Branch: master ▾

Find file Copy path

python_speech_features / python_speech_features / **sigproc.py** / Jump to ▾

hshteingart add documentation line to explain the flag

9f244cf on 13 Aug 2017

4 contributors

Raw Blame History

140 lines (109 sloc) 5.98 KB

Code navigation is available!

Navigate your code with ease. Click on function and method calls to jump to their definitions or references in the same repository. Learn more

● You're using code navigation to jump to definitions or references. Learn more or give us feedback

```python
# This file includes routines for basic signal processing including framing and computing power spectra.
# Author: James Lyons 2012
import decimal

import numpy
import math
import logging


def round_half_up(number):
    return int(decimal.Decimal(number).quantize(decimal.Decimal('1'), rounding=decimal.ROUND_HALF_UP))


def rolling_window(a, window, step=1):
    # http://ellisvalentiner.com/post/2017-03-21-np-strides-trick
    shape = a.shape[:-1] + (a.shape[-1] - window + 1, window)
    strides = a.strides + (a.strides[-1],)
    return numpy.lib.stride_tricks.as_strided(a, shape=shape, strides=strides)[::step]


def framesig(sig, frame_len, frame_step, winfunc=lambda x: numpy.ones((x,)), stride_trick=True):
    """Frame a signal into overlapping frames.

    :param sig: the audio signal to frame.
    :param frame_len: length of each frame measured in samples.
    :param frame_step: number of samples after the start of the previous frame that the next frame should begin.
    :param winfunc: the analysis window to apply to each frame. By default no window is applied.
    :param stride_trick: use stride trick to compute the rolling window and window multiplication faster
    :returns: an array of frames. Size is NUMFRAMES by frame_len.
    """
    slen = len(sig)
    frame_len = int(round_half_up(frame_len))
    frame_step = int(round_half_up(frame_step))
    if slen <= frame_len:
        numframes = 1
    else:
        numframes = 1 + int(math.ceil((1.0 * slen - frame_len) / frame_step))

    padlen = int((numframes - 1) * frame_step + frame_len)

    zeros = numpy.zeros((padlen - slen,))
    padsignal = numpy.concatenate((sig, zeros))
    if stride_trick:
        win = winfunc(frame_len)
        frames = rolling_window(padsignal, window=frame_len, step=frame_step)
    else:
        indices = numpy.tile(numpy.arange(0, frame_len), (numframes, 1)) + numpy.tile(
            numpy.arange(0, numframes * frame_step, frame_step), (frame_len, 1)).T
        indices = numpy.array(indices, dtype=numpy.int32)
        frames = padsignal[indices]
        win = numpy.tile(winfunc(frame_len), (numframes, 1))

    return frames * win


def deframesig(frames, siglen, frame_len, frame_step, winfunc=lambda x: numpy.ones((x,))):
    """Does overlap-add procedure to undo the action of framesig.

    :param frames: the array of frames.
    :param siglen: the length of the desired signal, use 0 if unknown. Output will be truncated to siglen samples.
    :param frame_len: length of each frame measured in samples.
    :param frame_step: number of samples after the start of the previous frame that the next frame should begin.
    :param winfunc: the analysis window to apply to each frame. By default no window is applied.
    :returns: a 1-D signal.
    """
    frame_len = round_half_up(frame_len)
    frame_step = round_half_up(frame_step)
    numframes = numpy.shape(frames)[0]
    assert numpy.shape(frames)[1] == frame_len, '"frames" matrix is wrong size, 2nd dim is not equal to frame_len'

    indices = numpy.tile(numpy.arange(0, frame_len), (numframes, 1)) + numpy.tile(
        numpy.arange(0, numframes * frame_step, frame_step), (frame_len, 1)).T
    indices = numpy.array(indices, dtype=numpy.int32)
    padlen = (numframes - 1) * frame_step + frame_len

    if siglen <= 0: siglen = padlen

    rec_signal = numpy.zeros((padlen,))
    window_correction = numpy.zeros((padlen,))
    win = winfunc(frame_len)

    for i in range(0, numframes):
        window_correction[indices[i, :]] = window_correction[
                                               indices[i, :]] + win + 1e-15  # add a little bit so it is never zero
        rec_signal[indices[i, :]] = rec_signal[indices[i, :]] + frames[i, :]

    rec_signal = rec_signal / window_correction
    return rec_signal[0:siglen]


def magspec(frames, NFFT):
    """Compute the magnitude spectrum of each frame in frames. If frames is an NxD matrix, output will be Nx(NFFT/2+1).

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are zero-padded.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row will be the magnitude spectrum of the corresponding frame.
    """
    if numpy.shape(frames)[1] > NFFT:
        logging.warn(
            'frame length (%d) is greater than FFT size (%d), frame will be truncated. Increase NFFT to avoid.',
            numpy.shape(frames)[1], NFFT)
    complex_spec = numpy.fft.rfft(frames, NFFT)
    return numpy.absolute(complex_spec)


def powspec(frames, NFFT):
    """Compute the power spectrum of each frame in frames. If frames is an NxD matrix, output will be Nx(NFFT/2+1).

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are zero-padded.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row will be the power spectrum of the corresponding frame.
    """
    return 1.0 / NFFT * numpy.square(magspec(frames, NFFT))


def logpowspec(frames, NFFT, norm=1):
    """Compute the log power spectrum of each frame in frames. If frames is an NxD matrix, output will be Nx(NFFT/2+1).

    :param frames: the array of frames. Each row is a frame.
    :param NFFT: the FFT length to use. If NFFT > frame_len, the frames are zero-padded.
    :param norm: If norm=1, the log power spectrum is normalised so that the max value (across all frames) is 0.
    :returns: If frames is an NxD matrix, output will be Nx(NFFT/2+1). Each row will be the log power spectrum of the corresponding frame.
    """
    ps = powspec(frames, NFFT);
    ps[ps <= 1e-30] = 1e-30
    lps = 10 * numpy.log10(ps)
    if norm:
        return lps - numpy.max(lps)
    else:
        return lps


def preemphasis(signal, coeff=0.95):
    """perform preemphasis on the input signal.

    :param signal: The signal to filter.
    :param coeff: The preemphasis coefficient. 0 is no filter, default is 0.95.
    :returns: the filtered signal.
    """
    return numpy.append(signal[0], signal[1:] - coeff * signal[:-1])
```