pybela: a Python library to interface scientific and physical computing

Teresa Pelinski
Centre for Digital Music
Queen Mary University of London
London, UK
t.pelinskiramos@qmul.ac.uk

Giulio Moro
Augmented Instruments Ltd
London, UK
giulio@bela.io

Andrew McPherson
Dyson School of Design Engineering
Imperial College London
London, UK
andrew.mcpherson@imperial.ac.uk

Abstract

Workflows to obtain, examine and prototype with sensor data often involve a back and forth between environments, platforms and programming languages. Usually, sensors are connected to physical computing platforms, and solutions to transmit data to the computer often rely on low-bandwidth communicating channels. It is not obvious how to interface physical computing platforms with data science environments, which also operate under distinct constraints and programming styles. We introduce pybela, a Python library that facilitates real-time, high-bandwidth, bidirectional data streaming between the Bela embedded computing platform and Python, bridging the gap between physical computing environments and data-driven workflows. In this paper, we outline its design, implementation and applications, including deep learning examples.

Keywords

prototyping, Bela, physical computing, workflows, Python, machine learning, deep learning, embedded AI

1 Introduction

Sensing interaction and the physical environment is central to fields like interactive media, wearable technology, or home automation. Sensors transmit information from the physical world into the digital domain, however they may not relay this information as transparently as expected: signals can be noisy or inconsistent. Often, they need significant observation, shaping and moulding before they are suitable for use in an interactive context. This process can require considerable trial and error, and sometimes, complex mapping strategies or numerical systems such as neural networks.

Workflows to obtain, examine and prototype with sensor data often entail back and forth between environments, platforms and programming languages. Moreover, particularly in maker and arts-technology communities, workflows may involve low-bandwidth solutions such as streaming data from a microcontroller to a computer over a USB-serial link. Alternatively, approaches based on logging data on an embedded computing platform face problems of limited storage and inconvenience transferring large files to the computer for analysis. If running the scientific computing code in an embedded computer such as Raspberry Pi or Bela, users might face slow processing times given the limited CPU performance of low-resource platforms.

To address these issues related to workflow complexity and low bandwidth, we present pybela. pybela facilitates both data collection and monitoring, as well as bidirectional real-time data streaming between Bela [18] and a computer[1]. The pybela API allows requesting data from –and sending data to– Bela interactively in Python, with great flexibility over *what*, *when* and *how* data is sent. pybela can be used for data collection (e.g., collecting training data for a deep learning model, debugging and calibrating sensors, capturing data during a user study) but also for streaming data back and forth between platform and computer (e.g. sending control data, running a deep learning model in a desktop computer on sensor data, and sending the output back to Bela for further processing).

By integrating physical computing (Bela embedded platform) with data science workflows (Python and Jupyter notebooks running on a computer), pybela attempts to bridge ("plug" [21]) two communities with distinct priorities and code practices: real-time vs. non real-time, low-resourced vs. computationally intensive, compiled vs. interpreted. Our goal is not to create an audio streaming environment with the lowest possible latency, a goal which could be better served through existent professional audio interfaces. Instead, the idea is to bridge two disparate communities by empowering them to map their knowledge into a new domain [21] while respecting their preexistent workflows. Alternatively, pybela can also be used in an educational setting to introduce students to physical computing with a more beginner-friendly language, Python, than C++.

A natural extension of bridging physical computing with data science environments is to train and run deep learning models on sensor signals. Along with the library[2] and its documentation[3], we provide a set of examples and tutorials[4] to run models either on Bela or on the computer (while streaming data back-and-forth with Bela). We also contribute a cross-compilation environment to speed-up compilation of Bela programs, as well as pre-built binaries for Bela of the most common C++ inference frameworks (Torch, ONNXRuntime, TensorFlow Lite). This aligns with

NIME '25, June 24–27, 2025, Canberra, Australia

[1] In this paper we use "computer" to refer to a desktop or laptop personal computer with a GUI, as opposed to a low-powered embedded computer platform.
[2] https://github.com/BelaPlatform/pybela
[3] https://belaplatform.github.io/pybela/
[4] https://github.com/pelinski/deep-learning-for-bela

the growing interest towards embedded AI, particularly in musical performance [11, 23, 31], forms of music AI beyond audio (e.g. using gesture data) [9, 27], as well as prototyping with machine learning models [3, 28].

2 Background

In this section we provide some context to the issues pybela attempts to address, as well as a summary of previous work and existent tools.

2.1 Fragmented prototyping

2.1.1 Different prototyping models. The ease of prototyping is central to both physical computing platforms and data science environments. Physical computing platforms such as Arduino or Bela facilitate tinkering with electronics by abstracting the complexities of compilation (through an IDE), offering GPIO ports, APIs that facilitate interfacing with peripherals, and supportive online communities [21]. However, given their limited computational resources and often stringent real-time requirements, physical computing environments are typically based on C++, which is usually the choice in performance-critical applications as it allows manual control over memory allocation. This memory control can be challenging for beginners and medium-skilled programmers, which along with compilation times, can obstruct the prototyping flow. Alternatively, in the microcontroller domain, MicroPython[5] and CircuitPython[6] are Python flavours designed to run directly on the hardware, however they offer a less optimised performance and limited interaction during runtime. On the other hand, data science environments such as Python Jupyter notebooks, allow running code snippets interactively while preserving the state between runs. Along with the fact that Python does not require compilation, and that it is a higher-level programming language with a large catalogue of libraries, Jupyter notebooks facilitate quick and dynamic prototyping with data. However, in terms of efficiency, as an interpreted language with automatic memory allocation, Python generally has a slower execution time than C++.

