Timeflux

Release 0.6.1

Pierre Clisson

Jun 01, 2020

TABLE OF CONTENTS

Timeflux is a free and open-source framework for the acquisition and real-time processing of biosignals. Use it to bootstrap your research, build brain-computer interfaces, closed-loop biofeedback applications, interactive installations, and more. Written in Python, it works on Linux, MacOS and Windows. Made for researchers and hackers alike.

It comes with integrated communication protocols (Lab Streaming Layer, ZeroMQ, OSC), HDF5 file handling (saving and replay) and generic data manipulation tools.

Currently available plugins include signal processing nodes, machine learning tools and a JavaScript API for precise stimulus presentation and bidirectional streaming. A signal monitoring interface is included and is accessible directly from your browser.

Drivers for open and proprietary hardware (EEG, ECG, PPG, EDA, respiration, eye tracking, etc.) have already been developed, with more coming. And if your equipment is compatible with LSL, you are already good to go!

What now?

If you are new to Timeflux, start by reading the *Core concepts* section and then follow the *Getting started* guide and the *Hello, World!* tutorial.

Attention: Right now, the documentation is a bit coarse, and some parts of the code need polishing. We are working on it! Meanwhile, do not hesitate to *get in touch*, we will be glad to help.

Part I

General

CHAPTER

ONE

ABOUT

1.1 Why yet another BCI framework?

We needed something that:

- fits well within the Python datascience and machine learning ecosystem
- has a permissive license, allowing commercial applications
- works both offline and online
- enables quick prototyping
- is developer-friendly
- is not only about EEG data or even about the brain, and that can works with any time series

We were not able to find a framework that satisfied all of our requirements, so we built it.

1.2 The team

Timeflux is primarily developed by Pierre Clisson and Raphaëlle Bertrand-Lalo, with the support of awesome contributors.

1.3 Citing

The introduction paper has been be published in the Proceedings of the 2019 Graz BCI Conference.

CHAPTER

TWO

CONCEPTS

As a framework, Timeflux enables the creation of **applications** that interact with **time series**. According to Wikipedia, "*a time series is a series of data points indexed in time order*".

Applications are defined by an ensemble of processing steps, called **nodes**, which are linked together using a simple YAML syntax. In order to be valid, an application must satisfy the requirements of a **directed acyclic graph** (DAG), that is, a set of nodes connected by edges, where information flows in a given direction, without any loop.

Multiple DAGs are authorized within the same application, optionally communicating with each other using one of the available network protocols. DAGs run simultaneously at their own adjustable **rate**. Within each DAG, nodes are executed sequentially according to the topological sorting of the graph.

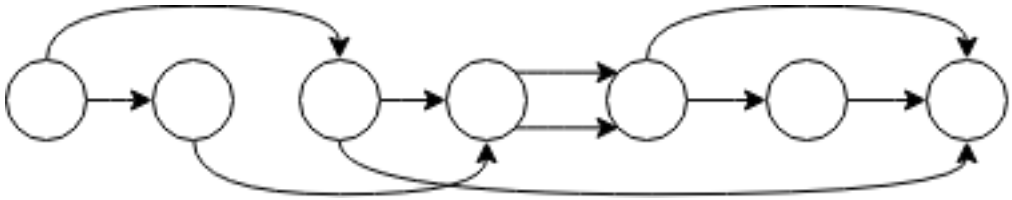


Fig. 1: An example directed acyclic graph (DAG), arranged in topological order. Circles represent **nodes**, arrows are **edges**. The connection points between nodes and edges are called **ports**. Information flows from left to right, at a frequency defined by the graph rate.

Nodes may expect one or more **inputs** and may provide any number of **outputs**. These I/O have two main properties. The **data** property is either a *datetime*-indexed Pandas DataFrame, or an Xarray structure with at least two dimensions: time and space. The **meta** property is a dictionary containing arbitrary keys, which can be used for example to indicate a stream nominal rate or to describe the context associated with an event.

There are different types of **ports** that may impact how an application is written:

- **Default ports** – Each node has one implicit input port and one implicit output port. If a node is expecting only one input, it will be on the default input port. If a node is providing only one output, it will be on the default output port.
- **Named ports** – If a node is expecting more than one input or is providing more than one output, it will explicitely define ports. These ports are identified by a name.
- **Dynamic ports** – They work just like named ports, except they are defined on the fly, according to the context.
- **Numbered ports** – In some cases, the number of inputs or outputs cannot be known in advance. For example, an epoching node running at a low rate may need to output an arbitrary number of *DataFrame*s during each execution.

Note: This is all there is to know to understand how Timeflux works. Confused? Don't worry. In the next sections, we will discuss a few examples in details and all this will soon make sense. You will be able to design your own

application from scratch in no time!

CHAPTER

THREE

GETTING HELP

If you have any question, please join our Slack workspace. We will be glad to help.

You can also contact us by email: contact@timeflux.io.

We give workshops on a regular basis. These are great opportunities to interact in person. Subscribe to our mailing-list to be informed of future events.

Depending on your requirements, we may be able to offer commercial support, personalized training and custom development. Please get in touch!

Tip: If you don't know how to use a particular node, have a look at the API reference section. You will often find useful information there.

Tip: Most nodes have working examples you can get inspiration from. You just need to know where to look :) Example YAML files can usually be found in the *examples* directory of any of the GIT repositories (Timeflux or any plugin, for instance, the DSP package).

Part II

Usage

CHAPTER

FOUR

GETTING STARTED

4.1 Installation

Before we can do anything, we need a Python 3.7+ distribution. We recommend Miniconda. If you don't already have it, go ahead and install it.

If you intend to install from source, you will also need Git.

Now that the prerequisites are satisfied, the next order of business is to install Timeflux and its dependencies. To keep things nice and clean, we will do this in a new virtual environment. Depending on your system, open a shell or command prompt:

```
conda create --name timeflux
conda activate timeflux
pip install timeflux
pip install timeflux_example
```

If everything went well, Timeflux is now installed. Hooray!

4.2 Basic usage

Applications are self-described in YAML files. Running an app is easy:

```
timeflux app.yaml
```

4.3 Let's try!

First, download a very simple app that we will use as an example:

```
curl -O https://raw.githubusercontent.com/timeflux/timeflux/master/examples/test.yaml
```

If the *timeflux* environment is not already activated, do it:

```
conda activate timeflux
```

You can now run the test app:

```
timeflux -d test.yaml
```

You should see and a bunch of random numbers every second. Hit *Ctrl+C* to stop.

Did you notice the `-d` flag in the command line? It's a shorthand for `--debug`, and this what allowed us to see the messages in the console.

4.4 Command line options

There are only a few options, and you can list them with:

```
timeflux --help
```

This should print:

```
usage: timeflux [-h] [-v] [-d] [-E ENV_FILE] [-e ENV] app

positional arguments:
  app                   path to the YAML or JSON application file

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -d, --debug           enable debug messages
  -E ENV_FILE, --env-file ENV_FILE
                        path to an environment file
  -e ENV, --env ENV     environment variables
```

Besides the `-d` flag we already discussed, two options are worth mentioning: `-E` or `--env-file` and `-e` or `--env`.

4.5 Environment

Storing an app configuration in the environment is a good practice. There are a few ways of doing this:

If a file named *.env* is found in the current directory or in any of its parent directories, it will be loaded. A *.env* file looks like this:

```
# A comment that will be ignored
FOO=bar
MEANING_OF_LIFE=42
```

As we saw earlier, you can also specify a custom path to an environment file with the `--env-file` option.

Another way of setting environment variables is with the `-e` option:

```
timeflux -e FOO="bar" -e MEANING_OF_LIFE=42 app.yaml
```

Finally, you can temporarily set environment variables for the duration of the session, directly from the console.

Windows:

```
set FOO "bar"
```

Linux, MacOS:

```
export FOO="bar"
```

The following environment variables are understood by Timeflux:

- `TIMEFLUX_LOG_LEVEL_CONSOLE` – This is the level of details printed in the console. Possible values are *DEBUG*, *INFO*, *WARNING*, *ERROR* and *CRITICAL*. The default value is *INFO*. Running the `timeflux` command with the `-d` flag is the same as setting this variable to *DEBUG*.
- `TIMEFLUX_LOG_LEVEL_FILE` – This is the logging level when the output of the application is written to a file. This variable accepts the same values as previously. The default value is `DEBUG`.
- `TIMEFLUX_LOG_FILE` – If set to a valid path, Timeflux will write the application output to a log file.
- `TIMEFLUX_HOOK_PRE` – Name of a Python module that will be run before executing the app.
- `TIMEFLUX_HOOK_POST` – Name of a Python module that will be run after executing the app.

Others variables may be used by specific nodes and plugins. Refer to the relevant documentation for details.

By combining environment variables and *templating*, you can add logic to your *YAML* files and build configurable applications.

4.6 Plugins

Timeflux is modular. The `timeflux` Python package contains the core features and the most essential nodes. Plugins are standard Python packages that provide one or several nodes. Officially supported plugins can be found on Timeflux GitHub page. Some plugins (especially those dealing with hardware) have special requirements. Please refer to each plugin repository for installation instructions.

Notable plugins include:

- User interface
- Digital Signal Processing

CHAPTER

FIVE

HELLO, WORLD!

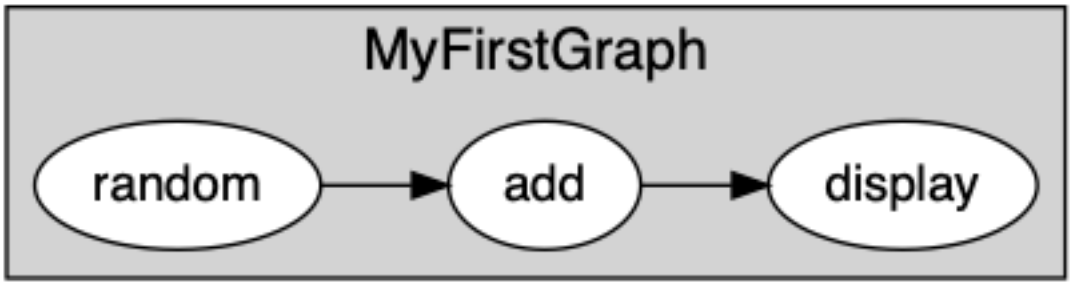
Let's get our feet wet! You will now write your first application, and progressively extend its capabilities.

Timeflux apps are written in YAML. If you don't know anything about YAML, we recommend to first spend three minutes watching this introduction video.

