Timeflux DSP

Release 0.3.4

Raphaëlle Bertrand-Lalo

Oct 15, 2022

CONTENTS

This plugin provides timeflux nodes and meta-nodes for real time digital signal processing of time series.

CHAPTER

ONE

INSTALLATION

First, make sure that Timeflux is installed.

You can then install this plugin in the *timeflux* environment:

```
$ conda activate timeflux
$ pip install timeflux_dsp
```

CHAPTER

TWO

MODULES

- `filters`: contains digital filters nodes (FIR, IIR, etc.) and resampling nodes.
- `spectral`: contains nodes for spectral analysis.
- `peaks`: contains nodes to detect peaks on 1D data, and estimates their characteristics.

2.1 API Reference

This page contains auto-generated API reference documentation.

timeflux_dsp

2.1.1 timeflux_dsp

timeflux_dsp.nodes

nodes

timeflux_dsp.nodes.filters

This module contains nodes for signal filtering.

filters

class timeflux_dsp.nodes.filters.**DropRows**(*factor*, *method=None*)

Bases: timeflux.core.node.Node

Decimate signal by an integer factor.

This node uses Pandas computationally efficient functions to drop rows. By default, it simply transfers one row out of `factor` and drops the others. If `method` is *mean* (resp. median), it applies a rolling window of length equals `factor`, computes the mean and returns one value per window. It maintains an internal state to ensure that every k'th sample is picked even across chunk boundaries.

Variables

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

Parameters

- **factor** (*int*) – Decimation factor. Only every k'th sample will be
- **output.** (*transferred into the*) –
- **method** (*str|None*) – Method to use to drop rows. If *None*, the values are transferred as it. If *mean* (resp. *median*), the mean (resp. median) of the samples is taken.

Example

```
graphs:

  - id: example
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random

    - id: droprows
      module: timeflux_dsp.nodes.filters
      class: DropRows
      params:
        factor: 2

    - id: display
      module: timeflux.nodes.debug
      class: Display
    edges:
    - source: random
      target: droprows
    - source: droprows
      target: display
    rate: 10
```

Example

In this exemple, we generate white noise to stream and we drop one sample out of two using DropRows, setting:

- `factor` = *2*
- `method` = *None* (see orange trace) | `method` = *"mean"* (see green trace)

Notes

Note that this node is not supposed to dejitter the timestamps, so if the input chunk is not uniformly sampled, the output chunk won't be either.

Also, this filter does not implement any anti-aliasing filter. Hence, it is recommended to precede this node by a low-pass filter (e.g., FIR or IIR) which cuts out below half of the new sampling rate.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**Resample**(*factor*, *window=None*)

Bases: timeflux.core.node.Node

Resample signal.

This node calls the *scipy.signal.resample* function to decimate the signal using Fourier method.

Variables

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

Parameters

- **factor** (*int*) – Decimation factor. Only every k'th sample will be transferred into the output.
- **window** (*str|list|float*) – Specifies the window applied to the signal in the Fourier domain. Default: *None*.

Example

```
graphs:

  - id: example
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random
    - id: window
      module: timeflux.nodes.window
      class: Window
      params:
        length: 2
    - id: resample
      module: timeflux_dsp.nodes.filters
      class: Resample
      params:
        factor: 2
        window: null

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

(continues on next page)

(continued from previous page)

