Package ‘soundgen’

March 13, 2019

Type Package
Title Parametric Voice Synthesis
Version 1.4.0
Date 2019-03-13
Maintainer Andrey Anikin <rty.anik@rambler.ru>
URL http://cogsci.se/soundgen.html
Description Tools for sound synthesis and acoustic analysis. Performs parametric synthesis of sounds with harmonic and noise components such as animal vocalizations or human voice. Also includes tools for spectral analysis, pitch tracking, audio segmentation, self-similarity matrices, morphing, etc.
License GPL (>= 2)
Encoding UTF-8
LazyData true
Imports stats, graphics, utils, tuneR, seewave (>= 2.1.0), zoo, shiny, reshape2, mvtnorm, plyr, dtw, phonTools
Depends R (>= 3.4)
RoxygenNote 6.1.1
Suggests knitr, rmarkdown, shinyBS
VignetteBuilder knitr
NeedsCompilation no
Author Andrey Anikin [aut, cre]
Repository CRAN
Date/Publication 2019-03-13 06:30:03 UTC

R topics documented:

addFormants ................................................................. 3
addVectors ................................................................. 5
R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze</td>
<td>6</td>
</tr>
<tr>
<td>analyzeFolder</td>
<td>12</td>
</tr>
<tr>
<td>beat</td>
<td>16</td>
</tr>
<tr>
<td>compareSounds</td>
<td>17</td>
</tr>
<tr>
<td>crossFade</td>
<td>19</td>
</tr>
<tr>
<td>defaults</td>
<td>21</td>
</tr>
<tr>
<td>estimateVTL</td>
<td>21</td>
</tr>
<tr>
<td>fade</td>
<td>22</td>
</tr>
<tr>
<td>fart</td>
<td>23</td>
</tr>
<tr>
<td>flatEnv</td>
<td>25</td>
</tr>
<tr>
<td>gaussianSmooth2D</td>
<td>26</td>
</tr>
<tr>
<td>generateNoise</td>
<td>27</td>
</tr>
<tr>
<td>getEntropy</td>
<td>29</td>
</tr>
<tr>
<td>getIntegerRandomWalk</td>
<td>30</td>
</tr>
<tr>
<td>getLoudness</td>
<td>31</td>
</tr>
<tr>
<td>getLoudnessFolder</td>
<td>33</td>
</tr>
<tr>
<td>getRandomWalk</td>
<td>34</td>
</tr>
<tr>
<td>getRMS</td>
<td>35</td>
</tr>
<tr>
<td>getRMSFolder</td>
<td>37</td>
</tr>
<tr>
<td>getRolloff</td>
<td>38</td>
</tr>
<tr>
<td>getSmoothContour</td>
<td>40</td>
</tr>
<tr>
<td>getSpectralEnvelope</td>
<td>42</td>
</tr>
<tr>
<td>HzToSemitones</td>
<td>44</td>
</tr>
<tr>
<td>matchPars</td>
<td>45</td>
</tr>
<tr>
<td>modulationSpectrum</td>
<td>47</td>
</tr>
<tr>
<td>modulationSpectrumFolder</td>
<td>50</td>
</tr>
<tr>
<td>morph</td>
<td>52</td>
</tr>
<tr>
<td>normalizeFolder</td>
<td>53</td>
</tr>
<tr>
<td>notesDict</td>
<td>55</td>
</tr>
<tr>
<td>optimizePars</td>
<td>55</td>
</tr>
<tr>
<td>osc_dB</td>
<td>58</td>
</tr>
<tr>
<td>permittedValues</td>
<td>59</td>
</tr>
<tr>
<td>pitchManual</td>
<td>60</td>
</tr>
<tr>
<td>playme</td>
<td>60</td>
</tr>
<tr>
<td>presets</td>
<td>61</td>
</tr>
<tr>
<td>reportTime</td>
<td>61</td>
</tr>
<tr>
<td>schwa</td>
<td>62</td>
</tr>
<tr>
<td>segment</td>
<td>62</td>
</tr>
<tr>
<td>segmentFolder</td>
<td>64</td>
</tr>
<tr>
<td>segmentManual</td>
<td>67</td>
</tr>
<tr>
<td>semitonesToHz</td>
<td>69</td>
</tr>
<tr>
<td>soundgen</td>
<td>70</td>
</tr>
<tr>
<td>soundgen_app</td>
<td>75</td>
</tr>
<tr>
<td>spectrogram</td>
<td>75</td>
</tr>
<tr>
<td>spectrogramFolder</td>
<td>78</td>
</tr>
<tr>
<td>ssm</td>
<td>79</td>
</tr>
</tbody>
</table>
**Description**

A spectral filter that either adds or removes formants from a sound - that is, amplifies or dampens certain frequency bands, as in human vowels. See `soundgen` and `getSpectralEnvelope` for more information. With action = 'remove' this function can perform inverse filtering to remove formants and obtain raw glottal output, provided that you can specify the correct formant structure.

**Usage**

```r
addFormants(sound, formants, spectralEnvelope = NULL, action = c("add", "remove")[,1], vocalTract = NA, formantDep = 1,
formantDepStoch = 20, formantWidth = 1, lipRad = 6, noseRad = 4,
mouthOpenThres = 0, mouth = NA, interpol = c("approx", "spline", "loess")[,3], temperature = 0.025, formDrift = 0.3, formDisp = 0.2,
samplingRate = 16000, windowLength_points = 800, overlap = 75,
normalize = TRUE)
```

**Arguments**

- **sound** numeric vector with samplingRate
- **formants** either a character string like "aai" referring to default presets for speaker "M1" or a list of formant times, frequencies, amplitudes, and bandwidths (see ex. below). `formants = NA` defaults to schwa. Time stamps for formants and mouthOpening can be specified in ms or an any other arbitrary scale. See `getSpectralEnvelope` for more details
- **spectralEnvelope** (optional): as an alternative to specifying formant frequencies, we can provide the exact filter - a vector of non-negative numbers specifying the power in each frequency bin on a linear scale (interpolated to length equal to `windowLength_points/2`). A matrix specifying the filter for each STFT step is also accepted. The easiest way to create this matrix is to call `soundgen:::.getSpectralEnvelope` or to use the spectrum of a recorded sound
- **action** 'add' = add formants to the sound, 'remove' = remove formants (inverse filtering)
- **vocalTract** the length of vocal tract, cm. Used for calculating formant dispersion (for adding extra formants) and formant transitions as the mouth opens and closes. If NULL or NA, the length is estimated based on specified formant frequencies (if any)
- **formantDep** scale factor of formant amplitude (1 = no change relative to amplitudes in `formants`)
- **formantDepStoch** the amplitude of additional stochastic formants added above the highest specified formant, dB (only if temperature > 0)
- **formantWidth** = scale factor of formant bandwidth (1 = no change)
The `addformants` function in R is a tool for modifying the spectrum of audio signals by adding formants. It works in conjunction with the `spectrogram` function and allows for fine-tuning the spectral content of sounds.

The key parameters are:

- **lipRad**: the effect of lip radiation on source spectrum, dB/oct. The default of +6 dB/oct produces a high-frequency boost when the mouth is open.
- **noseRad**: the effect of radiation through the nose on source spectrum, dB/oct. The alternative to `lipRad` when the mouth is closed.
- **mouthOpenThres**: open the lips (switch from nose radiation to lip radiation) when the mouth is open. 0 to 1.
- **mouth**: mouth opening (0 to 1, 0.5 = neutral, i.e., no modification). The anchor format.
- **interpol**: the method of smoothing envelopes based on provided mouth anchors: 'approx' = linear interpolation, 'spline' = cubic spline, 'loess' (default) = polynomial local smoothing function. NB: this does NOT affect the smoothing of formant anchors.
- **temperature**: hyperparameter for regulating the amount of stochasticity in sound generation.
- **formDrift**, **formDisp**: scaling factors for the effect of temperature on formant drift and dispersal, respectively.
- **samplingRate**: sampling frequency, Hz.
- **windowLength_points**: length of FFT window, points.
- **overlap**: FFT window overlap, %. For allowed values, see `istft`.
- **normalize**: if TRUE, normalizes the output to range from -1 to +1.

### Details

Algorithm: converts input from a time series (time domain) to a spectrogram (frequency domain) through short-term Fourier transform (STFT), multiples by the spectral filter containing the specified formants, and transforms back to a time series via inverse STFT. This is a subroutine in `soundgen`, but it can also be used on any existing sound.

### Examples

```r
# White noise
sound = c(rep(0, 1000), runif(16000), rep(0, 1000))  # white noise
# NB: pad with silence to avoid artefacts if removing formants
# playme(sound)
# spectrogram(sound, samplingRate = 16000)

# Add F1 = 900, F2 = 1300 Hz
sound_filtered = addFormants(sound, formants = c(900, 1300))
# playme(sound_filtered)
# spectrogram(sound_filtered, samplingRate = 16000)

# ... and remove them again (assuming we know what the formants are)
sound_inverse_filt = addFormants(sound_filtered, formants = c(900, 1300),
                                 action = 'remove')
# playme(sound_inverse_filt)
# spectrogram(sound_inverse_filt, samplingRate = 16000)
```

## Not run:
Description

Add two partly overlapping vectors, such as two waveforms, to produce a longer vector. The location at which vector 2 is pasted is defined by insertionPoint. Algorithm: both vectors are
padded with zeros to match in length and then added. All NA’s are converted to 0.

Usage

```r
addVectors(v1, v2, insertionPoint = 1, normalize = TRUE)
```

Arguments

- `v1, v2`: numeric vectors
- `insertionPoint`: the index of element in vector 1 at which vector 2 will be inserted (any integer, can also be negative)
- `normalize`: if TRUE, the output is normalized to range from -1 to +1

Examples

```r
v1 = 1:6
v2 = rep(100, 3)
addVectors(v1, v2, insertionPoint = 5, normalize = FALSE)
addVectors(v1, v2, insertionPoint = -4, normalize = FALSE)
# note the asymmetry: insertionPoint refers to the first arg
addVectors(v2, v1, insertionPoint = -4, normalize = FALSE)

v3 = rep(100, 15)
addVectors(v1, v3, insertionPoint = -4, normalize = FALSE)
addVectors(v2, v3, insertionPoint = 7, normalize = FALSE)
```

---

**analyze**

**Analyze sound**

Description

Acoustic analysis of a single sound file: pitch tracking, basic spectral characteristics, and estimated loudness (see `getLoudness`). The default values of arguments are optimized for human non-linguistic vocalizations. See vignette('acoustic_analysis', package = 'soundgen') for details.

Usage

```r
analyze(x, samplingRate = NULL, dynamicRange = 80, silence = 0.04,
        scale = NULL, SPL_measured = 70, Pref = 2e-05, windowLength = 50,
        step = NULL, overlap = 50, wn = "gaussian", zp = 0,
        cutFreq = 6000, nFormants = 3, pitchMethods = c("autocor", "spec", "dom"),
        entropyThres = 0.6, pitchFloor = 75, pitchCeiling = 3500,
        priorMean = HzToSemitones(300), priorSD = 6, priorPlot = FALSE,
        nCands = 1, minVoicedCands = 'autom', domThres = 0.1,
        domSmooth = 220, autocorThres = 0.7, autocorSmooth = NULL,
        cepThres = 0.3, cepSmooth = NULL, cepZp = 0, specThres = 0.3,
        specPeak = 0.35, specSinglePeakCert = 0.4, specHNRslope = 0.8,
        specSmooth = 150, specMerge = 1, shortestSyl = 20,
```
analyze

shortestPause = 60, interpWin = 3, interpTol = 0.3, interpCert = 0.3, pathfinding = c("none", "fast", "slow")[2], annealPars = list(maxit = 5000, temp = 1000), certWeight = 0.5, snakeStep = 0.05, snakePlot = FALSE, smooth = 1, smoothVars = c("pitch", "dom"), summary = FALSE, summaryStats = c("mean", "median", "sd"), plot = TRUE, showLegend = TRUE, savePath = NA, plotSpec = TRUE, pitchPlot = list(col = rgb(0, 0, 1, 0.75), lwd = 3), candPlot = list(), ylim = NULL, xlab = "Time, ms", ylab = "kHz", main = NULL, width = 900, height = 500, units = "px", res = NA, ...

Arguments

x                path to a .wav or .mp3 file or a vector of amplitudes with specified samplingRate
samplingRate    sampling rate of x (only needed if x is a numeric vector, rather than an audio file)
dynamicRange    dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
silence          (0 to 1) frames with RMS amplitude below silence threshold are not analyzed at all. NB: this number is dynamically updated: the actual silence threshold may be higher depending on the quietest frame, but it will never be lower than this specified number.
scale            maximum possible amplitude of input used for normalization (not needed for audio files)
SPL_measured     sound pressure level at which the sound is presented, dB
Pref             reference pressure, Pa
windowLength     length of FFT window, ms
step             you can override overlap by specifying FFT step, ms
overlap          overlap between successive FFT frames, %
wn               window type: gaussian, hanning, hamming, bartlett, rectangular, blackman, flat-top
zp               window length after zero padding, points
cutFreq          (>0 to Nyquist, Hz) repeat the calculation of spectral descriptives after discarding all info above cutFreq. Recommended if the original sampling rate varies across different analyzed audio files
nFormants        the number of formants to extract per FFT frame. Calls findformants with default settings
pitchMethods     methods of pitch estimation to consider for determining pitch contour: 'autocor' = autocorrelation (~PRAAT), 'cep' = cepstral, 'spec' = spectral (~BaNa), 'dom' = lowest dominant frequency band
entropyThres     pitch tracking is not performed for frames with Weiner entropy above entropyThres, but other spectral descriptives are still calculated
pitchFloor, pitchCeiling
absolute bounds for pitch candidates (Hz)
priorMean, priorSD
specifies the mean and sd of gamma distribution describing our prior knowledge about the most likely pitch values for this file. Specified in semitones: priorMean = HzToSemitones(300), priorSD = 6 gives a prior with mean = 300 Hz and SD of 6 semitones (half an octave)
priorPlot
if TRUE, produces a separate plot of the prior
nCands
maximum number of pitch candidates per method (except for dom, which returns at most one candidate per frame), normally 1...4
minVoicedCands
minimum number of pitch candidates that have to be defined to consider a frame voiced (defaults to 2 if dom is among other candidates and 1 otherwise)
domThres
(0 to 1) to find the lowest dominant frequency band, we do short-term FFT and take the lowest frequency with amplitude at least domThres
domSmooth
the width of smoothing interval (Hz) for finding dom
autocorThres, cepThres, specThres
(0 to 1) separate voicing thresholds for detecting pitch candidates with three different methods: autocorrelation, cepstrum, and BaNa algorithm (see Details). Note that HNR is calculated even for unvoiced frames.
autocorSmooth
the width of smoothing interval (in bins) for finding peaks in the autocorrelation function. Defaults to 7 for sampling rate 44100 and smaller odd numbers for lower values of sampling rate
cepSmooth
the width of smoothing interval (in bins) for finding peaks in the cepstrum. Defaults to 31 for sampling rate 44100 and smaller odd numbers for lower values of sampling rate
cepZp
zero-padding of the spectrum used for cepstral pitch detection (final length of spectrum after zero-padding in points, e.g. $2^{13}$)
specPeak, specHNR$\times$slope
when looking for putative harmonics in the spectrum, the threshold for peak detection is calculated as specPeak $\times$ (1 - HNR $\times$ specHNR$\times$slope)
specSinglePeakCert
(0 to 1) if F0 is calculated based on a single harmonic ratio (as opposed to several ratios converging on the same candidate), its certainty is taken to be specSinglePeakCert
specSmooth
the width of window for detecting peaks in the spectrum, Hz
specMerge
pitch candidates within specMerge semitones are merged with boosted certainty
shortestSyl
the smallest length of a voiced segment (ms) that constitutes a voiced syllable (shorter segments will be replaced by NA, as if unvoiced)
shortestPause
the smallest gap between voiced syllables (ms) that means they shouldn’t be merged into one voiced syllable
interpolWin, interpolTol, interpolCert
control the behavior of interpolation algorithm when postprocessing pitch candidates. To turn off interpolation, set interpolWin to NULL. See soundgen:::pathfinder for details.
analyze

pathfinding: method of finding the optimal path through pitch candidates: 'none' = best candidate per frame, 'fast' = simple heuristic, 'slow' = annealing. See soundgen::pathfinder

annealPars: a list of control parameters for postprocessing of pitch contour with SANN algorithm of optim. This is only relevant if pathfinding = 'slow'

certWeight: (0 to 1) in pitch postprocessing, specifies how much we prioritize the certainty of pitch candidates vs. pitch jumps / the internal tension of the resulting pitch curve

snakeStep: optimized path through pitch candidates is further processed to minimize the elastic force acting on pitch contour. To disable, set snakeStep to NULL

snakePlot: if TRUE, plots the snake

smooth, smoothVars:
if smooth is a positive number, outliers of the variables in smoothVars are adjusted with median smoothing. smooth of 1 corresponds to a window of ~100 ms and tolerated deviation of ~4 semitones. To disable, set smooth to NULL

summary: if TRUE, returns only a summary of the measured acoustic variables (mean, median and SD). If FALSE, returns a list containing frame-by-frame values

summaryStats: a vector of names of functions used to summarize each acoustic characteristic

plot: if TRUE, produces a spectrogram with pitch contour overlaid

showLegend: if TRUE, adds a legend with pitch tracking methods

savePath: if a valid path is specified, a plot is saved in this folder (defaults to NA)

plotSpec: if FALSE, the spectrogram will not be plotted

pitchPlot: a list of graphical parameters for displaying the final pitch contour. Set to NULL or NA to suppress

candPlot: a list of graphical parameters for displaying individual pitch candidates. Set to NULL or NA to suppress

ylim: frequency range to plot, kHz (defaults to 0 to Nyquist frequency)

xlab, ylab, main:
plotting parameters

width, height, units, res:
parameters passed to png if the plot is saved

Value

If summary = TRUE, returns a dataframe with one row and three columns per acoustic variable (mean / median / SD). If summary = FALSE, returns a dataframe with one row per FFT frame and one column per acoustic variable. The best guess at the pitch contour considering all available information is stored in the variable called "pitch". In addition, the output contains pitch estimates by separate algorithms included in pitchMethods and a number of other acoustic descriptors:

time: time of the middle of each frame (ms)
ampl: root mean square of amplitude per frame, calculated as sqrt(mean(frame ^ 2))
amplVoiced: the same as ampl for voiced frames and NA for unvoiced frames
analyze

dom  lowest dominant frequency band (Hz) (see “Pitch tracking methods / Dominant frequency” in the vignette)

entropy  Weiner entropy of the spectrum of the current frame. Close to 0: pure tone or tonal sound with nearly all energy in harmonics; close to 1: white noise

f1_freq, f1_width, ...  the frequency and bandwidth of the first nFormants formants per FFT frame, as calculated by phonTools::findformants with default settings

harmonics  the amount of energy in upper harmonics, namely the ratio of total spectral mass above 1.25 x F0 to the total spectral mass below 1.25 x F0 (dB)

