Package: soundgen (via r-universe)

October 20, 2026

Type Package

Title Sound Synthesis and Acoustic Analysis

Version 3.0.0

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

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

Description Parametric source-filter synthesis of harmonic-noise signals, such as animal vocalizations and human voice, with control over pitch, formants, noise, amplitude modulation, nonlinear phenomena, and morphing. General signal processing tools for audio analysis and manipulation: pitch tracking, formant and vocal tract length estimation, reassigned and auditory spectrograms, modulation spectra and psychoacoustic roughness, self-similarity and surprisal, audio segmentation, pitch and formant shifting, etc. Includes four interactive web apps for audio synthesis, annotation, formant analysis, and manually correcting pitch contours. Reference: Anikin (2019) <doi:10.3758/s13428-018-1095-7>.

License GPL (>= 2)

Encoding UTF-8

LazyData true

Imports stats, graphics, grDevices, utils, tuneR, signal, phonTools

Suggests base64enc, dtw, nonlinearTseries, shiny, shinyjs, bslib

Depends R (>= 4.1)

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Andrey Anikin [aut, cre]

Repository https://cran.r-universe.dev

Date/Publication 2026-09-20 12:29:57 UTC

RemoteUrl https://github.com/cran/soundgen

RemoteRef HEAD

RemoteSha ea27b027896d4aab7bddf349e6687013c6c78d98

Contents

acf_fft . . . 3
addAM . . . 4
addFormants . . . 7
analyze . . . 11
annotation_app . . . 20
audSpectrogram . . . 22
bandpass . . . 26
beat . . . 29
compareSounds . . . 31
crossFade . . . 35
detectNLP . . . 37
detectNLP_training_nonv . . . 40
estimateVTL . . . 41
fade . . . 43
filterSoundByMS . . . 45
findPeaks . . . 50
flatEnv . . . 51
formant_app . . . 54
gaussianSmooth2D . . . 56
getDuration . . . 57
getEntropy . . . 58
getEnv . . . 60
getFormantFilter . . . 61
getLoudness . . . 64
getRMS . . . 68
getRolloff . . . 71
getSpecEnv . . . 73
getSurprisal . . . 75
hilbert_exact . . . 83
hillenbrand . . . 84
HzToOther . . . 84
interpolate . . . 86
interpolateNA . . . 89
invertSpectrogram . . . 90
IR . . . 92
matchPars . . . 93
mixAtSNR . . . 95
modulationSpectrum . . . 97
morph . . . 105
naiveBayes . . . 107
noiseRemoval . . . 110
normalizeFolder . . . 113
osc . . . 115
phasegram . . . 117
pitch_app . . . 121
pitchDescriptives . . . 124

pitchSmoothPraat . . . 126
playme . . . 127
plotMS . . . 128
presets . . . 130
prosody . . . 131
reportCI . . . 133
reportTime . . . 134
resample . . . 135
reverb . . . 138
schwa . . . 141
segment . . . 144
segment_ann . . . 150
shiftFormants . . . 151
shiftPitch . . . 153
soundgen . . . 157
soundgen_app . . . 165
specToMS . . . 165
specToMS_1D . . . 166
spectrogram . . . 168
spectrum . . . 175
speechEx . . . 178
ssm . . . 178
stft_simple . . . 182
timeStretch . . . 185
transplantEnv . . . 187
transplantFormants . . . 189
winFun . . . 191

Index **195**

`acf_fft` *Autocorrelation with FFT*

Description

Analogous to `acf`, but based on the Fast Fourier Transform (Wiener-Khinchin theorem) and 5-10 times faster, especially for long input vectors.

Usage

```
acf_fft(x, center = TRUE, padToMult = 2)
```

Arguments

`x`	numeric vector
`center`	if TRUE (default), x is centered before padding with 0
`padToMult`	pad with 0 to the smallest power of 2 above `padToMult * length(x) - 1`, must be >=1 (padToMult = 1 means no padding, leading to circular ACF)

Value

Numeric vector that is usually longer than input because it is padded with zeros to the next power of two. Constant and zero inputs return NA.

Examples

```
len = 200
x = sin(2 * pi * 100 * (1:len) / 1000) + rnorm(len, 0, .5)
plot(x, type = 'l')
aut = acf(x, lag.max = len/2)
aut2 = acf_fft(x)
points(0:100, aut2[1:(len/2+1)], type = 'l', col = 'blue')
aut$acf[1:10]
aut2[1:10]

# compare execution time
system.time(for (i in 1:100) acf(x, lag.max = len/2, plot = FALSE))
system.time(for (i in 1:100) acf_fft(x))
```

---

addAM *Add amplitude modulation*

Description

Adds sinusoidal or logistic amplitude modulation to a sound. Sinusoidal AM creates a single pair of sidebands at ±amFreq around each original harmonic, whereas non-sinusoidal AM creates broader sidebands with more extra harmonics (see examples).

Usage

```
addAM(
  x,
  samplingRate = NULL,
  amDep = 25,
  amFreq = 30,
  amType = c("logistic", "sine"),
  amShape = 0,
  invalidArgAction = c("adjust", "abort", "ignore"),
  play = FALSE,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  reportEvery = NULL,
  cores = 1,
  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)
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	"logistic" = logistic (default), "sine" = sinusoidal
amShape	ignored 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 permitted range: 'adjust' = reset to default value, 'abort' = stop execution, 'ignore' = throw a warning and continue (may crash)
play	if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
plot	if TRUE, produces a plot of the results
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
width, height, units, res	graphical parameters for saving plots passed to png

Value

Returns the modified audio as a numeric vector with the original sampling rate.

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 ±amFreq
# around each f0 harmonic (amFreq, and thus the width of sidebands,
# may vary over time):
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 amFreq 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 amFreq on a linear scale, so we align them by extracting
# a long contour on a log scale for both)
con = soundgen:::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 = TRUE, 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 getFormantFilter 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 formantFilter argument (e.g., for a low/high/bandpass filter).

Usage

```
addFormants(
  x,
  samplingRate = NULL,
  formants = NULL,
  formantFilter = NULL,
  action = c("add", "remove"),
  dB = NULL,
  specificity = 1,
  zFun = NULL,
  vocalTract = NA,
  formantDep = 1,
  formantDepStoch = 1,
  formantWidth = 1,
  formantCeiling = NULL,
  lipRad = 6,
  noseRad = 4,
  mouthOpenThres = 0,
  mouth = NA,
  temperature = 0.025,
  formDrift = 0.3,
  formDisp = 0.2,
  smoothing = list(interpol = "splineFC"),
  windowLength = 50,
  step = NULL,
  overlap = 75,
  wn = "gaussian",
  normalize = c("orig", "max", "none"),
  play = FALSE,
  saveAudio = FALSE,
  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)
`formants`	a vector of formant frequencies (assuming formants are static throughout the sound); a list of formant times, frequencies, amplitudes, and bandwidths; or a character string referring to default presets for speaker "M1" (implemented: "aoieu0"). NA or NULL means no formants, only lip radiation (but a schwa is generated if `vocalTract` is specified). Time stamps for formants and mouth can be specified in ms relative to `sylLen` or on a scale of `[0, 1]`. See `getFormantFilter` for more details
`formantFilter`	(optional): as an alternative to specifying formant frequencies, we can provide the exact filter - a vector of non-negative numbers specifying the amplitude in each frequency bin on a linear scale. A matrix specifying the filter for each STFT step with frequency bins in rows and STFT frames in columns is also accepted. The easiest way to create this matrix is to call `getFormantFilter` or to use the spectrum of a recorded sound
`action`	'add' = add formants to the sound (default), 'remove' = remove formants (inverse filtering)
`dB`	if NULL (default), the spectral envelope is applied on the original scale; otherwise, it is set to range up 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 to avoid losing energy in the upper part of the spectrum due to unmodeled resonances above the Nyquist frequencies, specified in multiples of Nyquist. If NULL (default), a faster theoretical correction is used (may fail for unusual sounds)
`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 interpolate to control the interpolation and smoothing of contours drawn through anchors
windowLength	length of the analysis window, ms
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
wn	window type accepted by winFun: character string or function
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude for the given input scale, "none" = no normalization
play	if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
...	extra parameters passed to zFun

Details

Algorithm: converts input from a time series (time domain) to a spectrogram (frequency domain) through short-time Fourier transform (STFT), multiplies 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.

Value

The filtered waveform as a numeric vector of the original length with the original sampling rate, or a list if there are multiple inputs.

See Also

getFormantFilter transplantFormants soundgen

Examples

```
sound = c(rep(0, 1000), rnorm(8000) * 2 - 1, rep(0, 1000))  # white noise
# NB: pad with silence to avoid artifacts 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 50 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 formantFilter are defined, only lipRad has an effect
# For ex., we can boost low frequencies by 6 dB/oct
noise = rnorm(8000)
noise1 = addFormants(noise, 16000, lipRad = -6)
meanSpectrum(noise1, 16000, yScale = 'max0')

# Arbitrary spectra can be defined with formantFilter. 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, formantFilter = 10 ^ (sp_dB / 20))
```

```
meanSpectrum(noise2, 16000, yScale = 'max0')

## Use the spectral envelope of another recording
# (NB: this can also be achieved with a single call to transplantFormants)
sound_orig = soundgen(sylLen = 300, formants = 'a', addSilence = 5)
samplingRate = 16000
# 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_orig = spectrogram(sound_orig, blur = c(300, 50),
 samplingRate = samplingRate, output = 'original', plot = TRUE)

# Synthesize source only, with flat spectrum
sound_unfilt = soundgen(sylLen = 2500, pitch = pitch,
  rolloff = 0, rolloffOct = 0,
  temperature = 0, formants = NULL, lipRad = 0,
  samplingRate = samplingRate,
  invalidArgAction = 'ignore')  # prevent soundgen from increasing samplingRate
# playme(sound_unfilt, samplingRate)
# meanSpectrum(sound_unfilt, samplingRate, yScale = 'max0')  # ~flat

# Force spectral envelope to the shape of target
sound_filt = addFormants(sound_unfilt, formants = NULL,
  formantFilter = specEnv_orig, 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))
meanSpectrum(sound_orig, samplingRate, yScale = 'max0', alim = c(-50, 20))
meanSpectrum(sound_filt, samplingRate, yScale = 'max0', alim = c(-50, 20))
par(mfrow = c(1, 1))

## End(Not run)
```

---

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. 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. For more information, see https://cogsci.se/soundgen/acoustic_analysis.html

Usage

```
analyze(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  dynamicRange = 80,
  silence = 0.04,
  windowLength = 50,
  step = NULL,
  overlap = 50,
  wn = "gaussian",
  zp = 0,
  cutFreq = NULL,
  nFormants = 3,
  formants = list(),
  loudness = list(SPL_measured = 70),
 roughness = list(msType = "1D", specMethod = "spectrum", amRes = 1, specFun_pars =
    list(windowLength = 25, step = 2)),
  novelty = list(specFun = "melspec", kernelLen = 1000),
  pitchMethods = c("dom", "autocor"),
  pitchManual = NULL,
  pitchFloor = 75,
  pitchCeiling = 1000,
  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),
  flux = list(thres = 0.15),
  amRange = c(10, 60),
  fmRange = NULL,
  shortestSyl = 20,
```

```
  shortestPause = 60,
  interpolPitch = list(win = 75, tol = 0.3, cert = 0.3),
  certWeight = 0.5,
  smooth = 1,
  smoothVars = c("pitch", "dom"),
  summaryFun = c("mean", "median", "sd"),
  invalidArgAction = c("adjust", "abort", "ignore"),
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  osc = c("linear", "dB", "none"),
  showLegend = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  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 to normalize the input vector (only needed if `x` is a numeric vector)
`from, to`	if NULL (default), analyzes the whole sound, otherwise from...to (s)
`dynamicRange`	regions under `-dynamicRange` dB are treated as silent
`silence`	(0 to 1 as proportion of max amplitude of the anayzed sound) frames with RMS amplitude below `silence * max_ampl adjusted` by `scale` are not analyzed at all
`windowLength`	length of the analysis window, ms
`step`	step between successive windows, ms; if provided, overrides `overlap`; 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 - e.g., 24.98866 instead of 25.0 ms
`overlap`	overlap between successive windows, %
`wn`	window type accepted by `winFun`: character string or function

`zp` window length after zero padding, samples. No padding is performed if `zp` < analysis window length in samples. If NULL, the signal is padded to a good number for speeding up the FFT

`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.

`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` for LPC analysis

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

`roughness` a list of parameters passed to `modulationSpectrum` for measuring roughness and fluctuation strength. 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, 'zc' = zero crossings, 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 accepted - e.g., as returned by `pitch_app`

`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). NULL = no priors used at all; NA = no priors in the first pass, adaptive priors in the second pass if `priorAdapt = TRUE`

`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 to 4 (except for `dom` and `hps`, which return 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

`pitchAutocor` a list of control parameters for pitch tracking using the autocorrelation or "autocor" method

`pitchCep` a list of control parameters for pitch tracking using the cepstrum or "cep" method

`pitchSpec` a list of control parameters for pitch tracking using the BaNa or "spec" method

`pitchHps` a list of control parameters for pitch tracking using the harmonic product spectrum or "hps" method

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

`harmHeight` a list of control parameters for estimating how high harmonics reach in the spectrum

`subh` a list of control parameters for estimating the strength of subharmonics per frame - that is, spectral energy at integer fractions of f0: f0/2, f0/3, etc.

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

`amRange` target range of frequencies for amplitude modulation (`amFreq`, Hz): a vector of length 2, defaults to `c(10, 60)`. Affects both `amMsFreq` and `amEnvFreq`. NB: f0 should not fall into this range, or AM will treat glottal cycles as modulation

`fmRange` target range of frequencies for analyzing frequency modulation (`fmFreq`, Hz): a vector of length 2, defaults to `c(5, 1000 / step / 2)`

`shortestSyl` the smallest length of a voiced segment (ms) that constitutes a voiced syllable (shorter segments will be replaced by NA, as if voiceless)

`shortestPause` the smallest gap between voiced syllables (ms): large value = interpolate and merge, small value = treat as separate syllables separated by a voiceless gap; shortestPause < step disables any gap tolerance (a single voiceless frame terminates the syllable)

`interpolPitch` a list of parameters (currently `win, tol, cert`) for interpolating missing pitch candidates (NULL = no interpolation)

`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

`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 permitted range: '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 `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`

`cores` number of cores for parallel processing

`plot`	if TRUE, produces a spectrogram with pitch contour overlaid
`osc`	"linear" = on the original scale (default); "none" = no oscillogram; "dB" = in decibels
`showLegend`	if TRUE, adds a legend with pitch tracking methods
`savePlots`	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
`embed`	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
`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`	graphical parameters for saving plots passed to `png`
`...`	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
- `interpol` method of interpolating the ACF: "sinc" for maximum precision, "none" for speed

`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
- specMethod "commonFactor" = highest common factor of putative harmonics, "BaNa" = ratio of putative harmonics
- specRatios for method = "commonFactor", the number of harmonics and integer fractions to consider

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)

pitchZc **(zero crossings)**
- zcThres pitch candidates with certainty below this value are treated as noise and set to NA (0 = nothing discarded, 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)

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:

harmHeight **(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

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,amEnvPurity frequency (Hz), purity (0 to 1), and depth (0 to 100) of amplitude modulation estimated from a smoothed amplitude envelope

amMsFreq,amMsPurity frequency (Hz) and purity (dB) 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)

entropyW Wiener entropy of the spectrum of the current frame (=spectral flatness). 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

fluctuation strength of low-frequency modulation at ~4 Hz (0.25-30 Hz), calculated from a modulation spectrum as a complement to psychoacoustic roughness; see `modulationSpectrum`

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 (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 in the roughness range, see `modulationSpectrum` and Anikin 2025

sharpness psychoacoustic sharpness: related to spectral centroid, but calculated from a psychoacoustic loudness model, see `getLoudness`

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

References

Anikin, A. (2025) Acoustic estimation of voice roughness. Attention, Perception, & Psychophysics 87: 1771–1787.

See Also

pitch_app getLoudness segment getRMS

Examples

```
# Detailed documentation: https://cogsci.se/soundgen/acoustic_analysis.html

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
)

# Fancy plotting options:
a = analyze(sound, samplingRate = 44100, plot = TRUE,
  xlab = 'Time, ms', colorTheme = 'seewave', yScale = 'ERB',
  contrast = .5, ylim = c(0.05, 8), 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 = TRUE,  # save the spectrograms with pitch contours
  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]

# 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 pitchMethods = "dom" b/c of likely low-freq noise
  windowLength = 5, step = 2.5,
  shortestSyl = 10, shortestPause = 10,  # again, very short sounds
  interpolPitch = 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

## 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. Use hotkeys (eg spacebar to play/stop) and avoid working with very large files.

Usage

```
annotation_app(...)
```

Arguments

...	presets like `windowLength = 25`. Full list: dynamicRange, specContrast, specBrightness, blur_freq, blur_time, reass_cex, nColors, zp, spec_maxPoints, osc_maxPoints, windowLength, step, spec_xlim, spec_ylim, specType, spec_colorTheme, spec_yScale, osc, wn

Details

Recovering lost data

Every time you add an annotation or move in between files in the queue, the output you've got so far is saved in a temporary backup file. If the app crashes or is closed without saving the results, this backup file preserves your data. To recover it, restart annotation_app() - a dialog box will pop up and ask whether you want to append the old data to the new one. Even so, save your data regularly to be on the safe side!

Value

A list with two elements: `$settings` (a list of the last used settings) and `$annotations` (a data.frame with the annotations). Every time a new annotation is added, the app creates a backup csv file in the session's temporary directory, and it returns the final payload upon closing the app.

See Also

`formant_app`

Examples

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

ann = annotation_app(specType = 'reassigned', windowLength = 5, step = 1)

# full list of parameters that can be passed to annotation_app():
paste0(c(rownames(soundgen:::def_ann),
 'specType', 'spec_colorTheme', 'spec_yScale', 'osc', 'wn', 'audioMethod'),
 collapse = ', ')

# save the complete output, including the settings used
saveRDS(ann, 'my_annotations.rds')

# re-use the same settings in a future session
ann2 = do.call(annotation_app, ann$settings)

# 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_oct`, 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 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 = 10,
  dynamicRange = 80,
  filterType = c("gammatone", "butterworth", "chebyshev"),
  envelope = c("rms", "hil"),
  nFilters_oct = 6,
  nFilters = NULL,
  yScale = c("ERB", "bark", "mel", "log"),
  filterOrder = NULL,
  bandwidth = NULL,
  bandwidthMult = 1,
  minFreq = 20,
  maxFreq = NULL,
  minBandwidth = 10,
 output = c("all", "audSpec", "audSpec_processed", "filterbank", "filterbank_env",
    "filters"),
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  plotFilters = FALSE,
  osc = c("linear", "dB", "none"),
  heights = c(3, 1),
  ylim = NULL,
  contrast = 0,
```

```
  brightness = 0,
  maxPoints = c(1e+05, 5e+05),
  colorTheme = "bw",
  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 to normalize the 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 when envelope = "hil" (ncol of output = length of input audio) and a default of 5 ms when envelope = "rms"
dynamicRange	regions under -dynamicRange dB are treated as silent
filterType	"butterworth" = Butterworth filter (IIR) butter, "chebyshev" = Chebyshev filter (IIR) cheby1, "gammatone" = gammatone filter (FIR)
envelope	the method of computing the envelope of each channel: "rms" = root mean square per window, which is faster but gives limited time resolution (default), "hil" = analytic envelope obtained with a Hilbert transform, low-pass filtered and downsampled unless step = NULL, which is slower but gives the best possible time resolution. As a simple heuristic, you may want to use "hil" if your desired time step is smaller than ~5 ms
nFilters_oct	the approximate number of filters per octave between minFreq and maxFreq; the actual resolution depends on yScale: for instance, if yScale = 'ERB', center frequencies are equally spaced on the ERB scale (set plotFilters = TRUE to check)
nFilters	an alternative way to specify frequency resolution: if specified, overrides nFilters_oct
yScale	determines the location of center frequencies of the filters
filterOrder	filter order (defaults to 4 for gammatones, 3 otherwise)

`bandwidth` filter bandwidth, octaves; if NULL, defaults to ERB bandwidths

`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, which can make the rest of the spectrogram look empty

`minBandwidth` minimum filter bandwidth, Hz (otherwise filters may become too narrow when nFilters is high); only affects Butterworth and Chebyshev filters, not gammatones

`output` character vector specifying which 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 `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`

`cores` number of cores for parallel processing

`plot` if TRUE, produces a plot of the results

`savePlots` if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

`embed` if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

`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 yScale = bark, mel, or ERB

`contrast` controls the sharpness or contrast of the image: <0 = decrease contrast, 0 = no change, >0 increase contrast. Recommended range approximately (-1, 1). The spectrogram is raised to the power of `exp(3 * contrast)`

`brightness` makes the image lighter or darker, range [-1, 1] (default 0 = no change); for colorTheme = "bw", <0 = darker, >0 = lighter, range [-1, 1]. Values are remapped through a smooth sigmoid transfer curve that preserves the full color palette. To lighten or darken the palette itself, change the colors

`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

`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, but nonlinear yScale also warps the contour) that will be plotted over the spectrogram (eg pitch contour); can also be a list with extra graphical parameters such as `lwd`, `col`, `warp` (FALSE = plot as is, TRUE = warp to conform to nonlinear yScale), etc. (see examples)

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

`grid` if numeric, adds n = `grid` dotted lines per kHz and the same number of lines along the time axis

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

`...` other graphical parameters

Value

A list for each analyzed file, including:

audSpec auditory spectrogram: a matrix with frequency in rows (kHz) and time in columns (ms), offset by step/2

audSpec_processed same dimensions, rescaled for plotting (log-transformed, contrast/brightness-adjusted, range 0–1)

filterbank raw filter outputs: a matrix with one row per filter (ordered by center frequency) and one column per audio sample

filterbank_env Hilbert envelopes of the filterbank, same dimensions as `filterbank`; NA if `envelope = "rms"`

filters a dataframe giving the center frequencies, bandwidths, and lower/upper bounds of the used filters, all in Hz

Examples

```
data('speechEx', package = 'soundgen')

# auditory spectrogram
asp = audSpectrogram(speechEx, to = 1, step = 5)
dim(asp$audSpec)

# compare to STFT with similar time and frequency resolution (~100 times faster)
fs = spectrogram(speechEx, to = 1, yScale = 'ERB', windowLength = 5, step = 5)
dim(fs)

## Not run:
# add bells and whistles
audSpectrogram(speechEx,
  nFilters = 128,
  dynamicRange = 150,
  osc = 'none',
```

```
  heights = c(2, 1),  # spectro/osc height ratio
  contrast = .4,  # increase contrast
  brightness = -.2,  # reduce brightness
  colorTheme = 'matlab',  # 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(speechEx, to = 1, nFilters = 15, bandwidth = 1/2)
audSpectrogram(speechEx, to = 1, nFilters = 15, bandwidth = 1/10)
audSpectrogram(speechEx, to = 1, nFilters = 100, bandwidth = 1/2)
audSpectrogram(speechEx, to = 1, nFilters = 100, bandwidth = 1/10)
audSpectrogram(speechEx, to = 1, nFilters_oct = 5, bandwidth = 1/10)
audSpectrogram(speechEx, to = 1, nFilters = 200, bandwidthMult = 1/3)

# caution: if bandwidths are too narrow relative to nFilters, there may be gaps
audSpectrogram(speechEx, to = 1, nFilters = 30, bandwidthMult = 1/3,
  plotFilters = TRUE, plot = FALSE)

# different filter types
audSpectrogram(speechEx, to = 1, filterType = 'gammatone')
audSpectrogram(speechEx, to = 1, filterType = 'butterworth')
audSpectrogram(speechEx, to = 1, filterType = 'chebyshev')

# save auditory spectrograms of all audio files in a folder
audSpectrogram('~/Downloads/temp', savePlots = TRUE, 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 with flexible filter depth (dB) and adjustable bandwidths at filter boundaries. Similar to the "ffilter" function from the seewave package, 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"),
  dB = Inf,
  bw = 0,
  na.rm = TRUE,
  from = NULL,
  to = NULL,
  normalize = FALSE,
  reportEvery = NULL,
  cores = 1,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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 - avoid!), 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 reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
plot	if TRUE, produces a plot of the results
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
width, height, units, res	graphical parameters for saving plots passed to png
...	other graphical parameters passed to plot() and meanSpectrum

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.

Value

Numeric vector of the same length as input.

Examples

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

# low-pass
s1_lp = bandpass(s1, 16000, upr = 2000, bw = 100, plot = TRUE)
meanSpectrum(s1_lp, 16000)

# high-pass by 40 dB
s1_hp = bandpass(s1, 16000, lwr = 2000, dB = 40, bw = 50,
  plot = TRUE, windowLength = 5)
# windowLength is passed to meanSpectrum for plotting (ignore the warnings)

# bandstop
bandpass(s1, 16000, lwr = 1000, upr = 1800, bw = 50, action = 'stop', plot = TRUE)

# bandpass (turns noise into a whistle)
s1_bp = bandpass(s1, 16000, lwr = 2000, upr = 2100, bw = 25, plot = TRUE)
```

```
# playme(rep(s1_bp, 5))
# spectrogram(s1_bp, 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, bw = 1)
points(x_bandp, type = 'l', col = 'blue')

# boost or attenuate a particular frequency band by 20 dB
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 = rnorm(16000)
s5 = bandpass(whiteNoise, 16000, lwr = 30, upr = 40, normalize = TRUE,
              plot = TRUE, xlim = c(0, 0.5))
spectrum(s5, 16000, xlim = c(0, 0.1))
# playme(s5)
# spectrogram(s5, 16000, windowLength = 150, yScale = 'log')

## Not run:
# high-pass all audio files in a folder
bandpass('~/Downloads/temp', lwr = 2000, saveAudio = TRUE, savePlots = TRUE)

## 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 (numeric vector or anchor format, see soundgen)
samplingRate	sampling rate of the output (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

The synthesized waveform as a numeric vector.

See Also

soundgen

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 sounds*

Description

Computes distances between sounds based on comparing their spectrogram-like representations. `compareSounds` takes two sounds or feature matrices as input, whereas `compareFolder` takes a path to a folder with audio files or a list of feature matrices and returns a matrix of pairwise distances between them. Feature matrices are normalized and compared with Dynamic Time Warp (DTW), correlation, cosine distance, or pixel by pixel.

Usage

```
compareSounds(
  x,
  y,
  samplingRate = NULL,
  specFun = "melspec",
  specFun_pars = list(),
  logSpec = FALSE,
  method = c("cor", "cosine", "diff", "dtw"),
  padWith = NA,
  padDir = c("central", "left", "right"),
  dtw_pars = list()
)

compareFolder(
  myfolder = NULL,
  spectrograms = NULL,
  matchAllLengths = FALSE,
  specFun = "melspec",
  specFun_pars = list(),
  logSpec = FALSE,
  method = c("cor", "cosine", "diff", "dtw"),
  padWith = NA,
  padDir = c("central", "left", "right"),
  dtw_pars = list(),
  cores = 1,
  reportEvery = 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. This does not resample audio. For meaningful comparisons, audio inputs should already have the same sampling rate

specFun — the function used to extract a spectrogram-like feature matrix. Can be a string or a custom function that takes audio (numeric vector) as the first argument and returns a spectrogram-like matrix with time in columns and features in rows (see examples). Supported strings:

- stft_simple 'stft' / STFT / 'stft_simple' (amplitude spectrogram) Parameters in specFun_pars: samplingRate (optional if frequency and time labels are not needed), wl (samples), step (samples), wn, zp, padWithSilence.
- spectrogram 'spectrogram' (amplitude spectrogram - a wrapper around stft_simple with more options). Parameters in specFun_pars: see spectrogram.
- powspec 'powerspec' (power spectrogram with tuneR). Parameters in specFun_pars: wintime (s), steptime (s), dither.
- melfcc 'melspec' (mel-spectrogram with tuneR), 'mfcc' / 'melfcc' (MFCCs). Parameters in specFun_pars: windowLength (ms), step (ms), nbands, maxfreq (Hz), MFCC (integer vector).
- audSpectrogram 'audSpectrogram' / 'audSpec' (auditory spectrogram). Parameters in specFun_pars: see audSpectrogram.
- getRMS 'getRMS' / 'rms' (RMS amplitude envelope). Parameters in specFun_pars: see getRMS.
- getEnv 'getEnv' / 'env' (upsampled envelopes: RMS, analytical, peak, etc.). Parameters in specFun_pars: see getEnv.
- spectrum 'spectrum' (short-term spectrum). Parameters in specFun_pars: see spectrum.
- meanSpectrum 'meanSpectrum' / 'meanspec' / 'meanSpec' (long-term average spectrum). Parameters in specFun_pars: see meanSpectrum.
- ssm 'ssm' (self-similarity matrix). Parameters in specFun_pars: see ssm.
- modulationSpectrum 'ms' / 'modulationSpectrum' (modulation spectrum). Parameters in specFun_pars: see modulationSpectrum.

specFun_pars — a list of parameters passed to specFun

logSpec — if TRUE, applies a log transform to the spectrograms before normalization

method — method(s) of comparing spectrograms of two sounds: "cor" = Pearson's correlation distance; "cosine" = cosine distance; "diff" = normalized absolute difference; "dtw" = multivariate Dynamic Time Warp with dtw (NB: the "dtw" package must be installed for this method to work)

padWith — if the durations of x and y are not identical, the compared spectrograms are either padded with silence (padWith = 0) or truncated (padWith = NA) to have the same number of columns. Padding with NA is like truncating the longer sound to the short one's duration, whereas padding with 0 means that the shorter sound is padded with zero to the long one's duration

padDir — if padding, specify where to add zeros or NAs: before the sound ('left'), after the sound ('right'), or on both sides ('central')

dtw_pars — a list of parameters passed to dtw

`myfolder`	path to folder containing audio files to compare
`spectrograms`	a list of spectrogram-like feature matrices to use instead of analyzing the audio - extracted acoustic features, modulation spectra, similarity matrices, ... (overrides `myfolder`)
`matchAllLengths`	if TRUE, all spectrograms are length-matched - e.g., if 100 sounds are compared and `padWith = 0`, 99 of them are padded with silence to match the longest sound (faster); if FALSE, length matching is performed separately for each compared pair. Note that this also affects DTW matching if other distance metrics are used at the same time; thus, you may want to disable `matchAllLengths` if using DTW
`cores`	number of cores for parallel processing
`reportEvery`	when processing multiple inputs, report estimated time left every `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`

Details

If the input is audio, several methods of producing spectrograms are available ("specFun"). For more customized options, just prepare your spectrograms or feature matrices first (time in columns, features like pitch, peak frequency, etc. in rows), and then pass them to `compareSounds` (see examples). All methods except for DTW require that the compared matrices should be of the same size. Compared sounds should ideally have the same sampling rate. If they differ, row (frequency bin) truncation is performed by position, keeping only the first min(nrow) rows, which approximates keeping frequencies up to the lower Nyquist frequency when both spectrograms use the same frequency resolution. In case of differences in duration, the shorter sound is padded with 0 (silence) or NA, as controlled by arguments `padWith, padDir`. If passing custom feature matrices, ensure that they have the same dimensions or that padding with 0 (silence) makes sense.