2.1.2 Viscous workflows. When building digital artefacts in creative environments, technologists will often emphasise speed and ease of development to enable smooth prototyping [6]. In particular, Blackwell and Green use the term "viscosity" to refer to the number of actions it takes to accomplish a goal in programming [4]. In this sense, workflows to process sensor signals in data science environments can be viscous. Generally, sensors are connected to embedded platforms that often run in C++, which complicates accessing data interactively during runtime. A common workflow to obtain and process data is therefore to write the data into a file on the embedded platform, and then, from the computer, to copy this file through the terminal, and parse it and handle it in a Python environment. Other options include streaming data over USB serial (e.g. on an Arduino microcontroller), an option which offers poor bandwidth and timing resolution, or running Python directly on the board on higher-spec platforms such as Bela or Raspberry Pi – but still face slow processing times given their limited CPU capacity. There are indeed many options to transmit data from the embedded platform to the data science environment, yet they often involve low-bandwidth solutions and a back and forth between platforms, programming languages and environments.

2.1.3 Toolchain gaps in Embedded AI. An obvious application of combining physical computing with data science is to train and run deep learning models on real-time sensor data. However, running neural networks in real-time physical computing environments can be complex [22], although there are a few examples in digital musical instrument design [11, 16, 31]. The limited resources of physical computing platforms complicate training and running neural network models in real-time on-device. Models are typically trained on the computer and then deployed either on-device or run in the computer while streaming data back and forth with the embedded platform. Running models on-device involves compiling deep learning inference engines (e.g. Torch, ONNX, TFLite) for the platform of choice, which will generally require cross-compilation on the computer. It also may involve knowledge of compilation software such as CMake or Makefile – skills which are quite advanced for a beginner or mid-level physical computing or data science practitioner. Moreover, the real-time reliability of these inference engines has been questioned (due to e.g. issues with dynamic memory allocation) [8, 25], although significant improvements are being made to ensure it [1]. Alternatively, the model can be run on the computer, however there is no evident high-bandwidth option to stream the data back and forth between the computer and the embedded platform.

2.2 Previous work and existent tools

2.2.1 Improvement over previous pipeline. We addressed some of these issues related to integrating physical computing and data-intensive approaches (in particular, neural networks) in a previous publication [22]. In this earlier work, we presented a pipeline to record datasets, train deep learning models and deploy them in Bela. Although useful, that pipeline still involved significant back and forth between platforms in the data capture step: the data files were recorded in Bela and then manually transferred to the computer. There was no direct communication between Python and the Bela code, and hence no control from Python as to what data was recorded, and when and how it was sent.

pybela extends this work by enabling direct communication between Python and the code running in Bela, so that from Python, the user can precisely control which data and when is being sent, and how, e.g., at which sampling frequency. Moreover, pybela adds a streaming path from Python to Bela, which enables, for instance, outsourcing computation of models which are too expensive to run in Bela, or connecting to other software through e.g. another Python API. We have also extended the aforementioned pipeline with a flexible choice for inference engine, as we now provide pre-built binaries for Torch and ONNXRuntime besides TensorFlow Lite.

[5] https://micropython.org/ [accessed 02/02/2025]
[6] https://circuitpython.org/ [accessed 27/01/2025]

2.2.2 Existent tools. If the purpose is to exclusively capture sensor data with high accuracy, scientific data acquisition interfaces (e.g. NI mioDAQ[7]) offer high resolution data collection as well as accompanying analysis software. However, their price is high in comparison to physical computing platforms, and they are not programmable embedded computers that can be used in interactive contexts. Alternatively, MicroPython and CircuitPython run on various microcontrollers and offer some limited interaction with the code running in the microcontroller through a REPL serial console[8], as well as support for a variety of sensors including Adafruit's[9]. There also exist devices targeting specifically data collection in physical computing environments, such as a data logging shield for Arduino[10] – however, its application is limited to offline data logging to an SD card rather than dynamic control of when and how the data is collected. Notably, an inspiration for the pybela's early development were the course materials by Justin Bois for a Caltech course[11] on building custom scientific measuring instruments with Arduino, Python and serial communication.

These microcontroller-based environments can offer a real-time exchange of data with the computer; however, they often rely on the serial protocol. In comparison to network-based approaches (e.g. WebSockets running over TCP), serial typically offers limited bandwidth and limited assurances on data integrity as the connection is usually one-directional.

2.2.3 Choice of platform. As opposed to the approaches discussed above, pybela relies on the WebSocket protocol to communicate with Bela, which offers bidirectional communication and several features to ensure message consistency. In principle, pybela's implementation could be ported to any other platform with a network stack and enough CPU and memory to run the Watcher server (the Watcher will be introduced in Section 4.1), such as e.g. Raspberry Pi. However, our choice of platform is determined by Bela's strong timestamping implementation, in which sensor or audio signals are timestamped at the level of data capture rather than when the data is received by the computer. This allows, for instance, ensuring that data is sent and processed in an orderly manner and in real-time, as well as accurately measuring roundtrip latencies between Bela and Python (see Section 5). In this sense, pybela leverages Bela's strong timestamping, as well as low-latency and high-resolution sensing capabilities, since sensors are captured at audio rate [17, 18] – features which are critical in a musical performance context. Porting to other platforms would require, besides the technical specs outlined above, at least a similar strong timestamping functionality.

[7]https://www.ni.com/en/shop/data-acquisition/miodaq-devices.html [accessed 02/02/2025]
[8]https://learn.adafruit.com/welcome-to-circuitpython/the-repl [accessed 27/01/2025]
[9]https://learn.adafruit.com/welcome-to-circuitpython/beginner-boards [accessed 27/01/2025]
[10]Arduino Data Logger Shield https://learn.adafruit.com/adafruit-data-logger-shield/overview [accessed 27/01/2025]
[11]Caltech "Design and Construction of Biodevices" https://be189.github.io/ [accessed 27/01/2025].