```
  edges:
  - source: random
    target: window
  - source: window
    target: resample
  - source: resample
    target: display
  rate: 10
```

Notes

This node should be used after a buffer to assure that the FFT window has always the same length.

References

- scipy.signal.resample

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**IIRFilter**(*frequencies=None*, *rate=None*, *filter_type='bandpass'*, *sos=None*, ***kwargs*)

Bases: timeflux.core.node.Node

Apply IIR filter to signal.

If `sos` is *None*, this node uses adapted methods from mne.filters to design the filter coefficients based on the specified parameters. If no transition band is given, default is to use :

- l_trans_bandwidth = min(max(l_freq * 0.25, 2), l_freq)
- h_trans_bandwidth = min(max(h_freq * 0.25, 2.), rate / 2. - h_freq)

Else, it uses `sos` as filter coefficients.

Once the kernel has been estimated, the node applies the filtering to each columns in `columns` using *scipy.signal.sosfilt* to generate the output given the input, hence ensures continuity across chunk boundaries,

Variables

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

Parameters

- **rate** (*float*) – Nominal sampling rate of the input data. If None, rate is get from the meta.
- **order** (*int, optional*) – Filter order. Default: *None*.
- **frequencies** (*list|None*) – Transition frequencies. Ignored when sos is given.
- **filter_type** (*str|None*) – Filter mode (*lowpass*, *highpass*, *bandstop*, *bandpass*). Default: *bandpass*. Ignored when sos is given.
- **sos** (*array|None, optional*) – Array of second-order sections (sos) representation, must have shape (n_sections, 6). Default: *None*.

- **kwargs** – keyword arguments to pass to the filter constructor

Example

In this example, we generate a signal that is the sum of two sinus with respective periods of 1kHz and 15kHz and respective amplitudes of 1 and 0.5. We stream this signal using the IIRFilter node, designed for lowpass filtering at cutoff frequency 6kHz, order 3.

- `order` = *3*
- `freqs` = *[6000]*
- `mode` = *'lowpass'*

We plot the input signal, the output signal and the corresponding offline filtering.

Notes

This node ensures continuity across chunk boundaries, using a recursive algorithm, based on a cascade of biquads filters.

The filter is initialized to have a minimal step response, but needs a 'warm up' period for the filtering to be stable, leading to small artifacts on the first few chunks.

The IIR filter is faster than the FIR filter and delays the signal less but this delay is not constant and the stability not ensured.

References

- Real-Time IIR Digital Filters
- scipy.signal.sosfilt

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**IIRLineFilter**(*rate=None*, *edges_center=(50, 60, 100, 120)*, *orders=(2, 1, 1, 1)*, *edges_width=(3, 3, 3, 3)*)

Bases: timeflux.core.node.Node

Apply multiple Notch IIR Filter in series.

Attributes: i (Port): Default input, expects DataFrame. o (Port): Default output, provides DataFrame.

Parameters

- **rate** (*float*) – Nominal sampling rate of the input data. If None, rate is get from the meta.
- **edges_center** – List with center of the filters.
- **orders** (*tuple* | *int* | *None*) – List with orders of the filters. If int, the same order will be used for all filters. If None, order 2 will be used for all filters.
- **edges_width** – List with orders of the filters. If int, the same order will be used for all filters. If None, width of 3 (Hz) will be used for all filters.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**FIRFilter**(*frequencies*, *rate=None*, *columns='all'*, *order=20*, *filter_type='bandpass'*, *coeffs=None*, ***kwargs*)

Bases: timeflux.core.node.Node

Apply FIR filter to signal.

If `coeffs` is *None*, this node uses adapted methods from *mne.filters* to design the filter coefficients based on the specified parameters. If no transition band is given, default is to use:

- l_trans_bandwidth = min(max(l_freq * 0.25, 2), l_freq)
- h_trans_bandwidth = min(max(h_freq * 0.25, 2.), fs / 2. - h_freq)

Else, it uses `coeffs` as filter coefficients.

It applies the filtering to each columns in `columns` using *scipy.signal.lfilter* to generate the output given the input, hence ensures continuity across chunk boundaries,

The delay introduced is estimated and stored in the meta `FIRFilter`, `delay`.

Variables

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

Parameters

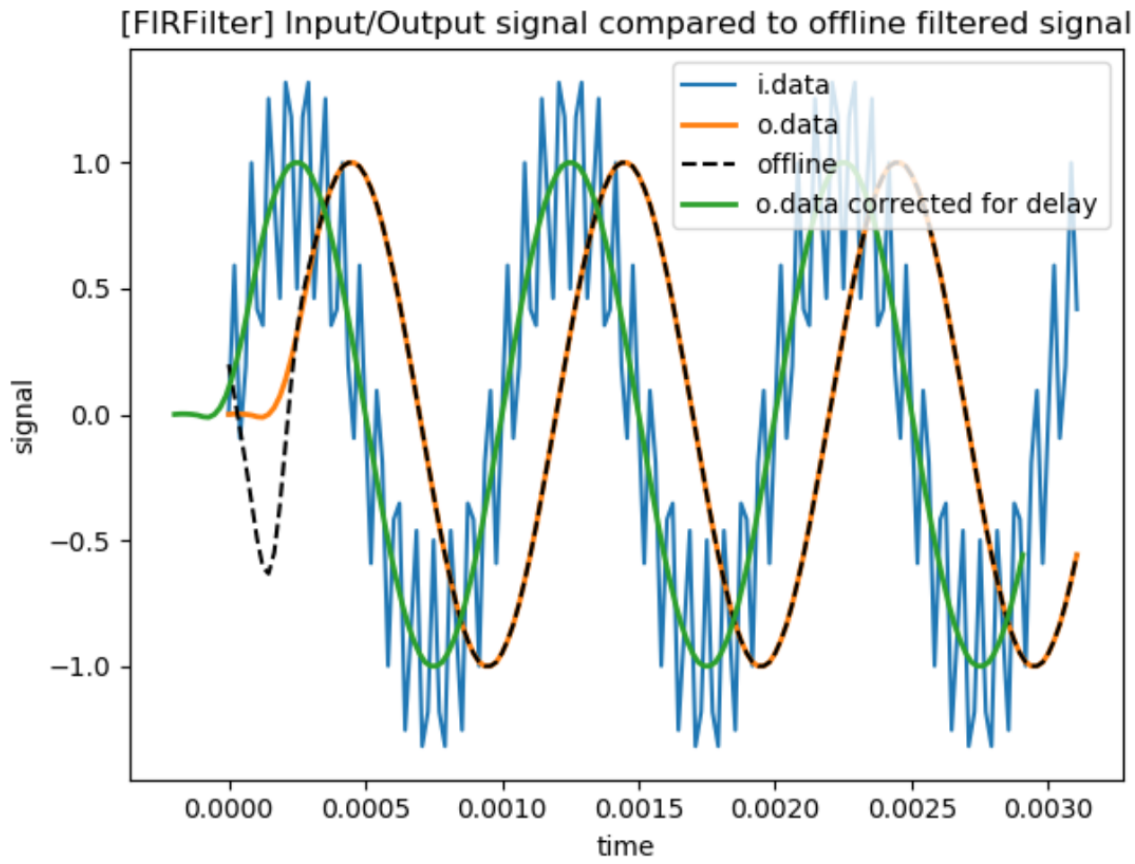
- **rate** (*float*) – Nominal sampling rate of the input data. If None, rate is get from the meta.
- **columns** (*list|'all', optional*) – Columns to apply filter on. Default: *all*.
- **order** (*int*) – Filter order.
- **frequencies** (*list*) – Transition frequencies.