Value

`compareSounds` returns a dataframe with two columns: "method" for the method(s) used, and "distance" for the distance between the two sounds calculated with that method. The range of distances is `[0, 1]` for "cor", "cosine", and "diff", and `[0, Inf)` for "dtw". `compareFolder` returns a list of distance matrices (`dist` objects), one for each method.

Examples

```
s1 = soundgen(sylLen = 100, pitch = c(80, 180), formants = 'a')
s2 = soundgen(sylLen = 120, pitch = c(150, 350), formants = 'u')
compareSounds(s1, s2, samplingRate = 16000, method = c('cor', 'cosine', 'diff'))
# spectrogram(s1); playme(s1)
# spectrogram(s2); playme(s2)

## Not run:
# NB: install the "dtw" library to run the examples

# compare all sounds in a folder, e.g.:
target = '~/Documents/Research/zz_test_audio/temp_long'
cf = compareFolder(target, logSpec = TRUE)
```

```
mds = as.data.frame(cmdscale(cf$cor))
plot(mds, type = 'n'); text(mds, labels = abbreviate(rownames(mds)))

# or use manually produced spectrograms
sp = spectrogram(target, windowLength = c(10, 40), overlap = 75,
  yScale = 'ERB', output = 'processed', plot = FALSE, cores = 4)
image(sp[[1]])
cf1 = compareFolder(spectrograms = sp)
mds1 = as.data.frame(cmdscale(cf1$cor))
plot(mds1, type = 'n'); text(mds1, labels = abbreviate(rownames(mds1)))

# extract a spectrogram-like representation using a custom function
# (e.g., full-resolution analytic envelopes instead of downsampled RMS)
compareSounds(s1, s2, samplingRate = 16000,
  specFun = function(x) matrix(hilbert_approx(x)$envelope, nrow = 1))

# some more examples
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/end 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, s4, samplingRate = 16000, padWith = 0)
compareSounds(s1, s4, samplingRate = 16000, padWith = NA)

# different types of spectrograms produce quite different results
compareSounds(s1, s3, samplingRate = 16000, specFun = 'stft')
compareSounds(s1, s3, samplingRate = 16000, specFun = 'melspec')
compareSounds(s1, s3, samplingRate = 16000, specFun = 'mfcc')
compareSounds(s1, s3, samplingRate = 16000, specFun = 'audSpec')

# pass additional control parameters to specFun and DTW
compareSounds(s1, s3, samplingRate = 16000,
              specFun = 'melspec',
              specFun_pars = list(nbands = 128),
              dtw_pars = list(dist.method = "Manhattan"))

# use feature matrices instead of spectrograms
# (time in columns, features in rows)
a1 = t(as.matrix(analyze(s1, samplingRate = 16000)$detailed))
a1 = a1[4:nrow(a1), ]; a1[is.na(a1)] = 0  # don't use dur and time stamps
a2 = t(as.matrix(analyze(s2, samplingRate = 16000)$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'))

## End(Not run)
```

---

crossFade *Join two waveforms by cross-fading*

Description

crossFade joins two input vectors (waveforms) by overlapping and cross-fading them. This is standard for joining two separately generated sounds without audible artifacts.

Usage

```
crossFade(
  x,
  y,
  crossLen = 15,
  crossLen_points = NULL,
  samplingRate = NULL,
  shape = c("equalPower", "lin", "exp", "log", "cos", "logistic", "gaussian"),
  cutToZC = FALSE,
  steepness = 1,
  SNR = NULL
)
```

Arguments

x, y	two numeric vectors (waveforms) to be joined
crossLen	overlap in ms
crossLen_points	overlap in points (overrides crossLen)
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: 'equalPower' = equal power (default), 'lin' = linear, 'exp' = exponential, 'log' = logarithmic, 'cos' = cosine, 'logistic' = logistic S-curve, 'gaussian' = Gaussian curve
cutToZC	if TRUE, input vectors are truncated so that x ends with a zero crossing and y starts with a zero crossing, both on an upward portion of the soundwave
steepness	scaling factor regulating the steepness of fading curves (except for shapes 'lin' and 'cos'): 0 = linear, >1 = steeper than default
SNR	if a number, the two sounds are normalized so that the RMS amplitude of x is SNR dB above that of y (NULL = no normalization; negative values make x quieter than y)

Value

A numeric vector with the original sampling rate. The length is going to be `length(x) + length(y) - crossLen_points`, or a bit less if cutting to nearest zero crossings.

See Also

`fade`

Examples

```
sound1 = sin(1:100 / 9)
sound2 = sin(7:106 / 3)
plot(c(sound1, sound2), type = 'b')
# an ugly discontinuity at 100 that will make an audible click

sound = crossFade(sound1, sound2, crossLen_points = 25)
plot(sound, type = 'b') # a nice, smooth transition
length(sound) # but note that cross-fading costs us crossLen_points = 25
# points even without cutting to zc, and much more if we trim to
# zero crossings and then overlap

sound = crossFade(sound1, sound2, crossLen_points = 25,
  SNR = 6)  # make sound1 6 dB louder (RMS) than sound2
plot(sound, type = 'b') # a nice, smooth transition

# Actual sounds, alternative shapes of fade-in/out
sound3 = soundgen(formants = 'a', pitch = 200, sylLen = 300,
                  addSilence = 0, attackLen = c(50, 0))
sound4 = soundgen(formants = 'u', pitch = 200, sylLen = 300,
                  addSilence = 0, attackLen = c(0, 50))

# simple concatenation (with a click)
cf1 = c(sound3, sound4)
osc(cf1, 16000, from = .25, to = .35)
# playme(cf1, 16000)

# concatenation from zc to zc (no click, but a rough transition)
cf2 = crossFade(sound3, sound4, crossLen_points = 0, cutToZC = TRUE)
osc(cf2, 16000, from = .25, to = .35)
# playme(cf2, 16000)

# crossFade over 35 ms - brief, but smooth
cf3 = crossFade(sound3, sound4, crossLen = 35, samplingRate = 16000)
osc(cf3, 16000, from = .25, to = .35)
# playme(cf3, 16000)

# equal-power cross-fade over 300 ms (shortens the sound by ~300 ms)
cf4 = crossFade(sound3, sound4, samplingRate = 16000,
                 crossLen = 300, shape = 'equalPower')
osc(cf4, 16000)
# playme(cf4, 16000)
```

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"), sidebands / 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 clumping to account for the fact that NLP typically occur in continuous segments spanning multiple frames.

Usage

```
detectNLP(
  x,
  samplingRate = NULL,
  predictors = NULL,
  thresProb = 0.4,
  voicelessToNone = FALSE,
  train = soundgen::detectNLP_training_nonv,
  scale = NULL,
  from = NULL,
  to = NULL,
  pitchManual = NULL,
 analyze_pars = list(windowLength = 50, roughness = list(msType = "1D", specMethod =
    "spectrum", specFun_pars = list(windowLength = 25, step = 2), amRes = NULL,
    roughRange = NULL, roughMean = 100, roughSD = 8), amRange = c(20, 150)),
  phasegram_pars = list(nonlinStats = "d2"),
  naiveBayes_pars = list(prior = "static", wlClumper = 3),
  jumpThres = 14,
  jumpWindow = 100,
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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. If NULL (default), uses the standard set of predictors if present in the training corpus: d2, subDep, amEnvDep, amMsPurity, entropyW, entropySh, HNR, CPP, roughness. If specified, uses all predictors as long as they are available 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)
`voicelessToNone`	if TRUE, frames treated as voiceless 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 to normalize the 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 accepted - e.g., as returned by `pitch_app`
`analyze_pars`	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")
`phasegram_pars`	arguments passed to `phasegram`. NB: only phasegram descriptives requested here are used for NLP detection; other nonlinear statistics are not calculated to save time.
`naiveBayes_pars`	arguments passed to `naiveBayes`. It is strongly recommended to use some clumping, with `wlClumper` given as frames (multiply by `step` to get the corresponding minimum 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 reportEvery iterations (NULL = default, NA = don't report); see reportTime

cores — number of cores for parallel processing

plot — if TRUE, produces a spectrogram with annotated NLP regimes

savePlots — if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed — if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

main, xlab, ylab, ... — graphical parameters passed to spectrogram

ylim — frequency range to plot, kHz

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

Value

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. The time step is equal to the larger of the steps passed to analyze() and phasegram().

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,
  plot = TRUE, ylim = c(0, 4))

# classifier trained on synthetic, soundgen()-generated sounds
nlp = detectNLP(target, 16000,
  train = soundgen::detectNLP_training_synth,
```

```
  plot = TRUE, ylim = c(0, 4))
head(nlp[, c('time', 'pr')])
table(nlp$pr)
plot(nlp$amEnvDep, 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))

## End(Not run)
```

`detectNLP_training_nonv`

Training datasets for NLP detection

Description

The results of running `naiveBayes_train` on acoustically analyzed 969 human nonverbal vocalizations (detectNLP_training_nonv) and on 5000 synthetic sounds with or without NLP created with soundgen (detectNLP_training_synth). Used by `detectNLP`.

Usage

```
detectNLP_training_nonv

detectNLP_training_synth
```

Format

An object of class `list` of length 12.

An object of class `list` of length 12.

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 $(2 * n - 1) * speedSound/(4 * Fn)$ for a closed-open tube (mouth open) and $n * speedSound/(2 * Fn)$ 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) and Anikin et al. (2024). For a review of VTL-related summary measures of formant frequencies, refer to Pisanski et al. (2014). See also `schwa` for VTL estimation with additional information on formant frequencies.

Usage

```
estimateVTL(
  formants,
  method = c("regression", "meanDispersion", "meanFormant"),
  interceptZero = TRUE,
  tube = c("closed-open", "open-open", "open-closed", "closed-closed"),
  speedSound = 35400,
  checkFormat = TRUE,
  output = c("simple", "detailed"),
  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 Description): "regression" (default), "meanDispersion", or "meanFormant"
`interceptZero`	if TRUE, forces the regression curve to pass through the origin, reducing the influence of highly variable lower formants (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 (for method = "regression" only)

`plot` if TRUE, plots the regression line whose slope gives formant dispersion dF (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.

References

- Reby, D., McComb, K., Cargnelutti, B., Darwin, C., Fitch, W. T., & Clutton-Brock, T. (2005). Red deer stags use formants as assessment cues during intrasexual agonistic interactions. Proceedings of the Royal Society B: Biological Sciences, 272(1566), 941-947.
- Pisanski et al. (2014). Vocal indicators of body size in men and women: a meta-analysis. Animal Behaviour, 95, 89-99.
- Anikin, A., Barreda, S. & Reby, D. (2024) A practical guide to calculating vocal tract length and scale-invariant formant patterns. Behavior Research Methods 56, 5588–5604.

See Also

schwa

Examples

```
estimateVTL(NA)
estimateVTL(500)
estimateVTL(c(600, 1850, 2800, 3600, 5000), plot = TRUE)
es = estimateVTL(c(600, 1850, 2800, 3600, 5000), plot = TRUE, output = 'detailed')
es

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

# Compare the results produced by the three methods
nIter = 100
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
```

`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,
  samplingRate = NULL,
  scale = NULL,
  shape = c("lin", "exp", "log", "cos", "logistic", "gaussian"),
  steepness = 1,
  reportEvery = NULL,
  cores = 1,
```

```
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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)
samplingRate	sampling rate of x (only needed if x is a numeric vector)
scale	maximum possible amplitude of input, used to normalize the 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, 'gaussian' = Gaussian 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 reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
plot	if TRUE, produces an oscillogram of the waveform after fading
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
width, height, units, res	graphical parameters for saving plots passed to png

Value

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(attackLen = 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(4000, min = -1, max = 1)  # make sure to zero-center input!!!
# plot(x, type = 'l')
y = fade(x, samplingRate = 1000, fadeIn = 1000, fadeOut = 0, plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1000, fadeOut = 1500,
         shape = 'exp', steepness = 1, plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1500, fadeOut = 500,
         shape = 'log', steepness = 1, plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1500, fadeOut = 500,
         shape = 'log', steepness = 3, plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1500, fadeOut = 1500,
         shape = 'cos', plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1500, fadeOut = 1500,
         shape = 'logistic', steepness = 1, plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1500, fadeOut = 1500,
         shape = 'logistic', steepness = 3, plot = TRUE)
y = fade(x, samplingRate = 1000, fadeIn = 1500, fadeOut = 1500,
         shape = 'gaussian', steepness = 1.5, plot = TRUE)

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

## End(Not run)
```

---

filterSoundByMS *Filter sound by modulation spectrum*

Description

Manipulates the modulation spectrum (MS) of a sound to attenuate or amplify 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 back to a spectrogram, 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 = TRUE,
  windowLength = 25,
  step = NULL,
  overlap = 80,
  wn = "hanning",
  amCond = NULL,
  fmCond = NULL,
  jointCond = NULL,
  action = c("stop", "pass"),
  dynamicRange = 80,
  dB = Inf,
  smooth = NULL,
  protect = c("level", "meanSpectrum"),
  warnAsymmetric = TRUE,
  initialPhase = c("orig", "spsi", "random", "zero"),
  nIter = 50,
  reportEvery = NULL,
  cores = 1,
  play = FALSE,
  saveAudio = FALSE,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  width = 900,
  height = 500,
  units = "px",
  res = NA
)

filterMS(
  ms,
  amCond = NULL,
  fmCond = NULL,
  jointCond = NULL,
  action = c("stop", "pass"),
  dB = Inf,
  smooth = NULL,
  protect = c("level", "meanSpectrum"),
  warnAsymmetric = TRUE,
  plot = TRUE
)
```

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-like representation is log-transformed prior to estimating the modulation spectrum. This applies to both 2D and 1D modulation spectra. A small constant is added to avoid non-finite values
`windowLength`, `step`, `overlap`, `wn`	STFT parameters used to create the original spectrogram; make sure `zp = 0`
`amCond`, `fmCond`	condition on amplitude and frequency modulation: character string, expression, or function (see examples)
`jointCond`	character string, expression, or function with a joint condition on am and fm
`action`	'stop' or 'pass'
`dynamicRange`	the minimum possible value of the magnitude spectrogram (dB, must be positive or NA). 80 (default) = 80 dB below the global maximum, NA or Inf = no floor (allow zero values). Lowering the dynamic range (e.g., to 60 dB) attenuates ripple effects caused by filtering, but adds some uniform residual noise at the floor level
`dB`	a positive number giving the strength of effect in dB (defaults to Inf - complete removal of selected frequencies)
`smooth`	smoothing of the MS filter mask. NULL = default smoothing (~3 bins); NA = no smoothing; numeric vector of length 2 = smoothing bandwidth for FM and AM in native units (FM: cycles/kHz, AM: Hz). A single number is not allowed because FM and AM have different units
`protect`	which zero-modulation components to protect from filtering: 'level' = the (0, 0) cell (overall mean level / gain); 'meanSpectrum' = the (am = 0) column (time-averaged spectrum); 'envelope' = the (fm = 0) row (broadband temporal envelope, including any AM). TRUE protects all three; FALSE, NULL or NA protects none. NB: protecting 'envelope' re-injects temporal modulation when filtering AM; protecting 'meanSpectrum' preserves static spectral modulation when filtering FM
`warnAsymmetric`	if TRUE, warns if the MS filter mask is asymmetric with respect to zero AM/FM. Asymmetric filters can produce complex-valued inverse transforms and phase-related artifacts
`initialPhase`	initial phase estimate: "orig" (default) = phase of the original sound; "spsi" = single-pass spectrogram inversion (Beauregard et al., 2015); "random" = uniformly distributed noise; "zero" = all phases zero. If `nIter = 0` and `initialPhase = "orig"`, the original phase is used directly
`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 `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`

`cores`	number of cores for parallel processing
`play`	if TRUE, plays back the reconstructed audio
`saveAudio`	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
`plot`	if TRUE, produces a triple plot: original MS, filtered MS, and the MS of the output sound
`savePlots`	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
`embed`	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
`width`, `height`, `units`, `res`	parameters passed to `png` if the plot is saved
`ms`	a modulation spectrum: matrix of real or complex values, AM in columns (Hz), FM in rows (cycles/kHz)

Details

By default, the filter mask is smoothed to avoid abrupt cutoffs. Smoothing is specified in physical units: FM in cycles/kHz and AM in Hz. Set `smooth = NA` to disable smoothing. If `logSpec = TRUE`, filtering is performed on the modulation spectrum of the log-magnitude spectrogram, which is often more robust and more physiologically plausible than filtering linear magnitudes. If `initialPhase = 'orig'`, the phase of the original sound is used. With `nIter = 0`, the original phase is used directly; with `nIter > 0`, the original phase is used to initialize the Griffin-Lim iterations.

Value

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

Choosing filtering settings

MS filtering operates globally on the spectrogram's envelope, so any transient whose modulation content overlaps the filtered band — onsets/offsets, plosives, amplitude steps — is partially removed together with the target, producing a ripple that decays inward from the transient. Smoothing (blurring the filter mask before applying it) shortens the ripple's tails but cannot eliminate it completely. Two parameters control the trade-off between successful filtering and introducing spectro-temporal artifacts: (1) `dynamicRange`: lower values shrink the silence-to-sound step in the log domain and lift the linear target above the zero-clipping knee, suppressing edge ripple at the cost of a uniform residual noise floor, most visible in silent regions, and (2) `dB`: finite attenuation, e.g. 20–40 dB, removes proportionally less of the transients' in-band content than dB = Inf. Use `dB = Inf` with a large `dynamicRange` for steady, sustained sounds and finite dB with dynamicRange = 60–80 for sounds with transients, such as speech. Use `logSpec = TRUE` for manipulating the temporal envelope /

AM in a perceptually meaningful manner. `logSpec = FALSE` weights components by absolute magnitude and avoids amplifying quiet regions, but shows stronger edge artifacts. Choose `protect = c('level','meanSpectrum')` (the default setting) when filtering AM. Do not protect `'envelope'`, which contains the very AM being removed. Set `protect = c('level', 'envelope')` when filtering FM, so as to keep the natural amplitude contour. Drop `'meanSpectrum'` only if you also want to flatten the static spectral shape. Whenever the condition includes 0 on an axis (e.g. removing slow AM with abs(am) < 3), keep the corresponding protection to avoid a collapsed, "AC-coupled" reconstruction. For mild filtering, `initialPhase = 'orig'` with `nIter = 0` is fast and preserves the original phase; increase `nIter` for stronger filtering.

See Also

invertSpectrogram

Examples

```
# Create a sound to be filtered
s = soundgen(sylLen = 500, samplingRate = 16000,
    amFreq = 25, amDep = 50,
    addSilence = 50, plot = TRUE)
# playme(s, 16000)

# Filter to remove the rapid AM at 25 Hz
s_filt = fade(filterSoundByMS(s, samplingRate = 16000,
  amCond = 'abs(am) > 22 & abs(am) < 28',
  action = 'stop',
  plot = TRUE))
# NB: clicks are often introduced at the onset and offset - fade and renormalize
s_filt = s_filt / max(abs(s_filt))
# playme(s_filt, samplingRate = 16000)
spectrogram(s_filt, 16000, windowLength = 25)

# Using filterMS()
ms = modulationSpectrum(s, 16000, returnComplex = TRUE)$detailed$complex

# Remove all AM over 25 Hz
filterMS(ms, amCond = 'abs(am) > 25', protect = FALSE)
filterMS(ms, amCond = 'abs(am) > 25', protect = TRUE)
filterMS(ms, amCond = 'abs(am) > 25', protect = TRUE, smooth = NA)

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

# jointCond is more flexible
filterMS(ms, jointCond = 'am * fm < 5', action = 'stop')
filterMS(ms, jointCond = 'am^2 + (fm*3)^2 < 200', action = 'pass')

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

```
filterMS(ms, amCond = 'abs(am) > 5', fmCond = 'abs(fm) < 5')  # faster

## Not run:
data('speechEx', package = 'soundgen')
samplingRate = speechEx@samp.rate
playme(speechEx)
spectrogram(speechEx)

# Remove AM above 3 Hz from a bit of speech (removes most temporal details)
s_filt1 = fade(filterSoundByMS(speechEx, amCond = 'abs(am) > 3',
                          action = 'stop',
                          smooth = c(0.5, 2)))
spectrogram(s_filt1, samplingRate = samplingRate)
playme(s_filt1, samplingRate)

# Remove FM above 3 cycles/kHz (masks f0 but preserves formants)
s_filt3 = fade(filterSoundByMS(speechEx,
  fmCond = 'abs(fm) > 3', action = 'stop',
  logSpec = TRUE, smooth = c(1, 5),
  protect = c('level', 'envelope')))
spectrogram(s_filt3, samplingRate = samplingRate)
playme(s_filt3, samplingRate)

# Joint spectro-temporal filtering with a symmetric condition
s_filt4 = fade(filterSoundByMS(speechEx,
  jointCond = 'am^2 + (fm * 3)^2 < 300', action = 'stop',
  logSpec = TRUE, dynamicRange = 60, smooth = c(1, 10),
  protect = c('level', 'meanSpectrum')))
spectrogram(s_filt4, samplingRate = samplingRate)
playme(s_filt4, samplingRate)

# Process all files in a folder, save filtered audio and plots
s_filt = filterSoundByMS('~/Downloads/temp2',
  saveAudio = TRUE, savePlots = TRUE,
  amCond = 'abs(am) > 15', fmCond = 'abs(fm) > 5',
  action = 'stop', smooth = c(0.5, 5),
  nIter = 10)

## End(Not run)
```

`findPeaks` *Find peaks*

Description

A bare-bones, very fast function to find local maxima (peaks) in a numeric vector. NAs are set to the minimum value of input and effectively ignored. The first and last values can also be peaks. Plateaus are NOT treated as peaks.

Usage

```
findPeaks(
  x,
  wl = 3,
  thres = NULL,
  thresRel = NULL,
  dropFirstLast = FALSE,
  plot = FALSE
)
```

Arguments

x	numeric vector
wl	rolling window over which we look for maxima: central value ± floor(wl/2), eg ±1 if wl=3; forced to be at least 3
thres	required absolute value of each peak
thresRel	required absolute excess over the window mean excluding the focal point
dropFirstLast	if TRUE, the first and last value cannot be peaks (only two-sided maxima accepted)
plot	if TRUE, plots the input vector with peaks marked.

Value

A vector of indices of local maxima.

Examples

```
x = c(5, 2, 3, 3, 2, 7, 3, 2, 5, 5, 1, 0, 2, NA)
findPeaks(x, wl = 3, plot = TRUE)
findPeaks(x, wl = 3, thres = 2, plot = TRUE)
findPeaks(x, wl = 3, thres = 100, plot = TRUE)
findPeaks(x, wl = 3, plot = TRUE)
findPeaks(x, wl = 6, plot = TRUE)
findPeaks(x, wl = 3, thresRel = 2, plot = TRUE)
findPeaks(x, wl = 3, thresRel = 4, plot = TRUE)
findPeaks(x, wl = 3, thres = 6, thresRel = 2, plot = TRUE)
findPeaks(x, dropFirstLast = TRUE)
```

---

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"),
  windowLength = 50,
  killDC = FALSE,
  dynamicRange = 40,
  reportEvery = NULL,
  cores = 1,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  col = "blue",
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  ...
)

compressor(
  x,
  samplingRate = NULL,
  scale = NULL,
  compression = 1,
  method = c("hil", "rms", "peak"),
  windowLength = 50,
  killDC = FALSE,
  dynamicRange = 40,
  reportEvery = NULL,
  cores = 1,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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 to normalize the 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
`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 `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`
`cores`	number of cores for parallel processing
`saveAudio`	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
`plot`	if TRUE, plots the original sound, the smoothed envelope, and the compressed sound
`savePlots`	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
`embed`	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
`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.

See Also

`transplantEnv`

Examples

```
a = rnorm(500) * seq(1, 0, length.out = 500)
b = flatEnv(a, 1000, plot = TRUE, windowLength = 5)    # too short
c = flatEnv(a, 1000, plot = TRUE, windowLength = 450)  # too long
d = flatEnv(a, 1000, plot = TRUE, windowLength = 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 = 50, killDC = FALSE)
s3 = flatEnv(s1, 16000, plot = TRUE, windowLength = 50, killDC = TRUE)

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

## 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(...)
```

Arguments

`...`	presets like `windowLength = 25`. Full list: samplingRate_mult, nFormants, minformant, maxbw, dynamicRange_lpc, zp_lpc, spec_ylim, dynamicRange, specContrast, specBrightness, blur_freq, blur_time, reass_cex, zp, spec_maxPoints,

osc_maxPoints, spectrum_smooth, spectrum_xlim, spectrum_len, silence, windowLength_lpc, step_lpc, windowLength, step, spec_colorTheme, osc, wn, wn_lpc, vtl_method, speedSound, coeffs, interceptZero, tube, nColors, spec_cex, pitch, summaryFun, audioMethod, specType, fmtSpacePlot, spec_col, normalizeInput, adaptivePitch, spectrum_plotSynth

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 three ways to adjust them: (1) click the spectrogram within selection (pick the formant number to adjust by clicking the formant boxes); (2) single-click the spectrum to use the cursor's position, or (3) 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.

Recovering lost data

Every time you add an annotation or move in between files in the queue, the output you've got so far is saved in a temporary backup file. If the app crashes or is closed without saving the results, this backup file preserves your data. To recover it, restart formant_app() - a dialog box will pop up and ask whether you want to append the old data to the new one. Even so, save your data regularly to be on the safe side!

Value

A list of the last used settings ($settings) plus a data.frame with the formant measurements. Every time a new annotation is added, the app creates a backup csv file in the session's temporary directory, and it returns the final payload upon closing the app.

See Also

pitch_app annotation_app

Examples

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

f1 = formant_app(specType = 'reassigned', windowLength = 5, step = 1)

# run the app with previously used settings
f2 = do.call(formant_app, f1$settings)

# save the complete output, including the settings used
saveRDS(f2, 'my_formant_analysis.rds')

# 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 smooths it by convolution with a symmetric Gaussian window function. Values outside the matrix are either treated as zero, attenuating the edges, or assumed to continue beyond the edges.

Usage

```
gaussianSmooth2D(
  m,
  kernelSize = 5,
  kernelSD = 0.5,
  action = c("blur", "unblur"),
  amount = 0.5,
  padWith = c("repeat", "zero"),
  plotKernel = FALSE
)
```

Arguments

m	input matrix (numeric, on any scale, doesn't have to be square)
kernelSize	vector of size 1 or 2: the size of the Gaussian kernel, in points (forced to odd values). kernelSize = 0 means no smoothing along that dimension.
kernelSD	the SD of the Gaussian kernel evaluated over [-1, 1]: for ex., if kernelSD = 0.5, the kernel spans approximately ±2 SDs
action	'blur' = kernel-weighted average, 'unblur' = unsharp masking
amount	the amount of residual to mix with the original when unblurring: $result = orig + amount * (orig - blurred)$
padWith	how to treat the edges of the matrix: 'repeat' = the edge rows/columns are assumed to continue beyond the matrix; 'zero' = values outside the matrix are assumed to be zero, attenuating the smoothed edges
plotKernel	if TRUE, plots the kernel

Value

A numeric matrix of the same dimensions as input.

