ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD076

PYCUMBIA: BRIDGING HIGH-PERFORMANCE CONTROL LOGIC AND PYTHON SIMPLICITY FOR MODERN UIS

G. Strangolino, L. Zambon, Elettra Sincrotrone Trieste, Italy

Abstract

Pycumbia is a Python binding to the high-performance C++ cumbia framework, designed to simplify the development of control system applications without sacrificing responsiveness or scalability. It offers a user-friendly interface while maintaining the speed, concurrency, and low memory footprint of its C++ backend. By releasing Python's GIL, pycumbia ensures that GUI applications and data workflows remain smooth and responsive even under heavy load, a key requirement in control system environments.

From a user experience perspective, pycumbia significantly reduces the complexity typically associated with integrating control systems into custom applications. Developers can build advanced data visualization tools and synoptic panels with just a few lines of Python code, without dealing with polling logic, event dispatching, or thread management. Pycumbia also ships with PYI stubs for full IDE code completion.

A key architectural advantage is its flexibility in deployment: pycumbia can run inside an isolated miniconda environment, allowing developers to use up-to-date Qt and Python packages independently of the operating system, or system-wide, provided the base OS supports a recent enough software stack. This enables modern Qt-based graphical applications to be developed and run consistently across platforms, bypassing limitations of outdated system packages. Pycumbia in real-world control system applications improves both the developer experience and application performance.

INTRODUCTION

In modern accelerator and laboratory environments, control system applications need to balance high performance with rapid development. While C++ frameworks such as cumbia provide robust concurrency, low latency, and minimal memory overhead, they can be less accessible to developers who prefer dynamic languages or rapid prototyping. Pycumbia bridges this gap by exposing the cumbia [1] C++ libraries directly to Python, combining the execution speed of C++ with the simplicity and flexibility of Python. This makes it possible to create responsive, real-time control system user interfaces (UIs) with minimal code, without compromising scalability.

BACKGROUND TECHNOLOGIES

Qt Framework

Qt is a cross-platform application development framework widely used in scientific software for building rich, interactive graphical interfaces [2]. The pycumbia project targets Qt 6, which brings C++17 support, improved rendering, better high-DPI scaling, and enhanced performance for both widgets and QML-based UIs.

Python for Control Systems

Python [3] enables fast iteration, scriptable automation, and the integration of data processing, visualization, and UI logic in a single language. Pycumbia releases the GIL during C++ operations to maintain smooth GUI updates even under heavy loads.

Shiboken Binding Generator

Shiboken generates Python bindings for C++ libraries with minimal overhead. Pycumbia uses it to map cumbia's C++ API to Python efficiently.

CMake Build System

CMake manages cross-platform compilation of cumbia and pycumbia, simplifying integration with other dependencies.

PYCUMBIA ARCHITECTURE AND ADVANTAGES

Pycumbia wraps the C++ cumbia libraries into Python, preserving their multithreaded architecture. Advantages include concurrency without complexity, near-native performance, UI responsiveness, reduced boilerplate, and IDE integration via `.pyi` stubs.

AVAILABLE MODULES

PyCumbia (Base Library)

The base module implements cumbia's multithreaded activity management in Python. Developers define activities (init, execute, exit), assign them to threads, and exchange data via CuData key–value bundles. This enables safe communication between background tasks and the main UI thread without manual locking.

PyCumbiaTango

Integrates cumbia with the Tango control system [4], providing asynchronous read/write access to device attributes and commands. It supports both event-driven and polled updates. Connections, database access, and event subscriptions are managed transparently.

PyCumbiaQtControls

Provides Qt widgets for control and display.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD076

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

PyCumbiaQtControls

Provides engine-agnostic Qt widgets for displaying and controlling data. Examples include labels, gauges, thermometers, combo boxes, and spectrum plots. Widgets implement a listener interface to receive updates from background activities, ensuring decoupling between UI and data source.

PyCumbiaTangoQt

Combines PyCumbiaTango with PyCumbiaQtControls, enabling Tango-aware widgets. Factory classes create Tango-specific readers and writers that can be linked directly to UI elements, reducing setup time.

