

Package: pladdrr (via r-universe)

September 14, 2026

Type Package

Title Direct Access to the Core Algorithms of Praat

Version 5.0.5

Description Provides direct access to Praat C implemented functionality from R. This package allows R users to leverage Praat's powerful phonetic analysis capabilities directly from R. The package exposes 39 Praat modules with 500+ methods for sound manipulation, robust formant tracking (FormantPath, FormantModeler), speech synthesis (KlattGrid), pitch analysis, intensity measurements, spectral analysis (ComplexSpectrogram), cepstral coefficients (MFCC, LFCC), statistical analysis (PCA, Discriminant), auditory modeling (Cochleagram, Excitation), TextGrid annotation, voice quality assessment, geometric operations (Polygon), and a persistent Praat script interpreter with bidirectional R-Praat object transfer.

License GPL-3 + file LICENSE

URL <https://github.com/humlab-speech/pladdrr>,
<https://humlab-speech.github.io/pladdrr/>

BugReports <https://github.com/humlab-speech/pladdrr/issues>

Encoding UTF-8

Depends R (>= 4.0.0)

Imports Rcpp (>= 1.0.0), R6 (>= 2.5.0), ggplot2, rlang, data.table (>= 1.14.0), methods, tools, utils, parallel, grid

LinkingTo Rcpp, RcppXsimd

Suggests testthat (>= 3.0.0), roxygen2 (>= 7.0.0), covr, knitr, rmarkdown, gridExtra, patchwork, scales, RcppSimdJson, RcppXPtrUtils, av, bench, microbenchmark, speakr, dplyr, purrr, tidyr

SystemRequirements C++17 compiler (GCC >= 7, Clang >= 5, MSVC >= 2017), GNU Scientific Library (GSL >= 1.10)

VignetteBuilder knitr

Config/roxygen2/version 8.1.0
Config/testthat/edition 3
Config/pak/sysreqs libgsl0-dev
Repository <https://ropensci-staging.r-universe.dev>
Date/Publication 2026-09-14 08:38:59 UTC
RemoteUrl <https://github.com/humlab-speech/pladdr>
RemoteRef HEAD
RemoteSha b9bec2e1ac805394f1989c7f3378a56f50ce6019

Contents

<code>\$.sound_constructor</code>	8
<code>\$.textgrid_constructor</code>	9
<code>aggregate_measurements</code>	9
<code>amplitude_tier_create</code>	10
<code>amplitude_tier_from_point_process</code>	11
<code>AmplitudeTier</code>	11
<code>analyze_files_parallel</code>	13
<code>apply_transform_xptr</code>	14
<code>apply_window_xptr</code>	15
<code>as-data-frame-missing</code>	16
<code>as.data.frame.Formant</code>	18
<code>as.data.frame.Intensity</code>	19
<code>as.data.frame.Pitch</code>	19
<code>as.data.frame.praat_formant</code>	21
<code>as.data.frame.praat_intensity</code>	21
<code>as.data.frame.praat_sound</code>	22
<code>as.data.frame.Sound</code>	23
<code>autoplot-methods</code>	24
<code>BarkSpectrogram</code>	36
<code>batch_process</code>	36
<code>batch_processing</code>	38
<code>benchmark_parallel</code>	38
<code>build_multiband_harmonicity</code>	39
<code>calculate_cppps_fast</code>	40
<code>calculate_cppps_ultra</code>	43
<code>calculate_durations_simd_bridge</code>	45
<code>calculate_f0_stats_ultra</code>	46
<code>calculate_midpoints_simd_bridge</code>	47
<code>calculate_minimum_intensity_ultra</code>	48
<code>calculate_multiband_hnr_ultra</code>	49
<code>Cepstrum</code>	50
<code>cepstrum_plots</code>	51
<code>check_audio_quality</code>	51
<code>Cochleagram</code>	53

ComplexSpectrogram	54
create_cepstrum_report	55
create_file_list	56
create_sound	57
create_sound_from_values	57
create_window_xptr	58
Discriminant	59
discriminant_from_matrix	60
DTW	60
duration_statistics_simd_bridge	62
DurationTier	63
Electroglottogram	63
electroglottogram_create	64
Excitation	65
extract_formant_parallel	66
extract_formants	67
extract_intensity	68
extract_intensity_parallel	69
extract_measurements	70
extract_measurements_custom	71
extract_pitch	72
extract_pitch_parallel	73
extract_textgrid_intervals	74
extract_voiced_segments	76
extract_voiced_segments_ultra	77
filter_by_duration_simd_bridge	79
Formant	80
FormantGrid	82
FormantModeler	83
FormantPath	83
FormantTier	84
formanttier_from_formant	85
format_quality_report	86
generate_noise	87
generate_sine_wave	88
get_cppps_fast	89
get_duration	90
get_durations_batch	91
get_formant_at_time	92
get_formant_bandwidths_at_times	93
get_formant_value_direct	94
get_formants_at_times	95
get_formants_direct	96
get_intensity_at_time	96
get_intensity_at_times	97
get_intensity_value_direct	98
get_interval_predicate	99
get_jitter_shimmer_batch	100

get_max_intensity	101
get_max_pitch	102
get_mean_formant	102
get_mean_intensity	103
get_mean_pitch	104
get_min_intensity	105
get_min_pitch	105
get_n_channels	106
get_n_samples	107
get_pitch_at_time	107
get_pitch_at_times	108
get_pitch_mean_direct	109
get_pitch_quantile_direct	109
get_pitch_quantiles_batch	110
get_pitch_stats_direct	112
get_pitch_stdev_direct	113
get_pitch_strengths_at_times	113
get_pitch_value_direct	114
get_pointprocess_intervals	115
get_pointprocess_nearest_indices	116
get_pointprocess_times	116
get_sampling_rate	117
get_sd_intensity	118
get_sound_times_fast	118
get_sound_values_fast	119
get_spectral_moments_batch	120
get_textgrid_interval_stats	120
get_textgrid_labels_all	121
get_voice_quality_ultra	122
Harmonicity	124
Intensity	126
intensity_tier_to_amplitude_tier	127
IntensityTier	128
is_fast_access	128
is_fast_vector	129
is_praat_formant	129
is_praat_intensity	130
is_praat_pitch	131
is_praat_sound	131
KlattGrid	132
klattgrid_create_example	133
klattgrid_create_from_vowel	133
LFCC	135
LongSound	135
longsound_get_buffer_size_pref_seconds	137
longsound_open	138
longsound_set_buffer_size_pref_seconds	139
LPC	139

Ltas	141
ltas_average	144
Manipulation	144
Matrix	145
matrix_create	146
matrix_create_simple	147
matrix_read	147
MelSpectrogram	148
MFCC	149
mfccs_to_dtw	150
multiband_hnr_stats	151
pair_files	152
pair_sound_textgrid	153
PCA	154
pca_from_matrix	155
Pitch	155
pitches_to_dtw	157
PitchModule	158
PitchTier	159
pladdr_simd	161
pladdr_threads	162
plot.Formant	163
plot.Harmonicity	164
plot.Intensity	165
plot.Ltas	166
plot.Matrix	167
plot.Pitch	168
plot.PointProcess	169
plot.PowerCepstrum	170
plot.Sound	171
plot.Spectrogram	173
plot.Spectrum	174
plot.TextGrid	175
plot_cpp_timeseries	176
plot_pitch_intensity	177
plot_powercepstrogram	178
plot_powercepstrum	179
plot_sound_pitch	180
plot_spectrogram_formants	182
plot_spectrogram_pitch	183
plot_textgrid_pitch	184
plot_textgrid_sound	185
plotting-combined	186
plotting-methods	187
PointProcess	187
Polygon	189
PowerCepstrogram	190
PowerCepstrum	191

pp_get_mean_period_direct	193
pp_get_stdev_period_direct	194
praat_direct	195
praat_eval_matrix	195
praat_eval_numeric	196
praat_eval_string	196
praat_eval_string_array	197
praat_eval_vector	197
praat_init	198
praat_initialized	198
praat_list_objects	199
praat_object_count	199
praat_run_script	200
praat_version	200
PraatInterpreter	201
print.fast_vector	205
print.praat_formant	205
print.praat_intensity	206
print.praat_pitch	207
print.praat_sound	207
print.zerocopy_vector	208
process_sounds_parallel	209
read_sound	210
set_textgrid_simd_enabled_bridge	210
simd_info	211
Sound	212
sound_as_matrix_fast	215
sound_auto_correlate	216
sound_concatenate_all	216
sound_create_pure_tone	217
sound_create_tone	218
sound_create_tone_complex	219
sound_deepen_band_modulation	220
sound_extract_and_formant	221
sound_extract_and_pitch	222
sound_extract_part	223
sound_extract_parts	224
sound_extract_parts_pooled	226
sound_filter_pass_hann_band	227
sound_filter_stop_hann_band	228
sound_from_values	228
sound_get_zcr	229
sound_lengthen	230
sound_load_window	231
sound_max	232
sound_mean	233
sound_min	233
sound_pool	234

sound_rms	235
sound_statistics	236
sound_times_fast	237
sound_to_formant_batch	237
sound_to_intensity_batch	238
sound_to_pitch_ac_batch	239
sound_to_pitch_batch	241
sound_to_pitch_cc_batch	242
sound_to_pitch_shs_batch	243
sound_to_pitch_spinet_batch	244
sound_to_textgrid_silences	245
sound_values_fast	247
sounds_append	248
sounds_convolve	249
sounds_cross_correlate	249
sounds_to_dtw	250
Spectrogram	251
spectrograms_to_dtw	252
Spectrum	253
spectrum_cepstral_smoothing	255
spectrum_pass_hann_band	255
spectrum_stop_hann_band	256
SpectrumTier	257
summary.praat_formant	258
summary.praat_intensity	259
summary.praat_pitch	259
summary.praat_sound	260
Table	261
table_create	261
TextGrid	262
textgrid_create	264
textgrid_extract_intervals_batch	265
textgrid_filter_xptr	266
textgrid_get_all_labels	268
textgrid_get_intervals_where	269
textgrid_interval_all_features_batch	270
textgrid_interval_formant_batch	271
textgrid_interval_intensity_batch	272
textgrid_interval_pitch_batch	273
textgrid_interval_statistics_batch	274
textgrid_merge	275
textgrid_simd_enabled	276
to_formant_direct	276
to_harmonicity_direct	278
to_intensity_direct	279
to_ltas_direct	279
to_pitch_ac_direct	280
to_pitch_cc_direct	281

to_pitch_direct 283

to_pitch_shs_direct 284

to_pitch_spinet_direct 285

to_point_process_direct 286

to_point_process_from_sound_and_pitch 287

to_powercepstrogram_fast 288

to_spectrogram_direct 289

to_spectrum_direct 290

two_pass_adaptive_pitch 290

vad 292

VocalTract 293

vocaltract_create_from_phone 294

with_pladdr_errors 295

Index 296

\$.sound_constructor	<i>\$ method for Sound constructor (enables Sound\$create_tone(), etc.)</i>
----------------------	---

Description

\$ method for Sound constructor (enables Sound\$create_tone(), etc.)

Usage

```
## S3 method for class 'sound_constructor'
x$name
```

Arguments

x	The Sound constructor function
name	Name of static method to access

Value

The requested static method function

Examples

```
tone_fn <- Sound$create_tone
sound <- tone_fn(frequency = 220, duration = 0.2, sampling_rate = 8000)
```

```
$.textgrid_constructor
```

```
$ method for TextGrid constructor (enables TextGrid$new(),
TextGrid$create())
```

Description

\$ method for TextGrid constructor (enables TextGrid\$new(), TextGrid\$create())

Usage

```
## S3 method for class 'textgrid_constructor'
x$name
```

Arguments

x	The TextGrid constructor function
name	Name of static method to access

Value

The requested static method function

Examples

```
create_fn <- TextGrid$create
tg <- create_fn(0, 1, "words")
```

```
aggregate_measurements
```

```
Aggregate Measurements by Label
```

Description

Aggregate extracted measurements by interval label (e.g., phoneme, word). Common workflow after extract_measurements().

Usage

```
aggregate_measurements(
  measurements,
  by = "label",
  stats = c("mean", "sd", "n")
)
```

Arguments

measurements	Data frame from extract_measurements().
by	Character. Column to group by (default: "label").
stats	Character vector. Statistics to compute: "mean", "sd", "median", "min", "max", "n".

Value

Data frame with aggregated statistics.

Examples

```
results <- data.frame(
  label = c("a", "a", "e", "e"),
  f0 = c(150, 155, 210, 205)
)
vowel_stats <- aggregate_measurements(
  measurements = results,
  by = "label",
  stats = c("mean", "sd", "n")
)
```

amplitude_tier_create *Create an empty AmplitudeTier*

Description

Creates a new AmplitudeTier object with no points.

Usage

```
amplitude_tier_create(tmin, tmax)
```

Arguments

tmin	Start time in seconds
tmax	End time in seconds

Value

An AmplitudeTier object

Examples

```
tier <- amplitude_tier_create(0, 1)
tier$add_point(0.5, 0.8)
```

amplitude_tier_from_point_process

Create AmplitudeTier from PointProcess and Sound

Description

Extracts amplitude values from a Sound at the times specified by a PointProcess.

Usage

```
amplitude_tier_from_point_process(point_process, sound)
```

Arguments

point_process	A PointProcess object
sound	A Sound object

Value

An AmplitudeTier object with amplitudes at each point time

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pp <- sound$to_point_process_periodic_cc(75, 600)
tier <- amplitude_tier_from_point_process(pp, sound)
tier$get_number_of_points()
```

AmplitudeTier

AmplitudeTier

Description

A Praat AmplitudeTier: sound pressure amplitude in Pascals as a function of time.

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ AmplitudeTier object; set internally when a method returns a new AmplitudeTier.
-------	--

Details

An AmplitudeTier stores amplitude as a sparse sequence of (time, value) points rather than a dense signal, with linear interpolation between points. It's the amplitude counterpart to an IntensityTier (which stores dB instead of Pa), and the two convert into each other. A common use is pulling amplitude at glottal pulse times (from a PointProcess) to compute shimmer.

Value

An AmplitudeTier object.

Usage

```
tier <- amplitude_tier_create(0, 1)
tier <- amplitude_tier_from_point_process(point_process, sound)
tier <- intensity_tier_to_amplitude_tier(intensity_tier)
```

Query methods

- `get_start_time()`, `get_end_time()` - time domain in seconds
- `get_number_of_points()` - number of (time, value) points
- `get_time_from_index(index)` - time at a 1-based point index
- `get_value_at_index(index)` - amplitude at a 1-based point index (Pa)
- `get_value_at_time(time)` - interpolated amplitude at a time (Pa)

Modification

- `add_point(time, value)` - add a (time, value) point
- `remove_point(index)` - remove the point at a 1-based index

Conversion and export

- `to_intensity_tier(threshold_db)` - convert amplitude to an IntensityTier
- `as_data_frame()` - points as a data frame with time and amplitude_pa columns
- `save(path)` - write to file

See Also

[`amplitude_tier_create`], [`amplitude_tier_from_point_process`], [`intensity_tier_to_amplitude_tier`]

Examples

```
tier <- amplitude_tier_create(0, 1)
tier$add_point(0.25, 0.5)
tier$add_point(0.75, 0.8)
tier$get_number_of_points()
tier$get_value_at_time(0.5)
```

analyze_files_parallel

Process Audio Files in Parallel

Description

Process multiple audio files in parallel across multiple CPU cores, instead of processing them sequentially.

Usage

```
analyze_files_parallel(
  files,
  analysis_func,
  n_cores = NULL,
  threads_per_worker = NULL,
  ...
)
```

Arguments

files	Character vector. Paths to audio files
analysis_func	Function. Analysis function to apply to each file. Should accept a Sound object and return results.
n_cores	Integer. Number of CPU cores to use (default: parallel::detectCores() - 1)
threads_per_worker	Integer or 'NULL'. C++ threads each worker may use for Praat kernels. 'NULL' (default) auto-divides cores among workers so total concurrency stays near the machine's core count, preventing oversubscription. Set '1' to force strictly single-threaded workers.
...	Additional arguments passed to analysis_func

Details

This function distributes batch analysis across worker processes. Each file is: 1. Loaded as a Sound object 2. Processed by analysis_func 3. Results collected and returned

Value

List of results from analysis_func, one per file

Examples

```
audio_dir <- tempfile("audio-")
dir.create(audio_dir)
tone <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)
```

```

tone$save(file.path(audio_dir, "tone1.wav"))
files <- list.files(audio_dir, pattern = "\\\\.wav$", full.names = TRUE)

analyze_pitch <- function(sound) {
  pitch <- sound$to_pitch()
  list(
    mean_f0 = pitch$get_mean(0, 0, "hertz"),
    sd_f0 = pitch$get_standard_deviation(0, 0, "hertz")
  )
}

# n_cores = 1 keeps this a single-process example (CRAN-safe)
results <- analyze_files_parallel(files, analyze_pitch, n_cores = 1)

```

apply_transform_xptr *Apply Compiled Transform Function (Advanced Performance API)*

Description

Apply a user-defined C++ transform function to sample values, compiled via RcppXPtrUtils, instead of calling an R callback per sample. Requires the RcppXPtrUtils package.

Usage

```
apply_transform_xptr(sound, transform_func)
```

Arguments

sound Sound object or external pointer
transform_func External pointer from RcppXPtrUtils::cppXPtr()

Details

****ADVANCED API**** - Requires RcppXPtrUtils package.

The transform function receives sample amplitude and returns transformed amplitude. Common transforms:

- Clipping: 'x > threshold ? threshold : (x < -threshold ? -threshold : x)'
- Soft clipping: 'tanh(x * gain)'
- Rectification: 'fabs(x)'
- Squaring: 'x * x'

Value

Sound object with transform function applied

Examples

```
# Compiling a C++ transform function takes a few seconds
if (requireNamespace("RcppXPtrUtils", quietly = TRUE)) {
  soft_clip <- RcppXPtrUtils::cppXPtr(
    "double softclip(double x) { return tanh(x * 2.0); }",
    depends = character()
  )

  sound <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
    16000)
  clipped <- apply_transform_xptr(sound, soft_clip)
}
```

apply_window_xptr	<i>Apply Compiled Window Function (Advanced Performance API)</i>
-------------------	--

Description

Apply a user-defined C++ window function to a Sound object, compiled via RcppXPtrUtils, instead of calling an R callback per sample. Requires the RcppXPtrUtils package.

Usage

```
apply_window_xptr(sound, window_func)
```

Arguments

sound	Sound object or external pointer
window_func	External pointer from RcppXPtrUtils::cppXPtr()

Details

****ADVANCED API**** - Requires RcppXPtrUtils package.

The window function receives normalized time (0 to 1) and returns a multiplier. Common window functions:

- Hamming: $0.54 - 0.46 * \cos(2 * M_PI * t)$
- Hanning: $0.5 * (1 - \cos(2 * M_PI * t))$
- Gaussian: $\exp(-18 * (t - 0.5)^2)$
- Triangular: $1 - \text{fabs}(2*t - 1)$

Value

Sound object with window function applied

Examples

```
# Compiling a C++ window function takes a few seconds
if (requireNamespace("RcppXPtrUtils", quietly = TRUE)) {
  gauss_window <- RcppXPtrUtils::cppXPtr(
    "double gauss(double t) {
      double x = t - 0.5;
      return exp(-18.0 * x * x);
    }",
    depends = character()
  )

  # Verify signature (optional but recommended)
  RcppXPtrUtils::checkXPtr(gauss_window, "double", "double")

  sound <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
    16000)
  windowed <- apply_window_xptr(sound, gauss_window)
}
```

as-data-frame-missing *as.data.frame Methods for pladdrr Objects*

Description

‘as.data.frame()’ methods for Praat object classes that expose their data as flat data frames (long format for 2-D matrix-like objects, one row per sample/point/frame for 1-D tiers and tracks).

Usage

```
## S3 method for class 'Discriminant'
as.data.frame(x, ...)

## S3 method for class 'Electroglottogram'
as.data.frame(x, ...)

## S3 method for class 'FormantModeler'
as.data.frame(x, ...)

## S3 method for class 'PCA'
as.data.frame(x, ...)

## S3 method for class 'PowerCepstrogram'
as.data.frame(x, ...)

## S3 method for class 'BarkSpectrogram'
as.data.frame(x, ...)
```



```

## S3 method for class 'Cepstrum'
as.data.frame(x, row.names = NULL, optional = FALSE, power = FALSE, ...)

## S3 method for class 'Cochleagram'
as.data.frame(x, ...)

## S3 method for class 'DTW'
as.data.frame(x, ...)

## S3 method for class 'KlattGrid'
as.data.frame(x, ...)

## S3 method for class 'LPC'
as.data.frame(x, ...)

## S3 method for class 'LongSound'
as.data.frame(x, ...)

## S3 method for class 'Matrix'
as.data.frame(x, ...)

## S3 method for class 'MelSpectrogram'
as.data.frame(x, ...)

## S3 method for class 'VocalTract'
as.data.frame(x, ...)

```

Arguments

<code>x</code>	A pladdrr R6 object (Discriminant, Electroglottogram, FormantModeler, PCA, PowerCepstrogram, BarkSpectrogram, Cepstrum, Cochleagram, DTW, KlattGrid, LPC, LongSound, Matrix, MelSpectrogram, or VocalTract).
<code>...</code>	Passed to methods; currently unused except ‘power’ on ‘Cepstrum’.
<code>row.names</code>	Unused (required by the ‘as.data.frame()’ generic).
<code>optional</code>	Unused (required by the ‘as.data.frame()’ generic).
<code>power</code>	If TRUE, convert to PowerCepstrum and return its nonnegative linear power values (column ‘power’) instead of Praat’s default raw signed cepstrum view. Default FALSE matches ‘Cepstrum_drawLinear’, Praat’s default "Draw..." command. Note: this returns linear power, not dB — ‘autoplot()’/‘autolayer()’ convert to dB (‘10 * log10(power)’) for display only; that conversion is not applied here.

Value

A data.frame. Column names vary by class; see each class’s `$as_data_frame()` method or the corresponding ‘autoplot.*’ method for the exact columns used.

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.2, sampling_rate =
  16000)
cepstrum <- sound$to_cepstrum()
df <- as.data.frame(cepstrum)
head(df)

```

as.data.frame.Formant *Convert R6 Formant to data frame*

Description

S3 method for converting R6 Formant objects to data frames. Delegates to the R6 ‘\$as_data_frame()’ method.

Usage

```

## S3 method for class 'Formant'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

```

Arguments

x	A Formant R6 object
row.names	Ignored
optional	Ignored
...	Additional arguments passed to ‘\$as_data_frame()’ (e.g. ‘max_formants’)

Value

A data.table (inherits from data.frame) in long format, one row per (frame, formant number): columns ‘time’, ‘formant’ (1-based formant number), ‘frequency’ (Hz), ‘bandwidth’ (Hz). Matches ‘as.data.frame.FormantPath()’.

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
formant <- sound$to_formant_burg()
df <- as.data.frame(formant)
head(df)

```

`as.data.frame.Intensity`*Convert R6 Intensity to data frame*

Description

S3 method for converting R6 Intensity objects to data frames. Delegates to the R6 ‘\$as_data_frame()’ method.

Usage

```
## S3 method for class 'Intensity'  
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

<code>x</code>	An Intensity R6 object
<code>row.names</code>	Ignored
<code>optional</code>	Ignored
<code>...</code>	Additional arguments passed to the underlying function or ignored.

Value

A data.table (inherits from data.frame) with time and intensity columns

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =  
  16000)  
intensity <- sound$to_intensity()  
df <- as.data.frame(intensity)  
head(df)
```

`as.data.frame.Pitch`*Convert R6 Pitch to data frame*

Description

S3 method for converting R6 Pitch objects to data frames. Delegates to the R6 ‘\$as_data_frame()’ method.

Usage

```
## S3 method for class 'Pitch'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'PointProcess'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'TextGrid'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'MFCC'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'LFCC'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

<code>x</code>	An LFCC R6 object
<code>row.names</code>	Ignored
<code>optional</code>	Ignored
<code>...</code>	Additional arguments passed to the underlying function or ignored.

Value

A `data.table` (inherits from `data.frame`) with pitch measurements

Functions

- `as.data.frame(PointProcess)`: Convert `PointProcess` to `data.frame`
- `as.data.frame(TextGrid)`: Convert `TextGrid` to `data.frame`
- `as.data.frame(MFCC)`: Convert `MFCC` to `data.frame`
- `as.data.frame(LFCC)`: Convert `LFCC` to `data.frame`

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)
pitch <- sound$to_pitch()
df <- as.data.frame(pitch)
head(df)
```

```
as.data.frame.praat_formant
```

Convert praat_formant to data.frame

Description

Convert praat_formant to data.frame

Usage

```
## S3 method for class 'praat_formant'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A praat_formant object
row.names	Not used
optional	Not used
...	Additional arguments (unused)

Value

The values data.table (inherits from data.frame) from the formant object

Examples

```
x <- list(values = data.frame(
  time = c(0.1, 0.2), formant_number = c(1, 1),
  frequency = c(500, 520), bandwidth = c(80, 82)
))
class(x) <- "praat_formant"
as.data.frame(x)
```

```
as.data.frame.praat_intensity
```

Convert praat_intensity to data.frame

Description

Convert praat_intensity to data.frame

Usage

```
## S3 method for class 'praat_intensity'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

<code>x</code>	A <code>praat_intensity</code> object
<code>row.names</code>	Not used
<code>optional</code>	Not used
<code>...</code>	Additional arguments (unused)

Value

The values `data.table` (inherits from `data.frame`) from the intensity object

Examples

```
x <- list(values = data.frame(time = c(0.1, 0.2), intensity_db = c(65.2,
66.1)))
class(x) <- "praat_intensity"
as.data.frame(x)
```

```
as.data.frame.praat_sound
```

Convert praat_sound to data frame

Description

Converts a `praat_sound` object to a data frame with time and amplitude columns. This is useful for plotting and further analysis in R.

Usage

```
## S3 method for class 'praat_sound'
as.data.frame(x, ...)
```

Arguments

<code>x</code>	A <code>praat_sound</code> object
<code>...</code>	Additional arguments passed to the underlying function or ignored.

Value

A `data.table` (inherits from `data.frame`) with two columns:

time Time in seconds

amplitude Amplitude values

Examples

```
values <- sin(2 * pi * 220 * seq(0, 0.1, length.out = 1000))
snd <- create_sound_from_values(values, sampling_rate = 10000)
df <- as.data.frame(snd)
head(df)
```

as.data.frame.Sound	<i>Convert R6 Sound to data frame</i>
---------------------	---------------------------------------

Description

S3 method for converting R6 Sound objects to data frames. Delegates to the R6 `$as_data_frame()` method.

Usage

```
## S3 method for class 'Sound'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A Sound R6 object
row.names	Ignored
optional	Ignored
...	Additional arguments passed to the underlying function or ignored.

Value

A data.table (inherits from data.frame) with time, channel, and value columns

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.1, sampling_rate =
  8000)
df <- as.data.frame(sound)
head(df)
```

Description

ggplot2-style autoplot() and autolayer() methods for Praat objects. These provide a tidyverse-idiomatic interface for plotting, allowing flexible composition of multi-object visualizations.

Usage

```
## S3 method for class 'Sound'
autoplot(object, from_time = NULL, to_time = NULL, color = "steelblue", ...)

## S3 method for class 'Sound'
autolayer(object, from_time = NULL, to_time = NULL, color = "steelblue", ...)

## S3 method for class 'Pitch'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  color = "darkgreen",
  show_voicing = FALSE,
  ...
)

## S3 method for class 'Pitch'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  color = "darkgreen",
  geom = c("line", "point"),
  ...
)

## S3 method for class 'Formant'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3,
  colors = NULL,
  ...
)

## S3 method for class 'Formant'
```



```
autolayer(  
  object,  
  from_time = NULL,  
  to_time = NULL,  
  max_formant = 3,  
  colors = NULL,  
  ...  
)  
  
## S3 method for class 'Intensity'  
autoplot(object, from_time = NULL, to_time = NULL, color = "darkorange", ...)  
  
## S3 method for class 'Intensity'  
autolayer(object, from_time = NULL, to_time = NULL, color = "darkorange", ...)  
  
## S3 method for class 'Spectrogram'  
autoplot(  
  object,  
  from_time = NULL,  
  to_time = NULL,  
  from_freq = NULL,  
  to_freq = NULL,  
  dynamic_range = 70,  
  ...  
)  
  
## S3 method for class 'Spectrogram'  
autolayer(  
  object,  
  from_time = NULL,  
  to_time = NULL,  
  from_freq = NULL,  
  to_freq = NULL,  
  dynamic_range = 70,  
  ...  
)  
  
## S3 method for class 'Spectrum'  
autoplot(  
  object,  
  from_freq = NULL,  
  to_freq = NULL,  
  log_freq = FALSE,  
  color = "navy",  
  ...  
)  
  
## S3 method for class 'Spectrum'
```

```
autolayer(object, from_freq = NULL, to_freq = NULL, color = "navy", ...)  
  
## S3 method for class 'Ltas'  
autoplot(  
  object,  
  from_freq = NULL,  
  to_freq = NULL,  
  log_freq = FALSE,  
  color = "darkred",  
  ...  
)  
  
## S3 method for class 'Ltas'  
autolayer(object, from_freq = NULL, to_freq = NULL, color = "darkred", ...)  
  
## S3 method for class 'Harmonicity'  
autoplot(object, from_time = NULL, to_time = NULL, color = "darkviolet", ...)  
  
## S3 method for class 'Harmonicity'  
autolayer(object, from_time = NULL, to_time = NULL, color = "darkviolet", ...)  
  
## S3 method for class 'PointProcess'  
autoplot(object, from_time = NULL, to_time = NULL, color = "black", ...)  
  
## S3 method for class 'PointProcess'  
autolayer(  
  object,  
  from_time = NULL,  
  to_time = NULL,  
  color = "black",  
  ymin = 0,  
  ymax = 1,  
  ...  
)  
  
## S3 method for class 'PowerCepstrum'  
autoplot(  
  object,  
  from_quefrency = NULL,  
  to_quefrency = NULL,  
  color = "darkblue",  
  mark_peak = TRUE,  
  ...  
)  
  
## S3 method for class 'PowerCepstrum'  
autolayer(  
  object,
```

```
    from_quefreny = NULL,
    to_quefreny = NULL,
    color = "darkblue",
    ...
)

## S3 method for class 'TextGrid'
autoplot(object, ...)

## S3 method for class 'TextGrid'
autolayer(
  object,
  tier = 1,
  from_time = NULL,
  to_time = NULL,
  color = "steelblue",
  alpha = 0.3,
  ...
)

## S3 method for class 'AmplitudeTier'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  color = "darkred",
  garnish = TRUE,
  ...
)

## S3 method for class 'AmplitudeTier'
autolayer(object, from_time = NULL, to_time = NULL, color = "darkred", ...)

## S3 method for class 'DurationTier'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  color = "steelblue",
  garnish = TRUE,
  ...
)

## S3 method for class 'DurationTier'
autolayer(object, from_time = NULL, to_time = NULL, color = "steelblue", ...)

## S3 method for class 'IntensityTier'
autoplot(
```

```

    object,
    from_time = NULL,
    to_time = NULL,
    color = "darkgreen",
    garnish = TRUE,
    ...
)

## S3 method for class 'IntensityTier'
autolayer(object, from_time = NULL, to_time = NULL, color = "darkgreen", ...)

## S3 method for class 'PitchTier'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  color = "blue",
  garnish = TRUE,
  ...
)

## S3 method for class 'PitchTier'
autolayer(object, from_time = NULL, to_time = NULL, color = "blue", ...)

## S3 method for class 'FormantTier'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3L,
  colors = NULL,
  time_step = 0.005,
  style = c("speckle", "line"),
  garnish = TRUE,
  ...
)

## S3 method for class 'FormantTier'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3L,
  colors = NULL,
  time_step = 0.005,
  style = c("speckle", "line"),
  ...
)

```

```
## S3 method for class 'FormantGrid'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3L,
  colors = NULL,
  garnish = TRUE,
  ...
)

## S3 method for class 'FormantGrid'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3L,
  colors = NULL,
  ...
)

## S3 method for class 'FormantPath'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3L,
  colors = NULL,
  show_candidates = TRUE,
  garnish = TRUE,
  ...
)

## S3 method for class 'FormantPath'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3L,
  colors = NULL,
  show_candidates = FALSE,
  ...
)

## S3 method for class 'Excitation'
autoplot(object, garnish = TRUE, color = "darkred", ...)
```

```
## S3 method for class 'Excitation'
autolayer(object, color = "darkred", ...)

## S3 method for class 'ComplexSpectrogram'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  from_freq = NULL,
  to_freq = NULL,
  dynamic_range = 70,
  garnish = TRUE,
  show_phase = FALSE,
  ...
)

## S3 method for class 'ComplexSpectrogram'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  from_freq = NULL,
  to_freq = NULL,
  dynamic_range = 70,
  ...
)

## S3 method for class 'Cepstrum'
autoplot(object, power = FALSE, garnish = TRUE, color = "darkblue", ...)

## S3 method for class 'Cepstrum'
autolayer(object, power = FALSE, color = "darkblue", ...)

## S3 method for class 'Cochleagram'
autoplot(object, from_time = NULL, to_time = NULL, garnish = TRUE, ...)

## S3 method for class 'Cochleagram'
autolayer(object, from_time = NULL, to_time = NULL, ...)

## S3 method for class 'PowerCepstrogram'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  quefreny_range = NULL,
  garnish = TRUE,
  ...
)
```

```
## S3 method for class 'PowerCepstrogram'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  quefrency_range = NULL,
  ...
)

## S3 method for class 'MFCC'
autoplot(object, coefficient_range = NULL, garnish = TRUE, ...)

## S3 method for class 'MFCC'
autolayer(object, coefficient_range = NULL, ...)

## S3 method for class 'LFCC'
autoplot(object, coefficient_range = NULL, garnish = TRUE, ...)

## S3 method for class 'LFCC'
autolayer(object, coefficient_range = NULL, ...)

## S3 method for class 'BarkSpectrogram'
autoplot(object, garnish = TRUE, ...)

## S3 method for class 'BarkSpectrogram'
autolayer(object, ...)

## S3 method for class 'MelSpectrogram'
autoplot(object, garnish = TRUE, ...)

## S3 method for class 'MelSpectrogram'
autolayer(object, ...)

## S3 method for class 'Matrix'
autoplot(
  object,
  x_col = NULL,
  y_col = NULL,
  fill_col = NULL,
  garnish = TRUE,
  ...
)

## S3 method for class 'Matrix'
autolayer(object, x_col = NULL, y_col = NULL, fill_col = NULL, ...)

## S3 method for class 'PCA'
```

```
autoplot(object, type = c("scree", "scores", "both"), garnish = TRUE, ...)
```

```
## S3 method for class 'PCA'
autolayer(object, ...)
```

```
## S3 method for class 'Discriminant'
autoplot(object, type = c("scree", "scores", "both"), garnish = TRUE, ...)
```

```
## S3 method for class 'Discriminant'
autolayer(object, ...)
```

```
## S3 method for class 'FormantModeler'
autoplot(object, from_track = 1L, to_track = 0L, garnish = TRUE, ...)
```

```
## S3 method for class 'FormantModeler'
autolayer(object, from_track = 1L, to_track = 0L, ...)
```

```
## S3 method for class 'Electroglottogram'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  color = "black",
  garnish = TRUE,
  ...
)
```

```
## S3 method for class 'Electroglottogram'
autolayer(object, from_time = NULL, to_time = NULL, color = "black", ...)
```

```
## S3 method for class 'LongSound'
autoplot(
  object,
  from_time = 0,
  to_time = 2,
  color = "black",
  garnish = TRUE,
  ...
)
```

```
## S3 method for class 'LongSound'
autolayer(object, from_time = 0, to_time = 2, color = "black", ...)
```

```
## S3 method for class 'DTW'
autoplot(object, garnish = TRUE, alpha_path = 0.8, ...)
```

```
## S3 method for class 'DTW'
autolayer(object, alpha_path = 0.8, ...)
```



```
## S3 method for class 'Polygon'
autoplot(
  object,
  garnish = TRUE,
  fill_polygon = FALSE,
  color = "black",
  fill_color = "grey80",
  ...
)

## S3 method for class 'Polygon'
autolayer(object, color = "black", ...)

## S3 method for class 'VocalTract'
autoplot(object, garnish = TRUE, plot_type = c("area", "line"), ...)

## S3 method for class 'VocalTract'
autolayer(object, ...)

## S3 method for class 'LPC'
autoplot(object, frame = 1L, garnish = TRUE, color = "darkred", ...)

## S3 method for class 'LPC'
autolayer(object, frame = 1L, color = "darkred", ...)

## S3 method for class 'KlattGrid'
autoplot(
  object,
  from_time = NULL,
  to_time = NULL,
  formant_type = c("all", "oral", "nasal", "frication", "tracheal", "delta"),
  max_formant = 6L,
  garnish = TRUE,
  ...
)

## S3 method for class 'KlattGrid'
autolayer(
  object,
  from_time = NULL,
  to_time = NULL,
  formant_type = c("oral", "nasal", "frication", "tracheal", "delta"),
  max_formant = 6L,
  ...
)
```

Arguments

<code>object</code>	A pladdrr S3 analysis object (Sound, Pitch, Formant, Intensity, Spectrogram, Spectrum, Ltas, Harmonicity, PointProcess, PowerCepstrum, or TextGrid)
<code>from_time</code>	Start time in seconds (NULL = from beginning)
<code>to_time</code>	End time in seconds (NULL = to end)
<code>color</code>	Line color (default: "steelblue")
<code>...</code>	Additional arguments passed to <code>geom_line</code>
<code>show_voicing</code>	Color by voicing strength (default: FALSE)
<code>geom</code>	For Pitch: type of geometry ("line" or "point")
<code>max_formant</code>	Maximum formant number to display (default: 3).
<code>colors</code>	Colors for each formant track (default: auto).
<code>from_freq</code>	Start frequency in Hz
<code>to_freq</code>	End frequency in Hz
<code>dynamic_range</code>	Dynamic range in dB (default: 70)
<code>log_freq</code>	Use logarithmic frequency scale (default: FALSE)
<code>ymin</code>	For PointProcess: minimum y value for vertical lines (default: 0)
<code>ymax</code>	For PointProcess: maximum y value for vertical lines (default: 1)
<code>from_quefrency</code>	Start quefrency in seconds
<code>to_quefrency</code>	End quefrency in seconds
<code>mark_peak</code>	Mark the cepstral peak (default: TRUE)
<code>tier</code>	Tier number or name to display (default: 1)
<code>alpha</code>	Fill transparency (default: 0.3)
<code>garnish</code>	Whether to add a title and axis labels (default TRUE).
<code>time_step</code>	Sampling interval in seconds, only used when <code>style="line"</code> (default: 0.005).
<code>style</code>	"speckle" (default, matches Praat's FormantTier_speckle: plots the tier's own stored points, unconnected) or "line" (interpolates on a <code>time_step</code> grid and connects with a line — not Praat's default view).
<code>show_candidates</code>	Logical. Overlay candidate paths (default: TRUE).
<code>show_phase</code>	Include phase as separate panel (default: FALSE).
<code>power</code>	If TRUE, plot as PowerCepstrum (dB). Default FALSE matches Praat's default 'Cepstrum_drawLinear' (raw signed cepstrum, linear scale).
<code>quefrency_range</code>	Optional 'c(min, max)' quefrency range to display (PowerCepstrogram); NULL shows the full range.
<code>coefficient_range</code>	Range of coefficients to display (e.g. 1:12).
<code>x_col</code>	Column name for x-axis (default: auto-detect).
<code>y_col</code>	Column name for y-axis (default: auto-detect).

<code>fill_col</code>	Column name for fill (default: auto-detect).
<code>type</code>	For PCA: "scree" (variance explained), "scores" (component scores), or "both" (combined via patchwork).
<code>from_track, to_track</code>	Track index range to display (FormantModeler).
<code>alpha_path</code>	Alpha for warping path line (default: 0.8).
<code>fill_polygon</code>	Fill the polygon interior (default: FALSE).
<code>fill_color</code>	Fill color for closed shapes (Polygon); distinct from 'fill_col', which selects a data column to map to a continuous fill scale.
<code>plot_type</code>	Plot style selector (VocalTract).
<code>frame</code>	Time or frame index to extract LPC spectrum at.
<code>formant_type</code>	Type of formant to plot: "oral", "nasal", "frication", "tracheal", "delta", or (autoplot.KlattGrid only) "all".

Details

The autoplot/autolayer pattern enables flexible plot composition:

- `autoplot(object)` - Creates a complete ggplot
- `autolayer(object)` - Returns a layer to add to existing plots

This allows combining multiple objects:

```
autoplot(spectrogram) +
  autolayer(formant, max_formant = 3) +
  autolayer(pitch, color = "blue")
```

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
p <- ggplot2::autoplot(sound)

pitch <- sound$to_pitch()
p2 <- ggplot2::autoplot(pitch)
```

BarkSpectrogram	<i>BarkSpectrogram</i>
-----------------	------------------------

Description

A Bark-scale spectrogram, used in psychoacoustic and perceptual analysis of speech.

Usage

```
BarkSpectrogram(.xptr = NULL)
```

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ BarkSpectrogram object; set internally by Sound\$to_bark_spectrogram().
-------	--

Value

A BarkSpectrogram object.

Examples

```
snd <- Sound$create_tone(frequency = 150, duration = 0.3)
bark <- snd$to_bark_spectrogram()

# Export as matrix
mat <- bark$as_matrix()
```

batch_process	<i>Batch Process Audio Files</i>
---------------	----------------------------------

Description

Process multiple audio files with a custom function. This replaces Praat's pattern of creating Strings objects and looping over files.

Usage

```
batch_process(
  directory,
  pattern = "\\*.wav$",
  func,
  recursive = FALSE,
  parallel = FALSE,
  ncores = NULL,
```

```

    progress = TRUE,
    ...
)

```

Arguments

directory	Character path to directory containing audio files
pattern	Regular expression pattern to match files (default: "\\wav\$")
func	Function to apply to each file. Should accept a Sound object as first argument and return a named list or data frame row
recursive	Logical, search directories recursively (default: FALSE)
parallel	Logical, use parallel processing (default: FALSE)
ncores	Integer, number of cores for parallel processing (default: NULL = all-1)
progress	Logical, show progress bar (default: TRUE)
...	Additional arguments passed to func

Value

Data frame with results from all files

Examples

```

audio_dir <- tempfile("audio_")
dir.create(audio_dir)
tone <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)
tone$save(file.path(audio_dir, "tone1.wav"))

results <- batch_process(
  directory = audio_dir,
  pattern = "\\wav$",
  progress = FALSE,
  func = function(sound) {
    pitch <- sound$to_pitch()
    list(
      mean_f0 = pitch$get_mean(from_time = 0, to_time = 0, unit = "hertz"),
      sd_f0 = pitch$get_standard_deviation(from_time = 0, to_time = 0, unit =
        "hertz")
    )
  }
)

```

batch_processing	<i>Batch Processing Utilities for Speaker Package</i>
------------------	---

Description

Functions for batch processing audio files, pairing Sound/TextGrid files, and extracting measurements across multiple files. These utilities replace the need for Praat's Strings and Table objects with R-idiomatic workflows.

Value

This is a documentation-only overview; see the individual functions ([batch_process](#), [create_file_list](#), [extract_measurements](#), [extract_measurements_custom](#), [aggregate_measurements](#)) for their return values.

Examples

```
# See individual functions, e.g. ?batch_process, ?create_file_list
```

benchmark_parallel	<i>Benchmark Parallel vs Sequential Processing</i>
--------------------	--

Description

Compare performance of parallel vs sequential processing. Useful for determining optimal core count for your workload.

Usage

```
benchmark_parallel(files, analysis_func, core_counts = c(1, 2, 4))
```

Arguments

files	Character vector of files (use subset for quick test)
analysis_func	Function to benchmark
core_counts	Integer vector. Core counts to test (default: c(1, 2, 4))

Value

Data frame with timing results

Examples

```

audio_dir <- tempfile("audio_")
dir.create(audio_dir)
tone <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)
tone$save(file.path(audio_dir, "tone1.wav"))
files <- list.files(audio_dir, pattern = "\\wav$", full.names = TRUE)

# CRAN examples should use at most 2 cores
results <- benchmark_parallel(
  files,
  function(s) s$to_pitch()$get_mean(0, 0, "hertz"),
  core_counts = c(1, 2)
)
print(results)

```

build_multiband_harmonicity

Build reusable multi-band Harmonicity objects

Description

Compute the 5 Harmonicity objects used by VQ's multiband HNR workflow once, then reuse them across repeated interval queries with [multiband_hnr_stats()].

Usage

```

build_multiband_harmonicity(
  sound,
  bands = c(0, 500, 1500, 2500, 3500),
  time_step = 0.005,
  min_pitch = 75
)

```

Arguments

sound	Sound object or external pointer
bands	Numeric vector of upper frequency limits in Hz (default 'c(0, 500, 1500, 2500, 3500)')
time_step	Time step for harmonicity in seconds (default '0.005')
min_pitch	Minimum pitch in Hz (default '75')

Details

Use this when the same 'Sound' is queried over many '[from_time, to_time]' windows, e.g. a TextGrid with multiple voiced intervals. The expensive band-pass filtering and Harmonicity computation are done once; only the summary stats are repeated.