See Also

modulationSpectrum spectrogram

Examples

```
data('speechEx', package = 'soundgen')
s = spectrogram(speechEx, from = 0, to = 1, windowLength = 10,
  output = 'original', plot = FALSE)
s = log(s + .001)
image(t(s))
s1 = gaussianSmooth2D(s, kernelSize = 5, plotKernel = TRUE)
image(t(s1))

# more smoothing in time than in frequency
s2 = gaussianSmooth2D(s, kernelSize = c(5, 15))
image(t(s2))

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

# smoothing only in one dimension
s4 = gaussianSmooth2D(s, kernelSize = c(25, 0))
image(t(s4))
s5 = gaussianSmooth2D(s, kernelSize = c(0, 15))
image(t(s5))

# sharpen the image
s6 = gaussianSmooth2D(s, kernelSize = 5, action = 'unblur', amount = .5)
image(t(s6))
```

---

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 pure silence.

Usage

```
getDuration(
  x,
  samplingRate = NULL,
  silence = 0.01,
  rms_pars = 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_pars	a list of control parameters passed to getRMS
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing

Value

For a single file, a one-row data frame with columns $duration and $duration_noSilence. For multiple files, a data frame with one row per file, including a $file column. If silence = NULL, duration_noSilence (duration without leading and trailing silence) is NA. If no RMS frame exceeds the threshold, duration_noSilence is 0.

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_pars = list(windowLength = 5, step = 1))

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

## End(Not run)
```

---

getEntropy *Entropy*

Description

Returns Wiener 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("wiener", "shannon"),
  normalize = FALSE,
  convertNonPositive = 1e-10
)
```

Arguments

`x`	vector of positive floats
`type`	'shannon' for Shannon (information) entropy, 'wiener' for Wiener entropy
`normalize`	if TRUE, Shannon entropy is normalized by the length of input vector to range from 0 to 1. It has no effect on Wiener entropy
`convertNonPositive`	occasional non-positive values are converted to `convertNonPositive` (if all inputs are non-positive, entropy = NA)

Value

The entropy of input vector as a single number.

Examples

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

# Wiener entropy is ~0 for periodic, NA for silent, 1 for white noise
lapply(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
lapply(s, function(x) round(getEntropy(x, type = 'shannon'), 2))

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

getEnv *Get amplitude envelope*

Description

Calculates a smoothed envelope of a waveform based on peaks, running average, root mean square (intensity), or envelope of the analytic signal with Hilbert transform. Algorithm: calculates one envelope value per frame of length `wl`, then upsamples to the original sampling rate with `spline`.

Usage

```
getEnv(
  x,
  method = c("rms", "hil", "peak", "mean"),
  wl = 200,
  overlap = 0,
  step = NULL,
  upsample = TRUE
)
```

Arguments

`x`	numeric vector at least `wl` long
`method`	"peak" for peak amplitude per window, "rms" for root mean square amplitude, "mean" for mean (for DC offset removal), "hil" for Hilbert envelope
`wl`	the length of smoothing window (samples)
`overlap`	overlap between successive windows, 0 to 100%
`step`	step between successive windows (samples); overrides overlap
`upsample`	if TRUE, upsamples the envelope to the length of the original sound; if FALSE, returns one sample per window

Value

The envelope as a numeric vector on the original scale. If `upsample = TRUE`, it has the same length as the input, regardless of the amount of smoothing.

Examples

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

```
# No upsampling (short output)
getEnv(1:16, 'mean', wl = 5, upsample = FALSE)
env_short = getEnv(a, 'rms', wl = wl, overlap = 50, upsample = FALSE)
plot(a, type = 'l')
lines(seq(1, length(a), length.out = length(env_short)), env_short, col = 'red')
```

---

getFormantFilter *Formant filter*

Description

Prepares a frequency-domain filter for adding formants to a sound. Formants are specified as a list containing time, frequency, amplitude, and width values for each vocal tract resonance (see examples). For more information, see https://cogsci.se/soundgen/sound_generation.html.

Usage

```
getFormantFilter(
  nr,
  nc,
  formants = NA,
  formantDep = 1,
  formantWidth = 1,
  lipRad = 6,
  noseRad = 4,
  mouth = NA,
  mouthOpenThres = 0,
  openMouthBoost = 0,
  vocalTract = NULL,
  temperature = 0.025,
  formDrift = 0.3,
  formDisp = 0.2,
  formantDepStoch = 1,
  formantCeiling = NULL,
  samplingRate = 16000,
  speedSound = 35400,
  smoothing = list(interpol = "splineFC"),
  output = c("simple", "detailed"),
  plot = FALSE,
  duration = NULL,
  colorTheme = "bw",
  col = NULL,
  xlab = "Time",
  ylab = "Frequency, kHz",
  ...
)
```

Arguments

`nr`	number of frequency bins (half the FFT window size)
`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 if `temperature > 0` and `vocalTract` is specified. Time stamps for formants and mouth opening can be specified in ms or 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`	the amplitude of additional stochastic formants added above the highest specified formant, dB (only if temperature > 0)
`formantCeiling`	frequency to which stochastic formants are calculated to avoid losing energy in the upper part of the spectrum due to unmodeled resonances above the Nyquist frequencies, specified in multiples of Nyquist. If NULL (default), a faster theoretical correction is used (may fail for unusual sounds)
`samplingRate`	sampling rate of the output (Hz)
`speedSound`	speed of sound in warm air, cm/s. Stevens (2000) "Acoustic phonetics", p. 138
`smoothing`	list of parameters passed to soundgen:::getSmoothContour
`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 and column names in the output)
colorTheme	black and white ('bw'), as in seewave package ('seewave'), or another color theme (e.g. 'heat.colors')
col	actual colors, e.g., 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

Spectral filter on a linear scale (not dB): 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 = getFormantFilter(nr = 512, nc = 50, duration = 300,
  formants = '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 = getFormantFilter(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 = getFormantFilter(nr = 512, nc = 50, 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 = getFormantFilter(nr = 512, nc = 1, 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 = getFormantFilter(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 = getFormantFilter(nr = 512, nc = 1,
  formants = 'a', formantWidth = 1, formantDep = 1)  # defaults
e2 = getFormantFilter(nr = 512, nc = 1,
  formants = '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 = getFormantFilter(
  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 = getFormantFilter(
  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)
```

`getLoudness` *Get loudness*

Description

Estimates subjective loudness and sharpness of audio. 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,
  input = c("spec", "audSpec"),
  windowLength = 50,
  step = NULL,
  overlap = 50,
  SPL_measured = 70,
  spreadSpectrum = FALSE,
  sharpnessMethod = c("aures", "DIN45692", "bismarck"),
  summaryFun = c("mean", "median", "sd"),
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  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 to normalize the input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
input	"spec" = power spectrogram warped to bark scale with audspec, "audSpec" = auditory spectrogram produced by convolving the signal with a bank of gammatone filters using audSpectrogram (much slower, but more physiologically accurate)
windowLength	length of the analysis window, ms
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
SPL_measured	sound pressure level at which the sound is presented relative to some reference (the conventional threshold is 2e-5 Pa), dB
spreadSpectrum	if TRUE, applies a spreading function to account for frequency masking

`sharpnessMethod`
: the method of calculating sharpness (mostly differ in weighting functions; only "aures" depends on SPL)

`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 `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`

`cores`
: number of cores for parallel processing

`plot`
: if TRUE, produces a plot of the results

`savePlots`
: if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

`embed`
: if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

`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 frequencies on the bark scale, optionally 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 and `spreadSpectrum = FALSE`. Sharpness is calculated as the weighted first moment of specific loudness on the Bark scale.

Value

A list with two top-level elements: `$detailed` and `$summary`.

`$detailed` contains per-file results. If multiple sounds are analyzed, `$detailed` is a list of per-sound lists. If a single sound is analyzed, it is simplified to a single list. Each list contains:

loudness a vector of loudness in sone units per STFT frame

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

loudnessPhon, specPhon same in phon instead of sone units

sharpness a vector of sharpness in acum units per STFT frame

audSpec auditory spectrogram used to calculate loudness and sharpness

`$summary` is a dataframe of summary loudness measures (one row per file). If `summaryFun` is `NULL`, `$summary` is `NULL`.

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(
  noise_1KHz = soundgen:::zeroOne(bandpass(rnorm(8000), 16000,
    lwr = 900, upr = 1100)) * 2 -1, # narrow-band noise at 1 KHz
  white_noise = runif(8000, -1, 1),  # white noise
  white_noise2 = runif(8000, -1/2, 1/2),  # ~6 dB quieter
  pure_tone_1KHz = sin(2*pi*1000/16000*(1:8000)),  # pure tone at 1 kHz
  pure_tone_100Hz = sin(2*pi*100/16000*(1:8000))  # pure tone at 100 Hz
)
# playme(sounds)
l = getLoudness(
    x = sounds, samplingRate = 16000, scale = 1,
    windowLength = 40, step = NULL, input = c('spec', 'audSpec')[1],
    overlap = 50, SPL_measured = 60,
    plot = FALSE)
l$summary
# loudness depends on amplitude if "scale" is provided (cf. sounds 2 and 3)
# narrowband noise / tone at 1 kHz, 60 dB: sharpness ~=1 acum, loudness ~=4 sone

# a steady glissando from 125 to 8000 Hz (constant on a musical scale)
pitch = exp(seq(log(125), log(8000), length.out = 16000))
s = sinpi(2 * cumsum(pitch) / 16000)
l1 = getLoudness(s, samplingRate = 16000, SPL_measured = 70)
# steady SPL, but variable loudness

# The estimated loudness and sharpness depend on target SPL
l2 = getLoudness(s, samplingRate = 16000, SPL_measured = 40, plot = FALSE)
l1$summary$loudness_mean
l2$summary$loudness_mean
```

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

## Not run:
# Using auditory spectrogram as input instead of STFT (slower)
l4 = getLoudness(s, samplingRate = 16000, SPL_measured = 40, input = 'audSpec')
l4$summary$loudness_mean

# Process all audio files in a folder
l5 = getLoudness('~/Downloads/temp', savePlots = TRUE)
l5$summary

## End(Not run)
```

---

getRMS *RMS amplitude*

Description

Calculates root mean square (RMS) amplitude in overlapping windows, providing an envelope 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"),
  killDC = FALSE,
  normalize = TRUE,
  windowDC = 200,
  summaryFun = "mean",
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
```

```
  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 to normalize the 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 analysis window, ms (longer windows = more smoothing)
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
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, removes 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, 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 reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
plot	if TRUE, plot a contour of RMS amplitude
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
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

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 = TRUE)
r$summary

# Compare:
analyze('~/Downloads/temp', pitchMethods = NULL,
        plot = FALSE)$summary$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` *Rolloff of harmonics in source spectrum*

Description

A periodic sound often contains many harmonics of the fundamental frequency (f0). For instance, soundgen generates voice source as a combination of pure tones - one for each partial in the source spectrum. Harmonics typically become weaker at higher frequencies, and the rolloff (roll-off) function describes the loss of energy in upper harmonics relative to f0. `getRolloff` provides flexible control over this rolloff function: simple exponential (`rolloff`), `rolloffOct` that varies per octave, and `rolloffKHz` for rolloff correction depending on f0. It calculates the rolloff function based on these parameters and returns the amplitude of each harmonic.

Usage

```
getRolloff(
  pitch_per_gc = c(440),
  nHarmonics = 1000,
  rolloff = -6,
  rolloffOct = 0,
  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 or partials to generate, including f0 (all harmonics above Nyquist frequency or with amplitude < `-dynamicRange` are discarded)
- `rolloff` — the rate at which f0 harmonics in the spectrum become weaker, dB/oct (anchor format for all rolloff-related parameters). More negative rolloff = weaker upper harmonics
- `rolloffOct`, `rolloffKHz` — rolloff may be constant throughout the spectrum, or it may vary with each octave above f0 (rolloffOct) or per kHz increase in f0 above the baseline of 200 Hz (rolloffKHz); for both parameters, positive values mean the rate of rolloff increases toward upper frequencies)
- `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

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
rolloff1 = getRolloff(pitch_per_gc = 150, rolloff = -12,
  rolloffOct = 0, rolloffKHz = 0, plot = TRUE)
# the rate of rolloff slows down by 1 dB each octave
r = getRolloff(pitch_per_gc = 100, rolloff = -6, rolloffOct = 0)
r1 = getRolloff(pitch_per_gc = 100, rolloff = -6, rolloffOct = -1)
round(20*log10(r[c(1, 2, 4, 8, 16)]), 1)
round(20*log10(r1[c(1, 2, 4, 8, 16)]), 1)

# 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)
```

```
# dynamic rolloff (varies over time)
rolloff = getRolloff(pitch_per_gc = c(150, 250),
                     rolloff = c(-12, -18, -24), plot = TRUE)
```

---

getSpecEnv *Get spectral envelope*

Description

Calculates a smoothed envelope of a magnitude spectrum or of each column of a spectrogram. This is good for removing the fine structure produced by harmonics of f0 and leaving only the overall spectral contour produced by resonances (formants). This is the source-filter separation step used by `shiftFormants`. All methods except "peak" smooth the log-magnitude spectrum and return the envelope on the original linear scale, with the same dimensions as the input.

Usage

```
getSpecEnv(
  spec,
  freqs = NULL,
  freqWindow = NULL,
  freqWindow_bins = NULL,
  method = c("cepstral", "gauss", "movavg", "peak"),
  plot = FALSE
)
```

Arguments

`spec` numeric vector (magnitude spectrum of one frame) or matrix (rows = frequency bins, columns = time frames), such as a spectrogram returned by `stft_simple` or `spectrogram`. Must have at least 3 rows (frequency bins) and only finite values; zeros and negative values are floored at 1e-10 before taking the log. The input spectrum is expected to be on a linear, not logarithmic (dB) scale

`freqs` frequency labels corresponding to `spec`, in kHz (not Hz!); not needed if `freqWindow_bins` is provided

`freqWindow, freqWindow_bins` the width of the smoothing window, in Hz (not kHz!) or frequency bins (>0): for example, if we are trying to smooth away the harmonics of f0 and leave only formants, `freqWindow` must exceed the expected spacing between harmonics. Larger values produce smoother envelopes. If `freqWindow_bins` is provided, it overrides `freqWindow`

`method` the method of smoothing: "cepstral" = Gaussian liftering of the real cepstrum (default), "gauss" = Gaussian blur of the log spectrum, "movavg" = moving average of the log spectrum, "peak" = moving maximum (upper envelope)

`plot` if TRUE, produces a simple plot of the original spectrum and the extracted envelope

Details

The amount of smoothing is controlled by `freqWindow_bins`, which should normally equal the expected spacing between harmonics (f0, in bins): spectral details that vary on a faster scale are treated as source fine structure and removed, while slower variations (formants) are retained. For high-pitched or variable calls, increase `freqWindow_bins` to smooth more. Methods:

- "cepstral" (default): Gaussian low-pass liftering of the real cepstrum (FFT of the log spectrum). The harmonic ripple, which has a period of `freqWindow_bins` bins, is attenuated to ~exp(-4) = 2 amplitude, while the formant envelope is preserved. By the convolution theorem, this is equivalent to Gaussian smoothing of the log spectrum with SD = `freqWindow_bins` / 2, but computed in the quefrency domain.
- "gauss": Gaussian blur of the log spectrum along the frequency axis with SD = `freqWindow_bins` / 2 - the same low-pass filter as "cepstral", but with explicit edge padding instead of circular (wrap-around) filtering.
- "movavg": moving average of the log spectrum with a rectangular window of width `freqWindow_bins` (rounded up to an odd number).
- "peak": moving maximum (morphological dilation), i.e. the upper envelope of the spectrum with a window of width `freqWindow_bins`. Much slower than the other methods on long inputs.

The edges of the spectrum are handled by padding (repeating the edge values) before smoothing and trimming afterwards, so the output always has the same length as the input. If `spec` is a matrix (rows = frequency bins, columns = time frames), all frames are smoothed at once.

Value

The spectral envelope on the original (linear magnitude) scale, as a numeric vector or matrix with the same dimensions as `spec`.

See Also

`getEnv` for the temporal envelope of a waveform; `shiftFormants`, which uses `getSpecEnv` to shift formants

Examples

```
# Synthetic spectrum: three formants plus harmonics 20 bins apart
N = 512
freq = 1:N
true_envelope = exp(-.5 * ((freq - 100) / 20)^2) +
  exp(-.5 * ((freq - 250) / 30)^2) +
  exp(-.5 * ((freq - 400) / 40)^2)
spectrum = true_envelope * (0.5 + 0.5 * abs(sin(pi * freq / 20)))

plot(freq, spectrum, type = 'l', log = 'y',
     main = 'Spectral envelope', xlab = 'Frequency, bins')
lines(freq, true_envelope, col = 'grey60', lwd = 4)
lines(getSpecEnv(spectrum, freqWindow_bins = 20, method = 'cepstral'),
      col = 'red', lwd = 2)
lines(getSpecEnv(spectrum, freqWindow_bins = 20, method = 'gauss'),
```

```
      col = 'orange', lwd = 2, lty = 2)
lines(getSpecEnv(spectrum, freqWindow_bins = 20, method = 'movavg'),
      col = 'blue', lwd = 2, lty = 3)
lines(getSpecEnv(spectrum, freqWindow_bins = 20, method = 'peak'),
      col = 'green', lwd = 2, lty = 4)
legend('bottom',
       legend = c('raw', 'truth', 'cepstral', 'gauss', 'movavg', 'peak'),
       col = c('black', 'grey60', 'red', 'orange', 'blue', 'green'),
       lwd = c(1, 4, 2, 2, 2, 2), lty = c(1, 1, 1, 2, 3, 4), bty = 'n')

# Smoothed spectral envelope of a single vowel
data(speechEx, package = 'soundgen')
spec = spectrum(speechEx, from = .15, to = .3, plot = FALSE)
env = getSpecEnv(spec, freqWindow = 500, plot = TRUE)

# Smooth a whole spectrogram at once (matrix input):
spec = stft_simple(speechEx@left[1:16000],
                   samplingRate = speechEx@samp.rate,
                   wl = 512, step = 256)
spec = Mod(spec[1:(nrow(spec) %/% 2 + 1), ])
env = getSpecEnv(spec, freqWindow = 500, plot = TRUE)
```

`getSurprisal` *Get surprisal*

Description

Tracks the unpredictability of spectro-temporal changes in a sound over time, returning continuous contours of Shannon surprisal (`$info`), Bayesian surprise (`$kl` for Kullback-Leibler divergence), and autocorrelation-based surprisal (`$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 listener's attention.

Usage

```
getSurprisal(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  winSurp = 2000,
  specFun = "audSpec",
  specFun_pars = list(),
  logSpec = TRUE,
  method = c("acf", "none"),
  sameLagAllFreqs = FALSE,
  weightByAmpl = TRUE,
  weightByPrecision = TRUE,
```

```
  onlyPeakAutocor = TRUE,
  rescale = FALSE,
  minProb = 1e-12,
  summaryFun = "mean",
  output = c("surprisal", "info", "kl"),
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  osc = c("linear", "dB", "none"),
  heights = c(3, 1),
  ylim = NULL,
  maxPoints = c(1e+05, 5e+05),
  colorTheme = "bw",
  col = NULL,
  extraContour = list(col = "blue", lwd = 3, lty = 1),
  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 to normalize the 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 means "from sound onset"; windows shorter than 3 frames produce NA
specFun	the function used to extract a spectrogram-like feature matrix. Can be a string or a custom function that takes audio (numeric vector) as the first argument and returns a spectrogram-like matrix with time in columns and features in rows (see examples). A precomputed spectrogram-like matrix is also accepted (features in rows, time in columns [ms], numeric rownames for plotting). Supported strings: stft_simple 'stft' / STFT / 'stft_simple' (amplitude spectrogram) Parameters in specFun_pars: samplingRate (optional if frequency and time labels are not needed), wl (samples), step (samples), wn, zp, padWithSilence.

spectrogram 'spectrogram' (amplitude spectrogram - a wrapper around stft_simple with more options). Parameters in specFun_pars: see spectrogram.

powspec 'powerspec' (power spectrogram with tuneR). Parameters in specFun_pars: wintime (s) or windowLength (ms), steptime (s) or step (ms), dither.

melfcc 'melspec' (mel-spectrogram), 'mfcc' / 'melfcc' (MFCCs). Parameters in specFun_pars: windowLength (ms), step (ms), nbands, maxfreq (Hz), MFCC (integer vector).

audSpectrogram 'audSpectrogram' / 'audSpec' (auditory spectrogram). Parameters in specFun_pars: see audSpectrogram.

getRMS 'getRMS' / 'rms' (RMS amplitude envelope). Parameters in specFun_pars: see getRMS.

getEnv 'getEnv' / 'env' (various smoothed envelopes: RMS, analytical, peak, etc.). Parameters in specFun_pars: see getEnv.

specFun_pars
: a list of parameters passed to specFun. Defaults for audSpectrogram, list(yScale = 'ERB', nFilters_oct = 6, step = 15,minFreq = 60); for melspec, windowLength = 20, step = 20, maxfreq = NULL, nbands = 128, MFCC = 2:13; for spectrogram, windowLength = 20,step = 20; for env, windowLength = 40, step = 20 converted to samples; for rms, windowLength = 40, step = 20. For convenience when using melspec, windowLength and step in ms are converted to wintime and steptime in seconds if no explicit values in seconds are supplied.

logSpec
: if TRUE, the output of specFun is log-transformed prior to calculating surprisal, offsetting as needed to avoid non-positive values

method
: affects $surprisal and $bestLag_mat only; has no effect on $info and $kl. acf = change in autocorrelation at the previously best lag after adding the final point; none = do not calculate $surprisal; $surprisal and $bestLag_mat are then NA.

sameLagAllFreqs
: only for method = 'acf'. If TRUE, the bestLag is calculated by averaging the ACFs of all channels, and the same bestLag is used to calculate the surprisal in each frequency channel (we expect the same "rhythm" for all frequencies). If FALSE, the bestLag is calculated separately for each frequency channel (we can track different "rhythms" at different frequencies).

weightByAmpl
: f TRUE, ACF averaging (when sameLagAllFreqs = TRUE) and aggregation of all per-channel contours (surprisal, info, infoW, kl, klW) are weighted by the maximum non-negative value per frequency channel in the current analysis window. For log-transformed, processed, or custom feature matrices, these weights are feature maxima, not necessarily physical amplitude.

weightByPrecision
: if TRUE, ACF-based surprisal is weighted by the current autocorrelation, so deviations from a previous pattern are more surprising if this pattern is strong.

onlyPeakAutocor
: if TRUE, only peaks of ACFs are considered (so bestLag can never be 1, and the first change after a string of static values results in surprisal = NA).

rescale
: if TRUE, aggregated surprisal is normalized from (-Inf, Inf) to approximately (-1, 1) using tanh(surprisal / 2). surprisal_mat is unaffected.

minProb — minimum probability used to cap Shannon surprisal: `info` and `infoW` cannot exceed `-log(minProb)`. Also used as a lower bound for the KL divergence before logging in `kl` and `klW`.

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

output — what to return, options: 'surprisal', 'loudness', 'dLoudness', 'surprisalLoudness', 'surprisal_mat', 'bestLag_mat', 'info', 'info_mat', 'infoW', 'infoW_mat', 'kl', 'kl_mat', 'klW', 'klW_mat', 'spectrogram', 'all' (see the Return section)

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

cores — number of cores for parallel processing

plot — If TRUE, plots the feature matrix and the surprisal contour. For `specFun = 'env'`, the surprisal contour is overlaid on an oscillogram.

savePlots — if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed — if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

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

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

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, but nonlinear yScale also warps the contour) that will be plotted over the spectrogram (eg pitch contour); can also be a list with extra graphical parameters such as `lwd, col, warp` (FALSE = plot as is, TRUE = warp to conform to nonlinear yScale), etc. (see examples)

`xlab, ylab, main, mar, xaxp` — graphical parameters for plotting

grid — if numeric, adds n = `grid` dotted lines per kHz and the same number of lines along the time axis

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

...
: other graphical parameters

Details

Algorithm: the sound is transformed into some spectrogram-like representation (e.g., an auditory spectrogram, a mel-warped STFT spectrogram, etc.) or an RMS amplitude envelope. Using just the envelope is very fast, but then we discard all spectral information. For each frequency channel, a sliding window is analyzed to compare the actually observed final value with its expected value. The resulting per-channel surprisal contours are aggregated by taking their mean - optionally, weighted by the maximum 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 product of its positive changes and surprisal.

Value

A list with two top-level elements: `$detailed` and `$summary`.

`$detailed` contains per-frame statistics selected with the `output` argument. If multiple sounds are analyzed, `$detailed` is a list of per-sound lists.

`$summary` contains per-file summaries of a fixed set of contours: `loudness`, `surprisal`, `surprisalLoudness`, `info`, `infoW`, `kl`, and `klW`. These are summarized even if not all of them are included in `output`. If `summaryFun` is `NULL`, `$summary` is `NULL`.

Available measures:

surprisal Aggregated surprisal contour: change in autocorrelation. Values are averaged across frequency channels, optionally weighted by channel amplitude. Positive values mean "an unexpected change", and negative values mean "a change that confirms the expectations, making signal periodicity more certain." If `rescale = TRUE`, the aggregated contour is transformed with `tanh(surprisal / 2)` to approximately `(-1, 1)`.

loudness Subjective loudness in sone, as per `getLoudness`, resampled to match the number of surprisal frames.

dLoudness First temporal derivative of max-normalized loudness: `diff(c(0, loudness / max(loudness)))`.

surprisalLoudness Product of the positive parts of surprisal and dLoudness: `max(surprisal, 0) * max(dLoudness, 0)`. This contour emphasizes surprising events that coincide with increases in loudness. If `rescale = TRUE`, this uses the rescaled surprisal contour.

surprisal_mat Matrix of per-channel surprisal values before aggregation across frequency channels (frequency channels in rows, time frames in columns). Column names are time in ms.

bestLag_mat Matrix of the autocorrelation lag used to calculate ACF-surprisal in each time-frequency bin, in seconds. `NA` where no lag was available or applicable, static analysis windows, or when `onlyPeakAutocor = TRUE` and no ACF peak was found. If `sameLagAllFreqs = TRUE`, the same lag is used for all non-static frequency channels; static channels still return `NA`.

info Shannon surprisal contour, calculated as `-log(rho)`, where `rho` is the Gaussian density at the next observation normalized by the maximum Gaussian density. Values are capped at `-log(minProb)`.

info_mat Matrix of per-channel Shannon surprisal values corresponding to `info`.

infoW Windowed Shannon surprisal: same as `info`, but using weighted means and standard deviations from a half-Gaussian taper that prioritizes more recent observations.

infoW_mat Matrix of per-channel windowed Shannon surprisal values corresponding to `infoW`.

kl Bayesian log-surprisal: natural logarithm of the Kullback-Leibler divergence between the Gaussian distributions before and after observing the next data point, with a window-length correction added as `2 * log(n)`. To avoid `-Inf`, the KL divergence is floored at `minProb` before taking the logarithm. Values can therefore be negative.

kl_mat Matrix of per-channel Bayesian log-surprisal values corresponding to `kl`.

klW Windowed Bayesian log-surprisal: same as `kl`, but using weighted means and variances from a half-Gaussian taper that prioritizes more recent observations. The full window length `n` is still used for the `2 * log(n)` correction.

klW_mat Matrix of per-channel windowed Bayesian log-surprisal values corresponding to `klW`.

spectrogram The spectrogram-like feature matrix actually analyzed (frequency channels or features in rows, time frames in columns), after any requested preprocessing such as log-transformation. Column names are time in ms. If `specFun = 'env'`, this is a one-row matrix containing the RMS envelope, possibly log-transformed if `logSpec = TRUE`.

References

- Anikin, A. (2026) Measuring surprisal in sound sequences. Behavior Research Methods. doi: 10.3758/s13428-026-03153-3.

Examples

```
# A quick example
data('speechEx', package = 'soundgen')
surp = getSurprisal(speechEx, from = 0.5, to = 1)
surp

## Not run:
# A few more meaningful examples

## Example 1: a temporal deviant
s0 = soundgen(nSyl = 8, sylLen = 150,
              pauseLen = c(rep(200, 7), 450), pitch = c(200, 150),
              temperature = .05, plot = FALSE)
sound = c(rep(0, 4000),
          addVectors(rnorm(16000 * 3.5, 0, .02), s0, insertionPoint = 4000),
          rep(0, 200))
spectrogram(sound, 16000, yScale = 'ERB')

# long window (Inf = from the beginning)
surp = getSurprisal(sound, 16000, winSurp = Inf, output = 'all')
plot(sound, type = 'l')
surp_cont = surp$detailed$surprisal
lines(seq(0, length(sound), length.out = length(surp_cont)),
  surp_cont / max(surp_cont, na.rm = TRUE), col = 'blue', lwd = 2)

# Which frequency-time bins are surprising?
```

```
filled.contour(x = as.numeric(colnames(surp$detailed$surprisal_mat)) / 1000,
               y = as.numeric(rownames(surp$detailed$surprisal_mat)),
               z = t(surp$detailed$surprisal_mat),
               xlab = 'Time, s',
               ylab = 'Frequency, kHz')
# Best lag (periodicity) over time
hist(surp$detailed$bestLag_mat, xlab = 'Period, s')
abline(v = .35, lty = 3, lwd = 3, col = 'blue')  # true period = 350 ms

# just use the amplitude envelope instead of an auditory spectrogram
surp = getSurprisal(sound, 16000, winSurp = Inf, specFun = 'env')

# increase spectral and temporal resolution (can be slow)
surp = getSurprisal(sound, 16000, winSurp = 2000,
  specFun_pars = list(nFilters = 50, step = 10,
  yScale = 'bark', bandwidth = 1/4), output = 'all')

