Shape-Link Documentation

Release 0.1.1.post15

Paul Müller

Nov 12, 2021

CONTENTS:

This is Shape-Link, a Python library and comman-line utility for interfacing with Shape-In, the acquisition software for RT-DC. This is the documentation of Shape-Link version 0.1.1.post15.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

To install Shape-Link, use one of the following methods:

- **from PyPI:** `pip install shapelink`
- **from sources:** `pip install .`

1.2 Citing shapelink

If you use shapelink in a scientific publication, please cite it with:

> Philipp Rosendahl, Paul Müller, Eoghan O'Connell (2021), Shape-Link version X.X.X: Python library for interfacing with Shape-In [Software]. Available at https://github.com/ZELLMECHANIK-DRESDEN/shapelink.

CHAPTER

TWO

COMMAND-LINE INTERFACE

Shape-Link comes with a command-line-interface (CLI) for running plugins.

2.1 shape-link

```
shape-link [OPTIONS] COMMAND [ARGS]...
```

2.1.1 run-plugin

Run a Shape-Link plugin file

Example usages:

```
# run a plugin
shape-link run-plugin plugins/slp_rolling_mean.py
# run a plugin with a simulator thread (for plugin testing)
shape-link run-plugin -w data.rtdc -f image,deform slp_rolling_mean.py
```

```
shape-link run-plugin [OPTIONS] PATH
```

Options

-w, --with-simulator <with_simulator>
Run the Shape-In simulator in the background using the RT-DC dataset specified (used for testing).

-f, --features <features>
Comma-separated list of features to send by the Shape-In simulator; Defaults to all innate features. A list of valid feature names can be found in the dclab docs (Advanced Usage -> Notation). The list of features will be ignored if any features are specified within the *choose_features* method of a plugin implementation.

Arguments

PATH
Required argument

2.1.2 run-simulator

Run the Shape-In simulator using data from an RT-DC dataset file

Example usage:

```
shape-link run-simulator --features image,deform /path/to/data.rtdc
```

```
shape-link run-simulator [OPTIONS] PATH
```

Options

-f, --features <features>
Comma-separated list of features to send by the Shape-In simulator; Defaults to all innate features. A list of valid feature names can be found in the dclab docs (Advanced Usage -> Notation). The list of features will be ignored if any features are specified within the *choose_features* method of a plugin implementation.

Arguments

PATH
Required argument

CHAPTER

THREE

WRITING PLUG-INS

A Shape-Link plug-in is a Python script with a class derived from *shapelink.ShapeLinkPlugin* and some additional meta data. Let's have a look at `this example plugin` which prints the rolling mean of a few scalar features to stdout:

```
import shutil

import numpy as np

from shapelink import ShapeLinkPlugin

# We use the terminal width to make sure a line doesn't get cluttered
# with prints from a previous line.
TERMINAL_WIDTH = shutil.get_terminal_size((80, 20))[0]


class RollingMeansPlugin(ShapeLinkPlugin):
    """Displays a rolling mean of a few scalar features"""
    def __init__(self, *args, **kwargs):
        super(RollingMeansPlugin, self).__init__(*args, **kwargs)
        self.window_size = 100
        self.scalar_data = {}

    def after_register(self):
        print(" Preparing for transmission")
        for feat in self.reg_features.scalars:
            self.scalar_data[feat] = np.zeros(self.window_size) * np.nan

    def after_transmission(self):
        print("\n End of transmission\n")

    def choose_features(self):
        return list()

    def handle_event(self, event_data):
        """Handle a new event"""
        window_index = event_data.id % self.window_size
        for ii, feat in enumerate(self.reg_features.scalars):
            self.scalar_data[feat][window_index] = event_data.scalars[ii]
        # print the first three features to stdout
        msgs = [" Rolling means: "]
        num_prints = min(3, len(self.reg_features.scalars))
```

(continues on next page)

(continued from previous page)

```
        for ii in range(num_prints):
            feat = self.reg_features.scalars[ii]
            msgs.append("{}: {:.3g}".format(feat,
                                            np.mean(self.scalar_data[feat])))
        line = "  ".join(msgs)
        if len(line) < TERMINAL_WIDTH:
            line += " " * (TERMINAL_WIDTH - len(line))
        print(line, end="\r", flush=True)

        return False


info = {
    "class": RollingMeansPlugin,
    "description": "Display the rolling mean of a few scalar features",
    "name": "Rolling Means",
    "version": "0.1.1",
}
```

The main action happens in the *handle_event* function. Your plugin **must** implement both this function and the *choose_features* function, which can be used to specify three lists of features (scalar, traces, images). The `Verify Aspect Ratio plugin` shows how to use the *choose_features* function. The two functions *after_register* and *after_transmission* can be used to set things up (e.g. creation of an additional output file) or to tear things down (e.g. closing that file). Use the `__init__` function for defining additional class properties. The `info` dictionary is required so that the plugin can be run via the *Command-line interface*.

