FFMPEG

FROM ZERO TO HERO

NICK FERRANDO

FFMPEG
From Zero to Hero

By Nick Ferrando

Cover Illustration by Tarik Vision, Licensed by Getty Images.

www.ffmpegfromzerotohero.com

Index

Acknowledgments

Thank you for reading this book.
This is actually my very first technical book on the subject and i do hope you will find it useful for your audio and video content production needings.
It has been written with the goal to provide a quick and effective way to understand and use FFMPEG along with many other great open source technologies used to create, edit and process audio, video and pictures at scale.
This book will provide several practical formulas and their syntax explanation.
FFMPEG won't be the only piece of software discussed in this book: there are quite few tools that works in conjunctions with FFMPEG and some formulas and tools that you will discover, or you may re-discover, that will assist you for your professional needs.
By writing this book I wanted to share my 20+ years of experience with content production, and particularly my last years of experience with video production and automation.

A special acknowledgment and a special thank you goes to my dear friend *Andy Lombardi*:
Andy: you are my digital guru and a true friend whose knowledge and humanity are a continuos ispiration for me.

Of course this book won't exist without the genius mind of *Fabrice Bellard*, the creator of FFMPEG and all the FFMPEG active developers around the world.

A special dedication goes to *Leonardo Chiariglione*, *Hiroshi Yasuda*, *Federico Faggin*, *Dennis Ritchie*, *Ken Thompson*, Stephen Bourne, Steve Wozniak, Steve Jobs, *Richard Stallman* and *Linus Torvalds*.

With infinite admiration to the entire Ferrando family.

What is FFMPEG

Ffmpeg is a very fast video and audio converter that can also grab from a live audio/video source. It also reads from an arbitrary number of input "files" which can be your own computer files, pipes[1], network streams or URLs, grabbing devices, etc.[2]

If you ever wondered how the developers of **YouTube** or **Vimeo** cope with billions of video uploads or how **Netflix** processes its catalogue at scale or, again, if you want to discover how to create and develop your own video platform, you may want to know more about FFMPEG.

This acronym stands for “**F**ast-**F**orward-**M**oving-**P**icture-**E**xpert **G**roup”.

The Moving Picture Experts Group, **MPEG**, is a working group of authorities that was formed in 1988 by the standard organization **ISO,** The International Organization for Standardization, and **IEC**, the International

[1] pipe is a technique for passing information from one program process to another.

[2] https://ffmpeg.org/ffmpeg.html#toc-Description

Electrotechnical Commission, to set standards for audio and video compression and transmission.

Since its establishment by Leonardo Chiariglione and Hiroshi Yasuda, the **M**oving **P**ictures **E**xperts **G**roup has made an indelible mark on the transition from analog to digital video[3].

FFMPEG is by definition a framework, which can be defined as a platform or a structure for developing software applications. FFMPEG is able to process pretty much anything that humans have created in the last 30 years or so, in terms of audio, video, data and pictures.

It supports the most obscure old formats up to the cutting edge, no matter if they were designed by some standards committee, the community or a corporation[4].

In the last 10 years the content creation has seen an incredible evolution and expansion: if you are a content creator yourself, you will be familiar with tons of the on-line tools, Apps, or subcription based platforms such as the

[3] https://www.streamingmedia.com/Articles/Editorial/Featured-Articles/MPEG-What-Happened-141678.aspx

[4] http://ffmpeg.org/about.html

Adobe Creative Cloud or cutting-edge editing softwares such as FinalCut Pro or Avid Media Composer.

FFMPEG is not a substitute of those softwares, but at the same time it can perform many of their tasks in a smarter, faster and costless way.

Intended Audience

This book is designed to address anyone who is just above the “raw beginner” level. This book will explain some basic process such as entering commands and execute simple code instructions using a **C**ommand-**L**ine-**I**nterface, or "CLI", instead of using high resource-intensive **G**raphical **U**ser **I**nterfaces, or "GUI".

You may review some basic definitions and concepts, or skip directly to the working Formulas, as you'll prefer.

Whether you are at the very beginning or an experienced developer, you will find several effective ways to execute many tasks for your audio/video/streaming needings.
A great deal of the technology discussed in this book is an evolution of discoveries in the field of computer science mainly developed in the early 1970 by Dennis Ritchie and

Ken Thompson: a lot of technology developed back then is still with us today and it will continue to be for a long time.

All the software discussed in this book is mostly free and open-source and is developed by extremely talented developers around the world.

Two Google engineers, for example, have been amongst the major contributors of the FFMPEG project[5].

A chapter of this book is entirely dedicated for the basic definitions of most of the technical terms used in this text.

Tested Platforms

All the instructions and Formulas described in this book have been successfully tested on a MacBook Pro with MacOS X Catalina 10.15.6, on Ubuntu 18.04 and 20.04.

For Windows users: while there is a way to install FFMPEG as a standalone executable .exe, i suggest you to install the BASH Shell for Windows by following the step-by-step guide available here:

`https://docs.microsoft.com/en-us/windows/wsl/install-win10`

[5] FFMPEG and a thousand fixes - Google Blog: https://security.googleblog.com/2014/01/ffmpeg-and-thousand-fixes.html

Basic Definitions

As mentioned before, in order to use all the programs and tools described in this book you will need to use a "Shell" and more specifically the BASH shell.

SHELL: Is a UNIX term for a user interface to the system: something that lets you communicate with the computer via the keyboard and the display thru direct instructions (Command Lines) rather than with a mouse and graphics and buttons. The shell's job, then, is to translate the user's command lines into operating system instructions[6].

BASH: Bash is the shell, or command language interpreter, for Unix-like systems.
The name is an acronym for the '`Bourne-Again SHell`', a pun on Stephen Bourne, the author of the direct ancestor of the current Unix shell "`sh`", which appeared in the 7th Edition Bell Labs Research version of Unix, in 1979.

6 Newham, Cameron. Learning the bash Shell (In a Nutshell) (O'Reilly)

ENCODE: The process to compress a file so to enable a faster transmission of data.

DECODE: The function of a program or a device that translates encoded data into its original format.

CODEC: A codec is the combination of two words en**CO**der and **DEC**oder. An encoder compress a source file with a particular algorithm: then a decoder can decompress and reproduce the resulting file.
Common examples of video codecs are: MPEG-1, MPEG-2, H.264 (aka AVC), H.265 (aka HEVC), H.266 (aka VVC), VP8, VP9, AV1, or audio codecs such as Mp3, AAC, Opus, Ogg Vorbis, HE-AAC, Dolby Digital, FLAC, ALAC.

BITRATE: Bitrate or data rate is the amount of data per second in the encoded video file, usually expressed in kilobits per second (kbps) or megabits per second (Mbps). The bitrate measurement is also applied to audio files.
An Mp3 file, for example, can reach a maximum bitrate of 320 kilobit per second, while a standard CD (non-compressed) audio track can have up to 1.411 kilobit per

second. A typical compressed h264 video in Full-HD has a bitrate in the range of 3.000 - 6.000 kbps, while a 4k video can reach a bitrate value up to 51.000 kbps.
A non-compressed video format, such as the Apple ProRes format in 4K resolution, can reach a bitrate of 253.900 kbps and higher.

CONTAINER: Like a box that contains important objects, containers exist to allow multiple data streams, such as video, audio, subtitles and other data, to be embedded into a single file. Amongst popular containers there are:
MP4 (.mp4), MKV (.mkv), WEBM (.webm), MOV (.mov), MXF (.mxf), ASF (.asf), MPEG Transport Stream (.ts), CAF (Core Audio Format, .caf), WEBP (.webp).

MUX: This is the process of taking encoded data in the form of 'packets' and write it into files, in a specific container format.

DEMUX: The process of reading a media file and split it into chunks of data.

TRANSMUXING: Also referred to as "repackaging" or "packetizing", is a process in which audio and video files are repackaged into different delivery formats without changing the original file content.

TRANSCODING: The process of converting a media file or object from one format to another.

RESOLUTION: Resolution defines the number of pixels (dots) that make up the picture on your screen.

For any given screen size the more dots in the picture, the higher the resolution and the higher the overall quality of the picture. TV resolution is often stated as the number of pixels or dots contained vertically in the picture.
Each of these resolutions also has a number and a name associated with it. For example: 480 is associated to SD (Standard Definition). 720 and 1080 are associated to HD (High-Definition), 2160 is associated to UHD (Ultra-High-Definition) and finally 4320 is associated to 8K UHD.

ASPECT RATIO: This is the ratio (relation) of width to height of the TV screen. Certain aspect ratios are designed to handle certain resolutions without any stretching or distortion of the picture and without any blank space around the picture. Common aspect ratios are the "4:3 aspect ratio" meaning that for every 4 inches of width in an image, you will have 3 inches of height. Another popular aspect ratio is 16:9, meaning thath for every 16 inches of width in an image, you will have 9 inches of height.

INTERLACED FORMAT: It's a technique invented in the 1920s to display a full picture by dividing it into 2 different set of lines: the even and the odd lines.
The even lines are called "even field", while the odd lines are called the "odd field".

The even lines are displayed on the screen, then the odd lines are displayed on the screen, each one every 1/60th of a second: both of these, even and odd fields, make up one video frame. This is one of the earliest video compression methods[7].

[7] https://becg.org.uk/2018/12/16/interlacing-the-hidden-story-of-1920s-video-compression-technology/

PROGRESSIVE: Progressive format refers to video that displays both the even and odd lines, meaning the entire video frame, at the same time.

LETTER BOX: Letterboxing is the practice of transferring film shot in a widescreen aspect ratio to standard-width video formats while preserving the content's original aspect ratio. The resulting videographic image has black bars above and below it. LBX or LTBX are the identifying abbreviations for films and images so formatted.

VOD/SVOD: Acronym for **V**ideo **O**n **D**emand / **S**ubscription-based **V**ideo **O**n **D**emand.

OTT: Abbreviation for "Over-the-top". A video streaming service offered directly to the public thru an internet connection, rather than thru an antenna, cable or satellite.

STREAMING: The process of delivering media thru small chunks of compress data and sent to a requesting device.

RTMP: Real-Time-Messaging-Protocol. A proprietary protocol originally developed by Macromedia, today Adobe.

HLS: **H**TTP-**L**ive-**S**treaming Protocol. Created by Apple for delivering video destined for Apple devices.

DASH: **D**ynamic **A**daptive **S**treaming over **H**TTP.
This protocol can be defined as the open-source HLS.

M3U8: A standard text file encoded in UTF-8[8] format and organized as a playlist of items with their location, to be reproduced in a particular sequence.

AV1: Stands for AOMedia Video 1, which is an acronym for Alliance for Open Media Video 1, a codec developed in joint-venture by Amazon, Cisco, Google, Intel, Microsoft, Mozilla, Neflix, ARM, NVidia, Apple and AMD.

8 UTF-8 stands for Universal Text Format 8

BATCH PROCESSING: The act of processing a group of catalogued files.

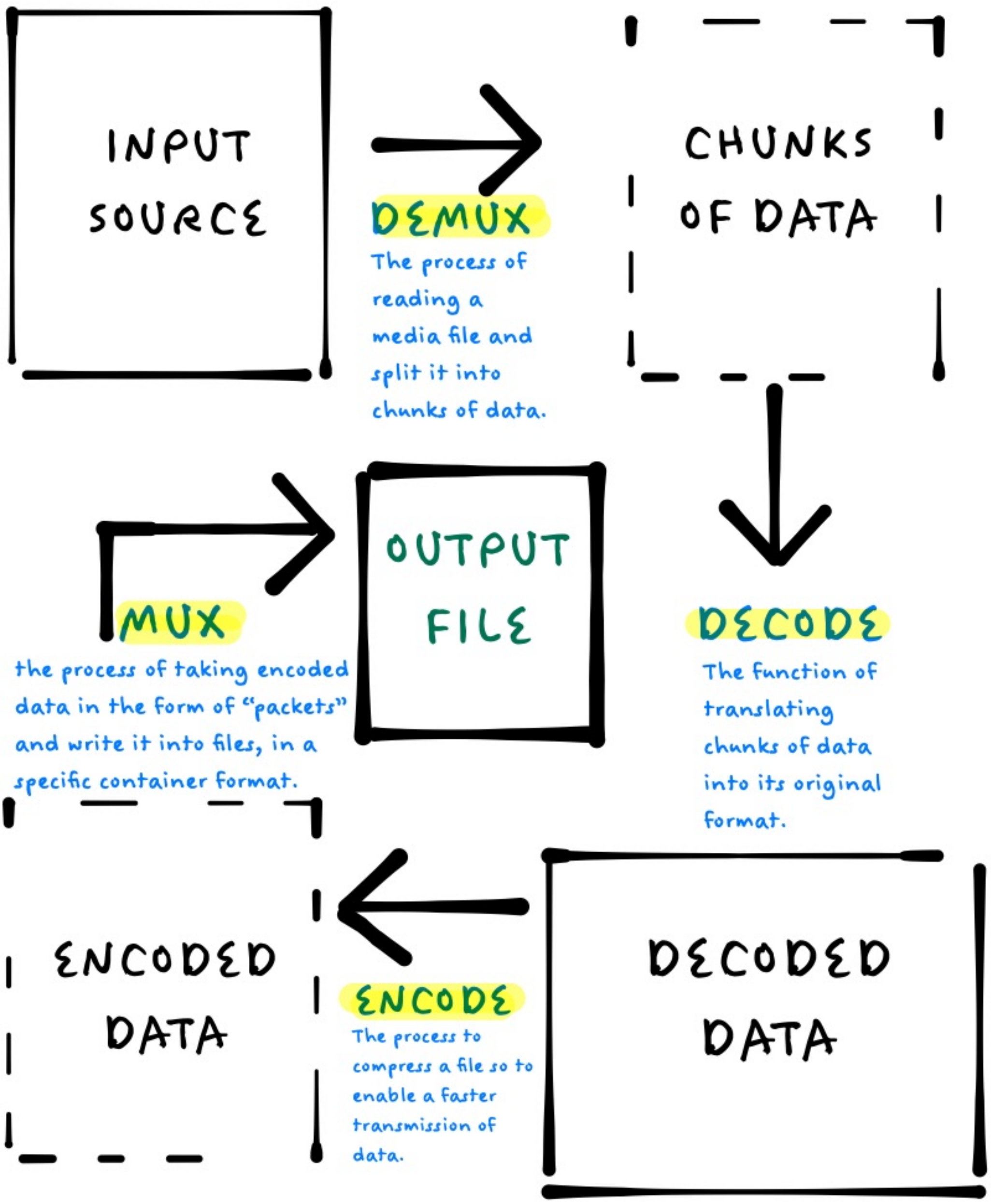
HDR: Acronym for High-Dynamic-Range. It is about recreating image realism from camera through postproduction to distribution and display[9].

h264: A standard of video compression defined by the Motion Expert Picture Group (MPEG) and the International Telecommunication Union (ITU). It is also referred to h264/AVC (Advanced Video Coding).

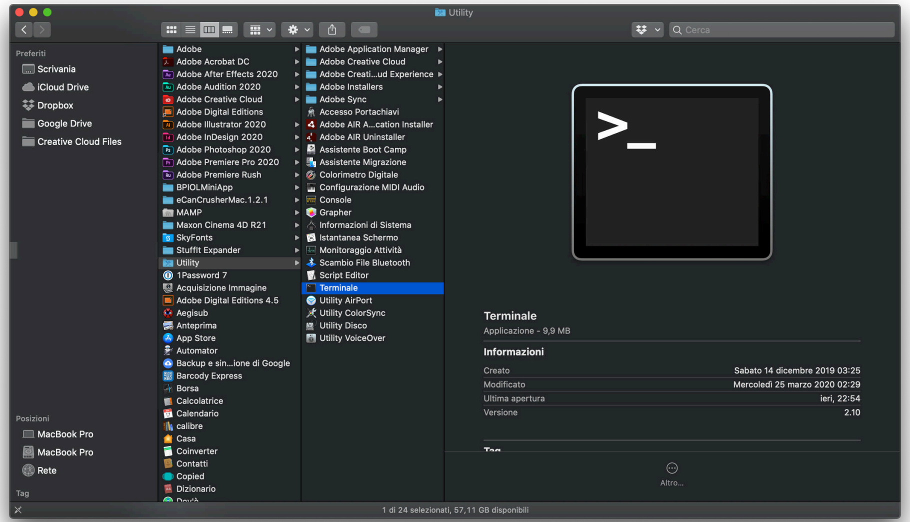
x264: a free software library and application for encoding video streams into the H.264/MPEG-4 AVC compression format.

[9] https://aws.amazon.com/media/tech/what-high-dynamic-range-hdr-video/#:~:text=High%20dynamic%20range%20(HDR)%20video%20technology%20is%20the%20next%20great,postproduction%20to%20distribution%20and%20display.

Basic FFMPEG Workflow


INPUT SOURCE
CHUNKS OF DATA
DEMUX
The process of reading a media file and split it into chunks of data.
OUTPUT FILE
MUX
the process of taking encoded data in the form of "packets" and write it into files, in a specific container format.
DECODE
The function of translating chunks of data into its original format.
ENCODED DATA
ENCODE
The process to compress a file so to enable a faster transmission of data.
DECODED DATA

How to Install FFMPEG



Terminale
Applicazione - 9,9 MB

Informazioni

Creato	Sabato 14 dicembre 2019 03:25
Modificato	Mercoledì 25 marzo 2020 02:29
Ultima apertura	ieri, 22:54
Versione	2.10

1 di 24 selezionati, 57,11 GB disponibili

MacOS X contains a great app called "Terminal" which is inside the Utility Folder.
The Terminal application on MacOS X it's a standard shell with a pre-installed BASH version 3.
Please note: the latest MacOS X version use an extended version of BASH named ZSH (aka "Zed Shell"). In order to use all the tools and commands described in this book i suggest you to switch from **ZSH** to **BASH**.
To do so, just type the word *bash* as soon as your Terminal will open, and press Enter. This will enable the BASH shell immediately.

First thing to do with Terminal: choose or locate your default working directory.

By default Terminal will process your commands on the current USER directory path (aka your Home Directory). You will find lots of tilde symbol (~) in this book: that tilde symbol means "your home directory".

To access your "Desktop" directory on MacOS X, for example, you can type the following command on Terminal:

```
cd ~/Desktop
```

The above command means "**C**hange the **D**irectory where i am located, to the user Desktop". You just want to make sure that everything you will output from your Terminal will be easily located on your computer, once the processes are done.

For example: you might want to process a large batch of videos. In this case it will a good idea to can create a folder on your desktop and call it "processed"

To create this folder you will type:

```
mkdir ~/Desktop/processed
```

To access this folder on your Desktop, within Terminal, you will type:

```
cd ~/Desktop/processed
```

While there are several ways to install FFMPEG on MacOS X or on Ubuntu, i suggest you to download and install one specific program called "HomeBrew" thru the Terminal Application.

Just open the Terminal Application on MacOS X and then paste the following code:

```
/bin/bash -c "$(curl -fsSL https://
raw.githubusercontent.com/Homebrew/install/
master/install.sh)"
```

The above mentioned code is a BASH command that basically says: "*Please BASH run the command within the quotes so to download and install HomeBrew from the HomeBrew's Website onto my computer*".

For the sake of simplicity, the above command won't be analyzed word by word. A complete techcnical explanation of the command can be founded on the **Additional Notes** of this book and/or by visiting the Bash Reference Manual available at the GNU.org website.

Install a Basic FFMPEG version thru HomeBrew

After installing HomeBrew it's time to install FFMPEG. At the time of writing this book the latest version is 4.3.1.

Now type the following code:

```
brew install homebrew-ffmpeg/ffmpeg/ffmpeg
```

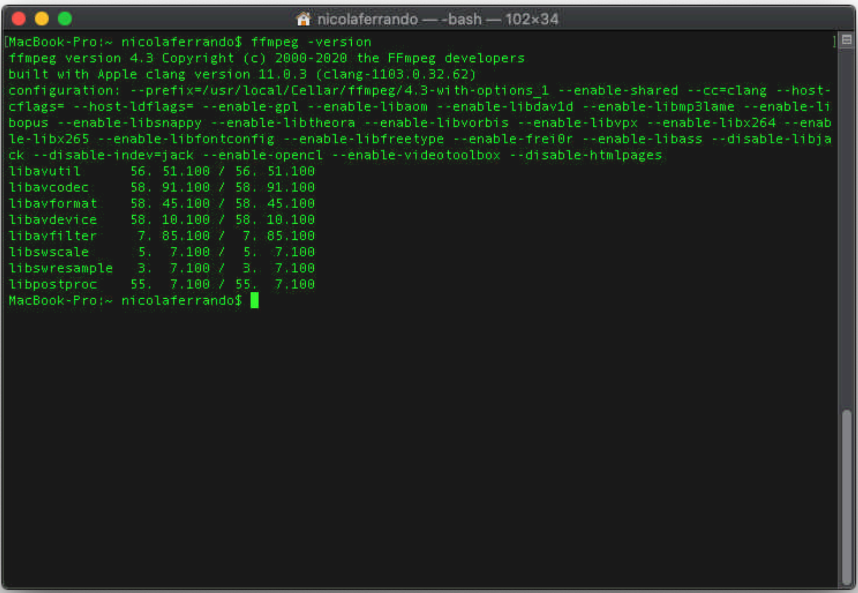
This might take 5 minutes or more.

After completion of the installation process, you might want to check if everything is fine with your installed FFMPEG.

To do so, within your Terminal window, just type the following code:

```
ffmpeg -version
```

Your Terminal will output something like this:



```
MacBook-Pro:~ nicolaferrando$ ffmpeg -version
ffmpeg version 4.3 Copyright (c) 2000-2020 the FFmpeg developers
built with Apple clang version 11.0.3 (clang-1103.0.32.62)
configuration: --prefix=/usr/local/Cellar/ffmpeg/4.3-with-options_1 --enable-shared --cc=clang --host-
cflags= --host-ldflags= --enable-gpl --enable-libaom --enable-libdav1d --enable-libmp3lame --enable-li
bopus --enable-libsnappy --enable-libtheora --enable-libvorbis --enable-libvpx --enable-libx264 --enab
le-libx265 --enable-libfontconfig --enable-libfreetype --enable-frei0r --enable-libass --disable-libja
ck --disable-indev=jack --enable-opencl --enable-videotoolbox --disable-htmlpages
libavutil      56. 51.100 / 56. 51.100
libavcodec     58. 91.100 / 58. 91.100
libavformat    58. 45.100 / 58. 45.100
libavdevice    58. 10.100 / 58. 10.100
libavfilter     7. 85.100 /  7. 85.100
libswscale      5.  7.100 /  5.  7.100
libswresample   3.  7.100 /  3.  7.100
libpostproc    55.  7.100 / 55.  7.100
MacBook-Pro:~ nicolaferrando$
```

If you are seeing a similar screen, after running the command above, that means that the installation process of FFMPEG on MacOS X has completed.

Optional: Custom Installation of FFMPEG on MacOS X with Options

You might want to use an FFMPEG version that has all the protocols and libraries needed for a particular task.

For example, you might need to use a **Decklink** acquisition card from BlackMagic Design.
In this case you can check the options available on Brew by typing the following command:

```
brew info homebrew-ffmpeg/ffmpeg/ffmpeg
```

This will print out all the supported options.
In order to install FFMPEG with the desired options, you will need to type the command along with the desired option to install. In the example of DeckLink support, you will have to install the Blackmagic Desktop Video and Driver libraries and then type the following command:

```
brew install homebrew-ffmpeg/ffmpeg/ffmpeg
--with-decklink
```

As mentioned before FFMPEG can be installed in many ways, and with some options that can process many formats and standards created in the last 30 years. Amongst these formats there are patented formats, such as the common Mp3 format, which is a registered patent of the **Fraunhofer Institute**, which deals with licenses and authorized uses for commercial purposes.

For example: if you want to use a proprietary codec such as the AAC into an App or a commercial software, and embed FFMPEG as a component of your App, you might want to study **this legal section of the FFMPEG website**.