3 Features and modes of operation

We have outlined two main issues with the existent workflows to integrate physical computing and data science environments: the back and forth between platforms, programming languages and environments (the viscosity of the workflow) and a low-bandwidth communication channel which prevents a real-time exchange of high-resolution data. pybela attempts to address these issues through the following design goals:

(1) Control how data is sent (at which sampling frequency, in which format) and when (immediately or at a point in the future, and for how long)
(2) Simple, beginner-friendly API that does not require Python asynchronous programming knowledge
(3) Controls to start and stop the exchange of data can be sent interactively from a Jupyter notebook (or programmed in a Python script)
(4) Ensuring consistency in data and latency so that the library is usable in a real-time scenario

Goals (2), (3) and (4) are related to the library implementation, which will be discussed in next section. Goal (1) involves implementing the library flexibly and make it suitable for several use cases, for instance continuous streaming of data or sampling data at given intervals to monitor sensor behaviour under changing conditions. To address these various use cases we have implemented four modes of operation: (a) streaming, to continuously send data between Bela and Python, (b) monitor, to also send data from Bela to Python but at given intervals, (c) logging, to reliably log data to files and (d) controlling, to force variables in the Bela code to take given values (sent by Python). Tables 1 and 2 summarise the purpose, use cases and functionality of each of these modalities. Detailed tutorials for each of the pybela modes are available in the pybela repository[12].

For brevity, below we will use "Bela variables" to refer to variables that have been defined in the Bela code and that can be accessed and interacted with using pybela – the technicalities which make this possible will be discussed in next section. Variables can represent anything in the Bela code: direct readings of sensor values, a processed version of the signal, or whatever the user wants to define (e.g. a boolean variable indicating if certain condition is happening or not).

3.1 Stream

The streaming mode allows continuous streaming of data between Bela and Python, in both directions. It allows scheduling the streaming to happen at future timeframes or requesting a specific number of data points. A more interesting feature is the ability to set callbacks to be run every time a buffer of data is received in Python. The callback can also be set for a "block" or packet of buffers, so that rather than running the callback per buffer received per Bela variable, the callback is called once the buffers of all variables (for a given timestamp) are received. The streaming mode ensures that the buffers are processed in order and with real-time guarantees. Moreover, the streaming

[12]https://github.com/pelinski/pybela/tree/main/tutorials/notebooks

	Streamer	Monitor	Logger	Controller
Purpose	continuous streaming of data between Bela and Python	sampling values of Bela variables	reliably recording data locally in Bela	setting values of Bela variables
Use case example	performance with real-time sensor data	sensor calibration	dataset recording	sending control values

Table 1: Purpose and use cases of pybela's modes of operation

	Streamer	Monitor	Logger	Controller
Sending data from Bela to Python	✓ continuous stream of data	✓ "sampled" stream of data	✓ as binary files	✗
Sending data from Python to Bela	✓ continous stream of data	✗	✗	✓ forces a Bela variable to take a given value
Saving data into a file	✓ saved in a .txt file on arrival to Python	✓ saved in a .txt file on arrival to Python	✓ saved locally in binary files in Bela, then transferred to computer	✗
Calling a callback function each time a buffer of data is received	✓	✓	✗	✗

Table 2: Functionality of pybela's modes of operation

mode allows sending buffers of data back to Bela. Additionally, the plotting method allows displaying the received data in Python in real-time.

The streaming mode can be used to interface with sensors from Python in real-time. The sensor data can be then processed in Python and sent back to Bela (e.g. to control an actuator) or to another software (e.g. MaxMSP). In particular, the callback method can be used to run a deep learning model (that might be too heavy to run directly in Bela) and send the output back to Bela – an example project is shown in Section 6.2. Datasets can also be recorded with the streamer mode, although the logging mode is a more reliable option in this case as it does not depend on the status of the connection between Python and Bela.

3.2 Monitor

In a typical scenario, the Bela variables correspond to sensor signals sampled at audio rate in Bela. In certain situations, however, such high sampling rates (44100 samples per second) are not necessary. For instance, when trying to find the range of operation of a sensor we might want to try a certain condition (e.g. stretch a stretch sensor), get a few values and then repeat with a different condition (e.g. stretch sensor at rest). In these cases, a few values per second suffice. The monitor mode works as a "sampled" streaming mode, it streams data from Bela to Python at a requested rate[13]. The monitor mode also allows to "peek" at a variable value, that is, requesting a single value rather than a continuous stream of data.

3.3 Log

Rather than streaming values from Bela to Python, the logging mode writes the data into binary files locally on Bela. This is to ensure that the data is recorded regardless of the status of the communication between Bela and Python. This is a safer option for recording datasets, the audio of a performance, or sensitive data from a user study. Files are saved in binary to save memory[14]. To avoid long transfer times, by default the logging mode transfers the files as they are being recorded. In the event of a connection error, the transfer does not need to be restarted but can be continued from the point it had been left at. Similarly to the streaming mode, a logging session can be scheduled for a particular time in the future.

3.4 Control

The control mode allows forcing a Bela variable to take a value set from Python. If `foo` is a Bela variable which has been assigned to the Bela audio input, regardless of what the audio input is, with the control mode we can force `foo` to take a given value. The variable will still hold that value

[13]In fact, the Monitor class is implemented as a child class of the Streamer class.

[14]Storage requirements grow quickly when recording data at audio rate. The Bela has just under 4GB of available memory, although it can be extended with an SD card.

regardless of what happens in the Bela code (i.e. even if it gets assigned a new value in the code), up until the control mode is disabled.

The control mode does not support sending buffers of data from Python to Bela variables, it only supports setting one value at a time for each Bela variable. It should be noted that while the Streamer does support sending buffers of data to the Bela code, it does not support setting or forcing those values into Bela variables – "forcing" values onto Bela variables is a functionality reserved to the control mode[15]. The control mode can be used in a performance context to send control values (e.g. parameters of a filter), or in a debugging scenario in which we might want to force certain conditions into the Bela code.

