The latest version of the rOpenSci [av package](https://docs.ropensci.org/av) includes some useful new tools for working with audio data. We have added functions for reading, cutting, converting, transforming, and plotting audio data in any popular audio / video format (mp3, mkv, aac, etc).

The functionality can either be used by itself, or to prepare audio data for further analysis in R using other packages. We hope this clears an important hurdle to use R for research on speech, music, and [whale mating calls](https://youtu.be/F8Zt3mYlOqU).

**Audio analysis in R**

CRAN has serveral cool packages for audio analysis, most notably the [tuneR](https://cran.r-project.org/package=tuneR) package, maintained by Uwe Ligges. However, getting your audio data into R can be tricky, because real-world audio come in all sorts of formats and codecs, with varying sampling rates, channel layouts, and so on.

I got stuck on this myself when trying to use sound from youtube videos, or extract a short fragment from a long recording. The existing packages mostly assume small wav files, but it is unclear where to get started to analyze the complete oeuvre of Taylor Swift or a 3 hour recording of bird sounds.

The av package builds on the [FFmpeg](http://ffmpeg.org/documentation.html) libav libraries, which provide a extensive, high performance implementations for streamable reading, editing, and writing of media in any format. The av package is already used in R for working with video, for example to create [animated graphics](https://ropensci.org/technotes/2018/10/06/av-release/) or [sampling pictures from a camera stream](https://ropensci.org/technotes/2019/09/27/ropensci-docs/) for image analysis. By adding audio functionality, we hope to make it more feasible to analyze real-world sound data directly in R.

**Converting between audio formats**

The simplest use of av is converting an audio file into another format. The [av\_audio\_convert](https://docs.ropensci.org/av/reference/encoding.html) function will convert any sound input (even from a video file) into another output format, optionally adjusting the number of channels, sampling rate, and start/end time.

Let’s run some examples. First make sure you install the latest version of the package from CRAN:

install.packages("av")

library(av)

Suppose we want to read a certain piece of audio from a youtube recording. We use the youtube-dl utility to download a full video from youtube:

# You need youtube-dl: <https://ytdl-org.github.io/youtube-dl/download.html>

# MacOS: brew install youtube-dl

system("youtube-dl <https://youtu.be/F8Zt3mYlOqU> -o whale.mp4")

# Show some info

av\_media\_info('whale.mp4')

#> $duration

#> [1] 138.507

#>

#> $video

#> width height codec frames framerate format

#> 1 1280 720 h264 3460 25 yuv420p

#>

#> $audio

#> channels sample\_rate codec frames bitrate layout

#> 1 2 44100 aac 5965 125588 stereo

This gives us a 75mb HD video with an AAC audio stream, which is not something other packages can read. Suppose that for your analysis you only need the first 10 seconds of sound, in mono wav format. Use [av\_audio\_convert](https://docs.ropensci.org/av/reference/encoding.html) to convert the input video into a shorter wav file:

av\_audio\_convert('whale.mp4', 'whale10.wav', channels = 1, total\_time = 10)

#> Output #0, wav, to 'short.wav':

#> Metadata:

#> ISFT : Lavf58.29.100

#> Stream #0:0: Audio: pcm\_s16le ([1][0][0][0] / 0x0001), 48000 Hz, mono, s16, 768 kb/s

#> Adding audio frame 503 at timestamp 10.01sec - audio stream completed!

Now we have a nice small wav file that we can read with any of the standard R packages. Note that this entire conversion above took 0.02 seconds and uses very little memory because it streams directly from the input to the output file.

**Reading raw audio samples**

For this example we use an MP3 demo that is included with the package. The [av\_media\_info](https://docs.ropensci.org/av/reference/info.html) function shows some information about this file (without reading it in memory).