Value

Named list of 5 ‘Harmonicity’ objects: ‘full’, ‘band500’, ‘band1500’, ‘band2500’, ‘band3500’ (or names derived from custom bands).

References

- VQ_measurements_V2.praat (Voice Quality measurements) - Maryn & Weenink (2015) - Multiband HNR for voice quality

See Also

[multiband_hnr_stats](#) and [calculate_multiband_hnr_ultra](#)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)

built <- build_multiband_harmonicity(sound)
hnr_full <- multiband_hnr_stats(built)
```

calculate_cppps_fast	<i>Smoothed Cepstral Peak Prominence (CPPS) in one call</i>
----------------------	---

Description

Computes the smoothed cepstral peak prominence (CPPS, in dB) of a ‘Sound’ — a widely used acoustic measure of voice quality/breathiness. It performs the whole PowerCepstrogram-then-CPPS pipeline in a single call and returns the same value as ‘sound\$to_powercepstrogram(...)\$get_cppps(...)’ with matching parameters. You supply the analysis parameters directly and are responsible for passing valid values (see the arguments below).

Usage

```
calculate_cppps_fast(
  sound,
  subtract_tilt = TRUE,
  time_averaging_window = 0.001,
  quefrency_averaging_window = 5e-04,
  pitch_floor = 60,
  pitch_ceiling = 333.3,
  delta_f0 = 0.05,
  interpolation = "parabolic",
  qstart_fit = 0.003,
  qend_fit = 0.04,
  trend_line_type = "straight",
```



```

    fit_method = "robust",
    cepstrogram_pitch_floor = 60,
    time_step = 0.002,
    max_frequency = 5000,
    pre_emphasis_from = 50
)

```

Arguments

sound	Sound object or external pointer
subtract_tilt	Logical, subtract tilt before calculating CPPS (default TRUE)
time_averaging_window	Numeric, time averaging window in seconds (default 0.001)
quefrequency_averaging_window	Numeric, quefrequency averaging window in seconds (default 0.0005)
pitch_floor	Numeric, minimum F0 in Hz (default 60)
pitch_ceiling	Numeric, maximum F0 in Hz (default 333.3)
delta_f0	Numeric, F0 fractional precision (default 0.05)
interpolation	Character, one of "parabolic", "none", "cubic", "sinc70", "sinc700" (default "parabolic")
qstart_fit	Numeric, quefrequency range start for fitting in seconds (default 0.003)
qend_fit	Numeric, quefrequency range end in seconds (default 0.04)
trend_line_type	Character, "straight" or "exponential" (default "straight")
fit_method	Character, "robust" (Siegel repeated median), "least_squares", or "robust slow" (Theil-Sen). Default "robust". ***"robust slow" is not reproducible*** : it samples randomly inside Praat's slope selection, so repeated runs on the same input differ (~0.8 dB observed) and can return values on the order of 1e290. That is an upstream Praat defect, reproduced faithfully here; pladdrr warns once per session when you select it.
cepstrogram_pitch_floor	Numeric, pitch floor for cepstrogram creation (default 60)
time_step	Numeric, time step for cepstrogram in seconds (default 0.002)
max_frequency	Numeric, max frequency for cepstrogram in Hz (default 5000)
pre_emphasis_from	Numeric, pre-emphasis frequency in Hz (default 50)

Details

Use this when analysing many files in a loop. For interactive or one-off use, the object API (`'sound$to_powercepstrogram(...)$get_cppps(...)'`) is equivalent and validates its inputs more forgivingly; this function skips that validation, so pass well-formed parameters.

This is a *****CPPS***** helper: it builds a whole-sound PowerCepstrogram and then computes a smoothed peak-prominence summary. If you need a single-interval *****CPP***** value that matches Praat's 'To PowerCepstrum' workflow, extract the interval, convert it to a Spectrum, and call `'to_power_cepstrum()$get_peak_prominence()'` instead. That path is much cheaper and is not the same metric.

Value

A single numeric CPPS value in dB.

Defaults differ from Praat's

These defaults follow the AVQI/clinical convention, **not** the defaults of Praat's 'PowerCepstrogram: Get CPPS...' dialog. On a 1 s test signal the two parameter sets give 9.92 dB and 4.82 dB respectively — a different measurement, not a rounding difference. Pass the Praat values explicitly to reproduce a Praat run:

```
| parameter | Praat default | pladdrr default | | | | | | | | 'time_averaging_window' | 0.02 | 0.001 | |
| 'quefrency_averaging_window' | 0.0005 | 0.0005 | | | 'pitch_floor' | 60 | 60 | | 'pitch_ceiling' | 330 |
333.3 | | 'qstart_fit' | 0.001 | 0.003 | | 'qend_fit' | 0.05 | 0.04 | | 'trend_line_type' | "exponential" |
"straight" | | 'fit_method' | "robust slow" | "robust" |
```

The 'fit_method' difference is deliberate beyond convention: Praat's "robust slow" (Theil-Sen) is not reproducible — see the 'fit_method' argument.

Examples

```
sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))

# Full-file CPPS calls on the 60 s demo file are CPU-heavy (> 5 s under
# R CMD check), so they live in \donttest{} (still run under
# --run-donttest).

# One-call CPPS with the default (AVQI-convention) parameters
calculate_cppps_fast(sound)

# The object API gives the identical value
pcep <- sound$to_powercepstrogram(60, 0.002, 5000, 50)
pcep$get_cppps()

# Reproduce Praat's own "Get CPPS..." defaults
calculate_cppps_fast(sound,
  time_averaging_window = 0.02, pitch_ceiling = 330,
  qstart_fit = 0.001, qend_fit = 0.05,
  trend_line_type = "exponential", fit_method = "least_squares"
)

# Single-interval CPP is a different, cheaper path
segment <- sound$extract_part(0, 0.5)
cpp <- segment$to_spectrum()$to_power_cepstrum()$get_peak_prominence(
  60, 333.3, "parabolic", 0.001, 0.05, "exponential decay", "robust slow"
)
```

calculate_cppps_ultra *Calculate CPPS with Optimized Single-Call (Tier 4 Ultra)*

Description

Computes CPPS from a ‘Sound’ in a single C++ call, building the PowerCepstrogram internally so no intermediate R object is created.

Returns the same value as `calculate_cppps_fast()` and as `sound$to_powercepstrogram(...)$get_cppps(...)`. The three paths cost about the same: the R/C++ boundary crossing is negligible next to the per-frame trend fit, so pick whichever reads best at the call site.

Usage

```
calculate_cppps_ultra(
  sound,
  time_averaging_window = 0.001,
  quefrency_averaging_window = 5e-04,
  pitch_floor = 60,
  pitch_ceiling = 333.3,
  subtract_trend = TRUE,
  time_step = 0.002,
  max_quefrency = 0.04,
  tolerance = 0.05,
  interpolation = "parabolic",
  tilt_line_quefrency = 0.003,
  line_type = "straight",
  fit_method = "robust",
  pre_emphasis_from = 50,
  max_frequency = 5000
)
```

Arguments

<code>sound</code>	Sound object or external pointer
<code>time_averaging_window</code>	Time averaging window in seconds (default 0.001)
<code>quefrency_averaging_window</code>	Quefrency averaging window in seconds (default 0.0005)
<code>pitch_floor</code>	Minimum F0 in Hz (default 60)
<code>pitch_ceiling</code>	Maximum F0 in Hz (default 333.3)
<code>subtract_trend</code>	Logical, subtract tilt before smoothing (default TRUE)
<code>time_step</code>	Time step for cepstrogram in seconds (default 0.002)
<code>max_quefrency</code>	End of the trend-fit quefrency window in seconds (default 0.04); 0 means autowindow to the full quefrency range (Praat convention).

tolerance	Tolerance for peak detection (default 0.05)
interpolation	Peak interpolation: "none", "parabolic", "cubic", "sinc70", "sinc700" (default "parabolic")
tilt_line_quefrequency	Start of the trend-fit quefrequency window in seconds (default 0.003).
line_type	Trend line type: "straight" or "exponential" (default "straight")
fit_method	Fitting method: "robust" (Siegel repeated median), "least_squares", or "robust slow" (Theil-Sen). Default "robust". **"robust slow" is not reproducible** — see 'calculate_cppps_fast()'.
pre_emphasis_from	Pre-emphasis frequency in Hz for the cepstrogram (default 50).
max_frequency	Maximum frequency in Hz for the cepstrogram (default 5000).

Details

Implements the complete CPPS pipeline in one C++ call, following the approach used in AVQI v2.03 and v3.01: PowerCepstrogram creation and CPPS extraction are consolidated, and no intermediate R object is allocated.

This does ****not**** make it meaningfully cheaper than the other CPPS entry points: the per-frame robust trend fit ('SlopeSelector::getSlope_Siegel') dominates CPPS runtime and is shared by every path; consolidating the PowerCepstrogram creation only removes the R/C++ boundary crossing, which is a small fraction of the total cost. Treat the choice as a matter of call-site convenience, not performance.

The defaults here match `calculate_cppps_fast()` and therefore also deviate from Praat's dialog defaults — see the "Defaults differ from Praat's" section of [calculate_cppps_fast](#).

This is still a ****CPPS**** helper. For a single-interval ****CPP**** measurement, use the segment's 'Spectrum -> PowerCepstrum -> get_peak_prominence()' path instead of 'calculate_cppps_ultra()'. It is both cheaper and closer to the Praat workflow used by voice-quality scripts that query one interval at a time.

****Use Cases:**** - AVQI v2.03/v3.01 implementation - High-throughput voice quality analysis - CPPS monitoring in latency-sensitive pipelines

Value

Numeric CPPS value in dB

Algorithm choice

Not applicable — this function is pitch-independent. It builds a 'PowerCepstrogram' directly from the Sound ('Sound_to_PowerCepstrogram()') and never extracts a 'Pitch' object, so there is no AC/CC or 'veryAccurate' choice to document here. See the CPPS parameter default table in 'CLAUDE.md' for the (non-pitch) parameters that do vary by caller, and the Tier 4 Ultra algorithm table in 'inst/agents/AGENT_GUIDE.md' for how this compares to the pitch-based Ultra functions.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)

# Tier 4 Ultra (same defaults as calculate_cppps_fast)
cpps <- calculate_cppps_ultra(sound)

# Should match calculate_cppps_fast() within 0.01 dB
cpps_fast <- calculate_cppps_fast(sound)
all.equal(cpps, cpps_fast, tolerance = 0.01)
```

`calculate_durations_simd_bridge`*Calculate Interval Durations with SIMD*

Description

Vectorized calculation of interval durations (end - start). Uses SIMD instructions on large interval counts.

Usage

```
calculate_durations_simd_bridge(start_times, end_times)
```

Arguments

<code>start_times</code>	Numeric vector of start times
<code>end_times</code>	Numeric vector of end times

Value

Numeric vector of durations

Examples

```
starts <- c(0, 1, 2.5)
ends <- c(0.8, 2, 3.2)
calculate_durations_simd_bridge(starts, ends)
```

 calculate_f0_stats_ultra

Calculate F0 Statistic in Single Call (Tier 4 Ultra)

Description

Performs pitch extraction AND statistic calculation entirely in C++, avoiding intermediate R6 object creation and the separate ‘to_pitch_cc()’ and ‘get_maximum()’ calls.

This is a **Tier 4 "Ultra"** function for batch DSI (Dysphonia Severity Index) calculations where maximum F0 is the FH (Highest Frequency) component.

Usage

```
calculate_f0_stats_ultra(
  sound,
  stat,
  min_pitch = 75,
  max_pitch = 600,
  time_step = 0,
  voicing_threshold = 0.45
)
```

Arguments

sound	A Sound object
stat	Statistic to compute: "max", "min", "mean", "median", or "sd"
min_pitch	Pitch floor in Hz (default: 75)
max_pitch	Pitch ceiling in Hz (default: 600)
time_step	Time step for pitch extraction (0 = auto)
voicing_threshold	Voicing threshold (default: 0.45)

Value

Single numeric value of the requested statistic in Hz

API Tier

This is a **Tier 4 "Ultra"** function. The entire pitch extraction and statistic computation happens in C++ with no intermediate R6 objects.

Algorithm choice

Pitch is always extracted with ‘Sound_to_Pitch_rawCc()’, ‘veryAccurate = TRUE’, ‘silenceThreshold = 0.03’ (matching DSI201.praat’s ‘To Pitch (cc)...’ step). Only ‘voicing_threshold’ is configurable; the AC/CC choice and ‘veryAccurate’ are not. See the Tier 4 Ultra algorithm table in ‘inst/agents/AGENT_GUIDE.md’ for how this compares to the other Ultra functions.

See Also

[Sound] for creating Sound objects [get_durations_batch()] for MPT component of DSI

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1, sampling_rate =
  16000)

# Get maximum F0 (DSI FH component)
max_f0 <- calculate_f0_stats_ultra(sound, stat = "max", min_pitch = 75,
  max_pitch = 600)

# Get mean F0
mean_f0 <- calculate_f0_stats_ultra(sound, stat = "mean")
```

calculate_midpoints_simd_bridge

Calculate Interval Midpoints with SIMD

Description

Calculate Interval Midpoints with SIMD

Usage

```
calculate_midpoints_simd_bridge(start_times, end_times)
```

Arguments

start_times	Numeric vector of start times
end_times	Numeric vector of end times

Value

Numeric vector of midpoints

Examples

```
calculate_midpoints_simd_bridge(c(0, 1, 2.5), c(0.8, 2, 3.2))
```

 calculate_minimum_intensity_ultra

Calculate Minimum Intensity in Voiced Regions (Tier 4 Ultra)

Description

Complete intensity pipeline in C++: Sound -> Pitch -> PointProcess -> TextGrid (VUV) -> Intensity -> Minimum in voiced regions, in one call instead of the equivalent Tier 2/3 workflow.

This is a **Tier 4 "Ultra"** function for batch DSI (Dysphonia Severity Index) calculations where minimum intensity is the IM (Intensity Minimum) component.

Usage

```
calculate_minimum_intensity_ultra(
  sound,
  min_pitch = 75,
  max_pitch = 600,
  time_step = 0,
  subtract_mean = TRUE
)
```

Arguments

sound	A Sound object
min_pitch	Pitch floor in Hz (default: 75)
max_pitch	Pitch ceiling in Hz (default: 600)
time_step	Time step for analysis (0 = auto)
subtract_mean	Whether to subtract mean for intensity calculation (default: TRUE)

Value

Minimum intensity in dB (in voiced regions only). Returns 'NA' if no voiced regions are detected.

API Tier

This is a **Tier 4 "Ultra"** function. The entire workflow (pitch extraction, PointProcess creation, VUV segmentation, intensity calculation, and minimum finding in voiced regions) happens in C++ with no intermediate R objects.

Algorithm choice

Pitch is always extracted with 'Sound_to_Pitch_rawCc()', 'veryAccurate = FALSE', 'silenceThreshold = 0.03', 'voicingThreshold = 0.8' — matching DSI201.praat's IM component, which uses a stricter voicing threshold than the jitter block. None of these are exposed as parameters (including 'voicing_threshold', unlike 'calculate_f0_stats_ultra()'). See the Tier 4 Ultra algorithm table in 'inst/agents/AGENT_GUIDE.md'.

See Also

[Sound] for creating Sound objects [calculate_f0_stats_ultra()] for FH component of DSI

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1, sampling_rate =
  16000)

# Get minimum intensity in voiced regions (DSI IM component)
min_int <- calculate_minimum_intensity_ultra(sound, min_pitch = 75)
```

```
calculate_multiband_hnr_ultra
```

Calculate multi-band HNR in a single call

Description

Optimized multi-band HNR calculation for VQ (Voice Quality) measurements. Computes HNR (mean + standard deviation) for 5 frequency bands in a single call: full spectrum, 0-500 Hz, 0-1500 Hz, 0-2500 Hz, 0-3500 Hz.

Usage

```
calculate_multiband_hnr_ultra(
  sound,
  bands = c(0, 500, 1500, 2500, 3500),
  time_step = 0.005,
  min_pitch = 75,
  from_time = 0,
  to_time = 0
)
```

Arguments

sound	Sound object or external pointer
bands	Numeric vector of upper frequency limits in Hz (default 'c(0, 500, 1500, 2500, 3500)', where '0' = full spectrum)
time_step	Time step for harmonicity in seconds (default '0.005')
min_pitch	Minimum pitch in Hz (default '75')
from_time	Start time for statistics extraction (default '0')
to_time	End time for statistics extraction (default '0')

Details

Use this when you only need one interval or whole-sound summary. For repeated interval queries on the same ‘Sound’, use [build_multiband_harmonicity()] once and then [multiband_hnr_stats()] for each interval.

Value

Named list with ‘*_mean’ and ‘*_sd’ values for each band.

Algorithm choice

Harmonicity is always computed with ‘Sound_to_Harmonicity_cc()’ (the CC method) for every band — there is no AC alternative and no way to configure it. This matches VQ_measurements_V2.praat lines 102-122. See the Tier 4 Ultra algorithm table in ‘inst/agents/AGENT_GUIDE.md’.

References

- VQ_measurements_V2.praat (Voice Quality measurements) - Maryn & Weenink (2015) - Multi-band HNR for voice quality

See Also

[build_multiband_harmonicity](#) and [multiband_hnr_stats](#) for the reusable multi-interval path

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)

hnr_results <- calculate_multiband_hnr_ultra(sound)
hnr_results$full_mean
```

Cepstrum

Praat Cepstrum Object

Description

Praat Cepstrum object with direct C++ module binding.

Details

The cepstrum is the inverse Fourier transform of the logarithm of the spectrum. Unlike PowerCepstrum, it preserves phase information.

Value

A Cepstrum object with methods for cepstral analysis and quefrency-domain processing.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
cepstrum <- sound$to_cepstrum()
spectrum <- cepstrum$to_spectrum()
```

cepstrum_plots	<i>PowerCepstrum and PowerCepstrogram Visualization Functions</i>
----------------	---

Description

ggplot2-based visualizations for power cepstrum objects, replacing Praat’s native graphics system with R’s powerful plotting capabilities.

Value

This is a documentation-only overview; see the individual functions ([plot_powercepstrum](#), [create_cepstrum_report](#)) for their return values.

Examples

```
# See individual functions, e.g. ?plot_powercepstrum
```

check_audio_quality	<i>Check Audio Quality Metrics</i>
---------------------	------------------------------------

Description

Analyzes a Sound object for common quality issues and returns diagnostic metrics. This function provides basic quality control checks useful for validating recording quality in production pipelines.

Usage

```
check_audio_quality(
  sound,
  clipping_threshold = 0.99,
  intensity_floor = 100,
  time_step = 0
)
```

Arguments

- sound A Sound object to analyze
- clipping_threshold Amplitude threshold for clipping detection (default 0.99)
- intensity_floor Minimum pitch for intensity calculation (default 100 Hz)
- time_step Time step for intensity analysis (0 = auto, default 0.0)

Details

This function is designed to catch common recording problems:

****Clipping Detection****: Identifies if the signal exceeds a threshold (default 0.99). Clipped recordings have distorted peaks and should typically be re-recorded.

****Intensity Analysis****: Uses Praat's intensity measurement to assess signal strength. Very low mean intensity may indicate recording level problems.

****Dynamic Range****: The difference between maximum and minimum intensity can help identify recordings with poor signal-to-noise ratio or excessive compression.

****Quality Criteria**** (general guidelines): - No clipping (is_clipped = FALSE) - Mean intensity: -20 to -10 dB for speech - Dynamic range: > 20 dB - Max amplitude: 0.7-0.9 range (good headroom without clipping)

Value

A list with the following components:

max_amplitude Maximum absolute amplitude in the recording

is_clipped Logical: TRUE if amplitude exceeds clipping_threshold

n_clipping_samples Number of samples above clipping threshold

clipping_percentage Percentage of samples that clip

mean_intensity_db Mean intensity in dB

min_intensity_db Minimum intensity in dB

max_intensity_db Maximum intensity in dB

intensity_range_db Dynamic range (max - min intensity)

rms_amplitude Root mean square amplitude

duration Total duration in seconds

sampling_frequency Sampling frequency in Hz

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
quality <- check_audio_quality(sound)

cat("Audio Quality Report:\n")
cat("  Duration:", quality$duration, "seconds\n")
cat("  Sampling rate:", quality$sampling_frequency, "Hz\n")
cat("  Clipped:", quality$is_clipped, "\n")
cat("  Max amplitude:", round(quality$max_amplitude, 3), "\n")
cat("  Mean intensity:", round(quality$mean_intensity_db, 1), "dB\n")
cat("  Dynamic range:", round(quality$intensity_range_db, 1), "dB\n")
```

Cochleagram

*Cochleagram***Description**

A Praat Cochleagram: the output of a bank of auditory filters modeling the basilar membrane, over time.

Arguments

.xptr Not for direct use. External pointer to the underlying C++ Cochleagram object; set internally when a method returns a new Cochleagram.

Details

Frequency runs along the Bark scale (0-25.6 Bark) rather than Hertz, matching the perceptual frequency resolution of the ear. Create one from a Sound with `sound$to_cochleagram()` or `sound$to_cochleagram_edb()`; slice it at a time point to get an Excitation pattern, from which loudness can be read off.

Value

A Cochleagram object.

Usage

```
cochleagram <- sound$to_cochleagram()
```

Query methods

- `get_start_time()`, `get_end_time()`, `get_duration()` - time domain in seconds
- `get_number_of_frames()`, `get_time_step()` - time sampling
- `get_time_from_column(i_col)` - time at a 1-based frame index
- `get_lowest_frequency()`, `get_highest_frequency()` - frequency range in Bark
- `get_number_of_frequency_bands()`, `get_frequency_step()` - frequency sampling
- `get_frequency_from_row()` - frequency at a 1-based band index
- `get_value_at_time_and_frequency(time, freq_bark)` - excitation level at a point
- `get_loudness_at_time(time)` - loudness (sone) at a time

Transformation and export

- `to_excitation(time)` - excitation pattern (Excitation object) at a time
- `get_difference(other, tmin, tmax)` - distance between two cochleagrams
- `as_matrix()` - values as a numeric matrix (frequency bands x frames)

See Also

[Excitation], [Sound]

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  44100)
cochleagram <- sound$to_cochleagram()
cochleagram$get_duration()
cochleagram$get_loudness_at_time(0.15)
```

ComplexSpectrogram	<i>ComplexSpectrogram</i>
--------------------	---------------------------

Description

Create a ComplexSpectrogram object from a Sound: a phase-preserving spectrogram computed with a complex FFT.

Arguments

sound	Sound object.
window_length	Window length in seconds. Default 0.005.
maximum_frequency	Maximum frequency to analyze, in Hz. Default 5000.

Value

An object of class ComplexSpectrogram (a list with methods, dispatched via the shared PraatObject pattern).

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
cs <- ComplexSpectrogram(sound)
cs$get_amplitude(0.15, 150)
```

create_cepstrum_report

Create Cepstrum Report Plot

Description

Creates a multi-panel diagnostic plot combining power cepstrum, cepstrogram, and CPP time series for comprehensive analysis.

Usage

```
create_cepstrum_report(
  cepstrogram,
  time_slice = NULL,
  save_path = NULL,
  format = c("png", "pdf", "svg"),
  dpi = 300
)
```

Arguments

cepstrogram	PowerCepstrogram object
time_slice	Numeric. Time point for extracting single cepstrum (default: middle)
save_path	Character. Path to save plot (optional)
format	Character. Output format: "png", "pdf", "svg" (default: "png")
dpi	Numeric. Resolution for raster formats (default: 300)

Value

A combined plot object (invisibly)

Examples

```
if (requireNamespace("gridExtra", quietly = TRUE)) {
  sound <- Sound$create_tone(frequency = 150, duration = 0.6, sampling_rate =
    16000)
  cepstrogram <- sound$to_powercepstrogram(pitch_floor = 60)

  # Create comprehensive report at t = 0.3s (mid-signal)
  create_cepstrum_report(cepstrogram, time_slice = 0.3)
}
```

create_file_list	<i>Create File List (Replaces Praat's Strings Object)</i>
------------------	---

Description

Create a file list similar to Praat's "Create Strings as file list". Returns a simple character vector of file paths.

Usage

```
create_file_list(  
  directory,  
  pattern = NULL,  
  full_names = TRUE,  
  recursive = FALSE  
)
```

Arguments

directory	Character path to directory
pattern	Regular expression pattern to match files
full_names	Logical, return full paths (default: TRUE)
recursive	Logical, search recursively (default: FALSE)

Value

Character vector of file paths

Examples

```
audio_dir <- tempfile("audio_")  
dir.create(audio_dir)  
tone <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =  
  16000)  
tone$save(file.path(audio_dir, "tone1.wav"))  
  
# Equivalent to: Create Strings as file list: "list", "*.wav"  
wav_files <- create_file_list(audio_dir, pattern = "\\*.wav$")
```

create_sound	Create a sound object from numeric values (DEPRECATED)
--------------	--

Description

****DEPRECATED:**** This S3 function is deprecated. Use the R6 interface instead: Sound\$from_values(values, sampling_rate)

Usage

```
create_sound(values, sampling_rate = 44100, start_time = 0)
```

Arguments

values	Numeric vector of amplitude values
sampling_rate	Sampling rate in Hz (default: 44100)
start_time	Start time in seconds (default: 0.0)

Value

Sound R6 object

Examples

```
# Old S3 approach (DEPRECATED, shown for reference)
sound <- create_sound(c(0.1, 0.2), sampling_rate = 1000)

# New R6 approach (RECOMMENDED)
sound2 <- Sound$from_values(c(0.1, 0.2), sampling_rate = 1000)
```

create_sound_from_values	Create a sound object from numeric vector
--------------------------	---

Description

Creates a praat_sound object structure from R numeric data

Usage

```
create_sound_from_values(values, sampling_rate = 44100, start_time = 0)
```

Arguments

values	Numeric vector of sound amplitude values
sampling_rate	Sampling rate in Hz (default: 44100)
start_time	Start time in seconds (default: 0.0)

Value

List representing a `praat_sound` object with values and metadata

Examples

```
values <- sin(2 * pi * 220 * seq(0, 0.1, length.out = 1000))
snd <- create_sound_from_values(values, sampling_rate = 10000)
```

create_window_xptr	<i>Create Common Window Function XPtr</i>
--------------------	---

Description

Convenience function to create pre-defined window functions as compiled XPtrs. Requires the `RcppXPtrUtils` package.

Usage

```
create_window_xptr(
  type = c("hamming", "hanning", "gaussian", "triangular", "blackman", "rectangular"),
  sigma = 0.25
)
```

Arguments

type	Character, one of "hamming", "hanning", "gaussian", "triangular", "blackman", "rectangular"
sigma	Numeric, standard deviation for Gaussian window (default 0.25)

Value

External pointer to compiled window function

Examples

```
# Compiling a C++ window function takes a few seconds
if (requireNamespace("RcppXPtrUtils", quietly = TRUE)) {
  hamming <- create_window_xptr("hamming")

  sound <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
    16000)
  windowed <- apply_window_xptr(sound, hamming)
}
```

Discriminant

*Discriminant***Description**

Linear Discriminant Analysis (LDA) for vowel classification, speaker identification, and other multivariate acoustic classification tasks.

Arguments

`.xptr` Not for direct use. External pointer to the underlying C++ Discriminant object; set internally when a method returns a new Discriminant (for example [discriminant_from_matrix](#)).

Details

Discriminant analysis is commonly used in phonetics for vowel classification, speaker identification, dialect/accent classification, and phoneme recognition.

Value

A Discriminant object with methods for querying discriminant functions, eigenvalues, and group statistics.

See Also

[PCA](#)

Examples

```
set.seed(1)
data <- matrix(c(rnorm(10, 500), rnorm(10, 700)), ncol = 1,
  dimnames = list(NULL, "f1"))
labels <- rep(c("a", "i"), each = 10)
disc <- discriminant_from_matrix(data, labels)
disc$get_number_of_groups()
disc$get_eigenvalues()
```

discriminant_from_matrix

Create Discriminant Analysis from labeled data

Description

Performs Linear Discriminant Analysis on a labeled numeric matrix.

Usage

```
discriminant_from_matrix(data, labels)
```

Arguments

data	Numeric matrix where rows are observations and columns are variables
labels	Character vector of group labels (one per row in data)

Value

A Discriminant object

Examples

```
# NOTE: only a single predictor column is used here; multi-column
# matrices are currently unreliable (see Discriminant class docs).
set.seed(1)
data <- matrix(c(rnorm(10, 500), rnorm(10, 700)), ncol = 1,
               dimnames = list(NULL, "f1"))
labels <- rep(c("a", "i"), each = 10)
disc <- discriminant_from_matrix(data, labels)
disc$get_number_of_groups()
```

DTW

DTW

Description

Temporal alignment between two acoustic signals.

Usage

```
DTW(.xptr = NULL)
```

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ DTW object; set internally when wrapping an existing alignment.
-------	--

Details

DTW aligns a candidate signal (x-axis) to a prototype or reference (y-axis), computing an optimal path through a distance matrix that minimizes the total distance while respecting slope constraints.

Value

A DTW object wrapping the alignment path and distance matrix.

Creating DTW objects

DTW objects are created by aligning two acoustic objects:

- `sounds_to_dtw(reference, candidate)` - align two sounds (most common)
- `mfccs_to_dtw(mfcc1, mfcc2)` - align MFCCs (speech recognition)
- `spectrograms_to_dtw(spec1, spec2)` - align spectrograms
- `pitches_to_dtw(pitch1, pitch2)` - align pitch contours

Time mapping

- `$get_y_time_from_x_time(tx)` - map candidate time to reference time
- `$get_x_time_from_y_time(ty)` - map reference time to candidate time
- `$map_times(times, direction)` - vectorized time mapping

TextGrid warping

- `$warp_textgrid(textgrid)` - warp annotation times

Path analysis

- `$get_weighted_distance()` - global alignment distance
- `$get_path_length()` - number of cells in optimal path
- `$get_path()` - full path as a data.frame
- `$get_maximum_consecutive_steps("x"|"y")` - path regularity

Slope constraints

Controls path flexibility:

- 1: No constraint (any slope)
- 2: $1/3 < \text{slope} < 3$
- 3: $1/2 < \text{slope} < 2$ (recommended)
- 4: $2/3 < \text{slope} < 3/2$ (strict)

See Also

[Sound](#), [Pitch](#), [Spectrogram](#), [MFCC](#)

Examples

```
# Basic alignment workflow
reference <- Sound$create_tone(frequency = 150, duration = 0.3)
candidate <- Sound$create_tone(frequency = 160, duration = 0.3)

# Create DTW alignment
dtw <- sounds_to_dtw(reference, candidate,
  analysis_width = 0.015,
  time_step = 0.005,
  band = 0.0,
  slope = 3
)

# Check alignment quality
cat("Distance:", dtw$get_weighted_distance(), "\n")

# Map time point from candidate to reference
ref_time <- dtw$get_y_time_from_x_time(0.1)
```

`duration_statistics_simd_bridge`*Calculate Duration Statistics with SIMD*

Description

Computes mean, standard deviation, min, and max of durations using SIMD.

Usage

```
duration_statistics_simd_bridge(durations)
```

Arguments

`durations` Numeric vector of durations

Value

List with mean, stdev, min, max

Examples

```
duration_statistics_simd_bridge(c(0.5, 0.8, 1.2, 0.3))
```

DurationTier

DurationTier

Description

Praat DurationTier object for duration manipulation, created via direct C++ module binding.

Arguments

tmin	Start time in seconds. Used with tmax to create a new, empty DurationTier.
tmax	End time in seconds. Used with tmin to create a new, empty DurationTier.
.xptr	Not for direct use. External pointer to the underlying C++ DurationTier object; set internally when a method returns a new DurationTier.

Details

DurationTiers are used together with Manipulation objects to modify the duration/tempo of sounds. Values represent duration multiplication factors:

- 1.0 - normal speed
- 2.0 - half speed (doubled duration)
- 0.5 - double speed (halved duration)

Value

A DurationTier object with methods for duration and tempo manipulation via time-value points.

Examples

```
dt <- DurationTier(0, 1)
dt$add_point(0.25, 1.0)
dt$add_point(0.75, 1.5)
dt$get_number_of_points()
dt$get_value_at_time(0.5)
```

Electroglottogram

Electroglottogram

Description

Praat Electroglottogram (EGG) object. Measures electrical impedance across the larynx, varying with vocal fold contact during phonation.

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ Electroglottogram object; set internally when a method returns a new Electroglottogram.
-------	--

Details

Electroglottogram inherits from Sound and represents a specialized single-channel sound that records vocal fold contact area.

Value

An Electroglottogram object (triple-class c("Electroglottogram", "Sound", "PraatObject")) that inherits Sound's methods in addition to its own.

Examples

```
egg <- electroglottogram_create(xmin = 0, xmax = 1, nx = 16000, dx = 1 /
  16000, x1 = 0)
egg$get_duration()
egg$get_number_of_samples()
egg$is_valid()
```

electroglottogram_create

Create an Electroglottogram object

Description

Create an Electroglottogram object

Usage

```
electroglottogram_create(xmin, xmax, nx, dx, x1)
```

Arguments

xmin	Start time in seconds
xmax	End time in seconds
nx	Number of samples
dx	Sampling period in seconds
x1	Time of first sample in seconds

Value

Electroglottogram object

Examples

```
egg <- electroglottogram_create(xmin = 0, xmax = 1, nx = 16000, dx = 1 /
  16000, x1 = 0)
egg$get_duration()
```

Excitation

Excitation

Description

Represents a Praat auditory excitation pattern.

Arguments

`.xptr` Not for direct use. External pointer to the underlying C++ Excitation object; set internally when a method returns a new Excitation.

Details

An excitation pattern is the perceptual loudness distribution across the auditory frequency range, measured on the Bark scale. It is produced from a Spectrum or a Cochleagram, not created directly.

Value

An Excitation object.

Query methods

- `get_loudness()` - total loudness in sones
- `get_value_at_frequency(freq_bark)` - excitation level at a given Bark frequency
- `get_distance(other)` - distance to another Excitation object

Conversion

- `to_formant(max_formants = 20)` - estimate formants from the excitation pattern

Export

- `as_vector()` - export as a numeric vector, one value per Bark bin
- `as_data_frame()` - export as a data.frame of frequency (Bark) and excitation

See Also

[Spectrum](#), [Formant](#)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
spectrum <- sound$to_spectrum()
excitation <- spectrum$to_excitation()
excitation$get_loudness()
```

```
extract_formant_parallel
```

Parallel Formant Extraction

Description

Extract formants from multiple files in parallel.

Usage

```
extract_formant_parallel(
  files,
  n_cores = NULL,
  time_step = 0.005,
  max_formants = 5,
  max_frequency = 5500
)
```

Arguments

files	Character vector of file paths
n_cores	Integer. Number of cores (default: auto)
time_step	Numeric. Time step in seconds (default: 0.005)
max_formants	Numeric. Max number of formants (default: 5)
max_frequency	Numeric. Max frequency in Hz (default: 5500)

Value

List of Formant objects

Examples

```
audio_dir <- tempfile("audio_")
dir.create(audio_dir)
tone <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
tone$save(file.path(audio_dir, "tone1.wav"))
files <- list.files(audio_dir, pattern = "\\wav$", full.names = TRUE)

# n_cores = 1 keeps this a single-process example (CRAN-safe)
formants <- extract_formant_parallel(files, n_cores = 1)
```

extract_formants	<i>Extract formants from a sound object (DEPRECATED)</i>
------------------	--

Description

****DEPRECATED:**** This function is deprecated in favor of the R6 interface. Use ‘sound\$to_formant_burg()’ instead.

Usage

```
extract_formants(
  sound,
  time_step = 0,
  max_formant = 5500,
  n_formants = 5,
  window_length = 0.025,
  pre_emphasis_from = 50
)
```

Arguments

sound	A <code>praat_sound</code> object created by read_sound or create_sound
time_step	Time step in seconds for formant analysis (0 = auto: 4x Nyquist)
max_formant	Maximum formant frequency in Hz (default: 5500 for adult female, use 5000 for adult male, 8000 for child)
n_formants	Number of formants to track (default: 5)
window_length	Analysis window length in seconds (default: 0.025)
pre_emphasis_from	Pre-emphasis frequency in Hz (default: 50)

Details

Analyzes formant frequencies (vocal tract resonances) from a sound object using Praat’s Burg algorithm.

Value

Depends on the class of sound:

- If sound is an R6 Sound object (the normal case — `Sound()/Sound$create_tone()` always create one), this function delegates entirely to `sound$to_formant_burg()` and returns an R6 Formant object. *This is what [get_formant_at_time](#)/[get_mean_formant](#) do NOT accept* — those two expect the legacy list below.

- If sound is a legacy (pre-R6) `praat_sound` list, returns a plain, **unclassed** list with elements values (a `data.frame` with columns `time`, `formant_number`, `frequency`, `bandwidth`), `n_frames`, `time_step`, `max_formant`, and `n_formants`. This list is no longer given a `"praat_formant"` class (removed when the package's S3 object system was fully migrated to R6), so it will not satisfy `is_praat_formant()` / `get_formant_at_time` / `get_mean_formant` without manually adding `class(x) <- "praat_formant"` first.

Examples

```
# sound is an R6 Sound object here, so this delegates to to_formant_burg()
# and returns an R6 Formant object (see the second value's \value above).
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
formants <- extract_formants(sound, max_formant = 5500)
f1_mean <- formants$get_mean(formant_number = 1)

# Equivalent, and the recommended way to spell it directly:
formants2 <- sound$to_formant_burg(max_frequency = 5500)
f1_mean2 <- formants2$get_mean(formant_number = 1)
```

<code>extract_intensity</code>	<i>Extract intensity from a sound object (DEPRECATED)</i>
--------------------------------	---

Description

****DEPRECATED:**** Use `sound$to_intensity()` instead.

Usage

```
extract_intensity(
  sound,
  time_step = 0,
  minimum_pitch = 100,
  subtract_mean = TRUE
)
```

Arguments

<code>sound</code>	A Sound R6 object
<code>time_step</code>	Time step in seconds
<code>minimum_pitch</code>	Minimum pitch in Hz
<code>subtract_mean</code>	Subtract mean intensity

Value

Intensity R6 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
intensity <- suppressWarnings(extract_intensity(sound))
intensity$get_mean(from_time = 0, to_time = 0)
```

 extract_intensity_parallel

Parallel Intensity Extraction

Description

Extract intensity from multiple files in parallel.

Usage

```
extract_intensity_parallel(
  files,
  n_cores = NULL,
  minimum_pitch = 100,
  time_step = 0
)
```

Arguments

files	Character vector of file paths
n_cores	Integer. Number of cores (default: auto)
minimum_pitch	Numeric. Minimum pitch for analysis (default: 100)
time_step	Numeric. Time step (default: 0, auto)

Value

List of Intensity objects

Examples

```
audio_dir <- tempfile("audio_")
dir.create(audio_dir)
tone <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
tone$save(file.path(audio_dir, "tone1.wav"))
files <- list.files(audio_dir, pattern = "\\wav$", full.names = TRUE)

# n_cores = 1 keeps this a single-process example (CRAN-safe)
intensities <- extract_intensity_parallel(files, n_cores = 1)
```

extract_measurements *Extract Measurements from Sound and TextGrid Pairs*

Description

High-level function to extract acoustic measurements aligned with TextGrid intervals. Automates the common Praat workflow of measuring formants/pitch at interval midpoints.

Usage

```
extract_measurements(
  sound,
  textgrid,
  tier = 1,
  measurements = c("pitch", "formants", "intensity"),
  time_point = c("midpoint", "start", "end", "mean"),
  pitch_params = list(time_step = 0.01, pitch_floor = 75, pitch_ceiling = 600),
  formant_params = list(time_step = 0.01, max_formants = 5, max_frequency = 5500,
    window_length = 0.025, pre_emphasis = 50),
  intensity_params = list(minimum_pitch = 100, time_step = 0, subtract_mean = TRUE)
)
```

Arguments

sound	Sound object or file path.
textgrid	TextGrid object or file path.
tier	Integer. Tier number to use for segmentation.
measurements	Character vector. Measurements to extract: "pitch", "formants", "intensity", etc.
time_point	Character. Where to measure: "midpoint" (default), "start", "end", or "mean".
pitch_params	List. Parameters for pitch extraction (time_step, pitch_floor, pitch_ceiling).
formant_params	List. Parameters for formant extraction (max_formants, max_frequency, etc.).
intensity_params	List. Parameters for intensity extraction.

Value

Data frame with one row per interval, columns for label and requested measurements.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.6, sampling_rate =
  16000)
tg <- textgrid_create(0, 0.6, "phones")
tg$insert_boundary("phones", 0.3)
tg$set_interval_text("phones", 1, "a")
tg$set_interval_text("phones", 2, "e")
```

```

results <- extract_measurements(
  sound = sound,
  textgrid = tg,
  tier = 1,
  measurements = c("pitch", "formants"),
  time_point = "midpoint"
)

```

extract_measurements_custom

Extract Custom Measurements from TextGrid Intervals

Description

Extract acoustic measurements from intervals or points in TextGrid annotations. This replaces complex Praat scripts that loop over TextGrid intervals.

Usage

```

extract_measurements_custom(
  sound,
  textgrid,
  tier,
  measures,
  interval_filter = NULL,
  aggregate_by = "interval"
)

```

Arguments

sound	Sound object or path to sound file
textgrid	TextGrid object or path to TextGrid file
tier	Integer or character, tier number or name
measures	Named list of measurement functions. Each function should accept (sound, tmin, tmax) and return a single value or named list
interval_filter	Optional function to filter intervals, receives interval label and should return TRUE/FALSE
aggregate_by	Character, how to aggregate: "interval" (default), "label", or "tier"

Value

Data frame with measurements for each interval/point

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 0.6, sampling_rate =
  16000)
tg <- textgrid_create(0, 0.6, "phones")
tg$insert_boundary("phones", 0.3)
tg$set_interval_text("phones", 1, "a")
tg$set_interval_text("phones", 2, "e")

measurements <- extract_measurements_custom(
  sound = sound,
  textgrid = tg,
  tier = "phones",
  measures = list(
    mean_f0 = function(snd, t1, t2) snd$to_pitch()$get_mean(0, 0, "hertz"),
    mean_intensity = function(snd, t1, t2) snd$to_intensity()$get_mean(0, 0)
  ),
  interval_filter = function(label) label %in% c("a", "e", "i", "o", "u")
)

```

extract_pitch

Extract pitch contour from sound (DEPRECATED)

Description

****DEPRECATED:**** This S3 function is deprecated. Use the R6 interface instead: `sound$to_pitch(time_step, pitch_floor, pitch_ceiling)`

Usage

```
extract_pitch(sound, pitch_floor = 75, pitch_ceiling = 600, time_step = 0.01)
```

Arguments

sound	A Sound R6 object
pitch_floor	Minimum pitch in Hz (default: 75)
pitch_ceiling	Maximum pitch in Hz (default: 600)
time_step	Time step in seconds (default: 0.01)

Value

Pitch R6 object

Examples

```
# Old S3 approach (DEPRECATED, shown for reference)
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- extract_pitch(sound, pitch_floor = 75, pitch_ceiling = 300)

# New R6 approach (RECOMMENDED)
pitch2 <- sound$to_pitch(pitch_floor = 75, pitch_ceiling = 300)
mean_f0 <- pitch2$get_mean()
```

 extract_pitch_parallel

Parallel Pitch Extraction

Description

Extract pitch from multiple files in parallel. Convenience wrapper around `analyze_files_parallel`.

Usage

```
extract_pitch_parallel(
  files,
  n_cores = NULL,
  pitch_floor = 75,
  pitch_ceiling = 600,
  time_step = 0
)
```

Arguments

<code>files</code>	Character vector of file paths
<code>n_cores</code>	Integer. Number of cores (default: auto)
<code>pitch_floor</code>	Numeric. Minimum pitch in Hz (default: 75)
<code>pitch_ceiling</code>	Numeric. Maximum pitch in Hz (default: 600)
<code>time_step</code>	Numeric. Time step (default: 0, auto)

Value

List of Pitch objects

Examples

```

audio_dir <- tempfile("audio_")
dir.create(audio_dir)
tone <- Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)
tone$save(file.path(audio_dir, "tone1.wav"))
files <- list.files(audio_dir, pattern = "\\wav$", full.names = TRUE)

# n_cores = 1 keeps this a single-process example (CRAN-safe)
pitches <- extract_pitch_parallel(files, n_cores = 1)

```

extract_textgrid_intervals

Extract Intervals Matching Criteria (Batch)

Description

Extract all intervals from a TextGrid tier that match specified text criteria, in a single call instead of a manual R loop.

Usage

```

extract_textgrid_intervals(
  textgrid,
  sound = NULL,
  tier,
  text_equals = NULL,
  text_contains = NULL,
  text_starts_with = NULL,
  extract_sounds = FALSE
)

```

Arguments

textgrid	A TextGrid object
sound	A Sound object (optional, required if extract_sounds = TRUE)
tier	Tier number (1-based) or tier name
text_equals	Exact label match (e.g., "V" for voiced)
text_contains	Substring match (e.g., "vowel")
text_starts_with	Prefix match (e.g., "IPA_")
extract_sounds	Logical. If TRUE, extract Sound parts for matched intervals

Value

List with components: - 'indices': Integer vector of matching interval indices - 'labels': Character vector of matching labels - 'start_times': Numeric vector of start times - 'end_times': Numeric vector of end times - 'n_total': Total number of intervals in tier - 'n_matched': Number of matching intervals - 'sounds': List of Sound objects (if extract_sounds = TRUE)

How It Works

A manual R loop makes one R<->C++ call per interval; this function does the whole tier scan in a single C++ call.

Comparison Types

Specify exactly ONE comparison criterion: - 'text_equals': Exact match (fastest) - 'text_contains': Substring search - 'text_starts_with': Prefix match

See Also

- [get_textgrid_labels_all()] to get all labels from a tier - [get_textgrid_interval_stats()] to compute statistics for all intervals

Examples

```
# Create sound and a voiced/unvoiced TextGrid
sound <- Sound$create_tone(frequency = 150, duration = 1, sampling_rate =
  16000)
pitch <- sound$to_pitch()
tg <- pitch$to_textgrid_vuv(0.02, 0.01)

# Extract all voiced intervals (batch), with the matching Sound parts
result <- extract_textgrid_intervals(
  textgrid = tg,
  sound = sound,
  tier = 1,
  text_equals = "V",
  extract_sounds = TRUE
)
voiced_sounds <- result$sounds

# Get interval durations without extracting sounds
result2 <- extract_textgrid_intervals(
  textgrid = tg,
  tier = 1,
  text_equals = "V",
  extract_sounds = FALSE
)
voiced_durations <- result2$end_times - result2$start_times
```

extract_voiced_segments

Extract Voiced Segments from Speech

Description

Voice activity detection combining intensity-based detection with optional Zero Crossing Rate (ZCR) filtering. Returns concatenated voiced segments as a Sound object. This matches the AVQI v2.03/v3.01 voiced extraction.

Usage

```
extract_voiced_segments(
    sound,
    minimum_pitch = 50,
    time_step = 0.003,
    silence_threshold = -25,
    min_silent_interval = 0.1,
    min_sounding_interval = 0.1,
    zcr_threshold = 3000,
    zcr_window = 0.03,
    use_zcr = TRUE,
    return_textgrid = FALSE
)
```

Arguments

sound	Sound object (continuous speech)
minimum_pitch	Numeric. Minimum pitch for intensity detection (Hz, default: 50)
time_step	Numeric. Time step for intensity analysis (s, default: 0.003)
silence_threshold	Numeric. Silence threshold in dB below max (default: -25)
min_silent_interval	Numeric. Minimum silence duration (s, default: 0.1)
min_sounding_interval	Numeric. Minimum voiced duration (s, default: 0.1)
zcr_threshold	Numeric. Maximum ZCR for voiced speech (Hz, default: 3000)
zcr_window	Numeric. ZCR analysis window duration (s, default: 0.03)
use_zcr	Logical. Apply ZCR filtering (default: TRUE)
return_textgrid	Logical. Also return VAD TextGrid (default: FALSE)

Details

The detection pipeline: 1. Intensity-based: Find segments above silence threshold 2. ZCR filtering (if 'use_zcr = TRUE'): Reject high-ZCR segments (unvoiced)

AVQI uses both intensity AND ZCR filtering. Set 'use_zcr = FALSE' for intensity-only detection.

Value

If 'return_textgrid = FALSE': Sound object with concatenated voiced segments. If 'return_textgrid = TRUE': List with 'sound' and 'textgrid' elements.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1, sampling_rate =
  16000)

# Full AVQI-compatible extraction (default)
voiced <- extract_voiced_segments(sound)

# Intensity-only (no ZCR filtering)
voiced_no_zcr <- extract_voiced_segments(sound, use_zcr = FALSE)

# With TextGrid output
result <- extract_voiced_segments(sound, return_textgrid = TRUE)
cat("Duration:", result$sound$get_duration(), "s\n")
```

extract_voiced_segments_ultra

Extract Voiced Segments with AVQI Filtering (Tier 4 Ultra)

Description

Complete AVQI voiced segment extraction pipeline in a single C++ call. Supports both AVQI v2.03 (simple intensity-based) and v3.01 (with ZCR filtering). Performs: Sound -> TextGrid (silence) -> Extract sounding -> Concatenate -> [v3.01: Window power/ZCR filtering] -> Final concatenation, in a single C++ call instead of a multi-step R implementation.