4 Architecture and implementation

How does pybela request and obtain data from Bela on demand? The communication between a Bela C++ program and pybela happens through the WebSocket protocol, which establishes a bidirectional transmission channel over a TCP connection. pybela acts as a client that sends requests to and receives data from the Bela Watcher server. The Watcher is a Bela C++ library[16] that can be added to any Bela C++ program. Variables declared in the Bela code can be put on "watch", and then be streamed, monitored, logged or controlled, as requested by pybela. This avoids having to rewrite and recompile the Bela project each time the user wants to send data in a different mode.

Besides requesting the data, pybela needs to manage the arrival and processing of data sent from the Watcher, which involves asynchronous Python programming. In this section, we discuss some relevant aspects of the implementation of the Bela Watcher and pybela.

4.1 The Bela Watcher

To make the variables in the Bela code accessible to pybela, variables need to be defined as an instantiation of the Watcher template class. The Watcher class wraps a "watched variable" of a supported type (floating point and integer numerical types up to 64 bit) and connects it with the pybela API via a WebSocket. The Watcher `set()` method sets the value of the watched variable and notifies a WatcherManager instance of the update. The `get()` method returns either the last value that was assigned to the watched variable in the C++ code or a "control" value set via the pybela API. To minimise the code changes needed to move from using a native numeric type to using a Watcher object, the Watcher class also provides two overloads (redefinitions):

(a) the assignment (=) operator calls `set()`
(b) the casting operator, called when accessing the value of the variable, calls `get()`

Figure 1 shows an example of Bela C++ code before and after adding the Watcher. An object of the WatcherManager class manages the timestamping and WebSocket connection of one or more Watcher objects. Watcher objects are assigned a WatcherManager instance on creation and notify it when `set()` or the = operator overload are called. If pybela is currently monitoring the variable, the WatcherManager stores each assigned value in a buffer and sends it to Python when the buffer is full. Timestamping is provided by calling WatcherManager's `tick()` method. The Watcher adds a timestamp at the beginning of each buffer indicating the "tick" value (in the example, the audio frames elapsed) of the first item of the buffer. For cases in which the Watcher is not "ticked" at a constant rate, and in consequence, the data in buffers is not equally spaced in time, the Watcher offers a "dense" timestamping method in which each value in the buffer is accompanied by its timestamp.

4.2 Abstracting asynchronicity

At the beginning of Section 3 we outlined the design goals of pybela. In particular, goal (2) referred to a simple, beginner-friendly API that did not require asynchronous programming knowledge, and goal (3), that the library was usable in an interactive Python environment (Jupyter notebooks). The pybela implementation relies on asynchronous code: pybela requests a continuous stream of data, but besides waiting for the next incoming buffer, it might need to process the previous one, store it somewhere – in general, do other tasks while waiting for more data to arrive. This is a typical I/O-bound operations scenario: the performance of the program is limited by operations which are "waiting" for events. In this context, the "data listener" functions (the functions that wait for the data buffers to arrive) cannot be programmed synchronously, as they would block the execution of all other tasks[17].

4.2.1 Asynchronicity in Python. In Python, I/O-bound tasks and asynchronicity are managed with the `asyncio`[18] library. `asyncio` creates an "event loop" that acts as a scheduler. The event loop has a queue of tasks that are generally I/Obound, so they spend most of their time waiting for an event to happen. While waiting, they yield control back to the event loop, which in the meantime can run other coroutines. Coroutines are functions that are declared with the `async` keyword and may include the `await` keyword in their definition. The `await` is put in front of a function which is likely to spend most of its execution waiting for an event to happen, time in which the control is yield back to the event loop.

The event loop and `async`/`await` syntax for managing asynchronicity is not a standard practice in physical computing, at least in what relates to real-time sensor and

[15]The buffers sent from Python to Bela through the Streamer need to be decoded and manually assigned to Bela variables in the Bela code, this is to avoid ambiguity with regards to when (at which timeframe) each value of the buffer is set for each variable.
[16]https://github.com/BelaPlatform/Watcher

[17]Technically, in a multithread scenario the "data listener" functions could be programmed synchronously, i.e. without using `await` to explicitly tell the scheduler where the control can be yield to another thread. However, a multithreading approach scales poorly (it needs to create a new thread per task) and involves significant overhead when context-switching (changing threads) and creating threads, as each thread requires a stack memory allocation. Multithreading can also introduce race conditions when multiple threads attempt to modify the same data.
[18]https://docs.Python.org/3/library/asyncio.html [accessed 30/01/2025]. `asyncio` is part of the Python standard library, i.e. the modules and packages that come pre-installed in Python.

(a) Example Bela C++ code

```
float gFrequency;
int gButton;
// ... some code ...
void render(BelaContext* context, void*)
{
    for(unsigned int n = 0; n < context->audioFrames; ++n)
    {
    gButton = digitalRead(context, n, 0);
    gFrequency = analogRead(context, n, 0) * 1000 + 200;
    float out = oscillator.process(gFrequency);
    // ... some code ...
    }
}
```

(b) Example Bela C++ code after adding the Watcher

```
Watcher<float> gFrequency("frequency"); // uses the default WatcherManager
Watcher<int> gButton("button"); // uses the default WatcherManager
// ... some code ...
void render(BelaContext* context, void*)
{
    for(unsigned int n = 0; n < context->audioFrames; ++n)
    {
    // tick the  WatcherManager
    Bela_getDefaultWatcherManager()->tick(context->audioFramesElapsed + n, 0 == n);

    // operator overload calls the Watcher set() method
    gButton = digitalRead(context, n, 0);
    gFrequency = analogRead(context, n, 0) * 1000 + 200;

    // "out" can either use the value assigned above or the one set via the pybela "
        Controller" class
    float out = oscillator.process(gFrequency);

    // ... some code ...
    }
}
```

Figure 1: Example code of using the Watcher in a Bela C++ program.

