yodel Documentation

Release 0.3.0

Romain Clement

October 28, 2014

Contents

Contents:

CHAPTER 1

yodel package

1.1 Submodules

1.1.1 yodel.analysis module

This module provides classes for audio signal analysis such as spectral analysis.

class `yodel.analysis.DFT(size)`

Bases: `builtins.object`

The Discrete Fourier Transform allows to convert a time-domain signal into a frequency-domain spectrum.

Warning: It should not be used in practice for computational reasons, and should only be used for testing purposes. Instead, prefer using the `FFT`.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

`__init__(size)`

Initialize the Discrete Fourier Transform.

Parameters **size** – length of the DFT (should only be a power of 2)

`forward(real_signal, real_spec, imag_spec)`

Compute the complex spectrum of a given real time-domain signal

Parameters

- **real_signal** – real time-domain input signal
- **real_spec** – real-part of the output complex spectrum
- **imag_spec** – imaginary-part of the output complex spectrum

`inverse(real_spec, imag_spec, real_signal)`

Compute the real time-domain signal of a given complex spectrum

Parameters

- **real_spec** – real-part of the complex spectrum
- **imag_spec** – imaginary-part of the complex spectrum
- **real_signal** – real time-domain output signal

class `yodel.analysis.FFT(size)`

Bases: `builtins.object`

The Fast Fourier Transform is a faster algorithm for performing the `DFT`. It allows converting a time-domain signal into a frequency-domain spectrum.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

`__init__`(*size*)
Initialize the Fast Fourier Transform.

Parameters **size** – length of the FFT (should only be a power of 2)

`forward`(*real_signal*, *real_spec*, *imag_spec*)
Compute the complex spectrum of a given real time-domain signal

Parameters

- **real_signal** – real time-domain input signal
- **real_spec** – real-part of the output complex spectrum
- **imag_spec** – imaginary-part of the output complex spectrum

`inverse`(*real_spec*, *imag_spec*, *real_signal*)
Compute the real time-domain signal of a given complex spectrum

Parameters

- **real_spec** – real-part of the complex spectrum
- **imag_spec** – imaginary-part of the complex spectrum
- **real_signal** – real time-domain output signal

class `yodel.analysis.`**`Window`**(*size*)
Bases: `builtins.object`

An analysis window function allows to reduce unwanted frequencies when performing spectrum analysis.

`__init__`(*size*)
Initialize the analysis window. By default, a flat window is applied. Use one of the provided methods to make it Hanning or Hamming.

Parameters **size** – length of the analysis window

`blackman`(*size*)
Make a Blackman analysis window.

Parameters **size** – length of the analysis window

`hamming`(*size*)
Make a Hamming analysis window.

Parameters **size** – length of the analysis window

`hanning`(*size*)
Make a Hanning analysis window.

Parameters **size** – length of the analysis window

`process`(*input_signal*, *output_signal*)
Perform windowing on an input signal.

Parameters

- **input_signal** – input signal to be windowed
- **output_signal** – resulting windowed signal

rectangular(*size*)
Make a rectangular (flat) window. This type of window does not have any affect when applied on an input signal.

Parameters **size** – length of the analysis window

1.1.2 yodel.complex module

This module provides utility functions for complex numbers.

`yodel.complex.`**`modulus`**(*real*, *imag*)
Compute the modulus of a complex number.

Parameters

- **real** – real part of the complex number
- **imag** – imaginary part of the complex number

Return type modulus of complex number

`yodel.complex.`**`phase`**(*real*, *imag*)
Compute the phase of a complex number.

Parameters

- **real** – real part of the complex number
- **imag** – imaginary part of the complex number

Return type phase of complex number

1.1.3 yodel.conversion module

This module provides utility functions for various math conversions.

`yodel.conversion.`**`db2lin`**(*dbval*)
Convert a decibel (dB) value to the linear scale.

Parameters **dbval** – decibel value

Return type linear value

`yodel.conversion.`**`lin2db`**(*linval*)
Convert a linear value to the decibel (dB) scale.

Parameters **linval** – linear value

Return type decibel value

1.1.4 yodel.delay module

This module provides classes for delaying signals.

class `yodel.delay.`**`DelayLine`**(*samplerate*, *maxdelay=1000*, *delay=0*)
Bases: `builtins.object`

A delayline allows to delay a given signal by a certain amount of time or samples. Time-varying delay is allowed.

`__init__`(*samplerate*, *maxdelay=1000*, *delay=0*)
Create a delayline.

Parameters

- **samplerate** – sample-rate in Hz
- **maxdelay** – maximum allowed delay in ms
- **delay** – initial delay in ms

`clear`()
Clear the current samples in the delayline with zeros. Every other state is kept (current delay, max delay).

