monitor = sr.live.get_chromaticity_monitor("KSI")
chromaticity_monitor.measure(do_plot=True, alphac=alphac, callback=chroma_callback)
ksi = chromaticity_monitor.chromaticity.get()
print(ksi)
```



Tune response measurement

Configuration

```yaml
- type: pyaml.tuning_tools.tune
  name: DEFAULT_TUNE_CORRECTION
  quad_array: QForTune
  betatron_tune: BETATRON_TUNE
  delta: 1e-4
```

```python
from pyaml.accelerator import Accelerator
from pyaml.common.constants import ACTION_RESTORE
from pyaml.magnet.magnet import Magnet

def tune_callback(step: int, action: int, m: Magnet, dtune: np.array):
    if action == ACTION_RESTORE:
        # On action restore, the measured dq/dk is passed as argument
        print(f"Tune response: #{step} {m.get_name()} {dtune}")
    return True

sr = Accelerator.load("BESSY2Tune.yaml")
sr.design.tune.response.measure(callback=tune_callback)
sr.design.tune.response.save_json("tunemat-bessy.json")
```

ORBIT TUNING TOOL

Orbit correction

- Successfully tested on EBS
- Beam lost at 250 singular values (It was expected but PyAML should have check corrector strengths before applying, setpoint check has been added recently)

Configuration

```
- type: pyaml.tuning_tools.orbit
  bpm_array_name: BPM
  hcorr_array_name: HCorr
  vcorr_array_name: VCorr
  name: DEFAULT_ORBIT_CORRECTION
  singular_values: 162
  response_matrix: file:ideal_orm_disp.json
```

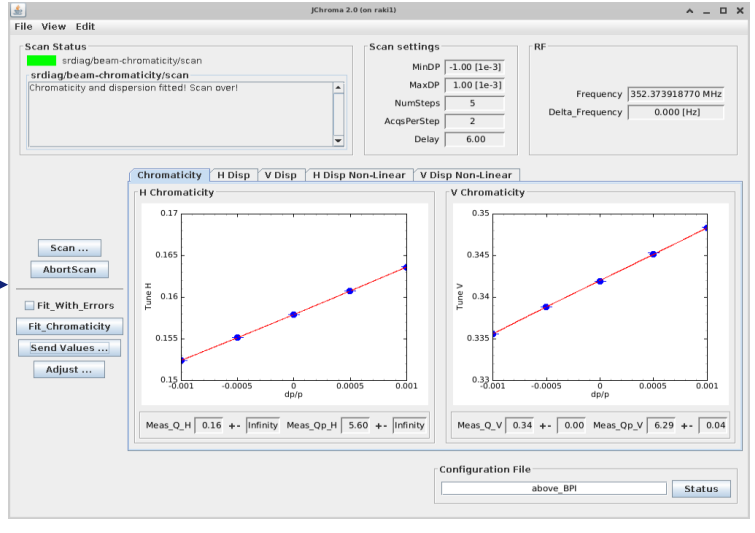
Usage

```
sr = Accelerator.load("EBSOrbit.yaml")
sr.live.orbit.correct()
```

NEXT OBJECTIVE FOR THE NEXT WORKSHOP ?

TODO List and challenge

- **Unit conversion (pint ?)**
- **Logging**
- **DOOCS bindings**
- **Dynamic generated API for Accelerator creation by code**
- **Unify measurement tools (interface, metadata, …)**
- **Chromaticity adjustment** (+ first high level application on top of a Tango server)
- **BBA**
- **LOCO**
- **…**



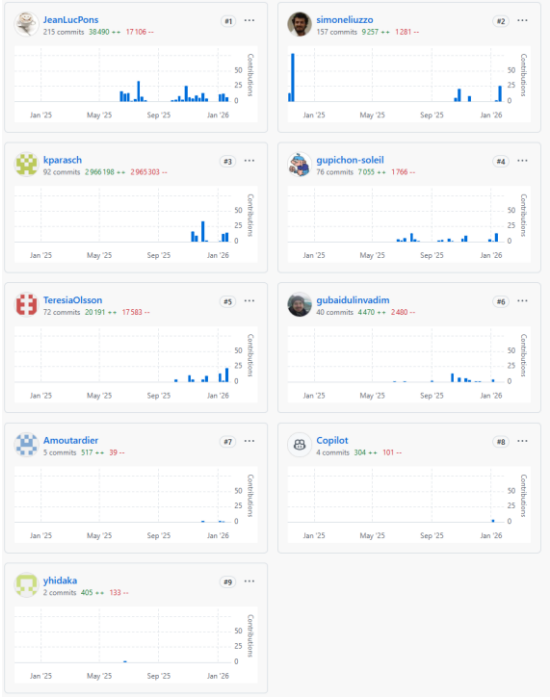
THANK YOU

Special thanks to all who contributed actively to the PyAML project

Kostas
@DESY
(pySC, PyAML)

Guillaume
@SOLEIL
(Tango, PyAML)

Simone
@ESRF
(PyAML)



Teresia
@HZB
(PyAML)

Yoshi
@BNL
(Epics, Tango, OphydAsync, PyAML)

Vadim
@SOLEIL
(PyAML)

Alexandre
@SOLEIL
(PyAML)