Usage

```
extract_voiced_segments_ultra(
  sound,
  version = "v3.01",
  min_pitch = 50,
  silence_threshold_db = -25,
  min_silent_duration = 0.1,
  min_sounding_duration = 0.1,
  power_threshold_factor = 0.3,
```

```

    max_zcr = 3000,
    window_width = 0.03
)

```

Arguments

<code>sound</code>	Sound object or external pointer
<code>version</code>	AVQI version: "v2.03" (simple) or "v3.01" (ZCR filtering, default)
<code>min_pitch</code>	Minimum pitch for silence detection in Hz (default 50)
<code>silence_threshold_db</code>	Silence threshold in dB (default -25)
<code>min_silent_duration</code>	Minimum silent interval duration in seconds (default 0.1)
<code>min_sounding_duration</code>	Minimum sounding interval duration in seconds (default 0.1)
<code>power_threshold_factor</code>	Power threshold as fraction of global power (default 0.3)
<code>max_zcr</code>	Maximum zero-crossing rate for voiced segments (default 3000)
<code>window_width</code>	Window width for v3.01 filtering in seconds (default 0.03)

Details

****TIER 4 ULTRA API****

This function implements the exact AVQI voiced extraction algorithm in a single C++ call. It replaces the R loops and multiple boundary crossings of the standard multi-step approach.

****Algorithm (AVQI v3.01):**** 1. Detect silences using intensity-based TextGrid creation 2. Extract all "sounding" intervals 3. Concatenate sounding intervals -> "loud_sound" 4. Slide 30ms windows through loud_sound 5. Filter windows by: power > 30 6. Concatenate passing windows -> final voiced sound

****Algorithm (AVQI v2.03):**** Steps 1-3 only (no window filtering)

****Version Differences:**** - v2.03: Simpler, keeps most voiced content (~37s from 37s input) - v3.01: Aggressive ZCR filtering, removes fricatives (~25-30s from 37s input)

Value

Sound object containing only voiced segments (concatenated)

Algorithm choice

No pitch algorithm is used here — voiced/silence segmentation is Intensity-threshold based ('Sound_to_Intensity()' + 'silence_threshold_db' relative to the maximum), not derived from a 'Pitch' object. 'min_pitch' only controls the Intensity analysis window length (via 'Sound_to_Intensity''s own floor parameter), not a pitch-detection algorithm choice. See the Tier 4 Ultra algorithm table in 'inst/agents/AGENT_GUIDE.md'.

References

- AVQI v3.01: Maryn et al. (2017) - with ZCR filtering - AVQI v2.03: Original Praat script - intensity-based only

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1, sampling_rate =
  16000)

# AVQI v3.01 (default, with ZCR filtering)
voiced_v3 <- extract_voiced_segments_ultra(sound, version = "v3.01")

# AVQI v2.03 (simple intensity-based)
voiced_v2 <- extract_voiced_segments_ultra(sound, version = "v2.03")

voiced_v3$get_total_duration()
voiced_v2$get_total_duration()
```

filter_by_duration_simd_bridge

Filter Intervals by Duration Range with SIMD

Description

Returns indices of intervals with duration in [min_dur, max_dur]. Uses SIMD for fast range comparison.

Usage

```
filter_by_duration_simd_bridge(durations, min_dur, max_dur)
```

Arguments

durations	Numeric vector of durations
min_dur	Minimum duration (inclusive)
max_dur	Maximum duration (inclusive)

Value

Integer vector of 1-based indices into durations whose value falls within [min_dur, max_dur]

Examples

```
filter_by_duration_simd_bridge(c(0.1, 0.5, 1.2, 0.3), 0.2, 0.8)
```

Formant

Formant

Description

Formant objects represent vocal tract resonance frequencies over time. Created from a Sound via formant tracking algorithms (Burg, Split-Levinson, or Willems). Formant frequencies and bandwidths are the primary acoustic correlates of vowel quality in speech.

Arguments

`.xptr` Not for direct use. External pointer to the underlying C++ Formant object; set internally when a method returns a new Formant.

Value

A Formant object with methods for querying formant frequencies and bandwidths at time points or across the full contour.

Information

- `get_number_of_frames()` - number of analysis frames
- `get_time_step()` - time step between frames (s)
- `get_min_num_formants()`, `get_max_num_formants()` - formant count range per frame

Point queries (single time)

- `get_value_at_time(formant_number, time, unit)` - formant frequency at time
- `get_bandwidth_at_time(formant_number, time, unit)` - formant bandwidth at time
- `get_all_values_at_time(time, max_formants, unit)` - all formant values at a time point

Statistics (over time range)

- `get_mean(formant_number, from_time, to_time, unit)` - mean formant frequency
- `get_standard_deviation(formant_number, from_time, to_time, unit)` - standard deviation
- `get_minimum(formant_number, from_time, to_time, unit)` - minimum value
- `get_maximum(formant_number, from_time, to_time, unit)` - maximum value
- `get_quantile(formant_number, quantile, from_time, to_time, unit)` - quantile
- `get_time_of_minimum(...)`, `get_time_of_maximum(...)` - time of extremum

Batch and vectorized

- `get_formant_track(formant_number, unit)` - full track for one formant
- `get_bandwidth_track(formant_number, unit)` - full bandwidth track
- `get_values_at_times(formant_number, times, unit)` - values at an arbitrary vector of times
- `get_all_formant_tracks(max_formants, unit)` - all formants as a matrix

Export

- `as_data_frame(max_formants)` - export as a `data.frame`, long format: one row per (frame, formant number), with columns time, formant, frequency (Hz), and bandwidth (Hz). Matches `FormantPath$as_data_frame()`.
- `save(filepath)` - save to a Praat binary file

Transform

- `to_formant_tier(formant_number)` - extract one formant as a `FormantTier`
- `to_formant_modeler()` - create a polynomial trajectory model
- `down_to_formant_tier()` - extract all formants as a `FormantTier`

See Also

[Sound](#), [LPC](#), [FormantPath](#), [FormantModeler](#)

Examples

```
# Self-contained example with generated tone
sound <- Sound$create_tone(duration = 1.0, frequency = 150, sampling_rate =
  44100)
formant <- sound$to_formant_burg(
  time_step = 0.01, max_number_of_formants = 5,
  maximum_formant = 5500, window_length = 0.025, pre_emphasis_from = 50
)
f1 <- formant$get_value_at_time(formant_number = 1, time = 0.5, unit =
  "hertz")

# The same analysis on a recording read from disk
sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))
formant <- sound$to_formant_burg(
  time_step = 0.01, max_number_of_formants = 5,
  maximum_formant = 5500, window_length = 0.025, pre_emphasis_from = 50
)
f1_at_02s <- formant$get_value_at_time(formant_number = 1, time = 0.2, unit =
  "hertz")
mean_f1 <- formant$get_mean(formant_number = 1, from_time = 0, to_time = 0,
  unit = "hertz")
```

FormantGrid

FormantGrid

Description

Praat FormantGrid object: formant frequencies and bandwidths over time, editable at arbitrary time points.

Arguments

tmin	Start time in seconds.
tmax	End time in seconds.
number_of_formants	Number of formants to track. Default 10.
initial_first_formant	Initial frequency of the first formant, in Hz. Default 550.
initial_formant_spacing	Initial spacing between formants, in Hz. Default 1100.
initial_first_bandwidth	Initial bandwidth of the first formant, in Hz. Default 60.
initial_bandwidth_spacing	Initial spacing between formant bandwidths, in Hz. Default 50.
.xptr	Not for direct use. External pointer to the underlying C++ FormantGrid object; set internally when a method returns a new FormantGrid.

Details

Used for voice transformation and synthesis. This is the editable counterpart to the read-only Formant object.

Value

A FormantGrid object with methods for formant frequency and bandwidth manipulation.

Examples

```
fg <- FormantGrid(0, 1, number_of_formants = 3)
fg$add_formant_point(1, 0.5, 700)
fg$get_formant_at_time(1, 0.5)
fg$get_number_of_formants()
```

FormantModeler

FormantModeler

Description

Robust formant trajectory modeling using polynomial fits with outlier detection.

Arguments

`.xptr` Not for direct use. External pointer to the underlying C++ FormantModeler object; set internally when a method returns a new FormantModeler.

Details

FormantModeler provides polynomial modeling of formant trajectories over time. It automatically identifies outliers and can find the optimal formant ceiling for a given sound, making it useful for robust formant analysis in noisy speech.

Value

A FormantModeler object with methods for polynomial modeling of formant trajectories with outlier detection.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
formant <- sound$to_formant_burg()
modeler <- formant$to_formant_modeler(tmin = 0, tmax = 0, num_tracks = 3,
  num_params = 3)
r2 <- modeler$get_coefficient_of_determination(1, 3)
f1_modeled <- modeler$get_track_model_values(1)
```

FormantPath

Create a FormantPath object from a Sound

Description

A FormantPath object represents multiple formant tracking candidates with different ceiling frequencies, allowing for robust formant analysis by automatically selecting the optimal tracking path.

Arguments

sound	A Sound object or path to audio file
time_step	Time step for analysis in seconds (must be > 0, typically 0.005)
max_num_formants	Maximum number of formants to track (typically 5)
formant_ceiling	Maximum formant frequency in Hz (typically 5000-5500)
window_length	Analysis window length in seconds (typically 0.025)
preemphasis_from	Preemphasis frequency in Hz (typically 50)
ceiling_step_fraction	Step size for ceiling frequency variation (0.05-0.1)
num_steps_up_down	Number of steps above/below ceiling (typically 2-4)

Value

An object of class FormantPath wrapping the set of candidate formant tracks (list with methods; dispatched via the shared PraatObject pattern).

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
fp <- FormantPath(sound)
fp$get_duration()
```

FormantTier	<i>FormantTier</i>
-------------	--------------------

Description

Praat FormantTier object: formant frequencies and bandwidths at discrete time points, with interpolation between points.

Arguments

tmin	Start time in seconds. Default 0.
tmax	End time in seconds. Default 1.
.xptr	Not for direct use. External pointer to the underlying C++ FormantTier object; set internally when a method returns a new FormantTier.

Details

The interpolated contour can be used to filter sounds for vowel modification or resynthesis.

Value

A FormantTier object with methods for formant frequency and bandwidth manipulation via time-value points.

Usage

```
FormantTier(tmin, tmax)          # create an empty FormantTier
FormantTier$from_formant(formant) # convert from a Formant object
```

Query methods

- `get_start_time()`, `get_end_time()`, `get_duration()` - time range in seconds
- `get_number_of_points()` - number of time points
- `get_min_num_formants()`, `get_max_num_formants()` - formant count across points
- `get_value_at_time(formant_number, time)` - formant frequency in Hz
- `get_bandwidth_at_time(formant_number, time)` - bandwidth in Hz

Transformation

- `filter_sound(sound, scale = TRUE)` - filter a sound through the formants
- `as_data_frame()` - export to a data frame
- `save(path)` - save to file

Examples

```
# Create from Formant analysis
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
formant <- sound$to_formant_burg()
ft <- FormantTier$from_formant(formant)

# Query formant values
f1 <- ft$get_value_at_time(1, 0.25) # F1 at 0.25s
f2 <- ft$get_value_at_time(2, 0.25) # F2 at 0.25s

# Filter a source sound
source <- Sound$create_tone(frequency = 100, duration = 0.5) # Buzz
vowel <- ft$filter_sound(source)
```

formanttier_from_formant

Create FormantTier from Formant

Description

Create FormantTier from Formant

Usage

```
formanttier_from_formant(formant)
```

Arguments

formant Formant object to convert

Value

FormantTier object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
formant <- sound$to_formant_burg()  
ft <- FormantTier$from_formant(formant)  
print(ft)
```

format_quality_report *Format Audio Quality Report*

Description

Creates a human-readable text report from audio quality metrics.

Usage

```
format_quality_report(quality_metrics, detailed = TRUE)
```

Arguments

quality_metrics Output from check_audio_quality()
detailed Include detailed metrics (default TRUE)

Value

Character string with formatted report

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
quality <- check_audio_quality(sound)  
report <- format_quality_report(quality)  
cat(report)
```

generate_noise	<i>Generate white noise</i>
----------------	-----------------------------

Description

Creates a `praat_sound` object containing white noise (random values from a normal distribution). Optionally specify a seed for reproducible noise.

Usage

```
generate_noise(duration, sampling_rate = 44100, amplitude = 1, seed = NULL)
```

Arguments

<code>duration</code>	Duration in seconds (must be positive)
<code>sampling_rate</code>	Sampling rate in Hz (default: 44100)
<code>amplitude</code>	Amplitude scaling factor (default: 1.0, must be positive). Controls the standard deviation of the noise.
<code>seed</code>	Optional random seed for reproducible noise generation. If <code>NULL</code> (default), noise will be different each time.

Details

White noise is generated using `rnorm()` with mean 0 and standard deviation controlled by the `amplitude` parameter. For reproducible results, specify a seed value.

Value

A `praat_sound` object containing white noise

Examples

```
# Generate 1 second of random noise
noise <- generate_noise(1.0)

# Generate reproducible noise
noise1 <- generate_noise(0.5, seed = 42)
noise2 <- generate_noise(0.5, seed = 42)
identical(noise1$values, noise2$values) # TRUE

# Generate quieter noise
quiet_noise <- generate_noise(1.0, amplitude = 0.1)
```

generate_sine_wave	<i>Generate a sine wave</i>
--------------------	-----------------------------

Description

Creates a `praat_sound` object containing a pure sine wave at a specified frequency. Useful for testing and creating reference signals.

Usage

```
generate_sine_wave(frequency, duration, sampling_rate = 44100, amplitude = 1)
```

Arguments

frequency	Frequency in Hz (must be positive)
duration	Duration in seconds (must be positive)
sampling_rate	Sampling rate in Hz (default: 44100)
amplitude	Peak amplitude (default: 1.0, must be positive)

Details

The generated sine wave follows the formula: $\text{amplitude} * \sin(2 * \pi * \text{frequency} * t)$ where t is time. The wave starts at phase 0 (value 0 at $t=0$).

Value

A `praat_sound` object containing the sine wave

Examples

```
# Generate A4 note (440 Hz) for 1 second
sine_a4 <- generate_sine_wave(440, 1.0)

# Generate lower amplitude sine at 1000 Hz
sine_quiet <- generate_sine_wave(1000, 0.5, amplitude = 0.3)
```

get_cppps_fast	<i>Get CPPS from PowerCepstrogram Pointer (Advanced Performance API)</i>
----------------	--

Description

Calculate CPPS from a PowerCepstrogram external pointer, bypassing R6 dispatch.

Usage

```
get_cppps_fast(
  powercepstrogram,
  subtract_tilt = FALSE,
  time_averaging_window = 0.01,
  quefrency_averaging_window = 0.001,
  pitch_floor = 60,
  pitch_ceiling = 330,
  delta_f0 = 0.05,
  interpolation = "parabolic",
  qstart_fit = 0.001,
  qend_fit = 0,
  trend_line_type = "straight",
  fit_method = "robust"
)
```

Arguments

powercepstrogram	External pointer to PowerCepstrogram object
subtract_tilt	Logical, subtract tilt before calculating CPPS (default FALSE)
time_averaging_window	Numeric, time averaging window in seconds (default 0.01)
quefrency_averaging_window	Numeric, quefrency averaging window in seconds (default 0.001)
pitch_floor	Numeric, minimum F0 in Hz (default 60)
pitch_ceiling	Numeric, maximum F0 in Hz (default 330)
delta_f0	Numeric, F0 fractional precision (default 0.05)
interpolation	Character, one of "parabolic", "none", "cubic", "sinc70", "sinc700" (default "parabolic")
qstart_fit	Numeric, quefrency range start for fitting in seconds (default 0.001)
qend_fit	Numeric, quefrency range end in seconds (default 0, means auto)
trend_line_type	Character, "straight" or "exponential" (default "straight")

fit_method Character, "robust" (Siegel repeated median), "least_squares", or "robust slow" (Theil-Sen). Default "robust". ****"robust slow" is not reproducible****: it samples randomly inside Praat's slope selection, so repeated runs on the same input differ (~0.8 dB observed) and can return values on the order of 1e290. That is an upstream Praat defect, reproduced faithfully here; pladdrr warns once per session when you select it.

Details

****ADVANCED API**** - Takes the external pointer returned by 'to_powercepstrogram_fast()' instead of a 'PowerCepstrogram' R6 object.

Useful when you need to calculate CPPS multiple times with different parameters from the same PowerCepstrogram object.

Value

Numeric CPPS value in dB

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)

# Create cepstrogram once
pcep_ptr <- to_powercepstrogram_fast(sound)

# Calculate CPPS with different parameters
cpps1 <- get_cppps_fast(pcep_ptr, subtract_tilt = FALSE, pitch_floor = 60)
cpps2 <- get_cppps_fast(pcep_ptr, subtract_tilt = TRUE, pitch_floor = 80)
```

get_duration	<i>Get duration of sound object (DEPRECATED)</i>
--------------	--

Description

****DEPRECATED:**** Use sound\$get_duration() instead.

Usage

```
get_duration(sound)
```

Arguments

sound A Sound R6 object

Value

Duration in seconds

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
suppressWarnings(get_duration(sound))
```

get_durations_batch *Get Audio File Durations via WAV Header Reading*

Description

Reads only the 44-byte WAV header to calculate duration, avoiding full file loading via ‘LongSound\$from_file()\$get_total_duration()’.

This is a **Tier 4 "Ultra"** function for batch DSI (Dysphonia Severity Index) calculations where file duration is the MPT (Maximum Phonation Time) component.

Usage

```
get_durations_batch(file_paths)
```

Arguments

file_paths Character vector of .wav file paths

Value

Numeric vector of durations (seconds). Returns ‘NA’ for files that cannot be read or are not valid WAV files.

Performance

This function avoids the overhead of the standard LongSound approach by:

- Reading only the first 44-100 bytes of the WAV header
- Avoiding memory allocation for audio samples
- Skipping all Praat object construction

API Tier

This is a **Tier 4 "Ultra"** function. Tier 4 functions keep entire workflows in the C++ layer to minimize R<->C++ boundary crossings, returning only final scalar or simple vector results.

See Also

[LongSound()] for full audio file access when you need more than duration

Examples

```
wav1 <- tempfile(fileext = ".wav")
wav2 <- tempfile(fileext = ".wav")
Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)$save(wav1)
Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)$save(wav2)

# Multiple files (DSI workflow)
durations <- get_durations_batch(c(wav1, wav2))
max_mpt <- max(durations, na.rm = TRUE)
```

get_formant_at_time	<i>Get formant frequency at a specific time (DEPRECATED)</i>
---------------------	--

Description

****DEPRECATED:**** Use the R6 interface instead: `formant$get_value_at_time()`

Usage

```
get_formant_at_time(formant, formant_number, time, interpolate = FALSE)
```

Arguments

formant	A legacy praat_formant-shaped list: a plain list with a values element (a data.frame with columns time, formant_number, frequency, bandwidth) and a class attribute of "praat_formant". extract_formants() no longer produces this (it now returns an R6 Formant object instead — see its extract_formants documentation); build one by hand for this legacy function, or use <code>formant\$get_value_at_time()</code> on an R6 Formant object directly instead of this deprecated wrapper.
formant_number	Which formant (1 = F1, 2 = F2, etc.)
time	Time in seconds
interpolate	Logical; if TRUE, interpolate between frames

Value

Formant frequency in Hz, or NA if undefined

Examples

```
# A praat_formant-shaped list (see the @param formant description
# above); built directly here for a self-contained example.
formant <- structure(
  list(
    values = data.frame(
```

```

        time = c(0.1, 0.1, 0.2, 0.2),
        formant_number = c(1, 2, 1, 2),
        frequency = c(500, 1500, 520, 1480),
        bandwidth = c(80, 120, 85, 110)
    ),
    n_frames = 2,
    n_formants = 2
),
class = "praat_formant"
)
get_formant_at_time(formant, formant_number = 1, time = 0.15)

```

get_formant_bandwidths_at_times

Batch Query Formant Bandwidths at Multiple Times

Description

Query formant bandwidths at multiple time points in a single function call.

Usage

```

get_formant_bandwidths_at_times(
  formant,
  times,
  formant_numbers = 1:4,
  unit = "hertz"
)

```

Arguments

formant	A Formant object
times	Numeric vector of time points (in seconds)
formant_numbers	Integer vector specifying which formants (default '1:4')
unit	Unit for bandwidth values: "hertz" (default) or "bark"

Value

A list with one element per formant number (e.g., 'B1', 'B2', ...), each containing a numeric vector of bandwidths.

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
formant <- sound$to_formant_burg()
times <- seq(0.1, 0.4, length.out = 5)
bandwidths <- get_formant_bandwidths_at_times(formant, times, 1:4)
```

`get_formant_value_direct`*Get Single Formant Value Directly*

Description

Get Single Formant Value Directly

Usage

```
get_formant_value_direct(formant, formant_number, time, unit = "hertz")
```

Arguments

<code>formant</code>	Formant object or external pointer
<code>formant_number</code>	Formant number (1=F1, etc)
<code>time</code>	Time in seconds
<code>unit</code>	Unit string

Value

Formant frequency

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
formant_ptr <- to_formant_direct(sound)
get_formant_value_direct(formant_ptr, 1, 0.25)
```

get_formants_at_times Batch Query Formant Frequencies at Multiple Times

Description

Query formant frequencies (F1, F2, F3, F4, etc.) at multiple time points in a single function call, instead of calling 'get_value_at_time()' repeatedly in a loop.

Usage

```
get_formants_at_times(formant, times, formant_numbers = 1:4, unit = "hertz")
```

Arguments

formant	A Formant object
times	Numeric vector of time points (in seconds)
formant_numbers	Integer vector specifying which formants to extract (e.g., '1:4' for F1-F4). Default is '1:4'.
unit	Unit for formant values: "hertz" (default) or "bark"

Value

A list with one element per formant number (e.g., 'F1', 'F2', ...), each containing a numeric vector of formant frequencies at the specified times.

Performance

This function reduces R<->C++ boundary crossings from '4n' calls (for 4 formants at n times) to just 1 call.

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
formant <- sound$to_formant_burg()

# Extract F1-F4 at 5 time points
times <- seq(0.1, 0.4, length.out = 5)
result <- get_formants_at_times(formant, times, formant_numbers = 1:4)

# Access individual formants
f1_vals <- result$F1
f2_vals <- result$F2
```

get_formants_direct *Get Formant F1-F4 at Time Directly*

Description

Get F1, F2, F3, F4 at a single time point in one call.

Usage

```
get_formants_direct(formant, time, unit = c("hertz", "bark"))
```

Arguments

formant	Formant object or external pointer
time	Time in seconds
unit	Character: "hertz" or "bark"

Value

Named numeric vector: F1, F2, F3, F4

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
formant <- sound$to_formant_burg()

# Get all 4 formants in one call
f1_f4 <- get_formants_direct(formant, time = 0.25)

# Equivalent R6 calls, one boundary crossing per formant:
f1 <- formant$get_value_at_time(1, 0.25, "hertz")
f2 <- formant$get_value_at_time(2, 0.25, "hertz")
```

get_intensity_at_time *Get intensity at a specific time (DEPRECATED)*

Description

****DEPRECATED:**** Use intensity\$get_value_at_time(time) instead.

Usage

```
get_intensity_at_time(intensity, time, interpolate = FALSE)
```


Arguments

intensity	An Intensity R6 object
time	Time in seconds
interpolate	Whether to interpolate

Value

Intensity in dB

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
intensity <- sound$to_intensity()  
suppressWarnings(get_intensity_at_time(intensity, 0.25))
```

get_intensity_at_times

Batch Query Intensity Values at Multiple Times

Description

Query intensity (amplitude in dB) at multiple time points in a single call.

Usage

```
get_intensity_at_times(  
  intensity,  
  times,  
  interpolate = "cubic",  
  interpolation = NULL  
)
```

Arguments

intensity	An Intensity object
times	Numeric vector of time points (in seconds)
interpolate	Interpolation method: "nearest", "linear", "cubic" (default), or "sinc70". Kept for backward compatibility; prefer 'interpolation'.
interpolation	Alias for 'interpolate' (consistent with R6 method naming). When provided, supersedes 'interpolate'.

Value

Numeric vector of intensity values (in dB) at the specified times

Performance

Uses existing optimized C++ code.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
intensity <- sound$to_intensity()  
times <- seq(0.05, 0.45, length.out = 10)  
intensities <- get_intensity_at_times(intensity, times)
```

get_intensity_value_direct

Get Single Intensity Value Directly

Description

Get Single Intensity Value Directly

Usage

```
get_intensity_value_direct(intensity, time, interpolation = "cubic")
```

Arguments

intensity	Intensity object or external pointer
time	Time in seconds
interpolation	Interpolation method

Value

Intensity in dB

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
intensity_ptr <- to_intensity_direct(sound)  
get_intensity_value_direct(intensity_ptr, 0.25)
```

`get_interval_predicate`*Create Built-in Interval Predicates*

Description

Returns external pointers to pre-compiled predicates for common filtering tasks. Use these instead of compiling your own for simple cases.

Usage

```
get_interval_predicate(type, threshold = 0)
```

Arguments

type	Predicate type: "non_empty", "min_duration", "max_duration"
threshold	Numeric threshold (for duration predicates)

Details

Available predicates: - "non_empty": Matches intervals with non-empty labels (label[0] != '\0') - "min_duration": Matches intervals with duration >= threshold - "max_duration": Matches intervals with duration <= threshold

Value

External pointer to predicate function

Examples

```
tg <- TextGrid$create(0, 1, "words")
tg$insert_boundary(1, 0.5)
tg$set_interval_text(1, 2, "hello")

# Get all non-empty intervals
pred <- get_interval_predicate("non_empty")
result <- textgrid_filter_xptr(tg$.xptr, 1, pred)

# Get intervals longer than 100ms
pred <- get_interval_predicate("min_duration", 0.1)
result <- textgrid_filter_xptr(tg$.xptr, 1, pred)
```

get_jitter_shimmer_batch

Get All Jitter and Shimmer Measures in One Call

Description

Returns 11 voice quality measures (5 jitter, 6 shimmer) in a single C++ call, instead of calling individual methods separately for each measure.

****Jitter measures**** (period perturbation): - 'jitter_local': Local jitter (relative, fraction) - 'jitter_local_abs': Local absolute jitter (seconds) - 'jitter_rap': Relative average perturbation - 'jitter_ppq5': 5-point period perturbation quotient - 'jitter_ddp': Difference of differences of periods

****Shimmer measures**** (amplitude perturbation): - 'shimmer_local': Local shimmer (relative, fraction) - 'shimmer_local_db': Local shimmer (dB) - 'shimmer_apq3': 3-point amplitude perturbation quotient - 'shimmer_apq5': 5-point amplitude perturbation quotient - 'shimmer_apq11': 11-point amplitude perturbation quotient - 'shimmer_dda': Difference of differences of amplitudes

Usage

```
get_jitter_shimmer_batch(
    pointprocess,
    sound,
    from_time = 0,
    to_time = 0,
    period_floor = 1e-04,
    period_ceiling = 0.02,
    max_period_factor = 1.3,
    max_amplitude_factor = 1.6
)
```

Arguments

pointprocess	PointProcess object or external pointer (glottal pulses)
sound	Sound object or external pointer (required for shimmer)
from_time	Start time (0 = beginning)
to_time	End time (0 = end)
period_floor	Minimum period in seconds (default 0.0001 = 10000 Hz)
period_ceiling	Maximum period in seconds (default 0.02 = 50 Hz)
max_period_factor	Maximum period factor (default 1.3)
max_amplitude_factor	Maximum amplitude factor (default 1.6)

Value

Named list with 11 voice quality measures

Performance

This function reduces R<->C++ boundary crossings from 11 calls to 1 call.

Workflow

For accurate voice quality analysis, use 'to_point_process_from_sound_and_pitch()' which uses the refined pitch contour to guide period detection: ““r # Recommended workflow sound <- Sound("voice.wav") pitch <- sound\$to_pitch_cc(voicing_threshold = 0.45) pp <- to_point_process_from_sound_and_pitch(sound, pitch) metrics <- get_jitter_shimmer_batch(pp, sound) ““

See Also

[two_pass_adaptive_pitch()] for robust pitch extraction [to_point_process_from_sound_and_pitch()] for accurate pulse detection

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pp <- sound$to_point_process_periodic_cc(75, 600)

# Get all voice quality measures at once
metrics <- get_jitter_shimmer_batch(pp, sound)

# Access individual measures
cat("Jitter (local):", metrics$jitter_local * 100, "%\n")
cat("Shimmer (local):", metrics$shimmer_local * 100, "%\n")
cat("Shimmer (dB):", metrics$shimmer_local_db, "dB\n")
```

get_max_intensity	<i>Get maximum intensity (DEPRECATED)</i>
-------------------	---

Description

****DEPRECATED:**** Use intensity\$get_maximum() instead.

Usage

```
get_max_intensity(intensity, time_range = NULL)
```

Arguments

intensity	An Intensity R6 object
time_range	Optional time range c(start, end)

Value

Maximum intensity in dB

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
intensity <- sound$to_intensity()
suppressWarnings(get_max_intensity(intensity))
```

get_max_pitch	<i>Get maximum pitch (DEPRECATED)</i>
---------------	---------------------------------------

Description

****DEPRECATED:**** Use pitch\$get_maximum(unit) instead.

Usage

```
get_max_pitch(pitch, unit = "Hz", time_range = NULL)
```

Arguments

pitch	A Pitch R6 object
unit	Unit: "Hz" or "semitones"
time_range	Optional time range c(start, end)

Value

Maximum pitch value

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- sound$to_pitch()
suppressWarnings(get_max_pitch(pitch))
```

get_mean_formant	<i>Get mean formant frequency (DEPRECATED)</i>
------------------	--

Description

****DEPRECATED:**** Use the R6 interface instead: 'formant\$get_mean()'

Usage

```
get_mean_formant(formant, formant_number, time_range = NULL)
```

Arguments

- formant** A legacy `praat_formant`-shaped list: a plain list with a `values` element (a `data.frame` with columns `time`, `formant_number`, `frequency`, `bandwidth`) and a `class` attribute of `"praat_formant"`. `extract_formants()` no longer produces this (it now returns an R6 `Formant` object instead — see its [extract_formants](#) documentation); build one by hand for this legacy function, or use `formant$get_mean()` on an R6 `Formant` object directly instead of this deprecated wrapper.
- formant_number** Which formant (1 = F1, 2 = F2, etc.)
- time_range** Optional numeric vector `c(start, end)` in seconds

Value

Mean formant frequency in Hz, or NA if undefined

Examples

```
# A praat_formant-shaped list (see the @param formant description
# above); built directly here for a self-contained example.
formant <- structure(
  list(
    values = data.frame(
      time = c(0.1, 0.1, 0.2, 0.2),
      formant_number = c(1, 2, 1, 2),
      frequency = c(500, 1500, 520, 1480),
      bandwidth = c(80, 120, 85, 110)
    ),
    n_frames = 2,
    n_formants = 2
  ),
  class = "praat_formant"
)
get_mean_formant(formant, formant_number = 1)
```

`get_mean_intensity` *Get mean intensity (DEPRECATED)*

Description

****DEPRECATED:**** Use `intensity$get_mean()` instead.

Usage

```
get_mean_intensity(intensity, time_range = NULL)
```

Arguments

- intensity** An `Intensity` R6 object
- time_range** Optional time range `c(start, end)`

Value

Mean intensity in dB

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
intensity <- sound$to_intensity()
suppressWarnings(get_mean_intensity(intensity))
```

get_mean_pitch	<i>Get mean pitch (DEPRECATED)</i>
----------------	------------------------------------

Description

****DEPRECATED:**** Use `pitch$get_mean(unit)` instead.

Usage

```
get_mean_pitch(pitch, unit = "Hz", time_range = NULL)
```

Arguments

pitch	A Pitch R6 object
unit	Unit: "Hz" or "semitones"
time_range	Optional time range c(start, end)

Value

Mean pitch value

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- sound$to_pitch()
suppressWarnings(get_mean_pitch(pitch))
```

get_min_intensity	<i>Get minimum intensity (DEPRECATED)</i>
-------------------	---

Description

****DEPRECATED:**** Use `intensity$get_minimum()` instead.

Usage

```
get_min_intensity(intensity, time_range = NULL)
```

Arguments

intensity	An Intensity R6 object
time_range	Optional time range c(start, end)

Value

Minimum intensity in dB

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
intensity <- sound$to_intensity()  
suppressWarnings(get_min_intensity(intensity))
```

get_min_pitch	<i>Get minimum pitch (DEPRECATED)</i>
---------------	---------------------------------------

Description

****DEPRECATED:**** Use `pitch$get_minimum(unit)` instead.

Usage

```
get_min_pitch(pitch, unit = "Hz", time_range = NULL)
```

Arguments

pitch	A Pitch R6 object
unit	Unit: "Hz" or "semitones"
time_range	Optional time range c(start, end)

Value

Minimum pitch value

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
pitch <- sound$to_pitch()  
suppressWarnings(get_min_pitch(pitch))
```

get_n_channels	<i>Get number of channels in sound object (DEPRECATED)</i>
----------------	--

Description

****DEPRECATED:**** Use `sound$get_number_of_channels()` instead.

Usage

```
get_n_channels(sound)
```

Arguments

sound	A Sound R6 object
-------	-------------------

Value

Number of channels

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
suppressWarnings(get_n_channels(sound))
```

get_n_samples	<i>Get number of samples in sound object (DEPRECATED)</i>
---------------	---

Description

****DEPRECATED:**** Use `sound$get_number_of_samples()` instead.

Usage

```
get_n_samples(sound)
```

Arguments

sound	A Sound R6 object
-------	-------------------

Value

Number of samples

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
suppressWarnings(get_n_samples(sound))
```

get_pitch_at_time	<i>Get pitch at specific time point (DEPRECATED)</i>
-------------------	--

Description

****DEPRECATED:**** Use `pitch$get_value_at_time(time, unit)` instead.

Usage

```
get_pitch_at_time(pitch, time, unit = "Hz", interpolate = FALSE)
```

Arguments

pitch	A Pitch R6 object
time	Time in seconds
unit	Unit: "Hz" or "semitones"
interpolate	Whether to interpolate

Value

Pitch value or NA

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- sound$to_pitch()
get_pitch_at_time(pitch, 0.25)

```

get_pitch_at_times	<i>Batch Query Pitch Values at Multiple Times</i>
--------------------	---

Description

Query pitch (F0) values at multiple time points in a single function call, instead of repeated calls to 'get_value_at_time()'.

Usage

```
get_pitch_at_times(pitch, times, unit = "hertz", interpolate = TRUE)
```

Arguments

pitch	A Pitch object
times	Numeric vector of time points (in seconds)
unit	Unit for pitch values: "hertz" (default), "mel", "loghertz", "semitones", or "erb"
interpolate	Logical; whether to interpolate between frames (default TRUE)

Value

Numeric vector of pitch values at the specified times

Performance

This function uses existing optimized C++ code.

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- sound$to_pitch()
times <- seq(0.05, 0.45, length.out = 10)
f0_contour <- get_pitch_at_times(pitch, times)

```

`get_pitch_mean_direct` *Get Pitch Mean Directly (Bypass R6)*

Description

Get mean pitch value without R6 wrapper overhead.

Usage

```
get_pitch_mean_direct(  
  pitch,  
  from_time = 0,  
  to_time = 0,  
  unit = c("hertz", "semitones", "mel", "erb", "loghertz")  
)
```

Arguments

<code>pitch</code>	Pitch object or external pointer
<code>from_time</code>	Start time (0 = beginning)
<code>to_time</code>	End time (0 = end)
<code>unit</code>	Pitch unit

Value

Mean pitch value

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
pitch_ptr <- to_pitch_cc_direct(sound)  
get_pitch_mean_direct(pitch_ptr)
```

`get_pitch_quantile_direct`*Get Pitch Quantile Directly (Bypass R6)*

Description

Get a specific quantile of pitch values without R6 wrapper overhead. Useful for VUV analysis workflows where you need Q1, Q3 for adaptive pitch range.

Usage

```
get_pitch_quantile_direct(
  pitch,
  quantile,
  from_time = 0,
  to_time = 0,
  unit = c("hertz", "semitones", "mel", "erb", "loghertz")
)
```

Arguments

pitch	Pitch object or external pointer
quantile	Quantile value (0.25 for Q1, 0.75 for Q3, 0.5 for median)
from_time	Start time (0 = beginning)
to_time	End time (0 = end)
unit	Pitch unit ("hertz", "semitones", "mel", "erb", "loghertz")

Value

Quantile value in specified unit

See Also

[get_pitch_quantiles_batch](#) for getting multiple quantiles at once

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch_ptr <- to_pitch_cc_direct(sound)
q1 <- get_pitch_quantile_direct(pitch_ptr, 0.25)
q3 <- get_pitch_quantile_direct(pitch_ptr, 0.75)
```

```
get_pitch_quantiles_batch
```

Get Multiple Pitch Quantiles in Single Call (NEW for VUV Performance)

Description

Extract multiple quantiles (e.g., Q1, Q3) from a Pitch object in a single C++ call instead of calling ‘get_quantile()’ multiple times. Specifically designed for VUV analysis workflows where adaptive pitch ranges are calculated from quantiles.

Usage

```
get_pitch_quantiles_batch(
  pitch,
  quantiles,
  from_time = 0,
  to_time = 0,
  unit = "hertz"
)
```

Arguments

pitch	A Pitch object
quantiles	Numeric vector of quantile values (e.g., c(0.25, 0.75) for Q1 and Q3)
from_time	Start time (0 = beginning of pitch object)
to_time	End time (0 = end of pitch object)
unit	Unit for pitch values: "hertz" (default), "mel", "loghertz", "semitones", or "erb"

Value

Named numeric vector with quantile values (names like "q0.25", "q0.75")

Performance

Reduces R<->C++ boundary crossings from n separate calls to 1 call.

Use Case - VUV Analysis

```
““r # Extract adaptive pitch range for refined pitch analysis
quantiles <- get_pitch_quantiles_batch(pitch_rough,
c(0.25, 0.75))
pitch_refined <- to_pitch_cc_direct( sound, pitch_floor = quantiles["q0.25"] * 0.75,
pitch_ceiling = quantiles["q0.75"] * 1.5 ) ““
```

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
16000)
pitch <- sound$to_pitch()

# Get Q1, median, Q3 in one call
quantiles <- get_pitch_quantiles_batch(pitch, c(0.25, 0.5, 0.75))

# Access by name
q1 <- quantiles["q0.25"]
median <- quantiles["q0.5"]
q3 <- quantiles["q0.75"]
```

```
get_pitch_stats_direct
```

Get Pitch Statistics Directly

Description

Get all common pitch statistics in a single call, bypassing R6 dispatch.

Usage

```
get_pitch_stats_direct(
  pitch,
  from_time = 0,
  to_time = 0,
  unit = c("hertz", "semitones", "mel", "erb", "loghertz")
)
```

Arguments

pitch	Pitch object or external pointer
from_time	Start time (0 = start of signal)
to_time	End time (0 = end of signal)
unit	Character: "hertz", "semitones", "mel", "erb", "loghertz"

Value

Named list with: min, max, mean, stdev, median, q25, q75, count_voiced

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- sound$to_pitch_cc()

# Direct call
stats <- get_pitch_stats_direct(pitch)

# Equivalent R6 calls, one boundary crossing per statistic:
min_val <- pitch$get_minimum(0, 0, "hertz")
max_val <- pitch$get_maximum(0, 0, "hertz")
```

get_pitch_stdev_direct

Get Pitch Standard Deviation Directly (Bypass R6)

Description

Get standard deviation of pitch values without R6 wrapper overhead.

Usage

```
get_pitch_stdev_direct(
  pitch,
  from_time = 0,
  to_time = 0,
  unit = c("hertz", "semitones", "mel", "erb", "loghertz")
)
```

Arguments

pitch	Pitch object or external pointer
from_time	Start time (0 = beginning)
to_time	End time (0 = end)
unit	Pitch unit

Value

Standard deviation

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch_ptr <- to_pitch_cc_direct(sound)
get_pitch_stdev_direct(pitch_ptr)
```

get_pitch_strengths_at_times

Batch Query Pitch Strengths at Multiple Times

Description

Query pitch strength (voicing confidence) at multiple time points.

Usage

```
get_pitch_strengths_at_times(pitch, times, unit = "hertz", interpolate = TRUE)
```

Arguments

<code>pitch</code>	A Pitch object
<code>times</code>	Numeric vector of time points (in seconds)
<code>unit</code>	Unit for pitch (used internally by Praat)
<code>interpolate</code>	Logical; whether to interpolate (default TRUE)

Value

Numeric vector of pitch strengths (0-1) at the specified times

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pitch <- sound$to_pitch()
times <- seq(0.05, 0.45, length.out = 10)
strengths <- get_pitch_strengths_at_times(pitch, times)
```

```
get_pitch_value_direct
```

Get Single Pitch Value Directly

Description

Get Single Pitch Value Directly

Usage

```
get_pitch_value_direct(pitch, time, unit = "hertz", interpolate = TRUE)
```

Arguments

<code>pitch</code>	Pitch object or external pointer
<code>time</code>	Time in seconds
<code>unit</code>	Unit string
<code>interpolate</code>	Whether to interpolate

Value

Pitch value

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
pitch_ptr <- to_pitch_cc_direct(sound)  
get_pitch_value_direct(pitch_ptr, 0.25)
```

`get_pointprocess_intervals`*Get Inter-Point Intervals from PointProcess*

Description

Compute the time intervals between consecutive points in a PointProcess. Useful for jitter analysis and prosody studies.