PyCumbiaHttp

Interfaces with the puma HTTP service [5], enabling control system access over HTTP/WebSocket. This is particularly useful for mobile or remote applications that cannot connect directly to Tango or EPICS servers.

PyCumbiaQtControlsNG

The "next generation" controls module offers improved performance, modernized widget designs, and new features such as enhanced plotting and graphics items.

PyCumbiaPlugins

Extends pycumbia with optional features such as:

- Formula Plugin: Combine multiple data sources using JavaScript expressions.
- Qt Designer Plugin: Configure sources/targets visually in Qt Designer.
- Extra Widgets Plugin: Add specialized widgets, such as real-time plots for array-returning device commands.
- Multi Reader Plugin: read from several sources concurrently or sequentially and receive updates at once
- Image Plugin: interface to images

DEPLOYMENT AND ENVIRONMENT MANAGEMENT

Pycumbia can be installed system-wide or within an isolated miniconda environment. The latter ensures access to the latest Python and Qt versions even on systems with outdated OS packages, while avoiding conflicts with system libraries. The pycumbia-conda-setup script automates installation, including dependencies and .pyi stub generation.

USAGE EXAMPLES

PyCumbia applications can range from extremely simple widgets bound to control system attributes, to complete graphical tools with complex data visualizations. This section presents two examples illustrating the incremental approach to building applications.

Minimal GUI with Bound Label

The first example creates a Qt window with a standard `QLabel` showing the bound attribute name, and a `QuLabel` displaying its live value.

```
from PyCumbia.Cumbia import CumbiaPool
from PyCumbia.QtControls import
    QCumbiaApplication, QuLabel
from PyCumbia.Apps import CuModuleLoader
import sys
from PySide6 import QtWidgets

if len(sys.argv) > 1:
    q = QCumbiaApplication(sys.argv)
    mlo = CuModuleLoader(q)

    # Main widget
    w = QtWidgets.QWidget()
    layout = QtWidgets.QHBoxLayout(w)

    # QLabel (static text) and QuLabel (live
        value)
    ql = QtWidgets.QLabel(w)
    label = QuLabel(w)
    layout.addWidget(ql)
    layout.addWidget(label)

    # Bind attribute source
    label.setSource("$1/double_scalar")
    ql.setText(label.source())

    w.resize(400, 120)
    w.show()
    q.exec()
else:
    print(f'usage:␣{sys.argv[0]}␣a/tg/dev,␣e.g
        :␣test/device/1')
```

Listing 1: Minimal PyCumbia application with a bound label.

Scalar Trend Plot

The second example connects to the same attribute but visualizes its values in real-time using `QuScalarPlot` (see Fig. 1).

```
import sys
from PyCumbia.QtControls import
    QCumbiaApplication, QwtPlot
from PyCumbia.Apps import CuModuleLoader
from PyCumbia.QtControlsNG import QuScalarPlot
    , QuPlotDataConnector, QuXTimeScale
from PySide6 import QtWidgets

if len(sys.argv) >= 2:
    q = QCumbiaApplication(sys.argv)
    mlo = CuModuleLoader(q)

    w = QtWidgets.QWidget()
    lo = QtWidgets.QGridLayout(w)

    # Label
    l = QtWidgets.QLabel("Trend", w)
    lo.addWidget(l, 0, 0, 1, 10)

    # Plot
    p1 = QuScalarPlot(w)
    p1.setAxisAutoScale(QwtPlot.yLeft, True)
```

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD076

```
    p1.setAxisAutoScale(QwtPlot.xBottom, True)
    lo.addWidget(p1, 1, 0, 5, 10)

    # Data connector
    c1 = QuPlotDataConnector(q.cumbiaPool(), q
        .fpool(), p1)
    QuXTimeScale(p1)
    c1.addSource("$1/double_scalar")

    w.resize(1000, 1200)
    w.show()
    q.exec()
else:
    print(f'usage␣{sys.argv[0]}␣tg_test1')
```

Listing 2: PyCumbia application displaying a scalar trend.

