

Package ‘tidymedia’

October 11, 2026

Title Media File Preprocessing and Metadata for the 'tidyverse'

Version 0.2.2

Description Batch preprocessing and metadata extraction for audio, video and image files, built on the command-line programs 'FFmpeg' (<<https://ffmpeg.org/>>) and 'MediaInfo' (<<https://mediaarea.net/en/MediaInfo>>). Trim, crop, scale, convert and standardize files one at a time or across a whole directory, and read container and stream metadata back as tibbles for use with the 'tidyverse'.

License GPL-3

URL <https://github.com/jmgirard/tidymedia>,
<https://jmgirard.github.io/tidymedia/>

BugReports <https://github.com/jmgirard/tidymedia/issues>

Depends R (>= 4.1.0)

Imports archive (>= 1.1.1), cli (>= 3.4.0), digest (>= 0.6.37), dplyr (>= 1.1.0), glue (>= 1.6.2), purrr (>= 1.0.0), rappdirs (>= 0.3.3), rlang (>= 1.2.0), tibble (>= 3.1.4), tools, utils, withr (>= 2.5.0)

Suggests furr (>= 0.3.0), future (>= 1.30.0), knitr (>= 1.40), rmarkdown (>= 2.16), roxygen2 (>= 7.2.0), spelling (>= 2.2), testthat (>= 3.0.0)

SystemRequirements FFmpeg (<https://ffmpeg.org/>), MediaInfo (<https://mediaarea.net/en/MediaInfo>)

VignetteBuilder knitr

Encoding UTF-8

Language en-US

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Config/Needs/website pkgdown

NeedsCompilation no

Author Jeffrey Girard [aut, cre] (ORCID:
<https://orcid.org/0000-0002-7359-3746>)

Maintainer Jeffrey Girard <me@jmgirard.com>

Repository CRAN

Date/Publication 2026-10-11 14:30:02 UTC

Contents

tidymedia-package	4
anonymize_video	5
anonymize_video_batch	8
audio_stream	11
compare_videos	13
compare_videos_batch	16
concatenate_videos	18
concatenate_videos_batch	19
convert_audio	20
convert_audio_batch	22
crop_video	24
crop_video_batch	27
extract_audio	30
extract_audio_batch	31
extract_frame	34
extract_frame_batch	35
ffmpeg	36
ffmpeg_codecs	37
ffmpeg_encoders	38
ffmpeg_batch	39
ffmpeg_codec	41
ffmpeg_compile	42
ffmpeg_concat	43
ffmpeg_copy	44
ffmpeg_crop	45
ffmpeg_drawbox	46
ffmpeg_drop	47
ffmpeg_files	48
ffmpeg_fps	49
ffmpeg_hstack	50
ffmpeg_jobs	51
ffmpeg_loudnorm	52
ffmpeg_manifest	54
ffmpeg_map	55
ffmpeg_output_options	56
ffmpeg_overlay	57
ffmpeg_pixel_format	59
ffmpeg_run	60
ffmpeg_scale	62

ffmpeg_seek	63
ffmpeg_trim	64
ffmpeg_vstack	65
ffmpeg_probe	66
find_ffmpeg	67
format_for_web	69
format_for_web_batch	71
get_duration	73
get_frame_rate	74
get_height	75
get_sample_rate	76
get_width	77
hardware_encoder	78
has_encoder	79
install_on_win	80
local_timeout	83
mediainfo	84
mediainfo_parameter	85
mediainfo_query	86
mediainfo_template	87
normalize_audio	89
normalize_audio_batch	92
picture_in_picture	96
picture_in_picture_batch	99
print.tidymedia_ffm	101
probe_all	102
probe_container	103
program_status	105
refresh_ffmpeg_capabilities	106
sample_frames	108
sample_frames_batch	109
segment_video	111
segment_video_batch	114
separate_audio_video	117
separate_audio_video_batch	121
set_program	125
standardize_video	127
standardize_video_batch	130
strip_metadata	133
strip_metadata_batch	134
unset_program	135
verify_media	136
with_timeout	138

Description

tidymedia prepares audio and video files for research. It trims, crops, converts and standardizes files, and reads file details back as tibbles. It runs the programs FFmpeg and MediaInfo. Start with `vignette("tidymedia")`.

Details

Task functions, such as `extract_audio()`, do one common job in one call. Pipeline functions, such as `ffmpeg_files()`, build an FFmpeg command one step at a time. Direct commands, `ffmpeg()`, `ffprobe()` and `mediainfo()`, pass your own arguments to the programs. `probe_all()` and `get_duration()` read file details.

See `vignette("tidymedia")` for the guided tour and a glossary of media terms. The other vignettes are "batch", "metadata", "verification" and "workflow".

Session options

A new option value takes effect at the next call. The workers of a `parallel = TRUE` run use your session's values.

- `options(tidymedia.timeout = 600)` sets a limit, in whole seconds, on each program the package starts. The default, 0, means no limit. See `with_timeout()`.
- `options(tidymedia.check_tracks = FALSE)` turns off the dropped-track warning of `extract_audio()`, `convert_audio()`, `normalize_audio()` and their `_batch` forms. The default is TRUE. The check runs one FFprobe call for each different input, when `run = TRUE` and the call or job row names no `audio_stream`. Turning it off skips those calls. A value other than TRUE or FALSE gives an error that names the option, when the check would run.
- `options(tidymedia.hardware_encoders = "h264_nvenc")` names the hardware video encoders of this computer, so the package does not ask FFmpeg. `character(0)` means none.

Errors when FFmpeg fails

- `tidymedia_ffmpeg_exit`: FFmpeg exited non-zero in `ffmpeg_run()`, or in the loudnorm analysis pass of `normalize_audio(two_pass = TRUE)`. The task functions that run their command with `ffmpeg_run()`, such as `extract_audio()` and `separate_audio_video()`, give it too. The `tm_status` field holds the exit status.
- `tidymedia_loudnorm_no_measurement`: the two-pass analysis gave no measurement. In the `_batch` form, `tm_rows` holds the rows, and `tm_row_status` holds each exit status or NA.
- `tidymedia_multitrack_separation`: `separate_audio_video()` could not write several audio tracks into one audio file. The `_batch` form gives a warning.

- `tidymedia_encoder_unavailable`: this FFmpeg build lists no encoder for a codec the call used. After a failed run, `ffmpeg_run()` adds it before `tidymedia_ffmpeg_exit`, and `ffmpeg_batch()` gives one warning for all its rows. A hardware encoder that `fallback = FALSE` cannot use gives it too. `tm_encoder` holds the names and `tm_stream` says "video" or "audio" for each. "When the build lacks an encoder" on `ffmpeg_run()` says more.

Author(s)

Maintainer: Jeffrey Girard <me@jmgirard.com> ([ORCID](#))

Authors:

- Jeffrey Girard <me@jmgirard.com> ([ORCID](#))

See Also

Useful links:

- <https://github.com/jmgirard/tidymedia>
- <https://jmgirard.github.io/tidymedia/>
- Report bugs at <https://github.com/jmgirard/tidymedia/issues>

anonymize_video

Cover fixed regions of a video with opaque boxes

Description

Anonymize a video by covering one or more fixed rectangular regions with opaque filled boxes. For example, you can redact a face, a name badge, or a screen that stays in one place for the whole clip. The regions are fixed (there is no face or object tracking), so this suits footage where the areas to cover do not move. The glossary in `vignette("tidymedia")` explains media terms such as codec, pixel format and stream copy.

Usage

```
anonymize_video(
  infile,
  outfile,
  regions,
  color = "black",
  video_codec = "libx264",
  audio_codec = "copy",
  pixel_format = "yuv420p",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code>	A string containing the path to a video file.
<code>outfile</code>	A string containing the path of the video file to write.
<code>regions</code>	A data frame with one row per box and columns <code>x</code> , <code>y</code> , <code>width</code> , <code>height</code> (and optionally <code>color</code>); see Details.
<code>color</code>	A string naming the default fill color in FFmpeg color syntax, used for any row without its own color (default "black").
<code>video_codec</code>	A string naming the output video codec (default "libx264"). NULL emits no <code>-codec:v</code> and lets the output container's default encoder decide. NULL is how you opt out of the H.264 default for a container that does not hold it. For a .webm output, pass <code>video_codec = NULL</code> <i>and</i> <code>audio_codec = NULL</code> , because the default <code>audio_codec = "copy"</code> would otherwise carry a codec WebM cannot hold.
<code>audio_codec</code>	A string naming the output audio codec. The default "copy" stream-copies the source audio unchanged. Name a real encoder, such as "aac", when the source audio codec cannot be copied into the output container. NULL emits no <code>-codec:a</code> and lets the container's default encoder decide.
<code>pixel_format</code>	A string naming the output pixel format (default "yuv420p").
<code>hardware</code>	The encoder backend. "none" (default) uses the software <code>video_codec</code> . "nvenc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). A backend uses its own encoder for the family of <code>video_codec</code> . For example, "libx264" becomes "h264_nvenc" or "h264_videotoolbox". See has_hardware_encoder for availability and its caveats. This applies to video only. <code>audio_codec</code> is never hardware-accelerated. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it.
<code>fallback</code>	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE re-encodes with the software <code>video_codec</code> and a message. FALSE (default) aborts instead. This keeps output reproducible by never changing the codec silently. A <code>video_codec</code> in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
<code>quality</code>	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264 and libx265 read it as <code>-crf</code> (0 to 51). The nvenc encoders read it as <code>-cq</code> (0 to 51), and the videotoolbox encoders as <code>-q:v</code> (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as libvpx-vp9, is refused with <code>quality</code> set. So is a <code>video_codec</code> of "copy", or of NULL under <code>hardware = "none"</code> . When <code>fallback = TRUE</code> falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale.

audio_stream	The audio track to carry into the output, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. The every-track family reads NULL this way: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

`regions` is a data frame with one row per box and the columns `x`, `y`, `width`, and `height`. Each value is a pixel number or an FFmpeg expression such as `"in_w/2"`. `x` and `y` give the top-left corner, and `width` and `height` give the size. An optional `color` column overrides the `color` argument for that row. Every box is a solid fill (FFmpeg's `drawbox` with `t=fill`). The function intentionally does not offer hollow outlines.

Because the function applies a filter, it re-encodes the video. The `video_codec` and `pixel_format` arguments set the encoding, and default to H.264 and `yuv420p`. The function floors odd source dimensions to even, so the output always encodes. `yuv420p` and `libx264` require even dimensions, and the step changes nothing for input that is already even. The function stream-copies the audio unchanged (`-c:a copy`) unless `audio_codec` names an encoder. The same input and `regions` therefore always compile to a byte-identical command.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

References

<https://ffmpeg.org/ffmpeg-filters.html#drawbox>

See Also

[ffmpeg_drawbox\(\)](#), the pipeline function it wraps; [has_hardware_encoder\(\)](#) for the hardware argument; [anonymize_video_batch\(\)](#) for the many-file (batch) form.

Other task functions: [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Cover two fixed regions with black boxes
regions <- data.frame(
  x = c(10, 200), y = c(10, 150),
  width = c(120, 80), height = c(90, 60)
)
anonymize_video(video, "anon.mp4", regions, run = FALSE)
# Carry only the second audio track instead of all of them
anonymize_video(video, "anon.mp4", regions, audio_stream = 1, run = FALSE)
```

`anonymize_video_batch` *Anonymize Many Videos From a Jobs Table*

Description

Cover fixed rectangular regions of many input videos with opaque filled boxes from a single jobs tibble. This is the **batch** (table-driven) form of `anonymize_video()`, for when you have more than one video to redact. Each row is one input with its own regions. The required columns name the source (input) and the boxes to cover (regions). This is a thin wrapper over `ffmpeg_batch`. It compiles one reproducible command per input. It shares the same box-fill pipeline (and per-region validation) as `anonymize_video()`. The glossary in `vignette("tidymedia")` explains media terms such as codec, pixel format and stream copy.

Usage

```
anonymize_video_batch(
  jobs,
  color = "black",
  video_codec = "libx264",
  audio_codec = "copy",
  pixel_format = "yuv420p",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per input and (at least) an input column (source path) and a regions list-column. Each regions cell is itself a data frame of boxes for that input. It has the same shape that anonymize_video takes: x, y, width, height and an optional per-box color. An optional output column names the destination. When it is absent, the function derives one per row. It appends <code>_anonymized</code> to each input's basename and keeps the input's extension (e.g. <code>clip.mkv</code> becomes <code>clip_anonymized.mkv</code>). Two rows naming the same output path are refused before any row runs. That is a path repeated in the output column, or a repeated input when there is no output column. Four encoding arguments may also appear as a column: <code>color</code> , <code>video_codec</code> , <code>audio_codec</code> and <code>pixel_format</code> . Such a column overrides the corresponding argument on a per-row basis. Rows (or arguments) that omit the column fall back to the argument's value. In either codec column, NA leaves that row's codec unset. This is the column form of <code>video_codec = NULL / audio_codec = NULL</code> . In a <code>color</code> or <code>pixel_format</code> column NA is an error, because those have no unset state. An <code>audio_stream</code> column overrides the <code>audio_stream</code> argument per row, where NA keeps that row on every audio track. A numeric quality column overrides the quality argument per row (see <code>quality</code>). Any other columns are ignored.
color	A string naming the default fill color (FFmpeg color syntax) applied to every row. A <code>color</code> column in <code>jobs</code> , or a box that supplies its own color, overrides it. (default = "black")
video_codec	A string naming the output video codec applied to every row, unless <code>jobs</code> carries a <code>video_codec</code> column. In that column, NA in a cell leaves that row's codec unset. The default is "libx264". NULL emits no <code>-codec:v</code> and lets the output container's default encoder decide. For a <code>.webm</code> output, pass <code>audio_codec = NULL</code> too, because the default "copy" would otherwise carry a codec WebM cannot hold.
audio_codec	A string naming the output audio codec applied to every row, unless <code>jobs</code> carries an <code>audio_codec</code> column. In that column, NA in a cell leaves that row's codec unset. "copy" (default) stream-copies the audio through untouched. Name an encoder (e.g. "aac") when the source audio cannot be copied into the output container.
pixel_format	A string naming the output pixel format applied to every row, unless <code>jobs</code> carries a <code>pixel_format</code> column. (default = "yuv420p")
hardware	The encoder backend applied to every row. "none" (default) uses the software <code>video_codec</code> . "nvenc" gives NVIDIA GPU encoding (H.264, HEVC and AV1). "videotoolbox" gives Apple GPU encoding (H.264 and HEVC). Batch-wide (a machine property), not a per-row column; a <code>hardware</code> column in <code>jobs</code> is ignored. See has_hardware_encoder . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also be wrong about a per-row value, for example a regions table that is missing a required

	column. The function refuses that call for the value first, whether or not this machine has the encoder.
fallback	A logical applied to every row. When a hardware other than "none" is requested but its encoder is unavailable, TRUE re-encodes with the software video_codec and a message. FALSE (default) aborts instead. It is batch-wide, not a per-row column. A video_codec in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
quality	A number, or NULL (default), applied to each row unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See anonymize_video() for the encoders, their flags and ranges, and the values it refuses.
audio_stream	The audio track to carry into each output, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. Without an audio_stream column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The every-track family reads NULL as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <i>_batch</i> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <i>_batch</i> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from audio_input , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run each input's command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical passed to ffmpeg_batch : anonymize in parallel with furrr (TRUE) or sequentially (FALSE, default). Parallelism follows the active future plan; TRUE under the default sequential plan runs one input at a time and warns. Set a plan first, e.g. <code>future::plan(future::multisession)</code> .
...	Additional arguments forwarded to ffmpeg_batch , such as verify , manifest , checksums , and progress .

Value

The [tibble](#) returned by [ffmpeg_batch](#): jobs with an added command column. When output was derived, it also has the resolved output column. When `run = TRUE`, it has a success column, plus any columns the forwarded arguments add, such as [verified](#).

See Also

[anonymize_video\(\)](#) for the single-input form. [has_hardware_encoder\(\)](#) for the hardware argument. [ffmpeg_batch\(\)](#) for the batch runner and the arguments forwarded through ... [standardize_video_batch\(\)](#) and [segment_video_batch\(\)](#) for the other table-driven functions.

Other task functions: `anonymize_video()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input  = c(video, video),
  output = c("a.mp4", "b.mp4"),
  regions = list(
    data.frame(x = 10, y = 10, width = 120, height = 90),
    data.frame(x = 200, y = 150, width = 80, height = 60)
  )
)
# run = FALSE compiles one command per input without calling FFmpeg
anonymize_video_batch(jobs, run = FALSE)
```

audio_stream

Audio track and audio input indices

Description

Two audio arguments in this package count different things: `audio_stream` and `audio_input`. Both count from 0, so 0 means the first one. This page explains which is which.

The glossary in `vignette("tidymedia")` explains media terms such as stream, container and codec.

Value

This page documents no function and returns no value. It explains two arguments the functions listed under See Also take. Each of those pages says what its own function returns.

The two indices

audio_stream counts **the audio tracks of one input file**. On `extract_audio()`, audio_stream = 1 is the second audio track of the file. Where that track sits among all the streams of the file does not matter. So audio_stream is not the index column of `probe_audio()`, which counts every stream, audio or not.

audio_input counts **the input files of a function**. The functions `compare_videos()` and `picture_in_picture()` combine several files into one output, so they must choose whose sound to keep. On these functions, audio_input = 1 is the second *file*. It says nothing about which track of that file is used.

You cannot work out one index from the other. So the package keeps two names, rather than one argument whose meaning depends on how many inputs a function takes.

What NULL means

audio_stream = NULL still selects audio. It does not mean "no audio". How much audio it selects depends on the function.

- The first-track family reads NULL as the first audio track only: `extract_audio`, `convert_audio` and `normalize_audio`, and their `_batch` forms. The every-track family reads it as every audio track: `separate_audio_video`, `standardize_video`, `anonymize_video`, `crop_video`, `segment_video` and `format_for_web`, and their `_batch` forms.
- The two readings have a reason. A function that writes one audio stream must pick one track when you name none. A function that carries audio through can keep all the tracks its container holds.
- On the functions that pass video through, an input with no audio gives an output with no audio, not an error. On `separate_audio_video()` and `normalize_audio()`, whose output *is* audio, that input gives an FFmpeg error.

audio_input = NULL is different: it selects no audio at all, so the output has **no audio**. A silent output is the default for `compare_videos()` and `picture_in_picture()`. With several inputs, no choice of which one to hear is better than another.

The two arguments also fail in different ways when a number is too large. An audio_input that names an input you did not pass gives an R error, before FFmpeg runs. An audio_stream that names a track the input does not have gives an FFmpeg error. The reason is that the number of tracks is a fact about the file, not about the call.

In a _batch jobs table

On a `_batch` function, both arguments follow one rule. The argument you pass is the default, and a jobs column with the same name overrides it row by row.

This rule is about these two arguments only. The arguments `hardware`, `parallel` and `two_pass` apply to the whole batch, and the function reads no column for them.

If the column is absent, the argument applies to every row. If the column is present, each row uses its own cell. An NA cell means NULL for that row. It does not fall back to the argument. So audio_stream = 2 with an NA cell in an audio_stream column gives that row the NULL reading of its family, not track 2.

The name audio alone is not an index

The pipeline functions use audio for two things that are not counts:

- an audio codec name on `ffmpeg_codec()`, where audio = "aac" names an encoder;
- a logical on `ffmpeg_copy()`, where audio = TRUE copies the audio stream without re-encoding it.

The input index is called `audio_input`, so that its name says what it counts, as `audio_stream` does.

See Also

`extract_audio`, `convert_audio` and `normalize_audio` read NULL as the first audio track. `separate_audio_video`, `standardize_video`, `anonymize_video`, `crop_video`, `segment_video` and `format_for_web` read it as every audio track. `compare_videos()` and `picture_in_picture()` take the input index. `probe_audio()` shows which audio tracks a file has.

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

compare_videos

Build a side-by-side comparison video

Description

Stack two or more videos into a single comparison video. The videos go side-by-side (direction = "horizontal") or one above the other (direction = "vertical"). This is a common need when reviewing annotations or before/after processing. Built on the stacking pipeline functions (`ffmpeg_hstack` / `ffmpeg_vstack`). The glossary in vignette("tidymedia") explains media terms such as codec, encoder and stream copy.

Usage

```
compare_videos(
  infiles,
  outfile,
  direction = c("horizontal", "vertical"),
  resize = TRUE,
  audio_input = NULL,
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code> s	A character vector of two or more video file paths. This function checks every path itself. A path that cannot be found or read aborts naming this function, and the error lists every such path. It is not reported against the internal builder that the path would otherwise reach.
<code>outfile</code>	A string giving the path to write the comparison video to.
<code>direction</code>	Either "horizontal" (side-by-side, the default) or "vertical" (stacked top to bottom).
<code>resize</code>	A logical indicating whether to resize the inputs to share an edge. Only supported for exactly two inputs. (default = TRUE)
<code>audio_input</code>	The input file whose audio to keep, as a number that counts from 0. 0 is the first file you pass and 1 is the second. This counts the function's inputs, not the audio tracks of one input. So it is a different index from <code>audio_stream</code> on the functions that take one input. NULL (default) selects no audio at all, so the output is silent. This differs from <code>audio_stream = NULL</code> , which still selects audio. An input number the call does not have gives an R error, before FFmpeg runs. See audio_stream . (default = NULL)
<code>video_codec</code>	A string naming the output video codec, or NULL (default) to leave it unset. Then the output container's default encoder is used, and the compiled command is the same as one that never named a codec.
<code>audio_codec</code>	A string naming the codec for the carried audio track. "copy" (default) stream-copies it through untouched. Name an encoder, such as "aac", to transcode it. NULL leaves the codec unset, so the output container's default encoder is used. When <code>audio_input</code> is NULL, no audio reaches the output, so nothing is emitted. Naming an encoder in that case is an error.
<code>hardware</code>	The encoder backend. "none" (default) uses the software <code>video_codec</code> . "nvenc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). A backend uses its own encoder for the family of <code>video_codec</code> . For example, "libx264" becomes "h264_nvenc" or "h264_videotoolbox". With the default <code>video_codec = NULL</code> , the H.264 family is assumed. So a non-H.264 container, such as .webm, needs an explicit HEVC- or AV1-family <code>video_codec</code> (AV1 only under "nvenc"). See has_hardware_encoder for availability and its caveats. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it.
<code>fallback</code>	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE encodes in software with a message. FALSE (default) aborts instead. With <code>video_codec = NULL</code> , the fallback leaves the codec unset rather than picking one, so the codec never changes silently. A <code>video_codec</code> in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
<code>quality</code>	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264

and libx265 read it as `-crf` (0 to 51). The nvenc encoders read it as `-cq` (0 to 51), and the videotoolbox encoders as `-q:v` (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as libvpx-vp9, is refused with quality set. So is a `video_codec` of "copy", or of NULL under hardware = "none". When `fallback = TRUE` falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale.

`run` A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

By default the two inputs are resized to share an edge (equal heights for a horizontal stack, equal widths for a vertical one). Resizing currently supports exactly two inputs, so pass `resize = FALSE` to compare more. Audio is dropped unless `audio_input` names an input to carry; a carried track is stream-copied unless `audio_codec` names an encoder.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

See Also

`ffmpeg_hstack()` and `ffmpeg_vstack()`, the pipeline functions it wraps; `has_hardware_encoder()` for the hardware argument; `picture_in_picture()` for inseting instead of stacking.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
compare_videos(c(video, video), "compare.mp4", run = FALSE)
```

compare_videos_batch *Build Many Comparison Videos From a Jobs Table*

Description

Stack videos side by side for many outputs from a single jobs tibble. This is the **batch** (table-driven) form of `compare_videos()`, for when you have more than one comparison to produce. Each row carries an inputs list-column (each cell two or more video paths) plus an output column. This is a thin wrapper over `ffmpeg_batch`: one reproducible stacking command per row, sharing the pipeline with `compare_videos()`. The glossary in `vignette("tidymedia")` explains media terms such as codec, encoder and stream copy.

Usage

```
compare_videos_batch(
  jobs,
  direction = c("horizontal", "vertical"),
  resize = TRUE,
  audio_input = NULL,
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs A data frame with one row per output and (at least) an inputs list-column and an output column (destination path). Each inputs cell is a character vector of **two or more** video paths. Optional direction, resize, audio_input, video_codec, and audio_codec columns override the like-named arguments per row (a row omitting one falls back to the argument). In an audio_input column, NA means "drop audio", the column's way of writing the scalar's NULL. In a video_codec or audio_codec column, it means "leave the codec unset". A numeric quality column overrides the quality argument per row (see quality). Two rows given the same output path are refused before any row runs; other columns are ignored.

direction, resize

Defaults applied to every row lacking the corresponding column. direction is "horizontal" (the default) or "vertical"; a direction column is held to the same two values, per row. See `compare_videos()` for their fuller meaning.

audio_input	The input file whose audio to keep, as a number that counts from 0. 0 is the first file you pass and 1 is the second. This counts the function's inputs, not the audio tracks of one input. So it is a different index from audio_stream on the functions that take one input. NULL (default) selects no audio at all, so the output is silent. This differs from audio_stream = NULL, which still selects audio. An input number the call does not have gives an R error, before FFmpeg runs. Without an audio_input column, the argument applies to every row. An NA cell in that column means NULL for that row, so that output has no audio. Each row's value is validated against that row's input count. See audio_stream . (default = NULL)
video_codec	A string naming the output video codec, applied to every row lacking a video_codec column. NULL (default) leaves it unset, so each output keeps its container's default encoder.
audio_codec	A string naming the codec for the carried audio track, applied to every row lacking an audio_codec column. "copy" (default) stream-copies it. Name an encoder to re-encode it, or NULL to leave the codec unset. A row carrying no audio emits no -codec:a, and naming an encoder on such a row is an error.
hardware, fallback	The encoder backend and its fallback behavior, applied to the whole batch. They are a property of the machine, not of a row, so neither is read as a jobs column. See compare_videos() . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under run = FALSE. The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also contradict itself by naming an audio_codec with no audio carried into the output. Such a call is refused for the contradiction first, whether or not this machine has the encoder. A per-row value error likewise reports ahead of the encoder check. Examples are an audio_input index past that row's input count, and a direction outside the two accepted values. A value error and a contradiction resolve the same way whether the value arrived as an argument or in a jobs column. The contradiction reports first.
quality	A number, or NULL (default), applied to each row unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See compare_videos() for the encoders, their flags and ranges, and the values it refuses.
run	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. verify, manifest, progress).

Value

The jobs tibble with an added command column. When `run = TRUE`, it also has a success column, plus verified or a provenance manifest, each when requested through `...`. See [ffmpeg_batch](#).

See Also

[compare_videos\(\)](#), the one-output function it wraps; [ffmpeg_batch\(\)](#), the batch runner; [has_hardware_encoder\(\)](#) for the hardware argument. [concatenate_videos_batch\(\)](#) and [picture_in_picture_batch\(\)](#), the other batch functions that take several inputs per row.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(inputs = list(c(video, video)), output = "compare.mp4")
compare_videos_batch(jobs, run = FALSE)
```

concatenate_videos	<i>Combine video files using the concat demuxer</i>
--------------------	---

Description

Combine multiple video files one after another without needing to re-encode them by using the [concat demuxer](#). This will be much faster than re-encoding but requires that the files have the same parameters (width, height, etc.) and formats/codecs. To concatenate videos using re-encoding, see the [concat video filter](#). The glossary in `vignette("tidymedia")` explains media terms such as codec and re-encode.

Usage

```
concatenate_videos(infiles, outfile, run = TRUE)
```

Arguments

infiles	A character vector containing the file paths to video files. This function checks every path itself. A path that cannot be found or read aborts naming this function, and the error lists every such path. It is not reported against the internal builder that the path would otherwise reach.
outfile	A string containing the desired file path to write the new, concatenated video file to.
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Value

The compiled FFmpeg command (invisibly when run = TRUE).

See Also

`ffmpeg_concat()`, the pipeline function it wraps.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
concatenate_videos(c(video, video), "joined.mp4", run = FALSE)
```

concatenate_videos_batch

Concatenate Many Videos From a Jobs Table

Description

Join clips end to end for many outputs from a single jobs tibble. This is the **batch** (table-driven) form of `concatenate_videos()`, for when you have more than one concatenation to produce. Unlike the single-input batch functions, each row's inputs are **many**. So jobs carries an inputs list-column (each cell a character vector of source paths) plus an output column. This is a thin wrapper over `ffmpeg_batch`: one reproducible concat-demuxer command per row, sharing the copy + map-0 pipeline with `concatenate_videos()`.

Usage

```
concatenate_videos_batch(jobs, run = TRUE, parallel = FALSE, ...)
```

Arguments

jobs	A data frame with one row per output and (at least) an inputs list-column and an output column (destination path). Each inputs cell is a character vector of the source paths to join, in order. An output column is required; this function derives no destination. Two rows given the same output path are refused before any row runs. Any other columns are ignored.
run	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. verify, manifest, progress).

Value

The jobs tibble with an added command column. When run = TRUE, it also has a success column, plus verified or a provenance manifest, each when requested through See [ffmpeg_batch](#).

See Also

[concatenate_videos\(\)](#), the one-output function it wraps; [ffmpeg_batch\(\)](#), the batch runner; [compare_videos_batch\(\)](#) and [picture_in_picture_batch\(\)](#), the other batch functions that take several inputs per row.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(inputs = list(c(video, video)), output = "joined.mp4")
concatenate_videos_batch(jobs, run = FALSE)
```

convert_audio

Extract or convert a media file's audio track

Description

Write the audio stream of infile into outfile. By default (audio_codec = NULL), the output format follows the outfile file extension, at the highest VBR quality (-q:a 0). For example, an .mp3 extension gives an MP3. Pass audio_codec to set the output audio codec yourself, whatever the extension is. The glossary in `vignette("tidymedia")` explains media terms such as codec and stream.

Usage

```
convert_audio(
  infile,
  outfile,
  audio_codec = NULL,
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code>	A string containing the path to a media file.
<code>outfile</code>	A string containing the path of the audio file to write.
<code>audio_codec</code>	An optional string naming the output audio codec (e.g. "libmp3lame", "aac", "flac"), passed to FFmpeg's <code>-c:a</code> . When NULL (default), FFmpeg infers the codec from the outfile extension and encodes at the highest VBR quality. On the other task functions, NULL means "leave the codec unset". Here NULL does <i>not</i> leave the codec unset. NULL selects <code>-q:a 0</code> .
<code>audio_stream</code>	The audio track to take, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) takes the first audio track. The first-track family reads NULL this way: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The every-track family reads it as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
<code>run</code>	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

When `infile` has more than one audio track, `audio_stream` names which one to take. With no `audio_stream`, the function takes the **first** one.

When no `audio_stream` is named and the input has tracks that the output will not carry, the function warns. The check costs **one FFprobe call per distinct input**, which is one call here, because this function takes a single `infile`. The warning is given when FFprobe is available and the input can be probed. Otherwise the check is skipped silently. It never runs under `run = FALSE`, and never changes the compiled command. Suppress it by naming a track with `audio_stream`, or by class with `suppressWarnings(classes = "tidymedia_dropped_audio")`.

To switch the check off and skip its FFprobe call, use `options(tidymedia.check_tracks = FALSE)` for the session. Use `withr::local_options(tidymedia.check_tracks = FALSE)` for the rest of one function.

Value

The compiled FFmpeg command (invisibly when run = TRUE).

See Also

`ffmpeg_codec()` and `ffmpeg_map()`, the pipeline functions it wraps. `extract_audio()` to copy audio without re-encoding. `convert_audio_batch()` for the many-file form.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
convert_audio(video, "audio.mp3", run = FALSE)
convert_audio(video, "audio.m4a", audio_codec = "aac", run = FALSE)
# Convert the second audio track instead of the first
convert_audio(video, "audio.mp3", audio_stream = 1, run = FALSE)
```

convert_audio_batch	<i>Convert the Audio of Many Files From a Jobs Table</i>
---------------------	--

Description

Extract or re-encode the audio track of many input files, using one jobs table. This is the **batch** form of `convert_audio()`, for when you have more than one file. Each row is one input. The input and output columns are required. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each input. Each command uses the same audio steps as `convert_audio()`, and the function checks each `audio_codec` value in the same way. The glossary in `vignette("tidymedia")` explains media terms such as codec and stream.