Usage

```
get_pointprocess_intervals(pointprocess)
```

Arguments

pointprocess A PointProcess object

Value

Numeric vector of intervals (in seconds). Length is 'n_points - 1'.

Performance

Computes all intervals in C++ without R<->C++ overhead.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
pp <- sound$to_point_process_periodic_cc(75, 600)  
intervals <- get_pointprocess_intervals(pp)  
jitter <- sd(intervals) / mean(intervals)
```

```
get_pointprocess_nearest_indices
```

Query PointProcess Nearest Indices at Multiple Times

Description

Find the nearest point index for each of multiple query times.

Usage

```
get_pointprocess_nearest_indices(pointprocess, times)
```

Arguments

pointprocess	A PointProcess object
times	Numeric vector of query times (in seconds)

Value

Integer vector of nearest point indices (1-based)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)
pp <- sound$to_point_process_periodic_cc(75, 600)
query_times <- seq(0.1, 0.4, by = 0.05)
indices <- get_pointprocess_nearest_indices(pp, query_times)
```

```
get_pointprocess_times
```

Get All Point Times from PointProcess

Description

Extract all point times from a PointProcess object as a numeric vector, in a single call instead of calling 'get_time(i)' in a loop.

Usage

```
get_pointprocess_times(pointprocess)
```

Arguments

pointprocess	A PointProcess object
--------------	-----------------------

Value

Numeric vector of all point times (in seconds)

Performance

Reduces R<->C++ calls from 'n' to 1.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
pp <- sound$to_point_process_periodic_cc(75, 600)  
times <- get_pointprocess_times(pp)
```

get_sampling_rate	<i>Get sampling rate of sound object (DEPRECATED)</i>
-------------------	---

Description

****DEPRECATED:**** Use sound\$get_sampling_frequency() instead.

Usage

```
get_sampling_rate(sound)
```

Arguments

sound	A Sound R6 object
-------	-------------------

Value

Sampling rate in Hz

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
get_sampling_rate(sound)
```

get_sd_intensity	<i>Get standard deviation of intensity (DEPRECATED)</i>
------------------	---

Description

****DEPRECATED:**** Use intensity\$get_standard_deviation() instead.

Usage

```
get_sd_intensity(intensity, time_range = NULL)
```

Arguments

intensity	An Intensity R6 object
time_range	Optional time range c(start, end)

Value

Standard deviation in dB

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
intensity <- sound$to_intensity()
suppressWarnings(get_sd_intensity(intensity))
```

get_sound_times_fast	<i>Get Sound Sample Times (Fast Computation)</i>
----------------------	--

Description

Returns time values for each sample using a direct computation, instead of 'sound\$get_sample_times()'. Still allocates memory for the result.

Usage

```
get_sound_times_fast(sound)
```

Arguments

sound	A Sound object
-------	----------------

Value

Numeric vector of sample times (in seconds)

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
times <- get_sound_times_fast(sound)
```

get_sound_values_fast *Get Sound Values (Fast Copy)*

Description

Copies Sound sample data via direct pointer access, instead of going through Praat's per-sample accessor as 'sound\$get_values()' does.

Usage

```
get_sound_values_fast(sound, channel = 1)

# Deprecated: use get_sound_values_fast() instead
```

Arguments

sound	A Sound object created with 'Sound()'
channel	Channel number (1-based, default 1)

Value

Numeric vector (independent copy of sample data). Has class 'c("fast_vector", "numeric")' and a 'readonly' attribute for backward compatibility.

The returned vector is an independent R copy — safe to modify, store, or use after the Sound object is garbage collected.

See Also

- [is_fast_vector()] to check if vector was created by fast access - [sound_as_matrix_fast()] for matrix output

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)

# Fast copy for analysis
samples <- get_sound_values_fast(sound, channel = 1)
rms <- sqrt(mean(samples^2))
peak <- max(abs(samples))

# Equivalent - also a copy
```

```
samples2 <- sound$get_values(channel = 1)
```

```
get_spectral_moments_batch
```

Get spectral moments for all frames of a Spectrogram

Description

Computes centre of gravity, standard deviation, skewness, and kurtosis for every frame in a Spectrogram in a single C++ pass, instead of calling the per-frame Spectrum methods in an R loop.

Usage

```
get_spectral_moments_batch(spectrogram, power = 2)
```

Arguments

spectrogram	A Spectrogram object
power	Numeric. Power for moment weighting (default 2.0, matching Praat)

Value

data.frame with columns time, cog, sd, skewness, kurtosis (one row per frame; NA where undefined)

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
spectrogram <- sound$to_spectrogram()
moments <- get_spectral_moments_batch(spectrogram)
head(moments)
```

```
get_textgrid_interval_stats
```

Get Interval Statistics for All Intervals (Batch)

Description

Compute statistics (start, end, duration, label) for all intervals in a tier in a single call, instead of a manual loop. Returns a data frame ready for analysis.

Usage

```
get_textgrid_interval_stats(textgrid, tier)
```


Arguments

textgrid	A TextGrid object
tier	Tier number (1-based) or tier name

Value

Data frame with columns: - 'index': Interval index (1-based) - 'label': Interval label - 'start': Start time (seconds) - 'end': End time (seconds) - 'duration': Duration (seconds)

Examples

```
tg <- TextGrid$create(0, 1, "words")
tg$insert_boundary(1, 0.5)
tg$set_interval_text(1, 2, "hello")
stats <- get_textgrid_interval_stats(tg, tier = 1)

# Analysis with base R
voiced <- stats[stats$label == "hello", ]
mean(voiced$duration)
```

`get_textgrid_labels_all`*Get All Labels from TextGrid Tier (Batch)*

Description

Extract all interval labels from a tier in a single operation, instead of calling 'textgrid\$get_interval_text()' repeatedly.

Usage

```
get_textgrid_labels_all(textgrid, tier)
```

Arguments

textgrid	A TextGrid object
tier	Tier number (1-based) or tier name

Value

Character vector of all interval labels

Examples

```
tg <- TextGrid$create(0, 1, "words")
tg$insert_boundary(1, 0.5)
tg$set_interval_text(1, 2, "hello")
labels <- get_textgrid_labels_all(tg, tier = 1)
table(labels) # Frequency table of labels
```

```
get_voice_quality_ultra
```

Get Voice Quality Metrics in Single Call (Tier 4 Ultra)

Description

Complete voice quality pipeline in C++: Sound -> Pitch -> PointProcess -> Jitter/Shimmer/HNR metrics, in one call instead of the equivalent Tier 2/3 workflow using separate function calls.

This is a **Tier 4 "Ultra"** function for batch DSI (Dysphonia Severity Index) calculations where jitter PPQ5 is the PPQ component.

By default, this keeps the existing Tier 4 pitch path ('pitch_method = "cc", 'very_accurate = TRUE'). If your reference workflow uses Praat's plain 'To Pitch...' command before 'To Point-Process (cc)' (for example the DSI jitter block), call this with 'pitch_method = "ac", very_accurate = FALSE' — or the equivalent shorthand 'pitch_method = "periodic_cc"' (see below).

Usage

```
get_voice_quality_ultra(
  sound,
  metrics = "all",
  min_pitch = 75,
  max_pitch = 600,
  time_step = 0,
  pitch_method = c("cc", "ac", "periodic_cc"),
  very_accurate = TRUE
)
```

Arguments

sound	A Sound object
metrics	Character vector of metrics to compute: "jitter", "shimmer", "hnr", or "all" for all metrics
min_pitch	Pitch floor in Hz for pitch extraction (default: 75). Note: HNR always uses 75 Hz as minimum pitch (Praat's CC harmonicity default), independent of this parameter.
max_pitch	Pitch ceiling in Hz (default: 600)
time_step	Time step for pitch/HNR (0 = auto; HNR auto uses 0.01 s)

<code>pitch_method</code>	Pitch algorithm for the jitter/shimmer pitch object: <code>"cc"</code> (default, preserves existing Tier 4 behaviour), <code>"ac"</code> , or <code>"periodic_cc"</code> — an alias for <code>"ac"</code> with <code>'very_accurate'</code> forced to <code>'FALSE'</code> , matching Praat's <code>'To PointProcess (periodic, cc)...</code> command (see Algorithm choice section below).
<code>very_accurate</code>	Logical; whether to use Praat's very accurate pitch path for jitter/shimmer pitch extraction (default: <code>'TRUE'</code> to preserve the existing Tier 4 output). Use <code>'FALSE'</code> with <code>'pitch_method = "ac"</code> to match Praat's plain <code>'To Pitch...</code> command. Ignored (forced <code>'FALSE'</code>) when <code>'pitch_method = "periodic_cc"</code> .

Value

Named list with requested voice quality metrics:

jitter_local Local jitter (relative, fraction)
jitter_local_abs Local absolute jitter (seconds)
jitter_rap Relative average perturbation
jitter_ppq5 5-point period perturbation quotient (DSI PPQ component)
jitter_ddp Difference of differences of periods
shimmer_local Local shimmer (relative, fraction)
shimmer_local_db Local shimmer (dB)
shimmer_apq3 3-point amplitude perturbation quotient
shimmer_apq5 5-point amplitude perturbation quotient
shimmer_apq11 11-point amplitude perturbation quotient
shimmer_dda Difference of differences of amplitudes
hnr_mean Mean harmonics-to-noise ratio (dB)
hnr_sd Standard deviation of HNR

Algorithm choice — `pitch_method = "periodic_cc"`

Praat's `'Sound: To PointProcess (periodic, cc)...` command is, per Praat's own source (`'Sound_to_PointProcess.cpp'`) and manual, exactly `'Sound_to_Pitch(sound, 0, floor, ceiling)'` (i.e. `'Sound_to_Pitch_rawAc'` with `'veryAccurate = FALSE'` and Praat's raw defaults `'0.03, 0.45, 0.01, 0.35, 0.14'`) followed by `'Sound_Pitch_to_PointProcess_cc'`. That is byte-for-byte what this function already computes when called with `'pitch_method = "ac", very_accurate = FALSE'`. `'pitch_method = "periodic_cc"` is a pure alias for that combination (it forces `'very_accurate = FALSE'` regardless of the `'very_accurate'` argument), so callers porting a Praat script that uses `'To PointProcess (periodic, cc)...` can request the matching Tier 4 path by name instead of having to know the two are equivalent.

API Tier

This is a `***Tier 4 "Ultra"***` function. The entire workflow (pitch extraction, `PointProcess` creation, and all voice quality calculations) happens in C++ with no intermediate R objects.

See Also

[`Sound`] for creating `Sound` objects [`get_jitter_shimmer_batch()`] for Tier 2/3 voice quality analysis

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =
  16000)

# Get all voice quality metrics
vq <- get_voice_quality_ultra(sound, metrics = "all", min_pitch = 75)

# Get only jitter metrics (DSI PPQ component)
vq <- get_voice_quality_ultra(sound, metrics = "jitter")
ppq5 <- vq$jitter_ppq5

# Match a plain pitch extraction + PointProcess (cc) DSI path
vq_ac <- get_voice_quality_ultra(
  sound,
  metrics = "jitter",
  pitch_method = "ac",
  very_accurate = FALSE
)

# Equivalent shorthand for the periodic-cc PointProcess path
vq_periodic_cc <- get_voice_quality_ultra(
  sound,
  metrics = "jitter",
  pitch_method = "periodic_cc"
)

```

Harmonicity

Harmonicity

Description

A Harmonicity object represents the degree of acoustic periodicity, the Harmonics-to-Noise Ratio (HNR), in a sound over time, measured in decibels.

Arguments

<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ Harmonicity object; set internally when a method returns a new Harmonicity.
--------------------	--

Details

Created from a Sound via cross-correlation (CC) or glottal-to-noise excitation (GNE) analysis. Higher values indicate more periodic (voiced) speech.

Value

A Harmonicity object with methods for harmonics-to-noise ratio (HNR) analysis.

Information methods

- `get_start_time()`, `get_end_time()` - time range (s)
- `get_sampling_period()` - time step between frames (s)
- `get_number_of_frames()` - number of analysis frames
- `get_time_from_frame(frame)` - time for a frame index
- `get_frame_from_time(time)` - frame index for a time

Point query methods

- `get_value_at_time(time, interpolation)` - HNR at a time point (dB)
- `get_values_at_times(times, interpolation)` - HNR at a vector of times (batch)
- `get_values_vector()` - raw HNR values for all frames
- `get_times_vector()` - frame time points

Statistics methods (over a time range)

- `get_mean(from_time, to_time)` - mean HNR (dB)
- `get_minimum(from_time, to_time, interpolation)` - minimum HNR
- `get_maximum(from_time, to_time, interpolation)` - maximum HNR
- `get_standard_deviation(from_time, to_time)` - standard deviation
- `get_time_of_minimum(...)`, `get_time_of_maximum(...)` - time of extremum

Batch methods

- `get_statistics_batch(from_times, to_times)` - statistics for multiple intervals in a single C++ call

Export methods

- `as_data_frame()`, `as_matrix()` - export as a data.frame or matrix

See Also

[Sound](#), [Pitch](#), [PointProcess](#)

Examples

```
sound <- Sound$create_tone(duration = 1.0, frequency = 200, sampling_rate =
  44100)
hnr <- sound$to_harmonicity_cc(time_step = 0.01, min_pitch = 75)
mean_hnr <- hnr$get_mean()
hnr_at_05 <- hnr$get_value_at_time(0.5)

# The same analysis on a recording read from disk
sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))
hnr <- sound$to_harmonicity_ac(time_step = 0.01, min_pitch = 75)
df <- hnr$as_data_frame()
```

Intensity

Intensity

Description

Intensity objects represent sound power (loudness) over time, measured in decibels (dB) relative to the auditory threshold.

Arguments

`.xptr` Not for direct use. External pointer to the underlying C++ Intensity object; set internally when a method returns a new Intensity.

Details

Created from a Sound using intensity contour extraction.

Value

An Intensity object with methods for querying intensity values (in dB) at time points or across the full contour.

Information methods

- `get_duration()` - duration of the intensity contour (s)
- `get_time_step()` - time step between frames (s)

Point query methods

- `get_value_at_time(time, interpolation)` - intensity at a time point (dB)
- `get_values_at_times(times, interpolation)` - intensity at a vector of times (batch)

Statistics methods (over a time range)

- `get_mean(from_time, to_time, averaging_method)` - mean intensity (dB)
- `get_standard_deviation(from_time, to_time)` - standard deviation (dB)
- `get_minimum(from_time, to_time, interpolation)` - minimum intensity
- `get_maximum(from_time, to_time, interpolation)` - maximum intensity
- `get_quantile(quantile, from_time, to_time)` - quantile
- `get_time_of_minimum(...)`, `get_time_of_maximum(...)` - time of extremum

Export methods

- `as_vector()` - raw intensity values (dB)
- `as_data_frame()` - export as a `data.frame` (time, intensity)
- `save(filepath)` - save to a Praat binary file

Interpolation

Codes: "nearest" (0), "linear" (1), "cubic" (2, default), "sinc70" (3), "sinc700" (4). Averaging methods: "energy" (0, default), "sones" (1), "db" (2).

See Also

[Sound](#), [Pitch](#), [IntensityTier](#)

Examples

```
sound <- Sound$create_tone(duration = 1.0, frequency = 200, sampling_rate =
  44100)
intensity <- sound$to_intensity(minimum_pitch = 100, time_step = 0.0)
mean_int <- intensity$get_mean()
df <- intensity$as_data_frame()

# The same analysis on a recording read from disk
sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))
intensity <- sound$to_intensity(minimum_pitch = 100, time_step = 0.0)
int_at_02s <- intensity$get_value_at_time(0.2)
```

intensity_tier_to_amplitude_tier

Convert IntensityTier to AmplitudeTier

Description

Converts intensity values (dB) to amplitude values.

Usage

```
intensity_tier_to_amplitude_tier(intensity_tier)
```

Arguments

`intensity_tier` An IntensityTier object

Value

An AmplitudeTier object

Examples

```
it <- IntensityTier(0, 1)
it$add_point(0.25, 70)
it$add_point(0.75, 60)
at <- intensity_tier_to_amplitude_tier(it)
at$get_number_of_points()
```

IntensityTier	<i>Praat IntensityTier Object</i>
---------------	-----------------------------------

Description

Praat IntensityTier object with direct C++ module binding for intensity manipulation.

Details

IntensityTiers contain discrete time-value pairs representing intensity in dB SPL. They can be used to modify the amplitude envelope of sounds.

Value

An IntensityTier object with methods for intensity (dB SPL) manipulation via time-value points.

Examples

```
it <- IntensityTier(0, 1)
it$add_point(0.25, 70)
it$add_point(0.75, 60)
it$get_number_of_points()
it$get_value_at_time(0.5)
```

is_fast_access	<i>Check if Vector is a Fast-Access Vector</i>
----------------	--

Description

Check if Vector is a Fast-Access Vector

Usage

```
is_fast_access(x)
```

Arguments

x	Numeric vector
---	----------------

Value

TRUE if x has the fast_vector or zerocopy_vector class

Examples

```
is_fast_access(1:10)
sound <- Sound$create_tone(frequency = 220, duration = 0.1, sampling_rate = 8000)
is_fast_access(get_sound_values_fast(sound))
```

is_fast_vector	<i>Check if Vector is a Fast-Access Vector</i>
----------------	--

Description

Tests whether a numeric vector was created by [get_sound_values_fast()].

Usage

```
is_fast_vector(x)

# Deprecated: use is_fast_vector() instead
```

Arguments

x	A vector to test
---	------------------

Value

Logical. TRUE if vector has fast_vector/zerocopy_vector class.

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)

fast_vec <- get_sound_values_fast(sound, 1)
regular_vec <- sound$get_values(1)

is_fast_vector(fast_vec)      # TRUE
is_fast_vector(regular_vec)  # FALSE
```

is_praat_formant	<i>Check if object is a valid Formant (R6 or legacy S3)</i>
------------------	---

Description

Validates that an object is a Formant R6 object or legacy praat_formant

Usage

```
is_praat_formant(x)
```

Arguments

x	Object to check
---	-----------------

Value

Logical indicating validity

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
formant <- sound$to_formant_burg()  
is_praat_formant(formant)  
is_praat_formant(42)
```

is_praat_intensity	<i>Check if object is a valid Intensity (R6 or legacy S3)</i>
--------------------	---

Description

Validates that an object is an Intensity R6 object or legacy praat_intensity

Usage

```
is_praat_intensity(x)
```

Arguments

x	Object to check
---	-----------------

Value

Logical indicating validity

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =  
  16000)  
intensity <- sound$to_intensity()  
is_praat_intensity(intensity)  
is_praat_intensity(42)
```

is_praat_pitch	<i>Check if object is a valid Pitch (R6 or legacy S3)</i>
----------------	---

Description

Validates that an object is a Pitch R6 object or legacy praat_pitch

Usage

```
is_praat_pitch(x)
```

Arguments

x	Object to check
---	-----------------

Value

Logical indicating validity

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5, sampling_rate =  
  16000)  
pitch <- sound$to_pitch()  
is_praat_pitch(pitch)  
is_praat_pitch(42)
```

is_praat_sound	<i>Check if object is a valid Sound (R6 or legacy S3)</i>
----------------	---

Description

Validates that an object is a Sound R6 object or legacy praat_sound

Usage

```
is_praat_sound(x)
```

Arguments

x	Object to check
---	-----------------

Value

Logical indicating validity

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
is_praat_sound(sound)
is_praat_sound(42)

```

KlattGrid

*Create a KlattGrid object***Description**

A KlattGrid is a speech synthesizer based on the Klatt formant synthesizer. It allows detailed control over phonation, vocal tract resonances, frication, and other articulatory parameters.

Arguments

```

numberOfFormants
    Number of oral formants (typically 6)
numberOfNasalFormants
    Number of nasal formants (typically 1)
numberOfNasalAntiFormants
    Number of nasal antiformants (typically 1)
numberOfTrachealFormants
    Number of tracheal formants (typically 1)
numberOfTrachealAntiFormants
    Number of tracheal antiformants (typically 1)
numberOfFricationFormants
    Number of frication formants (typically 6)
numberOfDeltaFormants
    Number of delta formants (typically 1)

```

Value

KlattGrid object with S3 class

Examples

```

# Create empty KlattGrid
kg <- KlattGrid(0, 1, numberOfFormants = 6)

# Set pitch contour and voicing amplitude (both required for synthesis)
kg$add_pitch_point(0.5, 100) # 100 Hz at 0.5s
kg$add_voicing_amplitude_point(0.5, 90)

# Set oral formant frequencies (formantType 1 = ORAL)
kg$add_formant_point(1, 1, 0.5, 500) # F1 = 500 Hz
kg$add_formant_point(1, 2, 0.5, 1500) # F2 = 1500 Hz

```

```
# Synthesize  
sound <- kg$to_sound()
```

```
klattgrid_create_example  
Create example KlattGrid
```

Description

Creates a demonstration KlattGrid with pre-configured parameters for testing the synthesizer.

Usage

```
klattgrid_create_example()  
  
# Deprecated: use klattgrid_create_example() instead
```

Value

KlattGrid example object

Examples

```
kg <- klattgrid_create_example()  
sound <- kg$to_sound()
```

```
klattgrid_create_from_vowel  
Create KlattGrid from vowel parameters
```

Description

Creates a KlattGrid pre-configured for synthesizing a vowel sound with specified formant frequencies and bandwidths.

Usage

```
klattgrid_create_from_vowel(
  duration = 0.5,
  f0start = 100,
  f1 = 500,
  b1 = 50,
  f2 = 1500,
  b2 = 100,
  f3 = 2500,
  b3 = 150,
  f4 = 3500,
  bandwidthFraction = 0.05,
  formantFrequencyInterval = 1000
)

# Deprecated: use klattgrid_create_from_vowel() instead
```

Arguments

<code>duration</code>	Duration in seconds
<code>f0start</code>	Starting F0 in Hz
<code>f1</code>	First formant frequency in Hz
<code>b1</code>	First formant bandwidth in Hz
<code>f2</code>	Second formant frequency in Hz
<code>b2</code>	Second formant bandwidth in Hz
<code>f3</code>	Third formant frequency in Hz
<code>b3</code>	Third formant bandwidth in Hz
<code>f4</code>	Fourth formant frequency in Hz (optional)
<code>bandwidthFraction</code>	Bandwidth as fraction of frequency (default 0.05)
<code>formantFrequencyInterval</code>	Formant spacing interval in Hz (default 1000)

Value

KlattGrid object configured for vowel

Examples

```
kg <- klattgrid_create_from_vowel(duration = 0.3, f0start = 120)
sound <- kg$to_sound()
```

LFCC

*LFCC***Description**

Linear-frequency cepstral coefficients for speaker recognition.

Arguments

.xptr Not for direct use. External pointer to the underlying C++ LFCC object; set internally when a method returns a new LFCC.

Details

LFCCs are similar to MFCCs but use a linear frequency scale instead of the mel scale. They are derived from LPC analysis and are useful for speaker recognition tasks. Uses a shared dispatch table for minimal memory per object.

Value

An LFCC object with methods for linear-frequency cepstral coefficient analysis.

See Also

[Sound](#), [LPC](#)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
lpc <- sound$to_lpc_burg(prediction_order = 16)
lfcc <- lpc$to_lfcc(num_coefficients = 12)
coefs <- lfcc$get_all_coefficients()
lpc2 <- lfcc$to_lpc(num_coefficients = 16)
```

LongSound

*LongSound***Description**

Represents a Praat LongSound object: an audio file that stays on disk instead of loading into memory.

Arguments

.xptr Not for direct use. External pointer to the underlying C++ LongSound object; set internally when a method returns a new LongSound.

Details

A regular Sound loads the whole recording into RAM, which can exceed available memory on a multi-hour field recording, or just crowd out everything else you're working on. LongSound opens the file, reads its header, and streams samples on demand, so you can check duration, inspect amplitude, or pull out short clips from a file far bigger than memory allows. Use `extract_part()` whenever you need actual waveform data for pitch, formant, or intensity analysis: it reads just the requested time window and hands back a normal Sound.

Value

A LongSound object.

Usage

```
ls <- LongSound$open("recording.wav")
part <- ls$extract_part(10, 15) # 5 seconds as a Sound
```

Query methods

- `get_duration()` - total duration in seconds
- `get_start_time()`, `get_end_time()` - time domain in seconds
- `get_sample_rate()` - sampling frequency in Hz
- `get_number_of_samples()` - number of samples per channel
- `get_number_of_channels()` - number of channels (1 = mono, 2 = stereo)
- `get_file_path()` - path to the underlying audio file
- `get_dx()` - sampling period in seconds ($1 / \text{get_sample_rate}()$)
- `get_x1()` - time of the first sample

Time/sample conversion

- `get_time_from_sample(sample)` - time, in seconds, of a given sample index
- `get_sample_from_time(time)` - index of the sample nearest a given time

Streaming

- `extract_part(from, to, preserve_times = FALSE)` - read a time range from disk and return it as a Sound. Set `preserve_times = TRUE` to keep the extracted Sound's time domain aligned with the original file instead of starting it at time 0.
- `have_window(from, to)` - TRUE if that time range is already held in the internal read buffer, so the next `extract_part()` over it will be fast
- `get_window_extrema(from, to, channel = 1)` - minimum and maximum amplitude in a time range, without building a Sound

Save

- `save_part(from, to, path, format = "wav")` - write a time range straight to an audio file, without holding the whole clip in memory. format is one of "wav", "aiff", "aifc", "flac", or "wav24" (24-bit WAV).
- `save_channel(channel, path, format = "wav")` - write a single channel to an audio file

Utility

- `is_valid()` - FALSE if the underlying file handle was closed or garbage collected
- `print()` - summary of file path, duration, sample rate, and channel count

Streaming through a very large file repeatedly touches the same disk regions; see [longsound_get_buffer_size_pref_seconds](#) to tune how much Praat keeps cached between reads.

See Also

[Sound](#)

Examples

```
# LongSound streams from disk, so the example writes a small WAV file first
tmp <- tempfile(fileext = ".wav")
Sound$create_tone(frequency = 150, duration = 1.0)$save(tmp)

ls <- LongSound$open(tmp)
print(ls)

cat("Duration:", ls$get_duration(), "s\n")
cat("Sample rate:", ls$get_sample_rate(), "Hz\n")

# Extract first 0.5 seconds as a Sound
sound <- ls$extract_part(0, 0.5)

# Convert a sample index to a time and back
t <- ls$get_time_from_sample(100)
ls$get_sample_from_time(t)

unlink(tmp)
```

Description

Controls how much audio Praat keeps resident in memory while streaming from a LongSound. This is a global setting: it applies to every LongSound opened after it is changed, not to one specific object. The default is 60 seconds. Raise it to reduce disk re-reads when repeatedly querying nearby windows of a very large file; lower it to shrink the package's memory footprint when working with many LongSound objects at once.

Usage

```
longsound_get_buffer_size_pref_seconds()
```

Value

Buffer size in seconds.

Examples

```
longsound_get_buffer_size_pref_seconds()
```

longsound_open	<i>Open a LongSound from file</i>
----------------	-----------------------------------

Description

Open a LongSound from file

Usage

```
longsound_open(path)
```

Arguments

path	Path to audio file (WAV, AIFF, FLAC, MP3, etc.)
------	---

Value

LongSound object

Examples

```
wav <- tempfile(fileext = ".wav")
Sound$create_tone(frequency = 220, duration = 1, sampling_rate =
  16000)$save(wav)

ls <- longsound_open(wav)
print(ls)

# Extract portion
sound <- ls$extract_part(0, 0.5)
```

longsound_set_buffer_size_pref_seconds

Set the LongSound streaming buffer size preference

Description

Set the LongSound streaming buffer size preference

Usage

```
longsound_set_buffer_size_pref_seconds(seconds)
```

Arguments

seconds	Buffer size in seconds. See longsound_get_buffer_size_pref_seconds for what this controls.
---------	--

Value

Invisibly, the previous buffer size in seconds.

Examples

```
old <- longsound_set_buffer_size_pref_seconds(120)
longsound_get_buffer_size_pref_seconds()
longsound_set_buffer_size_pref_seconds(old)
```

LPC

LPC

Description

Praat LPC object for linear predictive coding analysis, created via direct C++ module binding.

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ LPC object; set internally when a method returns a new LPC.
-------	--

Details

LPC (Linear Predictive Coding) estimates the spectral envelope of a sound by modeling it as an autoregressive process. The LPC coefficients describe the vocal tract filter and can be converted to formants, spectra, or other representations.

Value

An LPC object with methods for linear predictive coding analysis and inverse filtering.

Creating LPC objects

LPC objects are created from Sound objects using one of several methods:

- `sound$to_lpc_burg()` - Burg method (fastest, most robust)
- `sound$to_lpc_auto()` - autocorrelation method
- `sound$to_lpc_covariance()` - covariance method
- `sound$to_lpc_marple()` - Marple method (slowest, most accurate)

Query methods

- `get_number_of_frames()` - number of analysis frames
- `get_time_step()` - time step between frames
- `get_sampling_period()` - sampling period of the original sound
- `get_max_num_coefficients()` - maximum number of LPC coefficients
- `get_gain_at_frame(frame)` - gain value for a specific frame
- `get_coefficients_at_frame(frame)` - LPC coefficients for a specific frame
- `get_all_gains()` - vector of all gain values
- `get_all_coefficients()` - matrix of all LPC coefficients

Conversion methods

- `to_formant(margin)` - not available in this build (requires CLAPACK); use `Sound$to_formant_burg()` for formant extraction instead
- `to_spectrum(time, ...)` - convert to a Spectrum at a specific time
- `to_matrix()` - convert to a Matrix object

Voice source extraction (inverse filtering)

- `filter_inverse(sound)` - extract glottal flow by inverse filtering
- `filter_inverse_at_time(sound, time, channel)` - use the filter from a specific time

These methods remove vocal tract resonances to reveal the voice source (glottal flow waveform), useful for voice quality research and vocal fold dynamics.

See Also

[Sound](#), [Formant](#), [Spectrum](#), [LFCC](#)

Examples

```
# Load sound
sound <- Sound$create_tone(frequency = 150, duration = 0.3)

# Compute LPC (Burg method is recommended)
lpc <- sound$to_lpc_burg(
  prediction_order = 16,
  analysis_width = 0.025,
  time_step = 0.005,
  pre_emphasis_frequency = 50.0
)

# Query properties
n_frames <- lpc$get_number_of_frames()
gains <- lpc$get_all_gains()
coeffs <- lpc$get_all_coefficients()

# Get coefficients for a specific frame
coef_frame1 <- lpc$get_coefficients_at_frame(1)

# Convert to other representations
spectrum <- lpc$to_spectrum(time = 0.15, df_min = 20)

# Extract voice source (glottal flow) via inverse filtering at a given time
midpoint <- sound$get_duration() / 2
glottal_flow <- lpc$filter_inverse_at_time(sound, time = midpoint)
```

Ltas

Ltas

Description

Represents a long-term average spectrum (LTAS): the frequency spectrum of a sound averaged over its whole duration.

Arguments

<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ object; set internally when a method returns a new Ltas.
--------------------	---

Details

An Ltas collapses a recording into a single spectral shape: how much energy sits in each frequency band once the moment-to-moment detail is averaged out. It's the standard tool for describing overall spectral shape (tilt, resonance pattern, noise floor), useful for comparing voice quality across speakers or checking a recording's frequency balance.

Value

An Ltas object with methods for long-term average spectrum analysis.

Usage

```
ltas <- sound$to_ltas(bandwidth = 100)
ltas <- sound$to_ltas_pitch_corrected(pitch_floor = 75, pitch_ceiling = 600)
```

Units and interpolation

Power values default to decibels (`unit = "dB"`); pass `"energy"` or `"sones"` for Praat's other two averaging units. Methods that locate a peak or trough take an interpolation of `"none"`, `"parabolic"` (the default), `"cubic"`, `"sinc70"`, or `"sinc700"` - parabolic and cubic refine the peak's position between bins instead of snapping to the nearest one. `get_value_at_frequency()` takes a plain `interpolate` flag (`TRUE/FALSE`) rather than a named method.

Frequency domain

- `get_lowest_frequency()`, `get_highest_frequency()` - the frequency range, in Hz
- `get_frequency_range()` - the width of that range, in Hz
- `get_number_of_bins()` - number of frequency bins
- `get_bin_width()` - width of each bin, in Hz
- `get_frequency_from_bin(bin)` - centre frequency of a bin
- `get_bin_from_frequency(frequency)` - the bin nearest a frequency

Query values

- `get_value_at_frequency(frequency, unit, interpolate)` - power at a given frequency
- `get_value_in_bin(bin)` - power in one bin, read directly with no interpolation
- `get_minimum(fmin, fmax, unit, interpolation)`, `get_maximum(fmin, fmax, unit, interpolation)` - lowest/highest power in a frequency range
- `get_frequency_of_minimum(fmin, fmax, interpolation)`, `get_frequency_of_maximum(fmin, fmax, interpolation)` - where those extremes occur
- `get_mean(fmin, fmax, unit)` - average power across a frequency range
- `get_standard_deviation(fmin, fmax, unit)` - how much the power varies across a frequency range
- `get_slope(f1min, f1max, f2min, f2max, unit)` - the difference in average power between two bands, e.g. spectral tilt (high-frequency band minus low-frequency band)
- `get_local_peak_height(environment_min, environment_max, peak_min, peak_max, unit)` - how far a peak rises above its surrounding band, e.g. harmonic-to-noise prominence

Batch queries

Faster than calling the single-value methods in a loop, since each call crosses into C++ once for the whole set of frequencies or ranges.

- `get_values_at_frequencies(frequencies, interpolation)` - values at many frequencies at once
- `get_means_batch(fmins, fmaxs, unit)` - means for many frequency ranges at once
- `get_peaks_batch(fmins, fmaxs, interpolation), get_minima_batch(fmins, fmaxs, interpolation)` - peaks or troughs for many ranges at once, returned as a data frame

Trend and peaks

- `compute_trend_line(fmin, fmax)` - the fitted trend line, as its own Ltas
- `subtract_trend_line(fmin, fmax)` - the spectrum with that trend removed, e.g. to flatten spectral tilt before comparing bands
- `report_spectral_trend(fmin, fmax, frequency_scale, fit_method)` - slope, intercept, and fit quality (R-squared, residual error) as a structured result; print it directly or read `$fitted_values` for the per-point predictions
- `get_spectral_slope(fmin, fmax, frequency_scale, fit_method)` - just the slope from the above, when that's all you need
- `to_spectrum_tier_peaks()` - the local maxima across the whole spectrum, as a [SpectrumTier](#)

Export

- `as_data_frame()` - frequency and power as a data frame
- `as_matrix()` - frequency and power as a plain matrix
- `to_matrix()` - the Ltas converted to a full [Matrix](#) object
- `save(path)` - write to a Praat text file

See Also

[Sound](#), [Spectrum](#), [Spectrogram](#), [SpectrumTier](#)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
ltas <- sound$to_ltas(bandwidth = 100)
ltas$get_mean()
ltas$get_number_of_bins()
```

ltas_average	<i>Average multiple Ltas objects</i>
--------------	--------------------------------------

Description

Average multiple Ltas objects

Usage

```
ltas_average(...)
```

Arguments

... Ltas objects to average

Value

A new Ltas object representing the average

Examples

```
s1 <- Sound$create_tone(frequency = 200, duration = 0.5, sampling_rate =
16000)
s2 <- Sound$create_tone(frequency = 240, duration = 0.5, sampling_rate =
16000)
avg <- ltas_average(s1$to_ltas(), s2$to_ltas())
```

Manipulation	<i>Praat Manipulation Object</i>
--------------	----------------------------------

Description

Praat Manipulation object with direct C++ module binding for PSOLA-based pitch and duration modification.

Arguments

.xptr Not for direct use. External pointer to the underlying C++ Manipulation object; set internally when a method returns a new Manipulation, e.g. `sound$to_manipulation()`.

Details

The Manipulation object is Praat's main tool for modifying pitch and duration of speech sounds using PSOLA (Pitch-Synchronous Overlap-Add) resynthesis.

Value

A Manipulation object.

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
manip <- sound$to_manipulation(pitch_floor = 75, pitch_ceiling = 300)
manip$has_pitch_tier()
pt <- manip$extract_pitch_tier()
```

Matrix	<i>Praat Matrix Object</i>
--------	----------------------------

Description

Praat Matrix object with direct C++ module binding for matrix operations.

Details

Matrix objects represent two-dimensional sampled data with x and y axes.

Value

A Matrix object with methods for two-dimensional sampled data access.

Examples

```
m <- Matrix(numberOfRows = 3, numberOfColumns = 4)
m$set_value(1, 1, 5.0)
m$get_value(1, 1)
m$get_number_of_rows()
as.matrix(m)

# Save and read back
mat_file <- tempfile(fileext = ".Matrix")
m$save(mat_file)
m2 <- matrix_read(mat_file)
m2$get_value(1, 1)
```

`matrix_create`*Create a Praat Matrix with full parameters*

Description

Creates a new Matrix object with explicit domain and sampling parameters.

Usage

```
matrix_create(xmin, xmax, nx, dx, x1, ymin, ymax, ny, dy, y1)
```

Arguments

<code>xmin</code>	Minimum x value (start of x domain)
<code>xmax</code>	Maximum x value (end of x domain)
<code>nx</code>	Number of columns
<code>dx</code>	X sampling period (step between columns)
<code>x1</code>	X value of first column
<code>ymin</code>	Minimum y value (start of y domain)
<code>ymax</code>	Maximum y value (end of y domain)
<code>ny</code>	Number of rows
<code>dy</code>	Y sampling period (step between rows)
<code>y1</code>	Y value of first row

Value

A Matrix object

See Also

[matrix_create_simple](#) for simpler creation, [Matrix](#) for object methods

Examples

```
m <- matrix_create(xmin = 0, xmax = 2, nx = 2, dx = 1, x1 = 0.5,
                  ymin = 0, ymax = 1, ny = 1, dy = 1, y1 = 0.5)
m$get_number_of_rows()
m$get_number_of_columns()
```

matrix_create_simple	Create a simple Praat Matrix
----------------------	------------------------------

Description

Creates a new Matrix object with given dimensions. Domain defaults to [0,1] for both axes.

Usage

```
matrix_create_simple(numberOfRows, numberOfColumns)
```

Arguments

numberOfRows	Number of rows
numberOfColumns	Number of columns

Value

A Matrix object

See Also

[matrix_create](#) for full parameter control, [Matrix](#) for object methods

Examples

```
m <- matrix_create_simple(3, 4)
m$set_value(1, 1, 5.0)
m$get_value(1, 1)
```

matrix_read	Read a Matrix from file
-------------	-------------------------

Description

Reads a Praat Matrix object from a text or binary file.

Usage

```
matrix_read(path)
```

Arguments

path	Path to the Matrix file
------	-------------------------

Value

A Matrix object

See Also

[matrix_create](#), [matrix_create_simple](#)

Examples

```
mat_file <- tempfile(fileext = ".Matrix")
m <- matrix_create_simple(1, 2)
m$set_value(1, 1, 1)
m$set_value(1, 2, 2)
m$save(mat_file)

m2 <- matrix_read(mat_file)
m2$get_number_of_rows()
m2$get_value(1, 1)
```

MelSpectrogram

MelSpectrogram

Description

A mel-scale spectrogram, commonly used in speech technology (ASR, speaker identification).

Usage

```
MelSpectrogram(.xptr = NULL)
```

Arguments

<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ MelSpectrogram object; set internally by <code>Sound\$to_mel_spectrogram()</code> .
--------------------	--

Value

A MelSpectrogram object.

Examples

```
snd <- Sound$create_tone(frequency = 150, duration = 0.3)
mel <- snd$to_mel_spectrogram()

# Convert to MFCC
mfcc <- mel$to_mfcc(number_of_coefficients = 12)

# Export as matrix
mat <- mel$as_matrix()
```

MFCC

*MFCC***Description**

Mel-frequency cepstral coefficients for speech and speaker recognition.

Arguments

`.xptr` Not for direct use. External pointer to the underlying C++ MFCC object; set internally when a method returns a new MFCC.

Details

MFCCs are widely used features in speech and speaker recognition systems. They represent the short-term power spectrum of a sound on a mel scale, which approximates human auditory perception. Uses a shared dispatch table for minimal memory per object.

Value

An MFCC object with methods for Mel-frequency cepstral coefficient analysis.

Creating MFCC objects

- `sound$to_mfcc()` - extract MFCCs with default parameters

Query methods

- `get_number_of_frames()` - number of analysis frames
- `get_time_step()` - time step between frames
- `get_max_num_coefficients()` - maximum number of coefficients
- `get_fmin()`, `get_fmax()` - frequency range (mel)
- `get_c0_at_frame(frame)` - C0 (energy) for a specific frame
- `get_value_in_frame(frame, coef)` - coefficient value at a frame
- `get_coefficients_at_frame(frame)` - all coefficients for a frame
- `get_all_coefficients()` - matrix of all coefficients
- `get_all_c0()` - vector of all C0 values

Liftering

- `lifter(L)` - apply cepstral liftering (weighting)

Export

- `as_data_frame(include_c0)` - convert to a `data.frame`/`data.table`
- `to_matrix()` - convert to a `Matrix` object

See Also

[Sound](#), [MelSpectrogram](#)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
mfcc <- sound$to_mfcc(
  num_coefficients = 13,
  analysis_width = 0.025,
  time_step = 0.01,
  f1_mel = 100,
  fmax_mel = 7800,
  df_mel = 100
)
n_frames <- mfcc$get_number_of_frames()
coefs <- mfcc$get_all_coefficients()
mfcc$lifter(22)
df <- mfcc$as_data_frame(include_c0 = TRUE)
```

mfccs_to_dtw	Create DTW from two MFCC objects
--------------	----------------------------------

Description

Create DTW from two MFCC objects

Usage

```
mfccs_to_dtw(
  mfcc1,
  mfcc2,
  coefficient_weight = 1,
  log_energy_weight = 0,
  coefficient_regression_weight = 0,
  log_energy_regression_weight = 0,
  regression_window_length = 0
)
```

Arguments

- mfcc1 First MFCC object (candidate, x-axis)
- mfcc2 Second MFCC object (reference, y-axis)
- coefficient_weight Weight for cepstral coefficients (default: 1.0)
- log_energy_weight Weight for log energy (c0) (default: 0.0)

coefficient_regression_weight
 Weight for coefficient regression (default: 0.0)

log_energy_regression_weight
 Weight for energy regression (default: 0.0)

regression_window_length
 Window for regression calculation (default: 0.0)

Value

A DTW object

Examples

```
s1 <- Sound$create_tone(frequency = 200, duration = 0.3, sampling_rate =
  16000)
s2 <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
dtw <- mfccs_to_dtw(s1$to_mfcc(), s2$to_mfcc())
```

multiband_hnr_stats	<i>Query reusable multiband HNR statistics</i>
---------------------	--

Description

Extract mean and standard deviation from a reusable set of multiband ‘Harmonicity’ objects built by [build_multiband_harmonicity()]. This is the cheap query step for repeated interval workflows.

Usage

```
multiband_hnr_stats(multiband, from_time = 0, to_time = 0)
```

Arguments

multiband	Named list of 5 ‘Harmonicity’ objects (or Harmonicity external pointers), typically returned by [build_multiband_harmonicity()].
from_time	Start time for statistics extraction (default ‘0’)
to_time	End time for statistics extraction (default ‘0’)

Value

Named list with the same ‘*_mean’ / ‘*_sd’ fields as [calculate_multiband_hnr_ultra()].