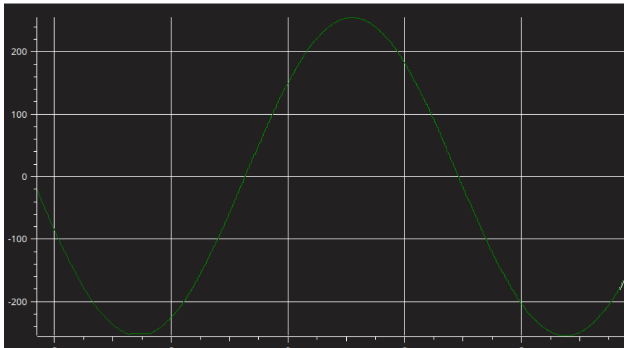

Figure 1: Live scalar trend plot generated by the PyCumbia plotting example in Listing 2.

This structure makes the learning curve clear:
- Label binding → minimal, quick start.
- Plotting → richer UI, still compact code.

COMPARISON WITH TAURUS

Taurus is a Python-based framework widely used in scientific and industrial environments to create both command-line and graphical control system applications [6]. It supports a variety of control systems, including Tango and EPICS, and offers both CLI and GUI creation capabilities.

A standout feature of Taurus is its user-friendly "wizard" interface—developers and non-programmers alike can generate complete GUIs in minutes using drag-and-drop elements and execute-time customization.

Under the hood, Taurus leverages a Model-View-Controller (MVC) architecture: data sources are abstracted as "models" (typically identified by URIs like tango:device/attribute), and a rich set of pre-built widgets—such as forms, trend plots, and synoptic panels—can be easily attached to those models programmatically or from scripts.

In comparison, Pycumbia takes a different engineering approach. It provides a direct and lightweight binding between the high-performance, multithreaded C++ cumbia framework and Python, preserving efficiency, low latency, and minimal memory overhead. While Taurus wraps control logic into Python via an MVC abstraction, PyCumbia exposes the full architectural performance benefits of cumbia—including automatic GIL release, activity-thread dispatching, and efficient CuData exchange—within a familiar Pythonic API. PyCumbia thus combines the best of both worlds: Pythonic simplicity and real-time responsiveness underpinned by native C++ performance. As a result, PyCumbia is especially suited for environments where responsiveness, scalability, and fine-grained control over threading and performance are crucial—without sacrificing rapid development.

CPU Usage Plot

A CPU usage plot follows and shows how a Taurus application compares with the pycumbia equivalent (see Fig. 2). Both arrange in a grid 196 labels displaying scalar values read at a 10Hz frequency. We monitored the CPU usage of both applications using taurus trend with a model like `eval:@psutil.*/Process(${pid}).cpu_percent(1.0)`. Just a few applications were running in the system at the time the test was taken (no web browsers, no qtcreator).

In this first test, the application is "pure" *PyCumbia*, meaning that there is no data conversion between Python and C++.

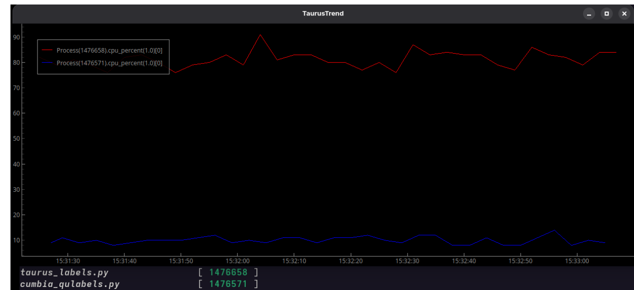

Figure 2: CPU usage comparison between PyCumbia and Taurus implementations.

Startup Time

Both Taurus and pycumbia code has then been modified in order to measure the time elapsed between the start of the application and the completion of the showEvent.

Averaging 10 startups, the result is that *PyCumbia* version takes from 190 to 215 ms, *Taurus* from 2044 to 2310 (a factor greater than 10.)

Tests Involving Data Conversion

In this second test, we used the same Taurus application, with exactly the same arguments, but we compared it to a flavor of its pycumbia counterpart which updates QuLabel objects throught a Multireader (PyCumbia.Plugins.Multireader). In this case, again we employed 196 labels refreshed at 10 Hz, exactly as in the previous example.