As per the Blackmagic Design option above described, if you need to use FFMPEG on a specific configuration or output a special format you may want to install a custom version of FFMPEG.

Let's take an example: the AAC audio compression codec.

This audio codec can be used freely by calling the native AAC encoder option with the command `-c:a aac` (which means "Codec for Audio: AAC"). But you might need to use a special version of the AAC codec, such as the HE-AAC (High Efficiency Advance Audio Coding).

To use this patented format, which is yet another Fraunhofer Institute Patent, you will need to install FFMPEG with the **`libfdk_aac`** library.

To enable this option you will have to type:

```
brew install homebrew-ffmpeg/ffmpeg/ffmpeg
--with-fdk-aac
```

From a quality standpoint the **`libfdk_aac`** option will produce superior results from the native FFMPEG's AAC encoder. You might want to investigate further this point, by reading this **FFMPEG Guideline** for high-quality lossy audio encoding.

On MacOS X and with HomeBrew installed, you can check all the available install options of FFMPEG, by typing on your Terminal the following command:

```
brew options homebrew-ffmpeg/ffmpeg/ffmpeg
```

Installing FFMPEG 3.X on Linux (Ubuntu)

On Ubuntu releases you can open the Terminal Application which is already installed on the operating system.

To install FFMPEG, open Terminal and type the following set of 2 command:

```
sudo apt update&&sudo apt install ffmpeg
```

Please keep in mind that this Ubuntu version of FFMPEG might not be the same version installed with HomeBrew on a Mac OS system.

sudo = “Super User Do!”. This means you are asking the system to perform actions as the main administrator, with full privileges. You might be asked to enter an Administrator Password before proceeding.

apt = Advance Package Tool. Is the most efficient and preferred way of managing software from the command line for Debian and Debian based Linux distributions like Ubuntu.

update = This will update all the software. This is the step that actually retrieves information about what packages can be installed on your Ubuntu system, including what updates to currently installed packages are available.

&& = this is used to chain commands together, such that the next command is run *if and only if* the preceding command exited without errors

apt install ffmpeg = this is the command used to install the package FFMPEG, available on the Advance Package Tool.

Installing FFMPEG 4.x on Ubuntu

```
sudo apt install snapd&&sudo snap install
ffmpeg
```

Please note that this version of FFMPEG will install also the "non-free" option and licensed codecs, such as the **`libfkd_aac`**. Please refer to the above Custom Installation section for legal terms and conditions that might apply in your specific development case.

Installing FFMPEG 4.x on Ubuntu with all the Bells and Whistles

FFMPEG can be installed in many ways. As discussed earlier you might use a Mac, therefore you might want to install FFMPEG thru HomeBrew or by following the official instructions on how to compile FFMPEG with custom dependancies, available here:

https://trac.ffmpeg.org/wiki/CompilationGuide

Note: you might encounter an error message using SNAP package, similar to this one:

The command could not be located because '/snap/bin' is not included in the PATH environment variable[10].

If this is the case, you can edit a file called **`/etc/environment`** and add **`/snap/bin`** in the list then restart your system.

[10] **PATH** is an **environment variable** on Unix-like operating systems, DOS, OS/2, and Microsoft Windows, specifying a set of directories where executable programs are located

For example:

```
nano /etc/enviroment
```

Then edit the file, by adding **`/snap/bin`** at the end of the list:

```
  PATH="/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/
local/games:/snap/bin"
```

Then you can restart your system:

```
sudo reboot
```

Installing FFMPEG 4.x on Ubuntu with HomeBrew

If you want to install FFMPEG with the Homebrew Package Manager, you can do so by typing:

```
/bin/bash -c "$(curl -fsSL https://
raw.githubusercontent.com/Homebrew/install/
master/install.sh)"
```

And then by typing:

```
brew install homebrew-ffmpeg/ffmpeg/ffmpeg
```

If you need to install additional codec or protocols, you can follow the same instructions as above for MacOS X's HomeBrew.

Installing FFMPEG on Windows

As per the other platforms, on Windows machines you can install FFMPEG in several ways as described before. Particularly with Windows 10 you can install a component named “*Windows Subsystem for Linux*”.
This can give you the possibility to install a full Ubuntu release. To use the formula described in this book I reccomend you to install Ubuntu 18.04 LTS or 20.04 LTS. For more information about installing BASH on Windows 10 please refer to this page:

https://docs.microsoft.com/en-us/windows/wsl/install-win10

Once installed the “Windows Subsystem for Linux” and a release of Ubuntu, you can then type the following line:

```
sudo apt-get install ffmpeg
```

Pre-Requisites

Although FFMPEG can be installed on a very old machine even with no graphic card installed, much faster performances can be achieved with newer machines and one or more graphic cards. If you have a configuration of one or more GPUs you can also enable a specific option called “Hardware Acceleration” on FFMPEG, and achieve an even faster experience on some operations.

However, for the sake of simplicity, this specific options and all their variants won’t be covered on this book.

If you are interested in discovering and enabling the “Hardware Acceleration” option for your FFMPEG installation please take a look at the following article:

https://trac.ffmpeg.org/wiki/HWAccelIntro

Approximately every 6 months the FFmpeg project makes a new major release. Between major releases point releases will appear that add important bug fixes but no new features[11].

[11] https://ffmpeg.org/download.html

Basic Syntax Concepts of FFMPEG

Once you have FFMPEG installed, you can run the program just by typing the word `ffmpeg` on your Terminal. The basic flow of FFMPEG is pretty straight-forward.

The program will expect an input file, might expect some options to trasform it, or to keep it as the original, and then it will create an output file.

Let's take the following very basic example.
I want to take an audio file in ".wav" format (e.g.:"mysong.wav") and convert it into an Mp3 audio file, therefore creating a new file called "mysong.mp3".

Assuming that the file "mysong.wav" is already in the current working directory, i will type a command just like the following:

```
ffmpeg -i mysong.wav mysong.mp3
```

The **"-i"** stands for "input". This input can be a local file in your computer or a file from a remote URL.

The file extension specified at the end of my output file, will make FFMPEG automatically choose the Mp3 format with some of the best options for that kind of file.

But having to deal with many videos or different audio files formats can be a little more complicated than this.

You might want to select a specific bitrate for that Mp3 output, or your WAV file might need to be pre-processed for example with a standard loudness normalizer filter, *before* the conversion happens.

You might end with a more complex command such as this one:

```
ffmpeg -i mysong.wav -af loudnorm -c:a mp3
-b:a 256k -ar 48000 mysong.mp3
```

This will take the same original file, but this time it will process it and then convert it to an Mp3 file.

Let's breakdown the above example:

`ffmpeg -i mysong.wav:` will process the input file "mysong.wav"

`-af loudnorm` : will apply an "Audio Filter Loudnorm" which will perform a standard audio loudness normalization

`-c:a mp3:` stands for "codec audio Mp3". This option will specify that we want to export an Mp3 file.

`-b:a 256k:` stands for "bitrate of the audio" and will produce an audio file with "Bitrate Audio at 256 Kbit/s"

`-ar 48000:` stands for "Audio Rate". This option will specify furthemore that we want to create an output file with a specific audio sampling rate (in this example 48000 KHz).

The same syntax structure is meant for video files.

Let's make an example: an Apple ProRes file ("master.mov") must be converted in h264.

The syntax will be pretty simple:

```
ffmpeg -i master.mov master.mp4
```

The extension used on the output file will trigger several default options of FFMPEG in order to produce a good quality h264 file in a mp4 container.

But we may want to have a smaller file or a particular audio option, or may need to have a stereo downmix from a 5.1 soundtrack, or again, we might want to process and output just the first 2 minutes and 30 seconds of a 3 hours long video from an URL.

Thus, we might have an example case like this:

```
ffmpeg -i https://www.dropbox.com/master.mov
-t 00:02:30 -c:v h264 -b:v 3M -c:a aac -b:a
128k -ar 44100 -ac 2 master.mp4
```

-i: stands for "input"

-t: stands for "process only for a specified duration"

-c:v: stands for "codec video"

-b:v: stands for "video bitrate" which is the amount of desired video data per second, expressed in Megabit

-c:a: stands for "audio codec"

-b:a: stands for "audio bitrate", which is the amount of the desired audio data per second, expressed in kilobit

`-ar:` stands for "Audio Rate", which is also known as "sampling rate".

A *sample* is a measurement – a snapshot, if you will – at one specific time in that audio track, described in the binary language of 1s and 0s.

Repeat that measurement tens of thousands of times each second; how often that snapshot is taken represents the sample rate or sampling frequency, measured in kiloHertz, a unit meaning 1,000 times per second.

Audio CDs, for example, have a sample rate of 44.1kHz, which means that the analog signal is sampled 44,100 times per second.

`-ac 2`: stands for "2 Audio Channels (Left+Right)".

This option is useful when you have to down-mix an audio file that contains multiple tracks, such in the case of Dolby Surround 5.1/7.1 and similar audio tracks.

Keyframes: Basic Concepts

Any video content is made up of a series of frames. Usually denoted as FPS (frames per second), each frame is a still image that when played in sequence creates a moving picture.

A content at 30 FPS means that there are 30 “still images” that will play for every second of video[12].

In the video compression world a frame can be compressed in many different ways, depending on the algorithm used. These different algorithms are mostly referred as "picture types" or "frame types".

Some of the most advanced codecs uses 3 frame types or picture types, namely:

• The **I-frame:** a frame that stores the whole picture

• The **P-frame:** a frame that stores only the changes between the current picture and previous ones

• The **B-frame:** a frame that stores differences with previous or future pictures

These 3 frame types forms a **G**roup **O**f **P**ictures (or GOP).

[12] https://blog.video.ibm.com/streaming-video-tips/keyframes-interframe-video-compression/

The I-frame: Short for *intraframe,* a video compression method used by the MPEG standard.

In a motion sequence, individual frames of pictures are grouped together, to form the above mentioned GOP, and played back so that the viewer registers the sense of motion.

An I-frame, also called *keyframe,* is a single frame of digital content that the compressor examines independently of the frames that precede and follow it *and stores all of the data needed to display that frame*.

Typically, the I-frames are interspersed with P-frames and B-frames in a compressed video.

The more I-frames that are contained the better quality the video will be. However, I-frames contains the most amount of bits and therefore take up more space on the storage medium[13].

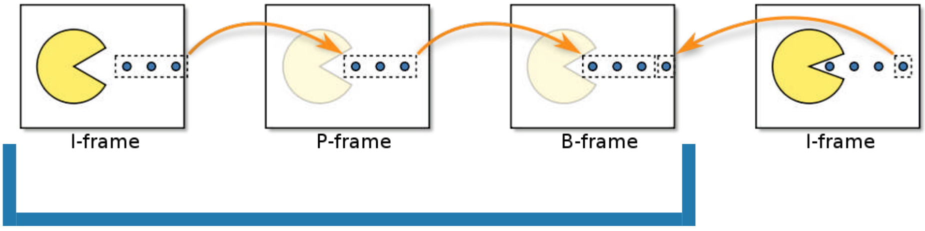
---

[13] https://www.webopedia.com/TERM/I/I_frame.html#:~:text=Short%20for%20intraframe%2C%20a%20video,%EF%BF%BD%EF%BF%BDs%20spatial%20motion.

The P-Frame: Short for *predictive frame*, or *predicted frame*. It follows I-frames and *contain only the data that have changed from the preceding I-frame,* such as color or content changes. Because of this, P-frames depend on the I-frames to fill in most of the data.

The B-Frame: Short for *bi-directional frame*, or *bi-directional predictive frame*. As the name suggests, *B-frames rely on the frames preceding and following them*. B-frames contain only the data that have changed from the preceding frame or are different from the data in the very next frame.

The Group of Pictures: A group of pictures, or GOP structure, that specifies **the order** in which *I-frames, B-frames* and *P-frames* are arranged.



Group of Pictures (GOP)

How Do You Set A Keyframe Interval and Why?

Assuming that you need to stream a file at 25FPS, using the h264 codec, a good idea will be to specify a keyframe interval of 2 seconds, using the following formula:

```
ffmpeg -i YOUR_INPUT -c:v h264 -keyint_min
25 -g 50 -sc_threshold 0 OUTPUT.mp4
```

`-i`: your input file

`-c:v h264`: is the h264 codec selector

`-key_int_min`: specifies the minimum interval to set a keyrame. In this example will set a minimum keyframe interval every 25 frames.

`-g 50`: this option stands for "Group of Pictures" and will instruct FFMPEG to sets the maximum keyframe interval every 50 frames, or 2 seconds, assuming that your input runs at 25 FPS.

`-sc_threshold 0`: This "SC" stands for Scene Change. FFmpeg defaults to a keyframe interval of 250 frames and inserts keyframes at scene changes, meaning when a picture's content change. This option will make sure not to add any new keyframe at scene changes.

The reason why a short keyframe interval is very important is because we need to provide the end user with a fast experience during playback or seeking (fast forward or rewind) a video, especially for the Adaptive Bitrate case scenario, where the end user automatically receives the highest or the lowest quality, based on his own available bandwidth. Also, a player can not start playback on a *p-frame* or *b-frame.*

How to check the Keyframe Interval on a existing video?

With the installation of FFMPEG you will also install a utility called FFPROBE. FFPROBE gathers information from multimedia streams and prints them in human- and machine-readable fashion[14]. To check the keyframe interval with FFPROBE you can use the following formula:

```
ffprobe -loglevel error -skip_frame nokey
-select_streams v:0 -show_entries
frame=pkt_pts_time -of csv=print_section=0
YOUR_FILE.mp4
```

This will list the exact time for every I-Frame (Keyframe), expressed in seconds, skipping P-Frames and B-Frames.

To define all the command of this formula, please refer to the Additional Notes of this book.

14 https://ffmpeg.org/ffprobe.html

Metadata and FFPROBE

When it comes to video, metadata, which is *a set of data that describes and gives information about other data,* is used to describe information related to the video asset.

The metadata can be displayed as a visible text to the end user, such as tags and descriptions, but also invisible to them, such as keywords or Digital Rights Management (DRM) information used to secure an asset against piracy concerns[15].

Metadata can also contain important informations on the audio track of a video, including the authors, composers, performers, the music genre, all the copyright informations, the credits, etc.

As mentioned earlier, FFMPEG comes with 3 different programs. The main program is FFMPEG itself, then there is FFPLAY, a simple player mainly used for testing purposes and then FFPROBE, which gathers informations from your input. It may be employed both as a standalone application or in combination with a textual filter, which may perform

15 https://www.ibm.com/downloads/cas/XVYMEYGM

more sophisticated processing, e.g. statistical processing or plotting, meaning illustrating by the use of a graph.

A basic formula for FFPROBE is:

```
ffprobe YOUR_INPUT.mp4
```

This will output lots of informations (metadata), including the lenght of a file, the type of the codec used for audio and video, the size, the display size, the number of frames-per-second, etc.

You may just want to extract only the time of a stream. This function is called "Stream Duration" and the formula is:

```
ffprobe -v error -select_streams v:0 
-show_entries stream=duration -of 
default=noprint_wrappers=1:nokey=1 
-sexagesimal YOUR_INPUT.mp4
```

Here's the command breakdown:

`ffprobe`: will launch the program

`-v error`: select the "error" log, which will display every errors, including ones which can be recovered from

`-select_streams v:0`: this will select the very first stream of the video input (FFMPEG and FFPROBE starts to count from 0)

`-show_entries stream=duration`: this will instruct FFPROBE to display the duration of the stream

`-of default=noprint_wrappers=1:nokey=1`: this means "print format" (-of). The other options are meant to instruct to display only the requested information (stream duration) without any other additional information.

`-sexagesimal`: this will instruct FFPROBE to display the restuls in sexagesimal format (HH:MM:SS:ms)

The above command will output something like this:

```
0:06:52.100000
```

The duration will be expressed in HH:MM:SS:ms, or sexagesimal (also known as base 60 or sexagenary), which is a numeral system with sixty as its base: originated with the ancient Sumerians in the 3rd millennium BC, was passed down to the ancient Babylonians, and is still used–in a modified form–for measuring time, angles, and geographic coordinates[16].

Exporting FFPROBE Data in TXT

You may need to extract the data into a Text file.
In this case you will need to type the following command:

```
ffprobe YOUR_INPUT.mp4 > output.txt 2>&1
```

Extracting Data in JSON format

JSON stands for JavaScript Object Notation. JSON is a lightweight format for storing and transporting data.

16 https://arstechnica.com/science/2017/08/ancient-tablet-reveals-babylonians-discovered-trigonometry/

It's often used when data is sent from a server to a web page[17] and used mainly by developers.

The formula to export FFPROBE's data into JSON format is as follow:

```
ffprobe -v quiet -print_format json -show_format
-show_streams YOUR_INPUT.mp4 > YOUR_INPUT.json
```

To know more about this command please check the FFPROBE's Official Guide.

You can also check out the FFPROBE Tips Wiki.

[17] https://www.w3schools.com/whatis/whatis_json.asp

Extracting Metadata with FFMPEG

As mentioned earlier, Metadata are a group of data, of the technical aspects of a file and the descriptive aspects of it, including the title, the author, composers, the name of the album, the comments of the original producers or record label, the framerate, the languages, etc.

Extracting Metadata from Audio Files Only

To extract the existing metadata from an audio file you can use FFMPEG by typing this command:

```
ffmpeg -i YOUR_INPUT -f ffmetadata out.txt
```

FFMPEG's **`-f`** options stands for: *Force input or output file format*. And the value **`ffmetadata`** means "FFMPEG's Dump of Metadata".

Extracting Specific Streams

As the official documentation describes it: "The **`-map`** option is used to choose which streams from the input(s) should be included in the output(s)."

In other words, if you need to extract only a specific stream of your file, this is the right option for you.

How FFMPEG works by default

If you do not use the **`-map`** option then the default stream selection behavior will automatically choose streams[18].

Please note:

• Default stream selection will not automatically choose all of the streams;

• Only one stream per type will be selected. For example, if the input has 3 video streams it will only choose 1;

• The default stream selection will choose streams based upon specific criteria.

[18] https://trac.ffmpeg.org/wiki/Map

Let's take an example file called "home.mov" of a video containing 3 audio tracks: 1) english, 2) italian and 3) spanish, along with their respective subtitles track.

FFMPEG's Map Option can work like this:

`-map 0`: with this option you will select all streams

```
ffmpeg -i home.mov -map 0 -c:a aac -c:v h265
YOUR_OUTPUT.mkv
```

In the above example command you will select all the audio and video streams from the "home.mov" file and convert them into an h265 file, with audio in AAC format, in a default configuration, into an MKV container.

`-map 0:v -map 0:2`: with this option you will extract the first video stream and the second audio stream. In our "home.mov" example will be the "italian" audio track. Please note that FFMPEG starts to count the streams from 0, not from number 1.

To know more about more MAP's options check out the MAP Section on the FFMPEG's documentation.

Extracting Audio Only from a Video

To extract only the audio stream from a video track, you can use the following formula:

```
ffmpeg -i YOUR_INPUT.mp4 -vn -c:a copy YOUR_AUDIO.aac
```

Where **-vn** stands for "no video".

The above command will copy the audio track without converting it, assuming that your input contains an AAC (Advanced Audio Codec) audio track.

If your input contains a different audio codec, then you can change the output's file extension, accordingly to the codec you need.

Otherwise you can choose to convert the audio input, selecting the desired codec and parameters, for example:

```
ffmpeg -i YOUR_VIDEO.mov -vn -c:a mp3 -b:a 128k -ar 44100 -ac 2 YOUR_AUDIO.mp3
```

This will extract the audio track from input "YOUR_VIDEO.mov" and will convert it to an Mp3 file, at 128 Kbps, 44.100 kHz, in Stereo.

Extracting Video Only without Audio

If you need to extract only the video from a file, without any audio, you can use the following formula:

```
ffmpeg -i YOUR_VIDEO -an -c:v copy OUTPUT
```

Where **`-an`** stands for "no audio".

The above example will produce an exact copy of the video, without re-encoding it (**`-c:v copy`**).

If you need to convert your input's video without the audio, then you can use a formula like this:

```
ffmpeg -i YOUR_VIDEO.mov -an
YOUR_VIDEO_WITHOUT_AUDIO.h265
```

The above command will trigger some of the best options in order to convert the example ".mov" input into a standard h265 file, without the original audio.

Cutting Videos with FFMPEG

With FFMPEG you can extract portions of video with the Seeking function.

With the function **-ss** (before **-i**) you can set a desired "In" marker and with the function **-t** you can set an "Out" marker for the specified duration.

Please note that if you want to cut a video without re-encoding it, using the **-c copy** option, there won't be general rule on how correctly set both time points for **-ss** and **-t** options, because *those depend on the keyframe interval used when the input was encoded*.

To give some orientation, the x264 encoder by default uses a GOP size of 250, which means 1 keyframe each 10 seconds if the input frame rate is 25 fps.

Extracting 20 seconds without re-encoding

```
ffmpeg -ss 00:01:30.000 -i YOUR_VIDEO.mp4 -t
00:00:20.000 -c copy
YOUR_EXTRACTED_VIDEO.mp4
```

Let's break it down:

`-ss 00:01:30.000`: If putted before the **`-i`** input, this seeks in the input file at the specified time value. Time can be expressed in hours, minutes, seconds and milliseconds. `hh:mm:ss.ms` format.

`-i YOUR_VIDEO.mp4`: this is an example input video

`-t 00:00:20.000`: the duration of the desired portion, that you can specify in sexasegimal format (**`hh:mm:ss.ms`**). In this example a duration of 20 seconds.

`-c copy`: this option will copy the audio and video streams, without re-encoding them

Extracting 20 seconds with encoding

```
ffmpeg -ss 00:01:30.000 -i YOUR_VIDEO.mp4 -t 
00:00:20.000 YOUR_EXTRACTED_VIDEO.mp4
```

In this example FFMPEG will re-encode your input and will use basic options to create a new output file, based on the extension used in your output file.

With this re-encoding, you will be sure to have an accurate portion, withtou any black screen or audio drops that might occur with the **`-c copy`** option used in the previous formula, but at a quality expense if you are working with h264 or other lossy formats.

Producing h264/AVC videos

In 2003 the International Telecomunication Union (ITU) stated: "As the costs for both processing power and memory have reduced, network support for coded video data has diversified, and advances in video coding technology have progressed, the need has arisen for an industry standard for compressed video representation with substantially increased coding efficiency and enhanced robustness to network environments."

At the time of writing this book the h264 codec, aka AVC - Advanced Video Codec, it's still currently the most popular codec online, according to a recent survey from BitMovin[19] which enpowers video streaming for Twitter's Periscope and the BBC, amongst others.

This codec has been presented to the public on May 2003.

Netflix and other big player such as Amazon, Cisco, Google and Intel have now formed the "Alliance for Open Media" (AOM) for a more advanced (and royalty-free) type

[19] https://bitmovin.com/bitmovin-2019-video-developer-report-av1-codec-ai-machine-learning-low-latency/

of codec which perphaps will likely replace the h264 standard sooner or later.

But until that day arrives, the most common video format around it's still the h264/AVC codec.

Once again, when working for video streaming purposes the h264/AVC is by recent surveys still the preferred choice.

To better understand the options of compression offered by h264 FFMPEG offers a great H264 encoding guide.

Different h264 encoding approaches

When working with h264 you might want to decide first wheter you want to work with a Constant Rate Factor, **CRF**, which keeps the best quality and care less about the file size, or using a **Two-Pass Encoding**, if you are targeting a specific output file size, and if the output quality from frame to frame is of less importance.

Using a Costant Rate Factor

As stated on the FFMPEG documentation, the range of the CRF scale is 0–51, where 0 is lossless, 23 is the default, and 51 is worst quality possible. A lower value generally leads to higher quality, and a subjectively sane range is 17-28. Consider 17 or 18 to be visually lossless or nearly so; it should look the same or nearly the same as the input but it isn't technically lossless.

Presets

A preset is a collection of options that will provide a certain encoding speed to compression ratio.