`process`(*input_signal*, *output_signal*)
Delay an input signal by the current amount of delay.

Parameters

- **input_signal** – signal to be delayed
- **output_signal** – resulting delayed signal

`process_sample`(*input_sample*)
Delay an input sample by the current amount of delay.

Parameters **input_signal** – sample to be delayed

Returns resulting delayed sample

`set_delay`(*delay*)
Specify a new time delay value.

Parameters **delay** – new delay value in ms

1.1.5 yodel.filter module

This module provides classes for audio signal filtering.

class `yodel.filter.Biquad`
Bases: `builtins.object`

A biquad filter is a 2-poles/2-zeros filter allowing to perform various kind of filtering. Signal attenuation is at a rate of 12 dB per octave.

Reference: "Cookbook formulae for audio EQ biquad filter coefficients", Robert Bristow-Johnson (http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt)

`__init__`()
Create an inactive biquad filter with a flat frequency response. To make the filter active, use one of the provided methods: `low_pass()`, `high_pass()`, `band_pass()`, `all_pass()`, `notch()`, `peak()`, `low_shelf()`, `high_shelf()` and `custom()`.

`all_pass`(*samplerate*, *center*, *resonance*)
Make an all-pass filter.

Parameters

- **samplerate** – sample-rate in Hz
- **center** – center frequency in Hz
- **resonance** – resonance or Q-factor

band_pass (*samplerate*, *center*, *resonance*)
Make a band-pass filter.

Parameters

- **samplerate** – sample-rate in Hz
- **center** – center frequency in Hz
- **resonance** – resonance or Q-factor

custom (*a0*, *a1*, *a2*, *b0*, *b1*, *b2*)
Make a custom filter.

Parameters

- **a0** – a[0] coefficient
- **a1** – a[1] coefficient
- **a2** – a[2] coefficient
- **b0** – b[0] coefficient
- **b1** – b[1] coefficient
- **b2** – b[2] coefficient

high_pass (*samplerate*, *cutoff*, *resonance*)
Make a high-pass filter.

Parameters

- **samplerate** – sample-rate in Hz
- **cutoff** – cut-off frequency in Hz
- **resonance** – resonance or Q-factor

high_shelf (*samplerate*, *cutoff*, *resonance*, *dbgain*)
Make a high-shelf filter.

Parameters

- **samplerate** – sample-rate in Hz
- **cutoff** – cut-off frequency in Hz
- **resonance** – resonance or Q-factor
- **dbgain** – gain in dB

low_pass (*samplerate*, *cutoff*, *resonance*)
Make a low-pass filter.

Parameters

- **samplerate** – sample-rate in Hz
- **cutoff** – cut-off frequency in Hz
- **resonance** – resonance or Q-factor

low_shelf (*samplerate*, *cutoff*, *resonance*, *dbgain*)
Make a low-shelf filter.

Parameters

- **samplerate** – sample-rate in Hz

- **cutoff** – cut-off frequency in Hz
- **resonance** – resonance or Q-factor
- **dbgain** – gain in dB

notch (*samplerate*, *center*, *resonance*)
Make a notch filter.

Parameters

- **samplerate** – sample-rate in Hz
- **center** – center frequency in Hz
- **resonance** – resonance or Q-factor

peak (*samplerate*, *center*, *resonance*, *dbgain*)
Make a peak filter.

Parameters

- **samplerate** – sample-rate in Hz
- **center** – center frequency in Hz
- **resonance** – resonance or Q-factor
- **dbgain** – gain in dB

process (*x*, *y*)
Filter an input signal. Can be used for in-place filtering.

Parameters

- **x** – input buffer
- **y** – output buffer

process_sample (*x*)
Filter a single sample and return the filtered sample.

Parameters **x** – input sample

Return type filtered sample

reset ()
Make the filter inactive with a flat frequency response.

class `yodel.filter.`**Comb** (*samplerate*, *delay*, *gain*)
Bases: `builtins.object`

A comb filter combines the input signal with a delayed copy of itself. Three types are available: feedback, feedforward and allpass.

References: "Physical Audio Signal Processing", Julius O. Smith (https://ccrma.stanford.edu/~jos/pasp/Comb_Filters.html)

"Introduction to Computer Music", Nick Collins

__init__ (*samplerate*, *delay*, *gain*)
Create a comb filter. By default, it is an allpass but one can select another type with `feedback()`, `feedforward()` and `allpass()` methods.

Parameters

- **samplerate** – sample-rate in Hz
- **delay** – delay in ms

- **gain** – gain between -1 and +1

allpass (*delay*, *gain*)
Make a feedforward comb filter.

Parameters

- **delay** – delay in ms
- **gain** – gain between -1 and + 1

feedback (*delay*, *gain*)
Make a feedback comb filter.

