zounds Documentation

Release 0.46.0

John Vinyard

Mar 05, 2020

Contents

Zounds is a python library for organizing machine learning experiments dealing with sound.

CHAPTER 1

API documentation

1.1 Timeseries

The timeseries module introduces classes for dealing with time as it relates to audio signals

1.1.1 Audio Samples

class zounds.timeseries.**AudioSamples**

AudioSamples represents constant-rate samples of a continuous audio signal at common sampling rates.

It is a special case of an *ArrayWithUnits* whose first dimension is a *TimeDimension* that has a common audio sampling rate (e.g. *SR44100*).

Parameters

- **array** (*np.ndarray*) – The raw sample data
- **samplerate** (SampleRate) – The rate at which data was sampled

Raises

- ValueError – When array has a second dimension with size greater than 2
- TypeError – When samplerate is not a AudioSampleRate (e.g. *SR22050*)

Examples::

```
>>> from zounds import AudioSamples, SR44100, TimeSlice, Seconds
>>> import numpy as np
>>> raw = np.random.normal(0, 1, 44100*10)
>>> samples = AudioSamples(raw, SR44100())
>>> samples.samples_per_second
44100
>>> samples.channels
1
```

(continues on next page)

(continued from previous page)

```
>>> sliced = samples[TimeSlice(Seconds(2))]
>>> sliced.shape
(88200,)
```

classmethod from_example(*arr*, *example*)
Produce a new `ArrayWithUnits` instance given some raw data and an example instance that has the desired dimensions

mono
Return this instance summed to mono. If the instance is already mono, this is a no-op.

encode(*flo=None*, *fmt='WAV'*, *subtype='PCM_16'*)
Return audio samples encoded as bytes given a particular audio format

Parameters

- **flo** (*file-like*) – A file-like object to write the bytes to. If flo is not supplied, a new `io.BytesIO` instance will be created and returned
- **fmt** (*str*) – A libsndfile-friendly identifier for an audio encoding (detailed here: http://www.mega-nerd.com/libsndfile/api.html)
- **subtype** (*str*) – A libsndfile-friendly identifier for an audio encoding subtype (detailed here: http://www.mega-nerd.com/libsndfile/api.html)

Examples

```
>>> from zounds import SR11025, AudioSamples
>>> import numpy as np
>>> silence = np.zeros(11025*10)
>>> samples = AudioSamples(silence, SR11025())
>>> bio = samples.encode()
>>> bio.read(10)
'RIFFx]\x03\x00WA'
```

1.1.2 The Time Dimension

class zounds.timeseries.**TimeDimension**(*frequency=None*, *duration=None*, *size=None*)
When applied to an axis of *ArrayWithUnits*, that axis can be viewed as representing a constant-rate time series sampled at a given *SampleRate*.

Parameters

- **frequency** (*np.timedelta64*) – The sampling frequency for this dimension
- **duration** (*np.timedelta64*) – The sampling duration for this dimension. When not provided it defaults to the sampling frequency
- **size** (*int*) – The size/length of the dimension

Raises `ValueError` – when frequency and/or duration are not `np.timedelta64` instances

Examples

```
>>> from zounds import ArrayWithUnits, TimeDimension, Seconds, TimeSlice
>>> import numpy as np
>>> raw = np.zeros(100)
>>> timeseries = ArrayWithUnits(raw, [TimeDimension(Seconds(1))])
>>> timeseries.dimensions[0]
TimeDimension(f=1.0, d=1.0)
>>> timeseries.dimensions[0].end_seconds
100.0
>>> sliced = timeseries[TimeSlice(Seconds(50))]
>>> sliced.shape
(50,)
```

metaslice(*index*, *size*)
Produce a new instance of this dimension, given a custom slice

integer_based_slice(*ts*)
Transform a *TimeSlice* into integer indices that numpy can work with

Parameters **ts** (*slice, TimeSlice*) – the time slice to translate into integer indices

class zounds.timeseries.**TimeSlice**(*duration=None*, *start=None*)
A slice that can be applied to a *TimeDimension* to return a subset of samples.

Parameters

- **duration** (*np.timedelta64*) – The duration of the slice
- **start** (*np.timedelta64*) – A duration representing the start position of this slice, relative to zero or the beginning. If not provided, defaults to zero

Raises ValueError – when duration and/or start are not numpy.timedelta64 instances

Examples

```
>>> from zounds import ArrayWithUnits, TimeDimension, TimeSlice, Seconds
>>> import numpy as np
>>> raw = np.zeros(100)
>>> ts = ArrayWithUnits(raw, [TimeDimension(Seconds(1))])
>>> sliced = ts[TimeSlice(duration=Seconds(5), start=Seconds(50))]
>>> sliced.shape
(5,)
```

See also:

TimeDimension

1.1.3 Sample Rates

class zounds.timeseries.**SR96000**
A *SampleRate* representing the common audio sampling rate 96kHz

Examples

```
>>> from zounds import SR96000
>>> sr = SR96000()
>>> sr.samples_per_second
96000
>>> int(sr)
96000
>>> sr.nyquist
48000
```

class zounds.timeseries.**SR48000**
A *SampleRate* representing the common audio sampling rate 48kHz

Examples

```
>>> from zounds import SR48000
>>> sr = SR48000()
>>> sr.samples_per_second
48000
>>> int(sr)
48000
>>> sr.nyquist
24000
```

class zounds.timeseries.**SR44100**
A *SampleRate* representing the common audio sampling rate 44.1kHz

Examples

```
>>> from zounds import SR44100
>>> sr = SR44100()
>>> sr.samples_per_second
44100
>>> int(sr)
44100
>>> sr.nyquist
22050
```

class zounds.timeseries.**SR22050**
A *SampleRate* representing the common audio sampling rate 22.025kHz

Examples

```
>>> from zounds import SR22050
>>> sr = SR22050()
>>> sr.samples_per_second
22050
>>> int(sr)
22050
>>> sr.nyquist
11025
```

class zounds.timeseries.**SR11025**
A *SampleRate* representing the common audio sampling rate 11.025kHz

Examples

```
>>> from zounds import SR11025
>>> sr = SR11025()
>>> sr.samples_per_second
11025
>>> int(sr)
11025
>>> sr.nyquist
5512
```

class zounds.timeseries.**SampleRate**(*frequency*, *duration*)
SampleRate describes the constant frequency at which samples are taken from a continuous signal, and the duration of each sample.

Instances of this class could describe an audio sampling rate (e.g. 44.1kHz) or the strided windows often used in short-time fourier transforms

Parameters

- **frequency** (*numpy.timedelta64*) – The frequency at which the signal is sampled
- **duration** (*numpy.timedelta64*) – The duration of each sample

Raises ValueError – when frequency or duration are less than or equal to zero

Examples

```
>>> from zounds import Seconds, SampleRate
>>> sr = SampleRate(Seconds(1), Seconds(2))
>>> sr.frequency
numpy.timedelta64(1,'s')
>>> sr.duration
numpy.timedelta64(2,'s')
>>> sr.overlap
numpy.timedelta64(1,'s')
>>> sr.overlap_ratio
0.5
```

See also:

SR96000 *SR48000* *SR44100* *SR22050* *SR11025*

overlap
For sampling schemes that overlap, return a numpy.timedelta64 instance representing the duration of overlap between each sample

overlap_ratio
For sampling schemes that overlap, return the ratio of overlap to sample duration

1.1.4 Durations

Zounds includes several convenience classes that make it possible to create time durations as numpy.timedelta64 instances without remembering or using magic strings to designate units.

class zounds.timeseries.**Hours**(**args*, ***kwargs*)
Convenience class for creating a duration in hours

Parameters **hours** (*int*) – duration in hours

Examples

```
>>> from zounds import Hours
>>> hours = Hours(3)
>>> hours
numpy.timedelta(3, 'h')
```

class zounds.timeseries.**Minutes**(**args*, ***kwargs*)
Convenience class for creating a duration in minutes

Parameters **minutes** (*int*) – duration in minutes

Examples

```
>>> from zounds import Minutes
>>> minutes = Minutes(3)
>>> minutes
numpy.timedelta(3, 'm')
```

class zounds.timeseries.**Seconds**(**args*, ***kwargs*)
Convenience class for creating a duration in seconds

Parameters **seconds** (*int*) – duration in seconds

Examples

```
>>> from zounds import Seconds
>>> seconds = Seconds(3)
>>> seconds
numpy.timedelta(3, 's')
```

class zounds.timeseries.**Milliseconds**(**args*, ***kwargs*)
Convenience class for creating a duration in milliseconds