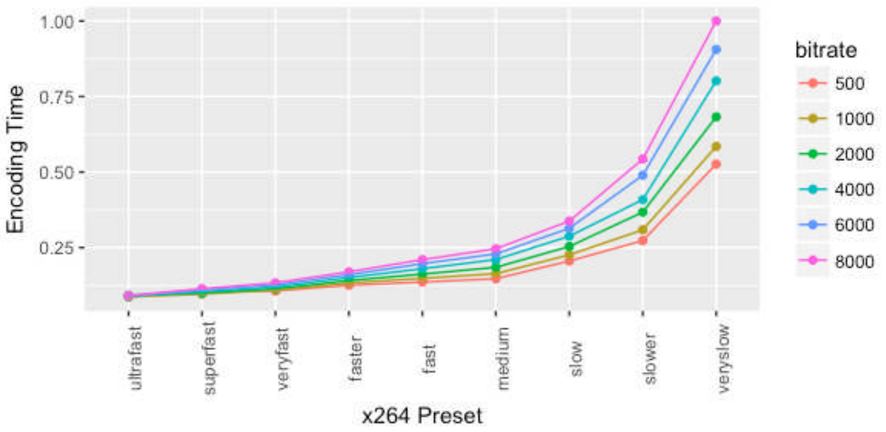
A slower preset will provide better compression (compression is quality per filesize).

This means that, for example, if you target a certain file size or constant bit rate, you will achieve better quality with a slower preset. Similarly, for constant quality encoding, you will simply save bitrate by choosing a slower preset.

Use the slowest preset that you have patience for.

The available presets in descending order of speed are:

- ultrafast
- superfast
- veryfast
- faster
- fast
- medium - (FFMPEG's default preset)
- slow
- slower
- veryslow
- placebo



How do the different presets influence encoding time?

Take a look at the above table.

Going from *medium* to *slow*, the time needed increases by about 40%. Going to slower instead would result in about 100% more time needed (i.e. it will take twice as long). Compared to *medium*, *veryslow* requires 280% of the original encoding time, with only minimal improvements over slower in terms of quality.

Using fast saves about 10% encoding time, faster 25%. ultrafast will save 55% at the expense of much lower quality.

The Tune Option

FFMPEG gives you also a **-tune** option to change settings based upon the specifics of your input.

The tune options are:

```
film – use for high quality movie content;
animation – good for cartoons;
grain – preserves the grain structure in old, grainy film material;
stillimage – good for slideshow-like content;
fastdecode – allows faster decoding by disabling certain filters;
zerolatency – good for fast encoding and low-latency streaming;
psnr – only used for codec development;
ssim – only used for codec development;
```

Example:

```
ffmpeg -i your_video.mov -c:v h264 -crf 23
-tune film your_output.mp4
```

Profile in h264

h264 has been built with different profiles for different devices destination.

Although there are several profiles in the h264 codec, FFMPEG has 3 profiles available:

Baseline: Some devices (very old and/or obsolete) only support the more limited Constrained Baseline or Main profiles.

Main: Same as above

High: Most modern devices support the more advanced High profile.

To specify a particular profile in FFMPEG you can set:

```
 -profile:v baseline
```

or

```
- profile:v main
```

or

```
-profile:v high
```

With that being said, unless you need to support limited devices, the FFMPEG recommendation is **to omit setting the profile** which will allow FFMPEG's x264 to automatically select the appropriate profile based on the input and the options inserted.

With the above considerations in mind, a basic formula for converting a source file into a standard quality h264 video, using the CRF approach, will be as follow:

```
ffmpeg -i [YOUR_SOURCE_VIDEO] -c:v h264
-preset medium -tune film -crf 23 -c:a copy
output.mp4
```

The 2-Pass Encoding

If you are targeting a specific output file size, and if output quality from frame to frame is of less importance, then you may want to use a 2-Pass encoding method.

For two-pass, you need to run FFMPEG twice, with almost the same settings, except for:

In pass 1 and 2, use the **`-pass 1`** and **`-pass 2`** options, respectively.

In pass 1, output to a null file descriptor, not an actual file. This will generate a logfile that FFMPEG needs for the second pass.

In pass 1 you need to specify an output format (with **-f**) that matches the output format you will use in pass 2.

In pass 1 you can leave audio out by specifying **`-an`.**

Example Formula for 2-Pass Encoding with h264

```
ffmpeg -y -i [YOUR_INPUT] -c:v h264 -b:v 
2600k -pass 1 -an -f mp4 /dev/null && \
ffmpeg -i [YOUR_INPUT] -c:v h264 -b:v 2600k 
-pass 2 -c:a aac -b:a 128k output.mp4
```

Lossless h264 Encoding

As mentioned earlier, the **-crf** option gives you the option to encode a video using a "constant rate factor".

You can use **`-crf 0`** to create a lossless video.

Two useful presets for this case scenario are *ultrafast* or *veryslow* since either a fast encoding speed or best compression are usually the most important factors.

Example:

```
ffmpeg -i [YOUR_INPUT] -c:v h264 -preset 
ultrafast -crf 0 [YOUR_OUTPUT]
```

or:

```
ffmpeg -i [YOUR_INPUT] -c:v libx264 -preset 
veryslow -crf 0 [YOUR_OUTPUT]
```

Note that lossless output files will be huge in terms of file size and most non-FFmpeg based players, such as Quicktime, proably won't play them.
Therefore, if compatibility or file size are an issue, you should not use lossless. The FFMPEG documentation gives also a tip, which can be useful for achievieng similar results without using a **`-crf 0`** command.

"If you're looking for an output that is roughly "visually lossless" but not technically lossless, use a -crf value of around 17 or 18 (you'll have to experiment to see which value is acceptable for you). It will likely be indistinguishable from the source and not result in a huge, possibly incompatible file like true lossless mode."

Constant Bit Rate (CBR)

According to the FFMPEG Documentation, there is no native or true CBR mode, but you can "simulate" a constant bit rate setting by tuning the parameters of a one-pass average bitrate encode:

```
ffmpeg -i input.mp4 -c:v h264 -x264-params
"nal-hrd=cbr" -b:v 1M -minrate 1M -maxrate
1M -bufsize 2M output.ts
```

In this formula, **`-bufsize`** is the "rate control buffer", so it will enforce your requested "average" (1 MBit/s in this case) across each 2 MBit worth of video.

Here it is assumed that the receiver / player will buffer that much data, meaning that a fluctuation within that range is acceptable. The `.ts` extension in the above example is a '*transport stream*' container, a standard container used often for streaming purposes.

Costrained Encoding

If you want to constrain the maximum bitrate used, or keep the stream's bitrate within certain bounds, there is a formula to achieve this result. This is particularly useful for online streaming, where the client expects a certain average bitrate, but you still want the encoder to adjust the bitrate per-frame.

You can use **`-crf`** or **`-b:v`** with a maximum bit rate by specifying both **`-maxrate`** and **`-bufsize`**:

```
ffmpeg -i input -c:v h264 -crf 23 -maxrate
1M -bufsize 2M output.mp4
```

In another example, instead of using constant quality (CRF) as a target, the average bitrate is set.

A two-pass approach is preferred here.

First Pass:

```
ffmpeg -i input -c:v h264 -b:v 1M -maxrate
1M -bufsize 2M -pass 1 -f mp4 /dev/null
```

And then:

```
ffmpeg -i input -c:v h264 -b:v 1M -maxrate
1M -bufsize 2M -pass 2 output.mp4
```

Faststart

As mentioned in the FFMPEG documentation, you can add **`-movflags +faststart`** as an output option if your videos are going to be viewed in a browser.

This will move some informations at the beginning of your file and allow the video to begin playing, before it is completely downloaded by the viewer.

It is also recommended by YouTube.

Example:

```
ffmpeg -i input -c:v h264 -crf 23 -maxrate 
1M -bufsize 2M -movflags +faststart 
output.mp4
```

Producing h265/HEVC Videos

Developed by Multicoreware in 2013, the h265 aka High-Efficiency-Video-Codec (or HEVC) was created in order to meet the increasing demand for high definition and ultra-high definition video, along with an increasing desire for video on demand which led to exponential growth in demand for bandwidth and storage requirements[20].

The h265 codec can offer around 25-50% bitrate savings compared to H.264 video encoded with h264, while retaining the same visual quality. These gains will be most pronounced at resolutions of 1080p and higher.

To encode video in h265 format FFMPEG needs to be built with the `--enable-gpl --enable-libx265` configuration flags and requires x265 to be installed on your system. Please refer to the chapter "How to Install FFMPEG" for know more about custom installations.

[20] https://x265.readthedocs.io/en/default/introduction.html#about-hevc

Rate Control Modes

Similar to x264, the x265 encoder has multiple rate control algorithms, including:

- 1-pass target bitrate (by setting **`-b:v`**)
- 2-pass target bitrate
- Constant Rate Factor (CRF)

FFMPEG documentation states that 1-pass target bitrate encoding is not recommended, thus we'll explore the the CRF method and the 2-pass target bitrate.

Costant Rate Factor (CRF)

This approach can be used if you want to retain good visual quality and don't care about the exact bitrate or filesize of the encoded file.

As with h264, As with x264, you need to make two choices.

1) Choose a CRF: the default is 28, and it should visually correspond to FFMPEG's h264 compression at CRF 23, but result in about half the file size. Other than that, CRF works just like the FFMPEG's options with h264 codec.

2) Choose a preset: the default is *medium*.

The preset determines how fast the encoding process will be at the expense of detail. Put differently, if you choose ultrafast, the encoding process is going to run fast, and the file size will be smaller (at expense of quality) when compared to medium. Slower presets use more memory. Valid presets are:

- ultrafast
- superfast
- veryfast
- faster
- fast
- medium
- slow
- slower
- veryslow
- placebo

Optionally you can choose a **`-tune`** option.

By default, this is disabled, and it is *generally not required* to set a tune option. The h265 codec supports the following **-tune** options:

`psnr`, `ssim`, `grain`, `zerolatency` and `fastdecode`.

The above options are discussed in the FFMPEG h264 guide.

For more presets and options discussions a good resource is available here: https://x265.readthedocs.io/en/default/presets.html

Basic FFMPEG formula for encoding a video in h265/HEVC format, using CRF method:

```
ffmpeg -i input -c:v libx265 -crf 28 -c:a
aac -b:a 128k output.mp4
```

This example uses AAC audio at 128 kBit/s.

This uses the native FFmpeg AAC encoder, but you may use the `libfdk_aac` option (requires `libfdk_acc` installed. Please refer to the chapter "How to inInstall FFMPEG".

Two-pass encoding

As the official FFMPEG documentations states, the two-pass encoding is generally used if you are targeting a specific output file size and output quality from frame to frame is of less importance.

This is best explained with an example.

Your video is 10 minutes (600 seconds) long and an output of 200 MiB (Mebibyte, which is near 210 Megabyte) is desired. Since bitrate = file size / duration:

(200 MiB * 8192 [converts MiB to kBit]) / 600 seconds = ~2730 kBit/s total bitrate

2730 - 128 kBit/s (desired audio bitrate) = 2602 kBit/s video bitrate

You can also forgot the bitrate calculation if you already know what final (average) bitrate you need.

For two-pass, you need to run ffmpeg twice, with almost the same settings, except for:

In pass 1 and 2, use the **`-x265-params pass=1`** and **`-x265-params pass=2`** options, respectively.

In pass 1, output to a null file descriptor[21], not an actual file. This will generate a logfile that ffmpeg needs for the second pass. In pass 1, you need to specify an output format (with **`-f`**) that matches the output format you will use in pass 2.

In pass 1, you can leave audio out by specifying **`-an`**.

For libx265, the **`-pass`** option (that you would use for libx264) is not applicable.

```
ffmpeg -y -i input -c:v libx265 -b:v 2600k
-x265-params pass=1 -an -f mp4 /dev/null &&
\
ffmpeg -i input -c:v libx265 -b:v 2600k
-x265-params pass=2 -c:a aac -b:a 128k
output.mp4
```

This example uses AAC audio at 128 kBit/s. This uses the native FFmpeg AAC encoder, but you may use the

`-c:a libfdk_aac` option (requires *libfdk_acc* installed.

Please refer to the chapter "How to inInstall FFMPEG".)

[21] https://medium.com/@codenameyau/step-by-step-breakdown-of-dev-null-a0f516f53158

As with CRF, choose the slowest **`-preset`** you can tolerate, and optionally apply a **`-tune`** setting.

Note that when using faster presets with the same target bitrate, the resulting quality will be lower and vice-versa.

For more options such as the Passing Option, Lossless encoding amd more, please refer to this section of the official FFMPEG Documentation.

h266 - Versatile Video Codec (VVC)

This standard approved in July 2020 has been created by Fraunhofer Heinrich Hertz Institute[22].

By their own words: "Through a reduction of data requirements, H.266/VVC makes video transmission in mobile networks (where data capacity is limited) more efficient. For instance, the previous standard H.265/HEVC requires ca. 10 gigabytes of data to transmit a 90-min UHD video. With this new technology, only 5 gigabytes of data are required to achieve the same quality.

Because H.266/VVC was developed with ultra-high-resolution video content in mind, the new standard is particularly beneficial when streaming 4K or 8K videos on a flat screen TV. Furthermore, H.266/VVC is ideal for all types of moving images: from high-resolution 360° video panoramas to screen sharing contents."

At the time of writing this book there are no H266 codec available for FFMPEG and they should be available in late 2020.

22 https://jvet.hhi.fraunhofer.de

Producing VP8 Videos

The WebM Project is dedicated to developing a high-quality, open video format for the web that's freely available to everyone.

WEBM's codec VP8 is installed on FFMPEG with the **libvpx** library, which is an open, royalty-free media file format.

While any recent FFMPEG repository should have this option enabled by default, the **libvpx** library can be installed with the help of the Compilation Guide and by compiling FFmpeg with the **`--enable-libvpx`** option, or by making sure that libvpx option it's enabled in the Homebrew's installation formula.

Note: The VP8 successor **VP9** provides better video quality at a lower bitrate.

In this chapter we'll explore commands for producing Variable Bitrate, Constant Bitrate and Alpha Channel videos.

Variable Bitrate (Default)

```
ffmpeg -i INPUT -c:v libvpx -b:v 1M -c:a
libvorbis OUTPUT.webm
```

Libvpx offers a variable bitrate mode by default.

In this mode, it will simply try to reach the specified bit rate on average, e.g. 1 MBit/s. This is the "target bitrate".

Choose a higher bit rate if you want better quality.

Note: you shouldn't leave out the **-b:v** option as the default settings will produce mediocre quality output.

Variable Bitrate with CRF (Reccomended)

In addition to the "default" VBR mode, there's a constant quality mode, like in the x264 encoder, that will ensure that every frame gets the number of bits it deserves to achieve a certain quality level, rather than forcing the stream to have an average bit rate. This results in better overall quality and should be your method of choice when you encode video with VP8. In this case, the target bitrate becomes the maximum allowed bitrate.

You enable the constant quality mode with the CRF parameter:

```
ffmpeg -i INPUT -c:v libvpx -crf 10
-b:v 1M -c:a libvorbis OUTPUT.webm
```

By default the CRF value can be from 4-63, and 10 is a good starting point. Lower values mean better quality[23].

Important: If neither **`-b:v`** nor **`-crf`** are set, the encoder will use a low default bitrate and your result will probably look very bad. Always supply one of these options *–ideally both*.

If you want to tweak the quality even further, you can set two more parameters:

`-qmin` - the minimum quantizer (default 4, range 0-63)

`-qmax` - the maximum quantizer (default 63, range `-qmin` up to 63).

"Quantize" is the process of converting a continuous range of values into a finite range of discreet values.

---

23 https://trac.ffmpeg.org/wiki/Encode/VP8

These "q values" are quantization parameters, and lower generally means "better quality". If you set the bounds from 0 to 63, this means the encoder has free choice of how to assign the quality. For a better overall quality, you can try to set **`-qmin`** to **`0`** and **`-qmax`** to **`50`** or lower.

For example:

```
ffmpeg -i INPUT -c:v libvpx -qmin 0 -qmax 50
-crf 5 -b:v 1M -c:a libvorbis OUTPUT.webm
```

As stated in the FFMPEG's documentation:

"Needless to say that this requires a bit of tweaking.

If in CRF mode you set the maximum bit rate too low, or the quality too high (i.e., low qmin, qmax or crf), libvpx will "saturate" the bitrate and you'll end up with constant bit rate encoding again, which is not ideal."

Constant Bitrate

Like most other encoders, libvpx offers a constant bitrate encoding mode as well, which tries to encode the video in such a way that an average bitrate is reached.

This doesn't mean that every frame is encoded with the same amount of bits (since it would harm quality), but the bitrate will be very constrained.

You should use constant bitrate encoding if you need your video files to have a certain size, or if you're streaming the videos over a channel that only allows a certain bit rate.[24]

Example Formula:

```
ffmpeg -i INPUT -c:v libvpx -minrate 1M
-maxrate 1M -b:v 1M -c:a libvorbis
OUTPUT.webm
```

Of course you can choose different values for the bitrate other than 1M, e.g. 500K, but you must set all options (i.e., **`-minrate`**, **`-maxrate`** and **`-b:v`**) to the same value.

Alpha Channel

VP8 includes alpha channel support that can be seen at http://simpl.info/videoalpha/ using Google Chrome or

[24] https://trac.ffmpeg.org/wiki/Encode/VP8

Firefox. To create a similar video with ffmpeg from a series of png input images:

```
ffmpeg -i %03d.png -c:v libvpx -pix_fmt
yuva420p -metadata:s:v:0 alpha_mode="1"
output.webm
```

To know more about VP8 Encode Parameters, please check the VP8 Encode Parameter Guide.

Producing VP9 videos

Created and developed by Google in 2013, VP9 is the direct competitor of h265, mainly used for its YouTube platform and the evolution of the VP8 codec.

To use VP9 with FFMPEG you will use the **`libvpx-vp9`** which is the VP9 video encoder for WebM, an open, royalty-free media file format. The complete description of VP9 is available at the Google's Developer VP9 section.

Google offers also full FFMPEG commands and reccomended settings specifically for Live Encoding, Video-On-Demand scenarios and for HDR scenarios.

Similarly to the h264 or h265 codecs, VP9 offers different rate control modes, which determine the quality and file size:

- 1-pass and 2-Pass Average bitrate
- Constant quality
- Constrained quality
- 2-pass constant quality
- Constant bitrate
- Lossless

1-Pass Average Bitrate

In this example you have just to set a desired birate (2M) and FFMPEG will take care of the rest.

```
ffmpeg -i input.mp4 -c:v libvpx-vp9 -b:v 2M 
output.webm
```

2-Pass Average Bitrate (Reccomended)

As the official FFMPEG's docuementation states:

"Two-pass is the recommended encoding method for libvpx-vp9 as some quality-enhancing encoder features are only available in 2-pass mode."

There are 2 distinct 2-Pass methods for producing VP9 videos with FFMPEG: a target bitrate method for targeting an average bitrate, and a costant quality one.

Again, as mentioned in the official FFMPEG documentation, the second method "*uses the more contemporary CRF-style approach for the final pass to achieve a certain perceptual quality level while still gaining the aforementioned compression benefits by also doing a first pass."*

For two-pass, you need to run FFMPEG twice, with almost the same settings, except for:

• In pass 1 and 2, use the **`-pass 1`** and **`-pass 2`** options, respectively.

• In pass 1, FFMPEG will output to a null file descriptor, not an actual file. (*This will generate a logfile that ffmpeg needs for the second pass*)

• In pass 1, you need to specify an output format, with **`-f,`** that matches the output format you will use in pass 2.

In pass 1, you can leave audio out by specifying **`-an`**. ("no audio" option.)

First Method: 2-Pass "Targeting Bitrate Mode"

Example Formula:

```
ffmpeg -i input.mp4 -c:v libvpx-vp9 -b:v 2M
-pass 1 -an -f webm /dev/null&&ffmpeg -i
input.mp4 -c:v libvpx-vp9 -b:v 2M -pass 2
-c:a libopus output.webm
```

Second Method: 2-Pass "Costant Quality"

Constant quality 2-pass is invoked by setting **`-b:v`** to zero and specifiying a quality level using the **`-crf`** switch:

Example Formula:

```
ffmpeg -i input.mp4 -c:v libvpx-vp9 -b:v 0
-crf 30 -pass 1 -an -f webm /dev/
null&&ffmpeg -i input.mp4 -c:v libvpx-vp9
-b:v 0 -crf 30 -pass 2 -c:a libopus
output.webm
```

Costant Quality

FFMPEG's VP9 gives you also a constant quality mode (CQ) which is similar to the CRF option in the h264 codec, that targets a certain perceptual quality level while only using a single pass.

As the FFMPEG's documentation states:

"*While using this single-pass mode will result in less effi cient compression due to libvpx's preference for 2-pass encoding, this mode may still be useful if the extra time required for the fi rst pass and the additional CPU cycles used for better compression in 2-pass mode aren't worth it for your use case.*"

To trigger this mode, you must use a combination of **`-crf`** and **`-b:v 0`**. Note that **`-b:v`** MUST be `0`.

Setting it to anything higher or omitting it entirely will instead invoke the Constrained Quality mode, described below.

Example Formula:

```
ffmpeg -i input.mp4 -c:v libvpx-vp9 -crf 30
-b:v 0 output.webm
```

The CRF value can be from 0-63. Lower values mean better quality. Recommended values range from 15-35, with 31 being recommended for 1080p HD video.

Google's Reccomended Settings for 2Pass VOD

Google's VP9 Developer website contains lots of useful information and FFMPEG formulas in order to produce VP9 videos for different case scenarios.

These recommendations are designed for the following goals:

- A balance between quality and encoding speed
- The minimal bit rate to achieve reasonable quality
- Settings to accommodate a wide range of content types

An example Formula taking those specs in consideration, for a 1080p video, but with an higher bitrate, would be:

```
ffmpeg -i INPUT -b:v 3600k -minrate 1800k
-maxrate 5220k -tile-columns 2 -g 240
-threads 8 -quality good -crf 27 -c:v
libvpx-vp9 -map 0 -sn -c:a libopus -ac 2
-b:a 128k -pass 1 -speed 4
OUTPUT.webm&&ffmpeg -i INPUT -b:v 3600k
-minrate 1800k -maxrate 5220k -tile-columns
3 -g 240 -threads 8 -quality good -crf 27
-c:v libvpx-vp9 -map 0 -sn -c:a libopus -ac
2 -b:a 128k -pass 2 -speed 4 -y OUTPUT.webm
```

Please refer to the official Google documentation to learn more about this command, or the WebM VOD Reccomendend Settings.

Constrained Quality Mode

With the constrained quality method VP9 will ensure that a constant (perceptual) quality is reached while keeping the bitrate below a specified upper bound or within a certain bound. While the caveats of single-pass encoding mentioned above stil apply, this method can still be useful for bulk encoding videos in a generally consistent fashion.

Example Formula:

```
ffmpeg -i input.mp4 -c:v libvpx-vp9 -crf 30
-b:v 2000k output.webm
```

The quality is determined by the **`-crf`** option, and the bitrate limit by the **`-b:v`** where the bitrate MUST be non-zero.

Constant Bitrate Mode

As FFMPEG's Docs states: "*Like most other encoders, libvpx offers a constant bitrate (CBR) encoding mode as well, which tries to encode the video in such a way that an average bitrate is reached. This doesn't mean that every frame is encoded with the same amount of bits (since it would harm quality), but the bitrate will be very constrained. You should* *use constant bitrate encoding if you need your video files to have a certain size, or if you're streaming the videos over a channel that only allows a certain bit rate. Generally though, using constrained quality is the recommended option in this case.*"

Example Formula:

```
ffmpeg -i input.mp4 -c:v libvpx-vp9 -minrate 
1M -maxrate 1M -b:v 1M output.webm
```

Here, you can choose different values for the bitrate other than 1M, e.g. 500K, but you must set all options (i.e., **`-minrate`**, **`-maxrate`** and **`-b:v`**) to the same value.