(a) pybela current implementation

```
streamer.start_streaming()
streamer.wait(30)
streamer.stop_streaming()
```

(b) pybela usage if coroutines were not wrapped in synchronous functions

```
async def stream_for_30_seconds():
    loop = asyncio.get_event_loop()
    loop.create_task(streamer.start_streaming_coroutine())
    await asyncio.sleep(30)
    loop.create_task(streamer.stop_streaming_coroutine())
asyncio.run(stream_for_30_seconds)
```

Figure 2: (a) The pybela current API and (b) the API if the coroutines were exposed.

audio processing[19]. In such cases, asynchronicity is typically managed by either multithreading (for tasks which do not require scaling, e.g. running a given CPU-intensive task in a block of audio), loop-based polling (i.e. periodically checking if an OSC message has been received), or with user-defined callbacks managed by the platform's API[20]. The event loop and the `async/await` syntax style for managing asynchronicity is also not habitual in deep learning practices, where code is generally synchronous.

4.2.2 Wrapping coroutines. Figure 2 shows a comparison of the current API implementation vs. how the same functionality would look if coroutines were exposed. The `async/await` syntax is cumbersome, even for programmers familiar with asynchronicity. In pybela, we wrapped all coroutines under synchronous functions to avoid exposing the `asyncio` syntax to the user.

`asyncio` does not allow nested event loops, which prevents running coroutines inside of other coroutines; instead, they need to be scheduled as tasks (see lines 4 and 6 of the example in Figure 2). This complicates wrapping coroutines in synchronous functions, as their execution context needs to be considered. The solution involved duplicating certain methods as both asynchronous (awaiting a result) and synchronous (immediately returning even if the result is not ready). This avoided less robust alternatives such as the `nest_asyncio`[21] patch library, which bypasses `asyncio`'s prohibition of nested event loops, but introduces the overhead of creating and closing event loops every time a coroutine is executed[22]), and possible inefficient scheduling. This patch, however, is still necessary to run the library in Jupyter notebooks, as this environment runs its own event loop, conflicting with `asyncio`'s single event loop per thread constraint.

5 Benchmarking

In this section we provide measurements of the roundtrip latency for data travelling from Bela to Python and back to Bela. Before discussing these results, it is important to clarify that our goal is not to create an audio streaming environment with the lowest possible latency, a goal which would be better served through existing professional audio interfaces. Instead, the value of pybela lies in its ability to integrate physical computing with data science workflows, facilitating prototyping with real-time sensor data. Moreover, the data presented in Table 3 corresponds to the roundtrip latency of buffers filled at audio rate (44.1 kHz), a resolution that exceeds the requirements of many sensing scenarios.

The roundtrip latency was measured as the time it takes for a sample of data to travel from Bela to Python and back[23]. Samples are not sent one-by-one but rather collected in buffers (as discussed in Section 4.1) of 1024 samples[24], filled at audio rate. Once full, each buffer is sent from Bela to Python, which immediately sends it back. In this scenario, each streamed variable represents a channel: at every time frame, for n streamed variables, Bela appends one sample to each of the n buffers.

Measurements were taken across different number of streamed variables and under varying CPU load conditions. To force CPU load[25], an oscillator bank was added to the Bela audio thread. Data was transmitted over USB[26] and the average latencies were recorded over a 60-second period. The worst-case (WC) latency reflects the highest observed value over that period, and the jitter is calculated as the difference between the latency at percentiles 97.5% and 2.5%. Tests were conducted on a 2019 MacBook Pro running macOS Sequoia 15.3.2, with a 2.6 GHz 6-core Intel Core i7 CPU and 16 GB of DDR4-2667 RAM.

num. streamed variables	num. oscillators (avg. CPU load)	mean latency (ms)	jitter (ms)	WC (ms)
1	0 (37%)	7.6	10.1	33.0
	20 (76%)	12.3	23.7	71.8
	40 (98%)	43.0	107.7	195.9
5	0 (68%)	25.0	36.2	122.6
	20 (90%)	47.7	95.1	221.0
	40 (97%)	193.2	375.4	775.3
10	0 (67%)	47.8	65.9	196.6
	20 (90%)	93.6	145.7	352.7
	40 (97%)	380.9	556.1	998.1
20	0 (66%)	95.9	120.1	292.1
	20 (89%)	186.7	254.6	546.0
	40 (98%)	754.7	1194.4	2181.6

Table 3: Roundtrip (Bela→ Python→Bela) latency measurements for varying numbers of streamed variables and CPU load conditions. Data was transmitted in buffers of 1024 samples, filled at 44.1 kHz, and streamed over USB for a duration of 60 seconds.

As expected, latency increases consistently with the number of streamed variables. This is due to the fact that for each additional variable, an extra buffer must be sent to Python and returned to Bela, scaling the amount of

[19]It is, of course, a different story in the Internet of Things (IoT) domain.

[20]E.g. the Bela API has a `setBinaryDataCallback()` method to which the user can pass a function that is called every time a buffer of data is received through WebSockets.

[21]`nest_asyncio` is a patch library for `asyncio` developed by Ewald de Wit https://github.com/erdewit/nest_asyncio [accessed 30/01/2025].

[22]If the coroutine is executed with `ayncio.run()`, which would be the most straightforward way of wrapping a coroutine under a synchronous function.

[23]The benchmarking code is available in https://github.com/BelaPlatform/pybela/tree/main/benchmark

[24]The streamer buffer size is determined by the data type of the Bela variable and the timestamping mode (sparse or dense, as discussed in Section 4.1). In this case, we streamed integers with sparse timestamping (only one timestamp per buffer). The buffer size of the data sent from Python to Bela is not fixed, and in this case was set to 1024 to match the size of the buffer sent by Bela.

