SuMPF Documentation

Release 0.16 alpha

Jonas Schulte-Coerne

Oct 29, 2019

Contents

The *SuMPF* package provides some classes, that implement offline (non-realtime) signal processing functionalities. *SuMPF* is being developed with a focus on acoustics, but it might be applicable for the analysis of other time series data as well.

Here is a brief example of *SuMPF* in action:

```
>>> import sumpf
>>> noise = sumpf.GaussianNoise(mean=0.0,
...                             standard_deviation=1.0,
...                             sampling_rate=48000.0,
...                             length=2 ** 14)
>>> filter_ = sumpf.ButterworthFilter(cutoff_frequency=1000.0, order=4, highpass=True)
>>> filtered = noise * filter_
>>> spectrum = filtered.fourier_transform()
```

CHAPTER 1

Contents

1.1 Reference

This section contains the API reference for the *SuMPF* package.

1.1.1 Signals

This section contains the API reference for the `Signal` class and its subclasses.

The base class

Waves

This section documents classes for periodic waves.

The inherited methods of the `Signal` class are not documented here.

Noise

This section documents classes for random noise signals.

The inherited methods of the `Signal` class are not documented here.

Sweeps

This section documents classes for sine sweeps and their inverses.

Linear sweeps

Exponential sweeps

Window functions

This section documents classes for window functions. Some classes require `scipy` to be available. If `scipy` is not available, these classes will be missing in the *`sumpf`* module.

The inherited methods of the `Signal` class are not documented here.

Other

This section documents classes for other signals.

The inherited methods of the `Signal` class are not documented here.

1.1.2 Spectrums

This section contains the API reference for the `Spectrum` class and its subclasses.

The base class

Pseudo-noise spectrums

This section documents classes for spectrums of pseudo-noise. These spectrums can be transformed to the time domain to have a noise signal with a defined magnitude spectrum.

The inherited methods of the `Spectrum` class are not documented here.

1.1.3 Filters

This section contains the API reference for the `Filter` class and its subclasses.

The base class

IIR filters

This section contains the API reference for classes, that implement common IIR filters.

The inherited methods of the `Filter` class are not documented here.

Bands filter

This section contains the API reference for the bands filter class.

The inherited methods of the `Filter` class are not documented here.

Simple operations

This section contains the API reference for filters, that do simple mathematical operations.

The inherited methods of the `Filter` class are not documented here.

1.1.4 Other data containers

This section contains the API reference for data containers, that do not have subclasses.

Spectrogram

This section contains the API reference for the `Spectrogram` class.

1.1.5 Signal processing blocks

This section contains the API reference for classes, that implement operations in a signal processing chain.

Combining data sets

This section documents classes, that combine multiple data sets into one.

Input/Output

This section documents classes, that interface *SuMPF* with the outside world.

1.1.6 Internal features

This section documents features, that are not part of the public API of the *SuMPF* package. They are meant to be used internally in the package and its tests and may change without further notice.

Enumerations

This section documents the enumeration classes.

Filter terms

This section documents the terms, from which the transfer functions of the `Filter` class can be built.

The base class

Primitive terms

Unary terms

Binary terms

Other Filter functionalities

This section documents some internal functionalities, that are related to the `Filter` class.

Computation related

Persistence related

Functions

This section documents internally used helper functions.

Persistence functionalities

File formats

This section documents enumeration classes, that define flags for file formats, in which *SuMPF*'s data containers can be stored.

1.2 Organisation

This section contains organisational information and instructions for the installation of the *SuMPF* package.

1.2.1 Installation

Installation from source

To retrieve the sources, the `git`-repository of the *SuMPF* package has to be cloned.

```
git clone https://github.com/JonasSC/SuMPF.git SuMPF
```

The `SuMPF` at the end of this command specifies the directory, in which the local copy of the repository shall be created. After cloning, move to that directory.

```
cd SuMPF
```

Now the *SuMPF* package can be installed system wide with the following command:

```
python3 setup.py install
```

Alternatively, the package can be installed only for the current user.

```
python3 setup.py install --user
```

1.2.2 Dependencies

Python version

The *SuMPF* package is currently developed and tested with Python 3.7. Most features should be available with Python 3.6 as well.