For this app, we will use a node that comes from the timeflux_example plugin. The installation is straightforward:

```
conda activate timeflux
pip install timeflux_example
```

For your first iteration, here is what you will build:



This is a very simple app with one graph and three nodes. The first node generates random time series, sends its output to another node that adds 1 to each value of the matrix, and that itself sends its own output to a debugging node.

Open a code editor, and copy-paste the following:

```
graphs:
  - id: MyFirstGraph
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random
      params:
        columns: 5
        rows_min: 1
        rows_max: 10
        value_min: 0
        value_max: 5
        seed: 1
    - id: add
      module: timeflux_example.nodes.arithmetic
```

(continues on next page)

(continued from previous page)

```
    class: Add
    params:
      value: 1
  - id: display
    module: timeflux.nodes.debug
    class: Display
  edges:
  - source: random
    target: add
  - source: add
    target: display
  rate: 1
```

Save the file as *helloworld.yaml*, and run the app (don't forget to activate the *timeflux* environment if it is not already the case!):

```
timeflux -d helloworld.yaml
```

Interrupt with *Ctrl+C*.

Let's take a step back and examine the code. The first thing we notice is the `graphs` list. Here, we only have one graph. Graphs have four properties:

- `id` – This is not mandatory, but very useful when you have more complex applications. Each time something is logged in the console, the graph `id` will be printed, so you know exactly where it comes from.
- `nodes` – This is the list of individual computing units in our graph.
- `edges` – This is where you connect the nodes together.
- `rate` – The frequency of the rate. A value of *25* means that each node in the graph will be executed 25 times per second. The default value is *1*, so here, we could have omitted it.

Nodes have also four properties:

- `id` – A unique identifier for the node. Unlike graph `id`s, node `id`s are mandatory.
- `module` – The name of the Python module where the node is defined.
- `class` – The name of the Python class that defines the node.
- `params` – A dictionary of parameters passed during the initialization of the node.

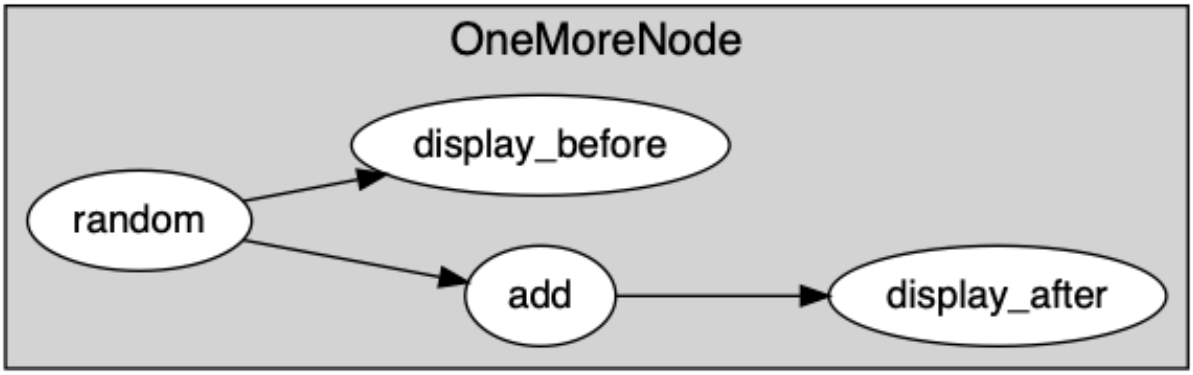
The list of parameters and their meaning can be found in the API documentation. In our case:

- *`timeflux.nodes.random.Random()`*
- `timeflux_example.nodes.arithmetic.Add()`
- *`timeflux.nodes.debug.Display()`*

Edges connect nodes together. They have two properties, `source` and `target`. Each of them takes a node `id` as value. Here we connect the default output to the default input, so we don't need to specify the port name. We will see how to connect I/O to named and dynamic ports in a moment.

Right now, we have a very simple app that prints the modified matrix in the screen. What if we want to *also* display the original matrix *before* we add 1?

This can be represented schematically as:



And in YAML as:

```yaml
graphs:
  - id: OneMoreNode
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random
    - id: add
      module: timeflux_example.nodes.arithmetic
      class: Add
      params:
        value: 1
    - id: display_before
      module: timeflux.nodes.debug
      class: Display
    - id: display_after
      module: timeflux.nodes.debug
      class: Display
    edges:
    - source: random
      target: add
    - source: random
      target: display_before
    - source: add
      target: display_after
```

We just added one `Display` node and one edge. For brevity, we also removed the `params` property of the `Random` node and used the default values.

All this console printing is boring. We want glitter and unicorns! Or at least, we want to display our signal in a slightly more graphical way. Enters the timeflux_ui plugin. This plugin enables the development of web interfaces, and exposes a JavaScript API to interact with Timeflux instances through WebSockets. Bidirectional streaming and events are supported, as well as stimulus scheduling with sub-millisecond precision. A monitoring interface is included in the package.

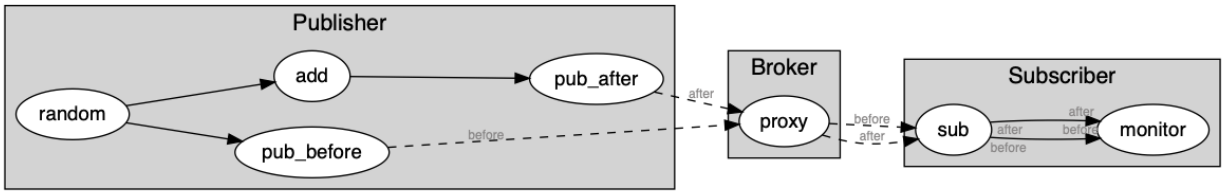
Without further ado, let's install it:

```
pip install timeflux_ui
```

We need to refactor our code a little bit. For better performances (and also for readability), it is a good practice to divide the code into multiple graphs. Remember that nodes inside a graph run sequentially and that graphs inside an application run in parallel.

Of course, in our simple example, we could get away with only one graph, but this is an opportunity to introduce a fundamental notion: inter-graph communication. To send and receive information between nodes from different graphs, we can use any of the asynchronous network nodes provided with Timeflux. If you come from the Brain-Computer Interface field, you probably already know about the Lab Streaming Layer system. We have nodes for that (see: `timeflux.nodes.lsl`). Here we will use a *Publish/Subscribe* protocol, built on top of the ZeroMQ library, and available in the `timeflux.nodes.zmq` module. In the *Pub/Sub* pattern, subscribers express interest in topics, and receive data matching these topics. There can be more than one publisher per topic. Our implementation provides a broker that centralizes messages emitted by publishers.

This will look like this:



It can be rendered in YAML as follows:

```yaml
graphs:

  - id: Broker
    nodes:
    - id: proxy
      module: timeflux.nodes.zmq
      class: Broker

  - id: Publisher
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random
      params:
        columns: 2
        seed: 1
    - id: add
      module: timeflux_example.nodes.arithmetic
      class: Add
      params:
        value: 1
    - id: pub_before
      module: timeflux.nodes.zmq
      class: Pub
      params:
        topic: before
    - id: pub_after
      module: timeflux.nodes.zmq
      class: Pub
      params:
        topic: after
    edges:
    - source: random
      target: add
    - source: random
      target: pub_before
    - source: add
```

(continues on next page)

(continued from previous page)

```
    target: pub_after

- id: Subscriber
  nodes:
  - id: sub
    module: timeflux.nodes.zmq
    class: Sub
    params:
      topics: [ before, after ]
  - id: monitor
    module: timeflux_ui.nodes.ui
    class: UI
  edges:
    - source: sub:before
      target: monitor:before
    - source: sub:after
      target: monitor:after
```

What changed:

- We added a graph called *Broker* with only one node. Its role is to centralize the messages. It acts as a proxy between publishers and subscribers.
- Our main graph is now called *Publisher*. We replaced the `Display` nodes by `Pub` nodes. Notice that these nodes take one parameter: `topic`.
- A third graph named *Subscriber* was introduced. The `Sub` node subscribes to the two existing topics. The `UI` node is responsible for handling the web server and displaying the data. The `Sub` node has dynamic output ports, meaning that it will create output ports on the fly, named after the `topics` parameter. The `UI` has dynamic input ports, created automatically according to the second part of the `target` property of the edges. `sub:before` to `monitor:before` means: *connect the* `before` *output port of the* `sub` *node to the* `before` *input port of the* `monitor` node*.

Launch the app, and visit http://localhost:8000/monitor in your browser. From the *Streams* panel, select one signal, then select the channel you want to display (or choose *all channels*). Click the *Display* button, and voilà!

Note: Agreed, this first application does not do anything useful. But by now, you have grasped the essential concepts and are well on your way to a real-world app!

CHAPTER

SIX

A NEUROFEEDBACK APP

Let's apply what we learned and build a simple alpha neurofeedback app.

Tip: This use case assumes a little bit of background in EEG processing. If this is not the case for you, focus on the data flow rather than the processing itself, and make sure that you understand the syntax.

Note: This example does not include the sound generation. This can be accomplished with any multimedia programming software that understand the Open Sound Control protocol, such as Pure Data or Max.

The application consists of three graphs. In the main one, we assume that the EEG data is acquired through a LSL inlet. Data is accumulated into a rolling window, on which the classical Welch's method is applied with default parameters. The frequency bands are then extracted from the periodogram. Finally, the relative alpha power is sent to an OSC outlet. An external application receives this data and plays a sound when the feedback signal crosses a defined threshold. The other two graphs are not strictly required, but illustrate some important principles. The graph containing the Broker node acts as a proxy. It receives data from publishers (in our example, the raw EEG stream and the computed frequency bands) and redistributes it to subscribers. In the last graph, these two data streams are aggregated and saved to a HDF5 file.