Usage

```
convert_audio_batch(
  jobs,
  audio_codec = NULL,
  audio_stream = NULL,
```

```

    run = TRUE,
    parallel = FALSE,
    ...
)

```

Arguments

jobs	A data frame with one row per input. It needs at least an input column (source path) and an output column (destination path). The output column is required . The function cannot name an audio destination for you, because its extension picks the output format. An optional audio_codec column overrides the audio_codec argument per row. There, NA means "use the highest-VBR-quality default". Rows without a value use the argument. An optional audio_stream column overrides the audio_stream argument per row in the same way, and NA keeps that row on the first audio track. The function refuses two rows with the same output path, before any row runs. The function ignores any other columns, with one exception. A format column is an error, not a silent no-op. The package retired that column with the argument of the same name.
audio_codec	The output audio codec applied to every row, unless jobs has an audio_codec column. With NULL (default), FFmpeg infers the codec from each output extension, at the highest VBR quality. Name a codec (e.g. "aac", "flac") to set -c:a.
audio_stream	The audio track to take, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) takes the first audio track. Without an audio_stream column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The first-track family reads NULL as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The every-track family reads it as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. <code>verify</code> , <code>manifest</code> , <code>progress</code>).

Details

When a row names no `audio_stream` and its input has tracks that the output will not carry, the function warns **once** for the whole batch. The warning names every affected row. The check costs **one FFprobe call per distinct input** it has to probe. A repeated input is probed once, and a row that names a track is not probed at all. The warning is given when FFprobe is available and the input can be probed. Otherwise the check is skipped silently. Those probes run **one at a time**,

before any row starts, so `parallel` does not reach them. A sweep long enough to look like a hang reports its progress. The check never runs under `run = FALSE`, never changes any compiled command, and is skipped entirely when every row names a track. Suppress it by class with `suppressWarnings(classes = "tidymedia_dropped_audio")`.

To switch the check off and skip the whole sweep, use `options(tidymedia.check_tracks = FALSE)` for the session. Use `withr::local_options(tidymedia.check_tracks = FALSE)` for the rest of one function.

Value

The jobs tibble with an added `command` column. When `run = TRUE`, it also has a `success` column, plus `verified` or a `provenance` manifest, each when requested through `...`. See [ffmpeg_batch](#).

See Also

[convert_audio\(\)](#), the single-input form it wraps. [ffmpeg_batch\(\)](#), the batch runner. [extract_audio_batch\(\)](#) to stream-copy audio in batch.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(input = c(video, video), output = c("a.mp3", "b.mp3"))
convert_audio_batch(jobs, run = FALSE)
```

crop_video

Crop a video to a rectangular region

Description

Crop a video to a rectangular region. The glossary in `vignette("tidymedia")` explains media terms such as codec, container and stream copy.

Usage

```
crop_video(
  infile,
  outfile,
  width,
  height,
  x = "(in_w-out_w)/2",
  y = "(in_h-out_h)/2",
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code>	A string containing the path to a video file.
<code>outfile</code>	A string containing the path of the video file to write.
<code>width</code>	The width of the output video, in pixels.
<code>height</code>	The height of the output video, in pixels.
<code>x</code>	The horizontal offset, in pixels, of the left edge of the crop. (default = centered)
<code>y</code>	The vertical offset, in pixels, of the top edge of the crop. (default = centered)
<code>video_codec</code>	A string naming the output video codec, or NULL (default) to leave it unset. Then the output container's default encoder is used, and the compiled command is the same as one that never named a codec.
<code>audio_codec</code>	A string naming the output audio codec. "copy" (default) stream-copies the audio through untouched. Name an encoder, such as "aac", to transcode it. NULL leaves the codec unset, so the output container's default encoder is used. Stream-copying fails if the output container cannot hold the source audio codec, for example FLAC in .mp4. In that case, name an encoder.
<code>hardware</code>	The encoder backend. "none" (default) uses the software <code>video_codec</code> . "nvenc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). A backend uses its own encoder for the family of <code>video_codec</code> . For example, "libx264" becomes "h264_nvenc" or "h264_videotoolbox". With the default <code>video_codec = NULL</code> , the H.264 family is assumed. So a non-H.264 container, such as .webm, needs an explicit HEVC- or AV1-family <code>video_codec</code> (AV1 only under "nvenc"). See has_hardware_encoder for availability and its caveats. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it.

fallback	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE encodes in software with a message. FALSE (default) aborts instead. With video_codec = NULL, the fallback leaves the codec unset rather than picking one, so the codec never changes silently. A video_codec in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
quality	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264 and libx265 read it as -crf (0 to 51). The nvenc encoders read it as -cq (0 to 51), and the videotoolbox encoders as -q:v (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as libvpx-vp9, is refused with quality set. So is a video_codec of "copy", or of NULL under hardware = "none". When fallback = TRUE falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale.
audio_stream	The audio track to carry into the output, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. The every-track family reads NULL this way: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <i>_batch</i> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <i>_batch</i> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from audio_input, the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Value

The compiled FFmpeg command (invisibly when run = TRUE).

See Also

[ffmpeg_crop\(\)](#), the pipeline function it wraps; [has_hardware_encoder\(\)](#) for the hardware toggle; [crop_video_batch\(\)](#) for the many-file form.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video_batch\(\)](#),

```
extract_audio(), extract_audio_batch(), format_for_web(), format_for_web_batch(), normalize_audio(),
normalize_audio_batch(), picture_in_picture(), picture_in_picture_batch(), segment_video(),
segment_video_batch(), separate_audio_video(), separate_audio_video_batch(), standardize_video(),
standardize_video_batch()
```

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
crop_video(video, "cropped.mp4", width = 160, height = 120, run = FALSE)
```

crop_video_batch

Crop Many Videos From a Jobs Table

Description

Crop many videos to a rectangular region, using one jobs table. This is the **batch** form of `crop_video()`, for when you have more than one file. Each row is one input. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each input, with the same crop steps as `crop_video()`. This function checks each row's crop size and position before any command runs. So a bad cell is refused with an error that names this function. The glossary in `vignette("tidymedia")` explains media terms such as codec, container and stream copy.

Usage

```
crop_video_batch(
  jobs,
  width = NULL,
  height = NULL,
  x = "(in_w-out_w)/2",
  y = "(in_h-out_h)/2",
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per input. It needs at least an input column, the source path. An optional output column names the destination. Without it, each row's output name adds <code>_cropped</code> to the input's base name and keeps its extension. For example, <code>clip.mp4</code> becomes <code>clip_cropped.mp4</code> . A width,
------	---

height, x or y column overrides that argument for each row. A dimension with no column uses the argument. A video_codec column overrides that argument for each row, and NA leaves the codec unset. That is the column form of the argument's NULL. An audio_codec column works the same way. An audio_stream column overrides that argument for each row, and NA keeps every audio track. That is the column form of that argument's NULL. Two rows with the same destination path are refused before any row runs. That happens with a repeated output, or with a repeated input when there is no output column. A numeric quality column overrides the quality argument per row (see quality). Any other columns are ignored.

width, height	The output crop size in pixels, for every row unless jobs has a column of the same name. Each is required: pass it as an argument or as a column. There is no default crop size.
x, y	The offset in pixels of the crop's left and top edge, for every row unless jobs has a column of the same name. The default centers the crop.
video_codec	A string naming the output video codec, applied to every row lacking a video_codec column. NULL (default) leaves it unset, so each output keeps its container's default encoder.
audio_codec	A string naming the output audio codec, for every row when jobs has no audio_codec column. "copy" (default) stream-copies the audio. Name an encoder to transcode it. NULL leaves the codec unset, so each output keeps its container's default encoder.
hardware, fallback	The encoder backend and its fallback behavior, applied to the whole batch. They are a property of the machine, not of a row, so neither is read as a jobs column. See crop_video() . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under run = FALSE. The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also have a per-row width or height that is neither a positive number nor an FFmpeg expression. Such a call is refused for the value first, whether or not this machine has the encoder.
quality	A number, or NULL (default), applied to each row unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See crop_video() for the encoders, their flags and ranges, and the values it refuses.
audio_stream	The audio track to carry into each output, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. Without an audio_stream column, the argument applies to every row. An NA cell in that column means NULL for that row. It does

not fall back to the argument. The every-track family reads NULL as every audio track: [separate_audio_video](#), [standardize_video](#), [anonymize_video](#), [crop_video](#), [segment_video](#) and [format_for_web](#), and their `_batch` forms. The first-track family reads it as the first audio track only: [extract_audio](#), [convert_audio](#) and [normalize_audio](#), and their `_batch` forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See [audio_stream](#) for how this differs from `audio_input`, the input index on [compare_videos](#) and [picture_in_picture](#). (default = NULL)

<code>run</code>	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
<code>parallel</code>	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffm_batch for the future plan requirement.
<code>...</code>	Additional arguments forwarded to ffm_batch (e.g. <code>verify</code> , <code>manifest</code> , <code>progress</code>).

Value

The jobs tibble with an added `command` column. When `run = TRUE`, it also has a `success` column, plus `verified` or a `provenance manifest`, each when requested through `...`. See [ffm_batch](#).

See Also

[crop_video\(\)](#), the single-input form it wraps; [ffm_batch\(\)](#), the batch runner; [has_hardware_encoder\(\)](#) for the hardware toggle; [standardize_video_batch\(\)](#) to re-encode in batch.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(input = c(video, video), output = c("a.mp4", "b.mp4"),
                      width = c(160, 80), height = c(120, 60))
crop_video_batch(jobs, run = FALSE)
```

extract_audio	<i>Extract the audio stream from a media file</i>
---------------	---

Description

Take one audio track out of infile, and drop the video. When the input has more than one audio track, `audio_stream` names which one to take. With no `audio_stream`, the function takes the **first** audio track. The glossary in `vignette("tidymedia")` explains media terms such as codec, container and stream copy.

Usage

```
extract_audio(
  infile,
  outfile,
  audio_codec = "copy",
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code>	A string containing the path to a media file.
<code>outfile</code>	A string containing the path of the audio file to write.
<code>audio_codec</code>	A string naming the audio codec for the output stream. The default "copy" copies the stream into the new container without re-encoding. NULL writes no -codec:a, so the output container's default encoder decides. That is useful when FFmpeg cannot copy the source codec into the extension you asked for.
<code>audio_stream</code>	The audio track to take, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) takes the first audio track. The first-track family reads NULL this way: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The every-track family reads it as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
<code>run</code>	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

When no `audio_stream` is named and the input has tracks that the output will not carry, the function warns. The check costs **one FFprobe call per distinct input**, which is one call here, because this function takes a single `infile`. The warning is given when FFprobe is available and the input can

be probed. Otherwise the check is skipped silently. It never runs under `run = FALSE`, and never changes the compiled command. Suppress it by naming a track with `audio_stream`, or by class with `suppressWarnings(classes = "tidymedia_dropped_audio")`.

To switch the check off and skip its FFprobe call, use `options(tidymedia.check_tracks = FALSE)` for the session. Use `withr::local_options(tidymedia.check_tracks = FALSE)` for the rest of one function.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

See Also

`ffmpeg_drop()` and `ffmpeg_codec()`, the pipeline functions it wraps. `convert_audio()` to re-encode the extracted audio. `extract_audio_batch()` for the many-file form.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
extract_audio(video, "audio.aac", run = FALSE)
# Take the second audio track instead of the first
extract_audio(video, "audio.aac", audio_stream = 1, run = FALSE)
```

extract_audio_batch	<i>Extract Audio From Many Files From a Jobs Table</i>
---------------------	--

Description

Take the audio track out of many input files, using one jobs table. This is the **batch** form of `extract_audio()`, for when you have more than one file. Each row is one input. The input and output columns are required. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each input, with the same steps as `extract_audio()`: select the audio track and drop the video. The glossary in `vignette("tidymedia")` explains media terms such as codec, container and stream copy.

Usage

```
extract_audio_batch(
  jobs,
  audio_codec = "copy",
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per input. It needs at least an input column (source path) and an output column (destination path). The output column is required . Unlike the video batch functions, this function cannot name an audio destination for you, because the extension is the instruction. The extension picks the container, and with <code>audio_codec = "copy"</code> it must match the source codec. An optional <code>audio_codec</code> column overrides the <code>audio_codec</code> argument per row. Rows without a value use the argument. NA in a cell leaves that row's codec unset, which is the column form of <code>audio_codec = NULL</code> . An optional <code>audio_stream</code> column overrides the <code>audio_stream</code> argument per row in the same way, and NA keeps that row on the first audio track. The function refuses two rows with the same output path, before any row runs. The function ignores any other columns.
audio_codec	The audio codec applied to every row, unless <code>jobs</code> has an <code>audio_codec</code> column. In that column, NA in a cell leaves that row's codec unset. "copy" (default) copies the audio stream with no quality loss. Name an encoder (e.g. "aac") to re-encode. Or pass NULL to write no <code>-codec:a</code> , so the output container's default encoder decides.
audio_stream	The audio track to take, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) takes the first audio track. Without an <code>audio_stream</code> column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The first-track family reads NULL as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The every-track family reads it as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. <code>verify</code> , <code>manifest</code> , <code>progress</code>).

Details

When a row names no `audio_stream` and its input has tracks that the output will not carry, the function warns **once** for the whole batch. The warning names every affected row. The check costs **one FFprobe call per distinct input** it has to probe. A repeated input is probed once, and a row that names a track is not probed at all. The warning is given when FFprobe is available and the input can be probed. Otherwise the check is skipped silently. Those probes run **one at a time, before any row starts**, so parallel does not reach them. A sweep long enough to look like a hang reports its progress. The check never runs under `run = FALSE`, never changes any compiled command, and is skipped entirely when every row names a track. Suppress it by class with `suppressWarnings(classes = "tidymedia_dropped_audio")`.

To switch the check off and skip the whole sweep, use `options(tidymedia.check_tracks = FALSE)` for the session. Use `withr::local_options(tidymedia.check_tracks = FALSE)` for the rest of one function.

Value

The jobs tibble with an added `command` column. When `run = TRUE`, it also has a `success` column, plus `verified` or a `provenance` manifest, each when requested through `...`. See [ffmpeg_batch](#).

See Also

[extract_audio\(\)](#), the single-input form it wraps. [ffmpeg_batch\(\)](#), the batch runner. [convert_audio_batch\(\)](#) to re-encode audio in batch.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(input = c(video, video), output = c("a.aac", "b.aac"))
extract_audio_batch(jobs, run = FALSE)
```

extract_frame	<i>Extract a single frame from a video</i>
---------------	--

Description

Save one frame of a video to an image file, selected either by timestamp or by frame number. Provide exactly one of timestamp or frame.

Usage

```
extract_frame(infile, outfile, timestamp = NULL, frame = NULL, run = TRUE)
```

Arguments

infile	A string containing the path to a video file.
outfile	A string containing the path of the image file to write.
timestamp	Either a number of seconds, a time-duration-syntax string, or NULL. Provide exactly one of timestamp or frame.
frame	Either an integerish frame number or NULL. Provide exactly one of timestamp or frame.
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Value

The compiled FFmpeg command (invisibly when run = TRUE).

See Also

[ffmpeg_seek\(\)](#), the pipeline function it uses to get the frame. [extract_frame_batch\(\)](#) for the many-file (batch) form.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# run = FALSE returns the reproducible command instead of executing it
extract_frame(video, "frame.png", timestamp = 0.5, run = FALSE)
```

extract_frame_batch	<i>Extract Still Frames From Many Videos From a Jobs Table</i>
---------------------	--

Description

Save one still image for each row, across many input files, using one jobs table. This is the **batch** form of `extract_frame()`, for when your frames come from more than one input. Each row is one frame. The required columns name its source and the moment to capture. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each frame. The glossary in `vignette("tidymedia")` explains media terms such as frame rate.

Usage

```
extract_frame_batch(jobs, format = "png", run = TRUE, parallel = FALSE, ...)
```

Arguments

jobs	A data frame with one row per frame. It needs at least an input column (source path). It also needs exactly one of a timestamp column and a frame column. A timestamp holds seconds, or FFmpeg time-duration strings. A frame holds whole frame numbers. The function converts each one to a timestamp with the input's frame rate, as <code>extract_frame</code> does. An optional output column names the destination image. When it is absent, the function derives one per row by appending <code>_<n>.<format></code> to each input's basename. The frame number restarts at 1 for each input file. The function refuses two rows whose destination is the same path, before any row runs. That covers a repeated output, and two derived names that match. For example, <code>clip.mp4</code> and <code>clip.mkv</code> both give <code>clip_1.png</code> . The function ignores any other columns.
format	A string giving the image file extension used when the function derives output. The function ignores it when jobs has an output column. (default = "png")
run	A logical: run each frame's command through FFmpeg (TRUE, default) or only build the commands for inspection (FALSE).
parallel	A logical passed to <code>ffmpeg_batch</code> : save frames in parallel with furrr (TRUE) or one after another (FALSE, default). Parallel work follows the active <code>future</code> plan. TRUE under the default sequential plan runs one frame at a time and warns.
...	Additional arguments forwarded to <code>ffmpeg_batch</code> , such as <code>verify</code> , <code>manifest</code> , <code>checksums</code> , and <code>progress</code> .

Value

The `tibble` returned by `ffmpeg_batch`: jobs with an added command column. When output was derived, it also has the resolved output column. When `run = TRUE`, it has a success column, plus any columns the forwarded arguments add, such as `verified`.

References

<https://ffmpeg.org/ffmpeg-utils.html#time-duration-syntax>

See Also

[extract_frame\(\)](#) for the single-frame form. [ffmpeg_batch\(\)](#) for the batch runner and the arguments passed on through ... [segment_video_batch\(\)](#) for the batch function that cuts segments.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input      = c(video, video),
  output     = c("a.png", "b.png"),
  timestamp  = c(0.25, 0.75)
)
# run = FALSE compiles one command per frame without calling FFmpeg
extract_frame_batch(jobs, run = FALSE)
```

 ffmpeg

Run a raw FFmpeg command

Description

Send a raw argument string to the FFmpeg command-line program. This is a direct command. The function passes the string to FFmpeg unchanged, after the path of the FFmpeg program. So you are responsible for the quoting and the order of the options.

Usage

```
ffmpeg(command)
```

Arguments

command A string containing the arguments to pass to FFmpeg.

Value

A character vector containing the text output by FFmpeg.

See Also

[ffmpeg_codecs\(\)](#) and [ffmpeg_encoders\(\)](#) ask FFmpeg what it supports. The [ffmpeg_*](#)() pipeline functions, such as [ffmpeg_run\(\)](#), are a safer way to build a command.

Other direct command functions: [ffprobe\(\)](#), [mediainfo\(\)](#)

Examples

```
# A direct command: the function passes the string to FFmpeg unchanged
ffmpeg("-version")
```

ffmpeg_codecs

Get a data frame of all installed codecs

Description

Ask FFmpeg for its list of installed codecs, and return the list as a data frame with information about each codec. The glossary in `vignette("tidymedia")` explains media terms such as codec and encoder.

Usage

```
ffmpeg_codecs(sort_by_type = TRUE)
```

Arguments

`sort_by_type` A logical. TRUE sorts the tibble by type and then by name. FALSE sorts it by name only. (default = TRUE)

Value

A [tibble](#) with the following variables:

<code>name</code>	A character vector including the name/code of each codec
<code>details</code>	A character vector including details about each codec
<code>type</code>	A factor vector indicating whether each codec supports "Video", "Audio" or "Subtitles"
<code>decoding</code>	A logical vector indicating whether each codec supports decoding
<code>encoding</code>	A logical vector indicating whether each codec supports encoding
<code>intraframe</code>	A logical vector indicating whether each codec is an intra-frame-only codec
<code>lossy</code>	A logical vector indicating whether each codec supports lossy compression
<code>lossless</code>	A logical vector indicating whether each codec supports lossless compression

See Also

[ffmpeg_encoders\(\)](#) for the encoder list, [ffm_codec\(\)](#) to set a codec in a pipeline, and [ffmpeg\(\)](#) for the direct command.

Other capability functions: [ffmpeg_encoders\(\)](#), [hardware_encoder\(\)](#), [has_encoder\(\)](#), [refresh_ffmpeg_capabilities\(\)](#)

Examples

```
head(ffmpeg_codecs())
ffmpeg_codecs(sort_by_type = FALSE)
```

ffmpeg_encoders	<i>Get a data frame of all installed encoders</i>
-----------------	---

Description

Ask FFmpeg for its list of installed encoders, and return the list as a data frame with information about each encoder. The glossary in `vignette("tidymedia")` explains media terms such as codec and encoder.

Usage

```
ffmpeg_encoders(sort_by_type = TRUE)
```

Arguments

sort_by_type	A logical. TRUE sorts the tibble by type and then by name. FALSE sorts it by name only. (default = TRUE)
--------------	--

Value

A [tibble](#) with the following variables:

name	A character vector including the name/code of each encoder
details	A character vector including details about each encoder
type	A factor vector indicating whether each encoder supports "Video", "Audio" or "Subtitles"
frame_mt	A logical vector indicating whether each encoder supports frame-level multi-threading
slice_mt	A logical vector indicating whether each encoder supports slice-level multi-threading
experimental	A logical vector indicating whether each encoder is experimental
horiz_band	A logical vector indicating whether each encoder supports draw_horiz_band
direct_render	A logical vector indicating whether each encoders supports direct rendering method 1

See Also

[ffmpeg_codecs\(\)](#) for the codec list, [fm_codec\(\)](#) to set a codec in a pipeline, and [ffmpeg\(\)](#) for the direct command.

Other capability functions: [ffmpeg_codecs\(\)](#), [hardware_encoder\(\)](#), [has_encoder\(\)](#), [refresh_ffmpeg_capabilities\(\)](#)

Examples

```
head(ffmpeg_encoders())
ffmpeg_encoders(sort_by_type = FALSE)
```

ffmpeg_batch

Run an FFmpeg Pipeline Over Many Files

Description

Apply a pipeline-building function to every row of a jobs table and compile (and optionally run) the resulting FFmpeg command for each. This is the package's main batch function. It gives one reproducible compiled command per job, collected back into a tibble.

Usage

```
ffmpeg_batch(
  jobs,
  .f,
  ...,
  run = TRUE,
  parallel = FALSE,
  verify = NULL,
  progress = FALSE,
  manifest = FALSE,
  checksums = FALSE
)
```

Arguments

jobs	A data frame with one row per job. Its column names are the arguments passed to .f.
.f	A function that takes a job's columns (by name) and returns an ffmpeg pipeline object.
...	Additional arguments passed on to every call of .f.
run	A logical: run each compiled command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE, a dry run).
parallel	A logical: map over jobs in parallel with furrr (TRUE) or sequentially (FALSE, default). Parallel runs follow the future plan that you set. With TRUE and the default sequential plan, jobs still run one at a time, and you get a warning. Set a plan first, for example <code>future::plan(future::multisession)</code> .
verify	An optional output check applied to each job (only when run = TRUE). Give a named list of expected properties, or a function. A list, for example <code>list(width = 1920)</code> , applies the same checks to every job. A function takes the job columns like .f (called pmap -style) and returns such a list for each job. Each job's output is passed to verify_media . Unlike ffmpeg_run , a failed check is <i>recorded</i> , and does not stop the call. Adds a logical <code>verified</code> column (all checks passed), NA for jobs that did not run successfully.
progress	A logical: display a cli progress bar as the jobs run (TRUE) or run quietly (FALSE, default). Only applies when run = TRUE; safe (a no-op animation) in non-interactive sessions.

manifest	A logical. When TRUE (and run = TRUE), the batch records a provenance manifest and attaches it to the result. The manifest has each job's command, the FFmpeg and FFprobe versions, a timestamp and the output size. Read it with ffmpeg_manifest . (default = FALSE)
checksums	A logical: when TRUE, the manifest also captures md5 checksums of each job's input(s) and output. Ignored unless manifest = TRUE. (default = FALSE)

Details

Each column of jobs is passed by name to `.f` (as [purrr::pmap\(\)](#) does), so a job table with columns `input`, `output` and `start` calls `.f(input = ..., output = ..., start = ...)`. `.f` must return a pipeline (see [ffmpeg_files](#)). Give `.f` a `...` argument if jobs carries columns it does not use.

Two jobs whose pipelines write to the same output path are refused before any job runs, under `run = FALSE` as well as `run = TRUE`. Paths are compared exactly as written. An output that writes no file may repeat. Such outputs are `-` (standard output), a pipe: URL, and an output whose last `-f` option is `-f null`, as `ffmpeg_output_options("-f null")` gives.

[with_timeout\(\)](#) explains how to limit how long R waits for each program in a job, and what happens when a program reaches the limit.

Value

jobs as a [tibble](#) with an added `command` column, which holds the compiled FFmpeg command for each job. When `run = TRUE`, it also has a logical `success` column. When `verify` is supplied, it also has a `verified` column. When `manifest = TRUE`, a provenance manifest is attached as an attribute; read it with [ffmpeg_manifest](#).

When the build lacks an encoder

A job can fail while its pipeline names a codec this FFmpeg build does not list as encodable. The section of the same name on [ffmpeg_run](#) says how that is decided. Such a job still runs, and it still gets `success = FALSE`. After all the jobs run, `ffmpeg_batch()` gives one warning of class `tidymedia_encoder_unavailable` for all such jobs. It names each encoder and the jobs that used it. Its fields hold one element for each job and codec: `tm_encoder`, `tm_stream` and `tm_row`, the job's row number in jobs. A task function's `_batch` form can pass `ffmpeg_batch()` a table of its own, so these can differ from the rows of the table you gave: [separate_audio_video_batch](#) gives `ffmpeg_batch()` two rows for each input, and `normalize_audio_batch(two_pass = TRUE)` leaves out silent inputs.

The same class is also an error, for example when a hardware encoder is missing and `fallback = FALSE`. A `tryCatch()` handler for the class therefore also catches this warning and loses the batch's result. Use `withCallingHandlers()` to act on the warning and keep the result.

If reading FFmpeg's encoder lists reaches the `tidymedia.timeout` limit, the batch gives one warning of class `tidymedia_probe_timeout`. The later jobs do not read the lists again. With `parallel = TRUE`, each worker reads them up to once. That warning comes before the `tidymedia_encoder_unavailable` one, so a `tryCatch()` handler for it also loses the result; use `withCallingHandlers()` for it too.

See Also

`segment_video()`, which is built on `ffmpeg_batch()`; `verify_media()` for the verification spec and `ffmpeg_manifest()` for the provenance manifest.

Other pipeline functions: `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input = c(video, video),
  output = c("a.mp3", "b.mp3")
)
# run = FALSE compiles one command per job without calling FFmpeg
ffmpeg_batch(jobs, run = FALSE, .f = function(input, output, ...) {
  ffmpeg_files(input, output) |>
  ffmpeg_drop("video") |>
  ffmpeg_codec(audio = "libmp3lame")
})
```

ffmpeg_codec

*Set Codecs in an FFmpeg Pipeline***Description**

Set the audio codec, the video codec, or both, for the output file. Use `ffmpeg_codecs()` to see a list of the codecs in your FFmpeg version. The glossary in `vignette("tidymedia")` explains media terms such as codec and stream copy.

Usage

```
ffmpeg_codec(object, audio = NULL, video = NULL)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
audio	A string that names the audio codec, or NULL to set only the video codec. The default is NULL. See audio_stream for the two things that the name audio means in the pipeline functions, and for the input index <code>audio_input</code> .
video	A string that names the video codec, or NULL to set only the audio codec. The default is NULL.

Value

object with an added instruction to change the codecs.

References

<https://ffmpeg.org/ffmpeg-codecs.html>

See Also

`ffmpeg_copy()`, the shortcut for stream copy, `ffmpeg_codecs()` to list the codecs you can use, and `standardize_video()`, a task function built on it.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffm()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_codec(video = "libx264", audio = "aac") |>
  ffmpeg_compile()
```

ffmpeg_compile

Compile the tidymedia pipeline into an FFmpeg command

Description

Compile all the instructions into one string, the FFmpeg command that runs them.

Usage

```
ffmpeg_compile(object)
```

Arguments

object An FFmpeg pipeline (ffm) object created by `ffmpeg_files()`.

Value

A string with the FFmpeg command that carries out all the instructions in the tidymedia pipeline.

See Also

`ffmpeg_run()` to compile and run in one step, and `ffmpeg_batch()` to compile over many files.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffm()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# ffmpeg_compile() returns the reproducible FFmpeg command as a string
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_trim(start = 1, end = 5) |>
  ffmpeg_crop(width = 160, height = 120) |>
  ffmpeg_codec(video = "libx264") |>
  ffmpeg_compile()
```

 ffmpeg_concat

 Concatenate Multiple Inputs in an FFmpeg Pipeline

Description

Join the pipeline's input files one after another with FFmpeg's **concat demuxer**. Like [ffmpeg_hstack](#), this is a pipeline function for several inputs. It uses stream copy, so it is fast and lossless. But every input must share the same parameters, such as codec, resolution and frame rate. The glossary in `vignette("tidymedia")` explains media terms such as codec, re-encode and stream copy.

Usage

```
ffmpeg_concat(object)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> with more than one input file.
--------	--

Details

To join inputs with different parameters, you must re-encode with the concat filter. The package does not wrap that filter yet, so use [ffmpegpeg](#).

The demuxer needs a list file that names the inputs. When you call `ffmpeg_concat()`, it writes one to a temporary path and stores it in the pipeline, so the compiled command can refer to it. It also copies codecs and maps all streams, as [ffmpeg_copy](#) would.

Value

object with an added instruction to concatenate the inputs.

See Also

[concatenate_videos\(\)](#), the task function built on this function.