Other packages and tools

Many core features of *SuMPF* depend on `numpy` and `connectors`. Importing *SuMPF* will fail, if these packages are not available.

Other packages are optional, but not installing them, will reduce the number of features of SuMPF.

- some computations require `scipy`.
- saving certain audio files requires `soundfile`.
- `numexpr` is used for performance gains.
- playing back and recording audio signals is done with `jack`.
- the setup is done with `setuptools`.
- **the tests are run with `pytest`.**
  - thorough testing is achieved with `hypothesis`.
  - the test coverage is assessed with `pytest-cov`.
  - the code is analyzed with *Pylint* and `flake8`.
  - spell-checking is done with *pyenchant*.
- **the documentation is built with `sphinx`.**
  - the documentation uses the `sphinx_rtd_theme`.
  - graphs are drawn with `sphinx.ext.graphviz`.
  - some examples rely on `matplotlib`.

1.2.3 Makefile targets

The Makefile in the source code repository of the *SuMPF* package has the following targets:

- `make test` runs the unit tests.
- `make test_coverage` runs the unit tests and prints information about their test coverage.
- `make test_without_optional_dependencies` runs the unit tests with the *optional dependencies* being made unavailable, so that it's tested, if *SuMPF* degrades gracefully.
- `make lint` checks the package and the unit tests with *Pylint* and `flake8`.
- `make docs` builds the documentation.

The `test`, `test_coverage` and `test_without_optional_dependencies` targets also accept parameters, which are passed to the `pytest` call. This allows to run only specific tests, for example `make test documentation` will only run the `doctest` tests of the files in the documentation directory.

1.2.4 Licenses

LGPLv3+ for the source code

The source code of the *SuMPF* package can be distributed and modified under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. A copy of this license can be found in the source code repository in the file `LICENSE.txt` or on the website of the GNU project.

CC0 for the documentation

The documentation for the *SuMPF* package and the source code snippets in it can be distributed and modified under the terms of the CC0 license as published by the Creative Commons Corporation. This means, that the documentation and the source code snippets are practically in the public domain. The full text of the license can be found on the website of the Creative Commons Corporation.

It would be great, if the redistributed pieces of this work were marked with a reference to this project, but this is by no means mandatory.

1.3 Tutorials

This section contains tutorials, which demonstrate and explain the functionalities of the *SuMPF* package.

1.3.1 Basic concepts

This section explains the basic concepts of the *SuMPF* package.

Data containers and signal processing blocks

SuMPF provides mainly two sets of classes. One is a set of data containers, while the other classes implement signal processing operations.

The data containers are used to store measurements or analysis results, which are passed around between the signal processing steps. Data containers should be considered immutable.

The instances of signal processing steps are mutable objects, which have setter methods for data and parameters. Their getter methods return the processing results. The methods of the processing objects can be connected to each other, so that the whole processing chain is updated, once a parameter is changed. See the `connectors` package for further information on this.

While the connections between signal processing classes are handy in interactive applications, the instantiation of these classes is tedious in simple analysis scripts. For this reason, the data containers provide many methods and overloaded operators, which implement a readable API for signal processing operations.

Derived classes of data containers

So far, *SuMPF* features three base classes for signal processing related data:

- `sumpf.Signal` stores equidistantly sampled time series data.
- `sumpf.Spectrum` stores equidistantly sampled frequency domain data.
- `sumpf.Filter` provides some functions to give an analytical description of a transfer function.

The classes `Signal` and `Spectrum` are basically wrappers around `numpy.array()`s, that add metadata and convenience methods.

SuMPF provides sub-classes of these data containers, that allow the generation of specific data sets, such as `ExponentialSweep` or `ButterworthFilter`. These sub-classes take some parameters as constructor arguments and initialize the respective data container accordingly. Often they also feature additional methods to those, that are already provided by their base class.

1.3.2 Measuring the impulse responses of harmonics

This tutorial shows the measurement and analysis of the impulse response of a nonlinear system. This demonstrates the features of *SuMPF*, that are useful for writing concise analysis scripts, such as the data generation classes, their overloaded operators and other methods.

