Package 'soundgen'

July 23, 2025

Type Package

Title Sound Synthesis and Acoustic Analysis

Version 2.7.3

Date 2025-07-17

Maintainer Andrey Anikin <andrey.anikin@cogsci.se>

URL http://cogsci.se/soundgen.html

Description Performs parametric synthesis of sounds with harmonic and noise components such as animal vocalizations or human voice. Also offers tools for audio manipulation and acoustic analysis, including pitch tracking, spectral analysis, audio segmentation, pitch and formant shifting, etc. Includes four interactive web apps for synthesizing and annotating audio, manually correcting pitch contours, and measuring formant frequencies. Reference: Anikin (2019) <doi:10.3758/s13428-018-1095-7>.

License GPL (>= 2)

Encoding UTF-8

LazyData true

Imports stats (>= 4.0.0), graphics, utils, tuneR, seewave (>= 2.1.6), zoo, mvtnorm, dtw, phonTools, signal, shiny, shinyjs, foreach, doParallel, nonlinearTseries, data.table

Depends R (>= 4.0), shinyBS

RoxygenNote 7.3.2

NeedsCompilation no

Author Andrey Anikin [aut, cre]

Repository CRAN

Date/Publication 2025-07-17 10:40:02 UTC

Contents

addAM *Add amplitude modulation*

Description

Adds sinusoidal or logistic amplitude modulation to a sound. This produces additional harmonics in the spectrum at ±am_freq around each original harmonic and makes the sound rough. The optimal frequency for creating a perception of roughness is ~70 Hz (Fastl & Zwicker "Psychoacoustics"). Sinusoidal AM creates a single pair of new harmonics, while non-sinusoidal AM creates more extra harmonics (see examples).

Usage

```
addAM(
  x,
  samplingRate = NULL,
  amDep = 25,
  amFreq = 30,
  amType = c("logistic", "sine")[1],
  amShape = 0,
  invalidArgAction = c("adjust", "abort", "ignore")[1],
  plot = FALSE,
  play = FALSE,
  saveAudio = NULL,
  reportEvery = NULL,
  cores = 1
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
amDep	amplitude modulation (AM) depth, %. 0: no change; 100: AM with amplitude range equal to the dynamic range of the sound (anchor format)
amFreq	AM frequency, Hz (anchor format)
amType	"sine" = sinusoidal, "logistic" = logistic (default)
amShape	ignore if amType = "sine", otherwise determines the shape of non-sinusoidal AM: 0 = ~sine, -1 = notches, +1 = clicks (anchor format)
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 the amplitude modulation
play	if TRUE, plays the processed audio

saveAudio	full (!) path to folder for saving the processed audio; NULL = don't save, '' = same as input folder (NB: overwrites the originals!)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing

Examples

```
sound1 = soundgen(pitch = c(200, 300), addSilence = 0)
s1 = addAM(sound1, 16000, amDep = c(0, 50, 0), amFreq = 75, plot = TRUE)
# playme(s1)
## Not run:
# Parameters can be specified as in the soundgen() function, eg:
s2 = addAM(sound1, 16000,
         amDep = list(time = c(0, 50, 52, 200, 201, 300),
                      value = c(0, 0, 35, 25, 0, 0)),
         plot = TRUE, play = TRUE)

# Sinusoidal AM produces exactly 2 extra harmonics at ±am_freq
# around each f0 harmonic:
s3 = addAM(sound1, 16000, amDep = 30, amFreq = c(50, 80),
           amType = 'sine', plot = TRUE, play = TRUE)
spectrogram(s3, 16000, windowLength = 150, ylim = c(0, 2))

# Non-sinusoidal AM produces multiple new harmonics,
# which can resemble subharmonics...
s4 = addAM(sound1, 16000, amDep = 70, amFreq = 50, amShape = -1,
           plot = TRUE, play = TRUE)
spectrogram(s4, 16000, windowLength = 150, ylim = c(0, 2))

# ...but more often look like sidebands
sound3 = soundgen(sylLen = 600, pitch = c(800, 1300, 1100), addSilence = 0)
s5 = addAM(sound3, 16000, amDep = c(0, 30, 100, 40, 0),
           amFreq = 105, amShape = -.3,
           plot = TRUE, play = TRUE)
spectrogram(s5, 16000, ylim = c(0, 5))

# Feel free to add AM stochastically:
s6 = addAM(sound1, 16000,
           amDep = rnorm(10, 40, 20), amFreq = rnorm(20, 70, 20),
           plot = TRUE, play = TRUE)
spectrogram(s6, 16000, windowLength = 150, ylim = c(0, 2))

# If am_freq is locked to an integer ratio of f0, we can get subharmonics
# For ex., here is with pitch 400-600-400 Hz (soundgen interpolates pitch
# on a log scale and am_freq on a linear scale, so we align them by extracting
# a long contour on a log scale for both)
con = getSmoothContour(anchors = c(400, 600, 400),
                       len = 20, thisIsPitch = TRUE)
s = soundgen(sylLen = 1500, pitch = con, amFreq = con/3, amDep = 30,
             plot = TRUE, play = TRUE, ylim = c(0, 3))
```

```
# Process all files in a folder and save the modified audio
addAM('~/Downloads/temp', saveAudio = '~/Downloads/temp/AM',
      amFreq = 70, amDep = c(0, 50))

## End(Not run)
```

---

addFormants *Add formants*

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. Instead of formants, any arbitrary spectral filtering function can be applied using the spectralEnvelope argument (eg for a low/high/bandpass filter).

Usage

```
addFormants(
  x,
  samplingRate = NULL,
  formants = NULL,
  spectralEnvelope = NULL,
  action = c("add", "remove")[1],
  dB = NULL,
  specificity = 1,
  zFun = NULL,
  vocalTract = NA,
  formantDep = 1,
  formantDepStoch = 1,
  formantWidth = 1,
  formantCeiling = 2,
  lipRad = 6,
  noseRad = 4,
  mouthOpenThres = 0,
  mouth = NA,
  temperature = 0.025,
  formDrift = 0.3,
  formDisp = 0.2,
  smoothing = list(),
  windowLength_points = 800,
  overlap = 75,
  normalize = c("max", "orig", "none")[1],
  play = FALSE,
  saveAudio = NULL,
```

```
  reportEvery = NULL,
  cores = 1,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling frequency, Hz
formants	either a character string referring to default presets for speaker "M1" (implemented: "aoieu0") or a list of formant times, frequencies, amplitudes, and bandwidths (see examples). NA or NULL means no formants, only lip radiation. Time stamps for formants and mouthOpening can be specified in ms relative to sylLen or on a scale of [0, 1]. 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)
dB	if NULL (default), the spectral envelope is applied on the original scale; otherwise, it is set to range from 1 to 10 ^ (dB / 20)
specificity	a way to sharpen or blur the spectral envelope (spectrum ^ specificity) : 1 = no change, >1 = sharper, <1 = blurred
zFun	(optional) an arbitrary function to apply to the spectrogram prior to iSTFT, where "z" is the spectrogram - a matrix of complex values (see examples)
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 (anchor format)
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)
formantCeiling	frequency to which stochastic formants are calculated, in multiples of the Nyquist frequency; increase up to ~10 for long vocal tracts to avoid losing energy in the upper part of the spectrum
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
mouth	mouth opening (0 to 1, 0.5 = neutral, i.e. no modification) (anchor format)
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
smoothing	a list of parameters passed to getSmoothContour to control the interpolation and smoothing of contours: interpol (approx / spline / loess), loessSpan, discontThres, jumpThres
windowLength_points	length of FFT window, points
overlap	FFT window overlap, %. For allowed values, see istft
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude, "none" = no normalization
play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
saveAudio	path + filename for saving the output, e.g. '~/Downloads/temp.wav'. If NULL = doesn't save
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
...	other plotting parameters passed to spectrogram

Details

Algorithm: converts input from a time series (time domain) to a spectrogram (frequency domain) through short-time 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 for voice synthesis in soundgen, but it can also be applied to a recording.

See Also

getSpectralEnvelope transplantFormants soundgen

Examples

```
sound = c(rep(0, 1000), runif(8000) * 2 - 1, 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, samplingRate = 16000,
                             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,
                                 samplingRate = 16000,
                                 formants = c(900, 1300),
                                 action = 'remove')
# playme(sound_inverse_filt)
# spectrogram(sound_inverse_filt, samplingRate = 16000)

## Not run:
## Perform some user-defined manipulation of the spectrogram with zFun
# Ex.: noise removal - silence all bins under threshold,
# say -0 dB below the max value
s_noisy = soundgen(sylLen = 200, addSilence = 0,
                   noise = list(time = c(-100, 300), value = -20))
spectrogram(s_noisy, 16000)
# playme(s_noisy)
zFun = function(z, cutoff = -50) {
  az = abs(z)
  thres = max(az) * 10 ^ (cutoff / 20)
  z[which(az < thres)] = 0
  return(z)
}
s_denoised = addFormants(s_noisy, samplingRate = 16000,
                         formants = NA, zFun = zFun, cutoff = -40)
spectrogram(s_denoised, 16000)
# playme(s_denoised)

# If neither formants nor spectralEnvelope are defined, only lipRad has an effect
# For ex., we can boost low frequencies by 6 dB/oct
noise = runif(8000)
noise1 = addFormants(noise, 16000, lipRad = -6)
seewave::meanspec(noise1, f = 16000, dB = 'max0')

# Arbitrary spectra can be defined with spectralEnvelope. For ex., we can
# have a flat spectrum up to 2 kHz (Nyquist / 4) and -3 dB/kHz above:
freqs = seq(0, 16000 / 2, length.out = 100)
n = length(freqs)
idx = (n / 4):n
sp_dB = c(rep(0, n / 4 - 1), (freqs[idx] - freqs[idx[1]]) / 1000 * (-3))
plot(freqs, sp_dB, type = 'b')
noise2 = addFormants(noise, 16000, lipRad = 0, spectralEnvelope = 10 ^ (sp_dB / 20))
seewave::meanspec(noise2, f = 16000, dB = 'max0')

## Use the spectral envelope of an existing recording (bleating of a sheep)
# (see also the same example with noise as source in ?generateNoise)
# (NB: this can also be achieved with a single call to transplantFormants)
data(sheep, package = 'seewave')  # import a recording from seewave
sound_orig = as.numeric(scale(sheep@left))
```

```
samplingRate = sheep@samp.rate
sound_orig = sound_orig / max(abs(sound_orig))  # range -1 to +1
# playme(sound_orig, samplingRate)

# get a few pitch anchors to reproduce the original intonation
pitch = analyze(sound_orig, samplingRate = samplingRate,
  pitchMethod = c('autocor', 'dom'))$detailed$pitch
pitch = pitch[!is.na(pitch)]

# extract a frequency-smoothed version of the original spectrogram
# to use as filter
specEnv_bleating = spectrogram(sound_orig, windowLength = 5,
 samplingRate = samplingRate, output = 'original', plot = FALSE)
# image(t(log(specEnv_bleating)))

# Synthesize source only, with flat spectrum
sound_unfilt = soundgen(sylLen = 2500, pitch = pitch,
  rolloff = 0, rolloffOct = 0, rolloffKHz = 0,
  temperature = 0, jitterDep = 0, subDep = 0,
  formants = NULL, lipRad = 0, samplingRate = samplingRate,
  invalidArgAction = 'ignore')  # prevent soundgen from increasing samplingRate
# playme(sound_unfilt, samplingRate)
# seewave::meanspec(sound_unfilt, f = samplingRate, dB = 'max0')  # ~flat

# Force spectral envelope to the shape of target
sound_filt = addFormants(sound_unfilt, formants = NULL,
  spectralEnvelope = specEnv_bleating, samplingRate = samplingRate)
# playme(sound_filt, samplingRate)  # playme(sound_orig, samplingRate)
# spectrogram(sound_filt, samplingRate)  # spectrogram(sound_orig, samplingRate)

# The spectral envelope is now similar to the original recording. Compare:
par(mfrow = c(1, 2))
seewave::meanspec(sound_orig, f = samplingRate, dB = 'max0', alim = c(-50, 20))
seewave::meanspec(sound_filt, f = samplingRate, dB = 'max0', alim = c(-50, 20))
par(mfrow = c(1, 1))
# NB: but the source of excitation in the original is actually a mix of
# harmonics and noise, while the new sound is purely tonal

## End(Not run)
```

`addPitchJumps` *Add pitch jumps*

Description

Internal soundgen function

Usage

```
addPitchJumps(pitch, magn, nj = 1, prop = 0.1, plot = FALSE)
```

Arguments

pitch	numeric vector of f0 values over time (any step is OK)
magn	magnitude of jump(s) in semitones, a numeric vector of length 1 or nj
nj	number of jump pairs = affected segments (e.g., a single fragment is transposed if nj = 1)
prop	duration of transposed episode(s) a a proportion of the total voiced duration (length of pitch)
plot	if TRUE, plots the original and modified contours

Details

Adds random discontinuities (jumps) to a pitch contour in a manner that shifts a segment of pitch contour up or down. Careful when adding several jumps: one can land on top of another, and it gets rather weird rather quickly.

Examples

```
pitch = getSmoothContour(c(100, 350, 320, 110), len = 100, interpol = 'loess')
addPitchJumps(pitch, magn = runif(1, 3, 12), plot = TRUE)
addPitchJumps(pitch, magn = c(6, 1), nj = 2, plot = TRUE)
addPitchJumps(pitch, magn = 3, nj = 5, plot = TRUE)

pitch2 = c(rep(NA, 10), pitch[1:50], rep(NA, 25), pitch[51:100], rep(NA, 17))
addPitchJumps(pitch2, magn = c(6, 1), nj = 2, plot = TRUE)
```

---

addVectors *Add overlapping vectors*

Description

Adds 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

```
addVectors(v1, v2, insertionPoint = 1L, 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

See Also

soundgen

Examples

```
v1 = 1:6
v2 = rep(100, 3)
addVectors(v1, v2, insertionPoint = 5, normalize = FALSE)
addVectors(v1, v2, insertionPoint = -4, normalize = FALSE)
addVectors(v1, rep(100, 15), 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)
addVectors(1:6, 3:6, insertionPoint = 3, normalize = FALSE)
```

---

analyze *Acoustic analysis*

Description

Acoustic analysis of one or more sounds: pitch tracking, basic spectral characteristics, formants, estimated loudness (see getLoudness), roughness (see modulationSpectrum), novelty (see ssm), etc. The default values of arguments are optimized for human non-linguistic vocalizations. See vignette('acoustic_analysis', package = 'soundgen') for details. The defaults and reasonable ranges of all arguments can be found in defaults_analyze. For high-precision work, first extract and manually correct pitch contours with pitch_app, PRAAT, or whatever, and then run analyze(pitchManual = ...) with these manual contours.

Usage

```
analyze(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  dynamicRange = 80,
  silence = 0.04,
  windowLength = 50,
  step = windowLength/2,
  overlap = 50,
  specType = c("spectrum", "reassign", "spectralDerivative")[1],
  wn = "gaussian",
  zp = 0,
  cutFreq = NULL,
```

```
  nFormants = 3,
  formants = list(),
  loudness = list(SPL_measured = 70),
  roughness = list(msType = "1D", windowLength = 25, step = 2, amRes = 5),
  novelty = list(input = "melspec", kernelLen = 100),
  pitchMethods = c("dom", "autocor"),
  pitchManual = NULL,
  entropyThres = 0.6,
  pitchFloor = 75,
  pitchCeiling = 3500,
  priorMean = 300,
  priorSD = 6,
  priorAdapt = TRUE,
  nCands = 1,
  minVoicedCands = NULL,
  pitchDom = list(domThres = 0.1, domSmooth = 220),
  pitchAutocor = list(autocorThres = 0.7, autocorSmooth = 7, autocorUpsample = 25,
    autocorBestPeak = 0.975, interpol = "sinc"),
  pitchCep = list(cepThres = 0.75, cepZp = 0),
  pitchSpec = list(specThres = 0.05, specPeak = 0.25, specHNRslope = 0.8, specSmooth =
    150, specMerge = 0.1, specSinglePeakCert = 0.4, specRatios = 3),
  pitchHps = list(hpsNum = 5, hpsThres = 0.1, hpsNorm = 2, hpsPenalty = 2),
  pitchZc = list(zcThres = 0.1, zcWin = 5),
  harmHeight = list(harmThres = 3, harmTol = 0.25, harmPerSel = 5),
  subh = list(method = c("cep", "pitchCands", "harm")[1], nSubh = 5, tol = 0.05, nHarm =
    5, harmThres = 12, harmTol = 0.25, amRange = c(10, 200)),
  flux = list(thres = 0.15, smoothWin = 100),
  amRange = c(10, 200),
  fmRange = c(5, 1000/step/2),
  shortestSyl = 20,
  shortestPause = 60,
  interpol = list(win = 75, tol = 0.3, cert = 0.3),
  pathfinding = c("none", "fast", "slow")[2],
  annealPars = list(maxit = 5000, temp = 1000),
  certWeight = 0.5,
  snakeStep = 0,
  snakePlot = FALSE,
  smooth = 1,
  smoothVars = c("pitch", "dom"),
  summaryFun = c("mean", "median", "sd"),
  invalidArgAction = c("adjust", "abort", "ignore")[1],
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  osc = "linear",
  showLegend = TRUE,
  savePlots = NULL,
  pitchPlot = list(col = rgb(0, 0, 1, 0.75), lwd = 3, showPrior = TRUE),
```

```
  extraContour = NULL,
  ylim = NULL,
  xlab = "Time",
  ylab = NULL,
  main = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
silence	(0 to 1 as proportion of max amplitude) frames with RMS amplitude below `silence * max_ampl` adjusted by `scale` are not analyzed at all.
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override `overlap` by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
specType	plot the original FFT ('spectrum'), reassigned spectrogram ('reassigned'), or spectral derivative ('spectralDerivative')
wn	window type accepted by ftwindow, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle
zp	window length after zero padding, points
cutFreq	if specified, spectral descriptives (peakFreq, specCentroid, specSlope, and quartiles) are calculated only between `cutFreq[1]` and `cutFreq[2]`, Hz. If a single number is given, analyzes frequencies from 0 to `cutFreq`. For ex., when analyzing recordings with varying sampling rates, set to half the lowest sampling rate to make the spectra more comparable. Note that "entropyThres" applies only to this frequency range, which also affects which frames will not be analyzed with pitchAutocor.
nFormants	the number of formants to extract per STFT frame (0 = no formant analysis, NULL = as many as possible)

formants — a list of arguments passed to findformants - an external function called to perform LPC analysis

loudness — a list of parameters passed to getLoudness for measuring subjective loudness, namely SPL_measured, Pref, spreadSpectrum. NULL = skip loudness analysis

roughness — a list of parameters passed to modulationSpectrum for measuring roughness. NULL = skip roughness analysis

novelty — a list of parameters passed to ssm for measuring spectral novelty. NULL = skip novelty analysis

pitchMethods — methods of pitch estimation to consider for determining pitch contour: 'autocor' = autocorrelation (~PRAAT), 'cep' = cepstral, 'spec' = spectral (~BaNa), 'dom' = lowest dominant frequency band, 'hps' = harmonic product spectrum, NULL = no pitch analysis

pitchManual — manually corrected pitch contour. For a single sound, provide a numeric vector of any length. For multiple sounds, provide a dataframe with columns "file" and "pitch" (or path to a csv file) as returned by pitch_app, ideally with the same windowLength and step as in current call to analyze. A named list with pitch vectors per file is also OK (eg as returned by pitch_app)

entropyThres — pitch tracking is only performed for frames with Weiner entropy below entropyThres, but other spectral descriptives are still calculated (NULL = analyze everything)

pitchFloor, pitchCeiling — absolute bounds for pitch candidates (Hz)

priorMean, priorSD — specifies the mean (Hz) and standard deviation (semitones) of gamma distribution describing our prior knowledge about the most likely pitch values for this file. For ex., priorMean = 300, priorSD = 6 gives a prior with mean = 300 Hz and SD = 6 semitones (half an octave). To avoid using any priors, set priorMean = NA, priorAdapt = FALSE

priorAdapt — adaptive second-pass prior: if TRUE, optimal pitch contours are estimated first with a prior determined by priorMean, priorSD, and then with a new prior adjusted according to this first-pass pitch contour

nCands — maximum number of pitch candidates per method, normally 1...4 (except for dom, which returns at most one candidate per frame)

minVoicedCands — minimum number of pitch candidates that have to be defined to consider a frame voiced (if NULL, defaults to 2 if dom is among other candidates and 1 otherwise)

pitchDom — a list of control parameters for pitch tracking using the lowest dominant frequency band or "dom" method; see details and ?soundgen:::getDom

pitchAutocor — a list of control parameters for pitch tracking using the autocorrelation or "autocor" method; see details and ?soundgen:::getPitchAutocor

pitchCep — a list of control parameters for pitch tracking using the cepstrum or "cep" method; see details and ?soundgen:::getPitchCep

pitchSpec — a list of control parameters for pitch tracking using the BaNa or "spec" method; see details and ?soundgen:::getPitchSpec

`pitchHps` a list of control parameters for pitch tracking using the harmonic product spectrum or "hps" method; see details and `?soundgen:::getPitchHps`

`pitchZc` a list of control parameters for pitch tracking based on zero crossings in bandpass-filtered audio or "zc" method; see `getPitchZc`

`harmHeight` a list of control parameters for estimating how high harmonics reach in the spectrum; see details and `?soundgen:::harmHeight`

`subh` a list of control parameters for estimating the strength of subharmonics per frame - that is, spectral energy at integer ratios of f0: see `?soundgen:::getSHR`

`flux` a list of control parameters for calculating feature-based flux (not spectral flux) passed to `getFeatureFlux`

`amRange` target range of frequencies for amplitude modulation, Hz: a vector of length 2 (affects both `amMsFreq` and `amEnvFreq`)

`fmRange` target range of frequencies for analyzing frequency modulation, Hz (`fmFreq`): a vector of length 2

`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): large value = interpolate and merge, small value = treat as separate syllables separated by an unvoiced gap

`interpol` a list of parameters (currently `win, tol, cert`) passed to `soundgen:::pathfinder` for interpolating missing pitch candidates (NULL = no interpolation)

`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 = 0`

`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 = 0`

`summaryFun` functions used to summarize each acoustic characteristic, eg "c('mean', 'sd')"; user-defined functions are fine (see examples); NAs are omitted automatically for mean/median/sd/min/max/range/sum, otherwise take care of NAs yourself

`invalidArgAction` what to do if an argument is invalid or outside the range in `defaults_analyze`: 'adjust' = reset to default value, 'abort' = stop execution, 'ignore' = throw a warning and continue (may crash)

`reportEvery` when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)

cores	number of cores for parallel processing
plot	if TRUE, produces a spectrogram with pitch contour overlaid
osc	"none" = no oscillogram; "linear" = on the original scale; "dB" = in decibels
showLegend	if TRUE, adds a legend with pitch tracking methods
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
pitchPlot	a list of graphical parameters for displaying the final pitch contour. Set to `list(type = 'n')` to suppress
extraContour	name of an output variable to overlap on the pitch contour plot, eg 'peakFreq' or 'loudness'; can also be a list with extra graphical parameters, eg `extraContour = list(x = 'harmHeight', col = 'red')`
ylim	frequency range to plot, kHz (defaults to 0 to Nyquist frequency). NB: still in kHz, even if ySCale = bark, mel, or ERB
xlab, ylab, main	plotting parameters
width, height, units, res	parameters passed to png if the plot is saved
...	other graphical parameters passed to spectrogram

Details

Each pitch tracker is controlled by its own list of settings, as follows:

`pitchDom` **(lowest dominant frequency band)**
- `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`

`pitchAutocor` **(autocorrelation)**
- `autocorThres` voicing threshold (unitless, ~0 to 1)
- `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
- `autocorUpsample` upsamples acf to this resolution (Hz) to improve accuracy in high frequencies
- `autocorBestPeak` amplitude of the lowest best candidate relative to the absolute max of the acf

`pitchCep` **(cepstrum)**
- `cepThres` voicing threshold (unitless, ~0 to 1)
- `cepZp` zero-padding of the spectrum used for cepstral pitch detection (final length of spectrum after zero-padding in points, e.g. 2 ^ 13)

`pitchSpec` **(ratio of harmonics - BaNa algorithm)**
- `specThres` voicing threshold (unitless, ~0 to 1)
- `specPeak`, `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

- `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`

pitchHps (harmonic product spectrum)

- `hpsNum` the number of times to downsample the spectrum
- `hpsThres` voicing threshold (unitless, ~0 to 1)
- `hpsNorm` the amount of inflation of hps pitch certainty (0 = none)
- `hpsPenalty` the amount of penalizing hps candidates in low frequencies (0 = none)

Each of these lists also accepts graphical parameters that affect how pitch candidates are plotted, eg `pitchDom = list(domThres = .5, col = 'yellow')`. Other arguments that are lists of subroutine-specific settings include:

`harmonicHeight` **(finding how high harmonics reach in the spectrum)**

- `harmThres` minimum height of spectral peak, dB
- `harmPerSel` the number of harmonics per sliding selection
- `harmTol` maximum tolerated deviation of peak frequency from multiples of f0, proportion of f0

Value

Returns a list with `$detailed` frame-by-frame descriptives and a `$summary` with one row per file, as determined by `summaryFun` (e.g., mean / median / SD of each acoustic variable across all STFT frames). Output measures include:

duration total duration, s

duration_noSilence duration from the beginning of the first non-silent STFT frame to the end of the last non-silent STFT frame, s (NB: depends strongly on `windowLength` and `silence` settings)

time time of the middle of each frame (ms)

amEnvFreq,amEnvDep frequency (Hz) and depth (0 to 1) of amplitude modulation estimated from a smoothed amplitude envelope

amMsFreq,amMsPurity frequency and purity of amplitude modulation estimated via `modulationSpectrum`

ampl root mean square of amplitude per frame, calculated as sqrt(mean(frame ^ 2))

ampl_noSilence same as `ampl`, but ignoring silent frames

CPP Cepstral Peak Prominence, dB (a measure of pitch quality, the ratio of the highest peak in the cepstrum to the regression line drawn through it)

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

entropySh Normalized Shannon entropy of the spectrum of the current frame: 0 = pure tone, 1 = white noise

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

flux feature-based flux, the rate of change in acoustic features such as pitch, HNR, etc. (0 = none, 1 = max); "epoch" is an audio segment between two peaks of flux that exceed a threshold of `flux = list(thres = ...)` (listed in output$detailed only)

fmFreq frequency of frequency modulation (FM) such as vibrato or jitter, Hz

fmDep depth of FM, semitones

fmPurity purity or dominance of the main FM frequency (fmFreq), 0 to 1

harmEnergy 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)

harmHeight how high harmonics reach in the spectrum, based on the best guess at pitch (or the manually provided pitch values)

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`

novelty spectral novelty - a measure of how variable the spectrum is on a particular time scale, as estimated by `ssm`

peakFreq the frequency with maximum spectral power (Hz)

pitch post-processed pitch contour based on all F0 estimates

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

roughness the amount of amplitude modulation, see modulationSpectrum

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

specSlope the slope of linear regression fit to the spectrum below cutFreq (dB/kHz)

subDep estimated depth of subharmonics per frame: 0 = none, 1 = as strong as f0. NB: this depends critically on accurate pitch tracking

subRatio the ratio of f0 to subharmonics frequency with strength subDep: 2 = period doubling, 3 = f0 / 3, etc.

voiced is the current STFT frame voiced? TRUE / FALSE

See Also

`pitch_app` `getLoudness` `segment` `getRMS`

Examples

```
sound = soundgen(sylLen = 300, pitch = c(500, 400, 600),
  noise = list(time = c(0, 300), value = c(-40, 0)),
  temperature = 0.001,
  addSilence = 50)  # NB: always have some silence before and after!!!
# playme(sound, 16000)
a = analyze(sound, samplingRate = 16000, plot = TRUE)
str(a$detailed)  # frame-by-frame
a$summary        # summary per sound
```

```
## Not run:
# For maximum processing speed (just basic spectral descriptives):
a = analyze(sound, samplingRate = 16000,
  plot = FALSE,         # no plotting
  pitchMethods = NULL,  # no pitch tracking
  loudness = NULL,      # no loudness analysis
  novelty = NULL,       # no novelty analysis
  roughness = NULL,     # no roughness analysis
  nFormants = 0         # no formant analysis
)

# Take a sound hard to analyze b/c of subharmonics and jitter
sound2 = soundgen(sylLen = 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 = 10, jitterDep = 0.5,
  temperature = 0.001, samplingRate = 44100, pitchSamplingRate = 44100)
# playme(sound2, 44100)
a2 = analyze(sound2, samplingRate = 44100, priorSD = 24,
             plot = TRUE, ylim = c(0, 5))

# Compare the available pitch trackers
analyze(sound2, 44100,
  pitchMethods = c('dom', 'autocor', 'spec', 'cep', 'hps', 'zc'),
  # don't use priors to see weird pitch candidates better
  priorMean = NA, priorAdapt = FALSE,
  plot = TRUE, yScale = 'bark')

# Fancy plotting options:
a = analyze(sound2, samplingRate = 44100, plot = TRUE,
  xlab = 'Time, ms', colorTheme = 'seewave',
  contrast = .5, ylim = c(0, 4), main = 'My plot',
  pitchMethods = c('dom', 'autocor', 'spec', 'hps', 'cep'),
  priorMean = NA,  # no prior info at all
  pitchDom = list(col = 'red', domThres = .25),
  pitchPlot = list(col = 'black', pch = 9, lty = 3, lwd = 3),
  extraContour = list(x = 'peakFreq', type = 'b', pch = 4, col = 'brown'),
  osc = 'dB', heights = c(2, 1))

# Analyze an entire folder in one go, saving spectrograms with pitch contours
# plus an html file for easy access
s2 = analyze('~/Downloads/temp',
  savePlots = '',  # save in the same folder as audio
  showLegend = TRUE, yScale = 'bark',
  width = 20, height = 12,
  units = 'cm', res = 300, ylim = c(0, 5),
  cores = 4)  # use multiple cores to speed up processing
s2$summary[, 1:5]

# Different options for summarizing the output
a = analyze(sound2, 44100,
            summaryFun = c('mean', 'range'))
a$summary  # one row per sound
```

```
# ...with custom summaryFun, eg time of peak relative to duration (0 to 1)
timePeak = function(x) which.max(x) / length(x)  # without omitting NAs
timeTrough = function(x) which.min(x) / length(x)
a = analyze(sound2, samplingRate = 16000,
            summaryFun = c('mean', 'timePeak', 'timeTrough'))
colnames(a$summary)

# Analyze a selection rather than the whole sound
a = analyze(sound, samplingRate = 16000, from = .1, to = .3, plot = TRUE)

# Use only a range of frequencies when calculating spectral descriptives
# (ignore everything below 100 Hz and above 8000 Hz as irrelevant noise)
a = analyze(sound, samplingRate = 16000, cutFreq = c(100, 8000))

## Amplitude and loudness: analyze() should give the same results as
# dedicated functions getRMS() / getLoudness()
# Create 1 kHz tone
samplingRate = 16000; dur_ms = 50
sound3 = sin(2*pi*1000/samplingRate*(1:(dur_ms/1000*samplingRate)))
a1 = analyze(sound3, samplingRate = samplingRate, scale = 1,
             windowLength = 25, overlap = 50,
             loudness = list(SPL_measured = 40),
             pitchMethods = NULL, plot = FALSE)
a1$detailed$loudness  # loudness per STFT frame (1 sone by definition)
getLoudness(sound3, samplingRate = samplingRate, windowLength = 25,
            overlap = 50, SPL_measured = 40, scale = 1)$loudness
a1$detailed$ampl  # RMS amplitude per STFT frame
getRMS(sound3, samplingRate = samplingRate, windowLength = 25,
       overlap = 50, scale = 1)$detailed
# or even simply: sqrt(mean(sound3 ^ 2))

# The same sound as above, but with half the amplitude
a_half = analyze(sound3 / 2, samplingRate = samplingRate, scale = 1,
                 windowLength = 25, overlap = 50,
                 loudness = list(SPL_measured = 40),
                 pitchMethods = NULL, plot = FALSE)
a1$detailed$ampl / a_half$detailed$ampl  # rms amplitude halved
a1$detailed$loudness/ a_half$detailed$loudness
# loudness is not a linear function of amplitude

# Analyzing ultrasounds (slow but possible, just adjust pitchCeiling)
s = soundgen(sylLen = 100, addSilence = 10,
  pitch = c(25000, 35000, 30000),
  formants = NA, rolloff = -12, rolloffKHz = 0,
  pitchSamplingRate = 350000, samplingRate = 350000, windowLength = 5,
  pitchCeiling = 45000, invalidArgAction = 'ignore',
  plot = TRUE)
# s is a bat-like ultrasound inaudible to humans
a = analyze(
  s, 350000, plot = TRUE,
  pitchFloor = 10000, pitchCeiling = 90000, priorMean = NA,
  pitchMethods = c('autocor', 'spec'),
  # probably shouldn't use pitchMethod = "dom" b/c of likely low-freq noise
```

```
  windowLength = 5, step = 2.5,
  shortestSyl = 10, shortestPause = 10,  # again, very short sounds
  interpol = list(win = 10),  # again, very short sounds
  smooth = 0.1,  # might need less smoothing if very rapid f0 changes
  nFormants = 0, loudness = NULL, roughness = NULL, novelty = NULL)
# NB: ignore formants and loudness estimates for such non-human sounds

# download 260 sounds from Anikin & Persson (2017)
# http://cogsci.se/publications/anikin-persson_2017_nonlinguistic-vocs/260sounds_wav.zip
# unzip them into a folder, say '~/Downloads/temp'
myfolder = '~/Downloads/temp'  # 260 .wav files live here
s = analyze(myfolder)  # ~ 10-20 minutes!
# s = write.csv(s, paste0(myfolder, '/temp.csv'))  # save a backup

# Check accuracy: import manually verified pitch values (our "key")
# pitchManual   # "ground truth" of mean pitch per sound
# pitchContour  # "ground truth" of complete pitch contours per sound
files_manual = paste0(names(pitchManual), '.wav')
idx = match(s$file, files_manual)  # in case the order is wrong
s$key = pitchManual[idx]

# Compare manually verified mean pitch with the output of analyze:
cor(s$key, s$summary$pitch_median, use = 'pairwise.complete.obs')
plot(s$key, s$summary$pitch_median, log = 'xy')
abline(a=0, b=1, col='red')

# Re-running analyze with manually corrected contours gives correct
pitch-related descriptives like amplVoiced and harmonics (NB: you get it "for
free" when running pitch_app)
s1 = analyze(myfolder, pitchManual = pitchContour)
plot(s$summary$harmonics_median, s1$summary$harmonics_median)
abline(a=0, b=1, col='red')

## End(Not run)
```

`annotation_app` *Annotation app*

Description

Starts a shiny app for annotating audio. This is a simplified and faster version of `formant_app` intended only for making annotations.