# weight by increase in loudness
spectrogram(sound, 16000, extraContour = surp$detailed$surprisalLoudness /
  max(surp$detailed$surprisalLoudness, na.rm = TRUE) * 8000)

par(mfrow = c(3, 1))
plot(surp$detailed$surprisal, type = 'l', xlab = '',
  ylab = '', main = 'surprisal')
abline(h = 0, lty = 2)
plot(surp$detailed$dLoudness, type = 'l', xlab = '',
  ylab = '', main = 'd-loudness')
abline(h = 0, lty = 2)
plot(surp$detailed$surprisalLoudness, type = 'l', xlab = '',
  ylab = '', main = 'surprisal * d-loudness')
par(mfrow = c(1, 1))

# short window = amnesia (every new sound is surprising)
getSurprisal(sound, 16000, winSurp = 300)

# add bells and whistles
surp = getSurprisal(sound, samplingRate = 16000,
  osc = 'dB',  # plot oscillogram in dB
  heights = c(2, 1),  # spectro/osc height ratio
  # 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
)

## Example 2: a spectral deviant
s1 = soundgen(
  nSyl = 11, sylLen = 150, invalidArgAction = 'ignore',
  formants = NULL, lipRad = 0,  # so all syls have the same envelope
  pauseLen = 90, pitch = c(1000, 750), rolloff = -20,
```

```
  pitchGlobal = c(rep(0, 5), 18, rep(0, 5)),
  temperature = .01, pitchCeiling = 7000,
  plot = TRUE, windowLength = 35)
surp = getSurprisal(s1, 16000, winSurp = 1500, output = 'all')
filled.contour(x = as.numeric(colnames(surp$detailed$surprisal_mat)) / 1000,
               y = as.numeric(rownames(surp$detailed$surprisal_mat)),
               z = t(surp$detailed$surprisal_mat),
               xlab = 'Time, s',
               ylab = 'Frequency, kHz')
# deviant surprising both at 1 kHz (expected tone omitted) and at the new freq
surp = getSurprisal(s1, 16000, winSurp = 1500,
  specFun = 'env')  # doesn't work - need spectral info

## Example 3: different rhythms in different frequency bins
s6_1 = soundgen(nSyl = 23, sylLen = 100, pauseLen = 50, pitch = 1200,
  rolloffExact = 1, invalidArgAction = 'ignore', plot = TRUE)
s6_2 = soundgen(nSyl = 10, sylLen = 250, pauseLen = 100, pitch = 400,
  rolloffExact = 1, invalidArgAction = 'ignore', plot = TRUE)
s6_3 = soundgen(nSyl = 5, sylLen = 400, pauseLen = 200, pitch = 3400,
  rolloffExact = 1, invalidArgAction = 'ignore', plot = TRUE)
s6 = addVectors(s6_1, s6_2)
s6 = addVectors(s6, s6_3)

surp = getSurprisal(s6, 16000, winSurp = Inf, sameLagAllFreqs = TRUE,
  specFun_pars = list(nFilters = 32), output = 'all')
surp = getSurprisal(s6, 16000, winSurp = Inf, sameLagAllFreqs = FALSE,
  specFun_pars = list(nFilters = 32), output = 'all')  # learns all 3 rhythms
filled.contour(x = as.numeric(colnames(surp$detailed$surprisal_mat)) / 1000,
               y = as.numeric(rownames(surp$detailed$surprisal_mat)),
               z = t(surp$detailed$surprisal_mat),
               xlab = 'Time, s',
               ylab = 'Frequency, kHz')

## Example 4: different time scales
s8 = soundgen(nSyl = 4, sylLen = 75, pauseLen = 50)
s8 = rep(c(s8, rep(0, 2000)), 8)
getSurprisal(s8, 16000, specFun = 'env', winSurp = Inf)
# ACF picks up first the fast rhythm, then after a few cycles switches to
# the slow rhythm

# Custom input: produce a nice spectrogram first, then use it as input
sp = spectrogram(s0, 16000, windowLength = 10, step = 10, contrast = .3,
  output = 'processed')  # return the modified spectrogram
colnames(sp) = as.numeric(colnames(sp)) / 1000  # convert ms to s
getSurprisal(s0, 16000, specFun = sp, logSpec = FALSE)

# Custom input: use acoustic features returned by analyze()
an = analyze(sound, 16000, windowLength = 20, novelty = NULL)
feature_mat = t(an$detailed[, 4:ncol(an$detailed)]) # or select pitch, HNR, ...
feature_mat = t(apply(feature_mat, 1, scale))  # z-transform all variables
feature_mat[is.na(feature_mat)] = 0  # get rid of NAs
colnames(feature_mat) = an$detailed$time  # time stamps in ms
rownames(feature_mat) = 1:nrow(feature_mat)
```

```
image(t(feature_mat))  # not a spectrogram, just a feature matrix
getSurprisal(sound, 16000, specFun = feature_mat, logSpec = FALSE)

# analyze all sounds in a folder
surp = getSurprisal('~/Downloads/temp/', savePlots = TRUE)
surp$summary

## End(Not run)
```

`hilbert_exact` *Hilbert transforms*

Description

`hilbert_exact` treats the input as one period of an infinite periodic signal (circular convolution), while `hilbert_approx` pads the input on both sides to a good length for FFT, which is faster at the cost of possible slight artifacts at the edges (~linear convolution). If the input is <3 samples long, the envelope is calculated simply as `Mod(x)`. Note: only `hilbert_approx` is fast enough to be used for extracting the envelopes of typical audio.

Usage

```
hilbert_exact(x)

hilbert_approx(x)
```

Arguments

`x`	numeric vector

Value

A list with two components:

hilbert Hilbert transform

envelope Instantaneous amplitude of the analytic signal (envelope)

Examples

```
# signal: amplitude-modulated sine wave
t = seq(0, 1, length.out = 477)
carrier = cos(2 * pi * 50 * t)
modulator = 1 + 0.5 * cos(2 * pi * 5 * t)
s = modulator * carrier  # amplitude modulated signal
hil_exact = hilbert_exact(s)
hil_approx = hilbert_approx(s)
plot(s, type = 'l')
points(hil_exact$envelope, type = 'l', col = 'blue')
points(hil_approx$envelope, type = 'l', col = 'red')
```

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)
```

HzToOther *Convert between Hz and other frequency scales*

Description

Converts between Hz and ERB, bark, mel, log, semitones relative to a reference frequency, or musical notes. Accepts vectors and missing values.

Usage

```
HzToOther(
  x,
  scale = c("ERB", "bark", "mel", "log", "semitones", "notes", "linear", "orig"),
  ...
)

otherToHz(
  x,
  scale = c("ERB", "bark", "mel", "log", "semitones", "notes", "linear", "orig"),
  ...
)
```

Arguments

`x`	vector or matrix of frequencies
`scale`	target scale: "bark" = Zwicker's critical bandwidth scale calculated as in Wang et al. 1992 (see https://en.wikipedia.org/wiki/Bark_scale), "mel" = O'Shaughnessy's original formula (https://en.wikipedia.org/wiki/Mel_scale), "ERB" = Equivalent Rectangular Bandwidth rate (see Moore & Glasberg 1983, 1990), "log" = log2, "semitones" = semitones relative to a reference value, "notes" = musical notation, "linear" or "orig" = no change
`...`	other arguments passed on to the scale-specific function

Value

Numeric vector of converted frequencies.

Scale-specific arguments

The following arguments can be passed via `...`. They are only used when the corresponding `scale` is selected; otherwise, they are silently ignored.

method Character, used when `scale = "ERB"`. Choose between "linear" (Glasberg & Moore, 1990) and "quadratic" (Moore & Glasberg, 1983). Default is "linear".

ref Numeric, used when `scale = "semitones"`. Reference frequency (Hz) for the semitone scale. Default is C-5 (0.5109875 Hz).

A4 Numeric, used when `scale = "notes"`. Frequency of the note A4. Default is 440 Hz (modern standard ISO 16, concert pitch).

showCents Logical, used only with HzToOther() when `scale = "notes"`. If `TRUE`, appends the cent deviation (rounded to the nearest integer) to the note name. Default is `FALSE`. Not applicable to otherToHz().

No additional arguments are required or usedor scale = "bark", "mel", "log", or "linear".

References

- Moore, B. C., & Glasberg, B. R. (1983). Suggested formulae for calculating auditory-filter bandwidths and excitation patterns. The journal of the acoustical society of America, 74(3), 750-753.
- Glasberg, B. R., & Moore, B. C. (1990). Derivation of auditory filter shapes from notched-noise data. Hearing research, 47(1-2), 103-138.

Examples

```
x = c(-20, 20, 100, 440, 1000, NA)
HzToOther(x, 'ERB')
HzToOther(x, 'ERB', 'quadratic')
HzToOther(x, 'bark')
HzToOther(x, 'mel')
HzToOther(x, 'log')
HzToOther(x, 'semitones', ref = 16)
HzToOther(x, 'notes', showCents = TRUE)

# ...and back to Hz
x = c(0:10, NA)

otherToHz(x, 'ERB')
otherToHz(x, 'ERB', method = 'quadratic')
otherToHz(HzToOther(c(100, 440, 2000), 'ERB'), 'ERB')

otherToHz(x, 'bark')
otherToHz(HzToOther(c(100, 440, 2000), 'bark'), 'bark')

otherToHz(x, 'mel')
otherToHz(HzToOther(c(100, 440, 2000), 'mel'), 'mel')

otherToHz(x, 'log')

otherToHz(x, 'semitones')
HzToOther(c(440, 210, 880), 'semitones', ref = 440)
otherToHz(HzToOther(c(440, 210, 880), 'semitones'), 'semitones')

otherToHz(c('A4', 'C#6', 'blabla', 'C0', 'C-2', NA), 'notes')
HzToOther(c(440, 293, 115, 16.35, 4), 'notes')

HzToOther(c(440, 415, 80, 81), 'notes', 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
HzToOther(c(440, 415, 16.35), 'notes', A4 = 415)
otherToHz(c("A4", "D4", "A#2", "C0", "C-2"), 'notes', A4 = 415)
```

`interpolate` *Interpolation*

Description

Interpolates unevenly spaced points into a relatively smooth curve. If the points are evenly spaced but there are missing values and/or aliasing should be avoided, use resample instead.

Usage

```
interpolate(
  x,
  y,
  xout,
 method = c("splineFC", "spline", "constant", "linear", "approxLowPass", "sgolay",
    "pchip", "cosine", "cardinal", "hermite", "loess"),
  plot = FALSE,
  ...
)

interpol_approxLowPass(bandwidth = 0.1)

interpol_sgolay(p = 3, n = 15)

interpol_cardinal(tension = 0.5)

interpol_loess(span = 0.5)
```

Arguments

`x, y`	numeric vectors giving the coordinates of the points to be interpolated (no NAs)
`xout`	numeric vector of target x-coordinates where interpolation is to take place
`method`	interpolation method to use. Accepts either a character string naming an in-built method (see "Interpolation methods" below) or a constructor function such as `interpol_loess(span = 0.3)` or a custom function (see examples). The default is `"splineFC"`. When calling `interpolate()` directly, a string plus method-specific arguments in `...` is the simplest option. When passing `method` through a higher-level function (e.g. `resample()`), use a constructor because it packages all method-specific arguments into a single object
`plot`	logical; if TRUE, a quick diagnostic plot is drawn
`...`	extra arguments specific to the chosen interpolation method, e.g. `span = 0.7` for `"loess"` (see "Interpolation methods" for per-method arguments). Used only when `method` is a character string; when `method` is a constructor function, parameters are passed to the constructor directly. Do not pass graphical arguments here
`bandwidth`	(interpol_approxLowPass) the amount of smoothing, a number between 0 and 1: close to 0 = more smoothing, close to 1 = less smoothing. Defaults to `len_x / len_out` when called via the string dispatch of `interpolate()`.
`p`	(interpol_sgolay) polynomial order for Savitzky-Golay smoothing (positive integer, defaults to 3).

`n` (interpol_sgolay) filter length for Savitzky-Golay smoothing (odd positive integer, defaults to `floor(len_out / len_x / 2) * 2 + 1` when called via the string dispatch of `interpolate()`).

`tension` (interpol_cardinal) a number between 0 and 1 controlling the tightness of the cardinal spline: `tension = 0` gives Catmull-Rom (smooth but may overshoot), `tension = 0.5` is a good balance, and `tension = 1` makes the curve pass through midpoints between the input points. Defaults to 0.5.

`span` (interpol_loess) the amount of LOESS smoothing, a number between 0 and 1: a larger `span` gives more smoothing, a smaller `span` captures finer detail but may overfit. Defaults to 0.5.

Value

A numeric vector of interpolated y-values at the requested `xout` locations.

Interpolation methods

constant Constant interpolation via `approx`. Fast, but no smoothing.

linear aka approx Linear interpolation via `approx`. Fast, but not smooth.

spline Cubic spline interpolation (FMM method) via `spline`. Fast, but overshoots.

splineFC Monotone cubic interpolation using the Fritsch-Carlson method (see `splinefun`). Moderately fast, less overshooting than the FMM spline.

approxLowPass Linear interpolation followed by low-pass filtering. Fast, smooth, but reduced range compared to original y. Constructor: `interpol_approxLowPass(bandwidth)`.

sgolay Linear interpolation followed by Savitzky-Golay smoothing (see `sgolayfilt`). Fairly similar to approxLowPass, but much slower. Constructor: `interpol_sgolay(p, n)`.

pchip Piecewise Cubic Hermite Interpolating Polynomial (preserves monotonicity). Calls `interp1` with `method = "pchip"`.

cosine Cosine-eased (smoothstep) interpolation. Eases between anchor points using a cosine curve. Fast, passes exactly through input points.

cardinal Cardinal spline interpolation (a generalization of Catmull-Rom). Fast, can be forced to pass exactly through input points. Constructor: `interpol_cardinal(tension)`.

hermite Hermite spline interpolation with forced zero slope at local extrema to prevent overshoot. Fast, passes exactly through input points.

loess Locally estimated scatterplot smoothing (LOESS) via `loess`. Smooth, but slow; may overshoot. Constructor: `interpol_loess(span)`.

See Also

interpolateNA resample

Examples

```
x  = c(0, .15, .2, .3, .7, 1)
y  = c(360, 116, 550, 350, 700, 610)
xout = seq(0, 1, length.out = 100)

# Compare inbuilt interpolation methods
ms = c('constant', 'linear', 'spline', 'splineFC', 'approxLowPass',
        'sgolay', 'cosine', 'cardinal', 'hermite', 'loess')
op = par(c('mfrow', 'mar')); par(mfrow = c(4, 3), mar = c(2, 2, 3, 1))
for (m in ms) {interpolate(x, y, xout, method = m, plot = TRUE); title(m)}
par(op)

# Passing method-specific parameters via ...
interpolate(x, y, xout, method = 'cardinal', tension = 0, plot = TRUE)
interpolate(x, y, xout, method = 'loess', span = 0.2, plot = TRUE)
interpolate(x, y, xout, method = 'loess', span = 0.9, plot = TRUE)

# Equivalent: passing a constructor (useful when forwarding through
# higher-level functions like resample())
interpolate(x, y, xout, method = interpol_cardinal(tension = 0), plot = TRUE)
interpolate(x, y, xout, method = interpol_loess(span = 0.2), plot = TRUE)

# Passing a fully custom function
interpolate(x, y, xout, plot = TRUE,
  method = function(...) spline(x, y, xout = xout, method = 'natural')$y)
```

`interpolateNA` *Interpolate NAs*

Description

Takes a numeric vector, such as a pitch contour, and fills in the NAs, first by linear interpolation in the middle and then by constant or linear interpolation at the ends.

Usage

```
interpolateNA(x, idx_na = NULL, nPoints = 1)
```

Arguments

`x`	numeric vector
`idx_na`	which(is.na(x))
`nPoints`	the number of points to use for interpolating leading and trailing NAs: 1 = constant interpolation, 2 = use the first two non-NAs at the beginning and the last two non-NAs at the end (possibly after interpolating NAs in the middle), etc.

Value

The input numeric vector with NAs filled in by interpolation.

Examples

```
a = c(NA, 405, NA, 460, NA, NA, NA, 480, 490, NA, NA)
interpolateNA(a)
interpolateNA(a, nPoints = 3)
plot(interpolateNA(a), type = 'l', col = 'blue'); points(a)
plot(interpolateNA(a, nPoints = 3), type = 'l', col = 'blue'); points(a)
# Compare to approx - terminal NAs are simply trimmed
approx(a, na.rm = TRUE, n = length(a))$y
```

---

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 spectro-temporal modulation domain.

Usage

```
invertSpectrogram(
  spec,
  samplingRate = NULL,
  windowLength,
  step,
  overlap = NULL,
  wn,
  specScale = c("spec", "power", "log", "dB"),
  initialPhase = c("spsi", "random", "zero"),
  nIter = 50,
  normalize = TRUE,
  play = FALSE,
  verbose = FALSE,
  plotError = TRUE
)
```

Arguments

spec	the spectrogram that is to be transform to a time series: numeric matrix of real value (no phase) with frequency bins in rows (kHz) and time frames in columns (ms)
samplingRate	sampling rate (not needed if the spectrogram has rownames corresponding to frequency bins with the last one at Nyquist = samplingRate / 2)

`windowLength`, `step`, `overlap`, `wn`
: STFT parameters used to create the original spectrogram; make sure `zp = 0`

`specScale`
: the scale of target spectrogram: 'spec' = untransformed amplitude spectrum, 'power' = power spectrum, 'log' = log-transformed, 'dB' = in decibels

`initialPhase`
: initial phase estimate: "spsi" (default) = single-pass spectrogram inversion (Beauregard et al., 2015); "zero" = set all phases to zero; "random" = Gaussian noise

`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, plots the error during GL iterations and prints the final reconstruction error (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

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. NB: do NOT use zero-padding
samplingRate = 16000
windowLength = 40
step = 5
wn = 'gaussian'
# NB: the more detailed a spectrogram, the more precisely it can be inverted,
# so a relatively long window AND a short step make for best results
```

```
s = soundgen(samplingRate = samplingRate, addSilence = 50)
spec = spectrogram(s, samplingRate = samplingRate,
  wn = wn, windowLength = windowLength, step = step,
  zp = 0,  # otherwise it changes the window length and messes up istft
  padWithSilence = FALSE, output = 'original')

# Invert the spectrogram, attempting to guess the phase
# Note that we need to know the original windowLength, step, and wn
# (i.e., you have to know how the spectrogram was created)
s_new = invertSpectrogram(spec, samplingRate = samplingRate,
  windowLength = windowLength, step = step, wn = wn,
  initialPhase = 'spsi', nIter = 50, play = FALSE)

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

## Not run:
# to improve the quality of reconstruction, increase the number of iterations
s_new = invertSpectrogram(spec, samplingRate = samplingRate,
  windowLength = windowLength, step = step, wn = wn,
  initialPhase = 'spsi', nIter = 500, play = FALSE)
playme(s, samplingRate); playme(s_new, samplingRate)
spectrogram(s, samplingRate)
spectrogram(s_new, samplingRate)

## End(Not run)
```

IR *Impulse response*

Description

A finger snap in a room as an approximation to an impulse response

Usage

```
IR
```

Format

An object of class `Wave` of length 3252.

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 https://cogsci.se/soundgen/sound_generation.html and https://cogsci.se/soundgen/matching/matching.html for more information.

Usage

```
matchPars(
  target,
  samplingRate = NULL,
  pars = NULL,
  specFun = "melspec",
  specFun_pars = list(),
  init = NULL,
  probMutation = 0.25,
  stepVariance = 0.1,
  maxIter = 50,
  minExpectedDelta = 0.001,
  compareSounds_pars = list(),
  verbose = TRUE,
  play = FALSE
)
```

Arguments

target	the sound we want to reproduce using soundgen: path to an audio 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
specFun	the function used to extract a spectrogram-like feature matrix. Can be a string or a custom function that takes audio (numeric vector) as the first argument and returns a spectrogram-like matrix with time in columns and features in rows (see examples). Supported strings: stft_simple 'stft' / STFT / 'stft_simple' (amplitude spectrogram) Parameters in specFun_pars: samplingRate (optional if frequency and time labels are not needed), wl (samples), step (samples), wn, zp, padWithSilence. spectrogram 'spectrogram' (amplitude spectrogram - a wrapper around stft_simple with more options). Parameters in specFun_pars: see spectrogram.

powspec 'powerspec' (power spectrogram with tuneR). Parameters in specFun_pars: wintime (s), steptime (s), dither.

melfcc 'melspec' (mel-spectrogram with tuneR), 'mfcc' / 'melfcc' (MFCCs). Parameters in specFun_pars: windowLength (ms), step (ms), nbands, maxfreq (Hz), MFCC (integer vector).

audSpectrogram 'audSpectrogram' / 'audSpec' (auditory spectrogram). Parameters in specFun_pars: see audSpectrogram.

getRMS 'getRMS' / 'rms' (RMS amplitude envelope). Parameters in specFun_pars: see getRMS.

getEnv 'getEnv' / 'env' (upsampled envelopes: RMS, analytical, peak, etc.). Parameters in specFun_pars: see getEnv.

spectrum 'spectrum' (short-term spectrum). Parameters in specFun_pars: see spectrum.

meanSpectrum 'meanSpectrum' / 'meanspec' / 'meanSpec' (long-term average spectrum). Parameters in specFun_pars: see meanSpectrum.

ssm 'ssm' (self-similarity matrix). Parameters in specFun_pars: see ssm.

modulationSpectrum 'ms' / 'modulationSpectrum' (modulation spectrum). Parameters in specFun_pars: see modulationSpectrum.

specFun_pars — a list of parameters passed to specFun

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; iter = 0 means acoustic analysis only, no optimization

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

compareSounds_pars — a list of control parameters passed to compareSounds

verbose — if TRUE, reports the outcome at each iteration

play — if TRUE, plays back the accepted candidate at each iteration

Value

A list containing the history of parameters tried and their final values (pars).

Examples

```
## Not run:
target = soundgen(sylLen = 600, pitch = c(300, 200),
                  rolloff = -20, 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 (-12) to target (-20):
lapply(m2$history, function(x) x$pars$rolloff)
cand2 = do.call(soundgen, c(m2$pars, list(play = TRUE, plot = TRUE)))

## End(Not run)
```

`mixAtSNR` *Mix signals at specific SNR*

Description

Adds two partly overlapping vectors, such as two waveforms or "signal" and "noise", at specific RMS or loudness levels. The location at which vector 2 ("noise") is pasted is defined by `insertionPoint`. Algorithm: both vectors are padded with zeros to match in length, rescaled, added, and re-normalized. All NA's are converted to 0. `addVectors` is the former name from soundgen 2.x, preserved for back-compatibility.

Usage

```
mixAtSNR(
  signal,
  noise,
  insertionPoint = 1L,
  normalize = TRUE,
  SNR = NULL,
  type = c("rms", "RMS", "peak", "loudness"),
  loudness_pars = list()
)

addVectors(
  signal,
  noise,
  insertionPoint = 1L,
  normalize = TRUE,
```

```
  SNR = NULL,
  type = c("rms", "RMS", "peak", "loudness"),
  loudness_pars = list()
)
```

Arguments

signal, noise	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
SNR	signal-to-noise ratio, dB
type	rescale noise to target SNR relative to signal based on root mean square amplitude ('rms' or 'RMS', default), peak amplitude ('peak'), or mean subjective loudness in sone ('loudness')
loudness_pars	a list of parameters passed to getLoudness if type = 'loudness'

Value

A numeric vector.

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(v1, v2, 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)

# add noise to a sound at a particular SNR
signal = rep(c(rep(0, 500), cospi(2 * 440 * (1:1000) / 16000)), 2)
noise = rnorm(length(signal))
osc(mixAtSNR(signal, noise, SNR = 6), samplingRate = 16000)
osc(mixAtSNR(signal, noise, SNR = 6, type = 'peak'), samplingRate = 16000)
osc(mixAtSNR(signal, noise, SNR = 6, type = 'loudness',
  loudness_pars = list(samplingRate = 16000, SPL = 80)), samplingRate = 16000)
osc(mixAtSNR(signal, noise, SNR = 12), samplingRate = 16000)
osc(mixAtSNR(signal, noise, SNR = 0), samplingRate = 16000)

# no rescaling (SNR = NULL)
osc(mixAtSNR(signal, noise, SNR = NULL), samplingRate = 16000)
osc(mixAtSNR(signal, noise/4, SNR = NULL), samplingRate = 16000)
```

modulationSpectrum *Modulation spectrum*

Description

Produces a modulation spectrum of waveform(s) or audio file(s). It begins with a spectrogram-like time-frequency representation and analyzes the modulation of the envelope in each frequency band. If `specFun = 'audSpec'`, the sound is passed through a bank of bandpass filters with `audSpectrogram`. If `specFun = '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.