HNR  harmonics-to-noise ratio (dB), a measure of harmonicity returned by soundgen:::getPitchAutocor (see “Pitch tracking methods / Autocorrelation”). If HNR = 0 dB, there is as much energy in harmonics as in noise

loudness  subjective loudness, in sone, corresponding to the chosen SPL_measured - see getLoudness

medianFreq  50th quantile of the frame’s spectrum

peakFreq  the frequency with maximum spectral power (Hz)

peakFreqCut  the frequency with maximum spectral power below cutFreq (Hz)

pitch  post-processed pitch contour based on all F0 estimates

pitchAutocor  autocorrelation estimate of F0

pitchCep  cepstral estimate of F0

pitchSpec  BaNa estimate of F0

quartile25, quartile50, quartile75  the 25th, 50th, and 75th quantiles of the spectrum below cutFreq (Hz)

specCentroid  the center of gravity of the frame’s spectrum, first spectral moment (Hz)

specCentroidCut  the center of gravity of the frame’s spectrum below cutFreq

specSlope  the slope of linear regression fit to the spectrum below cutFreq

voiced  is the current FFT frame voiced? TRUE / FALSE

Examples

```r
sound = soundgen(syllLen = 300, pitch = c(900, 400, 2300),
noise = list(time = c(0, 300), value = c(-40, 00)),
temperature = 0.001, addSilence = 0)
# playme(sound, 16000)
a = analyze(sound, samplingRate = 16000, plot = TRUE)

## Not run:
sound1 = soundgen(syllLen = 900, pitch = list(
  time = c(0, .3, .9, 1), value = c(300, 900, 400, 2300)),
noise = list(time = c(0, 300), value = c(-40, 00)),
temperature = 0.001, addSilence = 0)
# improve the quality of postprocessing:
a1 = analyze(sound1, samplingRate = 16000, plot = TRUE, pathfinding = 'slow')
median(a1$pitch, na.rm = TRUE)
# (can vary, since postprocessing is stochastic)
# compare to the true value:
median(getSmoothContour(anchors = list(time = c(0, .3, .8, 1)),
```
analyze

```r
value = c(300, 900, 400, 2300)), len = 1000))

# the same pitch contour, but harder b/c of subharmonics and jitter
sound2 = soundgen(syllLen = 900, pitch = list(  
  time = c(0, .3, .8, 1), value = c(300, 900, 400, 2300)),  
  noise = list(time = c(0, 900), value = c(-40, 20)),  
  subdep = 100, jitterDep = 0.5, nonlinBalance = 100, temperature = 0.001)

# playme(sound2, 16000)
a2 = analyze(sound2, samplingRate = 16000, plot = TRUE, pathfinding = 'slow')
# many candidates are off, but the overall contour should be mostly accurate

# Fancy plotting options:
a = analyze(sound2, samplingRate = 16000, plot = TRUE,  
xlab = 'Time, ms', colorTheme = 'seewave',  
contrast = .5, ylim = c(0, 4),  
pitchMethods = c('dom', 'autocor', 'spec'),  
candPlot = list(  
  col = c('gray70', 'yellow', 'purple'), # same order as pitchMethods  
pch = c(1, 3, 5),  
cex = 3),  
pitchPlot = list(col = 'black', lty = 3, lwd = 3))

# Plot pitch candidates w/o a spectrogram
a = analyze(sound2, samplingRate = 16000, plot = TRUE, plotSpec = FALSE)

# Different formatting options for output
a = analyze(sound2, samplingRate = 16000, summary = FALSE) # frame-by-frame
a = analyze(sound2, samplingRate = 16000, summary = TRUE,  
  summaryStats = c('mean', 'range')) # one row per sound

# Save the plot
a = analyze(sound, samplingRate = 16000,  
  savePath = '~/Downloads/',  
  width = 20, height = 15, units = 'cm', res = 300)

## Amplitude and loudness: analyze() should give the same results as  
# dedicated functions getRMS() / getLoudness()
# Create 1 kHz tone
samplingRate = 16000; dur_ms = 50  
sound1 = sin(2*pi*1000/samplingRate*(1:(dur_ms/1000*samplingRate))))
a1 = analyze(sound1, samplingRate = samplingRate, windowLength = 25,  
  overlap = 50, SPL_measured = 40, scale = 1,  
  pitchMethods = NULL, plot = FALSE)
a1$loudness # loudness per STFT frame (1 sone by definition)
getLoudness(sound1, samplingRate = samplingRate, windowLength = 25,  
  overlap = 50, SPL_measured = 40, scale = 1)$loudness
a1$samp # RMS amplitude per STFT frame
getRMS(sound1, samplingRate = samplingRate, windowLength = 25,  
  overlap = 50, scale = 1)
# or even simply: sqrt(mean(sound1 ^ 2))

# The same sound as above, but with half the amplitude
a_half = analyze(sound1/2, samplingRate = samplingRate, windowLength = 25,
```

---

The text you've provided includes a block of code written in R, a widely used programming language for statistical computing and graphics. The code appears to be part of a larger script that likely deals with audio signal processing, analyzing pitch contours, and plotting the results. The code includes various functions and imports, such as `analyze()`, `soundgen()`, and `plot()`, which are typically used for audio analysis and visualization. The script also incorporates options for fancy plotting, output formatting, and saving plots, which are essential for detailed analysis and presentation of results. The context suggests that the script is designed to handle audio signals, analyze their pitch contours, and generate plots that visualize the pitch data, allowing for a thorough examination of the audio signal's characteristics.
analyzeFolder

Description

Acoustic analysis of all wav/mp3 files in a folder. See `analyze` and vignette('acoustic_analysis', package = 'soundgen') for further details.

Usage

```r
analyzeFolder(myfolder, htmlPlots = TRUE, verbose = TRUE,
              samplingRate = NULL, dynamicRange = 80, silence = 0.04,
              SPL_measured = 70, Pref = 2e-05, windowLength = 50, step = NULL,
              overlap = 50, wn = "gaussian", zp = 0, cutFreq = 6000,
              nFormants = 3, pitchMethods = c("autocor", "spec", "dom"),
              entropyThres = 0.6, pitchFloor = 75, pitchCeiling = 3500,
              priorMean = HzToSemitones(300), priorSD = 6, priorPlot = FALSE,
              nCands = 1, minVoicedCands = "autom", domThres = 0.1,
              domSmooth = 220, autocorThres = 0.7, autocorSmooth = NULL,
              cepThres = 0.3, cepSmooth = NULL, cepZp = 0, specThres = 0.3,
              specPeak = 0.35, specSinglePeakCert = 0.4, specHNRslope = 0.8,
              specSmooth = 150, specMerge = 1, shortestSyl = 20,
              shortestPause = 60, interpoltWin = 3, interpoltol = 0.3,
              interpoltCert = 0.3, pathfinding = c("none", "fast", "slow")[2],
              annealPars = list(maxit = 5000, temp = 1000), certWeight = 0.5,
              snakeStep = 0.05, snakePlot = FALSE, smooth = 1,
              smoothVars = c("pitch", "dom"), summary = TRUE,
              summaryStats = c("mean", "median", "sd"), plot = FALSE,
              showLegend = TRUE, savePlots = FALSE, plotSpec = TRUE,
              pitchPlot = list(col = rgb(0, 0, 1, 0.75), lwd = 3),
              candPlot = list(levels = c("autocor", "spec", "dom", "cep"), col =
                              c("green", "red", "orange", "violet"), pch = c(16, 2, 3, 7), cex = 2)
```

```r
overlap = 50, SPL_measured = 40, scale = 1,
pitchMethods = NULL, plot = FALSE)
a1$amp1 / a_half$amp1 # rms amplitude halved
a1$loudness/ a_half$loudness # loudness is not a linear function of amplitude
```

### # Amplitude & loudness of an existing audio file

```r
sound2 = '~/Downloads/temp/032_ut_anger_30-m-roar-curse.wav'
a2 = analyze(sound2, windowLength = 25, overlap = 50, SPL_measured = 40,
pitchMethods = NULL, plot = FALSE)
apply(a2[, c('loudness', 'amp1')], 2, median, na.rm = TRUE)
median(getLoudness(sound2, windowLength = 25, overlap = 50,
SPL_measured = 40)$loudness)
median(getRMS(sound2, windowLength = 25, overlap = 50, scale = 1))
```

```r
## End(Not run)
```
Arguments

myfolder               full path to target folder
htmlPlots             if TRUE, saves an html file with clickable plots
verbose               if TRUE, reports progress and estimated time left
samplingRate         sampling rate of \( x \) (only needed if \( x \) is a numeric vector, rather than an audio file)
dynamicRange         dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
silence               (0 to 1) frames with RMS amplitude below silence threshold are not analyzed at all. NB: this number is dynamically updated: the actual silence threshold may be higher depending on the quietest frame, but it will never be lower than this specified number.
SPL_measured         sound pressure level at which the sound is presented, dB
Pref                  reference pressure, Pa
windowLength         length of FFT window, ms
step                  you can override overlap by specifying FFT step, ms
overlap               overlap between successive FFT frames, %
wn                    window type: gaussian, hanning, hamming, bartlett, rectangular, blackman, flattop
zp                    window length after zero padding, points
cutFreq               (>0 to Nyquist, Hz) repeat the calculation of spectral descriptives after discarding all info above cutFreq. Recommended if the original sampling rate varies across different analyzed audio files
nFormants            the number of formants to extract per FFT frame. Calls findformants with default settings
pitchMethods         methods of pitch estimation to consider for determining pitch contour: ‘autocor’ = autocorrelation (~PRAAT), ‘cep’ = cepstral, ‘spec’ = spectral (~BaNa), ‘dom’ = lowest dominant frequency band
entropyThres          pitch tracking is not performed for frames with Weiner entropy above entropyThres, but other spectral descriptives are still calculated
pitchFloor           absolute bounds for pitch candidates (Hz)
pitchCeiling          absolute bounds for pitch candidates (Hz)
priorMean            specifies the mean and sd of gamma distribution describing our prior knowledge about the most likely pitch values for this file. Specified in semitones:
                      priorMean = HzToSemitones(300), priorSD = 6 gives a prior with mean = 300 Hz and SD of 6 semitones (half an octave)
priorSD              specifies the mean and sd of gamma distribution describing our prior knowledge about the most likely pitch values for this file. Specified in semitones:
                      priorMean = HzToSemitones(300), priorSD = 6 gives a prior with mean = 300 Hz and SD of 6 semitones (half an octave)
priorPlot if TRUE, produces a separate plot of the prior
nCands maximum number of pitch candidates per method (except for dom, which returns
at most one candidate per frame), normally 1...4
minVoicedCands minimum number of pitch candidates that have to be defined to consider a frame
voiced (defaults to 2 if dom is among other candidates and 1 otherwise)
domThres (0 to 1) to find the lowest dominant frequency band, we do short-term FFT and
take the lowest frequency with amplitude at least domThres
domSmooth the width of smoothing interval (Hz) for finding dom
autocorThres (0 to 1) separate voicing thresholds for detecting pitch candidates with three
different methods: autocorrelation, cepstrum, and BaNa algorithm (see Details).
Note that HNR is calculated even for unvoiced frames.
autocorSmooth the width of smoothing interval (in bins) for finding peaks in the autocorrelation
function. Defaults to 7 for sampling rate 44100 and smaller odd numbers for lower values of sampling rate
cepThres (0 to 1) separate voicing thresholds for detecting pitch candidates with three
different methods: autocorrelation, cepstrum, and BaNa algorithm (see Details).
Note that HNR is calculated even for unvoiced frames.
cepSmooth the width of smoothing interval (in bins) for finding peaks in the cepstrum. De-
defaults to 31 for sampling rate 44100 and smaller odd numbers for lower values of sampling rate
cepZp zero-padding of the spectrum used for cepstral pitch detection (final length of
spectrum after zero-padding in points, e.g. 2 ^ 13)
specThres (0 to 1) separate voicing thresholds for detecting pitch candidates with three
different methods: autocorrelation, cepstrum, and BaNa algorithm (see Details).
Note that HNR is calculated even for unvoiced frames.
specPeak when looking for putative harmonics in the spectrum, the threshold for peak
detection is calculated as specPeak * (1 - HNR * specHNRslope)
specSinglePeakCert (0 to 1) if F0 is calculated based on a single harmonic ratio (as opposed to
several ratios converging on the same candidate), its certainty is taken to be
specSinglePeakCert
specHNRslope when looking for putative harmonics in the spectrum, the threshold for peak
detection is calculated as specPeak * (1 - HNR * specHNRslope)
specSmooth the width of window for detecting peaks in the spectrum, Hz
specMerge pitch candidates within specMerge semitones are merged with boosted certainty
shortestSyl the smallest length of a voiced segment (ms) that constitutes a voiced syllable
(shorter segments will be replaced by NA, as if unvoiced)
shortestPause the smallest gap between voiced syllables (ms) that means they shouldn’t be
merged into one voiced syllable
interpolWin control the behavior of interpolation algorithm when postprocessing pitch candi-
dates. To turn off interpolation, set interpolWin to NULL. See soundgen:::pathfinder
for details.
analyzeFolder

**interpoltol** control the behavior of interpolation algorithm when postprocessing pitch candidates. To turn off interpolation, set `interpolWin` to NULL. See `soundgen::pathfinder` for details.

**interpCert** control the behavior of interpolation algorithm when postprocessing pitch candidates. To turn off interpolation, set `interpolWin` to NULL. See `soundgen::pathfinder` for details.

**pathfinding** method of finding the optimal path through pitch candidates: 'none' = best candidate per frame, 'fast' = simple heuristic, 'slow' = annealing. See `soundgen::pathfinder` for details.

**annealPars** a list of control parameters for postprocessing of pitch contour with SANN algorithm of `optim`. This is only relevant if `pathfinding = 'slow'`

**certWeight** (0 to 1) in pitch postprocessing, specifies how much we prioritize the certainty of pitch candidates vs. pitch jumps / the internal tension of the resulting pitch curve

**snakeStep** optimized path through pitch candidates is further processed to minimize the elastic force acting on pitch contour. To disable, set `snakeStep` to NULL

**snakePlot** if TRUE, plots the snake

**smooth** if smooth is a positive number, outliers of the variables in `smoothVars` are adjusted with median smoothing. smooth of 1 corresponds to a window of ~100 ms and tolerated deviation of ~4 semitones. To disable, set smooth to NULL

**smoothVars** if smooth is a positive number, outliers of the variables in `smoothVars` are adjusted with median smoothing. smooth of 1 corresponds to a window of ~100 ms and tolerated deviation of ~4 semitones. To disable, set smooth to NULL

**summary** if TRUE, returns only a summary of the measured acoustic variables (mean, median and SD). If FALSE, returns a list containing frame-by-frame values

**summaryStats** a vector of names of functions used to summarize each acoustic characteristic

**plot** if TRUE, produces a spectrogram with pitch contour overlaid

**showLegend** if TRUE, adds a legend with pitch tracking methods

**savePlots** if TRUE, saves plots as .png files

**plotSpec** if FALSE, the spectrogram will not be plotted

**pitchPlot** a list of graphical parameters for displaying the final pitch contour. Set to NULL or NA to suppress

**candPlot** a list of graphical parameters for displaying individual pitch candidates. Set to NULL or NA to suppress

**ylim** frequency range to plot, kHz (defaults to 0 to Nyquist frequency)

**xlab** plotting parameters

**ylab** plotting parameters

**main** plotting parameters

**width** parameters passed to `png` if the plot is saved

**height** parameters passed to `png` if the plot is saved

**units** parameters passed to `png` if the plot is saved

**res** parameters passed to `png` if the plot is saved

... other graphical parameters passed to `spectrogram`
Value

If summary is TRUE, returns a dataframe with one row per audio file. If summary is FALSE, returns a list of detailed descriptives.

Examples

```r
## Not run:
# download 260 sounds from Anikin & Persson (2017)
# unzip them into a folder, say '~/Downloads/temp'
myFolder = '~/Downloads/temp' # 260 .wav files live here
s = analyzeFolder(myFolder, verbose = TRUE) # ~ 15-30 minutes!

# Save spectrograms with pitch contours plus an html file for easy access
a = analyzeFolder('~/Downloads/temp', savePlots = TRUE,
  showLegend = TRUE,
  width = 20, height = 12,
  units = 'cm', res = 300)

# Check accuracy: import manually verified pitch values (our "key")
key = pitchManual # a vector of 260 floats
trial = s$pitch_median
cor(key, trial, use = 'pairwise.complete.obs')
plot(log(key), log(trial))
abline(a=0, b=1, col='red')

## End(Not run)
```

**beat**

Generate beat

**Description**

Generates percussive sounds from clicks through drum-like beats to sliding tones. The principle is to create a sine wave with rapid frequency modulation and to add a fade-out. No extra harmonics or formants are added. For this specific purpose, this is vastly faster and easier than to tinker with soundgen settings, especially since percussive syllables tend to be very short.

**Usage**

```r
beat(nSyl = 10, syllLen = 200, pauseLen = 50, pitch = c(200, 10),
  samplingRate = 16000, fadeOut = TRUE, play = FALSE)
```

**Arguments**

- `nSyl` the number of syllables to generate
- `syllLen` average duration of each syllable, ms
- `pauseLen` average duration of pauses between syllables, ms
comparesounds

**Description**

Computes similarity between two sounds based on correlating mel-transformed spectra (auditory spectra). Called by `matchPars`.

**Usage**

```r
comparesounds(target, targetSpec = NULL, cand, samplingRate = NULL, method = c("cor", "cosine", "pixel", "dtw")[1:4], windowLength = 40, overlap = 50, step = NULL, padWith = NA, penalizeLengthDiff = TRUE, dynamicRange = 80, maxFreq = NULL, summary = TRUE)
```

**Value**

Returns a non-normalized waveform centered at zero.

**Examples**

```r
playback = c(TRUE, FALSE)[2]
# a drum-like sound
s = beat(nSyl = 1, syllLen = 200, 
  pitch = c(200, 100), play = playback)
# plot(s, type = 'l')

# a dry, muted drum
s = beat(nSyl = 1, syllLen = 200, 
  pitch = c(200, 10), play = playback)

# sci-fi laser guns
s = beat(nSyl = 3, syllLen = 300, 
  pitch = c(1000, 50), play = playback)

# machine guns
s = beat(nSyl = 10, syllLen = 10, pauseLen = 50, 
  pitch = c(2300, 300), play = playback)
```
compareSounds

Arguments

target  the sound we want to reproduce using soundgen: path to a .wav file or numeric vector
targetSpec if already calculated, the target auditory spectrum can be provided to speed things up
cand  the sound to be compared to target
samplingRate  sampling rate of target (only needed if target is a numeric vector, rather than a .wav file)
method  method of comparing mel-transformed spectra of two sounds: "cor" = average Pearson’s correlation of mel-transformed spectra of individual FFT frames; "cosine" = same as "cor" but with cosine similarity instead of Pearson’s correlation; "pixel" = absolute difference between each point in the two spectra; "dtw" = discrete time warp with dtw
windowLength  length of FFT window, ms
overlap overlap between successive FFT frames, %
step  you can override overlap by specifying FFT step, ms
padWith  compared spectra are padded with either silence (padWith = 0) or with NA's (padWith = NA) to have the same number of columns. When the sounds are of different duration, padding with zeros rather than NA's improves the fit to target measured by method = 'pixel' and 'dtw', but it has no effect on 'cor' and 'cosine'.
penalizeLengthDiff  if TRUE, sounds of different length are considered to be less similar; if FALSE, only the overlapping parts of two sounds are compared
dynamicRange  parts of the spectra quieter than ~dynamicRange dB are not compared
maxFreq  parts of the spectra above maxFreq Hz are not compared
summary  if TRUE, returns the mean of similarity values calculated by all methods in method

Examples

## Not run:

```r
target = soundgen(syllLen = 500, formants = 'a',
   pitch = data.frame(time = c(0, 0.1, 0.9, 1),
      value = c(100, 150, 135, 100)),
   temperature = 0.001)
targetSpec = soundgen:::getMelSpec(target, samplingRate = 16000)
parsToTry = list(
   list(formants = 'i',
      pitch = data.frame(time = c(0, 1),
         value = c(200, 300))), # wrong
   list(formants = 'i',
      pitch = data.frame(time = c(0, 0.1, 0.9, 1),
         value = c(100, 150, 135, 100))), # right
   list(formants = 'a',
      pitch = data.frame(time = c(0, 1)), # right
         value = c(100, 150, 135, 100)))
```

```r
target = soundgen(syllLen = 500, formants = 'a',
   pitch = data.frame(time = c(0, 0.1, 0.9, 1),
      value = c(100, 150, 135, 100)),
   temperature = 0.001)
targetSpec = soundgen:::getMelSpec(target, samplingRate = 16000)
parsToTry = list(
   list(formants = 'i',
      pitch = data.frame(time = c(0, 1),
         value = c(200, 300))), # wrong
   list(formants = 'i',
      pitch = data.frame(time = c(0, 0.1, 0.9, 1),
         value = c(100, 150, 135, 100))), # right
   list(formants = 'a',
      pitch = data.frame(time = c(0, 1)), # right
         value = c(100, 150, 135, 100)))
```
crossFade

Join two waveforms by cross-fading

description
crossFade joins two input vectors (waveforms) by cross-fading. First it truncates both input vectors, so that amp1 ends with a zero crossing and amp2 starts with a zero crossing, both on an upward portion of the soundwave. Then it cross-fades both vectors linearly with an overlap of crossLen or crossLenPoints. If the input vectors are too short for the specified length of cross-faded region, the two vectors are concatenated at zero crossings instead of cross-fading. Soundgen uses crossFade for gluing together epochs with different regimes of pitch effects (see the vignette on sound generation), but it can also be useful for joining two separately generated sounds without audible artifacts.
crossFade(amp1, amp2, crossLenPoints = 240, crossLen = NULL, samplingRate = NULL, shape = c("lin", "exp", "log", "cos", "logistic")[1], steepness = 1)

Arguments

amp1, amp2 two numeric vectors (waveforms) to be joined
crossLenPoints (optional) the length of overlap in points
crossLen the length of overlap in ms (overrides crossLenPoints)
samplingRate the sampling rate of input vectors, Hz (needed only if crossLen is given in ms rather than points)
shape controls the type of fade function: 'lin' = linear, 'exp' = exponential, 'log' = logarithmic, 'cos' = cosine, 'logistic' = logistic S-curve
steepness scaling factor regulating the steepness of fading curves if the shape is 'exp', 'log', or 'logistic' (0 = linear, >1 = steeper than default)

Value

Returns a numeric vector.