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1, sampling_rate =
  16000)

built <- build_multiband_harmonicity(sound)
hnr_interval1 <- multiband_hnr_stats(built, 0, 0.5)
hnr_interval2 <- multiband_hnr_stats(built, 0.5, 1.0)
```

pair_files

Pair Sound and TextGrid Files

Description

Find matching pairs of Sound and TextGrid files based on filename matching. Equivalent to Praat's manual file pairing but automated.

Usage

```
pair_files(
  sound_dir,
  textgrid_dir = sound_dir,
  sound_pattern = "\\wav$",
  textgrid_pattern = "\\TextGrid$",
  by = "basename",
  require_both = TRUE
)
```

Arguments

sound_dir	Character. Directory containing sound files.
textgrid_dir	Character. Directory containing TextGrid files (default: same as sound_dir).
sound_pattern	Character. Pattern for sound files (default: "\\wav\$").
textgrid_pattern	Character. Pattern for TextGrid files (default: "\\TextGrid\$").
by	Character. Matching strategy: "basename" (default), "exact", or a custom function.
require_both	Logical. Require both files to exist (default: TRUE).

Value

Data frame with columns: sound_file, textgrid_file, basename.

Examples

```

audio_dir <- tempfile("audio_")
dir.create(audio_dir)
Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)$save(
  file.path(audio_dir, "utt1.wav")
)
tg <- TextGrid$create(0, 0.3, "words")
tg$save(file.path(audio_dir, "utt1.TextGrid"))

# Find all matching pairs
pairs <- pair_files(sound_dir = audio_dir)

# Process each pair
results <- lapply(seq_len(nrow(pairs)), function(i) {
  sound <- Sound(pairs$sound_file[i])
  textgrid <- TextGrid(pairs$textgrid_file[i])
  # ... analysis ...
})

```

pair_sound_textgrid *Pair Sound and TextGrid Files*

Description

Find matching Sound and TextGrid files in directories. This replaces Praat's pattern of manually pairing files based on naming conventions.

Usage

```

pair_sound_textgrid(
  sound_dir,
  textgrid_dir = sound_dir,
  sound_pattern = "\\*.wav$",
  textgrid_pattern = "\\*.TextGrid$",
  by = "basename",
  require_both = TRUE
)

```

Arguments

sound_dir	Character path to directory containing audio files
textgrid_dir	Character path to directory containing TextGrid files (default: same as sound_dir)
sound_pattern	Pattern to match sound files (default: "*.wav\$")
textgrid_pattern	Pattern to match TextGrid files (default: "*.TextGrid\$")
by	Matching strategy: "basename" (default), "full", or a custom function
require_both	Logical, only return pairs where both files exist (default: TRUE)

Value

Data frame with columns: sound_file, textgrid_file, basename

Examples

```
audio_dir <- tempfile("audio_")
dir.create(audio_dir)
Sound$create_tone(frequency = 150, duration = 0.3, sampling_rate =
  16000)$save(
  file.path(audio_dir, "utt1.wav")
)
tg <- TextGrid$create(0, 0.3, "words")
tg$save(file.path(audio_dir, "utt1.TextGrid"))

# Find all matching pairs
pairs <- pair_sound_textgrid(sound_dir = audio_dir)

# Process each pair
for (i in seq_len(nrow(pairs))) {
  sound <- Sound(pairs$sound_file[i])
  textgrid <- TextGrid(pairs$textgrid_file[i])
  # ... process ...
}
```

PCA	PCA
-----	-----

Description

Principal Component Analysis (PCA) for dimensionality reduction and analysis of multivariate acoustic data.

Arguments

.xptr Not for direct use. External pointer to the underlying C++ PCA object; set internally when a method returns a new PCA (for example [pca_from_matrix](#)).

Details

PCA is commonly used in phonetics for vowel space analysis, speaker normalization, acoustic feature extraction, and data visualization.

Value

A PCA object with methods for querying components, eigenvalues, and projections.

See Also

[Discriminant](#)

Examples

```
set.seed(1)
data <- cbind(f1 = rnorm(20, 500, 50), f2 = rnorm(20, 1500, 100))
pca <- pca_from_matrix(data)
pca$get_number_of_components()
pca$get_eigenvalues()
```

pca_from_matrix	Create PCA from data matrix
-----------------	-----------------------------

Description

Performs Principal Component Analysis on a numeric matrix.

Usage

```
pca_from_matrix(data)
```

Arguments

data	Numeric matrix where rows are observations and columns are variables
------	--

Value

A PCA object

Examples

```
set.seed(1)
data <- matrix(rnorm(50), nrow = 10, ncol = 5)
pca <- pca_from_matrix(data)
print(pca)
```

Pitch	<i>Pitch</i>
-------	--------------

Description

Fundamental frequency (F0) contour of a sound (a Praat Pitch object).

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ Pitch object; set internally when a method returns a new Pitch.
-------	--

Details

Created from a Sound via autocorrelation or cross-correlation pitch tracking. Supports multiple unit systems: hertz, semitones (re 1 Hz, 100 Hz, or custom), mel, and erb; see the Units section for the full set of codes.

Value

A Pitch object.

Information

- `get_number_of_frames()` - number of analysis frames
- `get_time_step()` - time step between frames, in seconds
- `count_voiced_frames()` - number of frames with voiced (non-zero) pitch

Point queries

- `get_value_at_time(time, unit, interpolate)` - F0 at a time point
- `get_values_at_times(times, unit, interpolate)` - F0 at a vector of times (batch)
- `get_strength_at_time(time)` - strength (voicing likelihood) at a time
- `get_strengths_at_times(times)` - strengths at a vector of times (batch)

Statistics

Computed over a time range.

- `get_mean(from_time, to_time, unit)` - mean F0
- `get_standard_deviation(from_time, to_time, unit)` - standard deviation
- `get_minimum(from_time, to_time, unit, interpolate)` - minimum F0
- `get_maximum(from_time, to_time, unit, interpolate)` - maximum F0
- `get_quantile(quantile, from_time, to_time, unit)` - quantile of F0
- `get_time_of_minimum(...)`, `get_time_of_maximum(...)` - time of extremum

Export

- `as_vector()`, `as_data_frame()` - export as vector or data.frame
- `get_times_vector()` - frame time points
- `as_matrix()` - F0 values as a matrix (frames by candidates)

Transform

- `to_point_process(voicing_threshold, octave_cost, ...)` - convert to a PointProcess (glottal pulses)
- `down_to_pitch_tier()` - convert to a PitchTier (editable pitch contour)

Units

F0 unit codes: "hertz" (0), "semitones re 1 Hz" (1), "semitones re 100 Hz" (2), "semitones re 200 Hz" (3), "semitones re 440 Hz" (4), "mel" (5), "log hertz" (6), "erb" (7). Default is "hertz".

See Also

[Sound](#), [PointProcess](#), [PitchTier](#)

Examples

```
sound <- Sound$create_tone(duration = 1.0, frequency = 200, sampling_rate =
  44100)
pitch <- sound$to_pitch(time_step = 0.01, pitch_floor = 75, pitch_ceiling =
  600)
mean_f0 <- pitch$get_mean(from_time = 0, to_time = 0, unit = "hertz")
min_f0 <- pitch$get_minimum(from_time = 0, to_time = 0, unit = "hertz")
df <- as.data.frame(pitch)

# The same analysis on a recording read from disk
sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))
pitch <- sound$to_pitch(time_step = 0.01, pitch_floor = 75, pitch_ceiling =
  600)
n_voiced <- pitch$count_voiced_frames()
```

pitches_to_dtw

Create DTW from two Pitch objects

Description

Create DTW from two Pitch objects

Usage

```
pitches_to_dtw(
  pitch1,
  pitch2,
  vuv_costs = 24,
  time_weight = 10,
  match_start = TRUE,
  match_end = TRUE,
  slope = 1
)
```

Arguments

pitch1	First pitch contour (candidate)
pitch2	Second pitch contour (reference)
vuv_costs	Cost for voiced-unvoiced mismatches (default: 24)
time_weight	Weight for temporal distance (default: 10)
match_start	Force path to start at (1,1)
match_end	Force path to end at (nx,ny)
slope	Slope constraint (1-4)

Value

A DTW object

Examples

```
sound1 <- Sound$create_tone(frequency = 220, duration = 0.5)
sound2 <- Sound$create_tone(frequency = 440, duration = 0.5)
pitch1 <- sound1$to_pitch()
pitch2 <- sound2$to_pitch()
dtw <- pitches_to_dtw(pitch1, pitch2)
```

PitchModule	<i>Create a Pitch Object from Module</i>
-------------	--

Description

Creates a Pitch object using the new Rcpp Modules architecture. This is the constructor function for Pitch objects in pladdr 2.0.

Usage

```
PitchModule(.ptr = NULL)
```

Arguments

.ptr	External pointer to a Praat Pitch object (internal use)
------	---

Details

Pitch objects are typically created via ‘Sound\$to_pitch()’ rather than directly. The returned object provides methods for querying pitch values, statistics, and conversions.

Value

A Pitch object (reference class from Rcpp Module)

Methods

'get_value_at_time(time, unit, interpolate)' Get pitch at time
'get_mean(from_time, to_time, unit)' Get mean pitch
'get_standard_deviation(from_time, to_time, unit)' Get pitch SD
'get_minimum(from_time, to_time, unit, interpolate)' Get min pitch
'get_maximum(from_time, to_time, unit, interpolate)' Get max pitch
'count_voiced_frames()' Count voiced frames
'as_data_frame(include_strength, include_intensity)' Convert to data.frame

Properties

'duration' Duration in seconds
'nx' Number of frames
'dx' Time step between frames
'xmin', 'xmax' Start and end times
'ceiling' Pitch ceiling (Hz)

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 0.5)
pitch <- sound$to_pitch()

# Properties
pitch$duration
pitch$nx

# Query methods
pitch$get_mean(0, 0, "hertz") # Mean F0 in Hz
pitch$count_voiced_frames()

# Export
df <- pitch$as_data_frame(FALSE, FALSE)

```

PitchTier

PitchTier

Description

Praat PitchTier object: a sequence of time-value points describing a pitch contour.

Arguments

tmin	Start time in seconds, for creating an empty PitchTier.
tmax	End time in seconds, for creating an empty PitchTier.
.xptr	Not for direct use. External pointer to the underlying C++ PitchTier object; set internally when a method returns a new PitchTier.

Details

PitchTiers are used together with Manipulation objects to modify the pitch contour of sounds. Unlike Pitch objects, which hold sampled data, PitchTiers hold discrete time-value pairs that can be edited directly.

Value

A PitchTier object with methods for pitch-contour manipulation via time-value points.

Usage

```
PitchTier$new(path)           # load from file
PitchTier(tmin, tmax)         # create an empty PitchTier
pitch$down_to_pitch_tier()    # extract from a Pitch object
```

Query methods

- `get_number_of_points()` - number of pitch points
- `get_value_at_time(time)` - interpolated F0 at a time
- `get_value_at_index(index)` - F0 of a specific point
- `get_time_from_index(index)` - time of a specific point
- `get_minimum()`, `get_maximum()` - F0 range
- `get_mean(tmin, tmax)` - mean F0 (interpolated curve)
- `get_standard_deviation(tmin, tmax)` - standard deviation
- `get_area(tmin, tmax)` - area under the curve

Modification

- `add_point(time, value)` - add a pitch point (Hz)
- `remove_point(index)` - remove a point by index
- `remove_points_between(tmin, tmax)` - remove points in a time range
- `multiply_frequencies(factor)` - scale all frequencies
- `multiply_frequencies_in_range(tmin, tmax, factor)` - scale in a range
- `shift_frequencies(shift, unit)` - add to all frequencies
- `shift_frequencies_in_range(tmin, tmax, shift, unit)` - shift in a range
- `stylize(frequency_resolution, use_semitones)` - simplify the contour
- `interpolate_quadratically(points_per_parabola, logarithmically)` - smooth the contour

Conversion

- `to_sound_pulse_train(sample_rate)` - synthesize a pulse train
- `to_sound_phonation(sample_rate)` - synthesize phonation
- `to_sound_sine(sample_rate)` - synthesize a sine wave
- `down_to_point_process()` - extract time points
- `to_pitch(time_step, pitch_floor, pitch_ceiling)` - convert to a Pitch object

Export

- `as_data_frame()` - convert to a `data.table`
- `as_matrix()` - convert to a matrix
- `save(path)` - write to file

Examples

```
# Create empty PitchTier and add points
pt <- PitchTier(0, 1)
pt$add_point(0.1, 120)
pt$add_point(0.5, 150)
pt$add_point(0.9, 100)

# Create from Pitch
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
pitch <- sound$to_pitch()
pitch_tier <- pitch$down_to_pitch_tier()

# Modify pitch
pitch_tier$multiply_frequencies(1.5) # Raise pitch 50%

# Query
f0_at_mid <- pitch_tier$get_value_at_time(0.25)
n_points <- pitch_tier$get_number_of_points()
f0_min <- pitch_tier$get_minimum()
f0_max <- pitch_tier$get_maximum()

# Synthesize
synth <- pitch_tier$to_sound_sine(16000)

# Export
df <- pitch_tier$as_data_frame()
```

pladdr_simd

Get or set runtime SIMD usage

Description

Get or set runtime SIMD usage

Usage

```
pladdr_simd(enabled = NULL)
```

Arguments

enabled	Logical scalar to enable or disable SIMD at runtime. Use 'NULL' (default) to query the current state without changing it.
---------	---

Value

Invisibly, the same list returned by [simd_info()].

Examples

```
pladdr_simd()      # query current state
pladdr_simd(FALSE) # force scalar fallbacks when available
pladdr_simd(TRUE)  # restore SIMD
```

pladdr_threads	<i>Control multi-threading of Praat analyses</i>
----------------	--

Description

Praat's compute kernels (pitch, formant, spectrogram, CPPS, ...) run multi-threaded by default, using all available processor cores. Use this function to cap or disable threading, or to restore the default.

Usage

```
pladdr_threads(n = NULL)
```

Arguments

n	Maximum number of concurrent threads. Use '1' to disable threading, 'NULL' (default) to leave the setting unchanged and just return the current state, or '0' to restore automatic mode (all cores).
---	--

Details

The package's own batch helpers ('analyze_files_parallel()', 'process_sounds_parallel()', 'batch_process()') already cap each worker's threads automatically, so you only need this when running your own parallel loop (e.g. a hand-written 'mclapply()' where each worker should stay single-threaded).

Threading never changes results: threads partition analysis frames and each frame is computed exactly as in single-threaded mode.

Value

Invisibly, a list describing the current state:

processors Number of logical cores detected.

enabled Whether multi-threading is enabled.

max_threads Effective maximum concurrent threads.

min_elements_per_thread Minimum work per thread; 0 means each analysis routine uses its own tuned threshold.

Examples

```
pladdr_threads()      # query current state
pladdr_threads(1)     # single-threaded (e.g. inside mclapply workers)
pladdr_threads(0)     # back to automatic (all cores)
```

plot.Formant

*Plot Formant Tracks***Description**

Creates a formant trajectory visualization showing F1, F2, F3, etc.

Usage

```
## S3 method for class 'Formant'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3,
  garnish = TRUE,
  title = "Formant",
  colors = NULL,
  ...
)
```

Arguments

x	Formant object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
max_formant	Maximum formant number to display (default: 3)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Formant")
colors	Character vector. Colors for each formant (default: auto)
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.5)
formant <- sound$to_formant_burg()

# Basic plot
plot(formant)

# Show first 5 formants
plot(formant, max_formant = 5)

# Customize
plot(formant, max_formant = 2,
      colors = c("red", "blue"))

```

plot.Harmonicity	<i>Plot Harmonicity (HNR) Contour</i>
------------------	---------------------------------------

Description

Creates a harmonics-to-noise ratio contour visualization.

Usage

```

## S3 method for class 'Harmonicity'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  garnish = TRUE,
  title = "Harmonicity (HNR)",
  color = "darkviolet",
  ...
)

```

Arguments

x	Harmonicity object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Harmonicity")
color	Character. Line color (default: "darkviolet")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 1.0)
harmoniccity <- sound$to_harmoniccity_cc()

# Basic plot
plot(harmoniccity)

# Time range
plot(harmoniccity, from_time = 0.2, to_time = 0.8)
```

plot.Intensity	<i>Plot Intensity Contour</i>
----------------	-------------------------------

Description

Creates an intensity (loudness) contour visualization.

Usage

```
## S3 method for class 'Intensity'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  garnish = TRUE,
  title = "Intensity",
  color = "darkorange",
  ...
)
```

Arguments

x	Intensity object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Intensity")
color	Character. Line color (default: "darkorange")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 1.0)
intensity <- sound$to_intensity()

# Basic plot
plot(intensity)

# Time range
plot(intensity, from_time = 0.2, to_time = 0.8)
```

plot.Ltas	<i>Plot Long-Term Average Spectrum</i>
-----------	--

Description

Creates a LTAS (long-term average spectrum) visualization.

Usage

```
## S3 method for class 'Ltas'
plot(
  x,
  from_freq = NULL,
  to_freq = NULL,
  log_freq = FALSE,
  garnish = TRUE,
  title = "LTAS",
  color = "darkred",
  ...
)
```

Arguments

x	Ltas object
from_freq	Start frequency in Hz (NULL = from 0)
to_freq	End frequency in Hz (NULL = to maximum)
log_freq	Logical. Use logarithmic frequency scale (default: FALSE)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "LTAS")
color	Character. Line color (default: "darkred")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 1.0)
ltas <- sound$to_ltas()

# Basic plot
plot(ltas)

# Speech frequency range
plot(ltas, to_freq = 5000)
```

plot.Matrix	<i>Plot Matrix as Heatmap</i>
-------------	-------------------------------

Description

Creates a heatmap visualization of a Matrix object. Supports any Matrix-derived objects including generic matrices, spectrograms, etc.

Usage

```
## S3 method for class 'Matrix'
plot(
  x,
  from_x = NULL,
  to_x = NULL,
  from_y = NULL,
  to_y = NULL,
  garnish = TRUE,
  title = "Matrix",
  x_label = "X",
  y_label = "Y",
  color_scale = "viridis",
  ...
)
```

Arguments

x	Matrix object
from_x	Start value for x-axis (NULL = from beginning)
to_x	End value for x-axis (NULL = to end)
from_y	Start value for y-axis (NULL = from beginning)

to_y	End value for y-axis (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Matrix")
x_label	Character. X-axis label (default: "X")
y_label	Character. Y-axis label (default: "Y")
color_scale	Character. Color scale to use: "viridis", "magma", "plasma", "inferno", "cividis", or "greyscale" (default: "viridis")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
m <- matrix_create_simple(3, 4)

# Basic heatmap
plot(m)

# Custom color scale
plot(m, color_scale = "magma", title = "My Matrix")
```

plot.Pitch

Plot Pitch Contour

Description

Creates a pitch (F0) contour visualization of a Pitch object.

Usage

```
## S3 method for class 'Pitch'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  garnish = TRUE,
  title = "Pitch",
  color = "darkgreen",
  show_voicing = TRUE,
  ...
)
```


Arguments

x	Pitch object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Pitch")
color	Character. Line color (default: "darkgreen")
show_voicing	Logical. Color by voicing (default: TRUE)
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 1.0)
pitch <- sound$to_pitch()

# Basic plot
plot(pitch)

# Time range
plot(pitch, from_time = 0.2, to_time = 0.8)

# Customize
plot(pitch, show_voicing = FALSE, color = "blue") +
  ggplot2::geom_hline(yintercept = 120, linetype = "dashed")

```

plot.PointProcess	<i>Plot PointProcess Events</i>
-------------------	---------------------------------

Description

Creates a visualization of PointProcess events (e.g., glottal pulses).

Usage

```

## S3 method for class 'PointProcess'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  garnish = TRUE,
  title = "PointProcess",

```

```

    color = "black",
    ...
)

```

Arguments

x	PointProcess object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "PointProcess")
color	Character. Line color (default: "black")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 1.0)
pulses <- sound$to_pointprocess_periodic_cc()

# Basic plot
plot(pulses)

# Time range
plot(pulses, from_time = 0.2, to_time = 0.5)

```

plot.PowerCepstrum	<i>Plot PowerCepstrum</i>
--------------------	---------------------------

Description

Creates a visualization of a PowerCepstrum object showing the cepstral coefficients as a function of quefrency. Optionally highlights the peak related to pitch.

Usage

```

## S3 method for class 'PowerCepstrum'
plot(
  x,
  from_quefrency = NULL,
  to_quefrency = NULL,
  garnish = TRUE,

```

```
    title = "Power Cepstrum",
    color = "darkblue",
    mark_peak = TRUE,
    ...
)
```

Arguments

x	PowerCepstrum object
from_quefrency	Start quefrency in seconds (NULL = from beginning)
to_quefrency	End quefrency in seconds (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Power Cepstrum")
color	Character. Line color (default: "darkblue")
mark_peak	Logical. Mark the peak prominence if available (default: TRUE)
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)
spectrum <- sound$to_spectrum()
pc <- spectrum$to_power_cepstrum()

# Basic plot
plot(pc)

# Focus on vocal range (60-500 Hz = 0.002-0.0167 s quefrency)
plot(pc, from_quefrency = 0.002, to_quefrency = 0.017)

# Customize
plot(pc, color = "red", title = "Cepstral Analysis")
```

plot.Sound	<i>Plot Sound Waveform</i>
------------	----------------------------

Description

Creates a waveform visualization of a Sound object.

Usage

```
## S3 method for class 'Sound'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  garnish = TRUE,
  title = "Sound",
  color = "steelblue",
  ...
)
```

Arguments

x	Sound object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Sound")
color	Character. Line color (default: "steelblue")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 440, duration = 1.0)

# Basic plot
plot(sound)

# Time range
plot(sound, from_time = 0.2, to_time = 0.8)

# Customize
plot(sound, color = "darkblue", title = "Speech Recording") +
  ggplot2::geom_hline(yintercept = 0, linetype = "dashed", alpha = 0.5)
```

plot.Spectrogram	<i>Plot Spectrogram Heatmap</i>
------------------	---------------------------------

Description

Creates a time-frequency heatmap visualization of a Spectrogram.

Usage

```
## S3 method for class 'Spectrogram'
plot(
  x,
  from_time = NULL,
  to_time = NULL,
  from_freq = NULL,
  to_freq = NULL,
  garnish = TRUE,
  title = "Spectrogram",
  dynamic_range = 70,
  ...
)
```

Arguments

x	Spectrogram object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
from_freq	Start frequency in Hz (NULL = from 0)
to_freq	End frequency in Hz (NULL = to Nyquist)
garnish	Logical. Add axis labels and title (default: TRUE)
title	Character. Plot title (default: "Spectrogram")
dynamic_range	Numeric. Dynamic range in dB (default: 70)
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)
spectrogram <- sound$to_spectrogram()

# Basic plot
plot(spectrogram)
```

```
# Focus on speech range
plot(spectrogram, to_freq = 5000)
```

<code>plot.Spectrum</code>	<i>Plot Spectrum</i>
----------------------------	----------------------

Description

Creates a frequency spectrum visualization.

Usage

```
## S3 method for class 'Spectrum'
plot(
  x,
  from_freq = NULL,
  to_freq = NULL,
  log_freq = FALSE,
  garnish = TRUE,
  title = "Spectrum",
  color = "navy",
  ...
)
```

Arguments

<code>x</code>	Spectrum object
<code>from_freq</code>	Start frequency in Hz (NULL = from 0)
<code>to_freq</code>	End frequency in Hz (NULL = to Nyquist)
<code>log_freq</code>	Logical. Use logarithmic frequency scale (default: FALSE)
<code>garnish</code>	Logical. Add axis labels and title (default: TRUE)
<code>title</code>	Character. Plot title (default: "Spectrum")
<code>color</code>	Character. Line color (default: "navy")
<code>...</code>	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.5)
spectrum <- sound$to_spectrum()

# Basic plot
plot(spectrum)

# Logarithmic frequency
plot(spectrum, log_freq = TRUE)

```

plot.TextGrid

*Plot TextGrid Annotations***Description**

Visualize tier labels and boundaries as a standalone plot.

Usage

```

## S3 method for class 'TextGrid'
plot(x, tier = NULL, from_time = NULL, to_time = NULL, ...)

```

Arguments

x	A TextGrid object
tier	Integer or character specifying which tier to plot (default: all tiers)
from_time	Start time in seconds (NULL = beginning)
to_time	End time in seconds (NULL = end)
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot2 object

Examples

```

tg <- TextGrid$create(0, 1, "words")
tg$set_interval_text("words", 1, "hello")
plot(tg)
plot(tg, tier = 1)

```

plot_cpp_timeseries *Plot CPP Time Series*

Description

Creates a line plot of Cepstral Peak Prominence (CPP) values over time from a PowerCepstrogram object. Useful for tracking voice quality variation.

Usage

```
plot_cpp_timeseries(
  cepstrogram,
  time_range = NULL,
  qmin = 0.001,
  qmax = 0,
  n_samples = 100,
  smooth = FALSE,
  smooth_span = 0.1,
  reference_lines = NULL,
  title = NULL,
  theme = c("minimal", "bw", "classic")
)
```

Arguments

cepstrogram	PowerCepstrogram object
time_range	Numeric vector. c(start, end) time range to display (default: NULL = auto)
qmin	Numeric. Minimum quefrency for peak search (default: 0.001)
qmax	Numeric. Maximum quefrency for peak search (default: 0)
n_samples	Integer. Number of time points to sample (default: 100)
smooth	Logical. Apply smoothing to CPP contour (default: FALSE)
smooth_span	Numeric. Smoothing span for loess (default: 0.1)
reference_lines	Numeric vector. Reference CPP values to plot as horizontal lines (default: NULL)
title	Character. Plot title (default: auto-generated)
theme	Character. ggplot2 theme (default: "minimal")

Value

A ggplot2 object

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.5)
cepstrogram <- sound$to_powercepstrogram(pitch_floor = 60, time_step = 0.002)

# CPP time series
plot_cpp_timeseries(cepstrogram, n_samples = 20)

# With smoothing and reference lines
plot_cpp_timeseries(cepstrogram,
                    n_samples = 20,
                    smooth = TRUE,
                    reference_lines = c(5, 10, 15))

```

plot_pitch_intensity *Plot Pitch and Intensity Together*

Description

Creates a dual-axis visualization showing pitch and intensity contours together. This replicates Praat's Pitch_Intensity_draw() function.

Usage

```

plot_pitch_intensity(
  pitch,
  intensity,
  from_time = NULL,
  to_time = NULL,
  pitch_color = "darkgreen",
  intensity_color = "darkorange",
  title = "Pitch and Intensity",
  ...
)

```

Arguments

pitch	Pitch object
intensity	Intensity object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
pitch_color	Character. Pitch line color (default: "darkgreen")
intensity_color	Character. Intensity line color (default: "darkorange")
title	Character. Plot title (default: "Pitch and Intensity")
...	Additional arguments passed to the underlying function or ignored.

Value

A ggplot object with dual y-axes

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 1.0)
pitch <- sound$to_pitch()
intensity <- sound$to_intensity()

# Combined plot
plot_pitch_intensity(pitch, intensity)

# Time range
plot_pitch_intensity(pitch, intensity,
                     from_time = 0.2, to_time = 0.8)
```

plot_powercepstrogram *Plot PowerCepstrogram*

Description

Creates a heatmap visualization of a power cepstrogram showing how the cepstral spectrum varies over time, similar to a spectrogram.

Usage

```
plot_powercepstrogram(
  cepstrogram,
  time_range = NULL,
  quefrequency_range = c(0, 0.05),
  db_range = NULL,
  color_scale = c("viridis", "inferno", "magma", "plasma"),
  show_cpp_contour = FALSE,
  contour_color = "white",
  title = NULL,
  theme = c("minimal", "bw", "classic")
)
```

Arguments

cepstrogram	PowerCepstrogram object
time_range	Numeric vector. c(start, end) time range to display (default: NULL = auto)
quefrequency_range	Numeric vector. c(min, max) quefrequency range to display (default: c(0, 0.05))
db_range	Numeric vector. c(min, max) dB range for color scale (default: NULL = auto)

color_scale	Character. Color palette: "viridis", "inferno", "magma", "plasma" (default: "viridis")
show_cpp_contour	Logical. Overlay CPP contour over time (default: FALSE)
contour_color	Character. Color for CPP contour line (default: "white")
title	Character. Plot title (default: auto-generated)
theme	Character. ggplot2 theme (default: "minimal")

Value

A ggplot2 object

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 0.5)
cepstrogram <- sound$to_powercepstrogram(pitch_floor = 60, time_step = 0.002)

# Basic plot
plot_powercepstrogram(cepstrogram)

# With CPP contour
plot_powercepstrogram(cepstrogram,
                      show_cpp_contour = TRUE,
                      quefreny_range = c(0.001, 0.02))

```

plot_powercepstrum	<i>Plot PowerCepstrum</i>
--------------------	---------------------------

Description

Creates a visualization of a power cepstrum showing the cepstral values across quefrencies, with optional peak and trend line annotations.

Usage

```

plot_powercepstrum(
  cepstrum,
  show_peak = TRUE,
  show_trendline = TRUE,
  qmin = 0.001,
  qmax = 0,
  fit_method = c("straight", "exponential decay", "parabolic"),
  quefreny_range = NULL,
  db_range = NULL,
  title = NULL,
  theme = c("minimal", "bw", "classic")
)

```

Arguments

cepstrum	PowerCepstrum object
show_peak	Logical. Highlight the cepstral peak (default: TRUE)
show_trendline	Logical. Show regression trend line (default: TRUE)
qmin	Numeric. Minimum quefrency for peak search (seconds, default: 0.001)
qmax	Numeric. Maximum quefrency for peak search (seconds, default: 0)
fit_method	Character. Trend line fit method (default: "straight")
quefrency_range	Numeric vector. c(min, max) quefrency range to display (default: NULL = auto)
db_range	Numeric vector. c(min, max) dB range to display (default: NULL = auto)
title	Character. Plot title (default: auto-generated)
theme	Character. ggplot2 theme: "minimal", "bw", "classic" (default: "minimal")

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)
spectrum <- sound$to_spectrum()
cepstrum <- spectrum$to_power_cepstrum()

# Basic plot
plot_powercepstrum(cepstrum)

# Customized plot
plot_powercepstrum(cepstrum,
  show_peak = TRUE,
  show_trendline = TRUE,
  quefrency_range = c(0.001, 0.02),
  title = "Voice Quality Analysis")
```

plot_sound_pitch	<i>Plot Sound Waveform with Pitch Contour</i>
------------------	---

Description

Creates a two-panel visualization showing the sound waveform in the top panel and pitch contour in the bottom panel, aligned by time. This is a common Praat visualization pattern.

Usage

```
plot_sound_pitch(
  sound,
  pitch,
  from_time = NULL,
  to_time = NULL,
  waveform_color = "steelblue",
  pitch_color = "darkblue",
  title = NULL,
  ...
)
```

Arguments

sound	Sound object
pitch	Pitch object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
waveform_color	Character. Waveform color (default: "steelblue")
pitch_color	Character. Pitch color (default: "darkblue")
title	Character. Overall plot title (default: NULL)
...	Additional arguments passed to the underlying function or ignored.

Value

A combined plot object (requires patchwork or gridExtra)

Examples

```
if (requireNamespace("patchwork", quietly = TRUE) ||
    requireNamespace("gridExtra", quietly = TRUE)) {
  sound <- Sound$create_tone(frequency = 220, duration = 1.0)
  pitch <- sound$to_pitch()

  # Basic two-panel plot
  plot_sound_pitch(sound, pitch)

  # Time range and custom colors
  plot_sound_pitch(sound, pitch,
    from_time = 0.2, to_time = 0.8,
    waveform_color = "black",
    pitch_color = "red")
}
```

plot_spectrogram_formants

Plot Spectrogram with Formant Overlay

Description

Creates a spectrogram heatmap with formant trajectories overlaid. This is a common visualization pattern in Praat for vowel analysis.

Usage

```
plot_spectrogram_formants(
  spectrogram,
  formant,
  from_time = NULL,
  to_time = NULL,
  max_formant = 3,
  formant_colors = NULL,
  dynamic_range = 70,
  title = "Spectrogram + Formants",
  ...
)
```

Arguments

spectrogram	Spectrogram object
formant	Formant object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
max_formant	Maximum formant number to display (default: 3)
formant_colors	Character vector. Colors for formants (default: auto)
dynamic_range	Numeric. Spectrogram dynamic range in dB (default: 70)
title	Character. Plot title (default: "Spectrogram + Formants")
...	Additional arguments passed to plot.Spectrogram

Value

A ggplot object with formant tracks overlaid on spectrogram

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)
spectrogram <- sound$to_spectrogram()
formant <- sound$to_formant_burg()
```

```
# Combined plot
plot_spectrogram_formants(spectrogram, formant)

# Show F1-F5
plot_spectrogram_formants(spectrogram, formant, max_formant = 5)
```

plot_spectrogram_pitch

Plot Spectrogram with Pitch Overlay

Description

Creates a combined visualization showing a spectrogram with pitch contour overlaid. This is one of the most common Praat visualizations for voice analysis.

Usage

```
plot_spectrogram_pitch(
  spectrogram,
  pitch,
  from_time = NULL,
  to_time = NULL,
  freq_max = 5000,
  pitch_color = "blue",
  pitch_floor = NULL,
  pitch_ceiling = NULL,
  title = "Spectrogram with Pitch",
  ...
)
```

Arguments

spectrogram	Spectrogram object
pitch	Pitch object
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)
freq_max	Maximum frequency to display in Hz (default: 5000)
pitch_color	Character. Pitch track color (default: "blue")
pitch_floor	Minimum F0 to display in Hz (default: NULL = auto)
pitch_ceiling	Maximum F0 to display in Hz (default: NULL = auto)
title	Character. Plot title (default: "Spectrogram with Pitch")
...	Additional arguments passed to plot methods

Value

A ggplot2 object

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)
spectrogram <- sound$to_spectrogram()
pitch <- sound$to_pitch()

# Basic combined plot
plot_spectrogram_pitch(spectrogram, pitch)

# Customize pitch range
plot_spectrogram_pitch(spectrogram, pitch,
  pitch_floor = 75, pitch_ceiling = 500,
  pitch_color = "red")
```

plot_textgrid_pitch *Plot TextGrid with Pitch Contour*

Description

Creates a combined visualization showing a pitch contour with TextGrid annotation tiers. This replicates Praat's TextGrid_Pitch_draw() function.

Usage

```
plot_textgrid_pitch(
  textgrid,
  pitch,
  tier = NULL,
  from_time = NULL,
  to_time = NULL,
  pitch_color = "darkgreen",
  tier_colors = NULL,
  title = NULL,
  ...
)
```

Arguments

textgrid	TextGrid object
pitch	Pitch object
tier	Integer or character. Tier number or name to display (default: all)
from_time	Start time in seconds (NULL = from beginning)
to_time	End time in seconds (NULL = to end)

pitch_color	Character. Pitch line color (default: "darkgreen")
tier_colors	Character vector. Colors for each tier (default: auto)
title	Character. Plot title (default: NULL)
...	Additional arguments passed to plot methods

Value

A combined ggplot object

Examples

```
if (requireNamespace("patchwork", quietly = TRUE) ||
    requireNamespace("gridExtra", quietly = TRUE)) {
  sound <- Sound$create_tone(frequency = 220, duration = 1.0)
  pitch <- sound$to_pitch()
  textgrid <- TextGrid$create(0, 1, "phonemes")
  textgrid$set_interval_text("phonemes", 1, "a")

  # Combined pitch + TextGrid
  plot_textgrid_pitch(textgrid, pitch)

  # Single tier
  plot_textgrid_pitch(textgrid, pitch, tier = "phonemes")
}
```

plot_textgrid_sound *Plot TextGrid with Sound Waveform*

Description

Creates a combined visualization showing a waveform with TextGrid annotation tiers overlaid. This replicates Praat's TextGrid_Sound_draw() function.

Usage

```
plot_textgrid_sound(
  textgrid,
  sound,
  tier = NULL,
  from_time = NULL,
  to_time = NULL,
  waveform_color = "steelblue",
  tier_colors = NULL,
  title = NULL,
  ...
)
```

Arguments

<code>textgrid</code>	TextGrid object
<code>sound</code>	Sound object
<code>tier</code>	Integer or character. Tier number or name to display (default: all)
<code>from_time</code>	Start time in seconds (NULL = from beginning)
<code>to_time</code>	End time in seconds (NULL = to end)
<code>waveform_color</code>	Character. Waveform color (default: "steelblue")
<code>tier_colors</code>	Character vector. Colors for each tier (default: auto)
<code>title</code>	Character. Plot title (default: NULL)
<code>...</code>	Additional arguments passed to plot methods

Value

A combined ggplot object (requires patchwork or gridExtra)

Examples

```
if (requireNamespace("patchwork", quietly = TRUE) ||
    requireNamespace("gridExtra", quietly = TRUE)) {
  sound <- Sound$create_tone(frequency = 440, duration = 1.0)
  textgrid <- TextGrid$create(0, 1, "words")
  textgrid$set_interval_text("words", 1, "tone")

  # Basic combined plot
  plot_textgrid_sound(textgrid, sound)

  # Single tier with time range
  plot_textgrid_sound(textgrid, sound, tier = 1,
                      from_time = 0.2, to_time = 0.8)
}
```

plotting-combined

Combined Visualization Functions

Description

Convenience functions for common multi-object visualizations that combine multiple Praat objects in a single plot. These replicate common Praat plotting patterns using ggplot2 and patchwork/gridExtra.

Value

This is a documentation-only overview; see the individual functions (e.g. [plot_sound_pitch](#), [plot_textgrid_sound](#)) for their return values.

Examples

```
# See individual functions, e.g. ?plot_sound_pitch
```

plotting-methods

S3 Plot Methods for pladdrr Objects

Description

Convenient S3 plot methods for Praat objects. These are ggplot2-based wrappers that provide sensible defaults while allowing full customization. All methods return ggplot2 objects that can be further customized.

Details

These methods follow R's S3 generic dispatch system, allowing you to use the standard 'plot()' function with pladdrr objects. Each method converts the Praat object to a data frame and creates an appropriate ggplot2 visualization.

All plot methods support: - Time range filtering via 'from_time' and 'to_time' parameters - Axis labels and titles via 'garnish' parameter - ggplot2 object return for further customization

Value

This is a documentation-only overview; see the individual methods (e.g. [plot.Sound](#), [plot.Pitch](#)) for their return values.

Examples

```
# See individual methods, e.g. ?plot.Sound
```

PointProcess

Praat PointProcess object

Description

A PointProcess is a sequence of time points, typically representing glottal pulse boundaries (moments of vocal fold closure). Used for voice quality analysis: jitter (period perturbation) and shimmer (amplitude perturbation).

Arguments

tmin, tmax	When creating an empty PointProcess, the time span of the object. Leave NULL (the default) when supplying .xptr.
.xptr	Not for direct use. External pointer to the underlying C++ PointProcess object; set internally when a method returns a new PointProcess.

Value

A PointProcess object with methods for glottal pulse analysis, including jitter and shimmer.

Query methods

- `get_number_of_points()` - total number of time points
- `get_time(index)` - time of point at index
- `get_nearest_index(time)` - index of point nearest to a given time
- `get_low_index(time), get_high_index(time)` - index bounding a time
- `get_interval(time)` - time between surrounding points
- `get_mean_period(from_time, to_time, ...)` - mean glottal period
- `get_stdev_period(from_time, to_time, ...)` - SD of glottal period
- `get_number_of_periods(from_time, to_time, ...)` - number of periods in range
- `get_voice_breaks(from_time, to_time, ...)` - number of voice breaks

Jitter methods

Period perturbation measures. See the Note section for units.

- `get_jitter_local(from_time, to_time, ...)` - local jitter (Jloc)
- `get_jitter_local_absolute(from_time, to_time, ...)` - local absolute jitter
- `get_jitter_rap(from_time, to_time, ...)` - relative average perturbation
- `get_jitter_ppq5(from_time, to_time, ...)` - 5-period perturbation quotient
- `get_jitter_ddp(from_time, to_time, ...)` - difference of differences of periods

Shimmer methods

Amplitude perturbation measures; each takes a Sound. See the Note section for units.

- `get_shimmer_local(sound, from_time, to_time, ...)` - local shimmer (Shim)
- `get_shimmer_local_db(sound, from_time, to_time, ...)` - local shimmer in dB
- `get_shimmer_apq3(sound, from_time, to_time, ...)` - 3-period amplitude perturbation quotient
- `get_shimmer_apq5(sound, from_time, to_time, ...)` - 5-period APQ
- `get_shimmer_apq11(sound, from_time, to_time, ...)` - 11-period APQ
- `get_shimmer_dda(sound, from_time, to_time, ...)` - difference of differences of amplitude

Batch and report methods

- `get_jitter_shimmer_batch(pointprocess, sound, ...)` - all jitter and shimmer values as a named list, computed in one C++ call. Cache-aware: later calls with the same parameters return instantly.
- `voice_report(sound, pitch, ...)` - combined voice report (jitter, shimmer, HNR)

Export and conversion methods

- `as_vector()`, `as_data_frame()` - export as a vector or data.frame
- `to_textgrid_vuv(max_voiced_period, ...)` - create a voiced/unvoiced TextGrid
- `to_pitch_tier()` - convert to a PitchTier
- `to_sound_pulse_train(...)` - create a pulse train Sound

Note

Jitter and shimmer values are returned as decimals (0-1), not percentages. Multiply by 100 for percentage display. The first shimmer call caches all 11 metrics; later jitter or shimmer calls with matching parameters return from cache (no additional C++ call).

See Also

[Sound](#), [Pitch](#), [AmplitudeTier](#)

Examples

```
sound <- Sound$create_tone(duration = 1.0, frequency = 150, sampling_rate =
  44100)
pp <- sound$to_point_process_periodic_cc(pitch_floor = 75, pitch_ceiling =
  600)
jitter <- pp$get_jitter_local()
shimmer <- pp$get_shimmer_local(sound)
```

Polygon

Polygon Object

Description

2D polygon for geometric operations and spatial analysis.

Arguments

<code>x</code>	Numeric vector of x coordinates. Required unless <code>.xptr</code> is given.
<code>y</code>	Numeric vector of y coordinates. Required unless <code>.xptr</code> is given.
<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ Polygon object; set internally when a method returns a new Polygon.

Details

Uses Rcpp Modules for high-performance access to Praat's Polygon object.

Value

A Polygon object with methods for geometry operations.

Common uses

- Vowel space boundaries
- Formant space visualization
- Acoustic space analysis
- Convex hull computation

Examples

```
# Create polygon from formant data
poly <- Polygon(x = c(730, 1090, 2440), y = c(1090, 1220, 2440))

# Geometry queries
perimeter <- poly$get_perimeter()
n_points <- poly$n_points()

# Optimize path (traveling salesman)
poly$optimize_salesperson(iterations = 100)

# Export
df <- as.data.frame(poly)
```

PowerCepstrogram	<i>PowerCepstrogram</i>
------------------	-------------------------

Description

A Praat PowerCepstrogram: a power cepstrum computed at every time frame of a sound, forming a quefrency-by-time surface.

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ PowerCepstrogram object; set internally when a method returns a new PowerCepstrogram.
-------	--

Details

It's the standard input for CPPS (cepstral peak prominence, smoothed), a widely used measure of voice quality: high CPPS means a strong, regular harmonic structure, low CPPS means a noisy or aperiodic voice. Create one with `sound$to_powercepstrogram()`, then either read `get_cppps()` directly or slice out a single frame as a PowerCepstrum for closer inspection.

Value

A PowerCepstrogram object.

Usage

```
cepstrogram <- sound$to_powercepstrogram(pitch_floor = 60)
```

Query methods

- `get_cpp_at_time(time, interpolation, qmin, qmax, fit_method, tolerance)` - cepstral peak prominence at a single time
- `get_mean_cpp(from_time, to_time, qmin, qmax, fit_method, tolerance)` - mean cepstral peak prominence over a time range
- `get_cppts(subtract_tilt, time_averaging_window, quefrency_averaging_window, pitch_floor, pitch_ceiling, delta_f0, interpolation, qstart_fit, qend_fit, trend_type, fit_method)` - CPPS, smoothed over time and quefrency

Transformation and export

- `get_power_cepstrum_at_time(time)` - single frame as a PowerCepstrum object
- `smooth(time_averaging_window, quefrency_averaging_window)` - smoothed copy as a new PowerCepstrogram
- `to_matrix()` - values as a Matrix object
- `as_matrix()` - values as a plain numeric matrix (quefrency x time)

See Also

```
[PowerCepstrum], [Sound], [get_cppts_fast]
```

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.3)
cepstrogram <- sound$to_powercepstrogram(pitch_floor = 60)
mean_cpp <- cepstrogram$get_mean_cpp()
```

PowerCepstrum	<i>PowerCepstrum</i>
---------------	----------------------

Description

A PowerCepstrum is the power spectrum of the log power spectrum, a representation that separates the source (glottal pulse, low quefrency) from the filter (vocal tract, high quefrency). Created from a Spectrum or extracted from a PowerCepstrogram at a specific time. The primary voice quality metric from this object is CPP (Cepstral Peak Prominence).

Value

A PowerCepstrum object.

Query methods

- `get_qmin()`, `get_qmax()` - quefrency range (s)
- `get_quefrency_range()` - quefrency range as a vector
- `get_n_bins()` - number of quefrency bins
- `get_dq()` - quefrency step (s)
- `get_q1()` - starting quefrency value (s)

Peak analysis

- `get_peak_prominence(pitch_floor, pitch_ceiling, ...)` - CPP value (dB), the main voice quality metric
- `get_peak_prominence_hillenbrand(pitch_floor, pitch_ceiling)` - CPP using the Hillenbrand algorithm
- `get_quefrency_of_peak(interpolation)` - quefrency of the cepstral peak (s)
- `get_value_at_quefrency(quefrency, interpolation, unit)` - cepstral amplitude at a quefrency

Trend and smoothing

- `smooth(averaging_window)` - smooth the cepstrum
- `fit_trend_line(qmin, qmax, trend_type, fit_method)` - fit a regression trend line
- `get_trend_line_value(quefrency, ...)` - value of the fitted trend at a quefrency
- `subtract_trend(qstart_fit, qend_fit, ...)` - subtract the regression trend (returns a new PowerCepstrum)
- `subtract_trend_inplace(qstart_fit, qend_fit, ...)` - subtract the trend in place (mutates)

Export and transform

- `as_matrix()`, `as_data_frame()` - export
- `to_spectrum(random_phases)` - convert back to Spectrum
- `to_matrix()` - export as a matrix

See Also

[Spectrum](#), [PowerCepstrogram](#), [Sound](#)

Examples

```
sound <- Sound$create_tone(duration = 0.5, frequency = 200, sampling_rate =
  44100)
spectrum <- sound$to_spectrum()
cepstrum <- spectrum$to_power_cepstrum()
cpp <- cepstrum$get_peak_prominence()

# The same analysis on a recording read from disk
```



```
wav <- system.file("extdata", "test.wav", package = "pladdrr")
cepstrum <- Sound(wav)$to_spectrum()$to_power_cepstrum()
```

pp_get_mean_period_direct

Get PointProcess Mean Period Directly (Bypass R6)

Description

Get mean period from PointProcess without R6 wrapper overhead. Critical for VUV analysis workflows.