Theoretical background

A sweep excites only one frequency at a time and it starts with the lowest frequency. A nonlinear system, that is excited with a frequency does not only respond with its excitation frequency, but also with its integer multiples. This means, that when excited with a sweep, a nonlinear system responds with frequencies, that are excited at a later point in time. When computing the impulse response of the system, these components are shifted in the non-causal part of the impulse response. An exponential sweep has the convenient property, that this shift is constant for each frequency, which means, that the seemingly non-causal artifacts add up to impulse responses for the harmonic distortions.

This idea has been explored and described by Angelo Farina and Antonin Novak. Novak also developed a variation of the exponential sweep, the synchronized sweep, which allows measuring the impulse responses of the harmonics with the correct phase. In this simple example, this variation is omitted, since it is not (yet) implemented in *SuMPF*.

Importing the required packages

We need *SuMPF* for the signal processing and `matplotlib` for rendering the plots.

```
>>> import sumpf
>>> from matplotlib import pyplot
```

Defining a nonlinear system

In this section, we define a nonlinear system, that shall be measured with *SuMPF*. This is a simple function, that expects the excitation signal as an argument and returns the response signal.

```
>>> def system(excitation):
...     x = excitation
...     distorted = 0.5 * x ** 3 - 0.6 * x ** 2 + 0.1 * x + 0.02
...     highpass = sumpf.ChebyshevFilter(cutoff_frequency=100.0,
...                                      ripple=4.0,
...                                      order=4,
...                                      highpass=True)
...     lowpass = sumpf.ButterworthFilter(cutoff_frequency=3000.0,
...                                       order=2,
...                                       highpass=False)
...     filtered = distorted * highpass * lowpass
...     shifted = filtered.shift(50)
...     return shifted
```

the excitation is relabeled to `x`, so it becomes clearer, that `distorted` is basically a polynomial of the excitation signal. Note, how the `Signal` class has overloaded its math operators, so that the power, the multiplication and the addition can be written, as if `x` was an ordinary number.

`lowpass` and `highpass` are filters, that contain an analytical description of a filter's transfer function. Note, that we did not instantiate the `Filter` class and built those IIR filters from scratch. Instead, we used subclasses of `Filter`, which generate the desired transfer functions from common filter parameters.

The filters can be applied to a signal by multiplying the two. Behind the scenes, this is an element-wise multiplication in the frequency domain.

The `Signal` class has an `offset` parameter, which defines, where the first sample of the signal is located in relation to the sample of the zero point in time. This allows to create signals, that start before (negative offset) or after (positive offset) that zero point in time. The `shift()` method is used in this example to increase the offset of the system's response `Signal` by 50 samples, which means, that the response is delayed.

Creating an excitation signal

At first, we need an exponential sweep. *SuMPF* provides a subclass of `Signal` to generate one.

```
>>> sweep = sumpf.ExponentialSweep(start_frequency=20.0,
...                                stop_frequency=5000.0,
...                                interval=(4096, -4096),
...                                sampling_rate=48000,
...                                length=2 ** 16)
```

The `interval` parameter specifies, that the sweep shall sweep from the start to the stop frequency between the given sample indices. The stop index is given as a negative number, which means that it shall be counted from the back of the signal.

If a signal starts or stops abruptly, this jump in amplitude excites many frequencies at once, which spoils the sweep's property, that it only excites one frequency at a time. To avoid these abrupt jumps, the sweep must be faded in and out gently. If these fades are applied outside the interval, we know, that the specified frequency range of the sweep is unaffected by the fade.

We can now generate a signal, that defines the fade in and the fade out of the excitation signal. In accordance with the sweep's `interval` parameter, the fade in should happen in the first 4096 samples, while the fade out should affect the last 4096 samples. The fade signal is basically a mask, that rises from 0.0 to 1.0 during the rise interval, stays at 1.0 for a while and falls back to 0.0 during the fall interval.

```
>>> fade = sumpf.Fade(rise_interval=(0, 4096),
...                   fall_interval=(-4096, 1.0),
...                   sampling_rate=48000.0,
...                   length=2 ** 16)
>>> excitation = sweep * fade
```

Note how the sample indices of the `fall_interval` parameter are defined. As above, the start index is given as a negative number, which means, that the index is counted from the back of the signal. The stop index is given as a float. *SuMPF* accepts floats between 0.0 and 1.0 as sample indices, which will be multiplied with the length of the data set and then rounded to the next integer. In this case, the 1.0 means, that the fall interval shall span until the end of the signal.