[25]The CPU load is calculated as the sum of all threads, running in Xenomai or Linux mode, scaled by the Xenomai CPU usage.

[26]All latency measurements reported here were performed using USB data transmission. For readers interested in how performance may vary when using WiFi instead, we refer to the evaluation presented in [32].

data transferred by the number of variables. The maximum latency (i.e. worst-case latency) is significantly higher than the mean, and the jitter values are also relatively large. This is not uncommon for protocols that are not specifically optimised for low streaming latency. It was observed that the latency peaked every 5s, which seems to suggest a timing behaviour introduced by the desktop OS or Python's runtime environment. Beyond the jump from 1 to 5 streamed variables, CPU usage remained relatively stable across different number of oscillators, though –as expected– latency continues to increase with more streamed variables.

6 In context: developing tools for deep learning in Bela

pybela has been developed within a larger research project that focuses on facilitating deep learning workflows involving physical computing platforms, in the context of real-time audio and interaction. Earlier, in Section 2.1.3, we outlined some of the workflow difficulties specific to embedded AI. In this context, we have developed a series of resources to support prototyping with deep learning models and embedded computing platforms. All tools, tutorials, and examples mentioned in this section are included in the provided repository[27].

In a deep learning pipeline, pybela is of use at two stages: for capturing a dataset, and, if the model is running in the computer, for streaming data in real-time between Bela and the computer. If the model is run directly on Bela (on-device) instead, we provide an updated toolchain (with respect to our previous publication [22]) to cross-compile the Bela inference code. So far, we have not considered on-device training given the limited computational resources of the platform. However, recent promising work in embedded interactive machine learning [13] suggests it could be an interesting addition. Below, we discuss both inference cases (on-device and outsourced to the computer).

6.1 Inference on-device

Embedded computers have a significantly limited CPU capacity when compared to desktop computers. In deep learning, models are typically run in GPUs, although there is an increasing interest for lighter models deployed on laptop or embedded CPUs [20, 24, 33]. There also exist techniques to compress the size of an existent deep learning model (pruning) to make it suitable for an embedded context [7, 10, 26]. In creative contexts, however, a perhaps more interesting approach is to move away from one-size-fits-all large models and work with smaller custom datasets and models instead [30]. This becomes a hard constraint rather than an artistic choice when working with an embedded platform with limited resources – and arguably a source for creative inspiration[28] [5]. Additionally, many AI approaches in interactive media are inspired by information retrieval techniques and are often directed towards analysing audio, text or visual data rather than engaging with the embodied aspects of creative practice (some counterexamples are [9, 12, 15]).

pybela supports this light and custom AI approach by facilitating the collection of sensor signals dataset from a Jupyter notebook (or vanilla Python) environment, where practitioners can prototype with deep learning models trained on those signals. To avoid toolchain complications when deploying the models back to the embedded platform, in this case Bela, we have provided a dockerised[29] cross-compilation environment which avoids tedious local installs – only a couple of Docker commands are necessary to pull the image and create a container. We also provide a tutorial with an example project. In the tutorial, a Jupyter notebook is run in the Docker container and all the steps (dataset collection, model training and export, and cross-compilation of Bela inference code) happen in the Jupyter notebook. The tutorial uses Torch as the C++ on-device inference engine, but we also provide the ONNXRuntime and TensorFlow Lite libraries compiled for Bela. The example project[30] in the tutorial uses audio from a microphone to control a drum synthesiser using onset detection and feature extraction, which map to synthesis parameter updates through a multi-layer perceptron model.

6.2 Inference on desktop computer

For situations in which running a deep learning model directly in the embedded platform might be unfeasible (due to the computational and/or real-time constraints), or in which there might be a substantial amount of data processing involved (e.g. expensive filters or adaptive algorithms), the model inference and additional data processing can be outsourced to the computer. Streaming the sensor data from Bela to pybela and back to Bela adds an overhead which might however be compensated by running the model in the computer's higher-spec CPU or GPU, if available. The output of the model can be sent back to Bela for e.g. further processing, controlling an actuator, or integrating into an Eurorack modular system[31], or alternatively, to another software running on the computer (e.g. MaxMSP, SuperCollider, Processing).

We provide an example project which explores the sonification of the latent space of a number of deep learning models, all variations of the same architecture. The architecture, a custom Transformer [29] autoencoder, compresses 8 piezo signals into 4 signals. The piezo sensors are attached to the body of a feedback double-bass[32], and the signals are sent in blocks from Bela to Python. Each block (8 signals, 1024 samples) is passed to the model, which then outputs a block of 4 signals. The output block can be either sent back to Bela, which runs integrated in a Eurorack modular setup or to SuperCollider through OSC. In this project, we also implemented an algorithm that uses the output of the model to determine which model should run the next block's inference. The algorithm takes into account the rate of variation and amplitude of the 4 compressed piezo signals as well as the history of the system to determine if

[27] https://github.com/pelinski/deep-learning-for-bela

[28] See [14] for a discussion on designing with constraints in the context of digital musical instruments.

[29] Docker (https://www.docker.com/ [accessed 05/02/2025]) is a tool for packaging applications. It creates isolated environments with all the necessary libraries to run a given application.

[30] The example project was originally developed by Jordie Shier.

[31] Using Pepper, a Eurorack module that allows mounting Bela into a Eurorack setup (https://learn.bela.io/products/modular/pepper/ [accessed 05/02/2025]).

[32] This project is an ongoing collaboration with Adam Pultz Melbye and their Feedback-Actuated Augmented Bass [19].

there should be a change in model or not. The prototyping of this algorithm involved a good amount of parameter tweaking in a rehearsal setting, which would have been significantly slowed down if it had involved recompiling the Bela C++ project each time.