Parameters **milliseconds** (*int*) – duration in milliseconds

Examples

```
>>> from zounds import Milliseconds
>>> ms = Milliseconds(3)
>>> ms
numpy.timedelta(3, 'ms')
```

class zounds.timeseries.**Microseconds**(**args*, ***kwargs*)
Convenience class for creating a duration in microseconds

Parameters **microseconds** (*int*) – duration in microseconds

Examples

```
>>> from zounds import Microseconds
>>> us = Microseconds(3)
>>> us
numpy.timedelta(3, 'us')
```

class zounds.timeseries.**Nanoseconds**(**args*, ***kwargs*)

Convenience class for creating a duration in nanoseconds

Parameters **nanoseconds** (*int*) – duration in nanoseconds

Examples

```
>>> from zounds import Nanoseconds
>>> ns = Nanoseconds(3)
>>> ns
numpy.timedelta(3, 'ns')
```

class zounds.timeseries.**Picoseconds**(**args*, ***kwargs*)

Convenience class for creating a duration in picoseconds

Parameters **picoseconds** (*int*) – duration in picoseconds

Examples

```
>>> from zounds import Picoseconds
>>> ps = Picoseconds(3)
>>> ps
numpy.timedelta(3, 'ps')
```

1.2 Soundfile

The soundfile module introduces `featureflow.Node` subclasses that know how to process low-level audio samples and common audio encodings.

1.2.1 Input

class zounds.soundfile.**AudioMetaData**(*uri=None*, *samplerate=None*, *channels=None*, *licensing=None*, *description=None*, *tags=None*, ***kwargs*)

Encapsulates metadata about a source audio file, including things like text descriptions and licensing information.

Parameters

- **uri** (*requests.Request or str*) – uri may be either a string representing a network resource or a local file, or a `requests.Request` instance
- **samplerate** (*int*) – the samplerate of the source audio
- **channels** (*int*) – the number of channels of the source audio
- **licensing** (*str*) – The licensing agreement (if any) that applies to the source audio

- **description** (*str*) – a text description of the source audio
- **tags** (*str*) – text tags that apply to the source audio
- **kwargs** (*dict*) – other arbitrary properties about the source audio

Raises ValueError – when *uri* is not provided

See also:

zounds.datasets.FreeSoundSearch *zounds.datasets.InternetArchive* *zounds.datasets.PhatDrumLoops*

1.2.2 Chunksize

class zounds.soundfile.**ChunkSizeBytes**(*samplerate*, *duration*, *channels=2*, *bit_depth=16*)

A convenience class to help describe a chunksize in bytes for the featureflow.ByteStream in terms of audio sample batch sizes.

Parameters

- **samplerate** (SampleRate) – The samples-per-second factor
- **duration** (*numpy.timedelta64*) – The length of desired chunks in seconds
- **channels** (*int*) – Then audio channels factor
- **bit_depth** (*int*) – The bit depth factor

Examples

```
>>> from zounds import ChunkSizeBytes, Seconds, SR44100
>>> chunksize = ChunkSizeBytes(SR44100(), Seconds(30))
>>> chunksize
ChunkSizeBytes(samplerate=SR44100(f=2.2675736e-05, d=2.2675736e-05)...
>>> int(chunksize)
5292000
```

1.2.3 Processing Nodes

class zounds.soundfile.**AudioStream**(*sum_to_mono=True*, *needs=None*)

AudioStream expects to process a raw stream of bytes (e.g. one produced by featureflow.ByteStream) and produces chunks of *AudioSamples*

Parameters

- **sum_to_mono** (*bool*) – True if this node should return a *AudioSamples* instance with a single channel
- **needs** (*Feature*) – a processing node that produces a byte stream (e.g. ByteStream

Here's how'd you typically see *AudioStream* used in a processing graph.

```
import featureflow as ff
import zounds

chunksize = zounds.ChunkSizeBytes(
    samplerate=zounds.SR44100(),
```

(continues on next page)

(continued from previous page)

```
    duration=zounds.Seconds(30),
    bit_depth=16,
    channels=2)

@zounds.simple_in_memory_settings
class Document(ff.BaseModel):
    meta = ff.JSONFeature(
        zounds.MetaData,
        store=True,
        encoder=zounds.AudioMetaDataEncoder)

    raw = ff.ByteStreamFeature(
        ff.ByteStream,
        chunksize=chunksize,
        needs=meta,
        store=False)

    pcm = zounds.AudioSamplesFeature(
        zounds.AudioStream,
        needs=raw,
        store=True)


synth = zounds.NoiseSynthesizer(zounds.SR11025())
samples = synth.synthesize(zounds.Seconds(10))
raw_bytes = samples.encode()
_id = Document.process(meta=raw_bytes)
doc = Document(_id)
print doc.pcm.__class__  # returns an AudioSamples instance
```

class zounds.soundfile.**Resampler**(*samplerate=None*, *needs=None*)

Resampler expects to process *AudioSamples* instances (e.g., those produced by a *AudioStream* node), and will produce a new stream of AudioSamples at a new sampling rate.

Parameters

- **samplerate** (*AudioSampleRate*) – the desired sampling rate. If none is provided, the default is *SR44100*
- **needs** (*Feature*) – a processing node that produces *AudioSamples*

Here's how you'd typically see *Resampler* used in a processing graph.

```
import featureflow as ff
import zounds

chunksize = zounds.ChunkSizeBytes(
    samplerate=zounds.SR44100(),
    duration=zounds.Seconds(30),
    bit_depth=16,
    channels=2)

@zounds.simple_in_memory_settings
class Document(ff.BaseModel):
    meta = ff.JSONFeature(
        zounds.MetaData,
        store=True,
        encoder=zounds.AudioMetaDataEncoder)
```

(continues on next page)

(continued from previous page)

```
    raw = ff.ByteStreamFeature(
        ff.ByteStream,
        chunksize=chunksize,
        needs=meta,
        store=False)

    pcm = zounds.AudioSamplesFeature(
        zounds.AudioStream,
        needs=raw,
        store=True)

    resampled = zounds.AudioSamplesFeature(
        zounds.Resampler,
        samplerate=zounds.SR22050(),
        needs=pcm,
        store=True)


synth = zounds.NoiseSynthesizer(zounds.SR11025())
samples = synth.synthesize(zounds.Seconds(10))
raw_bytes = samples.encode()
_id = Document.process(meta=raw_bytes)
doc = Document(_id)
print doc.pcm.samplerate.__class__.__name__  # SR11025
print doc.resampled.samplerate.__class__.__name__  # SR22050
```

class zounds.soundfile.**OggVorbis**(*needs=None*)

OggVorbis expects to process a stream of raw bytes (e.g. one produced by `featureflow.ByteStream`) and produces a new byte stream where the original audio samples are ogg-vorbis encoded

Parameters **needs** (*Feature*) – a feature that produces a byte stream (e.g. `featureflow.Bytestream`)

Here's how you'd typically see *OggVorbis* used in a processing graph.

```
import featureflow as ff
import zounds


chunksize = zounds.ChunkSizeBytes(
    samplerate=zounds.SR44100(),
    duration=zounds.Seconds(30),
    bit_depth=16,
    channels=2)

@zounds.simple_in_memory_settings
class Document(ff.BaseModel):
    meta = ff.JSONFeature(
        zounds.MetaData,
        store=True,
        encoder=zounds.AudioMetaDataEncoder)

    raw = ff.ByteStreamFeature(
        ff.ByteStream,
        chunksize=chunksize,
        needs=meta,
```

(continues on next page)

(continued from previous page)

```
        store=False)

    ogg = zounds.OggVorbisFeature(
        zounds.OggVorbis,
        needs=raw,
        store=True)


synth = zounds.NoiseSynthesizer(zounds.SR11025())
samples = synth.synthesize(zounds.Seconds(10))
raw_bytes = samples.encode()
_id = Document.process(meta=raw_bytes)
doc = Document(_id)
# fetch and decode a section of audio
ts = zounds.TimeSlice(zounds.Seconds(2))
print doc.ogg[ts].shape  # 22050
```

1.3 Spectral

The spectral module contains classes that aid in dealing with frequency-domain representations of sound

1.3.1 Representations

class zounds.spectral.**FrequencyDimension**(*scale*)

When applied to an axis of *ArrayWithUnits*, that axis can be viewed as representing the energy present in a series of frequency bands