The whole application is expressed in YAML as follows:

```yaml
graphs:

  # The publish/subscribe broker graph
  - id: PubSubBroker
    nodes:
    # Allow communication between graphs
    - id: Broker
      module: timeflux.nodes.zmq
      class: Broker

  # The main processing graph
  - id: Processing
    nodes:
    # Receive EEG signal from the network
    - id: LSL
      module: timeflux.nodes.lsl
      class: Receive
      params:
        name: signal
    # Continuously buffer the signal
    - id: Rolling
```

(continues on next page)

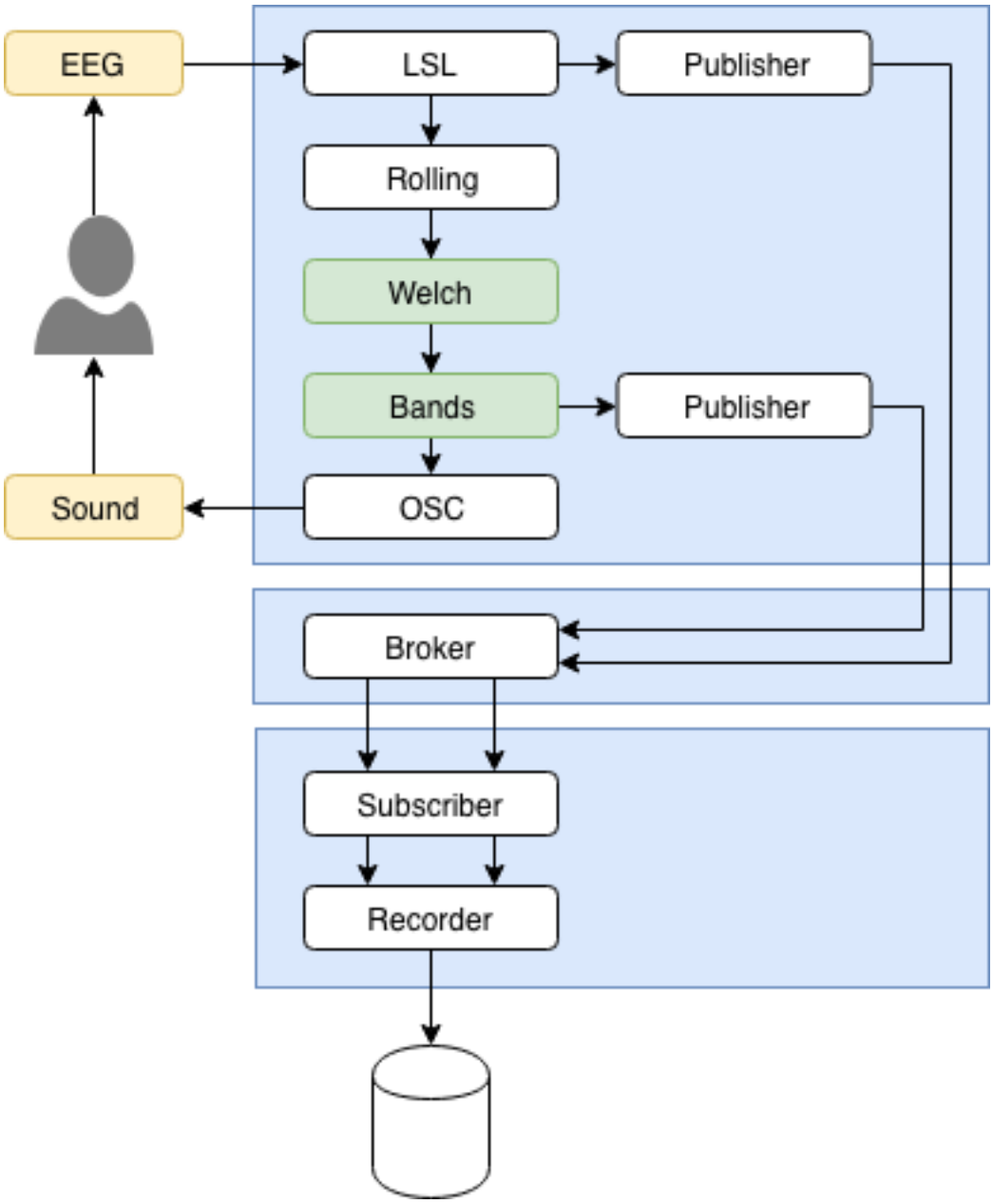


Fig. 1: Schematic representation of a basic neurofeedback application. Each blue box is a DAG. Together, they constitute the Timeflux application. The white boxes are core nodes. The green boxes are plugin nodes. Yellow boxes indicate external components.

(continued from previous page)

```
      module: timeflux.nodes.window
      class: Window
      params:
        length: 1.5
        step: 0.5
    # Compute the power spectral density
    - id: Welch
      module: timeflux_dsp.nodes.spectral
      class: Welch
    # Average the power over band frequencies
    - id: Bands
      module: timeflux_dsp.nodes.spectral
      class: Bands
    # Send to an external application
    - id: OSC
      module: timeflux.nodes.osc
      class: Client
      params:
        address: /alpha
    # Publish the raw EEG signal
    - id: PublisherRaw
      module: timeflux.nodes.zmq
      class: Pub
      params:
        topic: raw
    # Publish the frequency bands
    - id: PublisherBands
      module: timeflux.nodes.zmq
      class: Pub
      params:
        topic: bands
    # Connect nodes
    edges:
      - source: LSL
        target: Rolling
      - source: Rolling
        target: Welch
      - source: Welch
        target: Bands
      - source: Bands:alpha
        target: OSC
      - source: LSL
        target: PublisherRaw
      - source: Bands
        target: PublisherBands
    # Run this graph 25 times per second
    rate: 25

  # The recorder graph
  - id: SaveToFile
    nodes:
    # Receive data streams from the broker
    - id: Subscriber
      module: timeflux.nodes.zmq
      class: Sub
      params:
        topics:
```

(continues on next page)

(continued from previous page)

```
      - raw
      - bands
  # Record to file
  - id: Recorder
    module: timeflux.nodes.hdf5
    class: Save
  # Connect nodes
  edges:
    - source: Subscriber:raw
      target: Recorder:eeg_raw
    - source: Subscriber:bands
      target: Recorder:eeg_bands
  # Update file every second
  rate: 1
```

CHAPTER

SEVEN

GOING FURTHER

7.1 Imports

When your application becomes complex, you may want to split it into digestible and reusable parts. `import` is a special keyword you can use to combine multiple YAML files into one application.

A few examples are available here: import.yaml, import2.yaml, import3.yaml. They are self-explanatory.

7.2 Templates

You can add logic to your *YAML* files, make your apps configurable, and manipulate variables. Timeflux uses Jinja under the hood.

Take the following *app.yaml* example:

```
graphs:
  - nodes:
    - id: my_node
      module: my_module
      class: {{ FOOBAR }}
```

Setting an environment variable and invoking Timeflux:

```
timeflux -e FOOBAR="MyClass" app.yaml
```

Will render the template as:

```
graphs:
  - nodes:
    - id: my_node
      module: my_module
      class: MyClass
```

You are not limited to mere variable substitution. You have the full power of Jinja at your disposal, including control structures, macros, filters, and more.

7.3 Nodes

Explore the API reference for a list of available nodes and the *test/graphs* directory of the corresponding GitHub repositories for examples.

7.4 Tools

Useful tools and helpers can be found here: `timeflux.helpers`.

In particular, you may want to have a look at:

- `timeflux.helpers.viz` to generate images from applications, which is very useful to visually debug your application
- `timeflux.helpers.handler` if you need to deploy experiments on Windows systems and need to integrate with other software components
- `timeflux.helpers.lsl` to enumerate available LSL streams

If you are developing plugins:

- `timeflux.helpers.clock` has a set of functions to manipulate time
- `timeflux.helpers.testing` is useful to generate dummy data and simulate the graph scheduler for unit testing

7.5 Interfaces

Todo: Work in progress!

The timeflux_ui plugin exposes a powerful JavaScript API to build web apps and interact with Timeflux instances from a browser. Extensive documentation is on its way. Meanwhile, we invite you to explore the available example apps.

Part III

Extending

CHAPTER

EIGHT

YOUR FIRST PLUGIN

Todo: Work in progress! Extensive developer documentation will soon be available. Meanwhile, we invite you to explore the GitHub repositories and learn by example.

8.1 Setting a developement environment

- Go to the timeflux_example repository and click the button: *Use this template* to create a new repository from this plugin.
- Clone your new repository, modify *setup.cfg* and rename the subfolder called *timeflux_example*.
- Install the new plugin with: `pip install -e ".[dev]"`
- Work your magic.

CHAPTER

NINE

BRANCHES

Todo: Work in progress!

The *timeflux.core.branch.Branch()* class is based on the main `Node` class. Branches are useful in several cases:

- To build complex nodes by combining other nodes
- To embed Timeflux graphs in non-Timeflux applications
- To develop and test algorithms offline (for instance in a Jupyter notebook) and thus ensuring that they will also work online

See the corresponding unit test for an example.

CHAPTER

TEN

BEST PRACTICES

Todo: Work in progress!

For consistency, we invite you to respect these guidelines while developing new plugins.

10.1 Style

- Black
- PEP 8
- Google Python Style Guide

10.2 Tests

- Write tests. We use pytest.

10.3 Documentation

- Write good documentation. We use the Google Style Python Docstrings.

Part IV

Plugins

Part V

Reference

CHAPTER

ELEVEN

API REFERENCE

This page contains auto-generated API reference documentation.

timeflux

Timeflux

11.1 timeflux

timeflux.core

11.1.1 core

timeflux.core.branch

Branch base class.

branch

class timeflux.core.branch.**Branch**(*graph=None*)

Bases: *timeflux.core.node.Node*

Create instance and initialize the logger.

update(*self*)

load(*self*, *graph*)

Initialize the graph.

Parameters **graph** (*dict*) – The graph.

run(*self*)

Execute the graph once.

get_port(*self*, *node_id*, *port_id='o'*)

Get a port from the graph.

Parameters

- **node_id** (*string*) – The node id.

- **port_id** (*string*) – The port name. Default: *o*.

Returns A reference to the requested port.

Return type *Port*

set_port(*self*, *node_id*, *port_id='i'*, *data=None*, *meta=None*, *persistent=True*)
Set a port's data and meta.

Parameters

- **node_id** (*string*) – The node id.
- **port_id** (*string*) – The port name. Default: *i*.
- **data** (*DataFrame, DataArray*) – The data. Default: *None*.
- **meta** (*dict*) – The meta. Default: *None*.
- **persistent** (*boolean*) – Set the persistence of data and meta. If *True*, the port will not be cleared during graph execution. Default: *True*.

timeflux.core.exceptions

---

timeflux.core.exceptions: define exceptions

exceptions

exception timeflux.core.exceptions.**GraphDuplicateNode**(*message*, **args*)
Bases: timeflux.core.exceptions.TimefluxException

Raised when a duplicate node is found.

Create and return a new object. See help(type) for accurate signature.

exception timeflux.core.exceptions.**GraphUndefinedNode**(*message*, **args*)
Bases: timeflux.core.exceptions.TimefluxException

Raised when an undefined node is found in edges.

Create and return a new object. See help(type) for accurate signature.

exception timeflux.core.exceptions.**WorkerLoadError**(*message*, **args*)
Bases: timeflux.core.exceptions.TimefluxException

Raised when a worker cannot be loaded.