7 Conclusion

This paper introduced pybela, a Python library designed to integrate physical computing and data science workflows by enabling real-time, bidirectional communication between the Bela embedded computing platform and Python. We addressed the "fragmented prototyping" that occurs when working across physical computing and data-driven workflows, caused by their distinct prototyping models and the lack of efficient and interactive routes to exchange high-bandwidth real-time data. pybela supports multiple modes of operation – streaming, monitoring, logging and controlling – providing flexibility for diverse applications, from real-time performance to dataset collection. Moreover, the pybela implementation abstracts all complexity related to asynchronous Python programming. Similarly, the modifications required to use pybela with an existing Bela project are minimal. Beyond the technical contribution, we provide tutorials and example projects, which we hope will be of valuable use for researchers, educators, and practitioners exploring data-driven approaches on physical computing environments.

Ethical Standards

Giulio Moro and Andrew McPherson are part of Augmented Instruments Ltd, the company that produces the Bela platform. Teresa Pelinski's PhD is also partly supported by the same company, and part of this work was conducted during her internship at Bela between April and September 2023.

pybela has been improved through its implementation in the authors' and colleagues' projects, as well as in workshops such as [2, 16]. This work did not involve a formal human subjects research protocol and therefore did not require approval from our institution's ethic board.

Acknowledgments

This research has been supported by Bela (Augmented Instruments Ltd), the EPSRC UKRI Centre for Doctoral Training in Artificial Intelligence and Music (EP/S022694/1), a UKRI Frontier Research (Consolidator) Grant (EP/X023478/1, "RUDIMENTS") and by the Royal Academy of Engineering under the Research Chairs and Senior Research Fellowships scheme. The authors would like to also thank Jordie Shier and Adam Pultz Melbye for their involvement in the example projects, as well as Adán L. Benito Temprano and Rodrigo Diaz for their contributions in earlier iterations of the project. The authors also appreciate the insightful feedback provided by Andrea Martelloni, Chris Kiefer, Francisco Bernardo, Jordie Shier and the anonymous NIME reviewers.

References

[1] Valentin Ackva and Fares Schulz. 2024. ANIRA: An Architecture for Neural Network Inference in Real-Time Audio Applications. In 2024 IEEE 5th International Symposium on the Internet of Sounds (IS2). IEEE, Erlangen, Germany, 1–10. https://ieeexplore.ieee.org/document/10704099/

[2] Jack Armitage, Victor Shepardson, Nicola Privato, Teresa Pelinski, Adan L Benito Temprano, Lewis Wolstanholme, Andrea Martelloni, Franco Santiago Caspe, Courtney N. Reed, Sophie Skach, Rodrigo Diaz, Sean Patrick O'Brien, and Jordie Shier. 2023. Agential Instruments Design Workshop. In AIMC 2023. The University of Sussex, Falmer, UK. https://qmro.qmul.ac.uk/xmlui/handle/123456789/91547

[3] Francisco Bernardo, Michael Zbyszyński, Mick Grierson, and Rebecca Fiebrink. 2020. Designing and Evaluating the Usability of a Machine Learning API for Rapid Prototyping Music Technology. Frontiers in Artificial Intelligence 3 (April 2020). https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2020.00013/full

[4] Alan F. Blackwell and Thomas Green. 2003. Notational Systems—The Cognitive Dimensions of Notations Framework. In HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science, John M. Carroll (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 103–133.

[5] Margaret A. Boden. 2004 (1991). The Creative Mind: Myths and Mechanisms (2 ed.). Routledge, London. https://doi.org/10.4324/9780203508527

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. 2008. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. In Proceedings of the 4th International Workshop on End-user Software Engineering (WEUSE '08). Association for Computing Machinery, New York, NY, USA, 1–5. https://dl.acm.org/doi/10.1145/1370847.1370848

[7] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2020. A Survey of Model Compression and Acceleration for Deep Neural Networks. arXiv:1710.09282 [cs] http://arxiv.org/abs/1710.09282

[8] Jatin Chowdhury. 2021. RTNeural: Fast Neural Inferencing for Real-Time Systems [Preprint]. arXiv:2106.03037 [eess] http://arxiv.org/abs/2106.03037

[9] Çağrı Erdem, Ricardo Simionato, Sayed Mojtaba Karbasi, and Alexander Refsum Jensenius. 2022. Embodied Perspectives on Musical AI (EmAI) - RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion. https://www.uio.no/ritmo/english/news-and-events/events/workshops/2022/embodied-ai/index.html

[10] Philippe Esling, Ninon Devis, Adrien Bitton, Antoine Caillon, Axel Chemla–Romeu-Santos, and Constance Douwes. 2020. Diet Deep Generative Audio Models with Structured Lottery. In Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx-20). Vienna, Austria. arXiv:2007.16170 [cs, eess, stat] http://arxiv.org/abs/2007.16170

[11] Nicholas Evans, Behzad Haki, and Sergi Jordà. 2024. GrooveTransformer: A Generative Drum Sequencer Eurorack Module. In Proceedings of the International Conference on New Interfaces for Musical Expression. Zenodo, 261–265. https://zenodo.org/records/13904848

[12] Rebecca Fiebrink, Dan Trueman, and Perry R. Cook. 2009. A Meta-Instrument For Interactive, On-The-Fly Machine Learning. In Proceedings of the International Conference on New Interfaces for Musical Expression. Zenodo, Pittsburgh, PA, United States, 280–285. https://zenodo.org/record/1177513

[13] Chris Kiefer and Andrea Martelloni. 2024. Musically Embedded Machine Learning Workshop. In CHIME Annual Conference 2024. The Open University, Milton Keynes, UK. https://static1.squarespace.com/static/6227c31a43daf21135453605/t/6723af17d5a2ce1e889a8d6a/1730391832196/13+Andrea+Martelloni+and+Chris+Kiefer.pdf