Usage

```
annotation_app()
```

Value

Every time a new annotation is added, the app creates a backup csv file and creates or updates a global object called "my_annot", which contains all the annotations. When the app is terminated, it also returns the results as a dataframe.

Examples

```
## Not run:
ann = annotation_app()  # runs in default browser such as Firefox or Chrome

# To change system default browser, run something like:
options('browser' = '/usr/bin/firefox')  # path to the executable on Linux

## End(Not run)
```

---

audSpectrogram *Auditory spectrogram*

Description

Produces an auditory spectrogram by convolving the sound with a bank of bandpass filters. The main difference from STFT is that we don't window the signal and de facto get variable temporal resolution in different frequency channels, as with a wavelet transform. The key settings are `filterType`, `nFilters`, and `yScale`, which determine the type, number, and spacing of the filters, respectively. Gammatone filters were designed as a simple approximation of human perception - see `gammatone` and Slaney 1993 "An Efficient Implementation of the Patterson–Holdsworth Auditory Filter Bank". Butterworth or Chebyshev filters are not meant to model perception, but can be useful for quickly plotting a sound.

Usage

```
audSpectrogram(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  step = 1,
  dynamicRange = 80,
  filterType = c("butterworth", "chebyshev", "gammatone")[1],
  nFilters = 128,
  nFilters_oct = NULL,
  filterOrder = if (filterType == "gammatone") 4 else 3,
  bandwidth = NULL,
  bandwidthMult = 1,
  minFreq = 20,
  maxFreq = samplingRate/2,
```

```
  minBandwidth = 10,
 output = c("audSpec", "audSpec_processed", "filterbank", "filterbank_env", "roughness"),
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = NULL,
  plotFilters = FALSE,
  osc = c("none", "linear", "dB")[2],
  heights = c(3, 1),
  ylim = NULL,
  yScale = c("bark", "mel", "ERB", "log")[1],
  contrast = 0.2,
  brightness = 0,
  maxPoints = c(1e+05, 5e+05),
  padWithSilence = TRUE,
  colorTheme = c("bw", "seewave", "heat.colors", "...")[1],
  col = NULL,
  extraContour = NULL,
  xlab = NULL,
  ylab = NULL,
  xaxp = NULL,
  mar = c(5.1, 4.1, 4.1, 2),
  main = NULL,
  grid = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
step	step, ms (determines time resolution of the plot, but not of the returned envelopes per channel). step = NULL means no downsampling at all (ncol of output = length of input audio)
dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
filterType	"butterworth" = Butterworth filter butter, "chebyshev" = Chebyshev filter butter, "gammatone" = gammatone

nFilters	the number of filters between minFreq and maxFreq (determines frequency resolution, while ySc ale determines the location of center frequencies)
nFilters_oct	an alternative way to specify frequency resolution: the number of filters per octave
filterOrder	filter order (defaults to 4 for gammatones, 3 otherwise)
bandwidth	filter bandwidth, octaves. If NULL, defaults to ERB bandwidths as in gammatone
bandwidthMult	a scaling factor for all bandwidths (1 = no effect)
minFreq, maxFreq	the range of frequencies to analyze. If the spectrogram looks empty, try increasing minFreq - the lowest filters are prone to returning very large values
minBandwidth	minimum filter bandwidth, Hz (otherwise filters may become too narrow when nFilters is high; has no effect if filterType = 'gammatone')
output	a list of measures to return. Defaults to everything, but this takes a lot of RAM, so shorten to what's needed if analyzing many files at once
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
plotFilters	if TRUE, plots the filters as central frequencies ± bandwidth/2
osc	"none" = no oscillogram; "linear" = on the original scale; "dB" = in decibels
heights	a vector of length two specifying the relative height of the spectrogram and the oscillogram (including time axes labels)
ylim	frequency range to plot, kHz (defaults to 0 to Nyquist frequency). NB: still in kHz, even if ySc ale = bark, mel, or ERB
ySc ale	determines the location of center frequencies of the filters
contrast	a number, recommended range -1 to +1. The spectrogram is raised to the power of exp(3 * contrast). Contrast >0 increases sharpness, <0 decreases sharpness
brightness	how much to "lighten" the image (>0 = lighter, <0 = darker)
maxPoints	the maximum number of "pixels" in the oscillogram (if any) and spectrogram; good for quickly plotting long audio files; defaults to c(1e5, 5e5); does not affect reassigned spectrograms
padWithSilence	if TRUE, pads the sound with just enough silence to resolve the edges properly (only the original region is plotted, so the apparent duration doesn't change)
colorTheme	black and white ('bw'), as in seewave package ('seewave'), matlab-type palette ('matlab'), or any palette from palette such as 'heat.colors', 'cm.colors', etc
col	actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)
extraContour	a vector of arbitrary length scaled in Hz (regardless of ySc ale!) that will be plotted over the spectrogram (eg pitch contour); can also be a list with extra graphical parameters such as lwd, col, etc. (see examples)

xlab, ylab, main, mar, xaxp
: graphical parameters for plotting

grid
: if numeric, adds n = grid dotted lines per kHz

width, height, units, res
: graphical parameters for saving plots passed to png

...
: other graphical parameters

Value

Returns a list for each analyzed file, including:

audSpec auditory spectrogram with frequencies in rows and time in columns

audSpec_processed same but rescaled for plotting

filterbank raw output of the filters

roughness roughness per channel (as many as nFilters)

Examples

```
# synthesize a sound with gradually increasing hissing noise
sound = soundgen(sylLen = 200, temperature = 0.001,
  noise = list(time = c(0, 350), value = c(-40, 0)),
  formantsNoise = list(f1 = list(freq = 5000, width = 10000)),
  addSilence = 25)
# playme(sound, samplingRate = 16000)

# auditory spectrogram
as = audSpectrogram(sound, samplingRate = 16000, nFilters = 48)
dim(as$audSpec)

# compare to FFT-based spectrogram with similar time and frequency resolution
fs = spectrogram(sound, samplingRate = 16000, yScale = 'bark',
                 windowLength = 5, step = 1)
dim(fs)

## Not run:
# add bells and whistles
audSpectrogram(sound, samplingRate = 16000,
  filterType = 'butterworth',
  nFilters = 128,
  yScale = 'ERB',
  bandwidth = 1/6,
  dynamicRange = 150,
  osc = 'dB',  # plot oscillogram in dB
  heights = c(2, 1),  # spectro/osc height ratio
  contrast = .4,  # increase contrast
  brightness = -.2,  # reduce brightness
  # colorTheme = 'heat.colors',  # pick color theme...
  col = hcl.colors(100, palette = 'Plasma'),  # ...or specify the colors
  cex.lab = .75, cex.axis = .75,  # text size and other base graphics pars
  grid = 5,  # to customize, add manually with graphics::grid()
  ylim = c(0.05, 8),  # always in kHz
```

```
  main = 'My auditory spectrogram' # title
  # + axis labels, etc
)

# NB: frequency resolution is controlled by both nFilters and bandwidth
audSpectrogram(sound, 16000, nFilters = 15, bandwidth = 1/2)
audSpectrogram(sound, 16000, nFilters = 15, bandwidth = 1/10)
audSpectrogram(sound, 16000, nFilters = 100, bandwidth = 1/2)
audSpectrogram(sound, 16000, nFilters = 100, bandwidth = 1/10)
audSpectrogram(sound, 16000, nFilters_oct = 5, bandwidth = 1/10)

# remove the oscillogram
audSpectrogram(sound, samplingRate = 16000, osc = 'none')

# save auditory spectrograms of all audio files in a folder
audSpectrogram('~/Downloads/temp',
  savePlots = '~/Downloads/temp/audSpec', cores = 4)

## End(Not run)
```

---

bandpass *Bandpass/stop filters*

Description

Filtering in the frequency domain with FFT-iFFT: low-pass, high-pass, bandpass, and bandstop filters. Similar to `ffilter`, but here we use FFT instead of STFT - that is, the entire sound is processed at once. This works best for relatively short sounds (seconds), but gives us maximum precision (e.g., for precise notch filtering) and doesn't affect the attack and decay. NAs are accepted and can be interpolated or preserved in the output. Because we don't do STFT, arbitrarily short vectors are also fine as input - for example, we can apply a low-pass filter prior to decimation when changing the sampling rate without aliasing. Note that, unlike `pitchSmoothPraat`, `bandpass` by default applies an abrupt cutoff instead of a smooth gaussian filter, but this behavior can be adjusted with the `bw` argument.

Usage

```
bandpass(
  x,
  samplingRate = NULL,
  lwr = NULL,
  upr = NULL,
  action = c("pass", "stop")[1],
  dB = Inf,
  bw = 0,
  na.rm = TRUE,
  from = NULL,
  to = NULL,
```

```
  normalize = FALSE,
  reportEvery = NULL,
  cores = 1,
  saveAudio = NULL,
  plot = FALSE,
  savePlots = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
lwr, upr	cutoff frequencies, Hz. Specifying just lwr gives a high-pass filter, just upr low-pass filter with action = 'pass' (or vice versa with action = 'stop'). Specifying both lwr and upr a bandpass/bandstop filter, depending on 'action'
action	"pass" = preserve the selected frequency range (bandpass), "stop" = remove the selected frequency range (bandstop)
dB	a positive number giving the strength of effect in dB (defaults to Inf - complete removal of selected frequencies)
bw	bandwidth of the filter cutoffs, Hz. Defaults to 0 (abrupt, step function), a positive number corresponds to the standard deviation of a Gaussian curve, and two numbers set different bandwidths for the lower and upper cutoff points
na.rm	if TRUE, NAs are interpolated, otherwise they are preserved in the output
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
normalize	if TRUE, resets the output to the original scale (otherwise filtering often reduces the amplitude)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
saveAudio	full path to the folder in which to save the processed audio
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters passed to plot() as well as to meanspec

Details

Algorithm: fill in NAs with constant interpolation at the edges and linear interpolation in the middle; perform FFT; set the frequency ranges to be filtered out to 0; perform inverse FFT; set to the original scale; put the NAs back in.

Examples

```
# Filter white noise
s1 = fade(c(rnorm(2000, 0, 1)), samplingRate = 16000)

# low-pass
bandpass(s1, 16000, upr = 2000, plot = TRUE)

# high-pass by 40 dB
bandpass(s1, 16000, lwr = 2000, dB = 40, plot = TRUE, wl = 1024)
# wl is passed to seewave::meanspec for plotting

# bandstop
bandpass(s1, 16000, lwr = 1000, upr = 1800, action = 'stop', plot = TRUE)

# bandpass
s2 = bandpass(s1, 16000, lwr = 2000, upr = 2100, plot = TRUE)
# playme(rep(s2, 5))
# spectrogram(s2, 16000)

# low-pass and interpolate a short vector with some NAs
x = rnorm(150, 10) + 3 * sin((1:50) / 5)
x[sample(seq_along(x), 50)] = NA
plot(x, type = 'l')
x_bandp = bandpass(x, samplingRate = 100, upr = 10)
points(x_bandp, type = 'l', col = 'blue')

## Not run:
# add 20 dB with a Gaussian-shaped filter instead of step function
s3 = bandpass(s1, 16000, lwr = 1700, upr = 2100, bw = 200,
  dB = 20, plot = TRUE)
spectrogram(s3, 16000)
s4 = bandpass(s1, 16000, lwr = 2000, upr = 4300, bw = c(100, 500),
  dB = 60, action = 'stop', plot = TRUE)
spectrogram(s4, 16000)

# precise notch filtering is possible, even in low frequencies
whiteNoise = runif(16000, -1, 1)
s3 = bandpass(whiteNoise, 16000, lwr = 30, upr = 40, normalize = TRUE,
              plot = TRUE, xlim = c(0, 500))
playme(rep(s3, 5))
spectrogram(s3, 16000, windowLength = 150, yScale = 'log')

# compare the same with STFT
s4 = seewave::ffilter(whiteNoise, f = 16000, from = 30, to = 40)
spectrogram(s4, 16000, windowLength = 150, yScale = 'log')
# (note: works better as wl approaches length(s4))
```

```
# high-pass all audio files in a folder
bandpass('~/Downloads/temp', saveAudio = '~/Downloads/temp/hp2000/',
         lwr = 2000, savePlots = '~/Downloads/temp/hp2000/')

## 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

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

Arguments

nSyl	the number of syllables to generate
sylLen	average duration of each syllable, ms
pauseLen	average duration of pauses between syllables, ms
pitch	fundamental frequency, Hz - a vector or data.frame(time = ..., value = ...)
samplingRate	sampling frequency, Hz
fadeOut	if TRUE, a linear fade-out is applied to the entire syllable
play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play

Value

Returns a non-normalized waveform centered at zero.

See Also

soundgen generateNoise fart

Examples

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

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

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

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

---

compareSounds *Compare two sounds*

Description

Computes similarity between two sounds based on comparing their spectrogram-like representations. If the input is audio, two methods of producing spectrograms are available: `specType = 'linear'` calls powspec for an power spectrogram with frequencies in Hz, and `specType = 'mel'` calls melfcc for an auditory spectrogram with frequencies in Mel. For more customized options, just produce your spectrograms or feature matrices (time in column, features like pitch, peak frequency etc in rows) with your favorite function before calling `compareSounds` because it also accepts matrices as input. To be directly comparable, the two matrices are made into matrices of the same size. In case of differences in sampling rates, only frequencies below the lower Nyquist frequency or below `maxFreq` are kept. In case of differences in duration, the shorter sound is padded with 0 (silence) or NA, as controlled by arguments `padWith`, `padDir`. Then the matrices are compared using methods like cross-correlation or Dynamic Time Warp.

Usage

```
compareSounds(
  x,
  y,
  samplingRate = NULL,
  windowLength = 40,
  overlap = 50,
```

```
  step = NULL,
  dynamicRange = 80,
  method = c("cor", "cosine", "diff", "dtw"),
  specType = c("linear", "mel")[2],
  specPars = list(),
  dtwPars = list(),
  padWith = NA,
  padDir = c("central", "left", "right")[1],
  maxFreq = NULL
)
```

Arguments

x, y	either two matrices (spectrograms or feature matrices) or two sounds to be compared (numeric vectors, Wave objects, or paths to wav/mp3 files)
samplingRate	if one or both inputs are numeric vectors, specify sampling rate, Hz. A vector of length 2 means the two inputs have different sampling rates, in which case spectrograms are compared only up to the lower Nyquist frequency
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
overlap	overlap between successive FFT frames, %
step	you can override `overlap` by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
dynamicRange	parts of the spectra quieter than `-dynamicRange` dB are not compared
method	method of comparing mel-transformed spectra of two sounds: "cor" = Pearson's correlation; "cosine" = cosine similarity; "diff" = absolute difference between each bin in the two spectrograms; "dtw" = multivariate Dynamic Time Warp with `dtw`
specType	"linear" = power spectrogram with `powspec`, "mel" = mel-frequency spectrogram with `melfcc`
specPars	a list of parameters passed to `melfcc`
dtwPars	a list of parameters passed to `dtw`
padWith	if the duration of x and y is not identical, the compared spectrograms are padded with either silence (`padWith = 0`) or with NA's (`padWith = NA`) to have the same number of columns. Padding with NA implies that only the overlapping part is of relevance, whereas padding with 0 means that the added silent part is also compared with the longer sound, usually resulting in lower similarity (see examples)
padDir	if padding, specify where to add zeros or NAs: before the sound ('left'), after the sound ('right'), or on both sides ('central')
maxFreq	parts of the spectra above `maxFreq` Hz are not compared

Value

Returns a dataframe with two columns: "method" for the method(s) used, and "sim" for the similarity between the two sounds calculated with that method. The range of similarity measures is [-1, 1] for "cor", [0, 1] for "cosine" and "diff", and (-Inf, Inf) for "dtw".

Examples

```
data(orni, peewit, package = 'seewave')
compareSounds(orni, peewit)
# spectrogram(orni); playme(orni)
# spectrogram(peewit); playme(peewit)

## Not run:
s1 = soundgen(formants = 'a', play = TRUE)
s2 = soundgen(formants = 'ae', play = TRUE)
s3 = soundgen(formants = 'eae', sylLen = 700, play = TRUE)
s4 = runif(8000, -1, 1)  # white noise
compareSounds(s1, s2, samplingRate = 16000)
compareSounds(s1, s4, samplingRate = 16000)

# the central section of s3 is more similar to s1 than is the beg/eng of s3
compareSounds(s1, s3, samplingRate = 16000, padDir = 'left')
compareSounds(s1, s3, samplingRate = 16000, padDir = 'central')

# padding with 0 penalizes differences in duration, whereas padding with NA
# is like saying we only care about the overlapping part
compareSounds(s1, s3, samplingRate = 16000, padWith = 0)
compareSounds(s1, s3, samplingRate = 16000, padWith = NA)

# comparing linear (Hz) vs mel-spectrograms produces quite different results
compareSounds(s1, s3, samplingRate = 16000, specType = 'linear')
compareSounds(s1, s3, samplingRate = 16000, specType = 'mel')

# pass additional control parameters to dtw and melfcc
compareSounds(s1, s3, samplingRate = 16000,
              specPars = list(nbands = 128),
              dtwPars = list(dist.method = "Manhattan"))

# use feature matrices instead of spectrograms (time in columns, features in rows)
a1 = t(as.matrix(analyze(s1, samplingRate = 16000)$detailed))
a1 = a1[4:nrow(a1), ]; a1[is.na(a1)] = 0
a2 = t(as.matrix(analyze(s2, samplingRate = 16000)$detailed))
a2 = a2[4:nrow(a2), ]; a2[is.na(a2)] = 0
a4 = t(as.matrix(analyze(s4, samplingRate = 16000)$detailed))
a4 = a4[4:nrow(a4), ]; a4[is.na(a4)] = 0
compareSounds(a1, a2, method = c('cosine', 'dtw'))
compareSounds(a1, a4, method = c('cosine', 'dtw'))

# a demo for comparing different similarity metrics
target = soundgen(sylLen = 500, formants = 'a',
                  pitch = data.frame(time = c(0, 0.1, 0.9, 1),
                                     value = c(100, 150, 135, 100)),
```

```
                  temperature = 0.001)
spec1 = soundgen:::getMelSpec(target, samplingRate = 16000)

parsToTry = list(
  list(formants = 'i',                                          # wrong
       pitch = data.frame(time = c(0, 1),                       # wrong
                          value = c(200, 300))),
  list(formants = 'i',                                          # wrong
       pitch = data.frame(time = c(0, 0.1, 0.9, 1),             # right
                                value = c(100, 150, 135, 100))),
  list(formants = 'a',                                          # right
       pitch = data.frame(time = c(0,1),                        # wrong
                                value = c(200, 300))),
  list(formants = 'a',
       pitch = data.frame(time = c(0, 0.1, 0.9, 1),             # right
                                value = c(100, 150, 135, 100)))  # right
)

sounds = list()
for (s in seq_along(parsToTry)) {
  sounds[[length(sounds) + 1]] =  do.call(soundgen,
    c(parsToTry[[s]], list(temperature = 0.001, sylLen = 500)))
}
lapply(sounds, playme)

method = c('cor', 'cosine', 'diff', 'dtw')
df = matrix(NA, nrow = length(parsToTry), ncol = length(method))
colnames(df) = method
df = as.data.frame(df)
for (i in 1:nrow(df)) {
  df[i, ] = compareSounds(
    x = spec1,  # faster to calculate spec1 once
    y = sounds[[i]],
    samplingRate = 16000,
    method = method
  )[, 2]
}
df$av = rowMeans(df, na.rm = TRUE)
# row 1 = wrong pitch & formants, ..., row 4 = right pitch & formants
df$formants = c('wrong', 'wrong', 'right', 'right')
df$pitch = c('wrong', 'right', 'wrong', 'right')
df

## End(Not run)
```

---

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 ampl1 ends with a zero crossing and ampl2 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.

Usage

```
crossFade(
  ampl1,
  ampl2,
  crossLenPoints = 240,
  crossLen = NULL,
  samplingRate = NULL,
  shape = c("lin", "exp", "log", "cos", "logistic", "gaussian")[1],
  steepness = 1
)
```

Arguments

ampl1, ampl2	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 (except for shapes 'lin' and 'cos'): 0 = linear, >1 = steeper than default

Value

Returns a numeric vector.

See Also

fade

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)

# concatentation 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)

# s-shaped cross-fade over 300 ms (shortens the sound by ~300 ms)
playme(crossFade(sound3, sound4, samplingRate = 16000,
                 crossLen = 300, shape = 'cos'), 16000)

## End(Not run)
```

`defaults` *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
```

Format

An object of class `list` of length 69.

defaults_analyze *Defaults and ranges for analyze()*

Description

A dataset containing defaults and ranges of key variables for analyze() and pitch_app(). Adjust as needed.

Usage

```
defaults_analyze
```

Format

A matrix with 58 rows and 4 columns:

default default value

low lowest permitted value

high highest permitted value

step increment for adjustment ...

defaults_analyze_pitchCand *Defaults for plotting with analyze()*

Description

Default plotting settings for each pitch tracker in analyze() and pitch_app(). Adjust as needed.

Usage

```
defaults_analyze_pitchCand
```

Format

A dataframe with 8 rows and 5 columns:

method pitch tracking method

col color

pch point character

lwd line width

lty line type ...

detectNLP *Detect NLP*

Description

(Experimental) A function for automatically detecting and annotating nonlinear vocal phenomena (NLP). Algorithm: analyze the audio using analyze and phasegram, then use the extracted frame-by-frame descriptives to classify each frame as having no NLP ("none"), subharmonics ("sh"), sibebands / amplitude modulation ("sb"), or deterministic chaos ("chaos"). The classification is performed by a naiveBayes algorithm adapted to autocorrelated time series and pretrained on a manually annotated corpus of vocalizations. Whenever possible, check and correct pitch tracks prior to running the algorithm. See naiveBayes for tips on using adaptive priors and "clumpering" to account for the fact that NLP typically occur in continuous segments spanning multiple frames.

Usage

```
detectNLP(
  x,
  samplingRate = NULL,
 predictors = c("d2", "subDep", "amEnvDep", "amMsPurity", "entropy", "HNR", "CPP",
    "roughness"),
  thresProb = 0.4,
  unvoicedToNone = FALSE,
  train = soundgen::detectNLP_training_nonv,
  scale = NULL,
  from = NULL,
  to = NULL,
  pitchManual = NULL,
 pars_analyze = list(windowLength = 50, roughness = list(windowLength = 15, step = 3)),
  pars_phasegram = list(nonlinStats = "d2"),
  pars_naiveBayes = list(prior = "static", wlClumper = 3),
  jumpThres = 14,
  jumpWindow = 100,
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = NULL,
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  ylim = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
predictors	variables to include in NLP classification. The default is to include all 7 variables in the training corpus. NA values are fine (they do not cause the entire frame to be dropped as long as at least one variable is measured).
thresProb	minimum probability of NLP for the frame to be classified as non-"none", which is good for reducing false alarms (<1/nClasses means just go for the highest probability)
unvoicedToNone	if TRUE, frames treated as unvoiced are set to "none" (mostly makes sense with manual pitch tracking)
train	training corpus, namely the result of running naiveBayes_train on audio with known NLP episodes. Currently implemented: soundgen::detectNLP_training_nonv = manually annotated human nonverbal vocalizations, soundgen::detectNLP_training_synth = synthetic, soundgen()-generated sounds with various NLP. To train your own, run detectNLP on a collection of recordings, provide ground truth classification of NLP per frame (normally this would be converted from NLP annotations), and run naiveBayes_train.
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
pitchManual	manually corrected pitch contour. For a single sound, provide a numeric vector of any length. For multiple sounds, provide a dataframe with columns "file" and "pitch" (or path to a csv file) as returned by pitch_app, ideally with the same windowLength and step as in current call to analyze. A named list with pitch vectors per file is also OK (eg as returned by pitch_app)
pars_analyze	arguments passed to analyze. NB: drop everything unnecessary to speed up the process, e.g. nFormants = 0, loudness = NULL, etc. If you have manual pitch contours, pass them as pitchManual = ... Make sure the "silence" threshold is appropriate, and ideally normalize the audio (silent frames are automatically assigned to "none")
pars_phasegram	arguments passed to phasegram. NB: only d2 and nPeaks are used for NLP detection because they proved effective in the training corpus; other nonlinear statistics are not calculated to save time.
pars_naiveBayes	arguments passed to naiveBayes. It is strongly recommended to use some clumpering, with wlClumper given as frames (multiple by step to get the corresponding minumum duration of an NLP segment in ms), and/or dynamic priors.
jumpThres	frames in which pitch changes by jumpThres octaves/s more than in the surrounding frames are classified as containing "pitch jumps". Note that this is the rate of frequency change PER SECOND, not from one frame to the next
jumpWindow	the window for calculating the median pitch slope around the analyzed frame, ms

reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	if TRUE, produces a spectrogram with annotated NLP regimes
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
main, xlab, ylab, ...	graphical parameters passed to spectrogram
ylim	frequency range to plot, kHz (defaults to 0 to Nyquist frequency). NB: still in kHz, even if yScale = bark, mel, or ERB
width, height, units, res	parameters passed to png if the plot is saved

Value

Returns a dataframe with frame-by-frame descriptives, posterior probabilities of each NLP type per frame, and the tentative classification (the NLP type with the highest posterior probability, possibly corrected by clumpering). The time step is equal to the larger of the steps passed to analyze() and phasegram().

Returns a list of datasets, one per input file, with acoustic descriptives per frame (returned by analyze and phasegram), probabilities of each NLP type per frame, and the putative classification of NLP per frame.

Examples

```
## Not run:
target = soundgen(sylLen = 2000, addSilence = 0, temperature = 1e-2,
  pitch = c(380, 550, 500, 220), subDep = c(0, 0, 40, 0, 0, 0, 0, 0),
  amDep = c(0, 0, 0, 0, 80, 0, 0, 0), amFreq = 80,
  noise = c(-10, rep(-40, 5)),
  jitterDep = c(0, 0, 0, 0, 0, 3),
  plot = TRUE, play = TRUE)

# classifier trained on manually annotated recordings of human nonverbal
# vocalizations
nlp = detectNLP(target, 16000,
  predictors = c('subDep', 'amEnvDep', 'amMsPurity', 'HNR', 'CPP'),
  plot = TRUE, ylim = c(0, 4))

# classifier trained on synthetic, soundgen()-generated sounds
nlp = detectNLP(target, 16000,
  train = soundgen::detectNLP_training_synth,
  predictors = c('subDep', 'amEnvDep', 'amMsPurity', 'HNR', 'CPP'),
  plot = TRUE, ylim = c(0, 4))
head(nlp[, c('time', 'pr')])
table(nlp$pr)
plot(nlp$amEnvDep, type = 'l')
plot(nlp$subDep, type = 'l')
plot(nlp$entropy, type = 'l')
```

```
plot(nlp$none, type = 'l')
points(nlp$sb, type = 'l', col = 'blue')
points(nlp$sh, type = 'l', col = 'green')
points(nlp$chaos, type = 'l', col = 'red')

# detection of pitch jumps
s1 = soundgen(sylLen = 1200, temperature = .001, pitch = list(
  time = c(0, 350, 351, 890, 891, 1200),
  value = c(140, 230, 460, 330, 220, 200)))
playme(s1, 16000)
nlp1 = detectNLP(s1, 16000, plot = TRUE, ylim = c(0, 3),
  predictors = c('subDep', 'amEnvDep', 'amMsPurity', 'HNR', 'CPP'),
  train = soundgen::detectNLP_training_synth)

# process all files in a folder
nlp = detectNLP('/home/allgoodguys/Downloads/temp260/',
  pitchManual = soundgen::pitchContour, cores = 4, plot = TRUE,
  savePlots = '', ylim = c(0, 3))

## End(Not run)
```

`detectNLP_training_nonv`

Nonlinear phenomena: Naive Bayes classifier trained on human nonverbal vocalizations

---

Description

The results of running `naiveBayes_train` on acoustically analyzed 969 human nonverbal vocalizations (>83K frames). It is used by `detectNLP`.

Usage

```
detectNLP_training_nonv
```

Format

An object of class `list` of length 9.

`detectNLP_training_synth`

Nonlinear phenomena: Naive Bayes classifier trained on synthetic sounds

---

Description

The results of running `naiveBayes_train` on 5000 synthetic sounds with or without NLP created with soundgen(). It is used by `detectNLP`.

Usage

```
detectNLP_training_synth
```

Format

An object of class `list` of length 9.

`ERBToHz` *Convert Hz to ERB rate*

Description

Converts from Hz to the number of Equivalent Rectangular Bandwidths (ERBs) below input frequency. See https://www2.ling.su.se/staff/hartmut/bark.htm and https://en.wikipedia.org/wiki/Equivalent_rectangular_bandw

Usage

```
ERBToHz(e, method = c("linear", "quadratic")[1])
```

Arguments

`e`	vector or matrix of frequencies in ERB rate
`method`	approximation to use

See Also

`HzToERB` `HzToSemitones` `HzToNotes`

Examples

```
freqs_Hz = c(-20, 20, 100, 440, 1000, 20000, NA)
e_lin = HzToERB(freqs_Hz, 'linear')
ERBToHz(e_lin, 'linear')

e_quad = HzToERB(freqs_Hz, 'quadratic')
ERBToHz(e_quad, 'quadratic')
```

`estimateVTL` *Estimate vocal tract length*

Description

Estimates the length of vocal tract based on formant frequencies. If `method = 'meanFormant'`, vocal tract length (VTL) is calculated separately for each formant, and then the resulting VTLs are averaged. The equation used is $(2*formant_number-1)*speedSound/(4*formant_frequency)$ for a closed-open tube (mouth open) and $formant_number*speedSound/(2*formant_frequency)$ for an open-open or closed-closed tube (eg closed mouth in mmm or open mouth and open glottis in whispering). If `method = 'meanDispersion'`, formant dispersion is calculated as the mean distance between formants, and then VTL is calculated as $speedofsound/2/formantdispersion$. If `method = 'regression'`, 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". For a review of these and other VTL-related summary measures of formant frequencies, refer to Pisanski et al. (2014) "Vocal indicators of body size in men and women: a meta-analysis". See also `schwa` for VTL estimation with additional information on formant frequencies.