Using the Multireader implies converting data from C++ to Python and viceversa.

© Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THPD076

This implementation shows that when Python comes into play, much more CPU time is needed In Fig. 3), you can see four instances of the same panel running on four different layers:

1. Taurus configured like in 1a (red)
2. Pycumbia with no C++/Python data conversion (the same used in the experiment 1a) (blue)
3. Cumbia pure C++ (green)
4. Pycumbia + Multireader (violet)

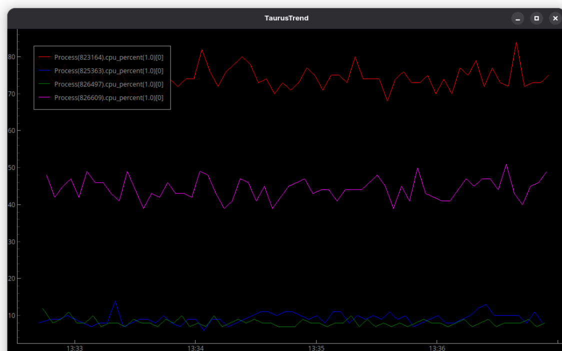

Figure 3: Performance with data conversion using PyCumbia Multireader.

A more in depth analysis of the comparison between PyCumbia and Taurus can be found in [7].

Notes on Software Versions

Taurus Taurus was installed with *pip install taurus* within a virtual environment, version 5.2.4 which should include the Taurus Performance Optimization number 1. Using PyQt5 (v5.15.11 with Qt 5.15.14 and Python 3.13.3) and PyTango 10.0.2.

PyCumbia

- pycumbia version 1.0.1 with cumbia 2.4.0,
- cumbia-qtcontrols 2.5.0,
- cumbia-tango 2.4.0,
- cumbia-qtcontrols-ng 2.5.0
- cumbia-http 2.5.0,
- cumbia-tango-qt 2.5.0.

Qt and Qwt

- Qt,
- PySide6,
- Shiboken6 version 6.9.0,
- Qwt 6.3.0 on Qt 6.9.0

Conclusions About Performance Comparison: PyCumbia vs. Taurus When evaluating PyCumbia and Taurus in terms of performance, the key observations from benchmarking are:

Startup time: PyCumbia applications generally start faster because they only load the necessary modules, whereas Taurus often initializes a larger Tango-specific framework.

Runtime efficiency: Thanks to its C++ backend and Shiboken bindings, PyCumbia handles high-frequency data updates with lower CPU usage.

Memory footprint: In tests with multiple widgets updating at high rates, PyCumbia showed lower steady-state memory usage compared to Taurus.

Scalability (just a guess at this stage of development): PyCumbia's modular engine system seems to allow distributing data processing across threads more flexibly, improving responsiveness under heavy load.

These differences are most noticeable in applications that must display many rapidly updating attributes or handle data from mixed sources.

CONCLUSIONS

Pycumbia provides a streamlined pathway to develop modern, responsive, and high-performance control system UIs in Python without sacrificing the efficiency of a C++ backend. By building directly on top of the cumbia libraries, pycumbia inherits the constant performance improvements and optimizations made to the C++ core. As the cumbia framework evolves—adopting new concurrency models, optimizing data structures, and improving cross-platform support—these advancements automatically benefit pycumbia applications. This guarantees that developers can focus on features and usability while relying on a robust, actively maintained, and future-proof foundation. The combination of Python's ease of use, Qt 6's modern graphical capabilities, and cumbia's high-performance architecture makes pycumbia an ideal choice for the next generation of control system user interfaces in accelerator and laboratory environments.

REFERENCES

[1] cumbia libraries, modules and plugins, https://gitlab.elettra.eu/cs/lib/cumbia

[2] Qt, https://www.qt.io/.

[3] Python, https://www.python.org

[4] Tango, http://www.tango-controls.org

[5] puma service, https://gitlab.elettra.eu/puma/server

[6] taurus, https://taurus-scada.org/docs.html

[7] pycumbia taurus comparison, https://gitlab.elettra.eu/cs/lib/cumbia/taurus-evaluation

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2025). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.