CHAPTER

FOUR

INFORMATION FOR DEVELOPERS

4.1 Plug-ins

You may need to install extra dependencies for some plug-ins, such as the autofocus plugin.

```
pip install -r tests/requirements.txt
```

To write plug-ins, see the *Writing Plug-ins* section. To run plugins, see the *Command-line interface* section.

4.2 Testing

Running tests

```
pytest tests
```

If you don't wish to run the benchmarking tests (which can take some time) use

```
pytest tests --ignore=tests/benchmarking_tests
```

4.3 Benchmark Testing

For more information on benchmarking, see pytest-benchmark.readthedocs.io. One can run the benchmarking tests locally with

```
pip install -r tests/benchmarking_tests/requirements.txt
pytest tests/benchmarking_tests
```

To create a local benchmark file (with which you can compare further tests), use

```
pytest tests/benchmarking_tests --benchmark-save="NAME"
```

where "NAME" should be similar to "user_date_otherinfo" for tracking purposes, e.g., "eoghan_190321_WINpy38.json". Note that a counter is appended as a prefix in the saved file, e.g., "0001_eoghan_190321_WINpy38.json"

Then, when you need to make sure new changes aren't regressing Shape-Link, use

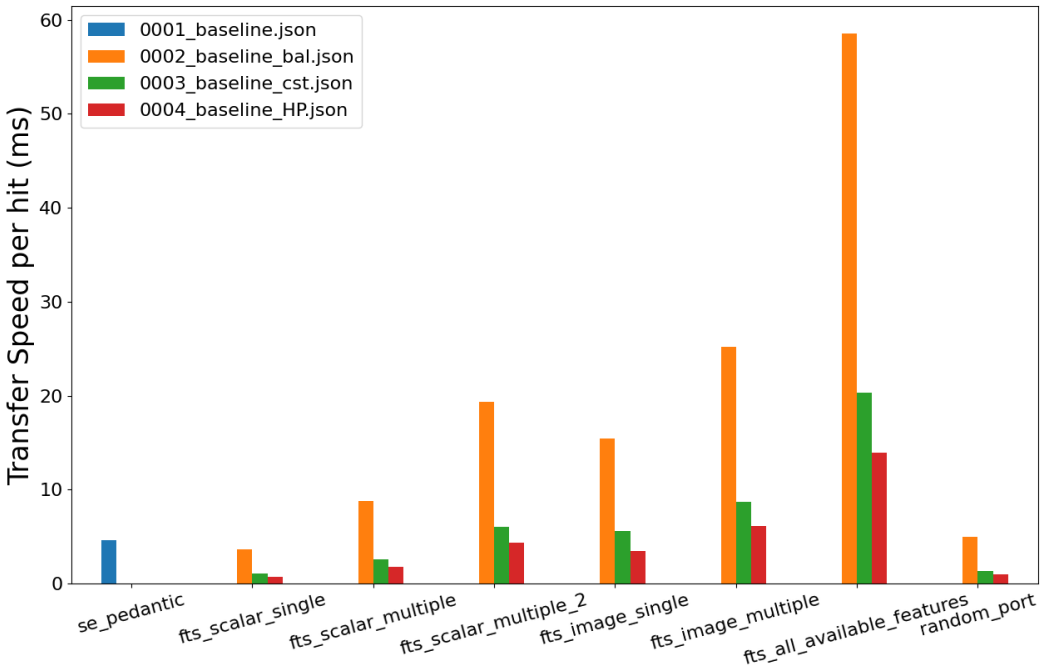
```
pytest tests/benchmarking_tests --benchmark-compare="*/0001_eoghan_21-03-19_WINpy38" --
↪benchmark-compare-fail=median:5%
```

4.3.1 Feature transfer speeds

You can output plots that compare benchmark tests. They show how long each feature transfer takes in milliseconds. This is helpful for understanding how fast each feature can be transferred during acquisition.