The fading mask is then applied to the sweep, by multiplying the two.

Of course, the sweep has to start at a lower frequency and end at a higher frequency, than the given start and stop frequencies, because of the additional samples outside the interval. Since we know, that the nonlinear system contains a third degree polynomial, we know, that it produces nonlinearities up to the third harmonic. This means, that we should not excite more than 8kHz, because otherwise, the third harmonic will contain frequencies above 24kHz,

which is more than half the sampling rate of 48kHz and therefore will cause aliasing. Since the sweep's start and stop frequencies are defined for the given interval and the sweep continues outside that interval, we don't know the minimum and maximum frequencies, that are actually excited by the sweep. In addition to the functionality of the `Signal` class, the `ExponentialSweep` class provides methods, that compute these frequencies.

```
>>> sweep.maximum_frequency()
7417.395449686136
```

Measuring the response of our system

The response of our example system is computed by calling the function.

```
>>> response = system(excitation)
```

Computing the impulse response of the system

Since the response, that we have got from our system is not the one to an impulse, but the one to an exponential sweep, we have to compensate for the differences between the sweep and an impulse. One way to do that is to convolve the response with an inverse exponential sweep, which is a signal, whose convolution with an exponential sweep results in an impulse. *SuMPF* offers a class to create such a signal.

```
>>> inverse = sumpf.InverseExponentialSweep(start_frequency=20.0,
...                                         stop_frequency=5000.0,
...                                         interval=(4096, -4096),
...                                         sampling_rate=48000,
...                                         length=2 ** 16)
```

Note that the inverse sweep takes exactly the same parameter values as the sweep, to which it shall be the inverse.

The convolution is computed with the `convolve()` method. This method accepts a `mode`-parameter, which specifies, how the convolution shall be computed. In this case, the convolution is computed in the frequency domain, which is faster than the time domain implementations.

```
>>> impulse_response = response.convolve(inverse, mode=sumpf.Signal.convolution_modes.
↪SPECTRUM_PADDED)
```

Properties of the impulse response

Now, the impulse response can be plotted.

```
>>> pyplot.plot(impulse_response.time_samples(), impulse_response.channels()[0])     #␣
↪doctest: +SKIP
>>> pyplot.xlabel("time")                                                            #␣
↪doctest: +SKIP
>>> pyplot.ylabel("amplitude")                                                       #␣
↪doctest: +SKIP
>>> pyplot.show()                                                                    #␣
↪doctest: +SKIP
```

There are two things to point out here. First, the `Signal` class provides the method `time_samples()`, which creates an array, that contains the time values of the signal's samples. This array can be used for the x-values of the plot. And second, all data sets in *SuMPF* support multiple channels. For the `Signal` class, this means, that the `channels()` method returns a two dimensional array, in which the rows correspond to a channel. For the plot, a single channel of this array has to be extracted.

The plot shows three impulses. The largest one around time point zero is the one, that corresponds to the linear components of the system's response. The two impulses in the negative time domain are the impulse responses of the second and third harmonic.

Cutting out the impulse responses

The `ExponentialSweep` and `InverseExponentialSweep` classes provide the `harmonic_impulse_response()` method, with which the impulse responses of the harmonics can be cut out of the measured impulse response.

```
>>> harmonic1 = sweep.harmonic_impulse_response(impulse_response=impulse_response,
...                                             harmonic=1)
>>> harmonic2 = sweep.harmonic_impulse_response(impulse_response=impulse_response,
...                                             harmonic=2)
>>> harmonic3 = sweep.harmonic_impulse_response(impulse_response=impulse_response,
...                                             harmonic=3)
```

The resulting impulse responses of the harmonics are ordinary signals, that can be plotted like described above.

```
>>> for harmonic in [harmonic1, harmonic2, harmonic3]:
↪             # doctest: +SKIP
...     pyplot.plot(harmonic.time_samples(), harmonic.channels()[0], label=harmonic.
↪labels()[0])   # doctest: +SKIP
>>> pyplot.xlabel("time")
↪             # doctest: +SKIP
>>> pyplot.ylabel("amplitude")
↪             # doctest: +SKIP
>>> pyplot.legend()
↪             # doctest: +SKIP
>>> pyplot.show()
↪             # doctest: +SKIP
```

Note, that this time, the plot has a legend, that was created from the impulse responses' labels.