Parameters **scale** (FrequencyScale) – A scale whose frequency bands correspond to the items along the frequency axis

Examples

```
>>> from zounds import LinearScale, FrequencyBand, ArrayWithUnits
>>> from zounds import FrequencyDimension
>>> import numpy as np
>>> band = FrequencyBand(20, 20000)
>>> scale = LinearScale(frequency_band=band, n_bands=100)
>>> raw = np.hanning(100)
>>> arr = ArrayWithUnits(raw, [FrequencyDimension(scale)])
>>> sliced = arr[FrequencyBand(100, 1000)]
>>> sliced.shape
(5,)
>>> sliced.dimensions
(FrequencyDimension(scale=LinearScale(band=FrequencyBand(
start_hz=20.0,
stop_hz=1019.0,
center=519.5,
bandwidth=999.0), n_bands=5)),)
```

metaslice(*index*, *size*)

Produce a new instance of this dimension, given a custom slice

integer_based_slice(*index*)
Subclasses define behavior that transforms a custom, user-defined slice into integer indices that numpy can understand

Parameters **index** (*custom slice*) – A user-defined slice instance

validate(*size*)
Ensure that the size of the dimension matches the number of bands in the scale

Raises ValueError – when the dimension size and number of bands don't match

class zounds.spectral.**ExplicitFrequencyDimension**(*scale*, *slices*)
A frequency dimension where the mapping from frequency bands to integer indices is provided explicitly, rather than computed

Parameters

- **scale** (ExplicitScale) – the explicit frequency scale that defines how slices are extracted from this dimension
- **slices** (*iterable of slices*) – An iterable of python.slice instances which correspond to each frequency band from scale

Raises ValueError – when the number of slices and number of bands in scale don't match

metaslice(*index*, *size*)
Produce a new instance of this dimension, given a custom slice

integer_based_slice(*index*)
Subclasses define behavior that transforms a custom, user-defined slice into integer indices that numpy can understand

Parameters **index** (*custom slice*) – A user-defined slice instance

validate(*size*)
Subclasses check to ensure that the dimensions size does not validate any assumptions made by this instance

class zounds.spectral.**FrequencyAdaptive**
TODO: This needs some love. Mutually exclusive constructor arguments are no bueno

Parameters

- **arrs** – TODO
- **time_dimension** (TimeDimension) – the time dimension of the first axis of this array
- **scale** (FrequencyScale) – The frequency scale corresponding to the first axis of this array, mutually exclusive with the explicit_freq_dimension argument
- **explicit_freq_dimension** (ExplicitFrequencyDimension) – TODO

See also:

FrequencyAdaptiveTransform

square(*n_coeffs*, *do_overlap_add=False*)
Compute a "square" view of the frequency adaptive transform, by resampling each frequency band such that they all contain the same number of samples, and performing an overlap-add procedure in the case where the sample frequency and duration differ :param n_coeffs: The common size to which each frequency band should be resampled

1.3.2 Functions

zounds.spectral.**fft**(*x*, *axis=-1*, *padding_samples=0*)

Apply an FFT along the given dimension, and with the specified amount of zero-padding

Parameters

- **x** (ArrayWithUnits) – an *ArrayWithUnits* instance which has one or more *TimeDimension* axes
- **axis** (*int*) – The axis along which the fft should be applied
- **padding_samples** (*int*) – The number of padding zeros to apply along axis before performing the FFT

zounds.spectral.**morlet_filter_bank**(*samplerate*, *kernel_size*, *scale*, *scaling_factor*, *normalize=True*)

Create a *ArrayWithUnits* instance with a *TimeDimension* and a *FrequencyDimension* representing a bank of morlet wavelets centered on the sub-bands of the scale.

Parameters

- **samplerate** (SampleRate) – the samplerate of the input signal
- **kernel_size** (*int*) – the length in samples of each filter
- **scale** (FrequencyScale) – a scale whose center frequencies determine the fundamental frequency of each filer
- **scaling_factor** (*int or list of int*) – Scaling factors for each band, which determine the time-frequency resolution tradeoff. The number(s) should fall between 0 and 1, with smaller numbers achieving better frequency resolution, and larget numbers better time resolution
- **normalize** (*bool*) – When true, ensure that each filter in the bank has unit norm

See also:

FrequencyScale *SampleRate*

1.3.3 Processing Nodes

class zounds.spectral.**SlidingWindow**(*wscheme*, *wfunc=None*, *padwith=0*, *needs=None*)

SlidingWindow is a processing node that provides a very common precursor to many frequency domain transforms: a lapped and windowed view of the time- domain signal.

Parameters

- **wscheme** (SampleRate) – a sample rate that describes the frequency and duration af the sliding window
- **wfunc** (WindowingFunc) – a windowing function to apply to each frame
- **needs** (*Node*) – A processing node on which this node relies for its data. This will generally be a time-domain signal

Here's how you'd typically see *SlidingWindow* used in a processing graph

```
import zounds

Resampled = zounds.resampled(resample_to=zounds.SR11025())
```

(continues on next page)

(continued from previous page)

```
@zounds.simple_in_memory_settings
class Sound(Resampled):
    windowed = zounds.ArrayWithUnitsFeature(
        zounds.SlidingWindow,
        needs=Resampled.resampled,
        wscheme=zounds.SampleRate(
            frequency=zounds.Milliseconds(250),
            duration=zounds.Milliseconds(500)),
        wfunc=zounds.OggVorbisWindowingFunc(),
        store=True)


synth = zounds.SineSynthesizer(zounds.SR44100())
samples = synth.synthesize(zounds.Seconds(5), [220., 440., 880.])

# process the audio, and fetch features from our in-memory store
_id = Sound.process(meta=samples.encode())
sound = Sound(_id)

print sound.windowed.dimensions[0]
# TimeDimension(f=0.250068024879, d=0.500045346811)
print sound.windowed.dimensions[1]
# TimeDimension(f=9.0702947e-05, d=9.0702947e-05)
```

See also:

WindowingFunc *SampleRate*

class zounds.spectral.**FrequencyWeighting**(*weighting=None*, *needs=None*)

FrequencyWeighting is a processing node that expects to be passed an *ArrayWithUnits* instance whose last dimension is a *FrequencyDimension*

Parameters

- **weighting** (FrequencyWeighting) – the frequency weighting to apply
- **needs** (*Node*) – a processing node on which this node depends whose last dimension is a *FrequencyDimension*

class zounds.spectral.**FFT**(*needs=None*, *axis=-1*, *padding_samples=0*)

A processing node that performs an FFT of a real-valued signal

Parameters

- **axis** (*int*) – The axis over which the FFT should be computed
- **padding_samples** (*int*) – number of zero samples to pad each window with before applying the FFT
- **needs** (*Node*) – a processing node on which this one depends

See also:

FFTSynthesizer

class zounds.spectral.**DCT**(*axis=-1*, *scale_always_even=False*, *needs=None*)

A processing node that performs a Type II Discrete Cosine Transform (https://en.wikipedia.org/wiki/Discrete_cosine_transform#DCT-II) of the input

Parameters

- **axis** (*int*) – The axis over which to perform the DCT transform

- **needs** (*Node*) – a processing node on which this one depends

See also:

`DctSynthesizer`

class zounds.spectral.**DCTIV**(*scale_always_even=False*, *needs=None*)

A processing node that performs a Type IV Discrete Cosine Transform (https://en.wikipedia.org/wiki/Discrete_cosine_transform#DCT-IV) of the input

Parameters **needs** (*Node*) – a processing node on which this one depends

See also:

DCTIVSynthesizer

class zounds.spectral.**MDCT**(*needs=None*)

A processing node that performs a modified discrete cosine transform (https://en.wikipedia.org/wiki/Modified_discrete_cosine_transform) of the input.