Create and return a new object. See help(type) for accurate signature.

exception timeflux.core.exceptions.**WorkerInterrupt**(*message='Interrupting'*, **args*)
Bases: timeflux.core.exceptions.TimefluxException

Raised when a process stops prematurely.

Create and return a new object. See help(type) for accurate signature.

exception timeflux.core.exceptions.**ValidationError**(*param_name*, *message*, **args*)
Bases: timeflux.core.exceptions.TimefluxException

Raised when input validation fails.

Create and return a new object. See help(type) for accurate signature.

timeflux.core.graph

---

timeflux.core.graph: handle graphs

graph

class timeflux.core.graph.**Graph**(*graph*)
Graph helpers

Initialize graph.

Parameters **graph** (*dict*) – The graph, which structure can be found in *test.yaml*. Must be an acyclic directed graph. Multiple edges between two nodes are allowed.

See also:

build()

build(*self*)
Build graph.

Returns A graph object, suitable for computation.

Return type MultiDiGraph

traverse(*self*)
Sort nodes in a format suitable for traversal.

Returns List of node ids and their predecessors.

Return type list of dicts

timeflux.core.io

---

timeflux.core.io: lightweight i/o wrapper

io

class timeflux.core.io.**Port**(*persistent=False*)

clear(*self*)

ready(*self*)

set(*self*, *rows*, *timestamps=None*, *names=None*, *meta={}*)

timeflux.core.logging

---

logging

class timeflux.core.logging.**UTCFormatterConsole**(*fmt=None*, *datefmt=None*, *style=DEFAULT_FORMAT_STYLE*, *level_styles=None*, *field_styles=None*)

Bases: coloredlogs.ColoredFormatter

Initialize a ColoredFormatter object.

Parameters

- **fmt** – A log format string (defaults to DEFAULT_LOG_FORMAT).
- **datefmt** – A date/time format string (defaults to None, but see the documentation of BasicFormatter.formatTime()).
- **style** – One of the characters %, { or $ (defaults to DEFAULT_FORMAT_STYLE)
- **level_styles** – A dictionary with custom level styles (defaults to DEFAULT_LEVEL_STYLES).
- **field_styles** – A dictionary with custom field styles (defaults to DEFAULT_FIELD_STYLES).

Raises Refer to check_style().

This initializer uses colorize_format() to inject ANSI escape sequences in the log format string before it is passed to the initializer of the base class.

converter

class timeflux.core.logging.**UTCFormatterFile**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*)

Bases: logging.Formatter

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, str.format() ({}) formatting or string.Template formatting in your format string.

Changed in version 3.2: Added the style parameter.

converter

class timeflux.core.logging.**Handler**

handle(*self*, *record*)

timeflux.core.logging.**init_listener**(*level_console='INFO'*, *level_file='DEBUG'*, *file=None*)

timeflux.core.logging.**terminate_listener**()

timeflux.core.logging.**init_worker**(*queue*)

timeflux.core.logging.**get_queue**()

timeflux.core.manager

---

timeflux.core.manager: manage workers

manager

class timeflux.core.manager.**Manager**(*config*)
Load configuration and spawn workers.

Load configuration

Parameters **config** (*string|dict*) – The configuration can either be a path to a YAML or JSON file or a dict.

run(*self*)
Launch as many workers as there are graphs.

timeflux.core.message

---

timeflux.core.message: serialize and unserialize dataframes

message

timeflux.core.message.**pickle_serialize**(*message*)

timeflux.core.message.**pickle_deserialize**(*message*)

timeflux.core.message.**msgpack_serialize**(*message*)

timeflux.core.message.**msgpack_deserialize**(*message*)

timeflux.core.node

Node base class.

node

class timeflux.core.node.**Node**
Bases: abc.ABC

Instantiate the node.

bind(*self*, *source*, *target*)
Create an alias of a port

Parameters

- **source** (*string*) – The name of the source port
- **target** (*string*) – The name of the target port

iterate(*self*, *name='*'*)
Iterate through ports.

If name ends with the globbing character (*), the generator iterates through all existing ports matching that pattern. Otherwise, only one port is returned. If it does not already exist, it is automatically created.

Parameters **name** (*string*) – The matching pattern.

Yields *(tupple)* – A tupple containing:

- name (*string*): The full port name.
- suffix (*string*): The part of the name matching the globbing character.

- port (*Port*): The port object.

clear (*self*)
Reset all ports.

It is assumed that numbered ports (i.e. those with a name ending with an underscore followed by numbers) are temporary and must be completely removed. The only exception is when they are bound to a default or named port. All other ports are simply emptied to avoid the cost of reinstanciating a new *Port* object before each update.

abstract update (*self*)
Update the input and output ports.

terminate (*self*)
Perform cleanup upon termination.

timeflux.core.registry

Share context between nodes

registry

class timeflux.core.registry.**Registry**

cycle_start

rate

effective_rate

timeflux.core.scheduler

---

timeflux.core.schedule: run nodes

scheduler

class timeflux.core.scheduler.**Scheduler** (*path*, *nodes*, *rate*)

run (*self*)

next (*self*)

terminate (*self*)

timeflux.core.sync

---

timeflux.core.sync: time synchronisation based on NTP

sync

class timeflux.core.sync.**Server**(*host=''*, *port=12300*, *now=time.perf_counter*)

start(*self*)

stop(*self*)

class timeflux.core.sync.**Client**(*host='localhost'*, *port=12300*, *rounds=600*, *timeout=1*, *now=time.perf_counter*)

sync(*self*)

stop(*self*)

timeflux.core.validate

---

validate

timeflux.core.validate.**LOGGER**

timeflux.core.validate.**RESOLVER**

timeflux.core.validate.**extend_with_defaults**(*validator_class*)
Extends the validator class to set defaults automatically.

timeflux.core.validate.**Validator**

timeflux.core.validate.**resolver**()
Load the schema and returns a resolver.

timeflux.core.validate.**validate**(*instance*, *definition='app'*)
Validate a Timeflux application or a graph.

Parameters

- **instance** (*dict*) – The application to validate.
- **definition** (*string*) – The subschema to validate against.

timeflux.core.worker

---

timeflux.core.worker: spawn processes.

worker

class timeflux.core.worker.**Worker**(*graph*)
Spawn a process and launch a scheduler.

run(*self*)
Run the process

load(*self*)

timeflux.estimators

11.1.2 estimators

timeflux.estimators.transformers

---

transformers

timeflux.estimators.transformers.shape

---

shape

class timeflux.estimators.transformers.shape.**Expand**(*axis=0*, *dimensions=3*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

fit(*self*, *X*, *y=None*)

transform(*self*, *X*)

fit_transform(*self*, *X*, *y=None*)

class timeflux.estimators.transformers.shape.**Reduce**(*axis=0*)
Bases: sklearn.base.BaseEstimator, sklearn.base.TransformerMixin

fit(*self*, *X*, *y=None*)

transform(*self*, *X*)

fit_transform(*self*, *X*, *y=None*)

timeflux.helpers

11.1.3 helpers

timeflux.helpers.background

Run a Python method in a lightweight background process.

Example

```
from timeflux.helpers.background import Task
from my.module import MyClass

task = Task(MyClass(), 'my_method', my_arg=42).start()
while not task.done:
    status = task.status()
print(status)
```

background

class timeflux.helpers.background.**Runner**
Background base class. Provides common methods.

> **Warning:** Do not use directly!

class timeflux.helpers.background.**Task**(*instance*, *method*, **args*, ***kwargs*)
Bases: *timeflux.helpers.background.Runner*

Background task.

Launch a 0MQ PAIR server, start a client and dispatch the task.

Variables **done** (*bool*) – Indicates if the task is complete.

Parameters

- **instance** (*object*) – A picklable class instance.
- **method** (*string*) – The method name to call from the instance.
- ***args** – Arbitrary variable arguments to be passed to the method.
- ****kwargs** – Arbitrary keyword arguments to be passed to the method.

start(*self*)
Run the task.

stop(*self*)
Terminate the task.

status(*self*)
Get the task status.

Returns

None if the task is not complete or a dict containing the following keys.

- `success`: A boolean indicating if the task ran successfully.
- `instance`: The (possibly modified) instance.
- `result`: The result of the method call, if *success* is *True*.
- `exception`: The exception, if *success* is *False*.
- `traceback`: The traceback, if *success* is *False*.
- `time`: The time it took to run the task.

class timeflux.helpers.background.**Worker**(*port*)
Bases: *timeflux.helpers.background.Runner*

Background worker. Connects to the server and executes the task.

> **Warning:** Do not use directly!

execute(*self*)
Get the task from the socket and run it.

timeflux.helpers.background.**port**

timeflux.helpers.clock

Time and rate helpers

clock

timeflux.helpers.clock.**now**()
Return the current time as *np.datetime64['us']*.

timeflux.helpers.clock.**float_to_time**(*timestamp*)
Convert a *np.float64* to a *np.datetime64['us']*.

timeflux.helpers.clock.**float_index_to_time_index**(*df*)
Convert a dataframe float indices to *datetime64['us']* indices.

timeflux.helpers.clock.**time_to_float**(*timestamp*)
Convert a *np.datetime64['us']* to a *np.float64*.

timeflux.helpers.clock.**min_time**(*unit='ns'*)
Return the minimum datetime for this platform.

timeflux.helpers.clock.**max_time**(*unit='ns'*)
Return the maximum datetime for this platform.

timeflux.helpers.clock.**effective_rate**(*df*)
A simple method to compute the effective rate.

timeflux.helpers.clock.**absolute_offset**()
Return the offset between the UTC timestamp and a precision timer such as the LSL precision clock.

timeflux.helpers.clock.**time_range**(*start*, *stop*, *num*)
Return `num` evenly spaced timestamps between `start` and `stop` (*np.datetime64*).

timeflux.helpers.handler

Launch and terminate Timeflux instances on both POSIX and Windows systems.

Sometimes, it's okay to kill. But only if you do it gracefully, and give your victim a chance to say goodbye to his or her children and let them commit suicide (again, with grace) upon hearing such news.

On POSIX systems, it's easy: just launch your process normally and terminate it by sending a SIGINT signal.

On Windows, well, that's another story. SIGINT can't be captured, and the only way is to send a CTRL+C event. Any other signal (except CTRL+BREAK) will terminate without notice. Simple enough, you might say. Not quite. It just happens that CTRL events can only be captured by processes attached to the current console. Which is pretty useless in most cases. But do not abandon all hope! Here is a (hacky) solution: have a launcher script that will start a simple TCP server and run your program. When a client connect to the server, it will send a CTRL+C event to its subprocess.