Lossless Mode

The libvpx-vp9 has a lossless encoding mode that can be activated using the option **`-lossless 1`**:

Example Formula:

```
fmpeg -i input.mp4 -c:v libvpx-vp9 -lossless 1 
output.webm
```

For additional informations and details about FFMPEG's usage of VP9, please refer to this section of the FFMPEG guide.

For specific options of VP9, please refer to this section of the FFMPEG's guide.

The OPUS Audio Codec

Opus is a totally open, royalty-free, highly versatile audio codec. Opus is unmatched for interactive speech and music transmission over the Internet, but is also intended for storage and streaming applications. It is standardized by the Internet Engineering Task Force (IETF) as RFC 6716[25] which incorporated technology from Skype's SILK codec and Xiph.Org's CELT codec[26].

Opus can handle a wide range of audio applications, including Voice over IP, Videoconferencing, In-Game Chat, and even Remote Live Music performances.

It can scale from low bitrate narrowband speech to very high quality stereo music.

Note that the default audio encoder for WebM, VP8 + VP9, is OPUS (**`libopus`**). If not installed, FFMPEG will use Vorbis (**`libvorbis`**) instead.

[25] A Request for Comments (RFC) is a publication from the Internet Society (ISOC) and its associated bodies, most prominently the Internet Engineering Task Force (IETF), the principal technical development and standards-setting bodies for the Internet.

[26] https://opus-codec.org/

To install OPUS, make sure to compile FFMPEG with the **`--enable-libopus`** option.

An example formula for converting from WAV file, or AIFF to a stereo Opus file is:

```
ffmpeg -i INPUT.aiff -c:a libopus -b:a 96k
-ac 2 OUTPUT.opus
```

An example formula to convert from Dolby 5.1 AC3 file format to Opus, preserving all the separate channels:

```
ffmpeg -i INPUT.ac3 -c:a libopus
-af channelmap=channel_layout=5.1
-b:a 96k OUTPUT.opus
```

Another example formula for converting a Dolby 5.1 track to a stereo Opus file will be:

```
ffmpeg -i INPUT.ac3 -c:a libopus
-ac 2 -b:a 96k OUTPUT.opus
```

Audio Parameters

If you want to increment the value of your bitrate, you can set the **`-b:a`** value to an higher one, such as 128k or above, up to 256k.

The same applies with the optional **`-ar`** value, which by default is set at 48 kHz. If you require an higher bitrate, such as 96 kHz or 192 kHz, then you should consider to use a Lossless audio codec, such as the FLAC audio codec, described in the next chapter.

For more informations about the OPUS codec, please refer to the OPUS Recommended Settings and the OPUS FAQ.

Batch Conversion

To convert an entire directory from WAV to OPUS you will need to write a simple bash script. Make sure to locate yourself into the working directory, meaning the directory where your have the original files, such as the example below:

```
cd ~/Desktop/WAV_FILES_DIRECTORY
```

Then just type:

```
nano opus.sh
```

Then type the following instructions:

```
mkdir OPUS
for i in *.wav;
  do name=`echo "$i" | cut -d'.' -f1`
  echo "$name"
  ffmpeg -i "$i" -c:a libopus -b:a 96k
"OPUS/${name}.opus"
done
```

Save your "opus.sh" file and exit from Nano (or your favourite Text editor), then run the bash script:

```
source opus.sh
```

FFMPEG will perform the batch conversion for every WAV file available in the working directory and will place the converted files into a directory named "OPUS".

To know more about OPUS audio format, visit the OPUS audio codec website and the OPUS FAQ website.

Embed OPUS Files in Safari / iOS Environments

To play or embed an OPUS file into Safari or into an iOS App you can simply change the .opus container into the .caf container (**C**ore **A**udio **F**ormat) using this simple formula:

```
ffmpeg -i YOUR_OPUS_FILE.opus -c:a copy 
OUTPUT.caf
```

This command will change the container of the file from OPUS to CAF without re-encoding the file and keeping the same file size. To know more about the CAF format you can check the Apple's Developer CAF File Overview website.

The FLAC Audio Codec

FLAC stands for Free Lossless Audio Codec, an audio format similar to MP3, but lossless, meaning that audio is compressed without any loss in quality and supports High Resolution Audio, or HD Audio.[27]

FFMPEG can convert your existing audio files into FLAC, using the syntax:

```
ffmpeg -i YOUR_AUDIO.wav -c:a flac 
OUTPUT.flac
```

For futher settings please refer to the General Usage section and the FAQ Section of the FLAC's website.

27 https://xiph.org/flac/

Producing AV1 Video

AV1 stands for **A**lliance for Open Media **V**ideo **1** (AV1) and it's a royalty-free and open source codec developed in alliance with major developers, such as Google, Adobe, Netflix, Cisco, Microsoft amongst others.

It's considered as a successor of VP9 codec, and used with the Opus Codec for audio and "packed" into a WebM container, so to be used in HTML5 video and used as a WebRTC standard. The WebRTC standard is manily used for real-time communications between browsers or Apps.

Amazon's AWS describes AV1 as a solution where "*you can deliver high-quality SD and HD video to mobile and other devices over congested or bandwidth-constrained networks at bitrates unachievable with traditional codecs.*"[28]

FFMPEG, of course, can handle the AV1 standard, with the **`libaom`** (encoder) and the VLC+FFMPEG's **`dav1d`** (decoder) libraries.

[28] https://aws.amazon.com/about-aws/whats-new/2020/03/av1-encoding-now-available-with-aws-elemental-mediaconvert/?nc1=h_ls

For video-on-demand cases, an exaple formula for converting a source video into AV1 could be:

```
ffmpeg -i YOUR_INPUT -c:v libaom-av1 -cpu-used 8 -crf 27 -b:v 0 -strict experimental -c:a libopus -b:a 128k  -movflags +faststart YOUR_OUTPUT.webm
```

To know more informations about AV1 encoding options, pleaser refer to the official FFMPEG AV1 Documentation.

Producing AV1 files can take really long, even of faster machines. With the 2020's FFMPEG 4.3 release, and later versions, the encoding times are better than in 2019, along with the latest LibAOM and Dav1d releases, as Jan Ozer pointed out, "AV1 encoding times dropped to near-reasonable levels."[29]

Please note also that AV1 will insert a very long key-frame interval by default (every 9999 frames): this will translates into a very slow, or impossible seeking.

To avoid this, you need to specify a max interval key-frame, using the **-g** (group of picture) option with a sane range (2 seconds or less.)

[29] https://www.streamingmedia.com/Articles/Editorial/Featured-Articles/Good-News-AV1-Encoding-Times-Drop-to-Near-Reasonable-Levels-130284.aspx

FFMPEG's official documentation suggests also to insert the **-movflags +faststart** instruction, if your video is destined for streaming purposes.

Note for audio: if you have an AC3 5.1 audio source, you migth need to add the

-af "channelmap=channel_layout=5.1" option into your command, such as the example below:

```
ffmpeg -i YOUR_SOURCE -vn -c:a libopus -b:a 
128k -af "channelmap=channel_layout=5.1" 
OUTPUT.opus
```

Netflix/Intel AV1 SVT-AV1

The SVT-AV1 is an open-source AV1 codec implementation hosted on GitHub[30].

A complete technical explanation of this codec is available on the Netflix's blog.

Defined as the *Scalable AV1 technology* this codec can be faster than the one used on the LibAOM library, especially on x86 (Intel) processors.

To install SVT-AV1 codec you will need to manually compile FFMPEG and install the codec thru the **`--enable-libsvtav1`** during installation, or by cloning the repository on GitHub and following the Installation process instructions.

An example formula will be:

```
ffmpeg -i [input.mp4] -nostdin -f rawvideo
-pix_fmt yuv420p - | ./SvtAv1EncApp -i stdin
-n [number_of_frames_to_encode] -w [width]
-h [height]
```

30 https://netflixtechblog.com/svt-av1-an-open-source-av1-encoder-and-decoder-ad295d9b5ca2

AV1AN - All-in-one Tool

AV1AN is a Python tool defined as "*an easy way to start using AV1 / VP9 / VP8 encoding. AOM, rav1e, SVT-AV1, VPX are supported.*"[31]

This tool aims to solve a current problem in AV1 encoding times by using "spliting video by scenes" for parallel encoding, because AV1 encoders currently are not good at multithreading and encoding is limited to single or couple of threads at the same time.

To use and install AV1AN please refer to the official AV1AN documentation, available here.

An example formula will be:

```
av1an -i [YOUR_INPUT] -enc aom -v " --cpu-used=3 --end-usage=q --cq-level=30 --threads=8 " -w 10 -p 2
 -ff " -vf scale=-1:720 "  -a " -c:a libopus -b:a 24k " -s scenes.csv -log my_log -o [YOUR_OUTPUT]
```

[31] https://pypi.org/project/Av1an/#description

Streaming on Social Media with RTMP

The RTMP abbreviation stands for "**R**eal-**T**ime-**M**essaging-**P**rotocol" and RTMPS stands for the same, but over a TLS, **T**ransport **L**ayer **S**ecurity[32], or a SSL[33] connection.

RTMP has been originally created by Macromedia for enabling audio and video streaming between Flash Servers. Now Adobe, which bought Macromedia, has opened this proprietary protocol for public use.

In 2020 it's still the standard format for broadcasting live videos on Facebook, Instagram, YouTube, Twitch, Twitter's Periscope and many others platforms.

FFMPEG can stream to an RTMP/RTMPS server, without any particular problem. For RTMPS protocol you must have installed Open SSL, which is already installed with the Brew Installation of FFMPEG on MacOSX and Ubuntu's Snap package, as described earlier.

---

[32] Transport Layer Security, or TLS, is a widely adopted security protocol designed to facilitate privacy and data security for communications over the Internet.

[33] SSL stands for Secure Sockets Layer and, in short, it's the standard technology for keeping an internet connection secure and safeguarding any sensitive data that is being sent between two systems, preventing criminals from reading and modifying any information transferred, including potential personal details.

Streaming Pre-Recorded Content

There are many methods to stream a pre-recorded video to a RTMP server and in this book we'll explore 2 of them. The first way is to take your input "as-is" and convert it in real-time to FLV, which is the standard format for the RTMP protocol.
This methods uses lots of CPU, because it will process your input in real-time in order to convert it and stream it.

For live streaming or VOD (Video-On-Demand) services you will need to produce streams that follows a specific "keyframe interval", meaning that the interval between one I-frame and another must adhere to a specific value, so to give the best possible user experience to your viewers.

For example: YouTube Live requires a keyframe frequency of 2 seconds, not exceeding 4 seconds.

Facebook Live requires 2 seconds keyframe interval.

Twitch also requires 2 seconds keyframe interval.

The official FFMPEG documentation explains furthermore the different case scenarios for Live Streaming and Video-On-Demand.

Streaming for YouTube Live

As mentioned earlier, YouTube Live has some requirements in order to stream your signals on its platform. For HD 1080p streams, in example, you will need to adhere at the following requirements:

- **Resolution**: 1920x1080
- **Video Bitrate Range**: 3.000-6.000 Kbps
- **Video Codec**: H.264, Level 4.1 for up to 1080p 30 FPS or Level 4.2 for 60 FPS
- **Keyframe Frequency**: 2 seconds
- **Audio Codec**: AAC or Mp3
- **Bitrate Encoding**: CBR (Constant Bit Rate)

With all of that in mind, assuming to have a Full-HD 1080p 25 FPS video to be broadcasted Live on YouTube, an example command will be:

```
ffmpeg -re -i INPUT.mp4 -c:v libx264
-profile:v high -level:v 4.1 -preset
veryfast -b:v 3000k -maxrate 3000k -bufsize
6000k -pix_fmt yuv420p -g 50 -keyint_min 50
-sc_threshold 0 -c:a aac -b:a 128k -ac 2 -ar
44100 -f flv "rtmp://a.rtmp.youtube.com/
live2/YOUR_STREAMING_KEY"
```

Let's break down the above command:

`ffmpeg -re -i input.mp4`: This is the input command. The **`-re`** option (before the **`-i`**) will instruct FFMPEG to read the source file at its native frame rate.

This slows the stream down to simulate live streaming and mitigates buffering and memory buildup that can disrupt playback

`-c:v libx264 -profile:v high -level:v 4.1 -preset veryfast`: This will instruct FFMPEG to use the libx264 for producing an optimized h264 file, with the "High Profile" setting, which is the primary profile for broadcast and disc storage applications, particularly for high-definition television applications, adopted also for HD DVD and Blu-ray Disc, and level 4.1 as explained here, and by using the veryfast preset.

`-b:v 3000k -maxrate 3000k -bufsize 6000k`: this will instruct FFMPEG to produce a file that stays in the 3.000-6000 video bitrate range required by YouTube. More specifically the **`-b:v`** option specifies the video bitrate (you can either type 3000k or 3M), **`-maxrate`** stands for the maximum bitrate to stream and the **`-bufsize`** (buffer size) option will calculate and correct the average bit rate produced.

`-pix_fmt yuv420p`: This will instruct FFMPEG to use a specific chroma subsampling scheme named 4:2:0 planar. This instruction is specified for compatibility reasons, since your output must be playable across different players and platforms. *Chroma subsampling* is the practice of encoding images by implementing less resolution for chroma information[34], than for luma information[35], taking advantage of the human visual system's lower acuity for color differences than for luminance[36].

"YUV" is a type of video signal that consists of three separate signals: 1 for luminance (brightness) and two for chrominance (colours).

To know more about Chroma subsampling, please refer to this VLC Resource.

`-g 50`: In order to abide to the required 2 second keyframe interval, this will set a value of 50 GOP, Group of Pictures. This value of "50" is just an example.
To know the exact value for your case, simply multiply your output frame rate * 2. For example, if your original input runs at 24 FPS, then you will use **`-g 48`**.

[34] the degree of vividness of a color, or how pure it is compared to its representative on the color wheel

[35] the brightness in an image

[36] S. Winkler, C. J. van den Branden Lambrecht, and M. Kunt (2001). "Vision and Video: Models and Applications". In Christian J. van den Branden Lambrecht (ed.). Vision models and applications to image and video processing. Springer. p. 209. ISBN 978-0-7923-7422-0.

`-keyint_min value 50:` this will specify the minimum distance between I-frames and must be same as the **`-g`** value

`-sc_threshold 0`: This stands for **S**cene **C**hange **T**hreshold. FFmpeg defaults to a keyframe interval of 250 frames and inserts keyframes at scene changes. This option will make sure to not adding any new keyframe when the content of a picture changes.

`-c:a aac -b:a 160k -ac 2 -ar 44100`: with this instructions you will make FFMPEG encode the audio using the built-in **A**dvanced **A**udio **C**odec (**`-c:a aac`**), with a bitrate of 160 Kbps (**`-b:a 160k`**), in stereo ("**`-ac 2`**" stands for Stereo), with an audio sampling rate of 44.100 kHz (**`-ar 44100`**).

`-f flv rtmp://a.rtmp.youtube.com/live2/[YOUR_STREAM_KEY]:` this will instruct FFMPEG to output everything into the required FLV format to the YouTube RTMP server.

Please note that the above mentioned command will perform a real-time conversion of your input source.

If your machine is slow and just can't handle the real-time encoding process, such in the case of Full-HD videos or 4K or above, you can always pre-process the file before streaming it.

You have simply to create a file with the same command as above, but with a different destination
(a file), such as in this example for a Full-HD 1080p/25fps video:

```
ffmpeg -i INPUT.mp4 -c:v libx264 -profile:v
high -level:v 4.1 -preset veryfast -b:v
3000k -maxrate 3000k -bufsize 6000k -pix_fmt
yuv420p -g 50 -keyint_min 50 -sc_threshold 0
-c:a aac -b:a 128k -ac 2 -ar 44100
OUTPUT.mp4
```

The preset can also be adjusted to a slower one, in order to have a better compression.

Then you just need to "stream-copy" the file to the YouTube or to another RTMP destination, such as in this example formula:

```
ffmpeg -re -i OUTPUT.mp4 -c copy -f flv
"rtmp://a.rtmp.youtube.com/live2/
YOUR_STREAMING_KEY"
```

Loop pre-processed video

To make an infinite loop of a pre-processed video you can simply type:

```
ffmpeg -re -stream_loop -1
-i OUTPUT.mp4 -c copy -f flv "rtmp://
a.rtmp.youtube.com/live2/YOUR_STREAMING_KEY"
```

Loop a definite number of times (example 4 times)

```
ffmpeg -re -stream_loop 4
-i OUTPUT.mp4 -c copy -f flv "rtmp://
a.rtmp.youtube.com/live2/YOUR_STREAMING_KEY"
```

Each major social media platform has a technical guide, in order to produce the best experience for final users.

YouTube Live Specs:

https://support.google.com/youtube/answer/2853702?hl=en

Facebook Live Specs:

https://www.facebook.com/business/help/1625401111070395?id=1123223941353904

Twitter's Periscope Live Specs:

https://help.twitter.com/content/dam/help-twitter/periscope/periscopeproducer.pdf

Twitch Live Specs:

https://stream.twitch.tv/encoding/

Pre-Process Files in Batch

In case of large volumes of videos you might want to automate the pre-process action described previously.

In order to do so, you can type one single command in your Terminal, within your working directory:

```
for i in *.mov; do ffmpeg -i "$i" -c:v 
libx264 -profile:v high -level:v 4.1 -preset 
veryfast -b:v 3000k -maxrate 3000k -bufsize 
6000k -pix_fmt yuv420p -r 25 -g 50 
-keyint_min 50 -sc_threshold 0 -c:a aac -b:a 
128k -ac 2 -ar 44100 "${i%.*}.mp4"; done
```

The above example script will pre-process every *.mov* file contained in the directory, in line with the YouTube's requirements for streaming a Full-HD 1080p at 25FPS.

If your source inputs have the same extension (e.g.: `.mp4`) you can use the following script, that will add the suffix **`-converted`** to your output file:

```
for i in *.mp4; do ffmpeg -i "$i" -c:v
libx264 -profile:v high -level:v 4.1 -preset
veryfast -b:v 3000k -maxrate 3000k -bufsize
6000k -pix_fmt yuv420p -r 25 -g 50
-keyint_min 50 -sc_threshold 0 -c:a aac -b:a
128k -ac 2 -ar 44100 "${i%.*}-
converted.mp4"; done
```

Re-Stream to multiple destinations

FFMPEG offers the `-tee` option, which can be used to write the same data to several outputs, such as files or streams. It can be used also, for example, to stream a video over a network and save it to disk at the same time.

As stated in the official FFMPEG manual: "*the* **`-tee`** *option is different from specifying several outputs to the ffmpeg command-line tool. With the tee muxer, the audio and video data will be encoded only once.*"

Example Formula:

```
ffmpeg -i INPUT -c:v h264 -c:a aac -f tee
-map 0:v -map 0:a "output.mkv|
[f=mpegts]udp://10.0.1.255:1234/"
```

A complete description of this option can be found on the "Creating Multiple Outputs" section of the official FFMPEG's guide.

Concatenate Video Playlists

In this chapter we'll discover how to concatenate multiple videos in order to stream a playlist of different content to a social media destination such as YouTube Live, Facebook Live, Twitch, Twitter's Periscope, etc.

The process requires 4 separate steps, so to ensure maximum compatibility with the technical requirements of most of the major streaming platforms.

The 4 steps are:

1) Conform your files to the platform's tech specs;

2) Segment the files in HLS format with Bento4;

3) Create a playlist in M3U8 format;

4) Stream the M3U8 playlist to the RTMP server;

Please note: usually after 24 hours (or less), the FFMPEG streaming process could be interrupted by a "Broken Pipe"[37] error. This can happen if you stream directly to the RTMP address provided by YouTube by or any other major platform. From my direct experience if you stream your signal to a video server such as Nimble Streamer or Wowza, and then from there you re-stream the signal to the YouTube/FB/Periscope RTMP server, you will be able to fix such "Broken Pipe" error and obtain a non-stop 24/7 streaming signal.

With that being said, let's explore the 4 steps required.

1) Pre-Process Your Files

To pre-process the files we'll take as an example the 1080p, 25 FPS requirements of YouTube:

- **Resolution**: 1920x1080
- **Video Bitrate Range**: 3.000-6.000 Kbps
- **Video Codec**: H.264, Level 4.1 for up to 1080p 30 FPS or Level 4.2 for 60 FPS

[37] A pipe is a data stream, typically data being read from a file or from a network socket. A broken pipe occurs when this pipe is suddenly closed from the other end. For a flie, this could be if the file is mounted on a disc or a remote network which has become disconnected.

- **Keyframe Frequency**: 2 seconds
- **Audio Codec**: AAC or Mp3
- **Bitrate Encoding**: CBR (Constant Bit Rate)

The command to produce a file with the above mentioned specs will be as follow:

```
ffmpeg -i YOUR_INPUT -c:v libx264 -preset
veryfast -b:v 3000k -maxrate 3000k -bufsize
6000k -pix_fmt yuv420p -g 50 -keyint_min 50
-sc_threshold 0 -c:a aac -b:a 128k -ac 2 -ar
44100 OUTPUT.mp4
```

The above command will assume that you will have a 1920x1080p/25 FPS input video.

2) Segment and convert files to HLS with Bento4

Bento4 is a set of tools designed to produce standard HLS and DASH packages. HLS stands for Apple HTTP Live Streaming, while DASH, Dynamic Adaptive Streaming over HTTP, is the open-source alternative.

In addition to supporting ISO-MP4, Bento4 includes support for parsing and multiplexing H.264 and H.265 elementary streams, converting ISO-MP4 to MPEG2-TS, packaging HLS and MPEG-DASH, content encryption, decryption, and much more.

To install Bento4 on Mac, Windows and Linux platform please refer to the official Bento4 website:

https://www.bento4.com/downloads/

To segment your file with Bento4 you will use the following formula for every video pre-processed during Step 1:

```
mp42hls --output-single-file OUTPUT.mp4
```

This command will store segment data in a single output file per input file, without producing tens or hundreds of segments.

This command will also create 3 distinct files:

1) The **`stream.ts`** file which is the segmented file itself, in MPEG Transport Stream format (`.ts`);
2) The **`stream.m3u8`** file, which is the playlist of the segments;
3) A **`iframe.m3u8`** file, which is not required for this specific purpose, but required for VOD development purposes.

To test the **`stream.m3u8`** file on YouTube Live, just type the following command:

```
ffmpeg -re -i stream.m3u8 -c copy -f flv
"rtmp://a.rtmp.youtube.com/live2/
YOUR_STREAMING_KEY"
```

3) Create your Playlist in M3U8 Format

Once you will have conformed and segmented your videos, you will need to create a text file in order to create your desired playlist of videos to be broadcasted live.

In order to do so, just open your favourite text editor, or as in this example the Nano text editor and type the following:

```
nano playlist.txt
```

Then:

```
file 'video1/stream.m3u8'
file 'video2/stream.m3u8'
file 'video3/stream.m3u8'
file 'video4/stream.m3u8'
file 'video5/stream.m3u8'
```

And so forth, depending on how many videos you will need to concatenate.

4) Stream your playlist

Once you have created your playlist in the desired order of reproduction, you will be able to stream it by using the following formula:

```
ffmpeg -re -f concat -i playlist.txt -c copy
-f flv "rtmp://a.rtmp.youtube.com/live2/
YOUR_STREAMING_KEY"
```

Optionally you can loop infinitely your playlist with the instruction **`-stream_loop -1`** before the **`-i playlist.txt`** section, or specify a number based on how many loop you want, with the **`-stream_loop`** [DESIRED NUMBER OF TIMES].

Producing HLS with FFMPEG and Bento4

With FFMPEG and Bento4[38] your can create standard HLS streaming formats.

HLS it's a standard created by Apple[39] in order to deliver Audio/Video streamings using ordinary web servers and content delivery networks (CDNs), for Apple systems, such as iPhone, iPad, Mac and Apple TV.