# Our example data

wonderland <- system.file('samples/Synapsis-Wonderland.mp3', package='av')

av\_media\_info(wonderland)

#> $duration

#> [1] 30.04082

#>

#> $video

#> NULL

#>

#> $audio

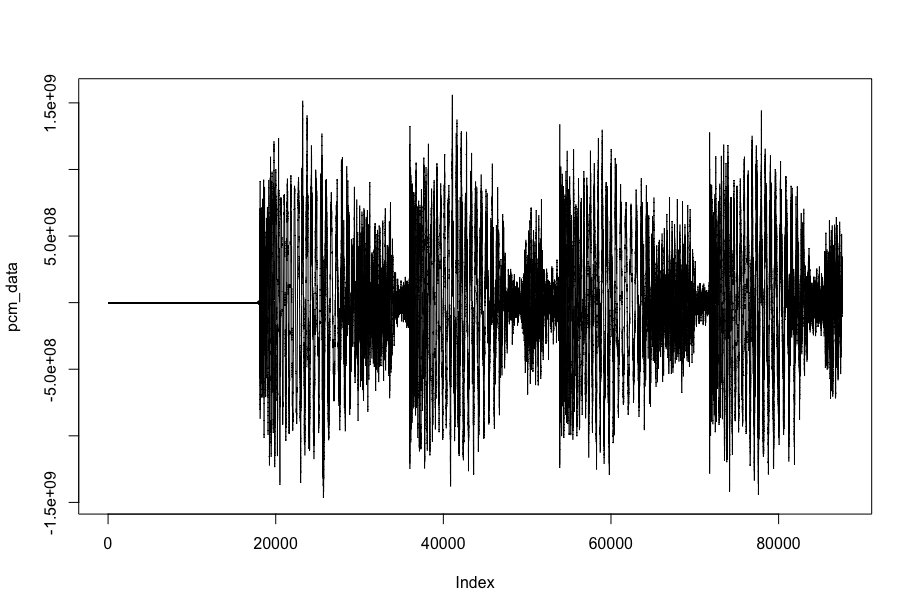
#> channels sample\_rate codec frames bitrate layout

#> 1 2 44100 mp3float NA 192000 stereo

As we can see, this file has one audio stream and no video streams, which is what we would expect. The function [read\_audio\_bin](https://docs.ropensci.org/av/reference/read_audio.html) reads audio files (in any format) into binary PCM samples. By setting end\_time = 2 we only read the first two seconds (approximately), keeping the data relatively small.

pcm\_data <- read\_audio\_bin(wonderland, channels = 1, end\_time = 2.0)

plot(pcm\_data, type = 'l')



The docs page for [read\_audio\_bin](https://docs.ropensci.org/av/reference/read_audio.html) shows parameters for reading audio, in particular to reduce the number of channels (i.e. convert to mono) or downsample to make data a bit more more managable.

**Frequency data and spectrograms**

Raw data is not very informative, most analyses require that the signal is transformed into frequency data. The [read\_audio\_fft](https://docs.ropensci.org/av/reference/read_audio.html) function converts audio directly into frequency data using FFmpeg’s built-in FFT. Because the FFT is done on-the-fly when reading the audio, it requires relatively little memory. It returns a matrix with the time-frequency data, from which we can plot the spectrogram.

# Read 5 sec of data and directly transform to frequency

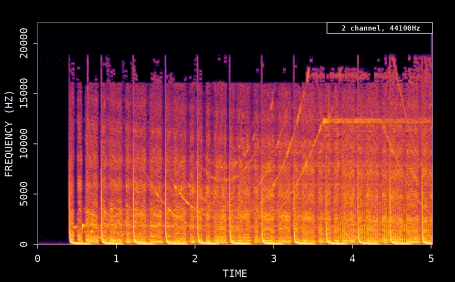
fft\_data <- read\_audio\_fft(wonderland, end\_time = 5.0)

dim(fft\_data)

#> [1] 512 860

# Plot the spectrogram

plot(fft\_data)



The docs page of the av package has an entire article dedicated to [spectograms](https://docs.ropensci.org/av/articles/articles/spectrograms.html), comparing the various R packages. This vignette also shows the [av\_spectrogram\_video](https://docs.ropensci.org/av/reference/capturing.html) function which generates a video that overlays the audio on an animated spectrogram with moving status bar:

# Create new audio file with first 5 sec

av\_audio\_convert(wonderland, 'short.mp3', total\_time = 5)

av\_spectrogram\_video('short.mp3', output = 'spectrogram.mp4', width = 1280, height = 720, res = 144)