Note: These transfer values include the initial time taken for the server and client to connect. Therefore, the plots overestimate the transfer values. To get a more accurate transfer value, use *line_profiler (kernprof)*.

```
python tests\benchmarking_tests\benchmark_utils.py
```



These plots will be saved locally in the same directory.

4.4 Adding a new Benchmark for Github Actions

Shape-Link uses continuous integration with GitHub Actions. The benchmarking tests are run under the "Benchmark with pytest-benchmark" step. Any push or pull requests will trigger this step. To add a new benchmark file for GitHub Actions, follow the steps below:

Note: GitHub Actions currently builds a matrix of OS and Python versions. Therefore, minor warnings will appear stating that the OS or Python versions don't match the current benchmark comparison files. You can ignore this warning. We recommend using the output from the Ubuntu-py3.7 build to create the new benchmark file, as it is the slowest.

1. Push your changes. Then go to the GitHub Actions build tab on GitHub. If the benchmarking tests passed, open the "Benchmark with pytest-benchmark" output.

2. Under the "===== passed =====" log, copy the contents of the *output.json* file. Do not copy the ZMQ errors. Paste in a new *.json* file in your local repo in the *./.benchmarks_github_actions* folder. This file should be named similar to: *ActionsBenchmark_21-03-19_UBUNTUpy38.json*, where the date should change (yy-mm-dd).
3. Open the *./.github/workflows/checks.yml* file and replace the name of the –benchmark-compare="actions_benchmarks/ActionsBenchmark_190321_ubuntu_py38" to the name of your file.
4. Commit and push your changes. Now the github actions workflow will compare its live benchmark run to the new file you just created.

4.5 Using line_profiler (kernprof)

The above benchmarking tests are good but not perfect estimations of feature transfer speed. They have a flaw: the initial time taken to connect the server and client is included in the transfer speed value. This initial time is sometimes a substantial (~33%) part of the overall time.

line_profiler, formally kernprof, is a package that tests the speed of excecution of each line of code in your program. You just need to install line_profiler and decorate the relevant functions with the `@profile` decorator.

4.5.1 Installing line_profiler

See installation details on the official pyutils repository. First try to install with pip:

```
pip install line_profiler
```

If you are using windows and `pip install line_profiler` does not work, use one of these pre-built wheels. Once you have downloaded the correct wheel for your computer architecture, you can install the wheel with pip:

```
pip install path\to\the\wheel
```

4.5.2 Using line_profiler

The official pyutils repository has a short guide. Place `@profile` above the function you wish to profile. Then, run the following in terminal:

```
kernprof -l -v path/to/file.py
```

CHAPTER

FIVE

CODE REFERENCE

5.1 shapelink.shapelink_plugin

Receive data in real-time from a Shape-In instance via zmq

class shapelink.shapelink_plugin.**EventData**

class shapelink.shapelink_plugin.**ShapeLinkPlugin**(*bind_to='tcp://*:6666'*, *random_port=False*, *verbose=False*)

Shape-Link plug-in meta class

Parameters

- **bind_to** (*str*) – IP and port to bind to (where Shape-In runs)
- **random_port** (*bool*) – If set to *True*, ZMQ will use its *socket.bind_to_random_port* method. This will override only the port number of the *bind_to* ShapeLinkPlugin argument.
- **verbose** (*bool*) – Set to *True* to see additional debugging information.

after_register()
Called after registration with Shape-In is complete

after_transmission()
Called after Shape-In ends data transmission

abstract choose_features()
Abstract method to be overridden by plugins implementations.

Notes

When features are chosen by a plugin implementation, only those chosen features will be transferred between ShapeIn and the plugin. This has the effect of ignoring any features specified by the user in the –features (-f) option of the command line interface.

abstract handle_event(*event_data:* shapelink.shapelink_plugin.EventData) → bool
Abstract method to be overridden by plugins implementations

handle_messages()
Handle messages from Shape-In

Please don't override this function. Use *ShapeLinkPlugin.handle_event()* for your customized plugins.

run_EOT_message(*send_stream*)

run_event_message(*r*, *rcv_stream*)

run_features_request_message(*send_stream*)
Called before registration. The user can specify features for Shape-In to send. This limits the data being transferred. This can be useful for plugins that require only specific features.

feats is a list of three lists. The sublists are sc, tr, and im

run_register_message(*rcv_stream*, *send_stream*)

5.2 shapelink.util

Utility functions

These functions replicate QDataStreams behavior in C++. In PySide2 QDataStream does not accept array type data.