Other pipeline functions: [ffmpeg_batch\(\)](#), [ffmpeg_codec\(\)](#), [ffmpeg_compile\(\)](#), [ffmpeg_copy\(\)](#), [ffmpeg_crop\(\)](#), [ffmpeg_drawbox\(\)](#), [ffmpeg_drop\(\)](#), [ffmpeg_files\(\)](#), [ffmpeg_fps\(\)](#), [ffmpeg_hstack\(\)](#), [ffmpeg_jobs\(\)](#), [ffmpeg_loudnorm\(\)](#), [ffmpeg_map\(\)](#), [ffmpeg_output_options\(\)](#), [ffmpeg_overlay\(\)](#), [ffmpeg_pixel_format\(\)](#), [ffmpeg_run\(\)](#), [ffmpeg_scale\(\)](#), [ffmpeg_seek\(\)](#), [ffmpeg_trim\(\)](#), [ffmpeg_vstack\(\)](#), [print.tidymedia_ffmpeg\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Join two inputs end-to-end (they must share codec/resolution/frame rate)
ffmpeg_files(c(video, video), "output.mp4") |>
  ffmpeg_concat() |>
  ffmpeg_compile()
```

 ffmpeg_copy

Copy the codecs and map all streams

Description

Copy the audio, the video, or both, with stream copy and no re-encoding. It can also map all streams from the input. This is the fast, lossless path when you only need to put the streams in a new container or cut on keyframes. The glossary in `vignette("tidymedia")` explains media terms such as codec, container, keyframe and stream copy.

Usage

```
ffmpeg_copy(object, audio = TRUE, video = TRUE, streams = TRUE)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
audio	A logical. TRUE (the default) copies the audio codec. See audio_stream for the two things that the name audio means in the pipeline functions, and for the input index <code>audio_input</code> .
video	A logical. TRUE (the default) copies the video codec.
streams	A logical. TRUE (the default) maps all streams from the input. It sets the mapping to the all-streams specifier <code>"0"</code> , and does not add to it. So two <code>ffmpeg_copy()</code> calls compile one <code>-map "0"</code> , not two. If the pipeline already has a <i>different</i> mapping, <code>ffmpeg_copy()</code> gives an error and does not discard that mapping silently. To keep the mapping you set, pass <code>streams = FALSE</code> . Or call <code>ffmpeg_copy()</code> first, and then narrow the mapping with <code>ffmpeg_map(replace = TRUE)</code> .

Value

object with an added instruction to copy codecs, map all streams, or both.

See Also

[ffmpeg_codec\(\)](#) and [ffmpeg_map\(\)](#), which `ffmpeg_copy()` calls, and [segment_video\(\)](#), which uses it for fast copy cuts.

Other pipeline functions: [ffmpeg_batch\(\)](#), [ffmpeg_codec\(\)](#), [ffmpeg_compile\(\)](#), [ffmpeg_concat\(\)](#), [ffmpeg_crop\(\)](#), [ffmpeg_drawbox\(\)](#), [ffmpeg_drop\(\)](#), [ffmpeg_files\(\)](#), [ffmpeg_fps\(\)](#), [ffmpeg_hstack\(\)](#), [ffmpeg_jobs\(\)](#), [ffmpeg_loudnorm\(\)](#), [ffmpeg_map\(\)](#), [ffmpeg_output_options\(\)](#), [ffmpeg_overlay\(\)](#), [ffmpeg_pixel_format\(\)](#), [ffmpeg_run\(\)](#), [ffmpeg_scale\(\)](#), [ffmpeg_seek\(\)](#), [ffmpeg_trim\(\)](#), [ffmpeg_vstack\(\)](#), [print.tidymedia_ffmpeg\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_copy() |>
  ffmpeg_compile()
```

ffmpeg_crop

*Crop Frames in an FFmpeg Pipeline***Description**

Make the video's frames smaller by cropping them.

Usage

```
ffmpeg_crop(object, width, height, x = "(in_w-out_w)/2", y = "(in_h-out_h)/2")
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
width	The width of the output video, in pixels. Give a positive real number or a string that contains an FFmpeg expression.
height	The height of the output video, in pixels. Give a positive real number or a string that contains an FFmpeg expression.
x	The horizontal position of the left edge of the output video, in pixels of the input video. Give a positive real number or a string that contains an FFmpeg expression. The default is <code>"(in_w-out_w)/2"</code> .
y	The vertical position of the top edge of the output video, in pixels of the input video. Give a positive real number or a string that contains an FFmpeg expression. The default is <code>"(in_h-out_h)/2"</code> .

Value

object with an added instruction to crop the frames.

References

<https://ffmpeg.org/ffmpeg-filters.html#crop>

See Also

`ffmpeg_scale()` to resize instead of crop. `crop_video()` and `format_for_web()` are the task functions built on `ffmpeg_crop()`.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Crop to a centered 160x120 region
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_crop(width = 160, height = 120) |>
  ffmpeg_compile()
```

ffmpeg_drawbox
Draw a Colored Box on the Videos in an FFmpeg Pipeline

Description

Add a video filter that draws a colored rectangle on the input video.

Usage

```
ffmpeg_drawbox(
  object,
  x = 0,
  y = 0,
  width = "in_w",
  height = "in_h",
  color = "black",
  thickness = "fill"
)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
x	The horizontal position of the left edge of the box, in pixels of the input video. Give a nonnegative real number or a string that contains an FFmpeg expression. The default is 0.
y	The vertical position of the top edge of the box, in pixels of the input video. Give a nonnegative real number or a string that contains an FFmpeg expression. The default is 0.
width	The width of the box, in pixels. Give a positive real number or a string that contains an FFmpeg expression. The default is "in_w".
height	The height of the box, in pixels. Give a positive real number or a string that contains an FFmpeg expression. The default is "in_h".
color	A string with the color of the box, in FFmpeg color syntax. The reference link below explains that syntax. With the special value "invert", the box has the color of the video with inverted luma. The default is "black".
thickness	The thickness of the box edge, in pixels. The value "fill" draws a filled box. The default is "fill".

Value

object with an added instruction to apply the drawbox filter.

References

<https://ffmpeg.org/ffmpeg-filters.html#drawbox>

<https://ffmpeg.org/ffmpeg-utils.html#color-syntax>

See Also

`anonymize_video()`, the task function that uses `ffmpeg_drawbox()` to fill regions.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Draw a filled red box covering the top-left quarter of the frame
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_drawbox(width = "in_w/2", height = "in_h/2", color = "red") |>
  ffmpeg_compile()
```

 ffmpeg_drop

Drop Streams from an FFmpeg Pipeline

Description

Remove one or more streams from the media file. For example, remove the video, audio, subtitles or data stream from a media file. The glossary in `vignette("tidymedia")` explains media terms such as stream.

Usage

```
ffmpeg_drop(object, streams = c("video", "audio", "subtitles", "data"))
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
streams	A character vector with one or more of these strings: "video", "audio", "subtitles" and "data".

Value

object with an added instruction to drop these streams from the output file when the pipeline runs.

See Also

`extract_audio()`, the task function that uses `ffmpeg_drop()` to drop the video stream.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Drop the audio stream (keep video only)
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_drop(streams = "audio") |>
  ffmpeg_compile()
```

ffmpeg_files

Specify Files in an FFmpeg Pipeline

Description

Start an FFmpeg pipeline by specifying input and output files.

Usage

```
ffmpeg_files(input, output, overwrite = TRUE)
```

Arguments

input	A character vector of paths to the input media files of the pipeline. Give more than one path for stacking.
output	A string with the path of the output media file of the pipeline.
overwrite	A logical. If TRUE (the default), an output media file that already exists is overwritten.

Value

An FFmpeg pipeline object.

See Also

`ffmpeg_compile()` to build the command and `ffmpeg_run()` to run it. The task functions, such as `standardize_video()` and `segment_video()`, are built on the pipeline functions.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_compile()
```

ffmpeg_fps

*Set the Frame Rate in an FFmpeg Pipeline***Description**

Resample the video to a constant frame rate with FFmpeg's fps filter. The filter duplicates or drops frames as needed. It is added to the end of the video filters, like the other filters that take one input. The glossary in vignette("tidymedia") explains media terms such as frame rate.

Usage

```
ffmpeg_fps(object, fps)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by ffmpeg_files().
fps	The target frame rate. Give a positive real number of frames per second, or a string that contains an FFmpeg frame rate expression. For example, "30000/1001" is the NTSC rate.

Value

object with an added instruction to resample the frame rate.

See Also

[standardize_video\(\)](#), the task function that uses ffmpeg_fps() to set the frame rate.

Other pipeline functions: [ffmpeg_batch\(\)](#), [ffmpeg_codec\(\)](#), [ffmpeg_compile\(\)](#), [ffmpeg_concat\(\)](#), [ffmpeg_copy\(\)](#), [ffmpeg_crop\(\)](#), [ffmpeg_drawbox\(\)](#), [ffmpeg_drop\(\)](#), [ffmpeg_files\(\)](#), [ffmpeg_hstack\(\)](#), [ffmpeg_jobs\(\)](#), [ffmpeg_loudnorm\(\)](#), [ffmpeg_map\(\)](#), [ffmpeg_output_options\(\)](#), [ffmpeg_overlay\(\)](#), [ffmpeg_pixel_format\(\)](#), [ffmpeg_run\(\)](#), [ffmpeg_scale\(\)](#), [ffmpeg_seek\(\)](#), [ffmpeg_trim\(\)](#), [ffmpeg_vstack\(\)](#), [print.tidymedia_ffmpeg\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_fps(fps = 30) |>
  ffmpeg_compile()
```

ffmpeg_hstack

*Horizontally Stack Multiple Videos in an FFmpeg Pipeline***Description**

Add a complex video filter that stacks several videos horizontally (side by side). It can also resize the videos to the same height.

Usage

```
ffmpeg_hstack(object, shortest = FALSE, resize = FALSE)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
shortest	A logical that says whether to trim the duration of all videos to that of the shortest video. The default is FALSE.
resize	A logical that says whether to resize the input videos to the same height. Resizing takes longer, and for now it works only with two inputs. It fits both inputs to the same aspect ratio, so it assumes the inputs share one.

Value

object with an added instruction to stack the videos horizontally.

See Also

`ffmpeg_vstack()` for vertical stacking, and `compare_videos()`, the task function built on both.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Stack two inputs side-by-side (pass more than one input to ffmpeg_files())
ffmpeg_files(c(video, video), "output.mp4") |>
  ffmpeg_hstack() |>
  ffmpeg_compile()
```

ffmpeg_jobs

*Build a Jobs Table From a Directory***Description**

List the media files in a directory and return them as a jobs table for `ffmpeg_batch()`. The table is a tibble with one row per file and an input column of full paths. Start a batch here, instead of with your own `list.files()` call.

Usage

```
ffmpeg_jobs(directory, type, extension = NULL, recursive = FALSE)
```

Arguments

directory	A single string naming an existing directory.
type	The media category to list, one of "video", "audio" or "image". You must give it, because it has no default.
extension	An optional character vector of file extensions narrowing the search within type, with or without a leading dot ("mp4" and ".mp4" both work). Each must be one of the extensions type covers, and the error message lists them. NULL (the default) lists every extension of that type.
recursive	A logical: descend into subdirectories (TRUE) or list only the top level (FALSE, default).

Details

The table has only the input column. `ffmpeg_batch()` passes every column of the jobs table to `.f` by name. So if `.f` has no argument for a column, and no `...` argument, the batch stops with R's "unused argument" error.

Add the columns your pipeline needs with the usual data-frame tools. Some `*_batch()` task functions need an output column. Others need columns for their task, such as start and end. The examples below make an output column from input.

`crop_video_batch()` and `extract_audio_batch()` handle the other columns of the table as follows:

- They read a column named like one of their per-row arguments in place of that argument, row by row. Each help page lists these arguments. Examples are a width column in `crop_video_batch()` and an audio_codec column in `extract_audio_batch()`.
- They replace a column named like one that `ffmpeg_batch()` adds. For example, the compiled command replaces a command column. The Value section of `ffmpeg_batch()` lists the added columns.
- They return every other column unchanged.

Value

A [tibble](#) with one row per matching file and one character column, `input`, with each file's full path. Files whose names start with a dot are left out. Rows are in the order `list.files` returns them.

Every row is a path that exists and is not a directory. A subdirectory whose own name ends in a listed extension is never a row. On macOS and Linux, a symbolic link whose target is gone is never a row either. Windows reports such a link as existing, so there it can still be a row. The call gives an error, instead of zero rows, when nothing matches. With `recursive = TRUE` the search follows a symbolic link to a directory, so a row can name a file outside directory.

The scanned extensions include `.mka` as audio and `.ts` as video. `separate_audio_video()` recommends `.mka` or `.m4a` for multi-track audio, and this function reads both as audio, so it lists a folder of that output. Not every container that can hold several audio streams is audio here. `.ts`, like `.mp4` and `.mkv`, is video, so multi-track audio written to one of those is a row under `type = "video"`. The name `.ts` also belongs to TypeScript source files, and this function reads names rather than file contents. A folder of TypeScript sources therefore comes back as video rows when you ask for `type = "video"`.

See Also

`ffmpeg_batch()`, which consumes the returned table.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
folder <- system.file("extdata", package = "tidymedia")
jobs <- ffmpeg_jobs(folder, type = "video")
jobs

# Derive an output column, then hand the whole table to ffmpeg_batch().
# Two inputs sharing a name (a.mp4 and a.mkv) would derive one output here;
# ffmpeg_batch() refuses jobs that share an output before any of them runs.
jobs$output <- file.path(tempdir(), paste0(
  tools::file_path_sans_ext(basename(jobs$input)), ".mp3"
))
ffmpeg_batch(jobs, run = FALSE, .f = function(input, output, ...) {
  ffmpeg_files(input, output) |> ffmpeg_drop("video")
})
```

Description

Add FFmpeg's loudnorm (EBU R128) audio filter. It normalizes the input's perceived loudness toward a target integrated loudness, true-peak ceiling and loudness range. The filter compiles to `-af`, or joins an existing audio filter chain in the order the filters were added. The glossary in `vignette("tidymedia")` explains media terms such as LUFS and true peak.

Usage

```
ffmpeg_loudnorm(  
  object,  
  target_loudness = -23,  
  true_peak = -1,  
  loudness_range = 7,  
  measured_i = NULL,  
  measured_tp = NULL,  
  measured_lra = NULL,  
  measured_thresh = NULL,  
  offset = NULL,  
  linear = FALSE,  
  print_format = NULL  
)
```

Arguments

<code>object</code>	An FFmpeg pipeline (<code>ffmpeg</code>) object created by <code>ffmpeg_files()</code> .
<code>target_loudness</code>	The target integrated loudness, in LUFS (a number in $-70..-5$). The default, <code>-23</code> , is the EBU R128 target.
<code>true_peak</code>	The maximum true peak, in dBTP (a number in $-9..0$). The default, <code>-1</code> , is the EBU R128 ceiling.
<code>loudness_range</code>	The target loudness range, in LU (a number in $1..50$). The default is <code>7</code> .
<code>measured_i</code> , <code>measured_tp</code> , <code>measured_lra</code> , <code>measured_thresh</code>	Measured input values from an earlier loudnorm analysis pass: integrated loudness, true peak, loudness range and threshold. Give them together for an accurate two-pass (linear) correction. Give these values and <code>offset</code> as one set, or give none of them. <code>NULL</code> , the default, gives single-pass dynamic normalization. These map to FFmpeg's <code>measured_I</code> , <code>measured_TP</code> , <code>measured_LRA</code> and <code>measured_thresh</code> options.
<code>offset</code>	The <code>target_offset</code> (offset gain) that the analysis pass reports. It is part of the measured set (see <code>measured_i</code>). The default is <code>NULL</code> .
<code>linear</code>	A logical. <code>TRUE</code> requests linear normalization (<code>linear=true</code>), which needs the measured values to hit the target precisely. <code>FALSE</code> (the default) leaves out the option, so the single-pass dynamic behavior does not change.
<code>print_format</code>	The format of the measurement report for an analysis pass: <code>"json"</code> , <code>"summary"</code> or <code>"none"</code> . <code>NULL</code> (the default) leaves out the option. Use <code>"json"</code> for an analysis pass that a program can parse.

Details

This is single-pass (dynamic) loudnorm. The pipeline stays one reproducible command, with no measurement pass. The defaults follow EBU Recommendation R 128 (2014): `target_loudness = -23 LUFS` and `true_peak = -1 dBTP`. Loudness is measured per ITU-R BS.1770-4. The default `loudness_range = 7` is FFmpeg's own loudnorm default. EBU R128 does not prescribe a single value.

Two filters are added, not one. loudnorm is followed by `asetnsamples`, which regroups the filtered audio into frames of 4096 samples and does not pad the last one. Dynamic loudnorm resamples to 192 kHz and gives frames of 192000 samples. Some encoders accept whatever frame they are given, FLAC and Vorbis among them. Even those encoders refuse to open at all on frames of 192000 samples.

Value

object with an added instruction to normalize loudness.

References

EBU Recommendation R 128 (2014), *Loudness normalisation and permitted maximum level of audio signals*; ITU-R BS.1770-4. <https://ffmpeg.org/ffmpeg-filters.html#loudnorm>

See Also

`normalize_audio()`, the task function built on this filter.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_loudnorm() |>
  ffmpeg_compile()
```

 ffmpeg_manifest

 Read a Batch Provenance Manifest

Description

Retrieve the reproducibility manifest recorded by `ffmpeg_batch()` when it was run with `manifest = TRUE`. The manifest is a tibble with one row per job. Each row has the compiled command and the FFmpeg and FFprobe versions used. It also has the time of the run, the input and output paths, and the output file size. When the batch ran with `checksums = TRUE`, each row also has md5 checksums of the input and output files. With the manifest, a batch run leaves a record that others can check.

Usage

```
ffmpeg_manifest(x, path = NULL)
```

Arguments

x A tibble returned by `ffmpeg_batch()` with `manifest = TRUE`.

path An optional file path. When supplied, the manifest is also written there as CSV (via `utils::write.csv`, no row names) and returned invisibly.

Value

The manifest [tibble](#); invisibly when path is written.

See Also

[ffmpeg_batch\(\)](#), which records the manifest.

Other verification functions: [verify_media\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(input = video, output = tempfile(fileext = ".mp3"))
res <- ffmpeg_batch(jobs, manifest = TRUE, .f = function(input, output, ...) {
  ffmpeg_files(input, output) |>
  ffmpeg_drop("video") |>
  ffmpeg_codec(audio = "libmp3lame")
})
ffmpeg_manifest(res)
```

ffmpeg_map

Set the Stream Mapping in an FFmpeg Pipeline

Description

Choose which input streams go into the output, with FFmpeg's `-map` option. The default, `"0"`, maps every stream from the first input. The glossary in `vignette("tidymedia")` explains media terms such as stream.

Usage

```
ffmpeg_map(object, mapping = "0", replace = FALSE)
```

Arguments

object An FFmpeg pipeline (`ffmpeg`) object created by `ffmpeg_files()`.

mapping A character vector of one or more stream specifiers. Each one adds one `-map`.

replace A logical. `TRUE` discards any mapping already set on object. `FALSE` (the default) adds to it.

Details

mapping can be a character vector. Each element adds one `-map`, in the order given. For example, `ffmpeg_map(object, c("0:v", "0:a:1"))` keeps the video and the *second* audio track of the input.

A second `ffmpeg_map()` call **adds** to the maps already set. It does not replace them. Pass `replace = TRUE` to discard them instead. That is how you narrow the all-streams map that `ffmpeg_copy` sets. Adding to that map puts the stream in the output twice, and does not select it.

`ffmpeg_map()` is the only pipeline function that adds to earlier calls. Every other `ffmpeg_*` function that sets a value, `ffmpeg_copy` included, replaces it. `ffmpeg_map()` is different because its arguments are *partial* choices that combine. For example, you keep the video, then name one audio track.

When the pipeline uses a function with several inputs, such as `ffmpeg_hstack`, your mapping is added *beside* the automatic `-map "[vout]"` of the filtered stream. For example, `ffmpeg_map(object, "0:a")` keeps the audio of the first input next to the stacked video.

Value

object with an added instruction to map streams.

See Also

`ffmpeg_copy()`, which maps all streams, and `separate_audio_video()`, a task function built on `ffmpeg_map()`.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia-ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_map(mapping = "0") |>
  ffmpeg_compile()

# Keep the video and the second audio track only
ffmpeg_files(video, "output.mkv") |>
  ffmpeg_map(mapping = c("0:v", "0:a:1")) |>
  ffmpeg_compile()
```

ffmpeg_output_options	Add Raw Output Options to an FFmpeg Pipeline
-----------------------	--

Description

Add one or more raw FFmpeg output options to the pipeline, after any added before. Output options are the flags after the input and before the output file. Use this function for an option that has no pipeline function of its own. `ffmpeg_compile()` still decides where the options go and how the rest of the command is quoted. So this is not the same as writing the command string yourself.

Usage

```
ffmpeg_output_options(object, ...)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
...	One or more strings. Each string is a group of options separated by white space, for example <code>"-q:v 1"</code> or <code>"-frames:v 1"</code> . They are added in the order given. When the command runs, each word between white space becomes one FFmpeg argument. So an option value must not contain spaces.

Value

object with the added output options.

See Also

`ffmpeg()`, the direct command that takes any FFmpeg arguments, and `ffmpeg_compile()`, which places these options.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Extract a single frame by adding a raw output option
ffmpeg_files(video, "frame.png") |>
  ffmpeg_output_options("-frames:v 1") |>
  ffmpeg_compile()
```

```
ffmpeg_overlay
```

Overlay One Video on Another in an FFmpeg Pipeline

Description

Draw the second input (the overlay) on top of the first input (the main video) at position `x` and `y`. Like `ffmpeg_hstack`, this is a pipeline function for several inputs. It forces the `-filter_complex` path and manages its own stream labels internally. It needs exactly two inputs. The first is the background, and the second is drawn over it. The glossary in `vignette("tidymedia")` explains media terms such as stream.

Usage

```
ffmpeg_overlay(object, x = 0, y = 0, shortest = FALSE, scale = NULL)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> with exactly two input files.
x	The horizontal position of the overlay's left edge, as a number of pixels or an FFmpeg expression. The default is 0.
y	The vertical position of the overlay's top edge, as a number of pixels or an FFmpeg expression. The default is 0.
shortest	A logical that says whether to end the output when the shorter input ends. The default is FALSE.
scale	An optional fraction ($0 < \text{scale} \leq 1$). Before the overlay is drawn, it is resized to scale times the main video's width, and its aspect ratio is kept. NULL (the default) draws the overlay at its own size. When scale is set, <code>overlay_w</code> and <code>overlay_h</code> in <code>x</code> and <code>y</code> refer to the resized overlay.

Details

`x` and `y` accept plain numbers or FFmpeg overlay expressions. A plain number counts pixels from the top-left of the main video. In an expression, `main_w` and `main_h` are the main video's dimensions. `overlay_w` and `overlay_h` are the overlay's dimensions. For example, `x = "main_w-overlay_w-16"` puts the overlay 16 pixels from the right edge.

When `scale` is set, the overlay is first resized to a fraction of the main video's width, and its aspect ratio is kept. The task function `picture_in_picture` uses this resize. Otherwise, to resize the overlay yourself, filter it in a separate pipeline first.

Value

object with an added instruction to draw the second input on the first.

See Also

`picture_in_picture()`, the task function built on this function.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Draw the second input over the first, 16px in from the top-right corner
ffmpeg_files(c(video, video), "output.mp4") |>
  ffmpeg_overlay(x = "main_w-overlay_w-16", y = 16) |>
  ffmpeg_compile()
```

ffmpeg_pixel_format	<i>Set the Pixel Format in an FFmpeg Pipeline</i>
---------------------	---

Description

Set the pixel format of the output with FFmpeg's `-pix_fmt` option. For example, use "yuv420p" for broad player compatibility. The glossary in `vignette("tidymedia")` explains media terms such as pixel format.

Usage

```
ffmpeg_pixel_format(object, format)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
format	A string that names the pixel format of the output file.

Value

object with an added instruction to set the pixel format.

See Also

`standardize_video()` and `format_for_web()`, the task functions that use `ffmpeg_pixel_format()` to set the pixel format.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_pixel_format("yuv420p") |>
  ffmpeg_compile()
```

ffmpeg_run

*Run the FFmpeg Pipeline***Description**

Compile the instructions in the pipeline and run them all through FFmpeg.

Usage

```
ffmpeg_run(object, verify = NULL)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
verify	An optional named list of the properties you expect the output to have, for example <code>list(width = 1920, video_codec = "h264")</code> . It is passed to verify_media . After a successful run, the output is probed. If a check fails, <code>ffmpeg_run()</code> gives an error with the failed checks. It also gives an error when FFmpeg exits non-zero. NULL (the default) skips the checks.

Value

FFmpeg's standard output as a character vector, returned invisibly. On a non-zero exit it has a `status` attribute. You call `ffmpeg_run()` to write the output file, not for its return value. The pipeline runs as a vector of arguments and never through a shell. So paths with spaces or special characters are safe.

When FFmpeg exits non-zero

If FFmpeg refuses a run, `ffmpeg_run()` gives an error of class `tidymedia_ffmpeg_exit`. A caller can catch a failed run without reading the error text:

```
tryCatch(
  ffmpeg_run(pipeline),
  tidymedia_ffmpeg_exit = function(cnd) cnd$tm_status
)
```

The `tm_status` field is one integer, the exit status exactly as `system2()` reported it. If a signal stopped FFmpeg, the field holds the shell's number, 128 plus the signal number, unchanged. That number stands for the signal, not for a status FFmpeg chose to return.

Two other paths give this class and carry this field, so one handler covers all three:

- the loudnorm analysis pass of `normalize_audio(two_pass = TRUE)`, when FFmpeg exits non-zero.
- the error about several audio tracks that [separate_audio_video](#) adds to a failed audio output.

Each of those two paths also gives a second, narrower class before this one. In the same order, they are `tidymedia_loudnorm_no_measurement` and `tidymedia_multitrack_separation`. Catch that class when you want only that failure.

Two related paths do **not** give this class, each for its own reason:

- `normalize_audio(two_pass = TRUE)` also gives an error when the analysis pass exits zero and prints no measurement block that can be read. FFmpeg did not exit non-zero there. So that error has only the class `tidymedia_loudnorm_no_measurement`, and no `tm_status`.
- `normalize_audio_batch(two_pass = TRUE)` reports in one error every row that failed in its analysis phase. The failed rows can include rows that exited zero and rows that FFmpeg refused. So a non-zero exit is one of its causes, not the fact it reports, and no single status can stand for the mix. It also has only the class `tidymedia_loudnorm_no_measurement`. It carries `tm_rows`, the failed rows counted from 1. It also carries `tm_row_status`, their exit statuses in the same order, with NA where a row exited zero.

So `tidymedia_loudnorm_no_measurement` is the one class that covers the analysis pass in both forms.

When the build lacks an encoder

After FFmpeg exits non-zero, `ffmpeg_run()` looks at the video and audio codecs that `ffmpeg_codec` set on the pipeline. A name is encodable if `ffmpeg_encoders` lists it, or if `ffmpeg_codecs` lists it as a codec this build can encode, such as "h264". If a codec is not encodable, the error also has the class `tidymedia_encoder_unavailable`, before `tidymedia_ffmpeg_exit`. It carries `tm_encoder`, the names, and `tm_stream`, which is "video" or "audio" for each name.

The message says that this build lists no such encoder. It does not say that the missing encoder made the run fail. FFmpeg uses no video encoder for an input that has no video, so a run can fail for another reason. When a task function such as `standardize_video` runs its command, the error names that function.

The check runs only after a failed run. A successful run, and a call with `run = FALSE`, never start it. It skips "copy", a codec the pipeline does not set, the hardware encoders that `hardware_encoder` names, and encoders given through `ffmpeg_output_options`. It asks FFmpeg for the two lists, and the package remembers each list for the session once it reads it (see `refresh_ffmpeg_capabilities`). If the package cannot read a list, the check says nothing, and the error is the `tidymedia_ffmpeg_exit` error alone. If reading a list reaches the `tidymedia.timeout` limit, a warning of class `tidymedia_probe_timeout` also says so, once per call. `ffmpeg_batch` gives one `tidymedia_encoder_unavailable` warning for all its rows instead of an error for each.

See Also

`ffmpeg_compile()` to get the command without running it, `ffmpeg_batch()` to run many files, and `verify_media()` for the verify list.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
out <- tempfile(fileext = ".mp4")
ffmpeg_files(video, out) |>
  ffmpeg_scale(width = 160, height = 120) |>
  ffmpeg_codec(video = "libx264", audio = "aac") |>
  ffmpeg_run(verify = list(width = 160, height = 120))
```

 ffmpeg_scale

Scale (Resize) Frames in an FFmpeg Pipeline

Description

Scale (resize) the input video's frames to a width and height in pixels, or with an FFmpeg expression.

Usage

```
ffmpeg_scale(object, width, height)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
width	The width of the output video, in pixels. Give a positive real number or a string that contains an FFmpeg expression.
height	The height of the output video, in pixels. Give a positive real number or a string that contains an FFmpeg expression.

Value

object with an added instruction to resize the frames.

See Also

`ffmpeg_crop()` to crop instead of resize. `standardize_video()` is the task function built on `ffmpeg_scale()`.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_scale(width = 160, height = 120) |>
  ffmpeg_compile()
```

Description

Keep one continuous section of the input with FFmpeg's fast `-ss` and `-to` seek options. It does not use the *trim filter* of `ffmpeg_trim`. Unlike the filter, seeking can use stream copy, so it is the tool for fast, lossless cuts. The glossary in `vignette("tidymedia")` explains media terms such as keyframe, re-encode and stream copy.

Usage

```
ffmpeg_seek(object, start = NULL, end = NULL, reencode = TRUE)
```

Arguments

<code>object</code>	An FFmpeg pipeline (<code>ffmpeg</code>) object created by <code>ffmpeg_files()</code> .
<code>start</code>	The start of the kept section, in seconds or in FFmpeg time duration syntax. <code>NULL</code> keeps the section from the beginning.
<code>end</code>	The end of the kept section, in seconds or in FFmpeg time duration syntax. <code>NULL</code> keeps the section to the end.
<code>reencode</code>	A logical. <code>TRUE</code> (the default) re-encodes for a frame-accurate cut. <code>FALSE</code> makes a fast seek that is safe to copy, and its cut points move to keyframes.

Details

The `reencode` argument trades accuracy against speed:

- `reencode = TRUE` (the default) is **frame-accurate**. The section is re-encoded, so it starts and ends on the exact frames you ask for. This is the safe default.
- `reencode = FALSE` is a **fast, lossless copy**. But the cut points move to the nearest keyframes. So the output duration can differ from the request by up to the gap between two keyframes. Use it with `ffmpeg_copy` for the fastest path.

Value

object with an added instruction to cut the input by seeking.