Usage

```
estimateVTL(
  formants,
  method = c("regression", "meanDispersion", "meanFormant")[1],
  interceptZero = TRUE,
  tube = c("closed-open", "open-open")[1],
  speedSound = 35400,
  checkFormat = TRUE,
  output = c("simple", "detailed")[1],
  plot = FALSE
)
```

Arguments

`formants`	formant frequencies in any format recognized by `soundgen`: a vector of formant frequencies like `c(550,1600, 3200)`; a list with multiple values per formant like `list(f1 = c(500, 550), f2 = 1200)`); or a character string like `aaui` referring to default presets for speaker "M1" in soundgen presets
`method`	the method of estimating vocal tract length (see details)
`interceptZero`	if TRUE, forces the regression curve to pass through the origin. This reduces the influence of highly variable lower formants, but we have to commit to a particular model of the vocal tract: closed-open or open-open/closed-closed (method = "regression" only)
`tube`	the vocal tract is assumed to be a cylindrical tube that is either "closed-open" or "open-open" (same as closed-closed)
`speedSound`	speed of sound in warm air, by default 35400 cm/s. Stevens (2000) "Acoustic phonetics", p. 138

checkFormat	if FALSE, only a list of properly formatted formant frequencies is accepted
output	"simple" (default) = just the VTL; "detailed" = a list of additional stats (see Value below)
plot	if TRUE, plots the regression line whose slope gives formant dispersion (method = "regression" only). Label sizes show the influence of each formant, and the blue line corresponds to each formant being an integer multiple of F1 (as when harmonics are misidentified as formants); the second plot shows how VTL varies depending on the number of formants used

Value

If `output = 'simple'` (default), returns the estimated vocal tract length in cm. If `output = 'detailed'` and `method = 'regression'`, returns a list with extra stats used for plotting. Namely, `$regressionInfo$infl` gives the influence of each observation calculated as the absolute change in VTL with vs without the observation * 10 + 1 (the size of labels on the first plot). `$vtlPerFormant$vtl` gives the VTL as it would be estimated if only the first `nFormants` were used.

See Also

schwa

Examples

```
estimateVTL(NA)
estimateVTL(500)
estimateVTL(c(600, 1850, 2800, 3600, 5000), plot = TRUE)
estimateVTL(c(600, 1850, 2800, 3600, 5000), plot = TRUE, output = 'detailed')
estimateVTL(c(1200, 2000, 2800, 3800, 5400, 6400),
  tube = 'open-open', interceptZero = FALSE, plot = TRUE)
estimateVTL(c(1200, 2000, 2800, 3800, 5400, 6400),
  tube = 'open-open', interceptZero = TRUE, plot = TRUE)

# Multiple measurements are OK
estimateVTL(
  formants = list(f1 = c(540, 600, 550),
  f2 = 1650, f3 = c(2400, 2550)),
  plot = TRUE, output = 'detailed')
# NB: this is better than averaging formant values. Cf.:
estimateVTL(
  formants = list(f1 = mean(c(540, 600, 550)),
  f2 = 1650, f3 = mean(c(2400, 2550))),
  plot = TRUE)

# Missing values are OK
estimateVTL(c(600, 1850, 3100, NA, 5000), plot = TRUE)
estimateVTL(list(f1 = 500, f2 = c(1650, NA, 1400), f3 = 2700), plot = TRUE)

# Note that VTL estimates based on the commonly reported 'meanDispersion'
# depend only on the first and last formants
estimateVTL(c(500, 1400, 2800, 4100), method = 'meanDispersion')
estimateVTL(c(500, 1100, 2300, 4100), method = 'meanDispersion') # identical
```

```
# ...but this is not the case for 'meanFormant' and 'regression' methods
estimateVTL(c(500, 1400, 2800, 4100), method = 'meanFormant')
estimateVTL(c(500, 1100, 2300, 4100), method = 'meanFormant') # much longer

## Not run:
# Compare the results produced by the three methods
nIter = 1000
out = data.frame(meanFormant = rep(NA, nIter), meanDispersion = NA, regression = NA)
for (i in 1:nIter) {
  # generate a random formant configuration
  f = runif(1, 300, 900) + (1:6) * rnorm(6, 1000, 200)
  out$meanFormant[i]    = estimateVTL(f, method = 'meanFormant')
  out$meanDispersion[i] = estimateVTL(f, method = 'meanDispersion')
  out$regression[i]     = estimateVTL(f, method = 'regression')
}
pairs(out)
cor(out)
# 'meanDispersion' is pretty different, while 'meanFormant' and 'regression'
# give broadly comparable results

## End(Not run)
```

---

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

```
fade(
  x,
  fadeIn = 50,
  fadeOut = 50,
  fadeIn_points = NULL,
  fadeOut_points = NULL,
  samplingRate = NULL,
  scale = NULL,
  shape = c("lin", "exp", "log", "cos", "logistic", "gaussian")[1],
  steepness = 1,
  reportEvery = NULL,
  cores = 1,
  saveAudio = NULL,
  plot = FALSE,
  savePlots = NULL,
  width = 900,
  height = 500,
```

```
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
fadeIn, fadeOut	length of segments for fading in and out, ms (0 = no fade)
fadeIn_points, fadeOut_points	length of segments for fading in and out, points (if specified, override fadeIn/fadeOut)
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
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 (except for shapes 'lin' and 'cos'): 0 = linear, >1 = steeper than default
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
saveAudio	full path to the folder in which to save audio files (one per detected syllable)
plot	if TRUE, produces an oscillogram of the waveform after fading
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters

Value

Returns a numeric vector of the same length as input

See Also

crossFade

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)
s1 = fade(s, fadeIn = 40, fadeOut = 350,
          samplingRate = 16000, shape = 'cos', plot = TRUE)
```

```
# playme(s1)

# 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_points = 1000, fadeOut_points = 0, plot = TRUE)
y = fade(x, fadeIn_points = 1000, fadeOut_points = 1500,
         shape = 'exp', steepness = 1, plot = TRUE)
y = fade(x, fadeIn_points = 1500, fadeOut_points = 500,
         shape = 'log', steepness = 1, plot = TRUE)
y = fade(x, fadeIn_points = 1500, fadeOut_points = 500,
         shape = 'log', steepness = 3, plot = TRUE)
y = fade(x, fadeIn_points = 1500, fadeOut_points = 1500,
         shape = 'cos', plot = TRUE)
y = fade(x, fadeIn_points = 1500, fadeOut_points = 1500,
         shape = 'logistic', steepness = 1, plot = TRUE)
y = fade(x, fadeIn_points = 1500, fadeOut_points = 1500,
         shape = 'logistic', steepness = 3, plot = TRUE)
y = fade(x, fadeIn_points = 1500, fadeOut_points = 1500,
         shape = 'gaussian', steepness = 1.5, plot = TRUE)

## Not run:
  fade('~/Downloads/temp', fadeIn = 500, fadeOut = 500, savePlots = '')

## End(Not run)
```

`fart` *Fart*

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

```
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	rolloff of harmonics in source spectrum, dB/octave (not vectorized)
samplingRate	sampling frequency, Hz
play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
plot	if TRUE, plots the waveform

Value

Returns a normalized waveform.

See Also

soundgen generateNoise beat

Examples

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

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

## End(Not run)
```

---

filterMS *Filter modulation spectrum*

Description

Filters a modulation spectrum by removing a certain range of amplitude modulation (AM) and frequency modulation (FM) frequencies. Conditions can be specified either separately for AM and FM with amCond = ..., fmCond = ..., implying an OR combination of conditions, or jointly on AM and FM with jointCond. jointCond is more general, but using amCond/fmCond is ~100 times faster.

Usage

```
filterMS(
  ms,
  amCond = NULL,
  fmCond = NULL,
  jointCond = NULL,
  action = c("remove", "preserve")[1],
  plot = TRUE
)
```

Arguments

ms	a modulation spectrum as returned by modulationSpectrum - a matrix of real or complex values, AM in columns, FM in rows
amCond, fmCond	character strings with valid conditions on amplitude and frequency modulation (see examples)
jointCond	character string with a valid joint condition amplitude and frequency modulation
action	should the defined AM-FM region be removed ('remove') or preserved, while everything else is removed ('preserve')?
plot	if TRUE, plots the filtered modulation spectrum

Value

Returns the filtered modulation spectrum - a matrix of the original dimensions, real or complex.

Examples

```
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
                        returnComplex = TRUE)$complex
# Remove all AM over 25 Hz
ms_filt = filterMS(ms, amCond = 'abs(am) > 25')

# amCond and fmCond are OR-conditions
filterMS(ms, amCond = 'abs(am) > 15', fmCond = 'abs(fm) > 5', action = 'remove')
filterMS(ms, amCond = 'abs(am) > 15', fmCond = 'abs(fm) > 5', action = 'preserve')
filterMS(ms, amCond = 'abs(am) > 10 & abs(am) < 25', action = 'remove')

# jointCond is an AND-condition
filterMS(ms, jointCond = 'am * fm < 5', action = 'remove')
filterMS(ms, jointCond = 'am^2 + (fm*3)^2 < 200', action = 'preserve')

# So:
filterMS(ms, jointCond = 'abs(am) > 5 | abs(fm) < 5')  # slow but general
# ...is the same as:
filterMS(ms, amCond = 'abs(am) > 5', fmCond = 'abs(fm) < 5')  # fast
```

filterSoundByMS *Filter sound by modulation spectrum*

Description

Manipulates the modulation spectrum (MS) of a sound so as to remove certain frequencies of amplitude modulation (AM) and frequency modulation (FM). Algorithm: produces a modulation spectrum with `modulationSpectrum`, modifies it with `filterMS`, converts the modified MS to a spectrogram with `msToSpec`, and finally inverts the spectrogram with `invertSpectrogram`, thus producing a sound with (approximately) the desired characteristics of the MS. Note that the last step of inverting the spectrogram introduces some noise, so the resulting MS is not precisely the same as the intermediate filtered version. In practice this means that some residual energy will still be present in the filtered-out frequency range (see examples).

Usage

```
filterSoundByMS(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  logSpec = FALSE,
  windowLength = 25,
  step = NULL,
  overlap = 80,
  wn = "hamming",
  zp = 0,
  amCond = NULL,
  fmCond = NULL,
  jointCond = NULL,
  action = c("remove", "preserve")[1],
  initialPhase = c("zero", "random", "spsi")[3],
  nIter = 50,
  reportEvery = NULL,
  cores = 1,
  play = FALSE,
  saveAudio = NULL,
  plot = TRUE,
  savePlots = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
logSpec	if TRUE, the spectrogram is log-transformed prior to taking 2D FFT
windowLength, step, wn, zp	parameters for extracting a spectrogram if specType = 'STFT'. Window length and step are specified in ms (see spectrogram). If specType = 'audSpec', these settings have no effect
overlap	overlap between successive FFT frames, %
amCond, fmCond	character strings with valid conditions on amplitude and frequency modulation (see examples)
jointCond	character string with a valid joint condition amplitude and frequency modulation
action	should the defined AM-FM region be removed ('remove') or preserved, while everything else is removed ('preserve')?
initialPhase	initial phase estimate: "zero" = set all phases to zero; "random" = Gaussian noise; "spsi" (default) = single-pass spectrogram inversion (Beauregard et al., 2015)
nIter	the number of iterations of the GL algorithm (Griffin & Lim, 1984), 0 = don't run
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
play	if TRUE, plays back the reconstructed audio
saveAudio	full (!) path to folder for saving the processed audio; NULL = don't save, '' = same as input folder (NB: overwrites the originals!)
plot	if TRUE, produces a triple plot: original MS, filtered MS, and the MS of the output sound
savePlots	if a valid path is specified, a plot is saved in this folder (defaults to NA)
width, height, units, res	parameters passed to png if the plot is saved

Value

Returns the filtered audio as a numeric vector normalized to [-1, 1] with the same sampling rate as input.

See Also

invertSpectrogram filterMS

Examples

```
# Create a sound to be filtered
s = soundgen(pitch = rnorm(n = 20, mean = 200, sd = 25),
  amFreq = 25, amDep = 50, samplingRate = 16000,
  addSilence = 50, plot = TRUE, osc = TRUE)
# playme(s, 16000)

# Filter
s_filt = filterSoundByMS(s, samplingRate = 16000,
  amCond = 'abs(am) > 15', fmCond = 'abs(fm) > 5',
  nIter = 10,  # increase nIter for best results!
  action = 'remove', plot = TRUE)
# playme(s_filt, samplingRate = 16000)

## Not run:
# Process all files in a folder, save filtered audio and plots
s_filt = filterSoundByMS('~/Downloads/temp2',
  saveAudio = '~/Downloads/temp2/ms', savePlots = '',
  amCond = 'abs(am) > 15', fmCond = 'abs(fm) > 5',
  action = 'remove', nIter = 10)

# Download an example - a bit of speech (sampled at 16000 Hz)
download.file('http://cogsci.se/soundgen/audio/speechEx.wav',
              destfile = '~/Downloads/speechEx.wav')  # modify as needed
target = '~/Downloads/speechEx.wav'
samplingRate = tuneR::readWave(target)@samp.rate
playme(target)
spectrogram(target, osc = TRUE)

# Remove AM above 3 Hz from a bit of speech (remove most temporal details)
s_filt1 = filterSoundByMS(target, amCond = 'abs(am) > 3',
                          action = 'remove', nIter = 15)
playme(s_filt1, samplingRate)
spectrogram(s_filt1, samplingRate = samplingRate, osc = TRUE)

# Intelligigble when AM in 5-25 Hz is preserved:
s_filt2 = filterSoundByMS(target, amCond = 'abs(am) > 5 & abs(am) < 25',
                          action = 'preserve', nIter = 15)
playme(s_filt2, samplingRate)
spectrogram(s_filt2, samplingRate = samplingRate, osc = TRUE)

# Remove slow AM/FM (prosody) to achieve a "robotic" voice
s_filt3 = filterSoundByMS(target, jointCond = 'am^2 + (fm*3)^2 < 300',
                          nIter = 15)
playme(s_filt3, samplingRate)
spectrogram(s_filt3, samplingRate = samplingRate, osc = TRUE)


## An alternative manual workflow w/o calling filterSoundByMS()
# This way you can modify the MS directly and more flexibly
# than with the filterMS() function called by filterSoundByMS()
```

```
# (optional) Check that the target spectrogram can be successfully inverted
spec = spectrogram(s, 16000, windowLength = 50, step = NULL, overlap = 80,
  wn = 'hanning', osc = TRUE, padWithSilence = FALSE)
s_rev = invertSpectrogram(spec, samplingRate = 16000,
  windowLength = 50, overlap = 80, wn = 'hamming', play = FALSE)
# playme(s_rev, 16000)  # should be close to the original
spectrogram(s_rev, 16000, osc = TRUE)

# Get modulation spectrum starting from the sound...
ms = modulationSpectrum(s, samplingRate = 16000, windowLength = 25,
  overlap = 80, wn = 'hanning', amRes = NULL, maxDur = Inf, logSpec = FALSE,
  power = NA, returnComplex = TRUE, plot = FALSE)$complex
# ... or starting from the spectrogram:
# ms = specToMS(spec)
plotMS(abs(ms))  # this is the original MS

# Filter as needed - for ex., remove AM > 10 Hz and FM > 3 cycles/kHz
# (removes f0, preserves formants)
am = as.numeric(colnames(ms))
fm = as.numeric(rownames(ms))
idx_row = which(abs(fm) > 3)
idx_col = which(abs(am) > 10)
ms_filt = ms
ms_filt[idx_row, ] = 0
ms_filt[, idx_col] = 0
plotMS(abs(ms_filt))  # this is the filtered MS

# Convert back to a spectrogram
spec_filt = msToSpec(ms_filt)
image(t(log(abs(spec_filt))))

# Invert the spectrogram
s_filt = invertSpectrogram(abs(spec_filt), samplingRate = 16000,
  windowLength = 25, overlap = 80, wn = 'hanning')
# NB: use the same settings as in "spec = spectrogram(s, ...)" above

# Compare with the original
playme(s, 16000)
spectrogram(s, 16000, osc = TRUE)
playme(s_filt, 16000)
spectrogram(s_filt, 16000, osc = TRUE)

ms_new = modulationSpectrum(s_filt, samplingRate = 16000,
  windowLength = 25, overlap = 80, wn = 'hanning', maxDur = Inf,
  plot = TRUE, returnComplex = TRUE)$complex
image(x = as.numeric(colnames(ms_new)), y = as.numeric(rownames(ms_new)),
  z = t(log(abs(ms_new))))
plot(as.numeric(colnames(ms)), log(abs(ms[nrow(ms) / 2, ])), type = 'l')
points(as.numeric(colnames(ms_new)), log(ms_new[nrow(ms_new) / 2, ]), type = 'l',
  col = 'red', lty = 3)
# AM peaks at 25 Hz are removed, but inverting the spectrogram adds a lot of noise

## End(Not run)
```

`findInflections` *Find inflections*

Description

Finds inflections in discrete time series such as pitch contours. When there are no missing values and no thresholds, this can be accomplished with a fast one-liner like `which(diff(diff(x) > 0) != 0) + 1`. Missing values are interpolated by repeating the first and last non-missing values at the head and tail, respectively, and by linear interpolation in the middle. Setting a threshold means that small "wiggling" no longer counts. To use an analogy with ocean waves, smoothing (low-pass filtering) removes the ripples and only leaves the slow roll, while thresholding preserves only waves that are sufficiently high, whatever their period.

Usage

```
findInflections(x, thres = NULL, step = NULL, plot = FALSE, main = "")
```

Arguments

`x`	numeric vector with or without NAs
`thres`	minimum vertical distance between two extrema for them to count as two independent inflections
`step`	distance between values in s (only needed for plotting)
`plot`	if TRUE, produces a simple plot
`main`	plot title

Value

Returns a vector of indices giving the location of inflections.

See Also

`findPeaks`

Examples

```
x = sin(2 * pi * (1:100) / 15) * seq(1, 5, length.out = 100)
idx_na = c(1:4, 6, 7, 14, 25, 30:36, 39, 40, 42, 45:50,
           57, 59, 62, 66, 71:79, 98)
x[idx_na] = NA
soundgen:::findInflections(x, plot = TRUE)
soundgen:::findInflections(x, thres = 5, plot = TRUE)

for (i in 1:10) {
  temp = soundgen:::getRandomWalk(len = runif(1, 10, 100), rw_range = 10,
                                  rw_smoothing = runif(1, 0, 1))
  soundgen:::findInflections(temp, thres = 1, plot = TRUE)
```

```
  invisible(readline(prompt="Press [enter] to continue"))
}
```

---

findJumps *Find frequency jumps*

Description

This function flags frames with apparent pith jumps (frequency jumps, voice breaks), defined as relatively large and sudden changes in voice pitch or some other frequency measure (peak frequency, a formant frequency, etc). It is called by detectNLP. Algorithm: a frame is considered to contain a frequency jump if the absolute slope at this frame exceeds the average slope over ±jumpWindow around it by more than jumpThres. Note that the slope is considered per second rather than per time step - that is, taking into account the sampling rate of the frequency track. Thus, it's not just the change from frame to frame that defines what is considered a jump, but a change that differs from the trend in the surrounding frames (see examples). If several consecutive frames contain apparent jumps, only the greatest of them is preserved.

Usage

```
findJumps(
  pitch,
  step,
  jumpThres = 8,
  jumpWindow = 80,
  plot = FALSE,
  xlab = "Time, ms",
  ylab = "f0, Hz",
  ...
)
```

Arguments

pitch	vector of frequencies per frame, Hz
step	time step between frames, ms
jumpThres	frames in which pitch changes by jumpThres octaves/s more than in the surrounding frames are classified as containing "pitch jumps". Note that this is the rate of frequency change PER SECOND, not from one frame to the next
jumpWindow	the window for calculating the median pitch slope around the analyzed frame, ms
plot	if TRUE, plots the pitch contour with putative frequency jumps marked by arrows
xlab, ylab, ...	graphical parameters passed to plot

Value

Returns a boolean vector of the same length as `pitch`, where TRUE values correspond to frames with detected pitch jumps.

Examples

```
pitch = getSmoothContour(anchors = list(
  time = c(0, 350, 351, 890, 891, 1200),
  value = c(140, 230, 460, 330, 220, 200)), len = 40)
step = 25
pj = findJumps(pitch, step, plot = TRUE)

# convert frame indices to time in ms
step = 25
which(pj) * step
# or consider pj's to occur midway between the two frames
which(pj) * step - step / 2

# even very rapid changes are not considered jumps if they match
# the surrounding trend
pitch = getSmoothContour(anchors = list(
  time = c(0, 350, 351, 700),
  value = c(340, 710, 850, 1200)), len = 20)
findJumps(pitch, step, plot = TRUE)
diff(HzToSemitones(pitch)) * (1000 / step) / 12
# the slope at frame 10 (10.4 oct/s) exceeds the jumpThres (8 oct/s), but not
# 10.4 minus the average slope around frame 10 (~3 oct/s, so 10 - 3 < 8)
```

`findPeaks` *Find peaks*

Description

A bare-bones, very fast function to find local maxima (peaks) in a numeric vector.

Usage

```
findPeaks(x, wl = 3, thres = NULL)
```

Arguments

`x`	numeric vector
`wl`	rolling window over which we look for maxima: central value ± floor(wl/2), eg ±1 if wl=3
`thres`	required absolute value of each peak

Value

Returns a vector with indices of local maxima

See Also

findInflections

Examples

```
x = rnorm(100)
findPeaks(x, wl = 3)
findPeaks(x, wl = 3, thres = 1)
findPeaks(x, wl = 5)
idx = findPeaks(x, wl = 5, thres = 1)
plot(x, type = 'b'); abline(h = 1, lty = 3)
points(idx, x[idx], col = 'blue', pch = 8)
```

---

flatEnv *Flat envelope / compressor*

Description

Applies a compressor - that is, flattens the amplitude envelope of a waveform, reducing the difference in amplitude between loud and quiet sections. This is achieved by dividing the waveform by some function of its smoothed amplitude envelope (Hilbert, peak or root mean square).

Usage

```
flatEnv(
  x,
  samplingRate = NULL,
  scale = NULL,
  compression = 1,
  method = c("hil", "rms", "peak")[1],
  windowLength = 50,
  windowLength_points = NULL,
  killDC = FALSE,
  dynamicRange = 40,
  reportEvery = NULL,
  cores = 1,
  saveAudio = NULL,
  plot = FALSE,
  savePlots = NULL,
  col = "blue",
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

```
compressor(
  x,
  samplingRate = NULL,
  scale = NULL,
  compression = 1,
  method = c("hil", "rms", "peak")[1],
  windowLength = 50,
  windowLength_points = NULL,
  killDC = FALSE,
  dynamicRange = 40,
  reportEvery = NULL,
  cores = 1,
  saveAudio = NULL,
  plot = FALSE,
  savePlots = NULL,
  col = "blue",
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
compression	the amount of compression to apply: 0 = none, 1 = maximum
method	hil = Hilbert envelope, rms = root mean square amplitude, peak = peak amplitude per window
windowLength	the length of smoothing window, ms
windowLength_points	the length of smoothing window, points. If specified, overrides windowLength
killDC	if TRUE, dynamically removes DC offset or similar deviations of average waveform from zero (see examples)
dynamicRange	parts of sound quieter than -dynamicRange dB will not be amplified
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
saveAudio	full path to the folder in which to save the compressed sound(s)
plot	if TRUE, plots the original sound, the smoothed envelope, and the compressed sound

savePlots — full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)

col — the color of amplitude contours

width, height, units, res — graphical parameters for saving plots passed to png

... — other graphical parameters passed to points() that control the appearance of amplitude contours, eg lwd, lty, etc.

Value

If the input is a single audio (file, Wave, or numeric vector), returns the compressed waveform as a numeric vector with the original sampling rate and scale. If the input is a folder with several audio files, returns a list of compressed waveforms, one for each file.

Examples

```
a = rnorm(500) * seq(1, 0, length.out = 500)
b = flatEnv(a, 1000, plot = TRUE, windowLength_points = 5)    # too short
c = flatEnv(a, 1000, plot = TRUE, windowLength_points = 450)  # too long
d = flatEnv(a, 1000, plot = TRUE, windowLength_points = 100)  # about right

## Not run:
s = soundgen(sylLen = 1000, ampl = c(0, -40, 0), plot = TRUE)
# playme(s)
s_flat1 = flatEnv(s, 16000, dynamicRange = 60, plot = TRUE,
                  windowLength = 50, method = 'hil')
s_flat2 = flatEnv(s, 16000, dynamicRange = 60, plot = TRUE,
                  windowLength = 10, method = 'rms')
s_flat3 = flatEnv(s, 16000, dynamicRange = 60, plot = TRUE,
                  windowLength = 10, method = 'peak')
# playme(s_flat2)

# Remove DC offset
s1 = c(rep(0, 50), runif(1000, -1, 1), rep(0, 50)) +
     seq(.3, 1, length.out = 1100)
s2 = flatEnv(s1, 16000, plot = TRUE, windowLength_points = 50, killDC = FALSE)
s3 = flatEnv(s1, 16000, plot = TRUE, windowLength_points = 50, killDC = TRUE)

# Compress and save all audio files in a folder
s4 = flatEnv('~/Downloads/temp',
             method = 'peak', compression = .5,
             saveAudio = '~/Downloads/temp/compressed',
             savePlots = '~/Downloads/temp/compressed',
             col = 'green', lwd = 5)
osc(s4[[1]])

## End(Not run)
```

`flatSpectrum` *Flat spectrum*

Description

Flattens the spectrum of a sound by smoothing in the frequency domain. Can be used for removing formants without modifying pitch contour or voice quality (the balance of harmonic and noise components), followed by the addition of a new spectral envelope (cf. `transplantFormants`). Algorithm: makes a spectrogram, flattens the real part of the smoothed spectrum of each STFT frame, and transforms back into time domain with inverse STFT (see also `addFormants`).

Usage

```
flatSpectrum(
  x,
  samplingRate = NULL,
  freqWindow = NULL,
  dynamicRange = 80,
  windowLength = 50,
  step = NULL,
  overlap = 90,
  wn = "gaussian",
  zp = 0,
  play = FALSE,
  saveAudio = NULL,
  reportEvery = NULL,
  cores = 1
)
```

Arguments

`x`	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
`samplingRate`	sampling rate of `x` (only needed if `x` is a numeric vector)
`freqWindow`	the width of smoothing window, Hz. Defaults to median pitch estimated by `analyze`
`dynamicRange`	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
`windowLength`	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
`step`	you can override `overlap` by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
`overlap`	overlap between successive FFT frames, %

wn	window type accepted by ftwindow, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle
zp	window length after zero padding, points
play	if TRUE, plays the processed audio
saveAudio	full (!) path to folder for saving the processed audio; NULL = don't save, '' = same as input folder (NB: overwrites the originals!)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing

Value

Returns a numeric vector with the same sampling rate as the input.

See Also

addFormants transplantFormants

Examples

```
sound_aii = soundgen(formants = 'aii')
# playme(sound_aii, 16000)
seewave::meanspec(sound_aii, f = 16000, dB = 'max0')

sound_flat = flatSpectrum(sound_aii, freqWindow = 150, samplingRate = 16000)
# playme(sound_flat, 16000)
seewave::meanspec(sound_flat, f = 16000, dB = 'max0')
# harmonics are still there, but formants are gone and can be replaced

## Not run:
# Now let's make a sheep say "aii"
data(sheep, package = 'seewave')  # import a recording from seewave
playme(sheep)
sheep_flat = flatSpectrum(sheep)
playme(sheep_flat, sheep@samp.rate)
seewave::spec(sheep_flat, f = sheep@samp.rate, dB = 'max0')

# So far we have a sheep bleating with a flat spectrum;
# now let's add new formants
sheep_aii = addFormants(sheep_flat,
  samplingRate = sheep@samp.rate,
  formants = 'aii',
  lipRad = -3)  # negative lipRad to counter unnatural flat source
playme(sheep_aii, sheep@samp.rate)
spectrogram(sheep_aii, sheep@samp.rate)
seewave::spec(sheep_aii, f = sheep@samp.rate, dB = 'max0')

## End(Not run)
```

formant_app *Interactive formant tracker*

Description

Starts a shiny app for manually correcting formant measurements. For more tips, see pitch_app and http://cogsci.se/soundgen.html.

Usage

```
formant_app()
```

Details

Suggested workflow: load one or several audio files (wav/mp3), preferably not longer than a minute or so. Select a region of interest in the spectrogram - for example, a sustained vowel with clear and relatively steady formants. Double-click within the selection to create a new annotation (you may add a text label if needed). If you are satisfied with the automatically calculated formant frequencies, proceed to the next region of interest. If not, there are 4 ways to adjust them: (1) type in the correct number in one of the formant boxes in the top right corner; (2) click a spectrogram within selection (pick the formant number to adjust by clicking the formant boxes); (3) single-click the spectrum to use the cursor's position, or (4) double-click the spectrum to use the nearest spectral peak. When done with a file, move on to the next one in the queue. Use the orange button to download the results. To continue work, upload the output file from the previous session together with the audio files (you can rename it, but keep the .csv extension). Use hotkeys (eg spacebar to play/stop) and avoid working with very large files.

Value

Every time a new annotation is added, the app creates a backup csv file and creates or updates a global object called "my_formants", which contains all the annotations. When the app is terminated, it also returns the results as a dataframe.

See Also

pitch_app

Examples

```
## Not run:
f = formant_app()  # runs in default browser such as Firefox or Chrome

# To change system default browser, run something like:
options('browser' = '/usr/bin/firefox')  # path to the executable on Linux

## End(Not run)
```

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,
  action = c("blur", "unblur")[1],
  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)
action	'blur' = kernel-weighted average, 'unblur' = subtract kernel-weighted average
plotKernel	if TRUE, plots the kernel

Value

Returns a numeric matrix of the same dimensions as input.

See Also

modulationSpectrum

Examples

```
s = spectrogram(soundgen(), samplingRate = 16000, windowLength = 10,
  output = 'original', plot = FALSE)
s = log(s + .001)
# image(s)
s1 = gaussianSmooth2D(s, kernelSize = 5, plotKernel = TRUE)
# image(s1)

## Not run:
# more smoothing in time than in frequency
s2 = gaussianSmooth2D(s, kernelSize = c(5, 15))
image(s2)
```

```
# vice versa - more smoothing in frequency
s3 = gaussianSmooth2D(s, kernelSize = c(25, 3))
image(s3)

# sharpen the image by deconvolution with the kernel
s4 = gaussianSmooth2D(s1, kernelSize = 5, action = 'unblur')
image(s4)

s5 = gaussianSmooth2D(s, kernelSize = c(15, 1), action = 'unblur')
image(s5)

## End(Not run)
```

`generateNoise` *Generate noise*

Description

Generates noise of length `len` and with spectrum defined by rolloff parameters 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

```
generateNoise(
  len,
  rolloffNoise = 0,
  noiseFlatSpec = 1200,
  rolloffNoiseExp = 0,
  spectralEnvelope = NULL,
  noise = NULL,
  temperature = 0.1,
  attackLen = 10,
  windowLength_points = 1024,
  samplingRate = 16000,
  overlap = 75,
  dynamicRange = 80,
  smoothing = list(),
  invalidArgAction = c("adjust", "abort", "ignore")[1],
  play = FALSE
)
```

Arguments

`len`	length of output

`rolloffNoise`, `noiseFlatSpec`
: linear rolloff of the excitation source for the unvoiced component, `rolloffNoise` dB/kHz (anchor format) applied above `noiseFlatSpec` Hz

`rolloffNoiseExp`
: exponential rolloff of the excitation source for the unvoiced component, dB/oct (anchor format) applied above 0 Hz

`spectralEnvelope`
: (optional): as an alternative to using rolloffNoise, we can provide the exact filter - a vector of non-negative numbers specifying the desired spectrum on a linear scale up to Nyquist frequency (samplingRate / 2). The length doesn't matter as it can be interpolated internally to windowLength_points/2. A matrix specifying the filter for each STFT step is also accepted. The easiest way to obtain spectralEnvelope is to call soundgen:::getSpectralEnvelope or to use the spectrum / spectrogram 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

`smoothing`
: a list of parameters passed to `getSmoothContour` to control the interpolation and smoothing of contours: interpol (approx / spline / loess), loessSpan, discontThres, jumpThres

`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)

`play`
: if TRUE, plays the synthesized sound using the default player on your system. If character, passed to `play` as the name of player to use, eg "aplay", "play", "vlc", etc. 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.

See Also

`soundgen` `fart` `beat`

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 = list(TRUE, FALSE, 'aplay', 'vlc')[[1]]
# 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@samp.rate
# 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', padWithSilence = FALSE)
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)
```

`getDuration` *Get duration*

Description

Returns the duration of one or more audio files (mostly useful for running on an entire folder). If `threshold` is set, it also removes the leading and trailing silences or near-silences, thus returning the duration of relatively loud central fragments of each sound. Silences are located based on the amplitude of root mean square (RMS) amplitude with `getRMS`. Note that the threshold is set relative to the observed maximum RMS, just as in `analyze`. This means that even very quiet sounds are not treated as nothing but silence.

