ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP005

A NEW PYTHON MIDDLE LAYER FRAMEWORK: Particle Accelerator MIddle LAyer (PAMILA)*

Yoshiteru Hidaka†, Daniel Allan, Max Rakitin, Brookhaven National Laboratory, Upton, NY, USA

Abstract

MATLAB Middle Layer (MML) for accelerator control has been used by many facilities worldwide over the years. With the rise of Python's popularity, particularly for leveraging its advanced artificial intelligence and machine learning libraries, an international collaboration is underway to develop a similar software framework in Python. As part of this effort, we propose a new Python middle layer package, which is built on top of the pint unit-conversion package, and capable of handling any type of device-dependent unit conversion (including multiple-input multiple-output) for magnets and other equipment. This package is also compatible with a suite of modern experimental orchestration and data management tools widely used by beamlines at many light sources (i.e., bluesky, ophyd, and tiled), and provides a more modular approach for implementing high-level applications, facilitating reuse, while exposing comprehensive, yet manageable, options to end users.

INTRODUCTION

MATLAB Middle Layer (MML) [1] has been developed over many years and is still in wide use at synchrotron light sources for its user-friendly interface, GUI tools, and simulation capabilities that can significantly reduce beam time required for application testing. It promotes collaboration by enabling cross-facility reuse of common tools. However, MML has fragmented into different versions, and modernizing it would require rewriting its core in MATLAB, a proprietary language increasingly unfamiliar to newer generations of scientists.

In contrast, Python is an open-source programming language, widely adopted and supported by a rich ecosystem of scientific libraries. Replacing MML with a Python-based alternative broadens accessibility, simplifies integration with other modern tools (especially for AI and ML), and better supports the future of accelerator development. At NSLS-II, aphla [2] was developed for this purpose, and has been used since commissioning. However, it has never achieved the original goal of becoming a Python equivalent of MML.

With this background, an international collaboration among accelerator physicists has recently begun developing the Python Accelerator Middle Layer (pyAML) [3]. During its initial exploratory phase [4], a concept code called Particle Accelerator for MIddle LAyer (PAMILA) [5] was created to test new ideas, some of which might eventually be incorporated into pyAML. This paper describes the key features of PAMILA.

CORE FEATURES

PAMILA introduces three key features. First, it supports not only typical single-input single-output (SISO) unit conversions but also more complex multi-input multi-output (MIMO) conversions. Second, it is compatible with the bluesky ecosystem (e.g., bluesky, ophyd, and tiled). Lastly, high-level applications (HLAs) are implemented as *flows* composed of *stages*, rather than conventional function-based HLAs as in MML, thereby maximizing its modular nature and hence reusability.

UNIT CONVERSIONS

PAMILA distinguishes between two types of unit conversion. The first is *universal unit conversion*, the type that scientists commonly associate with the term "unit conversion," where conversions are always intra-dimensional, that is, the unit dimensions remain unchanged. Examples include converting milliamperes to amperes, millimeters to nanometers, or gigahertz to hertz. These are handled automatically by the unit-handling Python package pint [6].

The second type is what we refer to as *representation conversion*. Unlike universal conversion, this is always device dependent and often inter-dimensional (i.e., the unit dimensions do change), though this is not a strict requirement. For instance, a power supply current in [A] may be converted to the kick angle of an orbit corrector electromagnet in [mrad], or an integrated quadrupole strength in [m^{-1}] may be mapped to the corresponding power supply current [A] driving the quadrupole magnet. The term *representation* is used because both the input and output values of a conversion describe the magnitude of the same physical quantity (e.g., magnet strength), but in different forms or *representations*.

In MML, this distinction is not clearly made. However, in cases involving MIMO conversions, it becomes evident that 'unit' conversion and 'representation' conversion are fundamentally different processes. Consider a combined-function magnet powered by two separate power supplies generating dipole and quadrupole field components. The currents I_1 and I_2 influence the dipole strength b_1 and quadrupole strength b_2. Even though both currents have the same unit (amperes), they must be treated as separate representations to convert to and from b_1 and b_2.

Figure 1 illustrates the data flows for get/put operations in a SISO case. For get, a value is first retrieved from a process variable (PV) at the low level and wrapped in a pint

* Work supported by U.S. DOE under Contract No. DE-SC0012704. The research described herein is Fundamental Research as defined in EAR (15 CFR §734.8), or Part 810 (10 CFR §810.3), as applicable, and as described in the USD (AT&L) memoranda on Fundamental Research, dated May 24, 2010, and on Contracted Fundamental Research, dated June 26, 2008.
† yhidaka@bnl.gov