Examples

sound1 = sin(1:100 / 9)
sound2 = sin(7:107 / 3)
plot(c(sound1, sound2), type = 'b')
# an ugly discontinuity at 100 that will make an audible click

sound = crossFade(sound1, sound2, crossLenPoints = 5)
plot(sound, type = 'b') # a nice, smooth transition
length(sound) # but note that cross-fading costs us ~60 points
# because of trimming to zero crossings and then overlapping

## Not run:
# Actual sounds, alternative shapes of fade-in/out
sound3 = soundgen(formants = 'a', pitch = 200, addSilence = 0, attackLen = c(50, 0))
sound4 = soundgen(formants = 'u', pitch = 200, addSilence = 0, attackLen = c(0, 50))

# simple concatenation (with a click)
playme(c(sound3, sound4), 16000)

# concatenation from zc to zc (no click, but a rough transition)
playme(crossFade(sound3, sound4, crossLen = 0), 16000)

# linear crossFade over 35 ms - brief, but smooth
playme(crossFade(sound3, sound4, crossLen = 35, samplingRate = 16000), 16000)
Shiny app defaults

Description
A list of default values for Shiny app soundgen_app() - mostly the same as the defaults for soundgen(). NB: if defaults change, this has to be updated!!!

Usage
defaults
defaults

Format
An object of class list of length 67.

estimateVTL Estimate vocal tract length

Description
Estimates the length of vocal tract based on formant frequencies, assuming that the vocal tract can be modeled as a tube open at both ends. Algorithm: first formant dispersion is estimated using the regression method described in Reby et al. (2005) "Red deer stags use formants as assessment cues during intrasexual agonistic interactions". The length of vocal tract is then calculated as \( \frac{\text{speed of sound}}{2} \times \text{formant dispersion} \). See also schwa for VTL estimation with additional information on formant frequencies.

Usage
estimateVTL(formants, speedSound = 35400, checkFormat = TRUE)

Arguments
formants a character string like "aaui" referring to default presets for speaker "M1"; a vector of formant frequencies; or a list of formant times, frequencies, amplitudes, and bandwidths, with a single value of each for static or multiple values of each for moving formants.
speedSound speed of sound in warm air, cm/s. Stevens (2000) "Acoustic phonetics", p. 138
checkFormat if TRUE, expands shorthand format specifications into the canonical form of a list with four components: time, frequency, amplitude and bandwidth for each format (as returned by the internal function reformatFormants)
Value

Returns the estimated vocal tract length in cm.

Examples

```r
estimateVTL(NA)
estimateVTL(500)
estimateVTL(c(600, 1850, 3100))
estimateVTL(formants = list(f1 = 600, f2 = 1650, f3 = 2400))
```

# for moving formants, frequencies are averaged over time,
# i.e. this is identical to the previous example
```r
estimateVTL(formants = list(f1 = c(500, 700), f2 = 1650, f3 = c(2200, 2600)))
```

---

## fade

**Fade**

### Description

Applies fade-in and/or fade-out of variable length, shape, and steepness. The resulting effect softens the attack and release of a waveform.

### Usage

```r
fade(x, fadeIn = 1000, fadeOut = 1000, samplingRate = NULL,
     shape = c("lin", "exp", "log", "cos", "logistic")[1], steepness = 1,
     plot = FALSE)
```

### Arguments

- `x` zero-centered (!) numeric vector such as a waveform
- `fadeIn, fadeOut` length of segments for fading in and out, interpreted as points if `samplingRate = NULL` and as ms otherwise (0 = no fade)
- `samplingRate` sampling rate of the input vector, Hz
- `shape` controls the type of fade function: 'lin' = linear, 'exp' = exponential, 'log' = logarithmic, 'cos' = cosine, 'logistic' = logistic S-curve
- `steepness` scaling factor regulating the steepness of fading curves if the shape is 'exp', 'log', or 'logistic' (0 = linear, >1 = steeper than default)
- `plot` if TRUE, produces an oscillogram of the waveform after fading

### Value

Returns a numeric vector of the same length as input
Examples

# Fading a real sound: say we want fast attack and slow release
s = soundgen(attack = 0, windowLength = 10,
        syllen = 500, addSilence = 0)
# playme(s)
# plot(s, type = 'l')
s1 = fade(s, fadeIn = 10, fadeOut = 350,
         samplingRate = 16000, shape = 'cos')
# playme(s1)
# plot(s1, type = 'l')

# Illustration of fade shapes
x = runif(5000, min = -1, max = 1)  # make sure to zero-center input!!!
# plot(x, type = 'l')
y = fade(x, fadeIn = 1000, fadeOut = 0, plot = TRUE)
y = fade(x,
        fadeIn = 1000,
        fadeOut = 1500,
        shape = 'exp',
        plot = TRUE)
y = fade(x,
        fadeIn = 1500,
        fadeOut = 500,
        shape = 'log',
        plot = TRUE)
y = fade(x,
        fadeIn = 1500,
        fadeOut = 500,
        shape = 'log',
        steepness = 8,
        plot = TRUE)
y = fade(x,
        fadeIn = 1000,
        fadeOut = 1500,
        shape = 'cos',
        plot = TRUE)
y = fade(x,
        fadeIn = 1500,
        fadeOut = 500,
        shape = 'logistic',
        steepness = 4,
        plot = TRUE)

Description

While the same sounds can be created with soundgen(), this facetious function produces the same effect more efficiently and with very few control parameters. With default settings, execution time
is ~ 10 ms per second of audio sampled at 16000 Hz. Principle: creates separate glottal cycles with harmonics, but no formants. See soundgen for more details.

Usage

```r
fart(glottis = c(50, 200), pitch = 65, temperature = 0.25,
    syllen = 600, rolloff = -10, samplingRate = 16000, play = FALSE,
    plot = FALSE)
```

Arguments

- **glottis**: anchors for specifying the proportion of a glottal cycle with closed glottis, % (0 = no modification, 100 = closed phase as long as open phase); numeric vector or dataframe specifying time and value (anchor format)
- **pitch**: a numeric vector of f0 values in Hz or a dataframe specifying the time (ms or 0 to 1) and value (Hz) of each anchor, hereafter "anchor format". These anchors are used to create a smooth contour of fundamental frequency f0 (pitch) within one syllable
- **temperature**: hyperparameter for regulating the amount of stochasticity in sound generation
- **syllen**: syllable length, ms (not vectorized)
- **rolloff**: rollof of harmonics in source spectrum, dB/octave (not vectorized)
- **samplingRate**: sampling frequency, Hz
- **play**: if TRUE, plays the synthesized sound. In case of errors, try setting another default player for `play`
- **plot**: if TRUE, plots the waveform

Value

Returns a normalized waveform.

Examples

```r
f = fart()
# playme(f)

## Not run:
while (TRUE) {
  fart(syllen = 300, temperature = .5, play = TRUE)
  Sys.sleep(rexp(1, rate = 1))
}

## End(Not run)
```
**Description**

Flattens the amplitude envelope of a waveform. This is achieved by dividing the waveform by some function of its smoothed amplitude envelope (Hilbert, peak or root mean square).

**Usage**

```r
flatEnv(sound, windowLength = 200, samplingRate = 16000,
method = c("hil", "rms", "peak")[[1]], windowLength_points = NULL,
killDC = FALSE, dynamicRange = 80, plot = FALSE)
```

**Arguments**

- **sound**: input vector oscillating about zero
- **windowLength**: the length of smoothing window, ms
- **samplingRate**: the sampling rate, Hz. Only needed if the length of smoothing window is specified in ms rather than points
- **method**: 'hil' for Hilbert envelope, 'rms' for root mean square amplitude, 'peak' for peak amplitude per window
- **windowLength_points**: the length of smoothing window, points. If specified, overrides both `windowLength` and `samplingRate`
- **killDC**: if TRUE, dynamically removes DC offset or similar deviations of average waveform from zero
- **dynamicRange**: parts of sound quieter than ~`dynamicRange` dB will not be amplified
- **plot**: if TRUE, plots the original sound, smoothed envelope, and flattened sound

**Examples**

```r
a = rnorm(500) * seq(1, 0, length.out = 500)
b = flatEnv(a, plot = TRUE, windowLength_points = 5)  # too short
c = flatEnv(a, plot = TRUE, windowLength_points = 250)  # too long
d = flatEnv(a, plot = TRUE, windowLength_points = 50)  # about right

# Not run:
s = soundgen(syllen = 1000, ampl = c(0, -40, 0), plot = TRUE, osc = TRUE)
# playme(s)
s_flat1 = flatEnv(s, plot = TRUE, windowLength = 50, method = 'hil')
s_flat2 = flatEnv(s, plot = TRUE, windowLength = 10, method = 'rms')
# playme(s_flat2)

# Remove DC offset
s1 = c(rep(0, 50), runif(1000, -1, 1), rep(0, 50)) +
```


gaussianSmooth2D

Gaussian smoothing in 2D

Description

Takes a matrix of numeric values and smoothes it by convolution with a symmetric Gaussian window function.

Usage

gaussianSmooth2D(m, kernelSize = 5, kernelSD = 0.5, plotKernel = FALSE)

Arguments

m input matrix (numeric, on any scale, doesn’t have to be square)
kernelSize the size of the Gaussian kernel, in points
kernelSD the SD of the Gaussian kernel relative to its size (.5 = the edge is two SD’s away)
plotKernel if TRUE, plots the kernel

Value

Returns a numeric matrix of the same dimensions as input.

Examples

s = spectrogram(soundgen(), samplingRate = 16000,
    output = 'original', plot = FALSE)
# image(log(s))
s1 = gaussianSmooth2D(s, kernelSize = 11, plotKernel = TRUE)
# image(log(s1))

seq(.3, 1, length.out = 1100)
s2 = flatEnv(s1, plot = TRUE, windowLength_points = 50, killDC = FALSE)
s3 = flatEnv(s1, plot = TRUE, windowLength_points = 50, killDC = TRUE)

## End(Not run)
generateNoise

**Description**

Generates noise of length `len` and with spectrum defined by linear decay of `rolloffNoise` dB/kHz above `noiseflatSpec` Hz OR by a specified filter `spectralEnvelope`. This function is called internally by `soundgen`, but it may be more convenient to call it directly when synthesizing non-biological noises defined by specific spectral and amplitude envelopes rather than formants: the wind, whistles, impact noises, etc. See `fart` and `beat` for similarly simplified functions for tonal non-biological sounds.

**Usage**

```r
generateNoise(len, rolloffNoise = 0, noiseflatSpec = 1200,
              spectralEnvelope = NULL, noise = NULL, temperature = 0.1,
              attackLen = 10, windowLength_points = 1024, samplingRate = 16000,
              overlap = 75, dynamicRange = 80, play = FALSE)
```

**Arguments**

- `len` length of output
- `rolloffNoise` linear rolloff of the excitation source for the unvoiced component, dB/kHz (anchor format)
- `noiseflatSpec` keeps noise spectrum flat to this frequency, Hz
- `spectralEnvelope` (optional): as an alternative to using `rolloffNoise`, we can provide the exact filter - a vector of non-negative numbers specifying the power in each frequency bin on a linear scale (interpolated to length equal to `windowLength_points/2`). A matrix specifying the filter for each STFT step is also accepted. The easiest way to create this matrix is to call `soundgen::getSpectralEnvelope` or to use the spectrum of a recorded sound
- `noise` loudness of turbulent noise (0 dB = as loud as voiced component, negative values = quieter) such as aspiration, hissing, etc (anchor format)
- `temperature` hyperparameter for regulating the amount of stochasticity in sound generation
- `attackLen` duration of fade-in / fade-out at each end of syllables and noise (ms): a vector of length 1 (symmetric) or 2 (separately for fade-in and fade-out)
- `windowLength_points` the length of fft window, points
- `samplingRate` sampling frequency, Hz
- `overlap` FFT window overlap, %. For allowed values, see `istft`
- `dynamicRange` dynamic range, dB. Harmonics and noise more than `dynamicRange` under maximum amplitude are discarded to save computational resources
- `play` if TRUE, plays the synthesized sound. In case of errors, try setting another default player for `play`
Details

Algorithm: paints a spectrogram with desired characteristics, sets phase to zero, and generates a time sequence via inverse FFT.

Examples

# .5 s of white noise
samplingRate = 16000
noise1 = generateNoise(len = samplingRate * .5, samplingRate = samplingRate)
# playme(noise1, samplingRate)
# seeWave::meanspec(noise1, f = samplingRate)

# Percussion (run a few times to notice stochasticity due to temperature = .25)
noise2 = generateNoise(len = samplingRate * .15, noise = c(0, -80),
rolloffNoise = c(4, -6), attackLen = 5, temperature = .25)
noise3 = generateNoise(len = samplingRate * .25, noise = c(0, -40),
rolloffNoise = c(4, -20), attackLen = 5, temperature = .25)
# playme(c(noise2, noise3), samplingRate)

## Not run:
playback = c(TRUE, FALSE)[2]
# 1.2 s of noise with rolloff changing from 0 to -12 dB above 2 kHz
noise = generateNoise(len = samplingRate * 1.2,
rolloffNoise = c(0, -12), noiseFlatSpec = 2000,
samplingRate = samplingRate, play = playback)
# spectrogram(noise, samplingRate, osc = TRUE)

# Similar, but using the dataframe format to specify a more complicated
# contour for rolloffNoise:
noise = generateNoise(len = samplingRate * 1.2,
rolloffNoise = data.frame(time = c(0, .3, 1), value = c(-12, 0, -12)),
noiseFlatSpec = 2000, samplingRate = samplingRate, play = playback)
# spectrogram(noise, samplingRate, osc = TRUE)

# To create a sibilant [s], specify a single strong, broad formant at ~7 kHz:
windowLength_points = 1024
spectralEnvelope = soundgen::getSpectralEnvelope(
    nr = windowLength_points / 2, nc = 1, samplingRate = samplingRate,
    formants = list('f1' = data.frame(time = 0, freq = 7000,
    amp = 50, width = 2000)))
noise = generateNoise(len = samplingRate,
samplingRate = samplingRate, spectralEnvelope = as.numeric(spectralEnvelope),
play = playback)
# plot(spectralEnvelope, type = 'l')

# Low-frequency, wind-like noise
spectralEnvelope = soundgen::getSpectralEnvelope(
    nr = windowLength_points / 2, nc = 1, lipRad = 0,
samplingRate = samplingRate, formants = list('f1' = data.frame(
    time = 0, freq = 150, amp = 30, width = 90)))
noise = generateNoise(len = samplingRate,
samplingRate = samplingRate, spectralEnvelope = as.numeric(spectralEnvelope),
play = playback)

# Manual filter, e.g. for a kettle-like whistle (narrow-band noise)
spectralEnvelope = c(rep(0, 100), 120, rep(0, 100)) # any length is fine
# plot(spectralEnvelope, type = 'b') # notch filter at Nyquist / 2, here 4 kHz
noise = generateNoise(len = samplingRate, spectralEnvelope = spectralEnvelope,
samplingRate = samplingRate, play = playback)

# Compare to a similar sound created with soundgen()
# (unvoiced only, a single formant at 4 kHz)
noise_s = soundgen(pitch = NULL,
noise = data.frame(time = c(0, 1000), value = c(0, 0)),
formants = list(f1 = data.frame(freq = 4000, amp = 80, width = 20)),
play = playback)

# Use the spectral envelope of an existing recording (bleating of a sheep)
# (see also the same example with tonal source in addformants)
data(sheep, package = 'seewave') # import a recording from seewave
sound_orig = as.numeric(sheep@left)
samplingRate = sheep@smpRate
# playme(sound_orig, samplingRate)

# extract the original spectrogram
windowLength = c(5, 10, 50, 100)[1] # try both narrow-band (eg 100 ms)
# to get "harmonics" and wide-band (5 ms) to get only formants
spectralEnvelope = spectrogram(sound_orig, windowLength = windowLength,
samplingRate = samplingRate, output = 'original')
sound_noise = generateNoise(len = length(sound_orig),
spectralEnvelope = spectralEnvelope, rolloffNoise = 0,
samplingRate = samplingRate, play = playback)
# playme(sound_noise, samplingRate)

# The spectral envelope is similar to the original recording. Compare:
par(mfrow = c(1, 2))
seewave::meanspec(sound_orig, f = samplingRate, dB = 'max0')
seewave::meanspec(sound_noise, f = samplingRate, dB = 'max0')
par(mfrow = c(1, 1))
# However, the excitation source is now white noise
# (which sounds like noise if windowLength is ~5-10 ms,
# but becomes more and more like the original at longer window lengths)

## End(Not run)
**getIntegerRandomWalk**

*Discrete random walk*

**Description**

Takes a continuous random walk and converts it to continuous epochs of repeated values 0/1/2, each at least minLength points long. 0/1/2 correspond to different noise regimes: 0 = no noise, 1 = subharmonics, 2 = subharmonics and jitter/shimmer.
getLoudness

Usage

getIntegerRandomWalk(rw, nonlinBalance = 50, minLength = 50,
q1 = NULL, q2 = NULL, plot = FALSE)

Arguments

- **rw**: a random walk generated by `getRandomWalk` (expected range 0 to 100)
- **nonlinBalance**: a number between 0 to 100: 0 = returns all zeros; 100 = returns all twos
- **minLength**: the minimum length of each epoch
- **q1, q2**: cutoff points for transitioning from regime 0 to 1 (q1) or from regime 1 to 2 (q2). See `noiseThresholdsDict` for defaults
- **plot**: if TRUE, plots the random walk underlying nonlinear regimes

Value

Returns a vector of integers (0/1/2) of the same length as rw.

Examples

```r
rw = getRandomWalk(len = 100, rw_range = 100, rw_smoothing = .2)
r = getIntegerRandomWalk(rw, nonlinBalance = 75,
minLength = 10, plot = TRUE)
r = getIntegerRandomWalk(rw, nonlinBalance = 15,
q1 = 30, q2 = 70,
minLength = 10, plot = TRUE)
```

getLoudness

*Get loudness*

Description

Estimates subjective loudness per frame, in sone. Based on EMBSD speech quality measure, particularly the matlab code in Yang (1999) and Timoney et al. (2004). Note that there are many ways to estimate loudness and many other factors, ignored by this model, that could influence subjectively experienced loudness. Please treat the output with a healthy dose of skepticism! Also note that the absolute value of calculated loudness critically depends on the chosen "measured" sound pressure level (SPL). `getLoudness` estimates how loud a sound will be experienced if it is played back at an SPL of `SPL_measured` dB. The most meaningful way to use the output is to compare the loudness of several sounds analyzed with identical settings or of different segments within the same recording.