- **filter_type** (*str, optional*) – Filter mode (*lowpass*, *highpass*, *bandstop* or *bandpass*). Default: *bandpass*.
- **coeffs** (*array|None, optional*) – Custom coeffs to pass as `b` in *signal.filter*. Default: *None*.
- **kwargs** – keyword arguments to pass to the filter constructor (window, phase,...)

Example

In this exemple, we generate a signal that is the sum of two sinus with respective periods of 1kHz and 15kHz and respective amplitudes of 1 and 0.5. We stream this signal using the FIRFilter node, designed for lowpass filtering at cutoff frequency 6kHz, order 20.

- `order` = *20*
- `freqs` = *[6000, 6100]*
- `mode` = *'lowpass'*

The FIR is a linear phase filter, so it allows one to correct for the introduced delay. Here, we retrieve the input sinus of period 1kHz. We plot the input signal, the output signal, the corresponding offline filtering and the output signal after delay correction.



Notes

The FIR filter ensures a linear phase response, but is computationnaly more costly than the IIR filter.

The filter is initialized to have a minimal step response, but needs a 'warmup' period for the filtering to be stable, leeding to small artifacts on the first few chunks.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**Scaler**(*method='sklearn.preprocessing.StandardScaler'*, ***kwargs*)

Bases: timeflux.core.node.Node

Apply a sklearn scaler

Variables

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

Parameters

- **method** (*str*) – Name of the scaler object (see https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing)
- ****kwargs** – keyword arguments to initialize the scaler.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**AdaptiveScaler**(*length*, *method='sklearn.preprocessing.StandardScaler'*, *dropna=False*, ***kwargs*)

Bases: timeflux.nodes.window.TimeWindow

Scale the data adaptively. This nodes transforms the data using a sklearn scaler object that is continuously fitted on a rolling window.

Variables

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

Parameters

- **length** (*float*) – The length of the window, in seconds.
- **method** (*str*) – Name of the scaler object (see https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing)
- **dropna** (*bool*) – Whether or not NaN should be dropped before fitting the estimator. Default to False.
- ****kwargs** – keyword arguments to initialize the scaler.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.filters.**FilterBank**(*filters*, *method='IIRFilter'*, *rate=None*, ***kwargs*)

Bases: timeflux.core.branch.Branch

Apply multiple IIR Filters to the signal and stack the components horizontally

Variables

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

Parameters

- **rate** (*float*) – Nominal sampling rate of the input data. If None, rate is get from the meta.
- **filters** (*dict|None*) – Define the iir filter to apply given its name and its params.

Instantiate the node.

update()

Update the input and output ports.

timeflux_dsp.nodes.helpers

helpers

class timeflux_dsp.nodes.helpers.**Concat**(*axis=1*, ***kwargs*)

Bases: timeflux.core.node.Node

Concat list of data ports . :ivar i_*: Dynamic inputs, expects DataFrame and meta. :vartype i_*: Port :ivar o: Default output, provides DataFrame. :vartype o: Port

Parameters

- **axis** (*str* | *int*) – The axis to concatenate along.
- **kwargs** – Keyword arguments to pass to pd.concat function

Notes

There is no shape verification in this node, that won't result in any Exception but may introduce NaN in the DataFrame. The responsability is left to the user.

Instantiate the node.

update()

Update the input and output ports.

timeflux_dsp.nodes.peaks

---

peaks

class timeflux_dsp.nodes.peaks.**LocalDetect**(*delta*, *tol*, *reset=None*)

Bases: timeflux.core.node.Node

Detect peaks and valleys in live 1D signal

This node uses a simple algorithm to detect peaks in real time. When a local extrema (peak or valley) is detected, an event is sent with the nature specified in the label column (`peak`/ `valley`) andcthe characteristics in the data column, giving:

- **value**: Amplitude of the extrema.
- **lag**: Time laps between the extrema and its detection.
- **interval**: Duration between two extrema os same nature.

Variables

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

Parameters

- **delta** (*float*) – Threshold for peak/valley matching in amplitude. This can be seen as the minimum significant change enough to detect a peak/valley.
- **tol** (*float*) – Tolerence for peak/valley matching, in seconds. This can be seen as the minimum time difference between two peaks/valleys.
- **reset** (*float*) – Reset threshold, in seconds. This can be seen as the maximum duration of plausible transitions between peaks and valleys. Default: None.

Example