Parameters

- **delay** – delay in ms
- **gain** – gain between -1 and + 1

feedforward (*delay*, *gain*)
Make a feedforward comb filter.

Parameters

- **delay** – delay in ms
- **gain** – gain between -1 and + 1

process (*input_signal*, *output_signal*)
Filter an input signal.

Parameters

- **input_signal** – input signal
- **output_signal** – filtered signal

process_sample (*input_sample*)
Filter a single sample.

Parameters **input_sample** – input sample

Returns filtered sample

reset ()
Clear the current comb filter state.

set_delay (*delay*)
Change the current delay of the comb filter.

Parameters **delay** – delay in ms

set_gain (*gain*)
Change the current gain of the comb filter.

Parameters **gain** – gain between -1 and + 1

class `yodel.filter.`**Convolution** (*framesize*, *impulse_response*)
Bases: `builtins.object`

The convolution filter performs FIR filtering using a provided impulse response signal.

Warning: It should not be used in practice for computational reasons, and should only be used for testing purposes. Instead, prefer using the `FastConvolution`.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

`__init__` (*framesize*, *impulse_response*)
Create a convolution filter.

Parameters

- **framesize** – framesize of input buffers to be filtered
- **impulse_response** – the impulse response signal to used

`process` (*input_signal*, *output_signal*)
Filter an input signal with the impulse response. The length of the input signal must be the one defined at filter creation.

The filtered output signal will be of the same length. The 'tail' of the convolution will be added to the beginning of the next filtered signal.

To obtain the 'tail' of the convolution without filtering another signal, simply process an input signal filled with zeros.

Parameters

- **input_signal** – input signal to be filtered
- **output_signal** – filtered signal

class `yodel.filter.`**`Custom`** (*samplerate*, *framesize*)
Bases: `builtins.object`

A custom filter allows to design precisely the frequency response of a digital filter. The filtering is then performed with a `FastConvolution` filter.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

`__init__` (*samplerate*, *framesize*)
Create a custom filter with a flat frequency response. By default, the filter has a latency of (framesize/2) samples.

Parameters

- **samplerate** – sample-rate in Hz
- **framesize** – framesize of input buffers to be filtered

`design` (*freqresponse*, *db=True*)
Create the filter impulse response from the specified frequency response.

The response must represent the desired spectrum, and of size (Nfft/2+1). The latency of the filter will be of (Nfft/2) samples.

The values of the frequency bands can either be specified in linear scale (1 being flat) or in dB scale (0 being flat).

Parameters

- **freqresponse** – desired frequency response
- **db** – True if the frequency response is specified in dB

`process` (*input_signal*, *output_signal*)
Filter an input signal with the custom impulse response. The length of the input signal must be the one defined at filter creation.

As with `Convolution`, the filtered output signal will be of the same length. The 'tail' of the convolution will be added to the beginning of the next filtered signal.

To obtain the 'tail' of the convolution without filtering another signal, simply process an input signal filled with zeros.

Parameters

- **input_signal** – input signal to be filtered
- **output_signal** – filtered signal

class `yodel.filter.`**`FastConvolution`**(*framesize*, *impulse_response*)

Bases: `builtins.object`

The fast convolution filter performs FIR filtering using a provided impulse response signal.

This filter uses a faster algorithm than standard `Convolution`, based on the `yodel.analysis.FFT`.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

`__init__`(*framesize*, *impulse_response*)

Create a fast convolution filter.

Parameters

- **framesize** – framesize of input buffers to be filtered
- **impulse_response** – the impulse response signal to used

`process`(*input_signal*, *output_signal*)

Filter an input signal with the impulse response. The length of the input signal must be the one defined at filter creation.

The filtered output signal will be of the same length. The 'tail' of the convolution will be added to the beginning of the next filtered signal.

To obtain the 'tail' of the convolution without filtering another signal, simply process an input signal filled with zeros.

Parameters

- **input_signal** – input signal to be filtered
- **output_signal** – filtered signal

class `yodel.filter.`**`ParametricEQ`**(*samplerate*, *bands*)

Bases: `builtins.object`

A parametric equalizer provides multi-band equalization of audio signals. The center frequency, the resonance and the amplification (in dB) can be controlled individually for each frequency band.

`__init__`(*samplerate*, *bands*)

Create a parametric equalizer with a given number of frequency bands.

Parameters

- **samplerate** – sample-rate in Hz
- **bands** – number of bands (at least 2)

`process`(*input_signal*, *output_signal*)

Filter an input signal. Can be used for in-place filtering.

Parameters

- **input_signal** – input buffer
- **output_signal** – filtered buffer

set_band (*band*, *center*, *resonance*, *dbgain*)
Change the parameters for the selected frequency band.