References

<https://ffmpeg.org/ffmpeg.html#Main-options>

See Also

`ffmpeg_trim()` for the filter that cuts, `ffmpeg_copy()` for the fast copy path, and `segment_video()`, the task function built on it.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_trim()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Fast, lossless copy cut (snaps to keyframes)
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_seek(start = 1, end = 5, reencode = FALSE) |>
  ffmpeg_copy() |>
  ffmpeg_compile()
```

ffmpeg_trim

Trim the Duration of the FFmpeg Pipeline

Description

Trim the input so that the output keeps one continuous part of the input. If `start` is `NULL`, the kept section starts at the beginning of the input. If both `end` and `duration` are `NULL`, the kept section ends at the end of the input. The glossary in `vignette("tidymedia")` explains media terms such as stream copy.

Usage

```
ffmpeg_trim(
  object,
  start = NULL,
  end = NULL,
  duration = NULL,
  units = c("tds", "pts", "frame"),
  setpts = TRUE
)
```

Arguments

<code>object</code>	An FFmpeg pipeline (<code>ffmpeg</code>) object created by <code>ffmpeg_files()</code> .
<code>start</code>	The time of the start of the kept section, given in units. The frame at this time is the first frame of the output.
<code>end</code>	The time of the first frame that is dropped, given in units. The frame just before it is the last frame of the output.
<code>duration</code>	The maximum duration of the output, given in time duration syntax.

units	A string that says how start and end are given: time duration syntax ("tds"), timebase units ("pts") or frame numbers ("frame"). The default is "tds".
setpts	A logical that says whether the output timestamps change to start at zero. If TRUE, a setpts filter is added after the trim.

Value

object with added instructions to trim the duration.

References

<https://ffmpeg.org/ffmpeg-filters.html#trim>

<https://ffmpeg.org/ffmpeg-utils.html#time-duration-syntax>

See Also

`ffmpeg_seek()`, the faster cut by seeking, which can use stream copy. `ffmpeg_trim()` is the filter that cuts on exact frames.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_vstack()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffmpeg_files(video, "output.mp4") |>
  ffmpeg_trim(start = 1, end = 5) |>
  ffmpeg_compile()
```

ffmpeg_vstack

Vertically Stack Multiple Videos in an FFmpeg Pipeline

Description

Add a complex video filter that stacks several videos vertically (one above the other). It can also resize the videos to the same width.

Usage

```
ffmpeg_vstack(object, shortest = FALSE, resize = FALSE)
```

Arguments

object	An FFmpeg pipeline (ffmpeg) object created by <code>ffmpeg_files()</code> .
shortest	A logical that says whether to trim the duration of all videos to that of the shortest video. The default is FALSE.
resize	A logical that says whether to resize the input videos to the same width. Resizing takes longer, and for now it works only with two inputs. It fits both inputs to the same aspect ratio, so it assumes the inputs share one.

Details

This is the vertical form of `ffmpeg_hstack`. Both are pipeline functions for several inputs. They force the `-filter_complex` path and manage their own stream labels internally. The glossary in `vignette("tidymedia")` explains media terms such as stream.

Value

object with an added instruction to stack the videos vertically.

See Also

`ffmpeg_hstack()` for horizontal stacking, and `compare_videos()`, the task function built on both.

Other pipeline functions: `ffmpeg_batch()`, `ffmpeg_codec()`, `ffmpeg_compile()`, `ffmpeg_concat()`, `ffmpeg_copy()`, `ffmpeg_crop()`, `ffmpeg_drawbox()`, `ffmpeg_drop()`, `ffmpeg_files()`, `ffmpeg_fps()`, `ffmpeg_hstack()`, `ffmpeg_jobs()`, `ffmpeg_loudnorm()`, `ffmpeg_map()`, `ffmpeg_output_options()`, `ffmpeg_overlay()`, `ffmpeg_pixel_format()`, `ffmpeg_run()`, `ffmpeg_scale()`, `ffmpeg_seek()`, `ffmpeg_trim()`, `print.tidymedia_ffmpeg()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Stack two inputs one above the other (pass more than one input to ffmpeg_files())
ffmpeg_files(c(video, video), "output.mp4") |>
  ffmpeg_vstack() |>
  ffmpeg_compile()
```

ffprobe

Send a command to the FFprobe program

Description

`ffprobe()` runs the FFprobe program with the arguments in `command` and returns its output. FFprobe reads information about media files.

Usage

```
ffprobe(command)
```

Arguments

command A string with the arguments to give FFprobe.

Details

ffprobe() is a direct command. The package passes command to FFprobe exactly as you wrote it, so you must add any quotes that it needs. To get tibbles instead, use [probe_all\(\)](#) and the other probe_*() functions. These functions quote their arguments for you.

Value

A character vector with the text that FFprobe writes to standard output, one element for each line. Messages on standard error, such as FFprobe's banner and errors, are not returned. On macOS and Linux, a shell redirect such as 2>&1 in command returns them too.

See Also

[probe_all\(\)](#) and the other probe_*() functions, which return tibbles.

Other direct command functions: [ffmpeg\(\)](#), [mediainfo\(\)](#)

Examples

```
ffprobe("--version")
```

find_ffmpeg	<i>Find the location of a dependency program</i>
-------------	--

Description

Each of these functions returns the location of one program as a string: [find_ffmpeg\(\)](#), [find_ffprobe\(\)](#), [find_ffplay\(\)](#) and [find_mediainfo\(\)](#).

Usage

```
find_ffmpeg()

find_mediainfo()

find_ffprobe()

find_ffplay()
```

Details

The function looks on the PATH first. If the program is not there, the function reads the location that [set_program\(\)](#) saved. That location is in a file under tools::R_user_dir("tidymedia", "config"). If neither place has the program, the function gives a warning and returns NULL.

Value

The location of the program as a string, or NULL when it could not be found.

Problems with a saved location

A saved location that no longer works gives a warning, and the function returns NULL. The warning has one of three classes that you can catch:

- `tidymedia_location_gone`: the function read the location, but no program is there now. The condition holds the program name in `tm_program` and the location in `tm_location`, as the file holds it.
- `tidymedia_location_not_executable`: a file is at the saved location, and it cannot be run. Give it permission to execute, or use the `set_*`() function to point at another copy. The condition holds the program name in `tm_program` and the location in `tm_location`, as the file holds it. This case does not arise on Windows, which has no execute bit. There a location is read the way R's own `Sys.which()` reads it, except that a folder never counts and a leading `~` is expanded.
- `tidymedia_location_unreadable`: the file does not hold exactly one location. It is empty, has more than one line, or has one empty line. The condition holds the program name in `tm_program` and the file in `tm_file`.

A line that holds only spaces counts as a location. So it gives `tidymedia_location_gone`, not `tidymedia_location_unreadable`.

To fix any of the three, forget the location with `unset_program()`, or replace it with `set_program()`. A file that cannot be run can also be fixed by giving it permission to execute.

Locations saved by earlier versions

Versions of the package before 0.2.0 saved locations in a different folder, `rappdirs::user_config_dir("tidymedia", "R")`. The functions still read a file in that folder. They read it only when the current folder has no file for the program. `program_status()` looks in the same places, in the same order.

A `tidymedia_location_unreadable` warning names the file that the function read. That file can be the one in the old folder.

`set_program()` writes to the current folder only. After it writes a file for a program, the old file for that program is not read.

`unset_program()` removes the file in both folders. So the old location does not come back after the current file is gone.

See Also

`set_program()` to save the location of a program that is not on the PATH, and `install_on_win()` to download FFmpeg on Windows.

Other program management functions: `install_on_win()`, `program_status()`, `set_program()`, `unset_program()`

Examples

```
# Returns the path to the binary, or NULL with a warning if it is not found
find_ffmpeg()
find_mediainfo()
```

format_for_web

Re-encode a video for web playback

Description

Re-encode a video into a widely compatible, web-friendly form: H.264 video with yuv420p and +faststart, and AAC audio. Odd dimensions are padded down to even values, as the codec requires. The glossary in `vignette("tidymedia")` explains media terms such as codec, pixel format and re-encode.

Usage

```
format_for_web(
  infile,
  outfile,
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code>	A string containing the path to a video file.
<code>outfile</code>	A string containing the path of the video file to write.
<code>hardware</code>	The encoder backend. "none" (default) uses software libx264. "nvenc" uses NVIDIA GPU H.264 encoding ("h264_nvenc"), and "videotoolbox" uses Apple GPU H.264 encoding ("h264_videotoolbox"). The backend you name is the one used. An unavailable one aborts unless <code>fallback = TRUE</code> . See has_hardware_encoder . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it.
<code>fallback</code>	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE re-encodes with software libx264 and a message. FALSE (default) aborts instead. A video_codec in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.

quality	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264 reads it as -crf (0 to 51). h264_nvenc reads it as -cq (0 to 51), and h264_videotoolbox as -q:v (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. The encoder is libx264, h264_nvenc or h264_videotoolbox, as hardware chooses. When <code>fallback = TRUE</code> falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale.
audio_stream	The audio track to carry into the output, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. The every-track family reads NULL this way: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

See Also

[ffmpeg_codec\(\)](#) and [ffmpeg_pixel_format\(\)](#), among the pipeline functions it wraps; [has_hardware_encoder\(\)](#) for the hardware toggle; [standardize_video\(\)](#) for a configurable re-encode; [format_for_web_batch\(\)](#) for the many-file form.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
```

```
format_for_web(video, "web.mp4", run = FALSE)
```

format_for_web_batch *Re-encode Many Videos for the Web From a Jobs Table*

Description

Re-encode many videos into a widely compatible, web-friendly form, using one jobs table. This is the **batch** form of `format_for_web()`, for when you have more than one file. Each row is one input. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each input. Each command uses the same fixed H.264, AAC and +faststart steps as `format_for_web()`, with no per-row settings. The glossary in vignette("tidymedia") explains media terms such as codec and re-encode.

Usage

```
format_for_web_batch(
  jobs,
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per input. It needs at least an input column, the source path. An optional output column names the destination. Without it, each row's output name adds <code>_web</code> to the input's base name, with an <code>.mp4</code> extension. The web re-encode always writes H.264 in mp4. For example, <code>clip.mkv</code> becomes <code>clip_web.mp4</code> . Two rows with the same destination path are refused before any row runs. That happens with a repeated output, or with two derived names that match. For example, <code>clip.mov</code> and <code>clip.mkv</code> both give <code>clip_web.mp4</code> . An optional numeric <code>audio_stream</code> column overrides the <code>audio_stream</code> argument for each row. NA keeps every audio track in that row. A numeric <code>quality</code> column overrides the <code>quality</code> argument per row (see <code>quality</code>). Any other columns are ignored, <code>video_codec</code> and <code>audio_codec</code> included. The sibling batch functions read those two columns as per-row overrides, but this one does not. The web recipe fixes which codecs the output uses: H.264 video and AAC audio. For per-row codecs, use a function that has them, such as standardize_video_batch or crop_video_batch .
hardware	The encoder backend for every row. "none" (default) uses software libx264. "nvenc" uses NVIDIA GPU H.264 encoding, and "videotoolbox" uses Apple GPU H.264 encoding. It applies to the whole batch and is not read as a column.

	See has_hardware_encoder . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows.
fallback	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE re-encodes with software libx264 and a message. FALSE (default) aborts instead. A video_codec in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
quality	A number, or NULL (default), applied to each row unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See format_for_web() for the encoders, their flags and ranges, and the values it refuses.
audio_stream	The audio track to carry into each output, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. Without an audio_stream column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The every-track family reads NULL as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. <code>verify</code> , <code>manifest</code> , <code>progress</code>).

Value

The jobs tibble with an added command column. When `run = TRUE`, it also has a success column, plus verified or a provenance manifest, each when requested through ... See [ffmpeg_batch](#).

See Also

[format_for_web\(\)](#), the single-input form it wraps; [ffmpeg_batch\(\)](#), the batch runner; [standardize_video_batch\(\)](#) for a configurable re-encode.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(input = c(video, video), output = c("a.mp4", "b.mp4"))
format_for_web_batch(jobs, run = FALSE)
```

get_duration

Get the duration of a media file

Description

`get_duration()` uses the MediaInfo program to look up the duration of a media file. You choose the section of the file and the unit.

Usage

```
get_duration(
  file,
  section = c("General", "Video", "Audio"),
  unit = c("ms", "sec", "min", "hour")
)
```

Arguments

file	A character vector of one or more media file paths.
section	A string indicating the MediaInfo section from which to query the duration value. Can be either "General", "Video", or "Audio" (default = "General").
unit	A string indicating whether the duration should be returned in milliseconds ("ms"), seconds ("sec"), minutes ("min"), or hours ("hour") (default = "ms").

Details

The function returns one number for each file. The `probe_*`() functions, `mediainfo_query()` and `mediainfo_template()` return tibbles instead.

Value

A double vector (one per file) giving the duration of the specified section in the specified units.

See Also

[mediainfo_parameter\(\)](#) for arbitrary MediaInfo fields, and [probe_all\(\)](#) to read information with FFprobe.

Other metadata functions: [get_frame_rate\(\)](#), [get_height\(\)](#), [get_sample_rate\(\)](#), [get_width\(\)](#), [mediainfo_parameter\(\)](#), [mediainfo_query\(\)](#), [mediainfo_template\(\)](#), [probe_all\(\)](#), [probe_container\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
get_duration(video, unit = "sec")
```

get_frame_rate	<i>Get the video frame rate of a media file</i>
----------------	---

Description

`get_frame_rate()` uses the MediaInfo program to look up the video frame rate of a media file, in frames per second (fps). The glossary in `vignette("tidymedia")` explains media terms such as frame rate.

Usage

```
get_frame_rate(file)
```

Arguments

`file` A character vector of one or more media file paths.

Details

The function returns one number for each file. The `probe_*`() functions, [mediainfo_query\(\)](#) and [mediainfo_template\(\)](#) return tibbles instead.

Value

A double vector (one per file) giving the video frame rate in fps.

See Also

[mediainfo_parameter\(\)](#) for arbitrary MediaInfo fields, and [probe_all\(\)](#) to read information with FFprobe.

Other metadata functions: [get_duration\(\)](#), [get_height\(\)](#), [get_sample_rate\(\)](#), [get_width\(\)](#), [mediainfo_parameter\(\)](#), [mediainfo_query\(\)](#), [mediainfo_template\(\)](#), [probe_all\(\)](#), [probe_container\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
get_frame_rate(video)
```

get_height*Get the video height of a media file*

Description

get_height() uses the MediaInfo program to look up the video height of a media file, in pixels (px).

Usage

```
get_height(file)
```

Arguments

file A character vector of one or more media file paths.

Details

The function returns one number for each file. The probe_*() functions, [mediainfo_query\(\)](#) and [mediainfo_template\(\)](#) return tibbles instead.

Value

A double vector (one per file) giving the video height in px.

See Also

[mediainfo_parameter\(\)](#) for arbitrary MediaInfo fields, and [probe_all\(\)](#) to read information with FFprobe.

Other metadata functions: [get_duration\(\)](#), [get_frame_rate\(\)](#), [get_sample_rate\(\)](#), [get_width\(\)](#), [mediainfo_parameter\(\)](#), [mediainfo_query\(\)](#), [mediainfo_template\(\)](#), [probe_all\(\)](#), [probe_container\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
get_height(video)
```

get_sample_rate	<i>Get the audio sample rate of a media file</i>
-----------------	--

Description

`get_sample_rate()` uses the MediaInfo program to look up the audio sample rate of a media file, in hertz (Hz). The glossary in `vignette("tidymedia")` explains media terms such as sample rate.

Usage

```
get_sample_rate(file)
```

Arguments

file	A character vector of one or more media file paths.
------	---

Details

The function returns one number for each file. The `probe_*`() functions, `mediainfo_query()` and `mediainfo_template()` return tibbles instead.

Value

A double vector (one per file) giving the audio sample rate in Hz.

See Also

`mediainfo_parameter()` for arbitrary MediaInfo fields, and `probe_all()` to read information with FFprobe.

Other metadata functions: `get_duration()`, `get_frame_rate()`, `get_height()`, `get_width()`, `mediainfo_parameter()`, `mediainfo_query()`, `mediainfo_template()`, `probe_all()`, `probe_container()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
get_sample_rate(video)
```

get_width	<i>Get the video width of a media file</i>
-----------	--

Description

`get_width()` uses the MediaInfo program to look up the video width of a media file, in pixels (px).

Usage

```
get_width(file)
```

Arguments

`file` A character vector of one or more media file paths.

Details

The function returns one number for each file. The `probe_*`() functions, `mediainfo_query()` and `mediainfo_template()` return tibbles instead.

Value

A double vector (one per file) giving the video width in px.

See Also

`mediainfo_parameter()` for arbitrary MediaInfo fields, and `probe_all()` to read information with FFprobe.

Other metadata functions: `get_duration()`, `get_frame_rate()`, `get_height()`, `get_sample_rate()`, `mediainfo_parameter()`, `mediainfo_query()`, `mediainfo_template()`, `probe_all()`, `probe_container()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
get_width(video)
```

hardware_encoder	<i>Hardware video encoders</i>
------------------	--------------------------------

Description

These functions help with optional hardware video encoding. `hardware_encoder()` gives the hardware encoder name for a codec family. `has_hardware_encoder()` reports whether that encoder is available in the local FFmpeg build. The package supports two backends: NVIDIA `nvenc` (H.264, HEVC and AV1) and Apple `videotoolbox` (H.264 and HEVC). So `hardware_encoder("h264", "nvenc")` is `"h264_nvenc"`, and `hardware_encoder("h264", "videotoolbox")` is `"h264_videotoolbox"`. The glossary in `vignette("tidymedia")` explains media terms such as codec, container and hardware encoder.

Usage

```
hardware_encoder(codec = c("h264", "hevc", "av1", "prores"), hardware)
```

```
has_hardware_encoder(codec = c("h264", "hevc", "av1", "prores"), hardware)
```

Arguments

codec	The video codec family: one of <code>"h264"</code> , <code>"hevc"</code> , <code>"av1"</code> , or <code>"prores"</code> . These are the families the package recognizes, not the families a given backend covers. If the chosen hardware backend has no encoder for a family, the function refuses the call. The error names both the backend and the family (e.g. <code>"av1"</code> under <code>"videotoolbox"</code>). Both backends refuse <code>"prores"</code> today.
hardware	The backend: <code>"nvenc"</code> or <code>"videotoolbox"</code> . Required, with no default. This set is narrower than the hardware argument of the task functions. There, <code>"none"</code> means <code>"use no backend"</code> . That has no meaning here, so the function refuses it.

Details

`has_hardware_encoder()` is a *cheap* check. It asks whether FFmpeg lists the encoder (via [ffmpeg_encoders](#)). That list reflects how FFmpeg was built. It does not reflect whether working hardware and a driver are present at run time. An encode can still fail at run time on a machine with no capable GPU. To override detection in a known environment (or in tests), set `options(tidymedia.hardware_encoders = )` to a character vector of encoder names to treat as available.

The hardware argument of the task functions uses the same encoder names and the same check. These task functions have that argument: [standardize_video](#), [format_for_web](#), [anonymize_video](#), [crop_video](#), [segment_video](#), [compare_videos](#), [picture_in_picture](#), and [separate_audio_video](#) (and their `_batch` forms). Some of these functions have a `video_codec` that defaults to `NULL` (no codec named), and they assume the H.264 family. So a container that does not take H.264 (e.g. `.webm`) needs an explicit HEVC- or AV1-family `video_codec`. AV1 works only under `"nvenc"`. Hardware *decoding* (`-hwaccel`) and GPU filter pipelines are out of scope. Use the [ffmpeg](#) direct command for those.

Value

`hardware_encoder()` returns a single encoder-name string (e.g. "h264_nvenc"). `has_hardware_encoder()` returns a length-one logical. Neither returns for a codec that the chosen hardware backend has no encoder for. That pair is a wrong argument, not a machine without something. So both give the error that codec describes above. `has_hardware_encoder()` returns FALSE only for a pair that the chosen backend has an encoder for and this FFmpeg build does not list.

See Also

[ffmpeg_encoders](#) for the full encoder list. These task functions have the hardware argument: [standardize_video](#), [format_for_web](#), [anonymize_video](#), [crop_video](#), [segment_video](#), [compare_videos](#), [picture_in_picture](#), and [separate_audio_video](#).

Other capability functions: [ffmpeg_codecs\(\)](#), [ffmpeg_encoders\(\)](#), [has_encoder\(\)](#), [refresh_ffmpeg_capabilities\(\)](#)

Examples

```
hardware_encoder("h264", "nvenc")
has_hardware_encoder("h264", "nvenc")
```

has_encoder

Ask whether your FFmpeg build has an encoder

Description

FFmpeg builds differ in the encoders they were compiled with. A build without libx264 refuses every call that names it, and some Linux distributions ship exactly that build, because x264 is not free software. This function answers whether one encoder is in the build this package resolves. You can then name another encoder, or skip the work, before you run anything.

Usage

```
has_encoder(name)
```

Arguments

name	A string naming one encoder, spelled as ffmpeg_encoders spells it, such as "libx264", "aac" or "libmp3lame".
------	--

Details

The answer comes from the list [ffmpeg_encoders](#) returns. The list is asked of FFmpeg once per session and then remembered, so a repeated call does not start FFmpeg again. [refresh_ffmpeg_capabilities](#) makes the next call ask again.

The `tidymedia.hardware_encoders` option does not change this answer. That option names the hardware encoders of the machine, and this question is about the whole build. Use [has_hardware_encoder](#) for the GPU backends.

Where the package finds no FFmpeg at all, the answer is FALSE: nothing can be encoded, so no encoder is available. A location you saved with [set_ffmpeg](#) that no longer works still gives its warning beside the FALSE. So a machine you can repair is never reported silently as a build without the encoder.

Value

A single TRUE or FALSE.

See Also

[ffmpeg_encoders](#) for the whole list, [has_hardware_encoder](#) for the GPU backends, and [refresh_ffmpeg_capabilities](#) to make the next call ask FFmpeg again.

Other capability functions: [ffmpeg_codecs\(\)](#), [ffmpeg_encoders\(\)](#), [hardware_encoder\(\)](#), [refresh_ffmpeg_capabilities\(\)](#)

Examples

```
has_encoder("libx264")
has_encoder("no_such_encoder")
```

install_on_win	<i>Install FFmpeg on Windows</i>
----------------	----------------------------------

Description

`install_on_win()` downloads a Windows build of FFmpeg and unpacks it. Then it saves the locations of `ffmpeg`, `ffprobe` and `ffplay`, as [set_program\(\)](#) does. After that, the package can find these programs.

By default, the call downloads the latest "essentials" build from [gyan.dev](#). It unpacks the build into the `ffmpeg` folder under `tools::R_user_dir("tidymedia", "data")`.

By default, the call asks you to confirm before it does anything. The question names each file it will download and the folder it will unpack into. It also names the saved program locations that the install can replace. If you say no, the call returns FALSE and changes nothing.

This function works on Windows only. On any other system, it gives an error before it asks, writes or downloads anything. The error names the system it found. On macOS, you can install FFmpeg with `brew install ffmpeg`. On Linux, you can use `sudo apt-get install ffmpeg`. On any system, [set_program\(\)](#) tells the package where an installed FFmpeg is.

Usage

```
install_on_win(
  download_url = NULL,
  install_dir = NULL,
  confirm = TRUE,
  archive_checksum = NULL
)
```

Arguments

download_url	A string with the address of the FFmpeg archive. If NULL, the call uses the latest "essentials" build from gyan.dev, a .7z archive.
install_dir	A string with the folder to install FFmpeg into. If NULL, the call uses the ffmpeg folder under tools::R_user_dir("tidymedia", "data"). CRAN allows packages to keep user data in that place.
confirm	TRUE or FALSE. Whether to ask before the call downloads or installs anything. Defaults to TRUE. In a session where no one can answer, TRUE gives an error that names the same items as the question. Pass confirm = FALSE to install without the question.
archive_checksum	A string with the expected SHA-256 checksum of the archive, as 64 hexadecimal characters in upper or lower case. Defaults to NULL. If you give a checksum, the call uses it for any source and downloads no checksum. If it is NULL and download_url is not the default source, the call checks nothing and says so.

Details

Before the call unpacks the archive, it checks the archive against a SHA-256 checksum. A checksum is a fingerprint of the file's contents. For the default source, the call downloads the checksum that gyan.dev publishes next to each build. That file has the archive's address with .sha256 added. For any other source, give the checksum in archive_checksum.

The published checksum comes from the same site as the archive, over the same connection. So the check finds a damaged or incomplete download. It does not find a source that someone has tampered with.

After the unpack, the call checks each program before it saves any location. The path must be one that R finds as a program. It must be a file, not a folder, and the file must not be empty. The call does not run the program. So a build for the wrong type of processor can pass this check.

The package needs ffmpeg and ffprobe. If either one fails the check, the call saves no location and gives an error. The error names each failed program and its full path. If ffplay is missing or fails the check, the install finishes. A message says that the call did not save ffplay.

Value

TRUE when the install finished. FALSE when you said no to the question, or when the call could not create the install folder. Other failures give an error. The section "Errors" lists the error classes the call gives. A wrong argument gives an error before any of these.

What a failed install leaves behind

When the call gives an error, it tries to leave the install folder as it found it. It removes the files that a failed unpack wrote. It removes a folder that the call created. It does not touch the files that were already in the folder, with one exception.

The exception is a file of yours that the failed unpack wrote over. The call removes that file too, because the file no longer holds what you put there.

On Windows, the removal can fail. After a failed unpack, the unpack library can still hold a file open. Windows does not delete a file that is open.

The error names by full path the entries of the first case below that applies:

- each unpacked file that the call could not remove
- each folder that the call created and could not remove
- each file of yours that the call removed

Two errors come after a successful unpack: `tidymedia_program_not_extracted` and `tidymedia_program_unusable`. These errors leave the unpacked files in the folder, and they say so.

The call learns which files the unpack made from the archive's own list and from the folder. A program that the list names but that is not in the folder counts as not unpacked. For example, antivirus software can remove a program right after the unpack. The error then says that the unpack reported writing that file.

If none of the unpacked files are in the folder, `tidymedia_program_not_extracted` follows the usual rule. The call removes a folder that it created, and the error says so.

Errors

The call gives an error of its own class in these cases:

- `tidymedia_wrong_platform`: the session is not running on Windows.
- `tidymedia_confirmation_unavailable`: the call must ask you to confirm, but no one can answer in this session.
- `tidymedia_download_unavailable`: the archive did not download, or nothing readable arrived.
- `tidymedia_checksum_unavailable`: the call could not download or read the published checksum.
- `tidymedia_checksum_mismatch`: the downloaded archive does not match its checksum.
- `tidymedia_archive_unreadable`: the call could not unpack the archive.
- `tidymedia_program_not_extracted`: `ffmpeg` or `ffprobe` is not at the path where the install would put it.
- `tidymedia_program_unusable`: the archive made `ffmpeg` or `ffprobe`, but the file cannot be used.

See Also

[`set\_program\(\)`](#) to save the location of a program you already have, and [`find\_ffmpeg\(\)`](#) to check where the package finds a program.

Other program management functions: [`find\_ffmpeg\(\)`](#), [`program\_status\(\)`](#), [`set\_program\(\)`](#), [`unset\_program\(\)`](#)

Examples

```
## Not run:
# Download and install a static FFmpeg build (Windows)
install_on_win()

## End(Not run)
```

local_timeout	<i>Set a time limit for the rest of a function</i>
---------------	--

Description

local_timeout() sets a time limit for the rest of the function that calls it. The limit applies to each FFmpeg, FFprobe or MediaInfo program that the function starts after this call. When the function returns, or stops with an error, the caller's own limit is back, except in the cases in Details.

Use with_timeout() to set a limit on one expression. Use local_timeout() to set a limit on the rest of a function, or on several calls that are hard to wrap in one expression.

Usage

```
local_timeout(seconds, .local_envir = parent.frame())
```

Arguments

seconds	A whole number of seconds. 0 means no limit, so local_timeout(0) removes a session limit for the rest of the function. A fraction, a negative number, a string or NULL gives an error, and the limit does not change.
.local_envir	The environment that holds the limit. The default is the function that calls local_timeout(). Change it only when you write a helper that sets a limit for its own caller. Inside a function, the environment must belong to a function that is still running. Suppose it belongs to a function that has returned, or it is an environment such as new.env(). Then the limit stays set with no error. withr::local_options() works the same way. At the top level of a script or the console, withr::defer() decides when the limit is undone.

Details

The limit applies to each program, not to the whole function. In a 100-row batch after local_timeout(600), each program that a row starts gets 600 seconds, plus the delay that with_timeout() describes. The workers of a parallel = TRUE run use the same limit.

R can wait up to 40 seconds past the limit, and with_timeout() explains why. It also explains what happens when a limit is reached.

Two calls in one function work like any two local_*() calls. The second limit applies until the function ends. Then both are undone, and the caller's limit is back.

In three cases, the caller's limit is not back when the function ends, and there is no error.

- The function calls on.exit() without add = TRUE. That call removes the undo step. Write on.exit(..., add = TRUE) instead.
- The .local_envir belongs to a function that has returned, or it is an environment such as new.env().
- local_timeout() is called directly inside the expression of with_timeout(). There, local_timeout() belongs to the function around it. So when that function ends, the limit that with_timeout() set is in force. Put the inner limit in a function of its own, or use only one of the two.

The first two cases also apply to `withr::local_options()`, because R's exit handlers work this way. They do not apply to `withr::with_options()`, which puts the option back itself.

Value

The caller's earlier setting, invisibly. It is a list with one element, the same form that `withr::local_options()` returns.

See Also

`with_timeout()` to set a limit for one expression. [tidymedia-package](#) describes the session options.

Examples

```
bounded <- function() {
  local_timeout(30)
  getOption("tidymedia.timeout")
}

# In force for the rest of that function...
bounded()

# ...and gone once it has returned.
getOption("tidymedia.timeout", default = "unset")

# Bound every program a whole function starts, at five minutes. Defining the
# function starts nothing; the limit applies when you call it.
convert_all <- function(files) {
  local_timeout(300)
  for (f in files) extract_audio(f, sub("[.][^.]*$", ".wav", f))
}
```

mediainfo

Run a MediaInfo command

Description

`mediainfo()` runs the MediaInfo program with the arguments in `command` and returns its output. MediaInfo reads information about media files.

Usage

```
mediainfo(command)
```

Arguments

`command` A string with the arguments to give MediaInfo.

Details

`mediainfo()` is a direct command. The package passes command to MediaInfo exactly as you wrote it, so you must add any quotes that it needs. To get a tibble or a value instead, use `mediainfo_template()`, `mediainfo_query()` or `mediainfo_parameter()`. These functions quote their arguments for you.

Value

A character vector with the text that MediaInfo writes to standard output, one element for each line. Messages on standard error are not returned. On macOS and Linux, a shell redirect such as `2>&1` in command returns them too.

See Also

`mediainfo_template()`, `mediainfo_query()` and `mediainfo_parameter()` for a tibble or a value. `get_duration()` and the other `get_*`() functions for common single values.

Other direct command functions: `ffmpeg()`, `ffprobe()`

Examples

```
mediainfo("--Version")
```

<code>mediainfo_parameter</code>	<i>Query a single parameter from a single MediaInfo section</i>
----------------------------------	---

Description

`mediainfo_parameter()` uses the MediaInfo program to read one value, such as the video width, from media files. MediaInfo groups its values in sections, such as "General", "Video" and "Audio". You name the section and the parameter to read.

Usage

```
mediainfo_parameter(file, section, parameter, typed = TRUE)
```

Arguments

<code>file</code>	A character vector of one or more media file paths.
<code>section</code>	A string. The name of the MediaInfo section to read parameter from.
<code>parameter</code>	A string. The name of the MediaInfo parameter to read from section.
<code>typed</code>	A logical. If TRUE (the default), the function converts the values to their natural type, for example to numbers. If FALSE, it returns strings.