Usage

```r
getLoudness(x, samplingRate = NULL, scale = NULL, windowLength = 50,
step = NULL, overlap = 50, SPL_measured = 70, Pref = 2e-05,
spreadSpectrum = TRUE, plot = TRUE, mar = c(5.1, 4.1, 4.1, 4.1),
...)
```
getLoudness

Arguments

- **x**: path to a .wav or .mp3 file or a vector of amplitudes with specified samplingRate
- **samplingRate**: sampling rate of x (only needed if x is a numeric vector, rather than an audio file), must be > 2000 Hz
- **scale**: the maximum possible value of x (only needed if x is a numeric vector, rather than an audio file); defaults to observed \(\max(|x|)\) if it is greater than 1 and to 1 otherwise
- **windowLength**: length of FFT window, ms
- **step**: you can override overlap by specifying FFT step, ms
- **overlap**: overlap between successive FFT frames, %
- **SPL_measured**: sound pressure level at which the sound is presented, dB
- **Pref**: reference pressure, Pa
- **spreadSpectrum**: if TRUE, applies a spreading function to account for frequency masking
- **plot**: should a spectrogram be plotted? TRUE / FALSE
- **mar**: margins of the spectrogram
- **...**: other plotting parameters passed to `spectrogram`

Details

Algorithm: calibrates the sound to the desired SPL (Timoney et al., 2004), extracts a spectrogram, converts to bark scale (`audspec`), spreads the spectrum to account for frequency masking across the critical bands (Yang, 1999), converts dB to phon by using standard equal loudness curves (ISO 226), converts phon to sone (Timoney et al., 2004), sums across all critical bands, and applies a correction coefficient to standardize output. Calibrated so as to return a loudness of 1 sone for a 1 kHz pure tone with SPL of 40 dB.

Value

Returns a list of length two:

- **specSone**: spectrum in sone: a matrix with frequency on the bark scale in rows and time (STFT frames) in columns
- **loudness**: a vector of loudness per STFT frame (sone)

References

getLoudnessFolder

Examples

```
sounds = list(
    white_noise = runif(8000, -1, 1),
    white_noise2 = runif(8000, -1, 1) / 2,  # ~6 dB quieter
    pure_tone_1KHz = sin(2*pi*1000/16000*(1:8000))  # pure tone at 1 kHz
)

loud = rep(0, length(sounds)); names(loud) = names(sounds)
for (i in 1:length(sounds)) {
    # playme(sounds[[i]], 16000)
    l = getLoudness(
        x = sounds[[i]], samplingRate = 16000, scale = 1,
        windowLength = 20, step = NULL,
        overlap = 50, SPL_measured = 40,
        Pref = 2e-5, plot = FALSE)
    loud[i] = mean(l$loudness)
}

# white noise (sound 1) is twice as loud as pure tone at 1 KHz (sound 3),
# and note that the same white noise with lower amplitude has lower loudness
# (provided that "scale" is specified)
# compare: lapply(sounds, range)

## Not run:
s = soundgen()
l = getLoudness(s, SPL_measured = 70,
    samplingRate = 16000, plot = TRUE, osc = TRUE)

# The estimated loudness in sone depends on target SPL
l = getLoudness(s, SPL_measured = 40,
    samplingRate = 16000, plot = TRUE)

# ...but not (much) on windowLength and samplingRate
l = getLoudness(soundgen(), SPL_measured = 40, windowLength = 50,
    samplingRate = 16000, plot = TRUE)

# input can be an audio file
getLoudness(~Downloads/temp/032_ut_anger_30-m-roar-curse.wav)
```

## End(Not run)

getLoudnessFolder  Loudness per folder

Description

A wrapper around `getLoudness` that goes through all wav/mp3 files in a folder and returns either a list with loudness values per STFT frame from each file or, if `summary = TRUE`, a dataframe with a single summary value of loudness per file. This summary value can be mean, max and so on, as per `summaryFun`. 
Usage

```r
getLoudnessFolder(myfolder, windowLength = 50, step = NULL,
                  overlap = 50, SPL_measured = 70, Prep = 2e-05,
                  spreadSpectrum = TRUE, summary = TRUE, summaryFun = "mean",
                  verbose = TRUE)
```

Arguments

- **myfolder**: path to folder containing wav/mp3 files
- **windowLength**: length of FFT window, ms
- **step**: you can override overlap by specifying FFT step, ms
- **overlap**: overlap between successive FFT frames, %
- **SPL_measured**: sound pressure level at which the sound is presented, dB
- **Prep**: reference pressure, Pa
- **spreadSpectrum**: if TRUE, applies a spreading function to account for frequency masking
- **summary**: if TRUE, returns only a single value of loudness per file
- **summaryFun**: the function used to summarize loudness values across all STFT frames (if summary = TRUE)
- **verbose**: if TRUE, reports estimated time left

Examples

```r
## Not run:
getLoudnessFolder("~/Downloads/temp")
# Compare:
analyzeFolder("~/Downloads/temp", pitchMethods = NULL,
             plot = FALSE)$loudness_mean
# (per STFT frame; should be very similar, but not identical, because
# analyze() discards frames considered silent or too noisy)

getLoudnessFolder("~/Downloads/temp", summaryFun = function(x) diff(range(x)))

# save loudness values per frame without summarizing
l = getLoudnessFolder("~/Downloads/temp", summary = FALSE)

## End(Not run)
```

getRandomWalk

**Random walk**

Description

Generates a random walk with flexible control over its range, trend, and smoothness. It works by calling `rnorm` at each step and taking a cumulative sum of the generated values. Smoothness is controlled by initially generating a shorter random walk and upsampling.
Usage

g getRandomWalk(len, rw_range = 1, rw_smoothing = 0.2,
   method = c("linear", "spline")[2], trend = 0)

Arguments

len an integer specifying the required length of random walk. If len is 1, returns a
   single draw from a gamma distribution with mean=1 and sd=rw_range
rw_range the upper bound of the generated random walk (the lower bound is set to 0)
rw_smoothing specifies the amount of smoothing, from 0 (no smoothing) to 1 (maximum
   smoothing to a straight line)
method specifies the method of smoothing: either linear interpolation ('linear', see approx)
   or cubic splines ('spline', see spline)
trend mean of generated normal distribution (vectors are also acceptable, as long as
   their length is an integer multiple of len). If positive, the random walk has an
   overall upwards trend (good values are between 0 and 0.5 or -0.5). Trend =
   c(1,-1) gives a roughly bell-shaped rw with an upward and a downward curve.
   Larger absolute values of trend produce less and less random behavior

Value

Returns a numeric vector of length len and range from 0 to rw_range.

Examples

plot(getRandomWalk(len = 1000, rw_range = 5, rw_smoothing = .2))
plot(getRandomWalk(len = 1000, rw_range = 5, rw_smoothing = .5))
plot(getRandomWalk(len = 1000, rw_range = 15,
   rw_smoothing = .2, trend = c(.5, -.5)))
plot(getRandomWalk(len = 1000, rw_range = 15,
   rw_smoothing = .2, trend = c(15, -1)))

getRMS

RMS amplitude per frame

Description

Calculates root mean square (RMS) amplitude in overlapping frames, providing an envelope of
RMS amplitude as a measure of sound intensity. Longer windows provide smoother, more robust
estimates; shorter windows and more overlap improve temporal resolution, but they also increase
processing time and make the contour less smooth.

Usage

g getRMS(x, samplingRate = NULL, windowLength = 50, step = NULL,
   overlap = 75, killDC = FALSE, scale = NULL, normalize = TRUE,
   windowDC = 200, plot = TRUE, xlab = "Time, ms", ylab = "",
   type = "b", col = "blue", lwd = 2, ...)
getRMS

Arguments

- **x**: path to a .wav or .mp3 file or a vector of amplitudes with specified samplingRate
- **samplingRate**: sampling rate of x (only needed if x is a numeric vector, rather than an audio file)
- **windowLength**: length of FFT window, ms
- **step**: you can override overlap by specifying FFT step, ms
- **overlap**: overlap between successive FFT frames, %
- **killDC**: if TRUE, removed DC offset (see also flatEnv)
- **scale**: maximum possible amplitude of input used for normalization (not needed for audio files)
- **normalize**: if TRUE, RMS amplitude is normalized to [0, 1]
- **windowDC**: the window for calculating DC offset, ms
- **plot**: should a spectrogram be plotted? TRUE / FALSE
- **xlab, ylab**: general graphical parameters
- **type, col, lwd**: graphical parameters pertaining to the RMS envelope
- **...**: other graphical parameters

Details

Note that you can also get similar estimates per frame from analyze on a normalized scale of 0 to 1, but getRMS is much faster, operates on the original scale, and plots the amplitude contour. If you need RMS for the entire sound instead of per frame, you can simply calculate it as \( \sqrt{\text{mean}(x^2)} \), where \( x \) is your waveform. Having RMS estimates per frame gives more flexibility: RMS per sound can be calculated as the mean / median / max of RMS values per frame.

Value

Returns a numeric vector of RMS amplitudes per frame on the scale of input. Names give time stamps for the center of each frame, in ms.

Examples

```r
s = soundgen() + .1  # with added DC offset
plot(s, type = 'l')

r = getRMS(s, samplingRate = 16000,
          windowLength = 40, overlap = 50, killDC = TRUE,
          col = 'green', lty = 2, main = 'RMS envelope')
# short window = jagged envelope

r = getRMS(s, samplingRate = 16000,
          windowLength = 5, overlap = 0, killDC = TRUE,
          col = 'green', lty = 2, main = 'RMS envelope')

## Not run:

r = getRMS(~Downloads/temp/032_ut_anger_30-m-roar-curse.wav')

## End(Not run)
```
getRMSFolder

RMS amplitude per folder

Description

A wrapper around getRMS that goes through all wav/mp3 files in a folder and returns either a list with RMS values per frame from each file or, if summary = TRUE, a dataframe with a single summary value of RMS per file. This summary value can be mean, max and so on, as per summaryFun.

Usage

getRMSFolder(myfolder, windowLength = 50, step = NULL, overlap = 70, normalize = TRUE, killDC = FALSE, windowDC = 200, summary = TRUE, summaryFun = "mean", verbose = TRUE)

Arguments

- **myfolder**: path to folder containing wav/mp3 files
- **windowLength**: length of FFT window, ms
- **step**: you can override overlap by specifying FFT step, ms
- **overlap**: overlap between successive FFT frames, %
- **normalize**: if TRUE, RMS amplitude is normalized to [0, 1]
- **killDC**: if TRUE, removed DC offset (see also flatEnv)
- **windowDC**: the window for calculating DC offset, ms
- **summary**: if TRUE, returns only a single value of RMS per file
- **summaryFun**: the function used to summarize RMS values across all frames (if summary = TRUE)
- **verbose**: if TRUE, reports estimated time left

Examples

```r
## Not run:
getRMSFolder(’~/Downloads/temp’)
# Compare:
analyzeFolder(’~/Downloads/temp’, pitchMethods = NULL, plot = FALSE)$ampl_mean
# (per STFT frame, but should be very similar)

User-defined summary functions:
getRMSFolder(’~/Downloads/temp’, summaryFun = function(x) diff(range(x)))
getRMSFolder(’~/Downloads/temp’,
  summaryFun = function(x) paste0(‘mean = ‘, round(mean(x), 2), ‘; sd = ‘, round(sd(x), 2)))

## End(Not run)
```
**getRolloff**

**Control rolloff of harmonics**

**Description**

Harmonics are generated as separate sine waves. But we don’t want each harmonic to be equally strong, so we normally specify some rolloff function that describes the loss of energy in upper harmonics relative to the fundamental frequency (f0). *getRolloff* provides flexible control over this rolloff function, going beyond simple exponential decay (rolloff). Use quadratic terms to modify the behavior of a few lower harmonics, rolloffOct to adjust the rate of decay per octave, and rolloffKHz for rolloff correction depending on f0. Plot the output with different parameter values and see examples below and the vignette to get a feel for how to use *getRolloff* effectively.

**Usage**

```
getRolloff(pitch_per_gc = c(440), nHarmonics = NULL, rolloff = -6, rolloffOct = 0, rolloffParab = 0, rolloffParabHarm = 3, rolloffParabCeiling = NULL, rolloffKHz = 0, baseline = 200, dynamicRange = 80, samplingRate = 16000, plot = FALSE)
```

**Arguments**

- **pitch_per_gc** a vector of f0 per glottal cycle, Hz
- **nHarmonics** maximum number of harmonics to generate (very weak harmonics with amplitude < -dynamicRange will be discarded)
- **rolloff** basic rolloff from lower to upper harmonics, dB/octave (exponential decay). All rolloff parameters are in anchor format. See *getRolloff* for more details
- **rolloffOct** basic rolloff changes from lower to upper harmonics (regardless of f0) by rolloffOct dB/oct. For example, we can get steeper rolloff in the upper part of the spectrum
- **rolloffParab** an optional quadratic term affecting only the first rolloffParabHarm harmonics. The middle harmonic of the first rolloffParabHarm harmonics is amplified or dampened by rolloffParab dB relative to the basic exponential decay
- **rolloffParabHarm** the number of harmonics affected by rolloffParab
- **rolloffParabCeiling** quadratic adjustment is applied only up to rolloffParabCeiling, Hz. If not NULL, it overrides rolloffParabHarm
- **rolloffKHz** rolloff changes linearly with f0 by rolloffKHz dB/kHz. For ex., -6 dB/kHz gives a 6 dB steeper basic rolloff as f0 goes up by 1000 Hz
- **baseline** The "neutral" f0, at which no adjustment of rolloff takes place regardless of rolloffKHz
- **dynamicRange** dynamic range, dB. Harmonics and noise more than dynamicRange under maximum amplitude are discarded to save computational resources
- **samplingRate** sampling rate (needed to stop at Nyquist frequency and for plotting purposes)
- **plot** if TRUE, produces a plot
getRolloff

**Value**

Returns a matrix of amplitude multiplication factors for adjusting the amplitude of harmonics relative to f0 (1 = no adjustment, 0 = silent). Each row of output contains one harmonic, and each column contains one glottal cycle.

**Examples**

```r
# steady exponential rolloff of -12 dB per octave
rolloff = getRolloff(pitch_per_gc = 150, rolloff = -12,
                    rolloffOct = 0, rolloffKHz = 0, plot = TRUE)

# the rate of rolloff slows down by 1 dB each octave
rolloff = getRolloff(pitch_per_gc = 150, rolloff = -12,
                    rolloffOct = 1, rolloffKHz = 0, plot = TRUE)

# rolloff can be made to depend on f0 using rolloffKHz
rolloff = getRolloff(pitch_per_gc = c(150, 400, 800),
                    rolloffOct = 0, rolloffKHz = -3, plot = TRUE)

# without the correction for f0 (rolloffKHz),
# high-pitched sounds have the same rolloff as low-pitched sounds,
# producing unnaturally strong high-frequency harmonics
rolloff = getRolloff(pitch_per_gc = c(150, 400, 800),
                    rolloffOct = 0, rolloffKHz = 0, plot = TRUE)

# parabolic adjustment of lower harmonics
rolloff = getRolloff(pitch_per_gc = 350, rolloffParab = 0,
                    rolloffParabHarm = 2, plot = TRUE)

# rolloffParabHarm = 1 affects only f0
rolloff = getRolloff(pitch_per_gc = 150, rolloffParab = 30,
                    rolloffParabHarm = 1, plot = TRUE)

# rolloffParabHarm = 2 or 3 affects only h1
rolloff = getRolloff(pitch_per_gc = 150, rolloffParab = 30,
                    rolloffParabHarm = 2, plot = TRUE)

# rolloffParabHarm = 4 affects h1 and h2, etc
rolloff = getRolloff(pitch_per_gc = 150, rolloffParab = 30,
                    rolloffParabHarm = 4, plot = TRUE)

# negative rolloffParab weakens lower harmonics
rolloff = getRolloff(pitch_per_gc = 150, rolloffParab = -20,
                    rolloffParabHarm = 7, plot = TRUE)

# only harmonics below 2000 Hz are affected
rolloff = getRolloff(pitch_per_gc = c(150, 600),
                    rolloffParab = -20, rolloffParabCeiling = 2000,
                    plot = TRUE)

# dynamic rolloff (varies over time)
rolloff = getRolloff(pitch_per_gc = c(150, 250),
                    rolloff = c(-12, -18, -24), plot = TRUE)
rolloff = getRolloff(pitch_per_gc = c(150, 250), rolloffParab = 40,
                    rolloffParabHarm = 1:5, plot = TRUE)

## Not run:
# Note: getRolloff() is called internally by soundgen()
# using the data.frame format for all vectorized parameters
getSmoothContour

Smooth contour from anchors

Description

Returns a smooth contour based on an arbitrary number of anchors. Used by soundgen for generating intonation contour, mouth opening, etc. Note that pitch contours are treated as a special case: values are log-transformed prior to smoothing, so that with 2 anchors we get a linear transition on a log scale (as if we were operating with musical notes rather than frequencies in Hz). Pitch plots have two Y axes: one showing Hz and the other showing musical notation.

Usage

getSmoothContour(anchors = data.frame(time = c(0, 1), value = c(0, 1)), len = NULL, thisIsPitch = FALSE, normalizeTime = TRUE, interpol = c("approx", "spline", "loess")[3], discontThres = 0.05, jumpThres = 0.01, valueFloor = NULL, valueCeiling = NULL, plot = FALSE, main = "", xlim = NULL, ylim = NULL, samplingRate = 16000, voiced = NULL, contourLabel = NULL, ...)
getSmoothContour

interpol

the method of smoothing envelopes based on provided anchors: 'approx' = linear interpolation, 'spline' = cubic spline, 'loess' (default) = polynomial local smoothing function. NB: this does not affect contours for "noise", "glottal", and the smoothing of formants

discontThres

if two anchors are closer in time than discontThres, the contour is broken into segments with a linear transition between these anchors; if anchors are closer than jumpThres, a new section starts with no transition at all (e.g. for adding pitch jumps)

jumpThres

if two anchors are closer in time than discontThres, the contour is broken into segments with a linear transition between these anchors; if anchors are closer than jumpThres, a new section starts with no transition at all (e.g. for adding pitch jumps)

valueFloor, valueCeiling

lower/upper bounds for the contour

plot

(boolean) produce a plot?

main, xlim, ylim

plotting options

samplingRate

sampling rate used to convert time values to points (Hz)

voiced, contourLabel

graphical pars for plotting breathing contours (see examples below)

... other plotting options passed to plot()

Value

Returns a numeric vector.

Examples

# long format: anchors are a dataframe
a = getSmoothContour(anchors = data.frame(
  time = c(50, 137, 300), value = c(0.03, 0.78, 0.5)),
  normalizeTime = FALSE,
  voiced = 200, valueFloor = 0, plot = TRUE, main = '',
  samplingRate = 10000) # breathing

# short format: anchors are a vector (equal time steps assumed)
a = getSmoothContour(anchors = c(350, 800, 600),
  len = 5500, thisIsPitch = TRUE, plot = TRUE,
  samplingRate = 3500) # pitch

# a single anchor gives constant value
a = getSmoothContour(anchors = 800,
  len = 500, thisIsPitch = TRUE, plot = TRUE,
  samplingRate = 500)

# two pitch anchors give loglinear F0 change
a = getSmoothContour(anchors = c(220, 440),
  len = 500, thisIsPitch = TRUE, plot = TRUE,
  samplingRate = 500)

## Two closely spaced anchors produce a pitch jump
# one loess for the entire contour
a1 = getSmoothContour(anchors = list(time = c(0, .15, .2, .7, 1),
value = c(360, 116, 550, 700, 610)), len = 500, thisisPitch = TRUE,
plot = TRUE, samplingRate = 500)
# two segments with a linear transition
a2 = getSmoothContour(anchors = list(time = c(0, .15, .17, .7, 1),
value = c(360, 116, 550, 700, 610)), len = 500, thisisPitch = TRUE,
plot = TRUE, samplingRate = 500)
# two segments with an abrupt jump
a3 = getSmoothContour(anchors = list(time = c(0, .15, .155, .7, 1),
value = c(360, 116, 550, 700, 610)), len = 500, thisisPitch = TRUE,
plot = TRUE, samplingRate = 500)

getSpectralEnvelope

Spectral envelope

Description

Prepares a spectral envelope for filtering a sound to add formants, lip radiation, and some stochastic component regulated by temperature. Formants are specified as a list containing time, frequency, amplitude, and bandwidth values for each formant (see examples). See vignette('sound_generation', package = 'soundgen') for more information.

Usage

getSpectralEnvelope(nr, nc, formants = NA, formantDep = 1,
formantWidth = 1, lipRad = 6, noseRad = 4, mouth = NA,
interpol = c("approx", "spline", "loess")[3], mouthOpenThres = 0.2,
openMouthBoost = 0, vocalTract = NULL, temperature = 0.05,
formDrift = 0.3, formDisp = 0.2, formantDepStoch = 20,
smoothLinearFactor = 1, samplingRate = 16000, speedSound = 35400,
plot = FALSE, duration = NULL, colorTheme = c("bw", "seewave",
"...")[1], nCols = 100, xlab = "Time", ylab = "Frequency, kHz",
...)

Arguments

nr      the number of frequency bins = windowLength_points/2, where windowLength_points is the size of window for Fourier transform
nc      the number of time steps for Fourier transform
formants a character string like "auii" referring to default presets for speaker "M1"; a vector of formant frequencies; or a list of formant times, frequencies, amplitudes, and bandwidths, with a single value of each for static or multiple values of each for moving formants. formants = NA defaults to schwa. Time stamps for formants and mouthOpening can be specified in ms or any other arbitrary scale.
**getSpectralEnvelope**

- **formantDep**: scale factor of formant amplitude (1 = no change relative to amplitudes in formants)
- **formantWidth**: scale factor of formant bandwidth (1 = no change)
- **lipRad**: the effect of lip radiation on source spectrum, dB/oct (the default of +6 dB/oct produces a high-frequency boost when the mouth is open)
- **noseRad**: the effect of radiation through the nose on source spectrum, dB/oct (the alternative to `lipRad` when the mouth is closed)
- **mouth**: mouth opening (0 to 1, 0.5 = neutral, i.e. no modification) (anchor format)
- **interpol**: the method of smoothing envelopes based on provided mouth anchors: 'approx' = linear interpolation, 'spline' = cubic spline, 'loess' (default) = polynomial local smoothing function. NB: this does NOT affect the smoothing of formant anchors
- **mouthOpenThres**: open the lips (switch from nose radiation to lip radiation) when the mouth is open > `mouthOpenThres`, 0 to 1
- **openMouthBoost**: amplify the voice when the mouth is open by `openMouthBoost` dB
- **vocalTract**: the length of vocal tract, cm. Used for calculating formant dispersion (for adding extra formants) and formant transitions as the mouth opens and closes. If NULL or NA, the length is estimated based on specified formant frequencies (if any)
- **temperature**: hyperparameter for regulating the amount of stochasticity in sound generation
- **formDrift**: scale factor regulating the effect of temperature on the depth of random drift of all formants (user-defined and stochastic): the higher, the more formants drift at a given temperature
- **formDisp**: scale factor regulating the effect of temperature on the irregularity of the dispersion of stochastic formants: the higher, the more unevenly stochastic formants are spaced at a given temperature
- **formantDepStoch**: the amplitude of additional formants added above the highest specified formant (only if temperature > 0)
- **smoothLinearFactor**: regulates smoothing of formant anchors (0 to +Inf) as they are upsampled to the number of fft steps `nc`. This is necessary because the input formants normally contains fewer sets of formant values than the number of fft steps. `smoothLinearFactor = 0`: close to default spline; >3: approaches linear extrapolation
- **samplingRate**: sampling frequency, Hz
- **speedSound**: speed of sound in warm air, cm/s. Stevens (2000) "Acoustic phonetics", p. 138
- **plot**: if TRUE, produces a plot of the spectral envelope
- **duration**: duration of the sound, ms (for plotting purposes only)
- **colorTheme**: black and white ('bw'), as in seewave package ('seewave'), or another color theme (e.g. 'heat.colors')
- **nCols**: number of colors in the palette
- **xlab, ylab**: labels of axes
- **...**: other graphical parameters passed on to `image()`
Value

Returns a spectral filter (matrix nr x nc, where nr is the number of frequency bins = windowLength_points/2 and nc is the number of time steps)

Examples