Usage

```
pp_get_mean_period_direct(
  pointprocess,
  from_time = 0,
  to_time = 0,
  period_floor = 1e-04,
  period_ceiling = 0.02,
  max_period_factor = 1.3
)
```

Arguments

pointprocess	PointProcess object or external pointer
from_time	Start time (0 = beginning)
to_time	End time (0 = end)
period_floor	Minimum period (default: 0.0001)
period_ceiling	Maximum period (default: 0.02)
max_period_factor	Maximum period factor (default: 1.3)

Value

Mean period in seconds

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1.0)
pitch <- sound$to_pitch()
pp_ptr <- to_point_process_from_sound_and_pitch(sound, pitch)
mean_period <- pp_get_mean_period_direct(pp_ptr)
```

`pp_get_stdev_period_direct`*Get PointProcess Period Standard Deviation Directly (Bypass R6)*

Description

Get standard deviation of periods from PointProcess without R6 wrapper overhead.

Usage

```
pp_get_stdev_period_direct(  
  pointprocess,  
  from_time = 0,  
  to_time = 0,  
  period_floor = 1e-04,  
  period_ceiling = 0.02,  
  max_period_factor = 1.3  
)
```

Arguments

<code>pointprocess</code>	PointProcess object or external pointer
<code>from_time</code>	Start time (0 = beginning)
<code>to_time</code>	End time (0 = end)
<code>period_floor</code>	Minimum period
<code>period_ceiling</code>	Maximum period
<code>max_period_factor</code>	Maximum period factor

Value

Standard deviation of periods

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1.0)  
pitch <- sound$to_pitch()  
pp_ptr <- to_point_process_from_sound_and_pitch(sound, pitch)  
sd_period <- pp_get_stdev_period_direct(pp_ptr)
```

praat_direct	<i>Direct Function Dispatch API</i>
--------------	-------------------------------------

Description

These functions provide direct access to Praat operations without R6 class dispatch overhead — no R6 environment lookup, no named parameter matching, no result wrapping. Use them in tight loops where you are comfortable working with external pointers instead of R6 objects.

****Output:**** Numerically identical to R6 methods.

Value

This is a documentation-only overview; see the individual functions (e.g. [to_point_process_direct](#), [pp_get_mean_period_direct](#)) for their return values.

When to Use

- Processing many files in a batch loop
- Latency-sensitive analysis pipelines
- Tight loops with many queries
- When profiling shows R6 dispatch overhead as a bottleneck

When NOT to Use

- Interactive exploration (use R6 for convenience)
- Small datasets (overhead is negligible)
- When you need method chaining

Examples

```
# See individual functions, e.g. ?pp_get_mean_period_direct
```

praat_eval_matrix	<i>Evaluate a matrix Praat expression</i>
-------------------	---

Description

Evaluate a matrix Praat expression

Usage

```
praat_eval_matrix(expression)
```

Arguments

expression	Character string containing a Praat matrix formula
------------	--

Value

Numeric matrix

Examples

```
# Create matrix
mat <- praat_eval_matrix("{ { 1, 2 }, { 3, 4 } }")
```

praat_eval_numeric	<i>Evaluate a numeric Praat expression</i>
--------------------	--

Description

Evaluate a numeric Praat expression

Usage

```
praat_eval_numeric(expression)
```

Arguments

expression	Character string containing a Praat formula expression
------------	--

Value

Numeric value

Examples

```
# Calculate value
result <- praat_eval_numeric("sqrt(16) + 2^3")
```

praat_eval_string	<i>Evaluate a string Praat expression</i>
-------------------	---

Description

Evaluate a string Praat expression

Usage

```
praat_eval_string(expression)
```

Arguments

expression	Character string containing a Praat string formula
------------	--

Value

Character string

Examples

```
# String operations
result <- praat_eval_string('"Hello" + " " + "World"')
```

praat_eval_string_array

Evaluate a string array Praat expression

Description

Evaluate a string array Praat expression

Usage

```
praat_eval_string_array(expression)
```

Arguments

expression Character string containing a Praat string array formula

Value

Character vector

Examples

```
# Create string array
arr <- praat_eval_string_array('{ "hello", "world" }')
```

praat_eval_vector

Evaluate a vector Praat expression

Description

Evaluate a vector Praat expression

Usage

```
praat_eval_vector(expression)
```

Arguments

expression Character string containing a Praat vector formula

Value

Numeric vector

Examples

```
# Create vector
vec <- praat_eval_vector("{ 1, 2, 3, 4, 5 }")
```

praat_init	<i>Initialize Praat interpreter</i>
------------	-------------------------------------

Description

Manually initialize the Praat interpreter. Normally called automatically when needed, but can be called explicitly to control timing.

Usage

```
praat_init()
```

Value

Invisibly returns NULL

Examples

```
praat_init()
```

praat_initialized	<i>Check if Praat interpreter is initialized</i>
-------------------	--

Description

Check if Praat interpreter is initialized

Usage

```
praat_initialized()
```

Value

Logical; TRUE if initialized, FALSE otherwise

Examples

```
praat_init()
praat_initialized()
```

praat_list_objects	<i>List all objects in Praat object list</i>
--------------------	--

Description

List all objects in Praat object list

Usage

```
praat_list_objects()
```

Value

Data frame with columns: id, name, class, selected

Examples

```
objects <- praat_list_objects()
print(objects)
```

praat_object_count	<i>Get count of objects in Praat object list</i>
--------------------	--

Description

Get count of objects in Praat object list

Usage

```
praat_object_count()
```

Value

Integer count of objects

Examples

```
count <- praat_object_count()
```

praat_run_script	<i>Execute a Praat script</i>
------------------	-------------------------------

Description

Executes a Praat script with automatic interpreter initialization. Objects created during script execution remain in Praat's internal object list.

Usage

```
praat_run_script(script)
```

Arguments

script	Character string containing Praat script code
--------	---

Value

Invisibly returns NULL

Examples

```
# Create a sound and extract pitch
praat_run_script('
  Create Sound from formula: "test", 1, 0, 1, 44100, "0.5 * sin(2*pi*440*x)"
  pitch = To Pitch: 0.0, 75, 600
')
```

praat_version	<i>Get Praat version information</i>
---------------	--------------------------------------

Description

Returns the version string for the Praat library integration

Usage

```
praat_version()
```

Value

Character string with version information

Examples

```
praat_version()
```

PraatInterpreter	<i>Praat Script Interpreter</i>
------------------	---------------------------------

Description

The PraatInterpreter provides a persistent Praat scripting environment within R. Unlike one-shot script execution, the interpreter maintains state between calls, enabling incremental script development and interactive exploration.

Details

R6 class for executing Praat scripts with persistent interpreter state. Allows running multiple scripts while maintaining variables and state between runs. Provides bidirectional object transfer between R and Praat's object list.

Value

An R6 object of class PraatInterpreter.

Methods

Public methods:

- `PraatInterpreter$new()`
- `PraatInterpreter$run()`
- `PraatInterpreter$get_variable()`
- `PraatInterpreter$set_variable()`
- `PraatInterpreter$eval()`
- `PraatInterpreter$object_count()`
- `PraatInterpreter$list_objects()`
- `PraatInterpreter$get_object()`
- `PraatInterpreter$get_object_by_id()`
- `PraatInterpreter$set_object()`
- `PraatInterpreter$remove_object()`
- `PraatInterpreter$remove_object_by_id()`
- `PraatInterpreter$select_object()`
- `PraatInterpreter$clear_objects()`
- `PraatInterpreter$print()`
- `PraatInterpreter$clone()`

`PraatInterpreter$new()`: Create a new interpreter instance with empty state.

Usage:

```
PraatInterpreter$new()
```

`PraatInterpreter$run()`: Execute Praat script code.

Usage:

```
PraatInterpreter$run(script)
```

Arguments:

script Character string with Praat script.

Returns: Self (invisibly), for method chaining.

PraatInterpreter\$get_variable(): Get a variable's value from the interpreter.

Usage:

```
PraatInterpreter$get_variable(name)
```

Arguments:

name Variable name.

Returns: Variable value.

PraatInterpreter\$set_variable(): Set a variable's value in the interpreter.

Usage:

```
PraatInterpreter$set_variable(name, value)
```

Arguments:

name Variable name.

value Variable value.

Returns: Self (invisibly).

PraatInterpreter\$eval(): Evaluate a Praat expression and return the result.

Usage:

```
PraatInterpreter$eval(expression)
```

Arguments:

expression Praat expression.

Returns: Result of the expression.

PraatInterpreter\$object_count(): Get the count of objects in the Praat object list.

Usage:

```
PraatInterpreter$object_count()
```

Returns: Integer count.

PraatInterpreter\$list_objects(): List all objects in the Praat object list.

Usage:

```
PraatInterpreter$list_objects()
```

Returns: A data.frame with id, name, class, and selected columns.

PraatInterpreter\$get_object(): Get a Praat object from the interpreter's object list.

Usage:

```
PraatInterpreter$get_object(name, type = NULL)
```

Arguments:

name Object name.

type Expected type (optional).

Returns: An R6 object.

PraatInterpreter\$get_object_by_id(): Get a Praat object by ID.

Usage:

```
PraatInterpreter$get_object_by_id(id)
```

Arguments:

id Object ID.

Returns: An R6 object.

PraatInterpreter\$set_object(): Add an R object to Praat's object list.

Usage:

```
PraatInterpreter$set_object(name, object)
```

Arguments:

name Object name.

object A PraatObject.

Returns: Object ID (invisibly).

PraatInterpreter\$remove_object(): Remove an object from Praat's object list by name.

Usage:

```
PraatInterpreter$remove_object(name)
```

Arguments:

name Object name.

Returns: Self (invisibly).

PraatInterpreter\$remove_object_by_id(): Remove an object from Praat's object list by ID.

Usage:

```
PraatInterpreter$remove_object_by_id(id)
```

Arguments:

id Object ID.

Returns: Self (invisibly).

PraatInterpreter\$select_object(): Select an object in Praat's object list.

Usage:

```
PraatInterpreter$select_object(name, add = FALSE)
```

Arguments:

name Object name.

add If TRUE, add to the current selection.

Returns: Self (invisibly).

`PraatInterpreter$clear_objects()`: Clear all objects from Praat's object list.

Usage:

`PraatInterpreter$clear_objects()`

Returns: Self (invisibly).

`PraatInterpreter$print()`: Print a summary of the interpreter's current object list.

Usage:

`PraatInterpreter$print()`

Returns: Self (invisibly).

`PraatInterpreter$clone()`: The objects of this class are cloneable with this method.

Usage:

`PraatInterpreter$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

[Sound](#), [Pitch](#) for Praat object classes

Examples

```
# Create interpreter
interp <- PraatInterpreter$new()

# Execute script
interp$run("
  Create Sound as pure tone: \"tone\", 1, 0, 1, 44100, 440, 0.2, 0.01, 0.01
  pitch = To Pitch: 0, 75, 600
")

# Get objects
interp$list_objects()
sound <- interp$get_object("tone")

# Set and get variables
interp$set_variable("freq", 440)
interp$get_variable("freq")

# Evaluate expressions
interp$eval("sqrt(2)")
```

print.fast_vector	<i>Print Method for Fast-Access Vectors</i>
-------------------	---

Description

Print Method for Fast-Access Vectors

Usage

```
## S3 method for class 'fast_vector'  
print(x, ...)
```

Arguments

x	A fast_vector
...	Additional arguments passed to print

Value

x, invisibly.

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.2, sampling_rate =  
  16000)  
samples <- get_sound_values_fast(sound, channel = 1)  
print(samples)
```

print.praat_formant	<i>Print method for praat_formant objects</i>
---------------------	---

Description

Print method for praat_formant objects

Usage

```
## S3 method for class 'praat_formant'  
print(x, ...)
```

Arguments

x	A praat_formant object
...	Additional arguments (unused)

Value

x, invisibly.

Examples

```
x <- list(
  n_frames = 2, n_formants = 1, time_step = 0.01,
  max_formant = 5000, window_length = 0.025,
  values = data.frame(
    time = c(0.1, 0.2), formant_number = c(1, 1),
    frequency = c(500, 520), bandwidth = c(80, 82)
  )
)
class(x) <- "praat_formant"
print(x)
```

`print.praat_intensity` *Print method for praat_intensity objects*

Description

Print method for praat_intensity objects

Usage

```
## S3 method for class 'praat_intensity'
print(x, ...)
```

Arguments

<i>x</i>	A praat_intensity object
<i>...</i>	Additional arguments (unused)

Value

x, invisibly.

Examples

```
x <- list(
  n_frames = 2, time_step = 0.01, minimum_pitch = 100,
  window_length = 0.032, subtract_mean = TRUE,
  values = data.frame(time = c(0.1, 0.2), intensity_db = c(65.2, 66.1))
)
class(x) <- "praat_intensity"
print(x)
```

print.praat_pitch	<i>Print method for praat_pitch objects</i>
-------------------	---

Description

Provides a concise display of a pitch contour.

Usage

```
## S3 method for class 'praat_pitch'
print(x, ...)
```

Arguments

x	A praat_pitch object
...	Additional arguments passed to the underlying function or ignored.

Value

The object x, invisibly

Examples

```
x <- data.frame(time = c(0.1, 0.2, 0.3), frequency = c(120, 125, NA))
class(x) <- c("praat_pitch", "data.frame")
print(x)
```

print.praat_sound	<i>Print method for praat_sound objects</i>
-------------------	---

Description

Provides a concise, informative display of a praat_sound object.

Usage

```
## S3 method for class 'praat_sound'
print(x, ...)
```

Arguments

x	A praat_sound object
...	Additional arguments passed to the underlying function or ignored.

Value

The object x, invisibly

Examples

```
sound <- create_sound(rep(0, 1000), sampling_rate = 44100)
print(sound)
```

`print.zerocopy_vector` *Print method for legacy zerocopy_vector class (deprecated)*

Description

Print method for legacy zerocopy_vector class (deprecated)

Usage

```
## S3 method for class 'zerocopy_vector'
print(x, ...)
```

Arguments

x	A zerocopy_vector
...	Additional arguments

Value

x, invisibly.

Examples

```
# zerocopy_vector is a deprecated class name; fast_vector is current.
legacy_vec <- structure(c(0.1, 0.2, 0.3), class = c("zerocopy_vector",
  "numeric"))
print(legacy_vec)
```

process_sounds_parallel

Batch Process Sounds in Parallel

Description

Apply analysis to pre-loaded Sound objects in parallel. Use when sounds are already in memory.

Usage

```
process_sounds_parallel(
  sounds,
  analysis_func,
  n_cores = NULL,
  threads_per_worker = NULL,
  ...
)
```

Arguments

sounds	List of Sound objects or external pointers
analysis_func	Function. Analysis function to apply. Should accept a Sound object/pointer and return results.
n_cores	Integer. Number of CPU cores (default: auto)
threads_per_worker	Integer or 'NULL'. C++ threads each worker may use for Praat kernels. 'NULL' (default) auto-divides cores among workers to avoid oversubscription; set '1' to force single-threaded workers.
...	Additional arguments passed to analysis_func

Value

List of results

Examples

```
sounds <- list(
  Sound$create_tone(frequency = 220, duration = 0.2),
  Sound$create_tone(frequency = 440, duration = 0.2)
)

# Process using at most 2 cores (CRAN example policy)
results <- process_sounds_parallel(sounds, function(s) {
  pitch <- s$to_pitch()
  pitch$get_mean(0, 0, "hertz")
}, n_cores = 2)
```

read_sound	<i>Read sound from audio file (DEPRECATED)</i>
------------	--

Description

****DEPRECATED:**** This S3 function is deprecated. Use the R6 interface instead: Sound\$new(file_path)

Usage

```
read_sound(file_path, channel = 0)
```

Arguments

file_path	Path to audio file (WAV/AIFF/FLAC/MP3 via Praat, others via av fallback)
channel	Channel to read (0 = left, 1 = right) - ignored in R6

Value

Sound R6 object

Examples

```
tmp <- tempfile(fileext = ".wav")
Sound$create_tone(frequency = 440, duration = 0.2)$save(tmp)

# Old S3 approach (DEPRECATED)
sound <- read_sound(tmp)

# New R6 approach (RECOMMENDED)
sound <- Sound$new(tmp)
unlink(tmp)
```

set_textgrid_simd_enabled_bridge	<i>Enable/Disable SIMD for TextGrid Operations</i>
----------------------------------	--

Description

Enable/Disable SIMD for TextGrid Operations

Usage

```
set_textgrid_simd_enabled_bridge(enabled)
```

Arguments

enabled Logical, TRUE to enable SIMD, FALSE for scalar

Value

Invisibly returns NULL.

Examples

```
set_textgrid_simd_enabled_bridge(TRUE)
textgrid_simd_enabled()
set_textgrid_simd_enabled_bridge(FALSE)
```

simd_info	<i>Get SIMD Capabilities</i>
-----------	------------------------------

Description

Reports the SIMD (Single Instruction Multiple Data) capabilities available in the current installation, for diagnostics and debugging.

Usage

```
simd_info()
```

Details

The pladdrr package uses SIMD acceleration for computationally intensive operations like autocorrelation, windowing, and statistical computations. The effect varies by code path and platform; use [pladdrr_simd()] to toggle SIMD at runtime and compare a given workload directly.

Common SIMD instruction sets:

- **AVX2**: 256-bit vectors (4 doubles or 8 floats) - Intel/AMD x86_64
- **SSE4.2**: 128-bit vectors (2 doubles or 4 floats) - Older x86_64
- **NEON**: 128-bit vectors (2 doubles or 4 floats) - ARM (Apple Silicon)

Set options(pladdrr.use_simd = FALSE) before loading the package to start in scalar mode, or use [pladdrr_simd()] after load for runtime A/B checks. The option is read during .onLoad.

Value

A list with the following components:

- enabled** Logical indicating if SIMD is currently enabled
- available** Logical indicating if SIMD support was compiled in
- architecture** Character string describing the SIMD instruction set (e.g., "AVX2", "SSE4.2", "NEON")
- batch_size_double** Integer number of doubles processed per SIMD operation

batch_size_float Integer number of floats processed per SIMD operation

version Character string describing the SIMD library in use

debug_build Logical indicating the shared object was compiled without NDEBUG, i.e. by `devtools::load_all()/pkgbuild::compile_dll()`, which force `-UNDEBUG -Wall -pedantic -g -O0`. Such a build is not representative of normal operation and must not be used for timing comparisons; reinstall with `R CMD INSTALL --preclean .` to get optimised objects.

Examples

```
# Check SIMD capabilities
info <- simd_info()
print(info)

if (info$architecture == "AVX2") {
  message("AVX2 SIMD support detected")
}

# Disable SIMD temporarily for testing
pladdrr_simd(FALSE)
# ... run tests ...
pladdrr_simd(TRUE)
```

Sound

Sound

Description

Represents a digitized acoustic signal (a Praat Sound object).

Arguments

<code>path</code>	Path to an audio file. See the File I/O section for supported formats.
<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ Sound object; set internally when a method returns a new Sound.

Details

A Sound holds one or more channels of audio sampled at regular intervals. It is the starting point for most acoustic analyses in `pladdrr`: pitch, formants, intensity, and spectral measures all begin from a Sound.

Value

A Sound object.

File I/O

The constructor tries two readers in order:

1. The native Praat reader, which handles WAV, AIFF, AIFC, NIST, and NeXT/Sun files without extra packages, and is the fastest path.
2. The av package, used as a fallback for everything else, including FLAC, MP3, and OGG Vorbis. The FLAC and MP3 decoder sources were dropped from the vendored Praat tree in v4.9.5 to keep the CRAN tarball within limits, so those formats need av installed (`install.packages("av")`). Without it, reading a FLAC or MP3 file raises an error naming the missing package.

Convert to WAV up front if you want to avoid the av dependency entirely.

Usage

```
# From file
sound <- Sound(path = "audio.wav")

# From numeric data
sound <- Sound$from_values(values, sampling_rate = 44100)

# Synthetic tone
sound <- Sound$create_tone(frequency = 440, duration = 1.0)
```

Query methods

- `get_duration()` - duration in seconds
- `get_sampling_frequency()` - sampling rate in Hz
- `get_number_of_samples()` - number of samples
- `get_number_of_channels()` - number of channels
- `get_value_at_time()` - amplitude at a given time
- `get_rms()`, `get_energy()`, `get_power()` - energy measures
- `get_intensity_db()` - intensity in dB
- `get_minimum()`, `get_maximum()`, `get_mean()` - amplitude statistics
- `get_values(channel)` - sample values as a numeric vector
- `get_sample_times()` - sample times as a numeric vector

Analysis methods

- `to_pitch()` - extract pitch contour (F0)
- `to_formant_burg()` - extract formants (F1, F2, F3, ...)
- `to_intensity()` - extract intensity contour
- `to_harmonicity_cc()` - harmonics-to-noise ratio
- `to_harmonicity_gne()` - glottal-to-noise excitation ratio (GNE)
- `extract_electroglottogram(channel, invert)` - extract an electroglottogram (EGG) from a channel

- `to_spectrum()` - frequency spectrum
- `to_spectrogram()` - time-frequency representation
- `to_ltas()` - long-term average spectrum
- `to_ltas_pitch_corrected()` - pitch-corrected LTAS (voice quality)
- `to_formant_robust()` - outlier-resistant formant tracking
- `to_mel_spectrogram()` - mel-scale spectrogram
- `to_bark_spectrogram()` - Bark-scale spectrogram
- `to_point_process_periodic_cc()` - extract glottal pulses

Signal processing

- `lengthen()` - time-stretch using overlap-add
- `autocorrelate()` - autocorrelation function
- `convolve()` - convolve with another sound
- `cross_correlate()` - cross-correlate with another sound
- `deepen_band_modulation()` - hearing enhancement
- `filter_by_formant()` - filter with a Formant object
- `filter_by_formant_noscale()` - filter without scaling

Extraction

- `extract_channel()` - extract a single channel
- `extract_part(from, to, window_shape, relative_width, preserve_times)` - extract a time range, with optional windowing. Supports 12 window shapes (rectangular, triangular, parabolic, hanning, hamming, gaussian1-5, kaiser1-2); see https://www.fon.hum.uva.nl/praat/manual/Sound__Extract_part____.html.

Modification

- `scale_intensity()` - scale to a target dB level (in place)
- `scale_peak()` - scale peak amplitude (in place)
- `pre_emphasize()` - high-pass filter (in place)
- `de_emphasize()` - low-pass filter (in place)
- `resample()` - resample to a different rate (returns a new object)
- `convert_to_mono()` - average channels to mono (returns a new object)
- `concatenate()` - append another sound (returns a new object)
- `mix()` - mix with another sound (returns a new object)

Export

- `as_matrix()` - export as a numeric matrix
- `as_data_frame()` - export as a `data.frame`
- `save()` - save to an audio file

See Also

[Pitch](#), [Formant](#), [Intensity](#), [Spectrum](#)

Examples

```
# Synthetic tone, no external file needed
sound <- Sound$create_tone(frequency = 440, duration = 1.0)
pitch <- sound$to_pitch()
formants <- sound$to_formant_burg()

cat("Duration:", sound$get_duration(), "s\n")
cat("Sample rate:", sound$get_sampling_frequency(), "Hz\n")

part <- sound$extract_part(0.2, 0.5)
```

sound_as_matrix_fast *Convert Sound to Matrix (Fast Copy)*

Description

Copies Sound data into a matrix (samples x channels) via direct pointer access.

Usage

```
sound_as_matrix_fast(sound, zerocopy = FALSE)

# Deprecated: use sound_as_matrix_fast() instead
```

Arguments

sound	A Sound object
zerocopy	Ignored (kept for backward compatibility). All paths copy.

Value

Numeric matrix (samples x channels)

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5, sampling_rate =
  16000)
mat <- sound_as_matrix_fast(sound)
```

sound_auto_correlate *Auto-correlate a sound with itself*

Description

Auto-correlate a sound with itself

Usage

```
sound_auto_correlate(sound, scaling = 4L, signal_outside = 1L)
```

Arguments

sound Sound object
 scaling Scaling: 1=integral, 2=sum, 3=normalize, 4=peak_0.99
 signal_outside Signal outside time domain: 1=zero, 2=similar

Value

New Sound object (auto-correlation function)

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)
ac <- sound_auto_correlate(sound)
```

sound_concatenate_all *Concatenate Multiple Sounds in Single C++ Call*

Description

Concatenates a list of Sound objects at the C++ level, avoiding the $O(n)$ $R \rightarrow C$ boundary crossings that occur with 'Reduce(function(a,b) a\$concatenate(b), sounds)'.

Usage

```
sound_concatenate_all(sounds, overlap = 0, return_r6 = TRUE)
```

Arguments

sounds List of Sound objects (R6) or external pointers
 overlap Numeric. Overlap duration in seconds (default: 0)
 return_r6 Logical. Return R6 Sound object (TRUE) or raw xptr (FALSE)

Value

Sound object (R6 or xptr depending on return_r6)

Examples

```
sound_list <- list(
  Sound$create_tone(frequency = 440, duration = 0.2),
  Sound$create_tone(frequency = 880, duration = 0.2)
)
result <- sound_concatenate_all(sound_list)
```

sound_create_pure_tone

Create a pure tone with fade in/out

Description

Creates a sinusoidal pure tone with optional fade in/out envelopes.

Usage

```
sound_create_pure_tone(
  frequency = 440,
  duration = 1,
  sampling_rate = 44100,
  amplitude = 0.99,
  fade_in_duration = 0.01,
  fade_out_duration = 0.01,
  channels = 1L
)
```

Arguments

frequency	Frequency in Hz (default: 440)
duration	Duration in seconds (default: 1.0)
sampling_rate	Sampling rate in Hz (default: 44100)
amplitude	Peak amplitude (default: 0.99)
fade_in_duration	Fade-in duration in seconds (default: 0.01)
fade_out_duration	Fade-out duration in seconds (default: 0.01)
channels	Number of channels (default: 1)

Value

A Sound object

Examples

```
tone <- sound_create_pure_tone(frequency = 440, duration = 0.5)
tone <- Sound$create_pure_tone(frequency = 880, fade_in_duration = 0.05)
```

sound_create_tone	<i>Create a pure tone Sound</i>
-------------------	---------------------------------

Description

Factory function to generate a pure sine wave tone. Use ‘Sound\$create_tone()’ for backward compatibility (calls this function).

Usage

```
sound_create_tone(
  duration = 1,
  sampling_rate = 44100,
  frequency = 440,
  amplitude = 0.99
)
```

Arguments

duration	Duration in seconds (default: 1.0)
sampling_rate	Sampling rate in Hz (default: 44100)
frequency	Frequency in Hz (default: 440)
amplitude	Amplitude 0-1 (default: 0.99)

Value

A Sound object

Examples

```
# Create 440 Hz tone
sound <- sound_create_tone(frequency = 440, duration = 1.0)

# Or using Sound$create_tone() (same thing)
sound <- Sound$create_tone(frequency = 440, duration = 1.0)
```

`sound_create_tone_complex`*Create a tone complex (harmonic series)*

Description

Creates a sound consisting of multiple sinusoids at equal frequency intervals.

Usage

```
sound_create_tone_complex(  
  frequency_step = 100,  
  duration = 1,  
  sampling_rate = 44100,  
  phase = c("sine", "cosine"),  
  first_frequency = 0,  
  ceiling = 0,  
  number_of_components = 0L  
)
```

Arguments

<code>frequency_step</code>	Step between harmonics in Hz (default: 100)
<code>duration</code>	Duration in seconds (default: 1.0)
<code>sampling_rate</code>	Sampling rate in Hz (default: 44100)
<code>phase</code>	Phase type: "sine" or "cosine" (default: "sine")
<code>first_frequency</code>	Lowest component frequency in Hz (default: 0, uses <code>frequency_step</code>)
<code>ceiling</code>	Maximum frequency to include in Hz (default: Nyquist)
<code>number_of_components</code>	Number of components (default: 0 = all up to ceiling)

Value

A Sound object

Examples

```
# Harmonic tone with 10 components at 100 Hz intervals  
tone <- sound_create_tone_complex(frequency_step = 100, number_of_components  
  = 10)  
tone <- Sound$create_tone_complex(frequency_step = 200, phase = "cosine")
```

`sound_deepen_band_modulation`*Deepen band modulation (hearing enhancement)*

Description

Deepen band modulation (hearing enhancement)

Usage

```
sound_deepen_band_modulation(  
  sound,  
  enhancement_db = 10,  
  flow = 300,  
  fhigh = 4000,  
  slow_modulation = 3,  
  fast_modulation = 30,  
  band_smoothing = 100  
)
```

Arguments

<code>sound</code>	Sound object
<code>enhancement_db</code>	Enhancement in dB
<code>flow</code>	Low frequency bound (Hz)
<code>fhigh</code>	High frequency bound (Hz)
<code>slow_modulation</code>	Slow modulation frequency (Hz)
<code>fast_modulation</code>	Fast modulation frequency (Hz)
<code>band_smoothing</code>	Band smoothing (Hz)

Value

New Sound object

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1.0)  
enhanced <- sound_deepen_band_modulation(sound, enhancement_db = 10)
```

sound_extract_and_formant

Extract Parts and Analyze Formants in Single C++ Call

Description

Extract Parts and Analyze Formants in Single C++ Call

Usage

```
sound_extract_and_formant(
  sound,
  from_times,
  to_times,
  time_step = 0.005,
  max_formants = 5,
  max_frequency = 5500,
  window_length = 0.025,
  pre_emphasis_from = 50,
  return_r6 = TRUE
)
```

Arguments

sound	Sound object (R6) or external pointer
from_times	Numeric vector of start times
to_times	Numeric vector of end times
time_step	Numeric. Formant time step (default: 0.005)
max_formants	Numeric. Maximum number of formants (default: 5)
max_frequency	Numeric. Maximum frequency (default: 5500)
window_length	Numeric. Window length (default: 0.025)
pre_emphasis_from	Numeric. Pre-emphasis frequency (default: 50)
return_r6	Logical. Return R6 Formant objects (TRUE) or raw xptrs (FALSE)

Value

List of Formant objects (R6 or xptr depending on return_r6)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 2.0)
from_times <- c(0.2, 1.0)
to_times <- c(0.6, 1.4)
formants <- sound_extract_and_formant(sound, from_times, to_times)
```

 sound_extract_and_pitch

Extract Parts and Analyze Pitch in Single C++ Call

Description

Combines interval extraction and pitch analysis in a single C++ call, avoiding intermediate R6 object creation, as an alternative to extracting each part and calling ‘\$to_pitch()’ on it in a loop.

Usage

```
sound_extract_and_pitch(
  sound,
  from_times,
  to_times,
  time_step = 0,
  pitch_floor = 75,
  pitch_ceiling = 600,
  return_r6 = TRUE
)
```

Arguments

sound	Sound object (R6) or external pointer
from_times	Numeric vector of start times
to_times	Numeric vector of end times
time_step	Numeric. Pitch time step (0 = automatic)
pitch_floor	Numeric. Pitch floor in Hz (default: 75)
pitch_ceiling	Numeric. Pitch ceiling in Hz (default: 600)
return_r6	Logical. Return R6 Pitch objects (TRUE) or raw xptrs (FALSE)

Value

List of Pitch objects (R6 or xptr depending on return_r6)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 2.0)
from_times <- c(0.2, 1.0)
to_times <- c(0.6, 1.4)
pitches <- sound_extract_and_pitch(sound, from_times, to_times)
```

sound_extract_part	<i>Extract part of Sound with optional windowing</i>
--------------------	--

Description

Extracts time range from Sound, applying optional window function. Implements Praat's "Sound: Extract part..." with full window shape support.

Usage

```
sound_extract_part(
    sound,
    t1,
    t2,
    window_shape = 1L,
    relative_width = 1,
    preserve_times = FALSE
)
```

Arguments

sound	Sound object
t1	Start time (seconds)
t2	End time (seconds)
window_shape	Window shape code or name: * 0 or "rectangular" - Rectangular (no tapering) * 1 or "triangular" - Triangular (Bartlett) * 2 or "parabolic" - Parabolic (Welch) * 3 or "hanning" - Hanning * 4 or "hamming" - Hamming * 5 or "gaussian1" - Gaussian with sd=0.42466 (relative to duration) * 6 or "gaussian2" - Gaussian with sd=0.21233 (narrower, use relative_width=2.0) * 7 or "gaussian3" - Gaussian with sd=0.14155 (use relative_width=3.0) * 8 or "gaussian4" - Gaussian with sd=0.10616 (use relative_width=4.0) * 9 or "gaussian5" - Gaussian with sd=0.08493 (use relative_width=5.0) * 10 or "kaiser1" - Kaiser-Bessel with alpha=20.7 * 11 or "kaiser2" - Kaiser-Bessel with alpha=40.5 (use relative_width=2.0)
relative_width	Relative width for windowing (default: 1.0). For gaussian2/kaiser2, use 2.0 to maintain effective window duration. For gaussian3, use 3.0. For gaussian4, use 4.0. For gaussian5, use 5.0. This extends physical extraction beyond [t1,t2] while keeping effective duration.
preserve_times	If TRUE, preserve original time domain (result spans t1 to t2). If FALSE, time-shift result to start at 0.

Details

Window shapes with higher numbers (gaussian2-5, kaiser2) have narrower effective windows. To maintain comparable effective duration to gaussian1/kaiser1, use relative_width > 1.0, which extracts a longer physical segment while applying a more aggressive taper.

For spectral analysis, Kaiser2 and Gaussian2 with `relative_width=2.0` are commonly used (e.g., in Praat's "To Spectrogram..." and "To Pitch (ac)... Very accurate").

Value

New Sound object with extracted and windowed portion

References

Praat documentation: https://www.fon.hum.uva.nl/praat/manual/Sound__Extract_part____.html

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 3.0)

# Rectangular window (no tapering)
rect <- sound_extract_part(sound, 1.0, 2.0, window_shape = 0L)

# Gaussian1 window (standard)
gauss1 <- sound_extract_part(sound, 1.0, 2.0, window_shape = 5L,
  relative_width = 1.0)

# Gaussian2 with wider physical extraction
gauss2 <- sound_extract_part(sound, 1.0, 2.0, window_shape = 6L,
  relative_width = 2.0)

# Kaiser2 for spectral analysis
kaiser <- sound_extract_part(sound, 1.0, 2.0, window_shape = 11L,
  relative_width = 2.0)
```

sound_extract_parts *Extract Multiple Parts from a Sound*

Description

Extracts multiple time intervals from a Sound object and returns them as a list of Sound objects. Useful for extracting voiced segments identified by voice activity detection.

Usage

```
sound_extract_parts(
  sound,
  start_times,
  end_times,
  window_shape = "rectangular",
  relative_width = 1,
  preserve_times = FALSE,
  return_r6 = TRUE
)
```


Arguments

sound	Sound object
start_times	Numeric vector of interval start times (seconds)
end_times	Numeric vector of interval end times (seconds)
window_shape	Character. Window shape for extraction (default: "rectangular"). See details for all options.
relative_width	Numeric. Relative width of window (default: 1.0). For gaussian2/kaiser2, use 2.0. For gaussian3-5, use 3.0-5.0 respectively.
preserve_times	Logical. Preserve original time stamps (default: FALSE)
return_r6	Logical. Return R6 Sound objects (TRUE) or raw xptrs (FALSE). Using FALSE skips R6 wrapper construction.

Details

This function is vectorized to extract multiple intervals efficiently. Each extracted sound can then be concatenated or analyzed separately.

Available window shapes (see Praat manual for details): - "rectangular" (default) - No tapering - "triangular" - Triangular (Bartlett) taper - "parabolic" - Parabolic (Welch) taper - "hanning" - Hanning window - "hamming" - Hamming window - "gaussian1" - Gaussian window (sd=0.42466) - "gaussian2" - Narrower Gaussian (sd=0.21233), use relative_width=2.0 - "gaussian3" - Even narrower (sd=0.14155), use relative_width=3.0 - "gaussian4" - Very narrow (sd=0.10616), use relative_width=4.0 - "gaussian5" - Extremely narrow (sd=0.08493), use relative_width=5.0 - "kaiser1" - Kaiser-Bessel window (alpha=20.7) - "kaiser2" - Narrower Kaiser-Bessel (alpha=40.5), use relative_width=2.0

Value

List of Sound objects, one for each extracted interval

References

Praat documentation: https://www.fon.hum.uva.nl/praat/manual/Sound__Extract_part____.html

Examples

```
sound <- sounds_append(
  Sound$create_tone(frequency = 200, duration = 0.5, amplitude = 0.8),
  Sound$create_tone(frequency = 200, duration = 0.3, amplitude = 0.001)
)
vad_grid <- sound_to_textgrid_silences(sound)
voiced_intervals <- textgrid_get_intervals_where(vad_grid, 1, "equals",
  "sounding")

voiced_sounds <- sound_extract_parts(
  sound,
  voiced_intervals$xmin,
  voiced_intervals$xmax
```

```

)

# Analyze each segment separately
for (i in seq_along(voiced_sounds)) {
  cat("Segment", i, "duration:", voiced_sounds[[i]]$get_total_duration(),
      "\n")
}

```

sound_extract_parts_pooled

Extract multiple Sound parts using object pool

Description

Batch extraction using pooled memory reuse, for large numbers of segments.

Usage

```
sound_extract_parts_pooled(sound_xptr, start_times, end_times, use_pool = TRUE)
```

Arguments

sound_xptr	External pointer to source Sound
start_times	Numeric vector of start times
end_times	Numeric vector of end times
use_pool	Whether to use object pool (default TRUE)

Details

When use_pool is TRUE, Sound objects are acquired from a pool and should be released back with sound_pool_release() when done.

****Important:**** Pool-acquired Sounds should not be modified as they may be reused. Copy if modification is needed.

Value

List of external pointers to extracted Sound segments

Examples

```

sound <- Sound$create_tone(frequency = 200, duration = 2.0)
starts <- c(0.1, 0.5, 1.0)
ends <- c(0.3, 0.7, 1.2)

# Batch extraction with pooling
segments <- sound_extract_parts_pooled(sound$.xptr, starts, ends)

```

```
# Release back to pool when done
for (seg in segments) {
  pladdr:::sound_pool_release(seg)
}
```

`sound_filter_pass_hann_band`*Apply Hann band-pass filter*

Description

Apply Hann band-pass filter

Usage

```
sound_filter_pass_hann_band(sound, fmin, fmax, smooth = 100)
```

Arguments

<code>sound</code>	Sound object
<code>fmin</code>	Low frequency cutoff (Hz)
<code>fmax</code>	High frequency cutoff (Hz)
<code>smooth</code>	Smoothing bandwidth (Hz)

Value

New Sound object

Examples

```
sound <- Sound$create_tone(frequency = 1000, duration = 0.5)
filtered <- sound_filter_pass_hann_band(sound, fmin = 300, fmax = 3000,
  smooth = 100)
```

sound_filter_stop_hann_band	<i>Apply Hann band-stop filter</i>
-----------------------------	------------------------------------

Description

Apply Hann band-stop filter

Usage

```
sound_filter_stop_hann_band(sound, fmin, fmax, smooth = 100)
```

Arguments

sound	Sound object
fmin	Low frequency cutoff (Hz)
fmax	High frequency cutoff (Hz)
smooth	Smoothing bandwidth (Hz)

Value

New Sound object

Examples

```
sound <- Sound$create_tone(frequency = 1000, duration = 0.5)
filtered <- sound_filter_stop_hann_band(sound, fmin = 300, fmax = 3000,
smooth = 100)
```

sound_from_values	<i>Create Sound from numeric values</i>
-------------------	---

Description

Factory function to create a Sound object from a numeric vector or matrix. Use ‘Sound\$from_values()’ for backward compatibility (calls this function).

Usage

```
sound_from_values(values, sampling_rate = 44100, start_time = 0)
```

Arguments

values	Numeric matrix with channels as rows, samples as columns (or vector for mono)
sampling_rate	Sampling rate in Hz (default: 44100)
start_time	Start time of the sound in seconds (default: 0.0)

Value

A Sound object

Examples

```
# Create from vector (mono)
values <- sin(2 * pi * 440 * seq(0, 1, length.out = 4410))
sound <- sound_from_values(values, 4410)

# Or using Sound$from_values() (same thing)
sound <- Sound$from_values(values, 4410)
```

sound_get_zcr	<i>Calculate Zero Crossing Rate for Sound</i>
---------------	---

Description

Calculates Zero Crossing Rate (ZCR) per frame using Praat's built-in zero crossing detection. ZCR is the rate at which the signal changes sign, useful for distinguishing voiced (low ZCR) from unvoiced (high ZCR) speech.

Usage

```
sound_get_zcr(
  sound,
  window_duration = 0.03,
  hop_duration = 0.01,
  channel = 1L,
  avqi_compatible = TRUE
)
```

Arguments

sound	Sound object
window_duration	Numeric. Window duration in seconds (default: 0.03)
hop_duration	Numeric. Hop between windows in seconds (default: 0.01)
channel	Integer. Channel to analyze for stereo (default: 1)
avqi_compatible	Logical. Use AVQI-compatible analysis window (0.0025-0.0275s within each frame) instead of full frame (default: TRUE)

Details

Uses Praat’s ‘to_point_process_zeros()’ for accurate zero crossing detection with interpolation.

When ‘avqi_compatible = TRUE’ (default), uses AVQI203.praat’s checkZeros procedure: analyzes zeros within 0.0025-0.0275s of each frame (25ms analysis window within 30ms frame). This matches Praat’s AVQI implementation.

Typical ZCR values: - Voiced speech: 500-2000 crossings/second - Unvoiced speech: 3000-6000 crossings/second - Silence: variable, depends on noise

For AVQI, segments with ZCR > 3000 Hz are typically rejected as unvoiced.

Value

Named list with: - ‘times’: Numeric vector of frame center times - ‘zcr’: Numeric vector of zero crossing rates (crossings per second) - ‘window_duration’: Window duration used - ‘hop_duration’: Hop duration used

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1.0)
zcr_data <- sound_get_zcr(sound, window_duration = 0.03)
str(zcr_data)
```

sound_lengthen	<i>Time-stretch a sound using overlap-add</i>
----------------	---

Description

Time-stretch a sound using overlap-add

Usage

```
sound_lengthen(sound, fmin = 75, fmax = 600, factor = 1.5)
```

Arguments

sound	Sound object
fmin	Minimum pitch (Hz)
fmax	Maximum pitch (Hz)
factor	Lengthening factor (>1 = slower, <1 = faster)

Value

New Sound object

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 1.0)
slower <- sound_lengthen(sound, fmin = 75, fmax = 600, factor = 1.5)
faster <- sound_lengthen(sound, fmin = 75, fmax = 600, factor = 0.8)

```

sound_load_window	<i>Load Sound Window from File with Optional Resampling</i>
-------------------	---

Description

Extracts a time window from a sound file without loading the entire file. Optionally resamples the window to a target sampling rate. Uses LongSound for lazy loading - only the requested window is loaded from disk.

Usage

```
sound_load_window(path, start, end, resample_to = NULL, preserve_times = FALSE)
```

Arguments

path	Path to sound file (WAV, AIFF, FLAC, MP3, etc.)
start	Start time of window in seconds
end	End time of window in seconds
resample_to	Target sampling rate in Hz (optional). If NULL, no resampling.
preserve_times	If TRUE, keep original time domain. If FALSE, shift to start at 0 (default: FALSE)

Details

A traditional workflow loads the entire file, resamples the entire file, and then extracts the window of interest — reading and processing far more samples than needed. This function instead opens the file as a ‘LongSound’ (lazily, reading only the header), extracts just the requested window from disk, and resamples only that window.

****Use cases:**** - Pharyngeal analysis: extracting short vowel windows from long recordings - Formant tracking: analyzing specific time points - Batch extraction: processing windows from multiple files - Large file processing: working with hours-long recordings

Value

Sound object containing the windowed (and optionally resampled) audio

See Also

[Sound](#), [LongSound](#)

Other batch-ops: [textgrid_merge\(\)](#)

Examples

```
# Write a short synthetic recording, then load only part of it
sound <- Sound$create_tone(frequency = 220, duration = 2.0)
path <- tempfile(fileext = ".wav")
sound$save(path)

window <- sound_load_window(path, start = 0.5, end = 0.6)

# Extract and resample (for spectral analysis)
window_10k <- sound_load_window(path, start = 0.5, end = 0.6, resample_to =
  10000)

# Preserve original time domain (window starts at 0.5, not 0.0)
window_timed <- sound_load_window(path, start = 0.5, end = 0.6,
  preserve_times = TRUE)

unlink(path)
```

sound_max

Compute maximum amplitude

Description

Finds the maximum amplitude value in a sound object. Works with both S3 `praat_sound` objects and R6 `Sound` objects.

Usage

```
sound_max(sound)
```

Arguments

sound A `praat_sound` (S3) or `Sound` (R6) object

Value

Maximum amplitude (numeric scalar)

Examples

```
sound <- Sound$from_values(c(0.5, -0.8, 1.0), sampling_rate = 1000)
sound_max(sound) # Returns 1.0
```

sound_mean	<i>Compute mean amplitude</i>
------------	-------------------------------

Description

Calculates the mean (average) of all amplitude values in a sound object. Works with both S3 `praat_sound` objects and R6 `Sound` objects.

Usage

```
sound_mean(sound)
```

Arguments

sound	A <code>praat_sound</code> (S3) or <code>Sound</code> (R6) object
-------	---

Value

Mean amplitude (numeric scalar)

Examples

```
sound <- Sound$from_values(c(-1, 0, 1), sampling_rate = 1000)
sound_mean(sound) # Returns 0
```

sound_min	<i>Compute minimum amplitude</i>
-----------	----------------------------------

Description

Finds the minimum amplitude value in a sound object. Works with both S3 `praat_sound` objects and R6 `Sound` objects.

Usage

```
sound_min(sound)
```

Arguments

sound	A <code>praat_sound</code> (S3) or <code>Sound</code> (R6) object
-------	---

Value

Minimum amplitude (numeric scalar)

Examples

```
sound <- Sound$from_values(c(0.5, -0.8, 0.2), sampling_rate = 1000)
sound_min(sound) # Returns -0.8
```

sound_pool

Sound Object Pool for Batch Processing

Description

Memory optimization for batch operations that extract many Sound segments. Reuses Sound object allocations instead of creating/destroying each time.

****Numerical Impact:**** None - output is identical to non-pooled version

Usage

```
sound_pool_stats()

sound_pool_clear()

sound_pool_resize(max_size)
```

Arguments

max_size Maximum number of Sound objects to keep in pool

Details

The pool automatically manages Sound object reuse: - 'sound_pool_acquire()' - get a Sound from pool (or create new) - 'sound_pool_release()' - return Sound to pool for reuse - 'sound_pool_stats()' - get hit/miss statistics - 'sound_pool_clear()' - clear the pool - 'sound_pool_resize()' - change pool capacity

Value

sound_pool_stats() returns a named list with elements hits, misses, hit_rate, pool_size, and in_use.

Invisibly returns NULL.

Invisibly returns NULL.

Examples

```
# Pool is used automatically by batch extraction functions
# For manual control:

# Check pool statistics
stats <- sound_pool_stats()
stats$hits
stats$misses

# Clear pool to free memory
sound_pool_clear()
```

sound_rms

Compute RMS (root mean square) amplitude

Description

Calculates the RMS amplitude of a sound object. RMS is a measure of the signal's power and is computed as $\sqrt{\text{mean}(x^2)}$.

Usage

```
sound_rms(sound)
```

Arguments

sound A `praat_sound` object

Details

For a sine wave with amplitude A , the RMS value is $A/\sqrt{2} \sim 0.707 \cdot A$. RMS is useful for comparing signal levels and measuring acoustic intensity.

Value

RMS amplitude (numeric scalar)

Examples

```
# RMS of a sine wave
sine <- generate_sine_wave(440, 1.0, amplitude = 1.0)
sound_rms(sine) # Approximately 0.707
```

sound_statistics	<i>Compute comprehensive sound statistics</i>
------------------	---

Description

Calculates a comprehensive set of statistics for a sound object, including amplitude statistics and metadata. Works with both S3 `praat_sound` objects and R6 `Sound` objects.

Usage

```
sound_statistics(sound)
```

Arguments

sound	A <code>praat_sound</code> (S3) or <code>Sound</code> (R6) object
-------	---

Value

A named list containing:

mean Mean amplitude

min Minimum amplitude

max Maximum amplitude

rms RMS amplitude

duration Duration in seconds

n_samples Number of samples

sampling_rate Sampling rate in Hz

Examples

```
sound <- generate_sine_wave(440, 0.5)
stats <- sound_statistics(sound)
print(stats)
```

sound_times_fast	<i>Fast Sound Time Vector</i>
------------------	-------------------------------

Description

Returns time values for each sample via optimized computation.

Usage

```
sound_times_fast(sound_xptr)
```

Arguments

sound_xptr	External pointer to Sound object
------------	----------------------------------

Details

Computes sample times directly from Sound metadata ($t_0 + i \cdot dt$) rather than going through Praat's accessor functions.

Value

Numeric vector of sample times

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.1)
times <- sound_times_fast(sound$.xptr)
head(times)
```

sound_to_formant_batch	<i>Extract Formants from Multiple Sounds in Single C++ Call</i>
------------------------	---

Description

Extract Formants from Multiple Sounds in Single C++ Call

Usage

```

sound_to_formant_batch(
  sounds,
  time_step = 0.005,
  max_formants = 5,
  max_frequency = 5500,
  window_length = 0.025,
  pre_emphasis_from = 50,
  return_r6 = TRUE
)

```

Arguments

sounds	List of Sound objects (R6) or external pointers
time_step	Numeric. Time step in seconds (default: 0.005)
max_formants	Numeric. Maximum number of formants (default: 5)
max_frequency	Numeric. Maximum frequency in Hz (default: 5500)
window_length	Numeric. Window length in seconds (default: 0.025)
pre_emphasis_from	Numeric. Pre-emphasis from frequency (default: 50)
return_r6	Logical. Return R6 Formant objects (TRUE) or raw xptrs (FALSE)

Value

List of Formant objects (R6 or xptr depending on return_r6)

Examples

```

sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
formants <- sound_to_formant_batch(sounds)

```

sound_to_intensity_batch

Extract Intensity from Multiple Sounds in Single C++ Call

Description

Extract Intensity from Multiple Sounds in Single C++ Call

Usage

```
sound_to_intensity_batch(
  sounds,
  minimum_pitch = 100,
  time_step = 0,
  subtract_mean = TRUE,
  return_r6 = TRUE
)
```

Arguments

sounds	List of Sound objects (R6) or external pointers
minimum_pitch	Numeric. Minimum pitch for analysis (default: 100)
time_step	Numeric. Time step (0 = automatic)
subtract_mean	Logical. Subtract mean pressure (default: TRUE)
return_r6	Logical. Return R6 Intensity objects (TRUE) or raw xptrs (FALSE)

Value

List of Intensity objects (R6 or xptr depending on return_r6)

Examples

```
sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
intensities <- sound_to_intensity_batch(sounds)
```

sound_to_pitch_ac_batch

Extract Pitch (AC) from Multiple Sounds in Single C++ Call

Description

Batch version of to_pitch_ac with full voicing parameters. Avoids $O(n)$ R→C boundary crossings for VUV analysis workflows.