Usage

```
modulationSpectrum(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  msType = c("2D", "1D"),
  specFun = "STFT",
  specFun_pars = list(),
  amRes = 5,
  maxDur = 5,
  specMethod = c("meanSpectrum", "spectrum"),
  logSpec = FALSE,
  logMPS = FALSE,
  power = 1,
  normalize = TRUE,
  roughRange = NULL,
  roughMean = 100,
  roughSD = 8,
  roughMinFreq = 1,
  amRange = c(10, 200),
  fluctRange = c(0.25, 30),
  fluctMean = 4,
  fluctSD = 12,
  returnMS = TRUE,
  returnComplex = FALSE,
  summaryFun = c("mean", "sd"),
  output = "all",
  averageMS = FALSE,
```

```
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  logWarpX = NULL,
  logWarpY = NULL,
  quantiles = c(0.5, 0.8, 0.9),
  kernelSize = 5,
  kernelSD = 0.5,
  colorTheme = "bw",
  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 to normalize the 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
specFun	'STFT' or 'stft' = ordinary spectrogram, 'audSpec' = auditory spectrogram with audSpectrogram, 'env' = amplitude envelope with getEnv
specFun_pars	a list of parameters passed to specFun. Defaults: STFT list(windowLength = 15, overlap = 90, wn = 'hanning', zp = NULL) (window length and step are specified in ms), audSpec list(yScale = 'ERB', step = 1, envelope = 'hil', nFilters = 32), env list(). Whatever your specFun, careful with choosing the time step as it determines the temporal modulation resolution
amRes	approximate frequency resolution of amplitude modulation, Hz; equivalently, the number of independent MS / roughness / fluctuation values extracted per second of audio. Thus, larger amRes means more chunks (better temporal resolution of roughness), but each chunk is smaller (worse frequency resolution); lower amRes means fewer chunks, but better frequency resolution in each chunk. If NULL, the entire sound is analyzed at once, unless it is longer than maxDur, in

which case it is analyzed in chunks maxDur s long. If amRes is set, it overrides maxDur and the sound is analyzed in approximately equal chunks of about 1000 / amRes ms. For STFT-based spectrograms, each chunk contains at least three time frames whenever possible; if the requested resolution cannot be achieved, the effective resolution is lower. amRes also affects the width of the window over which local AM peaks are evaluated

maxDur — sounds longer than maxDur s are split into fragments, and the modulation spectra of all fragments are averaged. This parameter is used only when amRes = NULL

specMethod — the function to call when calculating the spectrum of each frequency band, only used when msType = '1D'; meanSpectrum is faster and less noisy, whereas spectrum gives higher resolution

logSpec — if TRUE, the spectrogram-like representation is log-transformed prior to estimating the modulation spectrum. This applies to both 2D and 1D modulation spectra. A small constant is added to avoid non-finite values

logMPS — if TRUE, the modulation spectrum is log-transformed prior to applying power, normalization, roughness calculation, and AM estimation. A small constant is added to avoid non-finite values

power — numeric exponent applied to the modulation spectrum, e.g. power = 2 for a squared modulation spectrum. This affects roughness and AM measures

normalize — if TRUE, the modulation spectrum of each analyzed fragment is min-max normalized to the range [0, 1], unless all values are identical

roughRange — the range of temporal modulation frequencies that constitute the "roughness" zone, Hz. If NULL, roughness can still be calculated using roughMean and roughSD

roughMean, roughSD — the mean (Hz) and standard deviation (semitones) of a lognormal distribution used to weight roughness estimates. Weighted roughness requires both roughMean and roughSD. If either is NULL, weighting is disabled and roughness is calculated using roughRange, if supplied. If neither a valid roughRange nor complete weighting parameters are supplied, roughness is undefined and is currently reported as 0. If both roughMean/roughSD and roughRange are defined, weights outside roughRange are set to 0; a very large SD, i.e. a flat weighting function, gives approximately the same result as using roughRange without weighting

roughMinFreq — frequencies below roughMinFreq (Hz) are ignored when calculating roughness and fluctuation, i.e. the estimated roughness and fluctuation strength increase if we disregard ~static components near AM = 0, which are often strong

amRange — the range of temporal modulation frequencies in which we look for systematic amplitude modulation, Hz

fluctRange, fluctMean, fluctSD — same as roughRange, roughMean,roughSD, but for calculating low-frequency fluctuations centered around 4 Hz (~0.25 to 30 Hz). Ensure that temporal windows are long enough to analyze slow fluctuations: e.g., use specMethod = 'spectrum' (not 'meanSpectrum') if doing 1D MS with msType = '1D', avoid splitting the sound into short chunks - set amRes = NULL, etc.

returnMS — if FALSE, only the summary measures are returned and plotting is disabled. This is faster and uses less memory. Careful with exporting the modulation spectra of many sounds at once, as this can require a lot of RAM

returnComplex — if TRUE and msType = '2D', returns the complex modulation spectrum prior to log transformation, power, and normalization. Ignored for msType = '1D'

summaryFun — summary functions used to summarize per-fragment roughness, amMsFreq, and amMsPurity across time; see analyze

output — what to return, options: 'original', 'original_list', 'modulation_spectrogram', 'processed', 'complex', 'roughness', 'roughness_spectrogram', 'roughness_list', 'fluctuation', 'fluctuation_list', 'fluctuation_spectrogram', 'amMsFreq', 'amMsPurity', 'ampl', 'all' (see the Return section)

averageMS — if TRUE, the modulation spectra of all successful inputs are averaged into a single output matrix; if FALSE, a separate MS is returned for each input. Note that summary statistics are still returned per input unless summarized separately

reportEvery — when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime

cores — number of cores for parallel processing

plot — if TRUE, plots the modulation spectrum of each sound (see plotMS). Plotting requires returnMS = TRUE

savePlots — if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed — if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

logWarpX, logWarpY — numeric vector of length 2: c(sigma, base) for pseudolog-warping the modulation spectrum in the plot, as in pseudo_log_trans() from the **scales** package. These arguments affect plotting only; the returned matrices are not log-warped

quantiles — numeric vector of cumulative contours in (0, 1). For example, 0.5 marks a region that contains 50% of the total modulation-spectrum amplitude. Set to NULL to omit contour lines

kernelSize — the size of the Gaussian kernel used for smoothing the modulation spectrum prior to plotting; 1 means no smoothing

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

colorTheme — plot color theme

col — custom color palette; if supplied, overrides colorTheme

main, xlab, ylab, 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)

Details

To calculate roughness and fluctuation depth, the symmetric modulation spectrum is folded by averaging its ±AM halves. Roughness is calculated as the proportion of the modulation spectrum within `roughRange` of temporal modulation frequencies or as a weighted version thereof. Fluctuation is calculated in the same manner as roughness, but in a lower frequency range of ~4 Hz (0.25 to 30 Hz). The frequency of amplitude modulation (`amMsFreq`, Hz) is calculated as the highest peak of the folded AM function within `amRange`, and its purity (`amMsPurity`, dB) as the ratio of this peak to the average of the folded AM function around the peak within 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, but inspect the MS visually to make sure the relevant modulation frequencies are resolved. For multiple inputs, such as a list of waveforms or a 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`.

The full STFT or auditory spectrogram is computed once and then sliced into temporal chunks. Roughness, fluctuation strength, `amMsFreq`, and `amMsPurity` are calculated from the per-fragment modulation spectra before Gaussian smoothing and before any log-warping used for plotting. Log-warps specified via `logWarpX` and `logWarpY` affect the plot only; the returned matrices are not log-warped.

Value

A list with three top-level elements: `$summary`, `$detailed`, and `$aggregated`.

`$summary` is a dataframe summarizing roughness, `amMsFreq`, and `amMsPurity` for each input (one row per file), or `NULL` if `summaryFun = NULL`.

`$detailed` contains detailed statistics per sound. If multiple sounds are analyzed, it is a named list of per-sound lists. If a single sound is analyzed, it is simplified to a single list. Components are selected via the `output` argument and may include:

original averaged modulation spectrum across fragments, after `logMPS`, `power`, and `normalize` if requested, but before Gaussian smoothing and plotting transforms. Colnames are temporal modulation frequencies (Hz). Rownames are spectral modulation frequencies if `msType = '2D'` and frequencies of filters or spectrogram bands if `msType = '1D'`.

original_list list of per-fragment modulation spectra.

modulation_spectrogram a spectrogram-like representation showing how the frequency-averaged modulation spectrum changes over time; `NA` if there is only one fragment.

processed modulation spectrum after Gaussian smoothing. This matrix is not log-warped; log-warping is applied only during plotting.

complex complex modulation spectrum, returned only if `returnComplex = TRUE`, `returnMS = TRUE`, and `msType = '2D'`.

roughness proportion of the modulation spectrum within `roughRange`, or a weighted version thereof if `roughMean` and `roughSD` are supplied, in percent. This is a vector if the sound is analyzed in multiple fragments, otherwise a single number.

roughness_list a list containing frequencies, band-wise roughness values, and band amplitudes for each fragment.

roughness_spectrogram a spectrogram-like matrix showing roughness per frequency band and per fragment.

fluctuation proportion of the modulation spectrum within `fluctRange`, or a weighted version thereof if `fluctMean` and `fluctSD` are supplied, in percent. This is a vector if the sound is analyzed in multiple fragments, otherwise a single number.

fluctuation_list a list containing frequencies, band-wise fluctuation values, and band amplitudes for each fragment.

fluctuation_spectrogram a spectrogram-like matrix showing fluctuation strength per frequency band and per fragment.

amMsFreq frequency of the highest AM peak within `amRange`. Like `roughness`, this can be a single number or a vector, depending on whether the sound is analyzed as a whole or in chunks.

amMsPurity ratio of the AM peak at `amMsFreq` to the mean amplitude of the other folded AM values within `amRange`, in dB.

ampl Root Mean Square amplitude of each analyzed fragment, divided by the internal audio scaling factor if present.

`$aggregated` is a list containing the `original`, `processed`, and (if requested) `complex` modulation spectra averaged across all successfully analyzed sounds. Only present if `averageMS = TRUE` and multiple sounds are analyzed.

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.
- Anikin, A. (2025) Acoustic estimation of voice roughness. Attention, Perception, & Psychophysics 87: 1771–1787.

See Also

`plotMS` `spectrogram` `audSpectrogram` `analyze`

Examples

```
s = soundgen(pitch = 440, amFreq = 100, amDep = 50)
ms = modulationSpectrum(s, samplingRate = 16000, amRes = NULL)
ms$detailed[c('roughness', 'amMsFreq', 'amMsPurity')]  # a single value for each
ms1 = modulationSpectrum(s, samplingRate = 16000, amRes = 5)
ms1$detailed[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)

# Plot a modulation spectrogram (the peak at 100 Hz shows AM)
spectrogram(s, 16000, specManual = ms$detailed$modulation_spectrogram,
  ylab = 'Modulation frequency, kHz', main = 'Modulation spectrogram')
```

```
# Plot a roughness spectrogram
spectrogram(s, 16000, specManual = ms$detailed$roughness_spectrogram,
  yScale = 'ERB', main = 'Roughness spectrogram')

# 1D instead of 2D
modulationSpectrum(s, 16000, msType = '1D', quantiles = NULL,
  col = soundgen:::jet.col(50), logWarpX = c(10, 2))

## 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), plot = TRUE, yScale = 'ERB')
playme(s_long)
ms = modulationSpectrum(s_long, 16000)
# plot AM over time
plot(names(ms$detailed$amMsFreq), ms$detailed$amMsFreq,
  cex = ms$detailed$amMsPurity/5,
  xlab = 'Time, ms', ylab = 'AM frequency, Hz')
# plot roughness over time
spectrogram(s_long, 16000, ylim = c(0, 4),
 extraContour = list(x = ms$detailed$roughness / max(ms$detailed$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, amRes = NULL,
  specFun_pars = list(windowLength = 50, step = 50)) # poor temporal resolution
ms = modulationSpectrum(s, samplingRate = 44100, amRes = NULL,
  specFun_pars = list(windowLength = 5, step = 1))  # poor frequency resolution
ms = modulationSpectrum(s, samplingRate = 44100, amRes = NULL,
  specFun_pars = list(windowLength = 15, step = 3))  # a reasonable compromise

# Start with an auditory spectrogram instead of STFT
modulationSpectrum(s, 44100, specFun = 'audSpec', xlim = c(-100, 100))
modulationSpectrum(s, 44100, specFun = 'audSpec',
  logWarpX = c(10, 2), xlim = c(-500, 500),
  specFun_pars = list(nFilters = 32, filterType = 'gammatone', bandwidth = NULL))

# customize the plot
ms = modulationSpectrum(s, samplingRate = 44100,
  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 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$summary
names(ms1$detailed)  # separate MS and other descriptives per file
# ...or a single MS can be calculated by averaging across all files:
ms2 = modulationSpectrum('~/Downloads/temp', kernelSize = 11,
                         plot = FALSE, averageMS = TRUE)
plotMS(ms2$aggregated$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$detailed$roughness
# compare:
modulationSpectrum(s, samplingRate = 16000, maxDur = .5,
  xlim = c(-25, 25), colorTheme = 'seewave',
  power = 1)$detailed$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))$detailed$roughness
modulationSpectrum(s, samplingRate = 16000, logWarpX = NA)$detailed$roughness
modulationSpectrum(s, samplingRate = 16000, kernelSize = 17)$detailed$roughness

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

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

```
  roughMean = NULL)$detailed$roughness  # same as above b/c SD --> Inf

# Complex modulation spectrum with phase preserved
ms = modulationSpectrum(soundgen(), samplingRate = 16000,
                        returnComplex = TRUE)
plotMS(log(abs(ms$detailed$complex + 1e-8)))  # note the symmetry
# compare:
plotMS(ms$detailed$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. Undefined values are set to reasonably intelligent defaults, but it is safer to specify the same parameters in both formulas. For instance, if formula1 includes "noise" and "mouth", set these parameters explicitly in formula2 as well.

Usage

```
morph(
  formula1,
  formula2,
  nMorphs,
  play = FALSE,
  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

`play`
: if TRUE or a character referring to a valid player, the morphs will be played as they are produced

`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 containing two lists, (`$formulas` and `$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, play = 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, play = 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(-20, -20, -30)),
    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(-25, -15, -35)),
    mouth = c(.4, .5), rolloffNoise = -5, attackLen = 30)',
  nMorphs = 8, play = 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(-40, -20, -70)),
    rolloffNoise = c(-5, -15))",
 nMorphs = 5, play = playback
)

## 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 static posterior probabilities of the preceding `wlPrior` observations if `prior = 'dynamic'`. `naiveBayes_train` calculates conditional means and standard deviations per class and 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(
  formula,
  train,
  test = train,
  prior = c("static", "dynamic", "flat"),
  wlPrior = 3,
  wlClumper = NULL,
```

```
  runBack = TRUE,
  plot = FALSE
)

naiveBayes_train(formula, train)
```

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 priori, "static" = use class probabilities in the training dataset, "dynamic" = update prior probabilities from static posterior probabilities of wlPrior preceding observations
wlPrior	the length of the dynamic-prior window in observations
wlClumper	the minimum length of a predicted class run, enforced by clumper()
runBack	if TRUE, the dynamic prior is calculated both forward and backward and averaged (only has an effect if prior = 'dynamic')
plot	if TRUE, produces diagnostic plots

Value

A dataframe with new columns: "pr" = the predicted class membership, "[outcome][class]" = posterior probabilities per class, "like_[class]" = normalized likelihoods, "prior_[class]" = normalized priors, "priorF_[class]" / "priorB_[class]" = forward / backward normalized priors (only filled for prior = 'dynamic').

Functions

- naiveBayes_train(): Train a naive Bayes classifier

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, train = 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)

# 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)

# 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)

# 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 clump the predicted classes (the latter imposes strong
# constraints on the predictions, but may be worth it if the data is known to
# be strongly "clumped", 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)

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)

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)

# 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 = test1,
  prior = 'dynamic', wlPrior = 10, plot = TRUE)
table(nb4$group, nb4$pr, useNA = 'ifany')
mean(nb4$group == nb4$pr, na.rm = TRUE)
```

---

noiseRemoval *Noise removal*

Description

Removes noise by log-spectral subtraction. 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 log-spectrum can be simply subtracted from that of the signal. Algorithm: STFT to produce a log-spectrogram, subtract normalized noise spectrum, iSTFT to reconstitute the signal. Most of the work is done by addFormants.

Usage

```
noiseRemoval(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  noise,
  dB = 6,
  specificity = 1,
  windowLength = 50,
  step = NULL,
  overlap = 75,
  dynamicRange = 120,
  normalize = c("orig", "max", "none"),
  reportEvery = NULL,
  cores = 1,
  play = FALSE,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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 specified (in seconds), only this section of input is denoised
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; path to file, Wave object, or numeric vector (with the same sampling rate as x) representing pure noise
dB	controls the amount of noise removal: larger values are more aggressive
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 the analysis window, ms
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
dynamicRange	regions under -dynamicRange dB are treated as silent
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude for the given input scale, "none" = no normalization
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
play	if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
plot	if TRUE, produces a plot of the results
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed — if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

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

... — extra graphical parameters passed to spectrogram

Value

The denoised audio as a numeric vector (multiple inputs return a list).

See Also

addFormants

Examples

```
s = soundgen(noise = list(time = c(-100, 400), value = -10),
  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)

## Not run:
# 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)

# play with gain and specificity
s4 = 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
s5 = noiseRemoval(s, samplingRate = 16000, from = .3, to = .5,
  noise = c(0.05, 0.15), dB = 60, plot = TRUE)

## 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. If `preserveRelativeDif` is set to `TRUE`, all recordings in the target folder get a boost in amplitude, so the loudest one reaches `maxAmp`, but the relative differences between the recordings are preserved (i.e., all files are boosted by the same amount - as much as possible to avoid clipping the loudest one).

Usage

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

Arguments

`myfolder`	full path to folder containing input audio files
`type`	normalize so the output files have 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')
`preserveRelativeDif`	(only for `type = "peak"`) if FALSE (default), all files are normalized to the same level; if TRUE, the peak amplitude of the loudest file is set to `maxAmp`, while the remaining files are quieter so the original relative peak amplitude across files is preserved
`windowLength`	length of analysis window, ms (longer windows = more smoothing)

step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
killDC	if TRUE, removes DC offset (see also flatEnv)
windowDC	the window for calculating DC offset, ms
cores	number of cores for parallel processing
savePath	full path to where the normalized files should be saved; defaults to NULL = 'myfolder/normalized'; NA = do not save the processed files
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime

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'. NB: because loudness is not a simple linear function of SPL, loudness normalization is only approximate; reiterate the process several times to improve the precision of loudness normalization.

Value

Does not return anything, only saves the normalized audio files.

See Also

getRMS analyze getLoudness

Examples

```
## Not run:
# put a few short audio files in a folder, eg '~/Downloads/temp'
target = '~/Downloads/temp'
save_in_folder = paste0(target, '/normalized')
getRMS(target, summaryFun = 'mean')$summary  # different
normalizeFolder(target, type = 'rms', summaryFun = 'mean',
  savePath = save_in_folder)
getRMS(save_in_folder, summaryFun = 'mean')$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)
```

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 the "oscillo" function in the seewave package.

Usage

```
osc(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  dynamicRange = 80,
  dB = FALSE,
  returnWave = FALSE,
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  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 to normalize the input vector (only needed if x is a numeric vector)

from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
dynamicRange	regions under -dynamicRange dB are treated as silent
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 reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
plot	if TRUE, produces a plot of the results
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
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 aliasing artifacts)
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 waveform as a numeric vector on the original or dB scale; otherwise, does not return anything.

Examples

```
sound = sin(1:2000/10) * approx(c(1, .01, .5), n = 2000)$y

# 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

# Wave object
data(speechEx, package = 'soundgen')
osc(speechEx, from = 2, to = 2.5, dB = TRUE)

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

## Not run:
# saves oscillograms of all audio files in a folder
osc('~/Downloads/temp', savePlots = TRUE)

## End(Not run)
```

---

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 = NULL,
  overlap = 50,
  timeLag = NULL,
  theilerWindow = NULL,
  nonlinStats = c("ed", "d2", "ml", "sur"),
  ed_pars = list(max.embedding.dim = 15),
 d2_pars = list(min.embedding.dim = 2, min.radius = 0.001, n.points.radius = 20),
  ml_pars = list(min.embedding.dim = 2, radius = 0.001),
  sur_pars = list(FUN = nonlinearTseries::timeAsymmetry, K = 20),
```

```
  bw = 0.01,
  bins = 5/bw,
  reportEvery = NULL,
  cores = 1,
  rasterize = FALSE,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  colorTheme = "bw",
  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`	length of the analysis window, ms
`step`	step between successive windows, ms; if provided, overrides `overlap`; 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 - e.g., 24.98866 instead of 25.0 ms
`overlap`	overlap between successive windows, %
`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`. If automatic estimation fails, defaults to 1 sample
`theilerWindow`	time lag between two points that are considered locally independent and can be treated as neighbors in the reconstructed phase space (ms). Converted internally to samples. 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 can be slow. Set to `NULL` or `character(0)` to calculate only the phasegram and basic descriptives. The default is to compute all available nonlinear statistics

ed_pars	a list of control parameters passed to estimateEmbeddingDim. If ed_pars$time.lag is NULL, it is set to the estimated time lag in samples
d2_pars	a list of control parameters passed to corrDim. If d2_pars$time.lag, d2_pars$max.radius, or d2_pars$theiler.window are NULL, they are filled in automatically
ml_pars	a list of control parameters passed to maxLyapunov. If ml_pars$time.lag or ml_pars$theiler.window are NULL, they are filled in automatically
sur_pars	a list of control parameters passed to surrogateTest
bw	standard deviation of the smoothing kernel, as in density. Must be a single positive finite number
bins	the number of bins along the Y axis after rasterizing (has no effect if rasterize = FALSE). Coerced to an integer of at least 2
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
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. The rasterized matrix is returned even if plot = FALSE
plot	if TRUE, produces a plot of the results
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
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 density values in orig are normalized by the global maximum across all frames. The resulting phasegram can optionally be rasterized to smooth it for plotting (output: rasterized); the rasterized matrix is additionally normalized row by row for display.

Value

For a single input, a list of three components:

orig the full phasegram as a data frame. $time is the middle of each frame (ms), $x is the coordinate along the Poincare section (approximately on the normalized audio scale), and $y is the density of intersections of system trajectories with the Poincare section, normalized by the global maximum across all frames. Failed or flat frames are represented by NA rows

rasterized the rasterized phasegram as a numeric matrix, or NULL if rasterize = FALSE. Rows correspond to time frames and columns correspond to bins along the Poincare-section coordinate. Values are row-normalized for display, so each non-empty row has a maximum of 1. If no valid Poincare intersections are found, a zero-valued matrix is returned

descriptives per-frame descriptives as a data frame. Always included are time (ms), shannon = normalized Shannon entropy of Poincare sections, and nPeaks = log-normalized number of peaks in the density distribution of Poincare sections. If requested via nonlinStats, also includes ed = optimal number of embedding dimensions, d2 = correlation dimension, ml = maximum Lyapunov exponent (positive values suggest chaos), and sur = stochasticity index from surrogate data testing, rescaled to approximately 0 = deterministic and 1 = stochastic

For multiple inputs, a list of such per-input results is returned.

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), noise = -30, jitterDep = .4,
  rolloff = -5, addSilence = 0, plot = TRUE)
ph2 = phasegram(target2, 16000, nonlinStats = NULL)

# low-pass filtering may help a bit
target2_lowpass = bandpass(target2, 16000, upr = 2500)
phasegram(target2_lowpass, 16000, nonlinStats = NULL)

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), plot = TRUE, yScale = 'bark')
phasegram(s2, 16000, windowLength = 10, nonlinStats = NULL, bw = .001)
phasegram(s2, 16000, windowLength = 10, nonlinStats = NULL, bw = .02)

## End(Not run)
```

`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 https://cogsci.se/soundgen/acoustic_analysis.html for help and examples. You can verify the pitch contours first, and then feed them back into `analyze` (see examples). Use hotkeys (eg spacebar to play/stop) and avoid working with very large files.

Usage

```
pitch_app(...)
```

Arguments

`...`	presets like `windowLength = 25, pitchMethods = c('autocor','cep')`. Full list: dynamicRange, zp, nCands, minVoicedCands, domThres, domSmooth, autocorThres, autocorSmooth, autocorUpsample, autocorBestPeak, cepThres, cepZp, specThres, specPeak, specRatios, specHNRslope, specSmooth, specMerge, specSinglePeakCert, hpsThres, hpsNum, hpsNorm, hpsPenalty, zcThres, zcWin, certWeight, smooth, interpolCert, spec_maxPoints, specContrast, specBrightness, blur_freq, blur_time, reass_cex, osc_maxPoints, windowLength, step, silence, pitchFloor, pitchCeiling, priorMean, priorSD, shortestSyl, shortestPause, interpolWin, interpolTol, spec_cex, nColors, reass_windowLength, reass_step, pitchMethods, summaryFun, summaryFun_text, spec_ylim, spec_colorTheme, osc, wn

Value

A list with the last used settings ($settings) plus the output of analyze() for each file from the last file queue with two additional columns: "time" and "pitch". NB: only the results of the most recent file queue are returned, so don't press "Load audio" repeatedly if you need the output returned to R (the csv file with results should still be saved correctly). When proceeding to the next file in the queue, the app saves to disk a backup .csv file with one row per audio file. When the orange "Download results" button is clicked, a context menu pops up offering to terminate the app - if that happens, the results are also returned directly into R. To process pitch contours further in R, work directly with `my_pitch[[myfile]]$detailed$time` and `my_pitch[[myfile]]$detailed$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 temporary backup file. If the app crashes or is closed without saving the results, this backup file preserves your data. To recover it, restart pitch_app() - a dialog box will pop up and ask whether you want to append the old data to the new one. Even so, save your data regularly to be on the safe side!

See Also

`formant_app` `annotation_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

# You can pass presets with your preferred parameter values:
my_pitch = pitch_app(windowLength = 20, step = 10,
  pitchMethods = c('dom', 'autocor', 'cep'), spec_ylim = c(0, 6))

# Object "my_pitch" contains the output, notably the time-pitch matrix
plot(my_pitch[[1]]$detailed$time, my_pitch[[1]]$detailed$pitch, type = 'b',
  xlab = 'Time, ms', ylab = 'Pitch, Hz')

# Run the app with previously used settings
my_pitch2 = do.call(pitch_app, my_pitch$settings)

# save the complete output, including the settings used
saveRDS(my_pitch2, 'my_pitch_analysis.rds')

# 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',  # needed for calculating 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
)

# 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)
```

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 = NULL,
  smoothBW = c(NA, 10, 1),
  inflThres = 0.2,
  summaryFun = c("mean", "sd"),
  extraSummaryFun = c(),
  ref = 16.3516,
  ptvStep = NULL,
  ptvTime = 0.5,
  ptvFreq = 1,
  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`	if NULL (default), guesses "ms" if step > 1 and "s" otherwise; specify "s" or "ms" explicitly to override
`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
`summaryFun`	summary function(s) to apply to the syllable descriptives (not to the pitch contours themselves)
`extraSummaryFun`	additional summary function(s) applied to pitch contours themselves (not to extracted pitch descriptives) 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 (16.3516 Hz)

`ptvStep`, `ptvTime`, `ptvFreq`
the instantaneous proportion of time vocalizing (PTV) is calculated by producing a binary (sound on/off) contour with a step of ptvStep ms and convolving it with a half-Gaussian filter with SD = ptvTime s ($ptv_conv) and by low-pass filtering it over ptvFreq Hz ($ptv_lowpass)

`plot` if TRUE, plots the inflections for manual verification

Value

A list with three elements: `summary` (a dataframe with columns containing summaries of one or multiple inputs, one input per row), `syllables` (a dataframe or list of dataframes with syllable timings, where a syllable is a contiguous non-NA run of the pitch contour), and `ptv` (a dataframe or list of dataframes with the instantaneous PTV contour). The descriptives in `summary` are as follows:

duration total duration, s

durDefined duration after omitting leading and trailing NAs

propDefined percentage of input with non-NA value, eg percentage 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,
  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(speechEx, package = 'soundgen')
a = analyze(speechEx)
pd1 = pitchDescriptives(a$detailed[, c('time', 'pitch')],
                        inflThres = NA, plot = TRUE)
pd2 = pitchDescriptives(a$detailed[, c('time', 'pitch')],
                        inflThres = c(0.1, 0.1, .5), plot = TRUE)

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

## End(Not run)
```

`pitchSmoothPraat` *Pitch smoothing as in Praat*

Description

Smooths an intonation (pitch) contour with a low-pass filter, as in Praat (http://www.fon.hum.uva.nl/praat/). Algorithm: interpolates missing values (voiceless 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 controls the width of the low-pass filter; larger values retain more high-frequency detail. Similar to a special case of low-pass filtering with `bandpass`, but faster and can handle NAs.

Usage

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

Arguments

pitch	numeric vector of pitch values (NA = voiceless)
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
preprocess	if TRUE, makes input non-negative and handles NA; if FALSE, expects a non-negative vector without any missing values (faster)

Value

Numeric vector of the same length as input.

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)
```

---

playme *Play audio*

Description

Plays one or more sounds: wav/mp3 file(s), Wave objects, or numeric vectors. Recommended players on Linux: "play" from the "SoX" library (default), "aplay" (ALSA).

Usage

```
playme(x, samplingRate = NULL, 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 options()$soundgen.player –> tuneR::getWavPlayer() –> OS-specific defaults: "play" on Linux, "afplay" on MacOS, and VLC –> FFmpeg –> PowerShell on Windows. In case of errors, try setting another default player (see examples).
from, to	play a selected time range (s)

Value

Does not return anything, only plays the audio.

Examples

```
## Not run:
# Play an audio file:
data('speechEx', package = 'soundgen')
playme(speechEx, from = 1, to = 2)
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

# Set default player for the current R session
# "play" (SoX library, default on Linux)
options(soundgen.player = 'play')
# "afplay" (ALSA, default on MacOS)
options(soundgen.player = 'afplay')
# VLC path (default on Windows)
options(soundgen.player = "C:\\Program Files\\VideoLAN\\VLC\\vlc.exe")
# view currently set player
options()$soundgen.player

# Or specify player per call
playme(sound, player = "C:\\Program Files\\VideoLAN\\VLC\\vlc.exe")

## 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 = "bw",
  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	colnames and rownames of ms, respectively
quantiles	numeric vector of cumulative contours in (0, 1). For example, 0.5 marks a region that contains 50% of the total modulation-spectrum amplitude. Set to NULL to omit contour lines
colorTheme	plot color theme
col	custom color palette; if supplied, overrides colorTheme
logWarpX, logWarpY	numeric vector of length 2: c(sigma, base) for pseudolog-warping the modulation spectrum in the plot, as in pseudo_log_trans() from the **scales** package. These arguments affect plotting only; the returned matrices are not log-warped
main, xlab, ylab, xlim, ylim	graphical parameters
audio	(internal) a list of audio attributes
extraY	if TRUE, another Y-axis is plotted on the right showing 1 / (Y-axis value) as a frequency in Hz (assuming that ms is derived from an ordinary STFT with linearly spaced frequency bins)

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

Value

Does not return anything, just plots the MS.

Examples

```
ms1 = modulationSpectrum(runif(4000), samplingRate = 16000, plot = TRUE)
plotMS(ms1$detailed$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$detailed$original,
  nr = nrow(ms1$detailed$original), nc = ncol(ms1$detailed$original))
# plot the difference
plotMS(log(ms1$detailed$original / ms2_resized), quantiles = NULL,
  col = colorRampPalette(c('blue', 'yellow')) (50))
```

---

presets *Presets*

Description

A library of presets for easy generation of a few representative sounds with soundgen.

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. Any NAs in pitch contour are treated as voiceless fragments, and their pitch is not modified. If the NAs are not actual silences / voiceless frames, they should be interpolated before passing the pitch contour to prosody().

Usage

```
prosody(
  x,
  samplingRate = NULL,
  multProsody,
  analyze_pars = list(),
  shiftPitch_pars = list(),
  pitchManual = NULL,
  reportEvery = NULL,
  cores = 1,
  play = FALSE,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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; numeric vector or anchor format `list(time = ..., value = ...)`

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 accepted - e.g., as returned by pitch_app

reportEvery
: when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime

cores
: number of cores for parallel processing

play
: if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme

saveAudio
: if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory

plot
: if TRUE, produces a plot of the results

savePlots
: if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed
: if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

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

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:
data('speechEx', package = 'soundgen')
samplingRate = speechEx@samp.rate
spectrogram(speechEx, yScale = 'log', ylim = c(.05, 4))
# playme(speechEx)

# start with exaggerated prosody, then flat towards the end
speech1 = prosody(speechEx,
  multProsody = list(time = c(0, 1), value = c(1.5, 0.1)),
  analyze_pars = list(windowLength = 40, step = 10,
  pitchMethods = c('dom', 'autocor', 'cep', 'spec')),
  shiftPitch_pars = list(freqWindow = 400))
spectrogram(speech1, samplingRate, yScale = 'log')
# playme(speech1, samplingRate)

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

## End(Not run)
```

`reportCI` *Report CI*

Description

A simple function for formatting and printing estimates with their confidence intervals or similar statistics that follow a "X [Y, Z]" format. Takes a numeric vector or matrix with three elements / columns: estimate, lower quantile from a CI, and upper quantile from a CI. For each row, it prints the result as "estimate [CI]".

Usage

```
reportCI(n, digits = 2, suffix = NULL)
```

Arguments

n	numeric vector or matrix-like object
digits	number of decimal points to preserve
suffix	a string to print between the first value and the values in []

Value

Does not return anything, just prints the estimate and CI.

Examples

```
n = rnorm(100)
reportCI(quantile(n, probs = c(.5, .025, .975)))

a = data.frame(fit = c(3, 5, 7),
               lwr = c(1, 4, 6.5),
               upr = c(5, 6, 7.1))
reportCI(a, 1)
reportCI(a, 1, ' cm')
reportCI(a, 1, '%, 95% CI')
```

---

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: "

Value

Does not return anything, just prints the time left.

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: ')
}

# 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)
}

## Not run:
# when analyzing a bunch of audio files, their size is a good estimate
# of how long each will take to process
time_start = proc.time()
filenames = list.files('~/Downloads/temp', pattern = "*.wav|.mp3",
  full.names = TRUE)
filesizes = file.info(filenames)$size
for (i in 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 of a sound or another time series without aliasing. Similar in spirit and speed to resample in the signal package, but also works for short vectors and preserves NAs

approximately after resampling. Use resample() for interpolation when the points are evenly spaced, when there are missing values, or when we care about aliasing. When the points are unevenly spaced, use interpolate instead, but then there should not be any missing values.

Usage

```
resample(
  x,
  samplingRate = NULL,
  samplingRate_new = NULL,
  mult = NULL,
  len = NULL,
  lowPass = TRUE,
  interpol = "splineFC",
  na.rm = FALSE,
  reportEvery = NULL,
  cores = 1,
  saveAudio = FALSE,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  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 known (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
`interpol`	the method of interpolation (only used for upsampling, not downsampling); see interpolate for details
`na.rm`	if TRUE, NAs are linearly interpolated before resampling; if FALSE they are preserved in the interpolated output in (approximately) the same position
`reportEvery`	when processing multiple inputs, report estimated time left every `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`

cores	number of cores for parallel processing
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
plot	if TRUE, produces a plot of the results
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
width, height, units, res	graphical parameters for saving plots passed to png

Details

Algorithm: to downsample, applies a low-pass filter, then decimates with `spline`; to upsample, performs a specified type of interpolation (defaults to monotone Fritsch-Carlson splines, see interpolate), 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 when `na.rm = FALSE` (see examples).