This is really just a lapped version of the DCT-IV transform

Parameters **needs** (*Node*) – a processing node on which this one depends

See also:

MDCTSynthesizer

class zounds.spectral.**FrequencyAdaptiveTransform**(*transform=None*, *scale=None*, *window_func=None*, *check_scale_overlap_ratio=False*, *needs=None*)

A processing node that expects to receive the input from a frequency domain transformation (e.g. *FFT*), and produces a *FrequencyAdaptive* instance where time resolution can vary by frequency. This is similar to, but not precisely the same as ideas introduced in:

- A quasi-orthogonal, invertible, and perceptually relevant time-frequency transform for audio coding
- A FRAMEWORK FOR INVERTIBLE, REAL-TIME CONSTANT-Q TRANSFORMS

Parameters

- **transform** (*function*) – the transform to be applied to each frequency band
- **scale** (*FrequencyScale*) – the scale used to take frequency band slices
- **window_func** (*numpy.ndarray*) – the windowing function to apply each band before the transform is applied
- **check_scale_overlap_ratio** (*bool*) – If this feature is to be used for resynthesis later, ensure that each frequency band overlaps with the previous one by at least half, to ensure artifact-free synthesis

See also:

FrequencyAdaptive *FrequencyAdaptiveDCTSynthesizer* *FrequencyAdaptiveFFTSynthesizer*

class zounds.spectral.**Chroma**(*frequency_band*, *window=<zounds.spectral.sliding_window.HanningWindowingFunc object>*, *needs=None*)

class zounds.spectral.**BarkBands**(*frequency_band*, *n_bands=100*, *window=<zounds.spectral.sliding_window.HanningWindowingFunc object>*, *needs=None*)

class zounds.spectral.**SpectralCentroid**(*needs=None*)
Indicates where the "center of mass" of the spectrum is. Perceptually, it has a robust connection with the impression of "brightness" of a sound. It is calculated as the weighted mean of the frequencies present in the signal, determined using a Fourier transform, with their magnitudes as the weights...

– http://en.wikipedia.org/wiki/Spectral_centroid

class zounds.spectral.**SpectralFlatness**(*needs=None*)
Spectral flatness or tonality coefficient, also known as Wiener entropy, is a measure used in digital signal processing to characterize an audio spectrum. Spectral flatness is typically measured in decibels, and provides a way to quantify how tone-like a sound is, as opposed to being noise-like. The meaning of tonal in this context is in the sense of the amount of peaks or resonant structure in a power spectrum, as opposed to flat spectrum of a white noise. A high spectral flatness indicates that the spectrum has a similar amount of power in all spectral bands - this would sound similar to white noise, and the graph of the spectrum would appear relatively flat and smooth. A low spectral flatness indicates that the spectral power is concentrated in a relatively small number of bands - this would typically sound like a mixture of sine waves, and the spectrum would appear "spiky"...

– http://en.wikipedia.org/wiki/Spectral_flatness

class zounds.spectral.**BFCC**(*needs=None*, *n_coeffs=13*, *exclude=1*)
Bark frequency cepstral coefficients

1.3.4 Windowing Functions

class zounds.spectral.**WindowingFunc**(*windowing_func=None*)
WindowingFunc is mostly a convenient wrapper around numpy's handy windowing functions, or any function that takes a size parameter and returns a numpy array-like object.

A *WindowingFunc* instance can be multiplied with a nother array of any size.

Parameters **windowing_func** (*function*) – A function that takes a size parameter, and returns a numpy array-like object

Examples

```
>>> from zounds import WindowingFunc
>>> import numpy as np
>>> wf = WindowingFunc(lambda size: np.hanning(size))
>>> np.ones(5) * wf
array([ 0. ,  0.5,  1. ,  0.5,  0. ])
>>> np.ones(10) * wf
array([ 0.        ,  0.11697778,  0.41317591,  0.75      ,  0.96984631,
        0.96984631,  0.75      ,  0.41317591,  0.11697778,  0.        ])
```

See also:

IdentityWindowingFunc *OggVorbisWindowingFunc* *HanningWindowingFunc*

class zounds.spectral.**IdentityWindowingFunc**
An identity windowing function

class zounds.spectral.**OggVorbisWindowingFunc**
The windowing function described in the ogg vorbis specification

class zounds.spectral.**HanningWindowingFunc**
A hanning window function

1.3.5 Scales

class zounds.spectral.**LinearScale**(*frequency_band*, *n_bands*, *always_even=False*)

A linear frequency scale with constant bandwidth. Appropriate for use with transforms whose coefficients also lie on a linear frequency scale, e.g. the FFT or DCT transforms.

Parameters

- **frequency_band** (*FrequencyBand*) – A band representing the entire span of this scale. E.g., one might want to generate a scale spanning the entire range of human hearing by starting with `FrequencyBand(20, 20000)`
- **n_bands** (*int*) – The number of bands in this scale
- **always_even** (*bool*) – when converting frequency slices to integer indices that numpy can understand, should the slice size always be even?

Examples

```
>>> from zounds import FrequencyBand, LinearScale
>>> scale = LinearScale(FrequencyBand(20, 20000), 10)
>>> scale
LinearScale(band=FrequencyBand(
start_hz=20,
stop_hz=20000,
center=10010.0,
bandwidth=19980), n_bands=10)
>>> scale.Q
array([ 0.51001001,  1.51001001,  2.51001001,  3.51001001,  4.51001001,
        5.51001001,  6.51001001,  7.51001001,  8.51001001,  9.51001001])
```

static from_sample_rate(*sample_rate*, *n_bands*, *always_even=False*)

Return a *LinearScale* instance whose upper frequency bound is informed by the nyquist frequency of the sample rate.

Parameters

- **sample_rate** (*SamplingRate*) – the sample rate whose nyquist frequency will serve as the upper frequency bound of this scale
- **n_bands** (*int*) – the number of evenly-spaced frequency bands

class zounds.spectral.**GeometricScale**(*start_center_hz*, *stop_center_hz*, *bandwidth_ratio*, *n_bands*, *always_even=False*)

A constant-Q scale whose center frequencies progress geometrically rather than linearly

Parameters

- **start_center_hz** (*int*) – the center frequency of the first band in the scale
- **stop_center_hz** (*int*) – the center frequency of the last band in the scale
- **bandwidth_ratio** (*float*) – the center frequency to bandwidth ratio
- **n_bands** (*int*) – the total number of bands

Examples

```
>>> from zounds import GeometricScale
>>> scale = GeometricScale(20, 20000, 0.05, 10)
>>> scale
GeometricScale(band=FrequencyBand(
start_hz=19.5,
stop_hz=20500.0,
center=10259.75,
bandwidth=20480.5), n_bands=10)
>>> scale.Q
array([ 20.,  20.,  20.,  20.,  20.,  20.,  20.,  20.,  20.,  20.])
>>> list(scale.center_frequencies)
[20.000000000000004, 43.088693800637671, 92.831776672255558,
    200.00000000000003, 430.88693800637651, 928.31776672255558,
    2000.0000000000005, 4308.8693800637648, 9283.1776672255564,
    20000.000000000004]
```

class zounds.spectral.**ExplicitScale**(*bands*)

A scale where the frequency bands are provided explicitly, rather than computed

Parameters **bands** (*list of FrequencyBand*) – The explicit bands used by this scale

See also:

FrequencyAdaptive

class zounds.spectral.**FrequencyScale**(*frequency_band*, *n_bands*, *always_even=False*)

Represents a set of frequency bands with monotonically increasing start frequencies

Parameters

- **frequency_band** (FrequencyBand) – A band representing the entire span of this scale. E.g., one might want to generate a scale spanning the entire range of human hearing by starting with `FrequencyBand(20, 20000)`
- **n_bands** (*int*) – The number of bands in this scale
- **always_even** (*bool*) – when converting frequency slices to integer indices that numpy can understand, should the slice size always be even?

See also:

LinearScale *GeometricScale*

bands

An iterable of all bands in this scale

center_frequencies

An iterable of the center frequencies of each band in this scale

bandwidths

An iterable of the bandwidths of each band in this scale

ensure_overlap_ratio(*required_ratio=0.5*)

Ensure that every adjacent pair of frequency bands meets the overlap ratio criteria. This can be helpful in scenarios where a scale is being used in an invertible transform, and something like the constant overlap add constraint must be met in order to not introduce artifacts in the reconstruction.

Parameters **required_ratio** (*float*) – The required overlap ratio between all adjacent frequency band pairs

Raises AssertionError – when the overlap ratio for one or more adjacent frequency band pairs is not met

Q
The quality factor of the scale, or, the ratio of center frequencies to bandwidths

start_hz
The lower bound of this frequency scale

stop_hz
The upper bound of this frequency scale

get_slice(*frequency_band*)
Given a frequency band, and a frequency dimension comprised of n_samples, return a slice using integer indices that may be used to extract only the frequency samples that intersect with the frequency band

class zounds.spectral.**FrequencyBand**(*start_hz*, *stop_hz*)
Represents an interval, or band of frequencies in hertz (cycles per second)

Parameters

- **start_hz** (*float*) – The lower bound of the frequency band in hertz
- **stop_hz** (*float*) – The upper bound of the frequency band in hertz

Examples::

```
>>> import zounds
>>> band = zounds.FrequencyBand(500, 1000)
>>> band.center_frequency
750.0
>>> band.bandwidth
500
```

intersect(*other*)
Return the intersection between this frequency band and another.