```yaml
graphs:

  - id: example
    nodes:
    - id: random
      module: timeflux.nodes.random
      class: Random

    - id: droprows
      module: timeflux_dsp.nodes.filters
      class: DropRows
      params:
        factor: 2

    - id: display
      module: timeflux.nodes.debug
      class: Display
    edges:
    - source: random
      target: droprows
    - source: droprows
      target: display
    rate: 10
```

Example

In this example, we stream a photoplethysmogram signal scaled between -1 and 1 and we use the node RealTimeDetect to detect peaks and valleys.

- `delta` = *0.1*
- `tol` = *0.5*
- `reset` = None

self.o.data:

```
                             label                                                 ␣
↪data
2018-11-19 11:06:39.620900000    peak  {'value': [1.0054607391357422], 'lag': 0.
↪03125, 'interval': 0.654236268}
2018-11-19 11:06:39.794709043  valley  {'value': [-1.0110111236572266], 'lag': 0.
↪046875, 'interval': 0.654236268}
2018-11-19 11:06:40.605209027    peak  {'value': [0.9566234350204468],  'lag': 0.
↪033197539, 'interval': 0.984309027}
2018-11-19 11:06:40.761455675  valley  {'value': [-1.0549353361129759], 'lag': 0.
↪048816963, 'interval': 0.810499984}
```

Notes

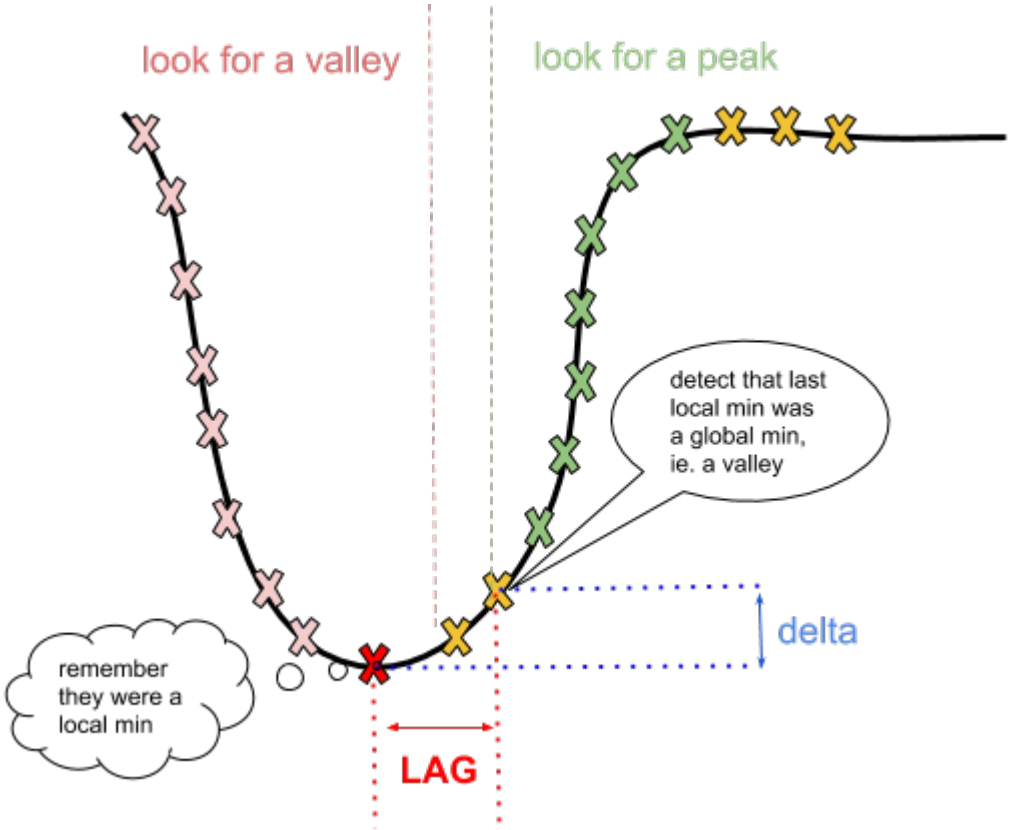
This peak detection is considered real-time since it does not require buffering the data. However, the detection method necessarily involve a lag between the actual peak and its detection. Indeed, the internal state of the node is either "looking for a peak" or "looking for a valley".

- **If the node is "looking for a peak", it means it computes local maxima**
  until the signal drops significantly (ie. more than `delta`)
- **If the node is "looking for a valley", it means it computes local minima**
  until the signal rises significantly (ie. more than `delta`)

The "last local extrema" is set to a peak (resp. valley) as soon as the signal drops (resp. rises) significantly. Hence, there is an intrinsic lag in the detection, that is directly linked to parameter `delta`.

Hence, by decreasing delta, we minimize the lag. But if delta is too small, we'll suffer from false positive detection, unless `tol` is tuned to avoid too closed detections. The parameters should be tuned depending on the nature of the data, ie. their dynamic, quality, shapes.

See the illustration above:



References

- Matlab function
- Publication

Todo

- allow for adaptive parametrization.

Instantiate the node.

`update()`

Update the input and output ports.

`class timeflux_dsp.nodes.peaks.RollingDetect(length: object = 0.5, tol: object = 0.1, rate: object = None)`

Bases: `timeflux.core.node.Node`

Detect peaks and valleys on a rolling window of analysis in 1D signal This node uses a buffer to compute local extrema and detect peaks in real time. When a local extrema (peak or valley) is detected, an event is sent with the nature specified in the label column ("peak"/"valley) and the characteristics in the data column, giving:

- **value**: Amplitude of the extrema.
- **lag**: Time laps between the extrema and its detection.
- **interval**: Duration between two extrema os same nature.

Parameters

- **window** (*float*) – Window of analysis in seconds, on which local max/min is computed.
- **tol** (*float*) – Tolerance for peak/valley matching, in seconds.
- **difference** (*This can be seen as the minimum time*) –
- **peaks/valleys.** (*between two*) –

Variables

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

Instantiate the node.

`update()`

Update the input and output ports.