Value

A numeric vector.

See Also

interpolate interpolateNA

Examples

```
## Example 1: a short vector with NAs
x = c(NA, 1, 2, 3, NA, NA, 6, 9, 8, NA)

# downsample
print(resample(x, len = 5))  # NAs are propagated
print(resample(x, len = 5, na.rm = TRUE))  # NAs are interpolated
print(resample(x, mult = 1/2, na.rm = TRUE, plot = TRUE)) # same

# 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)
```

```
# change the method of interpolation
resample(x, mult = 15, lowPass = FALSE, interpol = 'pchip', plot = TRUE)
resample(x, mult = 15, lowPass = FALSE,
  interpol = interpol_loess(span = .6), plot = TRUE)

## 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 artifacts

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

## End(Not run)
```

`reverb` *Reverb & echo*

Description

Adds reverberation and/or echo to a sound parametrically or based on the impulse response recorded in a particular real environment. Algorithm for parametric reverb: adds time-shifted copies of the signal weighted by a decay function. In simple terms: we specify how much and when the sound rebounds back (e.g., from the walls of a room) 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",
  IR = NULL,
  filter = list(),
  dynamicRange = 80,
  output = c("audio", "detailed"),
  play = FALSE,
  saveAudio = FALSE,
  reportEvery = NULL,
  cores = 1,
  seed = 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 (NULL = no echo)
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 (NULL = no reverb)
reverbDensity	the number of echos or "voices" added
reverbType	so far only "gaussian" has been implemented
IR	recording of an impulse response in a particular environment - ideally, an instant loud click, but just clapping your hands or snapping your fingers also works as a first approximation
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 output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory

reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
seed	if NULL, the algorithm is stochastic; provide a positive integer for reproducibility

Value

If output = "audio", returns the processed waveform as a numeric vector. If output = "detailed", returns a list: $rvb_win = Gaussian window used to create the reverberation, $rvb = reverberation effect, $echo = echo effect, $effect = both reverb and echo mixed together, $audio = processed audio.

Examples

```
data('speechEx', package = 'soundgen')
sr = speechEx@samp.rate
s = speechEx@left[1:16000]
# playme(s)

# parametric reverb
s_rev = reverb(s, sr)
# playme(s_rev)

# reverb of a particular real place based on its recorded IR
data('IR', package = 'soundgen')
s_ir = reverb(s, sr, IR = IR)
# playme(s_ir)

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

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

# reverb (caves)
s3 = reverb(s, sr, echoDelay = NULL,
            reverbDelay = 600, reverbSpread = 1500,
            reverbLevel = -10, reverbDensity = 100)
playme(s3)
spectrogram(s3, 16000, ylim = c(0, 4))

# both echo and reverb with high frequencies emphasized
```

```
s4 = reverb(s, sr,
            echoDelay = 250, echoLevel = -20,
            reverbDelay = 70, reverbSpread = 120,
            reverbLevel = -25, reverbDensity = 50,
            filter = list(formants = NULL, lipRad = 3))
playme(s4)
spectrogram(s4, 16000, ylim = c(0, 4))

# recorded IR
s5 = soundgen(nSyl = 2, sylLen = 250, pitch = c(500, 300))
playme(s5)
spectrogram(s5, 16000, yScale = 'ERB')
s5r = reverb(s5, 16000, IR = IR)
playme(s5r)
spectrogram(s5r, 16000, yScale = 'ERB')

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

## 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 and `estimateVTL` for details on the algorithm for estimating formant dispersion if VTL is not known (note that `schwa` calls `estimateVTL` with the option `method = 'regression'`).

Usage

```
schwa(
  formants = NULL,
  vocalTract = NULL,
  formants_relative = NULL,
  nFormants = 8,
```

```
  interceptZero = TRUE,
  tube = c("closed-open", "open-open", "open-closed", "closed-closed"),
  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)
`nFormants`	the number of formants to estimate (integer)
`interceptZero`	if TRUE, forces the regression curve to pass through the origin, reducing the influence of highly variable lower formants (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

Value

A list with the following components:

vtl_measured VTL as provided by the user, cm

vtl_apparent VTL estimated based on formants frequencies provided by the user, cm

dF 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

References

- Stevens, K. N. (2000). Acoustic phonetics (Vol. 30). MIT press.
- Anikin, A., Barreda, S. & Reby, D. (2024) A practical guide to calculating vocal tract length and scale-invariant formant patterns. Behavior Research Methods 56, 5588–5604.

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, nFormants = 5)
schwa(vocalTract = 13, nFormants = 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 (%, semitones, or dF units) in observed vs expected
#  formant frequencies: s_a[c('ff_relative', 'ff_relative_semitones', 'ff_relative_dF')]
# [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 / beats separated by background noise in long recordings (up to 1-2 hours of audio per file). Syllables are defined as continuous segments that differ from background noise based on amplitude and/or spectral contrast. Bursts are defined as local maxima in the detection contour that are high enough relative to the surrounding region. A note on long recordings: an hour of audio takes ~1 min to process, but watch your memory usage and perhaps decrease `maxDur` to avoid running out of RAM. This has little effect on processing speed, but dramatically cuts memory requirements. Another advantage of using shorter `maxDur` is that signal-noise separation may improve because noise amounts and profiles can be estimated in each chunk, adjusting for variable recording conditions over time. The downside is that small segments at chunk boundaries may be missed.

Usage

```
segment(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  shortestSyl = 40,
  shortestPause = 40,
  input = c("mel", "env", "spec"),
  propNoise = NULL,
  SNR = NULL,
  specDiffMeasure = c("cosine", "logSpecDist", "specDiff"),
  amplWeight = c("add", "gate", "scale", "none"),
  noiseLevelStabWeight = c(1, 0.25),
  windowLength = 40,
  step = NULL,
  overlap = 80,
  reverb_pars = list(reverbDelay = 70, reverbSpread = 130, reverbLevel = -35,
    reverbDensity = 50, echoLevel = -Inf),
  interburst = NULL,
```

```
  peakToTrough = NULL,
  summaryFun = c("median", "sd"),
  maxDur = 120,
  ptvStep = NULL,
  ptvTime = 0.5,
  ptvFreq = 1,
  reportEvery = NULL,
  cores = 1,
  plot = FALSE,
  savePlots = FALSE,
  embed = FALSE,
  saveAudio = FALSE,
  addSilence = 50,
  main = NULL,
  xlab = "",
  ylab = NULL,
  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 to normalize the 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)
`input`	the contour used to search for syllables: 'env' = smoothed RMS amplitude envelope; 'spec' = power spectrum returned by `tuneR::melfcc`; 'mel' = mel-filterbank amplitude spectrum from `tuneR::melfcc`
`propNoise`	the proportion of analysis frames assumed to represent background noise, 0 to 1. If NULL, this proportion is estimated automatically in each chunk. If 0, correction for background noise is skipped, in which case `SNR` must be specified

`SNR` expected signal-to-noise ratio (dB above noise), which determines the threshold for syllable detection. If NULL, SNR is estimated automatically in each chunk unless `propNoise = 0`. The meaning of "dB" here is approximate because the detection contour may not be sound intensity, depending on `input`

`specDiffMeasure`
similarity measure used to compare each spectrum with the estimated noise spectrum (ignored when `input = 'env'`):

cosine cosine distance without centering

logSpecDist root-mean-square log-spectral distance

specDiff mean absolute log-spectral difference

`amplWeight` amplitude weighting applied to the spectral contrast (ignored when `input = 'env'`):

add combines spectral distinctiveness with amplitude elevation; the default behavior in soundgen v2.x when `specDiffMeasure = 'cosine'`

gate amplitude-gated contrast: spectral contrast contributes mainly when the frame is also above the estimated noise amplitude

scale spectral contrast scaled to the dynamic range of the amplitude contour

none unnormalized spectral contrast, mainly useful for diagnostics

`noiseLevelStabWeight`
a vector of length 2 specifying the relative weights of the overall signal level vs. stability (time derivative) when attempting to automatically locate the regions that represent noise. Increasing the weight of stability prioritizes sudden changes as marking the beginning and end of a syllable

`windowLength` length of the analysis window, ms

`step` step between successive windows, ms; if provided, overrides `overlap`; 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 - e.g., 24.98866 instead of 25.0 ms

`overlap` overlap between successive windows, %

`reverb_pars` 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. This is particularly useful if you have a recording of an impulse response in the same environment to pass on to `reverb`. The default disables echo by setting `echoLevel = -Inf`. Set to NULL to use a static detection threshold without reverb compensation

`interburst` minimum time between two consecutive bursts, ms. Also determines the analysis window used for detecting burst peaks. Defaults to the median detected `sylLen + pauseLen` in each chunk, falling back to `shortestSyl` if no syllables are detected

`peakToTrough` to qualify as a burst, a local maximum has to be at least `peakToTrough` dB above the surrounding contour. Defaults to `SNR + 3` dB in each chunk when `peakToTrough = NULL`

`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 for chunked files. Note that the noise profile is estimated in each chunk separately, so set it low if the background noise is highly variable

ptvStep, ptvTime, ptvFreq
: the instantaneous proportion of time vocalizing (PTV) is calculated by producing a binary (sound on/off) contour with a step of ptvStep ms and convolving it with a half-Gaussian filter with SD = ptvTime seconds ($ptv_conv), and by low-pass filtering it over ptvFreq Hz ($ptv_lowpass). ptvStep defaults to the same value as step

reportEvery
: when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime

cores
: number of cores for parallel processing

plot
: if TRUE, produces a plot of the results

savePlots
: if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed
: if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

saveAudio
: if TRUE, saves the extracted syllables in a "/segment" subdirectory created in the input directory (if input is a file or folder) or in the working directory

addSilence
: if syllables are saved as separate audio files, they are padded with this much silence before and after, ms

xlab, ylab, main
: main plotting parameters

showLegend
: if TRUE, shows a legend for thresholds

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

maxPoints
: maximum number of points to plot for the waveform and the segmentation contour. Longer contours are downsampled to avoid slow plotting

specPlot
: a list of graphical parameters for displaying the spectrogram (if input = '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

sylPlot
: a list of graphical parameters for displaying the syllable threshold

burstPlot
: a list of graphical parameters for displaying the bursts

...
: other graphical parameters passed to graphics::plot

Details

Algorithm: the sound is analyzed in chunks of at most `maxDur` seconds. In each chunk, the quietest and most stable regions are located, and a noise threshold is derived either from a user-specified proportion of noise (`propNoise`) or, if `propNoise = NULL`, automatically from the distribution of a weighted product of amplitude and stability. The detection contour is then compared against the estimated noise. If `input = 'env'`, the contour is a smoothed log RMS amplitude envelope. If `input = 'spec'` or `'mel'`, the contour is computed by comparing the spectrum of each frame with the estimated noise spectrum using `specDiffMeasure` and `amplWeight`.

Syllables are detected as continuous regions of the contour that exceed the noise threshold by approximately `SNR` dB. Pauses shorter than `shortestPause` are merged. Syllable start and end times correspond to the edges of envelope bins, and `pauseLen` is the time between the end of the previous syllable and the start of the current syllable.

Bursts are detected as local maxima of the contour above the syllable detection threshold. The minimum spacing/scale for burst detection is controlled by `interburst`, and the required peak prominence is controlled by `peakToTrough`.

Value

A list with the following components:

syllables data frame with columns `syllable`, `start`, `end`, `pauseLen`, `sylLen`, `sylRate`, and `ptv`. Times are in ms. `start` and `end` correspond to syllable edges; `pauseLen` is the time between the end of the previous syllable and the start of the current syllable. $ptv gives the proportion of time vocalizing for each syllable-pause pair.

bursts data frame with columns `time`, `ampl`, and `interburst`. `time` and `interburst` are in ms

summary data frame summarizing temporal descriptives per file, including global PTV = sum of all syllable durations / total duration of audio. NULL if `summaryFun = NULL`

ptv contour of instantaneous proportion of time vocalizing, with columns `time` (ms), `on` (1 if this frame is part of a syllable, 0 otherwise), `ptv_lowpass`, and `ptv_conv`. If PTV cannot be computed (e.g., if there are <2 syllables), this may be NA or 0

If more than one file is analyzed, `syllables`, `bursts`, and `ptv` are lists with one element per file, while `summary` remains a single data frame.

See Also

`analyze` `ssm` `segment_ann`

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)
# playme(sound, samplingRate = 16000)
```

```
s = segment(sound, samplingRate = 16000, plot = TRUE)
str(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')

# set SNR manually to control detection threshold
s = segment(sound, samplingRate = 16000, SNR = 1, plot = TRUE)

# simple intensity threshold (anything >5 dB is signal)
segment(sound, 16000, input = 'env', SNR = 5, plot = TRUE,
  # less smoothing gives more precise timing
  windowLength = 10, step = 5,
  # don't correct SNR based on estimated background noise
  propNoise = 0,
  # don't use dynamic thresholds to cancel reverb
  reverb_pars = NULL
)

# same with automatic threshold setting
segment(sound, 16000, input = 'env', plot = TRUE,
  windowLength = 10, step = 5, reverb_pars = NULL)

## Not run:
# plot the PTV contour (proportion of time vocalizing)
plot(s$ptv$time, s$ptv$on, type = 'l', xlab = 'Time, ms',
  ylab = 'Prop. time voc.')
points(s$ptv$time, s$ptv$ptv_conv, type = 'l', col = 'blue')
points(s$ptv$time, s$ptv$ptv_lowpass, type = 'l', col = 'red')
s$summary$ptv; mean(s$ptv$ptv_conv); mean(s$ptv$ptv_lowpass) # similar

# different ways to calculate instantaneous PTV
s2 = segment(sound, 16000, ptvTime = 2, ptvFreq = 5)
s3 = segment(sound, 16000, ptvTime = 0.05, ptvFreq = 0.5)
plot(s2$ptv$time, s2$ptv$on, type = 'l', xlab = 'Time, ms',
  ylab = 'Prop. time voc.')
points(s2$ptv$time, s2$ptv$ptv_conv, type = 'l', col = 'blue')
points(s2$ptv$time, s2$ptv$ptv_lowpass, type = 'l', col = 'yellow')
points(s3$ptv$time, s3$ptv$ptv_conv, type = 'l', col = 'purple')
points(s3$ptv$time, s3$ptv$ptv_lowpass, type = 'l', col = 'orange')

# segment all files in a folder and save the segments as separate files
segment('~/Downloads/temp',
  saveAudio = TRUE, savePlots = TRUE)

## End(Not run)
```

segment_ann *Extract annotated syllables*

Description

Takes a table of syllable annotations, formatted as the `syllables` dataframe returned by `segment`, extracts the corresponding audio segments, and saves each segment as a separate .wav file in a subdirectory named "segment".

Usage

```
segment_ann(
  syllables,
  x,
  samplingRate = NULL,
  scale = NULL,
  addSilence = 50,
  path = NULL
)
```

Arguments

`syllables`	data frame with at least columns `start` and `end` (ms), e.g. the `syllables` data frame returned by `segment`
`x`	path to a .wav or .mp3 file, a `Wave` object, or a numeric vector
`samplingRate`	sampling rate of `x`; only needed if `x` is a numeric vector
`scale`	maximum possible amplitude of `x`; only needed if `x` is a numeric vector. If NULL, the scale is guessed from the maximum observed amplitude
`addSilence`	silence added before and after each extracted segment, ms
`path`	optional full path to the directory in which to save the extracted segments. If NULL, a subdirectory named "segment" is created either in the folder containing the audio file (if `x` is a file) or in the working directory (if `x` is a `Wave` object or numeric vector)

Details

If `x` is a path to an audio file, the default output directory is `segment` inside the folder containing that file. If `x` is a `Wave` object or a numeric vector, the default output directory is `segment` inside the current working directory. Use `path` to override the output location.

Value

Does not return anything, just saves the files.

See Also

`segment`

Examples

```
sound = soundgen(nSyl = 3, sylLen = 100, pauseLen = 70,
                 pitch = 250, temperature = 0.001)
s = segment(sound, samplingRate = 16000, plot = FALSE)
s$syllables
# the annotation table should look like this, with start/end times in ms!

## Not run:
# extract and save all syllables
segment_ann(s$syllables, sound, samplingRate = 16000)

# save to a custom folder
segment_ann(s$syllables, sound, samplingRate = 16000,
            path = '~/Downloads/temp_segment')

## End(Not run)
```

`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, it is set to the median detected f0, which is slow and requires detectable pitch; if pitch detection fails, `freqWindow` defaults to 400 Hz with a message.

Usage

```
shiftFormants(
  x,
  multFormants,
  samplingRate = NULL,
  freqWindow = NULL,
  dynamicRange = 80,
  windowLength = 50,
  step = NULL,
  overlap = 75,
  wn = "gaussian",
  specEnvMethod = c("cepstral", "gauss", "movavg", "peak"),
  interpol = "splineFC",
  normalize = c("orig", "max", "none"),
  play = FALSE,
  saveAudio = FALSE,
  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: a single positive number, roughly the expected spacing between harmonics. Defaults to median detected f0
dynamicRange	regions under -dynamicRange dB are treated as silent
windowLength	length of the analysis window, ms
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
wn	window type accepted by winFun: character string or function
specEnvMethod	the method of extracting a smoothed spectral envelope: "cepstral" = Gaussian liftering of the real cepstrum (default), "gauss" = Gaussian blur of the log spectrum, "movavg" = moving average of the log spectrum, "peak" = moving maximum (upper envelope). See getSpecEnv for details
interpol	the method for interpolating scaled spectra: any method supported by interpolate, defaults to "splineFC"
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude for the given input scale, "none" = no normalization
play	if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing

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 envelope of the magnitude spectrum (the amount of

smoothing is controlled by freqWindow) and then dividing the complex spectrum by this envelope. To avoid amplifying noise, bins more than dynamicRange dB below the frame peak are divided by a floored envelope. This division 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.

Value

The processed waveform as a numeric vector with the original sampling rate (a list if there are multiple inputs).

See Also

shiftPitch transplantFormants getSpecEnv

Examples

```
data('speechEx', package = 'soundgen')
# playme(speechEx)
# spectrogram(speechEx)

# Lower formants by 4 semitones or ~20% = 2 ^ (-4 / 12)
speech1 = shiftFormants(speechEx, multFormants = 2 ^ (-4 / 12), freqWindow = 150)
# playme(speech1, speechEx@samp.rate)
# spectrogram(speech1, speechEx@samp.rate)

orig = meanSpectrum(speechEx, plot = FALSE)
shifted = meanSpectrum(speech1, speechEx@samp.rate, plot = FALSE)
plot(ampl ~ freq, orig, log = 'y', type = 'l')
lines(ampl ~ freq, shifted, col = 'blue')

# dynamic change: raise formants at the beginning, lower at the end
speech2 = shiftFormants(speechEx, multFormants = c(1.3, .7), freqWindow = 150)
# playme(speech2, speechEx@samp.rate)
# spectrogram(speech2, speechEx@samp.rate)
```

---

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. Likewise, use the much faster timeStretch for slowing down or speeding up a recording without preserving either pitch or formants. Tip: increase overlap to >90% for best quality.

Usage

```
shiftPitch(
  x,
  samplingRate = NULL,
  multPitch = 1,
  multFormants = multPitch,
  timeStretch = 1,
  freqWindow = NULL,
  dynamicRange = 80,
  windowLength = 40,
  step = NULL,
  overlap = 90,
  wn = "hanning",
  interpol = "splineFC",
  propagation = c("time", "adaptive"),
  specEnvMethod = c("cepstral", "gauss", "movavg", "peak"),
  preserveEnv = FALSE,
  transplantEnv_pars = list(windowLength = 10),
  normalize = c("orig", "max", "none"),
  play = FALSE,
  saveAudio = FALSE,
  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)
multPitch	1 = no change, >1 = raise pitch (eg 1.1 = 10% up, 2 = one octave up), <1 = lower pitch. Anchor format accepted for multPitch / multFormants / timeStretch (see soundgen)
multFormants	1 = no change, >1 = raise formants (eg 1.1 = 10% up, 2 = one octave up), <1 = lower formants. The default behavior is for formants to follow pitch with multFormants = multPitch
timeStretch	1 = no change, >1 = longer, <1 = shorter
freqWindow	the width of spectral smoothing window, Hz - see shiftFormants for discussion and examples. Defaults to detected f0 prior to pitch shifting times mean(multPitch). Note that formant shifting is applied AFTER pitch shifting, so freqWindow should be multiplied by multPitch compared to what you would use on the original audio
dynamicRange	dynamic range (dB) that silences spectrogram regions below -dyamicRange dB and also control the tol parameter in istft_timevar
windowLength	length of the analysis window, ms

step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
wn	window type accepted by winFun: character string or function
interpol	the method for interpolating scaled spectra and anchors: any method supported by interpolate, defaults to "splineFC"
propagation	the method for propagating phase: "time" (default) = single-pass horizontal propagation, several times faster; "adaptive" = a modified "vocoder done right" (Prusa & Holighaus 2017), high-quality but relatively slow
specEnvMethod	the method of extracting a smoothed spectral envelope: "cepstral" = Gaussian liftering of the real cepstrum (default), "gauss" = Gaussian blur of the log spectrum, "movavg" = moving average of the log spectrum, "peak" = moving maximum (upper envelope). See getSpecEnv for details
preserveEnv	if TRUE, transplants the amplitude envelope from the original to the modified sound with transplantEnv. Mostly this makes sense if there is no time stretching
transplantEnv_pars	a list of parameters passed on to transplantEnv if preserveEnv = TRUE
normalize	"orig" = same as input (default), "max" = maximum possible peak amplitude for the given input scale, "none" = no normalization
play	if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to playme as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for playme
saveAudio	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
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, the recipient's spectral envelope is flattened and the donor envelope is imposed on the same spectrogram before iSTFT. See Prusa 2017 "Phase vocoder done right", Royer 2019 "Pitch-shifting algorithm design and applications in music".

Value

The processed waveform as a numeric vector (a list if there are multiple inputs).

See Also

shiftFormants transplantFormants

Examples

```
data(speechEx, package = 'soundgen')
samplingRate = speechEx@samp.rate
# playme(speechEx)
spectrogram(speechEx, yScale = 'ERB')

# raise pitch, lower formants
s1 = shiftPitch(speechEx, freqWindow = 200, multPitch = 1.2, multFormants = .85)
# spectrogram(s1, samplingRate, yScale = 'ERB')
# playme(s1, samplingRate)

## Not run:
# Tips for best quality: high overlap (slow), pad with silence
# and fade a bit before processing; try adaptive phase propagation (slow)
s = c(
  rep(0, 1000),
  fade(speechEx@left, samplingRate, fadeIn = 50, fadeOut = 50),
  rep(0, 1000)
)
s1a = shiftPitch(s, samplingRate, freqWindow = 200, multPitch = 1.2,
  multFormants = .85, propagation = 'adaptive', overlap = 95)
spectrogram(s1a, samplingRate, yScale = 'ERB')
playme(s1a, samplingRate)
# cf:
playme(s1, samplingRate)

## Dynamic manipulations
# Add a chevron-shaped contour to both pitch and formants
s2 = shiftPitch(speechEx, multPitch = c(1.1, 1.3, .8))
playme(s2, samplingRate)
spectrogram(s2, samplingRate, yScale = 'ERB')

# Time-stretch only the middle
s3 = shiftPitch(speechEx, overlap = 95, timeStretch = list(
  time = c(0, .25, .31, .5, .55, 1),
  value = c(1, 1, 3, 3, 1, 1))
)
playme(s3, samplingRate)

# Raise pitch and formants by 3 semitones, shorten
s4 = shiftPitch(speechEx, multPitch = 2 ^ (3 / 12), timeStretch = 0.75)
playme(s4, samplingRate)
spectrogram(s4, samplingRate, yScale = 'ERB')

# Just speed up
shiftPitch(speechEx, multPitch = 1, timeStretch = 0.75, play = TRUE)

# Raise pitch, preserve formants
```

```
s5 = shiftPitch(speechEx, multPitch = 1.2, multFormants = 1, freqWindow = 150)
playme(s5, samplingRate)
spectrogram(s5, samplingRate, yScale = 'ERB')

# Only modify voiced frames, preserving consonants / breathing etc
s1 = soundgen(nSyl = 2, sylLen = 300, pauseLen = 500, pitch = c(250, 200))
s2 = soundgen(sylLen = 150, noise = 0, pitch = NA, formants = list(f1 = 5000))
s = addVectors(s1, s2, insertionPoint = 16000 * .5)
s = s + rnorm(length(s), 0, .01)
spectrogram(s, 16000, yScale = 'ERB')
playme(s, 16000)  # we want to ignore the central /tch/ when shifting f0
# run analyze to get f0 contours, correct manually with pitch_app if needed
an = analyze(s, 16000, windowLength = 25, step = 5, plot = TRUE, yScale = 'ERB')
pitch = an$detailed$pitch
multPitch = ifelse(is.na(pitch), 1, 1.25)
s_shifted = fade(shiftPitch(s, 16000, multPitch = multPitch, wn = 'hanning', overlap = 90))
spectrogram(s_shifted, 16000, yScale = 'ERB')
playme(s_shifted, 16000)  # the /tch/ in the middle is unaffected by pitch-shifting

## End(Not run)
```

---

soundgen *Generate a sound*

Description

Generates a sequence ("bout") of one or more vocalizations ("syllables") with pauses between them. Two basic components are synthesized: a periodic component (the sum of sine waves with frequencies that are multiples of the fundamental frequency) and an aperiodic noise component. Both components can be filtered with independently specified vocal tract transfer functions ("formants"). Intonation and amplitude contours can be applied both within each syllable and across multiple syllables. Suggested application: synthesis of animal calls and human nonverbal vocalizations (not speech). For more information, see https://cogsci.se/soundgen/sound_generation.html.