HLS supports the following:

- Live broadcasts and pre-recorded content (VOD)
- Multiple alternate streams at different bit rates
- Intelligent switching of streams in response to network bandwidth changes
- Media encryption and user authentication

In order to prepare a video into a standard HLS stream, you will need both FFMPEG and Bento4 installed.

38 https://www.bento4.com

39 https://developer.apple.com/documentation/http_live_streaming

To install Bento4 for MacOSX, Windows or Linux, you can follow the instruction on the Bento4's Official Website or, if your are using Brew, you can open your Terminal and type:

```
brew install bento4
```

Once installed both FFMPEG and Bento4 you can prepare your master file. In the next examples I will assume to have an h264 file inside an mp4 container, with audio in AAC format.

Convert your Master into multi-bitrate with FFMPEG

To begin, you will have 1 master file that will need to be converted in different bitrates.

Let's say that your master is in Full-HD 1080p (1920x1080). For your streaming needs, let's say that you will need to deliver also the 720p (1280x720), the 480 (854x480) and the 240p (426x240) versions of the same file.

The following formula will produce a multi-bitrate version of your master, with h264 Constrained Encoding, as described earlier in the Chapter "***Different h264 encoding approaches***":

```
ffmpeg -i MASTER.mp4 -c:v h264 -crf 22 -tune film
-profile:v main -level:v 4.0 -maxrate 5000k
-bufsize 10000k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac
2 -pix_fmt yuv420p -movflags +faststart 1080.mp4
\
-s 1280x720 -c:v h264 -crf 24 -tune film
-profile:v main -level:v 4.0 -maxrate 2500k
-bufsize 5000k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac
2 -pix_fmt yuv420p -movflags +faststart 720.mp4 \
-s 854x480 -c:v h264 -crf 30 -tune film
-profile:v main -level:v 4.0 -maxrate 1250k
-bufsize 2500k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2
-pix_fmt yuv420p -movflags +faststart 480.mp4 \
-s 640x360 -c:v h264 -crf 33 -tune film
-profile:v main -level:v 4.0 -maxrate 900k
-bufsize 1800k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2
-pix_fmt yuv420p -movflags +faststart 360.mp4 \
-s 320x240 -c:v h264 -crf 36 -tune film
-profile:v main -level:v 4.0 -maxrate 625k
-bufsize 1250k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 22050 -b:a 64k -ac 1
-pix_fmt yuv420p -movflags +faststart 240.mp4
```

Let's breakdown this command:

`ffmpeg -i MASTER.mp4`: this will instruct to open FFMPEG and process a MASTER input's file called "MASTER.mp4".

`-c:v h264`: this will instruct FFMPEG to encode your MASTER video with the h264 codec

`-crf 22`: this will instruct FFMPEG to use a Constant Rate Factor of 20, which is a default value in order to produce a good quality video. The value can be anything from 0 (lossless) up to 50 (lowest).

`-tune film`: this option will instruct FFMPEG to use a specific h264 option, especially suitable for film content

`-profile:v main -level 4.0`: this will instruct FFMPEG to use a specific option of the h264 to compress frames and for compatibility reasons

`-maxrate 5000k -bufsize 10000k`: Along with the `-crf` option enabled this will instruct FFMPEG to encode your video with a specific constrained mode, as described in the previous chapter of this book "***Different h264 encoding approaches***"

`-r 25 -keyint_min 25 -g 50`: this instruction is for producing a 25 FPS file with the reccomended distance between keyframes every 2 seconds, assuming that your input is running at 25FPS, as explained in the previous chapter "***Keyframes: Basic Concepts***". You can change these values, accordingly to the Frame-Rate-Per-Second of your master input file.

`-sc_threshold 0`: this will instruct FFMPEG to not insert keyframes at scene changes, as explained in the chapter "***Keyframes: Basic Concepts***".

`-c:a aac -ar 44100 -b:a 128k -ac 2`: this will produce a standard AAC audio file, at 44.100 kHz, at 128 Kpbs, in stereo

`-pix_fmt yuv420p`: this option will produce a video compatible with major video players, including Apple's Quicktime.

`-movflags +faststart`: The `-movflags +faststart` instruction is advised whenever there is a streaming case-scenario. This particular instruction tells FFMPEG to move a specific part of an Mp4 File, called "atom", at the very beginning of the file. A full explanation of this mechanism is described in the chapter "***Streaming Mp4 Files - The Moov Atom***".

`1080.mp4`: this is the output file's name. I suggest you to name your files accordingly to the output definition, so to better recognize it later, for the Bento4 processing described below.

`\`: the backslash symbol means "Escape"[40], and using FFMPEG's syntax, it is used also as a new line delimiter[41].

---

40 https://ffmpeg.org/ffmpeg-utils.html#toc-Quoting-and-escaping

41 https://trac.ffmpeg.org/wiki/Creating%20multiple%20outputs

`-s 1280x720:` this will instruct FFMPEG to resize the `MASTER.mp4` input file, into the desired smaller size.

Accordingly to the above example, every new instruction after the **`-s`** value will produce half quality, half bitrate and smaller sizes, in parallel, so to have 4 different files for 4 different resolutions, in order to have:

`1080.mp4` – For the 1920x1080 resolution

`720.mp4` – For the 1280x720 resolution

`480.mp4` – For the 854x480 resolution

`360.mp4` – For the 640x360 resolution

`240.mp4` – For the 426x240 resolution

Please note that in the above formula, also the audio bitrate and quality will be adjusted accordingly to the version:

`1080.mp4` - AAC Audio at 128 Kbps, 44.100 kHz, Stereo

`720.mp4` - same as 1080.mp4

`480.mp4` - AAC Audio at 96 Kbps, 44.100 kHz, Stereo

`360.mp4` - same as 480.mp4

`240.mp4` - AAC Audio at 64 Kbpx, 22.050 kHz, Mono

At this point we are ready to use Bento4, so to create the HLS streaming for this 4 multi-bitrate files. Bento4 will segment each file and will produce a standard HLS playlist, according to the Apple's Technical Specification[42].

Create Multi-Bitrate HLS Playlist with Bento4

In order to create a multi-bitrate HLS Playlist with Bento4, using our above example, you will type the following formula:

```
mp4hls 240.mp4 360.mp4 480.mp4 720.mp4 
1080.mp4 --verbose --segment-duration 6 --
output-single-file
```

Let's break it down:

`mp4hls`: this will launch Bento4's Mp4HLS tool

`240.mp4 360.mp4 480.mp4 720.mp4 1080.mp4`: this will instruct Bento4 to process these 5 input files

42 https://tools.ietf.org/html/draft-pantos-hls-rfc8216bis-07

`--verbose`: this will enable a verbose mode, meaning that the progress of the process, including any error, will be displayed on your Terminal window

`--segment-duration 6`: this will produce standard 6 seconds segment, as per Apple's Technical Requirements[43]

`--output-single-file`: This will store segment data in a single output file per input file, thus reducing drastically the number of segmented files, to just 1.

The entire command will create a directory named "output" inside your working directory, containing all the HLS Playlists and media files, accordingly to the number of bitrate versions of your master video, previously processed with FFMPEG.

The whole directory can be then uploaded onto your webserver, so to stream and deliver your desired HLS playlists.

43 https://developer.apple.com/documentation/http_live_streaming/hls_authoring_specification_for_apple_devices

Producing DASH Streaming

Dash is the open-source equivalent of Apple's HLS technology, used for the same purpose: to deliver live videos or on-demand videos over the web, dynamically changing the quality based on the internet connection of the final user.

DASH stands for "**D**ynamic **A**daptive **S**treaming over **H**TTP".

In this chapter will go thru the same procedure of producing HLS, but with some slight modifications, in order to produce standard DASH fragments and playlists.

The same h264 codec settings applies in this case, and we'll also use Bento4 again, in order to use the DASH technology.

As discussed earlier in the chapter "Producing HLS with FFMPEG and Bento4", we will use the same formula, in order to produce a multi-bitrate version of your master file.

The following formula will produce a multi-bitrate version of your master, with h264 Constrained Encoding, as described earlier in the Chapter "***Different h264 encoding approaches***":

```
ffmpeg -i MASTER.mp4 -c:v h264 -crf 22 -tune film
-profile:v main -level:v 4.0 -maxrate 5000k
-bufsize 10000k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac
2 -pix_fmt yuv420p -movflags +faststart 1080.mp4
\
-s 1280x720 -c:v h264 -crf 24 -tune film
-profile:v main -level:v 4.0 -maxrate 2500k
-bufsize 5000k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac
2 -pix_fmt yuv420p -movflags +faststart 720.mp4 \
-s 854x480 -c:v h264 -crf 30 -tune film
-profile:v main -level:v 4.0 -maxrate 1250k
-bufsize 2500k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2
-pix_fmt yuv420p -movflags +faststart 480.mp4 \
-s 640x360 -c:v h264 -crf 33 -tune film
-profile:v main -level:v 4.0 -maxrate 900k
-bufsize 1800k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2
-pix_fmt yuv420p -movflags +faststart 360.mp4 \
-s 320x240 -c:v h264 -crf 36 -tune film
-profile:v main -level:v 4.0 -maxrate 625k
-bufsize 1250k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 22050 -b:a 64k -ac 1
-pix_fmt yuv420p -movflags +faststart 240.mp4
```

Create Multi-Bitrate DASH Playlist with Bento4

In order to create a multi-bitrate DASH Playlist with Bento4, using our above example, you will need to perform 2 separate actions. A first one dedicated at the fragmentation of each file (version) of your master video, using Bento4's utility "Mp4Fragment" with the following command:

```
mp4fragment 1080.mp4 1080-f.mp4
```

Let's break it down:

`mp4fragment`: this will launch Bento4's Mp4 Fragment tool

`1080.mp4`: this is the input file to be fragmented

`1080-f.mp4`: this is an example output's filename

Make sure to repeat this step with each variation of your master video, such in the example formula below:

```
mp4fragment 1080.mp4 1080-f.mp4&&mp4fragment
720.mp4 720-f.mp4&&mp4fragment 480.mp4 480-
f-.mp4&&mp4fragment 360.mp4 360-
f.mp4&&mp4fragment 240.mp4 240-f.mp4
```

Generate DASH Playlist

In order to generate a DASH Playlist with your segmented Mp4s, you will use another tool from Bento4 called "Mp4Dash", using the following command:

```
mp4dash 240-f.mp4 360-f.mp4 480-f.mp4 720-
f.mp4 1080-f.mp4
```

This will generate a folder titled "`output`" which will contain 2 separate folders: one called "`audio`" dedicated for the audio streaming and another called "`video`" containing the segments of each of the 5 variations of your master video. And, finally, you will find the "`stream.mpd`" file which is the DASH playlist itself.

Batch Processing for DASH and HLS Delivery

As discussed in the previous chapter, DASH can deliver content in multi-bitrate resolutions, adapting its quality based on the final user's bandwidth.

In case you will need to process tens or tens of thousands of videos you will surely need to automate this process, using BASH, as further discussed in the next chapter dedicated to the HLS Batch Processing.

To automate this process of multi-bitrate production from your masters, along with the segmentation and creation of DASH-Compliant MPD, or Media Presentation Descritpion Playlists[44], along with Apple's HLS Playlist in M3U8 format, you can create a script with you favourite text editor, such in the example fromula below, using Nano text editor.

First of all, let's create a new script called "`mpd.sh`":

```
nano mpd.sh
```

[44] A media presentation description (MPD) file is used to hold the information on the various streams and the bandwidths they are associated with.

Inside the Nano Text Editor you can type the following BASH script:

```
#! /bin/bash

for file in *.mp4;
do
mkdir "${file%.*}"
ffmpeg -i "$file" -c:v h264 -crf 22 -tune film -profile:v main
-level:v 4.0 -maxrate 5000k -bufsize 10000k -r 25 -keyint_min 25 -g
50 -sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac 2 -pix_fmt
yuv420p -movflags +faststart "${file%.*}/${file%.*}_1080.mp4" \
-s 1280x720 -c:v h264 -crf 24 -tune film -profile:v main -level:v
4.0 -maxrate 2500k -bufsize 5000k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac 2 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_720.mp4" \
-s 854x480 -c:v h264 -crf 30 -tune film -profile:v main -level:v
4.0 -maxrate 1250k -bufsize 2500k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_480.mp4" \
-s 640x360 -c:v h264 -crf 33 -tune film -profile:v main -level:v
4.0 -maxrate 900k -bufsize 1800k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_360.mp4" \
-s 320x240 -c:v h264 -crf 36 -tune film -profile:v main -level:v
4.0 -maxrate 625k -bufsize 1250k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 22050 -b:a 64k -ac 1 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_240.mp4"
mp4fragment "${file%.*}/${file%.*}_1080".mp4 "${file%.*}/${file%.*}
_1080-f".mp4
mp4fragment "${file%.*}/${file%.*}_720".mp4 "${file%.*}/${file%.*}
_720-f".mp4
mp4fragment "${file%.*}/${file%.*}_480".mp4 "${file%.*}/${file%.*}
_480-f".mp4
mp4fragment "${file%.*}/${file%.*}_360".mp4 "${file%.*}/${file%.*}
_360-f".mp4
mp4fragment "${file%.*}/${file%.*}_240".mp4 "${file%.*}/${file%.*}
_240-f".mp4
mp4dash --hls --output-dir="${file%.*}/${file%.*}_output" --use-
segment-list "${file%.*}/${file%.*}_1080-f".mp4 "${file%.*}/$
{file%.*}_720-f".mp4 "${file%.*}/${file%.*}_480-f".mp4 "${file%.*}/
${file%.*}_360-f".mp4 "${file%.*}/${file%.*}_240-f".mp4
mv "$file" "${file%.*}"
done
```

Full explanation of the script is available in the "Additional Notes and Syntax Definitions" chapter.

The BASH script will create:

• 5 versions of your Master video, at 25 FPS, in parallel;

• A folder containg each audio and video segments;

• DASH and HLS Playlists in `.mpd` and `.m3u8` format respectively for Non-Apple devices support and for iOS/MacOS devices;

The script will also move your original file in the respective directory, titled with the name of your master file.

You can then upload the entire directory to your own server, so to get ready for streaming your own content in both DASH and HLS formats.

Batch Processing for HLS Only

Let's say that you need to convert lots of videos into multi-bitrate versions for your own VOD platform, using HLS technology.

The following script will help you in accomplish just that, with the h264 constrained encoding method, as discussed in the previous chapter "***Different h264 encoding approaches***".

The formula will also create two new folder within your working directory: one titled as your input file, containing the various resolutions of your master file, and another one called "`{name_of_your_file_}_output`" containing the HLS segments and playlists ready to be streamed.

Please note: the script will assume that all of your input files are in Full-HD (1920x1080) and at 25 FPS. If your files have different FPS, you will need to convert them before running this script or by changing the `-r 25 -keyint_min 25 -g 50` sections of the script, accordingly to your input's FPS.

First of all, create a text file named as you will like, for example "`hls.sh`", with your favourite Text editor.

In this example I will use Nano:

```
nano hls.sh
```

Once in Nano, you can type the following BASH script:

```
#! /bin/bash

for file in *.mp4;
do
mkdir "${file%.*}"
ffmpeg -i "$file" -c:v h264 -crf 22 -tune film -profile:v main
-level:v 4.0 -maxrate 5000k -bufsize 10000k -r 25 -keyint_min 25 -g
50 -sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac 2 -pix_fmt
yuv420p -movflags +faststart "${file%.*}/${file%.*}_1080.mp4" \
-s 1280x720 -c:v h264 -crf 24 -tune film -profile:v main -level:v
4.0 -maxrate 2500k -bufsize 5000k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 128k -ac 2 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_720.mp4" \
-s 854x480 -c:v h264 -crf 30 -tune film -profile:v main -level:v
4.0 -maxrate 1250k -bufsize 2500k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_480.mp4" \
-s 640x360 -c:v h264 -crf 33 -tune film -profile:v main -level:v
4.0 -maxrate 900k -bufsize 1800k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 44100 -b:a 96k -ac 2 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_360.mp4" \
-s 320x240 -c:v h264 -crf 36 -tune film -profile:v main -level:v
4.0 -maxrate 625k -bufsize 1250k -r 25 -keyint_min 25 -g 50
-sc_threshold 0 -c:a aac -ar 22050 -b:a 64k -ac 1 -pix_fmt yuv420p
-movflags +faststart "${file%.*}/${file%.*}_240.mp4"
mp4hls --output-dir="${file%.*}/${file%.*}_output" "${file%.*}/$
{file%.*}_1080.mp4" "${file%.*}/${file%.*}_720.mp4" "${file%.*}/$
{file%.*}_480.mp4" "${file%.*}/${file%.*}_360.mp4" "${file%.*}/$
{file%.*}_240.mp4" --verbose --segment-duration 6 --output-single-
file
mv "$file" "${file%.*}"
done
```

Save and exit.

Then you can run the script by typing:

```
source hls.sh
```

The BASH script will produce the following 5 versions of your master files:

File Name	Size	Audio Codec	Audio Resolution	Audio Channels
{name-of-the-file}_1080.mp4	1920x1080	AAC	128 Kbps, 44.100 kHz	Stereo
{name-of-the-file}_720.mp4	1280x720	AAC	128 Kbps, 44.100 kHz	Stereo
{name-of-the-file}_480.mp4	854x480	AAC	128 Kbps, 44.100 kHz	Stereo
{name-of-the-file}_360.mp4	640x360	AAC	128 Kbps, 22.050 kHz	Stereo
{name-of-the-file}_240.mp4	426x240	AAC	128 Kbps, 22.050 kHz	Mono

And will produce the related HLS Playlists and segmented media, within the "`output`" folder generated by Bento4.

Streaming Mp4 Files - The Moov Atom

MP4 files consist of chunks of data called *atoms*[45].

There are atoms to store things like subtitles or chapters, as well as obvious things like the video and audio data. Metadata about where the video and audio atoms are, as well as information about how to play the video like the dimensions and frames per second, is all stored in a special atom called the `moov atom`.

You can think of the moov atom as a kind of table of contents for the MP4 file.

When you play a video, your video player looks through the MP4 file, locates the moov atom, and then uses that to find the start of the audio and video data and begin playing. Unfortunately, atoms can appear in any order, so the program doesn't know where the moov atom will be ahead of time.

Searching to find the moov works fine if you already have the entire video file[46].

45 https://rigor.com/blog/optimizing-mp4-video-for-fast-streaming/

46 https://www.adobe.com/devnet/video/articles/mp4_movie_atom.html

However another option is needed when you don't have the entire video yet, such as when you are streaming online. That's the whole point of streaming a video!

You can start watching it without having to download the entire video first.

When streaming, your browser requests the video and starts receiving the beginning of the file. It looks to see if the moov atom is near the start. If the moov atom is not near the start, it must either download the entire file to try and find the moov, or the browser can download small different pieces of the video file, starting with data from the very end, in an attempt to find the moov atom.

The FFMPEG option that moves the `moov atom` at the very beginning of the file is:

```
-movflags +faststart
```

If you already have lots of processed files that needs this particular option enabled, you can just type the following command, without re-encoding your original input:

```
  ffmpeg -i YOUR_INPUT.mp4 -c copy -movflags
+faststart -y YOUR_INPUT.mp4
```

The option `-y` will overwrite your original input, otherwise you can simply omit the `-y` option and change the output's file name.

Producing Adaptive WebM DASH Streaming

As described in the FFMPEG documentation[47], the DASH technology can be used also to deliver videos in WebM format.

This goal is accomplished following 3 steps:

1) By creating Audio and Video Streams in WebM
2) Creating the DASH Manifest
3) (Optional) By testing your stream using Dash.js

As described in the official WebM documentation[48], for streaming WebM files using DASH, the video and audio files have to be non-muxed and non-chunked, meaning that each video stream goes into it’s own file and each audio stream goes into it’s own file.

For more information please visit the WebM's Wiki.

47 https://ffmpeg.org/ffmpeg-formats.html#webm_005fdash_005fmanifest

48 http://wiki.webmproject.org/adaptive-streaming/instructions-to-playback-adaptive-webm-using-dash

Scaling with FFMPEG

The width and height are referred as the *resolution* of a video.

FFMPEG offers several ways to scale a video.

The fastest method is by using the `-s` option, such in this example:

```
ffmpeg -i YOUR_INPUT.mov -s 1280x720 
YOUR_OUTPUT.mp4
```

You can specify a width and height in pixel, such as in the above example, or you can use video resolution abbreviations, as described in the official FFMPEG's documentation[49], such in the following example:

```
ffmpeg -i YOUR_INPUT.mov -s 4kdci 
YOUR_OUTPUT.mov
```

By specifying `-s 4kdci` you will instruct FFMPEG to scale your input video at 4096x2160 pixels (4K resolution).

49 https://ffmpeg.org/ffmpeg-all.html#Video-size

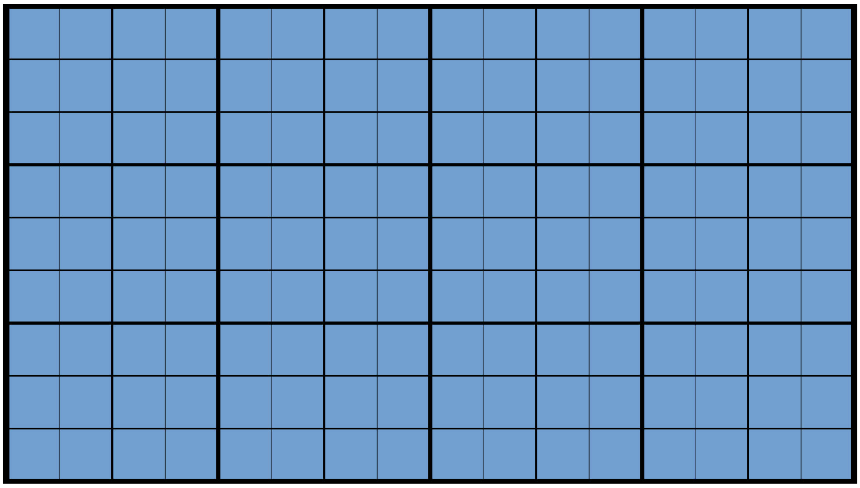
What is the Aspect Ratio

From Mirriam-Webster's dictionary:

Aspect Ratio: noun
The ratio of the width of a television or motion-picture image to its height.

If you need to process videos from different sources such as camera RAW footage, smartphones, old TV tapes, etc. you might end with different aspect ratios.

Common aspect ratios nowadays are 16:9 (TVs) or 9:16 (smartphones), but you might have situations where there are different aspect ratios involved, especially in the field of Cinema production[50].



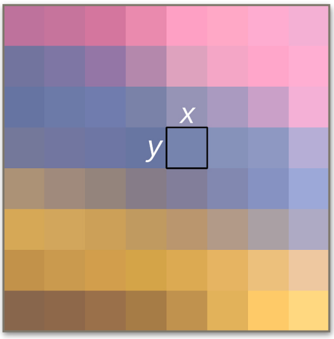
A 16:9 rectangle in which rectangles visualize the ratio.

50 https://www.red.com/red-101/video-aspect-ratios

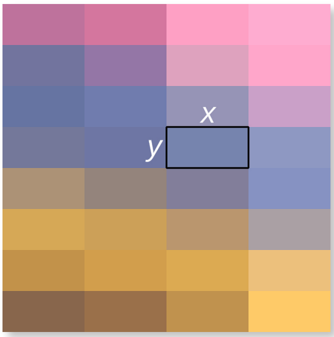
The Pixel Aspect Ratio (PAR)

PAR is a mathematical ratio that describes how the width of a pixel in a digital image compares to the height of that pixel.

Most digital imaging systems display an image as a grid of tiny, square pixels. However, some imaging systems, especially those that must be compatible with standard-definition television motion pictures, display an image as a grid of rectangular pixels, in which the pixel width and height are different. Pixel aspect ratio describes this difference[51]. Use of pixel aspect ratio mostly involves pictures pertaining to standard-definition television and some other exceptional cases. Most other imaging systems, including those that comply with SMPTE Standards and practices, use square pixels.