Details

Give several files in `file` to get one value for each file. The function returns a vector, not a tibble. The `probe_*`() functions read similar information with FFprobe and return tibbles.

Value

A vector with one value for each element of file. A value is NA when MediaInfo prints more than one line, for example for a section it does not know. A parameter that section does not have gives an empty value. That value is NA when typed = TRUE and "" when typed = FALSE. A value is also NA for a file that does not exist or that reaches the time limit.

The function does not stop at those files. It reads the other files, and then gives one warning that names the files that do not exist or reached the limit. See `with_timeout()` for the time limit.

See Also

`mediainfo_query()` to read several parameters at once. `mediainfo_template()` to apply a whole template. `probe_all()` to read information with FFprobe. `get_duration()` and the other `get_*`() functions for common single values.

Other metadata functions: `get_duration()`, `get_frame_rate()`, `get_height()`, `get_sample_rate()`, `get_width()`, `mediainfo_query()`, `mediainfo_template()`, `probe_all()`, `probe_container()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
mediainfo_parameter(video, section = "Video", parameter = "Width")
```

mediainfo_query	<i>Query multiple parameters from a single MediaInfo section</i>
-----------------	--

Description

`mediainfo_query()` uses the MediaInfo program to read several parameters from one section, and returns a tibble. To read parameters from more than one section in one call, use `mediainfo_template()`.

Usage

```
mediainfo_query(file, section, parameters, names = parameters, typed = TRUE)
```

Arguments

file	A character vector of one or more media file paths.
section	A string. The name of the MediaInfo section to read parameters from.
parameters	A character vector of one or more MediaInfo parameters to read from section.
names	A character vector of column names, one for each element of parameters. The default is parameters. The function keeps the names as you give them, but removes spaces at their start and end.
typed	A logical. If TRUE (the default), numeric columns become numbers and empty values become NA. If FALSE, all columns stay strings.

Details

Give several files in `file` to get one row for each file. The first column, `file`, names the input file. The `probe_*`() functions read similar information with `FFprobe`.

Value

A tibble with one row for each input file. The first column is `file`, and then there is one column for each parameter.

A file that the function could not read gets a row of NA values. The function reads the other files, and then gives one warning that names the files it could not read. A file that reaches the time limit counts as not read; see `with_timeout()`.

See Also

`mediainfo_parameter()` to read a single value. `mediainfo_template()` to apply a whole template. `probe_all()` to read information with `FFprobe`. `get_duration()` and the other `get_*`() functions for common single values.

Other metadata functions: `get_duration()`, `get_frame_rate()`, `get_height()`, `get_sample_rate()`, `get_width()`, `mediainfo_parameter()`, `mediainfo_template()`, `probe_all()`, `probe_container()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
mediainfo_query(video, section = "Video", parameters = c("Width", "Height"))
```

mediainfo_template	<i>Describe media files by applying a MediaInfo template</i>
--------------------	--

Description

`mediainfo_template()` uses the `MediaInfo` program to describe media files, and returns a tibble. It applies a `MediaInfo` template, which can read many parameters from many sections.

Usage

```
mediainfo_template(
  file,
  template = c("brief", "extended", "custom"),
  templatefile = NULL,
  typed = TRUE
)
```

Arguments

file	A character vector of one or more media file paths.
template	A string. Use "brief" or "extended" for a template that comes with the package. Use "custom" to apply the file in templatefile.
templatefile	The path to your own MediaInfo template, a .txt file that makes MediaInfo print comma-separated values. Give it when template is "custom", and only then. The default is NULL.
typed	A logical. If TRUE (the default), numeric columns become numbers and empty values become NA. If FALSE, all columns stay strings.

Details

The package comes with two templates, "brief" and "extended". You can also give your own template file. Give several files in `file` to get one row for each file. The first column, `file`, names the input file. The `probe_*`() functions read similar information with FFprobe.

Value

A tibble with one row for each input file. The template sets the columns, their names and their order. The function keeps the column names of a custom template, but removes spaces at their start and end.

A file that the function could not read gets a row of NA values. The function reads the other files, and then gives one warning that names the files it could not read. A file that reaches the time limit counts as not read; see `with_timeout()`.

See Also

`mediainfo_query()` to read one section. `mediainfo_parameter()` to read a single value. `probe_all()` to read information with FFprobe. `get_duration()` and the other `get_*`() functions for common single values.

Other metadata functions: `get_duration()`, `get_frame_rate()`, `get_height()`, `get_sample_rate()`, `get_width()`, `mediainfo_parameter()`, `mediainfo_query()`, `probe_all()`, `probe_container()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
mediainfo_template(video, template = "brief")
```

normalize_audio	<i>Normalize a file's audio loudness (EBU R128)</i>
-----------------	---

Description

Normalize the perceived loudness of a file's audio toward an EBU R128 target. The function uses FFmpeg's single-pass loudnorm filter. It can also downmix the channel count and resample. The output holds **one audio stream and no video**, whatever the input and whatever container outfile names. So the output of this function is audio, as with [extract_audio](#) and [convert_audio](#). It does not carry the other streams through. To normalize a recording's soundtrack *and* keep its picture, first normalize to an audio file. Then put the audio back with the picture using the [ffmpeg](#) direct command. The glossary in [vignette\("tidymedia"\)](#) explains media terms such as LUFS, true peak and sample rate.

Usage

```
normalize_audio(
  infile,
  outfile,
  target_loudness = -23,
  true_peak = -1,
  loudness_range = 7,
  channels = NULL,
  sample_rate = NULL,
  audio_codec = NULL,
  two_pass = FALSE,
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

infile	A string containing the path to a media file (with audio). An input with no audio stream is an FFmpeg error, not a silent copy of the video.
outfile	A string containing the path of the audio file to write. The function accepts any container that FFmpeg can write. The compiled command does not depend on which container it is. An audio container (.wav, .flac) holds the result exactly as a video container (.mkv) does. The video container carries one audio stream and nothing else.
target_loudness	The target integrated loudness, in LUFS (a number in -70..-5; default -23, the EBU R128 target).
true_peak	The maximum true peak, in dBTP (a number in -9..0; default -1, the EBU R128 ceiling).
loudness_range	The target loudness range, in LU (a number in 1..50; default 7).

channels	The output channel count, e.g. 1 to downmix to mono (a positive whole number), or NULL (default) to keep the source layout.
sample_rate	The output sample rate in Hz, e.g. 48000 (a positive whole number). NULL (default) lets loudnorm choose. It resamples, up to 192 kHz and capped by the encoder, and does not keep the source rate. Set this argument to fix the output rate.
audio_codec	An optional string naming the output audio encoder (e.g. "aac", "libmp3lame", "flac"), passed to FFmpeg's <code>-codec:a</code> . NULL (default) sets no <code>-codec:a</code> , which leaves the default encoder of the output container in place. "copy" is an error. Loudness normalization filters the audio, so the stream must be re-encoded and cannot be copied.
two_pass	A logical. When TRUE, the function uses accurate two-pass (measured/linear) normalization. The default (FALSE) is single-pass. A first <i>analysis pass</i> measures the loudness of the input. A second <i>correction pass</i> feeds those measurements back with <code>linear=true</code> , so the output hits the EBU R128 target precisely. So two-pass always runs the analysis pass through FFmpeg , even when <code>run = FALSE</code> . It needs the binary and a readable input. Under <code>run = FALSE</code> , the analysis still runs. The returned value is the exact correction command, which is not run. The single-pass default touches no binary under <code>run = FALSE</code> . If the input is silent , the analysis pass measures its loudness as <code>-inf</code> . Normalizing silence to a target is undefined, so two-pass aborts with a clear error. The single-pass default leaves silence untouched. The batch form differs here. normalize_audio_batch does not abort on a silent row. It sets that row aside, marks it in a silent column, and normalizes the rest. When the analysis pass gives no usable measurement at all, the abort has class <code>tidymedia_loudnorm_no_measurement</code> . The batch form raises the same class, so one handler covers both. Where FFmpeg exited non-zero, the abort also carries <code>tidymedia_ffmpeg_exit</code> , and the exit number on <code>tm_status</code> . Where FFmpeg exited zero but printed no measurement block that can be parsed, the abort carries the shared class alone. The silence abort above is neither: a silent input <i>was</i> measured.
audio_stream	The audio track to normalize, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) normalizes the first audio track. The first-track family reads NULL this way: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The every-track family reads it as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. This function reads NULL as the first track only. The two-pass analysis measures each audio track, but the correction uses one set of measurements. Normalizing several tracks at once would apply one track's measurements to all of them. Under <code>two_pass = TRUE</code> , the analysis pass measures this same track. Only the named track reaches the output, and no video does, whatever the container. So an output name with a video extension gives a video file that holds only audio. An input with no audio at all is an FFmpeg error. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)

run A logical: run the (correction) command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE). Under `two_pass = TRUE` this gates only the correction pass; the analysis pass runs regardless (see `two_pass`).

Details

The default targets follow EBU Recommendation R 128 (2014). They are `target_loudness = -23 LUFS` and `true_peak = -1 dBTP`. Loudness is measured per ITU-R BS.1770-4. The default `loudness_range` is 7. This is single-pass (dynamic) loudnorm. The same input and arguments always compile to one reproducible command, with no separate measurement pass. The filter changes the audio, so FFmpeg re-encodes it. Set `audio_codec` to name the output encoder, or leave it NULL to use the default of the output container. Leaving `channels` at NULL keeps the source channel layout. FFmpeg's loudnorm filter resamples its output, up to 192 kHz, capped by the encoder. So the output sample rate is *not* the source rate unless you set it. Set `sample_rate` to control the output rate.

The function warns when no `audio_stream` is named and `infile` carries tracks that the output will not. `extract_audio` and `convert_audio` emit the same warning. Naming a track with `audio_stream` silences it, as does `suppressWarnings(classes = "tidymedia_dropped_audio")`. The check costs **one FFprobe call per distinct input**, which is one call here, because this function takes a single `infile`. The warning is given when FFprobe is available and the input can be probed. Otherwise the check is skipped silently. It never runs under `run = FALSE`, and never changes the compiled command. Under `two_pass = TRUE`, the warning comes *before* the analysis pass. So it arrives while adding `audio_stream` can still save that pass.

To switch the check off and skip its FFprobe call, use `options(tidymedia.check_tracks = FALSE)` for the session. Use `withr::local_options(tidymedia.check_tracks = FALSE)` for the rest of one function.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`). Under `two_pass = TRUE` this is the correction command built from the measured values.

References

EBU Recommendation R 128 (2014), *Loudness normalisation and permitted maximum level of audio signals*; ITU-R BS.1770-4.

See Also

`ffmpeg_loudnorm()`, the pipeline function it wraps. `normalize_audio_batch()` for the many-file form. `extract_audio()` and `convert_audio()`, the other task functions whose output is one audio stream.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`,

`segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# The output holds audio only, so name an audio file for it
normalize_audio(video, "normalized.wav", run = FALSE)
# Normalize to a streaming target and downmix to mono
normalize_audio(video, "mono.wav", target_loudness = -16, channels = 1,
  run = FALSE)
# Name the output audio encoder instead of taking the container's default
normalize_audio(video, "normalized.m4a", audio_codec = "aac", run = FALSE)
```

`normalize_audio_batch` *Normalize Many Files' Audio Loudness From a Jobs Table*

Description

Normalize the audio loudness of many input files (EBU R128) from a single jobs tibble. This is the **batch** (table-driven) form of `normalize_audio()`, for when you have more than one file to normalize. Each row is one input, and the only required column names its source. The function is a thin wrapper over `ffmpeg_batch`. It gives one reproducible compiled command per input. Each row uses the same loudnorm pipeline, and the same check of each value, as `normalize_audio()`. Set `two_pass = TRUE` for accurate measured/linear normalization across the whole table (see `two_pass`). The glossary in `vignette("tidymedia")` explains media terms such as LUFS, encoder and sample rate.

Usage

```
normalize_audio_batch(
  jobs,
  target_loudness = -23,
  true_peak = -1,
  loudness_range = 7,
  channels = NULL,
  sample_rate = NULL,
  audio_codec = NULL,
  two_pass = FALSE,
  audio_stream = NULL,
  run = TRUE,
```

```
parallel = FALSE,
...
)
```

Arguments

jobs	A data frame with one row per input and (at least) an input column (source path). An optional output column names the destination. Without it, the function derives one per row. It appends <code>_normalized</code> to the basename of each input and keeps the extension of the input (e.g. <code>clip.mkv</code> becomes <code>clip_normalized.mkv</code>). The derived name keeps a <i>video</i> extension while the file itself holds audio only. So name an output column yourself when that matters. The function refuses two rows that name the same output path before any row runs. With <code>two_pass = TRUE</code> , that is before the analysis pass. The refusal covers a path repeated in the output column, or a repeated input when there is no output column. Five loudness arguments can also appear as a column that overrides the argument per row. They are <code>target_loudness</code> , <code>true_peak</code> , <code>loudness_range</code> , <code>channels</code> and <code>sample_rate</code> . Rows that omit the column fall back to the value of the argument. An optional <code>audio_codec</code> column (character) names the output audio encoder of each row. There, NA means "leave the encoder unset". Rows that omit it fall back to the <code>audio_codec</code> argument. An optional numeric <code>audio_stream</code> column likewise overrides the <code>audio_stream</code> argument per row. There, NA normalizes the first audio track of that row. Any other columns are ignored.
target_loudness, true_peak, loudness_range	The EBU R128 loudness targets applied to every row, unless <code>jobs</code> carries a column of the same name (see <code>jobs</code>). Defaults follow EBU Recommendation R 128 (2014): <code>target_loudness = -23 LUFS</code> , <code>true_peak = -1 dBTP</code> , <code>loudness_range = 7 LU</code> .
channels	The output channel count applied to every row, unless <code>jobs</code> carries a <code>channels</code> column, e.g. 1 to downmix to mono. NULL (default) keeps each source's channel layout.
sample_rate	The output sample rate in Hz applied to every row, unless <code>jobs</code> carries a <code>sample_rate</code> column. NULL (default) lets <code>loudnorm</code> choose. It resamples, up to 192 kHz and capped by the encoder, and does not keep the source rate. Set this argument to fix the output rate.
audio_codec	The output audio encoder applied to every row, unless <code>jobs</code> carries an <code>audio_codec</code> column, e.g. "aac". NULL (default) sets no <code>-codec:a</code> , which leaves the default encoder of the output container in place. "copy" is an error. Loudness normalization filters the audio, so it must be re-encoded. See normalize_audio .
two_pass	A logical that selects the normalization mode for <i>every</i> row. It applies to the whole table and is not a per-row column. FALSE (default) keeps the single-pass <code>loudnorm</code> pipeline. TRUE runs accurate two-pass (measured/linear) normalization in two phases. An <i>analysis pass</i> first measures the loudness of every input. It honors <code>parallel</code> and the targets of each row. A <i>correction pass</i> then feeds those measurements back with <code>linear=true</code> , so each output hits its EBU R128 target precisely. This is the table-wide form of <code>two_pass</code> in normalize_audio . The result shows the five measured values as columns <code>measured_I</code> , <code>measured_TP</code> ,

measured_LRA, measured_thresh and offset. Two-pass must measure each input. So it **always runs the analysis pass through FFmpeg**, even when `run = FALSE`. It needs the binary and readable inputs. If the analysis of any row fails or gives no measurement that can be parsed, the call aborts and names those rows. It aborts before it builds any correction command. That abort has class `tidymedia_loudnorm_no_measurement`, the same class that `normalize_audio` raises for this event. It carries the same row numbers on `tm_rows`, alongside `tm_row_status`. That field has the FFmpeg exit status of each row, or NA where the row exited zero but printed nothing that can be parsed. It carries no single exit status on `tm_status`, and it does not have class `tidymedia_ffmpeg_exit`. The reason is that it also fires for rows that exited zero. A batch can mix causes, so there is no one number to report. The one-file form carries both only where FFmpeg exited non-zero. Where FFmpeg exited zero and printed nothing that can be parsed, the one-file abort carries the shared class alone, with no `tm_status` either. **Silent** rows are the exception. A silent input (analysis loudness `-inf`) cannot be normalized to a target, but one silent row does not abort the batch. The function normalizes the rows that are not silent. It marks the silent rows in a logical silent column, with `success = FALSE` and no output written, and a warning names them. This is where the batch form and the one-file form differ. `normalize_audio` aborts on a silent input, because one silent input is the whole call. Here, the other rows still have work to do. The single-pass default touches no binary under `run = FALSE`.

audio_stream	The audio track to normalize, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) normalizes the first audio track. Without an <code>audio_stream</code> column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The first-track family reads NULL as the first audio track only: <code>extract_audio</code> , <code>convert_audio</code> and <code>normalize_audio</code> , and their <code>_batch</code> forms. The every-track family reads it as every audio track: <code>separate_audio_video</code> , <code>standardize_video</code> , <code>anonymize_video</code> , <code>crop_video</code> , <code>segment_video</code> and <code>format_for_web</code> , and their <code>_batch</code> forms. This function reads NULL as the first track only. The two-pass analysis measures each audio track, but the correction uses one set of measurements. Normalizing several tracks at once would apply one track's measurements to all of them. Under <code>two_pass = TRUE</code> , the analysis pass measures this same track. Only the named track reaches the output, and no video does, whatever the container. So an output name with a video extension gives a video file that holds only audio. An input with no audio at all is an FFmpeg error. A track the input does not have gives an FFmpeg error, not an R one. See <code>audio_stream</code> for how this differs from <code>audio_input</code> , the input index on <code>compare_videos</code> and <code>picture_in_picture</code> . (default = NULL)
run	A logical: run each input's command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE). Under <code>two_pass = TRUE</code> this gates only the correction pass; the analysis pass runs regardless (see <code>two_pass</code>).
parallel	A logical passed to <code>ffm_batch</code> : normalize in parallel with furrr (TRUE) or sequentially (FALSE, default). Parallelism follows the active <code>future</code> plan. TRUE under the default sequential plan runs one input at a time and warns. Set a plan

```

first, e.g. future::plan(future::multisession).
... Additional arguments forwarded to ffmpeg\_batch, such as verify, manifest,
checksums, and progress.

```

Details

The function warns **once** for the whole batch when a row names no `audio_stream` and its input carries tracks that the output will not. The warning names every affected row. Naming a track silences it. Use the `audio_stream` argument, or an `audio_stream` cell on every row. `suppressWarnings(classes = "tidymedia_dropped_audio")` silences it too. The check costs **one FFprobe call per distinct input** it has to probe. A repeated input is probed once, and a row that names a track is not probed at all. The warning is given when FFprobe is available and the input can be probed. Otherwise the check is skipped silently. Those probes run **one at a time, before any row starts**, so parallel does not reach them. A sweep long enough to look like a hang reports its progress. The check never runs under `run = FALSE` and never changes any compiled command. It is skipped entirely when every row names a track. Under `two_pass = TRUE`, the warning comes *before* the analysis pass. So it arrives while adding `audio_stream` can still save that pass.

To switch the check off and skip the whole sweep, use `options(tidymedia.check_tracks = FALSE)` for the session. Use `withr::local_options(tidymedia.check_tracks = FALSE)` for the rest of one function.

Value

The [tibble](#) returned by [ffmpeg_batch](#): jobs with an added `command` column. When output was derived, it also has the resolved output column. When `run = TRUE`, it has a `success` column, plus any columns the forwarded arguments add, such as `verified`. Under `two_pass = TRUE`, the result also carries the five measured columns (`measured_I` etc.) and a logical `silent` column. The `command` column then holds the linear correction commands. It is NA for silent rows, which carry NA measurements and are not normalized. The columns of the two-pass result do not depend on how many rows are silent. The `verified` column (under `verify`) and the provenance `manifest` (under `manifest`, read with [ffmpeg_manifest](#)) are present whenever requested. That holds even when *every* row is silent. Silent rows carry NA for those outputs.

References

EBU Recommendation R 128 (2014), *Loudness normalisation and permitted maximum level of audio signals*; ITU-R BS.1770-4.

See Also

[normalize_audio\(\)](#) for the single-input form. [ffmpeg_batch\(\)](#) for the batch runner and the arguments forwarded through [standardize_video_batch\(\)](#) for the table-driven form on the video side.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#),

```
segment_video(), segment_video_batch(), separate_audio_video(), separate_audio_video_batch(),
standardize_video(), standardize_video_batch(), strip_metadata(), strip_metadata_batch()
```

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input      = c(video, video),
  output     = c(tempfile(fileext = ".m4a"), tempfile(fileext = ".m4a")),
  target_loudness = c(-23, -16)
)
# run = FALSE compiles one command per input without calling FFmpeg
normalize_audio_batch(jobs, run = FALSE)

# Accurate two-pass (measured/linear) normalization across the whole table.
# This one runs FFmpeg to measure each input, so it needs the binary, and it
# re-encodes, so it needs an encoder this build has.
if (nzchar(Sys.which("ffmpeg")) && has_encoder("aac")) {
  normalize_audio_batch(jobs, two_pass = TRUE, audio_codec = "aac")
}
```

picture_in_picture	<i>Inset one video over another (picture-in-picture)</i>
--------------------	--

Description

Composite a smaller overlay video onto a main video in one corner (or the center). This is the classic picture-in-picture layout for pairing a speaker with a screen recording, or a stimulus with a webcam. Built on the `ffmpeg_overlay` pipeline function, which resizes the overlay to a fraction of the main video's width and positions it. The glossary in `vignette("tidymedia")` explains media terms such as codec, encoder and stream copy.

Usage

```
picture_in_picture(
  main,
  overlay,
  outfile,
  position = c("topright", "topleft", "bottomright", "bottomleft", "center"),
  scale = 0.25,
  margin = 16,
```

```

    audio_input = NULL,
    video_codec = NULL,
    audio_codec = "copy",
    hardware = c("none", "nvenc", "videotoolbox"),
    fallback = FALSE,
    quality = NULL,
    run = TRUE
  )

```

Arguments

<code>main</code>	A string giving the path to the background (full-size) video.
<code>overlay</code>	A string giving the path to the inset video.
<code>outfile</code>	A string giving the path to write the result to.
<code>position</code>	Where to place the inset: one of "topright" (default), "topleft", "bottomright", "bottomleft", or "center".
<code>scale</code>	The inset's width as a fraction of the main video's width, aspect preserved ($0 < \text{scale} \leq 1$). (default = 0.25)
<code>margin</code>	The gap in pixels between the inset and the video edges (ignored for position = "center"). (default = 16)
<code>audio_input</code>	The input file whose audio to keep, as a number that counts from 0. 0 is the first file you pass and 1 is the second. This counts the function's inputs, not the audio tracks of one input. So it is a different index from <code>audio_stream</code> on the functions that take one input. NULL (default) selects no audio at all, so the output is silent. This differs from <code>audio_stream = NULL</code> , which still selects audio. An input number the call does not have gives an R error, before FFmpeg runs. See audio_stream . (default = NULL)
<code>video_codec</code>	A string naming the output video codec, or NULL (default) to leave it unset. Then the output container's default encoder is used, and the compiled command is the same as one that never named a codec.
<code>audio_codec</code>	A string naming the codec for the carried audio track. "copy" (default) stream-copies it through untouched. Name an encoder, such as "aac", to transcode it. NULL leaves the codec unset, so the output container's default encoder is used. When <code>audio_input</code> is NULL, no audio reaches the output, so nothing is emitted. Naming an encoder in that case is an error.
<code>hardware</code>	The encoder backend. "none" (default) uses the software <code>video_codec</code> . "nvenc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). A backend uses its own encoder for the family of <code>video_codec</code> . For example, "libx264" becomes "h264_nvenc" or "h264_videotoolbox". With the default <code>video_codec = NULL</code> , the H.264 family is assumed. So a non-H.264 container, such as .webm, needs an explicit HEVC- or AV1-family <code>video_codec</code> (AV1 only under "nvenc"). See has_hardware_encoder for availability and its caveats. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it.

fallback	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE encodes in software with a message. FALSE (default) aborts instead. With video_codec = NULL, the fallback leaves the codec unset rather than picking one, so the codec never changes silently. A video_codec in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
quality	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264 and libx265 read it as -crf (0 to 51). The nvenc encoders read it as -cq (0 to 51), and the videotoolbox encoders as -q:v (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as libvpx-vp9, is refused with quality set. So is a video_codec of "copy", or of NULL under hardware = "none". When fallback = TRUE falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale.
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

Audio is dropped unless audio_input names an input to carry (0 = the main video, 1 = the overlay).
A carried track is stream-copied unless audio_codec names an encoder.

Value

The compiled FFmpeg command (invisibly when run = TRUE).

See Also

[ffmpeg_overlay\(\)](#), the pipeline function it wraps; [has_hardware_encoder\(\)](#) for the hardware argument; [compare_videos\(\)](#) for side-by-side stacking.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
picture_in_picture(video, video, "pip.mp4", run = FALSE)
```

picture_in_picture_batch

Inset One Video Over Another For Many Outputs From a Jobs Table

Description

Composite an inset (overlay) video onto a main video for many outputs from a single jobs tibble. This is the **batch** (table-driven) form of [picture_in_picture\(\)](#), for when you have more than one to produce. Its two inputs have distinct roles, so `jobs` carries fixed main and overlay columns (not a list-column) plus an output column. This is a thin wrapper over [ffmpeg_batch](#): one reproducible overlay command per row, sharing the pipeline with [picture_in_picture\(\)](#). The glossary in `vignette("tidymedia")` explains media terms such as codec, encoder and stream copy.

Usage

```
picture_in_picture_batch(
  jobs,
  position = c("topright", "topleft", "bottomright", "bottomleft", "center"),
  scale = 0.25,
  margin = 16,
  audio_input = NULL,
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per output and (at least) main (background path), overlay (inset path), and output (destination path) columns. Optional position, scale, margin, audio_input, video_codec, and audio_codec columns override the like-named arguments per row (a row omitting one falls back to the argument). In an audio_input column, NA means "drop audio", the column's way of writing the scalar's NULL. In a video_codec or audio_codec column, it means "leave the codec unset". A numeric quality column overrides the quality argument per row (see quality). Two rows given the same output path are refused before any row runs; other columns are ignored.
------	--

position, scale, margin	Defaults applied to every row lacking the corresponding column. position is one of "topright" (the default), "topleft", "bottomright", "bottomleft" or "center"; a position column is held to those same five values, per row. See picture_in_picture() for their fuller meaning.
audio_input	The input file whose audio to keep, as a number that counts from 0. 0 is the first file you pass and 1 is the second. This counts the function's inputs, not the audio tracks of one input. So it is a different index from audio_stream on the functions that take one input. NULL (default) selects no audio at all, so the output is silent. This differs from audio_stream = NULL, which still selects audio. An input number the call does not have gives an R error, before FFmpeg runs. Without an audio_input column, the argument applies to every row. An NA cell in that column means NULL for that row, so that output has no audio. See audio_stream . (default = NULL)
video_codec	A string naming the output video codec, applied to every row lacking a video_codec column. NULL (default) leaves it unset, so each output keeps its container's default encoder.
audio_codec	A string naming the codec for the carried audio track, applied to every row lacking an audio_codec column. "copy" (default) stream-copies it. Name an encoder to re-encode it, or NULL to leave the codec unset. A row carrying no audio emits no -codec:a, and naming an encoder on such a row is an error.
hardware, fallback	The encoder backend and its fallback behavior, applied to the whole batch. They are a property of the machine, not of a row, so neither is read as a jobs column. See picture_in_picture() . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under run = FALSE. The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also contradict itself by naming an audio_codec with no audio carried into the output. Such a call is refused for the contradiction first, whether or not this machine has the encoder. A per-row value error likewise reports ahead of the encoder check. Examples are a negative margin, an audio_input index outside the two inputs, and a position outside the five accepted values. A value error and a contradiction resolve the same way whether the value arrived as an argument or in a jobs column. The contradiction reports first.
quality	A number, or NULL (default), applied to each row unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See picture_in_picture() for the encoders, their flags and ranges, and the values it refuses.
run	A logical: run each command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).

parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. verify, manifest, progress).

Value

The jobs tibble with an added command column. When run = TRUE, it also has a success column, plus verified or a provenance manifest, each when requested through ... See [ffmpeg_batch](#).

See Also

[picture_in_picture\(\)](#), the one-output function it wraps; [ffmpeg_batch\(\)](#), the batch runner; [has_hardware_encoder\(\)](#) for the hardware argument. [concatenate_videos_batch\(\)](#) and [compare_videos_batch\(\)](#), the other batch functions that take several inputs per row.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(main = video, overlay = video, output = "pip.mp4")
picture_in_picture_batch(jobs, run = FALSE)
```

```
print.tidymedia_ffm    Print an FFmpeg pipeline
```

Description

Print a tidymedia ffmpeg pipeline by showing the FFmpeg command it currently compiles to (via [ffmpeg_compile](#)).

Usage

```
## S3 method for class 'tidymedia_ffm'
print(x, ...)
```

Arguments

`x` A tidymedia ffm pipeline object created by `ffm_files`.
`...` Ignored.

Value

`x`, invisibly.

See Also

`ffm_compile()`, which produces the printed command.

Other pipeline functions: `ffm_batch()`, `ffm_codec()`, `ffm_compile()`, `ffm_concat()`, `ffm_copy()`, `ffm_crop()`, `ffm_drawbox()`, `ffm_drop()`, `ffm_files()`, `ffm_fps()`, `ffm_hstack()`, `ffm_jobs()`, `ffm_loudnorm()`, `ffm_map()`, `ffm_output_options()`, `ffm_overlay()`, `ffm_pixel_format()`, `ffm_run()`, `ffm_scale()`, `ffm_seek()`, `ffm_trim()`, `ffm_vstack()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
ffm_files(video, "output.mp4") |>
  ffm_trim(start = 1, end = 5)
```

probe_all

Look up information about media files using FFprobe

Description

`probe_all()` uses the FFprobe program to read information about media files. It returns two tibbles. One describes each file as a whole, and one describes each stream in the files.

Usage

```
probe_all(infile, typed = TRUE, parallel = FALSE)
```

Arguments

`infile` A character vector of one or more media files to probe, as file paths or web links.
`typed` A logical. If TRUE (the default), numeric columns become integers or doubles, and FFprobe's "N/A" becomes NA. Fractions, ratios, hex identifiers and text stay as strings. If FALSE, every value stays a string.
`parallel` A logical. If TRUE, the function probes the files in parallel with **furrr**. If FALSE (the default), it probes them one at a time. A parallel run uses the active `future::plan()`. It warns when that plan is sequential, because the run is then no faster. The output is the same either way, with the same rows in the same order. `parallel = TRUE` needs the **furrr** package, and only then does the function check for it.

Details

Give several files in `infile` to read them all in one call. The function stacks the rows, and the first column, `file`, names the input file. So you can join and filter the results for a whole batch with `dplyr`.

The `MediaInfo` functions, `mediainfo_*`(), return tibbles or values. The `get_*`() functions return one value for each file.

The glossary in `vignette("tidymedia")` explains media terms such as container and stream.

Value

A list of two tibbles. `container` has one row for each input file. `streams` has one row for each stream. Both tibbles start with a `file` column that names the input file. A file with no readable streams gets one row in `streams`, with NA in every other column.