```r
def hz_to_semitones(freqs):
    semitones = hz_to_semitones(freqs)
    return semitones
```

# [a] with F1-F3 visible
```
# image(t(e)) # to plot the output on a linear scale instead of dB
```
```
e = getspectralEnvelope(nr = 512, nc = 50,
    formants = soundgen:::convertStringToFormants('a'),
    temperature = 0, plot = TRUE)
```
```
e = getspectralEnvelope(nr = 512, nc = 50,
    formants = soundgen:::convertStringToFormants('a'),
    temperature = 0.1, formantDepStoch = 20, plot = TRUE)
```
```
e = getspectralEnvelope(nr = 512, nc = 50,
    formants = soundgen:::convertStringToFormants('a'),
    temperature = 0.1, formantDepStoch = 20, plot = TRUE)
```
```
# a schwa based on the length of vocal tract = 15.5 cm
```
```
e = getspectralEnvelope(nr = 512, nc = 50, formants = NA,
    temperature = .1, vocalTract = 15.5, plot = TRUE)
```
```
# no formants at all, only lip radiation
```
```
e = getspectralEnvelope(nr = 512, nc = 50,
    formants = NA, temperature = 0, plot = TRUE)
```
```
# mouth opening
```
```
e = getspectralEnvelope(nr = 512, nc = 50,
    vocalTract = 16, plot = TRUE, lipRad = 6, noseRad = 4,
    mouth = data.frame(time = c(0, .5, 1), value = c(0, .5))))
```
```
# scale formant amplitude and/or bandwidth
```
```
e = getspectralEnvelope(nr = 512, nc = 50,
    formants = soundgen:::convertStringToFormants('a'),
    formantWidth = 2, formantDep = .5,
    temperature = 0, plot = TRUE)
```
```
# manual specification of formants
```
```
e = getspectralEnvelope(nr = 512, nc = 50, plot = TRUE, samplingRate = 16000,
    formants = list(f1 = data.frame(time = c(0, 1), freq = c(900, 500),
    amp = c(10, 50),
    width = c(80, 50)),
    f2 = data.frame(time = c(0, 1), freq = c(1200, 2500),
    amp = c(10, 50),
    width = 100),
    f3 = data.frame(time = 0, freq = 2900,
    amp = c(10, 50),
    width = 120)))
```
```
Description

Converts from Hz to semitones above C-5 (~0.5109875 Hz). This may not seem very useful, but note that this gives us a nice logarithmic scale for generating natural pitch transitions with the added benefit of getting musical notation for free from notesDict (see examples).

Usage

```r
HzToSemitones(h, ref = 0.5109875)
```

Arguments

- `h`: vector or matrix of frequencies (Hz)
- `ref`: frequency of the reference value (defaults to C-5, 0.51 Hz)

Examples

```r
s = HzToSemitones(c(440, 293, 115))
# to convert to musical notation
notesDict$note[1 + round(s)]
# note the "1 +": semitones ABOVE C-5, i.e. notesDict[1,] is C-5
```

matchPars

**Match soundgen pars (experimental)**

Description

Attempts to find settings for soundgen that will reproduce an existing sound. The principle is to mutate control parameters, trying to improve fit to target. The currently implemented optimization algorithm is simple hill climbing. Disclaimer: this function is experimental and may or may not work for particular tasks. It is intended as a supplement to - not replacement of - manual optimization. See vignette('sound_generation', package = 'soundgen') for more information.

Usage

```r
matchPars(target, samplingRate = NULL, pars = NULL, init = NULL,
 method = c("cor", "cosine", "pixel", "dtw"), probMutation = 0.25,
 stepVariance = 0.1, maxIter = 50, minExpectedDelta = 0.001,
 windowLength = 40, overlap = 50, step = NULL, verbose = TRUE,
 padWith = NA, penalizeLengthDif = TRUE, dynamicRange = 80,
 maxFreq = NULL)
```

Arguments

- `target`: the sound we want to reproduce using soundgen: path to a .wav file or numeric vector
- `samplingRate`: sampling rate of `target` (only needed if `target` is a numeric vector, rather than a .wav file)
pars arguments to `soundgen` that we are attempting to optimize

init a list of initial values for the optimized parameters `pars` and the values of other arguments to `soundgen` that are fixed at non-default values (if any)

method method of comparing mel-transformed spectra of two sounds: "cor" = average Pearson's correlation of mel-transformed spectra of individual FFT frames; "cosine" = same as "cor" but with cosine similarity instead of Pearson's correlation; "pixel" = absolute difference between each point in the two spectra; "dtw" = discrete time warp with `dtw`

probMutation the probability of a parameter mutating per iteration

stepVariance scale factor for calculating the size of mutations

maxIter maximum number of mutated sounds produced without improving the fit to target

minExpectedDelta minimum improvement in fit to target required to accept the new sound candidate

windowLength length of FFT window, ms

overlap overlap between successive FFT frames, %

step you can override `overlap` by specifying FFT step, ms

verbose if TRUE, plays back the accepted candidate at each iteration and reports the outcome

padWith compared spectra are padded with either silence (padWith = 0) or with NA's (padWith = NA) to have the same number of columns. When the sounds are of different duration, padding with zeros rather than NA's improves the fit to target measured by method = 'pixel' and 'dtw', but it has no effect on 'cor' and 'cosine'.

penalizeLengthDiff if TRUE, sounds of different length are considered to be less similar; if FALSE, only the overlapping parts of two sounds are compared

dynamicRange parts of the spectra quieter than `-dynamicRange` dB are not compared

maxFreq parts of the spectra above maxFreq Hz are not compared

Value

Returns a list of length 2: `$history` contains the tried parameter values together with their fit to target (`$history$sim`), and `$pars` contains a list of the final - hopefully the best - parameter settings.

Examples

```r
playback = c(TRUE, FALSE)[2]  # set to TRUE to play back the audio from examples

target = soundgen(repeatBout = 3, syllLen = 120, pauseLen = 70,
                  pitch = c(300, 200), rolloff = -5, play = playback)
# we hope to reproduce this sound
```
## Not run:

# Match pars based on acoustic analysis alone, without any optimization.
# This MAY match temporal structure, pitch, and stationary formants
m1 = matchPars(target = target,
    samplingRate = 16000,
    maxIter = 0,  # no optimization, only acoustic analysis
    verbose = playback)
cand1 = do.call(soundgen, c(m1$pars, list(play = playback, temperature = 0.001)))

# Try to improve the match by optimizing rolloff
# (this may take a few minutes to run, and the results may vary)

m2 = matchPars(target = target,
    samplingRate = 16000,
    pars = 'rolloff',
    maxIter = 100,
    verbose = playback)

# rolloff should be moving from default (-9) to target (-5):
sapply(m2$history, function(x) x$pars$rolloff)
cand2 = do.call(soundgen, c(m2$pars, list(play = playback, temperature = 0.001)))

## End(Not run)

---

### Description

Modulationspectrum produces a modulation spectrum of waveform(s) or audio file(s), with temporal modulation along the X axis (Hz) and spectral modulation (1/KHz) along the Y axis. A good visual analogy is decomposing the spectrogram into a sum of ripples of various frequencies and directions. Algorithm: prepare a spectrogram, take its logarithm (if logspec = TRUE), center, perform a 2D Fourier transform (see also specNfft), take the upper half of the resulting symmetric matrix, and raise it to power = 2. The result is returned as $original. Roughness is calculated as the proportion of energy / amplitude of the modulation spectrum within roughrange of temporal modulation frequencies. By default, the modulation matrix is then smoothed with Gaussian blur (see gaussianSmooth2D) and log-warped (if logWarp is a positive number) prior to plotting. This processed modulation spectrum is returned as $processed. For multiple inputs, such as a list of waveforms or path to a folder with audio files, the ensemble of modulation spectra is interpolated to the same spectral and temporal resolution and averaged. This is different from the behavior of modulationspectrumFolder, which produces a separate modulation spectrum per file, without averaging.

### Usage

```r
modulationspectrum(x, samplingRate = NULL, maxDur = 5,
    logSpec = FALSE, windowLength = 25, step = NULL, overlap = 80,
    wn = "gaussian", zp = 0, power = 1, roughRange = c(30, 150),
    plot = TRUE, savePath = NA, logWarp = 2, quantiles = c(0.5, 0.8, 0.9),
    kernelSize = 5, kernelSD = 0.5, colorTheme = c("bw",
```
"seewave", ",...")[1], xlab = "Hz", ylab = "1/KHz", main = NULL,
width = 900, height = 500, units = "px", res = NA, ...)

Arguments

x folder, path to a wav/mp3 file, a numeric vector representing a waveform, or a
list of numeric vectors

samplingRate sampling rate of x (only needed if x is a numeric vector, rather than an audio
file). For a list of sounds, give either one samplingRate (the same for all) or as
many values as there are input files

maxDur maximum allowed duration of a single sound, s (longer sounds are split)

logSpec if TRUE, the spectrogram is log-transformed prior to taking 2D FFT

windowLength length of FFT window, ms

step you can override overlap by specifying FFT step, ms

overlap overlap between successive FFT frames, %

wn window type: gaussian, hanning, hamming, bartlett, rectangular, blackman, flat-
top

zp window length after zero padding, points

power raise modulation spectrum to this power (eg power = 2 for \(^2\), or "power spec-
trum")

roughRange the range of temporal modulation frequencies that constitute the "roughness"
zone, Hz

plot if TRUE, plots the modulation spectrum

savePath if a valid path is specified, a plot is saved in this folder (defaults to NA)

logWarp the base of log for warping the modulation spectrum (ie log2 if logWarp = 2); set
to NULL or NA if you don’t want to log-warp

quantiles labeled contour values, % (e.g., "50" marks regions that contain 50% of the sum
total of the entire modulation spectrum)

kernelSize the size of Gaussian kernel used for smoothing (1 = no smoothing)

kernelSD the SD of Gaussian kernel used for smoothing, relative to its size

colorTheme black and white (’bw’), as in seewave package (‘seewave’), or any palette from
palette such as ‘heat.colors’, ‘cm.colors’, etc

xlab, ylab, main

graphical parameters

width, height, units, res

parameters passed to png if the plot is saved

... other graphical parameters passed on to filled.contour.modif2 and contour

Value

Returns a list with three components:
• Original modulation spectrum prior to blurring and log-warping, but after squaring if `power = TRUE`, a matrix of nonnegative values. Rownames are temporal modulation frequencies (Hz), and colnames are spectral modulation frequencies (cycles/KHz).

• Processed modulation spectrum after blurring and log-warping

• Roughness proportion of energy/amplitude of the modulation spectrum within `roughRange` of temporal modulation frequencies, %

References


Examples

```r
# white noise
ms = modulationSpectrum(runif(16000), samplingRate = 16000, 
  logSpec = FALSE, power = TRUE, logWarp = NULL)

# harmonic sound
s = soundgen()
ms = modulationSpectrum(s, samplingRate = 16000, 
  logSpec = FALSE, power = TRUE, logWarp = NULL)

# embellish
ms = modulationSpectrum(s, samplingRate = 16000, 
  xlab = 'Temporal modulation, Hz', ylab = 'Spectral modulation, 1/KHz', 
  colorTheme = 'seewave', main = 'Modulation spectrum', lty = 3)
## Not run:
# Input can also be a list of waveforms (numeric vectors)
ss = vector('list', 10)
for (i in 1:length(ss)) {
  ss[[i]] = soundgen(syllen = runif(1, 100, 1000), temperature = .4, 
    pitch = runif(3, 400, 600))
}
# lapply(ss, playme)
ms = modulationSpectrum(ss[[1]], samplingRate = 16000) # the first sound
ms = modulationSpectrum(ss, samplingRate = 16000) # all 10 sounds

# As with spectrograms, there is a tradeoff in time-frequency resolution
s = soundgen(pitch = 500, amFreq = 50, amDep = 100, samplingRate = 44100)
# playme(s, samplingRate = 44100)
ms = modulationSpectrum(s, samplingRate = 44100, 
  windowLength = 50, overlap = 0) # poor temporal resolution
ms = modulationSpectrum(s, samplingRate = 44100, 
  windowLength = 5, overlap = 80) # poor frequency resolution
ms = modulationSpectrum(s, samplingRate = 44100, 
  windowLength = 15, overlap = 80) # a reasonable compromise

# Input can be a wav/mp3 file
ms = modulationSpectrum('~/Downloads/temp/200_ut_fear-bungee_11.wav')
```
modulationSpectrumFolder

```
ms = modulationSpectrum '~/Downloads/temp/200_ut_fear-bungee_11.wav',
kernalsize = 17,  # more smoothing
xlim = c(-20, 20), ylim = c(0, 4),  # zoom in on the central region
quantiles = c(.25, .5, .75),  # customize contour lines
colorTheme = 'heat.colors',  # alternative palette
logWarp = NULL,  # don't log-warp the modulation spectrum
power = 2  # ^2
# NB: xlim/ylim currently won't work properly with logWarp on

# Input can be path to folder with audio files (average modulation spectrum)
ms = modulationSpectrum '~/Downloads/temp/', kernalsize = 17)
# NB: longer files will be split into fragments <maxDur in length

# "power = 2" returns squared modulation spectrum - note that this affects
# the roughness measure!
# A sound with ~3 syllables per second and only downsweeps in F0 contour
s = soundgen(nSyl = 8, sylLen = 200, pauseLen = 100, pitch = c(300, 200))
# playme(s)
ms = modulationSpectrum(s, samplingRate = 16000, maxDur = .5,
xlim = c(-25, 25), colorTheme = 'seewave', logWarp = NULL,
power = 2)
# note the asymmetry b/c of downsweeps
ms$roughness

# compare:
modulationSpectrum(s, samplingRate = 16000, maxDur = .5,
xlim = c(-25, 25), colorTheme = 'seewave', logWarp = NULL,
power = 1)$roughness  # much higher roughness

# Plotting with or without log-warping the modulation spectrum:
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
logWarp = NA, plot = T)
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
logWarp = 2, plot = T)
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
logWarp = 4.5, plot = T)

# logWarp and kernalsize have no effect on roughness
# because it is calculated before these transforms:
modulationSpectrum(s, samplingRate = 16000, logWarp = 5)$roughness
modulationSpectrum(s, samplingRate = 16000, logWarp = NA)$roughness
modulationSpectrum(s, samplingRate = 16000, kernalsize = 17)$roughness

# Log-transform the spectrogram prior to 2D FFT (affects roughness):
ms = modulationSpectrum(soundgen(), samplingRate = 16000, logSpec = FALSE)
ms = modulationSpectrum(soundgen(), samplingRate = 16000, logSpec = TRUE)
```

## End(Not run)
Modulation Spectrum Folder

Description

Extracts modulation spectra of all wav/mp3 files in a folder - separately for each file, without averaging. Good for saving plots of the modulation spectra and/or measuring the roughness of multiple files. See modulationSpectrum for further details.

Usage

modulationSpectrumFolder(myfolder, summary = TRUE, htmlPlots = TRUE, verbose = TRUE, maxDur = 5, logSpec = FALSE, windowLength = 25, step = NULL, overlap = 80, wn = "gaussian", zp = 0, power = 1, roughRange = c(30, 150), plot = FALSE, savePlots = FALSE, logWarp = 2, quantiles = c(0.5, 0.8, 0.9), kernelSize = 5, kernelSD = 0.5, colorTheme = c("bw", "seewave", "...")[1], xlab = "Hz", ylab = "1/KHz", width = 900, height = 500, units = "px", res = NA, ...)