Parameters **other** (FrequencyBand) – the instance to intersect with

Examples::

```
>>> import zounds
>>> b1 = zounds.FrequencyBand(500, 1000)
>>> b2 = zounds.FrequencyBand(900, 2000)
>>> intersection = b1.intersect(b2)
>>> intersection.start_hz, intersection.stop_hz
(900, 1000)
```

static from_start(*start_hz*, *bandwidth_hz*)
Produce a *FrequencyBand* instance from a lower bound and bandwidth

Parameters

- **start_hz** (*float*) – the lower bound of the desired FrequencyBand
- **bandwidth_hz** (*float*) – the bandwidth of the desired FrequencyBand

bandwidth
The span of this frequency band, in hertz

1.3.6 Frequency Weightings

class zounds.spectral.**AWeighting**

An A-weighting (https://en.wikipedia.org/wiki/A-weighting) that can be applied to a frequency axis via multiplication.

Examples

```
>>> from zounds import ArrayWithUnits, GeometricScale
>>> from zounds import FrequencyDimension, AWeighting
>>> import numpy as np
>>> scale = GeometricScale(20, 20000, 0.05, 10)
>>> raw = np.ones(len(scale))
>>> arr = ArrayWithUnits(raw, [FrequencyDimension(scale)])
>>> arr * AWeighting()
ArrayWithUnits([  1.        ,  18.3172567 ,  31.19918106,  40.54760374,
        47.15389876,  51.1554151 ,  52.59655479,  52.24516649,
        49.39906912,  42.05409205])
```

1.4 Core

The core module introduces the key building blocks of the representations zounds deals in: *ArrayWithUnits*, a numpy.ndarray-derived class that supports semantically meaningful indexing, and *Dimension*, a common base class for custom, user-defined dimensions.

1.4.1 Numpy Arrays with Semantically Meaningful Indexing

class zounds.core.**ArrayWithUnits**

ArrayWithUnits is an numpy.ndarray subclass that allows for indexing by more semantically meaningful slices.

It supports most methods on numpy.ndarray, and makes a best-effort to maintain meaningful dimensions throughout those operations.

Parameters

- **arr** (*ndarray*) – The numpy.ndarray instance containing the raw data for this instance
- **dimensions** (*list or tuple*) – list or tuple of *Dimension*-derived classes

Raises ValueError – when *arr.ndim* and *len(dimensions)* do not match

Examples

```
>>> from zounds import ArrayWithUnits, TimeDimension, Seconds, TimeSlice
>>> import numpy as np
>>> data = np.zeros(100)
>>> awu = ArrayWithUnits(data, [TimeDimension(Seconds(1))])
>>> sliced = awu[TimeSlice(Seconds(10))]
>>> sliced.shape
(10,)
```

See also:

IdentityDimension TimeDimension FrequencyDimension

classmethod from_example(*data*, *example*)
Produce a new *ArrayWithUnits* instance given some raw data and an example instance that has the desired dimensions

1.4.2 Custom Dimensions

class zounds.core.**Dimension**
Common base class representing one dimension of a numpy array. Sub-classes can define behavior making custom slices (e.g., time spans or frequency bands) possible.

Implementors are primarily responsible for determining how custom slices are transformed into integer indexes and slices that numpy can use directly.

See also:

IdentityDimension TimeDimension FrequencyDimension

metaslice(*index*, *size*)
Produce a new instance of this dimension, given a custom slice

integer_based_slice(*index*)
Subclasses define behavior that transforms a custom, user-defined slice into integer indices that numpy can understand

Parameters **index** (*custom slice*) – A user-defined slice instance

validate(*size*)
Subclasses check to ensure that the dimensions size does not validate any assumptions made by this instance

class zounds.core.**IdentityDimension**
A custom dimension that does not transform indices in any way, simply acting as a pass-through.

Examples

```
>>> from zounds import ArrayWithUnits, IdentityDimension
>>> import numpy as np
>>> data = np.zeros(100)
>>> arr = ArrayWithUnits(data, [IdentityDimension()])
>>> sliced = arr[4:6]
>>> sliced.shape
(2,)
```

integer_based_slice(*index*)
Subclasses define behavior that transforms a custom, user-defined slice into integer indices that numpy can understand

Parameters **index** (*custom slice*) – A user-defined slice instance

1.5 Synthesize

The *synthesize* module includes classes that can produce audio. Some, like SineSynthesize can produce simple signals from scratch that are often useful for test-cases, while others are able to invert common frequency-domain

transforms, like the *MDCTSynthesizer*

1.5.1 Short-Time Transform Synthesizers

class zounds.synthesize.**FFTSynthesizer**

Inverts the short-time fourier transform, e.g. the output of the *FFT* processing node.

Here's an example that extracts a short-time fourier transform, and then inverts it.

```
import zounds

STFT = zounds.stft(
    resample_to=zounds.SR11025(),
    store_fft=True)


@zounds.simple_in_memory_settings
class Sound(STFT):
    pass

# produce some additive sine waves
sine_synth = zounds.SineSynthesizer(zounds.SR22050())
samples = sine_synth.synthesize(
    zounds.Seconds(4), freqs_in_hz=[220, 400, 880])

# process the sound, including a short-time fourier transform feature
_id = Sound.process(meta=samples.encode())
snd = Sound(_id)

# invert the frequency-domain feature to reover the original audio
fft_synth = zounds.FFTSynthesizer()
recon = fft_synth.synthesize(snd.fft)
print recon.__class__  #  AudioSamples instance with reconstructed audio
```

See also:

FFT

class zounds.synthesize.**DCTSynthesizer**(*windowing_func=<zounds.spectral.sliding_window.IdentityWindowingFunc object>*)

Inverts the short-time discrete cosine transform (type II), e.g., the output of the *DCT* processing node

Here's an example that extracts a short-time discrete cosine transform, and then inverts it.

```
import zounds

Resampled = zounds.resampled(resample_to=zounds.SR11025())


@zounds.simple_in_memory_settings
class Sound(Resampled):
    windowed = zounds.ArrayWithUnitsFeature(
        zounds.SlidingWindow,
        needs=Resampled.resampled,
        wscheme=zounds.HalfLapped(),
        wfunc=zounds.OggVorbisWindowingFunc(),
        store=False)
```

(continues on next page)

(continued from previous page)

```
    dct = zounds.ArrayWithUnitsFeature(
        zounds.DCT,
        needs=windowed,
        store=True)

# produce some additive sine waves
sine_synth = zounds.SineSynthesizer(zounds.SR22050())
samples = sine_synth.synthesize(
    zounds.Seconds(4), freqs_in_hz=[220, 400, 880])

# process the sound, including a short-time fourier transform feature
_id = Sound.process(meta=samples.encode())
snd = Sound(_id)

# invert the frequency-domain feature to reover the original audio
dct_synth = zounds.DCTSynthesizer()
recon = dct_synth.synthesize(snd.dct)
print recon.__class__  # AudioSamples instance with reconstructed audio
```

See also:

DCT

class zounds.synthesize.**DCTIVSynthesizer**(*windowing_func=<zounds.spectral.sliding_window.IdentityWindowingFu object>*)

Inverts the short-time discrete cosine transform (type IV), e.g., the output of the *DCTIV* processing node.

Here's an example that extracts a short-time DCT-IV transform, and inverts it.

```
import zounds

Resampled = zounds.resampled(resample_to=zounds.SR11025())


@zounds.simple_in_memory_settings
class Sound(Resampled):
    windowed = zounds.ArrayWithUnitsFeature(
        zounds.SlidingWindow,
        needs=Resampled.resampled,
        wscheme=zounds.HalfLapped(),
        wfunc=zounds.OggVorbisWindowingFunc(),
        store=False)

    dct = zounds.ArrayWithUnitsFeature(
        zounds.DCTIV,
        needs=windowed,
        store=True)

# produce some additive sine waves
sine_synth = zounds.SineSynthesizer(zounds.SR22050())
samples = sine_synth.synthesize(
    zounds.Seconds(4), freqs_in_hz=[220, 400, 880])

# process the sound, including a short-time fourier transform feature
_id = Sound.process(meta=samples.encode())
snd = Sound(_id)

# invert the frequency-domain feature to reover the original audio
```

(continues on next page)

(continued from previous page)

```
dct_synth = zounds.DCTIVSynthesizer()
recon = dct_synth.synthesize(snd.dct)
print recon.__class__  # AudioSamples instance with reconstructed audio
```

See also:

DCTIV

class zounds.synthesize.**MDCTSynthesizer**

Inverts the modified discrete cosine transform, e.g., the output of the *MDCT* processing node.