Usage

```
getDuration(
  x,
  samplingRate = NULL,
  silence = 0.01,
  rms = list(windowLength = 20, step = 5),
  reportEvery = NULL,
  cores = 1
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
silence	leading and trailing sections quieter than this proportion of maximum RMS amplitude are removed when calculating duration_noSilence (NULL = don't calculate duration_noSilence to save time)
rms	a list of control parameters passed to getRMS
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing

Value

Returns duration (s) and duration_noSilence (duration without leading and trailing silences).

See Also

analyze getLoudness

Examples

```
s = c(rep(0, 550), runif(400, -1, 1), rep(0, 50))
osc(s, samplingRate = 1000)
# true duration_noSilence is 400 ms
getDuration(s, samplingRate = 1000, silence = .01)
getDuration(s, samplingRate = 1000, silence = .1,
            rms = list(windowLength = 5, step = 1))

## Not run:
d = getDuration('~/Downloads/temp')
hist(d$duration - d$duration_noSilence)

## End(Not run)
```

getEntropy *Entropy*

Description

Returns Weiner or Shannon entropy of an input vector such as the spectrum of a sound. Non-positive input values are converted to a small positive number (convertNonPositive). If all elements are zero, returns NA.

Usage

```
getEntropy(
  x,
  type = c("weiner", "shannon")[1],
  normalize = FALSE,
  convertNonPositive = 1e-10
)
```

Arguments

x	vector of positive floats
type	'shannon' for Shannon (information) entropy, 'weiner' for Weiner entropy
normalize	if TRUE, Shannon entropy is normalized by the length of input vector to range from 0 to 1. It has no affect on Weiner entropy
convertNonPositive	all non-positive values are converted to convertNonPositive

Examples

```
# Here are four simplified power spectra, each with 9 frequency bins:
s = list(
  c(rep(0, 4), 1, rep(0, 4)),       # a single peak in spectrum
  c(0, 0, 1, 0, 0, .75, 0, 0, .5),  # perfectly periodic, with 3 harmonics
  rep(0, 9),                        # a silent frame
  rep(1, 9)                         # white noise
)

# Weiner entropy is ~0 for periodic, NA for silent, 1 for white noise
sapply(s, function(x) round(getEntropy(x), 2))

# Shannon entropy is ~0 for periodic with a single harmonic, moderate for
# periodic with multiple harmonics, NA for silent, highest for white noise
sapply(s, function(x) round(getEntropy(x, type = 'shannon'), 2))

# Normalized Shannon entropy - same but forced to be 0 to 1
sapply(s, function(x) round(getEntropy(x,
  type = 'shannon', normalize = TRUE), 2))
```

getEnv *Get amplitude envelope*

Description

Returns the smoothed amplitude envelope of a waveform on the original scale. Unlike seewave::env, this function always returns an envelope of the same length as the original sound, regardless of the amount of smoothing.

Usage

```
getEnv(
  sound,
  windowLength_points,
  method = c("rms", "hil", "peak", "raw", "mean")[1]
)
```

Arguments

sound	numeric vector
windowLength_points	the length of smoothing window, in points
method	'peak' for peak amplitude per window, 'rms' for root mean square amplitude, 'mean' for mean (for DC offset removal), 'hil' for Hilbert, 'raw' for low-pass filtering the actual sound

Examples

```
a = rnorm(500) * seq(1, 0, length.out = 500)
windowLength_points = 50
scale = max(abs(a))
plot(a, type = 'l', ylim = c(-scale, scale))
points(soundgen:::getEnv(a, windowLength_points, 'rms'),
       type = 'l', col = 'red')
points(soundgen:::getEnv(a, windowLength_points, 'peak'),
       type = 'l', col = 'green')
points(soundgen:::getEnv(a, windowLength_points, 'hil'),
       type = 'l', col = 'blue')
points(soundgen:::getEnv(a, windowLength_points, 'mean'),
       type = 'l', lty = 3, lwd = 3)
```

getHNR *Get HNR*

Description

Calculates the harmonics-to-noise ratio (HNR) - that is, the ratio between the intensity (root mean square amplitude) of the harmonic component and the intensity of the noise component. Normally called by analyze.

Usage

```
getHNR(
  x = NULL,
  samplingRate = NA,
  acf_x = NULL,
  lag.min = 2,
  lag.max = length(x),
  interpol = c("none", "parab", "spline", "sinc")[4],
  wn = "hanning",
  idx_max = NULL
)
```

Arguments

x	time series (a numeric vector)
samplingRate	sampling rate
acf_x	pre-computed autocorrelation function of input x, if already available
lag.min, lag.max	minimum and maximum lag to consider when looking for peaks in the ACF; lag.min = samplingRate/pitchCeiling, lag.max = samplingRate/pitchFloor
interpol	method of improving the frequency resolution by interpolating the ACF: "none" = don't interpolate; "parab" = parabolic interpolation on three points (local peak and its neighbors); "spline" = spline interpolation; "sinc" = sin(x)/x interpolation to a continuous function followed by a search for local peaks using Brent's method
wn	window applied to x (unless acf_x is provided instead of x) as well as to the sinc interpolation
idx_max	(internal) the index of the peak to investigate, if already estimated

Value

A list of three components: f0 = frequency corresponding to the peak of the autocorrelation function; max_acf = amplitude of the peak of the autocorrelation function on a scale of (0, 1); HNR = 10 * log10(x / (1 - max_acf)).

References

Boersma, P. (1993). Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. In Proceedings of the institute of phonetic sciences (Vol. 17, No. 1193, pp. 97-110).

Examples

```
signal = sin(2 * pi * 150 * (1:16000)/16000)
signal = signal / sqrt(mean(signal^2))
noise = rnorm(16000)
noise = noise / sqrt(mean(noise^2))
SNR = 40
s = signal + noise * 10^(-SNR/20)
soundgen:::getHNR(s, 16000, lag.min = 16000/1000,
lag.max = 16000/75, interpol = 'none')
soundgen:::getHNR(s, 16000, lag.min = 16000/1000,
lag.max = 16000/75, interpol = 'parab')
soundgen:::getHNR(s, 16000, lag.min = 16000/1000,
lag.max = 16000/75, interpol = 'spline')
soundgen:::getHNR(s, 16000, lag.min = 16000/1000,
lag.max = 16000/75, interpol = 'sinc')
```

---

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.

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 mimimum 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

```
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

```
getLoudness(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  windowLength = 50,
  step = NULL,
  overlap = 50,
  SPL_measured = 70,
  Pref = 2e-05,
  spreadSpectrum = TRUE,
  summaryFun = c("mean", "median", "sd"),
  reportEvery = NULL,
```

```
  cores = 1,
  plot = TRUE,
  savePlots = NULL,
  main = NULL,
  ylim = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  mar = c(5.1, 4.1, 4.1, 4.1),
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
SPL_measured	sound pressure level at which the sound is presented, dB
Pref	reference pressure, Pa (currently has no effect on the estimate)
spreadSpectrum	if TRUE, applies a spreading function to account for frequency masking
summaryFun	functions used to summarize each acoustic characteristic, eg "c('mean', 'sd')"; user-defined functions are fine (see examples); NAs are omitted automatically for mean/median/sd/min/max/range/sum, otherwise take care of NAs yourself
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
main	plot title
ylim	frequency range to plot, kHz (defaults to 0 to Nyquist frequency). NB: still in kHz, even if yScale = bark, mel, or ERB

`width`, `height`, `units`, `res`
: graphical parameters for saving plots passed to `png`

`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 with `powspec`, converts to bark scale with (`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:

specSone spectrum in bark-sone (one per file): a matrix of loudness values in sone, with frequency on the bark scale in rows and time (STFT frames) in columns

loudness a vector of loudness in sone per STFT frame (one per file)

summary a dataframe of summary loudness measures (one row per file)

References

- ISO 226 as implemented by Jeff Tackett (2005) on https://www.mathworks.com/matlabcentral/fileexchange/7028-iso-226-equal-loudness-level-contour-signal
- Timoney, J., Lysaght, T., Schoenwiesner, M., & MacManus, L. (2004). Implementing loudness models in matlab.
- Yang, W. (1999). Enhanced Modified Bark Spectral Distortion (EMBSD): An Objective Speech Quality Measure Based on Audible Distortion and Cognitive Model. Temple University.

See Also

`getRMS` `analyze`

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
)
l = getLoudness(
    x = sounds, samplingRate = 16000, scale = 1,
    windowLength = 20, step = NULL,
    overlap = 50, SPL_measured = 40,
    Pref = 2e-5, plot = FALSE)
```

```
l$summary
# 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()
# playme(s)
l1 = getLoudness(s, samplingRate = 16000, SPL_measured = 70)
l1$summary
# The estimated loudness in sone depends on target SPL
l2 = getLoudness(s, samplingRate = 16000, SPL_measured = 40)
l2$summary

# ...but not (much) on windowLength and samplingRate
l3 = getLoudness(s, samplingRate = 16000, SPL_measured = 40, windowLength = 50)
l3$summary

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

...or a folder with multiple audio files
getLoudness('~/Downloads/temp2', plot = FALSE)$summary
# Compare:
analyze('~/Downloads/temp2', pitchMethods = NULL,
        plot = FALSE, silence = 0)$summary$loudness_mean
# (per STFT frame; should be similar if silence = 0, because
# otherwise analyze() discards frames considered silent)

# custom summaryFun
ran = function(x) diff(range(x))
getLoudness('~/Downloads/temp2', plot = FALSE,
            summaryFun = c('mean', 'ran'))$summary

## End(Not run)
```

`getPitchZc` *Zero-crossing rate*

Description

A less precise, but very quick method of pitch tracking based on measuring zero-crossing rate in low-pass-filtered audio. Recommended for processing long recordings with typical pitch values well below the first formant frequency, such as speech. Calling this function is considerably faster than using the same pitch-tracking method in `analyze`. Note that, unlike analyze(), it returns the times of individual zero crossings (hopefully corresponding to glottal cycles) instead of pitch values at fixed time intervals.

Usage

```
getPitchZc(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  pitchFloor = 50,
  pitchCeiling = 400,
  zcThres = 0.1,
  zcWin = 5,
  silence = 0.04,
  envWin = 5,
  summaryFun = c("mean", "sd"),
  reportEvery = NULL
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
pitchFloor, pitchCeiling	absolute bounds for pitch candidates (Hz)
zcThres	pitch candidates with certainty below this value are treated as noise and set to NA (0 = anything goes, 1 = pitch must be perfectly stable over zcWin)
zcWin	certainty in pitch candidates depends on how stable pitch is over zcWin glottal cycles (odd integer > 3)
silence	minimum root mean square (RMS) amplitude, below which pitch candidates are set to NA (NULL = don't consider RMS amplitude)
envWin	window length for calculating RMS envelope, ms
summaryFun	functions used to summarize each acoustic characteristic; see analyze
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)

Details

Algorithm: the audio is bandpass-filtered from pitchFloor to pitchCeiling, and the timing of all zero crossings is saved. This is not enough, however, because unvoiced sounds like white noise also have plenty of zero crossings. Accordingly, an attempt is made to detect voiced segments (or steady musical tones, etc.) by looking for stable regions, with several zero-crossings at relatively regular intervals (see parameters zcThres and zcWin). Very quiet parts of audio are also treated as not having a pitch.

Value

Returns a dataframe containing

time time stamps of all zero crossings except the last one, after bandpass-filtering

pitch pitch calculated from the time between consecutive zero crossings

cert certainty in each pitch candidate calculated from local pitch stability, 0 to 1

See Also

analyze

Examples

```
data(sheep, package = 'seewave')
# spectrogram(sheep)
zc = getPitchZc(sheep, pitchCeiling = 250)
plot(zc$detailed[, c('time', 'pitch')], type = 'b')

# Convert to a standard pitch contour sampled at regular time intervals:
pitch = getSmoothContour(
  anchors = data.frame(time = zc$detailed$time, value = zc$detailed$pitch),
  len = 1000, NA_to_zero = FALSE, discontThres = 0)
spectrogram(sheep, extraContour = pitch, ylim = c(0, 2))

## Not run:
# process all files in a folder
zc = getPitchZc('~/Downloads/temp')
zc$summary

## End(Not run)
```

---

getPrior *Get prior for pitch candidates*

Description

Prior for adjusting the estimated pitch certainties in analyze. For ex., if primarily working with speech, we could prioritize pitch candidates in the expected pitch range (100-1000 Hz) and decrease our confidence in candidates with very high or very low frequency as unlikely but still remotely possible. You can think of this as a "soft" alternative to setting absolute pitch floor and ceiling. Algorithm: the multiplier for each pitch candidate is the density of prior distribution with mean = priorMean (Hz) and sd = priorSD (semitones) normalized so max = 1 over [pitchFloor, pitchCeiling]. Useful for previewing the prior given to analyze.

Usage

```
getPrior(
  priorMean,
  priorSD,
  distribution = c("normal", "gamma")[1],
  pitchFloor = 75,
  pitchCeiling = 3000,
  len = 100,
  plot = TRUE,
  pitchCands = NULL,
  ...
)
```

Arguments

priorMean, priorSD	specifies the mean (Hz) and standard deviation (semitones) of gamma distribution describing our prior knowledge about the most likely pitch values for this file. For ex., priorMean = 300, priorSD = 6 gives a prior with mean = 300 Hz and SD = 6 semitones (half an octave). To avoid using any priors, set priorMean = NA, priorAdapt = FALSE
distribution	the shape of prior distribution on the musical scale: 'normal' (mode = priorMean) or 'gamma' (skewed to lower frequencies)
pitchFloor, pitchCeiling	absolute bounds for pitch candidates (Hz)
len	the required length of output vector (resolution)
plot	if TRUE, plots the prior
pitchCands	a matrix of pitch candidate frequencies (for internal soundgen use)
...	additional graphical parameters passed on to plot()

Value

Returns a numeric vector of certainties of length len if pitchCands is NULL and a numeric matrix of the same dimensions as pitchCands otherwise.

See Also

analyze pitch_app

Examples

```
soundgen:::getPrior(priorMean = 150,  # Hz
                    priorSD = 2)      # semitones
soundgen:::getPrior(150, 6)
s = soundgen:::getPrior(450, 24, pitchCeiling = 6000)
plot(s, type = 'l')
```

`getRandomWalk` *Random walk*

Description

Generates a random walk with flexible control over its range, trend, and smoothness. It works by calling stats::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

```
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, basically the number of points used to construct the rw as a proportion of len, from 0 (no smoothing) to 1 (maximum smoothing to a straight line)
`method`	specifies the method of smoothing: either linear interpolation ('linear', see stats::approx) or cubic splines ('spline', see stats::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 = 0))
plot(getRandomWalk(len = 1000, rw_range = 5, rw_smoothing = .2))
plot(getRandomWalk(len = 1000, rw_range = 5, rw_smoothing = .95))
plot(getRandomWalk(len = 1000, rw_range = 5, rw_smoothing = .99))
plot(getRandomWalk(len = 1000, rw_range = 5, rw_smoothing = 1))
plot(getRandomWalk(len = 1000, rw_range = 15,
```

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

---

getRMS *RMS amplitude*

Description

Calculates root mean square (RMS) amplitude in overlapping windows, providing an envelope of RMS amplitude - 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

```
getRMS(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  windowLength = 50,
  step = NULL,
  overlap = 70,
  stereo = c("left", "right", "average", "both")[1],
  killDC = FALSE,
  normalize = TRUE,
  windowDC = 200,
  summaryFun = "mean",
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = NULL,
  main = NULL,
  xlab = "",
  ylab = "",
  type = "b",
  col = "green",
  lwd = 2,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
stereo	'left' = only left channel, 'right' = only right channel, 'average' = take the mean of the two channels, 'both' = return RMS for both channels separately
killDC	if TRUE, removed DC offset (see also flatEnv)
normalize	if TRUE, the RMS amplitude is returned as proportion of the maximum possible amplitude as given by scale
windowDC	the window for calculating DC offset, ms
summaryFun	functions used to summarize each acoustic characteristic; see analyze
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	if TRUE, plot a contour of RMS amplitude
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
xlab, ylab, main	general graphical parameters
type, col, lwd	graphical parameters pertaining to the RMS envelope
width, height, units, res	graphical parameters for saving plots passed to png
...	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(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 list containing:

$detailed: a list of RMS amplitudes per frame for each sound, on the scale of input; names give time stamps for the center of each frame, in ms.

$summary: a dataframe with summary measures, one row per sound

See Also

analyze getLoudness

Examples

```
s = soundgen() + .25  # with added DC offset
# osc(s)
r = getRMS(s, samplingRate = 16000, from = .05,
  windowLength = 40, overlap = 50, killDC = TRUE,
  plot = TRUE, type = 'l', lty = 2, main = 'RMS envelope')
r
# short window = jagged envelope
r = getRMS(s, samplingRate = 16000,
  windowLength = 5, overlap = 0, killDC = TRUE,
  plot = TRUE, col = 'blue', pch = 13, main = 'RMS envelope')

 # stereo
 wave_stereo = tuneR::Wave(
   left = runif(1000, -1, 1) * 16000,
   right = runif(1000, -1, 1) / 3 * 16000,
   bit = 16, samp.rate = 4000)
 getRMS(wave_stereo)$summary
 getRMS(wave_stereo, stereo = 'right')$summary
 getRMS(wave_stereo, stereo = 'average')$summary
 getRMS(wave_stereo, from = .05,
   stereo = 'both', plot = TRUE)$summary

## Not run:
r = getRMS('~/Downloads/temp', savePlots = '~/Downloads/temp/plots')
r$summary

# Compare:
analyze('~/Downloads/temp', pitchMethods = NULL,
        plot = FALSE)$ampl_mean
# (per STFT frame, but should be very similar)

User-defined summary functions:
ran = function(x) diff(range(x))
meanSD = function(x) {
  paste0('mean = ', round(mean(x), 2), '; sd = ', round(sd(x), 2))
}
getRMS('~/Downloads/temp', summaryFun = c('mean', 'ran', 'meanSD'))$summary

## 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

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.

See Also

soundgen

Examples

```
# 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
# Compare:
s1 = soundgen(sylLen = 1000, pitch = 250,
              rolloff = c(-24, -2, -18), plot = TRUE)
s2 = soundgen(sylLen = 1000, pitch = 250,
              rolloff = data.frame(time = c(0, .2, 1),
                                   value = c(-24, -2, -18)),
              plot = TRUE)

# Also works for rolloffOct, rolloffParab, etc:
s3 = soundgen(sylLen = 1000, pitch = 250,
             rolloffParab = 20, rolloffParabHarm = 1:15, plot = TRUE)

## End(Not run)
```

`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. This function is mostly intended to be used internally by soundgen, more precisely to construct (upsample) smooth curves from a number of anchors. For general upsampling or downsampling of audio, use `resample`. 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],
  loessSpan = NULL,
  discontThres = 0.05,
  jumpThres = 0.01,
  valueFloor = NULL,
  valueCeiling = NULL,
  plot = FALSE,
  xlim = NULL,
  ylim = NULL,
  xlab = "Time, ms",
  ylab = ifelse(thisIsPitch, "Frequency, Hz", "Amplitude"),
  main = ifelse(thisIsPitch, "Pitch contour", ""),
  samplingRate = 16000,
  voiced = NULL,
  contourLabel = NULL,
  NA_to_zero = TRUE,
  ...
)
```

Arguments

anchors	a numeric vector of values or a list/dataframe with one column (value) or two columns (time and value). achors$time can be in ms (with len=NULL) or in arbitrary units, eg 0 to 1 (with duration determined by len, which must then be provided in ms). So anchors$time is assumed to be in ms if len=NULL and relative if len is specified. anchors$value can be on any scale.
len	the required length of the output contour. If NULL, it will be calculated based on the maximum time value (in ms) and samplingRate
thisIsPitch	(boolean) is this a pitch contour? If true, log-transforms before smoothing and plots in both Hz and musical notation
normalizeTime	if TRUE, normalizes anchors$time values to range from 0 to 1
interpol	method of interpolation between anchors: "approx" = linear with approx, "spline" = cubic splines with spline, "loess" = local polynomial regression with loess
loessSpan	controls the amount of smoothing when interpolating between anchors with loess, so only has an effect if interpol = 'loess' (1 = strong, 0.5 = weak smoothing)
discontThres	if two anchors are closer in time than discontThres (on a 0-1 scale, ie specified as proportion of total length), the contour is broken into segments with a linear transition between these segments
jumpThres	if anchors are closer than jumpThres, a new section starts with no transition at all (e.g. for adding pitch jumps)
valueFloor, valueCeiling	lowser/upper bounds for the contour

plot	(boolean) produce a plot?
xlim, ylim, xlab, ylab, main	plotting options
samplingRate	sampling rate used to convert time values to points (Hz)
voiced, contourLabel	graphical pars for plotting breathing contours (see examples below)
NA_to_zero	if TRUE, all NAs are replaced with zero
...	other plotting options passed to plot()

Value

Returns a numeric vector of length len.

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 = 16000) # 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)
# compare:
plot(a2)
plot(a3)  # NB: the segment before the jump is upsampled to compensate
```

```
## Control the amount of smoothing
getSmoothContour(c(1, 3, 9, 10, 9, 9, 2), len = 100, plot = TRUE,
  loessSpan = NULL)  # default amount of smoothing (depends on dur)
getSmoothContour(c(1, 3, 9, 10, 9, 9, 2), len = 100, plot = TRUE,
  loessSpan = .85)   # more smoothing than default
getSmoothContour(c(1, 3, 9, 10, 9, 9, 2), len = 100, plot = TRUE,
  loessSpan = .5)    # less smoothing
getSmoothContour(c(1, 3, 9, 10, 9, 9, 2), len = 100, plot = TRUE,
  interpol = 'approx')  # linear interpolation (no smoothing)

## Upsample preserving leading and trailing NAs
anchors = data.frame(time =  c(1,  4,  5,  7,  10, 20, 23, 25, 30),
                     value = c(NA, NA, 10, 15, 12, NA, 17, 15, NA))
plot(anchors, type = 'b')
anchors_ups = getSmoothContour(
  anchors, len = 200,
  interpol = 'approx',  # only approx can propagate NAs
  NA_to_zero = FALSE,   # preserve NAs
  discontThres = 0)     # don't break into sub-contours
plot(anchors_ups, type = 'b')
```

`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 width 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,
  mouthOpenThres = 0.2,
  openMouthBoost = 0,
  vocalTract = NULL,
  temperature = 0.05,
  formDrift = 0.3,
  formDisp = 0.2,
```

```
  formantDepStoch = 1,
  smoothLinearFactor = 1,
  formantCeiling = 2,
  samplingRate = 16000,
  speedSound = 35400,
  smoothing = list(),
  output = c("simple", "detailed")[1],
  plot = FALSE,
  duration = NULL,
  colorTheme = c("bw", "seewave", "...")[1],
  col = NULL,
  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 "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. `formants = NA` defaults to schwa. Time stamps for formants and mouthOpening can be specified in ms or an any other arbitrary scale.
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)
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 (anchor format)
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	multiplication factor for the amplitude of additional formants added above the highest specified formant (0 = none, 1 = default)
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
formantCeiling	frequency to which stochastic formants are calculated, in multiples of the Nyquist frequency; increase up to ~10 for long vocal tracts to avoid losing energy in the upper part of the spectrum
samplingRate	sampling frequency, Hz
speedSound	speed of sound in warm air, cm/s. Stevens (2000) "Acoustic phonetics", p. 138
smoothing	a list of parameters passed to getSmoothContour to control the interpolation and smoothing of contours: interpol (approx / spline / loess), loessSpan, discontThres, jumpThres
output	"simple" returns just the spectral filter, while "detailed" also returns a data.frame of formant frequencies over time (needed for internal purposes such as formant locking)
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')
col	actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)
xlab, ylab	labels of axes
...	other graphical parameters passed on to image()

Value

Returns a spectral filter: a matrix with frequency bins in rows and time steps in columns. Accordingly, rownames of the output give central frequency of each bin (in kHz), while colnames give time stamps (in ms if duration is specified, otherwise 0 to 1).

Examples

```
# [a] with only F1-F3 visible, with no stochasticity
e = getSpectralEnvelope(nr = 512, nc = 50, duration = 300,
  formants = soundgen:::convertStringToFormants('a'),
  temperature = 0, plot = TRUE, col = heat.colors(150))
# image(t(e))  # to plot the output on a linear scale instead of dB

# some "wiggling" of specified formants plus extra formants on top
```

```
e = getSpectralEnvelope(nr = 512, nc = 50,
  formants = c(860, 1430, 2900),
  temperature = 0.1, formantDepStoch = 1, plot = TRUE)

# a schwa based on variable length of vocal tract
e = getSpectralEnvelope(nr = 512, nc = 100, formants = NA,
  vocalTract = list(time = c(0, .4, 1), value = c(13, 18, 17)),
  temperature = .1, plot = TRUE)

# no formants at all, only lip radiation
e = getSpectralEnvelope(nr = 512, nc = 50, lipRad = 6,
  formants = NA, temperature = 0, plot = FALSE)
plot(e[, 1], type = 'l')              # linear scale
plot(20 * log10(e[, 1]), type = 'l')  # dB scale - 6 dB/oct

# 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, 0, .5)))

# scale formant amplitude and/or bandwidth
e1 = getSpectralEnvelope(nr = 512, nc = 50,
  formants = soundgen:::convertStringToFormants('a'),
  formantWidth = 1, formantDep = 1)  # defaults
e2 = getSpectralEnvelope(nr = 512, nc = 50,
  formants = soundgen:::convertStringToFormants('a'),
  formantWidth = 1.5, formantDep = 1.5)
plot(as.numeric(rownames(e2)), 20 * log10(e2[, 1]),
     type = 'l', xlab = 'KHz', ylab = 'dB', col = 'red', lty = 2)
points(as.numeric(rownames(e1)), 20 * log10(e1[, 1]), type = 'l')

# manual specification of formants
e3 = getSpectralEnvelope(
  nr = 512, nc = 50, samplingRate = 16000, plot = TRUE,
  formants = list(
    f1 = list(freq = c(900, 500), amp = c(30, 35), width = c(80, 50)),
    f2 = list(freq = c(1900, 2500), amp = c(25, 30), width = 100),
    f3 = list(freq = 3400, amp = 30, width = 120)
))

# extra zero-pole pair (doesn't affect estimated VTL and thus the extra
# formants added on top)
e4 = getSpectralEnvelope(
  nr = 512, nc = 50, samplingRate = 16000, plot = TRUE,
  formants = list(
    f1 = list(freq = c(900, 500), amp = c(30, 35), width = c(80, 50)),
    f1.5 = list(freq = 1300, amp = -15),
    f1.7 = list(freq = 1500, amp = 15),
    f2 = list(freq = c(1900, 2500), amp = c(25, 30), width = 100),
    f3 = list(freq = 3400, amp = 30, width = 120)
))
plot(as.numeric(rownames(e4)), 20 * log10(e3[, ncol(e3)]),
     type = 'l', xlab = 'KHz', ylab = 'dB')
```

```
points(as.numeric(rownames(e4)), 20 * log10(e4[, ncol(e4)]),
       type = 'l', col = 'red', lty = 2)
```

---

getSurprisal *Get surprisal*

Description

Tracks the (un)predictability of spectral changes in a sound over time, returning a continuous contour of "surprisal". This is an attempt to track auditory salience over time - that is, to identify parts of a sound that are likely to involuntarily attract the listeners' attention. The functions returns surprisal proper ('$surprisal') and its product with increases in loudness ('$surprisalLoudness'). Because getSurprisal() is slow and experimental, it is not called by analyze().

Usage

```
getSurprisal(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  winSurp = 2000,
 audSpec_pars = list(filterType = "butterworth", nFilters = 64, step = 20, yScale =
    "bark"),
  method = c("acf", "np")[1],
  summaryFun = "mean",
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = NULL,
  osc = c("none", "linear", "dB")[2],
  heights = c(3, 1),
  ylim = NULL,
  contrast = 0.2,
  brightness = 0,
  maxPoints = c(1e+05, 5e+05),
  padWithSilence = TRUE,
  colorTheme = c("bw", "seewave", "heat.colors", "...")[1],
  col = NULL,
  extraContour = NULL,
  xlab = NULL,
  ylab = NULL,
  xaxp = NULL,
  mar = c(5.1, 4.1, 4.1, 2),
  main = NULL,
  grid = NULL,
```

```
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
winSurp	surprisal analysis window, ms (Inf = from sound onset to each point)
audSpec_pars	a list of parameters passed to audSpectrogram
method	acf = change in maximum autocorrelation after adding the final point, np = non-linear prediction (see nonlinPred)
summaryFun	functions used to summarize each acoustic characteristic, eg "c('mean', 'sd')"; user-defined functions are fine (see examples); NAs are omitted automatically for mean/median/sd/min/max/range/sum, otherwise take care of NAs yourself
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	if TRUE, plots the auditory spectrogram and the suprisalLoudness contour
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
osc	"none" = no oscillogram; "linear" = on the original scale; "dB" = in decibels
heights	a vector of length two specifying the relative height of the spectrogram and the oscillogram (including time axes labels)
ylim	frequency range to plot, kHz (defaults to 0 to Nyquist frequency). NB: still in kHz, even if yScale = bark, mel, or ERB
contrast	a number, recommended range -1 to +1. The spectrogram is raised to the power of exp(3 * contrast). Contrast >0 increases sharpness, <0 decreases sharpness
brightness	how much to "lighten" the image (>0 = lighter, <0 = darker)
maxPoints	the maximum number of "pixels" in the oscillogram (if any) and spectrogram; good for quickly plotting long audio files; defaults to c(1e5, 5e5); does not affect reassigned spectrograms
padWithSilence	if TRUE, pads the sound with just enough silence to resolve the edges properly (only the original region is plotted, so the apparent duration doesn't change)
colorTheme	black and white ('bw'), as in seewave package ('seewave'), matlab-type palette ('matlab'), or any palette from palette such as 'heat.colors', 'cm.colors', etc

col	actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)
extraContour	a vector of arbitrary length scaled in Hz (regardless of yScale!) that will be plotted over the spectrogram (eg pitch contour); can also be a list with extra graphical parameters such as lwd, col, etc. (see examples)
xlab, ylab, main, mar, xaxp	graphical parameters for plotting
grid	if numeric, adds n = grid dotted lines per kHz
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters

Details

Algorithm: we start with an auditory spectrogram produced by applying a bank of bandpass filters to the signal, by default with central frequencies equally spaced on the bark scale (see audSpectrogram). For each frequency channel, a sliding window is analyzed to compare the actually observed final value with its expected value. There are many ways to extrapolate / predict time series and thus perform this comparison such as autocorrelation (method = 'acf') or nonlinear prediction (method = 'np'). The resulting per-channel surprisal contours are aggregated by taking their mean weighted by the average amplitude of each frequency channel across the analysis window. Because increases in loudness are known to be important predictors of auditory salience, loudness per frame is also returned, as well as the square root of the product of its derivative and surprisal.

Value

Returns a list with $detailed per-frame and $summary per-file results (see analyze for more information). Three measures are reported: loudness (in sone, as per getLoudness), the first derivative of loudness with respect to time (dLoudness), surprisal (non-negative), and suprisalLoudness (geometric mean of surprisal and dLoudness, treating negative values of dLoudness as zero).

Examples

```
# A quick example
s = soundgen(nSyl = 2, sylLen = 50, pauseLen = 25, addSilence = 15)
surp = getSurprisal(s, samplingRate = 16000)
surp

## Not run:
# A more meaningful example
sound = soundgen(nSyl = 5, sylLen = 150,
  pauseLen = c(50, 50, 50, 130), pitch = c(200, 150),
  noise = list(time = c(-300, 200), value = -20), plot = TRUE)
# playme(sound)
surp = getSurprisal(sound, samplingRate = 16000,
  yScale = 'bark', method = 'acf')
surp = getSurprisal(sound, samplingRate = 16000,
  yScale = 'bark', method = 'np')  # very slow
surp = getSurprisal(sound, samplingRate = 16000,
```

```
  yScale = 'bark', method = 'acf', audSpec_pars = list(
  nFilters = 128, yScale = 'ERB', bandwidth = 1/12))

# short window = amnesia (every event is equally surprising)
getSurprisal(sound, samplingRate = 16000, winSurp = 250)
# long window - remembers further into the past, Inf = from the beginning
surp = getSurprisal(sound, samplingRate = 16000, winSurp = Inf)

# plot "pure" surprisal, without weighting by loudness
spectrogram(sound, 16000, extraContour = surp$detailed$surprisal /
  max(surp$detailed$surprisal, na.rm = TRUE) * 8000)

# NB: surprisalLoudness contour is also log-transformed if yScale = 'log',
# so zeros become NAs
surp = getSurprisal(sound, samplingRate = 16000, yScale = 'log')

# add bells and whistles
surp = getSurprisal(sound, samplingRate = 16000,
  yScale = 'mel',
  osc = 'dB',  # plot oscillogram in dB
  heights = c(2, 1),  # spectro/osc height ratio
  brightness = -.1,  # reduce brightness
  # colorTheme = 'heat.colors',  # pick color theme...
  col = rev(hcl.colors(30, palette = 'Viridis')),  # ...or specify the colors
  cex.lab = .75, cex.axis = .75,  # text size and other base graphics pars
  ylim = c(0, 5),  # always in kHz
  main = 'Audiogram with surprisal contour', # title
  extraContour = list(col = 'blue', lty = 2, lwd = 2)
  # + axis labels, etc
)

surp = getSurprisal('~/Downloads/temp/', savePlots = '~/Downloads/temp/surp')
surp$summary

## End(Not run)
```

`hillenbrand` *Formants in American vowels*

Description

Typical relative frequencies of the first four formants measured in dF units (average spacing between formants, or formant dispersion) above or below schwa based on estimated VTL in American English, from Hillenbrand (1995), who measured F1-F4 in ~1.5K recordings (139 speakers, 12 vowels from each). Audio and formant measurements are freely available online: https://homepages.wmich.edu/~hillenbr/voweldata. The dataset below is the result of modeling Hillenbrand's data with brms: mvbind(F1rel, F2rel) ~ vowel + (vowel|speaker). It shows the most credible location of each vowel centroid in the F1Rel-F2Rel space.