The function does not stop at a file that it could not probe. That file gets a row of NA values in both tibbles, and the function gives a warning. A file that reaches the time limit counts as not probed; see `with_timeout()`.

See Also

`mediainfo_template()` and `mediainfo_query()` to read information with `MediaInfo`. `get_duration()` and the other `get_*`() functions for single values.

Other metadata functions: `get_duration()`, `get_frame_rate()`, `get_height()`, `get_sample_rate()`, `get_width()`, `mediainfo_parameter()`, `mediainfo_query()`, `mediainfo_template()`, `probe_container()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
info <- probe_all(video)
info$container
info$streams
```

probe_container

Shortcut functions for probing specific information

Description

These functions return one part of what `probe_all()` returns. `probe_container()` returns the container tibble, and `probe_streams()` returns the streams tibble. `probe_video()` and `probe_audio()` return only the video rows or the audio rows of streams.

Usage

```
probe_container(probe = NULL, infile = NULL, typed = TRUE, parallel = FALSE)
```

```
probe_streams(probe = NULL, infile = NULL, typed = TRUE, parallel = FALSE)
```

```
probe_video(probe = NULL, infile = NULL, typed = TRUE, parallel = FALSE)
```

```
probe_audio(probe = NULL, infile = NULL, typed = TRUE, parallel = FALSE)
```

Arguments

probe	A list made by probe_all() . Must be NULL if you give infile.
infile	A character vector of one or more media files. Must be NULL if you give probe.
typed	A logical that the function passes to probe_all() when you give infile. The default is TRUE. The function ignores it when you give probe.
parallel	A logical that the function passes to probe_all() when you give infile. If TRUE, it probes the files in parallel with furrr . If FALSE (the default), it probes them one at a time. The function ignores it when you give probe, because nothing is left to probe.

Details

Give each function either the output of [probe_all\(\)](#) in probe, or one or more files in infile. Give exactly one of the two, or the function gives an error. With infile, the function probes the files again. For large files, probe once with [probe_all\(\)](#) and reuse the result.

These functions use FFprobe and return tibbles. The MediaInfo functions, [mediainfo_*](#)(), and the [get_*](#)() functions are the other ways to read information. The glossary in [vignette\("tidymedia"\)](#) explains media terms such as container and stream.

Value

A tibble with only the requested information. When you give infile, a file that could not be probed gives a warning, as in [probe_all\(\)](#).

See Also

[probe_all\(\)](#) for the full probe. [mediainfo_query\(\)](#) to read information with MediaInfo. [get_width\(\)](#) and the other [get_*](#)() functions for single values.

Other metadata functions: [get_duration\(\)](#), [get_frame_rate\(\)](#), [get_height\(\)](#), [get_sample_rate\(\)](#), [get_width\(\)](#), [mediainfo_parameter\(\)](#), [mediainfo_query\(\)](#), [mediainfo_template\(\)](#), [probe_all\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Probe directly from a file location ...
probe_container(infile = video)
# ... or reuse a probe object to avoid reprobing large files
info <- probe_all(video)
```

```
probe_video(info)
probe_audio(info)
```

program_status	<i>Report which dependency programs tidymedia can find</i>
----------------	--

Description

`program_status()` looks for the four programs that the package uses: `ffmpeg`, `ffprobe`, `ffplay` and `mediainfo`. It returns a table with one row for each program. The row shows where the program is and which version it reports. The call does not install, write or change anything.

Usage

```
program_status()
```

Details

The call looks in the same places as `find_ffmpeg()`. First it looks on the `PATH`, then at a location saved by `set_program()`. For locations saved by versions before 0.2.0, see the section "Locations saved by earlier versions" in `find_ffmpeg()`.

A program that is not installed and has no saved location gets NA in both columns. This case gives no warning, so four missing programs give one table and not four warnings.

A saved location that cannot be used still gives a warning. Without it, the NA would look like a program you never had. Each warning names the saved location or the file that holds it, so you can fix it. There are three cases:

- `tidymedia_location_gone`: no program is at the saved location now.
- `tidymedia_location_not_executable`: a file is there, and it cannot be run. Give it permission to execute, or save another copy. This case does not arise on Windows, which has no execute bit.
- `tidymedia_location_unreadable`: the file that stores the location does not hold exactly one location.

In all three cases, the row has NA in both columns. `unset_program()` forgets the location, and `set_program()` replaces it.

The version is what the program reports about itself. For `ffmpeg`, `ffprobe` and `ffplay`, it is the FFmpeg build number. For `mediainfo`, it is the MediaInfo library version.

Sometimes the call finds a program but cannot get its version. Then the row has a location and an NA version. This happens when the program call fails. It also happens when the time limit in `options(tidymedia.timeout = )` stops it.

Value

A tibble with one row for each program and three columns:

- `program`, the name of the program.
- `location`, the path to the program, or NA.
- `version`, the version that the program reported, or NA.

See Also

[find_ffmpeg\(\)](#) and the other `find_*`() functions to look up one program. [set_program\(\)](#) to save the location of a program that is not on the PATH. [unset_program\(\)](#) to forget a saved location.

Other program management functions: [find_ffmpeg\(\)](#), [install_on_win\(\)](#), [set_program\(\)](#), [unset_program\(\)](#)

Examples

```
# One row per program; NA where the program was not found
program_status()
```

```
refresh_ffmpeg_capabilities
```

Forget what tidymedia remembers about your FFmpeg build

Description

Discard the package's record of which encoders and codecs your FFmpeg build has. The next query then asks FFmpeg again.

Usage

```
refresh_ffmpeg_capabilities()
```

Details

The first call in an R session to [has_encoder](#), or one that uses `hardware = "nvenc"` or `hardware = "videotoolbox"`, asks FFmpeg which encoders it has. The package remembers that answer for the rest of the session. Later calls reuse it and do not start FFmpeg again each time, so a large batch stays fast.

A run that FFmpeg refuses can also fill the record. After such a run, [ffm_run](#) can ask FFmpeg for its encoders and for the codecs it can encode, to name a missing encoder in the error. The package remembers both lists in the same way. A list that FFmpeg did not give is not remembered. The section "When the build lacks an encoder" on [ffm_run](#) says when the lists are asked for.

So the package does not see a change to your FFmpeg build until you discard the record. Examples of a change are a new FFmpeg install, a new graphics card (GPU) driver, or a different FFmpeg program. There are three ways to discard the record:

- Call `refresh_ffmpeg_capabilities()` yourself, at any time.
- Call `set_program` (or `set_ffmpeg`). It discards the record for you, because the record describes the old program.
- Call `unset_program` and have it remove something. When it forgets a saved location, the package can find a different program. A call that removed nothing keeps the record, because the program in use did not change. A call that removed one saved file and then failed on another discards the record. The file it removed may have named the program that the record came from.

The glossary in `vignette("tidymedia")` explains media terms such as encoder and hardware encoder.

Value

NULL, invisibly. Called for its side effect.

Parallel workers

Each R process keeps its own record, and a worker does not get the record of your session. So in a batch on `W` workers, each worker asks FFmpeg once. Your session can also ask once, before the jobs start. Discarding the record in your session does not reach the workers.

The `tidymedia.hardware_encoders` option works in a different way. The package copies your value into each worker for the duration of the call, and then puts back the worker's own value. So a batch under your setting does not ask FFmpeg for an encoder list at all. Every worker gives the same answer as your session.

Functions that never use the record

`ffmpeg_encoders` and `ffmpeg_codecs` ask FFmpeg on every call. So they always show the build as it is now, whether or not you called this function.

See Also

`has_hardware_encoder` uses the remembered answer. `hardware_encoder` gives the encoder name without asking FFmpeg. `ffmpeg_encoders` always gives a fresh encoder list. `set_program` points the package at a different FFmpeg program.

Other capability functions: `ffmpeg_codecs()`, `ffmpeg_encoders()`, `hardware_encoder()`, `has_encoder()`

Examples

```
# After installing FFmpeg, or a GPU driver or OS update mid-session:
refresh_ffmpeg_capabilities()
```

sample_frames	<i>Sample frames from a video at a fixed rate</i>
---------------	---

Description

Sample a video into a numbered image sequence. Sample at a fixed rate (fps) or at a fixed interval (interval, seconds between frames). This is the first step for per-frame coding and for computer-vision feature pipelines. Provide exactly one of fps or interval.

Usage

```
sample_frames(
    infile,
    outdir,
    fps = NULL,
    interval = NULL,
    format = "png",
    prefix = NULL,
    run = TRUE
)
```

Arguments

infile	A string containing the path to a video file.
outdir	A string naming the directory to write the image sequence to. The function creates it (recursively) if it does not exist.
fps	The sampling rate, in frames per second: either a positive number or an FFmpeg framerate expression string (for example "30000/1001"). Provide exactly one of fps or interval.
interval	The number of seconds between sampled frames (a positive number). The function uses the reciprocal as the frame rate. Provide exactly one of fps or interval.
format	A string giving the output image file extension (one of "png", "jpg", "jpeg", "bmp", "tif", "tiff", "webp"). (default = "png")
prefix	A string used as the basename stem of each image, or NULL to derive it from infile's basename. (default = NULL)
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

[extract_frame](#) saves one frame, and [extract_frame_batch](#) saves a set of frames that you list. This function is different. It builds a *single* FFmpeg command whose output is a printf-style file name pattern. FFmpeg's image2 muxer fills the pattern. FFmpeg decides the frame count when it decodes the video, and you do not list the frames. The function writes frames to outdir as <prefix>_<n>.<format>, where <n> is a zero-padded integer starting at 1. The glossary in [vignette\("tidymedia"\)](#) explains media terms such as frame rate.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

See Also

`ffmpeg_fps()`, the pipeline function it uses to set the sampling rate. `extract_frame()` for a single frame, and `extract_frame_batch()` for a set of frames that you list. `sample_frames_batch()` for the many-file form.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# run = FALSE returns the reproducible command instead of executing it
sample_frames(video, tempdir(), fps = 2, run = FALSE)
```

sample_frames_batch	<i>Sample frames from many videos at a fixed rate from a jobs table</i>
---------------------	---

Description

Sample many videos into numbered image sequences, using one jobs table. This is the **batch** form of `sample_frames()`. Each row is one input video, sampled at a fixed rate into its own image sequence. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each input. The glossary in `vignette("tidymedia")` explains media terms such as frame rate.

Usage

```
sample_frames_batch(
  jobs,
  fps = NULL,
  interval = NULL,
  outdir = NULL,
  format = "png",
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per input. It needs at least an input column (source path). An optional outdir column gives the output directory for that row's sequence. When it is absent, the function derives one as <input-base>_frames beside each input. Optional fps and interval columns override the rate per row. The function ignores any other columns. The function refuses two rows whose image sequences would share a file-name pattern, before any row runs. Two rows share a pattern when they have the same output directory path and the same input file name without its extension. The directory path can come from the column, from the outdir argument, or from the derived name.
fps, interval	The sampling rate applied to every row, as in sample_frames() . A per-row column of the same name overrides it. Supply exactly one of the two (as an argument or a column). (default = NULL)
outdir	An optional single output directory for all rows. An outdir column overrides it. When both are absent, the function derives one directory per input. (default = NULL)
format	A string giving the output image file extension, as in sample_frames() . (default = "png")
run	A logical: run each input's command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical passed to ffmpeg_batch : sample in parallel with furrr (TRUE) or one after another (FALSE, default). Parallel work follows the active future plan. TRUE under the default sequential plan runs one at a time and warns.
...	Additional arguments forwarded to ffmpeg_batch , such as verify, manifest, checksums, and progress.

Details

Supply the sampling rate once as the single fps or interval argument, which applies to every row. Or supply it per row as an fps or interval column, which overrides the argument of the same name. Supply exactly one of the two, *fps or interval*, across arguments and columns.

Value

The [tibble](#) returned by [ffmpeg_batch](#): jobs with an added command column. When outdir was derived, it also has the resolved outdir column. When run = TRUE, it has a success column, plus any columns the forwarded arguments add, such as verified.

See Also

[sample_frames\(\)](#) for the single-video form. [ffmpeg_batch\(\)](#) for the batch runner and the arguments passed on through ... [extract_frame_batch\(\)](#) for the batch function that takes a list of frames.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#),

```
normalize_audio_batch(), picture_in_picture(), picture_in_picture_batch(), sample_frames(),
segment_video(), segment_video_batch(), separate_audio_video(), separate_audio_video_batch(),
standardize_video(), standardize_video_batch(), strip_metadata(), strip_metadata_batch()
```

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input = c(video, video),
  outdir = c(file.path(tempdir(), "a"), file.path(tempdir(), "b"))
)
# run = FALSE compiles one command per input without calling FFmpeg
sample_frames_batch(jobs, fps = 2, run = FALSE)
```

segment_video

Segment Video

Description

Use FFmpeg to quickly break a single video file into multiple smaller video files (with the same encoding). Pairs of start and stop timestamps set the segments. The function names each segment file after infile. It appends a suffix of an underscore (_) and an integer indicating which segment, based on the order provided in start and end. The glossary in `vignette("tidymedia")` explains media terms such as codec, keyframe and stream copy.

Usage

```
segment_video(
  infile,
  start,
  end,
  outfiles = NULL,
  reencode = TRUE,
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE
)
```

Arguments

infile A string containing the path to a video file.

start	A vector containing one or more timestamps indicating the start of each segment to create. Can be either a numeric vector indicating seconds or a character vector with time duration syntax. Must have the same length as end.
end	A vector containing one or more timestamps indicating the stop of each segment to create. Can be either a numeric vector indicating seconds or a character vector with time duration syntax. Must have the same length as start.
outfiles	Either NULL or a character vector indicating the filename (with extension) for each segment to create. If NULL, will append a zero-padded integer to infile. If not NULL, must have the same length as start, and each element must be a single string. So the function accepts a list of strings as well as a character vector. This function itself refuses a missing value or a number in any position, before the per-segment step below it can. Two segments given the same path are refused before any segment is cut.
reencode	A logical passed to ffmpeg_seek : cut each segment frame-accurately by re-encoding (TRUE, default) or with a fast, lossless copy that snaps to keyframes (FALSE). See ffmpeg_seek for the trade-off.
video_codec	A string naming the output video codec, or NULL (default) to leave it unset. Then the output container's default encoder is used, and the compiled command is the same as one that never named a codec. A stream copy runs no encoder, so naming a codec (or a hardware backend) alongside reencode = FALSE is an error.
audio_codec	A string naming the output audio codec. "copy" (default) stream-copies the audio through untouched. Name an encoder, such as "aac", to transcode it. NULL leaves the codec unset, so the output container's default encoder is used. A stream copy (reencode = FALSE) always copies the audio, so any other value is an error there. Stream-copying fails if the output container cannot hold the source audio codec (e.g. FLAC in .mp4). Name an encoder instead.
hardware	The encoder backend. "none" (default) uses the software video_codec. "nvcenc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). A backend uses its own encoder for the family of video_codec. For example, "libx264" becomes "h264_nvcenc" or "h264_videotoolbox". With the default video_codec = NULL, the H.264 family is assumed. So a non-H.264 container, such as .webm, needs an explicit HEVC- or AV1-family video_codec (AV1 only under "nvcenc"). See has_hardware_encoder for availability and its caveats. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under run = FALSE. The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also contradict itself by asking for GPU encoding on a cut that stream-copies. Such a call is refused for the contradiction first, whether or not this machine has the encoder. The stream-copy conflict named under reencode is caught first, so such a call aborts without probing.
fallback	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE encodes in software with a message. FALSE (default) aborts

	instead. With <code>video_codec = NULL</code> , the fallback leaves the codec unset rather than picking one, so the codec never changes silently. A <code>video_codec</code> in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
quality	A number, or <code>NULL</code> (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. <code>libx264</code> and <code>libx265</code> read it as <code>-crf</code> (0 to 51). The <code>nvenc</code> encoders read it as <code>-cq</code> (0 to 51), and the <code>videotoolbox</code> encoders as <code>-q:v</code> (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as <code>libvpx-vp9</code> , is refused with <code>quality</code> set. So is a <code>video_codec</code> of "copy", or of <code>NULL</code> under <code>hardware = "none"</code> . When <code>fallback = TRUE</code> falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale. A stream-copying cut (<code>reencode = FALSE</code>) runs no encoder, so it refuses <code>quality</code> too.
audio_stream	The audio track to carry into the output, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. <code>NULL</code> (default) carries every audio track. The every-track family reads <code>NULL</code> this way: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = <code>NULL</code>)
run	A logical: run each segment's command (<code>TRUE</code> , default) or only compile them (<code>FALSE</code>).
parallel	A logical passed to ffmpeg_batch : cut segments in parallel with <code>furrr</code> (<code>TRUE</code>) or sequentially (<code>FALSE</code> , default). Parallelism follows the active future plan; <code>TRUE</code> under the default sequential plan runs one segment at a time and warns. Set a plan first, e.g. <code>future::plan(future::multisession)</code> .

Value

The [tibble](#) returned by [ffmpeg_batch](#): one row per segment with its command (and, when `run = TRUE`, success).

References

<https://ffmpeg.org/ffmpeg-utils.html#time-duration-syntax>

See Also

[ffmpeg_seek\(\)](#), the pipeline function it uses to cut; [ffmpeg_batch\(\)](#), the runner; [has_hardware_encoder\(\)](#) for the hardware argument; [segment_video_batch\(\)](#) for the many-file form.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#),

crop_video(), crop_video_batch(), extract_audio(), extract_audio_batch(), extract_frame(), extract_frame_batch(), format_for_web(), format_for_web_batch(), normalize_audio(), normalize_audio_batch(), picture_in_picture(), picture_in_picture_batch(), sample_frames(), sample_frames_batch(), segment_video_batch(), separate_audio_video(), separate_audio_video_batch(), standardize_video(), standardize_video_batch(), strip_metadata(), strip_metadata_batch()

Other audio selection functions: anonymize_video(), anonymize_video_batch(), audio_stream, compare_videos(), compare_videos_batch(), convert_audio(), convert_audio_batch(), crop_video(), crop_video_batch(), extract_audio(), extract_audio_batch(), format_for_web(), format_for_web_batch(), normalize_audio(), normalize_audio_batch(), picture_in_picture(), picture_in_picture_batch(), segment_video_batch(), separate_audio_video(), separate_audio_video_batch(), standardize_video(), standardize_video_batch()

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# Two segments; run = FALSE compiles one command per segment
segment_video(video, start = c(0, 0.5), end = c(0.5, 1), run = FALSE)
```

segment_video_batch	<i>Segment Many Videos From a Jobs Table</i>
---------------------	--

Description

Cut segments across many input files from a single jobs tibble. This is the **batch** (table-driven) form of `segment_video()`, for when your segments span more than one input. Each row is one segment; the four required columns name its source, destination, and cut points. This is a thin wrapper over `ffmpeg_batch`: one reproducible compiled command per segment. The glossary in `vignette("tidymedia")` explains media terms such as codec, keyframe and stream copy.

Usage

```
segment_video_batch(
  jobs,
  reencode = TRUE,
  video_codec = NULL,
  audio_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per segment and (at least) the columns input (source path), start and end (cut points). Each cut-point column is a numeric column of seconds or a character column with time-duration syntax. Two optional columns are recognized: output (destination path) and reencode (a logical; see the reencode argument). If output is absent, the function derives one per row by appending <code>_<n>.<ext></code> to each input's basename. The segment number restarts at 1 for each input file (the same rule as segment_video). Two rows given the same output path are refused before any row runs. A <code>video_codec</code> or <code>audio_codec</code> column overrides that argument per row, with NA meaning "leave the codec unset" (the column's way of writing the argument's NULL). An <code>audio_stream</code> column likewise overrides that argument per row, with NA meaning "keep every audio track" (the column's way of writing that argument's NULL). A numeric quality column overrides the quality argument per row (see quality). Any other columns are ignored.
reencode	A logical passed to ffmpeg_seek : cut each segment frame-accurately by re-encoding (TRUE, default) or with a fast, lossless copy that snaps to keyframes (FALSE). See ffmpeg_seek for the trade-off. Applies to every row, unless jobs carries a reencode column, which overrides this argument on a per-row basis.
video_codec	A string naming the output video codec, applied to every row lacking a <code>video_codec</code> column. NULL (default) leaves it unset, so each segment keeps its container's default encoder. A row can resolve to a codec while cutting by stream copy (<code>reencode = FALSE</code> , as an argument or a column). That row is an error, because no encoder runs on that path.
audio_codec	A string naming the output audio codec, applied to every row lacking an <code>audio_codec</code> column. "copy" (default) stream-copies the audio; name an encoder to re-encode it, or NULL to leave the codec unset. A row can resolve to anything but "copy" while cutting by stream copy (<code>reencode = FALSE</code> , as an argument or a column). That row is an error. So split a jobs table that mixes stream-copy rows with a re-encoding <code>audio_codec</code> into separate calls.
hardware, fallback	The encoder backend and its fallback behavior, applied to the whole batch. They are a property of the machine, not of a row, so neither is read as a jobs column. See segment_video() . Because hardware is batch-wide, a non-"none" value conflicts with a stream-copy row on its own, even one naming no codec. So split a jobs table that mixes <code>reencode = FALSE</code> rows with GPU encoding into separate calls. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also contradict itself by asking for GPU encoding on a cut that stream-copies. Such a call is refused for the contradiction first, whether or not this machine has the encoder. The stream-copy conflict named under <code>reencode</code> is caught first, so such a call aborts without probing.

quality	A number, or NULL (default), applied to each row unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See segment_video() for the encoders, their flags and ranges, and the values it refuses. A cell on a row that stream-copies (reencode = FALSE) is refused too, because no encoder runs on that row.
audio_stream	The audio track to carry into each output, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. Without an audio_stream column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The every-track family reads NULL as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <i>_batch</i> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <i>_batch</i> forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from audio_input, the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run each segment's command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical passed to ffmpeg_batch : cut segments in parallel with furrr (TRUE) or sequentially (FALSE, default). Parallelism follows the active future plan; TRUE under the default sequential plan runs one segment at a time and warns. Set a plan first, e.g. <code>future::plan(future::multisession)</code> .
...	Additional arguments forwarded to ffmpeg_batch , such as <code>verify</code> , <code>manifest</code> , <code>checksums</code> , and <code>progress</code> .

Value

The [tibble](#) returned by [ffmpeg_batch](#): jobs with an added command column. When output was derived, it also has the resolved output column. When run = TRUE, it has a success column, plus any columns the forwarded arguments add, such as `verified`.

References

<https://ffmpeg.org/ffmpeg-utils.html#time-duration-syntax>

See Also

[segment_video\(\)](#) for the single-input, parallel-vector form. [ffmpeg_batch\(\)](#) for the batch runner and the arguments forwarded through ... [has_hardware_encoder\(\)](#) for the hardware argument. [ffmpeg_seek\(\)](#) for the cut trade-off.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#),

crop_video(), crop_video_batch(), extract_audio(), extract_audio_batch(), extract_frame(), extract_frame_batch(), format_for_web(), format_for_web_batch(), normalize_audio(), normalize_audio_batch(), picture_in_picture(), picture_in_picture_batch(), sample_frames(), sample_frames_batch(), segment_video(), separate_audio_video(), separate_audio_video_batch(), standardize_video(), standardize_video_batch(), strip_metadata(), strip_metadata_batch()

Other audio selection functions: anonymize_video(), anonymize_video_batch(), audio_stream, compare_videos(), compare_videos_batch(), convert_audio(), convert_audio_batch(), crop_video(), crop_video_batch(), extract_audio(), extract_audio_batch(), format_for_web(), format_for_web_batch(), normalize_audio(), normalize_audio_batch(), picture_in_picture(), picture_in_picture_batch(), segment_video(), separate_audio_video(), separate_audio_video_batch(), standardize_video(), standardize_video_batch()

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input = c(video, video),
  output = c("a.mp4", "b.mp4"),
  start = c(0, 0.5),
  end = c(0.5, 1)
)
# run = FALSE compiles one command per segment without calling FFmpeg
segment_video_batch(jobs, run = FALSE)
```

separate_audio_video *Split a media file into separate audio and video files*

Description

By default, each stream is copied, not re-encoded (audio_codec = "copy", video_codec = "copy"). A copy loses no quality and is fast. Each output container must then support the source codec. For example, write AAC audio from an MP4 to .aac or .m4a, not to .mp3. To re-encode a stream, name an encoder (audio_codec = "libmp3lame"). Pass NULL to set no codec option, so the output extension picks the encoder. Each argument governs only its own output file. Where the video is re-encoded, hardware = "nvenc" or "videotoolbox" moves that encode onto a GPU. The audio output is never affected. The glossary in vignette("tidymedia") explains media terms such as codec, container and stream copy.

Usage

```
separate_audio_video(
  infile,
  audiofile,
  videofile,
  audio_codec = "copy",
  video_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
```

```

    quality = NULL,
    audio_stream = NULL,
    run = TRUE
  )

```

Arguments

<code>infile</code>	A string containing the path to a media file.
<code>audiofile</code>	A string containing the path of the audio file to write.
<code>videofile</code>	A string containing the path of the video file to write.
<code>audio_codec</code>	A string that names the encoder for <code>audiofile</code> . It goes to FFmpeg's <code>-codec:a</code> . The default "copy" copies the audio stream with no quality loss. A codec name (e.g. "libmp3lame") re-encodes it. NULL sets no <code>-codec:a</code> , so the <code>audiofile</code> extension picks the encoder.
<code>video_codec</code>	A string that names the encoder for <code>videofile</code> . It goes to FFmpeg's <code>-codec:v</code> . The default "copy" copies the video stream with no quality loss. A codec name (e.g. "libx264") re-encodes it. NULL sets no <code>-codec:v</code> , so the <code>videofile</code> extension picks the encoder.
<code>hardware</code>	The encoder backend for <code>videofile</code> . "none" (default) uses the software <code>video_codec</code> . "nvenc" uses NVIDIA GPU encoding, and "videotoolbox" uses Apple GPU encoding. Each uses its own encoder for the family of <code>video_codec</code> . For example, "libx264" becomes "h264_nvenc" or "h264_videotoolbox". With <code>video_codec = NULL</code> , the family is H.264. Only video is encoded on the GPU, so this never affects <code>audiofile</code> . The default <code>video_codec = "copy"</code> is a stream copy, which runs no encoder at all. So a hardware other than "none" with <code>video_codec = "copy"</code> is an error. Name an encoder or pass <code>video_codec = NULL</code> . See has_hardware_encoder for availability and its caveats. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. The stream-copy conflict above is caught first, so such a call aborts without asking FFmpeg.
<code>fallback</code>	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE encodes in software with a message. FALSE (default) aborts instead. With <code>video_codec = NULL</code> , the fallback leaves the codec unset rather than injecting one. A <code>video_codec</code> in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
<code>quality</code>	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264 and libx265 read it as <code>-crf</code> (0 to 51). The nvenc encoders read it as <code>-cq</code> (0 to 51), and the videotoolbox encoders as <code>-q:v</code> (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as libvpx-vp9, is refused with quality set. So is a <code>video_codec</code> of "copy", or of NULL under <code>hardware = "none"</code> . When <code>fallback = TRUE</code> falls back to

	software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale. It applies to videofile only.
audio_stream	The audio track to write to audiofile, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) keeps every audio track. The every-track family reads NULL this way: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <i>_batch</i> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <i>_batch</i> forms. A container that holds several audio streams (.mka, .m4a) gets them all. A container for one stream only (.aac, .mp3, .wav) makes FFmpeg fail, so name a track to write one of those. Count only the input's <i>audio</i> streams. Do not use the index column of probe_audio , which counts every stream. An input with no audio at all is an FFmpeg error here, because this function writes an audio file. Functions that write only a video file, such as standardize_video , do not fail in that case. videofile is never affected. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from audio_input , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run the commands through FFmpeg (TRUE, default) or return the compiled commands without running them (FALSE).

Value

A named character vector of the two compiled commands (audio, video). It is invisible when run = TRUE. Under run = TRUE, the audio command runs first and the video command runs second. The video command runs whether or not the audio command succeeded. A failed audio command still aborts the call. By then, the video command has written videofile, unless it failed too. See *When the audio output fails*.

When the audio output fails

The two commands run in order: audio first, video second. The video command runs even when the audio command failed, so a failed audio half does not cost you the video. In that case, the call still aborts with the audio failure. That error carries one added line that names the video file that was written. When the video command fails too, the added line is not there. The audio failure is still the error you get. FFmpeg's own output for the failed video command is printed above it.

A time limit reached on the audio command is held like any other audio failure, so the video command still runs. The video command gets a fresh limit of its own, because [with_timeout\(\)](#) limits each program that the call starts, not the call. So a call whose audio half reaches the limit can wait up to two limits, not one.

A failed command treats its own output path by the same rule on either path. It removes a partial file that the run wrote. It leaves a file that was already at that path, and that FFmpeg never wrote to, exactly as it was. So neither failure path promises that the path is empty afterwards. It promises only that nothing half-written is left there. The audio failure's own error says which of the two happened to audiofile. The same error's `tm_video_error` field says what became of the video command. It holds the condition that command raised when it failed too, and NULL when it succeeded.

The default keeps every audio track. So FFmpeg fails when it writes a multi-track input to a container that holds only one track (.aac, .mp3, .wav). When that happens, the error also reports how many audio tracks infile carries, and it names the two ways out. Use `audio_stream` to write one track, or use a container such as .mka or .m4a to keep them all.

The error carries that extra report only when all four of these hold:

- No `audio_stream` was named.
- FFmpeg returned a non-zero exit status.
- `infile` carries more than one audio track.
- The extension of `audiofile` is not among the containers named here as holding several audio streams.

Those containers are .mka, .m4a, .mp4, .mov, .mkv, .webm, .ogg, .opus and .ts. The nine are an exclusion list and not a survey. FFmpeg writes several audio streams into other containers too, .avi and .nut among them. A failure on one of those still gets the report. The container condition keeps the report off a call that already does what the report advises. When a call writes to one of the nine, the failure cannot be the container refusing a second audio stream. The report would then leave unnamed whatever FFmpeg did object to.

If any of the four does not hold, the error you get is the one the run itself raised, whatever that error is. It has the same class, the same status field and the same message. The one difference is the line saying that the video output was written. A failing audio half carries that line when the video command wrote its file and the audio failure is an `rlang` condition. When the exit status is the one that does not hold, there is no exit status to carry. A run that never reached FFmpeg has none.

The report states what the call *did*: the track count, and that every track was mapped into one output. It never states why FFmpeg refused. FFmpeg's own error and exit status are printed beneath it and carried on the condition. They remain the only authority on the cause. Several causes look alike from here. A stream copy into a container that will not hold the source codec fails on a multi-track input too. The default `audio_codec = "copy"` into .mp3 is one example. An unknown encoder and a missing output directory fail the same way.