Usage

```
sound_to_pitch_ac_batch(
  sounds,
  time_step = 0,
  pitch_floor = 75,
  pitch_ceiling = 600,
```

```

    max_candidates = 15L,
    very_accurate = FALSE,
    silence_threshold = 0.03,
    voicing_threshold = 0.45,
    octave_cost = 0.01,
    octave_jump_cost = 0.35,
    voiced_unvoiced_cost = 0.14,
    return_r6 = TRUE
  )

```

Arguments

<code>sounds</code>	List of Sound objects (R6) or external pointers
<code>time_step</code>	Numeric. Time step (0 = automatic)
<code>pitch_floor</code>	Numeric. Pitch floor in Hz (default: 75)
<code>pitch_ceiling</code>	Numeric. Pitch ceiling in Hz (default: 600)
<code>max_candidates</code>	Integer. Max candidates per frame (default: 15)
<code>very_accurate</code>	Logical. Use very accurate algorithm (default: FALSE)
<code>silence_threshold</code>	Numeric. Silence threshold (default: 0.03)
<code>voicing_threshold</code>	Numeric. Voicing threshold (default: 0.45)
<code>octave_cost</code>	Numeric. Octave cost (default: 0.01)
<code>octave_jump_cost</code>	Numeric. Octave jump cost (default: 0.35)
<code>voiced_unvoiced_cost</code>	Numeric. Voiced/unvoiced cost (default: 0.14)
<code>return_r6</code>	Logical. Return R6 Pitch objects (TRUE) or raw xptrs (FALSE)

Value

List of Pitch objects (R6 or xptr depending on `return_r6`)

Examples

```

sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
pitches <- sound_to_pitch_ac_batch(sounds)

```

sound_to_pitch_batch *Extract Pitch from Multiple Sounds in Single C++ Call*

Description

Processes multiple Sound objects and extracts Pitch at the C++ level, avoiding $O(n)$ R→C boundary crossings from calling ‘\$to_pitch()’ in a loop.

Usage

```
sound_to_pitch_batch(
  sounds,
  time_step = 0,
  pitch_floor = 75,
  pitch_ceiling = 600,
  return_r6 = TRUE
)
```

Arguments

sounds	List of Sound objects (R6) or external pointers
time_step	Numeric. Time step (0 = automatic)
pitch_floor	Numeric. Pitch floor in Hz (default: 75)
pitch_ceiling	Numeric. Pitch ceiling in Hz (default: 600)
return_r6	Logical. Return R6 Pitch objects (TRUE) or raw xptrs (FALSE)

Value

List of Pitch objects (R6 or xptr depending on return_r6)

Examples

```
sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
pitches <- sound_to_pitch_batch(sounds)
```

sound_to_pitch_cc_batch

Extract Pitch (CC) from Multiple Sounds in Single C++ Call

Description

Batch version of to_pitch_cc with full voicing parameters. Avoids $O(n)$ R→C boundary crossings for VUV analysis workflows. Use this when processing many sound segments in a loop.

Usage

```
sound_to_pitch_cc_batch(
    sounds,
    time_step = 0,
    pitch_floor = 75,
    pitch_ceiling = 600,
    max_candidates = 15L,
    very_accurate = FALSE,
    silence_threshold = 0.03,
    voicing_threshold = 0.45,
    octave_cost = 0.01,
    octave_jump_cost = 0.35,
    voiced_unvoiced_cost = 0.14,
    return_r6 = TRUE
)
```

Arguments

sounds	List of Sound objects (R6) or external pointers
time_step	Numeric. Time step (0 = automatic)
pitch_floor	Numeric. Pitch floor in Hz (default: 75)
pitch_ceiling	Numeric. Pitch ceiling in Hz (default: 600)
max_candidates	Integer. Max candidates per frame (default: 15)
very_accurate	Logical. Use very accurate algorithm (default: FALSE)
silence_threshold	Numeric. Silence threshold (default: 0.03)
voicing_threshold	Numeric. Voicing threshold (default: 0.45)
octave_cost	Numeric. Octave cost (default: 0.01)
octave_jump_cost	Numeric. Octave jump cost (default: 0.35)
voiced_unvoiced_cost	Numeric. Voiced/unvoiced cost (default: 0.14)
return_r6	Logical. Return R6 Pitch objects (TRUE) or raw xptrs (FALSE)

Value

List of Pitch objects (R6 or xptr depending on return_r6)

Examples

```
sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
# Instead of a loop (pitches <- lapply(sounds, function(s) s$to_pitch_cc())):
pitches <- sound_to_pitch_cc_batch(sounds)
```

sound_to_pitch_shs_batch

Extract Pitch (SHS) from Multiple Sounds

Description

Batch version of to_pitch_shs using Subharmonic Summation.

Usage

```
sound_to_pitch_shs_batch(
  sounds,
  time_step = 0.01,
  pitch_floor = 50,
  max_frequency = 1250,
  pitch_ceiling = 500,
  max_subharmonics = 15L,
  max_candidates = 15L,
  compression_factor = 0.84,
  n_points_per_octave = 48L,
  return_r6 = TRUE
)
```

Arguments

sounds	List of Sound objects (R6) or external pointers
time_step	Numeric. Time step (default: 0.01)
pitch_floor	Numeric. Pitch floor in Hz (default: 50)
max_frequency	Numeric. Maximum frequency in Hz (default: 1250)
pitch_ceiling	Numeric. Pitch ceiling in Hz (default: 500)
max_subharmonics	Integer. Number of subharmonics (default: 15)
max_candidates	Integer. Max candidates per frame (default: 15)

compression_factor Numeric. Compression factor (default: 0.84)
 n_points_per_octave Integer. Points per octave (default: 48)
 return_r6 Logical. Return R6 Pitch objects (TRUE) or raw xptrs (FALSE)

Value

List of Pitch objects (R6 or xptr depending on return_r6)

Examples

```
sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
pitches <- sound_to_pitch_shs_batch(sounds)
```

sound_to_pitch_spinet_batch

Extract Pitch (SPINET) from Multiple Sounds

Description

Batch version of to_pitch_spinet using spectral integration.

Usage

```
sound_to_pitch_spinet_batch(
  sounds,
  time_step = 0.005,
  window_duration = 0.04,
  min_frequency = 70,
  max_frequency = 5000,
  n_filters = 250L,
  pitch_ceiling = 500,
  max_candidates = 15L,
  return_r6 = TRUE
)
```

Arguments

sounds List of Sound objects (R6) or external pointers
 time_step Numeric. Time step (default: 0.005)
 window_duration Numeric. Analysis window (default: 0.04)

min_frequency	Numeric. Minimum frequency in Hz (default: 70)
max_frequency	Numeric. Maximum frequency in Hz (default: 5000)
n_filters	Integer. Number of gamma-tone filters (default: 250)
pitch_ceiling	Numeric. Pitch ceiling in Hz (default: 500)
max_candidates	Integer. Max candidates per frame (default: 15)
return_r6	Logical. Return R6 Pitch objects (TRUE) or raw xptrs (FALSE)

Value

List of Pitch objects (R6 or xptr depending on return_r6)

Examples

```
# The vendored Praat SPINET path has a rare, non-deterministic native
# flake ("all amplitudes equal to zero") unrelated to the input signal;
# tryCatch keeps this example from failing R CMD check when it strikes.
sounds <- list(
  Sound$create_tone(frequency = 150, duration = 0.5),
  Sound$create_tone(frequency = 200, duration = 0.5)
)
pitches <- tryCatch(sound_to_pitch_spinet_batch(sounds), error = function(e)
  NULL)
```

sound_to_textgrid_silences

Detect Silences in Sound and Create TextGrid

Description

Detects silent and sounding (voiced) intervals in a Sound object and returns a TextGrid with labeled intervals. This is essential for AVQI calculation, which requires extraction of voiced segments from continuous speech.

Usage

```
sound_to_textgrid_silences(
  sound,
  minimum_pitch = 100,
  time_step = 0,
  silence_threshold = -25,
  min_silent_interval = 0.1,
  min_sounding_interval = 0.1,
  silent_label = "silence",
  sounding_label = "sounding"
)
```

Arguments

sound	Sound object to analyze
minimum_pitch	Numeric. Minimum pitch for intensity calculation (Hz, default: 100)
time_step	Numeric. Time step for intensity calculation (s, default: 0.0 = auto)
silence_threshold	Numeric. Silence threshold in dB below maximum (default: -25)
min_silent_interval	Numeric. Minimum duration of silent interval (s, default: 0.1)
min_sounding_interval	Numeric. Minimum duration of sounding interval (s, default: 0.1)
silent_label	Character. Label for silent intervals (default: "silence")
sounding_label	Character. Label for sounding intervals (default: "sounding")

Details

The function works by: 1. Computing the intensity contour of the sound 2. Identifying regions where intensity falls below ‘silence_threshold’ dB relative to the maximum intensity 3. Merging nearby silent/sounding regions based on minimum duration criteria 4. Creating a TextGrid with labeled intervals

For AVQI, use these parameters (matching Praat AVQI script): - ‘minimum_pitch = 50’ - ‘time_step = 0.003’ - ‘silence_threshold = -25’ - ‘min_silent_interval = 0.1’ - ‘min_sounding_interval = 0.1’

Value

TextGrid object with one interval tier containing silent and sounding intervals

Examples

```
# Synthetic speech-like sound: loud / near-silent / loud
sound <- sounds_append(
  sounds_append(
    Sound$create_tone(frequency = 200, duration = 0.5, amplitude = 0.8),
    Sound$create_tone(frequency = 200, duration = 0.3, amplitude = 0.001)
  ),
  Sound$create_tone(frequency = 200, duration = 0.5, amplitude = 0.8)
)

# Create TextGrid with voice activity detection
vad_grid <- sound_to_textgrid_silences(
  sound,
  minimum_pitch = 50,
  time_step = 0.003,
  silence_threshold = -25,
  min_silent_interval = 0.1,
  min_sounding_interval = 0.1
)

# Extract voiced intervals
```

```
voiced_intervals <- textgrid_get_intervals_where(  
  vad_grid,  
  tier = 1,  
  condition = "equals",  
  text = "sounding"  
)  
  
# Extract voiced parts  
voiced_sounds <- sound_extract_parts(  
  sound,  
  voiced_intervals$xmin,  
  voiced_intervals$xmax,  
  window_shape = "rectangular",  
  relative_width = 1.0,  
  preserve_times = FALSE  
)
```

sound_values_fast	<i>Fast Sound Sample Access</i>
-------------------	---------------------------------

Description

Copies Sound sample data via direct pointer access into Praat's contiguous sample array, rather than going through the per-sample accessor in a loop.

Usage

```
sound_values_fast(sound_xptr, channel = 1L)
```

Arguments

sound_xptr	External pointer to Sound object
channel	Channel number (1-based, default 1)

Details

The returned vector is an independent R copy — safe to modify, store, or use after the Sound object is garbage collected.

Value

Numeric vector (independent copy of sample data). Has class `'c("fast_vector", "numeric")'` and a `'readonly'` attribute for backward compatibility with code that checked these.

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.5)

samples <- sound_values_fast(sound$.xptr, channel = 1)
rms <- sqrt(mean(samples^2))

# Regular copy - equivalent output
samples2 <- sound$get_values(channel = 1)
```

sounds_append	<i>Append two sounds with optional silence</i>
---------------	--

Description

Append two sounds with optional silence

Usage

```
sounds_append(sound1, sound2, silence_duration = 0)
```

Arguments

- sound1 First Sound object
- sound2 Second Sound object
- silence_duration Duration of silence to insert between sounds (seconds)

Value

New Sound object containing sound1, silence, and sound2

Examples

```
s1 <- Sound$create_tone(frequency = 220, duration = 0.2)
s2 <- Sound$create_tone(frequency = 440, duration = 0.2)
combined <- sounds_append(s1, s2, silence_duration = 0.1)
```

sounds_convolve	<i>Convolve two sounds</i>
-----------------	----------------------------

Description

Convolve two sounds

Usage

```
sounds_convolve(sound1, sound2, scaling = 4L, signal_outside = 1L)
```

Arguments

sound1	First Sound object
sound2	Second Sound object (filter/impulse response)
scaling	Scaling: 1=integral, 2=sum, 3=normalize, 4=peak_0.99
signal_outside	Signal outside time domain: 1=zero, 2=similar

Value

New Sound object

Examples

```
s1 <- Sound$create_tone(frequency = 220, duration = 0.2)
s2 <- Sound$create_tone(frequency = 440, duration = 0.05)
conv <- sounds_convolve(s1, s2)
```

sounds_cross_correlate	<i>Cross-correlate two sounds</i>
------------------------	-----------------------------------

Description

Cross-correlate two sounds

Usage

```
sounds_cross_correlate(sound1, sound2, scaling = 4L, signal_outside = 1L)
```

Arguments

sound1	First Sound object
sound2	Second Sound object
scaling	Scaling: 1=integral, 2=sum, 3=normalize, 4=peak_0.99
signal_outside	Signal outside time domain: 1=zero, 2=similar

Value

New Sound object (cross-correlation function)

Examples

```
s1 <- Sound$create_tone(frequency = 220, duration = 0.2)
s2 <- Sound$create_tone(frequency = 220, duration = 0.2)
xcorr <- sounds_cross_correlate(s1, s2)
```

sounds_to_dtw	Create DTW from two Sound objects
---------------	-----------------------------------

Description

Aligns a candidate sound to a reference sound using MFCC-based dynamic time warping.

Usage

```
sounds_to_dtw(
  reference,
  candidate,
  analysis_width = 0.015,
  time_step = 0.005,
  band = 0,
  slope = 3
)
```

Arguments

- reference Reference sound (prototype, y-axis)
- candidate Candidate sound (test, x-axis)
- analysis_width Window length in seconds (default: 0.015)
- time_step Time step between frames (default: 0.005)
- band Sakoe-Chiba band width in seconds (0 = no constraint)
- slope Slope constraint: 1=none, 2=1/3-3, 3=1/2-2, 4=2/3-3/2

Value

A DTW object

Examples

```
ref <- Sound$create_tone(frequency = 200, duration = 0.3, sampling_rate =
  16000)
test <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
dtw <- sounds_to_dtw(ref, test)
print(dtw$get_weighted_distance())
```

Spectrogram

*Spectrogram***Description**

A Spectrogram is a time-frequency representation of sound: a matrix of power values indexed by time (columns) and frequency (rows). Created from a Sound via short-time Fourier transform (STFT).

Value

A Spectrogram object.

Query methods

- `get_start_time()`, `get_end_time()` - time range (s)
- `get_time_step()` - time between frames (s)
- `get_number_of_time_bins()` - number of time frames
- `get_lowest_frequency()`, `get_highest_frequency()` - frequency range (Hz)
- `get_frequency_step()` - frequency resolution (Hz)
- `get_number_of_frequency_bins()` - number of frequency bins

Index mapping

- `get_time_from_frame(frame)` - time for a frame index
- `get_frame_from_time(time)` - frame index for a time
- `get_frequency_from_bin(bin)` - frequency for a bin index
- `get_bin_from_frequency(freq)` - bin index for a frequency

Power queries

- `get_power_at(time, frequency)` - power at a time-frequency point
- `get_power_at_points(times, frequencies)` - power at a vector of points (batch)
- `get_frame(time)` - full frequency spectrum at one time
- `get_frequency_slice(frequency)` - time series at one frequency

- `get_frames(times)` - matrix of frames at multiple times
- `get_band_power(fmin, fmax)` - integrated power in a frequency band
- `get_spectral_moments_batch(power)` - center of gravity, SD, skewness, and kurtosis per frame (single C++ call)

Export

- `as_matrix(include_dimnames)` - export as a numeric matrix
- `as_data_frame()` - export as a `data.frame`

Transform

- `to_spectrum(time)` - extract a spectrum at one time point
- `to_dtw(reference)` - dynamic time warping between spectrograms

See Also

[Sound](#), [Spectrum](#), [DTW](#), [ComplexSpectrogram](#)

Examples

```
snd <- Sound$create_tone(duration = 0.5, frequency = 440, sampling_rate =
  44100)
spec <- snd$to_spectrogram(window_length = 0.005, max_frequency = 5000)
power <- spec$get_power_at(time = 0.25, frequency = 440)
```

`spectrograms_to_dtw` *Create DTW from two Spectrogram objects*

Description

Create DTW from two Spectrogram objects

Usage

```
spectrograms_to_dtw(
  spectrogram1,
  spectrogram2,
  match_start = TRUE,
  match_end = TRUE,
  slope = 1,
  metric = 2
)
```

Arguments

spectrogram1	First spectrogram (candidate)
spectrogram2	Second spectrogram (reference)
match_start	Force path to start at (1,1)
match_end	Force path to end at (nx,ny)
slope	Slope constraint (1-4)
metric	Distance metric power (default: 2 = Euclidean)

Value

A DTW object

Examples

```
s1 <- Sound$create_tone(frequency = 200, duration = 0.3, sampling_rate =
  16000)
s2 <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
dtw <- spectrograms_to_dtw(s1$to_spectrogram(), s2$to_spectrogram())
```

Spectrum

Spectrum

Description

Praat Spectrum object with direct C++ module binding (complex FFT spectrum). Spectrum objects represent frequency-domain representations of sounds. Uses a shared dispatch table for minimal memory per object.

Arguments

.xptr	Not for direct use. External pointer to the underlying C++ Spectrum object; set internally when a method returns a new Spectrum.
-------	--

Value

A Spectrum object with methods for frequency-domain spectral analysis.

Query methods

- `get_lowest_frequency()` - lowest frequency (Hz)
- `get_highest_frequency()` - highest frequency (Hz)
- `get_number_of_bins()` - number of frequency bins
- `get_frequency_step()` - frequency step (Hz)

- `get_real_value_in_bin(bin)` - real part at a bin
- `get_imaginary_value_in_bin(bin)` - imaginary part at a bin
- `get_frequency_from_bin(bin)` - frequency for a bin number
- `get_bin_from_frequency(freq)` - bin number for a frequency
- `get_band_density(fmin, fmax)` - power density in a band (Pa^2/Hz^2)
- `get_band_energy(fmin, fmax)` - energy in a band ($\text{Pa}^2\cdot\text{s}$)
- `get_centre_of_gravity(power = 2.0)` - spectral center of gravity
- `get_standard_deviation(power = 2.0)` - spectral standard deviation
- `get_skewness(power = 2.0)` - spectral skewness
- `get_kurtosis(power = 2.0)` - spectral kurtosis
- `get_central_moment(moment, power = 2.0)` - central moment

Modification methods

- `pass_hann_band(fmin, fmax, smooth = 100)` - apply a Hann band-pass filter
- `stop_hann_band(fmin, fmax, smooth = 100)` - apply a Hann band-stop filter
- `cepstral_smoothing(bandwidth)` - smooth using the cepstral method

Transform methods

- `to_sound()` - convert to Sound (inverse FFT)
- `to_ltas(bandwidth)` - convert to long-term average spectrum
- `to_spectrogram(...)` - convert to Spectrogram
- `to_excitation(erb_density)` - convert to Excitation (auditory representation)

Export methods

- `as_matrix()` - export as a numeric matrix (real + imaginary)
- `as_data_frame()` - export as a data.frame (freq, real, imag, power)
- `save(path)` - save to a file

See Also

[Sound](#), [Spectrogram](#), [Ltas](#), [PowerCepstrum](#)

Examples

```
sound <- Sound$create_tone(duration = 0.5, frequency = 440, sampling_rate =
  44100)
spectrum <- sound$to_spectrum(fast = FALSE)
cog <- spectrum$get_centre_of_gravity(power = 2.0)
energy <- spectrum$get_band_energy(fmin = 400, fmax = 500)

# Create a spectrum from a recording read from disk
sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))
```

```
spectrum <- sound$to_spectrum(fast = TRUE)
spec_df <- spectrum$as_data_frame()
```

spectrum_cepstral_smoothing

Apply cepstral smoothing to spectrum

Description

Apply cepstral smoothing to spectrum

Usage

```
spectrum_cepstral_smoothing(spectrum, bandwidth = 500)
```

Arguments

spectrum	Spectrum object
bandwidth	Smoothing bandwidth (Hz)

Value

New Spectrum object with smoothing applied

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
spectrum <- sound$to_spectrum()
smoothed <- spectrum_cepstral_smoothing(spectrum, bandwidth = 500)
```

spectrum_pass_hann_band

Apply Hann band-pass filter to spectrum (in-place)

Description

Apply Hann band-pass filter to spectrum (in-place)

Usage

```
spectrum_pass_hann_band(spectrum, fmin, fmax, smooth = 100)
```

Arguments

spectrum	Spectrum object (will be modified)
fmin	Low frequency cutoff (Hz)
fmax	High frequency cutoff (Hz)
smooth	Smoothing bandwidth (Hz)

Value

NULL (modifies spectrum in place)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
spectrum <- sound$to_spectrum()
spectrum_pass_hann_band(spectrum, fmin = 300, fmax = 3000, smooth = 100)
# spectrum is now modified
```

spectrum_stop_hann_band

Apply Hann band-stop filter to spectrum (in-place)

Description

Apply Hann band-stop filter to spectrum (in-place)

Usage

```
spectrum_stop_hann_band(spectrum, fmin, fmax, smooth = 100)
```

Arguments

spectrum	Spectrum object (will be modified)
fmin	Low frequency cutoff (Hz)
fmax	High frequency cutoff (Hz)
smooth	Smoothing bandwidth (Hz)

Value

NULL (modifies spectrum in place)

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 0.5)
spectrum <- sound$to_spectrum()
spectrum_stop_hann_band(spectrum, fmin = 50, fmax = 100, smooth = 50)
# spectrum is now modified
```

SpectrumTier

SpectrumTier

Description

Holds the frequency peaks picked out of a long-term average spectrum.

Arguments

<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ object; set internally by <code>ltas\$to_spectrum_tier_peaks()</code> .
--------------------	--

Details

A SpectrumTier is the result of peak-picking an [Ltas](#): each point is one local maximum, recorded as a frequency and its power in dB. Use it to pull out harmonic or formant-like peaks from a spectral average without scanning the raw values by hand. It's read-only: build one with `ltas$to_spectrum_tier_peaks()`, then inspect or export it.

Value

A SpectrumTier object with methods for reading its frequency/power points.

Usage

```
peaks <- ltas$to_spectrum_tier_peaks()
```

Query methods

- `get_lowest_frequency()`, `get_highest_frequency()` - the frequency range the peaks were picked from, in Hz
- `get_number_of_points()` - number of peaks found
- `get_frequency_from_index(index)` - frequency of one peak
- `get_value_at_index(index)` - power of one peak, in dB

Export

- `as_data_frame()` - all peaks as a data frame, with `frequency` and `power_db` columns
- `as_matrix()` - the same data as a plain matrix
- `save(path)` - write to a Praat text file

See Also

[Ltas](#)

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 0.5)
ltas <- sound$to_ltas(bandwidth = 100)
peaks <- ltas$to_spectrum_tier_peaks()
peaks$get_number_of_points()
peaks$as_data_frame()

```

summary.praat_formant *Summary method for praat_formant objects*

Description

Summary method for praat_formant objects

Usage

```

## S3 method for class 'praat_formant'
summary(object, ...)

```

Arguments

object	A praat_formant object
...	Additional arguments (unused)

Value

object, invisibly.

Examples

```

x <- list(
  n_frames = 2, n_formants = 1, time_step = 0.01,
  max_formant = 5000, window_length = 0.025,
  values = data.frame(
    time = c(0.1, 0.2), formant_number = c(1, 1),
    frequency = c(500, 520), bandwidth = c(80, 82)
  )
)
class(x) <- "praat_formant"
summary(x)

```

`summary.praat_intensity`*Summary method for praat_intensity objects*

Description

Summary method for praat_intensity objects

Usage

```
## S3 method for class 'praat_intensity'  
summary(object, ...)
```

Arguments

<code>object</code>	A praat_intensity object
<code>...</code>	Additional arguments (unused)

Value

object, invisibly.

Examples

```
x <- list(  
  n_frames = 2, time_step = 0.01, minimum_pitch = 100,  
  values = data.frame(time = c(0.1, 0.2), intensity_db = c(65.2, 66.1))  
)  
class(x) <- "praat_intensity"  
summary(x)
```

`summary.praat_pitch`*Summary method for praat_pitch objects*

Description

Provides a detailed statistical summary of a pitch contour.

Usage

```
## S3 method for class 'praat_pitch'  
summary(object, ...)
```

Arguments

<code>object</code>	A praat_pitch object
<code>...</code>	Additional arguments passed to the underlying function or ignored.

Value

The object, invisibly

Examples

```
x <- data.frame(time = c(0.1, 0.2, 0.3), frequency = c(120, 125, NA))
class(x) <- c("praat_pitch", "data.frame")
summary(x)
```

summary.praat_sound	<i>Summary method for praat_sound objects</i>
---------------------	---

Description

Provides a statistical summary of a praat_sound object, including amplitude statistics and metadata.

Usage

```
## S3 method for class 'praat_sound'
summary(object, ...)
```

Arguments

object	A praat_sound object
...	Additional arguments passed to the underlying function or ignored.

Value

The object, invisibly

Examples

```
sound <- list(
  duration = 0.5, sampling_rate = 8000, n_samples = 4000, n_channels = 1,
  start_time = 0, end_time = 0.5,
  values = sin(2 * pi * 150 * seq(0, 0.5, length.out = 4000))
)
class(sound) <- "praat_sound"
summary(sound)
```

Table	<i>Praat Table Object</i>
-------	---------------------------

Description

Praat Table object with direct C++ module binding for tabular data operations.

Arguments

numberOfRows	Number of rows. Required unless .xptr is given.
numberOfColumns	Number of columns. Required unless columnNames or .xptr is given.
columnNames	Character vector of column names, used instead of numberOfColumns to name columns as they are created.
.xptr	Not for direct use. External pointer to the underlying C++ Table object; set internally when a method returns a new Table.

Details

Table objects store tabular data with named columns and support various statistical operations.

Value

A Table object with methods for tabular data access and statistics.

Examples

```
tbl <- Table(numberOfRows = 3, columnNames = c("word", "duration"))
tbl$set_string_value(1, "word", "cat")
tbl$set_numeric_value(1, "duration", 0.42)
tbl$get_number_of_rows()
tbl$get_string_value(1, "word")
tbl$get_numeric_value(1, "duration")
```

table_create	<i>Create a Praat Table</i>
--------------	-----------------------------

Description

Creates a new Table object with specified dimensions.

Usage

```
table_create(numberOfRows, numberOfColumns = NULL, columnNames = NULL)
```

Arguments

numberOfRows	Number of rows
numberOfColumns	Number of columns (optional if columnNames provided)
columnNames	Character vector of column names (optional)

Value

A Table object

See Also

[Table](#) for object methods

Examples

```
tbl1 <- table_create(numberOfRows = 3, columnNames = c("speaker", "f0"))
tbl2 <- table_create(numberOfRows = 3, numberOfColumns = 2)
```

TextGrid	<i>Praat TextGrid Object</i>
----------	------------------------------

Description

R wrapper for a Praat TextGrid object for linguistic annotation. Uses shared dispatch table for minimal memory per object.

Arguments

path	Path to a TextGrid file (Praat text or binary format). See the Creating TextGrid objects section for details.
.xptr	Not for direct use. External pointer to the underlying C++ TextGrid object; set internally when a method returns a new TextGrid.

Details

TextGrids are the primary tool for linguistic annotation in Praat. They contain one or more tiers, where each tier can be:

- IntervalTier: consecutive time intervals with labels (e.g., phonemes, words)
- PointTier (TextTier): time points with labels (e.g., events, tones)

Value

A TextGrid object with methods for tier and interval/point annotation access.

Creating TextGrid objects

- `TextGrid(path)` - read from file (Praat text or binary format)
- `textgrid_create(tmin, tmax, tier_names, point_tiers)` - create an empty grid

Querying tiers

- `get_number_of_tiers()` - number of tiers
- `get_tier_names()` - names of all tiers
- `tier_is_interval_tier(tier)` - check if a tier is an IntervalTier
- `tier_is_point_tier(tier)` - check if a tier is a PointTier

IntervalTier operations

- `get_number_of_intervals(tier)` - number of intervals in a tier
- `get_interval_text(tier, n)` - get the label of interval n
- `get_label_at_time(tier, time)` - get the label at a specific time
- `get_all_intervals(tier)` - get all intervals as a data.frame (fast)
- `extract_intervals_batch(tier, ...)` - extract matching intervals (fast)
- `extract_intervals_where(sound, tier, criterion, text, preserve_times)` - extract Sound intervals matching a text criterion
- `set_interval_text(tier, n, text)` - set the label of interval n
- `insert_boundary(tier, time)` - insert a new boundary
- `remove_boundary(tier, time)` - remove a boundary

PointTier operations

- `get_number_of_points(tier)` - number of points in a tier
- `get_point_text(tier, n)` - get the label of point n
- `insert_point(tier, time, mark)` - insert a new point
- `set_point_text(tier, n, text)` - set the label of point n
- `remove_point(tier, n)` - remove a point

Tier management

- `add_interval_tier(name)` - add a new IntervalTier
- `add_point_tier(name)` - add a new PointTier
- `remove_tier(tier)` - remove a tier
- `set_tier_name(tier, name)` - rename a tier
- `duplicate_tier(tier, new_name)` - duplicate a tier with a new name

Export

- as_data_frame(tiers) - convert to a long-format data frame
- save(path) - write to a file
- extract_part(start, end) - extract a time range

Examples

```
# Create new TextGrid with one interval tier and one point tier
# (tier_names lists all tiers; point_tiers names the subset that are point
# tiers)
tg <- textgrid_create(0, 1, "words tones", "tones")
tg$get_tier_names()

# Add boundaries and labels (IntervalTier)
tg$insert_boundary("words", 0.4)
tg$insert_boundary("words", 0.7)
tg$set_interval_text("words", 1, "hello")
tg$set_interval_text("words", 2, "world")

# Add points and labels (PointTier)
tg$insert_point("tones", 0.2, "H*")
tg$insert_point("tones", 0.8, "L-L%")

# Query
tg$get_number_of_intervals("words")
label <- tg$get_label_at_time("words", 0.5)

# Export to R
df <- tg$as_data_frame()
```

textgrid_create	Create TextGrid
-----------------	-----------------

Description

Create a new empty TextGrid with specified tiers

Usage

```
textgrid_create(tmin, tmax, tier_names = "", point_tiers = "")
```

Arguments

tmin	Start time in seconds
tmax	End time in seconds
tier_names	Space-separated tier names (e.g., "phones words syllables")
point_tiers	Space-separated names of tiers that should be PointTiers (default: all are IntervalTiers)

Value

A TextGrid object.

See Also

[Sound](#), [Pitch](#), [Formant](#), [PointProcess](#)

Examples

```
# Create TextGrid with 3 interval tiers
tg <- textgrid_create(0, 10, "phones words syllables")

# Create TextGrid with mixed tier types
tg2 <- textgrid_create(0, 10, "phones tones", "tones") # tones is PointTier
```

textgrid_extract_intervals_batch

Extract TextGrid Intervals by Label (Batch)

Description

Extract multiple intervals from a TextGrid tier that match specified criteria, using: - A single C++ call instead of 4n R<->C++ calls (n = number of intervals) - Comparisons done at the C++ level - A single memory allocation for the result

Usage

```
textgrid_extract_intervals_batch(
  textgrid_xptr,
  sound_xptr,
  tier_number,
  comparison_type = "equals",
  target_value = "",
  extract_sounds = FALSE
)
```

Arguments

textgrid_xptr	External pointer to TextGrid object
sound_xptr	External pointer to Sound object (optional, can be NULL)
tier_number	Tier number (1-based)
comparison_type	Type of comparison: "equals", "contains", "starts_with", "regex"
target_value	Value to match against interval labels
extract_sounds	If TRUE and sound_xptr provided, extract Sound parts

Details

****Comparison types:**** - "equals": Exact match (strcmp) - "contains": Substring match (strstr) - "starts_with": Prefix match - "regex": Regular expression (future)

Value

List with components: - indices: Integer vector of matching interval indices - labels: Character vector of matching labels - start_times: Numeric vector of start times - end_times: Numeric vector of end times - sounds: List of Sound xptrs (if extract_sounds = TRUE)

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 1.0)
tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.4)
tg$insert_boundary("phones", 0.7)
tg$set_interval_text("phones", 1, "sil")
tg$set_interval_text("phones", 2, "V")
tg$set_interval_text("phones", 3, "sil")

# Extract all "V" (voiced) intervals
result <- textgrid_extract_intervals_batch(
  tg$.xptr,
  sound$.xptr,
  tier_number = 1,
  comparison_type = "equals",
  target_value = "V",
  extract_sounds = TRUE
)

# Access results
n_voiced <- length(result$indices)
voiced_durations <- result$end_times - result$start_times
voiced_sounds <- result$sounds # List of Sound xptrs
```

textgrid_filter_xptr *Extract TextGrid Intervals Using Custom XPtr Predicate*

Description

Filter intervals using a user-compiled C++ predicate function. The predicate executes entirely in C++ without any R boundary crossings.

Usage

```
textgrid_filter_xptr(
  textgrid_xptr,
  tier_number,
```

```

    predicate_xptr,
    sound_xptr = NULL,
    extract_sounds = FALSE
)

```

Arguments

textgrid_xptr External pointer to TextGrid object

tier_number Tier number (1-based)

predicate_xptr External pointer to compiled predicate function created with 'RcppXPtrUtils::cppXPtr()'.
Signature must be: 'bool(const char* label, double start, double end)'

sound_xptr Optional external pointer to Sound for extraction

extract_sounds If TRUE and sound_xptr provided, extract Sound parts

Details

```

**Compiling a custom predicate (requires RcppXPtrUtils):**

“r # Example: Filter intervals with duration > 0.1s and label starting with 'V' my_pred <- RcppX-
PtrUtils::cppXPtr( "bool pred(cstr label, double start, double end) { double dur = end - start; return
dur > 0.1 && label[0] == 'V'; }", includes = "typedef const char* cstr;" )

result <- textgrid_filter_xptr( textgrid$.xptr, tier = 1, predicate_xptr = my_pred ) “

```

Value

List with components: - indices: Integer vector of matching interval indices - labels: Character vector of matching labels - start_times: Numeric vector of start times - end_times: Numeric vector of end times - sounds: List of Sound xptrs (if extract_sounds = TRUE)

See Also

[textgrid_extract_intervals_batch()] for simpler string matching

Examples

```

if (requireNamespace("RcppXPtrUtils", quietly = TRUE)) {
  tg <- textgrid_create(0, 1, "phones")
  tg$insert_boundary("phones", 0.4)
  tg$insert_boundary("phones", 0.7)
  tg$set_interval_text("phones", 1, "sil")
  tg$set_interval_text("phones", 2, "V")
  tg$set_interval_text("phones", 3, "sil")

  my_pred <- RcppXPtrUtils::cppXPtr(
    "bool pred(cstr label, double start, double end) {
      double dur = end - start;
      return dur > 0.1 && label[0] == 'V';
    }",
    includes = "typedef const char* cstr;"
  )
}

```

```

    result <- textgrid_filter_xptr(tg$.xptr, 1, my_pred)
  }

```

```
textgrid_get_all_labels
```

Get All Labels from TextGrid Tier (Batch)

Description

Extract all interval labels from a tier in a single call, instead of calling ‘get_interval_text()’ n times.

Usage

```
textgrid_get_all_labels(textgrid_xptr, tier_number)
```

Arguments

`textgrid_xptr` External pointer to TextGrid object
`tier_number` Tier number (1-based)

Value

Character vector of all interval labels

Examples

```

tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.4)
tg$set_interval_text("phones", 1, "sil")
tg$set_interval_text("phones", 2, "V")

labels <- textgrid_get_all_labels(tg$.xptr, tier_number = 1)
table(labels) # Frequency of each label

```

textgrid_get_intervals_where

Extract Intervals from TextGrid Matching Criteria

Description

Extracts intervals from a TextGrid tier that match a specified criterion. Returns time boundaries and labels of matching intervals.

Usage

```
textgrid_get_intervals_where(
  textgrid,
  tier = 1,
  condition = c("equals", "contains", "does not contain", "starts with", "ends with"),
  text
)
```

Arguments

textgrid	TextGrid object
tier	Integer. Tier number (1-indexed)
condition	Character. Matching condition: - "equals" or "is equal to" - Exact match - "contains" - Label contains text - "does not contain" - Label does not contain text - "starts with" - Label starts with text - "ends with" - Label ends with text
text	Character. Text to match

Value

Named list with: - 'xmin': Numeric vector of interval start times - 'xmax': Numeric vector of interval end times - 'text': Character vector of interval labels - 'count': Integer number of matching intervals

Examples

```
sound <- sounds_append(
  Sound$create_tone(frequency = 200, duration = 0.5, amplitude = 0.8),
  Sound$create_tone(frequency = 200, duration = 0.3, amplitude = 0.001)
)
vad_grid <- sound_to_textgrid_silences(sound)

# Get all "sounding" intervals
voiced <- textgrid_get_intervals_where(
  vad_grid,
  tier = 1,
  condition = "equals",
  text = "sounding"
```

```
)

cat("Found", voiced$count, "voiced segments\n")
cat("Total voiced duration:", sum(voiced$xmax - voiced$xmin), "s\n")
```

textgrid_interval_all_features_batch
<i>Extract All Acoustic Features for TextGrid Intervals (Batch, SIMD)</i>

Description

Comprehensive batch extraction of pitch, formant F1/F2, and intensity statistics for all intervals in a single call, as an alternative to calling ‘textgrid_interval_pitch_batch()’, ‘textgrid_interval_formant_batch()’, and ‘textgrid_interval_intensity_batch()’ separately.

Usage

```
textgrid_interval_all_features_batch(
  textgrid_xptr,
  pitch_xptr = NULL,
  formant_xptr = NULL,
  intensity_xptr = NULL,
  tier_number = 1L
)
```

Arguments

- textgrid_xptr External pointer to TextGrid
- pitch_xptr External pointer to Pitch (optional)
- formant_xptr External pointer to Formant (optional)
- intensity_xptr External pointer to Intensity (optional)
- tier_number Tier number (1-based)

Details

Duration calculation and statistics aggregation use SIMD where applicable.

Value

Data frame with all available features per interval

Examples

```

sound <- Sound$create_tone(frequency = 180, duration = 1.0)
pitch <- sound$to_pitch()
formant <- sound$to_formant_burg()
intensity <- sound$to_intensity()
tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.5)
tg$set_interval_text("phones", 1, "a")
tg$set_interval_text("phones", 2, "b")

stats <- textgrid_interval_all_features_batch(
  tg$.xptr, pitch$.xptr, formant$.xptr, intensity$.xptr, tier_number = 1
)

```

textgrid_interval_formant_batch

Extract Formant Statistics for All TextGrid Intervals (Batch, SIMD)

Description

Computes formant statistics for all intervals using SIMD.

Usage

```

textgrid_interval_formant_batch(
  textgrid_xptr,
  formant_xptr,
  tier_number,
  formant_number = 1L
)

```

Arguments

textgrid_xptr	External pointer to TextGrid
formant_xptr	External pointer to Formant object
tier_number	Tier number (1-based)
formant_number	Formant number to extract (1 = F1, 2 = F2, etc.)