Usage

```
hillenbrand
```

Format

An object of class `data.frame` with 12 rows and 5 columns.

Details

A dataframe of 12 observations and 5 columns: "vowel" = vowel (American English), "F1Rel" to "F4Rel" = formant frequencies in dF relative to their neutral, equidistant positions in a perfectly cylindrical vocal tract. See `schwa` - this is what schwa() returns as $ff_relative_dF

References

Hillenbrand, J., Getty, L. A., Clark, M. J., & Wheeler, K. (1995). Acoustic characteristics of American English vowels. The Journal of the Acoustical society of America, 97(5), 3099-3111.

Examples

```
plot(hillenbrand$F1Rel, hillenbrand$F2Rel, type = 'n')
text(hillenbrand$F1Rel, hillenbrand$F2Rel, labels = hillenbrand$vowel)
```

HzToERB *Convert Hz to ERB rate*

Description

Converts from Hz to the number of Equivalent Rectangular Bandwidths (ERBs) below input frequency. See https://www2.ling.su.se/staff/hartmut/bark.htm and https://en.wikipedia.org/wiki/Equivalent_rectangular_bandw

Usage

```
HzToERB(h, method = c("linear", "quadratic")[1])
```

Arguments

`h`	vector or matrix of frequencies (Hz)
`method`	approximation to use

See Also

`ERBToHz` `HzToSemitones` `HzToNotes`

Examples

```
HzToERB(c(-20, 20, 100, 440, 1000, NA))

f = 20:20000
erb_lin = HzToERB(f, 'linear')
erb_quadratic = HzToERB(f, 'quadratic')
plot(f, erb_lin, log = 'x', type = 'l')
points(f, erb_quadratic, col = 'blue', type = 'l')

# compare with the bark scale:
barks = tuneR::hz2bark(f)
points(f, barks / max(barks) * max(erb_lin),
  col = 'red', type = 'l', lty = 2)
```

HzToNotes *Convert Hz to notes*

Description

Converts from Hz to musical notation like A4 - note A of the fourth octave above C0 (16.35 Hz).

Usage

```
HzToNotes(h, showCents = FALSE, A4 = 440)
```

Arguments

h	vector or matrix of frequencies (Hz)
showCents	if TRUE, show cents to the nearest notes (cent = 1/100 of a semitone)
A4	frequency of note A in the fourth octave (modern standard ISO 16 or concert pitch = 440 Hz)

See Also

notesToHz HzToSemitones

Examples

```
HzToNotes(c(440, 293, 115, 16.35, 4))

HzToNotes(c(440, 415, 80, 81), showCents = TRUE)
# 80 Hz is almost exactly midway (+49 cents) between D#2 and E2

# Baroque tuning A415, half a semitone flat relative to concert pitch A440
HzToNotes(c(440, 415, 16.35), A4 = 415)
```

HzToSemitones *Convert Hz to semitones*

Description

Converts from Hz to semitones above C-5 (~0.5109875 Hz) or another reference frequency. This may not seem very useful, but note that this gives us a nice logarithmic scale for generating natural pitch transitions.

Usage

```
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)

See Also

semitonesToHz HzToNotes

Examples

```
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

# Any reference tone can be specified. For ex., for semitones above C0, use:
HzToSemitones(440, ref = 16.35)
# TIP: see notesDict for a table of Hz frequencies to musical notation
```

invertSpectrogram *Invert spectrogram*

Description

Transforms a spectrogram into a time series with inverse STFT. The problem is that an ordinary spectrogram preserves only the magnitude (modulus) of the complex STFT, while the phase is lost, and without phase it is impossible to reconstruct the original audio accurately. So there are a number of algorithms for "guessing" the phase that would produce an audio whose magnitude spectrogram is very similar to the target spectrogram. Useful for certain filtering operations that modify the magnitude spectrogram followed by inverse STFT, such as filtering in the spectrotemporal modulation domain.

Usage

```
invertSpectrogram(
  spec,
  samplingRate,
  windowLength,
  overlap,
  step = NULL,
  wn = "hanning",
  specType = c("abs", "log", "dB")[1],
  initialPhase = c("zero", "random", "spsi")[3],
  nIter = 50,
  normalize = TRUE,
  play = TRUE,
  verbose = FALSE,
  plotError = TRUE
)
```

Arguments

spec	the spectrogram that is to be transform to a time series: numeric matrix with frequency bins in rows and time frames in columns
samplingRate	sampling rate of x (only needed if x is a numeric vector)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
overlap	overlap between successive FFT frames, %
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
wn	window type accepted by ftwindow, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle
specType	the scale of target spectroram: 'abs' = absolute, 'log' = log-transformed, 'dB' = in decibels
initialPhase	initial phase estimate: "zero" = set all phases to zero; "random" = Gaussian noise; "spsi" (default) = single-pass spectrogram inversion (Beauregard et al., 2015)
nIter	the number of iterations of the GL algorithm (Griffin & Lim, 1984), 0 = don't run
normalize	if TRUE, normalizes the output to range from -1 to +1
play	if TRUE, plays back the reconstructed audio
verbose	if TRUE, prints estimated time left every 10% of GL iterations
plotError	if TRUE, produces a scree plot of squared error over GL iterations (useful for choosing 'nIter')

Details

Algorithm: takes the spectrogram, makes an initial guess at the phase (zero, noise, or a more intelligent estimate by the SPSI algorithm), fine-tunes over 'nIter' iterations with the GL algorithm, reconstructs the complex spectrogram using the best phase estimate, and performs inverse STFT. The single-pass spectrogram inversion (SPSI) algorithm is implemented as described in Beauregard et al. (2015) following the python code at https://github.com/lonce/SPSI_Python. The Griffin-Lim (GL) algorithm is based on Griffin & Lim (1984).

Value

Returns the reconstructed audio as a numeric vector.

References

- Griffin, D., & Lim, J. (1984). Signal estimation from modified short-time Fourier transform. IEEE Transactions on Acoustics, Speech, and Signal Processing, 32(2), 236-243.
- Beauregard, G. T., Harish, M., & Wyse, L. (2015, July). Single pass spectrogram inversion. In 2015 IEEE International Conference on Digital Signal Processing (DSP) (pp. 427-431). IEEE.

See Also

spectrogram filterSoundByMS

Examples

```
# Create a spectrogram
samplingRate = 16000
windowLength = 40
overlap = 75
wn = 'gaussian'

s = soundgen(samplingRate = samplingRate, addSilence = 100)
spec = spectrogram(s, samplingRate = samplingRate,
  wn = wn, windowLength = windowLength, step = NULL, overlap = overlap,
  padWithSilence = FALSE, output = 'original')

# Invert the spectrogram, attempting to guess the phase
# Note that samplingRate, wn, windowLength, and overlap must be the same as
# in the original (ie you have to know how the spectrogram was created)
s_new = invertSpectrogram(spec, samplingRate = samplingRate,
  windowLength = windowLength, overlap = overlap, wn = wn,
  initialPhase = 'spsi', nIter = 100, specType = 'abs', play = FALSE)

# Verify the quality of audio reconstruction
# playme(s, samplingRate); playme(s_new, samplingRate)
```

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

```
matchPars(
  target,
  samplingRate = NULL,
  pars = NULL,
  init = NULL,
  probMutation = 0.25,
  stepVariance = 0.1,
  maxIter = 50,
  minExpectedDelta = 0.001,
  compareSoundsPars = list(),
  verbose = TRUE
)
```

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)
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
compareSoundsPars	a list of control parameters passed to compareSounds
verbose	if TRUE, plays back the accepted candidate at each iteration and reports the outcome

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

```
## Not run:
target = soundgen(sylLen = 600, pitch = c(300, 200),
                  rolloff = -15, play = TRUE, plot = TRUE)
# we hope to reproduce this sound

# 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 = TRUE)
cand1 = do.call(soundgen, c(m1$pars, list(
  temperature = 0.001, play = TRUE, plot = TRUE)))

# 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 = TRUE)
# rolloff should be moving from default (-9) to target (-15):
sapply(m2$history, function(x) x$pars$rolloff)
cand2 = do.call(soundgen, c(m2$pars, list(play = TRUE, plot = TRUE)))

## End(Not run)
```

`modulationSpectrum` *Modulation spectrum*

Description

Produces a modulation spectrum of waveform(s) or audio file(s). It begins with some spectrogram-like time-frequency representation and analyzes the modulation of the envelope in each frequency band. if `specSource = 'audSpec'`, the sound is passed through a bank of bandpass filters with `audSpectrogram`. If `specSource = 'STFT'`, we begin with an ordinary spectrogram produced with a Short-Time Fourier Transform. If `msType = '2D'`, the modulation spectrum is a 2D Fourier transform of the spectrogram-like representation, with temporal modulation along the X axis and spectral modulation along the Y axis. A good visual analogy is decomposing the spectrogram into a sum of ripples of various frequencies and directions. If `msType = '1D'`, the modulation spectrum is a matrix containing 1D Fourier transforms of each frequency band in the spectrogram, so the result again has modulation frequencies along the X axis, but the Y axis now shows the frequency of

each analyzed band. Roughness is calculated as the proportion of the modulation spectrum within roughRange of temporal modulation frequencies or some weighted version thereof. The frequency of amplitude modulation (amMsFreq, Hz) is calculated as the highest peak in the smoothed AM function, and its purity (amMsPurity, dB) as the ratio of this peak to the median AM over amRange. For relatively short and steady sounds, set amRes = NULL and analyze the entire sound. For longer sounds and when roughness or AM vary over time, set amRes to get multiple measurements over time (see examples). For multiple inputs, such as a list of waveforms or path to a folder with audio files, the ensemble of modulation spectra can be interpolated to the same spectral and temporal resolution and averaged (if averageMS = TRUE).

Usage

```
modulationSpectrum(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  msType = c("1D", "2D")[2],
  specSource = c("STFT", "audSpec")[1],
  windowLength = 15,
  step = 1,
  wn = "hanning",
  zp = 0,
 audSpec_pars = list(filterType = "butterworth", nFilters = 32, bandwidth = 1/24, yScale
    = "bark", dynamicRange = 120),
  amRes = 5,
  maxDur = 5,
  specMethod = c("spec", "meanspec")[2],
  logSpec = FALSE,
  logMPS = FALSE,
  power = 1,
  normalize = TRUE,
  roughRange = c(30, 150),
  roughMean = NULL,
  roughSD = NULL,
  roughMinFreq = 1,
  amRange = c(10, 200),
  returnMS = TRUE,
  returnComplex = FALSE,
  summaryFun = c("mean", "median", "sd"),
  averageMS = FALSE,
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = NULL,
  logWarpX = NULL,
  logWarpY = NULL,
  quantiles = c(0.5, 0.8, 0.9),
```

```
  kernelSize = 5,
  kernelSD = 0.5,
  colorTheme = c("bw", "seewave", "heat.colors", "...")[1],
  col = NULL,
  main = NULL,
  xlab = "Hz",
  ylab = NULL,
  xlim = NULL,
  ylim = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
msType	'2D' = two-dimensional Fourier transform of a spectrogram; '1D' = separately calculated spectrum of each frequency band
specSource	'STFT' = Short-Time Fourier Transform; 'audSpec' = a bank of bandpass filters (see audSpectrogram)
windowLength, step, wn, zp	parameters for extracting a spectrogram if specType = 'STFT'. Window length and step are specified in ms (see spectrogram). If specType = 'audSpec', these settings have no effect
audSpec_pars	parameters for extracting an auditory spectrogram if specType = 'audSpec'. If specType = 'STFT', these settings have no effect
amRes	target resolution of amplitude modulation, Hz. If NULL, the entire sound is analyzed at once, resulting in a single roughness value (unless it is longer than maxDur, in which case it is analyzed in chunks maxDur s long). If amRes is set, roughness is calculated for windows ~1000/amRes ms long (but at least 3 STFT frames). amRes also affects the amount of smoothing when calculating amMsFreq and amMsPurity
maxDur	sounds longer than maxDur s are split into fragments, and the modulation spectra of all fragments are averaged
specMethod	the function to call when calculating the spectrum of each frequency band (only used when msType = '1D'); 'meanspec' is faster and less noisy, whereas 'spec' produces higher resolution
logSpec	if TRUE, the spectrogram is log-transformed prior to taking 2D FFT

logMPS	if TRUE, the modulation spectrum is log-transformed prior to calculating roughness
power	raise modulation spectrum to this power (eg power = 2 for ^2, or "power spectrum")
normalize	if TRUE, the modulation spectrum of each analyzed fragment maxDur in duration is separately normalized to have max = 1
roughRange	the range of temporal modulation frequencies that constitute the "roughness" zone, Hz
roughMean, roughSD	the mean (Hz) and standard deviation (semitones) of a lognormal distribution used to weight roughness estimates. If either is null, roughness is calculated simply as the proportion of spectrum within roughRange. If both roughMean and roughRange are defined, weights outside roughRange are set to 0; a very large SD (a flat weighting function) gives the same result as just roughRange without any weighting (see examples)
roughMinFreq	frequencies below roughMinFreq (Hz) are ignored when calculating roughness (ie the estimated roughness increases if we disregard very low-frequency modulation, which is often strong)
amRange	the range of temporal modulation frequencies that we are interested in as "amplitude modulation" (AM), Hz
returnMS	if FALSE, only roughness is returned (much faster). Careful with exporting the modulation spectra of a lot of sounds at once as this requires a lot of RAM
returnComplex	if TRUE, returns a complex modulation spectrum (without normalization and warping)
summaryFun	functions used to summarize each acoustic characteristic, eg "c('mean', 'sd')"; user-defined functions are fine (see examples); NAs are omitted automatically for mean/median/sd/min/max/range/sum, otherwise take care of NAs yourself
averageMS	if TRUE, the modulation spectra of all inputs are averaged into a single output; if FALSE, a separate MS is returned for each input
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	if TRUE, plots the modulation spectrum of each sound (see plotMS)
savePlots	if a valid path is specified, a plot is saved in this folder (defaults to NA)
logWarpX, logWarpY	numeric vector of length 2: c(sigma, base) of pseudolog-warping the modulation spectrum, as in function pseudo_log_trans() from the "scales" package
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'), matlab-type palette ('matlab'), or any palette from palette such as 'heat.colors', 'cm.colors', etc

col actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)

xlab, ylab, main, xlim, ylim
graphical parameters

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

... other graphical parameters passed on to filled.contour.mod and contour (see spectrogram)

Value

Returns a list with the following components:

- $original modulation spectrum prior to blurring and log-warping, but after squaring if power = TRUE, a matrix of nonnegative values. Colnames are temporal modulation frequencies (Hz). Rownames are spectral modulation frequencies (cycles/kHz) if msType = '2D' and frequencies of filters or spectrograms bands (kHz) if msType = '1D'.
- $original_list a list of modulation spectra for each analyzed fragment (is amRes is not NULL)
- $processed modulation spectrum after blurring and log-warping
- $complex untransformed complex modulation spectrum (returned only if returnComplex = TRUE)
- $roughness proportion of the modulation spectrum within roughRange of temporal modulation frequencies or a weighted average thereof if roughMean and roughSD are defined, % - a vector if amRes is numeric and the sound is long enough, otherwise a single number
- $roughness_list a list containing frequencies, amplitudes, and roughness values for each analyzed frequency band (1D) or frequency modulation band (2D)
- $amMsFreq frequency of the highest peak, within amRange, of the folded AM function (average AM across all FM bins for both negative and positive AM frequencies), where a peak is a local maximum over amRes Hz. Like roughness, amMsFreq and amMsPurity can be single numbers or vectors, depending on whether the sound is analyzed as a whole or in chunks
- $amMsPurity ratio of the peak at amMsFreq to the median AM over amRange, dB
- $summary dataframe with summaries of roughness, amMsFreq, and amMsPurity

References

- Singh, N. C., & Theunissen, F. E. (2003). Modulation spectra of natural sounds and ethological theories of auditory processing. The Journal of the Acoustical Society of America, 114(6), 3394-3411.

See Also

plotMS spectrogram audSpectrogram analyze

Examples

```
# White noise
ms = modulationSpectrum(rnorm(16000), samplingRate = 16000,
  logSpec = FALSE, power = TRUE,
  amRes = NULL)  # analyze the entire sound, giving a single roughness value
str(ms)

# Harmonic sound
s = soundgen(pitch = 440, amFreq = 100, amDep = 50)
ms = modulationSpectrum(s, samplingRate = 16000, amRes = NULL)
ms[c('roughness', 'amMsFreq', 'amMsPurity')]  # a single value for each
ms1 = modulationSpectrum(s, samplingRate = 16000, amRes = 5)
ms1[c('roughness', 'amMsFreq', 'amMsPurity')]
# measured over time (low values of amRes mean more precision, so we analyze
# longer segments and get fewer values per sound)

# Embellish
ms = modulationSpectrum(s, samplingRate = 16000, logMPS = TRUE,
  xlab = 'Temporal modulation, Hz', ylab = 'Spectral modulation, 1/kHz',
  colorTheme = 'matlab', main = 'Modulation spectrum', lty = 3)

# 1D instead of 2D
modulationSpectrum(s, 16000, msType = '1D', quantiles = NULL,
  col = soundgen:::jet.col(50))

## Not run:
# A long sound with varying AM and a bit of chaos at the end
s_long = soundgen(sylLen = 3500, pitch = c(250, 320, 280),
                  amFreq = c(30, 55), amDep = c(20, 60, 40),
                  jitterDep = c(0, 0, 2))
playme(s_long)
ms = modulationSpectrum(s_long, 16000)
# plot AM over time
plot(x = seq(1, 1500, length.out = length(ms$amMsFreq)), y = ms$amMsFreq,
     cex = 10^(ms$amMsPurity/20) * 10, xlab = 'Time, ms', ylab = 'AM frequency, Hz')
# plot roughness over time
spectrogram(s_long, 16000, ylim = c(0, 4),
  extraContour = list(ms$roughness / max(ms$roughness) * 4000, col = 'blue'))

# As with spectrograms, there is a tradeoff in time-frequency resolution
s = soundgen(pitch = 500, amFreq = 50, amDep = 100, sylLen = 500,
             samplingRate = 44100, plot = TRUE)
# playme(s, samplingRate = 44100)
ms = modulationSpectrum(s, samplingRate = 44100,
  windowLength = 50, step = 50, amRes = NULL)  # poor temporal resolution
ms = modulationSpectrum(s, samplingRate = 44100,
  windowLength = 5, step = 1, amRes = NULL)  # poor frequency resolution
ms = modulationSpectrum(s, samplingRate = 44100,
  windowLength = 15, step = 3, amRes = NULL)  # a reasonable compromise

# Start with an auditory spectrogram instead of STFT
modulationSpectrum(s, 44100, specSource = 'audSpec', xlim = c(-100, 100))
```

```
modulationSpectrum(s, 44100, specSource = 'audSpec',
  logWarpX = c(10, 2), xlim = c(-500, 500),
  audSpec_pars = list(nFilters = 32, filterType = 'gammatone', bandwidth = NULL))

# customize the plot
ms = modulationSpectrum(s, samplingRate = 44100,
  windowLength = 15, step = 2, amRes = NULL,
  kernelSize = 17,  # more smoothing
  xlim = c(-70, 70), ylim = c(0, 4),  # zoom in on the central region
  quantiles = c(.25, .5, .8),  # customize contour lines
  col = rev(rainbow(100)),  # alternative palette
  logWarpX = c(10, 2),  # pseudo-log transform
  power = 2)                    # ^2
# Note the peaks at FM = 2/kHz (from "pitch = 500") and AM = 50 Hz (from
# "amFreq = 50")

# Input can be a wav/mp3 file
ms = modulationSpectrum('~/Downloads/temp/16002_Faking_It_Large_clear.wav')

# Input can be path to folder with audio files. Each file is processed
# separately, and the output can contain an MS per file...
ms1 = modulationSpectrum('~/Downloads/temp', kernelSize = 11,
                         plot = FALSE, averageMS = FALSE)
ms1$summary
names(ms1$original)  # a separate MS per file
# ...or a single MS can be calculated:
ms2 = modulationSpectrum('~/Downloads/temp', kernelSize = 11,
                         plot = FALSE, averageMS = TRUE)
plotMS(ms2$original)
ms2$summary

# Input can also be a list of waveforms (numeric vectors)
ss = vector('list', 10)
for (i in seq_along(ss)) {
  ss[[i]] = soundgen(sylLen = runif(1, 100, 1000), temperature = .4,
    pitch = runif(3, 400, 600))
}
# lapply(ss, playme)
# MS of the first sound
ms1 = modulationSpectrum(ss[[1]], samplingRate = 16000, scale = 1)
# average MS of all 10 sounds
ms2 = modulationSpectrum(ss, samplingRate = 16000, scale = 1, averageMS = TRUE, plot = FALSE)
plotMS(ms2$original)

# 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',
  power = 2)
# note the asymmetry b/c of downsweeps

# "power = 2" returns squared modulation spectrum - note that this affects
```

```
# the roughness measure!
ms$roughness
# compare:
modulationSpectrum(s, samplingRate = 16000, maxDur = .5,
  xlim = c(-25, 25), colorTheme = 'seewave',
  power = 1)$roughness  # much higher roughness

# Plotting with or without log-warping the modulation spectrum:
ms = modulationSpectrum(soundgen(), samplingRate = 16000, plot = TRUE)
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
  logWarpX = c(2, 2), plot = TRUE)

# logWarp and kernelSize have no effect on roughness
# because it is calculated before these transforms:
modulationSpectrum(s, samplingRate = 16000, logWarpX = c(1, 10))$roughness
modulationSpectrum(s, samplingRate = 16000, logWarpX = NA)$roughness
modulationSpectrum(s, samplingRate = 16000, kernelSize = 17)$roughness

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

# Use a lognormal weighting function to calculate roughness
# (instead of just % in roughRange)
modulationSpectrum(s, 16000, roughRange = NULL,
  roughMean = 75, roughSD = 3)$roughness
modulationSpectrum(s, 16000, roughRange = NULL,
  roughMean = 100, roughSD = 12)$roughness
# truncate weights outside roughRange
modulationSpectrum(s, 16000, roughRange = c(30, 150),
  roughMean = 100, roughSD = 1000)$roughness  # very large SD
modulationSpectrum(s, 16000, roughRange = c(30, 150),
  roughMean = NULL)$roughness  # same as above b/c SD --> Inf

# Complex modulation spectrum with phase preserved
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
                        returnComplex = TRUE)
plotMS(abs(ms$complex))  # note the symmetry
# compare:
plotMS(ms$original)

## 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. Normally you will want to set `temperature` very low; the `tempEffects` argument is not supported.

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: overrides the values in `formula1` and `formula2`

Value

A list of two sublists (`$formulas` and `$sounds`), each of length `nMorphs`. For ex., the formula for the second hybrid is `m$formulas[[2]]`, and the waveform is `m$sounds[[2]]`

See Also

soundgen

Examples

```
## Not run:
# write two formulas or copy-paste them from soundgen_app() or presets:
playback = c(TRUE, FALSE)[1]
# [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]])

# 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, subDep = 15, 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
)

# from scream_010 to moan_515b
# (see online demos at http://cogsci.se/soundgen/humans/humans.html)
m = morph(
  formula1 = "soundgen(
    sylLen = 490,
    pitch = list(time = c(0, 80, 250, 370, 490),
    value = c(1000, 2900, 3200, 2900, 1000)),
    rolloff = c(-5, 0, -25), rolloffKHz = 0,
    temperature = 0.001,
    jitterDep = c(.5, 1, 0), shimmerDep = c(5, 15, 0),
    formants = c(1100, 2300, 3100, 4000, 5300, 6200),
    mouth = c(.3, .5, .6, .5, .3))",
  formula2 = "soundgen(sylLen = 520,
    pitch = c(300, 310, 300),
    ampl = c(0, -30),
    temperature = 0.001, rolloff = c(-18, -25),
    jitterDep = .05, shimmerDep = 2,
    formants = list(f1 = c(700, 900),
      f2 = c(1600, 1400),
      f3 = c(3600, 3500), f4 = c(4300, 4200)),
    mouth = c(.5, .3),
    noise = data.frame(time = c(0, 400, 660),
    value = c(-20, -10, -60)),
    rolloffNoise = c(-5, -15))",
 nMorphs = 5, playMorphs = playback
)

## End(Not run)
```

msToSpec *Modulation spectrum to spectrogram*

Description

Takes a complex MS and transforms it to a complex spectrogram with proper row (frequency) and column (time) labels.

Usage

```
msToSpec(ms, windowLength = NULL, step = NULL)
```

Arguments

ms	target modulation spectrum (matrix of complex numbers)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)

Value

Returns a spectrogram - a numeric matrix of complex numbers of the same dimensions as ms.

Examples

```
s = soundgen(sylLen = 250, amFreq = 25, amDep = 50,
             pitch = 250, samplingRate = 16000)
spec = spectrogram(s, samplingRate = 16000, windowLength = 25, step = 5)
ms = specToMS(spec)
plotMS(log(Mod(ms)), quantiles = NULL, col = soundgen:::jet.col(100))
spec_new = msToSpec(ms)
spectrogram(s, specManual = Mod(spec_new))
## Not run:
# or plot manually
image(x = as.numeric(colnames(spec_new)), y = as.numeric(rownames(spec_new)),
      z = t(log(abs(spec_new))), xlab = 'Time, ms',
      ylab = 'Frequency, kHz')

## End(Not run)
```

`naiveBayes` *Naive Bayes*

Description

An implementation of a Naive Bayes classifier adapted to autocorrelated time series such as the type of nonlinear vocal phenomena in consecutive audio frames. All predictors must be continuous, and the outcome must be categorical. Cases with missing values are not deleted because the posterior probabilities of each outcome class can be calculated from different combinations of predictors on a case-by-case basis. Two optional modifications of a standard Naive Bayes algorithm can be made: (1) classifications can be "clumped" at the final stage, ensuring that every run or "epoch" of a particular predicted class is at least `minLength` steps long, and (2) priors can be continuously adapted based on the likelihood function of the preceding `wlPrior` observations if `prior = 'dynamic'`.

Usage

```
naiveBayes(
  formula,
  train,
  test = train,
  prior = c("flat", "static", "dynamic")[2],
  wlPrior = 3,
  wlClumper = NULL,
  runBack = TRUE,
  plot = FALSE
)
```

Arguments

`formula`	model formula of the type outcome ~ predictor1 + predictor2 + ... (no interactions)
`train`	either the training dataframe or the output of `naiveBayes_train`. This data is used to calculate class-specific distributions of the predictors and prior class probabilities
`test`	the test dataframe. This data is used to make predictions - that is, outcome class probabilities given the values of predictors
`prior`	"flat" = all classes are equally likely a prior, "static" = use class probabilities in the training dataset, "dynamic" = update prior probabilities from weighted likelihoods of `wlPrior` preceding observations
`wlPrior`	the length of a Gaussian window used for updating dynamic priors
`wlClumper`	the minimum expected number of observations of the same class before the class can change
`runBack`	if TRUE, the dynamic prior is calculated both forward and backward and averaged (only has an effect f `prior = 'dynamic'`)
`plot`	if TRUE, produces diagnostic plots

Value

Returns the `test` dataframe with new columns: "pr" = the predicted class membership, "[class]" = posterior probabilities per class, "like_[class]" = log-likelihoods, "prior_[class]" = log-priors, "priorF_[class]" / "priorB_[class]" = forward / backward log-priors per class.

Examples

```
set.seed(151)
## create some fake data
df = data.frame(group = rep(c(
    rep('A', 150), rep('B', 50), rep('A', 120),
    rep('A', 100), rep('B', 30), rep('A', 90)
  ), 3))
df$group = as.factor(df$group)
df$x1 = rnorm(nrow(df), mean = ifelse(df$group == 'A', 3, 6), sd = 2)
df$x2 = rnorm(nrow(df), mean = ifelse(df$group == 'A', 2, -1), sd = 2)
boxplot(x1 ~ group, df)
boxplot(x2 ~ group, df)

## train the classifier
mod_train = naiveBayes_train(group ~ x1 + x2, data = df)
mod_train

## test on new data generated by the same process
test = data.frame(group = rep(c(
  rep('A', 90), rep('B', 40), rep('A', 150),
  rep('B', 40), rep('A', 130), rep('B', 30)
), 2))
test$group = as.factor(test$group)
test$x1 = rnorm(nrow(test), mean = ifelse(test$group == 'A', 3, 6), sd = 2)
test$x2 = rnorm(nrow(test), mean = ifelse(test$group == 'A', 2, -1), sd = 2)

# flat priors (same prior probability for each class)
nb_flat = naiveBayes(group ~ x1 + x2, train = mod_train, test = test,
  prior = 'flat', plot = TRUE)
# same as passing 'train' directly to the model, w/o calling naiveBayes_train():
nb_flat = naiveBayes(group ~ x1 + x2, train = df, test = test, prior = 'flat')
table(nb_flat$group, nb_flat$pr)
mean(nb_flat$group == nb_flat$pr) # 84% correct

# static priors (use original class proportions as prior class probabilities)
nb_static = naiveBayes(group ~ x1 + x2, train = mod_train, test = test,
  prior = 'static', wlClumper = NULL, plot = TRUE)
table(nb_static$group, nb_static$pr)
mean(nb_static$group == nb_static$pr) # 87% correct

# specify custom static priors
mod_train2 = mod_train
mod_train$table
mod_train2$table = list(A = .1, B = .9)  # sum to 1
nb_static2 = naiveBayes(group ~ x1 + x2, train = mod_train2, test = test,
  prior = 'static', wlClumper = NULL, plot = TRUE)
```

```
mean(nb_static2$group == nb_static2$pr)  # 61% correct

# if we expect autocorrelation, ie class X is more likely a priori if the
# last few observations were also likely to be class X, we can use dynamic
# priors and/or clumper the predicted classes (the latter imposes strong
# constraints on the predictions, but may be worth it if the data is known to
# be strongly "clumpered", ie if we know classes occur in long'ish runs)
nb1 = naiveBayes(group ~ x1 + x2, train = mod_train, test = test,
  prior = 'dynamic', wlPrior = 10, plot = TRUE)
table(nb1$group, nb1$pr)
mean(nb1$group == nb1$pr) # 94% correct

nb2 = naiveBayes(group ~ x1 + x2, train = mod_train, test = test,
  prior = 'static', wlClumper = 10, plot = TRUE)
table(nb2$group, nb2$pr)
mean(nb2$group == nb2$pr) # 89% correct

nb3 = naiveBayes(group ~ x1 + x2, train = mod_train, test = test,
  prior = 'dynamic', wlPrior = 10, wlClumper = 10, plot = TRUE)
table(nb3$group, nb3$pr)
mean(nb3$group == nb3$pr) # 98% correct

# NAs in the data are not a problem
test1 = test
test1$x1[sample(1:nrow(test1), 100)] = NA
test1$x2[sample(1:nrow(test1), 10)] = NA
summary(test1)

nb4 = naiveBayes(group ~ x1 + x2, train = mod_train, test = test,
  prior = 'dynamic', wlPrior = 10, plot = TRUE)
table(nb4$group, nb4$pr)
mean(nb4$group == nb4$pr)  # still 94% correct
```

`naiveBayes_train` *Train a naive Bayes classifier*

Description

Returns conditional means and standard deviations per class as well as a table with the global proportions of each class in the dataset. This is mostly useful because the output can be passed on to `naiveBayes` to save time if naiveBayes() is called in a loop with the same training dataset.

Usage

```
naiveBayes_train(formula, data)
```

Arguments

`formula`	outcome ~ predictor1 + predictor1 + ...
`data`	training dataset

noiseRemoval *Noise removal*

Description

Removes noise by spectral substraction. If a recording is affected by a steady noise with a relatively stable amplitude and spectrum (e.g., microphone hiss, crickets, MRI buzz, etc.), its spectrum can be simply subtracted from the signal. Algorithm: STFT to produce a spectrogram, divide by normalized noise spectrum, inverse STFT to reconstitute the signal. Most of the work is done by addFormants.

Usage

```
noiseRemoval(
  x,
  samplingRate = NULL,
  scale = NULL,
  noise,
  dB = 6,
  specificity = 1,
  windowLength = 50,
  step = windowLength/2,
  dynamicRange = 120,
  normalize = c("max", "orig", "none")[2],
  reportEvery = NULL,
  cores = 1,
  play = FALSE,
  saveAudio = NULL,
  plot = FALSE,
  savePlots = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
noise	a numeric vector of length two specifying the location of pure noise in input audio (in s); a matrix representing pure noise as a spectrum with frequency bins

	in rows; any input accepted by spectrogram if pure noise is found in a separate recording (eg path to file, numeric vector, etc.)
dB	if NULL (default), the spectral envelope is applied on the original scale; otherwise, it is set to range from 1 to 10 ^ (dB / 20)
specificity	a way to sharpen or blur the noise spectrum (we take noise spectrum ^ specificity) : 1 = no change, >1 = sharper (the loudest noise frequencies are preferentially removed), <1 = blurred (even quiet noise frequencies are removed)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
normalize	if TRUE, scales input prior to FFT
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
saveAudio	path + filename for saving the output, e.g. '~/Downloads/temp.wav'. If NULL = doesn't save
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters

Value

Returns the denoised audio

See Also

addFormants

Examples

```
s = soundgen(noise = list(time = c(-100, 400), value = -20),
  formantsNoise = list(f1 = list(freq = 3000, width = 25)),
  addSilence = 50, temperature = .001, plot = TRUE)
# Option 1: use part of the recording as noise profile
```

```
s1 = noiseRemoval(s, samplingRate = 16000, noise = c(0.05, 0.15),
  dB = 40, plot = TRUE)

# Option 2: use a separate recording as noise profile
noise = soundgen(pitch = NA, noise = 0,
  formantsNoise = list(f1 = list(freq = 3000, width = 25)))
spectrogram(noise, 16000)
s2 = noiseRemoval(s, samplingRate = 16000, noise = noise,
  dB = 40, plot = TRUE)

# Option 3: provide noise spectrum as a matrix
spec_noise = spectrogram(
      noise, samplingRate = 16000,
      output = 'original', plot = FALSE)
s3 = noiseRemoval(s, samplingRate = 16000, noise = spec_noise,
  dB = 40, plot = TRUE)

## Not run:
# play with gain and sensitivity
noiseRemoval(s, samplingRate = 16000, noise = c(0.05, 0.15),
  dB = 60, specificity = 2, plot = TRUE)

# remove noise only from a section of the audio
noiseRemoval(s, samplingRate = 16000, from = .3, to = .4,
  noise = c(0.05, 0.15), dB = 60, plot = TRUE, play = TRUE)

## End(Not run)
```

`nonlinPred` *Nonlinear prediction*

Description

Predicts new points in a time series. The functionality is provided by `nonLinearPrediction`. This function is just a simple wrapper "for dummies" that reconstructs the phase space under the hood, including the choice of time lag, embedding dimensions, etc. It can also predict not one but many points in a single step.