Note that the cut out impulse responses are all shifted to the zero point in time.

Merging the impulse responses into one multi-channel signal

For convenience (and to demonstrate that feature), the harmonics are merged into a single `Signal` instance with one channel per harmonic.

```
>>> harmonics = sumpf.MergeSignals([harmonic1, harmonic2, harmonic3]).output()
```

In a scripting application, like this tutorial, the API for merging signals is inconvenient. Rather than being a function, it requires instantiating the `MergeSignals` class and calling its `output()` method. This is due to a *design decision* in *SuMPF*, that all functionalities, that do not fit in the data container classes, are implemented in classes for signal processing blocks, that can be connected to form complex signal processing networks, in which value changes are automatically propagated.

Thanks to the `zip()` function, plotting the merged signal is a bit more convenient, than plotting the individual harmonics before.

```
>>> for channel, label in zip(harmonics.channels(), harmonics.labels()):    # doctest:␣
↪+SKIP
...     pyplot.plot(harmonics.time_samples(), channel, label=label)         # doctest:␣
↪+SKIP
```

(continues on next page)

(continued from previous page)

```
>>> pyplot.xlabel("time")                                                    # doctest:␣
↪+SKIP
>>> pyplot.ylabel("amplitude")                                               # doctest:␣
↪+SKIP
>>> pyplot.legend()                                                          # doctest:␣
↪+SKIP
>>> pyplot.show()                                                            # doctest:␣
↪+SKIP
```

Note, that now, all impulse responses have the same length. Internally, the `Signal` class uses a two-dimensional `numpy.array()`, that cannot store channels with different lengths. Therefore, the `MergeSignals` class fills missing samples with zeros.

Visualizing the transfer function

Computing the transfer function from an impulse response is done by transforming it to the frequency domain with the help of the fourier transform.

```
>>> transfer_function = harmonics.fourier_transform()
```

The resulting transfer function is stored in a `Spectrum` instance. Since the transfer function's channels store complex values, it is most common, to plot only its magnitude. The `Spectrum` class's `frequency_samples()` method provides an array of frequency values, that can be used as x-axis values for the plot.

```
>>> for magnitude, label in zip(transfer_function.magnitude(), transfer_function.
↪labels()):  # doctest: +SKIP
...     pyplot.plot(transfer_function.frequency_samples(), magnitude, label=label)    ␣
↪       # doctest: +SKIP
```

In addition to the transfer function's magnitude spectrum, it is possible to include a couple of frequencies, that have been used in this tutorial, as vertical lines in the plot:

- The red lines mark the defined start and stop frequencies of the exponential sweep.
- The black lines mark the minimum and maximum frequencies, that the sweep has excited, due to its fade in and fade out.
- The blue lines mark the cutoff frequencies of the filters in the system, that has been measured with the sweep.

```
>>> pyplot.axvline(20.0, linestyle="--", color="r")                          # doctest:␣
↪+SKIP
>>> pyplot.axvline(5000.0, linestyle="--", color="r")                        # doctest:␣
↪+SKIP
>>> pyplot.axvline(sweep.minimum_frequency(), linestyle="--", color="k")     # doctest:␣
↪+SKIP
>>> pyplot.axvline(sweep.maximum_frequency(), linestyle="--", color="k")     # doctest:␣
↪+SKIP
>>> pyplot.axvline(100.0, linestyle="--", color="b")                         # doctest:␣
↪+SKIP
>>> pyplot.axvline(3000.0, linestyle="--", color="b")                        # doctest:␣
↪+SKIP
```

And with that done, it's only a few lines of code to fine tune and display the plot.

```
>>> pyplot.xlabel("frequency")    # doctest: +SKIP
>>> pyplot.ylabel("magnitude")    # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
>>> pyplot.loglog()              # doctest: +SKIP
>>> pyplot.legend()              # doctest: +SKIP
>>> pyplot.xlim(10.0, 12000.0)   # doctest: +SKIP
>>> pyplot.ylim(0.001, 10.0)     # doctest: +SKIP
>>> pyplot.show()                # doctest: +SKIP
```

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

S

Index

S