Pixel Aspect Ratio 1:1



Pixel Aspect Ratio 2:1

[51] https://en.wikipedia.org/wiki/Pixel_aspect_ratio

The pixel aspect ratio (PAR) is a value that can be specified in the FFMPEG's output, to instruct the final user's player how to display the pixels, using the combination of `-vf scale` (Video Filter for scaling) and the `-setsar` options (Sample Aspect Ratio), which is the FFMPEG's equivalent of PAR.

Example formula Set SAR to 1:1 (Scaling)

```
ffmpeg -i YOUR_4:3_INPUT.mov -vf
scale=1280x720,setsar=1:1 -c:a copy
YOUR_16:9_OUTPUT.mov
```

`-vf scale=1280x720`

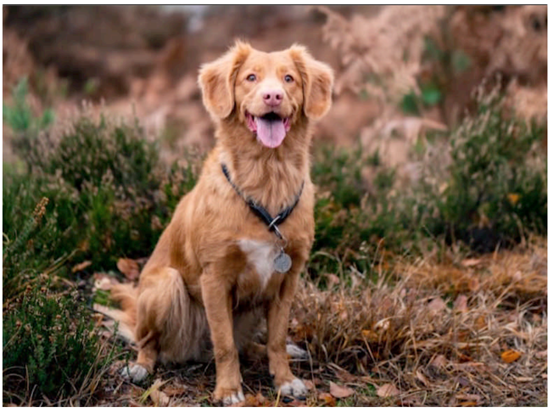
stands for "Video Filter Scale" and the 1280x720 are referred to the desired width and height output

`-setsar=1:1`

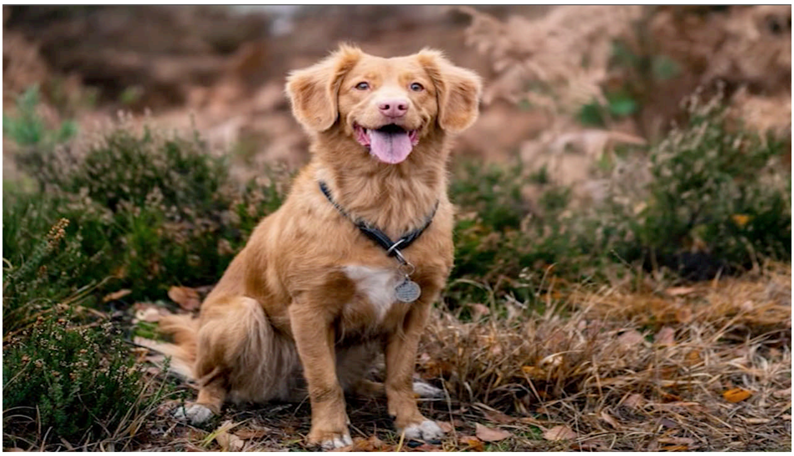
Stands for "Set the Pixel Aspect Ratio at 1:1 (Square)".

`—c:a copy`

This will copy the audio of the original input, without re-encoding.



Original 720x576 footage, 4:3

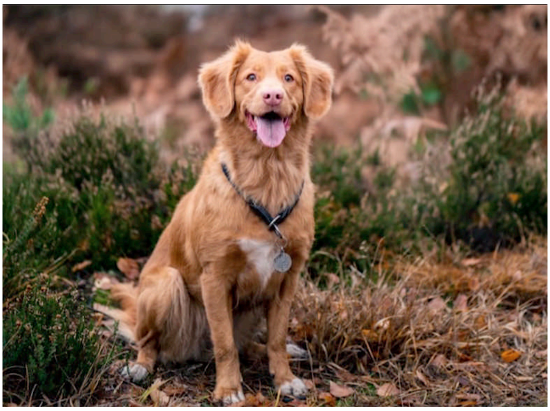


Final Output, scaled at 1280x720, 16:9

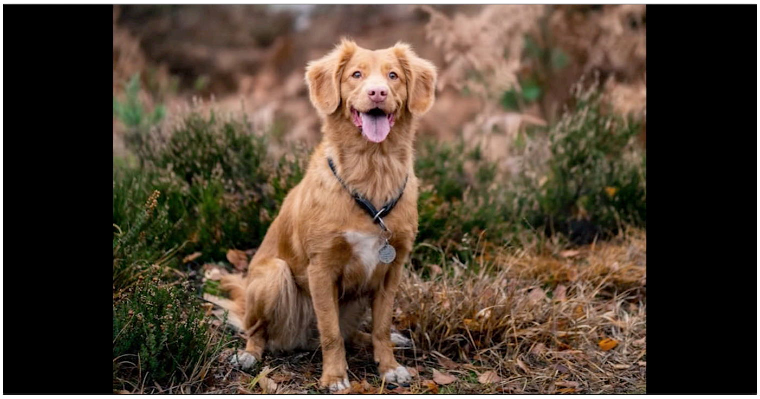
Example formula Set SAR to 1:1

(Preserve Original Ratio with Pillarbox[52])

```
ffmpeg -i INPUT.mov -vf
"scale=1280:720:force_original_aspect_ratio=
decrease,pad=1280:720:(ow-iw)/2:(oh-ih)/2"
-c:a copy OUTPUT.mov
```



Original 720x576 footage, 4:3



Final output at 1280x720, 16:9 with Pillar box

52 The pillarbox effect occurs in widescreen video displays when black bars (mattes or masking) are placed on the sides of the image. It becomes necessary when film or video that was not originally designed for widescreen

```
  -vf
"scale=1280:720:force_original_aspect_ratio=
decrease,pad=1280:720:(ow-iw)/2:(oh-ih)/2"
```

This `-vf` (video filter) will perform 2 distinct operations. The first will scale the input at 1280 (width) x 720 (height) and will preserve the original aspect ratio by scaling your input. The second (pad) will compute and insert a pillar box, so to produce a final 1280x720 output video. Letterboxing[53] will occur if your input's aspect ratio is wider than the output aspect ratio.

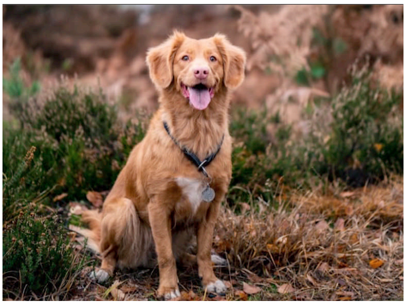
The word `"iw"` stands for "input width"; `"ow"` stands for "output width"; the word `"ih"` stands for "input height" and "`oh`" stands for output height. A mathematical expression is used to compute the size of the pillar/letter box, based on your input.

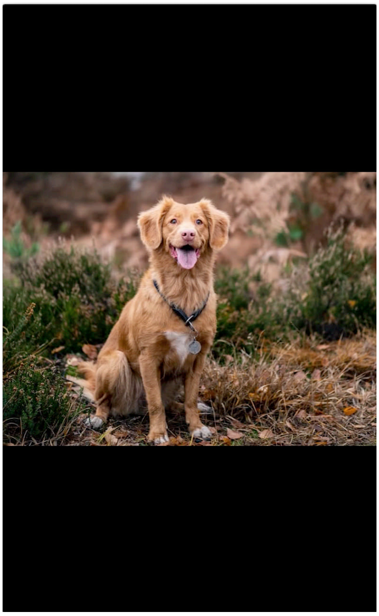
53 Letterboxing is the practice of transferring film shot in a widescreen aspect ratio to standard-width video formats while preserving the film's original aspect ratio. The resulting videographic image has mattes (black bars) above and below it.

The 9:16 Aspect Ratio

Many social networks such as Instagram or YouTube supports the 9:16 ratio. To produce a 9:16 video preserving the original aspect ratio you will have to invert the width with the height, as in this example formula for a 720x1280 video:

```
ffmpeg -i YOUR_16:9_INPUT.mov -vf
"scale=720:1280:force_original_aspect_ratio=
decrease,pad=720:1280:(ow-iw)/2:(oh-ih)/2"
-c:a copy YOUR_9:16_OUTPUT.mov
```

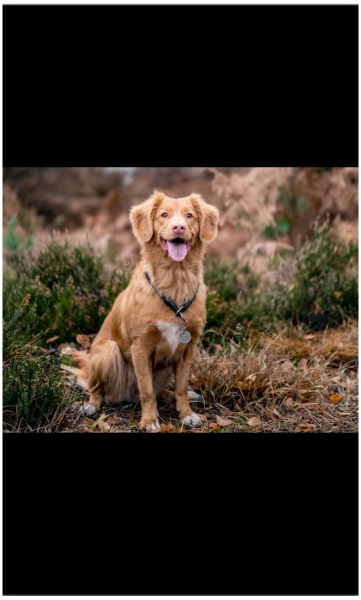

Original 720x576 footage, 4:3


Final Output, 1280x720, Letterboxed, 9:16

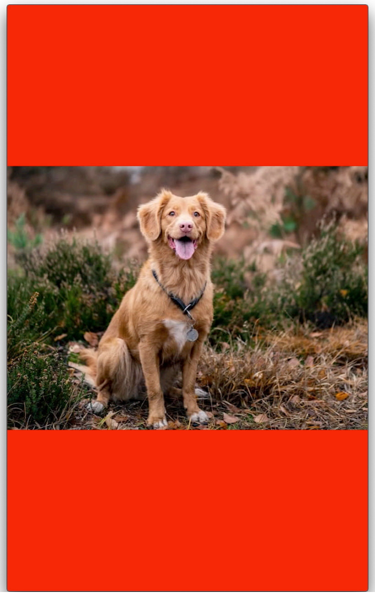
Colored Pillar/Letter Box

To change the color of the pillar/letter box bars, you can specify a value with the `color` option, such as in this example for producing red letterbox's bars:

```
ffmpeg -i INPUT.mov -vf
"scale=720:1280:force_original_aspect_ratio=
decrease,pad=720:1280:(ow-iw)/2:(oh-ih)/
2:color=red" OUTPUT.mov
```



Original 720x1280
Letterbox, Black Bars



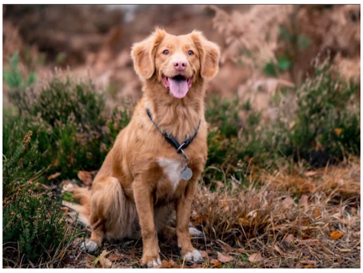
Final Output, 720x1280
Letterbox, Red Bars

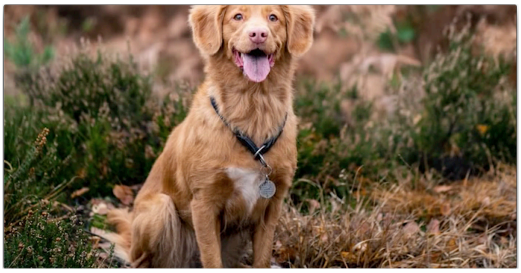
A list of FFMPEG's color options is available here.

Crop

When you want to avoid Pillar/Letter Boxes you can use the crop option, as in this example:

```
ffmpeg -i YOUR_4:3_INPUT.mov -vf
"scale=1280:720:force_original_aspect_ratio=
increase,crop=1280:720" CROP_OUTPUT.mov
```


Original 720x576 footage, 4:3


Final 1280x720 Cropped Footage, 16:9

Overlay Images on Video

With FFMPEG the formula is simple. Assuming that you are working with a Mp4 container and h264/AVC codec you just need to type:

```
ffmpeg -i input.mp4 -i logo.png
-filter_complex "overlay=0:0" -codec:a copy
output.mp4
```

Depending on which image you need to overlay on your video, usually better results are achieved with transparent PNG graphics.

This command's syntax is as follow:

`-i input.mp4`: this is an example video called "input.mp4"

`-i logo.png`: a second input is declared with an example "logo.png" file

`-filter_complex "overlay=0:0":` with this option you specify which filter to enable for the process

`-codec:a copy`: this instruction will keep the exact same audio parameters, without conversion.

When you need a better result, using the same Mp4 container and h264 codec, you might want to explore the different settings available for this kind of compression.

FFMPEG offers a great H264 encoding guide.

When working with h264 you might want to decide first wheter you want to work with a Constant Rate Factor, or CRF, which keeps the best quality and care less about the file size, or using a Two-Pass encoding, if you are targeting a specific output file size, and if output quality from frame to frame is of less importance.

Overlay Images on Pictures

Basic Overlay

Let’s take the following example:

Image #1

Photo Credit:Boxed Water Is Better on Unsplash

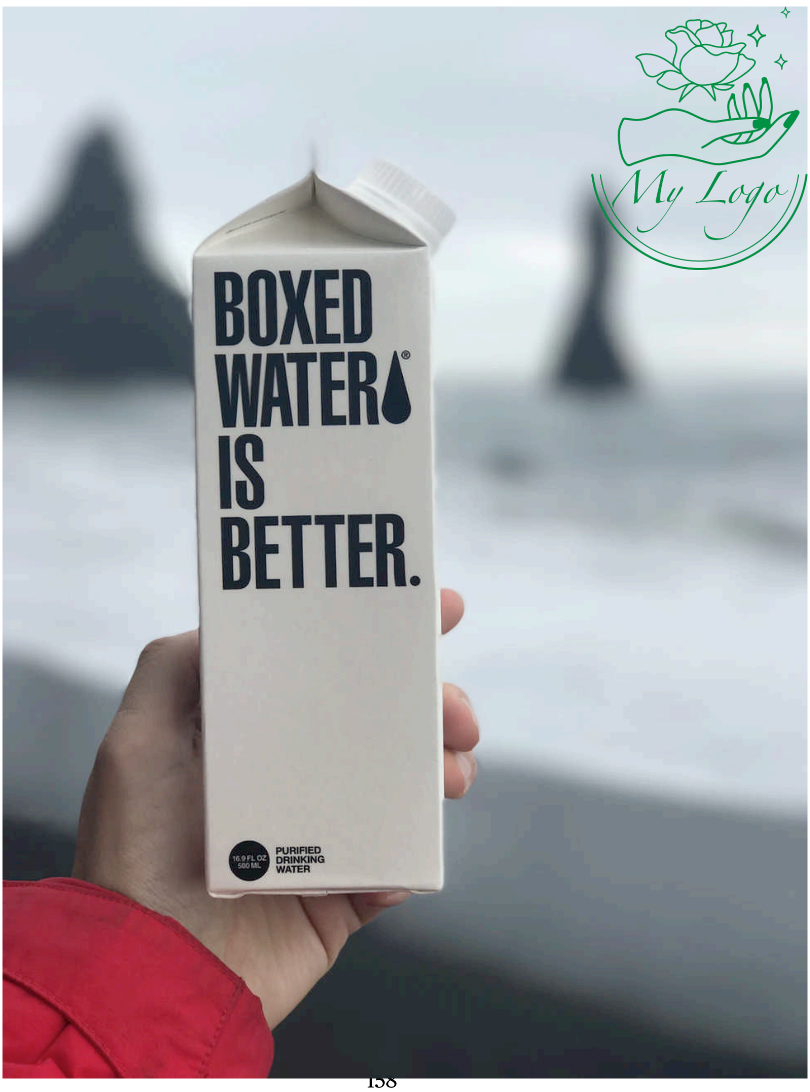
PNG Logo File

© Adobe Stock

What we want to accomplish here is a very simple task.

We have one image, the milk pack, and another image the logo in trasparent PNG format. We want to overlay the logo on the top-right corner of our main image.

Final result will be as the example below:



Assuming that we'll have the original image as a JPEG format and the logo as a transparent PNG format, this task will be accomplished by FFMPEG just typing this simple command on your Terminal:

```
ffmpeg -i backroundimage.jpg -i logoimage.png
-filter_complex "overlay=0:0" finaloutput.jpg
```

Let's breakdown this command so to better understand the FFMPEG syntax.

`ffmpeg`: this command will launch the program

`-i`: The minus symbol "-" before the "i" letter stands for "do process something". In other words this command tells the machine to execute a specific task. The "i" letter stands for "input": it tells the machine to expect an input somewhere.

`backroundimage.jpg`: in this example is our image #1

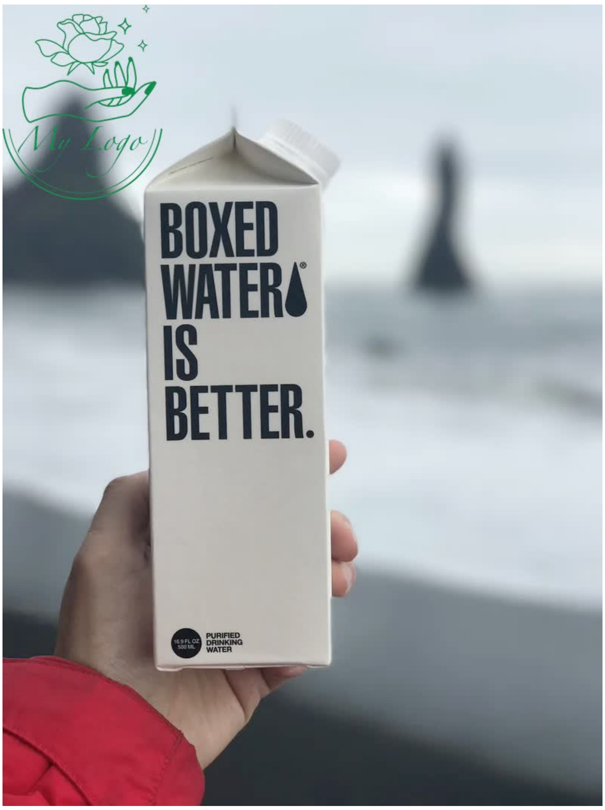
-i: in this formula, the second "-i" command tells FFMPEG to expect a secondary input

`logoimage.png`: in our example is the logo image file, in PNG format

`-filter_complex`: this tells FFMPEG to execute a particular Filter capabilty.

`"overlay=0:0"`: this tells FFMPEG to execute one particular filter called "*Overlay*" and furthermore specify that the overlay input must be placed at pixel position zero (0) horizontally and at pixel position zero (0) vertically.

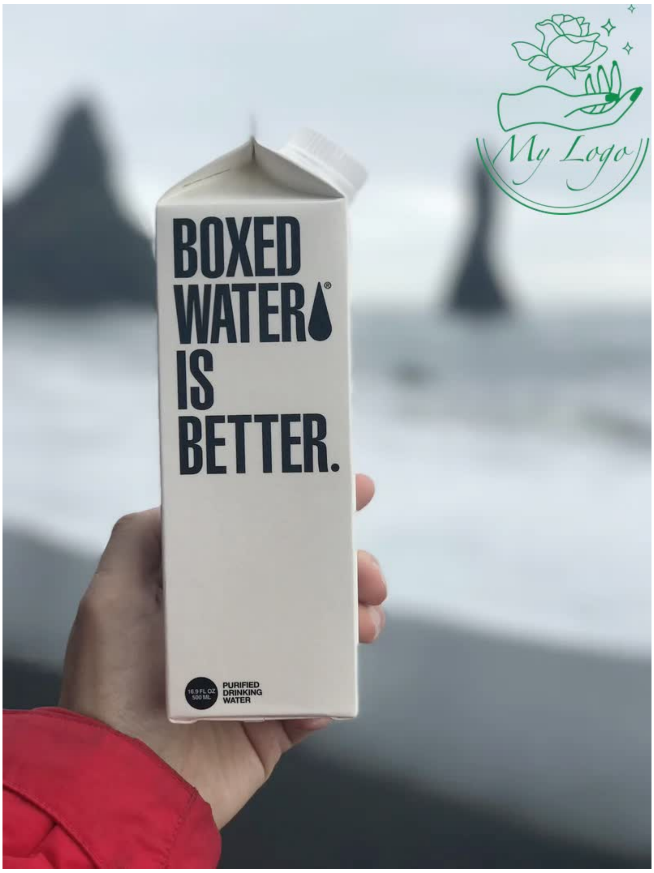
By using **“overlay=0:0”** the final results will be like this:



Notice that the logo is placed on the top-left corner of the image. But in our example we want to place the logo on the top-right. So we will need to adjust the horizontal values, in this case “**overlay=575:0**” which means:

“Move the logo by 575 pixel horizontally”

```
ffmpeg -i backroundimage.jpg -i logoimage.png
-filter_complex “overlay=575:0” finaloutput.jpg
```



If you need to place the overlay image on a different position, you can adjust the values “0:0” specified on the **`-filter_complex`** value, accordingly to your needs.

```
ffmpeg -i backroundimage.jpg -i logoimage.png
-filter_complex “overlay=0:0” finaloutput.jpg
```

Okay, we’ve just performed a very easy task with FFMPEG: overlay a logo on top of an image. So far, so good.

But what about processing an entire folder containing more than 1.000 images, all of which would need to overlay the same logo or a specific graphic?

By taking one image at the time with FFMPEG, it would just take ages. So it’s time to install and use a new program called Image Magick.

ImageMagick

ImageMagick is a suite of free and open-source software for displaying, creating, converting, modifying, and editing images. It can perform actions on over 200 different image file formats.
The history of the project can be found here:

https://imagemagick.org/script/history.php

Installing ImageMagick on a Mac

On MacOS, assuming that you have installed HomeBrew, as covered previously, just open the Terminal Application and type the following line:

```
brew install imagemagick
```

This might take a while, so be patient.

Installing ImageMagick on Ubuntu

1) Install Prerequisite PHP-Library

```
sudo apt-get update
sudo apt install php php-common gcc
```

2) Install LibOpenJP2-7

```
sudo apt install libopenjp2-7-dev
```

3) Install ImageMagick

```
sudo apt install imagemagick
```

4) Install ImageMagick for PHP

```
sudo apt install php-imagick
```

5) Restart Apache

```
sudo systemctl restart apache2
```

Alternatively you can download and build it from Unix Source:

`https://imagemagick.org/script/install-source.php`

Despite the first solution (PHP ImageMagick) is easier, i strongly advise you to follow the step **#3** (installing from Unix Source), since you might end up with missing options or other issues while running the Formulas described in this book. You may find useful to follow the steps described in my Gist Repositories[54], that will guides you thru the Source installation of ImageMagick on Ubuntu (tested on Ubuntu 20.04 LTS):

`https://bit.ly/magick-20`

For Ubuntu 18.04 you can find a specific GIST here:

`https://bit.ly/magick-18`

[54] Gist is an easy method to share snippets or excerpts of data with others on GitHub. A gist can be a string of code, a bash script or some other small piece of data. These bits of information are hosted by GitHub as a repository.

Installing ImageMagick on Windows

Make sure to follow the instructions of installing “*Linux Subsystem for Windows*” as covered in this technical article: After installing and following the above mentioned article, you can follow the same steps as for Ubuntu 18.04 or 20.04, depending on which Ubuntu version you've installed in your Windows machine.

Batch Process - Overlay to Multiple Images with Same Size

In order to perform a batch processing of multiple files with the same sizes, we need to use a combination of scripting and Imagemagick's programs.

First of all: always backup your files BEFORE any action. In this way, if anything will go wrong, you will be able to go back to a previous state.

Now, as i mentioned before, we need to create a simple script. A script is just a text file that contains several instructions that will run on your Terminal, in order to automate processes or perform special tasks.