Usage

```
nonlinPred(
  x,
  nPoints = 1,
  time.lag = NULL,
  embedding.dim = NULL,
  max.embedding.dim = 15,
  threshold = 0.95,
  max.relative.change = 0.1,
  radius = NULL,
```

```
  radius.increment = NULL,
  plot = FALSE
)
```

Arguments

x	numeric vector
nPoints	number of points to predict, ideally not more than length(x) / 2 (the function is called recursively to predict longer sequences, but don't expect miracles)
time.lag	time lag for constructing Takens vectors. Defaults to the time to the first exponential decay of mutual information. See timeLag
embedding.dim	the number of dimensions of the phase space. Defaults to an estimate based on estimateEmbeddingDim
max.embedding.dim, threshold, max.relative.change	parameters used to estimate the optimal number of embedding dimensions - see estimateEmbeddingDim
radius, radius.increment	the radius used for detecting neighbors in the phase space and its increment in case no neighbors are found - see nonLinearPrediction
plot	if TRUE, plots the original time series and the predictions

Value

Returns a numeric vector on the same scale as input x.

Examples

```
x = c(rep(1, 3), rep(0, 4), rep(1, 3), rep(0, 4), rep(1, 3), 0, 0)
nonlinPred(x, 5, plot = TRUE)

nonlinPred(sin(1:25), 22, plot = TRUE)

x = soundgen(sylLen = 50, addSilence = 0)[250:450]
nonlinPred(x, 100, plot = TRUE)

nonlinPred(c(rnorm(5), NA, rnorm(3)))
nonlinPred(1:4)
nonlinPred(1:6)

## Not run:
s1 = soundgen(sylLen = 500, pitch = rnorm(5, 200, 20),
              addSilence = 0, plot = TRUE)
playme(s1)
length(s1)
# we can predict output that is longer than the original time series by
# predicting a bit at a time and using the output as the new input
s2 = nonlinPred(s1, 16000)
spectrogram(c(s1, s2))
playme(c(s1, s2))
```

```
## 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.

Usage

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

Arguments

`myfolder`	full path to folder containing input audio 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 (multiple values in a vector produce a multi-resolution spectrogram)
`step`	you can override `overlap` by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
`overlap`	overlap between successive FFT frames, %
`killDC`	if TRUE, removed DC offset (see also `flatEnv`)
`windowDC`	the window for calculating DC offset, ms

saveAudio	full path to where the normalized files should be saved (defaults to 'myfolder/normalized')
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)

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 amplitude 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 sensitivity typical of human hearing. The following normalization procedure is similar to that for type = 'rms'.

See Also

getRMS analyze getLoudness

Examples

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

## End(Not run)
```

---

notesDict *Conversion table from Hz to musical notation*

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.

notesToHz *Convert notes to Hz*

Description

Converts to Hz from musical notation like A4 - note A of the fourth octave above C0 (16.35 Hz).

Usage

```
notesToHz(n, A4 = 440)
```

Arguments

n	vector or matrix of notes
A4	frequency of note A in the fourth octave (modern standard ISO 16 or concert pitch = 440 Hz)

See Also

HzToNotes HzToSemitones

Examples

```
notesToHz(c("A4", "D4", "A#2", "C0", "C-2"))

# Baroque tuning A415, half a semitone flat relative to concert pitch A440
notesToHz(c("A4", "D4", "A#2", "C0", "C-2"), A4 = 415)
```

optimizePars *Optimize parameters for acoustic analysis*

Description

This customized wrapper for optim attempts to optimize the parameters of segment or analyze 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 segment and analyze have been optimized for human non-linguistic vocalizations.

Usage

```
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),
  mygrid = NULL,
  verbose = TRUE
)
```

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 'segment' or 'analyze' (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 smoothOverlap, 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

```
## 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/temp260'  # 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 = 'segment', key = key,
  pars = c('shortestSyl', 'shortestPause'),
  fitnessPar = 'nBursts', otherPars = list(method = 'env'),
  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(segment, 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 = 'segment', key = segmentManual,
  pars = c('shortestSyl', 'shortestPause'),
  fitnessPar = 'nBursts', otherPars = list(method = 'env'),
  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!)
key = as.numeric(log(pitchManual))
```

```
res = optimizePars(
  myfolder = myfolder,
    myfun = 'analyze',
    key = key,  # log-scale better for pitch
    pars = c('windowLength', 'silence'),
    bounds = list(low = c(5, 0), high = c(200, .2)),
    fitnessPar = 'pitch_median',
    nIter = 2,
    otherPars = list(plot = FALSE, loudness = NULL, novelty = NULL,
                     roughness = NULL, nFormants = 0),
    fitnessFun = function(x) {
      1 - cor(log(x), key, use = 'pairwise.complete.obs') *
      (1 - mean(is.na(x) & is.finite(key))) # penalize failing to detect f0
})

## End(Not run)
```

---

osc *Oscillogram*

Description

Plots the oscillogram (waveform) of a sound on a linear or logarithmic scale (in dB). To get a dB scale, centers and normalizes the sound, then takes a logarithm of the positive part and a flipped negative part, which is analogous to "Waveform (dB)" view in Audacity. For more plotting options, check oscillo.

Usage

```
osc(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  dynamicRange = 80,
  dB = FALSE,
  returnWave = FALSE,
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = NULL,
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  ylim = NULL,
  bty = "n",
  midline = TRUE,
```

```
  maxPoints = 10000,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
dB	if TRUE, plots on a dB instead of linear scale
returnWave	if TRUE, returns a log-transformed waveform as a numeric vector
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	if TRUE, plots the oscillogram
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
main	plot title
xlab, ylab	axis labels
ylim	override default amplitude scale for non-centered sounds
bty	box type (see '?par')
midline	if TRUE, draws a line at 0 dB
maxPoints	the maximum number of points to plot (speeds up the plotting of long audio files, but beware of antialiasing)
width, height, units, res	graphical parameters for saving plots passed to png
...	Other graphical parameters passed on to 'plot()'

Value

If returnWave = TRUE, returns the input waveform on the original or dB scale: a vector with range from '-dynamicRange' to 'dynamicRange'.

Examples

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

# Oscillogram on a linear scale without bells and whistles, just base R
plot(sound, type = 'l')

# Oscillogram options with soundgen
osc(sound)             # linear
osc(sound, dB = TRUE)  # dB

# For numeric vectors, indicate samplingRate and scale (max amplitude)
osc(sound, samplingRate = 1000, scale = 100, dB = TRUE)

# Embellish and customize the plot
o = osc(sound, samplingRate = 1000, dB = TRUE, midline = FALSE,
        main = 'My waveform', col = 'blue', returnWave = TRUE)
abline(h = -80, col = 'orange', lty = 3)
o[1:10]  # the waveform in dB

## Not run:
# Wave object
data(sheep, package = 'seewave')
osc(sheep, dB = TRUE)

# Plot a section
osc(sheep, from = .5, to = 1.2)

# for long files, reduce the resolution to plot quickly (careful: if the
# resolution is too low, antialiasing may cause artifacts)
osc(sheep, dB = TRUE, maxPoints = 2500)
osc(sheep, samplingRate = 5000, maxPoints = 100)

# files several minutes long can be plotted in under a second
osc('~/Downloads/speechEx.wav', maxPoints = 20000)

# saves oscillograms of all audio files in a folder
osc('~/Downloads/temp2', savePlots = '')

## End(Not run)
```

---

permittedValues *Defaults and ranges for soundgen()*

Description

A dataset containing defaults and ranges of key variables for soundgen() and soundgen_app(). Adjust as needed.

Usage

```
permittedValues
```

Format

A matrix with 58 rows and 4 columns:

default default value

low lowest permitted value

high highest permitted value

step increment for adjustment ...

---

phasegram *Phasegram*

Description

Produces a phasegram of a sound or another time series, which is a collection of Poincare sections cut through phase portraits of consecutive frames. The x axis is time, just as in a spectrogram, the y axis is a slice through the phase portrait, and the color shows the density of trajectories at each point of the phase portrait.

Usage

```
phasegram(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  windowLength = 10,
  step = windowLength/2,
  timeLag = NULL,
  theilerWindow = NULL,
  nonlinStats = c("ed", "d2", "ml", "sur"),
  pars_ed = list(max.embedding.dim = 15),
 pars_d2 = list(min.embedding.dim = 2, min.radius = 0.001, n.points.radius = 20),
  pars_ml = list(min.embedding.dim = 2, radius = 0.001),
  pars_sur = list(FUN = nonlinearTseries::timeAsymmetry, K = 1),
  bw = 0.01,
  bins = 5/bw,
  reportEvery = NULL,
  cores = 1,
  rasterize = FALSE,
  plot = TRUE,
  savePlots = NULL,
  colorTheme = c("bw", "seewave", "heat.colors", "...")[1],
```

```
  col = NULL,
  xlab = "Time",
  ylab = "",
  main = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
windowLength	the length of each frame analyzed separately (ms)
step	time step between consecutive frames (ms)
timeLag	time lag between the original and time-shifted version of each frame that together represent the phase portrait (ms). Defaults to the number of steps beyond which the mutual information function reaches its minimum or, if that fails, the steps until mutual information experiences the first exponential decay - see timeLag
theilerWindow	time lag between two points that are considered locally independent and can be treated as neighbors in the reconstructed phase space. defaults to the first minimum or, if unavailable, the first zero of the autocorrelation function (or, failing that, to timeLag * 2)
nonlinStats	nonlinear statistics to report: "ed" = the optimal number of embedding dimensions, "d2" = correlation dimension D2, "ml" = maximum Lyapunov exponent, "sur" = the results of surrogate data testing for stochasticity. These are calculated using the functionality of the package nonlinearTseries, which is seriously slow, so the default is just to get the phasegram itself
pars_ed	a list of control parameters passed to estimateEmbeddingDim
pars_d2	a list of control parameters passed to corrDim
pars_ml	a list of control parameters passed to maxLyapunov
pars_sur	a list of control parameters passed to surrogateTest
bw	standard deviation of the smoothing kernel, as in density
bins	the number of bins along the Y axis after rasterizing (has no effect if rasterize = FALSE)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing

rasterize	if FALSE, only plots and returns Poincare sections on the original scale (most graphical parameters will then have no effect); if TRUE, rasterizes the phasegram matrix and plots it with more graphical parameters
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
colorTheme	black and white ('bw'), as in seewave package ('seewave'), matlab-type palette ('matlab'), or any palette from palette such as 'heat.colors', 'cm.colors', etc
col	actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)
xlab, ylab, main	graphical parameters passed to soundgen:::filled.contour.mod (if rasterize = TRUE) or plot (if rasterize = FALSE)
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters passed to soundgen:::filled.contour.mod (if rasterize = TRUE) or plot (if rasterize = FALSE)

Details

Algorithm: the input sound is normalized to [-1, 1] and divided into consecutive frames windowLength ms long without multiplying by any windowing function (unlike in STFT). For each frame, a phase portrait is obtained by time-shifting the frame by timeLag ms. A Poincare section is taken through the phase portrait (currently at a fixed angle, namely the default in poincareMap), giving the intersection points of trajectories with this bisecting line. The density of intersections is estimated with a smoothing kernel of bandwidth bw (as an alternative to using histogram bins). The density distributions per frame are stacked together into a phasegram (output: "orig"). The ranges of phase portraits depend on the amplitude of signal in each frame. The resulting phasegram can optionally be rasterized to smooth it for plotting (output: "rasterized").

Value

Returns a list of three components: "orig" = the full phasegram; "rasterized" = a rasterized version. For both, $time is the middle of each frame (ms), $x is the coordinate along a Poincare section (since the audio is normalized, the scale is [-1, 1]), and $y is the density of intersections of system trajectories with the Poincare section. The third component is $descriptives, which gives the result of nonlinear analysis per frame. Currently implemented: shannon = Shannon entropy of Poincare sections, nPeaks = log-number of peaks in the density distribution of Poincare sections, ml = maximum Lyapunov exponent (positive values suggest chaos), ed = optimal number of embedding dimensions (shows the complexity of the reconstructed attractor), d2 = correlation dimension, sur = probability of stochasticity according to surrogate data testing (0 = deterministic, 1 = stochastic).

References

- Herbst, C. T., Herzel, H., Švec, J. G., Wyman, M. T., & Fitch, W. T. (2013). Visualization of system dynamics using phasegrams. Journal of the Royal Society Interface, 10(85), 20130288.
- Huffaker, R., Huffaker, R. G., Bittelli, M., & Rosa, R. (2017). Nonlinear time series analysis with R. Oxford University Press.

Examples

```
target = soundgen(sylLen = 300, pitch = c(350, 420, 420, 410, 340) * 3,
  subDep = c(0, 0, 60, 50, 0, 0) / 2, addSilence = 0, plot = TRUE)
# Nonlinear statistics are also returned (slow - disable by setting
# nonlinStats = NULL if these are not needed)
ph = phasegram(target, 16000, nonlinStats = NULL)

## Not run:
ph = phasegram(target, 16000, windowLength = 20, step = 20,
  rasterize = TRUE, bw = .01, bins = 150)
ph$descriptives

# Unfortunately, phasegrams are greatly affected by noise. Compare:
target2 = soundgen(sylLen = 300, pitch = c(350, 420, 420, 410, 340) * 3,
  subDep = c(0, 0, 60, 50, 0, 0)/2, noise = -10, addSilence = 0, plot = TRUE)
ph2 = phasegram(target2, 16000)

s2 = soundgen(sylLen = 3000, addSilence = 0, temperature = 1e-6,
  pitch = c(380, 550, 500, 220), subDep = c(0, 0, 40, 0, 0, 0, 0, 0),
  amDep = c(0, 0, 0, 0, 80, 0, 0, 0), amFreq = 80,
  jitterDep = c(0, 0, 0, 0, 0, 3))
spectrogram(s2, 16000, yScale = 'bark')
phasegram(s2, 16000, windowLength = 10, nonlinStats = NULL, bw = .001)
phasegram(s2, 16000, windowLength = 10, nonlinStats = NULL, bw = .02)

## End(Not run)
```

---

pitchContour *Manually corrected pitch contours in 260 sounds*

Description

A dataframe of 260 rows and two columns: "file" for filename in the corpus (Anikin & Persson, 2017) and "pitch" for pitch values per frame. The corpus can be downloaded from http://cogsci.se/publications.html

Usage

```
pitchContour
```

Format

An object of class `data.frame` with 260 rows and 2 columns.

`pitchDescriptives` *Pitch descriptives*

Description

Provides common descriptives of time series such as pitch contours, including measures of average / range / variability / slope / inflections etc. Several degrees of smoothing can be applied consecutively. The summaries are produced on the original and log-transformed scales, so this is meant to be used on frequency-related variables in Hz.

Usage

```
pitchDescriptives(
  x,
  step = NULL,
  timeUnit,
  smoothBW = c(NA, 10, 1),
  inflThres = 0.2,
  extraSummaryFun = c(),
  ref = 16.35,
  plot = FALSE
)
```

Arguments

`x`	input: numeric vector, a list of time stamps and values in rows, a dataframe with one row per file and time/pitch values stored as characters (as exported by `pitch_app`), or path to csv file containing the output of `pitch_app` or `analyze`
`step`	distance between values in s (only needed if input is a vector)
`timeUnit`	specify whether the time stamps (if any) are in ms or s
`smoothBW`	a vector of bandwidths (Hz) for consecutive smoothing of input using `pitchSmoothPraat`; NA = no smoothing
`inflThres`	minimum difference (in semitones) between consecutive extrema to consider them inflections; to apply a different threshold at each smoothing level, provide `inflThres` as a vector of the same length as `smoothBW`; NA = no threshold
`extraSummaryFun`	additional summary function(s) that take a numeric vector with some NAs and return a single number, eg c('myFun1', 'myFun2')
`ref`	reference value for transforming Hz to semitones, defaults to C0 (16.35 Hz)
`plot`	if TRUE, plots the inflections for manual verification

Value

Returns a dataframe with columns containing summaries of one or multiple inputs (one input per row). The descriptives are as follows:

duration total duration, s

durDefined duration after omitting leading and trailing NAs

propDefined proportion of input with non-NA value, eg proportion of voiced frames if the input is pitch

start, start_oct, end, end_oct the first and last values on the original scale and in octaves above C0 (16.3516 Hz)

mean, median, max, min average and extreme values on the original scale

mean_oct, median_oct, min_oct, max_oct same in octaves above C0

time_max, time_min the location of minimum and maximum relative to durDefined, 0 to 1

range, range_sem, sd, sd_sem range and standard deviation on the original scale and in semitones

CV coefficient of variation = sd/mean (provided for historical reasons)

meanSlope, meanSlope_sem mean slope in Hz/s or semitones/s (NB: does not depend on duration or missing values)

meanAbsSlope, meanAbsSlope_sem mean absolute slope (modulus, ie rising and falling sections no longer cancel out)

maxAbsSlope, maxAbsSlope_sem the steepest slope

Examples

```
x = c(NA, NA, 405, 441, 459, 459, 460, 462, 462, 458, 458, 445, 458, 451,
444, 444, 430, 416, 409, 403, 403, 389, 375, NA, NA, NA, NA, NA, NA, NA, NA,
NA, 183, 677, 677, 846, 883, 886, 924, 938, 883, 946, 846, 911, 826, 826,
788, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 307,
307, 368, 377, 383, 383, 383, 380, 377, 377, 377, 374, 374, 375, 375, 375,
375, 368, 371, 374, 375, 361, 375, 389, 375, 375, 375, 375, 375, 314, 169,
NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 238, 285, 361, 374, 375, 375,
375, 375, 375, 389, 403, 389, 389, 375, 375, 389, 375, 348, 361, 375, 348,
348, 361, 348, 342, 361, 361, 361, 365, 365, 361, 966, 966, 966, 959, 959,
946, 1021, 1021, 1026, 1086, 1131, 1131, 1146, 1130, 1172, 1240, 1172, 1117,
1103, 1026, 1026, 966, 919, 946, 882, 832, NA, NA, NA, NA, NA, NA, NA, NA,
NA, NA)
plot(x, type = 'b')
ci95 = function(x) diff(quantile(na.omit(x), probs = c(.025, .975)))
pd = pitchDescriptives(
  x, step = .025, timeUnit = 's',
  smoothBW = c(NA, 10, 1),   # original + smoothed at 10 Hz and 1 Hz
  inflThres = c(NA, .2, .2), # different for each level of smoothing
  extraSummaryFun = 'ci95',  # user-defined, here 95% coverage interval
  plot = TRUE
)
pd

## Not run:
# a single file
data(sheep, package = 'seewave')
a = analyze(sheep)
pd1 = pitchDescriptives(a$detailed[, c('time', 'pitch')],
                        timeUnit = 'ms', inflThres = NA, plot = TRUE)
```

```
pd2 = pitchDescriptives(a$detailed[, c('time', 'pitch')],
                        timeUnit = 'ms', inflThres = c(0.1, 0.1, .5), plot = TRUE)

# multiple files returned by analyze()
an = analyze('~/Downloads/temp')
pd = pitchDescriptives(an$detailed, timeUnit = 'ms')
pd

# multiple files returned by pitch_app()
pd = pitchDescriptives(
  '~/Downloads/pitch_manual_1708.csv',
  timeUnit = 'ms', smoothBW = c(NA, 2), inflThres = .25)

# a single file, exported from Praat
par(mfrow = c(3, 1))
pd = pitchDescriptives(
  '~/Downloads/F-Hin-Om_jana.wav_F0contour.txt',
  timeUnit = 's', smoothBW = c(NA, 25, 2), inflThres = .25, plot = TRUE)
par(mfrow = c(1, 1))

## End(Not run)
```

`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.

`pitchSmoothPraat` *Pitch smoothing as in Praat*

Description

Smoothes an intonation (pitch) contour with a low-pass filter, as in Praat (http://www.fon.hum.uva.nl/praat/). Algorithm: interpolates missing values (unvoiced frames), performs FFT to obtain the spectrum, multiplies by a Gaussian filter, performs an inverse FFT, and fills the missing values back in. The `bandwidth` parameter is about half the cutoff frequency (ie some frequencies will still be present up to ~2 * bandwidth)

Usage

```
pitchSmoothPraat(pitch, bandwidth, samplingRate, plot = FALSE)
```

Arguments

pitch	numeric vector of pitch values (NA = unvoiced)
bandwidth	the bandwidth of low-pass filter, Hz (high = less smoothing, close to zero = more smoothing)
samplingRate	the number of pitch values per second
plot	if TRUE, plots the original and smoothed pitch contours

See Also

analyze

Examples

```
pitch = c(NA, NA, 405, 441, 459, 459, 460, 462, 462, 458, 458, 445, 458, 451,
444, 444, 430, 416, 409, 403, 403, 389, 375, NA, NA, NA, NA, NA, NA, NA, NA,
NA, 183, 677, 677, 846, 883, 886, 924, 938, 883, 946, 846, 911, 826, 826,
788, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 307,
307, 368, 377, 383, 383, 383, 380, 377, 377, 377, 374, 374, 375, 375, 375,
375, 368, 371, 374, 375, 361, 375, 389, 375, 375, 375, 375, 375, 314, 169,
NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 238, 285, 361, 374, 375, 375,
375, 375, 375, 389, 403, 389, 389, 375, 375, 389, 375, 348, 361, 375, 348,
348, 361, 348, 342, 361, 361, 361, 365, 365, 361, 966, 966, 966, 959, 959,
946, 1021, 1021, 1026, 1086, 1131, 1131, 1146, 1130, 1172, 1240, 1172, 1117,
1103, 1026, 1026, 966, 919, 946, 882, 832, NA, NA, NA, NA, NA, NA, NA, NA,
NA, NA)
pitchSmoothPraat(pitch, bandwidth = 10, samplingRate = 40, plot = TRUE)
pitchSmoothPraat(pitch, bandwidth = 2, samplingRate = 40, plot = TRUE)
```

---

pitch_app *Interactive pitch tracker*

Description

Starts a shiny app for manually editing pitch contours. The settings in the panels on the left correspond to arguments to analyze - see '?analyze' and the vignette on acoustic analysis for help and examples. You can verify the pitch contours first, and then feed them back into analyze (see examples).

Usage

```
pitch_app()
```

Value

When proceeding to the next file in the cue, two types of backups are created. (1) A global object called "my_pitch" is created or updated. This list becomes visible when the app is terminated, and it contains the usual outputs of analyze() ($detailed and $summary) plus lists of manually corrected voiced and unvoiced frames. (2) The app saves to disk a .csv file with one row per audio file. Apart from the usual descriptives from analyze(), there are two additional columns: "time" with time stamps (the midpoint of each STFT frame, ms) and "pitch" with the manually corrected pitch values for each frame (Hz). When the orange "Download results" button is clicked, a context menu pops up offering to terminate the app - if that happens, the results are also returned directly into R. To process pitch contours further in R, work directly with my_pitch[[myfile]]$time and my_pitch[[myfile]]$pitch or, if loading the csv file, do something like:

```
a = read.csv('~/Downloads/output.csv', stringsAsFactors = FALSE)
pitch = as.numeric(unlist(strsplit(a$pitch, ',')))
mean(pitch, na.rm = TRUE); sd(pitch, na.rm = TRUE)
```

Suggested workflow

Start by setting the basic analysis settings such as pitchFloor, pitchCeiling, silence, etc. Then click "Load audio" to upload one or several audio files (wav/mp3). Long files will be very slow, so please cut your audio into manageable chunks (ideally <10 s). If Shiny complains that maximum upload size is exceeded, you can increase it, say to 30 MB, with ‘options(shiny.maxRequestSize = 30 * 1024^2)‘. Once the audio has been uploaded to the browser, fine-tune the analysis settings as needed, edit the pitch contour in the first file to your satisfaction, then click "Next" to proceed to the next file, etc. Remember that setting a reasonable prior is often faster than adjusting the contour one anchor at a time. When done, click "Save results". If working with many files, you might want to save the results occasionally in case the app crashes (although you should still be able to recover your data if it does - see below).

How to edit pitch contours

Left-click to add a new anchor, double-click to remove it or unvoice the frame. Each time you make a change, the entire pitch contour is re-fit, so making a change in one frame can affect the path through candidates in adjacent frames. You can control this behavior by changing the settings in Out/Path and Out/Smoothing. If correctly configured, the app corrects the contour with only a few manual values - you shouldn’t need to manually edit every single frame. For longer files, you can zoom in/out and navigate within the file. You can also select a region to voice/unvoice or shift it as a whole or to set a prior based on selected frequency range.

Recovering lost data

Every time you click "next" or "last" to move in between files in the queue, the output you’ve got so far is saved in a backup file called "temp.csv", and the "my_pitch" global object is updated. If the app crashes or is closed without saving the results, this backup file preserves your data. To recover it, access this file manually on disk or simply restart pitch_app() - a dialog box will pop up and ask whether you wank to append the old data to the new one. Path to backup file: "[R_installation_folder]/soundgen/shiny/pitch_app/www/temp.csv", for example, "/home/allgoodguys/R/x86_64-pc-linux-gnu-library/3.6/soundgen/shiny/pitch_app/www/temp.csv"

See Also

formant_app

Examples

```
## Not run:
# Recommended workflow for analyzing a lot of short audio files
path_to_audio = '~/Downloads/temp'  # our audio lives here

# STEP 1: extract manually corrected pitch contours
my_pitch = pitch_app()  # runs in default browser such as Firefox or Chrome
# To change system default browser, run something like:
options('browser' = '/usr/bin/firefox')  # path to the executable on Linux

# STEP 2: run analyze() with manually corrected pitch contours to obtain
# accurate descriptives like the proportion of energy in harmonics above f0,
# etc. This also gives you formants and loudness estimates (disabled in
# pitch_app to speed things up)
df2 = analyze(path_to_audio,
  pitchMethods = 'autocor',  # only needed for HNR
  nFormants = 5,        # now we can measure formants as well
  pitchManual = my_pitch
  # or, if loading the output of pitch_app() from the disk:
  # pitchManual = '~/Downloads/output.csv'
  # pitchManual = '~/path_to_some_folder/my_pitch_contours.rds
  # etc
)

# STEP 3: add other acoustic descriptors, for ex.
df3 = segment(path_to_audio)

# STEP 4: merge df2, df3, df4, ... in R or a spreadsheet editor to have all
# acoustic descriptives together

# To verify your pitch contours and/or edit them later, copy output.csv to
# the folder with your audio, run pitch_app(), and load the audio + csv
# together. The saved pitch contours are treated as manual anchors

## End(Not run)
```

---

playme *Play audio*

Description

Plays one or more sounds: wav/mp3 file(s), Wave objects, or numeric vectors. This is a simple wrapper for the functionality provided by play. Recommended players on Linux: "play" from the "vox" library (default), "aplay".

Usage

```
playme(x, samplingRate = 16000, player = NULL, from = NULL, to = NULL)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
player	the name of player to use, eg "aplay", "play", "vlc", etc. Defaults to "play" on Linux, "afplay" on MacOS, and tuneR default on Windows. In case of errors, try setting another default player for play
from, to	play a selected time range (s)

Examples

```
## Not run:
# Play an audio file:
playme('pathToMyAudio/audio.wav')

# Create and play a numeric vector:
f0_Hz = 440
sound = sin(2 * pi * f0_Hz * (1:16000) / 16000)
playme(sound, 16000)
playme(sound, 16000, from = .1, to = .5)  # play from 100 to 500 ms

# In case of errors, look into tuneR::play(). For ex., you might need to
# specify which player to use:
playme(sound, 16000, player = 'aplay')

# To avoid doing it all the time, set the default player:
tuneR::setWavPlayer('aplay')
playme(sound, 16000)  # should now work without specifying the player

## End(Not run)
```

---

plotMS *Plot modulation spectrum*

Description

Plots a single modulation spectrum returned by modulationSpectrum. The result is the same as the plot produced by modulationSpectrum, but calling plotMS is handy for processed modulation spectra - for instance, for plotting the difference between the modulation spectra of two sounds or groups of sounds.

Usage

```
plotMS(
  ms,
  X = as.numeric(colnames(ms)),
  Y = as.numeric(rownames(ms)),
```

```
  quantiles = c(0.5, 0.8, 0.9),
  colorTheme = c("bw", "seewave", "heat.colors", "...")[1],
  col = NULL,
  logWarpX = NULL,
  logWarpY = NULL,
  main = NULL,
  xlab = "Hz",
  ylab = "1/kHz",
  xlim = NULL,
  ylim = NULL,
  audio = NULL,
  extraY = TRUE,
  ...
)
```

Arguments

ms	modulation spectrum - a matrix with temporal modulation in columns and spectral modulation in rows, as returned by modulationSpectrum
X, Y	rownames and colnames of ms, respectively
quantiles	labeled contour values, % (e.g., "50" marks regions that contain 50% of the sum total of the entire modulation spectrum)
colorTheme	black and white ('bw'), as in seewave package ('seewave'), matlab-type palette ('matlab'), or any palette from palette such as 'heat.colors', 'cm.colors', etc
col	actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)
logWarpX, logWarpY	numeric vector of length 2: c(sigma, base) of pseudolog-warping the modulation spectrum, as in function pseudo_log_trans() from the "scales" package
xlab, ylab, main, xlim, ylim	graphical parameters
audio	(internal) a list of audio attributes
extraY	if TRUE, another Y-axis is plotted on the right showing 1000/Y
...	other graphical parameters passed on to filled.contour.mod and contour (see spectrogram)

Examples

```
ms1 = modulationSpectrum(runif(4000), samplingRate = 16000, plot = TRUE)
plotMS(ms1$processed)  # identical to above

# compare two modulation spectra
ms2 = modulationSpectrum(soundgen(sylLen = 100, addSilence = 0),
                         samplingRate = 16000)
# ensure the two matrices have the same dimensions
ms2_resized = soundgen:::interpolMatrix(ms2$original,
  nr = nrow(ms1$original), nc = ncol(ms1$original))
```

```
# plot the difference
plotMS(log(ms1$original / ms2_resized), quantile = NULL,
  col = colorRampPalette(c('blue', 'yellow')) (50))
```

---

presets *Presets*

Description

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

Usage

```
presets
```

Format

A list of length 4.

---

prosody *Prosody*

Description

Exaggerates or flattens the intonation by performing a dynamic pitch shift, changing pitch excursion from its original median value without changing the formants. This is a particular case of pitch shifting, which is performed with `shiftPitch`. The result is likely to be improved if manually corrected pitch contours are provided. Depending on the nature of audio, the settings that control pitch shifting may also need to be fine-tuned with the `shiftPitch_pars` argument.