Here's an example that extracts a short-time MDCT transform, and inverts it.

```
import zounds

Resampled = zounds.resampled(resample_to=zounds.SR11025())


@zounds.simple_in_memory_settings
class Sound(Resampled):
    windowed = zounds.ArrayWithUnitsFeature(
        zounds.SlidingWindow,
        needs=Resampled.resampled,
        wscheme=zounds.HalfLapped(),
        wfunc=zounds.OggVorbisWindowingFunc(),
        store=False)

    mdct = zounds.ArrayWithUnitsFeature(
        zounds.MDCT,
        needs=windowed,
        store=True)

# produce some additive sine waves
sine_synth = zounds.SineSynthesizer(zounds.SR22050())
samples = sine_synth.synthesize(
    zounds.Seconds(4), freqs_in_hz=[220, 400, 880])

# process the sound, including a short-time fourier transform feature
_id = Sound.process(meta=samples.encode())
snd = Sound(_id)

# invert the frequency-domain feature to reover the original audio
mdct_synth = zounds.MDCTSynthesizer()
recon = mdct_synth.synthesize(snd.mdct)
print recon.__class__  # AudioSamples instance with reconstructed audio
```

See also:

MDCT

1.5.2 Frequency-Adaptive Transform Synthesizers

class zounds.synthesize.**FrequencyAdaptiveDCTSynthesizer**(*scale*, *samplerate*)

Invert a frequency-adaptive transform, e.g., one produced by the *zounds.spectral.FrequencyAdaptiveTransform* processing node which has used a discrete cosine transform in its *transform* parameter.

Parameters

- **scale** (FrequencyScale) – The scale used to produce the frequency-adaptive transform
- **samplerate** (SampleRate) – The audio samplerate of the audio that was originally transformed

Here's an example of how you might first extract a frequency-adaptive representation, and then invert it:

```
import zounds
import scipy
import numpy as np

samplerate = zounds.SR11025()
Resampled = zounds.resampled(resample_to=samplerate)

scale = zounds.GeometricScale(
    100, 5000, bandwidth_ratio=0.089, n_bands=100)
scale.ensure_overlap_ratio(0.5)


@zounds.simple_in_memory_settings
class Sound(Resampled):
    long_windowed = zounds.ArrayWithUnitsFeature(
        zounds.SlidingWindow,
        wscheme=zounds.SampleRate(
            frequency=zounds.Milliseconds(500),
            duration=zounds.Seconds(1)),
        wfunc=zounds.OggVorbisWindowingFunc(),
        needs=Resampled.resampled)

    dct = zounds.ArrayWithUnitsFeature(
        zounds.DCT,
        scale_always_even=True,
        needs=long_windowed)

    freq_adaptive = zounds.FrequencyAdaptiveFeature(
        zounds.FrequencyAdaptiveTransform,
        transform=scipy.fftpack.idct,
        window_func=np.hanning,
        scale=scale,
        needs=dct,
        store=True)


# produce some additive sine waves
sine_synth = zounds.SineSynthesizer(zounds.SR22050())
samples = sine_synth.synthesize(
    zounds.Seconds(10), freqs_in_hz=[220, 440, 880])

# process the sound, including a short-time fourier transform feature
_id = Sound.process(meta=samples.encode())
snd = Sound(_id)

# invert the sound
synth = zounds.FrequencyAdaptiveDCTSynthesizer(scale, samplerate)
recon = synth.synthesize(snd.freq_adaptive)
print recon  # AudioSamples instance with the reconstructed sound
```

See also:

DCT FrequencyAdaptive FrequencyAdaptiveTransform

class zounds.synthesize.**FrequencyAdaptiveFFTSynthesizer**(*scale*, *samplerate*)

Invert a frequency-adaptive transform, e.g., one produced by the *zounds.spectral.FrequencyAdaptiveTransform* processing node which has used a fast fouriter transform in its *transform* parameter.

Parameters

- **scale** (FrequencyScale) – The scale used to produce the frequency-adaptive transform
- **samplerate** (SampleRate) – The audio samplerate of the audio that was originally transformed

Here's an example of how you might first extract a frequency-adaptive representation, and then invert it:

```
import zounds
import numpy as np

samplerate = zounds.SR11025()
Resampled = zounds.resampled(resample_to=samplerate)

scale = zounds.GeometricScale(100, 5000, bandwidth_ratio=0.089, n_bands=100)
scale.ensure_overlap_ratio(0.5)


@zounds.simple_in_memory_settings
class Sound(Resampled):
    long_windowed = zounds.ArrayWithUnitsFeature(
        zounds.SlidingWindow,
        wscheme=zounds.SampleRate(
            frequency=zounds.Milliseconds(500),
            duration=zounds.Seconds(1)),
        wfunc=zounds.OggVorbisWindowingFunc(),
        needs=Resampled.resampled)

    fft = zounds.ArrayWithUnitsFeature(
        zounds.FFT,
        needs=long_windowed)

    freq_adaptive = zounds.FrequencyAdaptiveFeature(
        zounds.FrequencyAdaptiveTransform,
        transform=np.fft.irfft,
        window_func=np.hanning,
        scale=scale,
        needs=fft,
        store=True)


# produce some additive sine waves
sine_synth = zounds.SineSynthesizer(zounds.SR22050())
samples = sine_synth.synthesize(
    zounds.Seconds(10), freqs_in_hz=[220, 440, 880])

# process the sound, including a short-time fourier transform feature
_id = Sound.process(meta=samples.encode())
snd = Sound(_id)
```

(continues on next page)

(continued from previous page)

```
# invert the sound
synth = zounds.FrequencyAdaptiveFFTSynthesizer(scale, samplerate)
recon = synth.synthesize(snd.freq_adaptive)
print recon  # AudioSamples instance with the reconstructed sound
```

See also:

FFT FrequencyAdaptive FrequencyAdaptiveTransform

1.5.3 Simple Signals

class zounds.synthesize.**SineSynthesizer**(*samplerate*)

Synthesize sine waves

Parameters **samplerate** (*Samplerate*) – the samplerate at which the sine waves should be synthesized

Examples

```
>>> import zounds
>>> synth = zounds.SineSynthesizer(zounds.SR22050())
>>> samples = synth.synthesize(             zounds.Seconds(1), freqs_in_hz=[220.,␣
↪440.])
>>> samples
AudioSamples([ 0.        ,  0.09384942,  0.18659419, ..., -0.27714552,
       -0.18659419, -0.09384942])
>>> len(samples)
22050
```

See also:

TickSynthesizer NoiseSynthesizer SilenceSynthesizer

synthesize(*duration, freqs_in_hz=[440.0]*)

Synthesize one or more sine waves

Parameters

- **duration** (*numpy.timdelta64*) – The duration of the sound to be synthesized
- **freqs_in_hz** (*list of float*) – Numbers representing the frequencies in hz that should be synthesized

class zounds.synthesize.**NoiseSynthesizer**(*samplerate*)

Synthesize white noise

Parameters **samplerate** (SampleRate) – the samplerate at which the ticks should be synthesized

Examples

```
>>> import zounds
>>> synth = zounds.NoiseSynthesizer(zounds.SR44100())
>>> samples = synth.synthesize(zounds.Seconds(2))
```

(continues on next page)

(continued from previous page)

```
>>> samples
AudioSamples([ 0.1137964 , -0.02613194,  0.30963904, ..., -0.71398137,
       -0.99840281,  0.74310827])
```

See also:

SineSynthesizer TickSynthesizer SilenceSynthesizer

synthesize(*duration*)
Synthesize white noise

Parameters **duration** (*numpy.timedelta64*) – The duration of the synthesized sound

class zounds.synthesize.**TickSynthesizer**(*samplerate*)
Synthesize short, percussive, periodic "ticks"

Parameters **samplerate** (SampleRate) – the samplerate at which the ticks should be synthesized