Use this helper as a script or invoke with: `python -m timeflux.helpers.handler [cmd] [args]`.

Example

Launching a Timeflux instance: `python -m timeflux.helpers.handler launch timeflux -d foobar.yaml`

Example

Terminating a Timeflux instance gracefully: `python -m timeflux.helpers.handler terminate`

Example

Running Timeflux from a batch script:

```
@echo off
set ROOT=C:\Users\%USERNAME%\Miniconda3
call %ROOT%\Scripts\activate.bat %ROOT%
call conda activate timeflux
start python -m timeflux.helpers.handler launch timeflux -d foobar.yaml
pause
call python -m timeflux.helpers.handler terminate
```

References

- Python issue 26350

handler

timeflux.helpers.handler.**launch_posix**(*args*)
Launch a subprocess and exit.

timeflux.helpers.handler.**launch_windows**(*args*, *port=10000*)
Launch a subprocess and await connection to a TCP server.

timeflux.helpers.handler.**terminate_posix**(*match='timeflux'*)
Find oldest Timeflux process and terminate it.

timeflux.helpers.handler.**terminate_windows**(*port=10000*)
Terminate the Timeflux process by connecting to the TCP server.

timeflux.helpers.handler.**args**

timeflux.helpers.hdf5

Enumerate groups in a HFD5 file.

hdf5

timeflux.helpers.hdf5.**info**(*fname*)

timeflux.helpers.hdf5.**fname**

timeflux.helpers.lsl

Enumerate LSL streams.

lsl

timeflux.helpers.lsl.**enumerate**()

timeflux.helpers.mne

MNE helpers

mne

timeflux.helpers.mne.**logger**

timeflux.helpers.mne.**xarray_to_mne**(*data*, *meta*, *context_key*, *event_id*, *reporting='warn'*, *ch_types='eeg'*, ***kwargs*)

Convert DataArray and meta into mne Epochs object

Parameters

- **data** (*DataArray*) – Array of dimensions ('epoch', 'time', 'space')
- **meta** (*dict*) – Dictionary with keys 'epochs_context', 'rate', 'epochs_onset'
- **context_key** (*str|None*) – key to select the context label.
- **the context is a string, context_key should be set to None.** (*If*) –
- **event_id** (*dict*) – Associates context label to an event_id that should be an int. (eg. dict(auditory=1, visual=3))
- **reporting** (*'warn'|'error'| None*) – How this function handles epochs with invalid context: - 'error' will raise a TimefluxException - 'warn' will print a warning with `warnings.warn()` and skip the corrupted epochs - `None` will skip the corrupted epochs
- **ch_types** (*list|str*) – Channel type to

Returns mne object with the converted data.

Return type epochs (mne.Epochs)

timeflux.helpers.mne.**mne_to_xarray**(*epochs*, *context_key*, *event_id*, *output='dataarray'*)

Convert mne Epochs object into DataArray along with meta.

Parameters

- **epochs** (*mne.Epochs*) – mne object with the converted data.
- **context_key** (*str|None*) – key to select the context label.

- **the context is a string, context_key should be set to None.** (*If*) –
- **event_id** (*dict*) – Associates context label to an event_id that should be an int. (eg. dict(auditory=1, visual=3))
- **output** (*str*) – type of the expected output (DataArray or Dataset)

Returns Array of dimensions ('epoch', 'time', 'space') meta (dict): Dictionary with keys 'epochs_context', 'rate', 'epochs_onset'

Return type data (DataArray|Dataset)

timeflux.helpers.port

A set of Port helpers.

port

timeflux.helpers.port.**make_event**(*label*, *data={}*)
Create an event DataFrame

Parameters

- **label** (*str*) – The event label.
- **data** (*dict*) – The optional data dictionary.

Returns Dataframe

timeflux.helpers.port.**match_events**(*port*, *label*)
Find the given label in an event DataFrame

Parameters

- **port** (Port) – The event port.
- **label** (*str*) – The string to look for in the label column.

Returns The list of matched events, or *None* if there is no match.

Return type DataFrame

timeflux.helpers.port.**get_meta**(*port*, *keys*, *default=None*)
Find a deep value in a port's meta

Parameters

- **port** (Port) – The event port.
- **keys** (*tuple|str*) – The hiearchical list of keys.
- **default** – The default value if not found.

Returns The value, or *default* if not found.

timeflux.helpers.port.**traverse**(*dictionary*, *keys*, *default=None*)
Find a deep value in a dictionary

Parameters

- **dictionary** (*dict*) – The event port.
- **keys** (*tuple|str*) – The hiearchical list of keys.

- **default** – The default value if not found.

Returns The value, or *default* if not found.

timeflux.helpers.testing

A set of tools to facilitate code testing

testing

class timeflux.helpers.testing.**DummyData**(*num_rows=1000*, *num_cols=5*, *cols=None*, *rate=10*, *jitter=0.05*, *start_date='2018-01-01'*, *seed=42*, *round=6*)

Generate dummy data.

Initialize the dataframe.

Parameters

- **num_rows** (*int*) – Number of rows
- **num_cols** (*int*) – Number of columns
- **cols** (*list*) – List of column names
- **rate** (*float*) – Frequency, in Hertz
- **jitter** (*float*) – Amount of jitter, relative to rate
- **start_date** (*string*) – Start date
- **seed** (*int*) – Seed for random number generation
- **round** (*int*) – Number of decimals for random numbers

next(*self*, *num_rows=10*)

Get the next chunk of data.

Parameters **num_rows** (*int*) – Number of rows to fetch

reset(*self*)

Reset the cursor.

class timeflux.helpers.testing.**DummyXArray**(*num_time=1000*, *num_space=5*, *rate=10*, *jitter=0.05*, *start_date='2018-01-01'*, *seed=42*, *round=6*)

Generate dummy data of type XArray.

Initialize the dataframe.

Parameters

- **num_time** (*int*) – Number of rows
- **num_space** (*int*) – Number of columns
- **rate** (*float*) – Frequency, in Hertz
- **jitter** (*float*) – Amount of jitter, relative to rate
- **start_date** (*string*) – Start date
- **seed** (*int*) – Seed for random number generation
- **round** (*int*) – Number of decimals for random numbers

next (*self*, *num_rows=10*)
Get the next chunk of data.

Parameters **num_rows** (*int*) – Number of rows to fetch

reset (*self*)
Reset the cursor.

class timeflux.helpers.testing.**ReadData** (*data*)
Generate custom data.

Initialize the dataframe.

Parameters **data** (*DataFrame*) – custom data to stream.

next (*self*, *num_rows=10*)
Get the next chunk of data.

Parameters **num_rows** (*int*) – Number of rows to fetch

reset (*self*)
Reset the cursor.

class timeflux.helpers.testing.**Looper** (*generator*, *node*, *input_port='i'*, *output_port='o'*)
Mimics the scheduler behavior to allow testing the output of a node offline.

Initialize the helper :param generator (Node): timeflux node to test :param data (Object): data generator object with a method *next* and *reset*

run (*self*, *chunk_size=None*)
Loop across chunks of a generator, update the node and return data and meta. :param chunk_size (int): number of samples per chunk :return: output_data (DataFrame): concatenated output data output_meta: list of meta

timeflux.helpers.viz

Export Timeflux apps as images.

This module can be imported or used as a standalone tool. It is quite useful to visually inspect complex apps. It offers a graphical representation of multiple direcred acyclic graphs. It uses Graphviz under the hood, and outputs the *dot* representation on the standard output, so it can optionnally be redirected to a file for further processing.

Example

```
python -m timeflux.helpers.viz foobar.yaml
```

viz

timeflux.helpers.viz.**yaml_to_png** (*filename*, *format='png'*, *sort=False*)
Generate an image from a YAML application file.

Parameters

- **filename** (*string*) – The path to the YAML application file.
- **format** (*string*) – The image format. Default: *png*.
- **sort** (*boolean*) – If *True*, the graphs will be sorted in the same topological order that is used to run the application. Default: *False*.

timeflux.nodes

11.1.4 nodes

timeflux.nodes.accumulate

Accumulation nodes that either, stack, append or, concatenate data after a gate

accumulate

class timeflux.nodes.accumulate.**AppendDataFrame**(*meta_keys=None*, ***kwargs*)
Bases: timeflux.core.node.Node

Accumulates and appends data of type DataFrame after a gate.

This node should be plugged after a Gate. As long as it receives data, it appends them to an internal buffer. When it receives a meta with key *gate_status* set to *closed*, it releases the accumulated data and empty the buffer.

Variables

- **i** (Port) – Default data input, expects DataFrame and meta
- **o** (Port) – Default output, provides DataFrame

Parameters ****kwargs** – key word arguments to pass to pandas.DataFrame.append method.

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.accumulate.**AppendDataArray**(*dim*, *meta_keys=None*, ***kwargs*)
Bases: timeflux.core.node.Node

Accumulates and appends data of type XArray after a gate.

This node should be plugged after a Gate. As long as it receives DataArrays, it appends them to a buffer list. When it receives a meta with key *gate_status* set to *closed*, it concatenates the list of accumulated DataArray, releases it and empty the buffer list.

Variables

- **i** (Port) – Default data input, expects DataArray and meta
- **o** (Port) – Default output, provides DataArray

Parameters

- **dim** – Name of the dimension to concatenate along.
- ****kwargs** – key word arguments to pass to xarray.concat method.

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.apply

Arbitrary operations on DataFrames

apply

class timeflux.nodes.apply.**ApplyMethod**(*method*, *apply_mode='universal'*, *axis=0*, *closed='right'*, *func=None*, ***kwargs*)

Bases: *timeflux.core.node.Node*

Apply a function along an axis of the DataFrame.

This node applies a function along an axis of the DataFrame. Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis` = *0*) or the DataFrame's columns (`axis` = *1*).

Variables

- **i** (*Port*) – default data input, expects DataFrame.
- **o** (*Port*) – default output, provides DataFrame.

Parameters

- **func** (*func*) – custom function specified directely that takes as input a n_array (eg. lambda x: x+1). Default: None.
- **method** (*str*) – name of the module to import, in which the method is defined. eg. *numpy.mean*.
- **apply_mode** (*str`*) – {*universal*, *reduce*, *expand* }. Default: *universal*. - *universal* if function is a transformation from n_array to n_array - *reduce* if function is a transformation from n_array to scalar - *expand* if function is a transformation from n_array to nk_array [not yet implemented]
- **axis** (*int*) – if 0, the transformation is applied to columns, if 1 to rows. Default: *0*.
- **closed** (*str*) – {*left*, *right*, *center*}: timestamp to transfer in the output, only when method_type is "reduce" and axis = 0, in which case, the output port's lenght is 1. Default: *right*.
- **kwargs** – additional keyword arguments to pass as keywords arguments to *func*.