Arguments

myfolder full path to target folder
summary if TRUE, returns only a summary of the measured acoustic variables (mean, median and SD). If FALSE, returns a list containing frame-by-frame values
htmlPlots if TRUE, saves an html file with clickable plots
verbose if TRUE, reports progress and estimated time left
maxDur maximum allowed duration of a single sound, s (longer sounds are split)
logSpec if TRUE, the spectrogram is log-transformed prior to taking 2D FFT
windowLength length of FFT window, ms
step you can override overlap by specifying FFT step, ms
overlap overlap between successive FFT frames, %
wn window type: gaussian, hanning, hamming, bartlett, rectangular, blackman, flattop
zp window length after zero padding, points
power raise modulation spectrum to this power (eg power = 2 for \(2^2\), or "power spectrum")
roughRange the range of temporal modulation frequencies that constitute the "roughness" zone, Hz
plot if TRUE, produces a spectrogram with pitch contour overlaid
savePlots if TRUE, saves plots as .png files
logWarp the base of log for warping the modulation spectrum (ie log2 if logWarp = 2); set to NULL or NA if you don’t want to log-warp
quantiles labeled contour values, % (e.g., "50" marks regions that contain 50% of the sum total of the entire modulation spectrum)
kernelSize the size of Gaussian kernel used for smoothing (1 = no smoothing)
kernelSD the SD of Gaussian kernel used for smoothing, relative to its size
colorTheme black and white ('bw'), as in seewave package ('seewave'), or any palette from palette such as 'heat.colors', 'cm.colors', etc
xlab plotting parameters
ylab plotting parameters
width parameters passed to png if the plot is saved
height parameters passed to png if the plot is saved
units parameters passed to png if the plot is saved
res parameters passed to png if the plot is saved
... other graphical parameters passed to spectrogram

Value
If summary is TRUE, returns a dataframe with just the roughness measure per audio file. If summary is FALSE, returns a list with the actual modulation spectra.

Examples
## Not run:
ms = modulationSpectrumFolder('~/Downloads/temp', savePlots = TRUE, kernelSize = 15)

## End(Not run)

morph Morph sounds

Description
Takes two formulas for synthesizing two target sounds with soundgen and produces a number of intermediate forms (morphs), attempting to go from one target sound to the other in a specified number of equal steps.

Usage
morph(formula1, formula2, nMorphs, playMorphs = TRUE, savePath = NA, samplingRate = 16000)

Arguments
formula1, formula2 lists of parameters for calling soundgen that produce the two target sounds between which morphing will occur. Character strings containing the full call to soundgen are also accepted (see examples)
nMorphs the number of morphs to produce, including target sounds
playMorphs if TRUE, the morphs will be played
savePath if it is the path to an existing directory, morphs will be saved there as individual .wav files (defaults to NA)
samplingRate sampling rate of output, Hz. NB: must be the same as in formula1 and formula2!
normalizeFolder

Value

A list of two sublists ($Dformulas$ and $Dsounds$), each of length $nmorphs$. For ex., the formula for the second hybrid is $mDformulas[[2]]$, and the waveform is $mDsounds[[2]]$

Examples

```r
# write two formulas or copy-paste them from soundgen_app() or presets:
playback = c(TRUE, FALSE)[2]
# [a] to barking
m = morph(formula1 = list(repeatBout = 2),
          # equivalently: formula1 = 'soundgen(repeatBout = 2)',
          formula2 = presets$Misc$Dog_bark,
          nMorphs = 5, playMorphs = playback)
# use $formulas to access formulas for each morph, $sounds for waveforms
# m$formulas[[4]]
# playme(m$sounds[[3]])

## Not run:
## morph intonation and vowel quality
m = morph(
          'soundgen(pitch = c(300, 250, 400), formants = c(350, 2900, 3600, 4700))',
          'soundgen(pitch = c(300, 700, 500, 300), formants = c(800, 1250, 3100, 4500))',
          nMorphs = 5, playMorphs = playback)
)

# from a grunt of disgust to a moan of pleasure
m = morph(
          formula1 = 'soundgen(syllen = 180, pitch = c(160, 160, 120), rolloff = -12,
                        nonlinBalance = 70, subFreq = 75, subDep = 35, jitterDep = 2,
                        formants = c(550, 1200, 2100, 4300, 4700, 6500, 7300),
                        noise = data.frame(time = c(0, 180, 270), value = c(-25, -25, -40)),
                        rolloffNoise = 0),
          formula2 = 'soundgen(syllen = 320, pitch = c(340, 330, 300),
                        rolloff = c(-18, -16, -30), ampl = c(0, -10), formants = c(950, 1700, 3700),
                        noise = data.frame(time = c(0, 300, 440), value = c(-35, -25, -65)),
                        mouth = c(.4, .5), rolloffNoise = -5, attackLen = 30)',
          nMorphs = 8, playMorphs = playback)
)
## End(Not run)
```

normalizeFolder Normalize folder

Description

Normalizes the amplitude of all wav/mp3 files in a folder based on their peak or RMS amplitude or subjective loudness. This is good for playback experiments, which require that all sounds should have similar intensity or loudness.
normalizeFolder

Usage

normalizeFolder(myfolder, type = c("peak", "rms", "loudness") [1],
    maxAmp = 0, summaryFun = "mean", windowLength = 50, step = NULL,
    overlap = 70, killDC = FALSE, windowDC = 200, savepath = NULL,
    verbose = TRUE)

Arguments

myfolder path to folder containing wav/mp3 files

type normalize so the output files has the same peak amplitude ('peak'), root mean
    square amplitude ('rms'), or subjective loudness in sone ('loudness')

maxAmp maximum amplitude in dB (0 = max possible, -10 = 10 dB below max possible,
    etc.)

summaryFun should the output files have the same mean / median / max etc rms amplitude or
    loudness? (summaryFun has no effect if type = 'peak')

windowLength length of FFT window, ms

step you can override overlap by specifying FFT step, ms

overlap overlap between successive FFT frames, %

killDC if TRUE, removed DC offset (see also flatEnv)

windowDC the window for calculating DC offset, ms

savepath full path to where the normalized files should be saved (defaults to '/normalized')

verbose if TRUE, reports estimated time left

Details

Algorithm: first all files are rescaled to have the same peak amplitude of maxAmp dB. If type = 'peak',
the process ends here. If type = 'rms', there are two additional steps. First the original RMS am-
plitude of all files is calculated per frame by getRMS. The "quietest" sound with the lowest summary
RMS value is not modified, so its peak amplitude remains maxAmp dB. All the remaining sounds
are rescaled linearly, so that their summary RMS values becomes the same as that of the "quietest"
sound, and their peak amplitudes become smaller, <maxAmp. Finally, if type = 'loudness', the
subjective loudness of each sound is estimated by getLoudness, which assumes frequency sen-
sitivity typical of human hearing. The following normalization procedure is similar to that for
type = 'rms'.

Examples

## not run:
# put a few short audio files in a folder, eg '~/Downloads/temp'
getRMSFolder '~/Downloads/temp', summaryFun = 'mean') # different
normalizeFolder '~/Downloads/temp', type = 'rms', summaryFun = 'mean',
    savepath = '~/Downloads/temp/normalized')
getRMSFolder '~/Downloads/temp/normalized', summaryFun = 'mean') # same
# if the saved audio files are treated as stereo with one channel missing,
# try reconverting with ffmpeg (saving is handled by tuneR::writeWave)
notesDict

## End(Not run)

<table>
<thead>
<tr>
<th>notesDict</th>
<th>Conversion table from Hz to musical notation</th>
</tr>
</thead>
</table>

### Description

A dataframe of 192 rows and 2 columns: "note" and "freq" (Hz). Range: C-5 (0.51 Hz) to B10 (31608.53 Hz)

### Usage

notesDict

### Format

An object of class `data.frame` with 192 rows and 2 columns.

---

<table>
<thead>
<tr>
<th>optimizePars</th>
<th>Optimize parameters for acoustic analysis</th>
</tr>
</thead>
</table>

### Description

This customized wrapper for `optim` attempts to optimize the parameters of `segmentFolder` or `analyzeFolder` by comparing the results with a manually annotated "key". This optimization function uses a single measurement per audio file (e.g., median pitch or the number of syllables). For other purposes, you may want to adapt the optimization function so that the key specifies the exact timing of syllables, their median length, frame-by-frame pitch values, or any other characteristic that you want to optimize for. The general idea remains the same, however: we want to tune function parameters to fit our type of audio and research priorities. The default settings of `segmentFolder` and `analyzeFolder` have been optimized for human non-linguistic vocalizations.

### Usage

```r
optimizePars(myfolder, key, myfun, pars, bounds = NULL, fitnessPar, fitnessFun = function(x) 1 - cor(x, key, use = "pairwise.complete.obs"), nIter = 10, init = NULL, initSD = 0.2, control = list(maxit = 50, reltol = 0.01, trace = 0), otherPars = list(plot = FALSE, verbose = FALSE), mygrid = NULL, verbose = TRUE)
```
optimizePars

Arguments

- **myfolder**: path to where the .wav files live
- **key**: a vector containing the "correct" measurement that we are aiming to reproduce
- **myfun**: the function being optimized: either 'segmentFolder' or 'analyzeFolder' (in quotes)
- **pars**: names of arguments to myfun that should be optimized
- **bounds**: a list setting the lower and upper boundaries for possible values of optimized parameters. For ex., if we optimize smooth and smooth0verlap, reasonable bounds might be list(low = c(5, 0), high = c(500, 95))
- **fitnessPar**: the name of output variable that we are comparing with the key, e.g. 'nBursts' or 'pitch_median'
- **fitnessFun**: the function used to evaluate how well the output of myfun fits the key. Defaults to 1 - Pearson’s correlation (i.e. 0 is perfect fit, 1 is awful fit). For pitch, log scale is more meaningful, so a good fitness criterion is "function(x) 1 - cor(log(x), log(key), use = 'pairwise.complete.obs')"
- **nIter**: repeat the optimization several times to check convergence
- **init**: initial values of optimized parameters (if NULL, the default values are taken from the definition of myfun)
- **initSD**: each optimization begins with a random seed, and initSD specifies the SD of normal distribution used to generate random deviation of initial values from the defaults
- **control**: a list of control parameters passed on to optim. The method used is "Nelder-Mead"
- **otherpars**: a list of additional arguments to myfun
- **mygrid**: a dataframe with one column per parameter to optimize, with each row specifying the values to try. If not NULL, optimizePars simply evaluates each combination of parameter values, without calling optim (see examples)
- **verbose**: if TRUE, reports the values of parameters evaluated and fitness

Details

If your sounds are very different from human non-linguistic vocalizations, you may want to change the default values of other arguments to speed up convergence. Adapt the code to enforce suitable constraints, depending on your data.

Value

Returns a matrix with one row per iteration with fitness in the first column and the best values of each of the optimized parameters in the remaining columns.

Examples

```r
## Not run:
# Download 260 sounds from the supplements in Anikin & Persson (2017)
# - see http://cogsci.se/publications.html
```
```
# Unzip them into a folder, say '~/Downloads/temp'
myfolder = '~/Downloads/temp' # 260 .wav files live here

# Optimization of SEGMENTATION
# Import manual counts of syllables in 260 sounds from
# Anikin & Persson (2017) (our "key")
key = segmentManual # a vector of 260 integers

# Run optimization loop several times with random initial values
# to check convergence
# NB: with 260 sounds and default settings, this might take ~20 min per iteration!
res = optimizePars(myfolder = myfolder, myfun = 'segmentFolder', key = key,
            pars = c('shortestSyl', 'shortestPause', 'sylThres'),
            fitnessPar = 'nBursts',
            nIter = 3, control = list(maxit = 50, reltol = .01, trace = 0))

# Examine the results
print(res)
for (c in 2:ncol(res)) {
    plot(res[, c], res[, 1], main = colnames(res)[c])
}

pars = as.list(res[1, 2:ncol(res)]) # top candidate (best pars)
s = do.call(segmentFolder, c(myfolder, pars)) # segment with best pars
cor(key, as.numeric(s[, fitnessPar]))
boxplot(as.numeric(s[, fitnessPar]) ~ as.integer(key), xlab='key')
abline(a=0, b=1, col='red')

# Try a grid with particular parameter values instead of formal optimization
res = optimizePars(myfolder = myfolder, myfun = 'segmentFolder', key = segment_manual,
            pars = c('shortestSyl', 'shortestPause'),
            fitnessPar = 'nBursts',
            mygrid = expand.grid(shortestSyl = c(30, 40),
                                  shortestPause = c(30, 40, 50)))
1 - res$fit # correlations with key

# Optimization of PITCH TRACKING (takes several hours!)
res = optimizePars(myfolder = myfolder,
            myfun = 'analyzeFolder',
            key = log(pitchManual), # log-scale better for pitch
            pars = c('specThres', 'specSmooth'),
            bounds = list(low = c(0, 0), high = c(1, Inf)),
            fitnessPar = 'pitch_median',
            nIter = 2,
            otherPars = list(plot = FALSE, verbose = FALSE, step = 50,
                              pitchMethods = 'spec'),
            fitnessFun = function(x) {
                1 - cor(log(x), key, use = 'pairwise.complete.obs') *
                (1 - mean(is.na(x) & !is.na(key))) # penalize failing to detect F0
            })

## End(Not run)
```
Description

Plots the oscillogram (waveform) of a sound on a logarithmic scale, in dB. Analogous to "Waveform (dB)" view in Audacity.

Usage

```r
osc_db(x, dynamicRange = 80, maxAmpl = NULL, samplingRate = NULL,
       returnWave = FALSE, plot = TRUE, xlab = NULL, ylab = "dB",
       bty = "n", midline = TRUE, ...)
```

Arguments

- `x`: path to a .wav file or a CENTERED (mean \(\approx 0\)) vector of amplitudes with specified samplingRate
- `dynamicRange`: dynamic range of the oscillogram, dB
- `maxAmpl`: the maximum theoretically possible value indicating on which scale the sound is coded: 1 if the range is -1 to +1, \(2^{15}\) for 16-bit wav files, etc
- `samplingRate`: sampling rate of `x` (only needed if `x` is a numeric vector, rather than a .wav file)
- `returnWave`: if TRUE, returns a log-transformed waveform as a numeric vector
- `plot`: if TRUE, plots the oscillogram
- `xlab`, `ylab`: axis labels
- `bty`: box type (see `?par`)
- `midline`: if TRUE, draws a line at 0 dB
- `...`: Other graphical parameters passed on to `plot()`

Details

Algorithm: centers and normalizes the sound, then takes a logarithm of the positive part and a flipped negative part.

Value

Returns the input waveform on a dB scale: a vector with range from `-dynamicRange` to `dynamicRange`.
Examples

```r
sound = sin(1:2000/10) *
getSmoothContour(anchors = c(1, .01, .5), len = 2000)

# Oscillogram on a linear scale
plot(sound, type = 'l')
# or, for fancy plotting options: seewave::oscillo(sound, f = 1000)

# Oscillogram on a dB scale
osc_db(sound)

# Time in ms if samplingRate is specified
osc_db(sound, samplingRate = 5000)

# Assuming that the waveform can range up to 50 instead of 1
osc_db(sound, maxAmpl = 50)

# Embellish and customize the plot
o = osc_db(sound, samplingRate = 1000, midline = FALSE,
            main = 'My waveform', col = 'blue')
abline(h = 0, col = 'orange', lty = 3)
```

---

permittedValues  Defaults and ranges

Description

A dataset containing defaults and ranges of key variables in the Shiny app. Adjust as needed.

Usage

```r
permittedValues
```

Format

A matrix with 58 rows and 4 variables:

- **default** default value
- **low** lowest permitted value
- **high** highest permitted value
- **step** increment for adjustment ...
pitchManual

Manual pitch estimation in 260 sounds

Description

A vector of manually verified pitch values per sound in the corpus of 590 human non-linguistic emotional vocalizations from Anikin & Persson (2017). The corpus can be downloaded from http://cogsci.se/publications.html

Usage

pitchManual

Format

An object of class numeric of length 260.

playme

Play audio

Description

Plays an audio file or a numeric vector. This is a simple wrapper for the functionality provided by play

Usage

playme(sound, samplingRate = 16000)

Arguments

sound a vector of numbers on any scale or a path to a .wav file
samplingRate sampling rate (only needed if sound is a vector)

Examples

# playme('~/myfile.wav')
f0_Hz = 440
sound = sin(2 * pi * f0_Hz * (1:16000) / 16000)
# playme(sound, 16000)
# in case of errors, look into tuneR::play()
**Description**

A library of presets for easy generation of a few nice sounds.

**Usage**

```
presets
```

**Format**

A list of length 4.

---

**reportTime**

**Report time**

**Description**

Provides a nicely formatted "estimated time left" in loops plus a summary upon completion.

**Usage**

```
reportTime(i, nIter, time_start, jobs = NULL, reportEvery = 1)
```

**Arguments**

- `i` : current iteration
- `nIter` : total number of iterations
- `time_start` : time when the loop started running
- `jobs` : vector of length `nIter` specifying the relative difficulty of each iteration. If not NULL, estimated time left takes into account whether the jobs ahead will take more or less time than the jobs already completed
- `reportEvery` : report progress every `n` iterations
Examples

time_start = proc.time()
for (i in 1:20) {
    Sys.sleep(i ^ 2 / 10000)
    reportTime(i = i, nIter = 20, time_start = time_start,
                jobs = (1:20) ^ 2, reportEvery = 5)
}
# Not run:
# when analyzing a bunch of audio files, their size is a good estimate
# of how long each will take to process

time_start = proc.time()
filenames = list.files("~/Downloads/temp", pattern = "*.wav|*.mp3",
                     full.names = TRUE)
fileSizes = file.info(filenames)$size
for (i in 1:length(filenames)) {
    # do what you have to do with each file...
    reportTime(i = i, nIter = length(filenames),
               time_start = time_start, jobs = fileSizes)
}
# End(Not run)

schwa

Schwa-related formant conversion

Description

This function performs several conceptually related types of conversion of formant frequencies in relation to the neutral schwa sound based on the one-tube model of the vocal tract. Case 1: if we know vocal tract length (VTL) but not formant frequencies, schwa() estimates formants corresponding to a neutral schwa sound in this vocal tract, assuming that it is perfectly cylindrical. Case 2: if we know the frequencies of a few lower formants, schwa() estimates the deviation of observed formant frequencies from the neutral values expected in a perfectly cylindrical vocal tract (based on the VTL as specified or as estimated from formant dispersion). Case 3: if we want to generate a sound with particular relative formant frequencies (e.g. high F1 and low F2 relative to the schwa for this vocal tract), schwa() calculates the corresponding formant frequencies in Hz. See examples below for an illustration of these three suggested uses.

Usage

schwa(formants = NULL, vocalTract = NULL, formants_relative = NULL,
      nForm = 8, speedSound = 35400)

Arguments

formants a numeric vector of observed (measured) formant frequencies, Hz
vocalTract the length of vocal tract, cm
formants_relative
   a numeric vector of target relative formant frequencies, % deviation from schwa
   (see examples)

nForm
   the number of formants to estimate (integer)

speedSound
   speed of sound in warm air, cm/s. Stevens (2000) "Acoustic phonetics", p. 138

Details

Algorithm: the expected formant dispersion is given by $\frac{\text{speedSound}}{(2 \times \text{vocalTract})}$, and $F_1$ is expected at half the value of formant dispersion. See e.g. Stevens (2000) "Acoustic phonetics", p. 139. Basically, we estimate vocal tract length and see if each formant is higher or lower than expected for this vocal tract. For this to work, we have to know either the frequencies of enough formants (not just the first two) or the true length of the vocal tract. See also estimateVTL on the algorithm for estimating formant dispersion if VTL is not known.