Examples

```
>>> import zounds
>>> synth = zounds.TickSynthesizer(zounds.SR22050())
>>> samples = synth.synthesize(            duration=zounds.Seconds(3), tick_
↪frequency=zounds.Milliseconds(100))
>>> samples
AudioSamples([ -3.91624993e-01,  -8.96939666e-01,   4.18165378e-01, ...,
        -4.08054347e-04,  -2.32257899e-04,   0.00000000e+00])
```

See also:

SineSynthesizer NoiseSynthesizer SilenceSynthesizer

synthesize(*duration*, *tick_frequency*)
Synthesize periodic "ticks", generated from white noise and an envelope

Parameters

- **duration** (*numpy.timedelta64*) – The total duration of the sound to be synthesized
- **tick_frequency** (*numpy.timedelta64*) – The frequency of the ticking sound

class zounds.synthesize.**SilenceSynthesizer**(*samplerate*)
Synthesize silence

Parameters **samplerate** (SampleRate) – the samplerate at which the ticks should be synthesized

Examples

```
>>> import zounds
>>> synth = zounds.SilenceSynthesizer(zounds.SR11025())
>>> samples = synth.synthesize(zounds.Seconds(5))
>>> samples
AudioSamples([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

synthesize(*duration*)
Synthesize silence

Parameters **duration** (*numpy.timedelta64*) – The duration of the synthesized sound

1.6 Datasets

The datasets module provides access to some common sources of audio on the internet. In general, a dataset instance is an iterable of *zounds.soundfile.AudioMetaData* instances that can be passed to the root node of an audio processing graph.

class zounds.datasets.**FreeSoundSearch**(*api_key*, *query*, *n_results=10*, *delay=0.2*)

Produces an iterable of *zounds.soundfile.AudioMetaData* instances for every result from a https://freesound.org search

Parameters

- **api_key** (*str*) – Your freesound.org API key (get one here: (http://freesound.org/apiv2/apply/))
- **query** (*str*) – The text query to perform

Raises ValueError: when *api_key* and/or *query* are not supplied

Examples

```
>>> from zounds import FreeSoundSearch
>>> fss = FreeSoundSearch('YOUR_API_KEY', 'guitar')
>>> iter(fss).next()
{'description': u'Etude of Electric Guitar in Dm. Used chorus and reverberation␣
↪effects. Size 6/4. Tempo 100. Gloomy and sentimental.', 'tags': [u'Etude', u
↪'Experemental', u'Guitar', u'guitar', u'Electric', u'Chorus'], 'uri': <Request␣
↪[GET]>, 'channels': 2, 'licensing': u'http://creativecommons.org/licenses/by/3.
↪0/', 'samplerate': 44100.0}
```

See also:

InternetArchive *PhatDrumLoops* *zounds.soundfile.AudioMetaData*

class zounds.datasets.**InternetArchive**(*archive_id*, *format_filter=None*, ***attrs*)

Produces an iterable of *zounds.soundfile.AudioMetaData* instances for every file of a particular format from an internet archive id.

Parameters

- **archive_id** (*str*) – the Internet Archive identifier
- **format_filter** (*str*) – The file format to return
- **attrs** (*dict*) – Extra attributes to add to the AudioMetaData

Raises ValueError – when archive_id is not provided

Examples

```
>>> from zounds import InternetArchive
>>> ia = InternetArchive('Greatest_Speeches_of_the_20th_Century')
>>> iter(ia).next()
{'creator': u'John F. Kennedy', 'height': u'0', 'channels': None, 'genre': u'Folk
↪', 'licensing': None, 'mtime': u'1236666800', 'samplerate': None, 'size': u
↪'7264435', 'album': u'Great Speeches of the 20th Century [Box Set] Disc 2',
↪'title': u'The Cuban Missile Crisis', 'format': u'128Kbps MP3', 'source': u
↪'original', 'description': None, 'tags': None, 'track': u'15', 'crc32': u
↪'ace17eb5', 'md5': u'e00f4e7bd9df7bdba4db7098d1ccdfe0', 'sha1': u
↪'e42d1f348078a11ed9a6ea9c8934a1236235c7b3', 'artist': u'John F. Kennedy',
↪'external-identifier': [u'urn:acoustid:ff850a0c-2efa-450f-8034-efdb31a9b696', u
↪'urn:mb_recording_id:912cedd0-5530-4f26-972c-13d131fef06e'], 'uri': <Request␣
```

(continues on next page)

(continued from previous page)

See also:

FreeSoundSearch *PhatDrumLoops* *zounds.soundfile.AudioMetaData*

class zounds.datasets.**PhatDrumLoops**(***attrs*)

Produces an iterable of *zounds.soundfile.AudioMetaData* instances for every drum break from http://phatdrumloops.com/beats.php

Parameters **attrs** (*dict*) – Extra properties to add to the AudioMetaData

Examples

```
>>> from zounds import PhatDrumLoops
>>> pdl = PhatDrumLoops()
>>> iter(pdl).next()
{'description': None, 'tags': None, 'uri': <Request [GET]>, 'channels': None,
↪'licensing': None, 'samplerate': None}
```

See also:

InternetArchive *FreeSoundSearch* *zounds.soundfile.AudioMetaData*

class zounds.datasets.**CompositeDataset**(**datasets*)

A dataset composed of two or more others

Parameters **datasets** (*list of datasets*) – One or more other datasets

Examples

```
>>> from zounds import InternetArchive, CompositeDataset, ingest
>>> dataset1 = InternetArchive('beethoven_ingigong_850')
>>> dataset2 = InternetArchive('The_Four_Seasons_Vivaldi-10361')
>>> composite = CompositeDataset(dataset1, dataset2)
>>> ingest(composite, Sound) # ingest data from both datasets
```

1.7 Learn

The *learn* module includes classes that make it possible to define processing graphs whose leaves are trained machine learning models.

While much of *zounds.soundfile*, *zounds.spectral*, and *zounds.timeseries* focus on processing nodes that can be composed into a processing graph to extract features from a single piece of audio, the *learn* module focuses on defining graphs that extract features or trained models from an entire corpus of audio.

1.7.1 PyTorch Modules

class zounds.learn.**FilterBank**(*samplerate*, *kernel_size*, *scale*, *scaling_factors*, *normalize_filters=True*, *a_weighting=True*)

A torch module that convolves a 1D input signal with a bank of morlet filters.

Parameters

- **samplerate** (SampleRate) – the samplerate of the input signal
- **kernel_size** (*int*) – the length in samples of each filter
- **scale** (FrequencyScale) – a scale whose center frequencies determine the fundamental frequency of each filer
- **scaling_factors** (*int or list of int*) – Scaling factors for each band, which determine the time-frequency resolution tradeoff. The number(s) should fall between 0 and 1, with smaller numbers achieving better frequency resolution, and larget numbers better time resolution
- **normalize_filters** (*bool*) – When true, ensure that each filter in the bank has unit norm
- **a_weighting** (*bool*) – When true, apply a perceptually-motivated weighting of the filters

See also:

AWeighting morlet_filter_bank()

class zounds.learn.**SincLayer**(*scale*, *taps*, *samplerate*)

A layer as described in the paper "Speaker Recognition from raw waveform with SincNet"

> This paper proposes a novel CNN architecture, called SincNet, that encourages the first convolutional layer to discover more meaningful filters. SincNet is based on parametrized sinc functions, which implement band-pass filters. In contrast to standard CNNs, that learn all elements of each filter, only low and high cutoff frequencies are directly learned from data with the proposed method. This offers a very compact and efficient way to derive a customized filter bank specifically tuned for the desired application. Our experiments, conducted on both speaker identification and speaker verification tasks, show that the proposed architecture converges faster and performs better than a standard CNN on raw waveforms.
>
> —https://arxiv.org/abs/1808.00158

Parameters

- **scale** (FrequencyScale) – A scale defining the initial bandpass filters
- **taps** (*int*) – The length of the filter in samples
- **samplerate** (SampleRate) – The sampling rate of incoming samples

See also:

FrequencyScale SampleRate

1.7.2 The Basics

class zounds.learn.**PreprocessingPipeline**(*needs=None*)

A *PreprocessingPipeline* is a node in the graph that can be connected to one or more *Preprocessor* nodes, whose output it will assemble into a re-usable pipeline.

Parameters **needs** (*list or tuple of Node*) – the *Preprocessor* nodes on whose output this pipeline depends

Here's an example of a learning pipeline that will first find the feature-wise mean and standard deviation of a dataset, and will then learn K-Means clusters from the dataset. This will result in a re-usable pipeline that can use statistics from the original dataset to normalize new examples, assign them to a cluster, and finally, reconstruct them.