Usage

```
prosody(
  x,
  samplingRate = NULL,
  multProsody,
  analyze_pars = list(),
  shiftPitch_pars = list(),
  pitchManual = NULL,
  play = FALSE,
  saveAudio = NULL,
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = NULL,
  width = 900,
```

```
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
multProsody	multiplier of pitch excursion from median (on a logarithmic or musical scale): >1 = exaggerate intonation, 1 = no change, <1 = flatten, 0 = completely flat at the original median pitch
analyze_pars	a list of parameters to pass to analyze (only needed if pitchManual is NULL - that is, if we attempt to track pitch automatically)
shiftPitch_pars	a list of parameters to pass to shiftPitch to fine-tune the pitch-shifting algorithm
pitchManual	manually corrected pitch contour. For a single sound, provide a numeric vector of any length. For multiple sounds, provide a dataframe with columns "file" and "pitch" (or path to a csv file) as returned by pitch_app, ideally with the same windowLength and step as in current call to analyze. A named list with pitch vectors per file is also OK (eg as returned by pitch_app)
play	if TRUE, plays the processed audio
saveAudio	full (!) path to folder for saving the processed audio; NULL = don't save, '' = same as input folder (NB: overwrites the originals!)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters

Value

If the input is a single audio (file, Wave, or numeric vector), returns the processed waveform as a numeric vector with the original sampling rate and scale. If the input is a folder with several audio files, returns a list of processed waveforms, one for each file.

See Also

shiftPitch

Examples

```
s = soundgen(sylLen = 200, pitch = c(150, 220), addSilence = 50,
             plot = TRUE, yScale = 'log')
# playme(s)
s1 = prosody(s, 16000, multProsody = 2,
  analyze_pars = list(windowLength = 30, step = 15),
  shiftPitch_pars = list(windowLength = 20, step = 5, freqWindow = 300),
  plot = TRUE)
# playme(s1)
# spectrogram(s1, 16000, yScale = 'log')

## Not run:
# Flat intonation - remove all frequency modulation
s2 = prosody(s, 16000, multProsody = 0,
  analyze_pars = list(windowLength = 30, step = 15),
  shiftPitch_pars = list(windowLength = 20, step = 1, freqWindow = 500),
  plot = TRUE)
playme(s2)
spectrogram(s2, 16000, yScale = 'log')

# Download an example - a bit of speech (sampled at 16000 Hz)
download.file('http://cogsci.se/soundgen/audio/speechEx.wav',
              destfile = '~/Downloads/temp1/speechEx.wav')
target = '~/Downloads/temp1/speechEx.wav'
samplingRate = tuneR::readWave(target)@samp.rate
spectrogram(target, yScale = 'log')
playme(target)

s3 = prosody(target, multProsody = 1.5,
  analyze_pars = list(windowLength = 30, step = 15),
  shiftPitch_pars = list(freqWindow = 400, propagation = 'adaptive'))
spectrogram(s3, tuneR::readWave(target)@samp.rate, yScale = 'log')
playme(s3)

# process all audio files in a folder
s4 = prosody('~/Downloads/temp', multProsody = 2, savePlots = '',
             saveAudio = '~/Downloads/temp/prosody')
str(s4)  # returns a list with audio (+ saves it to disk)

## End(Not run)
```

---

reportTime *Report time*

Description

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

Usage

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

Arguments

i	current iteration
time_start	time when the loop started running
nIter	total number of iterations
reportEvery	report progress every n iterations
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
prefix	a string to print before "Done...", eg "Chain 1: "

Examples

```
time_start = proc.time()
nIter = 100
for (i in 1:nIter) {
  Sys.sleep(i ^ 1.02 / 10000)
  reportTime(i, time_start, nIter,
    jobs = (1:100) ^ 1.02, prefix = 'Chain 1: ')
}
## Not run:
# Unknown number of iterations:
time_start = proc.time()
for (i in 1:20) {
  Sys.sleep(i ^ 2 / 10000)
  reportTime(i = i, time_start = time_start,
  jobs = (1:20) ^ 2, reportEvery = 5)
}

# 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 seq_along(filenames)) {
  # ...do what you have to do with each file...
  reportTime(i = i, time_start = time_start, nIter = length(filenames),
             jobs = filesizes)
```

```
}

## End(Not run)
```

`resample` *Resample a vector*

Description

Changes the sampling rate without introducing artefacts like aliasing. Best for relatively short vectors that require special care (eg pitch contours that contain NAs, which need to be dropped or preserved) as the algorithm is too slow for long sounds. Algorithm: to downsample, applies a low-pass filter, then decimates with `approx`; to upsample, performs linear interpolation with `approx`, then applies a low-pass filter. NAs can be interpolated or preserved in the output. The length of output is determined, in order of precedence, by `len` / `mult` / `samplingRate_new`. For simple vector operations, this is very similar to approx, but the leading and trailing NAs are also preserved (see examples).

Usage

```
resample(
  x,
  samplingRate = NULL,
  samplingRate_new = NULL,
  mult = NULL,
  len = NULL,
  lowPass = TRUE,
  na.rm = FALSE,
  reportEvery = NULL,
  cores = 1,
  saveAudio = NULL,
  plot = FALSE,
  savePlots = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

`x`	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
`samplingRate`	sampling rate of `x` (only needed if `x` is a numeric vector)
`samplingRate_new`	an alternative to `mult` provided that the old `samplingRate` is know (NB: `mult` takes precedence)

mult	multiplier of sampling rate: new sampling rate = old sampling rate x mult, so 1 = no effect, >1 = upsample, <1 = downsample
len	if specified, overrides mult and samplingRate_new and simply returns a vector of length len
lowPass	if TRUE, applies a low-pass filter before decimating or after upsampling to avoid aliasing
na.rm	if TRUE, NAs are interpolated, otherwise they are preserved in the output
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
saveAudio	full path to the folder in which to save audio files (one per detected syllable)
plot	should a spectrogram be plotted? TRUE / FALSE
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters

Examples

```
## Example 1: a short vector with NAs
x = c(NA, 1, 2, 3, NA, NA, 6, 7, 8, NA)

# upsample
resample(x, mult = 3.5, lowPass = FALSE, plot = TRUE)  # just approx
resample(x, mult = 3.5, lowPass = TRUE, plot = TRUE) # low-pass + approx
resample(x, mult = 3.5, lowPass = FALSE, na.rm = TRUE, plot = TRUE)

# downsample
resample(x, mult = 0.5, lowPass = TRUE, plot = TRUE)
resample(x, mult = 0.5, na.rm = TRUE, plot = TRUE)
resample(x, len = 5, na.rm = TRUE, plot = TRUE) # same

# The most important TIP: use resample() for audio files and the internal
# soundgen:::.resample(list(sound = ...)) for simple vector operations because
# it's >1000 times faster. For example:
soundgen:::.resample(list(sound = x), mult = 3.5, lowPass = FALSE)

## Example 2: a sound
silence = rep(0, 10)
samplingRate = 1000
fr = seq(100, 300, length.out = 400)
x = c(silence, sin(cumsum(fr) * 2 * pi / samplingRate), silence)
spectrogram(x, samplingRate)

# downsample
x1 = resample(x, mult = 1 / 2.5)
spectrogram(x1, samplingRate / 2.5)  # no aliasing
```

```
# cf:
x1bad = resample(x, mult = 1 / 2.5, lowPass = FALSE)
spectrogram(x1bad, samplingRate / 2.5)  # aliasing

# upsample
x2 = resample(x, mult = 3)
spectrogram(x2, samplingRate * 3)  # nothing above the old Nyquist
# cf:
x2bad = resample(x, mult = 3, lowPass = FALSE)
spectrogram(x2bad, samplingRate * 3)  # high-frequency artefacts

## Not run:
# Example 3: resample all audio files in a folder to 8000 Hz
resample('~/Downloads/temp', saveAudio = '~/Downloads/temp/sr8000/',
         samplingRate_new = 8000, savePlots = '~/Downloads/temp/sr8000/')

## End(Not run)
```

---

reverb *Reverb & echo*

Description

Adds reverberation and/or echo to a sound. Algorithm for reverb: adds time-shifted copies of the signal weighted by a decay function, which is analogous to convoluting the input with a parametric model of some hypothetical impulse response function. In simple terms: we specify how much and when the sound rebounds back (as from a wall) and add these time-shifted copies to the original, optionally with some spectral filtering.

Usage

```
reverb(
  x,
  samplingRate = NULL,
  echoDelay = 200,
  echoLevel = -20,
  reverbDelay = 70,
  reverbSpread = 130,
  reverbLevel = -25,
  reverbDensity = 50,
  reverbType = "gaussian",
  filter = list(),
  dynamicRange = 80,
  output = c("audio", "detailed")[1],
  play = FALSE,
  reportEvery = NULL,
  cores = 1,
  saveAudio = NULL
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
echoDelay	the delay at which the echo appears, ms
echoLevel	the rate at which the echo weakens at each repetition, dB
reverbDelay	the time of maximum reverb density, ms
reverbSpread	standard deviation of reverb spread around time reverbDelay, ms
reverbLevel	the maximum amplitude of reverb, dB below input
reverbDensity	the number of echos or "voices" added
reverbType	so far only "gaussian" has been implemented
filter	(optional) a spectral filter to apply to the created reverb and echo (see addFormants for acceptable formats)
dynamicRange	the precision with which the reverb and echo are calculated, dB
output	"audio" = returns just the processed audio, "detailed" = returns a list with reverb window, the added reverb/echo, etc.
play	if TRUE, plays the processed audio
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
saveAudio	full (!) path to folder for saving the processed audio; NULL = don't save, '' = same as input folder (NB: overwrites the originals!)

Examples

```
s = soundgen()
s_rev = reverb(s, 16000)
# playme(s_rev)

## Not run:
# double echo, no reverb
s1 = reverb(s, samplingRate = 16000, reverbLevel = NULL,
            echoDelay = c(250, 800), echoLevel = c(-15, -25))
# playme(s1)
# spectrogram(s1, 16000, osc = TRUE, ylim = c(0, 4))

# only reverb (indoors)
s2 = reverb(s, samplingRate = 16000, echoDelay = NULL,
            reverbDelay = 70, reverbSpread = 130,
            reverbLevel = -20, reverbDensity = 20)
# playme(s2)
# spectrogram(s2, 16000, osc = TRUE, ylim = c(0, 4))

# reverb (caves)
s3 = reverb(s, samplingRate = 16000, echoDelay = NULL,
```

```
            reverbDelay = 600, reverbSpread = 1500,
            reverbLevel = -10, reverbDensity = 100)
# playme(s3)
# spectrogram(s3, 16000, osc = TRUE, ylim = c(0, 4))

# both echo and reverb with high frequencies emphasized
s4 = reverb(s, samplingRate = 16000,
            echoDelay = 250, echoLevel = -20,
            reverbDelay = 70, reverbSpread = 120,
            reverbLevel = -25, reverbDensity = 50,
            filter = list(formants = NULL, lipRad = 3))
# playme(s4)
# spectrogram(s4, 16000, osc = TRUE, ylim = c(0, 4))

# add reverb to a recording
s5 = reverb('~/Downloads/temp260/ut_fear_57-m-tone.wav',
            echoDelay = 850, echoLevel = -40)
# playme(s5, 44100)

# add reverb to all files in a folder, save the result
reverb('~/Downloads/temp2', saveAudio = '~/Downloads/temp2/rvb')

## 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. This is useful for speaker normalization because absolute formant frequencies measured in Hz depend strongly on overall vocal tract length (VTL). For example, adult men vs. children or grizzly bears vs. dog puppies have very different formant spaces in Hz, but it is possible to define a VTL-normalized formant space that is applicable to all species and sizes. 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,
  interceptZero = TRUE,
  tube = c("closed-open", "open-open")[1],
  speedSound = 35400,
  plot = FALSE
)
```

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)
interceptZero	if TRUE, forces the regression curve to pass through the origin. This reduces the influence of highly variable lower formants, but we have to commit to a particular model of the vocal tract: closed-open or open-open/closed-closed (method = "regression" only)
tube	the vocal tract is assumed to be a cylindrical tube that is either "closed-open" or "open-open" (same as closed-closed)
speedSound	speed of sound in warm air, cm/s. Stevens (2000) "Acoustic phonetics", p. 138
plot	if TRUE, plots vowel quality in speaker-normalized F1-F2 space

Details

Algorithm: the expected formant dispersion is given by $(2*formant_number-1)*speedSound/(4*formant_frequency)$ for a closed-open tube (mouth open) and $formant_number*speedSound/(2*formant_frequency)$ for an open-open or closed-closed tube. F1 is schwa 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 (note that `schwa` calls `estimateVTL` with the option `method = 'regression'`).

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 F1 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. Like `ff_relative`, this metric is invariant to vocal tract length, but the variance tends to be greater for lower vs. higher formants

ff_relative_dF deviation of formant frequencies from those expected for a schwa, proportion of formant spacing (dF). Unlike `ff_relative` and `ff_relative_semitones`, this metric has similar variance for lower and higher formants

See Also

`estimateVTL`

Examples

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

## CASE 2: known (observed) formant frequencies
# Let's take formant frequencies in four vocalizations, namely
# (/a/, /i/, /mmm/, /roar/) by the same male speaker:
formants_a = c(860, 1430, 2900, NA, 5200)  # NAs are OK - here F4 is unknown
s_a = schwa(formants = formants_a, plot = TRUE)
s_a
# We get an estimate of VTL (s_a$vtl_apparent),
#   same as with estimateVTL(formants_a)
# We also get theoretical schwa formants: s_a$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 (see plot)

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

formants_mmm = c(1200, 2000, 2800, 3800, 5400, 6400)
schwa(formants_mmm, tube = 'closed-closed', plot = TRUE)
# ~schwa, but with a closed mouth

formants_roar = c(550, 1000, 1460, 2280, 3350,
                  4300, 4900, 5800, 6900, 7900)
s_roar = schwa(formants = formants_roar, plot = TRUE)
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], plot = TRUE)
# 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, plot = TRUE)
# 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, plot = TRUE)
# 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 separated by background noise in long recordings (up to 1-2 hours of audio per file). Syllables are defined as continuous segments that seem to be different from noise based on amplitude and/or spectral similarity thresholds. Bursts are defined as local maxima in signal envelope that are high enough both in absolute terms (relative to the global maximum) and with respect to the surrounding region (relative to local minima). See vignette('acoustic_analysis', package = 'soundgen') for details.

Usage

```
segment(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  shortestSyl = 40,
  shortestPause = 40,
  method = c("env", "spec", "mel")[3],
  propNoise = NULL,
  SNR = NULL,
  noiseLevelStabWeight = c(1, 0.25),
```

```
  windowLength = 40,
  step = NULL,
  overlap = 80,
  reverbPars = list(reverbDelay = 70, reverbSpread = 130, reverbLevel = -35,
    reverbDensity = 50),
  interburst = NULL,
  peakToTrough = SNR + 3,
  troughLocation = c("left", "right", "both", "either")[4],
  summaryFun = c("median", "sd"),
  maxDur = 30,
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = NULL,
  saveAudio = NULL,
  addSilence = 50,
  main = NULL,
  xlab = "",
  ylab = "Signal, dB",
  showLegend = FALSE,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  maxPoints = c(1e+05, 5e+05),
  specPlot = list(colorTheme = "bw"),
  contourPlot = list(lty = 1, lwd = 2, col = "green"),
  sylPlot = list(lty = 1, lwd = 2, col = "blue"),
  burstPlot = list(pch = 8, cex = 3, col = "red"),
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
shortestSyl	minimum acceptable length of syllables, ms
shortestPause	minimum acceptable break between syllables, ms (syllables separated by shorter pauses are merged)
method	the signal used to search for syllables: 'env' = Hilbert-transformed amplitude envelope, 'spec' = spectrogram, 'mel' = mel-transformed spectrogram (see tuneR::melfcc)
propNoise	the proportion of non-zero sound assumed to represent background noise, 0 to 1 (note that complete silence is not considered, so padding with silence won't

affect the algorithm). Set to 0 to skip correcting SNR by background noise level

`SNR` expected signal-to-noise ratio (dB above noise), which determines the threshold for syllable detection. The meaning of "dB" here is approximate since the "signal" may be different from sound intensity, depending on the `method`

`noiseLevelStabWeight`
a vector of length 2 specifying the relative weights of the overall signal level vs. stability when attempting to automatically locate the regions that represent noise. Increasing the weight of stability tends to accentuate the beginning and end of each syllable.

`windowLength` length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)

`step` you can override `overlap` by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)

`overlap` overlap between successive FFT frames, %

`reverbPars` parameters passed on to `reverb` to attempt to cancel the effects of reverberation or echo, which otherwise tend to merge short and loud segments like rapid barks

`interburst` minimum time between two consecutive bursts (ms). Defaults to the average detected `(syllable + pause) / 2`

`peakToTrough` to qualify as a burst, a local maximum has to be at least `peakToTrough` dB above the left and/or right local trough(s) (controlled by `troughLocation`) over the analysis window (controlled by `interburst`). Defaults to SNR + 3 dB

`troughLocation` should local maxima be compared to the trough on the left and/or right of it? Values: 'left', 'right', 'both', 'either'

`summaryFun` functions used to summarize each acoustic characteristic; see `analyze`

`maxDur` long files are split into chunks `maxDur` s in duration to avoid running out of RAM; the outputs for all fragments are glued together, but plotting is switched off. Note that noise profile is estimated in each chunk separately, so set it low if the background noise is highly variable

`reportEvery` when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)

`cores` number of cores for parallel processing

`plot` if TRUE, produces a segmentation plot

`savePlots` full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)

`saveAudio` full path to the folder in which to save audio files (one per detected syllable)

`addSilence` if syllables are saved as separate audio files, they can be padded with some silence (ms)

`xlab`, `ylab`, `main` main plotting parameters

`showLegend` if TRUE, shows a legend for thresholds

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

maxPoints	the maximum number of "pixels" in the oscillogram (if any) and spectrogram; good for quickly plotting long audio files; defaults to c(1e5, 5e5); does not affect reassigned spectrograms
specPlot	a list of graphical parameters for displaying the spectrogram (if method = 'spec' or 'mel'); set to NULL to hide the spectrogram
contourPlot	a list of graphical parameters for displaying the signal contour used to detect syllables (see details)
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 graphics::plot

Details

Algorithm: for each chunk at most maxDur long, first the audio recording is partitioned into signal and noise regions: the quietest and most stable regions are located, and noise threshold is defined from a user-specified proportion of noise in the recording (propNoise) or, if propNoise = NULL, from the lowest local maximum in the density function of a weighted product of amplitude and stability (that is, we assume that quiet and stable regions are likely to represent noise). Once we know what the noise looks like - in terms of its typical amplitude and/or spectrum - we derive signal contour as its difference from noise at each time point. If method = 'env', this is Hilbert transform minus noise, and if method = 'spec' or 'mel', this is the inverse of cosine similarity between the spectrum of each frame and the estimated spectrum of noise weighted by amplitude. By default, signal-to-noise ratio (SNR) is estimated as half-median of above-noise signal, but it is recommended that this parameter is adjusted by hand to suit the purposes of segmentation, as it is the key setting that controls the balance between false negatives (missing faint signals) and false positives (hallucinating signals that are actually noise). Note also that effects of echo or reverberation can be taken into account: syllable detection threshold may be raised following powerful acoustic bursts with the help of the reverbPars argument. At the final stage, continuous "islands" SNR dB above noise level are detected as syllables, and "peaks" on the islands are detected as bursts. 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).

Value

If summaryFun = NULL, returns returns a list containing full stats on each syllable and burst (one row per syllable and per burst), otherwise returns only a dataframe with one row per file - a summary of the number and spacing of syllables and vocal bursts.

See Also

analyze ssm

Examples

```
sound = soundgen(nSyl = 4, sylLen = 100, pauseLen = 70,
                 attackLen = 20, amplGlobal = c(0, -20),
                 pitch = c(368, 284), temperature = .001)
```

```
# add noise so SNR decreases from 20 to 0 dB from syl1 to syl4
sound = sound + runif(length(sound), -10 ^ (-20 / 20), 10 ^ (-20 / 20))
# osc(sound, samplingRate = 16000, dB = TRUE)
# spectrogram(sound, samplingRate = 16000, osc = TRUE)
# playme(sound, samplingRate = 16000)

s = segment(sound, samplingRate = 16000, plot = TRUE)
s

# customizing the plot
segment(sound, samplingRate = 16000, plot = TRUE,
        sylPlot = list(lty = 2, col = 'gray20'),
        burstPlot = list(pch = 16, col = 'blue'),
        specPlot = list(col = rev(heat.colors(50))),
        xlab = 'Some custom label', cex.lab = 1.2,
        showLegend = TRUE,
        main = 'My awesome plot')
## Not run:
# set SNR manually to control detection threshold
s = segment(sound, samplingRate = 16000, SNR = 1, plot = TRUE)

# simple intensity threshold (anything >2 dB is signal)
segment(sound, 16000, method = 'env', SNR = 2, plot = TRUE,
  propNoise = 0, # don't correct SNR based on estimated background noise
  reverbPars = NULL  # don't use dynamic thresholds to cancel reverb
)

# Download 260 sounds from the supplements to Anikin & Persson (2017) at
# http://cogsci.se/publications.html
# unzip them into a folder, say '~/Downloads/temp'
myfolder = '~/Downloads/temp260'  # 260 .wav files live here
s = segment(myfolder, propNoise = .05, SNR = 3)

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

# or look at the detected syllables instead of bursts:
cor(key, s$summary$nSyl, use = 'pairwise.complete.obs')

## 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) or another reference frequency 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)

See Also

`HzToSemitones`

Examples

```
semitonesToHz(c(117, 105, 60))
```

`shiftFormants` *Shift formants*

Description

Raises or lowers formants (resonance frequencies), changing the voice quality or timbre of the sound without changing its pitch, statically or dynamically. Note that this is only possible when the fundamental frequency f0 is lower than the formant frequencies. For best results, `freqWindow` should be no lower than f0 and no higher than formant bandwidths. Obviously, this is impossible for many signals, so just try a few reasonable values, like ~200 Hz for speech. If `freqWindow` is not specified, soundgen sets it to the average detected f0, which is slow.

Usage

```
shiftFormants(
  x,
  multFormants,
  samplingRate = NULL,
  freqWindow = NULL,
  dynamicRange = 80,
  windowLength = 50,
  step = NULL,
  overlap = 75,
  wn = "gaussian",
  interpol = c("approx", "spline")[1],
  normalize = c("max", "orig", "none")[2],
  play = FALSE,
  saveAudio = NULL,
  reportEvery = NULL,
  cores = 1,
  ...
)
```

Arguments

`x`	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
`multFormants`	1 = no change, >1 = raise formants (eg 1.1 = 10% up, 2 = one octave up), <1 = lower formants. Anchor format accepted (see soundgen)
`samplingRate`	sampling rate of `x` (only needed if `x` is a numeric vector)
`freqWindow`	the width of spectral smoothing window, Hz. Defaults to detected f0
`dynamicRange`	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
`windowLength`	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)

step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
wn	window type accepted by ftwindow, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle
interpol	the method for interpolating scaled spectra
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude, "none" = no normalization
play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
saveAudio	full path to the folder in which to save audio files (one per detected syllable)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
...	other graphical parameters

Details

Algorithm: phase vocoder. In the frequency domain, we separate the complex spectrum of each STFT frame into two parts. The "receiver" is the flattened or smoothed complex spectrum, where smoothing is achieved by obtaining a smoothed magnitude envelope (the amount of smoothing is controlled by freqWindow) and then dividing the complex spectrum by this envelope. This basically removes the formants from the signal. The second component, "donor", is a scaled and interpolated version of the same smoothed magnitude envelope as above - these are the formants shifted up or down. Warping can be easily implemented instead of simple scaling if nonlinear spectral transformations are required. We then multiply the "receiver" and "donor" spectrograms and reconstruct the audio with iSTFT.

See Also

shiftPitch transplantFormants

Examples

```
s = soundgen(sylLen = 200, ampl = c(0,-10),
             pitch = c(250, 350), rolloff = c(-9, -15),
             noise = -40,
             formants = 'aii', addSilence = 50)
# playme(s)
s1 = shiftFormants(s, samplingRate = 16000, multFormants = 1.25,
                   freqWindow = 200)
# playme(s1)

## Not run:
```

```
data(sheep, package = 'seewave')  # import a recording from seewave
playme(sheep)
spectrogram(sheep)

# Lower formants by 4 semitones or ~20% = 2 ^ (-4 / 12)
sheep1 = shiftFormants(sheep, multFormants = 2 ^ (-4 / 12), freqWindow = 150)
playme(sheep1, sheep@samp.rate)
spectrogram(sheep1, sheep@samp.rate)

orig = seewave::meanspec(sheep, wl = 128, plot = FALSE)
shifted = seewave::meanspec(sheep1, wl = 128, f = sheep@samp.rate, plot = FALSE)
plot(orig[, 1], log(orig[, 2]), type = 'l')
points(shifted[, 1], log(shifted[, 2]), type = 'l', col = 'blue')

# dynamic change: raise formants at the beginning, lower at the end
sheep2 = shiftFormants(sheep, multFormants = c(1.3, .7), freqWindow = 150)
playme(sheep2, sheep@samp.rate)
spectrogram(sheep2, sheep@samp.rate)

## End(Not run)
```

`shiftPitch` *Shift pitch*

Description

Raises or lowers pitch with or without also shifting the formants (resonance frequencies) and performing a time-stretch. The three operations (pitch shift, formant shift, and time stretch) are independent and can be performed in any combination, statically or dynamically. `shiftPitch` can also be used to shift formants without changing pitch or duration, but the dedicated `shiftFormants` is faster for that task.

Usage

```
shiftPitch(
  x,
  multPitch = 1,
  multFormants = multPitch,
  timeStretch = 1,
  samplingRate = NULL,
  freqWindow = NULL,
  dynamicRange = 80,
  windowLength = 40,
  step = 2,
  overlap = NULL,
  wn = "gaussian",
  interpol = c("approx", "spline")[1],
  propagation = c("time", "adaptive")[1],
  preserveEnv = NULL,
```

```
  transplantEnv_pars = list(windowLength = 10),
  normalize = c("max", "orig", "none")[2],
  play = FALSE,
  saveAudio = NULL,
  reportEvery = NULL,
  cores = 1
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
multPitch	1 = no change, >1 = raise pitch (eg 1.1 = 10% up, 2 = one octave up), <1 = lower pitch. Anchor format accepted for multPitch / multFormant / timeStretch (see soundgen)
multFormants	1 = no change, >1 = raise formants (eg 1.1 = 10% up, 2 = one octave up), <1 = lower formants
timeStretch	1 = no change, >1 = longer, <1 = shorter
samplingRate	sampling rate of x (only needed if x is a numeric vector)
freqWindow	the width of spectral smoothing window, Hz. Defaults to detected f0 prior to pitch shifting - see shiftFormants for discussion and examples
dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
wn	window type accepted by ftwindow, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle
interpol	the method for interpolating scaled spectra and anchors
propagation	the method for propagating phase: "time" = horizontal propagation (default), "adaptive" = an experimental implementation of "vocoder done right" (Prusa & Holighaus 2017)
preserveEnv	if TRUE, transplants the amplitude envelope from the original to the modified sound with transplantEnv. Defaults to TRUE if no time stretching is performed and FALSE otherwise
transplantEnv_pars	a list of parameters passed on to transplantEnv if preserveEnv = TRUE
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude, "none" = no normalization

play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
saveAudio	full path to the folder in which to save audio files (one per detected syllable)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing

Details

Algorithm: phase vocoder. Pitch shifting is accomplished by performing a time stretch (at present, with horizontal or adaptive phase propagation) followed by resampling. This shifts both pitch and formants; to preserve the original formant frequencies or modify them independently of pitch, a variant of transplantFormants is performed to "transplant" the original or scaled formants onto the time-stretched new sound. See Prusa 2017 "Phase vocoder done right", Royer 2019 "Pitch-shifting algorithm design and applications in music".

See Also

shiftFormants transplantFormants

Examples

```
s = soundgen(sylLen = 200, ampl = c(0,-10),
             pitch = c(250, 350), rolloff = c(-9, -15),
             noise = -40,
             formants = 'aii', addSilence = 50)
# playme(s)
s1 = shiftPitch(s, samplingRate = 16000, freqWindow = 400,
                multPitch = 1.25, multFormants = .8)
# playme(s1)

## Not run:
## Dynamic manipulations
# Add a chevron-shaped pitch contour
s2 = shiftPitch(s, samplingRate = 16000, multPitch = c(1.1, 1.3, .8))
playme(s2)

# Time-stretch only the middle
s3 = shiftPitch(s, samplingRate = 16000, timeStretch = list(
  time = c(0, .25, .31, .5, .55, 1),
  value = c(1, 1, 3, 3, 1, 1))
)
playme(s3)


## Various combinations of 3 manipulations
data(sheep, package = 'seewave')  # import a recording from seewave
playme(sheep)
spectrogram(sheep)
```

```
# Raise pitch and formants by 3 semitones, shorten by half
sheep1 = shiftPitch(sheep, multPitch = 2 ^ (3 / 12), timeStretch = 0.5)
playme(sheep1, sheep@samp.rate)
spectrogram(sheep1, sheep@samp.rate)

# Just shorten
shiftPitch(sheep, multPitch = 1, timeStretch = 0.25, play = TRUE)

# Raise pitch preserving formants
sheep2 = shiftPitch(sheep, multPitch = 1.2, multFormants = 1, freqWindow = 150)
playme(sheep2, sheep@samp.rate)
spectrogram(sheep2, sheep@samp.rate)

## End(Not run)
```

---

soundgen *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 and vignette('sound_generation', package = 'soundgen').

Usage

```
soundgen(
  repeatBout = 1,
  nSyl = 1,
  sylLen = 300,
  pauseLen = 200,
  pitch = list(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 = 100,
  nonlinRandomWalk = NULL,
  subRatio = 2,
  subFreq = 0,
  subDep = 0,
  subWidth = 10000,
```

```
  shortestEpoch = 300,
  jitterLen = 1,
  jitterDep = 0,
  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 = 1,
  formantWidth = 1,
  formantCeiling = 2,
  formantLocking = 0,
  vocalTract = NA,
  amDep = 0,
  amFreq = 30,
  amType = c("logistic", "sine")[1],
  amShape = 0,
  noise = NULL,
  formantsNoise = NA,
  rolloffNoise = -4,
  noiseFlatSpec = 1200,
  rolloffNoiseExp = 0,
  noiseAmpRef = c("f0", "source", "filtered")[3],
  mouth = list(time = c(0, 1), value = c(0.5, 0.5)),
  ampl = NA,
  amplGlobal = NA,
 smoothing = list(interpol = c("approx", "spline", "loess")[3], loessSpan = NULL,
    discontThres = 0.05, jumpThres = 0.01),
  samplingRate = 16000,
  windowLength = 50,
  overlap = 75,
  addSilence = 100,
  pitchFloor = 1,
  pitchCeiling = 3500,
  pitchSamplingRate = 16000,
  dynamicRange = 80,
  invalidArgAction = c("adjust", "abort", "ignore")[1],
```

```
  plot = FALSE,
  play = FALSE,
  saveAudio = NA,
  ...
)
```

Arguments

repeatBout	number of times the whole bout should be repeated
nSyl	number of syllables in the bout. 'pitchGlobal', 'amplGlobal', and 'formants' span multiple syllables, but not multiple bouts
sylLen	average duration of each syllable, ms (vectorized)
pauseLen	average duration of pauses between syllables, ms (can be negative between bouts: force with invalidArgAction = 'ignore') (vectorized)
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 (1 = default, 0 = no stochastic behavior). amplDep, pitchDep, noiseDep: random fluctuations of user-specified amplitude / pitch / noise anchors; amplDriftDep: drift of amplitude mirroring pitch drift; formDisp: dispersion of stochastic formants; formDrift: formant frequencies; glottisDep: proportion of glottal cycle with closed glottis; pitchDriftDep: amount of slow random drift of f0; pitchDriftFreq: frequency of slow random drift of f0; rolloffDriftDep: drift of rolloff mirroring pitch drift; specDep: rolloff, rolloffNoise, nonlinear effects, attack; subDriftDep: drift of subharmonic frequency and bandwidth mirroring pitch drift; sylLenDep: duration of syllables and pauses
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

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

`subRatio`
: a positive integer giving the ratio of f0 (the main fundamental) to g0 (a lower frequency): 1 = no subharmonics, 2 = period doubling regardless of pitch changes, 3 = period tripling, etc; subRatio overrides subFreq (anchor format)

`subFreq`
: instead of a specific number of subharmonics (subRatio), we can specify the approximate g0 frequency (Hz), which is used only if subRatio = 1 and is adjusted to f0 so f0/g0 is always an integer (anchor format)

`subDep`
: the depth of subharmonics relative to the main frequency component (f0), %. 0: no subharmonics; 100: g0 harmonics are as strong as the nearest f0 harmonic (anchor format)

`subWidth`
: Width of subharmonic sidebands - regulates how rapidly g-harmonics weaken away from f-harmonics: large values like the default 10000 means that all g0 harmonics are equally strong (anchor format)

`shortestEpoch`
: minimum duration of each epoch with unchanging subharmonics regime or formant locking, in ms

`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 referring to default presets for speaker "M1" (implemented: "aoieu0") or a list of formant times, frequencies, amplitudes, and bandwidths (see examples). NA or NULL means no formants, only lip radiation. Time stamps for formants and mouthOpening can be specified in ms relative to `sylLen` or on a scale of [0, 1]. 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)

`formantCeiling` frequency to which stochastic formants are calculated, in multiples of the Nyquist frequency; increase up to ~10 for long vocal tracts to avoid losing energy in the upper part of the spectrum

`formantLocking` the approximate proportion of sound in which one of the harmonics is locked to the nearest formant, 0 = none, 1 = the entire sound (anchor format)

`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 (anchor format)

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

`amFreq` AM frequency, Hz (anchor format)

`amType` "sine" = sinusoidal, "logistic" = logistic (default)

`amShape` ignore if amType = "sine", otherwise determines the shape of non-sinusoidal AM: 0 = ~sine, -1 = notches, +1 = clicks (anchor format)

`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`, `noiseFlatSpec` linear rolloff of the excitation source for the unvoiced component, `rolloffNoise` dB/kHz (anchor format) applied above `noiseFlatSpec` Hz

`rolloffNoiseExp` exponential rolloff of the excitation source for the unvoiced component, dB/oct (anchor format) applied above 0 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)
smoothing	a list of parameters passed to getSmoothContour to control the interpolation and smoothing of contours: interpol (approx / spline / loess), loessSpan, discontThres, jumpThres
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: a vector of length 1 (symmetric) or 2 (different duration of silence before/after the sound)
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 using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
saveAudio	path + filename for saving the output, e.g. '~/Downloads/temp.wav'. If NULL = doesn't save
...	other plotting parameters passed to spectrogram

Value

Returns the synthesized waveform as a numeric vector.