Notes

Note that the passed function will receive ndarray objects for performance purposes. For universal functions, ie. transformation from n_array to n_array, input and output ports have the same size. For reducing function, ie. from n_array to scalar, output ports's index is set to first (if `closed` = *left*), last (if `closed` = *right*), or middle (if `closed` = *center*)

Todo: Allow expanding functions such as n_array to nk_array (with XArray usage)

Example

Universal function: in this example, we apply *numpy.sqrt* to each value of the data. Shapes of input and output data are the same.

- `method` = *numpy.sqrt*
- `method_type` = *universal*

If data in input port is `i` is:

```
                                    0
2018-10-25 07:33:41.871131          9.0
2018-10-25 07:33:41.873084          16.0
2018-10-25 07:33:41.875037          1.0
2018-10-25 07:33:41.876990          4.0
```

It returns the squared root of the data on port `o`:

```
                                    0
2018-10-25 07:33:41.871131          3.0
2018-10-25 07:33:41.873084          4.0
2018-10-25 07:33:41.875037          1.0
2018-10-25 07:33:41.876990          2.0
```

Example

Reducing function: in this example, we apply *numpy.sum* to each value of the data. Shapes of input and output data are not the same. We set:

- `method` = *numpy.sum*
- `method_type` = *reduce*
- `axis` = *0*
- `closed` = *right*

If data in input port is `i` is:

```
                                    0        1
2018-10-25 07:33:41.871131          9.0      10.0
2018-10-25 07:33:41.873084          16.0     2.0
2018-10-25 07:33:41.875037          1.0      5.0
2018-10-25 07:33:41.876990          4.0      2.0
```

It returns the sum amongst row axis on port `o`:

```
                                    0        1
2018-10-25 07:33:41.876990          30.0     19.0
```

References

See the documentation of pandas.apply .

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.axis

---

timeflux.nodes.axis: helpers to manipulate axis

axis

class timeflux.nodes.axis.**Rename**(***kwargs*)
Bases: *timeflux.core.node.Node*

Alter axes labels.

Variables

- **i** (Port) – Default data input, expects DataFrame.
- **o** (Port) – Default output, provides DataFrame and meta.

Parameters

- **kwargs** – see arguments from
- **method](https** (*[pandas.DataFrame.rename*) – //pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html)

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.axis.**RenameColumns**(*names*)
Bases: *timeflux.core.node.Node*

Rename column labels from a list

Variables

- **i** (Port) – Default data input, expects DataFrame.
- **o** (Port) – Default output, provides DataFrame and meta.

Parameters **names** (*list*) – New column names.

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.axis.**AddSuffix**(*suffix*)
Bases: *timeflux.core.node.Node*

Suffix labels with string suffix.

Variables

- **i** (Port) – Default data input, expects DataFrame.
- **o** (Port) – Default output, provides DataFrame and meta.

Parameters **suffix** (*string*) – The string to add after each column label.

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.debug

Simple debugging nodes

debug

class timeflux.nodes.debug.**Display**(*meta=False*, *data=True*)
Bases: *timeflux.core.node.Node*

Display input.

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.debug.**Dump**(*fname='/tmp/dump.csv'*)
Bases: *timeflux.core.node.Node*

Dump to CSV.

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.dejitter

Dejittering nodes

dejitter

class timeflux.nodes.dejitter.**Snap**(*rate=None*)
Bases: *timeflux.core.node.Node*

Snap time stamps to nearest occurring frequency.

Variables

- **i** (Port) – Default input, expects DataFrame and meta.
- **o** (Port) – Default output, provides DataArray and meta.

Parameters **rate** (*float | None*) – (optional) nominal sampling frequency of the data, to round the timestamps to (in Hz). If None, the rate will be get from the meta of the input port.

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.dejitter.**Interpolate**(*rate=None*, *method='cubic'*, *n_min=3*, *n_max=10*)
Bases: *timeflux.core.node.Node*

Dejitter data with values interpolation.

This nodes continuously buffers a small amount of data to allow for interpolating missing samples. The output data is resampled at a fixed rate. The interpolation is performed by Pandas methods.

Variables

- **i** (Port) – Default input, expects DataFrame and meta.
- **o** (Port) – Default output, provides DataArray and meta.

Parameters

- **rate** (*float | None*) – (optional) nominal sampling frequency of the data. If None,
- **rate will be get from the meta of the input port.** (*the*) –

- **method** – interpolation method. See the pandas.DataFrame.interpolate documentation.
- **n_min** – minimum number of samples to perform the interpolation.
- **n_max** – number of samples to keep in the buffer.

Notes

Computation cost mainly depends on the window size and the estimation is performed in the main thread. Hence, the user should be careful on the computation duration.

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.epoch

Epoching nodes

epoch

class timeflux.nodes.epoch.**Epoch**(*event_trigger*, *before=0.2*, *after=0.6*)
Bases: *timeflux.core.node.Node*

Event-triggered epoching.

This node continuously buffers a small amount of data (of a duration of `before` seconds) from the default input stream. When it detects a marker matching the `event_trigger` in the `label` column of the event input stream, it starts accumulating data for `after` seconds. It then sends the epoched data to an output stream, and sets the metadata to a dictionary containing the triggering marker and optional event data. Multiple, overlapping epochs are authorized. Each concurrent epoch is assigned its own *Port*. For convenience, the first epoch is bound to the default output, so you can avoid enumerating all output ports if you expects only one epoch.

Variables

- **i** (Port) – Default data input, expects DataFrame.
- **i_events** (Port) – Event input, expects DataFrame.
- **o** (Port) – Default output, provides DataFrame and meta.
- **o_*** (Port) – Dynamic outputs, provide DataFrame and meta.

Parameters

- **event_trigger** (*string*) – The marker name.
- **before** (*float*) – Length before onset, in seconds.
- **after** (*float*) – Length after onset, in seconds.

Example

```
graphs:

  # Nothing is displayed because the epoch does not have any event input
  - id: example
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random
    - id: epoch
      module: timeflux.nodes.epoch
      class: Epoch
      params:
        event_trigger: test
    - id: display
      module: timeflux.nodes.debug
      class: Display
    edges:
    - source: random
      target: epoch
    - source: epoch
      target: display
    rate: 10
```

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.epoch.**ToXArray**(*reporting='warn'*, *output='DataArray'*, *context_key=None*)

Bases: *timeflux.core.node.Node*

Convert multiple epochs to DataArray

This node iterates over input ports with valid epochs, concatenates them on the first axis, and creates a XArray with dimensions ('epoch', 'time', 'space') where epoch corresponds to th input ports, time to the ports data index and space to the ports data columns. A port is considered to be valid if it has meta with key 'epoch' and data with expected number of samples. If some epoch have an invalid length (which happens when the data has jitter), the node either raises a warning, an error or pass.

Variables

- **i_*** (Port) – Dynamic inputs, expects DataFrame and meta.
- **o** (Port) – Default output, provides DataArray and meta.

Parameters

- **reporting** (*string| None*) – How this function handles epochs with invalid length: *warn* will issue a warning with `warnings.warn()`, *error* will raise an exception, *None* will ignore it.
- **output** (*DataArray`|`Dataset*) – Type of output to return
- **context_key** (*string|None*) – If output type is *Dataset*, key to define the target of the event. If *None*, the whole context is considered.

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.events

Generate random events

events

class timeflux.nodes.events.**Events**(*rows_min=1*, *rows_max=10*, *string_min=3*, *string_max=12*, *items_min=0*, *items_max=5*, *seed=None*)

Bases: *timeflux.core.node.Node*

Return random integers from value_min to value_max (inclusive)

random_string(*self*, *length*)

update(*self*)

class timeflux.nodes.events.**Periodic**(*label='clock'*, *data=None*, *interval=None*, *phase=None*)

Bases: *timeflux.core.node.Node*

Node that sends events at a regular interval.

This node sends regular events after the first time the update method is called. If the update method is called at time *t*, and this node has a interval *ti* and phase *ts*, then the first event will be at *t* + *ts*. Then there will be one event at *t* + *ts* + *k ti* where *k* is 1, 2, ...

Parameters

- **label** (*str*) – Event name that will be generated by this node.
- **data** (*dict*) – Dictionary sent in each event.
- **interval** (*dict*) – Dictionary with keyword arguments passed to *datetime.timedelta* to define the time interval between events. This can be seconds, milliseconds, microseconds, etc.
- **phase** (*dict*) – Dictionary with keyword arguments passed to *datetime.timedelta* to define a phase for the stimulations. The first stimulation will happen after this time delta is observed. If not set, the phase will be as the interval.

Variables **o** (*Port*) – Default output, provides a pandas.DataFrame with events.

Examples

The following YAML can be used to generate events every half second but only after 5 seconds have elapsed

```
graphs:
  - nodes:
    - id: clock
      module: timeflux.nodes.events
      class: Periodic
      params:
        label: my-event-label
        interval:
          milliseconds: 500
        phase:
          seconds: 5
```

(continues on next page)

(continued from previous page)

```
    - id: display
      module: timeflux.nodes.debug
      class: Display

    rate: 20

    edges:
      - source: clock
        target: display
```

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.expression

---

expression

class timeflux.nodes.expression.**Expression**(*expr*, *eval_on*, ***kwargs*)

Bases: *timeflux.core.node.Node*

Evaluate a Python expression as a string.

This nodes uses *eval* method from pandas to evaluate a Python expression as a string on the data. The expression can be evaluated either on the input ports (`eval_on` = *ports*) or on the input columns (`eval_on` = *columns*).

The following arithmetic operations are supported: +, -, , /, *, %, // (python engine only) along with the following boolean operations: | (or), & (and), and ~ (not).

Variables

- **i** (Port) – default data input, expects DataFrame.
- **i_*** (Port, *optional*) – data inputs when `eval_on` is *ports*.
- **o** (Port) – default output, provides DataFrame.

Parameters

- **expr** (*str*) – Expression of the function to apply to each column or row.
- **eval_on** (*columns* | *ports*) – Variable on which the expression is evaluated. Default: *ports* If *columns*, the variables passed to the expression are the columns of the data in default input port. If *ports*, the variables passed to the expression are the data of the input ports.
- **kwargs** – Additional keyword arguments to pass as keywords arguments to *pandas.eval*: *{'parser': 'pandas', 'engine': None, 'resolvers': None,'level': None, 'target': None }*

Example

In this example, we eval arithmetic expression on the input ports : *o = i_1 + i_2*. Hence, the variables on which the expression is applied are the data of the ports.