Usage

```
soundgen(
  repeatBout = 1,
  nSyl = 1,
  sylLen = 500,
  pauseLen = 200,
  addSilence = 100,
  ampl = NA,
  amplGlobal = NA,
  attackLen = 50,
  pitch = c(100, 150, 100),
  pitchGlobal = NA,
  rolloff = -12,
  rolloffOct = 0,
```

```
rolloffKHz = 0,
rolloffExact = NULL,
glottis = 0,
pitchFloor = 1,
pitchCeiling = 3500,
pitchSamplingRate = 16000,
noise = NULL,
rolloffNoise = -4,
noiseFlatSpec = 1200,
rolloffNoiseExp = 0,
formants = c(860, 1430, 2900, 4100),
formantsNoise = NA,
formantDep = 1,
formantDepStoch = 1,
formantWidth = 1,
formantCeiling = NULL,
formantLocking = 0,
vocalTract = NA,
mouth = NULL,
lipRad = 6,
noseRad = 4,
mouthOpenThres = 0,
amDep = 0,
amFreq = 30,
amType = c("logistic", "sine"),
amShape = 0,
vibratoFreq = 5,
vibratoDep = 0,
jitterDep = 0,
jitterLen = 1,
shimmerDep = 0,
shimmerLen = 1,
subRatio = 2,
subDep = 0,
nonlinBalance = 100,
nonlinRandomWalk = NULL,
shortestEpoch = 300,
temperature = 0.025,
tempEffects = list(),
maleFemale = 0,
creakyBreathy = 0,
plot = FALSE,
play = FALSE,
saveAudio = FALSE,
invalidArgAction = c("adjust", "abort", "ignore"),
smoothing = list(interpol = "splineFC", discontThres = 0.05, jumpThres = 0.01),
samplingRate = 16000,
windowLength = 50,
```

```
  step = NULL,
  overlap = 75,
  wn = "gaussian",
  dynamicRange = 80,
  ...
)
```

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 to overlap them: force with invalidArgAction = 'ignore') (vectorized). If there are multiple bouts, the first value of `pauseLen` is used at the pause between bouts

`addSilence` silence before and after the bout, ms: a vector of length 1 (symmetric) or 2 (different duration of silence before/after the sound)

`ampl` amplitude envelope (dB, 0 = max amplitude): a numeric vector or anchor format

`amplGlobal` global amplitude envelope spanning multiple syllables (dB, 0 = no change) (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)

`pitch` fundamental frequency within one syllable (anchor format). NAs in pitch vectors are accepted and are converted into voiceless fragments, but then set `nSyl = 1`

`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)

`rolloff` the rate at which f0 harmonics in the spectrum become weaker, dB/oct (anchor format for all rolloff-related parameters). More negative rolloff = weaker upper harmonics; see `getRolloff` for more details

`rolloffOct`, `rolloffKHz` rolloff may be constant throughout the spectrum, or it may vary with each octave above f0 (rolloffOct) or per kHz increase in f0 above the baseline of 200 Hz (rolloffKHz); for both parameters, positive values mean the rate of rolloff increases toward upper frequencies)

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

`glottis` duration of the closed phase of a glottal cycle (silent) in relation to the open phase, % (0 = no closed phase, 100 = closed phase as long as open phase, 200 = twice as long as the open phase, etc.); numeric vector or dataframe specifying time and value (anchor format). Use this effect sparingly: it is slow, and high values may affect harmonic composition and require a high sampling rate, especially in combination with high pitch

`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

`noise`
: intensity of turbulent noise (0 dB = same RMS as that of the periodic (voiced) component, negative values = less intense; anchor format). In soundgen 3.0, the noise component is always calibrated relative to the filtered harmonic component. When noise is present, the harmonic and noise components are filtered separately, their RMS amplitudes are normalized after filtering, and they are then mixed. Because noise can begin before the voiced part and continue after it, the time of noise anchors MUST be in ms, not [0, 1]; this is different from all other soundgen arguments that accept the anchor format with time either in ms or [0, 1]

`rolloffNoise`, `rolloffNoiseExp`, `noiseFlatSpec`
: linear (rolloffNoise, dB/kHz, anchor format) or exponential (rolloffNoiseExp, dB/oct, anchor format) rolloff of the excitation source for the noise component (anchor format) applied above `noiseFlatSpec` (Hz, scalar). More negative rolloff = less high-frequency noise

`formants`
: a vector of formant frequencies (assuming formants are static throughout the sound); a list of formant times, frequencies, amplitudes, and bandwidths; or a character string referring to default presets for speaker "M1" (implemented: "aoieu0"). NA or NULL means no formants, only lip radiation (but a schwa is generated if `vocalTract` is specified). Time stamps for formants and mouth can be specified in ms relative to `sylLen` or on a scale of `[0, 1]`. See `getFormantFilter` for more details

`formantsNoise`
: the same as `formants`, but for the aperiodic noise rather than for the periodic component. If NA (default), the noise will be filtered through the same formants as the periodic component, approximating aspiration noise `[h]`

`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 to avoid losing energy in the upper part of the spectrum due to unmodeled resonances above the Nyquist frequencies, specified in multiples of Nyquist. If NULL (default), a faster theoretical correction is used (may fail for unusual sounds)

`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). In multisyllable sounds, formant locking is applied separately to each syllable using the corresponding portion of the bout-level formant filter

`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)

`mouth` mouth opening (0 to 1, 0.5 = neutral, i.e. no modification) (anchor format)

`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

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

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

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

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

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

`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)

`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)

`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.

`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)

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

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

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

`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 an adjustment of voice quality from creaky (-1) to breathy (+1); 0 = no change

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 if TRUE, saves the result as "soundgen.wav" in the working directory

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

smoothing a list of parameters passed to interpolate to control the interpolation and smoothing of contours drawn through anchors

samplingRate sampling rate of the output (Hz)

windowLength length of the FFT window (ms)

step step between successive windows (ms); if provided, overrides overlap

overlap overlap between successive windows (0–100%)

wn wn window type accepted by winFun: character string or function

dynamicRange dynamic range (dB). Harmonics and noise more than dynamicRange under maximum amplitude are discarded to save computational resources

... other plotting parameters passed to spectrogram

Value

The synthesized waveform as a numeric vector normalized to [-1, 1] and sampled at the specified samplingRate. Note that the sampling rate may be increased internally in case the pitch is high, with a warning.

Parameter groups

Temporal structure repeatBout, nSyl, sylLen, pauseLen, addSilence

Amplitude ampl, amplGlobal, attackLen

Pitch & periodic source pitch, pitchGlobal, rolloff, rolloffOct, rolloffKHz, rolloffExact, glottis, pitchFloor, pitchCeiling, pitchSamplingRate

Aperiodic source noise, rolloffNoise, rolloffNoiseExp, noiseFlatSpec

Filter formants, formantsNoise, formantDep, formantDepStoch, formantWidth, formantCeiling, formantLocking, vocalTract, lipRad, noseRad, mouth, mouthOpenThres

Amplitude & frequency modulation amDep, amFreq, amType, amShape, vibratoFreq, vibratoDep

Nonlinear phenomena jitterDep, jitterLen, shimmerDep, shimmerLen, subRatio, subDep, nonlinBalance, nonlinRandomWalk, shortestEpoch

Stochasticity & hyper-parameters temperature, tempEffects, maleFemale, creakyBreathy

I/O plot, play, saveAudio, ...

Technical smoothing, invalidArgAction, samplingRate, windowLength, step, overlap, wn, dynamicRange

Anchor format

soundgen() and some other functions in this package accept arguments that define a time series (e.g., pitch or amplitude contour). These arguments can be provided as either numeric vectors or a list of anchors - points through which the contour is interpolated. This "anchor format" can be a dataframe or list with two elements: `$time` (ms or 0 to 1) and `value` of each anchor. Ex.: `soundgen(pitch = list(time = c(0, 0.2, 1),value = c(310, 340, 280)))`.

See Also

`getFormantFilter` `beat`

Examples

```
# Detailed documentation: https://cogsci.se/soundgen/sound_generation.html

# A gallery of examples with code: https://cogsci.se/soundgen/demos.html

# A GUI for soundgen is available as a Shiny app.
# Type "soundgen_app()" to open it in your 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', etc. (see ?playme)

sound = soundgen(play = playback)
# spectrogram(sound, 16000)
# 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)),
  attackLen = 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:
# 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,
   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
 )

# soundgen plays Bach
dt = otherToHz(
  c('E4', 'D4', 'E4', 'C4', 'E4', 'B3', 'E4', 'A3', 'E4', 'G#3', 'E4',
  'A3', 'E4', 'B3', 'E4', 'C4', 'E4', 'E3', 'E4', 'F#3', 'E4', 'G#3',
  'E4', 'A3', 'E4', 'G#3', 'E4', 'A3', 'E4', 'B3', 'E4', 'C4'), 'notes')
out = numeric(0)
for (s in 1:length(dt)) {
  syl_s = soundgen(sylLen = 100, pitch = dt[s], rolloff = -15,
                   formants = c(750, 1400, 2900, 3800), noise = -45,
                   attackLen = 50, addSilence = 0, temperature = .01)
  syl_s = fade(soundgen:::matchLengths(syl_s, 0.1 * 16000), samplingRate = 16000)
  out = c(out, syl_s[1:(0.1 * 16000)])
}
spectrogram(out, 16000, yScale = 'ERB')
playme(out, 16000)
```

```
## End(Not run)
```

---

soundgen_app *Interactive sound synthesizer*

Description

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

Usage

```
soundgen_app()
```

Value

Does not return anything.

See Also

soundgen

Examples

```
## Not run:
soundgen_app()  # opens the app in your default browser

## End(Not run)
```

---

specToMS *Spectrogram to modulation spectrum and back*

Description

specToMS() takes a spectrogram (complex or magnitude) and returns its 2D Fourier transform - the modulation spectrum (MS). msToSpec() is the inverse: it takes a full, complex modulation spectrum and reconstructs the magnitude spectrogram.

Usage

```
specToMS(spec, center = TRUE)

msToSpec(ms)
```

Arguments

spec	target spectrogram (numeric matrix, frequency in rows, time in columns), with frequency labels (kHz) in rownames and time labels (ms) in colnames
center	if TRUE, the MS is centered with DC in the middle
ms	centered modulation spectrum (complex matrix, spectral modulation in rows, temporal modulation in columns), as returned by specToMS()

Details

Both functions assume that the input carries valid labels: a spectrogram must have frequency labels (kHz) in its rownames and time labels (ms) in its colnames, and a modulation spectrum must have spectral modulation labels in its rownames and temporal modulation labels in its colnames. This is the case for objects produced by soundgen functions (e.g. spectrogram, stft_simple, modulationSpectrum), and it also ensures that zero-padding is reflected correctly in the labels.

Value

specToMS() returns a list with $ms (the uncentered MS) and $ms_centered (the centered MS, or NULL if center = FALSE). Both are complex matrices of the same dimension as spec, with spectral modulation (FM) in rows and temporal modulation (AM) in columns. msToSpec() returns a magnitude spectrogram (real matrix, frequency in rows, time in columns).

See Also

modulationSpectrum, plotMS

Examples

```
s = soundgen(sylLen = 200, amFreq = 25, amDep = 50, pitch = 250,
             addSilence = 0, samplingRate = 16000)
spec = spectrogram(s, samplingRate = 16000, windowLength = 25, step = 5,
  plot = FALSE)
image(t(spec))
ms = specToMS(spec)
plotMS(log(Mod(ms$ms_centered)), quantiles = NULL)
# round-trip: reconstruct the magnitude spectrogram
spec_new = msToSpec(ms$ms_centered)
image(t(spec_new))
```

---

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(
  spec,
  samplingRate,
  windowLength = 250,
  step = windowLength/2,
  specMethod = c("meanSpectrum", "spectrum")
)
```

Arguments

spec	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); this only has an effect if specMethod = "meanSpectrum"
specMethod	calls either meanSpectrum (faster) or spectrum (higher resolution)

Value

A modulation spectrum - a matrix of real values, with center frequencies of original filters in rows (kHz) and modulation frequencies in columns (Hz).

Examples

```
data(speechEx, package = 'soundgen')

# auditory spectrogram
as = audSpectrogram(speechEx, from = 0, to = 1,
  envelope = 'hil', nFilters = 24, step = NULL, plot = FALSE)
ms = specToMS_1D(as$filterbank_env, samplingRate = speechEx@samp.rate)
plotMS(log(ms+.01), logWarpX = c(10, 2), quantiles = NULL, ylab = 'kHz')

# ordinary STFT spectrogram
sp = spectrogram(speechEx, from = 0, to = 1, windowLength = 15, step = 0.5,
  output = 'original', plot = FALSE)
ms2 = specToMS_1D(sp, 1000 / 0.5)  # 1000/0.5 frames per s
plotMS(log(ms2+.01), quantiles = NULL, ylab = 'kHz')

## Not run:
# full resolution
ms_spec = specToMS_1D(as$filterbank_env, samplingRate = speechEx@samp.rate,
  specMethod = 'spectrum')
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 with 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, etc. Gallery of examples: https://cogsci.se/soundgen/spectrograms.html.

Usage

```
spectrogram(
  x,
  samplingRate = NULL,
  scale = NULL,
  from = NULL,
  to = NULL,
  dynamicRange = 80,
  windowLength = 50,
  step = NULL,
  overlap = 50,
  specType = c("spectrum", "reassigned", "derivative"),
  logSpec = if (specType == "derivative") FALSE else TRUE,
  rasterize = FALSE,
  wn = "gaussian",
  zp = NULL,
  smoothFreq = 0,
  smoothTime = 0,
  qTime = 0,
  percentNoise = 10,
  noiseReduction = 0,
  output = c("original", "processed", "complex", "all", "none"),
  specManual = NULL,
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  osc = c("linear", "dB", "none"),
  heights = c(3, 1),
  ylim = NULL,
  yScale = c("linear", "log", "bark", "mel", "ERB"),
  contrast = 0,
  brightness = 0,
  blur = 0,
  maxPoints = c(1e+05, 5e+05),
```

```
  padWithSilence = TRUE,
  colorTheme = "bw",
  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 to normalize the input vector (only needed if x is a numeric vector)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
dynamicRange	regions under -dynamicRange dB are treated as silent
windowLength	length of FFT window, ms (multiple values in a vector produce a multi-resolution spectrogram, in which case step should have the same length)
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
specType	plot the original FFT ('spectrum'), reassigned spectrogram ('reassigned'), or spectral derivatives ('derivative')
logSpec	if TRUE, log-transforms the spectrogram; no effect if specType = 'derivative' because then the spectrogram is always log-transformed
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 winFun: character string or function
zp	window length after zero padding, samples (a vector of the same length as windowLength). No padding is performed if zp < analysis window length in samples. If NULL, the signal is padded to a good number for speeding up the FFT. For ex., if windowLength = 10 and samplingRate = 22050, each frame is

22050 * 0.01 = 220.5 samples long, so we can set zp = 512 or just default to nextn(22050 * 0.01) by setting zp = NULL. NB: do not use zero-padding (set zp = 0) if you need to invert the spectrogram afterwards with istft_simple or invertSpectrogram

smoothFreq, smoothTime
: length of the window for median smoothing in frequency (Hz) and time domains (ms), respectively

qTime
: the quantile to be subtracted for each frequency bin. For ex., if qTime = 0.5, the median of each frequency bin (over the entire sound duration) will be calculated and subtracted from each frame (see examples)

noiseReduction, percentNoise
: 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. percentNoise can range from 0 to 100 (default 10), and it gives the percentage of frames classified as noise

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

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 reportEvery iterations (NULL = default, NA = don't report); see reportTime

cores
: number of cores for parallel processing

plot
: if TRUE, produces a plot of the results

savePlots
: if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

embed
: if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

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, 'mel' = mel, 'ERB' = Equivalent Rectangular Bandwidths

contrast
: controls the sharpness or contrast of the image: <0 = decrease contrast, 0 = no change, >0 increase contrast. Recommended range approximately (-1, 1). The spectrogram is raised to the power of exp(3 * contrast)

`brightness` makes the image lighter or darker, range [-1, 1] (default 0 = no change); for colorTheme = "bw", <0 = darker, >0 = lighter, range [-1, 1]. Values are remapped through a smooth sigmoid transfer curve that preserves the full color palette. To lighten or darken the palette itself, change the colors

`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 lead to unsharp masking with the default amount of 0.5 instead of blurring - see `gaussianSmooth2D`

`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 half a window length of silence to resolve the edges properly (not needed if your sound is already padded with some silence)

`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, but nonlinear yScale also warps the contour) that will be plotted over the spectrogram (eg pitch contour); can also be a list with extra graphical parameters such as `lwd`, `col`, `warp` (FALSE = plot as is, TRUE = warp to conform to nonlinear yScale), etc. (see examples)

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

`grid` if numeric, adds n = `grid` dotted lines per kHz and the same number of lines along the time axis

`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

Depends on the `output` argument. All matrices have frequency in rows (rownames in kHz) and time in columns (colnames in ms):

'none' Nothing: only produces and/or saves the plot.

'original' Unprocessed spectral magnitudes (not power!), with frequency bins from 0 up to and including Nyquist. If `specType = 'derivative'`, the matrix contains spectral derivatives; if `specType = 'reassigned'`, it contains either the rasterized reassigned spectrogram (matrix) or the unrasterized reassigned spectrogram - a dataframe giving time (ms), frequency (KHz), and magnitude of each bin).

'processed' The same matrix as `'original'` after denoising, time- / frequency-smoothing, blurring, and applying contrast and brightness.

'complex' The complete complex STFT spectrogram with both positive and negative frequencies (rows beyond Nyquist correspond to negative frequencies, as in `stft_simple`). This is the raw Fourier transform prior to reassignment and any post-processing, suitable for custom frequency-domain manipulation followed by inverse STFT with `istft_simple` (NB: use `zp = 0` if you intend to invert the spectrogram).

'all' A list with all of the above.

For multi-resolution spectrograms (`windowLength` with several values), the plot combines all resolutions, but the returned matrices correspond to the last value of `windowLength`.

See Also

`osc` `modulationSpectrum` `ssm`

Examples

```
# Gallery of examples: https://cogsci.se/soundgen/spectrograms.html

# 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,
  windowLength = c(5, 40),  # multi-resolution
  osc = 'dB',  # plot oscillogram in dB
  heights = c(2, 1),  # spectro/osc height ratio
  contrast = .25,  # increase contrast
  brightness = -.3,  # 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 = TRUE, 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')

# 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')
sp = spectrogram(s, 22050, specType = 'reassigned', windowLength = 5,
  step = 1, yScale = 'bark', output = 'original')
# ...or it can be rasterized, but that sacrifices frequency resolution:
sp2 = spectrogram(s, 22050, specType = 'reassigned', rasterize = TRUE,
                  windowLength = 5, step = 1, yScale = 'bark')

# an unrasterized reassigned spectrogram is just a dataframe of points
sp$z1 = soundgen:::zeroOne(log(sp$magn))
plot(sp$time, sp$freq, col = rgb(sp$z1, sp$z1, 1 - sp$z1, 1),
  pch = 16, cex = 0.5, 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)
spectrogram(s, 22050, windowLength = 1:10, yScale = 'bark',
  specType = 'reassigned', dynamicRange = 50, rasterize = TRUE)

# Different combinations of specType, mono/multiresolution, and rasterization
spectrogram(s, 22050, windowLength = 5)
spectrogram(s, 22050, windowLength = c(5, 10))

spectrogram(s, 22050, windowLength = 5, specType = 'reassigned',
  rasterize = FALSE)
spectrogram(s, 22050, windowLength = c(5, 10), specType = 'reassigned',
  rasterize = FALSE)
```

```
spectrogram(s, 22050, windowLength = 5, specType = 'reassigned',
  rasterize = TRUE)
spectrogram(s, 22050, windowLength = c(5, 10), specType = 'reassigned',
  rasterize = TRUE)

# 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 case, breathing)
spectrogram(sound, samplingRate = 16000, noiseReduction = 0.2)

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

# increase brightness (drops quiet bins with the same color palette)
spectrogram(sound, samplingRate = 16000, brightness = .5)

# another approach is to just make the palette lighter:
spectrogram(sound, samplingRate = 16000, col = gray.colors(30, 1, .5))

# median smoothing in time and/or frequency
spectrogram(sound, samplingRate = 16000, smoothFreq = 250)
spectrogram(sound, samplingRate = 16000, smoothTime = 100)
spectrogram(sound, samplingRate = 16000, smoothFreq = 250, smoothTime = 100)

# Gaussian filter to blur or sharpen the image in time and/or frequency
spectrogram(sound, samplingRate = 16000, blur = c(100, 50))
# 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(speechEx, package = 'soundgen')
sp = spectrogram(speechEx, windowLength = c(5, 25), yScale = 'ERB',
  maxPoints = c(1e4, 1e4),  # limit the number of pixels in osc/spec
  output = 'original')
nrow(sp) * ncol(sp) / 5e3  # spec downsampled (~54 times fewer pixels)

# Plot some arbitrary contour over the spectrogram (simply calling lines()
# will not work if the osc is shown because 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 = TRUE, extraContour = 'dom')
spectrogram(s, 16000, extraContour = an$detailed$dom,
  ylim = c(0, 2), yScale = 'bark')
spectrogram(s, 16000, extraContour = list(x = an$detailed$dom, col = 'green'),
  ylim = c(0, 2), yScale = 'bark')
# or simply add whatever you like to a spectrogram with points(), lines(),
```

```
# etc., (but only works without an oscillogram):
spectrogram(s, 16000, ylim = c(0, 2), yScale = 'bark', osc = 'none')
points(an$detailed$time/1000,  # time in s
       HzToOther(an$detailed$dom, 'bark'),  # values in barks
       lwd = 2, col = 'green', lty = 2)  # any graphic pars

# For values that are not in Hz, normalize any way you like. NB: if yScale !=
# 'linear', the extra contour is by default warped to the same scale b/c it
# is assumed to be in Hz. Specify "warp = FALSE" to avoid this
spectrogram(s, 16000, yScale = 'ERB', ylim = c(0, 5), extraContour = list(
  x = an$detailed$loudness / max(an$detailed$loudness, na.rm = TRUE) * 5000,
  # because ylim[2] = 5000 Hz
  type = 'b', pch = 5, lwd = 2, lty = 2, col = 'blue', warp = FALSE))
# compare:
spectrogram(s, 16000, yScale = 'ERB', ylim = c(0, 5), extraContour = list(
  x = an$detailed$loudness / max(an$detailed$loudness, na.rm = TRUE) * 5000,
  # because ylim[2] = 5000 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$detailed$modulation_spectrogram,
  colorTheme = 'matlab', ylab = 'Modulation frequency, kHz',
  contrast = .25, blur = c(10, 10), yScale = 'log')

## End(Not run)
```

---

spectrum *Spectrum*

Description

spectrum computes the frequency spectrum of a sound using the Fast Fourier Transform (FFT). For a smoother appearance and faster processing of long sounds, meanSpectrum computes the time-averaged spectrum of successive windows. See the "spec" and "meanspec" functions in the seewave package for more plotting options. NB: soundgen::spectrum() masks stats::spectrum().

Usage

```
spectrum(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  zp = NULL,
  yScale = c("linear", "power", "dB", "max0"),
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
```

```
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  reportEvery = NULL,
  cores = 1,
  ...
)

meanSpectrum(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  windowLength = 50,
  step = windowLength/2,
  overlap = NULL,
  wn = "gaussian",
  zp = NULL,
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  yScale = c("linear", "power", "dB", "max0"),
  savePlots = FALSE,
  embed = FALSE,
  main = NULL,
  xlab = NULL,
  ylab = 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)
zp	window length after zero padding, samples. No padding is performed if zp < analysis window length in samples. If NULL, the signal is padded to a good number for speeding up the FFT

yScale	scale of the y-axis: "linear" for linear amplitude, "power" for power spectrum, "dB" for decibels (20*log10), "max0" for dB with the maximum set to 0 dB
plot	if TRUE, plots the spectrum
savePlots	if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg "spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening
embed	if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed
xlab, ylab, main	graphical parameters for plotting
width, height, units, res	graphical parameters for saving plots passed to png
reportEvery	when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime
cores	number of cores for parallel processing
...	other graphical parameters passed to plot()
windowLength	length of the analysis window, ms
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
wn	window type accepted by winFun: character string or function

Value

A dataframe with two columns: freq (frequency in kHz) and ampl (amplitude, in units determined by yScale).

Examples

```
# 500 Hz tone
sound = cos(2 * pi * 500 * (1:4000) / 16000) + rnorm(4000, 0, .05)

# Spectrum on linear scale
spectrum(sound, samplingRate = 16000, yScale = 'linear')
meanSpectrum(sound, samplingRate = 16000, yScale = 'linear')

# dB scale with custom labels
spectrum(sound, samplingRate = 16000, yScale = 'dB', col = 'blue',
         xlab = 'Frequency (kHz)', ylab = 'Amplitude (dB)')

# max0 scale with custom y-limits and extra graphical pars
```

```
meanSpectrum(sound, samplingRate = 16000, yScale = 'max0',
  xlim = c(0, 2), ylim = c(-50, 5), lty = 2, lwd = 3, col = 'blue')

# Return data without plotting
ms = meanSpectrum(sound, samplingRate = 16000, plot = FALSE)
head(ms)

# If windowLength is longer than the sound, meanSpectrum() = spectrum()
spectrum(sound, 16000)
meanSpectrum(sound, 16000, windowLength = 5000)

## Not run:
# Process all .wav files in a folder
spectrum('~/Downloads/temp', savePlots = TRUE, yScale = 'dB')

## End(Not run)
```

---

speechEx *Speech example*

Description

A few seconds of speech

Usage

```
speechEx
```

Format

An object of class Wave of length 53563.

---

ssm *Self-similarity matrix*

Description

Calculates the self-similarity matrix and novelty vector of a sound. The self-similarity matrix is produced by comparing all pairs of frames in the input sound. Novelty is calculated by convolving the self-similarity matrix with a tapered checkerboard kernel. The positive lobes of the kernel represent coherence (self-similarity within the regions on either side of the center point) and the negative lobes anti-coherence (cross-similarity between these two regions). Since novelty is the dot product of the checkerboard kernel with the SSM, it is high when the two regions are self-similar (internally consistent) but different from each other.

Usage

```
ssm(
  x,
  samplingRate = NULL,
  from = NULL,
  to = NULL,
  specFun = "melspec",
  specFun_pars = list(),
  logSpec = FALSE,
  normalize = FALSE,
  simil = c("cosine", "cor"),
  kernelLen = 1000,
  kernelSD = 0.5,
  padWith = 0,
  ssmWin = 1,
  summaryFun = c("mean", "sd"),
  output = c("ssm", "novelty"),
  reportEvery = NULL,
  cores = 1,
  plot = TRUE,
  savePlots = FALSE,
  embed = FALSE,
  main = NULL,
  heights = c(2, 1),
  width = 900,
  height = 500,
  units = "px",
  res = NA,
  specPars = list(colorTheme = c("bw", "seewave", "heat.colors", "...")[2], xlab =
    "Time"),
  ssmPars = list(colorTheme = c("bw", "seewave", "heat.colors", "...")[2], xlab = "Time",
    ylab = "Time"),
  noveltyPars = list(type = "l", col = "black", lwd = 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)
from, to	if NULL (default), analyzes the whole sound, otherwise from...to (s)
specFun	the function used to extract a spectrogram-like feature matrix. Can be a string or a custom function that takes audio (numeric vector) as the first argument and returns a spectrogram-like matrix with time in columns and features in rows (see examples). A precomputed matrix is also accepted (features in rows, time [ms] in columns, numeric rownames for plotting). Supported strings:

stft_simple 'stft' / STFT / 'stft_simple' (amplitude spectrogram) Parameters in specFun_pars: samplingRate (optional if frequency and time labels are not needed), wl (samples), step (samples), wn, zp, padWithSilence.

spectrogram 'spectrogram' (amplitude spectrogram - a wrapper around stft_simple with more options). Parameters in specFun_pars: see spectrogram.

powspec 'powerspec' (power spectrogram with tuneR). Parameters in specFun_pars: wintime (s) or windowLength (ms), steptime (s) or step (ms), dither.

melfcc 'melspec' (mel-spectrogram), 'mfcc' / 'melfcc' (MFCCs). Parameters in specFun_pars: windowLength (ms), step (ms), nbands, maxfreq (Hz), MFCC (integer vector).

audSpectrogram 'audSpectrogram' / 'audSpec' (auditory spectrogram). Parameters in specFun_pars: see audSpectrogram.

getRMS 'getRMS' / 'rms' (RMS amplitude envelope). Parameters in specFun_pars: see getRMS.

getEnv 'getEnv' / 'env' ("various smoothed envelopes: RMS, analytical, peak, etc.). Parameters in specFun_pars: see getEnv.

specFun_pars — a list of parameters passed to specFun. Defaults for specFun = "melspec": list(windowLength = 125, step = 25, nbands = 50); defaults for specFun = "audSpec": list(nFilters = 16, step = 10)

logSpec — if TRUE, the input is log-transformed prior to calculating self-similarity

normalize — if TRUE, the spectrum of each STFT frame (or each column of feature matrix) is normalized to the same range prior to calculating self-similarity

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 evaluated over [-1, 1]: for ex., if kernelSD = 0.5, the kernel spans approximately ±2 SDs

padWith — how to treat edges when calculating novelty: NA = pad with NA (ignores edges in correlation), 0 = pad with zeros

ssmWin — window for averaging SSM, frames (has a smoothing effect and speeds up the processing)

summaryFun — functions used to summarize novelty, 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

output — what to include in $detailed (drop "ssm" to save memory when analyzing a lot of files); options: 'ssm', 'novelty', 'all'

reportEvery — when processing multiple inputs, report estimated time left every reportEvery iterations (NULL = default, NA = don't report); see reportTime

cores — number of cores for parallel processing

plot — if TRUE, plots the SSM

savePlots — if TRUE, creates a subdirectory in the input directory (if input is a file or folder) or in the working directory (if input is a vector etc), named after the function (eg

"spectrogram/"). All plots and audio files (if any) are saved in this new directory. If there are multiple inputs, an html notebook is also created for easy viewing and listening

`embed` if TRUE and savePlots is set and there are multiple inputs, all saved images and audio (if any) are embedded in the exported html notebook for easy sharing; if FALSE, the html file links to separate images and audio files (but separate files are still saved). NB: for this to work, package "base64enc" must be installed

`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

A list with two top-level elements: `$detailed` and `$summary`.

`$detailed` contains per-file results selected with the `output` argument. If multiple sounds are analyzed, `$detailed` is a named list of per-sound lists. If a single sound is analyzed, it is simplified to a single list. Each list may contain:

ssm self-similarity matrix

novelty novelty vector

`$summary` is a dataframe with summaries of novelty per sound (one row per file), or `NULL` if `summaryFun = NULL`.

References

- 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)
# more global features (captures the transition b/w the two sounds)
s2 = ssm(sound, samplingRate = 16000, kernelLen = 400)

s2$summary
s2$detailed$novelty  # novelty contour
## Not run:
ssm(sound, samplingRate = 16000,
    specFun = 'mfcc', simil = 'cor', normalize = TRUE,
    ssmWin = 10,  # 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))

# Custom input: produce a nice spectrogram first, then feed it into ssm()
sp = spectrogram(sound, 16000, windowLength = c(5, 40), contrast = .3,
  output = 'processed')  # return the modified spectrogram
ssm(sound, 16000, kernelLen = 400, specFun = sp)

# Custom input: use acoustic features returned by analyze()
an = analyze(sound, 16000, windowLength = 20, novelty = NULL)
feature_mat = t(an$detailed[, 4:ncol(an$detailed)]) # or select pitch, HNR, ...
feature_mat = t(apply(feature_mat, 1, scale))  # z-transform all variables
feature_mat[is.na(feature_mat)] = 0  # get rid of NAs
colnames(feature_mat) = an$detailed$time  # time stamps in ms
rownames(feature_mat) = 1:nrow(feature_mat)
image(t(feature_mat))  # not a spectrogram, just a feature matrix
ssm(sound, 16000, kernelLen = 500, specFun = feature_mat, logSpec = FALSE,
  specPars = list(ylab = 'Feature'))

## End(Not run)
```

`stft_simple` *STFT and inverse STFT*

Description

Short-Time Fourier Transform and its inverse for converting a signal between time and frequency domains. `stft_simple` returns the full complex spectrogram containing both positive and negative frequencies. Any final samples that do not fit into a complete frame are silently dropped unless `padWithSilence = TRUE`. `istft_simple` performs inverse STFT.