See Also

generateNoise beat fart

Examples

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

```
# Set "playback" to TRUE for default system player or the name of preferred
# player (eg "aplay") to play back the audio from examples
playback = FALSE # or TRUE 'aplay', 'vlc', ...

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, 18320)

# presets of some vowels and consonants
names(presets$M1$Formants$vowels)
soundgen(sylLen = 500, formants = 'aoieu0', play = playback)

## 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,
  pitch = list(
    time = c(0, 0.65, 1),
    value = c(977, 1540, 826)),
  pitchGlobal = list(time = c(0, .5, 1), value = c(-6, 7, 0)),
  play = playback, plot = TRUE)

# Amplitude modulation
sound = soundgen(amFreq = 75, amDep = runif(10, 0, 60),
  pitch = list(
    time = c(0, .3, .9, 1), value = c(1200, 1547, 1487, 1154)),
  sylLen = 800,
  play = playback, plot = TRUE)

# Jitter and mouth opening (bark, dog-like)
```

```
sound = soundgen(repeatBout = 2, sylLen = 160, pauseLen = 100,
  jitterDep = 1,
  pitch = c(559, 785, 557),
  mouth = c(0, 0.5, 0),
  vocalTract = 5, formants = NULL,
  play = playback, plot = TRUE)

# Ultrasound - need to adjust some defaults:
 soundgen(
   sylLen = 10,  # just 10 ms
   attackLen = 1,  # should be very short for short vocalizations
   addSilence = 2,
   pitch = c(45000, 35000, 65000, 60000),  # 35-60 kHz
   rolloff = -12,
   rolloffKHz = 0,  # NB: the default is -3 dB/kHz, which we do NOT want here!
   formants = NA,  # no formants (or set vocal tract length)
   samplingRate = 350000,  # at least ~10 times the max f0
   pitchSamplingRate = 350000,  # the same as samplingRate
   windowLength = .25,  # need very short window lengths for USV
   pitchCeiling = 90000, # max allowed pitch
   invalidArgAction = 'ignore', # override the ranges allowed by default
   temperature = 1e-4,
   plot = TRUE
 )

# See the vignette on sound generation for more examples and in-depth
# explanation of the arguments to soundgen()
# Examples of code for creating human and animal vocalizations are available
# on project's homepage: http://cogsci.se/soundgen.html

## End(Not run)
```

---

soundgen_app *Interactive sound synthesizer*

Description

Starts a shiny app that provides an interactive wrapper to soundgen. Supported browsers: Firefox / Chrome. Note that the browser has to be able to playback WAV audio files, otherwise there will be no sound.

Usage

```
soundgen_app()
```

specToMS *Spectrogram to modulation spectrum*

Description

Takes a spectrogram (either complex or magnitude) and returns a MS with proper row and column labels.

Usage

```
specToMS(spec, windowLength = NULL, step = NULL)
```

Arguments

spec	target spectrogram (numeric matrix, frequency in rows, time in columns)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)

Value

Returns a MS - matrix of complex values of the same dimension as spec, with AM in rows and FM in columns.

Examples

```
s = soundgen(sylLen = 500, amFreq = 25, amDep = 50,
             pitch = 250, samplingRate = 16000)
spec = spectrogram(s, samplingRate = 16000, windowLength = 25,
  step = 5, plot = FALSE)
ms = specToMS(spec)
plotMS(log(Mod(ms)), quantiles = NULL, col = soundgen:::jet.col(100))
## Not run:
# or plot manually
image(x = as.numeric(colnames(ms)), y = as.numeric(rownames(ms)),
      z = t(log(abs(ms))), xlab = 'Amplitude modulation, Hz',
      ylab = 'Frequency modulation, cycles/kHz')
abline(h = 0, lty = 3); abline(v = 0, lty = 3)

## End(Not run)
```

specToMS_1D *Spectrogram to modulation spectrum 1D*

Description

Takes a spectrogram and returns the spectrum of each channel. The input can be an ordinary STFT spectrogram or an auditory spectrogram (a signal convolved with a bank of bandpass filters). The difference from specToMS is that, instead of taking a two-dimensional transform of the spectrogram, here the spectra are calculated independently for each frequency bin.

Usage

```
specToMS_1D(
  fb,
  samplingRate,
  windowLength = 250,
  step = windowLength/2,
  method = c("spec", "meanspec")[2]
)
```

Arguments

fb	input spectrogram (numeric matrix with frequency in rows and time in columns)
samplingRate	for auditory spectrogram, the sampling rate of input audio; for STFT spectrograms, the number of STFT frames per second
windowLength, step	determine the resolution of modulation spectra (both in ms)
method	calls either meanspec or spec

Value

Returns a modulation spectrum - a matrix of real values, with center frequencies of original filters in rows and modulation frequencies in columns.

Examples

```
data(sheep, package = 'seewave')

# auditory spectrogram
as = audSpectrogram(sheep, filterType = 'butterworth',
  nFilters = 24, plot = FALSE)
fb = t(do.call(cbind, as$filterbank_env))
rownames(fb) = names(as$filterbank_env)
ms = soundgen:::specToMS_1D(fb, sheep@samp.rate)
plotMS(log(ms+.01), logWarpX = c(10, 2), quantile = NULL, ylab = 'kHz')

# ordinary STFT spectrogram
```

```
sp = spectrogram(sheep, windowLength = 15, step = 0.5,
  output = 'original', plot = FALSE)
ms2 = soundgen:::specToMS_1D(sp, 1000 / 0.5)  # 1000/0.5 frames per s
plotMS(log(ms2+.01), quantile = NULL, ylab = 'kHz')
## Not run:
ms_spec = soundgen:::specToMS_1D(fb, sheep@samp.rate, method = 'spec')
plotMS(log(ms_spec+.01), logWarpX = c(10, 2), quantile = NULL, ylab = 'kHz')

## End(Not run)
```

`spectrogram` *Spectrogram*

Description

Produces the spectrogram of a sound using short-time Fourier transform. Inspired by `spectro`, this function offers added routines for reassignment, multi-resolution spectrograms, noise reduction, smoothing in time and frequency domains, manual control of contrast and brightness, plotting the oscillogram on a dB scale, grid, etc.

Usage

```
spectrogram(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  dynamicRange = 80,
  windowLength = 50,
  step = windowLength/2,
  overlap = NULL,
  specType = c("spectrum", "reassigned", "spectralDerivative")[1],
  logSpec = TRUE,
  rasterize = FALSE,
  wn = "gaussian",
  zp = 0,
  normalize = TRUE,
  smoothFreq = 0,
  smoothTime = 0,
  qTime = 0,
  percentNoise = 10,
  noiseReduction = 0,
  output = c("original", "processed", "complex", "all")[1],
  specManual = NULL,
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
```

```
  savePlots = NULL,
  osc = c("none", "linear", "dB")[2],
  heights = c(3, 1),
  ylim = NULL,
  yScale = c("linear", "log", "bark", "mel", "ERB")[1],
  contrast = 0.2,
  brightness = 0,
  blur = 0,
  maxPoints = c(1e+05, 5e+05),
  padWithSilence = TRUE,
  colorTheme = c("bw", "seewave", "heat.colors", "...")[1],
  col = NULL,
  extraContour = NULL,
  xlab = NULL,
  ylab = NULL,
  xaxp = NULL,
  mar = c(5.1, 4.1, 4.1, 2),
  main = NULL,
  grid = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input used for normalization of input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
specType	plot the original FFT ('spectrum'), reassigned spectrogram ('reassigned'), or spectral derivative ('spectralDerivative')
logSpec	if TRUE, log-transforms the spectrogram

`rasterize` (only applies if specType = 'reassigned') if TRUE, the reassigned spectrogram is plotted after rasterizing it: that is, showing density per time-frequency bins with the same resolution as an ordinary spectrogram

`wn` window type accepted by `ftwindow`, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle

`zp` window length after zero padding, points

`normalize` if TRUE, scales input prior to FFT

`smoothFreq, smoothTime` length of the window for median smoothing in frequency and time domains, respectively, points

`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 (non-negative number, 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`

`output` specifies what to return: nothing ('none'), unmodified spectrogram ('original'), denoised and/or smoothed spectrogram ('processed'), or unmodified spectrogram with the imaginary part giving phase ('complex')

`specManual` manually calculated spectrogram-like representation in the same format as the output of spectrogram(): rows = frequency in kHz, columns = time in ms

`reportEvery` when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)

`cores` number of cores for parallel processing

`plot` should a spectrogram be plotted? TRUE / FALSE

`savePlots` full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)

`osc` "none" = no oscillogram; "linear" = on the original scale; "dB" = in decibels

`heights` a vector of length two specifying the relative height of the spectrogram and the oscillogram (including time axes labels)

`ylim` frequency range to plot, kHz (defaults to 0 to Nyquist frequency). NB: still in kHz, even if yScale = bark, mel, or ERB

`yScale` scale of the frequency axis: 'linear' = linear, 'log' = logarithmic (musical), 'bark' = bark with `hz2bark`, 'mel' = mel with `hz2mel`, 'ERB' = Equivalent Rectangular Bandwidths with `HzToERB`

`contrast` a number, recommended range -1 to +1. The spectrogram is raised to the power of `exp(3 * contrast)`. Contrast >0 increases sharpness, <0 decreases sharpness

`brightness` how much to "lighten" the image (>0 = lighter, <0 = darker)

`blur` apply a Gaussian filter to blur or sharpen the image, two numbers: frequency (Hz), time (ms). A single number is interpreted as frequency, and a square filter

	is applied. NA / NULL / 0 means no blurring in that dimension. Negative numbers mean un-blurring (sharpening) the image by dividing instead of multiplying by the filter during convolution
maxPoints	the maximum number of "pixels" in the oscillogram (if any) and spectrogram; good for quickly plotting long audio files; defaults to c(1e5, 5e5); does not affect reassigned spectrograms
padWithSilence	if TRUE, pads the sound with just enough silence to resolve the edges properly (only the original region is plotted, so the apparent duration doesn't change)
colorTheme	black and white ('bw'), as in seewave package ('seewave'), matlab-type palette ('matlab'), or any palette from palette such as 'heat.colors', 'cm.colors', etc
col	actual colors, eg rev(rainbow(100)) - see ?hcl.colors for colors in base R (overrides colorTheme)
extraContour	a vector of arbitrary length scaled in Hz (regardless of yScale!) that will be plotted over the spectrogram (eg pitch contour); can also be a list with extra graphical parameters such as lwd, col, etc. (see examples)
xlab, ylab, main, mar, xaxp	graphical parameters for plotting
grid	if numeric, adds n = grid dotted lines per kHz
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters

Details

Many soundgen functions call spectrogram, and you can pass along most of its graphical parameters from functions like soundgen, analyze, etc. However, in some cases this will not work (eg for "units") or may produce unexpected results. If in doubt, omit extra graphical parameters or save your sound first, then call spectrogram() explicitly. Reassigned spectrograms are not affected by noise reduction or blurring.

Value

Returns nothing if output = 'none', spectral magnitudes - not power! - if output = 'original', denoised and/or smoothed spectrum if output = 'processed', or spectral derivatives if specType = 'spectralDerivative'. The output is a matrix of real numbers with time in columns (ms) and frequency in rows (kHz). For multi-resolution spectrograms, the complex matrix corresponds to the last value of windowLength.

See Also

osc modulationSpectrum ssm

Examples

```
# synthesize a sound 500 ms long, with gradually increasing hissing noise
sound = soundgen(sylLen = 500, temperature = 0.001, noise = list(
  time = c(0, 650), value = c(-40, 0)), formantsNoise = list(
```

```
  f1 = list(freq = 5000, width = 10000)))
# playme(sound, samplingRate = 16000)

# basic spectrogram
spectrogram(sound, samplingRate = 16000, yScale = 'bark')

# add bells and whistles
spectrogram(sound, samplingRate = 16000,
  osc = 'dB',  # plot oscillogram in dB
  heights = c(2, 1),  # spectro/osc height ratio
  noiseReduction = 1.1,  # subtract the spectrum of noisy parts
  brightness = -1,  # reduce brightness
  # pick color theme - see ?hcl.colors
  # colorTheme = 'heat.colors',
  # ...or just specify the actual colors
  col = colorRampPalette(c('white', 'yellow', 'red'))(50),
  cex.lab = .75, cex.axis = .75,  # text size and other base graphics pars
  grid = 5,  # lines per kHz; to customize, add manually with graphics::grid()
  ylim = c(0, 5),  # always in kHz
  main = 'My spectrogram' # title
  # + axis labels, etc
)
## Not run:
# save spectrograms of all sounds in a folder
spectrogram('~/Downloads/temp', savePlots = '', cores = 2)

# change dynamic range
spectrogram(sound, samplingRate = 16000, dynamicRange = 40)
spectrogram(sound, samplingRate = 16000, dynamicRange = 120)

# remove the oscillogram
spectrogram(sound, samplingRate = 16000, osc = 'none')  # or NULL etc

# frequencies on a logarithmic (musical) scale (mel/bark/ERB also available)
spectrogram(sound, samplingRate = 16000,
            yScale = 'log', ylim = c(.05, 8))

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

# reassigned spectrograms can be plotted without rasterizing, as a
# scatterplot instead of a contour plot
s = soundgen(sylLen = 500, pitch = c(100, 1100, 120, 1200, 90, 900, 110, 700),
  samplingRate = 22050, formants = NULL, lipRad = 0, rolloff = -20)
spectrogram(s, 22050, windowLength = 5, step = 1, yScale = 'bark')
spectrogram(s, 22050, specType = 'reassigned', windowLength = 5,
  step = 1, yScale = 'bark')
# ...or it can be rasterized, but that sacrifices frequency resolution:s
sp = spectrogram(s, 22050, specType = 'reassigned', rasterize = TRUE,
                 windowLength = 5, step = 1, yScale = 'bark', output = 'all')
# The raw reassigned version is saved if output = 'all' for custom plotting
df = sp$reassigned
df$z1 = soundgen:::zeroOne(log(df$magn))
```

```
plot(df$time, df$freq, col = rgb(df$z1, df$z1, 1 - df$z1, 1),
  pch = 16, cex = 0.25, ylim = c(0, 2))

# multi-resolution spectrograms
spectrogram(s, 22050, windowLength = c(1, 10, 20, 50), yScale = 'bark')
# (works well in combination with de-blurring)
spectrogram(s, 22050, windowLength = c(1, 10, 20, 50), yScale = 'bark',
  blur = c(-50, -50))
spectrogram(s, 22050, windowLength = 1:10, yScale = 'bark',
  specType = 'reassigned', dynamicRange = 50)

# 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

# increase contrast, reduce brightness
spectrogram(sound, samplingRate = 16000, contrast = .7, brightness = -.5)

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

# Gaussian filter to blur or sharpen ("unblur") the image in time and/or
# frequency domains
spectrogram(sound, samplingRate = 16000, blur = c(100, 500))
# TIP: when unblurring, set the first (frequency) parameter to the
# frequency resolution of interest, eg ~500-1000 Hz for human formants
spectrogram(sound, samplingRate = 16000, windowLength = 10, blur = c(-500, 50))

# specify location of tick marks etc - see ?par() for base graphics
spectrogram(sound, samplingRate = 16000,
            ylim = c(0, 3), yaxp = c(0, 3, 5), xaxp = c(0, .8, 10))

# Plot long audio files with reduced resolution
data(sheep, package = 'seewave')
sp = spectrogram(sheep, overlap = 0,
  maxPoints = c(1e4, 5e3),  # limit the number of pixels in osc/spec
  output = 'original')
nrow(sp) * ncol(sp) / 5e3  # spec downsampled by a factor of ~2

# Plot some arbitrary contour over the spectrogram (simply calling lines()
# will not work if osc = TRUE b/c the plot layout is modified)
s = soundgen(sylLen = 1500, pitch = c(250, 350, 320, 220),
  jitterDep = c(0, 0, 3, 2, 0, 0))
an = analyze(s, 16000, plot = FALSE)
spectrogram(s, 16000, extraContour = an$detailed$dom,
  ylim = c(0, 2), yScale = 'bark')
# or simply (but without an oscillogram):
spectrogram(s, 16000, ylim = c(0, 2), yScale = 'bark', osc = FALSE)
```

```
points(an$detailed$time/1000,  # time in s
       6 * asinh(an$detailed$dom/600),  # values in barks
       lwd = 2, col = 'green', lty = 2  # any graphic pars
)
# For values that are not in Hz, normalize any way you like
spectrogram(s, 16000, ylim = c(0, 2), extraContour = list(
  x = an$detailed$loudness / max(an$detailed$loudness, na.rm = TRUE) * 2000,
  # ylim[2] = 2000 Hz
  type = 'b', pch = 5, lwd = 2, lty = 2, col = 'blue'))

# Plot a spectrogram-like matrix paired with an osc
ms = modulationSpectrum(s, 16000, msType = '1D', amRes = 10)
spectrogram(s, 16000, specManual = ms$modulation_spectrogram,
  colorTheme = 'matlab', ylab = 'Modulation frequency, kHz',
  contrast = .25, blur = c(10, 10))

## End(Not run)
```

---

ssm *Self-similarity matrix*

Description

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

Usage

```
ssm(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  windowLength = 25,
  step = 5,
  overlap = NULL,
  ssmWin = NULL,
  sparse = FALSE,
  maxFreq = NULL,
  nBands = NULL,
  MFCC = 2:13,
  input = c("mfcc", "melspec", "spectrum")[2],
  norm = FALSE,
  simil = c("cosine", "cor")[1],
  kernelLen = 100,
  kernelSD = 0.5,
  padWith = 0,
  summaryFun = c("mean", "sd"),
  reportEvery = NULL,
```

```
  cores = 1,
  plot = TRUE,
  savePlots = NULL,
  main = NULL,
  heights = c(2, 1),
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  specPars = list(levels = seq(0, 1, length = 30), colorTheme = c("bw", "seewave",
    "heat.colors", "...")[2], xlab = "Time, s", ylab = "kHz"),
  ssmPars = list(levels = seq(0, 1, length = 30), colorTheme = c("bw", "seewave",
    "heat.colors", "...")[2], xlab = "Time, s", ylab = "Time, s"),
  noveltyPars = list(type = "b", pch = 16, col = "black", lwd = 3)
)
```

Arguments

x	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
samplingRate	sampling rate of x (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
ssmWin	window for averaging SSM, ms (has a smoothing effect and speeds up the processing)
sparse	if TRUE, the entire SSM is not calculated, but only the central region needed to extract the novelty contour (speeds up the processing)
maxFreq	highest band edge of mel filters, Hz. Defaults to samplingRate / 2. See melfcc
nBands	number of warped spectral bands to use. Defaults to 100 * windowLength / 20. See melfcc
MFCC	which mel-frequency cepstral coefficients to use; defaults to 2:13
input	the spectral representation used to calculate the SSM
norm	if TRUE, the spectrum of each STFT frame is normalized
simil	method for comparing frames: "cosine" = cosine similarity, "cor" = Pearson's correlation
kernelLen	length of checkerboard kernel for calculating novelty, ms (larger values favor global, slow vs. local, fast 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
summaryFun	functions used to summarize each acoustic characteristic, eg "c('mean', 'sd')"; user-defined functions are fine (see examples); NAs are omitted automatically for mean/median/sd/min/max/range/sum, otherwise take care of NAs yourself
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing
plot	if TRUE, plots the SSM
savePlots	full path to the folder in which to save the plots (NULL = don't save, '' = same folder as audio)
main	plot title
heights	relative sizes of the SSM and spectrogram/novelty plot
width, height, units, res	graphical parameters for saving plots passed to png
specPars	graphical parameters passed to filled.contour.mod and affecting the spectrogram
ssmPars	graphical parameters passed to filled.contour.mod and affecting the plot of SSM
noveltyPars	graphical parameters passed to lines and affecting the novelty contour

Value

Returns a list of two components: $ssm contains the self-similarity matrix, and $novelty contains the novelty vector.

References

- El Badawy, D., Marmaroli, P., & Lissek, H. (2013). Audio Novelty-Based Segmentation of Music Concerts. In Acoustics 2013 (No. EPFL-CONF-190844)
- Foote, J. (1999, October). Visualizing music and audio using self-similarity. In Proceedings of the seventh ACM international conference on Multimedia (Part 1) (pp. 77-80). ACM.
- Foote, J. (2000). Automatic audio segmentation using a measure of audio novelty. In Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on (Vol. 1, pp. 452-455). IEEE.

See Also

spectrogram modulationSpectrum segment

Examples

```
sound = c(soundgen(),
          soundgen(nSyl = 4, sylLen = 50, pauseLen = 70,
          formants = NA, pitch = c(500, 330)))
# playme(sound)
# detailed, local features (captures each syllable)
```

```
s1 = ssm(sound, samplingRate = 16000, kernelLen = 100,
         sparse = TRUE)  # much faster with 'sparse'
# more global features (captures the transition b/w the two sounds)
s2 = ssm(sound, samplingRate = 16000, kernelLen = 400, sparse = TRUE)

s2$summary
s2$novelty  # novelty contour
## Not run:
ssm(sound, samplingRate = 16000,
    input = 'mfcc', simil = 'cor', norm = TRUE,
    ssmWin = 25,  # speed up the processing
    kernelLen = 300,  # global features
    specPars = list(colorTheme = 'seewave'),
    ssmPars = list(col = rainbow(100)),
    noveltyPars = list(type = 'l', lty = 3, lwd = 2))

## End(Not run)
```

`timeStretch` *Time stretch*

Description

Dynamically time-stretches a sound without preserving its pitch or formants, as if gradually changing playback speed. Algorithm: the audio is resampled at time-varying steps. This is about 100 times faster than time-stretching with a phase vocoder in `shiftPitch`, but pitch and formants cannot be preserved, and large stretch factors may cause artifacts due to aliasing.

Usage

```
timeStretch(
  x,
  stretch = 1,
  samplingRate = NULL,
  precision = 1000,
  play = FALSE,
  saveAudio = NULL,
  reportEvery = NULL,
  cores = 1
)
```

Arguments

`x`	path to a folder, one or more wav or mp3 files c('file1.wav', 'file2.mp3'), Wave object, numeric vector, or a list of Wave objects or numeric vectors
`stretch`	1 = no change, >1 = longer, <1 = shorter. Single value, vector, or anchor format (see `soundgen`)
`samplingRate`	sampling rate of `x` (only needed if `x` is a numeric vector)

precision	the number of points used for estimating the duration of output (more = better, but slower)
play	if TRUE, plays the synthesized sound using the default player on your system. If character, passed to play as the name of player to use, eg "aplay", "play", "vlc", etc. In case of errors, try setting another default player for play
saveAudio	full path to the folder in which to save audio files (one per detected syllable)
reportEvery	when processing multiple inputs, report estimated time left every ... iterations (NULL = default, NA = don't report)
cores	number of cores for parallel processing

See Also

shiftPitch

Examples

```
data(sheep, package = 'seewave')  # import a recording from seewave
# playme(sheep)
# spectrogram(sheep)
s1 = timeStretch(sheep, stretch = c(1, 3))
# playme(s1, sheep@samp.rate)
# spectrogram(s1, sheep@samp.rate)

# compare to a similar effect achieved with a phase vocoder in pitchShift():
s2 = shiftPitch(
  sheep,
  timeStretch = c(1, 3),  # from 1 (original) to mult
  multPitch = c(1, 1/3),  # also drop pitch
  multFormants = c(1, 1/3)  # also drop formants (by the same proportion)
)
# playme(s2, sheep@samp.rate)
# spectrogram(s2, sheep@samp.rate)
# NB: because the two algorithms calculate transitions between stretch
# factors in different ways, the duration is not identical, even though the
# range of pitch change is the same
```

---

transplantEnv *Transplant envelope*

Description

Extracts a smoothed amplitude envelope of the donor sound and applies it to the recipient sound. Both sounds are provided as numeric vectors; they can differ in length and sampling rate. Note that the result depends on the amount of smoothing (controlled by windowLength) and the chosen method of calculating the envelope. Very similar to setenv, but with a different smoothing algorithm and with a choice of several types of envelope: hil, rms, or peak.

Usage

```
transplantEnv(
  donor,
  samplingRateD = NULL,
  recipient,
  samplingRateR = samplingRateD,
  windowLength = 50,
  method = c("hil", "rms", "peak")[3],
  killDC = FALSE,
  dynamicRange = 80,
  plot = FALSE
)
```

Arguments

donor	the sound that "donates" the amplitude envelope
samplingRateD, samplingRateR	sampling rate of the donor and recipient, respectively (only needed for vectors, not files)
recipient	the sound that needs to have its amplitude envelope adjusted
windowLength	the length of smoothing window, ms
method	hil = Hilbert envelope, rms = root mean square amplitude, peak = peak amplitude per window
killDC	if TRUE, dynamically removes DC offset or similar deviations of average waveform from zero (see examples)
dynamicRange	parts of sound quieter than -dynamicRange dB will not be amplified
plot	if TRUE, plots the original sound, the smoothed envelope, and the compressed sound

Value

Returns the recipient sound with the donor's amplitude envelope - a numeric vector with the same sampling rate as the recipient

See Also

flatEnv setenv

Examples

```
donor = rnorm(500) * seq(1, 0, length.out = 500)
recipient = soundgen(sylLen = 600, addSilence = 50)
transplantEnv(donor, samplingRateD = 200,
              recipient, samplingRateR = 16000,
              windowLength = 50, method = 'hil', plot = TRUE)
transplantEnv(donor, samplingRateD = 200,
              recipient, samplingRateR = 16000,
              windowLength = 10, method = 'peak', plot = TRUE)
```

`transplantFormants` *Transplant formants*

Description

Takes the general spectral envelope of one sound (`donor`) and "transplants" it onto another sound (`recipient`). For biological sounds like speech or animal vocalizations, this has the effect of replacing the formants in the recipient sound while preserving the original intonation and (to some extent) voice quality. Note that the amount of spectral smoothing (specified with `freqWindow` or `blur`) is a crucial parameter: too little smoothing, and noise between harmonics will be amplified, creasing artifacts; too much, and formants may be missed. The default is to set `freqWindow` to the estimated median pitch, but this is time-consuming and error-prone, so set it to a reasonable value manually if possible. Also ensure that both sounds have the same sampling rate.

Usage

```
transplantFormants(
  donor,
  recipient,
  samplingRate = NULL,
  freqWindow = NULL,
  blur = NULL,
  dynamicRange = 80,
  windowLength = 50,
  step = NULL,
  overlap = 90,
  wn = "gaussian",
  zp = 0
)
```

Arguments

`donor`	the sound that provides the formants (vector, Wave, or file) or the desired spectral filter (matrix) as returned by `getSpectralEnvelope`
`recipient`	the sound that receives the formants (vector, Wave, or file)
`samplingRate`	sampling rate of `x` (only needed if `x` is a numeric vector)
`freqWindow`	the width of smoothing window used to flatten the recipient's spectrum per frame. Defaults to median pitch of the donor (or of the recipient if donor is a filter matrix). If `blur` is NULL, `freqWindow` also controls the amount of smoothing applied to the donor's spectrogram
`blur`	the amount of Gaussian blur applied to the donor's spectrogram as a faster and more flexible alternative to smoothing it per bin with `freqWindow`. Provide two numbers: frequency (Hz, normally approximately equal to freqWindow), time (ms) (NA / NULL / 0 means no blurring in that dimension). See examples and `spectrogram`

dynamicRange	dynamic range, dB. All values more than one dynamicRange under maximum are treated as zero
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram)
step	you can override overlap by specifying FFT step, ms - a vector of the same length as windowLength (NB: because digital audio is sampled at discrete time intervals of 1/samplingRate, the actual step and thus the time stamps of STFT frames may be slightly different, eg 24.98866 instead of 25.0 ms)
overlap	overlap between successive FFT frames, %
wn	window type accepted by ftwindow, currently gaussian, hanning, hamming, bartlett, blackman, flattop, rectangle
zp	window length after zero padding, points

Details

Algorithm: makes spectrograms of both sounds, interpolates and smooths or blurs the donor spectrogram, flattens the recipient spectrogram, multiplies the spectrograms, and transforms back into time domain with inverse STFT.

See Also

transplantEnv getSpectralEnvelope addFormants spectrogram soundgen

Examples

```
## Not run:
# Objective: take formants from the bleating of a sheep and apply them to a
# synthetic sound with any arbitrary duration, intonation, nonlinearities etc
data(sheep, package = 'seewave')  # import a recording from seewave
playme(sheep)
spectrogram(sheep, osc = TRUE)

recipient = soundgen(
  sylLen = 1200,
  pitch = c(100, 300, 250, 200),
  vibratoFreq = 9, vibratoDep = 1,
  addSilence = 180,
  samplingRate = sheep@samp.rate,  # same as donor
  invalidArgAction = 'ignore')  # force to keep the low samplingRate
playme(recipient, sheep@samp.rate)
spectrogram(recipient, sheep@samp.rate, osc = TRUE)

s1 = transplantFormants(
  donor = sheep,
  recipient = recipient,
  samplingRate = sheep@samp.rate)
playme(s1, sheep@samp.rate)
spectrogram(s1, sheep@samp.rate, osc = TRUE)

# The spectral envelope of s1 will be similar to sheep's on a frequency scale
```

```
# determined by freqWindow. Compare the spectra:
par(mfrow = c(1, 2))
seewave::meanspec(sheep, dB = 'max0', alim = c(-50, 20), main = 'Donor')
seewave::meanspec(s1, f = sheep@samp.rate, dB = 'max0',
                  alim = c(-50, 20), main = 'Processed recipient')
par(mfrow = c(1, 1))

# if needed, transplant amplitude envelopes as well:
s2 = transplantEnv(donor = sheep, samplingRateD = sheep@samp.rate,
                   recipient = s1, windowLength = 10)
playme(s2, sheep@samp.rate)
spectrogram(s2, sheep@samp.rate, osc = TRUE)

# using "blur" to apply Gaussian blur to the donor's spectrogram instead of
# smoothing per frame with "freqWindow" (~2.5 times faster)
spectrogram(sheep, blur = c(150, 0))  # preview to select the amount of blur
s1b = transplantFormants(
  donor = sheep,
  recipient = recipient,
  samplingRate = sheep@samp.rate,
  freqWindow = 150,
  blur = c(150, 0))
  # blur: 150 = SD of 150 Hz along the frequency axis,
  #      0 = no smoothing along the time axis
playme(s1b, sheep@samp.rate)
spectrogram(s1b, sheep@samp.rate, osc = TRUE)

# Now we use human formants on sheep source: the sheep asks "why?"
s3 = transplantFormants(
  donor = getSpectralEnvelope(
            nr = 512, nc = 100,  # fairly arbitrary dimensions
            formants = 'uaaai',
            samplingRate = sheep@samp.rate),
  recipient = sheep,
  samplingRate = sheep@samp.rate)
playme(s3, sheep@samp.rate)
spectrogram(s3, sheep@samp.rate, osc = TRUE)

## End(Not run)
```

Index