The condition carries two class names, so a caller can catch it at either width. It is `tidymedia_ffmpeg_exit`, the class that every non-zero FFmpeg exit raises. An exit-status handler catches that class, and the number is on the condition's `tm_status` field. It is also `tidymedia_multitrack_separation`, the class of this report itself. Catch that class when it is this failure in particular you want:

```
tryCatch(
  separate_audio_video("three-tracks.mkv", "audio.mp3", "video.mp4"),
  tidymedia_ffmpeg_exit = function(cnd) cnd$tm_status
)
```

When the report is omitted, the error that reaches the caller is the one the run itself raised, apart from that video-output line. A non-zero exit still answers to `tidymedia_ffmpeg_exit`. A failure that is not an exit answers to neither class here. An FFmpeg that the package cannot locate raises an error with no `tidymedia_` class at all. A reached limit raises `tidymedia_timeout`.

Counting the tracks means running `FFprobe`, so the report is not guaranteed. The error has it when `FFprobe` is available and `infile` can be probed. Otherwise the package omits it silently and leaves FFmpeg's own error alone. So the report may not appear, and its absence is never itself a second failure. The count never runs under `run = FALSE` and never changes the compiled commands. It

is skipped when `audio_stream` names a track, or when `audiofile` names one of the multi-stream containers above. With one track mapped, the track count cannot be what FFmpeg objected to.

See Also

`ffmpeg_map()` and `ffmpeg_codec()`, the pipeline functions it wraps. `has_hardware_encoder()` for the hardware argument. `extract_audio()` to pull out just the audio. `probe_audio()` to list an input's audio tracks.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
separate_audio_video(video, "audio.aac", "video.mp4", run = FALSE)
# transcode the audio to MP3 while copying the video through untouched
separate_audio_video(video, "audio.mp3", "video.mp4",
  audio_codec = "libmp3lame", run = FALSE)
# write only the second audio track (this sample has one, so compile only)
separate_audio_video(video, "audio.aac", "video.mp4",
  audio_stream = 1, run = FALSE)
```

separate_audio_video_batch

Separate Audio and Video for Many Files From a Jobs Table

Description

Split the audio and video streams of many input files from a single jobs tibble. This is the **batch** (table-driven) form of `separate_audio_video()`, for when you have more than one file. Each row is one input that gives **two** outputs. The input, `audiofile` and `videofile` columns are all required. The function is a thin wrapper over `ffmpeg_batch`. It turns every input row into two single-output jobs, one per stream. So a jobs table of N rows returns 2N rows, with one reproducible compiled command per stream. Each stream uses the same map and stream-copy pipeline as `separate_audio_video()`. The glossary in `vignette("tidymedia")` explains media terms such as codec, container and stream copy.

Usage

```

separate_audio_video_batch(
  jobs,
  audio_codec = "copy",
  video_codec = "copy",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)

```

Arguments

jobs	A data frame with one row per input. It has at least an input column (source path), plus audiofile and videofile columns that name the two destinations. All three are required . Like separate_audio_video , this function derives no output paths, because the container extension of a copied stream is the instruction. That extension must match the source codec. No two destinations in a table can be the same path. That covers an audiofile and a videofile in one row, and any two across rows. The function refuses such a table before any row runs. Optional audio_codec and video_codec columns override the arguments of the same name per row. They are character, with NA to set no codec option for that stream. Rows that omit a column fall back to that argument. An optional numeric audio_stream column likewise overrides the audio_stream argument per row. There, NA keeps every audio track in that row's audiofile. A numeric quality column overrides the quality argument per row and applies to that row's videofile (see quality). Any other columns are ignored, with one exception. A reencode column, retired with the argument of the same name, is an error and not a silent no-op.
audio_codec	A string that names the encoder for every audiofile, unless jobs carries an audio_codec column. The default "copy" copies the audio stream with no quality loss. NULL sets no -codec:a. See separate_audio_video .
video_codec	A string that names the encoder for every videofile, unless jobs carries a video_codec column. The default "copy" copies the video stream with no quality loss. NULL sets no -codec:v. See separate_audio_video .
hardware, fallback	The encoder backend for every videofile and its fallback behavior. They apply to the whole batch. They are a property of the machine, not of a row, so neither is read as a jobs column. See separate_audio_video() . hardware is batch-wide, and a stream copy runs no encoder. So a hardware other than "none" conflicts with any row whose video codec resolves to "copy". That includes the default. So split a jobs table that mixes copied and re-encoded video into separate calls. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg

while the command is built, even under `run = FALSE`. The answer is remembered for the rest of the R session. See [refresh_ffmpeg_capabilities](#) to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows. A call can also contradict itself by asking for GPU encoding alongside a stream copy. Such a call is refused for the contradiction first, whether or not this machine has the encoder. The stream-copy conflict above is caught first, so such a call aborts without asking FFMpeg.

quality	A number, or NULL (default), applied to every videofile unless jobs carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See separate_audio_video() for the encoders, their flags and ranges, and the values it refuses. The audiofile never takes it. A cell on a row whose video codec is "copy", the default, is refused, because no encoder runs.
audio_stream	The audio track to write to each audiofile, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) keeps every audio track. Without an <code>audio_stream</code> column, the argument applies to every row. An NA cell in that column means NULL for that row. It does not fall back to the argument. The every-track family reads NULL as every audio track: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their <code>_batch</code> forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their <code>_batch</code> forms. A container that holds several audio streams (.mka, .m4a) gets them all. A container for one stream only (.aac, .mp3, .wav) makes FFMpeg fail, so name a track to write one of those. Count only the input's <i>audio</i> streams. Do not use the index column of probe_audio , which counts every stream. An input with no audio at all is an FFMpeg error here, because this function writes an audio file. Functions that write only a video file, such as standardize_video , do not fail in that case. videofile is never affected. A track the input does not have gives an FFMpeg error, not an R one. See audio_stream for how this differs from <code>audio_input</code> , the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run each command through FFMpeg (TRUE, default) or only compile them for inspection (FALSE).
parallel	A logical: process the jobs in parallel with furrr (TRUE) or one at a time (FALSE, default). See ffmpeg_batch for the future plan requirement.
...	Additional arguments forwarded to ffmpeg_batch (e.g. <code>verify</code> , <code>manifest</code> , <code>progress</code>).

Value

A [tibble](#) with **two rows per input**, one per stream. It has the reshaped input, a single output path, a stream marker ("audio" or "video"), and an added command column. When `run = TRUE`,

it also has a success column. A run also gives `verified` and the provenance manifest, each when requested via `...`. When jobs supplies either codec column, a single codec column carries each row's resolved encoder for its own stream (NA where none is set). When `audio_stream` is supplied as either the argument or a jobs column, an `audio_stream` column likewise carries each row's resolved track. That is the selected index on an audio row. It is NA on every video row, which takes no audio, and on an audio row that named no track. So NA does not by itself mark a video row. Read the `stream` column for that. The columns match the output of the other `_batch` functions, plus the stream marker. See [ffmpeg_batch](#).

Failed audio outputs

A row whose audio command does not finish cleanly is recorded as `success = FALSE`, and the batch does not abort. One warning for the whole batch, emitted **once**, names such rows. It lists every affected input row and the ways out.

A row reaches that warning only when all four of these hold:

- It named no `audio_stream`.
- The row is recorded `success = FALSE`.
- Its input carries more than one audio track.
- The extension of its audiofile is not among the containers named here as holding several audio streams.

Those containers are `.mka`, `.m4a`, `.mp4`, `.mov`, `.mkv`, `.webm`, `.ogg`, `.opus` and `.ts`. No exit status is among those conditions, and the difference from [separate_audio_video](#) is deliberate. The batch runner records *whether* a row succeeded and not *how*. So it records a non-zero exit, a hard error and a reached limit the same way. It treats alike a row put here by any of them. The nine are an exclusion list and not a survey. FFmpeg writes several audio streams into other containers too, `.avi` and `.nut` among them. A row that fails on one of those is still named. The container condition keeps a row off the list when it already does what the warning advises. Such a row is silently not named. A batch whose failed audio rows all write to those nine does not warn at all. The headline count follows the rows actually named.

Each bullet of the warning states what that row *did*: its track count, and that every track was mapped into one output. It never states why FFmpeg refused. Several causes look alike from here. Examples are a stream copy into a container that will not hold the source codec, an unknown encoder and a missing output directory.

The check runs FFprobe on the failed rows only. So the function emits the warning when FFprobe is available and the input can be probed, and skips it silently otherwise. The warning may not appear, and its absence is never itself a second failure. The check never runs under `run = FALSE` and never changes any compiled command. Suppress the warning with `suppressWarnings(classes = "tidymedia_multitrack_separation")`.

The warning names the same event as the error of [separate_audio_video](#) and answers to the same class. But it carries no exit status: no `tm_status` field, and no `tidymedia_ffmpeg_exit` class. The batch runner records, per row, *whether* the row succeeded, not *how* FFmpeg exited. The success column holds that record. So the exit number is gone by the time this warning is assembled. To catch a specific row's exit status, use [separate_audio_video\(\)](#).

See Also

`separate_audio_video()`, the one-file function it wraps. `ffmpeg_batch()`, the batch runner. `has_hardware_encoder()` for the hardware argument. `segment_video_batch()` for the other batch function where one input file can give several outputs.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `standardize_video()`, `standardize_video_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input      = c(video, video),
  audiofile  = c("a1.aac", "a2.aac"),
  videofile  = c("v1.mp4", "v2.mp4")
)
# run = FALSE compiles two commands per input without calling FFmpeg
separate_audio_video_batch(jobs, run = FALSE)
```

set_program

Set the location of a dependency program

Description

`set_program()` saves the location of a program, so the package can find it in later sessions. `set_ffmpeg()`, `set_ffprobe()`, `set_ffplay()` and `set_mediainfo()` do the same for one program each.

The location goes in a file named after the program, such as `ffmpeg_location.txt`. The file is under `tools::R_user_dir("tidymedia", "config")`. `find_ffmpeg()` and the other `find_*`() functions read it when the program is not on the PATH.

Usage

```
set_program(
  program = c("ffmpeg", "ffprobe", "ffplay", "mediainfo"),
  location,
```

```

    confirm = TRUE
)

set_mediainfo(location, confirm = TRUE)

set_ffmpeg(location, confirm = TRUE)

set_ffprobe(location, confirm = TRUE)

set_ffplay(location, confirm = TRUE)

```

Arguments

program	A string naming the program to set the location for.
location	A string with the location of the program.
confirm	Whether to ask before the call writes the location. TRUE, the default, asks. In a session where no one can answer, TRUE gives an error. FALSE writes without asking.

Details

The file stays after the session ends, so the call asks you to confirm first. It writes nothing until you agree. The question shows the location the call writes, which is the location as you typed it. The one change is a leading ~, which is written as your home folder in full. The question also shows the full path of the file. If you say no, the call changes nothing.

In a session where no one can answer, the call gives an error. Pass `confirm = FALSE` to write without the question, for example in a script that runs on its own.

For locations saved by versions before 0.2.0, see the section "Locations saved by earlier versions" in [find_ffmpeg\(\)](#).

Value

Invisibly, TRUE when the call wrote the location and FALSE when you said no.

See Also

[find_ffmpeg\(\)](#) and the other `find_*`() functions to find a program, and [install_on_win\(\)](#) to download FFmpeg on Windows.

Other program management functions: [find_ffmpeg\(\)](#), [install_on_win\(\)](#), [program_status\(\)](#), [unset_program\(\)](#)

Examples

```

## Not run:
# Point tidymedia at a binary in a non-standard location; asks first
set_mediainfo("C:/Program Files/MediaInfo/mediainfo.exe")

# In an unattended script, where there is no one to ask
set_mediainfo("C:/Program Files/MediaInfo/mediainfo.exe", confirm = FALSE)

```

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

standardize_video	<i>Standardize a video to a reproducible format</i>
-------------------	---

Description

Re-encode a video to a consistent, reproducible format for analysis. The format is one video codec and pixel format, and optionally a resolution and frame rate, with `+faststart` for smooth playback. `format_for_web` uses a fixed recipe for web delivery. Here, every part of the standard is an argument. So a lab can set its own house format once and apply it across a dataset. The glossary in `vignette("tidymedia")` explains media terms such as codec, pixel format and frame rate.

Usage

```
standardize_video(
  infile,
  outfile,
  width = NULL,
  height = NULL,
  fps = NULL,
  video_codec = "libx264",
  audio_codec = "copy",
  pixel_format = "yuv420p",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE
)
```

Arguments

<code>infile</code>	A string containing the path to a video file.
<code>outfile</code>	A string containing the path of the video file to write.
<code>width</code>	The output width in pixels (a positive number), or <code>NULL</code> (default) to leave the width unconstrained.
<code>height</code>	The output height in pixels (a positive number), or <code>NULL</code> (default) to leave the height unconstrained.
<code>fps</code>	The output frame rate (a positive number or FFmpeg framerate expression such as <code>"30000/1001"</code>), or <code>NULL</code> (default) to keep the input frame rate.
<code>video_codec</code>	A string naming the output video codec (default <code>"libx264"</code>). <code>NULL</code> emits no <code>-codec:v</code> and lets the output container's default encoder decide. <code>NULL</code> is how you opt out of the H.264 default for a container that does not hold it. For a <code>.webm</code> output, pass <code>video_codec = NULL</code> <i>and</i> <code>audio_codec = NULL</code> , because the

	default audio_codec = "copy" would otherwise carry a codec WebM cannot hold.
audio_codec	A string naming the output audio codec. The default "copy" stream-copies the source audio unchanged. Name a real encoder, such as "aac", when the source audio codec cannot be copied into the output container. NULL emits no -codec:a and lets the container's default encoder decide.
pixel_format	A string naming the output pixel format (default "yuv420p").
hardware	The encoder backend. "none" (default) uses the software video_codec. "nvc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). A backend uses its own encoder for the family of video_codec. For example, "libx264" becomes "h264_nvc" or "h264_videotoolbox". See has_hardware_encoder for availability and its caveats. This applies to video only. audio_codec is never hardware-accelerated. Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under run = FALSE. The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it.
fallback	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE re-encodes with the software video_codec and a message. FALSE (default) aborts instead. This keeps output reproducible by never changing the codec silently. A video_codec in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
quality	A number, or NULL (default) to leave the encoder's own default in place. It is the encoder's own rate-control value, passed through unchanged. libx264 and libx265 read it as -crf (0 to 51). The nvc encoders read it as -cq (0 to 51), and the videotoolbox encoders as -q:v (1 to 100). Each scale is its own: the same number means something different on each encoder. A value outside the encoder's range is refused. An encoder outside those seven, such as libvpx-vp9, is refused with quality set. So is a video_codec of "copy", or of NULL under hardware = "none". When fallback = TRUE falls back to software, the value is dropped and the message says so, because it belonged to the hardware encoder's scale.
audio_stream	The audio track to carry into the output, as a number that counts from 0 among the <i>audio tracks</i> of the input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. The every-track family reads NULL this way: separate_audio_video , standardize_video , anonymize_video , crop_video , segment_video and format_for_web , and their _batch forms. The first-track family reads it as the first audio track only: extract_audio , convert_audio and normalize_audio , and their _batch forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See audio_stream for how this differs from audio_input, the input index on compare_videos and picture_in_picture . (default = NULL)
run	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

The default standard, `standardize_video(infile, outfile)`, re-encodes to H.264 video (`video_codec = "libx264"`) with `pixel_format = "yuv420p"` and `-movflags +faststart`. It keeps the source resolution and frame rate. Audio is stream-copied unchanged (`-c:a copy`) unless `audio_codec` names an encoder. The same input therefore always compiles to a byte-identical command. Loudness standardization is out of scope. For that, see [normalize_audio](#).

Resolution follows width and height:

- With both, the output has exactly those dimensions.
- With only one, the aspect ratio is kept, and the other dimension is rounded to the nearest even number (FFmpeg's -2).
- With neither, the source resolution is kept, but odd dimensions are rounded down to the nearest even value. `yuv420p` and `libx264` require this, and it changes nothing for input that is already even. So the output always encodes.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

See Also

[ffmpeg_scale\(\)](#), [ffmpeg_codec\(\)](#), and [ffmpeg_pixel_format\(\)](#), among the pipeline functions it wraps; [has_hardware_encoder\(\)](#) for the hardware toggle; [standardize_video_batch\(\)](#) for the many-file form.

Other task functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [concatenate_videos\(\)](#), [concatenate_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [extract_frame\(\)](#), [extract_frame_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [sample_frames\(\)](#), [sample_frames_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video_batch\(\)](#), [strip_metadata\(\)](#), [strip_metadata_batch\(\)](#)

Other audio selection functions: [anonymize_video\(\)](#), [anonymize_video_batch\(\)](#), [audio_stream](#), [compare_videos\(\)](#), [compare_videos_batch\(\)](#), [convert_audio\(\)](#), [convert_audio_batch\(\)](#), [crop_video\(\)](#), [crop_video_batch\(\)](#), [extract_audio\(\)](#), [extract_audio_batch\(\)](#), [format_for_web\(\)](#), [format_for_web_batch\(\)](#), [normalize_audio\(\)](#), [normalize_audio_batch\(\)](#), [picture_in_picture\(\)](#), [picture_in_picture_batch\(\)](#), [segment_video\(\)](#), [segment_video_batch\(\)](#), [separate_audio_video\(\)](#), [separate_audio_video_batch\(\)](#), [standardize_video_batch\(\)](#)

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
# The documented default standard (H.264 / yuv420p / +faststart)
standardize_video(video, "std.mp4", run = FALSE)
# Pin resolution and frame rate too
standardize_video(video, "std.mp4", width = 1280, height = 720, fps = 30,
  run = FALSE)
# Carry only the second audio track instead of all of them
standardize_video(video, "std.mp4", audio_stream = 1, run = FALSE)
```

standardize_video_batch

Standardize Many Videos From a Jobs Table

Description

Re-encode many files to a reproducible format, using one jobs table. This is the **batch** form of `standardize_video()`, for when you have more than one video to standardize. Each row is one input, and the only required column names its source. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible command for each input. The glossary in `vignette("tidymedia")` explains media terms such as codec, pixel format and frame rate.

Usage

```
standardize_video_batch(
  jobs,
  width = NULL,
  height = NULL,
  fps = NULL,
  video_codec = "libx264",
  audio_codec = "copy",
  pixel_format = "yuv420p",
  hardware = c("none", "nvenc", "videotoolbox"),
  fallback = FALSE,
  quality = NULL,
  audio_stream = NULL,
  run = TRUE,
  parallel = FALSE,
  ...
)
```

Arguments

jobs	A data frame with one row per input. It needs at least an input column, the source path. An optional output column names the destination. Without it, each row's output name adds <code>_standardized</code> to the input's base name and keeps its extension. For example, <code>clip.mkv</code> becomes <code>clip_standardized.mkv</code> . Two rows naming the same output path are refused before any row runs. That is a path repeated in the output column, or a repeated input when there is no output column. A column can override any of the six format arguments for each row: <code>width</code> , <code>height</code> , <code>fps</code> , <code>video_codec</code> , <code>audio_codec</code> and <code>pixel_format</code> . An argument with no column applies its value to every row. In either codec column, NA leaves that row's codec unset. That is the column form of <code>video_codec = NULL</code> or <code>audio_codec = NULL</code> . In a <code>width</code> , <code>height</code> , <code>fps</code> or <code>pixel_format</code> column, NA is an error. <code>pixel_format</code> has no unset state to express. <code>width</code> , <code>height</code> and <code>fps</code> do accept NULL as arguments, but their columns have no NA form for it. An <code>audio_stream</code> column overrides the <code>audio_stream</code> argument for each row, and
------	--

	NA keeps that row on every audio track. A numeric quality column overrides the quality argument per row (see <code>quality</code>). Any other columns are ignored.
<code>width, height</code>	Optional target dimensions for every row, unless <code>jobs</code> has a column of the same name (see <code>jobs</code>). When only one is given, the other is derived to keep the aspect ratio. When neither is given, the frame is floor-cropped to even dimensions, so odd-sized sources encode. (default = NULL)
<code>fps</code>	Optional target frame rate applied to every row, unless <code>jobs</code> carries an <code>fps</code> column. (default = NULL, i.e. leave the frame rate unchanged)
<code>video_codec</code>	A string naming the video codec for every row, unless <code>jobs</code> has a <code>video_codec</code> column. In that column, NA leaves that row's codec unset. The default is "libx264". NULL emits no <code>-codec:v</code> and lets the output container's default encoder decide. For a .webm output, pass <code>audio_codec = NULL</code> too, because the default "copy" would otherwise carry a codec WebM cannot hold.
<code>audio_codec</code>	A string naming the audio codec for every row, unless <code>jobs</code> has an <code>audio_codec</code> column. In that column, NA leaves that row's codec unset. "copy" (default) stream-copies the audio through untouched. Name an encoder, such as "aac", when the source audio cannot be copied into the output container.
<code>pixel_format</code>	A string naming the pixel format applied to every row, unless <code>jobs</code> carries a <code>pixel_format</code> column. (default = "yuv420p")
<code>hardware</code>	The encoder backend for every row. "none" is the default. "nvenc" uses NVIDIA GPU encoding (H.264, HEVC and AV1), and "videotoolbox" uses Apple GPU encoding (H.264 and HEVC). It applies to the whole batch and is not read as a column. See standardize_video and has_hardware_encoder . Resolving a hardware backend asks this FFmpeg build which encoders it has. So the first such call that re-encodes the video runs FFmpeg while the command is built, even under <code>run = FALSE</code> . The answer is remembered for the rest of the R session. See refresh_ffmpeg_capabilities to discard it. This function checks that the encoder is available before any row runs. So an unavailable encoder aborts naming this function, not the internal step that runs the rows.
<code>fallback</code>	A logical. When a hardware other than "none" is requested but its encoder is unavailable, TRUE re-encodes with the software <code>video_codec</code> and a message. FALSE (default) aborts instead. A <code>video_codec</code> in a family that the backend has no encoder for is a wrong argument, not an absent encoder. So it aborts whatever fallback says.
<code>quality</code>	A number, or NULL (default), applied to each row unless <code>jobs</code> carries a numeric quality column. In that column, NA leaves that row's encoder default in place, whatever the argument says. The value is the encoder's own rate-control value, passed through unchanged. Each cell is checked against the encoder its own row resolves to. A wrong cell is refused before any row runs, and the error names this function and the row. See standardize_video() for the encoders, their flags and ranges, and the values it refuses.
<code>audio_stream</code>	The audio track to carry into each output, as a number that counts from 0 among the <i>audio tracks</i> of each row's input. 0 is the first audio track and 1 is the second. Other streams in the file, such as video, do not count. NULL (default) carries every audio track. Without an <code>audio_stream</code> column, the argument applies to every row. An NA cell in that column means NULL for that row. It does

not fall back to the argument. The every-track family reads NULL as every audio track: `separate_audio_video`, `standardize_video`, `anonymize_video`, `crop_video`, `segment_video` and `format_for_web`, and their `_batch` forms. The first-track family reads it as the first audio track only: `extract_audio`, `convert_audio` and `normalize_audio`, and their `_batch` forms. The function does not carry subtitle or data streams in either case. A track the input does not have gives an FFmpeg error, not an R one. See `audio_stream` for how this differs from `audio_input`, the input index on `compare_videos` and `picture_in_picture`. (default = NULL)

<code>run</code>	A logical: run each input's command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).
<code>parallel</code>	A logical passed to <code>ffmpeg_batch</code> : standardize in parallel with furrr (TRUE) or sequentially (FALSE, default). Parallelism follows the active <code>future</code> plan; TRUE under the default sequential plan runs one input at a time and warns. Set a plan first, e.g. <code>future::plan(future::multisession)</code> .
<code>...</code>	Additional arguments forwarded to <code>ffmpeg_batch</code> , such as <code>verify</code> , <code>manifest</code> , <code>checksums</code> , and <code>progress</code> .

Value

The `tibble` returned by `ffmpeg_batch`: jobs with an added `command` column. When output was derived, it also has the resolved output column. When `run = TRUE`, it has a `success` column, plus any columns the forwarded arguments add, such as `verified`.

See Also

`standardize_video()` for the single-input form; `ffmpeg_batch()` for the batch runner and the arguments forwarded through `...`; `segment_video_batch()` and `extract_frame_batch()` for the other batch task functions.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `strip_metadata()`, `strip_metadata_batch()`

Other audio selection functions: `anonymize_video()`, `anonymize_video_batch()`, `audio_stream`, `compare_videos()`, `compare_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(
  input = c(video, video),
```

```

    output = c("a.mp4", "b.mp4"),
    width = c(640, 320)
)
# run = FALSE compiles one command per input without calling FFmpeg
standardize_video_batch(jobs, run = FALSE)

```

strip_metadata

Strip identifying metadata from a media file

Description

Remove a media file's container and global metadata tags, together with any chapters, and write a de-identified copy. The tags include creation time, GPS or other location, device make and model, title and comment. Use it to de-identify research recordings, for example for an IRB (a research ethics board). The audio and video streams are **stream-copied**, not re-encoded. So the operation is lossless and fast, and the picture and sound are bit-for-bit unchanged. That includes any rotation display matrix, which is stream side data and not a metadata tag. The glossary in `vignette("tidymedia")` explains media terms such as container, stream and stream copy.

Usage

```
strip_metadata(infile, outfile, run = TRUE)
```

Arguments

<code>infile</code>	A string containing the path to a media file.
<code>outfile</code>	A string containing the path of the de-identified file to write. Use the same container extension as <code>infile</code> so the copied streams remux cleanly.
<code>run</code>	A logical: run the command through FFmpeg (TRUE, default) or return the compiled command without running it (FALSE).

Details

The output is written bit-exactly (`-fflags +bitexact`). So FFmpeg does not stamp the container again with a fresh `creation_time`, or with an encoder tag that names its own version. Either tag would defeat de-identification and reproducibility.

Because the streams are copied and not re-encoded, some data is not removed. Those are identifiers **inside** the encoded bitstream, and per-stream metadata such as `handler_name` or `language`. Removing them would require one of two things. One is re-encoding, which is out of scope for this function, so use the direct command `ffmpeg` for it. The other is per-stream metadata mapping that must probe the file first.

Value

The compiled FFmpeg command (invisibly when `run = TRUE`).

See Also

`anonymize_video()` removes faces or regions from the picture, for visual de-identification. `probe_container()` and `mediainfo_query()` inspect a file's metadata before and after. This function wraps the pipeline functions `ffmpeg_copy()` and `ffmpeg_output_options()`. `strip_metadata_batch()` is the many-file form.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata_batch()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
strip_metadata(video, "clean.mp4", run = FALSE)
```

strip_metadata_batch *Strip Metadata From Many Files From a Jobs Table*

Description

De-identify many files, using one jobs table. This is the **batch** form of `strip_metadata()`, for when you have more than one file to scrub. Each row is one input, and the only required column names its source. The function is a thin wrapper over `ffmpeg_batch`. It builds one reproducible stream-copy strip command for each input. Each command uses the same steps as `strip_metadata()`, so it is bit-exact and drops the same metadata. The glossary in `vignette("tidymedia")` explains media terms such as stream and stream copy.

Usage

```
strip_metadata_batch(jobs, run = TRUE, parallel = FALSE, ...)
```

Arguments

jobs	A data frame with one row per input. It needs at least an input column, the source path. An optional output column names the destination. Without it, each row's output name adds <code>_stripped</code> to the input's base name and keeps its extension. For example, <code>clip.mkv</code> becomes <code>clip_stripped.mkv</code> . Any two rows with the same output path are rejected, so one file cannot silently overwrite another. That happens with a duplicated input and no output column, or with a repeated explicit output. Any other columns are ignored, because the scrub has no per-row settings.
run	A logical: run each input's command through FFmpeg (TRUE, default) or only compile them for inspection (FALSE).

parallel	A logical passed to <code>ffmpeg_batch</code> : scrub in parallel with furrr (TRUE) or sequentially (FALSE, default). Parallelism follows the active <code>future</code> plan; TRUE under the default sequential plan runs one input at a time and warns. Set a plan first, e.g. <code>future::plan(future::multisession)</code> .
...	Additional arguments forwarded to <code>ffmpeg_batch</code> , such as <code>verify</code> , <code>manifest</code> , <code>checksums</code> , and <code>progress</code> .

Value

The `tibble` returned by `ffmpeg_batch`: jobs with an added `command` column. When output was derived, it also has the resolved output column. When `run = TRUE`, it has a `success` column, plus any columns the forwarded arguments add, such as `verified`.

See Also

`strip_metadata()` for the single-input form; `ffmpeg_batch()` for the batch runner and the arguments forwarded through ...; `standardize_video_batch()` and `anonymize_video_batch()` for the other batch task functions.

Other task functions: `anonymize_video()`, `anonymize_video_batch()`, `compare_videos()`, `compare_videos_batch()`, `concatenate_videos()`, `concatenate_videos_batch()`, `convert_audio()`, `convert_audio_batch()`, `crop_video()`, `crop_video_batch()`, `extract_audio()`, `extract_audio_batch()`, `extract_frame()`, `extract_frame_batch()`, `format_for_web()`, `format_for_web_batch()`, `normalize_audio()`, `normalize_audio_batch()`, `picture_in_picture()`, `picture_in_picture_batch()`, `sample_frames()`, `sample_frames_batch()`, `segment_video()`, `segment_video_batch()`, `separate_audio_video()`, `separate_audio_video_batch()`, `standardize_video()`, `standardize_video_batch()`, `strip_metadata()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
jobs <- tibble::tibble(input = video, output = "clean.mp4")
# run = FALSE compiles one command per input without calling FFmpeg
strip_metadata_batch(jobs, run = FALSE)
```

unset_program

Forget the location of a dependency program

Description

`unset_program()` forgets the location that `set_program()` saved for a program. After that, `find_ffmpeg()` and the other `find_*`() functions look for the program on the PATH only.

Usage

```
unset_program(program)
```

Arguments

program A string naming the program to forget. One of "ffmpeg", "ffprobe", "ffplay" or "mediainfo". There is no default, because the call deletes a file. A call that names no program gives an error.

Details

The call deletes the file that holds the location. It does not ask you to confirm first. It does not remove the program, and it does not change the PATH. A program on the PATH is still found afterwards.

If no location is saved for the program, the call gives a warning and returns FALSE. It does not give an error, because the program is already forgotten.

The call also clears a location saved by a version before 0.2.0. See the section "Locations saved by earlier versions" in [find_ffmpeg\(\)](#).

Value

Invisibly, TRUE when the call removed a saved location, and FALSE when there was none to remove.

See Also

[set_program\(\)](#) to save a location, and [program_status\(\)](#) to see where the package finds each program.

Other program management functions: [find_ffmpeg\(\)](#), [install_on_win\(\)](#), [program_status\(\)](#), [set_program\(\)](#)

Examples

```
## Not run:
# Forget a location set_program() remembered, so that find_mediainfo() goes
# back to answering from the PATH
unset_program("mediainfo")

## End(Not run)
```

 verify_media

Verify a Media File Against Expected Properties

Description

Probe a media file and check its structural metadata against a set of expectations, returning a tidy pass/fail tibble with one row per checked property. A reproducible command tells you what ran. This function tells you whether the *result* is what you asked for. Use it after a conversion to confirm that the output has the duration, dimensions, codecs and other properties the pipeline was meant to produce.