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP005

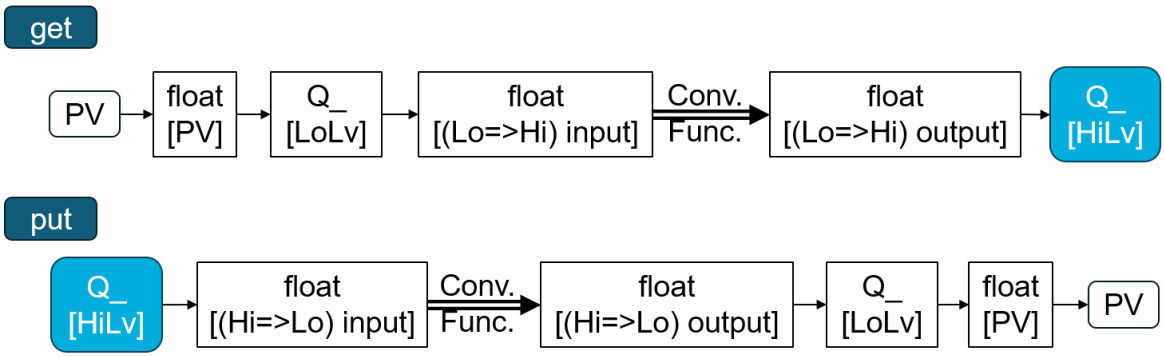

Figure 1: Data flows for SISO with conversions for `get` and `put` operations. `Q_` represents a `pint` quantity object.

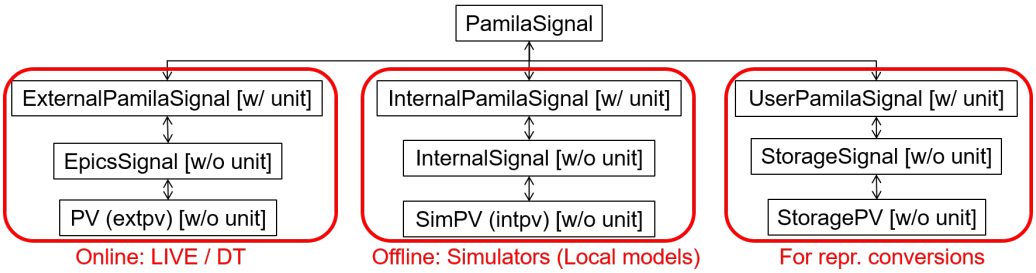

Figure 2: `PAMILA` signal class hierarchy.

quantity object. `pint` then converts it to the appropriate unit before passing it to the conversion function, which returns a float in the unit expected at the high level (i.e., the level at which the user interacts). This value is again wrapped in another quantity object and presented to the user. The `put` operation follows the reverse flow, except that it applies the inverse of the conversion function, if such an inverse exists. In MIMO cases (not shown), the only difference is the presence of multiple parallel data flows on both the input and output sides of the conversion function.

COMPATIBILITY WITH BLUESKY

`PAMILA` has been designed to be compatible with the Bluesky project [7], a suite of modern tools for experiment orchestration and data management that is widely used at beamlines across many light sources around the world.

Both online and offline operations are supported. In online (external) mode, users interface with a live accelerator or digital twin (i.e., a virtual accelerator) via a control system such as EPICS. In offline (internal) mode, users can simulate machine control scripts using an accelerator model in the local memory, without connecting to a control system.

`PAMILA` employs the signal class hierarchy shown in Fig. 2. The base class, `PamilaSignal`, for all signals holds `pint` quantity objects. The signal objects in the left two columns store values read from or written to the low-level PVs or SimPVs (simulated PVs backed by a simulation engine such as `pyAT` [8]). The signal objects in the right-most column store high-level user-facing values.

For online mode, `ExternalPamilaEpicsSignal` inherits from both `PamilaSignal` and `ophyd.EpicsSignal`. The latter class directly communicates with the low-level PVs and is only for EPICS. However, it can be replaced with an equivalent class for other control systems (e.g., Tango), if such a class is implemented in `ophyd`. For offline mode, `InternalPamilaSignal` inherits from `PamilaSignal` and `InternalSignal`. The latter custom signal class inherits from `ophyd.Signal` and connects to a SimPV.

`UserPamilaSignal` objects serve as the interface between users and the underlying system. For `get` operations, the low-level values passed from either external or internal signals are converted and stored into these signals. For `put` operations, they store user-supplied target values "as is", before they are converted and sent to the external or internal signals to be written to PVs or SimPVs.

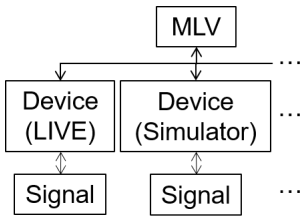

Figure 3: Hierarchy of signals, devices, and MLVs.

As shown in Fig. 3, `PamilaDevice` sits atop these signal classes. It handles application of representation conversion and includes a custom `ophyd Device` object as an attribute. This `ophyd` device object takes any of the `PamilaSignal` classes as its components.

Middle layer variables (MLVs) reside above these devices. Grouping objects called MLV lists and trees also exist for

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP005

convenient asynchronous (i.e., parallel) `get` / `put` operations on multiple devices. The primary role of MLVs is to enable transparent switching between online and offline modes. End users do not have to change their code, as long as they only work with MLVs or higher, not directly with signals or devices.

`PAMILA` also provides wrapper functions for `bluesky`: `get()` to retrieve current values from the specified MLVs, `abs/rel_put()` to put absolute or relative values to the MLVs, and `abs/rel_put_then_get()` to combine the previous functions. These wrappers automatically run plans using `bluesky`'s `RunEngine` such that end users do not have to learn how to use `bluesky` directly.