- `expr` = *i_1 + i_2*
- `eval_on` = *ports*

The nodes expects two data inputs :

On port `i_1`:

```
                     0  1
2018-01-01 00:00:00  5  8
2018-01-01 00:00:01  9  5
2018-01-01 00:00:02 10  4
2018-01-01 00:00:03  5  5
```

On port `i_2`:

```
                     0  1
2018-01-01 00:00:00  1  3
2018-01-01 00:00:01  1  5
2018-01-01 00:00:02 10  1
2018-01-01 00:00:03  2  1
```

It returns one data output that is `i_1.data + i_2.data` :

On port `o`:

```
                     0   1
2018-01-01 00:00:00  6  11
2018-01-01 00:00:01 10  10
2018-01-01 00:00:02 20   5
2018-01-01 00:00:03  7   6
```

Example

In this example, we eval an arithmetic expression on columns : *col3 = col2 + col1* Hence, the variables on which the expression is applied are the columns of the data from default input port. We set:

- `expr` = *col2 = col1 + col0*
- `eval_on` = *columns*

The node expects data with columns *col0* and *col1* on the default port `i`:

```
                     col0  col1
2018-01-01 00:00:00     5     8
2018-01-01 00:00:01     9     5
```

It returns data with an appended column *col2* on port `o`:

```
                     col0  col1  col2
2018-01-01 00:00:00     5     8    13
2018-01-01 00:00:01     9     5    14
```

References

See the documentation of pandas.eval .

Create instance and initialize the logger.

update (*self*)

timeflux.nodes.gate

Gate node that resume or stop the streaming data

gate

class timeflux.nodes.gate.**Gate**(*event_opens*, *event_closes*, *event_label='label'*, *truncate=False*)
Bases: *timeflux.core.node.Node*

Data-gate based on event triggers.

This node cuts off or puts through data depending on event triggers It has 3 operating mode/status: - silent: the node waits for an opening trigger in the events and returns nothing - opened: the node waits for a closing trigger in the events and free pass the data - closed: the node has just received a closing trigger, so free pass the data and resets its state.

It continuously iterates over events data to update its operating mode.

Variables

- **i** (Port) – Default data input, expects DataFrame or or XArray.
- **i_events** (Port) – Event input, expects DataFrame.
- **o** (Port) – Default output, provides DataFrame or XArray and meta.

Parameters

- **event_opens** (*string*) – The marker name on which the gate open.s
- **event_closes** (*string*) – The marker name on which the the gate closes.
- **event_label** (*string*) – The column to match for event_trigger.

Todo: allow for multiple input ports.

Create instance and initialize the logger.

update (*self*)

timeflux.nodes.hdf5

---

timeflux.nodes.hdf5: HDF5 nodes

hdf5

class timeflux.nodes.hdf5.**Replay**(*filename*, *keys*, *speed=1*, *timespan=None*, *resync=True*)
Bases: *timeflux.core.node.Node*

Replay a HDF5 file.

Initialize.

Parameters

- **filename** (*string*) – The path to the HDF5 file.
- **keys** (*list*) – The list of keys to replay.
- **speed** (*float*) – The speed at which the data must be replayed. 1 means real-time.
- **timespan** (*float*) – The timespan of each chunk, in seconds. If not None, will take precedence over the *speed* parameter
- **resync** (*boolean*) – If False, timestamps will not be resync'ed to current time

update(*self*)

terminate(*self*)

class timeflux.nodes.hdf5.**Save**(*filename=None*, *path='/tmp'*, *complib='zlib'*, *complevel=9*, *min_itemsize=None*)
Bases: *timeflux.core.node.Node*

Save to HDF5.

Initialize.

Parameters

- **filename** (*string*) – Name of the file (inside the path set by parameter). If not set, an auto-generated filename is used.
- **path** (*string*) – The directory where the HDF5 file will be written.
- **complib** (*string*) – The compression lib to be used. see: https://www.pytables.org/usersguide/libref/helper_classes.html
- **complevel** (*int*) – The compression level. A value of 0 disables compression. see: https://www.pytables.org/usersguide/libref/helper_classes.html
- **min_itemsize** (*int | dict*) – The string columns size see: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.HDFStore.append.html see: http://pandas.pydata.org/pandas-docs/stable/io.html#string-columns

update(*self*)

terminate(*self*)

timeflux.nodes.lsl

Lab streaming layer nodes

The lab streaming layer provides a set of functions to make instrument data accessible in real time within a lab network. From there, streams can be picked up by recording programs, viewing programs or custom experiment applications that access data streams in real time.

lsl

class timeflux.nodes.lsl.**Send**(*name*, *type='Signal'*, *format='double64'*, *rate=0.0*, *source=None*)
Bases: *timeflux.core.node.Node*

Send to a LSL stream.

Variables **i** (Port) – Default data input, expects DataFrame.

Parameters

- **name** (*string*) – The name of the stream.
- **type** (*string*) – The content type of the stream, .
- **format** (*string*) – The format type for each channel. Currently, only `double64` and `string` are supported.
- **rate** (*float*) – The nominal sampling rate. Set to `0.0` to indicate a variable sampling rate.
- **source** (*string, None*) – The unique identifier for the stream. If `None`, it will be auto-generated.

Example

```
graphs:

  - id: Sender
    nodes:
    - id: replay
      module: timeflux.nodes.hdf5
      class: Replay
      params:
        filename: data/data.hdf5
        keys:
          - /nexus/signal/nexus_signal_raw
        timespan: 1
    - id: outlet
      module: timeflux.nodes.lsl
      class: Send
      params:
        name: test
    edges:
      - source: replay:nexus_signal_nexus_signal_raw
        target: outlet
    rate: 1

  - id: Receiver
    nodes:
    - id: inlet
      module: timeflux.nodes.lsl
      class: Receive
      params:
        prop: name
        value: test
        unit: ns
    - id: display
      module: timeflux.nodes.debug
```

(continues on next page)

(continued from previous page)

```
    class: Display
  edges:
    - source: inlet
      target: display
  rate: 1
```

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.lsl.**Receive**(*name=None*, *prop='name'*, *value=None*, *timeout=1.0*, *unit='s'*, *offset_correction=False*, *sync='local'*, *channels=None*, *max_samples=1024*)

Bases: *timeflux.core.node.Node*

Receive from a LSL stream.

Variables **o** (Port) – Default output, provides DataFrame and meta.

Parameters

- **prop** (*string*) – The property to look for during stream resolution (e.g., `name`, `type`, `source_id`).
- **value** (*string*) – The value that the property should have (e.g., `EEG` for the type property).
- **timeout** (*float*) – The resolution timeout, in seconds.
- **unit** (*string*) – Unit of the timestamps (e.g., `s`, `ms`, `us`, `ns`). The LSL library uses seconds by default. Timeflux uses nanoseconds by default.
- **sync** (*string, None*) – The method used to synchronize timestamps. Use `local` if you receive the stream from another application on the same computer. Use `network` if you receive from another computer. Use `None` if you receive from a Timeflux instance on the same computer.
- **channels** (*list, None*) – Override the channel names. If `None`, the names defined in the LSL stream will be used.
- **max_samples** (*int*) – The maximum number of samples to return per call.

Example

```
graphs:

  - id: Sender
    nodes:
    - id: random_1
      module: timeflux.nodes.random
      class: Random
    - id: outlet_1
      module: timeflux.nodes.lsl
      class: Send
      params:
        name: test_1
        type: random
    - id: random_2
      module: timeflux.nodes.random
```

(continues on next page)

(continued from previous page)

```
      class: Random
    - id: outlet_2
      module: timeflux.nodes.lsl
      class: Send
      params:
        name: test_2
    edges:
      - source: random_1
        target: outlet_1
      - source: random_2
        target: outlet_2
    rate: 1

  - id: Receiver1
    nodes:
    - id: inlet
      module: timeflux.nodes.lsl
      class: Receive
      params:
        prop: type
        value: random
        unit: ns
    - id: display
      module: timeflux.nodes.debug
      class: Display
    edges:
      - source: inlet
        target: display
    rate: 1

  - id: Receiver2
    nodes:
    - id: inlet
      module: timeflux.nodes.lsl
      class: Receive
      params:
        name: test_2
        unit: ns
    - id: display
      module: timeflux.nodes.debug
      class: Display
    edges:
      - source: inlet
        target: display
    rate: 1
```

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.ml

Machine Learning

ml

timeflux.nodes.ml.**IDLE = 0**

timeflux.nodes.ml.**ACCUMULATING = 1**

timeflux.nodes.ml.**FITTING = 2**

timeflux.nodes.ml.**READY = 3**

class timeflux.nodes.ml.**Pipeline**(*steps*, *fit=True*, *mode='predict'*, *meta_label='epoch', 'context', 'target'*, *event_start_accumulation='accumulation_starts'*, *event_stop_accumulation='accumulation_stops'*, *event_start_training='training_starts'*, *buffer_size='5s'*, *passthrough=False*, *resample=False*, *resample_direction='right'*, *resample_rate=None*, *model=None*, *cv=None*)

Bases: *timeflux.core.node.Node*

Fit, transform and predict.

Training on continuous data is always unsupervised. Training on epoched data can either be supervised or unsupervised.

If fit is *False*, input events are ignored, and not initital training is performed. Automatically set to False if mode is either 'fit_predict' or fit_transform'. Automatically set to True if mode is either 'predict', 'predict_proba' or 'predict_log_proba'.

Variables

- **i** (Port) – Continuous data input, expects DataFrame.
- **i_*** (Port) – Epoched data input, expects DataFrame.
- **i_training** (Port) – Continuous training data input, expects DataFrame.
- **i_training_*** (Port) – Epoched training data input, expects DataFrame.
- **i_events** (Port) – Event input, expects DataFrame.
- **o** (Port) – Continuous data output, provides DataFrame.
- **o_*** (Port) – Epoched data output, provides DataFrame.
- **o_events** (Port) – Event output, provides DataFrame.