Usage

```

verify_media(
  file,
  duration = NULL,
  width = NULL,
  height = NULL,
  video_codec = NULL,
  audio_codec = NULL,
  sample_rate = NULL,
  ...,
  tolerance = 0.1
)

```

Arguments

file	A string naming a single media file to verify.
duration	Expected container duration in seconds (numeric).
width, height	Expected video-frame dimensions in pixels (numeric).
video_codec, audio_codec	Expected codec names for the first video and audio stream (strings, e.g. "h264", "aac").
sample_rate	Expected audio sample rate in Hz (numeric).
...	Further expectations given as name = value, each checked against the FFprobe field of that name (container first, then video, then audio).
tolerance	The absolute tolerance for numeric checks (default 0.1).

Details

The glossary in `vignette("tidymedia")` explains media terms such as codec, stream and keyframe.

Checks are structural (drawn from FFprobe metadata), not perceptual: this does not measure visual or audio quality. The named arguments cover the most common properties; pass any other FFprobe field by name through `...` (for example `pix_fmt = "yuv420p"` or `bit_rate = 141800`). Extra names are resolved against the probe columns in the order container, then video stream, then audio stream, and the first match wins.

Numeric checks pass when $\text{abs}(\text{actual} - \text{expected}) \leq \text{tolerance}$. With the default tolerance of 0.1, whole-number properties (width, height, sample rate) must match exactly. The duration check allows a small difference, for example for cuts that snap to a keyframe. String checks (the codecs) must match exactly. A property whose stream or column is absent yields an NA actual value and a failing check.

Value

A [tibble](#) with one row per checked property and columns file, check, expected, actual, and pass (logical).

See Also

`ffmpeg_run()` and `ffmpeg_batch()`, which accept a `verify = spec`.

Other verification functions: `ffmpeg_manifest()`

Examples

```
video <- system.file("extdata", "sample.mp4", package = "tidymedia")
verify_media(video, width = 320, height = 240, video_codec = "h264")
```

with_timeout	<i>Set a time limit for one call</i>
--------------	--------------------------------------

Description

`with_timeout()` runs `expr` with a time limit of its own. The limit applies to each FFmpeg, FFprobe or MediaInfo program that `expr` starts. When `with_timeout()` returns, or stops with an error, the limit that was in force before the call is back.

The session limit, `options(tidymedia.timeout = )`, applies to every call in the session. `with_timeout()` applies to one call. For example, you can give one test conversion five minutes in a session with a one-hour limit.

Usage

```
with_timeout(expr, seconds)
```

Arguments

<code>expr</code>	An expression. It is run once, where you wrote it, and its value is returned.
<code>seconds</code>	A whole number of seconds. <code>0</code> means no limit, so <code>with_timeout(expr, 0)</code> removes a session limit for one call. A fraction, a negative number, a string or <code>NULL</code> gives an error before <code>expr</code> runs.

Details

The limit applies to each program, not to the whole call. In a 100-row batch inside `with_timeout(expr, 600)`, each program that a row starts gets 600 seconds, plus the delay in "How long the wait can be". The workers of a `parallel = TRUE` run use the same limit.

The limit is a whole number of seconds. The package does not round a fraction, because R would read a limit below one second as no limit.

A limit set with `options(tidymedia.timeout = )` follows the same rule, with one difference. `options(tidymedia.timeout = NULL)` removes the option, so it means no limit. A function that can start a program gives an error for a wrong value, even when `run = FALSE`. `ffmpeg_batch()` gives that error before it starts any job. A function that starts no program gives no such error. For example, `has_hardware_encoder()` starts none when you set `tidymedia.hardware_encoders`. A

probe_*() function that you give a probe object also starts none. `has_encoder()` is the exception: it checks the limit first, even on a machine where it finds no FFmpeg to start.

Most functions check their own arguments before the limit. So a wrong argument gives its own error, even when the limit is also wrong. A few arguments of the `_batch` functions are checked inside each job, after the limit. An example is the `pixel_format` of `anonymize_video_batch()`. When the limit is also wrong, the error is about the limit.

Value

The value of `expr`.

How long the wait can be

The limit sets how long R waits for a program, and the wait can be longer. When the limit is reached, R asks the program to stop. R asks again 20 seconds later, and kills the program 20 seconds after that. So R can wait up to 40 seconds past the limit. For example, five hung files under a 1-second limit can take about three and a half minutes.

R does not guarantee that the program stops. A program can survive the attempts to stop it. How fast a program stops also depends on its version.

What happens when the limit is reached

A reached limit is never silent. The call gives an error or a warning.

These functions give an error with the class `tidymedia_timeout`, which names the program and the limit:

- the task functions whose names do not end in `_batch`, except `segment_video()`
- `ffmpeg_run()`, `ffmpeg()`, `ffprobe()` and `mediainfo()`
- `ffmpeg_codecs()` and `ffmpeg_encoders()`
- `has_encoder()` and `has_hardware_encoder()` when they ask FFmpeg
- `verify_media()`, because a check with no answer is not a "no"

These functions give a warning instead, so that one hung file does not lose the rest of the work:

- `probe_all()`, the other `probe_*`() functions, `mediainfo_parameter()`, `mediainfo_query()`, `mediainfo_template()` and the `get_*`() functions give NA for that file. One warning at the end says how many files timed out.
- `ffmpeg_batch()`, `segment_video()` and the `_batch` task functions set `success = FALSE` for that job. One warning at the end says how many jobs timed out. It has the class `tidymedia_batch_timeout`. Two steps of these calls give an error instead. One is the analysis pass of `normalize_audio_batch(two_pass = TRUE)`. The other is the check that FFmpeg has the hardware encoder that hardware names, such as "nvcnc". That check asks FFmpeg only when `tidymedia.hardware_encoders` is not set and the session has no stored answer. The glossary in `vignette("tidymedia")` explains hardware encoders.
- The dropped-track check of `extract_audio()`, `convert_audio()`, `normalize_audio()` and their `_batch` forms warns that it could not check. The track count that `separate_audio_video()` reads after a failed run warns the same way. So does the check of FFmpeg's encoder lists that

`ffmpeg_run()`, the task functions and `ffmpeg_batch()` make after a failed run, once per call (see "When the build lacks an encoder" on `ffmpeg_run()`). A batch manifest, see `ffmpeg_manifest()`, and `program_status()` warn when they cannot read a program version. These warnings have the class `tidymedia_probe_timeout`, and the call goes on as it would for an unreadable input.

`suppressWarnings(classes = "tidymedia_dropped_audio")` hides the dropped-track warning, but not the warning that the check timed out. To hide both, add `"tidymedia_probe_timeout"` to `classes`.

The task functions and `ffmpeg_run()` delete a part-written output file after a timeout, as they do after any failed run. `ffmpegpeg()` cannot tell which of your arguments is the output, so it leaves that file. Check the output of a timed-out `ffmpegpeg()` call yourself.

See Also

`local_timeout()` to set a limit for the rest of a function. [tidymedia-package](#) describes the session options.

Examples

```
# Inside the call, the limit is the one you gave.
with_timeout(getOption("tidymedia.timeout"), 30)

# Outside it, the session's own setting is untouched.
getOption("tidymedia.timeout", default = "unset")

# Bound one conversion at five minutes, whatever the session is set to.
# Needs FFmpeg, and writes to a temporary file.
if (nzchar(Sys.which("ffmpeg"))) {
  video <- system.file("extdata", "sample.mp4", package = "tidymedia")
  with_timeout(extract_audio(video, tempfile(fileext = ".wav")), 300)
}
```

Index

* audio selection functions

- [anonymize_video](#), [5](#)
- [anonymize_video_batch](#), [8](#)
- [audio_stream](#), [11](#)
- [compare_videos](#), [13](#)
- [compare_videos_batch](#), [16](#)
- [convert_audio](#), [20](#)
- [convert_audio_batch](#), [22](#)
- [crop_video](#), [24](#)
- [crop_video_batch](#), [27](#)
- [extract_audio](#), [30](#)
- [extract_audio_batch](#), [31](#)
- [format_for_web](#), [69](#)
- [format_for_web_batch](#), [71](#)
- [normalize_audio](#), [89](#)
- [normalize_audio_batch](#), [92](#)
- [picture_in_picture](#), [96](#)
- [picture_in_picture_batch](#), [99](#)
- [segment_video](#), [111](#)
- [segment_video_batch](#), [114](#)
- [separate_audio_video](#), [117](#)
- [separate_audio_video_batch](#), [121](#)
- [standardize_video](#), [127](#)
- [standardize_video_batch](#), [130](#)

* capability functions

- [ffmpeg_codecs](#), [37](#)
- [ffmpeg_encoders](#), [38](#)
- [hardware_encoder](#), [78](#)
- [has_encoder](#), [79](#)
- [refresh_ffmpeg_capabilities](#), [106](#)

* direct command functions

- [ffmpeg](#), [36](#)
- [ffprobe](#), [66](#)
- [mediainfo](#), [84](#)

* metadata functions

- [get_duration](#), [73](#)
- [get_frame_rate](#), [74](#)
- [get_height](#), [75](#)
- [get_sample_rate](#), [76](#)

- [get_width](#), [77](#)
- [mediainfo_parameter](#), [85](#)
- [mediainfo_query](#), [86](#)
- [mediainfo_template](#), [87](#)
- [probe_all](#), [102](#)
- [probe_container](#), [103](#)

* pipeline functions

- [ffmpeg_batch](#), [39](#)
- [ffmpeg_codec](#), [41](#)
- [ffmpeg_compile](#), [42](#)
- [ffmpeg_concat](#), [43](#)
- [ffmpeg_copy](#), [44](#)
- [ffmpeg_crop](#), [45](#)
- [ffmpeg_drawbox](#), [46](#)
- [ffmpeg_drop](#), [47](#)
- [ffmpeg_files](#), [48](#)
- [ffmpeg_fps](#), [49](#)
- [ffmpeg_hstack](#), [50](#)
- [ffmpeg_jobs](#), [51](#)
- [ffmpeg_loudnorm](#), [52](#)
- [ffmpeg_map](#), [55](#)
- [ffmpeg_output_options](#), [56](#)
- [ffmpeg_overlay](#), [57](#)
- [ffmpeg_pixel_format](#), [59](#)
- [ffmpeg_run](#), [60](#)
- [ffmpeg_scale](#), [62](#)
- [ffmpeg_seek](#), [63](#)
- [ffmpeg_trim](#), [64](#)
- [ffmpeg_vstack](#), [65](#)
- [print.tidymedia_ffmpeg](#), [101](#)

* program management functions

- [find_ffmpeg](#), [67](#)
- [install_on_win](#), [80](#)
- [program_status](#), [105](#)
- [set_program](#), [125](#)
- [unset_program](#), [135](#)

* task functions

- [anonymize_video](#), [5](#)
- [anonymize_video_batch](#), [8](#)

- compare_videos, [13](#)
- compare_videos_batch, [16](#)
- concatenate_videos, [18](#)
- concatenate_videos_batch, [19](#)
- convert_audio, [20](#)
- convert_audio_batch, [22](#)
- crop_video, [24](#)
- crop_video_batch, [27](#)
- extract_audio, [30](#)
- extract_audio_batch, [31](#)
- extract_frame, [34](#)
- extract_frame_batch, [35](#)
- format_for_web, [69](#)
- format_for_web_batch, [71](#)
- normalize_audio, [89](#)
- normalize_audio_batch, [92](#)
- picture_in_picture, [96](#)
- picture_in_picture_batch, [99](#)
- sample_frames, [108](#)
- sample_frames_batch, [109](#)
- segment_video, [111](#)
- segment_video_batch, [114](#)
- separate_audio_video, [117](#)
- separate_audio_video_batch, [121](#)
- standardize_video, [127](#)
- standardize_video_batch, [130](#)
- strip_metadata, [133](#)
- strip_metadata_batch, [134](#)
- * **verification functions**
 - ffm_manifest, [54](#)
 - verify_media, [136](#)
- anonymize_video, [5](#), [7](#), [9](#), [10](#), [12](#), [13](#), [21](#), [23](#), [26](#), [29](#), [30](#), [32](#), [70](#), [72](#), [78](#), [79](#), [90](#), [94](#), [113](#), [116](#), [119](#), [123](#), [128](#), [132](#)
- anonymize_video(), [8](#), [10](#), [11](#), [13](#), [15](#), [18–20](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [47](#), [70](#), [73](#), [91](#), [92](#), [95](#), [96](#), [98](#), [101](#), [109](#), [110](#), [113](#), [114](#), [116](#), [117](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#)
- anonymize_video_batch, [8](#)
- anonymize_video_batch(), [7](#), [8](#), [13](#), [15](#), [18–20](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [70](#), [73](#), [91](#), [92](#), [95](#), [96](#), [98](#), [101](#), [109](#), [110](#), [113](#), [114](#), [116](#), [117](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#), [139](#)
- audio_tracks(audio_stream), [11](#)
- audio_indices(audio_stream), [11](#)
- audio_stream, [7](#), [8](#), [10](#), [11](#), [11](#), [14](#), [15](#), [17](#), [18](#), [21–24](#), [26](#), [29–33](#), [41](#), [44](#), [70](#), [72](#), [73](#), [90](#), [92](#), [94](#), [96–98](#), [100](#), [101](#), [113](#), [114](#), [116](#), [117](#), [119](#), [121](#), [123](#), [125](#), [128](#), [129](#), [132](#)
- compare_videos, [7](#), [10](#), [13](#), [21](#), [23](#), [26](#), [29](#), [30](#), [32](#), [70](#), [72](#), [78](#), [79](#), [90](#), [94](#), [113](#), [116](#), [119](#), [123](#), [128](#), [132](#)
- compare_videos(), [7](#), [8](#), [11–13](#), [16–20](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [50](#), [66](#), [70](#), [73](#), [91](#), [92](#), [95](#), [96](#), [98](#), [101](#), [109](#), [110](#), [113](#), [114](#), [116](#), [117](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#)
- compare_videos_batch, [16](#)
- compare_videos_batch(), [7](#), [8](#), [11](#), [13](#), [15](#), [19](#), [20](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [70](#), [73](#), [91](#), [92](#), [95](#), [96](#), [98](#), [101](#), [109](#), [110](#), [113](#), [114](#), [116](#), [117](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#)
- concatenate_videos, [18](#)
- concatenate_videos(), [7](#), [11](#), [15](#), [18–20](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [43](#), [70](#), [73](#), [91](#), [95](#), [98](#), [101](#), [109](#), [110](#), [113](#), [116](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#)
- concatenate_videos_batch, [19](#)
- concatenate_videos_batch(), [7](#), [11](#), [15](#), [18](#), [19](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [70](#), [73](#), [91](#), [95](#), [98](#), [101](#), [109](#), [110](#), [113](#), [116](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#)
- convert_audio, [7](#), [10](#), [12](#), [13](#), [20](#), [21](#), [23](#), [26](#), [29](#), [30](#), [32](#), [70](#), [72](#), [89–91](#), [94](#), [113](#), [116](#), [119](#), [123](#), [128](#), [132](#)
- convert_audio(), [4](#), [7](#), [8](#), [11](#), [13](#), [15](#), [18–20](#), [22](#), [24](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [70](#), [73](#), [91](#), [92](#), [95](#), [96](#), [98](#), [101](#), [109](#), [110](#), [113](#), [114](#), [116](#), [117](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#), [139](#)
- convert_audio_batch, [22](#)
- convert_audio_batch(), [7](#), [8](#), [11](#), [13](#), [15](#), [18–20](#), [22](#), [26](#), [29](#), [31](#), [33](#), [34](#), [36](#), [70](#), [73](#), [91](#), [92](#), [95](#), [96](#), [98](#), [101](#), [109](#), [110](#), [113](#), [114](#), [116](#), [117](#), [121](#), [125](#), [129](#), [132](#), [134](#), [135](#)
- crop_video, [7](#), [10](#), [12](#), [13](#), [21](#), [23](#), [24](#), [26](#), [29](#), [30](#), [32](#), [70](#), [72](#), [78](#), [79](#), [90](#), [94](#), [113](#), [116](#), [119](#), [123](#), [128](#), [132](#)
- crop_video(), [7](#), [8](#), [11](#), [13](#), [15](#), [18–20](#), [22](#), [24](#), [27–29](#), [31](#), [33](#), [34](#), [36](#), [45](#), [70](#), [73](#), [91](#),

- 92, 95, 96, 98, 101, 109, 110, 114,
117, 121, 125, 129, 132, 134, 135
- crop_video_batch, 27, 71
- crop_video_batch(), 7, 8, 11, 13, 15, 18–20,
22, 24, 26, 31, 33, 34, 36, 51, 70, 73,
91, 92, 95, 96, 98, 101, 109, 110,
114, 117, 121, 125, 129, 132, 134,
135
- extract_audio, 7, 10, 12, 13, 21, 23, 26, 29,
30, 30, 32, 70, 72, 89–91, 94, 113,
116, 119, 123, 128, 132
- extract_audio(), 4, 7, 8, 11–13, 15, 18–20,
22, 24, 26, 27, 29, 31, 33, 34, 36, 48,
70, 73, 91, 92, 95, 96, 98, 101, 109,
110, 114, 117, 121, 125, 129, 132,
134, 135, 139
- extract_audio_batch, 31
- extract_audio_batch(), 7, 8, 11, 13, 15,
18–20, 22, 24, 26, 27, 29, 31, 34, 36,
51, 70, 73, 91, 92, 95, 96, 98, 101,
109, 110, 114, 117, 121, 125, 129,
132, 134, 135
- extract_frame, 34, 35, 108
- extract_frame(), 7, 11, 15, 18–20, 22, 24,
26, 29, 31, 33, 35, 36, 70, 73, 91, 95,
98, 101, 109, 110, 114, 117, 121,
125, 129, 132, 134, 135
- extract_frame_batch, 35, 108
- extract_frame_batch(), 7, 11, 15, 18–20,
22, 24, 26, 29, 31, 33, 34, 70, 73, 91,
95, 98, 101, 109, 110, 114, 117, 121,
125, 129, 132, 134, 135
- ffmpeg_batch, 8, 10, 16–20, 22–24, 27, 29,
31–33, 35, 39, 61, 71, 72, 92, 94, 95,
99, 101, 109, 110, 113, 114, 116,
121, 123, 124, 130, 132, 134, 135
- ffmpeg_batch(), 5, 10, 18, 20, 24, 29, 33, 36,
42–45, 47–52, 54–59, 61, 62, 64–66,
72, 95, 101, 102, 110, 113, 116, 125,
132, 135, 138–140
- ffmpeg_codec, 41, 61
- ffmpeg_codec(), 13, 22, 31, 37, 38, 41–45,
47–50, 52, 54, 56–59, 61, 62, 64–66,
70, 102, 121, 129
- ffmpeg_compile, 42, 101
- ffmpeg_compile(), 41–45, 47–50, 52, 54, 56–59,
61, 62, 64–66, 102
- ffmpeg_concat, 43
- ffmpeg_concat(), 19, 41, 42, 44, 45, 47–50, 52,
54, 56–59, 61, 62, 64–66, 102
- ffmpeg_copy, 43, 44, 56, 63
- ffmpeg_copy(), 13, 41–43, 45, 47–50, 52, 54,
56–59, 61, 62, 64–66, 102, 134
- ffmpeg_crop, 45
- ffmpeg_crop(), 26, 41–44, 47–50, 52, 54, 56–59,
61, 62, 64–66, 102
- ffmpeg_drawbox, 46
- ffmpeg_drawbox(), 7, 41–45, 48–50, 52, 54,
56–59, 61, 62, 64–66, 102
- ffmpeg_drop, 47
- ffmpeg_drop(), 31, 41–45, 47–50, 52, 54, 56–59,
61, 62, 64–66, 102
- ffmpeg_files, 40, 48, 102
- ffmpeg_files(), 4, 41–45, 47–50, 52, 54, 56–59,
61, 62, 64–66, 102
- ffmpeg_fps, 49
- ffmpeg_fps(), 41–45, 47, 48, 50, 52, 54, 56–59,
61, 62, 64–66, 102, 109
- ffmpeg_hstack, 13, 43, 50, 56, 57, 66
- ffmpeg_hstack(), 15, 41–45, 47–49, 52, 54,
56–59, 61, 62, 64–66, 102
- ffmpeg_jobs, 51
- ffmpeg_jobs(), 41–45, 47–50, 54, 56–59, 61, 62,
64–66, 102
- ffmpeg_loudnorm, 52
- ffmpeg_loudnorm(), 41–45, 47–50, 52, 56–59,
61, 62, 64–66, 91, 102
- ffmpeg_manifest, 40, 54, 95
- ffmpeg_manifest(), 41, 138, 140
- ffmpeg_map, 55
- ffmpeg_map(), 22, 41–45, 47–50, 52, 54, 57–59,
61, 62, 64–66, 102, 121
- ffmpeg_output_options, 56, 61
- ffmpeg_output_options(), 41–45, 47–50, 52,
54, 56, 58, 59, 61, 62, 64–66, 102,
134
- ffmpeg_overlay, 57, 96
- ffmpeg_overlay(), 41–45, 47–50, 52, 54, 56, 57,
59, 61, 62, 64–66, 98, 102
- ffmpeg_pixel_format, 59
- ffmpeg_pixel_format(), 41–45, 47–50, 52, 54,
56–58, 61, 62, 64–66, 70, 102, 129
- ffmpeg_run, 39, 40, 60, 106
- ffmpeg_run(), 4, 5, 36, 41–45, 47–50, 52, 54,
56–59, 62, 64–66, 102, 138–140

- ffmpeg_scale, 62
- ffmpeg_scale(), 41–45, 47–50, 52, 54, 56–59, 61, 64–66, 102, 129
- ffmpeg_seek, 63, 112, 115
- ffmpeg_seek(), 34, 41–45, 47–50, 52, 54, 56–59, 61, 62, 65, 66, 102, 113, 116
- ffmpeg_trim, 63, 64
- ffmpeg_trim(), 41–45, 47–50, 52, 54, 56–59, 61, 62, 64, 66, 102
- ffmpeg_vstack, 13, 65
- ffmpeg_vstack(), 15, 41–45, 47–50, 52, 54, 56–59, 61, 62, 64, 65, 102
- ffmpegpeg, 36, 43, 78, 89, 133
- ffmpegpeg(), 4, 37, 38, 57, 67, 85, 139, 140
- ffmpegpeg_codecs, 37, 61, 107
- ffmpegpeg_codecs(), 36, 38, 42, 79, 80, 107, 139
- ffmpegpeg_encoders, 38, 61, 78–80, 107
- ffmpegpeg_encoders(), 36, 37, 79, 80, 107, 139
- ffprobe, 66
- ffprobe(), 4, 36, 85, 139
- find_ffmpeg, 67
- find_ffmpeg(), 67, 82, 105, 106, 125, 126, 135, 136
- find_ffplay (find_ffmpeg), 67
- find_ffplay(), 67
- find_ffprobe (find_ffmpeg), 67
- find_ffprobe(), 67
- find_mediainfo (find_ffmpeg), 67
- find_mediainfo(), 67
- format_for_web, 7, 10, 12, 13, 21, 23, 26, 29, 30, 32, 69, 70, 72, 78, 79, 90, 94, 113, 116, 119, 123, 127, 128, 132
- format_for_web(), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 45, 59, 71–73, 91, 92, 95, 96, 98, 101, 109, 110, 114, 117, 121, 125, 129, 132, 134, 135
- format_for_web_batch, 71
- format_for_web_batch(), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 70, 91, 92, 95, 96, 98, 101, 109, 110, 114, 117, 121, 125, 129, 132, 134, 135
- future, 10, 35, 39, 94, 110, 113, 116, 132, 135
- get_duration, 73
- get_duration(), 4, 74–77, 85–88, 103, 104
- get_frame_rate, 74
- get_frame_rate(), 74–77, 86–88, 103, 104
- get_height, 75
- get_height(), 74, 76, 77, 86–88, 103, 104
- get_sample_rate, 76
- get_sample_rate(), 74, 75, 77, 86–88, 103, 104
- get_width, 77
- get_width(), 74–76, 86–88, 103, 104
- hardware_encoder, 61, 78, 107
- hardware_encoder(), 37, 38, 80, 107
- has_encoder, 79, 106
- has_encoder(), 37, 38, 79, 107, 139
- has_hardware_encoder, 6, 9, 14, 25, 69, 72, 79, 80, 97, 107, 112, 118, 128, 131
- has_hardware_encoder (hardware_encoder), 78
- has_hardware_encoder(), 7, 10, 15, 18, 26, 29, 70, 98, 101, 113, 116, 121, 125, 129, 138, 139
- install_on_win, 80
- install_on_win(), 68, 106, 126, 136
- list.files, 52
- local_timeout, 83
- local_timeout(), 140
- mediainfo, 84
- mediainfo(), 4, 36, 67, 139
- mediainfo_parameter, 85
- mediainfo_parameter(), 74–77, 85, 87, 88, 103, 104, 139
- mediainfo_query, 86
- mediainfo_query(), 73–77, 85, 86, 88, 103, 104, 134, 139
- mediainfo_template, 87
- mediainfo_template(), 73–77, 85–87, 103, 104, 139
- normalize_audio, 7, 10, 12, 13, 21, 23, 26, 29, 30, 32, 70, 72, 89, 90, 93, 94, 113, 116, 119, 123, 128, 129, 132
- normalize_audio(), 4, 7, 8, 11–13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 54, 70, 73, 92, 95, 96, 98, 101, 109, 110, 114, 117, 121, 125, 129, 132, 134, 135, 139
- normalize_audio_batch, 90, 92
- normalize_audio_batch(), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34,

- [36, 70, 73, 91, 92, 98, 101, 109, 111, 114, 117, 121, 125, 129, 132, 134, 135](#)
- [picture_in_picture, 7, 10, 21, 23, 26, 29, 30, 32, 58, 70, 72, 78, 79, 90, 94, 96, 113, 116, 119, 123, 128, 132](#)
- [picture_in_picture\(\), 7, 8, 11–13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 58, 70, 73, 91, 92, 95, 96, 99–101, 109, 111, 114, 117, 121, 125, 129, 132, 134, 135](#)
- [picture_in_picture_batch, 99](#)
- [picture_in_picture_batch\(\), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 70, 73, 91, 92, 95, 96, 98, 109, 111, 114, 117, 121, 125, 129, 132, 134, 135](#)
- [pmap, 39](#)
- [print.tidymedia_ffm, 101](#)
- [print.tidymedia_ffm\(\), 41–45, 47–50, 52, 54, 56–59, 61, 62, 64–66](#)
- [probe_all, 102](#)
- [probe_all\(\), 4, 67, 74–77, 86–88, 103, 104, 139](#)
- [probe_audio, 119, 123](#)
- [probe_audio \(probe_container\), 103](#)
- [probe_audio\(\), 12, 13, 121](#)
- [probe_container, 103](#)
- [probe_container\(\), 74–77, 86–88, 103, 134](#)
- [probe_streams \(probe_container\), 103](#)
- [probe_video \(probe_container\), 103](#)
- [program_status, 105](#)
- [program_status\(\), 68, 82, 126, 136, 140](#)
- [purrr::pmap\(\), 40](#)
-
- [refresh_ffmpeg_capabilities, 6, 9, 14, 17, 25, 28, 61, 69, 72, 79, 80, 97, 100, 106, 112, 115, 118, 123, 128, 131](#)
- [refresh_ffmpeg_capabilities\(\), 37, 38, 79, 80](#)
-
- [sample_frames, 108](#)
- [sample_frames\(\), 7, 11, 15, 18–20, 22, 24, 26, 29, 31, 33, 34, 36, 70, 73, 91, 95, 98, 101, 109–111, 114, 117, 121, 125, 129, 132, 134, 135](#)
- [sample_frames_batch, 109](#)
- [sample_frames_batch\(\), 7, 11, 15, 18–20, 22, 24, 26, 29, 31, 33, 34, 36, 70, 73, 91, 95, 98, 101, 109, 114, 117, 121, 125, 129, 132, 134, 135](#)
- [segment_video, 7, 10, 12, 13, 21, 23, 26, 29, 30, 32, 70, 72, 78, 79, 90, 94, 111, 113, 115, 116, 119, 123, 128, 132](#)
- [segment_video\(\), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 41, 44, 48, 64, 70, 73, 92, 96, 98, 101, 109, 111, 114–117, 121, 125, 129, 132, 134, 135, 139](#)
- [segment_video_batch, 114](#)
- [segment_video_batch\(\), 7, 8, 10, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 70, 73, 92, 96, 98, 101, 109, 111, 113, 114, 121, 125, 129, 132, 134, 135](#)
- [separate_audio_video, 7, 10, 12, 13, 21, 23, 26, 29, 30, 32, 60, 70, 72, 78, 79, 90, 94, 113, 116, 117, 119, 122–124, 128, 132](#)
- [separate_audio_video\(\), 4, 7, 8, 11–13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 52, 56, 70, 73, 92, 96, 98, 101, 109, 111, 114, 117, 121–125, 129, 132, 134, 135, 139](#)
- [separate_audio_video_batch, 40, 121](#)
- [separate_audio_video_batch\(\), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 70, 73, 92, 96, 98, 101, 109, 111, 114, 117, 121, 129, 132, 134, 135](#)
- [set_ffmpeg, 80, 107](#)
- [set_ffmpeg \(set_program\), 125](#)
- [set_ffplay \(set_program\), 125](#)
- [set_ffprobe \(set_program\), 125](#)
- [set_mediainfo \(set_program\), 125](#)
- [set_program, 107, 125](#)
- [set_program\(\), 67, 68, 80, 82, 105, 106, 135, 136](#)
- [standardize_video, 7, 10, 12, 13, 21, 23, 26, 29, 30, 32, 61, 70, 72, 78, 79, 90, 94, 113, 116, 119, 123, 127, 128, 131, 132](#)
- [standardize_video\(\), 7, 8, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 42, 48, 49, 59, 62, 70, 73, 92, 96,](#)

98, 101, 109, 111, 114, 117, 121, 125, 130–132, 134, 135

`standardize_video_batch`, *71, 130*

`standardize_video_batch()`, *7, 8, 10, 11, 13, 15, 18–20, 22, 24, 26, 27, 29, 31, 33, 34, 36, 70, 72, 73, 92, 95, 96, 98, 101, 109, 111, 114, 117, 121, 125, 129, 134, 135*

`strip_metadata`, *133*

`strip_metadata()`, *7, 11, 15, 18–20, 22, 24, 26, 29, 31, 33, 34, 36, 70, 73, 92, 96, 98, 101, 109, 111, 114, 117, 121, 125, 129, 132, 134, 135*

`strip_metadata_batch`, *134*

`strip_metadata_batch()`, *7, 11, 15, 18–20, 22, 24, 26, 29, 31, 33, 34, 36, 70, 73, 92, 96, 98, 101, 109, 111, 114, 117, 121, 125, 129, 132, 134*

`tibble`, *10, 35, 37, 38, 40, 52, 55, 95, 110, 113, 116, 123, 132, 135, 137*

`tidymedia` (`tidymedia-package`), *4*

`tidymedia-package`, *4, 84, 140*

`unset_program`, *107, 135*

`unset_program()`, *68, 82, 105, 106, 126*

`verify_media`, *39, 60, 136*

`verify_media()`, *41, 55, 61, 139*

`with_timeout`, *119, 138*

`with_timeout()`, *4, 40, 83, 84, 86–88, 103*

`withr::defer()`, *83*

`withr::local_options()`, *83, 84*

`withr::with_options()`, *84*