Value

Returns a list with the following components:

- **vtl_measured**: VTL as provided by the user, cm
- **vocalTract_apparent**: VTL estimated based on formants frequencies provided by the user, cm
- **formantDispersion**: average distance between formants, Hz
- **ff_measured**: formant frequencies as provided by the user, Hz
- **ff_schwa**: formant frequencies corresponding to a neutral schwa sound in this vocal tract, Hz
- **ff_theoretical**: formant frequencies corresponding to the user-provided relative formant frequencies, Hz
- **ff_relative**: deviation of formant frequencies from those expected for a schwa, % (e.g. if the first ff_relative is -25, it means that $F_1$ is 25% lower than expected for a schwa in this vocal tract)
- **ff_relative_semitones**: deviation of formant frequencies from those expected for a schwa, semitones

Examples

```r
## CASE 1: known VTL
# If vocal tract length is known, we calculate expected formant frequencies
schwa(vocalTract = 17.5)
schwa(vocalTract = 13, nForm = 5)

## CASE 2: known (observed) formant frequencies
# Let's take formant frequencies in three vocalizations
# (/a/, /i/, /roar/) by the same male speaker:
formants_a = c(860, 1430, 2900, 4200, 5200)
s_a = schwa(formants = formants_a)
s_a
# We get an estimate of VTL (s_a$vtl_apparent = 15.2 cm),
# same as with estimateVTL(formants_a)
# We also get theoretical schwa formants: s_a$ff_schwa
# And we get the difference (\% and semitones) in observed vs expected
```
# formant frequencies: s_a[c('ff_relative', 'ff_relative_semitones')]
# [a]: F1 much higher than expected, F2 slightly lower

formants_i = c(300, 2700, 3400, 4400, 5300, 6400)
s_i = schwa(formants = formants_i)
s_i
# The apparent VTL is slightly smaller (14.5 cm)
# [i]: very low F1, very high F2

formants_roar = c(550, 1000, 1460, 2280, 3350, 4300, 4900, 5800, 6900, 7900)
s_roar = schwa(formants = formants_roar)
s_roar
# Note the enormous apparent VTL (22.5 cm!)
# (lowered larynx and rounded lips exaggerate the apparent size)
# s_roar$ff_relative: high F1 and low F2-F4

schwa(formants = formants_roar[1:4])
# based on F1-F4, apparent VTL is almost 28 cm!
# Since the lowest formants are the most salient,
# the apparent size is exaggerated even further

# If you know VTL, a few lower formants are enough to get
# a good estimate of the relative formant values:
schwa(formants = formants_roar[1:4], vocalTract = 19)
# NB: in this case theoretical and relative formants are calculated
# based on user-provided VTL (vtl_measured) rather than vtl_apparent

## CASE 3: from relative to absolute formant frequencies
# Say we want to generate a vowel sound with F1 20% below schwa
# and F2 40% above schwa, with VTL = 15 cm
s = schwa(formants_relative = c(-20, 40), vocalTract = 15)
# s$ff_schwa gives formant frequencies for a schwa, while
# s$ff_theoretical gives formant frequencies for a sound with
# target relative formant values (low F1, high F2)
schwa(formants = s$ff_theoretical)

---

segment **Segment a sound**

Description

Finds syllables and bursts. Syllables are defined as continuous segments with amplitude above threshold. Bursts are defined as local maxima in amplitude envelope that are high enough both in absolute terms (relative to the global maximum) and with respect to the surrounding region (relative to local mimima). See vignette('acoustic_analysis', package = 'soundgen') for details.

Usage

segment(x, samplingRate = NULL, windowLength = 40, overlap = 80,
shortestSyl = 40, shortestPause = 40, sylThres = 0.9,
interburst = NULL, interburstMul = 1, burstThres = 0.075,
peakToTrough = 3, troughLeft = TRUE, troughRight = FALSE,
summary = FALSE, plot = FALSE, savePath = NA, col = "green",
xlab = "Time, ms", ylab = "Amplitude", main = NULL, width = 900,
height = 500, units = "px", res = NA, sylPlot = list(lty = 1, lwd = 2, col = "blue"), burstPlot = list(pch = 8, cex = 3, col = "red"),
...)

Arguments

x
path to a .wav or .mp3 file or a vector of amplitudes with specified samplingRate
samplingRate
sampling rate of x (only needed if x is a numeric vector, rather than an audio file)
windowLength, overlap
length (ms) and overlap (window used to produce the amplitude envelope, see env
shortestSyl
minimum acceptable length of syllables, ms
shortestPause
minimum acceptable break between syllables, ms. Syllables separated by less
time are merged. To avoid merging, specify shortestPause = NA
sylThres
amplitude threshold for syllable detection (as a proportion of global mean am-
plitude envelope)
interburst
minimum time between two consecutive bursts (ms). If specified, it overrides
interburstMul
multiplier of the default minimum interburst interval (median syllable length
or, if no syllables are detected, the same number as shortestSyl). Only used
if interburst is not specified. Larger values improve detection of unusually
broad shallow peaks, while smaller values improve the detection of sharp narrow
peaks
burstThres
to qualify as a burst, a local maximum has to be at least burstThres times the
height of the global maximum of amplitude envelope
peakToTrough
to qualify as a burst, a local maximum has to be at least peakToTrough times
the local minimum on the LEFT over analysis window (which is controlled by
interburst or interburstMul)
troughLeft, troughRight
should local maxima be compared to the trough on the left and/or right of it?
Default to TRUE and FALSE, respectively
summary
if TRUE, returns only a summary of the number and spacing of syllables and
vocal bursts. If FALSE, returns a list containing full stats on each syllable and
bursts (location, duration, amplitude,...)
plot
if TRUE, produces a segmentation plot
savePath
full path to the folder in which to save the plots. Defaults to NA
col, xlab, ylab, main
main plotting parameters
width, height, units, res
parameters passed to png if the plot is saved

sylPlot       a list of graphical parameters for displaying the syllables
burstPlot     a list of graphical parameters for displaying the bursts
...
other graphical parameters passed to plot

Details
The algorithm is very flexible, but the parameters may be hard to optimize by hand. If you
have an annotated sample of the sort of audio you are planning to analyze, with syllables and/or
bursts counted manually, you can use it for automatic optimization of control parameters (see
optimizepars. The defaults are the results of just such optimization against 260 human vocal-
izations in Anikin, A. & Persson, T. (2017). Non-linguistic vocalizations from online amateur

Value
If summary = TRUE, returns only a summary of the number and spacing of syllables and vocal
bursts. If summary = FALSE, returns a list containing full stats on each syllable and bursts (location,
duration, amplitude, ...).

Examples

```
sound = soundgen(nSyl = 8, sylLen = 50, pauseLen = 70,
pitch = c(368, 284), temperature = 0.1,
noise = list(time = c(0, 67, 86, 186), value = c(-45, -47, -89, -120)),
rolloff_noise = -8, amplAnchorsGlobal = c(0, -20))
spectrogram(sound, samplingRate = 16000, osc = TRUE)
# playme(sound, samplingRate = 16000)

s = segment(sound, samplingRate = 16000, plot = TRUE)
# accept quicker and quieter syllables
s = segment(sound, samplingRate = 16000, plot = TRUE,
shortestSyl = 25, shortestPause = 25, sylThres = .2, burstThres = .05)
# just a summary
segment(sound, samplingRate = 16000, summary = TRUE)

# customizing the plot
s = segment(sound, samplingRate = 16000, plot = TRUE,
shortestSyl = 25, shortestPause = 25,
sylThres = .2, burstThres = .05,
col = 'black', lwd = .5,
sylPlot = list(lty = 2, col = 'gray20'),
burstPlot = list(pch = 16, col = 'gray80'),
xlab = 'ms', cex.lab = 1.2, main = 'My awesome plot')
```

## Not run:
# customize the resolution of saved plot
s = segment(sound, samplingRate = 16000, savePath = '~/Downloads/',
width = 1920, height = 1080, units = 'px')
segmenFolder

## End(Not run)

### segmentFolder

**Segment all files in a folder**

**Description**

Finds syllables and bursts in all .wav files in a folder.

**Usage**

```r
segmentFolder(myfolder, htmlPlots = TRUE, shortestSyl = 40,
shortestPause = 40, sylThres = 0.9, interburst = NULL,
interburstMult = 1, burstThres = 0.075, peakToTrough = 3,
troughLeft = TRUE, troughRight = FALSE, windowLength = 40,
overlap = 80, summary = TRUE, plot = FALSE, savePlots = FALSE,
savePath = NA, verbose = TRUE, reportEvery = 10, col = "green",
xlab = "Time, ms", ylab = "Amplitude", main = NULL, width = 900,
height = 500, units = "px", res = NA, sylPlot = list(lty = 1, lwd
= 2, col = "blue"), burstPlot = list(pch = 8, cex = 3, col = "red"),
...)
```

**Arguments**

- `myfolder`: full path to target folder
- `htmlPlots`: if TRUE, saves an html file with clickable plots
- `shortestSyl`: minimum acceptable length of syllables, ms
- `shortestPause`: minimum acceptable break between syllables, ms. Syllables separated by less time are merged. To avoid merging, specify `shortestPause = NA`
- `sylThres`: amplitude threshold for syllable detection (as a proportion of global mean amplitude of smoothed envelope)
- `interburst`: minimum time between two consecutive bursts (ms). If specified, it overrides `interburstMult`
- `interburstMult`: multiplier of the default minimum interburst interval (median syllable length or, if no syllables are detected, the same number as `shortestSyl`). Only used if `interburst` is not specified. Larger values improve detection of unusually broad shallow peaks, while smaller values improve the detection of sharp narrow peaks
- `burstThres`: to qualify as a burst, a local maximum has to be at least `burstThres` times the height of the global maximum of amplitude envelope
- `peakToTrough`: to qualify as a burst, a local maximum has to be at least `peakToTrough` times the local minimum on the LEFT over analysis window (which is controlled by `interburst` or `interburstMult`)
troughLeft should local maxima be compared to the trough on the left and/or right of it? Default to TRUE and FALSE, respectively

troughRight should local maxima be compared to the trough on the left and/or right of it? Default to TRUE and FALSE, respectively

windowLength length (ms) and overlap ( window used to produce the amplitude envelope, see env

overlap length (ms) and overlap ( window used to produce the amplitude envelope, see env

summary if TRUE, returns only a summary of the number and spacing of syllables and vocal bursts. If FALSE, returns a list containing full stats on each syllable and bursts (location, duration, amplitude, ...)

plot if TRUE, produces a segmentation plot

savePlots if TRUE, saves plots as .png files

savePath full path to the folder in which to save the plots. Defaults to NA

verbose, reportEvery if TRUE, reports progress every reportEvery files and estimated time left

col main plotting parameters

xlab main plotting parameters

ylab main plotting parameters

main main plotting parameters

width parameters passed to png if the plot is saved

height parameters passed to png if the plot is saved

units parameters passed to png if the plot is saved

res parameters passed to png if the plot is saved

sylPlot a list of graphical parameters for displaying the syllables

burstPlot a list of graphical parameters for displaying the bursts

... other graphical parameters passed to plot

Details

This is just a convenient wrapper for segment intended for analyzing the syllables and bursts in a large number of audio files at a time. In verbose mode, it also reports ETA every ten iterations. With default settings, running time should be about a second per minute of audio.

Value

If summary is TRUE, returns a dataframe with one row per audio file. If summary is FALSE, returns a list of detailed descriptives.
Examples

```r
## Not run:
# Download 260 sounds from the supplements to Anikin & Persson (2017) at
# http://cogsci.se/publications.html
# unzip them into a folder, say '~/Downloads/temp'
mypath = '~/Downloads/temp' # 260 .wav files live here
s = segmentFolder(mypath, verbose = TRUE, savePlot = TRUE)

# Check accuracy: import a manual count of syllables (our "key")
key = segmentManual # a vector of 260 integers
trial = as.numeric(s$mBursts)
cor(key, trial, use = 'pairwise.complete.obs')
boxplot(trial ~ as.integer(key), xlab='key')
abline(a=0, b=1, col='red')

## End(Not run)
```

---

**segmentManual**

Manual counts of syllables in 260 sounds

Description

A vector of the number of syllables in the corpus of 260 human non-linguistic emotional vocalizations from Anikin & Persson (2017). The corpus can be downloaded from http://cogsci.se/publications.html

Usage

`segmentManual`

Format

An object of class `numeric` of length 260.

---

**semitonesToHz**

Convert semitones to Hz

Description

Converts from semitones above C-5 (~0.5109875 Hz) to Hz. See `HzToSemitones`

Usage

`semitonesToHz(s, ref = 0.5109875)`

Arguments

- `s`: vector or matrix of frequencies (semitones above C0)
- `ref`: frequency of the reference value (defaults to C-5, 0.51 Hz)
## Generate a sound

### Description

Generates a bout of one or more syllables with pauses between them. Two basic components are synthesized: the harmonic component (the sum of sine waves with frequencies that are multiples of the fundamental frequency) and the noise component. Both components can be filtered with independently specified formants. Intonation and amplitude contours can be applied both within each syllable and across multiple syllables. Suggested application: synthesis of animal or human non-linguistic vocalizations. For more information, see [http://cogsci.se/soundgen.html](http://cogsci.se/soundgen.html) and vignette('sound_generation', package = 'soundgen').

### Usage

```r
soundgen(repeatBout = 1, nSyl = 1, syllen = 300, pauseLen = 200, 
pitch = data.frame(time = c(0, 0.1, 0.9, 1), value = c(100, 150, 135, 100)), pitchGlobal = NA, glottis = 0, temperature = 0.025, 
tempEffects = list(), maleFemale = 0, creakyBreathy = 0, 
nonlinBalance = 0, nonlinDep = 50, nonlinRandomWalk = NULL, 
jitterLen = 1, jitterDep = 1, vibratoFreq = 5, vibratoDep = 0, 
shimmerDep = 0, shimmerLen = 1, attackLen = 50, rolloff = -9, 
rolloffOct = 0, rolloffKHz = -3, rolloffParab = 0, 
rolloffParabHarm = 3, rolloffExact = NULL, lipRad = 6, 
noseRad = 4, mouthOpenThres = 0, formants = c(860, 1430, 2900), 
formantDep = 1, formantDepStoch = 20, formantWidth = 1, 
vocalTract = NA, subFreq = 100, subDep = 100, 
shortestEpoch = 300, amDep = 0, amFreq = 30, amShape = 0, 
noise = NULL, formantsNoise = NA, rolloffNoise = -4, 
noiseFlatSpec = 1200, noiseAmpRef = c("f0", "source", "filtered")[3], 
mouth = data.frame(time = c(0, 1), value = c(0.5, 0.5)), ampl = NA, 
amplGlobal = NA, interpol = c("approx", "spline", "loess")[3], 
discontThres = 0.05, jumpThres = 0.01, samplingRate = 16000, 
windowLength = 50, overlap = 75, addSilence = 100, 
pitchFloor = 1, pitchCeiling = 3500, pitchSamplingRate = 3500, 
dynamicRange = 80, invalidArgAction = c("adjust", "abort", "ignore")[1], plot = FALSE, play = FALSE, savePath = NA, ...)
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeatBout</td>
<td>number of times the whole bout should be repeated</td>
</tr>
<tr>
<td>nSyl</td>
<td>number of syllables in the bout. ‘pitchGlobal’, ‘amplGlobal’, and ‘formants’ span multiple syllables, but not multiple bouts</td>
</tr>
<tr>
<td>syllen</td>
<td>average duration of each syllable, ms (vectorized)</td>
</tr>
<tr>
<td>pauseLen</td>
<td>average duration of pauses between syllables, ms (can be negative between bouts: force with invalidArgAction = ‘ignore’) (vectorized)</td>
</tr>
</tbody>
</table>
soundgen

pitch a numeric vector of f0 values in Hz or a dataframe specifying the time (ms or 0 to 1) and value (Hz) of each anchor, hereafter “anchor format”. These anchors are used to create a smooth contour of fundamental frequency f0 (pitch) within one syllable.

pitchGlobal unlike pitch, these anchors are used to create a smooth contour of average f0 across multiple syllables. The values are in semitones relative to the existing pitch, i.e., 0 = no change (anchor format).

glottis anchors for specifying the proportion of a glottal cycle with closed glottis, % (0 = no modification, 100 = closed phase as long as open phase); numeric vector or dataframe specifying time and value (anchor format).

temperature hyperparameter for regulating the amount of stochasticity in sound generation.

tempEffects a list of scaling coefficients regulating the effect of temperature on particular parameters. To change, specify just those pars that you want to modify (default is 1 for all of them). syllLenDep: duration of syllables and pauses; formDrift: formant frequencies; formDisp: dispersion of stochastic formants; pitchDriftDep: amount of slow random drift of f0; pitchDriftFreq: frequency of slow random drift of f0; amplDriftDep: drift of amplitude mirroring pitch drift; subDriftDep: drift of subharmonic frequency and bandwidth mirroring pitch drift; rolloffDriftDep: drift of rolloff mirroring pitch drift; pitchDep, noiseDep, amplDep: random fluctuations of user-specified pitch / noise / amplitude anchors; glottisDep: proportion of glottal cycle with closed glottis; specDep: rolloff, rolloffNoise, nonlinear effects, attack.

maleFemale hyperparameter for shifting f0 contour, formants, and vocalTract to make the speaker appear more male (-1...0) or more female (0...+1); 0 = no change.

creakyBreathy hyperparameter for a rough adjustment of voice quality from creaky (-1) to breathy (+1); 0 = no change.

nonlinBalance hyperparameter for regulating the (approximate) proportion of sound with different regimes of pitch effects (none / subharmonics only / subharmonics and jitter), 0% = no noise; 100% = the entire sound has jitter + subharmonics. Ignored if temperature = 0.

nonlinDep hyperparameter for regulating the intensity of subharmonics and jitter, 0 to 100% (50% = jitter and subharmonics are as specified, <50% weaker, >50% stronger). Ignored if temperature = 0.

nonlinRandomWalk a numeric vector specifying the timing of nonlinear regimes: 0 = none, 1 = subharmonics, 2 = subharmonics + jitter + shimmer.

jitterLen duration of stable periods between pitch jumps, ms. Use a low value for harsh noise, a high value for irregular vibrato or shaky voice (anchor format).

jitterDep cycle-to-cycle random pitch variation, semitones (anchor format).

vibratoFreq the rate of regular pitch modulation, or vibrato, Hz (anchor format).

vibratoDep the depth of vibrato, semitones (anchor format).

shimmerDep random variation in amplitude between individual glottal cycles (0 to 100% of original amplitude of each cycle) (anchor format).

shimmerLen duration of stable periods between amplitude jumps, ms. Use a low value for harsh noise, a high value for shaky voice (anchor format).
attackLen duration of fade-in / fade-out at each end of syllables and noise (ms): a vector of length 1 (symmetric) or 2 (separately for fade-in and fade-out)

rolloff basic rolloff from lower to upper harmonics, dB/octave (exponential decay). All rolloff parameters are in anchor format. See getRolloff for more details

rolloffOct basic rolloff changes from lower to upper harmonics (regardless of f0) by rolloffOct dB/oct. For example, we can get steeper rolloff in the upper part of the spectrum

rolloffKHz rolloff changes linearly with f0 by rolloffKHz dB/kHz. For ex., -6 dB/kHz gives a 6 dB steeper basic rolloff as f0 goes up by 1000 Hz

rolloffParab an optional quadratic term affecting only the first rolloffParabHarm harmonics. The middle harmonic of the first rolloffParabHarm harmonics is amplified or dampened by rolloffParab dB relative to the basic exponential decay

rolloffParabHarm the number of harmonics affected by rolloffParab

rolloffExact user-specified exact strength of harmonics: a vector or matrix with one row per harmonic, scale 0 to 1 (overrides all other rolloff parameters)

lipRad the effect of lip radiation on source spectrum, dB/oct (the default of +6 dB/oct produces a high-frequency boost when the mouth is open)

noseRad the effect of radiation through the nose on source spectrum, dB/oct (the alternative to lipRad when the mouth is closed)

mouthOpenThres open the lips (switch from nose radiation to lip radiation) when the mouth is open >mouthOpenThres, 0 to 1

formants either a character string like "aaui" referring to default presets for speaker "M1" or a list of formant times, frequencies, amplitudes, and bandwidths (see ex. below). formants = NA defaults to schwa. Time stamps for formants and mouthOpening can be specified in ms or an any other arbitrary scale. See getSpectralEnvelope for more details

formantDep scale factor of formant amplitude (1 = no change relative to amplitudes in formants)

formantDepStoch the amplitude of additional stochastic formants added above the highest specified formant, dB (only if temperature > 0)

formantWidth = scale factor of formant bandwidth (1 = no change)

vocalTract the length of vocal tract, cm. Used for calculating formant dispersion (for adding extra formants) and formant transitions as the mouth opens and closes. If NULL or NA, the length is estimated based on specified formant frequencies (if any)

subFreq target frequency of subharmonics, Hz (lower than f0, adjusted dynamically so f0 is always a multiple of subFreq) (anchor format)

subDep the width of subharmonic band, Hz. Regulates how quickly the strength of subharmonics fades as they move away from harmonics in f0 stack (anchor format)

shortestEpoch minimum duration of each epoch with unchanging subharmonics regime, in ms

amDep amplitude modulation depth, %. 0: no change; 100: amplitude modulation with amplitude range equal to the dynamic range of the sound (anchor format)

amFreq amplitude modulation frequency, Hz (anchor format)

amShape amplitude modulation shape (-1 to +1, defaults to 0) (anchor format)
soundgen

- **noise**: loudness of turbulent noise (0 dB = as loud as voiced component, negative values = quieter) such as aspiration, hissing, etc (anchor format)
- **formantsNoise**: the same as formants, but for unvoiced instead of voiced component. If NA (default), the unvoiced component will be filtered through the same formants as the voiced component, approximating aspiration noise [h]
- **rolloffNoise**: linear rolloff of the excitation source for the unvoiced component, dB/kHz (anchor format)
- **noiseFlatSpec**: keeps noise spectrum flat to this frequency, Hz
- **noiseAmpRef**: noise amplitude is defined relative to: "f0" = the amplitude of the first partial (fundamental frequency), "source" = the amplitude of the harmonic component prior to applying formants, "filtered" = the amplitude of the harmonic component after applying formants
- **mouth**: mouth opening (0 to 1, 0.5 = neutral, i.e. no modification) (anchor format)
- **ampl**: amplitude envelope (dB, 0 = max amplitude) (anchor format)
- **amplGlobal**: global amplitude envelope spanning multiple syllables (dB, 0 = no change) (anchor format)
- **interpol**: the method of smoothing envelopes based on provided anchors: 'approx' = linear interpolation, 'spline' = cubic spline, 'loess' (default) = polynomial local smoothing function. NB: this does not affect contours for "noise", "glottal", and the smoothing of formants
- **discontThres, jumpThres**: if two anchors are closer in time than discontThres, the contour is broken into segments with a linear transition between these anchors; if anchors are closer than jumpThres, a new section starts with no transition at all (e.g. for adding pitch jumps)
- **samplingRate**: sampling frequency, Hz
- **windowLength**: length of FFT window, ms
- **overlap**: FFT window overlap, %. For allowed values, see istft
- **addSilence**: silence before and after the bout, ms
- **pitchFloor, pitchCeiling**: lower & upper bounds of f0
- **pitchSamplingRate**: sampling frequency of the pitch contour only, Hz. Low values reduce processing time. Set to pitchCeiling for optimal speed or to samplingRate for optimal quality
- **dynamicRange**: dynamic range, dB. Harmonics and noise more than dynamicRange under maximum amplitude are discarded to save computational resources
- **invalidArgAction**: what to do if an argument is invalid or outside the range in permittedValues: 'adjust' = reset to default value, 'abort' = stop execution, 'ignore' = throw a warning and continue (may crash)
- **plot**: if TRUE, plots a spectrogram
- **play**: if TRUE, plays the synthesized sound. In case of errors, try setting another default player for play
savePath full path for saving the output, e.g. '~/Downloads/temp.wav'. If NA (default), doesn't save anything
...
other plotting parameters passed to `spectrogram`

**Value**

Returns the synthesized waveform as a numeric vector.

**Examples**