`class timeflux_dsp.nodes.peaks.Rate(event_trigger='peak', event_label='label')`

Bases: `timeflux.core.node.Node`

Computes rate of an event given its label.

This node computes the inverse duration (ie. instantaneous rate) between onsets of successive events with a marker matching the `event_trigger` in the `event_label` column of the event input,

Variables

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

Parameters

- **event_trigger** (*string*) – The marker name.
- **event_label** (*string*) – The column to match for event_trigger.

Instantiate the node.

update()

Update the input and output ports.

timeflux_dsp.nodes.quality

---

quality

class timeflux_dsp.nodes.quality.**Discretize**(*range*, *default=None*)

Bases: timeflux.core.node.Node

Discretize data based on amplitude range.

Attributes: i (Port): Default input, expects DataFrame. o (Port): Default output, provides DataFrame.

Parameters

- **range** (*dict*) – Dictionary with keys are discrete value and values
- **ranges.** (*are tuple with corresponding data*) –
- **default** – Default discrete value (for data that are not contained in any range)

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.quality.**ECGQuality**(*rate*, *length*, *step*)

Bases: timeflux.nodes.window.Window

Estimate ECG Quality

This nodes estimates ECG Quality using neurokit toolbox, by applying function *ecg_process* on a rolling window.

Variables

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

Parameters

- **rate** (*float*) – Nominal sampling rate of the input data. If None, rate is get from the meta.
- **length** (*float*) – The length of the window, in seconds.
- **step** (*float*) – The sliding step, in seconds.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.quality.**LineQuality**(*rate*, *range*, *window_length=3*, *window_step=0.5*, *bandpass_frequencies=(1, 65)*, *line_centers=(50, 100, 150)*)

Bases: timeflux.core.branch.Branch

Estimate level of line noise

This nodes estimates LineNoise as the ratio between good power and total power on a rolling window. Good power is defined as the sum of squared samples for signal after bandpass and Notch filtering. Total power is defined as the sum of squared samples for signal after bandpass filtering only.

Variables

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

Parameters

rate – Nominal sampling rate of the input data. If None, rate is get from the meta.

Instantiate the node.

update()

Update the input and output ports.

class timeflux_dsp.nodes.quality.**AmplitudeQuality**(*range*, *window_length=3*, *window_step=0.5*, *method='ptp'*)

Bases: timeflux.core.branch.Branch

Estimate discrete signal quality index based on a temporal feature from the amplitude.

This nodes rolls a window and applies a numpy function given by `method` (eg. ptp, max, min, mean...) over rows and discretize the result based on `range` .

Variables

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

Instantiate the node.

update()

Update the input and output ports.

timeflux_dsp.nodes.spectral

This module contains nodes for spectral analysis with Timeflux.

spectral

class timeflux_dsp.nodes.spectral.**FFT**(*fs=1.0*, *nfft=None*, *return_onesided=True*)

Bases: timeflux.core.node.Node

Compute the one-dimensional discrete Fourier Transform for each column using the Fast Fourier Tranform algorithm.

Variables

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

Example

In this exemple, we simulate a white noise and we apply FFT:

- `fs` = *10.0*
- `nfft` = *5*
- `return_onesided` = *False*

self.i.data:

```
                                   A         B         C
2017-12-31 23:59:59.998745401  0.185133  0.541901  0.872946
2018-01-01 00:00:00.104507143  0.732225  0.806561  0.658783
2018-01-01 00:00:00.202319939  0.692277  0.849196  0.249668
2018-01-01 00:00:00.300986584  0.489425  0.221209  0.987668
2018-01-01 00:00:00.396560186  0.944059  0.039427  0.705575
```

self.o.data:

```
xarray.DataArray (times: 1, freqs: 5, space: 3)
array([[[ 3.043119+0.j      ,  2.458294+0.j      ,  3.47464 +0.j      ],
        [-0.252884+0.082233j, -0.06265 -1.098709j,  0.29353 +0.478287j],
        [-0.805843+0.317437j,  0.188256+0.146341j,  0.151515-0.674376j],
        [-0.805843-0.317437j,  0.188256-0.146341j,  0.151515+0.674376j],
        [-0.252884-0.082233j, -0.06265 +1.098709j,  0.29353 -0.478287j]]])
Coordinates:
  * times    (times) datetime64[ns] 2018-01-01T00:00:00.396560186
  * freqs    (freqs) float64 0.0 2.0 4.0 -4.0 -2.0
  * space    (space) object 'A' 'B' 'C'
```

Notes

This node should be used after a buffer.

References

- scipy.fft

Parameters

- **fs** (*float*) – Nominal sampling rate of the input data.
- **nfft** (*int | None*) – Length of the Fourier transform. Default: length of the chunk.
- **return_onesided** (*bool*) – If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. (Note that for complex data, a two-sided spectrum is always returned.) Default: *True*.

`update()`

Update the input and output ports.

class timeflux_dsp.nodes.spectral.**Welch**(*rate=None*, *closed='right'*, ***kwargs*)

Bases: timeflux.core.node.Node

Estimate power spectral density using Welch's method.

Variables

- **i** (*Port*) – default input, expects DataFrame.
- **o** (*Port*) – default output, provides DataArray with dimensions (time, freq, space).

Example:

In this exemple, we simulate data with noisy sinus on three sensors (columns *a*, *b*, *c*):

- fs = *100.0*
- nfft = *24*

node.i.data::

s a b c 1970-01-01 00:00:00.000 -0.233920 -0.343296 0.157988 1970-01-01 00:00:00.010 0.460353 0.777296 0.957201 1970-01-01 00:00:00.020 0.768459 1.234923 1.942190 1970-01-01 00:00:00.030 1.255393 1.782445 2.326175 … … … … 1970-01-01 00:00:01.190 1.185759 2.603828 3.315607

node.o.data:

```
<xarray.DataArray (time: 1, freq: 13, space: 3)>
array([[[2.823924e-02, 1.087382e-01, 1.153163e-01],
    [1.703466e-01, 6.048703e-01, 6.310628e-01],
    ...            ...           ...
    [9.989429e-04, 8.519226e-04, 7.769918e-04],
    [1.239551e-03, 7.412518e-04, 9.863335e-04],
    [5.382880e-04, 4.999334e-04, 4.702757e-04]]])
Coordinates:
    * time     (time) datetime64[ns] 1970-01-01T00:00:01.190000
    * freq     (freq) float64 0.0 4.167 8.333 12.5 16.67 ... 37.5 41.67 45.83 50.0
    * space    (space) object 'a' 'b' 'c'
```

Notes

This node should be used after a Window with the appropriate length, with regard to the parameters *noverlap*, *nperseg* and *nfft*. It should be noted that a pipeline such as {LargeWindow-Welch} is in fact equivalent to a pipeline {SmallWindow-FFT-LargeWindow-Average} with SmallWindow 's parameters *length* and *step* respectively equivalent to *nperseg* and *step* and with FFT node with same kwargs.

Parameters

- **rate** (*float* | *None*) – Nominal sampling rate of the input data. If *None*, the rate will be taken from the input meta/
- **closed** (*str*) – Make the index closed on the *right*, *left* or *center*.
- **kwargs** – Keyword arguments to pass to scipy.signal.welch function. You can specify: window, nperseg, noverlap, nfft, detrend, return_onesided and scaling.

update()

Update the input and output ports.

class timeflux_dsp.nodes.spectral.**Bands**(*bands=None*, *relative=False*)

Bases: timeflux.core.node.Node

Averages the XArray values over freq dimension according to the frequencies bands given in arguments.

This node selects a subset of values over the chosen dimensions, averages them along this axis and convert the result into a flat dataframe. This node will output as many ports bands as given bands, with their respective name as suffix.

Attributes:

i (Port): default output, provides DataArray with 3 dimensions (time, freq, space). o (Port): Default output, provides DataFrame. o_* (Port): Dynamic outputs, provide DataFrame.

Parameters

bands (*dict*) – Define the band to extract given its name and its range. An output port will be created with the given names as suffix.

update()

Update the input and output ports.

timeflux_dsp.nodes.squashing

---

squashing

class timeflux_dsp.nodes.squashing.**Discretize**(*range*, *default=None*)

Bases: timeflux.core.node.Node

Discretize data based on defined ranges :ivar i: Default input, expects DataFrame. :vartype i: Port :ivar o: Default output, provides DataFrame.

Parameters

- **range** (*dict*) – dictionary with keys corresponding to the discrete labels and values are lists of tuple with boundaries.
- **default** (*float* | *str* | *None*) –

Instantiate the node.

update()

Update the input and output ports.

timeflux_dsp.nodes.time

This module contains nodes for time index processing

time

class timeflux_dsp.nodes.time.**DelayIndex**(*delay=None*)

Bases: timeflux.core.node.Node

Helper class that provides a standard way to create an ABC using inheritance.

Instantiate the node.

to_timedelta(*duration*)

update()

Update the input and output ports.

timeflux_dsp.utils

---

utils

timeflux_dsp.utils.filters

---

filters

timeflux_dsp.utils.filters.**LOGGER**

timeflux_dsp.utils.filters.**design_edges**(*frequencies*, *nyq*, *mode*)

Design filter edges.

Parameters

- **frequencies** (*list*) – Transition frequencies in Hz.
- **mode** (*str*) – Filter mode (*lowpass*, *highpass*, *bandstop* or *bandpass*).