In C++ an array is serialized by writing: 1) Uint32 number of elements 2) type array elements

more significant bytes are written first. (big-endian)

if numpy "to_bytes" is used the native little-endian format appears

shapelink.util.**qstream_read_array**(*stream: PySide2.QtCore.QDataStream*, *datatype: numpy.dtype*) → numpy.array
Read array data from a stream with a specified type

shapelink.util.**qstream_write_array**(*stream: PySide2.QtCore.QDataStream*, *array: numpy.array*) → int
Write array data to a stream with a specified type :param stream: :param array: :return:

5.3 shapelink.shapein_simulator

Simulate a Shape-In instance

The communication is based on a simple REQ REP pattern all methods return when the transmission was acknowledged by the peer.

class shapelink.shapein_simulator.**ShapeInSimulator**(*destination='tcp://localhost:6666'*, *verbose=False*)

register_parameters(*scalar_reg_features=None*, *vector_reg_features=None*, *image_reg_features=None*, *image_shape=None*, *settings_names=None*, *settings_values=None*)
Register parameters that are sent to other processes

send_end_of_transmission()
Send end of transmission packet

send_event(*event_id: int*, *scalar_values: numpy.array*, *vector_values: List[numpy.array]*, *image_values: List[numpy.array]*) → bool
Send a single event to the other process

send_request_for_features()

shapelink.shapein_simulator.**start_simulator**(*path*, *features=None*, *destination='tcp://localhost:6666'*, *verbose=1*)
Run a Shape-In simulator using data from an RT-DC dataset

Parameters

- **path** (*str*) – File path to a .rtdc file

- **features** (*list, default None*) – A list of RT-DC features e.g., [“image”, “circ”, “deform”]
- **destination** (*str*) – The socket to which the ShapeInSimulator will connect. By default it is set to “tcp://localhost:6666”. These are the protocol, host and port in the form “protocol://host:port”.
- **verbose** (*int*) – Prints extra information during the transfer process, such as simulator speed. Increment to increase verbosity.

See also:

`shapelink.cli.run_simulator`

5.4 shapelink.msg_def

Definitions for message ids (numeric)

CHAPTER

SIX

CHANGELOG

List of changes in-between Shape-Link releases.

6.1 version 0.2.1

- test: tests for sorting, plugin for sorting, Emod test dataset
- test: tests for autofocus, plugin for autofocus
- test: speed tests and benchmarking for features, new benchmarking tools and docs guide
- test: added image and scalar choose feature tests, verbose tests
- fix: contour shape is now correctly reshaped by the plugin. The image features are now not repeatedly reshaped.
- docs: added how-to guide for local and remote benchmark tests
- test: added first benchmark test
- test: allow Plugin to bind to random port, fixes ZMQ Address Error
- ref: reorganised message definitions into a dict
- test: added tests for message definitions and EventData
- enh: The user can now provide a single list of features in a plugin. The feature names are mapped to the correct scalar, trace, or image list required for data transfer. Now handles trace information.
- test: updated and expanded tests for choosing features. Added tests for the mapping of features.

6.2 version 0.2.0

- feat: allow user plugins to specify a list of RT-DC features. This stops unnecessary transfer of data. This will also over-ride the -f command line interface option (#1, #4)
- test: add a test function for the ShapeLinkPlugin.choose_features method, add a plugin example for choosing features. Use pytest instead of deprecated setup.py test.
- ref: clean up of the ShapeLinkPlugin.handle_messages method (#6)
- docs: add several lines in sec_plugins for slp_verify_aspect_ratio plugin

6.3 version 0.1.3

- enh: transfer mask images as binary (#3)

6.4 version 0.1.2

- fix: Running a plugin with the CLI did not work when run without –with-simulator option
- ref: write stream data using QByteArray which significantly improves the event rate (#2)

6.5 version 0.1.1

- docs: add section for writing plugins

6.6 version 0.1.0

- feat: rudimentary command-line interface for running plugins

6.7 version 0.0.1

- initial release

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

E

H

M

P

Q

R

S