```r
# NB: GUI for soundgen is available as a Shiny app.
# Type "soundgen_app()" to open it in default browser

playback = c(TRUE, FALSE)[2] # set to TRUE to play back the audio from examples

sound = soundgen(play = playback)
# spectrogram(sound, 16000, osc = TRUE)
# playme(sound)

# Control of intonation, amplitude envelope, formants
s0 = soundgen(
  pitch = c(300, 390, 250),
  ampl = data.frame(time = c(0, 50, 300), value = c(-5, -10, 0)),
  attack = c(10, 50),
  formants = c(600, 900, 2200),
  play = playback
)

# Use the in-built collection of presets:
# names(presets) # speakers
# names(presets$Chimpanzee) # calls per speaker
s1 = eval(parse(text = presets$Chimpanzee$Scream_conflict)) # screaming chimp
# playme(s1)

s2 = eval(parse(text = presets$F1$Scream)) # screaming woman
# playme(s2)

## Not run:
# unless temperature is 0, the sound is different every time
for (i in 1:3) sound = soundgen(play = playback, temperature = .2)

# Bouts versus syllables. Compare:
sound = soundgen(formants = 'uai', repeatBout = 3, play = playback)
sound = soundgen(formants = 'uai', nSyl = 3, play = playback)

# Intonation contours per syllable and globally:
sound = soundgen(nSyl = 5, sylLen = 200, pauseLen = 140,
  play = playback, pitch = data.frame(
    time = c(0, 0.65, 1), value = c(977, 1540, 826)),
  pitchGlobal = data.frame(time = c(0, .5, 1), value = c(-6, 7, 0)))

# Subharmonics in sidebands (noisy scream)
sound = soundgen (nonlinBalance = 100, subFreq = 75, subDep = 130,
pitch = data.frame(
```
soundgen_app

Description

Starts a shiny app, which provides an interactive wrapper to soundgen

Usage

soundgen_app()

spectrogram

Description

Produces the spectrogram of a sound using short-term Fourier transform. This is a simplified version of spectro with fewer plotting options, but with added routines for noise reduction, smoothing in time and frequency domains, and controlling contrast and brightness. It also provides an options to plot the oscillogram on a dB scale.
Usage

```r
spectrogram(x, samplingRate = NULL, dynamicRange = 80,
windowLength = 50, step = NULL, overlap = 70, wn = "gaussian",
zp = 0, normalize = TRUE, smoothFreq = 0, smoothTime = 0,
qTime = 0, percentNoise = 10, noiseReduction = 0, contrast = 0.2,
brightness = 0, method = c("spectrum", "spectralDerivative")[1],
output = c("none", "original", "processed")[1], ylim = NULL,
plot = TRUE, osc = FALSE, osc_dB = FALSE, heights = c(3, 1),
colorTheme = c("bw", "seewave", "...")[1], xlab = "Time, ms",
ylab = "Frequency, KHz", mar = c(5.1, 4.1, 4.1, 2), main = "",
frameBank = NULL, duration = NULL, ...)```

Arguments

- `x` path to a .wav or .mp3 file or a vector of amplitudes with specified `samplingRate`
- `samplingRate` sampling rate of `x` (only needed if `x` is a numeric vector, rather than an audio file)
- `dynamicRange` dynamic range, dB. All values more than one `dynamicRange` under maximum are treated as zero
- `windowLength` length of FFT window, ms
- `step` you can override `overlap` by specifying FFT step, ms
- `overlap` overlap between successive FFT frames, %
- `wn` window type: gaussian, hanning, hamming, bartlett, rectangular, blackman, flat-top
- `zp` window length after zero padding, points
- `normalize` if TRUE, scales input prior to FFT
- `smoothFreq`, `smoothTime` length of the window, in data points (0 to +inf), for calculating a rolling median. Applies median smoothing to spectrogram in frequency and time domains, respectively
- `qTime` the quantile to be subtracted for each frequency bin. For ex., if `qTime = 0.5`, the median of each frequency bin (over the entire sound duration) will be calculated and subtracted from each frame (see examples)
- `percentNoise` percentage of frames (0 to 100%) used for calculating noise spectrum
- `noiseReduction` how much noise to remove (0 to +inf, recommended 0 to 2). 0 = no noise reduction, 2 = strong noise reduction: `spectrum - (noiseReduction * noiseSpectrum)`, where `noiseSpectrum` is the average spectrum of frames with entropy exceeding the quantile set by `percentNoise`
- `contrast` spectrum is exponentiated by contrast (-inf to +inf, recommended -1 to +1). Contrast >0 increases sharpness, <0 decreases sharpness
- `brightness` how much to “lighten” the image (>0 = lighter, <0 = darker)
- `method` plot spectrum (‘spectrum’) or spectral derivative (‘spectralDerivative’) or denoised and/or smoothed spectrogram (‘processed’)
- `output` specifies what to return: nothing (‘none’), unmodified spectrogram (‘original’), or denoised and/or smoothed spectrogram (‘processed’).
```r
spectrogram

xlim frequency range to plot, kHz (defaults to 0 to Nyquist frequency)
plot should a spectrogram be plotted? TRUE / FALSE
osc, osc_db should an oscillogram be shown under the spectrogram? TRUE / FALSE. If 'osc_db', the oscillogram is displayed on a dB scale. See osc_db for details
heights a vector of length two specifying the relative height of the spectrogram and the oscillogram
colorTheme black and white ('bw'), as in seewave package ('seewave'), or any palette from palette such as 'heat.colors', 'cm.colors', etc
xlab, ylab, main, mar graphical parameters
frameBank ignore (only needed for pitch tracking)
duration ignore (only needed for pitch tracking)
... other graphical parameters passed to seewave::filled.contour.modifR

Value

Returns nothing (if output = 'none'), absolute - not power! - spectrum (if output = 'original'), denoised and/or smoothed spectrum (if output = 'processed'), or spectral derivatives (if method = 'spectralDerivative') as a matrix of real numbers.

Examples

# synthesize a sound 1 s long, with gradually increasing hissing noise
sound = soundgen(syllen = 1000, temperature = 0.001, noise = list(
    time = c(0, 1300), value = c(-40, 0)), formantsNoise = list(
    f1 = list(freq = 5000, width = 10000)))
# playme(sound, samplingRate = 16000)

# basic spectrogram
spectrogram(sound, samplingRate = 16000)

## Not run:
# change dynamic range
spectrogram(sound, samplingRate = 16000, dynamicRange = 40)
spectrogram(sound, samplingRate = 16000, dynamicRange = 120)

# add an oscillogram
spectrogram(sound, samplingRate = 16000, osc = TRUE)

# oscillogram on a dB scale, same height as spectrogram
spectrogram(sound, samplingRate = 16000,
    osc_db = TRUE, heights = c(1, 1))

# broad-band instead of narrow-band
spectrogram(sound, samplingRate = 16000, windowLength = 5)

# focus only on values in the upper 5% for each frequency bin
spectrogram(sound, samplingRate = 16000, qTime = 0.95)
```
# detect 10% of the noisiest frames based on entropy and remove the pattern
# found in those frames (in this cases, breathing)
spectrogram(sound, samplingRate = 16000, noiseReduction = 1.1,
  brightness = -2)  # white noise attenuated

# apply median smoothing in both time and frequency domains
spectrogram(sound, samplingRate = 16000, smoothFreq = 5,
  smoothTime = 5)

# increase contrast, reduce brightness
spectrogram(sound, samplingRate = 16000, contrast = 1, brightness = -1)

# add bells and whistles
spectrogram(sound, samplingRate = 16000, osc = TRUE, noiseReduction = 1.1,
  brightness = -1, colorTheme = 'heat.colors',
  ylim = c(0, 5), cex.lab = .75, main = 'My spectrogram')

## End(Not run)

---

**spectrogramFolder**  
*Save spectrograms per folder*

### Description

Creates spectrograms of all wav/mp3 files in a folder and saves them as .png files in the same folder. This is a lot faster than running `analyzeFolder` if you don’t need pitch tracking. By default it also creates an html file with a list of audio files and their spectrograms in the same folder. If you open it in a browser that supports playing .wav and/or .mp3 files (e.g. Firefox or Chrome), you can view the spetrograms and click on them to play each sound. Unlike `analyzeFolder`, spectrogramFolder supports plotting both a spectrogram and an oscillogram if `osc = TRUE`.

### Usage

```r
spectrogramFolder(myfolder, htmlPlots = TRUE, verbose = TRUE,
  windowLength = 50, step = NULL, overlap = 50, wn = "gaussian",
  zp = 0, ylim = NULL, osc = TRUE, xlab = "Time, ms",
  ylab = "kHz", width = 900, height = 500, units = "px",
  res = NA, ...)
```

### Arguments

- **myfolder** full path to the folder containing wav/mp3 files
- **htmlPlots** if TRUE, saves an html file with clickable plots
- **verbose** if TRUE, reports progress and estimated time left
- **windowLength** length of FFT window, ms
- **step** you can override overlap by specifying FFT step, ms
- **overlap** overlap between successive FFT frames, %
window type: gaussian, hanning, hamming, bartlett, rectangular, blackman, flat-top
window length after zero padding, points
frequency range to plot, kHz (defaults to 0 to Nyquist frequency)
should an oscillogram be shown under the spectrogram? TRUE/ FALSE. If
'osc_dB', the oscillogram is displayed on a dB scale. See osc_db for details
graphical parameters
parameters passed to png if the plot is saved
parameters passed to png if the plot is saved
parameters passed to png if the plot is saved
other parameters passed to spectrogram

Examples

## Not run:
spectrogramFolder(
  '~/Downloads/temp',
  windowLength = 40, overlap = 75,  # spectrogram pars
  width = 1500, height = 900,  # passed to png()
  osc = TRUE, osc_dB = TRUE, heights = c(1, 1)
)
# note that the folder now also contains an html file with clickable plots

## End(Not run)

Description

Calculates the self-similarity matrix and novelty vector of a sound.

Usage

```r
ssm(x, samplingRate = NULL, windowLength = 40, overlap = 75,
    step = NULL, ssmWin = 40, maxFreq = NULL, nBands = NULL,
    MFCC = 2:13, input = c("mfcc", "audiogram", "spectrum")\[1],
    norm = FALSE, simil = c("cosine", "cor")\[1], returnSSM = TRUE,
    kernelsLen = 200, kernelSD = 0.2, padWith = 0, plot = TRUE,
    heights = c(2, 1), specPars = list(levels = seq(0, 1, length = 30),
    color.palette = seewave::spectro.colors, xlab = "Time, s", ylab = "kHz",
    ylim = c(0, maxFreq/1000)), ssmPars = list(levels = seq(0, 1, length = 30),
    color.palette = seewave::spectro.colors, xlab = "Time, s", ylab =
    "Time, s", main = "Self-similarity matrix"), noveltyPars = list(type =
    "b", pch = 16, col = "black", lwd = 3))
```
### Arguments

- **x**
  - path to a .wav file or a vector of amplitudes with specified samplingRate
- **samplingRate**
  - sampling rate of x (only needed if x is a numeric vector, rather than a .wav file)
- **windowLength**
  - length of FFT window, ms
- **overlap**
  - overlap between successive FFT frames, %
- **step**
  - you can override overlap by specifying FFT step, ms
- **ssmWin**
  - window for averaging SSM, ms
- **maxFreq**
  - highest band edge of mel filters, Hz. Defaults to samplingRate / 2. See \texttt{melfcc}
- **nBands**
  - number of warped spectral bands to use. Defaults to 100 * windowLength / 20. See \texttt{melfcc}
- **MFCC**
  - which mel-frequency cepstral coefficients to use; defaults to 2:13
- **input**
  - either MFCCs ("cepstrum") or mel-filtered spectrum ("audiogram")
- **norm**
  - if TRUE, the spectrum of each STFT frame is normalized
- **simil**
  - method for comparing frames: "cosine" = cosine similarity, "cor" = Pearson’s correlation
- **returnSSM**
  - if TRUE, returns the SSM
- **kernelLen**
  - length of checkerboard kernel for calculating novelty, ms (larger values favor global vs. local novelty)
- **kernelSD**
  - SD of checkerboard kernel for calculating novelty
- **padWith**
  - how to treat edges when calculating novelty: NA = treat sound before and after the recording as unknown, 0 = treat it as silence
- **plot**
  - if TRUE, plots the SSM
- **heights**
  - relative sizes of the SSM and spectrogram/novelty plot
- **specPars**
  - graphical parameters passed to \texttt{seewave::filled.contour.modif2} and affecting the spectrogram
- **ssmPars**
  - graphical parameters passed to \texttt{seewave::filled.contour.modif2} and affecting the plot of SSM
- **noveltyPars**
  - graphical parameters passed to \texttt{lines} and affecting the novelty contour

### Value

If \texttt{returnSSM} is TRUE, returns a list of two components: Ssm contains the self-similarity matrix, and $novelty$ contains the novelty vector. If \texttt{returnSSM} is FALSE, only produces a plot.

### References

Examples

```r
sound = c(soundgen(), soundgen(nSyl = 4, syllen = 50, pauseLen = 70, formants = NA, pitch = c(500, 330)))
# playme(sound)
ml = ssm(sound, samplingRate = 16000,
        input = 'audiogram', simil = 'cor', norm = FALSE,
        ssmWin = 10, kernelLen = 150)  # detailed, local features
## Not run:
m2 = ssm(sound, samplingRate = 16000,
        input = 'mfcc', simil = 'cosine', norm = TRUE,
        ssmWin = 50, kernelLen = 600)  # more global
# plot(m2$novelty, type='b')  # use for peak detection, etc
## End(Not run)
```
Index

*Topic datasets
  defaults, 21
  notesDict, 55
  permittedValues, 59
  pitchManual, 60
  presets, 61
  segmentManual, 69

addFormants, 3
addVectors, 5
analyze, 6, 12, 36
analyzeFolder, 12, 55, 78
approx, 35
audspec, 32

beat, 16, 27

compareSounds, 17
contour, 48
crossFade, 19

defaults, 21
dtw, 18, 46

env, 65, 68
estimateVTL, 21, 63

fade, 22
fart, 23, 27
filled.contour.modif2, 48
findformants, 7, 13
flatEnv, 25, 36, 37, 54

gaussianSmooth2D, 26, 47
generateNoise, 27
getEntropy, 29
getIntegerRandomWalk, 30
getLoudness, 6, 10, 31, 33, 54
getLoudnessFolder, 33
getRandomWalk, 31, 34
getRMS, 35, 37, 54

generateNoise, 27
getEntropy, 29
getIntegerRandomWalk, 30
getLoudness, 6, 10, 31, 33, 54
getLoudnessFolder, 33
getRandomWalk, 31, 34
getRMS, 35, 37, 54

getRMSFolder, 37
getRolloff, 38, 38, 72
getSmoothContour, 40
getSpectralEnvelope, 3, 42, 72
HzToSemitones, 44, 69

istft, 4, 27, 73

lines, 80

matchPars, 17, 45
melfcc, 80
modulationSpectrum, 47, 51
modulationSpectrumFolder, 47, 50
morph, 52

normalizeFolder, 53
notesDict, 55

optim, 9, 15, 55, 56
optimizePars, 55, 66
osc_db, 58, 77, 79

palette, 48, 52, 77
permittedValues, 59
pitchManual, 60
play, 17, 24, 27, 60, 73
playme, 60
plot, 66, 68
png, 9, 15, 48, 52, 66, 68, 79
presets, 61

reportTime, 61
rnorm, 34

schwa, 21, 62
segment, 64, 68
segmentFolder, 55, 67
segmentManual, 69
semitonesToHz, 69
INDEX

soundgen, 3, 4, 16, 24, 27, 40, 45, 46, 52, 70, 75
soundgen_app, 75
spec.fft, 47
spectro, 75
spectrogram, 9, 15, 32, 47, 52, 74, 75, 79
spectrogramFolder, 78
spline, 35
ssm, 79