Parameters

- **band** – index of the band (from 0 to (total number of bands - 1))
- **cutoff** – cut-off frequency in Hz
- **resonance** – resonance or Q-factor
- **dbgain** – gain in dB

class yodel.filter.**SinglePole**
Bases: builtins.object

A single pole filter is used to perform low-pass and high-pass filtering. Signal attenuation is at a rate of 6 dB per octave.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

__init__ ()
Create an inactive single pole filter with a flat frequency response. To make the filter active, use one of the provided methods: low_pass() and high_pass().

high_pass (*samplerate*, *cutoff*)
Make a high-pass filter.

Parameters

- **samplerate** – sample-rate in Hz
- **cutoff** – cut-off frequency in Hz

low_pass (*samplerate*, *cutoff*)
Make a low-pass filter.

Parameters

- **samplerate** – sample-rate in Hz
- **cutoff** – cut-off frequency in Hz

process (*x*, *y*)
Filter an input signal. Can be used for in-place filtering.

Parameters

- **x** – input buffer
- **y** – output buffer

process_sample (*x*)
Filter a single sample and return the filtered sample.

Parameters **x** – input sample

Return type filtered sample

reset ()
Create an inactive single pole filter with a flat frequency response. To make the filter active, use one of the provided methods.

class yodel.filter.**StateVariable**
Bases: builtins.object

A state variable filter provides simultaneously low-pass, high-pass, band-pass and band-reject filtering. Like the `Biquad` filter, signal attenuation is at a rate of 12 dB per octave. Nevertheless, the filter becomes unstable at higher frequencies (around one sixth of the sample-rate).

__init__()
: Create an inactive state variable filter with a flat frequency response. To make the filter active, use the `set()` method.

process(*x*, *hp*, *bp*, *lp*, *br*)
: Filter an input signal. Can be used for in-place filtering.

Parameters

- **x** – input buffer
- **hp** – high-pass filtered output
- **bp** – band-pass filtered output
- **lp** – low-pass filtered output
- **br** – band-reject filtered output

process_sample(*x*)
: Filter a single sample and return the filtered samples.

Parameters **x** – input sample

Return type tuple (high-pass, band-pass, low-pass, band-reject)

reset()
: Make the filter inactive with a flat frequency response.

set(*samplerate*, *cutoff*, *resonance*)
: Specify the parameters of the filter.

Parameters

- **samplerate** – sample-rate in Hz
- **cutoff** – cut-off frequency in Hz
- **resonance** – resonance or Q-factor

class `yodel.filter.`**WindowedSinc**(*samplerate*, *framesize*)
: Bases: `builtins.object`

A windowed sinc filter allows to separate one frequency band from another, using `low_pass()`, `high_pass()`, `band_pass()` and `band_reject()` forms. Windowing is done using a Blackman `yodel.analysis.Window`. The filtering is performed with a `FastConvolution` filter.

Reference: "Digital Signal Processing, a practical guide for engineers and scientists", Steven W. Smith

__init__(*samplerate*, *framesize*)
: Create a windowed sinc filter with a flat frequency response.

Parameters

- **samplerate** – sample-rate in Hz
- **framesize** – framesize of input buffers to be filtered

band_pass(*center*, *bandwidth*)
: Make a band-pass filter with given center frequency and bandwidth. Lowering the bandwidth will increase the size of the kernel filter, thus increasing the roll-off rate but also the computation cost.

Parameters

- **center** – center frequency in Hz
- **bandwidth** – frequency band width in Hz

`band_reject` (*center*, *bandwidth*)
Make a band-reject filter with given center frequency and bandwidth. Lowering the bandwidth will increase the size of the kernel filter, thus increasing the roll-off rate but also the computation cost.

Parameters

- **center** – center frequency in Hz
- **bandwidth** – frequency band width in Hz

`high_pass` (*cutoff*, *bandwidth*)
Make a high-pass filter with given cutoff frequency and bandwidth. Lowering the bandwidth will increase the size of the kernel filter, thus increasing the roll-off rate but also the computation cost.

Parameters

- **cutoff** – cut-off frequency in Hz
- **bandwidth** – frequency band width in Hz

`low_pass` (*cutoff*, *bandwidth*)
Make a low-pass filter with given cutoff frequency and bandwidth. Lowering the bandwidth will increase the size of the kernel filter, thus increasing the roll-off rate but also the computation cost.

Parameters

- **cutoff** – cut-off frequency in Hz
- **bandwidth** – frequency band width in Hz

1.2 Module contents

Yodel (*the Swiss Army knife for your sound*) is an easy-to-use python package for digital audio signal processing, analysis and synthesis. It is meant to provide a comprehensive set of tools to manipulate audio signals. It can be used for prototyping as well as developing audio applications in Python.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

y

Symbols

A

B

C

D

F

H

I

L

M

N

P

R

S

W

Y