Usage

```
stft_simple(
  x,
  samplingRate = NULL,
  wl = 512,
```

```
  step = wl%/%2,
  wn = "gaussian",
  zp = 0,
  padWithSilence = FALSE
)

istft_simple(
  spec,
  wl,
  step,
  wn = "gaussian",
  wnSyn = c("wola", "rectangle"),
  type = c("half", "full"),
  fade = FALSE,
  tol = 1e-06
)

istft_timevar(
  spec,
  wl,
  step,
  wn = "gaussian",
  type = c("half", "full"),
  multPitch = 1,
  timeStretch = 1,
  fade = TRUE,
  tol = 1e-06
)
```

Arguments

x	numeric vector
samplingRate	sampling rate (Hz). If left NULL, time and frequency labels are not added (faster, but not by much)
wl	window length in samples (can be even or odd, but >=3). If wl exceeds the length of input vector, it is reset to wl = length(x). For maximum speed, set wl = `nextn(your_original_wl)`
step	step in samples
wn	window type accepted by `winFun`: character string or function
zp	window length after zero padding, samples. No padding is performed if zp < analysis window length in samples. If NULL, the signal is padded to a good number for speeding up the FFT
padWithSilence	if TRUE, pads the sound at both ends with half a window length of silence to resolve the edges properly and avoid dropping the last few samples (not needed if your sound is already padded with some silence)
spec	input complex spectrogram (rows = frequency, columns = time)

wnSyn	synthesis window: 'wola' = weighted overlap-add (WOLA) / STFT pseudoinverse, avoids spectral artifacts when the complex spectrogram is modified prior to iSTFT (default, recommended for most applications); 'rectangle' = no synthesis window, allows reconstructing the original signal exactly provided that the spectrogram is not modified between STFT and iSTFT
type	"full" = the full spectrogram returned by `stft_simple`; "half" = just the positive frequencies up to Nyquist (default)
fade	if TRUE, a linear fade-in and fade-out of length wl (but no more than 1/4 of input length) is applied to the output vector ("wola" method only)
tol	to avoid division by 0, denominator values smaller than `tol` are incremented by `tol` times sum of squared windows ("wola" method only)
multPitch	pitch multiplier interpolated across frames; > 1 –> raise pitch; must be positive
timeStretch	time stretch factor interpolated across frames; > 1 –> increase duration; must be positive

Details

If `wnSyn = "rectangle"`, no synthesis window is applied and the original signal can be recovered exactly, without any distortion at the beginning and the end (windows that taper to 0 give edge samples zero weight, so pad with silence to reconstruct them). The reconstruction is then exact for the covered portion of the signal when inverting an unnormalized windowed STFT produced by `stft_simple`, provided there is no zero-padding and the frame grid covers the signal. However, this works only if the complex spectrogram is not modified between STFT and iSTFT. When the spectrogram is modified - for example, when some filter is applied in the frequency domain - it is better to set `wnSyn = "wola"`, which applies the same windowing function to the iFFT of each frame. Finally, the much slower function `istft_timevar` works with time-variable step sizes in the context of dynamic pitch shifting or time stretching (see `shiftPitch`). The `zp` argument of `stft_simple()` is for analysis/display only. Spectrograms produced with `zp > wl` cannot be inverted by `istft_simple()` or `istft_timevar()` in their current form.

Value

`stft_simple` returns a complex spectrogram with both positive and negative frequencies as a matrix. If samplingRate is not NULL, row names give frequency in kHz and column names give time in ms. If samplingRate is NULL, no row or column names are added. Rows beyond the Nyquist frequency (half the sampling rate) correspond to negative frequencies. Time stamps correspond to the middle of each frame, starting with half a window length if `padWithSilence = FALSE` or starting at 0 ms if `padWithSilence = TRUE`. `istft_simple` returns a numeric vector.

Examples

```
## Ex. 1: obtaining a spectrogram
data(speechEx, package = "soundgen")
spec = stft_simple(speechEx@left[1:10000], samplingRate = speechEx@samp.rate,
  wl = 512, step = 100)
image(t(Mod(spec)))
# the spectrum of one frame
plot(as.numeric(rownames(spec)), Mod(spec[, 15]), type = 'l', xlab = 'Freq, kHz')
```

```
spec[1:5, 1:5]

# To get the positive frequencies only, use:
halfspec = spec[1:(nrow(spec) %/% 2 + 1), , drop = FALSE]
image(t(Mod(halfspec)))


## Ex. 2: manual narrow-band spectral filter to turn white noise into a whistle
samplingRate = 16000
wl = 317; step = 51
noise = fade(rnorm(samplingRate), samplingRate = samplingRate)
spec = stft_simple(noise, samplingRate, wl = wl, step = step, zp = 0)
spec = spec[1:(wl %/% 2 + 1), ]  # discard negative frequencies

# amplify one frequency band by 50 dB
spec_filtered = spec
spec_filtered[50, ] = spec_filtered[50, ] * 10^(50/20)
image(y = as.numeric(rownames(spec_filtered)), t(log(Mod(spec_filtered))))

# inverse STFT using wnSyn = 'wola'
noise_filtered = fade(istft_simple(spec_filtered, wl = wl, step = step,
  type = 'half', wnSyn = 'wola'), samplingRate = samplingRate)
spectrogram(noise_filtered, samplingRate)
# playme(noise_filtered, samplingRate)


## Ex. 3: reconstructing the input exactly with wnSyn = 'rectangle'
a = rnorm(64)
wl = 11; step = 3  # any wl and step are fine, even or odd
spec_full = stft_simple(a, wl = wl, step = step, wn = 'gaussian')
new_1 = istft_simple(spec_full, wl = wl, step = step, type = "full",
  wn = 'gaussian', wnSyn = 'rectangle')
plot(a, type = "l"); lines(new_1, col = 'green')
# note the missing bit at the end - incomplete final frame dropped

spec_half = spec_full[1:(nrow(spec_full) %/% 2 + 1), ]
new_2 = istft_simple(spec_half, wl = wl, step = step, type = "half",
  wn = 'gaussian', wnSyn = 'wola')  # wola is also exact here
plot(a, type = "l"); lines(new_2, col = 'green')


## Ex. 4: identity check for istft_timevar()
new_timevar = istft_timevar(spec_full, wl = wl, step = step,
  type = "full", multPitch = 1, timeStretch = 1, fade = FALSE)
plot(a, type = "l"); lines(new_timevar, col = 'green')
all(round(new_2, 5) == round(new_timevar, 5))  # should be TRUE
all(round(new_timevar, 5) == round(a[1:length(new_timevar)], 5))
# should be identical as well, except that two last samples in "a" are dropped
```

`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 compressing the sound (stretch < 1) may cause artifacts due to aliasing, since no anti-aliasing filter is applied (low-pass filter the input and/or output if needed). The `stretch` contour is interpreted over the output (stretched) duration, which is why the trajectory differs from `shiftPitch` with the same anchors.

Usage

```
timeStretch(
  x,
  stretch = 1,
  samplingRate = NULL,
  interpol = "splineFC",
  precision = 1000,
  play = FALSE,
  saveAudio = FALSE,
  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`). Must be positive
`samplingRate`	sampling rate of `x` (only needed if `x` is a numeric vector)
`interpol`	interpolation method supported by `interpolate`
`precision`	the number of points used for estimating the duration of output (more = better, but slower)
`play`	if TRUE, plays the output audio using the default player on your system. If a character string, it is passed to `playme` as the name of the player to use (e.g. 'aplay', 'play', 'vlc'). In case of errors, try setting another default player for `playme`
`saveAudio`	if TRUE, saves the processed audio in a subdirectory named after the function and created in the input directory (if input is a file or folder) or in the working directory
`reportEvery`	when processing multiple inputs, report estimated time left every `reportEvery` iterations (NULL = default, NA = don't report); see `reportTime`
`cores`	number of cores for parallel processing

Value

The processed waveform as a numeric vector for a single input or a list for multiple outputs. The duration is approximate: the output may end up a few samples shorter than intended if the resampling trajectory reaches the end of the input.

See Also

shiftPitch

Examples

```
data(speechEx, package = 'soundgen')  # import a recording
# playme(speechEx)
# spectrogram(speechEx)
s1 = timeStretch(speechEx, stretch = c(1, 3))
# playme(s1, speechEx@samp.rate)
# spectrogram(s1, speechEx@samp.rate)

# compare to a similar effect achieved with a phase vocoder in shiftPitch():
s2 = shiftPitch(
  speechEx,
  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, speechEx@samp.rate)
# spectrogram(s2, speechEx@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. The sounds 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. This is similar to "setenv" from the seewave package, but with a different smoothing algorithm and with a choice of several types of envelope: rms, hil, peak, etc. - see flatEnv.

Usage

```
transplantEnv(
  donor,
  recipient,
  samplingRateR = NULL,
```

```
  samplingRateD = samplingRateR,
  windowLength = 30,
  method = c("rms", "hil", "peak", "mean"),
  killDC = FALSE,
  dynamicRange = 30,
  plot = FALSE
)
```

Arguments

donor	the sound that "donates" the amplitude envelope
recipient	the sound that needs to have its amplitude envelope adjusted
samplingRateD, samplingRateR	sampling rate of the donor and recipient, respectively (only needed for vectors, not files); they don't hav to match
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

The recipient sound with the donor's amplitude envelope - a numeric vector with the same sampling rate and length as the recipient.

See Also

flatEnv

Examples

```
donor = c(rep(0, 50), rnorm(500)) * seq(1, 0, length.out = 550)
data('speechEx', package = 'soundgen')
recipient = speechEx@left[1000:4000]
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) is a crucial parameter: too little smoothing, and noise between harmonics will be amplified, creating 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; if pitch detection fails, freqWindow defaults to 400 Hz. Also ensure that both sounds have the same sampling rate. You may want to fade the output a little (a very short linear fade-in/out is applied internally).

Usage

```
transplantFormants(
  donor,
  recipient,
  samplingRate = NULL,
  freqWindow = NULL,
  specEnvMethod = c("cepstral", "gauss", "movavg", "peak"),
  dynamicRange = 80,
  windowLength = 50,
  step = NULL,
  overlap = 75,
  wn = "gaussian",
  normalize = c("orig", "max", "none")
)
```

Arguments

donor	either the sound that provides the formants (vector, Wave, or file) or the desired spectral filter (matrix) as returned by getFormantFilter or spectrogram - linear amplitude, frequency in rows, time in columns
recipient	the sound that receives the formants (vector, Wave, or file)
samplingRate	sampling rate (Hz) of both donor and recipient, which must match
freqWindow	the width of spectral smoothing window, Hz: a single positive number, roughly the expected spacing between harmonics. Defaults to the median pitch of the donor (or of the recipient if donor is a filter matrix); if pitch detection fails, defaults to 400 Hz with a message
specEnvMethod	the method of extracting a smoothed spectral envelope: "cepstral" = Gaussian liftering of the real cepstrum (default), "gauss" = Gaussian blur of the log spectrum, "movavg" = moving average of the log spectrum, "peak" = moving maximum (upper envelope). See getSpecEnv for details

dynamicRange	regions under -dynamicRange dB are treated as silent
windowLength	length of the analysis window, ms
step	step between successive windows, ms; if provided, overrides overlap; 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 - e.g., 24.98866 instead of 25.0 ms
overlap	overlap between successive windows, %
wn	window type accepted by winFun: character string or function
normalize	"orig" = same as donor / recipient (default), "max" = max possible amplitude of the donor given its scale (or of the recipient if donor is a filter matrix), "none" = no normalization

Details

Algorithm: makes spectrograms of both sounds, flattens the recipient spectrogram by dividing out its smoothed spectral envelope (obtained with getSpecEnv), smooths the donor spectrogram (or interpolates the supplied filter matrix) with getSpecEnv, multiplies the spectrograms, and transforms back into time domain with inverse STFT. To avoid amplifying noise, spectral bins more than dynamicRange dB below the peak of their frame are left untouched, and the original amplitude of each recipient frame is preserved. Anything more than dynamicRange dB below the global maximum is then zeroed out.

Value

The filtered waveform as a numeric vector with the original sampling rate, on a scale determined by the normalize argument and with the same duration as recipient.

See Also

transplantEnv getFormantFilter addFormants getSpecEnv shiftFormants shiftPitch

Examples

```
rec = rnorm(5000)  # white noise
donor = soundgen()  # voiced /a/
whisper = transplantFormants(donor = donor, recipient = rec,
  samplingRate = 16000, freqWindow = 300)  # whispered /a/
# playme(whisper)
meanSpectrum(whisper, 16000)

## Not run:
# Objective: take formants from one sound and apply them to another
s_orig = soundgen(pitch = 100, formants = 'ai')

recipient = soundgen(
  sylLen = 1200,
  pitch = c(100, 300, 250, 200),
  vibratoFreq = 9, vibratoDep = 1,
  formants = NULL,
```

```
  addSilence = 180,
  samplingRate = 16000,  # same as donor
  invalidArgAction = 'ignore')  # force to keep the low samplingRate
playme(recipient, 16000)
spectrogram(recipient, 16000)

s1 = transplantFormants(
  donor = s_orig,
  recipient = recipient,
  samplingRate = 16000)
playme(s1, 16000)
spectrogram(s1, 16000)

# The spectral envelope of s1 will be similar to that of the original on a
# frequency scale determined by freqWindow. Compare the spectra:
par(mfrow = c(1, 2))
meanSpectrum(s_orig, 16000, yScale = 'max0', ylim = c(-50, 0), main = 'Donor')
meanSpectrum(s1, 16000, yScale = 'max0', ylim = c(-50, 0),
             main = 'Processed recipient')
par(mfrow = c(1, 1))

# if needed, transplant amplitude envelopes as well:
s2 = transplantEnv(donor = s_orig, recipient = s1,
                   samplingRateR = 16000, samplingRateD = 16000,
                   windowLength = 10)
playme(s2, 16000)
spectrogram(s2, 16000)

## End(Not run)
```

---

winFun *Windowing functions for the Fourier transform*

Description

Generates an n-point window vector for the discrete Fourier transform and short-time Fourier analysis. Windows control spectral leakage by tapering the input signal, trading off main lobe width against sidelobe level. See the signal-package for more windows and/or different implementations. External and user-defined windowing functions can also be passed to all soundgen functions that call winFun - see examples.

Usage

```
winFun(n, wn, normalize = c("none", "amplitude", "energy"), ...)

winFun_tukey(fraction = 0.5)

winFun_gauss(sigma = 1/sqrt(24))
```

```
winFun_poisson(alpha = 1)

winFun_cauchy(alpha = 1)

winFun_kaiser(beta = 5)
```

Arguments

`n`	window length in samples (integer >= 3)
`wn`	window type (strings and functions can be passed to other soundgen functions that have a "wn" argument): **character string** name of a predefined window **function** a function that takes `n` and returns a window vector, e.g. a constructor such as `winFun_gauss(0.3)` or a user-defined function **numeric vector** a user-supplied window of length n
`normalize`	"none" = no normalization; "amplitude" = divide by sum of weights; "energy" = divide by sum of squared weights
`...`	extra arguments specific to parameterized windows, e.g. `sigma = 0.3` for `"gaussian"` (see "Predefined windows" for per-window arguments). Used only when `wn` is a character string; when `wn` is a constructor function, parameters are passed to the constructor directly. Do not pass graphical arguments here.
`fraction`	(winFun_tukey) fraction of the window that is tapered (0 = rectangular, 1 = Hann). Defaults to 0.5.
`sigma`	(winFun_gauss) standard deviation in units of half-window (positive). Defaults to $1/\sqrt{24} \approx 0.204$, matching the Praat/seewave Gaussian window.
`alpha`	(winFun_poisson, winFun_cauchy) shape parameter controlling the width. Defaults to 1.
`beta`	(winFun_kaiser) shape parameter controlling the trade-off between main lobe width and sidelobe level. `beta = 0` gives a rectangular window; `beta = 5` is similar to Hamming; `beta = 8.6` is similar to Blackman. Defaults to 5.

Value

A numeric vector of length n with values typically between 0 and 1 for `normalize = "none"` (or user-supplied values).

Predefined windows

"rectangle" Rectangular (uniform) window – no tapering. Minimizes mean square error, narrowest main lobe (-13 dB sidelobe). Good for transients or when leakage is not critical.

"hann" / "hanning" Hann window (raised cosine). Smooth ends, moderate sidelobe roll-off (-31.5 dB first sidelobe). Standard for general audio work.

"hamming" Hamming window. Almost cancels the first sidelobe of the Hann window (-43 dB). Popular in speech processing.

"blackman" Blackman window (3-term cosine sum). Lower sidelobes (-58 dB) than Hann/Hamming, slightly wider main lobe.

"flattop" Flat-top window (5-term cosine sum). Very wide main lobe but low sidelobes and excellent amplitude accuracy. Useful when precise amplitude measurement matters more than frequency resolution.

"bartlett" / "triangular" Bartlett (triangular) window – linearly tapering to zero. Simple, reasonably low sidelobes (-25 dB).

"welch" Welch (parabolic) window. Tapers to zero smoothly, sidelobe fall-off is asymptotic, close to sine window.

"parzen" Parzen window (cubic spline). Very smooth, continuous second derivative, excellent sidelobe roll-off. Common in non-parametric spectral density estimation.

"bspline" Cubic B-spline window (order 4). Extremely smooth; good when sidelobe structure must be suppressed heavily.

"bohman" Bohman window (time-domain convolution of two half-cosines). Sidelobes decay as $1/f^3$, flat at zero, very low leakage.

"blackmanharris" Blackman–Harris 4-term window (minimum sidelobe design). Very low sidelobes (-92 dB), suitable when dynamic range is important.

"nuttall" Nuttall 4-term window (symmetric, continuous first derivative). Slightly better sidelobes than Blackman–Harris in some metrics (-93 dB).

"barthannwin" Bartlett–Hann window (linear + cosine taper). Zero at edges, good compromise between Bartlett and Hann.

"sine" or "cosine" Half-cycle sine window $\sin(\pi k/(n-1))$. Simple, zero at edges, often used in audio coding (e.g., MP3).

"lanczos" Lanczos window (main lobe of sinc). Smooth, zero at edges, used in interpolation and image resampling.

"gaussian" Gaussian window. Minimizes time–frequency uncertainty; never quite reaches zero at the edges. The default `sigma = 1/sqrt(24)` (~= 0.204) reproduces the Praat/seewave window. Constructor: `winFun_gauss(sigma)`.

"tukey" Tukey (tapered cosine) window. "fraction" = fraction of the window devoted to cosine tapering (0 = rectangular, 1 = Hann). Constructor: `winFun_tukey(fraction)`.

"poisson" Poisson (exponential) window: $\exp(-\alpha\,|k - N/2|\,/\,(N/2))$. Sharp central peak, quick decay. Good for isolating fast transients. Constructor: `winFun_poisson(alpha)`.

"cauchy" Cauchy (Lorentzian) window: $1/(1 + (\alpha\,(k - N/2)/(N/2))^2)$. Smooth and heavy-tailed, with a parameter controlling the width. Constructor: `winFun_cauchy(alpha)`.

"kaiser" Kaiser window based on the modified Bessel function of the first kind. "beta" controls the trade-off between main lobe width and sidelobe level (0 = rectangular, ~5 ~= Hamming, ~8.6 ~= Blackman). Constructor: `winFun_kaiser(beta)`.

Examples

```
# "wn" as a character string
wns = c('rectangle', 'hann', 'hamming', 'blackman', 'flattop', 'bartlett',
'welch', 'parzen', 'bspline', 'bohman', 'blackmanharris', 'nuttall',
'barthannwin', 'sine', 'lanczos', 'gaussian', 'tukey', 'poisson',
'cauchy', 'kaiser')
op = par(c('mfrow', 'mar')); par(mfrow = c(5, 4), mar = c(0, 0, 3, 0))
for (w in wns)
```

```
  plot(winFun(256, w), xlab='', ylab='', bty='n', xaxt='n', yaxt='n', main=w)
par(op)

# Passing window-specific parameters via ...
plot(winFun(256, 'gaussian', sigma = 0.2), main = 'Gaussian, sigma = 0.2')
plot(winFun(256, 'gaussian', sigma = 0.5), main = 'Gaussian, sigma = 0.5')
plot(winFun(256, 'tukey', fraction = 0.8), main = 'Tukey, fraction = 0.8')
plot(winFun(256, 'kaiser', beta = 8.6), main = 'Kaiser, beta = 8.6')

# Equivalent: passing a constructor (useful when forwarding through
# higher-level functions like meanSpectrum(), spectrogram(), etc.)
plot(winFun(256, winFun_gauss(0.2)), main = 'Gaussian, sigma = 0.2')
plot(winFun(256, winFun_tukey(0.8)), main = 'Tukey, fraction = 0.8')
plot(winFun(256, winFun_kaiser(8.6)), main = 'Kaiser, beta = 8.6')

# use "wn" in other soundgen functions:
s = cos(2 * pi * 440 * (1:2000) / 2000) + 0.3 +
    cos(2 * pi * 880 * (1:2000) / 2000)
meanSpectrum(s, 2000, wn = 'blackman', yScale = 'dB')
meanSpectrum(s, 2000, wn = winFun_cauchy(1.2), yScale = 'dB')
meanSpectrum(s, 2000, wn = winFun_kaiser(8), yScale = 'dB')

# "wn" as a user-supplied function
plot(signal::kaiser(100, 4))
meanSpectrum(s, 2000, wn = function(n) signal::kaiser(n, 4), yScale = 'dB')
meanSpectrum(s, 2000, wn = function(n) signal::kaiser(n, 16), yScale = 'dB')

plot(winFun(256, signal::chebwin(256, 80)),
     main = 'Dolph-Chebyshev, -80 dB sidelobes')
meanSpectrum(s, 2000, wn = function(n) signal::chebwin(n, 80), yScale = 'dB')

halfsine = function(n) sin(pi * (0:(n-1)) / (n-1))
plot(winFun(25, halfsine), main = 'Half-sine')
meanSpectrum(s, 2000, wn = halfsine, yScale = 'dB')

# "wn" as a user-supplied numeric vector
custom = sin(pi * (0:255) / 255)   # half sine again
plot(winFun(256, custom), main = 'Half-sine')
```

Index

∗ **datasets**
detectNLP_training_nonv, 40
hillenbrand, 84
IR, 92
presets, 130
speechEx, 178

acf, *3*
acf_fft, 3
addAM, 4
addFormants, 7, *110*, *112*, *190*
addVectors *(*mixAtSNR*)*, 95
analyze, 11, *37, 38*, *57, 58*, *67*, *70*, *100*, *102*, *114*, *121*, *124*, *127*, *132*, *146*, *148*, *171*
annotation_app, 20, *55*, *122*
approx, *88*
audspec, *65*
audSpectrogram, 22, *32*, *65*, *77*, *94*, *97, 98*, *102*, *180*

bandpass, 26, *126*
beat, 29, *163*
butter, *23*

cheby1, *23*
compareFolder *(*compareSounds*)*, 31
compareSounds, 31, *94*
compressor *(*flatEnv*)*, 51
contour, *100*, *130*
corrDim, *119*
crossFade, 35, *45*

density, *119*
detectNLP, 37, *40*
detectNLP_training_nonv, 40
detectNLP_training_synth *(*detectNLP_training_nonv*)*, 40
dtw, *32*

estimateEmbeddingDim, *119*
estimateVTL, 41, *141*, *143*

fade, *36*, 43, *189*
filterMS, *45*
filterMS *(*filterSoundByMS*)*, 45
filterSoundByMS, 45, *91*
findformants, *14*
findPeaks, 50
flatEnv, 51, *69*, *114*, *187, 188*
formant_app, *20, 21*, 54, *122*

gaussianSmooth2D, 56, *171*
getDuration, 57
getEntropy, 58
getEnv, *32*, 60, *74*, *77*, *94*, *98*, *180*
getFormantFilter, *7–9*, 61, *160*, *163*, *189, 190*
getLoudness, *11*, *14*, *18, 19*, *58*, 64, *70*, *79*, *96*, *114*
getRMS, *19*, *32*, *57, 58*, *67*, 68, *77*, *94*, *114*, *180*
getRolloff, 71, *71*, *159*
getSpecEnv, 73, *152, 153*, *155*, *189, 190*
getSurprisal, 75

hilbert_approx *(*hilbert_exact*)*, 83
hilbert_exact, 83
hillenbrand, 84
HzToOther, 84

interp1, *88*
interpol_approxLowPass *(*interpolate*)*, 86
interpol_cardinal *(*interpolate*)*, 86
interpol_loess *(*interpolate*)*, 86
interpol_sgolay *(*interpolate*)*, 86
interpolate, *9*, 86, *136, 137*, *152*, *155*, *162*, *186*
interpolateNA, *88*, 89, *137*
invertSpectrogram, *45*, *49*, 90, *170*
IR, 92
istft_simple, *170*, *172*

istft_simple (stft_simple), 182
istft_timevar (stft_simple), 182

lines, *181*
loess, *88*

matchPars, 93
maxLyapunov, *119*
meanSpectrum, *28*, *32*, *94*, *99*, *167*, *175*
meanSpectrum (spectrum), 175
melfcc, *32*, *77*, *94*, *180*
mixAtSNR, 95
modulationSpectrum, *11*, *14*, *18*, *32*, *45*, *56*, *94*, 97, *129*, *166*, *172*, *181*
morph, 105
msToSpec (specToMS), 165

naiveBayes, *37*, *38*, 107, *107*
naiveBayes_train, *38*, *40*, *108*
naiveBayes_train (naiveBayes), 107
noiseRemoval, 110
normalizeFolder, 113

osc, 115, *172*
otherToHz (HzToOther), 84

palette, *24*, *78*, *119*, *171*
phasegram, *37*, *38*, 117
pitch_app, *11*, *14*, *19*, *38*, *54*, *55*, 121, *124*, *132*
pitchDescriptives, 124
pitchSmoothPraat, *26*, *124*, 126
play, *30*, *162*
playme, *5*, *9*, *111*, 127, *132*, *139*, *152*, *155*, *186*
plotMS, *100*, *102*, 128, *166*
png, *48*, *100*
poincareMap, *120*
powspec, *32*, *77*, *94*, *180*
presets, 130
prosody, 131

reportCI, 133
reportTime, *5*, *9*, *15*, *24*, *28*, *33*, *39*, *44*, *47*, *53*, *58*, *66*, *69*, *78*, *100*, *111*, *114*, *116*, *119*, *132*, 134, *136*, *140*, *147*, *152*, *155*, *170*, *177*, *180*, *186*
resample, *87*, *88*, 135, *135*
reverb, 138, *146*

schwa, *41*, *42*, *84*, 141
segment, *19*, 144, *150*, *181*
segment_ann, *148*, 150
sgolayfilt, *88*
shiftFormants, *73*, *74*, 151, *153*, *154*, *156*, *190*
shiftPitch, *131*, *132*, 153, *153*, *184*, *186*, *187*, *190*
soundgen, *7*, *9*, *29*, *30*, *41*, *71*, *72*, *93*, *105*, *106*, *130*, *152*, *154*, 157, *165*, *171*, *186*
soundgen_app, 165
specToMS, 165, *166*
specToMS_1D, 166
spectrogram, *16*, *32*, *39*, *56*, *66*, *73*, *77*, *91*, *93*, *98*, *100*, *102*, *112*, *130*, *162*, *166*, 168, *180*, *181*, *189*
spectrum, *32*, *94*, *99*, *167*, 175
speechEx, 178
spline, *60*, *88*
splinefun, *88*
ssm, *11*, *14*, *18*, *32*, *94*, *148*, *172*, 178
stft_simple, *32*, *73*, *76*, *93*, *166*, *172*, *180*, 182
surrogateTest, *119*

timeLag, *118*
timeStretch, *153*, 185
transplantEnv, *53*, *155*, 187, *190*
transplantFormants, *9*, *153*, *156*, 189

winFun, *9*, *13*, *152*, *155*, *162*, *169*, *177*, *183*, *190*, 191
winFun_cauchy (winFun), 191
winFun_gauss (winFun), 191
winFun_kaiser (winFun), 191
winFun_poisson (winFun), 191
winFun_tukey (winFun), 191