[14] Thor Magnusson. 2010. Designing Constraints: Composing and Performing with Digital Musical Systems. Computer Music Journal 34, 4 (Dec. 2010), 62–73. https://direct.mit.edu/comj/article/34/4/62-73/94484

[15] Charles Patrick Martin, Kyrre Glette, Tønnes Frostad Nygaard, and Jim Torresen. 2020. Understanding Musical Predictions With an Embodied Interface for Musical Machine Learning. Frontiers in Artificial Intelligence 3 (2020). https://www.frontiersin.org/article/10.3389/frai.2020.00006

[16] Charles Patrick Martin and Teresa Pelinski. 2024. Building NIMEs with Embedded AI. In Proceedings of the International Conference on New Interfaces for Musical Expression. Utrecht, Netherlands. https://smcclab.github.io/nime-embedded-ai/

[17] Andrew McPherson, Robert Jack, and Giulio Moro. 2016. Action-Sound Latency: Are Our Tools Fast Enough?. In Proceedings of the International Conference on New Interfaces

for Musical Expression. Brisbane, Australia, 20–25. https://www.zenodo.org/record/3964611

[18] Andrew McPherson and Victor Zappi. 2015. An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black. In 138th Audio Engineering Society Convention. Warsaw, Poland. http://www.aes.org/e-lib/browse.cfm?elib=17755

[19] Adam Pultz Melbye and Halldór Úlfarsson. 2020. Sculpting the Behaviour of the Feedback-Actuated Augmented Bass: Design Strategies for Subtle Manipulations of String Feedback Using Simple Adaptive Algorithms. In Proceedings of the International Conference on New Interfaces for Musical Expression. 221–226. https://zenodo.org/record/4813328

[20] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2022. A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations. IEEE Transactions on Neural Networks and Learning Systems 33, 10 (Oct. 2022), 5095–5115. https://ieeexplore.ieee.org/document/9410437

[21] Fabio Morreale, Giulio Moro, Alan Chamberlain, Steve Benford, and Andrew McPherson. 2017. Building a Maker Community Around an Open Hardware Platform. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. Denver, CO, USA. https://dl.acm.org/doi/10.1145/3025453.3026056

[22] Teresa Pelinski, Rodrigo Díaz, Adán L. Benito Temprano, and Andrew McPherson. 2023. Pipeline for Recording Datasets and Running Neural Networks on the Bela Embedded Hardware Platform. In Proceedings of the International Conference on New Interfaces for Musical Expression. Mexico City, Mexico. https://www.nime.org/proceedings/2023/nime2023_22.pdf

[23] Teresa Pelinski, Victor Shepardson, Steve Symons, Franco Santiago Caspe, Adán L. Benito Temprano, Jack Armitage, Chris Kiefer, Rebecca Fiebrink, Thor Magnusson, and Andrew McPherson. 2022. Embedded AI for NIME: Challenges and Opportunities. In International Conference on New Interfaces for Musical Expression. Auckland, New Zealand. https://nime.pubpub.org/pub/rwr2c3zs/release/1

[24] Wolfgang Roth, Günther Schindler, Bernhard Klein, Robert Peharz, Sebastian Tschiatschek, Holger Fröning, Franz Pernkopf, and Zoubin Ghahramani. 2024. Resource-Efficient Neural Networks for Embedded Systems. Journal of Machine Learning Research 25, 50 (2024), 1–51. http://jmlr.org/papers/v25/18-566.html

[25] Domenico Stefani, Simone Peroni, and Luca Turchet. 2022. A Comparison of Deep Learning Inference Engines for Embedded Real-Time Audio Classification. In Proceedings of the 25th International Conference on Digital Audio Effects (DAFx20in22). Vienna, Austria, 256–263. https://www.dafx.de/paper-archive/2022/papers/DAFx20in22_paper_16.pdf

[26] David Südholt, Alec Wright, Cumhur Erkut, and Vesa Välimäki. 2023. Pruning Deep Neural Network Models of Guitar Distortion Effects. IEEE/ACM Transactions on Audio, Speech, and Language Processing 31 (2023), 256–264. https://ieeexplore.ieee.org/document/9954902/

[27] Atau Tanaka, Federico Visi, Balandino Di Donato, Martin Klang, and Michael Zbyszyński. 2023. An End-to-End Musical Instrument System That Translates Electromyogram Biosignals to Synthesized Sound. Computer Music Journal 47, 1 (March 2023), 64–84. https://doi.org/10.1162/comj_a_00672

[28] Pierre Alexandre Tremblay, Gerard Roma, and Owen Green. 2021. Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit. Computer Music Journal 45, 2 (June 2021), 9–23. https://doi.org/10.1162/comj_a_00600

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In Advances in Neural Information Processing Systems, Vol. 2017-Decem. 5999–6009. https://arxiv.org/abs/1706.03762

[30] Gabriel Vigliensoni and Rebecca Fiebrink. 2025. Data- and Interaction-Driven Approaches for Sustained Musical Practices with Machine Learning. Journal of New Music Research (2025), 1–14. https://www.tandfonline.com/doi/full/10.1080/09298215.2024.2442361

[31] Federico Visi. 2024. The Sophtar: A Networkable Feedback String Instrument with Embedded Machine Learning. In Proceedings of the International Conference on New Interfaces for Musical Expression. Zenodo, 142–148. https://zenodo.org/records/13904810

[32] Johnty Wang, Eduardo Meneses, and Marcelo M Wanderley. 2020. The Scalability of WiFi for Mobile Embedded Sensor Interfaces. In Proceedings of the International Conference on New Interfaces for Musical Expression. Royal Birmingham Conservatoire, Birmingham City University, Birmingham, UK.

[33] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. arXiv:1907.10701 [cs, stat] http://arxiv.org/abs/1907.10701