Parameters

- **steps** (*dict*) – Pipeline steps and settings
- **fit** (*bool*) –
- **mode** (*'predict'|'predict_proba'|'predict_log_proba'|'transform'|'fit_predict'|* –
- **meta_label** (*str|tuple|None*) –
- **event_start_accumulation** (*str*) –
- **event_stop_accumulation** (*str*) –
- **event_start_training** (*str*) –
- **buffer_size** (*str*) –
- **passthrough** (*bool*) –
- **resample** (*bool*) –

- **resample_direction** (*'right'|'left'|'both'*) –
- **resample_rate** (*None|float*) –
- **model** – Load a pickle model - NOT IMPLEMENTED
- **cv** – Cross-validation - NOT IMPLEMENTED

Create instance and initialize the logger.

update (*self*)

terminate (*self*)

timeflux.nodes.monitor

Monitor a signal

monitor

class timeflux.nodes.monitor.**Monitor**
Bases: *timeflux.core.node.Node*

Instantiate the node.

update (*self*)

timeflux.nodes.osc

---

timeflux.nodes.osc: Simple OSC client and server

osc

class timeflux.nodes.osc.**Server**(*addresses=[]*, *ip='127.0.0.1'*, *port=5005*)
Bases: *timeflux.core.node.Node*

A simple OSC server.

Create instance and initialize the logger.

update (*self*)

terminate (*self*)

class timeflux.nodes.osc.**Client**(*address=''*, *ip='127.0.0.1'*, *port=5005*)
Bases: *timeflux.core.node.Node*

A simple OSC client.

Create instance and initialize the logger.

update (*self*)

timeflux.nodes.query

---

query

class timeflux.nodes.query.**SelectRange**(*ranges*, *axis=0*, *inclusive=False*)
Bases: *timeflux.core.node.Node*

Select a subset of the given data along vertical (index) or horizontal (columns) axis.

Variables

- **i** (Port) – default data input, expects DataFrame with eventually MultiIndex.
- **o** (Port) – default output, provides DataFrame with eventually MultiIndex.

Parameters

- **ranges** (*dict*) – Dict with keys are level names and values are selection ranges.
- **axis** (*int*) – If 0, the level concerns row index, if 1, columns index (*0* or *1*). Default: *0*.
- **inclusive** (*bool*) – Whether the boundaries are strict or included. Default: *False*.

Example

In this example, we have an input DataFrame with multi level columns and we want to select data with index from level of name *second* in range *[1,1.5]*. We set:

- `ranges` = *{"second": [1, 1.5]}*
- `axis` = *1*
- `inclusive` = *True*

If the data received on port `i` is:

```
first                                A                                 ...             B
second                             1.3       1.6       1.9                           1.3
↪    1.6       1.9
2017-12-31 23:59:59.998745401  0.185133  0.541901  0.806561    ...     0.732225
↪0.806561  0.658783
2018-01-01 00:00:00.104507143  0.692277  0.849196  0.987668    ...     0.489425
↪0.221209  0.987668
2018-01-01 00:00:00.202319939  0.944059  0.039427  0.567945    ...     0.925248
↪0.180575  0.567945
```

The data provided on port `o` will be:

```
first                                A         B
second                             1.3       1.3
2017-12-31 23:59:59.998745401  0.185133  0.732225
2018-01-01 00:00:00.104507143  0.692277  0.489425
2018-01-01 00:00:00.202319939  0.944059  0.925248
```

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.query.**XsQuery**(*key*, ***kwargs*)
Bases: *timeflux.core.node.Node*

Returns a cross-section (row(s) or column(s)) from the data.

Variables

- **i** (Port) – default input, expects DataFrame with eventually MultiIndex.
- **o** (Port) – default output, provides DataFrame with eventually MultiIndex.

Parameters

- **key** (*str|tuple*) – Some label contained in the index, or partially in a MultiIndex index.
- **axis** (*int*) – Axis to retrieve cross-section on (*0* or *1*). Default: *0*.
- **level** (*str|int|tuple*) – In case of a key partially contained in a MultiIndex, indicates which levels are used. Levels can be referred by label or position.
- **drop_level** (*bool*) – If False, returns DataFrame with same level. Default: *False*.

Example

In this example, we have an input DataFrame with multi level columns and we want to select cross section between *B* from level of name *first* and *1* from level of name *second*. We set:

- `key` = *("B", 1)*
- `axis` = *1*
- `level` = *["first", "second"]*
- `drop_level` = *False*

If the data received on port `i` is:

```
first                                A                 ...             B
second                               1         2       ...             1         2
2017-12-31 23:59:59.998745401  0.185133  0.541901     ...      0.297349  0.806561
2018-01-01 00:00:00.104507143  0.692277  0.849196     ...      0.844549  0.221209
2018-01-01 00:00:00.202319939  0.944059  0.039427     ...      0.120567  0.180575
```

The data provided on port `o` will be:

```
first                                B
second                               1
2018-01-01 00:00:00.300986584  0.297349
2018-01-01 00:00:00.396560186  0.844549
2018-01-01 00:00:00.496559945  0.120567
```

References

See the documentation of pandas.DataFrame.xs .

Parameters

- **key** (*str|tuple*) – Some label contained in the index, or partially in a MultiIndex index.
- **kwargs** – Keyword arguments to call pandas xs method: axis, level, drop_level

update(*self*)

class timeflux.nodes.query.**LocQuery**(*key*, *axis=1*)

Bases: *timeflux.core.node.Node*

Slices DataFrame on group of rows and columns by label(s)

Variables

- **i** (Port) – default data input, expects DataFrame.
- **o** (Port) – default output, provides DataFrame.

Parameters

- **key** (*str | list | tuple*) – Label selection specification.
- **axis** (*int*) – Axis to query the label from (*0* or *1*). Default: *1*.

Example

In this example, we have an input DataFrame with 5 columns *[A, B, C, D, E]* and we want to select columns A and E. We set:

- `key` = *["A", "E"]*
- `axis` = *1*

If the data received on port `i` is:

```
                                A         B       ...         E         F
2017-12-31 23:59:59.998745401  0.185133  0.541901  ...  0.806561  0.658783
2018-01-01 00:00:00.104507143  0.692277  0.849196  ...  0.221209  0.987668
2018-01-01 00:00:00.202319939  0.944059  0.039427  ...  0.180575  0.567945
```

The data provided on port `o` will be:

```
                                A         E
2017-12-31 23:59:59.998745401  0.185133  0.806561
2018-01-01 00:00:00.104507143  0.692277  0.221209
2018-01-01 00:00:00.202319939  0.944059  0.180575
```

References

See the documentation of pandas.DataFrame.loc .

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.random

---

timeflux.nodes.random: generate random data

random

class timeflux.nodes.random.**Random**(*columns=5*, *rows_min=2*, *rows_max=10*, *value_min=0*, *value_max=9*, *names=None*, *seed=None*)

Bases: *timeflux.core.node.Node*

Return random integers from value_min to value_max (inclusive)

update(*self*)

timeflux.nodes.sequence

---

timeflux.nodes.sequence: generate a sequence

sequence

class timeflux.nodes.sequence.**Sequence**
Bases: *timeflux.core.node.Node*

Generate a sequence

update(*self*)

timeflux.nodes.window

Sliding windows

window

class timeflux.nodes.window.**Window**(*length*, *step=None*, *index='time'*, *epochs=False*)
Bases: *timeflux.core.node.Node*

Provide sliding windows.

Variables **i** (Port) – Default data input, expects DataFrame.

Parameters

- **length** (*float | int*) – The length of the window, in seconds or samples.
- **step** (*float | int | None*) – The minimal sliding step, in seconds or samples. If None (the default), the step will be set to the length of the window. If 0, the data will be sent as soon as it is available.
- **index** (*string*) – If "time" (the default), the length of the window is in seconds. If "sample", the length of the window is in samples.
- **epochs** (*boolean*) – Whether the defaut output should be bound to an numbered output, thus simulating an epoch. This could be useful if piped to a Machine Learning node.

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.window.**TimeWindow**(*length*, *step=None*)
Bases: *timeflux.core.node.Node*

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.window.**SampleWindow**(*length*, *step=None*)
Bases: *timeflux.core.node.Node*

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.xarray

XArray

This module contains nodes to handle XArray data.

xarray

class timeflux.nodes.xarray.**Transpose**(*dims*)
Bases: *timeflux.core.node.Node*

Transpose dimensions of a DataArray.

This node reorders the dimensions of a DataArray object to `dims`.

Variables

- **i** (Port) – default data input, expects DataArray.
- **o** (Port) – default output, provides DataArray.

Parameters **dims** (*list, None*) – By default, reverse the dimensions. Otherwise, reorder the dimensions to this order.

Create instance and initialize the logger.

update(*self*)

class timeflux.nodes.xarray.**ToDataFrame**(*index_dim='time'*)
Bases: *timeflux.core.node.Node*

Convert XArray to DataFrame.

This node converts a XArray into a flat DataFrame with simple index given by dimension in `index_dim` and eventually MultiIndex columns (*nb_levels = n_dim -1*, where n_dim is the number of dimensions of the XArray in input).

Variables

- **i** (Port) – default data input, expects DataArray.
- **o** (Port) – default output, provides DataFrame.

Parameters **index_dim** (str, *time*) – Name of the dimension to set the index of the DataFrame.

Create instance and initialize the logger.

update(*self*)

timeflux.nodes.zmq

---

timeflux.nodes.zmq: a simple 0MQ pub/sub broker

zmq

class timeflux.nodes.zmq.**Broker**(*address_in='tcp://127.0.0.1:5559', address_out='tcp://127.0.0.1:5560'*)
Bases: *timeflux.core.node.Node*

Must run in its own graph.

Initialize frontend and backend. If used on a LAN, bind to tcp://*:5559 and tcp://*:5560 instead of localhost.

update(*self*)
Start a blocking proxy.

class timeflux.nodes.zmq.**BrokerMonitored**(*address_in='tcp://127.0.0.1:5559'*, *address_out='tcp://127.0.0.1:5560'*, *timeout=5*)

Bases: *timeflux.core.node.Node*

Run a monitored pub/sub proxy. Will shut itself down after [timeout] seconds if no data is received. Useful for unit testing and replays.

Create instance and initialize the logger.

update(*self*)

Monitor proxy

class timeflux.nodes.zmq.**BrokerLVC**(*address_in='tcp://127.0.0.1:5559'*, *address_out='tcp://127.0.0.1:5560'*, *timeout=1000*)

Bases: *timeflux.core.node.Node*

A monitored pub/sub broker with last value caching.

Create instance and initialize the logger.

update(*self*)

Main poll loop.

class timeflux.nodes.zmq.**Pub**(*topic*, *address='tcp://127.0.0.1:5559'*, *serializer='pickle'*, *wait=0*)

Bases: *timeflux.core.node.Node*

Create a publisher

update(*self*)

class timeflux.nodes.zmq.**Sub**(*topics=['']*, *address='tcp://127.0.0.1:5560'*, *deserializer='pickle'*)

Bases: *timeflux.core.node.Node*

Create a subscriber

update(*self*)

PYTHON MODULE INDEX

t

INDEX

I

L

M

N

P

R

S

T

U

V

W

X

Y