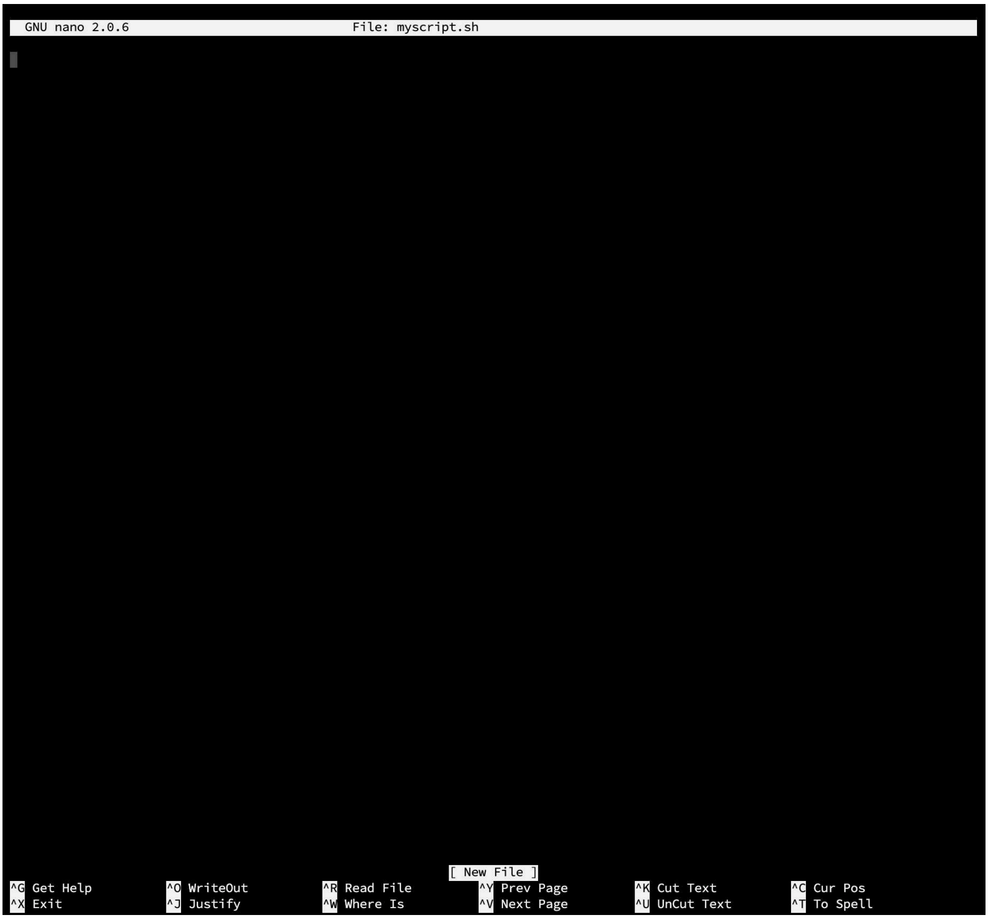
Step 1: Open your Terminal App

Step 2: Create a simple text file with a simple program called "Nano" which it's already installed on your system.

Then type the following:

```
nano myscript.sh
```

You will see a screen like this:



```
  GNU nano 2.0.6                            File: myscript.sh

                                            [ New File ]
^G Get Help     ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit         ^J Justify      ^W Where Is     ^V Next Page    ^U UnCut Text   ^T To Spell
```

This program is called "Nano" which is a basic text editor.

Now, within Nano just type the following script:

```
mkdir Processed
for f in *.png
do composite -gravity northeast yourlogo.png
$f Processed/$f
done
```

What this script means

`mkdir Processed`

= create a new folder called “Processed” within the current directory. Of course you can use any name instead of “Processed”.

`for f in *.png`

= expect to do something against a set of variable files, in this case all the files within the current directory that has the “.png” extension

```
  do composite -gravity northeast
yourlogo.png $f Processed/$f
```

= conditionally perform a command against each item.

This is what ImageMagick will do. It will overlay a file called "`yourlogo.png`" on all PNG files founded in the previous command and will put it on the top right corner (`northeast`) of the image. Then it will put the processed file into the directory called "Processed" or any name you might have choose in the previous steps.

```
  done
```

= tells the script to end when the previous tasks are completed.

When you're done just press CONTROL + the "X" key and then press the key "Y" on your keyboard. This will save your document called "`myscript.sh`"

Important Note: your files might have different extensions. So you just need to modify the line `for f in *.png` with the desired extesion.

You can also add as many extensions as contained in your folder, for example:

```
for f in *.png *.jpg *.tiff *.gif
```

Of course the 'yourlogo.png' is an example file.
You will need to insert your logo's filename (or any other main graphic's filename) accordingly.

Now you can execute the script, by typing:

```
source myscript.sh
```

What *Source* means?

= **`source`** is a command which is used to read and execute the content of a file, in this case "myscript.sh".

Please be patient since the Terminal will start to perform the above actions and this might take a while.
You won't see a typical 'Status bar' nor you will have any progress indications, unless you add to the command the option **`-x`** which enables a verbose mode during the execution of the command.

To display a "verbose mode" of the script, just type:

```
sh -x myscript.sh
```

Just wait until the script will finish. If you want to suspend the script you can always press **CTRL + z** on your keyboard, at any time.

Now you will find a new folder named “`Processed`” which will contains the output results of the actions performed by the script “`myscript.sh`”.

If you need to perform the same action on multiple images, but with different sizes involved, you will need to perform a different script, which is described on the next Formula.

Batch Process - Overlay to Multiple Images with Different Sizes

You might have a folder with thousands images, each one with a different width and height. And you might need to apply a Watermark or a Logo or a Graphic Overlay on each one of them.

We have a variable here: the main image, namely the background image destined to have your logo or any other graphic overlay.

By using the previous Formula and by using a logo, or a graphic with common extensions such as PNG, JPG or TIFF you might end with large images displaying a little logo or a big logo overlayed on a little image.

In order to batch process a folder with various and unknown sizes we need to use again the power of ImageMagick, but this time with a different approach: to proportionally resize the logo, based on the image sizes.

First it will be a good idea to have a logo available in a vector format,such as the SVG format or in Adobe Illustrator.

In this way you can produce a trasparent, big sized, PNG logo.

On this formula i've created a dedicated directory for the Logo file, in Transparent PNG, so to NOT process the `logo.png` file itself.

In the following example i will use a big logo (1900 x 2319 pixel): in this way the logo won't pixelate in case of a big image input. This logo size will be generally fine for pictures up to 4/5 thousands pixels. For bigger input files (8Megapixel, 12Megapixel, 25Megapixel foto, or above) you might want to have a bigger logo file at your disposal.

In the following example i've imagined a case scenario where i have both `.png` and `.jpg` files to process.
If you have just `.png` or just `.jpg` files or you have additional extensions, you can simply remove, or add, the desired file extension accordingly.

Now on Nano type:

```
nano resize.sh
```

By doing the previous command you will create an empty text file called "resize.sh" and you will find yourself into a new empy Nano window.

Now type the following script:

```
for f in *.png *.jpg
do magick $f -set option:sz "%[fx:min(w,h)*20/100]" \
\( ~/Desktop/testpict/logo/logobig.png -resize "%[sz]" \) \
-alpha on -channel rgba -gravity northeast \
-define compose:args=50,100 -composite $f
done
```

Let's breakdown this command.

`for f in *.png *.jpg`

=

For every file contained into the current directory, find all files that has the .png and the .jpg extension

`do magick $f`

=

run the program named **`magick`** which is part of the suites of softwares installed with Imagemagick and process the various files found in the previous command.

```
  -set option:sz
  =
```

This will assign a function called **`sz`** (there is no special meaning of the word "**`sz`**": it's just a name created for this function).

```
  "%[fx:mix(w,h)*20/100]" \
```

This is where the "magick" happens!
FX stands for "*Special Effects Image Operator*" and it's the collection of special expressions that you can enable with Imagemagick. There are a lots of expressions to discover: take a look here.
Basically the above mathematic expression will analyze the unknown width and height of your logo file and will calculate the 20% of its proportion.
If you want your logo to have a different percentage of this proportion on your final output you can just modify the ***20**/100* part of this command, with your desired percentage.

The backslash symbol \ means "escape".

Escape characters are used to remove the special meaning from a single character.

A non-quoted backslash \ is used as an escape character in BASH.
It preserves the literal value of the next character that follows, with the exception of a *newline*.

If a newline character appears immediately after the backslash, it marks the continuation of a line when it is longer that the width of the terminal; the backslash is removed from the input stream and effectively ignored.

```
  \( ~/Desktop/testpict/logo/logobig.png
-resize "%[sz]" \) \
```

The above command will specify the full path of your directory's logo input and apply a "-resize" command within the parenthesis, calling the previous mathematical proportion called '`sz`'.

The next \ backslash, after the closing parenthesis, will instruct bash to read the new following line.

Please make sure to add a blank space within the opening parenthesis and the closing (**`like this`**).

```
-alpha on -channel rgba -gravity northeast \
-alpha on -channel rgba
```

=

This command states that the logo will be applied with transparency on the background image, preserving the transparent channel (information) of the logo file.

```
-gravity northeast
```

=

indicates that the logo will be applied on the top right of the background image. Options available are: *north*, *east*, *west*, *south* and all the possible combination of these values.

```
-define compose:args=50,100 -composite $f
```

```
-define compose:args=50,100
```

=

{source percentage},{destination percentage}

This will call the "**`compose`**" options for the "**`composite`**" commands, so to create a new file which is the results of the previous operations.

Batch Resize Images

With the ImageMagick **`-resize`** option, you can quickly and easily batch scale images to a desired size.

Place all the images you want to scale in a directory and navigate to that location via command line, then type:

Resize Formula

```
mogrify -resize 800x800 *.png
```

With the above example formula you will resize to 800x800 pixel every .png file contained in your working directory. Of course you can change the "800x800" value and the **`*.png`** extension, based on your desired size and file format.

Resize preserving the aspect ratio

```
mogrify -resize 800 *.png
```

The above formula will resize the image width at 800 pixel and will calculate the height so to keep the aspect ratio of the image.

Resize with specific width + height and preserving aspect ratio

```
mogrify -resize 800x800! *.png
```

With the exclamation mark after your desired values, ImageMagick, when and where possible, will resize your image based on your input and will keep also the aspect ratio.

Batch Resize, Lower Quality and Convert Pictures

With this code you can perform 3 distinct action on a directory.

The first action will be to resize your pictures, as discussed in the previous chapter. The second action will lower the source quality, in case you might need to lower quality for a website destination or similar needs. The third action will convert the input into another format.

To do so, first of all, let's create a new folder and let's call it **`Converted`**. Let's assume that your pictures are in a folder named **`Images`** which is in your Desktop.

In this example you will type the following command:

```
mogrify -path ~/Desktop/Images/Converted 
-resize 800x800 -quality 50 -format jpg 
*.jpg
```

Based on your needings, you will adjust the **`-path`**, **`-resize`**, **`-quality`** and **`-format`** options accordingly.

Convert Images to WebP

The WebP format is a lossless and lossy compression for images for the web developed by Google. Using WebP, webmasters and developers can create smaller, richer images that make the web faster.[55] Lossless WebP supports transparency, also knows as Alpha Channel.

At the time of writing this book the WebP format is currently unsupported by Safari 13, but it will be, starting from Safari 14[56] across all the Mac and iOS ecosystems.

WebP is natively supported in Google Chrome, Firefox, Edge, the Opera browser, and by many other tools and software libraries, including FFMPEG.

On MacOSX, you can install the WebP tool with Brew, simply typing:

```
brew install webp
```

55 https://developers.google.com/speed/webp

56 https://www.macrumors.com/2020/06/22/webp-safari-14/

This will install a series of tool for WebP including:

- **`cwebp:`** webp encoder tool.

- **`dwebp:`** webp decoder tool.

- **`img2webp:`** tools for converting a sequence of images into an animated webp file.

- **`webpinfo:`** used to view info about a webp image file.

- **`webpmux:`** Create animated WebP files from non-animated WebP images, extract frames from animated WebP images, and manage XMP/EXIF[57] metadata and ICC profile[58]. More on this tool here.

[57] The Extensible Metadata Platform (XMP) is an ISO standard, originally created by Adobe Systems Inc., for the creation, processing and interchange of standardized and custom metadata for digital documents and data sets. Exchangeable image file format (EXIF) is a standard that specifies the formats for images, sound, and ancillary tags used by digital cameras, including smartphones, scanners and other systems handling image and sound files recorded by digital cameras.

[58] In color management, an ICC profile is a set of data that characterizes a color input or output device, or a color space, according to standards promulgated by the International Color Consortium (ICC)

Encode an image into WebP:

```
cwebp YOUR_INPUT.png -o YOUR_OUTPUT.webp
```

Your input can be in every supported extension format.

Decode a WebP image into PNG or other formats:

```
dwebp YOUR_INPUT.webp -o YOUR_OUTPUT.png
```

You can use any supported extension at the end of this command, accordingly to your needs.

Batch Convert from PNG to WebP format:

```
for i in *.png; do cwebp "${i}" -o "$
{i%}.webp"; done
```

In the above example every `.png` file founded will be converted in the WebP format using the **`cwebp`** encoder tool. You can change the `.png` extension accordingly to your needs.

Batch Decode from WebP format into JPG:

```
for i in *.webp; do dwebp "${i}" -o "$
{i%}.jpg"; done
```

In the above example every WebP file founded in the working directory will be converted in the JPEG format.

Remove Black Bars/Borders from Images and Trim

With ImageMagick you will be able to remove the black bars or black borders from any image.

The formula is simple:

```
convert example.png -bordercolor black
-border 1x1 -fuzz 20% -trim output.jpg
```

The **`-fuzz`** option[59] is better described in the ImageMagick's documentation.

To batch convert an entire folder of pictures with black bars/borders, simply locate the source directory of your pictures and positionate yourself within this directory, for example: **`cd ~/Desktop/Images`** and type:

```
convert *.png -bordercolor black -border 1x1
-fuzz 20% -trim converted.jpg
```

59 A number of algorithms search for a target color. By default the color must be exact.

Batch Convert Pictures from RAW to JPEG format

The RAW format is a proprietary digital camera format that contains all the information captured by the camera's sensors. RAW formats give the photographer the freedom and the artistic control to create the best result. A creative can open and edit a RAW file with popular softwares such as Adobe Lightroom or Photoshop.

There are dozens of raw formats in use by different manufacturers of digital image capture equipment.

Sometimes you will need to quickly process a RAW file, without having to open Adobe Photoshop, LightRoom or any other proprietary software.

In this chapter we will discuss the methods to do so.

For MacOS X users: the utility **SIPS** will do the magic.

For Ubuntu users: ImageMagick is the solution.

MacOS X:

SIPS stands for "**S**criptable **I**mage **P**rocessing **S**ystem" and it's used to convert or process lots of picture formats, including RAW, PNG, TIFF, JPG, BMP, CR2, CR3, etc.

SIPS has been included already in every MacOS X Release, since Panther (Oct. 2003).

A basic formula for converting from RAW to JPG one picture (i.e.: "IMG_001.cr2") will be:

```
sips -s format jpeg -s formatOptions 80
"IMG_001.cr2" --out "IMG_001.jpg"
```

Let's breakdown the command:

`sips`: will start the program

`-s format jpeg`: will specify the output format

`-s formatOptions 80`: will produce a JPG at 80% of quality

`--out:` will name the output file as specified in the quotation marks

A batch conversion of an entire directory will require just 3 steps:

1) Create a simple script named as you like, like the example "raw.sh" below:

```
nano raw.sh
```

2) Type the following formula:

```
for i in *.cr2; do sips -s format jpeg -s
formatOptions 80 "${i}" --out "${i%cr2}jpg";
done
```

3) Save and exit from your "raw.sh" text. Then simply type:

```
source raw.sh
```

The formula above will batch convert any ".cr2" file into a 80% quality JPG and will keep the original filename.

Please note that RAW files format extensions can vary from camera to camera. Hasselblad uses **`.fff`** or **`.3fr`** file extensions while many Sony cameras use its proprietary **`.arw`** format. You can modify the above formula, accordingly to the file extension to process.

For Example: in the Sony's ARW Format you will use the following:

```
for i in *.arw; do sips -s format jpeg -s
formatOptions 80 "${i}" --out "${i%cr2}jpg";
done
```

Ubuntu users:

The SIPS substitute in Ubuntu can be Imagemagik, as discussed in the ImageMagik's Chapter of this book.

A basic command to convert a RAW format into JPG, at 80% of its quality will be:

```
convert -auto-orient INPUT.fff -quality 80
-delete 0 OUTPUT.jpg
```

`convert`: will open ImageMagick's Convert Utility

`-auto-orient`: will try to understand the original orientation information in the RAW file

`INPUT.fff`: this is an example input file

`-quality 80`: will produce a JPG file at 80% of quality

`-delete 0`: it will discard additional RAW data and will output just one file. If something goes wrong with your output file, you can modify this value with `-delete 1`

`-OUTPUT.jpg`: this is the name of an example output file.

Please note: ImageMagick might output some warnings, such "*Unknown field with tag ...*". This is just a warning and not an error, and your final output should be fine.

It is just a warning that there is some special field in your input file that ImageMagick does not understand.

To batch convert an entire folder of RAW pictures, you can create a bash script (named for example "`raw.sh`") by typing:

```
nano raw.sh
```

At this point you can type:

```
for i in *.fff; do convert -quality 80
-delete 1 -auto-orient "${i}" "${i%fff}jpg";
done
```

Then run your bash script, by typing:

```
source raw.sh
```

Please note that proprietary RAW formats such as Hasselblad's **`.3fr`** and Sony's **`.arw`** might be not fully compatible with some installations of ImageMagick. You can solve this by using SIPS with MacOS X as described above.

Ghostscript for PDF processing

Ghostscript is a collection of software dedicated to PostScript™ and Portable-Document-Format (PDF) processing, both of which are Adobe standards.

In case of huge file size PDFs, Ghostscript can reduce drastically the overall size of it.

To install Ghostscript on MacOS X you simply type:

```
brew install ghostscript
```

To install Ghostscript for Ubuntu, you can type:

```
sudo apt update
sudo apt install ghostscript
```

The usual command syntax is as simple as:

```
ghostscript input.pdf output.pdf
```

On MacOS X you might use **`gs`** instead:

```
gs input.pdf output.pdf
```

Sometimes the above command will work just fine, but some PDFs might require additional tweaks in the command in order to produce greater results.

A good script that can produce better results with black-and-white as well as color content, is as follow:

```
gs \
  -o OUTPUT.pdf \
  -sDEVICE=pdfwrite \
  -dDownsampleColorImages=true \
  -dDownsampleGrayImages=true \
  -dDownsampleMonoImages=true \
  -dColorImageResolution=72 \
  -dGrayImageResolution=72 \
  -dMonoImageResolution=72 \
  -dColorImageDownsampleThreshold=1.0 \
  -dGrayImageDownsampleThreshold=1.0 \
  -dMonoImageDownsampleThreshold=1.0 \
   YOUR_INPUT_PDF_FILE.pdf
```

This is just an example formula, and must be tested against your PDF file, but should work in many cases, optimizing your file size.

Extract Images from PDF

In this chapter i will describe you how to extract images from one PDF file and convert it to JPG or PNG (lossless).

We will use ImageMagick for this purpose.

Please refer to the ImageMagick chapter in order to know how to install this program on MacOS X and Ubuntu.

JPG Extraction: first, create a directory to place the extracted images.

We will create a directory named "`extracted`":

```
mkdir extracted
```

Then you can type the following command:

```
convert example_input.pdf -quality 100
extracted/output.jpg
```

The above command will produce a sequential `output-0.jpg`, `output-1.jpg`, `output-3.jpg` accordingly to the numbers of the PDF source file.

In order to have a specific prefix with a specific sequence of numbers, you can modify the above command with:

```
convert example_input.pdf -quality 100 
extracted/img03%d.jpg
```

The suffix **`img03%d`** will produce a sequence of files with 3 digits number after the prefix **`img`**, ranging from 000 to 999. If you need to generate a different prefix followed by 3 digits, ranging from 000 to 999, you can modify it with any prefix, as in **`anyprefix%03d.png`**.

If you neede 4 digits, you can simply modify it with **`anyprefix%04d.png`**.

If you need a 2 digits suffix, you can type **`anyprefix%02d.png`**.

PNG Extraction: you can create the same "extracted" directory, by typing:

```
mkdir extracted
```

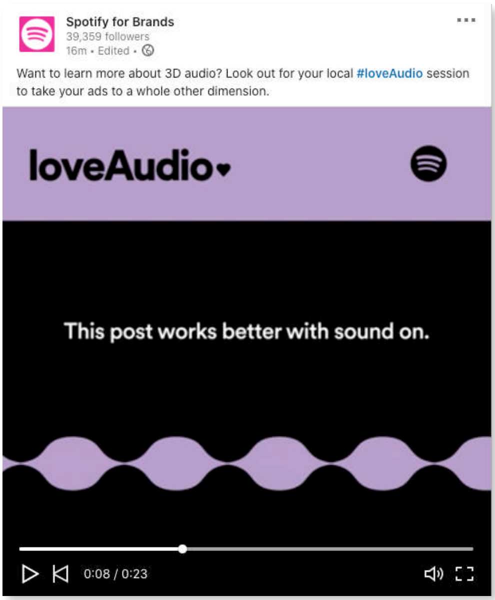
Then you can type:

```
convert example_input.pdf extracted/
img03%d.png
```

Generate Waveforms from Audio

A great deal of content in 2020 will be consumed by listening to Podcasts. Spotify invested over $500m in content acquisition and development, and this type of content is set to stay for a long time.

But how do you promote a PodCast by "visualizing" it, on social media and using a video? A great solution is to visualize the audio waveform of the podcast itself, inserting subtitles or a big banner describing what's the podcast is all about.


Spotify for Brands
39,359 followers
16m • Edited •
Want to learn more about 3D audio? Look out for your local #loveAudio session to take your ads to a whole other dimension.

Let's take the example post here, from Linkedin's Spotify for Brands account.

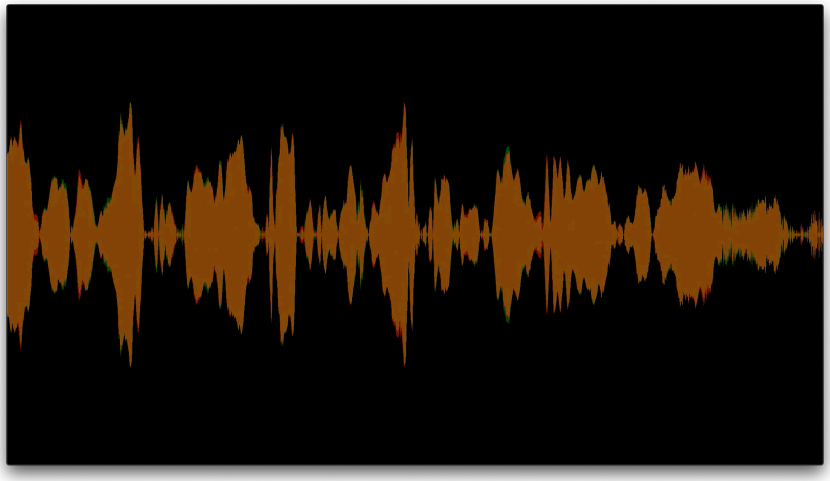
The post is a video which has a Top Header Banner and a colored waveform that goes in sync with an audio content.

In this chapter i will explain you how to achieve a similar results, without the need of any software or any subscription-based Apps, just with FFMPEG, which offers several styles of waveform generation.

The styles are described on the FFMPEG's Filters Section of the guide.

Let's explore the steps and the styles!

Style "Cline": Cline, draw a centered vertical line for each sample



Style "Cline"

Step 1: Generate the Waveform from your audio file, with this command:

```
ffmpeg -i MY_AUDIO.AAC -filter_complex
"[0:a]showwaves=mode=cline:s=1920x1080[v]"
-map '[v]' -map '0:a' -c:a copy -pix_fmt
yuv420p MY_VIDEO.mp4
```

Let's breakdown this command:

`-i MY_AUDIO.AAC`: this is an example audio input file

`-filter_complex:` this will call FFMPEG's Filter Complex Option

`"[0:a]showwaves=mode=cline:s=1920x1080[v]"`: this will genereate the waveforms for a video in Full-HD (1920x1080). You can modify and insert any size you want, based on the destination of your video.
The Audio Waveforms source will be taken from the stream number 0 (your input) and are named with an [a] standing for "audio", while the resulting video (called [v], for "video") will be passed on the next option.

`-map '[v]' -map '0:a' -c:a copy`:
The -MAP option will take the resulting 1920x1080 video '[v]' and merged with the audio track (your source file) in order to produce a final single file, wihout re-encoding your audio source (thus the `-c:a copy` in the command).