- **nyq** – Nyquist frequency (half sampling rate).

Returns

freqs. Updated frequencies with transition bands. array: gains. Filter gain at frequency sampling points. list: wp. Passband edge frequencies. list: ws. Stopband edge frequencies.

Return type

array

Filter edges design

The -6 dB point for all filters is in the middle of the transition band.

If no transition band is given, default is to use:

- l_trans_bandwidth = .. math:: min(max(l_{freq} * 0.25, 2), l_{freq})
- h_trans_bandwidth = .. math:: min(max(h_{freq} * 0.25, 2.), rate / 2. - h_{freq})

Band-pass filter

The frequency response is (approximately) given by:

Where:

- l_trans_bandwidth = Fp1 - Fs1 in Hz
- Fh_trans_bandwidth = Fs2 - Fp2 + in Hz
- `freqs` = *[Fp1, Fs1, Fs2, Fp2]*

Band-stop filter

The frequency response is (approximately) given by:

Where:

- l_trans_bandwidth = Fs1 - Fp1 in Hz
- Fh_trans_bandwidth = Fp2 - Fs2 + in Hz
- `freqs` = *[Fp1, Fs1, Fs2, Fp2]*

Low-pass filter

The frequency response is (approximately) given by:

Where :

- h_trans_bandwidth = Fstop - Fp in Hz
- `freqs` = *[Fp, Fstop]*

High-pass filter

The frequency response is (approximately) given by:

Where :

- l_trans_bandwidth = Fp - Fstop in Hz
- `freqs` = *[Fstop, Fp]*

Notes

Adapted from mne.filters, see the documentation of:

- mne.filters

`timeflux_dsp.utils.filters.`**`construct_fir_filter`**(*rate*, *frequencies*, *gains*, *order*, *phase*, *window*, *design*)

Construct coeffs of FIR filter.

Parameters

- **`rate`** (*`float`*) – Nominal sampling rate of the input data.
- **`order`** (*`int`*) – Filter order
- **`frequencies`** (*`list`*) – Transition frequencies in Hz.
- **`design`** (*`str`* | *`'firwin2'`*) – Design of the transfert function of the filter.
- **`phase`** (*`str`* | *`linear`*) – Phase response (“zero”, “zero-double” or “minimum”).
- **`window`** (*`float`* | *`hamming`*) – The window to use in FIR design, (“hamming”, “hann”, or “blackman”).

Returns
array h. FIR coeffs.

Notes

Adapted from mne.filters, see the documentation of:

- mne.filters
- scipy.signal.firwin2

timeflux_dsp.utils.filters.**construct_iir_filter**(*rate*, *frequencies*, *filter_type*, *order=None*, *design='butter'*, *pass_loss=3.0*, *stop_atten=50.0*, *output='sos'*)

Calculate an IIR filter kernel for a given sampling rate.

Parameters

- **rate** (*float*) – Nominal sampling rate of the input data.
- **order** (*int*) – Filter order
- **frequencies** (*list*) – Transition frequencies in Hz.
- **filter_type** (*str*) – Filter mode (*lowpass*, *highpass*, *bandstop* or *bandpass*).
- **design** (*str | 'butter'*) – Design of the transfert function of the filter (*butter*, *cheby1*, *cheby2*, *ellip*, *bessel*)
- **pass_loss** (*float | `3.0`*) – For Chebyshev and elliptic filters, provides the maximum ripple in the passband. (dB).
- **stop_atten** (*float | `50.0`*) – For Chebyshev and elliptic filters, provides the minimum attenuation in the stop band. (dB)

Returns
sos. IIR coeffs.

Return type
array

Notes

Adapted from mne.filters, see the documentation of:

- mne.filters
- scipy.signal.iirfilter
- scipy.signal.iirdesign

timeflux_dsp.utils.import_helpers

import_helpers

timeflux_dsp.utils.import_helpers.**make_object**(*fullname*, *params=None*)

Parameters

- **fullname** (*str*) – full name of the object (ie. '.'.join([o.__module__, o.__name__]))
- **params** (*dict|None*) – keyword arguments to initialize the object

Returns

Object, instance of the class.

PYTHON MODULE INDEX

t

INDEX

U

W