```
import featureflow as ff
import zounds
from random import choice

samplerate = zounds.SR44100()
STFT = zounds.stft(resample_to=samplerate)


@zounds.simple_in_memory_settings
class Sound(STFT):
    bark = zounds.ArrayWithUnitsFeature(
        zounds.BarkBands,
        samplerate=samplerate,
        needs=STFT.fft,
        store=True)


@zounds.simple_in_memory_settings
class ExamplePipeline(ff.BaseModel):
    docs = ff.PickleFeature(
        ff.IteratorNode,
        needs=None)

    shuffled = ff.PickleFeature(
        zounds.ShuffledSamples,
        nsamples=100,
        needs=docs,
        store=False)

    meanstd = ff.PickleFeature(
        zounds.MeanStdNormalization,
        needs=docs,
        store=False)

    kmeans = ff.PickleFeature(
        zounds.KMeans,
        needs=meanstd,
        centroids=32)

    pipeline = ff.PickleFeature(
        zounds.PreprocessingPipeline,
        needs=(meanstd, kmeans),
        store=True)

# apply the Sound processing graph to individual audio files
for metadata in zounds.InternetArchive('TheR.H.SFXLibrary'):
    print 'processing {url}'.format(url=metadata.request.url)
    Sound.process(meta=metadata)

# apply the ExamplePipeline processing graph to the entire corpus of audio
_id = ExamplePipeline.process(docs=(snd.bark for snd in Sound))
learned = ExamplePipeline(_id)

snd = choice(list(Sound))
result = learned.pipeline.transform(snd.bark)
print result.data  # print the assigned centroids for each FFT frame
inverted = result.inverse_transform()
print inverted  # the reconstructed FFT frames
```

See also:

Pipeline *Preprocessor* *PreprocessResult* *PipelineResult*

class zounds.learn.**Pipeline**(*preprocess_results*)

class zounds.learn.**Preprocessor**(*needs=None*)

Preprocessor is the common base class for nodes in a processing graph that will produce PreprocessingResult instances that end up as part of a *Pipeline*.

Parameters **needs** (*Node*) – previous processing node(s) on which this one depends for its data

See also:

PreprocessResult *PreprocessingPipeline* *PipelineResult*

class zounds.learn.**PreprocessResult**(*data*, *op*, *inversion_data=None*, *inverse=None*, *name=None*)

PreprocessResult are the output of *Preprocessor* nodes, and can participate in a *Pipeline*.

Parameters

- **data** – the data on which the node in the graph was originally trained
- **op** (*Op*) – a callable that can transform data
- **inversion_data** – data extracted in the forward pass of the model, that can be used to invert the result
- **inverse** (*Op*) – a callable that given the output of *op*, and *inversion_data*, can invert the result

class zounds.learn.**PipelineResult**(*data*, *processors*, *inversion_data*, *wrap_data*)

1.7.3 Custom Losses

class zounds.learn.**PerceptualLoss**(*scale*, *samplerate*, *frequency_window=<ZoundsDocsMock name='mock()' id='140484303083112'>*, *basis_size=512*, *lap=2*, *log_factor=100*, *frequency_weighting=None*, *cosine_similarity=True*)

PerceptualLoss computes loss/distance in a feature space that roughly approximates early stages of the human audio processing pipeline, instead of computing raw sample loss. It decomposes a 1D (audio) signal into frequency bands using an FIR filter bank whose frequencies are centered according to a user-defined scale, performs half-wave rectification, puts amplitudes on a log scale, and finally optionally applies a re-weighting of frequency bands.

Parameters

- **scale** (FrequencyScale) – a scale defining frequencies at which the FIR filters will be centered
- **samplerate** (SampleRate) – samplerate needed to construct the FIR filter bank
- **frequency_window** (*ndarray*) – window determining how narrow or wide filter responses should be
- **basis_size** (*int*) – The kernel size, or number of "taps" for each filter
- **lap** (*int*) – The filter stride
- **log_factor** (*int*) – How much compression should be applied in the log amplitude stage

- **frequency_weighting** (FrequencyWeighting) – an optional frequency weighting to be applied after log amplitude scaling
- **cosine_similarity** (*bool*) – If *True*, compute the cosine similarity between spectrograms, otherwise, compute the mean squared error

1.7.4 Data Preparation

class zounds.learn.**UnitNorm**(*needs=None*)

class zounds.learn.**MuLawCompressed**(*needs=None*)

class zounds.learn.**MeanStdNormalization**(*needs=None*)

class zounds.learn.**InstanceScaling**(*max_value=1*, *needs=None*)

class zounds.learn.**Weighted**(*weighting*, *needs=None*)

1.7.5 Sampling

class zounds.learn.**ShuffledSamples**(*nsamples=None*, *multiplexed=False*, *dtype=None*, *needs=None*)

1.7.6 Machine Learning Models

class zounds.learn.**KMeans**(*centroids=None*, *needs=None*)

class zounds.learn.**SklearnModel**(*model=None*, *needs=None*)

class zounds.learn.**PyTorchNetwork**(*trainer=None*, *post_training_func=None*, *needs=None*, *training_set_prep=None*, *chunksize=None*)

class zounds.learn.**PyTorchGan**(*apply_network='generator'*, *trainer=None*, *needs=None*)

class zounds.learn.**PyTorchAutoEncoder**(*trainer=None*, *needs=None*)

class zounds.learn.**SupervisedTrainer**(*model*, *loss*, *optimizer*, *epochs*, *batch_size*, *holdout_percent=0.0*, *data_preprocessor=<function SupervisedTrainer.<lambda>>*, *label_preprocessor=<function SupervisedTrainer.<lambda>>*, *checkpoint_epochs=1*)

class zounds.learn.**TripletEmbeddingTrainer**(*network*, *epochs*, *batch_size*, *anchor_slice*, *deformations=None*, *checkpoint_epochs=1*)

Learn an embedding by applying the triplet loss to anchor examples, negative examples, and deformed or adjacent examples, akin to:

- *UNSUPERVISED LEARNING OF SEMANTIC AUDIO REPRESENTATIONS* <https://arxiv.org/pdf/1711.02209.pdf>

Parameters

- **network** (*nn.Module*) – the neural network to train
- **epochs** (*int*) – the desired number of passes over the entire dataset
- **batch_size** (*int*) – the number of examples in each minibatch

- **anchor_slice** (*slice*) – since choosing examples near the anchor example is one possible transformation that can be applied to find a positive example, batches generally consist of examples that are longer (temporally) than the examples that will be fed to the network, so that adjacent examples may be chosen. This slice indicates which part of the minibatch examples comprises the anchor
- **deformations** (*callable*) – a collection of other deformations or transformations that can be applied to anchor examples to derive positive examples. These callables should take two arguments: the anchor examples from the minibatch, as well as the "wider" minibatch examples that include temporally adjacent events

class zounds.learn.**WassersteinGanTrainer**(*network*, *latent_dimension*, *n_critic_iterations*, *epochs*, *batch_size*, *preprocess_minibatch=None*, *kwargs_factory=None*, *debug_gradient=False*, *checkpoint_epochs=1*)

Parameters

- **network** (*nn.Module*) – the network to train
- **latent_dimension** (*tuple*) – A tuple that defines the shape of the latent dimension (noise) that is the generator's input
- **n_critic_iterations** (*int*) – The number of minibatches the critic sees for every minibatch the generator sees
- **epochs** – The total number of passes over the training set
- **batch_size** – The size of a minibatch
- **preprocess_minibatch** (*function*) – function that takes the current epoch, and a minibatch, and mutates the minibatch
- **kwargs_factory** (*callable*) – function that takes the current epoch and outputs args to pass to the generator and discriminator

1.7.7 Hashing

class zounds.learn.**SimHash**(*bits=None*, *packbits=False*, *needs=None*)

Hash feature vectors by computing on which side of N hyperplanes those features lie.

Parameters

- **bits** (*int*) – The number of hyperplanes, and hence, the number of bits in the resulting hash
- **packbits** (*bool*) – Should the result be bit-packed?
- **needs** (Preprocessor) – the processing node on which this node relies for its data

1.7.8 Learned Models in Audio Processing Graphs

class zounds.learn.**Learned**(*learned=None*, *version=None*, *wrapper=None*, *pipeline_func=None*, *needs=None*, *dtype=None*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

z

Index

T

U

V

W

Z