HLA STAGES AND FLOWS

HLAs in accelerator control, such as dispersion and chromaticity measurements, are often implemented as functions. This functional approach introduces key limitations, particularly when reusing one HLA within another. Name collisions for parameters (i.e., function options) can occur in nested functions. For instance, the parameter name `n_meas` may be used to indicate how many times a measurement should be repeated. When nesting HLAs, there could be multiple occurrence of `n_meas` in the top-level HLA parameters for different types of measurements (e.g., orbits and tunes). When a collision occurs, developers may resort to adding prefixes or suffixes to indicate different types of measurements. Furthermore, it is conceivable for a type of measurement to have multiple methods, each of which may have its own options, leading to the list of parameters growing uncontrollably. This clutters the docstrings, reduces readability, and complicates reuse. Another limitation is the inability to re-process or re-plot existing data, if all these processes are included in an HLA function. If plotting behavior changes or new postprocessing is needed, users must re-acquire data due to the tightly coupled design.

To address these issues, the concept of stages and flows for HLA applications has been introduced in `PAMILA`. The detailed explanation of this concept can be found at [9]. Here, some of the highlights are described.

An HLA can be divided into multiple "stages". Typical stages consist of "acquire", "postprocess", and "plot", as shown in Fig. 4. This allows each stage to be invoked independently, also enabling stage-by-stage testing and debugging. Parameters (or options) are defined for each stage as a separate class, not for the HLA as a whole. This encapsulation within each stage allows hierarchical parameter specifications, which avoids cluttering with many 'flat' (and often irrelevant) parameters.

An HLA "flow" specifies a sequence of these stages. Calling the `run` method of this flow object starts running the first stage, passes the output to the next stage, runs the next stage, and repeats until the final stage. A flow can enter and exit at any stage. For example, a "standalone" flow would enter the "acquire" stage (e.g., measure beam orbits), followed by the "postprocess" stage (e.g., calculate dispersion from the measured orbits), and finally exit from the "plot" stage after plotting the processed data. If you later decide to change the options used to postprocess the raw data and re-plot, you can create and execute another flow that starts from "postprocess" and ends with "plot". It is also easy to switch how you use this HLA: as a standalone program or as a library that provides data necessary for another HLA.

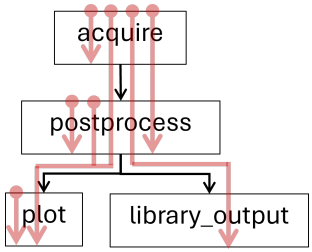

Figure 4: Stages (black boxes) and possible multiple flows (red arrows) that enters and exits at one of these stages for an HLA.

This architecture enhances modularity, reusability, and clarity. By structuring HLAs around logical stages with isolated parameters, and leveraging flows for orchestration of HLAs, developers can maintain clean interfaces while providing maximum flexibility.

CONCLUSION

`PAMILA` is a Python-based middle layer framework designed to modernize accelerator control by supporting advanced unit conversions, integration with the `bluesky` ecosystem, and a modular, reusable, stage-based architecture for HLAs. It enables seamless operation in both online and offline modes and facilitates clearer, more maintainable application design. As part of the broader `pyAML` initiative, `PAMILA` served as a testbed for novel ideas aimed at improving flexibility and interoperability in future accelerator control systems.

REFERENCES

[1] G. J. Portmann, W. J. Corbett, and A. Terebilo, "An Accelerator Control Middle Layer Using Matlab", in *Proc. PAC'05*, Knoxville, TN, USA, May 2005, pp. 4009-4011. doi:10.1109/PAC.2005.1591699

[2] L. Yang *et al.*, "The Design of NSLS-II High Level Physics Applications", in *Proc. ICALEPCS2013*, San Francisco, CA, USA, Oct. 2013, paper TUPPC130, pp. 890-892.

[3] Python Accelerator Middle Layer, `https://python-accelerator-middle-layer.github.io`

[4] S. Liuzzo *et al.*, "Exploratory tests for the design of a Python accelerator middle layer", presented at IPAC'25, Taipei, Taiwan, Jun. 2025, paper MOPB088, unpublished.

[5] NSLS2 / pamila `https://github.com/NSLS2/pamila/releases/tag/v0.1.0`

[6] hgrecco / pint, `https://github.com/hgrecco/pint/releases/tag/0.24.4`

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP005

[7] D. Allan, T. Caswell, S. Campbell, and M. Rakitin, "Bluesky's Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management", *Synchrotron Radiat. News*, vol. 32, no. 3, pp. 19-22, 2019. doi:10.1080/08940886.2019.1608121

[8] W. A. H. Rogers, N. Carmignani, L. Farvacque, and B. Nash, "pyAT: A Python Build of Accelerator Toolbox", in *Proc. IPAC'17*, Copenhagen, Denmark, May 2017, pp. 3855-3857. doi:10.18429/JACoW-IPAC2017-THPAB060

[9] NSLS2 / pamila, https://github.com/NSLS2/pamila/blob/main/doc/hla_flow_concept.md