Value

Data frame with interval info and formant statistics

Examples

```

sound <- Sound$create_tone(frequency = 220, duration = 1.0)
formant <- sound$to_formant_burg()
tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.5)
tg$set_interval_text("phones", 1, "a")
tg$set_interval_text("phones", 2, "b")

stats <- textgrid_interval_formant_batch(tg$.xptr, formant$.xptr,
  tier_number = 1, formant_number = 1)

```

textgrid_interval_intensity_batch

Extract Intensity Statistics for All TextGrid Intervals (Batch, SIMD)

Description

Extract Intensity Statistics for All TextGrid Intervals (Batch, SIMD)

Usage

```
textgrid_interval_intensity_batch(textgrid_xptr, intensity_xptr, tier_number)
```

Arguments

textgrid_xptr External pointer to TextGrid
intensity_xptr External pointer to Intensity object
tier_number Tier number (1-based)

Value

Data frame with interval info and intensity statistics

Examples

```

sound <- Sound$create_tone(frequency = 150, duration = 1.0)
intensity <- sound$to_intensity()
tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.5)
tg$set_interval_text("phones", 1, "a")
tg$set_interval_text("phones", 2, "b")

stats <- textgrid_interval_intensity_batch(tg$.xptr, intensity$.xptr, tier_number = 1)

```

textgrid_interval_pitch_batch

Extract Pitch Statistics for All TextGrid Intervals (Batch, SIMD)

Description

Computes pitch statistics (mean, stdev, min, max) for all intervals in a TextGrid tier using SIMD-accelerated batch processing.

Usage

```
textgrid_interval_pitch_batch(
  textgrid_xptr,
  pitch_xptr,
  tier_number,
  unit = "HERTZ"
)
```

Arguments

textgrid_xptr	External pointer to TextGrid
pitch_xptr	External pointer to Pitch object
tier_number	Tier number (1-based)
unit	Pitch unit: "HERTZ" or "SEMITONES"

Details

This function combines: 1. SIMD duration calculation for all intervals 2. SIMD statistics calculation for pitch values in each interval

Value

Data frame with interval index, label, start, end, duration, pitch_mean, pitch_stdev, pitch_min, pitch_max

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1.0)
pitch <- sound$to_pitch()
tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.5)
tg$set_interval_text("phones", 1, "a")
tg$set_interval_text("phones", 2, "b")

stats <- textgrid_interval_pitch_batch(tg$.xptr, pitch$.xptr, tier_number = 1, unit = "HERTZ")
```

textgrid_interval_statistics_batch

Compute Statistics for All Intervals (Batch, SIMD-Optimized)

Description

Compute statistics (duration, etc.) for all intervals in a tier. Single C++ call instead of looping in R. Uses SIMD for duration calculation when available.

Usage

```
textgrid_interval_statistics_batch(textgrid_xptr, tier_number)
```

Arguments

textgrid_xptr External pointer to TextGrid object

tier_number Tier number (1-based)

Details

Duration calculation uses SIMD vectorization when available, for large interval counts (>100).

Value

Data frame with columns: - index: Interval index - label: Interval label - start: Start time - end: End time - duration: Duration (end - start)

Examples

```
tg <- textgrid_create(0, 1, "phones")
tg$insert_boundary("phones", 0.4)
tg$set_interval_text("phones", 1, "sil")
tg$set_interval_text("phones", 2, "V")

stats <- textgrid_interval_statistics_batch(tg$.xptr, tier_number = 1)
mean(stats$duration[stats$label == "V"]) # Mean voiced interval duration
```

textgrid_merge	<i>Merge Multiple TextGrid Objects</i>
----------------	--

Description

Batch merging using Praat's $O(n)$ algorithm instead of $O(n^2)$ manual tier copying. Manual merge requires save/reload + insert_boundary for each interval (each insert shifts all later intervals). Batch merge is single-pass.

Usage

```
textgrid_merge(textgrids, equalize_domains = FALSE)
```

Arguments

textgrids	List of TextGrid objects (external pointers or R6 objects with .xptr)
equalize_domains	If TRUE, all tiers extended to same domain with empty intervals at edges if needed (default: FALSE)

Details

Manual merge requires save/reload plus an $O(n^2)$ sequence of 'insert_boundary' calls (each insert shifts all later intervals); batch merge does a single $O(n)$ pass with proper interval handling.

****Domain handling:**** - If 'equalize_domains = FALSE' (default): * New domain runs from min(xmin) to max(xmax) of all input TextGrids * Tiers retain their original domains

- If 'equalize_domains = TRUE': * All tiers extended to the new domain * Empty intervals added at edges if needed

****Use cases:**** - VUV analysis: Merging original TextGrid with VUV tier - Multi-annotator: Combining annotations from different annotators - Workflow: Adding automatic tiers to manual annotations

Value

TextGrid object (external pointer)

See Also

[TextGrid](#) for TextGrid object creation

Other batch-ops: [sound_load_window\(\)](#)

Examples

```
# Create test TextGrids
tg1 <- textgrid_create(0, 1, "words")
tg1$insert_boundary(1, 0.5)
tg1$set_interval_text(1, 1, "hello")
tg1$set_interval_text(1, 2, "world")

tg2 <- textgrid_create(0, 1, "events", "events")
tg2$insert_point(1, 0.25, "click")

# Batch merge
merged <- textgrid_merge(list(tg1, tg2))
# Result has 2 tiers: "words" (interval) + "events" (point)

# With domain equalization
merged_eq <- textgrid_merge(list(tg1, tg2), equalize_domains = TRUE)
```

`textgrid_simd_enabled` *Check if SIMD is Enabled for TextGrid*

Description

Check if SIMD is Enabled for TextGrid

Usage

```
textgrid_simd_enabled()
```

Value

Logical indicating SIMD status

Examples

```
textgrid_simd_enabled()
```

`to_formant_direct` *Create Formant from Sound Directly (returns XPtr)*

Description

Create Formant from Sound Directly (returns XPtr)

Usage

```
to_formant_direct(
  sound,
  time_step = 0,
  max_formants = 5,
  max_formant = 5500,
  window_length = 0.025,
  pre_emphasis = 50,
  max_number_of_formants = NULL,
  maximum_formant = NULL,
  pre_emphasis_from = NULL
)
```

Arguments

sound	Sound object or external pointer
time_step	Time step (0 = auto)
max_formants	Maximum number of formants
max_formant	Maximum formant frequency (Hz)
window_length	Window length (seconds)
pre_emphasis	Pre-emphasis frequency (Hz)
max_number_of_formants	Alias for 'max_formants' (maximum number of formants)
maximum_formant	Alias for 'max_formant' (maximum formant frequency, Hz)
pre_emphasis_from	Alias for 'pre_emphasis' (pre-emphasis frequency, Hz)

Value

External pointer to Formant

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)
formant_ptr <- to_formant_direct(sound)
f1 <- get_formant_value_direct(formant_ptr, 1, 0.25, "hertz")
```

to_harmonicity_direct *Create Harmonicity from Sound Directly (returns XPtr)*

Description

Create Harmonicity from Sound Directly (returns XPtr)

Usage

```
to_harmonicity_direct(  
  sound,  
  time_step = 0.01,  
  minimum_pitch = 75,  
  silence_threshold = 0.1,  
  periods_per_window = 1  
)
```

Arguments

sound	Sound object or external pointer
time_step	Time step
minimum_pitch	Minimum pitch (Hz)
silence_threshold	Silence threshold
periods_per_window	Periods per window

Value

External pointer to Harmonicity

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)  
hnr_ptr <- to_harmonicity_direct(sound)  
hnr <- Harmonicity(.xptr = hnr_ptr)  
hnr$get_mean(0, 0)
```

to_intensity_direct	Create Intensity from Sound Directly (returns XPtr)
---------------------	---

Description

Create Intensity from Sound Directly (returns XPtr)

Usage

```
to_intensity_direct(  
  sound,  
  minimum_pitch = 100,  
  time_step = 0,  
  subtract_mean = TRUE  
)
```

Arguments

sound	Sound object or external pointer
minimum_pitch	Minimum pitch (Hz)
time_step	Time step (0 = auto)
subtract_mean	Whether to subtract mean

Value

External pointer to Intensity

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)  
intensity_ptr <- to_intensity_direct(sound)  
db <- get_intensity_value_direct(intensity_ptr, 0.25)
```

to_ltas_direct	Create LTAS from Sound Directly
----------------	---------------------------------

Description

Create LTAS from Sound Directly

Usage

```
to_ltas_direct(sound, bandwidth = 100)
```

Arguments

sound	Sound object or external pointer
bandwidth	Numeric. Bandwidth in Hz (default: 100)

Value

A wrapped Ltas object

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)
ltas <- to_ltas_direct(sound, bandwidth = 100)
ltas$get_slope(0, 1000, 1000, 10000, "energy")
```

to_pitch_ac_direct	<i>Create Pitch from Sound Directly (Autocorrelation) - Full Parameters</i>
--------------------	---

Description

Create Pitch analysis using autocorrelation method with full control over all voicing parameters. Returns a raw external pointer instead of an R6 object.

****NEW in v4.0.1:**** Exposes all voicing parameters that were previously only available in Tier 1 (Standard) API.

Usage

```
to_pitch_ac_direct(
  sound,
  time_step = 0,
  pitch_floor = 75,
  pitch_ceiling = 600,
  max_candidates = 15,
  very_accurate = FALSE,
  silence_threshold = 0.03,
  voicing_threshold = 0.45,
  octave_cost = 0.01,
  octave_jump_cost = 0.35,
  voiced_unvoiced_cost = 0.14,
  max_number_of_candidates = NULL
)
```


Arguments

sound	Sound object or external pointer
time_step	Time step (0 = auto, typically 0.75/pitch_floor)
pitch_floor	Minimum pitch (Hz, default 75)
pitch_ceiling	Maximum pitch (Hz, default 600)
max_candidates	Maximum number of pitch candidates (default 15)
very_accurate	Use accurate but slower method (default FALSE)
silence_threshold	Frames below this relative intensity are unvoiced (default 0.03)
voicing_threshold	Strength required for voiced decision (default 0.45)
octave_cost	Cost per octave in path finding (default 0.01)
octave_jump_cost	Cost for octave jumps (default 0.35)
voiced_unvoiced_cost	Cost for voicing transitions (default 0.14)
max_number_of_candidates	Maximum number of pitch candidates per frame

Value

External pointer to Pitch (NOT R6 object)

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)

# With custom voicing threshold (stricter voicing detection)
pitch_ptr <- to_pitch_ac_direct(sound, voicing_threshold = 0.6)

# Use with query functions
f0 <- get_pitch_value_direct(pitch_ptr, 0.25, "hertz", TRUE)
```

to_pitch_cc_direct	<i>Create Pitch from Sound Directly (Cross-Correlation) - Full Parameters</i>
--------------------	---

Description

Create Pitch analysis using cross-correlation method with full control over all voicing parameters. Returns a raw external pointer instead of an R6 object.

****NEW in v4.0.1:**** Exposes all voicing parameters that were previously only available in Tier 1 (Standard) API.

Usage

```
to_pitch_cc_direct(
  sound,
  time_step = 0,
  pitch_floor = 75,
  pitch_ceiling = 600,
  max_candidates = 15,
  very_accurate = FALSE,
  silence_threshold = 0.03,
  voicing_threshold = 0.45,
  octave_cost = 0.01,
  octave_jump_cost = 0.35,
  voiced_unvoiced_cost = 0.14,
  max_number_of_candidates = NULL
)
```

Arguments

<code>sound</code>	Sound object or external pointer
<code>time_step</code>	Time step (0 = auto, typically 0.75/pitch_floor)
<code>pitch_floor</code>	Minimum pitch (Hz, default 75)
<code>pitch_ceiling</code>	Maximum pitch (Hz, default 600)
<code>max_candidates</code>	Maximum number of pitch candidates (default 15)
<code>very_accurate</code>	Use accurate but slower method (default FALSE)
<code>silence_threshold</code>	Frames below this relative intensity are unvoiced (default 0.03)
<code>voicing_threshold</code>	Strength required for voiced decision (default 0.45)
<code>octave_cost</code>	Cost per octave in path finding (default 0.01)
<code>octave_jump_cost</code>	Cost for octave jumps (default 0.35)
<code>voiced_unvoiced_cost</code>	Cost for voicing transitions (default 0.14)
<code>max_number_of_candidates</code>	Maximum number of pitch candidates per frame

Value

External pointer to Pitch (NOT R6 object)

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)

# With custom voicing threshold (stricter voicing detection)
pitch_ptr <- to_pitch_cc_direct(sound, voicing_threshold = 0.6)
```

```
# Use with query functions
f0 <- get_pitch_value_direct(pitch_ptr, 0.25, "hertz", TRUE)
```

to_pitch_direct	<i>Create Pitch from Sound Directly (returns XPtr) - Basic Parameters</i>
-----------------	---

Description

Create Pitch analysis directly, returning raw external pointer. This is a simplified version with basic parameters only.

****NOTE:**** For full control over voicing parameters (silence_threshold, voicing_threshold, etc.), use 'to_pitch_ac_direct()' or 'to_pitch_cc_direct()' instead.

Usage

```
to_pitch_direct(sound, time_step = 0, pitch_floor = 75, pitch_ceiling = 600)
```

Arguments

sound	Sound object or external pointer
time_step	Time step (0 = auto)
pitch_floor	Minimum pitch (Hz)
pitch_ceiling	Maximum pitch (Hz)

Value

External pointer to Pitch (NOT R6 object)

See Also

[to_pitch_ac_direct](#), [to_pitch_cc_direct](#) for full parameter control

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)

# Returns raw pointer (basic parameters)
pitch_ptr <- to_pitch_direct(sound)

# Use with other direct query functions
stats <- get_pitch_stats_direct(pitch_ptr)
```

to_pitch_shs_direct	Create Pitch from Sound using Subharmonic Summation (SHS) Directly (returns XPtr)
---------------------	---

Description

Create Pitch from Sound using Subharmonic Summation (SHS) Directly (returns XPtr)

Usage

```
to_pitch_shs_direct(
  sound,
  time_step = 0.01,
  pitch_floor = 50,
  max_frequency = 1250,
  pitch_ceiling = 500,
  max_subharmonics = 15L,
  max_candidates = 15L,
  compression_factor = 0.84,
  n_points_per_octave = 48L
)
```

Arguments

sound	Sound object or external pointer
time_step	Time step in seconds (default 0.01)
pitch_floor	Minimum pitch (Hz, default 50)
max_frequency	Maximum frequency for analysis (Hz, default 1250)
pitch_ceiling	Maximum pitch (Hz, default 500)
max_subharmonics	Number of subharmonics to sum (default 15)
max_candidates	Maximum number of pitch candidates per frame (default 15)
compression_factor	Compression factor for subharmonic weighting (default 0.84)
n_points_per_octave	Number of frequency points per octave (default 48)

Value

External pointer to Pitch (NOT R6 object)

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)
pitch_ptr <- to_pitch_shs_direct(sound)
f0 <- get_pitch_value_direct(pitch_ptr, 0.25, "hertz", TRUE)
```

to_pitch_spinet_direct

Create Pitch from Sound using SPINET Directly (returns XPtr)

Description

Create Pitch from Sound using SPINET Directly (returns XPtr)

Usage

```
to_pitch_spinet_direct(
  sound,
  time_step = 0.005,
  window_duration = 0.04,
  min_frequency = 70,
  max_frequency = 5000,
  n_filters = 250L,
  pitch_ceiling = 500,
  max_candidates = 15L
)
```

Arguments

sound	Sound object or external pointer
time_step	Time step in seconds (default 0.005)
window_duration	Analysis window duration (default 0.04)
min_frequency	Minimum frequency (Hz, default 70)
max_frequency	Maximum frequency (Hz, default 5000)
n_filters	Number of gamma-tone filters (default 250)
pitch_ceiling	Maximum pitch (Hz, default 500)
max_candidates	Maximum number of pitch candidates per frame (default 15)

Value

External pointer to Pitch (NOT R6 object)

Examples

```
# The vendored Praat SPINET path has a rare, non-deterministic native
# flake ("all amplitudes equal to zero") unrelated to the input signal;
# tryCatch keeps this example from failing R CMD check when it strikes.
sound <- Sound$create_tone(frequency = 200, duration = 0.5)
pitch_ptr <- tryCatch(to_pitch_spinet_direct(sound), error = function(e)
  NULL)
if (!is.null(pitch_ptr)) {
```

```
f0 <- get_pitch_value_direct(pitch_ptr, 0.25, "hertz", TRUE)
}
```

to_point_process_direct

Create PointProcess from Sound Directly (returns XPtr)

Description

Create PointProcess from Sound Directly (returns XPtr)

Usage

```
to_point_process_direct(
  sound,
  pitch_floor = 75,
  pitch_ceiling = 600,
  time_step = 0,
  max_period_factor = 1.3,
  max_amplitude_factor = 1.6
)
```

Arguments

sound	Sound object or external pointer
pitch_floor	Numeric. Minimum pitch in Hz (default: 75)
pitch_ceiling	Numeric. Maximum pitch in Hz (default: 600)
time_step	Numeric. Time step in seconds (0 = auto)
max_period_factor	Numeric. Max period factor (default: 1.3)
max_amplitude_factor	Numeric. Max amplitude factor (default: 1.6)

Value

External pointer to PointProcess

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)
# Extract glottal pulses
pp_ptr <- to_point_process_direct(sound, pitch_floor = 75, pitch_ceiling =
  300)
pp <- PointProcess(.xptr = pp_ptr)
pp$get_number_of_points()
```

to_point_process_from_sound_and_pitch

Create PointProcess from Sound and Pitch (Cross-Correlation)

Description

Creates a PointProcess using BOTH Sound and refined Pitch contour. This matches Praat's "To PointProcess (cc)" command when both Sound and Pitch objects are selected.

****IMPORTANT for VUV Analysis:**** This function uses the refined pitch contour to guide period detection, which is more accurate than using pitch range parameters alone. This is the correct method for voice quality analysis (jitter, shimmer, VUV detection).

****Algorithm Difference:**** - 'sound\$to_point_process_periodic_cc(floor, ceiling)' - Uses only pitch range - 'to_point_process_from_sound_and_pitch(sound, pitch)' - Uses refined pitch contour (recommended)

Usage

```
to_point_process_from_sound_and_pitch(sound, pitch)
```

Arguments

sound	Sound object or external pointer
pitch	Pitch object or external pointer (from to_pitch_ac/cc)

Value

External pointer to PointProcess

See Also

[to_point_process_direct](#) for the single-object Sound method

Examples

```
sound <- Sound$create_tone(frequency = 200, duration = 0.5)

# Create refined pitch analysis
pitch <- sound$to_pitch_cc(
  time_step = 0,
  pitch_floor = 75,
  pitch_ceiling = 600,
  voicing_threshold = 0.45
)

# RECOMMENDED: Use both Sound and Pitch for accurate pulse detection
pp <- to_point_process_from_sound_and_pitch(sound, pitch)

# Now calculate jitter with accurate pulse times
```

```
pp_r6 <- PointProcess(.xptr = pp)
jitter <- pp_r6$get_jitter_local()
```

to_powercepstrogram_fast

Fast PowerCepstrogram Creation (Advanced Performance API)

Description

Create a PowerCepstrogram object bypassing R6 method dispatch for maximum performance. Returns an external pointer that can be used with other fast path functions.

Usage

```
to_powercepstrogram_fast(
  sound,
  pitch_floor = 60,
  time_step = 0.002,
  max_frequency = 5000,
  pre_emphasis_from = 50
)
```

Arguments

sound	Sound object or external pointer
pitch_floor	Numeric, minimum pitch in Hz (default 60)
time_step	Numeric, time step in seconds (default 0.002)
max_frequency	Numeric, maximum frequency in Hz (default 5000)
pre_emphasis_from	Numeric, pre-emphasis frequency in Hz (default 50)

Details

****ADVANCED API**** - Returns raw external pointer, not R6 object.

Use this when you need to create multiple PowerCepstrogram objects in a loop and want to minimize R6 overhead. The returned pointer can be: - Passed to other fast path functions - Wrapped in R6 PowerCepstrogram object if needed

Value

External pointer to PowerCepstrogram object

Examples

```

sound <- Sound(system.file("extdata", "test.wav", package = "pladdrr"))

# Fast path: returns an external pointer, no wrapper object built
pcep_ptr <- to_powercepstrogram_fast(sound, 60, 0.002, 5000, 50)

# Wrap it when you want the method API
pcep <- PowerCepstrogram(.xptr = pcep_ptr)
pcep$get_cppts()

```

to_spectrogram_direct *Create Spectrogram from Sound Directly (returns XPtr)*

Description

Create Spectrogram from Sound Directly (returns XPtr)

Usage

```

to_spectrogram_direct(
  sound,
  window_length = 0.005,
  max_frequency = 5000,
  time_step = 0.002,
  frequency_step = 20,
  window_shape = "Gaussian"
)

```

Arguments

sound	Sound object or external pointer
window_length	Numeric. Window length in seconds (default: 0.005)
max_frequency	Numeric. Maximum frequency in Hz (default: 5000)
time_step	Numeric. Time step in seconds (default: 0.002)
frequency_step	Numeric. Frequency step in Hz (default: 20)
window_shape	Character. Window shape (default: "Gaussian")

Value

External pointer to Spectrogram

Examples

```

sound <- Sound$create_tone(frequency = 200, duration = 0.5)
spg_ptr <- to_spectrogram_direct(sound, window_length = 0.005)
spg <- Spectrogram(.xptr = spg_ptr)
spg$get_number_of_time_bins()

```

to_spectrum_direct	Create Spectrum from Sound Directly (returns XPtr)
--------------------	--

Description

Create Spectrum from Sound Directly (returns XPtr)

Usage

```
to_spectrum_direct(sound, fast = TRUE)
```

Arguments

sound	Sound object or external pointer
fast	Logical. Use fast algorithm (default: TRUE)

Value

External pointer to Spectrum

Examples

```

sound <- Sound$create_tone(frequency = 200, duration = 0.5)
spec_ptr <- to_spectrum_direct(sound)
spec <- Spectrum(.xptr = spec_ptr)
spec$get_number_of_bins()

```

two_pass_adaptive_pitch	Two-Pass Adaptive Pitch Extraction
-------------------------	------------------------------------

Description

Performs a two-pass pitch extraction where the first pass uses a wide range (50-800 Hz by default) to estimate the speaker's pitch distribution, then the second pass uses an adaptive range based on quartiles ($Q1*0.75$ to $Q3*1.5$).

This is a standard technique for robust pitch extraction across speakers with different voice ranges. Returns both the refined pitch contour and the computed range parameters for transparency.

Usage

```
two_pass_adaptive_pitch(
    sound,
    time_step = 0,
    voicing_threshold = 0.45,
    silence_threshold = 0.03,
    initial_floor = 50,
    initial_ceiling = 800,
    octave_cost = 0.01,
    octave_jump_cost = 0.35,
    voiced_unvoiced_cost = 0.14,
    q1_factor = 0.75,
    q3_factor = 1.5,
    method = c("cc", "ac")
)
```

Arguments

sound	Sound object or external pointer
time_step	Time step (0 = auto, typically 0.75/pitch_floor)
voicing_threshold	Voicing threshold (default 0.45)
silence_threshold	Silence threshold (default 0.03)
initial_floor	Initial pitch floor for pass 1 (default 50 Hz)
initial_ceiling	Initial pitch ceiling for pass 1 (default 800 Hz)
octave_cost	Octave cost (default 0.01)
octave_jump_cost	Octave jump cost (default 0.35)
voiced_unvoiced_cost	Voiced/unvoiced transition cost (default 0.14)
q1_factor	Factor to multiply Q1 for min_pitch (default 0.75)
q3_factor	Factor to multiply Q3 for max_pitch (default 1.5)
method	Pitch method: "cc" (cross-correlation, default) or "ac" (autocorrelation)

Value

Named list with: - 'pitch': External pointer to the refined Pitch object - 'min_pitch': Computed minimum pitch ($Q1 * q1_factor$) - 'max_pitch': Computed maximum pitch ($Q3 * q3_factor$) - 'q1': First quartile of pass 1 pitch values - 'q3': Third quartile of pass 1 pitch values

Algorithm

1. Pass 1: Extract pitch with wide range (initial_floor to initial_ceiling)
2. Compute Q1 and Q3 from voiced frames
3. Pass 2: Re-extract with adaptive range ($Q1 * 0.75$ to $Q3 * 1.5$)

Performance

This is a pure R wrapper calling existing direct functions. No C++ overhead beyond the two pitch extractions. Suitable for batch processing.

See Also

[to_pitch_cc_direct()], [to_pitch_ac_direct()] for single-pass extraction [pitch_get_adaptive_range()] for the single-call quartile + range computation [get_pitch_quantiles_batch()] for batch quartile extraction

Examples

```
sound <- Sound$create_tone(frequency = 150, duration = 1.0)

# Basic usage (returns XPtr)
result <- two_pass_adaptive_pitch(sound)
pitch_refined <- Pitch(.xptr = result$pitch)
cat("Adaptive range:", result$min_pitch, "-", result$max_pitch, "Hz\n")

# With custom parameters
result <- two_pass_adaptive_pitch(sound,
  voicing_threshold = 0.6, # Stricter voicing
  q1_factor = 0.7,        # Wider lower bound
  q3_factor = 1.6         # Wider upper bound
)
```

vad

Voice Activity Detection Functions

Description

Functions for detecting voiced and unvoiced segments in audio signals. Used primarily for AVQI implementation to extract voiced segments from continuous speech.

Value

This page documents a family of functions; see the ‘@return’ section of each individual function (e.g. [sound_to_textgrid_silences()]) for its specific return value.

Examples

```
# See individual functions, e.g. ?sound_to_textgrid_silences
```

VocalTract

VocalTract

Description

Praat VocalTract object: the cross-sectional areas of the vocal tract, from glottis to lips, divided into sections.

Arguments

<code>nx</code>	Number of sections. Default 17.
<code>dx</code>	Section length in metres. Default 0.01.
<code>.xptr</code>	Not for direct use. External pointer to the underlying C++ VocalTract object; set internally when a method returns a new VocalTract.

Details

Used for articulatory synthesis (convert to a Spectrum), vowel modeling (create from a phone), and acoustic tube modeling.

Value

A VocalTract object with methods for articulatory tube-model access.

Usage

```
VocalTract(nx, dx) # create an empty vocal tract with nx sections
VocalTract$create_from_phone(phone) # create from a phone name
```

Query methods

- `get_length()` - total length in metres
- `get_number_of_sections()` - number of sections
- `get_section_length()` - section length in metres
- `get_area(section)` - area at a section (m^2)
- `get_areas()` - all areas as a vector

Modification

- `set_area(section, area)` - set the area at a section
- `set_areas(areas)` - set all areas from a vector

Transformation

- `to_spectrum(...)` - convert to a Spectrum for synthesis
- `to_matrix()` - convert to a Matrix

See Also

[Spectrum], [Matrix]

Examples

```
vt <- VocalTract$create_from_phone("a")
print(vt)

areas <- vt$get_areas()

spectrum <- vt$to_spectrum()
```

vocaltract_create_from_phone
Create VocalTract from phone

Description

Create VocalTract from phone

Usage

```
vocaltract_create_from_phone(phone)
```

Arguments

phone	Phone name. Valid phones: • Vowels: a, e, i, o, u • Special vowels: y1, y2, y3, jery • Plosives: p, t, k, x • Syllables: pa, ta, ka, pi, ti, ki, pu, tu, ku
-------	---

Value

VocalTract object

Examples

```
vt_a <- VocalTract$create_from_phone("a")
vt_i <- VocalTract$create_from_phone("i")

# Compare spectra
spec_a <- vt_a$to_spectrum()
spec_i <- vt_i$to_spectrum()
```

with_pladdrr_errors	<i>Run an expression, reclassifying tagged pladdrr C++ errors/warnings.</i>
---------------------	---

Description

Any error or warning whose message matches the "[pladdrr_<class>:<routine>:<param>]<message>" tag is rethrown as a classed R condition. Untagged conditions pass through unchanged.

Usage

```
with_pladdrr_errors(expr)
```

Arguments

expr expression to evaluate

Details

Class hierarchy:

- pladdrr_input_error — invalid argument or precondition failed
- pladdrr_praat_error — Praat-internal failure
- pladdrr_data_loss — output incomplete vs requested range

All inherit from pladdrr_error so a single tryCatch(pladdrr_error = ...) catches them all.

Data-loss warnings additionally attach attr(., "pladdrr_data_loss") to the result, listing every routine that reported missing values during the call.

The reaction to data loss is controlled by options(pladdrr.data_loss =): "warn" (default) raises a classed warning per incident, "error" stops at the first incident, "silent" only records the attribute.

Value

value of expr, possibly with a pladdrr_data_loss attribute.

Examples

```
sound <- Sound$create_tone(frequency = 220, duration = 0.3, sampling_rate =
  16000)
formant <- sound$to_formant_burg()
tryCatch(
  with_pladdrr_errors(
    pladdrr:::formant_get_multiple_formants_at_times(
      formant$.xptr, times = c(0.1, 0.2), formant_numbers = 0L
    )
  ),
  pladdrr_input_error = function(e) message("bad input: ", conditionMessage(e))
)
```

Index

* batch-ops

- sound_load_window, 231
- textgrid_merge, 275
- \$.sound_constructor, 8
- \$.textgrid_constructor, 9
- aggregate_measurements, 9, 38
- amplitude_tier_create, 10
- amplitude_tier_from_point_process, 11
- AmplitudeTier, 11, 189
- analyze_files_parallel, 13
- apply_transform_xptr, 14
- apply_window_xptr, 15
- as.data.frame-missing, 16
- as.data.frame.BarkSpectrogram
 (as-data-frame-missing), 16
- as.data.frame.Cepstrum
 (as-data-frame-missing), 16
- as.data.frame.Cochleagram
 (as-data-frame-missing), 16
- as.data.frame.Discriminant
 (as-data-frame-missing), 16
- as.data.frame.DTW
 (as-data-frame-missing), 16
- as.data.frame.Electroglottogram
 (as-data-frame-missing), 16
- as.data.frame.Formant, 18
- as.data.frame.FormantModeler
 (as-data-frame-missing), 16
- as.data.frame.Intensity, 19
- as.data.frame.KlattGrid
 (as-data-frame-missing), 16
- as.data.frame.LFCC
 (as.data.frame.Pitch), 19
- as.data.frame.LongSound
 (as-data-frame-missing), 16
- as.data.frame.LPC
 (as-data-frame-missing), 16
- as.data.frame.Matrix
 (as-data-frame-missing), 16
- as.data.frame.MelSpectrogram
 (as-data-frame-missing), 16
- as.data.frame.MFCC
 (as.data.frame.Pitch), 19
- as.data.frame.PCA
 (as-data-frame-missing), 16
- as.data.frame.Pitch, 19
- as.data.frame.PointProcess
 (as.data.frame.Pitch), 19
- as.data.frame.PowerCepstrogram
 (as-data-frame-missing), 16
- as.data.frame.praat_formant, 21
- as.data.frame.praat_intensity, 21
- as.data.frame.praat_sound, 22
- as.data.frame.Sound, 23
- as.data.frame.TextGrid
 (as.data.frame.Pitch), 19
- as.data.frame.VocalTract
 (as-data-frame-missing), 16
- autolayer.AmplitudeTier
 (autoplot-methods), 24
- autolayer.BarkSpectrogram
 (autoplot-methods), 24
- autolayer.Cepstrum (autoplot-methods), 24
- autolayer.Cochleagram
 (autoplot-methods), 24
- autolayer.ComplexSpectrogram
 (autoplot-methods), 24
- autolayer.Discriminant
 (autoplot-methods), 24
- autolayer.DTW (autoplot-methods), 24
- autolayer.DurationTier
 (autoplot-methods), 24
- autolayer.Electroglottogram
 (autoplot-methods), 24
- autolayer.Excitation
 (autoplot-methods), 24
- autolayer.Formant (autoplot-methods), 24

- autolayer.FormantGrid
(autoplot-methods), 24
- autolayer.FormantModeler
(autoplot-methods), 24
- autolayer.FormantPath
(autoplot-methods), 24
- autolayer.FormantTier
(autoplot-methods), 24
- autolayer.Harmonicity
(autoplot-methods), 24
- autolayer.Intensity (autoplot-methods),
24
- autolayer.IntensityTier
(autoplot-methods), 24
- autolayer.KlattGrid (autoplot-methods),
24
- autolayer.LFCC (autoplot-methods), 24
- autolayer.LongSound (autoplot-methods),
24
- autolayer.LPC (autoplot-methods), 24
- autolayer.Ltas (autoplot-methods), 24
- autolayer.Matrix (autoplot-methods), 24
- autolayer.MelSpectrogram
(autoplot-methods), 24
- autolayer.MFCC (autoplot-methods), 24
- autolayer.PCA (autoplot-methods), 24
- autolayer.Pitch (autoplot-methods), 24
- autolayer.PitchTier (autoplot-methods),
24
- autolayer.PointProcess
(autoplot-methods), 24
- autolayer.Polygon (autoplot-methods), 24
- autolayer.PowerCepstrogram
(autoplot-methods), 24
- autolayer.PowerCepstrum
(autoplot-methods), 24
- autolayer.Sound (autoplot-methods), 24
- autolayer.Spectrogram
(autoplot-methods), 24
- autolayer.Spectrum (autoplot-methods),
24
- autolayer.TextGrid (autoplot-methods),
24
- autolayer.VocalTract
(autoplot-methods), 24
- autoplot-methods, 24
- autoplot.AmplitudeTier
(autoplot-methods), 24
- autoplot.BarkSpectrogram
(autoplot-methods), 24
- autoplot.Cepstrum (autoplot-methods), 24
- autoplot.Cochleagram
(autoplot-methods), 24
- autoplot.ComplexSpectrogram
(autoplot-methods), 24
- autoplot.Discriminant
(autoplot-methods), 24
- autoplot.DTW (autoplot-methods), 24
- autoplot.DurationTier
(autoplot-methods), 24
- autoplot.Electroglottogram
(autoplot-methods), 24
- autoplot.Excitation (autoplot-methods),
24
- autoplot.Formant (autoplot-methods), 24
- autoplot.FormantGrid
(autoplot-methods), 24
- autoplot.FormantModeler
(autoplot-methods), 24
- autoplot.FormantPath
(autoplot-methods), 24
- autoplot.FormantTier
(autoplot-methods), 24
- autoplot.Harmonicity
(autoplot-methods), 24
- autoplot.Intensity (autoplot-methods),
24
- autoplot.IntensityTier
(autoplot-methods), 24
- autoplot.KlattGrid (autoplot-methods),
24
- autoplot.LFCC (autoplot-methods), 24
- autoplot.LongSound (autoplot-methods),
24
- autoplot.LPC (autoplot-methods), 24
- autoplot.Ltas (autoplot-methods), 24
- autoplot.Matrix (autoplot-methods), 24
- autoplot.MelSpectrogram
(autoplot-methods), 24
- autoplot.MFCC (autoplot-methods), 24
- autoplot.PCA (autoplot-methods), 24
- autoplot.Pitch (autoplot-methods), 24
- autoplot.PitchTier (autoplot-methods),
24
- autoplot.PointProcess
(autoplot-methods), 24

- autoplot.Polygon (autoplot-methods), 24
- autoplot.PowerCepstrogram
 - (autoplot-methods), 24
- autoplot.PowerCepstrum
 - (autoplot-methods), 24
- autoplot.Sound (autoplot-methods), 24
- autoplot.Spectrogram
 - (autoplot-methods), 24
- autoplot.Spectrum (autoplot-methods), 24
- autoplot.TextGrid (autoplot-methods), 24
- autoplot.VocalTract (autoplot-methods), 24

- BarkSpectrogram, 36
- batch_process, 36, 38
- batch_processing, 38
- benchmark_parallel, 38
- build_multiband_harmonicity, 39, 50

- calculate_cppfs_fast, 40, 44
- calculate_cppfs_ultra, 43
- calculate_durations_simd_bridge, 45
- calculate_f0_stats_ultra, 46
- calculate_midpoints_simd_bridge, 47
- calculate_minimum_intensity_ultra, 48
- calculate_multiband_hnr_ultra, 40, 49
- Cepstrum, 50
- cepstrum_plots, 51
- check_audio_quality, 51
- Cochleagram, 53
- ComplexSpectrogram, 54, 252
- create_cepstrum_report, 51, 55
- create_file_list, 38, 56
- create_sound, 57, 67
- create_sound_from_values, 57
- create_window_xptr, 58

- Discriminant, 59, 154
- discriminant_from_matrix, 59, 60
- DTW, 60, 252
- duration_statistics_simd_bridge, 62
- DurationTier, 63

- Electroglottogram, 63
- electroglottogram_create, 64
- Excitation, 65
- extract_formant_parallel, 66
- extract_formants, 67, 92, 103
- extract_intensity, 68
- extract_intensity_parallel, 69
- extract_measurements, 38, 70
- extract_measurements_custom, 38, 71
- extract_pitch, 72
- extract_pitch_parallel, 73
- extract_textgrid_intervals, 74
- extract_voiced_segments, 76
- extract_voiced_segments_ultra, 77

- filter_by_duration_simd_bridge, 79
- Formant, 65, 80, 140, 215, 265
- FormantGrid, 82
- FormantModeler, 81, 83
- FormantPath, 81, 83
- FormantTier, 84
- formanttier_from_formant, 85
- format_quality_report, 86

- generate_noise, 87
- generate_sine_wave, 88
- get_cppfs_fast, 89
- get_duration, 90
- get_durations_batch, 91
- get_formant_at_time, 67, 68, 92
- get_formant_bandwidths_at_times, 93
- get_formant_value_direct, 94
- get_formants_at_times, 95
- get_formants_direct, 96
- get_intensity_at_time, 96
- get_intensity_at_times, 97
- get_intensity_value_direct, 98
- get_interval_predicate, 99
- get_jitter_shimmer_batch, 100
- get_max_intensity, 101
- get_max_pitch, 102
- get_mean_formant, 67, 68, 102
- get_mean_intensity, 103
- get_mean_pitch, 104
- get_min_intensity, 105
- get_min_pitch, 105
- get_n_channels, 106
- get_n_samples, 107
- get_pitch_at_time, 107
- get_pitch_at_times, 108
- get_pitch_mean_direct, 109
- get_pitch_quantile_direct, 109
- get_pitch_quantiles_batch, 110, 110
- get_pitch_stats_direct, 112
- get_pitch_stdev_direct, 113

- get_pitch_strengths_at_times, 113
- get_pitch_value_direct, 114
- get_pointprocess_intervals, 115
- get_pointprocess_nearest_indices, 116
- get_pointprocess_times, 116
- get_sampling_rate, 117
- get_sd_intensity, 118
- get_sound_times_fast, 118
- get_sound_values_fast, 119
- get_sound_values_zerocopy
 - (get_sound_values_fast), 119
- get_spectral_moments_batch, 120
- get_textgrid_interval_stats, 120
- get_textgrid_labels_all, 121
- get_voice_quality_ultra, 122
- Harmonicity, 124
- Intensity, 126, 215
- intensity_tier_to_amplitude_tier, 127
- IntensityTier, 127, 128
- is_fast_access, 128
- is_fast_vector, 129
- is_praat_formant, 68, 129
- is_praat_intensity, 130
- is_praat_pitch, 131
- is_praat_sound, 131
- is_zerocopy_vector (is_fast_vector), 129
- KlattGrid, 132
- klattgrid_create_example, 133
- klattgrid_create_from_vowel, 133
- KlattGrid_createExample
 - (klattgrid_create_example), 133
- KlattGrid_createFromVowel
 - (klattgrid_create_from_vowel), 133
- LFCC, 135, 140
- LongSound, 135, 231
- longsound_get_buffer_size_pref_seconds,
 - 137, 137, 139
- longsound_open, 138
- longsound_set_buffer_size_pref_seconds,
 - 139
- LPC, 81, 135, 139
- Ltas, 141, 254, 257
- ltas_average, 144
- Manipulation, 144
- Matrix, 143, 145, 146, 147
- matrix_create, 146, 147, 148
- matrix_create_simple, 146, 147, 148
- matrix_read, 147
- MelSpectrogram, 148, 150
- MFCC, 61, 149
- mfccs_to_dtw, 150
- multiband_hnr_stats, 40, 50, 151
- pair_files, 152
- pair_sound_textgrid, 153
- PCA, 59, 154
- pca_from_matrix, 154, 155
- Pitch, 61, 125, 127, 155, 189, 204, 215, 265
- pitches_to_dtw, 157
- PitchModule, 158
- PitchTier, 157, 159
- pladdr_simd, 161
- pladdr_threads, 162
- plot.Formant, 163
- plot.Harmonicity, 164
- plot.Intensity, 165
- plot.Ltas, 166
- plot.Matrix, 167
- plot.Pitch, 168, 187
- plot.PointProcess, 169
- plot.PowerCepstrum, 170
- plot.Sound, 171, 187
- plot.Spectrogram, 173
- plot.Spectrum, 174
- plot.TextGrid, 175
- plot_cpp_timeseries, 176
- plot_pitch_intensity, 177
- plot_powercepstrogram, 178
- plot_powercepstrum, 51, 179
- plot_sound_pitch, 180, 186
- plot_spectrogram_formants, 182
- plot_spectrogram_pitch, 183
- plot_textgrid_pitch, 184
- plot_textgrid_sound, 185, 186
- plotting-combined, 186
- plotting-methods, 187
- PointProcess, 125, 157, 187, 265
- Polygon, 189
- PowerCepstrogram, 190, 192
- PowerCepstrum, 191, 254
- pp_get_mean_period_direct, 193, 195
- pp_get_stddev_period_direct, 194
- praat_direct, 195

- praat_eval_matrix, 195
- praat_eval_numeric, 196
- praat_eval_string, 196
- praat_eval_string_array, 197
- praat_eval_vector, 197
- praat_init, 198
- praat_initialized, 198
- praat_list_objects, 199
- praat_object_count, 199
- praat_run_script, 200
- praat_version, 200
- PraatInterpreter, 201
- print.fast_vector, 205
- print.praat_formant, 205
- print.praat_intensity, 206
- print.praat_pitch, 207
- print.praat_sound, 207
- print.zerocopy_vector, 208
- process_sounds_parallel, 209
- read_sound, 67, 210
- set_textgrid_simd_enabled_bridge, 210
- simd_info, 211
- Sound, 61, 81, 125, 127, 135, 137, 140, 143, 150, 157, 189, 192, 204, 212, 231, 252, 254, 265
- sound_as_matrix_fast, 215
- sound_as_matrix_zerocopy (sound_as_matrix_fast), 215
- sound_auto_correlate, 216
- sound_concatenate_all, 216
- sound_create_pure_tone, 217
- sound_create_tone, 218
- sound_create_tone_complex, 219
- sound_deepen_band_modulation, 220
- sound_extract_and_formant, 221
- sound_extract_and_pitch, 222
- sound_extract_part, 223
- sound_extract_parts, 224
- sound_extract_parts_pooled, 226
- sound_filter_pass_hann_band, 227
- sound_filter_stop_hann_band, 228
- sound_from_values, 228
- sound_get_zcr, 229
- sound_lengthen, 230
- sound_load_window, 231
- sound_load_window(), 275
- sound_max, 232
- sound_mean, 233
- sound_min, 233
- sound_pool, 234
- sound_pool_clear (sound_pool), 234
- sound_pool_resize (sound_pool), 234
- sound_pool_stats (sound_pool), 234
- sound_rms, 235
- sound_statistics, 236
- sound_times_fast, 237
- sound_to_formant_batch, 237
- sound_to_intensity_batch, 238
- sound_to_pitch_ac_batch, 239
- sound_to_pitch_batch, 241
- sound_to_pitch_cc_batch, 242
- sound_to_pitch_shs_batch, 243
- sound_to_pitch_spinet_batch, 244
- sound_to_textgrid_silences, 245
- sound_values_fast, 247
- sounds_append, 248
- sounds_convolve, 249
- sounds_cross_correlate, 249
- sounds_to_dtw, 250
- Spectrogram, 61, 143, 251, 254
- spectrograms_to_dtw, 252
- Spectrum, 65, 140, 143, 192, 215, 252, 253
- spectrum_cepstral_smoothing, 255
- spectrum_pass_hann_band, 255
- spectrum_stop_hann_band, 256
- SpectrumTier, 143, 257
- summary.praat_formant, 258
- summary.praat_intensity, 259
- summary.praat_pitch, 259
- summary.praat_sound, 260
- Table, 261, 262
- table_create, 261
- TextGrid, 262, 275
- textgrid_create, 264
- textgrid_extract_intervals_batch, 265
- textgrid_filter_xptr, 266
- textgrid_get_all_labels, 268
- textgrid_get_intervals_where, 269
- textgrid_interval_all_features_batch, 270
- textgrid_interval_formant_batch, 271
- textgrid_interval_intensity_batch, 272
- textgrid_interval_pitch_batch, 273
- textgrid_interval_statistics_batch, 274

textgrid_merge, [275](#)
textgrid_merge(), [231](#)
textgrid_simd_enabled, [276](#)
to_formant_direct, [276](#)
to_harmonicity_direct, [278](#)
to_intensity_direct, [279](#)
to_ltas_direct, [279](#)
to_pitch_ac_direct, [280](#), [283](#)
to_pitch_cc_direct, [281](#), [283](#)
to_pitch_direct, [283](#)
to_pitch_shs_direct, [284](#)
to_pitch_spinet_direct, [285](#)
to_point_process_direct, [195](#), [286](#), [287](#)
to_point_process_from_sound_and_pitch,
 [287](#)
to_powercepstrogram_fast, [288](#)
to_spectrogram_direct, [289](#)
to_spectrum_direct, [290](#)
two_pass_adaptive_pitch, [290](#)

vad, [292](#)
VocalTract, [293](#)
vocaltract_create_from_phone, [294](#)

with_pladdr_errors, [295](#)