`-pix_fmt yuv420p`: this command is used for playback compatibility with Quicktime player and other players which aren't build with FFMPEG, such as VLC Player.

`MY_VIDEO.mp4`: this is an example output filename

To change color (for example **`white`**) you can add a colon (**:**) after the size specifier (**`s=1920x1080`**), like in the following formula:

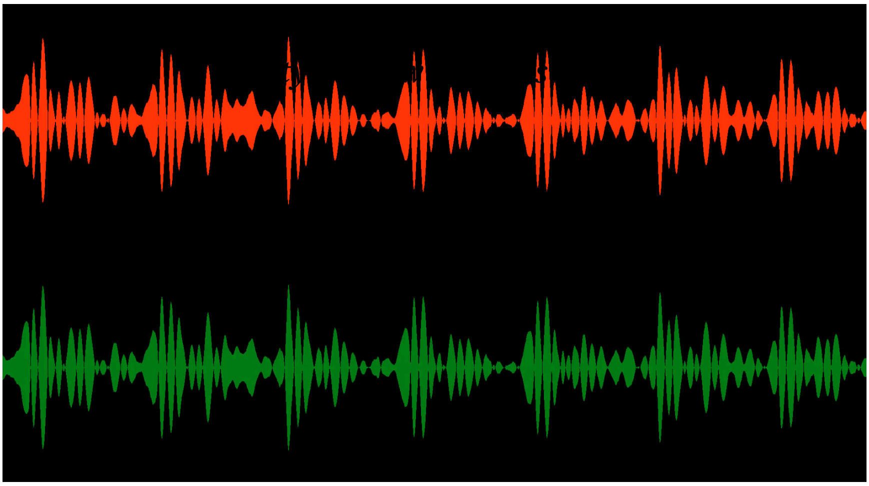
```
ffmpeg -i MY_AUDIO.AAC -filter_complex
"[0:a]showwaves=mode=cline:s=1920x1080:colors=white[v]"
-map '[v]' -map '0:a' -c:a copy -pix_fmt yuv420p
MY_VIDEO.mp4
```

To know the full color list and how to modify it, please refer to the Color section of the FFMPEG documentation: https://ffmpeg.org/ffmpeg-utils.html#Color

Style "Split Channels": The Left and the Right audio channels are visualized separately.

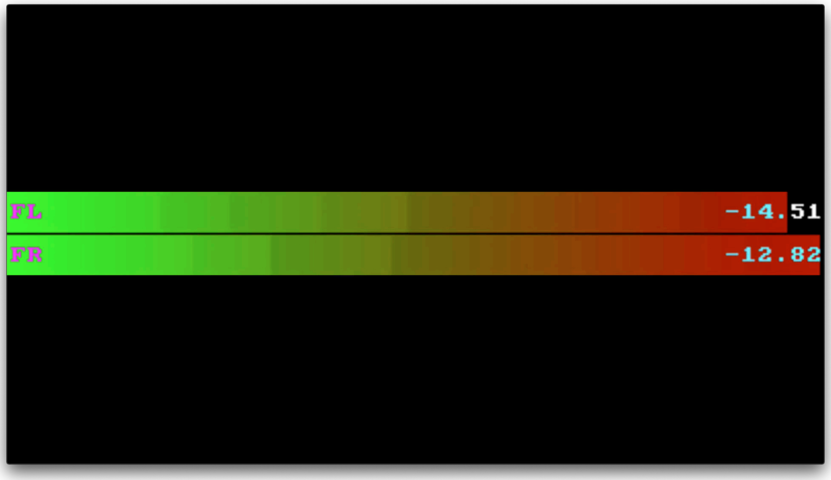
The formula is:

```
ffmpeg -i MY_AUDIO.aac -filter_complex
"[0:a]showwaves=split_channels=1:mode=cline:
s=1920x1080[v]" -map '[v]' -map '0:a' -c:a
copy -pix_fmt yuv420p MY_VIDEO.mp4
```



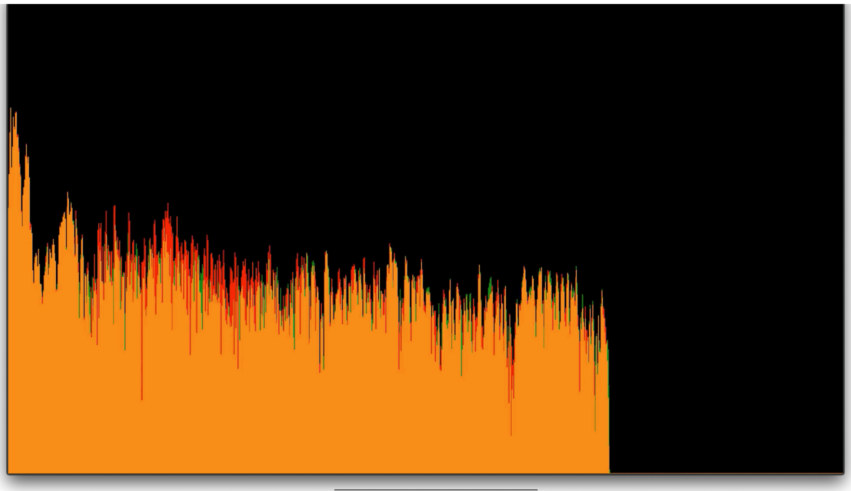
Style "Split Channels"

Style "Show Volumes": This will generate the Audio Left + Audio Right Volume Animation based on your input audio



The formula is:

```
ffmpeg -i MY_AUDIO.aac -t 00:01:00
-filter_complex
"[0:a]showvolume,scale=1920:-1,pad=1920:1080
:(ow-iw)/2:(oh-ih)/2[v]" -map '[v]' -map
'0:a' -c:a copy -pix_fmt yuv420p
MY_VIDEO.mp4
```



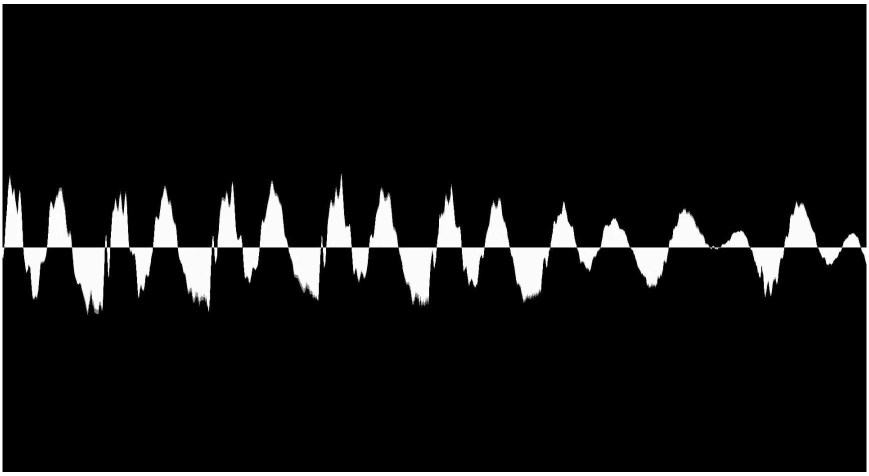
Style "Audio Frequency"

Style "Audio Frequency": to generate the Audio Frequency of your input source.

The formula is:

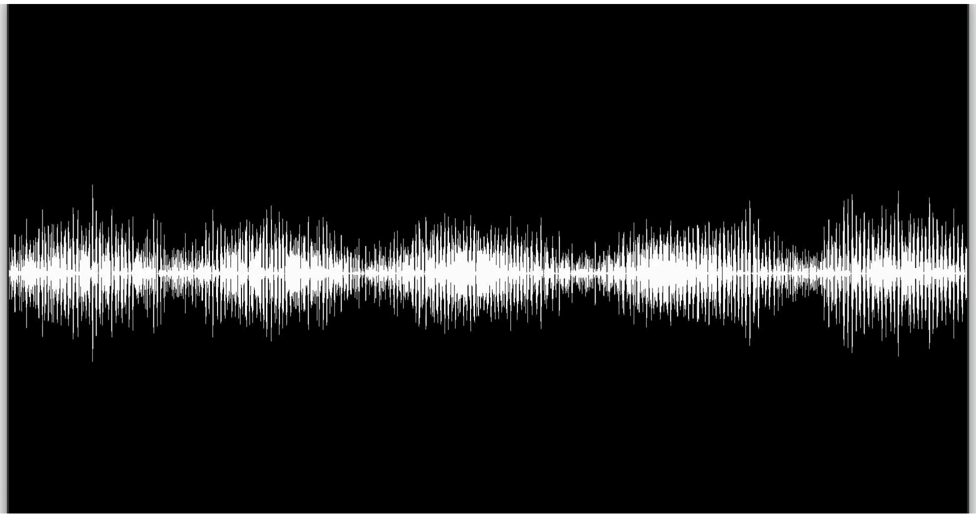
```
ffmpeg -i MY_AUDIO.aac -t 00:01:00
-filter_complex
"[0:a]showfreqs=s=1920x1080[v]" -map '[v]'
-map '0:a' -c:a copy -pix_fmt yuv420p
MY_VIDEO.mp4
```

Style "Black & White Lines": this will generate a simple black and white Waveform by taking your audio source and draw a vertical line for each sample.


Style "Black & White Lines"

```
ffmpeg -i MY_AUDIO.aac -filter_complex
"[0:a]showwaves=mode=line:s=1920x1080:colors
=white[v]" -map '[v]' -map '0:a' -c:a copy
-pix_fmt yuv420p MY_VIDEO.mp4
```

Style "Black & White Clines": this will generate a simple black and white waveform animation by taking your audio source and draw a centered vertical line for each sample.

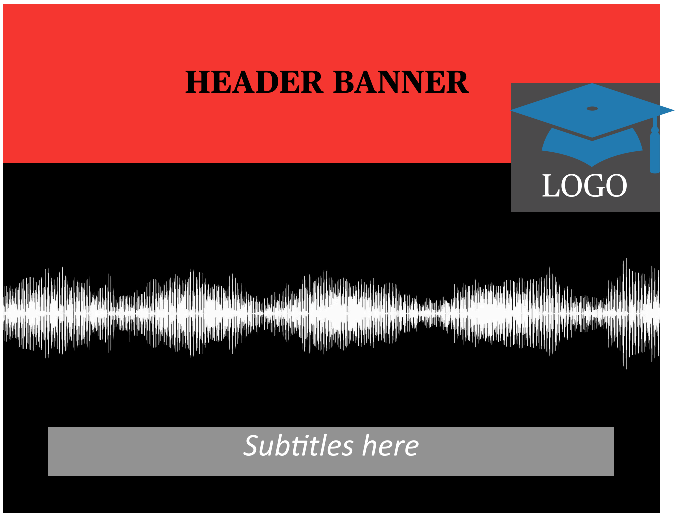

Style "Black & White Clines"

```
ffmpeg -i MY_AUDIO.aac -filter_complex
"[0:a]showwaves=mode=cline:s=1920x1080:color
s=white[v]" -map '[v]' -map '0:a' -c:a copy
-pix_fmt yuv420p MY_VIDEO.mp4
```

Generate Animated Video from Audio

In the previous chapter you have learned how to generate a waveform from an audio file and output a video.

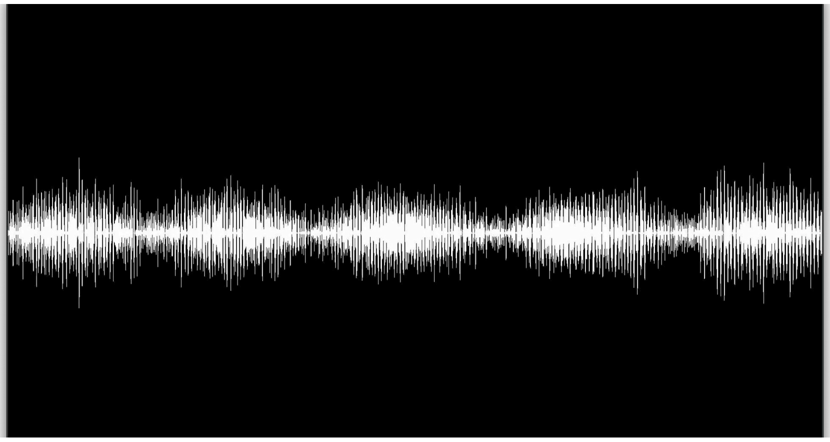
In this chapter we will go a bit deeper and we'll learn how to create a social media post, by using only your audio and the transcription, and by adding a backround image, and an overlay banner, such as in the example below.



This can become a great tool for content re-purposing. The first step will be to generate the audio waveform video with the desired style, as discussed in the previous chapter.

Example:

```
ffmpeg -i MY_AUDIO.aac -filter_complex
"[0:a]showwaves=mode=cline:s=1920x1080:color
s=white[v]" -map '[v]' -map '0:a' -c:a copy
-pix_fmt yuv420p MY_VIDEO.mp4
```



Style "Black & White Clines"

Second step will be to apply the transcription.

If your video is available on YouTube, you can extract the subtitles by using YouTube-DL and extracting the subtitles that you have created when you uploaded the video on YouTube.

The command for extracting the subitles is:

```
youtube-dl --write-sub --sub-lang en --skip-
download YOUR_YOUTUBE_VIDEO_URL
```

This command will download the English Language only of your available subtitles.

If you don't have any subtitles available, you can still download the auto-generated subtitles, created by the automatic voice recognition system of YouTube.

In this case the command will be as follow:

```
youtube-dl --write-auto-sub --skip-download
YOUR_YOUTUBE_VIDEO_URL
```

If you want to add a simple subtitle overlay you can convert the YouTube's **`.vtt`** format, into the SubStationAlpha format, with **`.ass`** extension.

```
ffmpeg -i YOUTUBE_SUBTITLES.vtt YOUR_SUB.ass
```

Then you can edit the **`.ass`** file with nano:

```
nano YOUR_SUB.ass
```

Then at the line "Style:" you can apply the desired style, for example:

```
Style:
Default,Lato,20,&H00FFFFFF,&H000000FF,&H80000000,&H80000000,-1,0,0,0,100,100,0,0,4,0,0,2,10,10,10,1
```

Alternatively you can use YouTube's `.vtt` (Web Video Text Track) to display words in real-time.

Create Animated Slides from Still Pictures

There is a basic way to create a slideshow with FFMPEG.

Sequential with Fixed time values

```
ffmpeg -framerate 1 -i img%03d.png -r 25
-pix_fmt yuv420p output.mp4
```

The above formula assumes that your input images are sequentially numbered with the following criteria:

```
img001.png
img002.png
img003.png
```

etc.

If you have a folder with different and mixed filenames you can use this formula to batch rename your files according to the prefix "`img000.extension`".

Just move yourself into your pictures directory and type:

```
ls | cat -n | while read n f; do mv "$f"
`printf "img%03d.png" $n`; done
```

You can replace **`img%03d.png`** with your desired extension, accordingly to your input file format.

The duration of each slide will depends on the -framerate option.

For example:

`-framerate 1`: a picture every second (1 fps)

`-framerate 1/2`: a picture every 2 seconds (0,50 fps)

`-framerate 1/3`: a picture every 3 seconds (0,33 fps)

`-framerate 1/5`: a picture every 5 seconds (0,20 fps)

`-framerate 1/10`: a picture every 10 seconds (0,10 fps)

If you need a faster or a slower framerate you can set your desired value with the **`-framerate`** option.

The above example assume 1 frame per each second.

The **`-r`** option will produce a 25 fps output file.

You can change this value accordingly to your output needs.

The **`-pix_fmt yuv420p`** option is for ensuring that your file will play across every major player, including Quicktime Player. This might produce a warning in your Terminal output, similar to this:

"deprecated pixel format used, make sure you did set range correctly"

This warning can be ignored, since it will process your files just fine.

To know more about **`-pix_fmt`** options, please take a look at this section of the FFMPEG guide:

https://trac.ffmpeg.org/wiki/Chroma%20Subsampling

Producing Slides with Ken Burns Effect[60]

A great technical article written by Remko Tronçon is available here and explains how to accomplish this task with FFMPEG.

60 Ken Burns effect refers to a wonderful pan and zoom effect introduced by an American director and documentarian, Ken Burns.

Extract Images from Video

This FFMPEG command will take a video input and extract still frames at a specific time interval.

Example Formula

(Images extracted every 5 seconds):

```
ffmpeg -i INPUT -f image2 -bt 20M -vf fps=1/5
%03d.png
```

Here's the command explanation:

`-i INPUT`: your input's file

`-f image2`: this is the FFMPEG's image muxer and writes video frames to image files.

`-bt 20M`: depending on your input's file format, this option will be required for PNG file format export and it's explained technically here. This option is called "Bitrate Tollerance" and in this example it's set at 20 Mbps.

`-vf fps=1/45`: this is the time interval value that can be adjusted accordingly to your needs (in this example, 1 picture every 45 seconds). The same can be achieved with the `-r` (rate) option (`-r 1/45`).

`%03d.png`: this will output a file titled with 3 digits, starting from "000.png"

Example formula for a specific width and height:

```
ffmpeg -i INPUT -r 1 -s 1280x720 -f image2 
%03d.jpeg
```

This will extract one video frame per second from the video and will output them in files named **`000.jpg`**, **`001.jpeg`**, etc. Images will be scaled to fit a specific (W)idth x (H)eight values, as in the above example at 1280x720.

Example Formula with Scaling, Padding and Letterbox every 45 seconds at 1280x720:

```
ffmpeg -i INPUT -r 1/45 -s 1280x720
-vf "scale=(iw*sar)*min(1280/(iw*sar)\,720/
ih):ih*min(1280/(iw*sar)\,720/ih),
pad=1280:720:(1280-iw*min(1280/iw\,720/ih))/
2:(720-ih*min(1280/iw\,720/ih))/2" -bt 10M
-f image2 %03d.png
```

The first filter is the scale filter (**`-vf scale`**).

The scale filter is powerful because it can perform both value substitution and math. In this case, the math computes the new scaled width and height. It will scale the input image to fit the width and/or height of the desired output format without distorting the image.

You won't need to make any modifications to the scale parameters because it will automatically substitute the actual values for "**`iw`**" (image width), "**`sar`**" (sample aspect radio) and "**`ih`**" (image height).

The second filter is the pad filter (**`-vf pad`**).

It has the same math/substitution features as the scale feature. So it will figure out the exact numbers for you.

You don't need to change anything to make it work.

The parameters tell the pad filter to make the output 720x1280 and to place the input image at the center of the frame. The pad filter will fill the output image with black anywhere that the input image doesn't cover. Black bars will be eventually added to the sides to make up the difference[61]. More info: https://ffmpeg.org/ffmpeg-filters.html#pad

If you want to extract just a limited number of frames, you can use the above command in combination with the **`-frames:v`** or **`-t`** option, or in combination with **`-ss`** to start extracting from a certain point in time, such in this example:

Starting from 00h:00m:15ss.00ms for 240 frames:

```
ffmpeg -ss 00:00:15.00 -i INPUT -frames:v 24
-f image2 -bt 10M %03d.png
```

Starting from time 00h:00m:15s.00ms and extract 5 seconds of pictures:

```
ffmpeg -ss 00:00:15.00 -i INPUT -t
00:00:05.00 -f image2 -bt 10M %03d.png
```

61 https://superuser.com/questions/891145/ffmpeg-upscale-and-letterbox-a-video

Extract Audio from Video

The formula for extracting an audio track is:

```
ffmpeg -i YOUR_VIDEO.mp4 -vn -c:a
copy youraudio.aac
```

In the above formula i'm assuming you have an H264 video along with an AAC audio track.

With the same command, but the relative extensions, you can extract audio tracks in mono, stereo or 7.1 Dolby AC3 formats. The formula above will also prevent the re-encoding of your source audio, unless you want to extract the audio and convert it at the same time.

In this case you will need to specify the audio encoder.

Taking the same example above, to extract an AAC audio track from a video and convert it to Mp3, you will need to type the following:

```
ffmpeg -i YOUR_VIDEO.mp4 -vn -c:a mp3 -b:a
128k -ac 2 -ar 48000 YOUR_AUDIO.mp3
```

This will produce a standard Mp3 file at 128 Kbit/s, 48000 kHz and, if the source has multiple audio tracks, such as in the Dolby AC3 format, it will produce a stereo mixdown of the source video. You can modify the **`-b:a`** (bitrate), the **`-ar`** (Audio Rate) and the **`-ac`** (Audio Channels) parameters, accordingly to your needings.

If you are unsure about the audio source of your video, but still you want to avoid a re-encoding of the audio, you can check your source video with the FFPROBE utility which is discussed previously in this book.

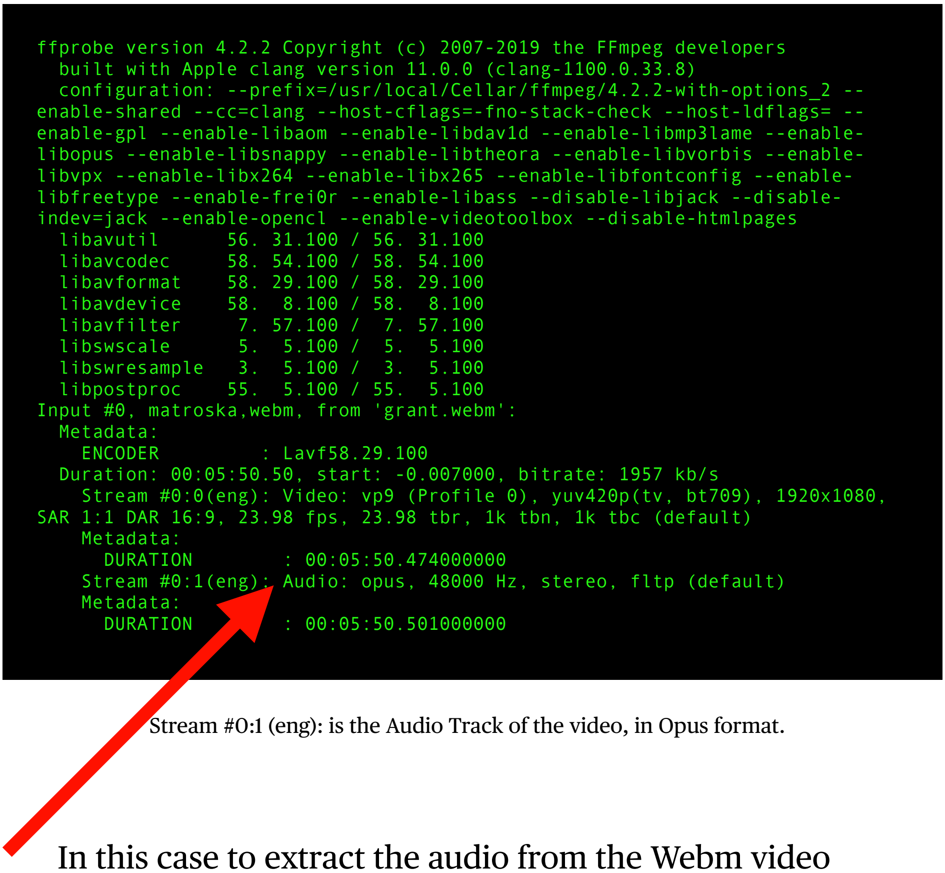
To check your source file just type:

```
ffprobe YOUR_VIDEO_SOURCE.[YOUR_EXTENSION]
```

For example: you might have a WEBM file. In this case, to check which format is the audio source, you can just type:

```
ffprobe video.webm
```

This will produce something like the following:



```
ffprobe version 4.2.2 Copyright (c) 2007-2019 the FFmpeg developers
  built with Apple clang version 11.0.0 (clang-1100.0.33.8)
  configuration: --prefix=/usr/local/Cellar/ffmpeg/4.2.2-with-options_2 --
enable-shared --cc=clang --host-cflags=-fno-stack-check --host-ldflags= --
enable-gpl --enable-libaom --enable-libdav1d --enable-libmp3lame --enable-
libopus --enable-libsnappy --enable-libtheora --enable-libvorbis --enable-
libvpx --enable-libx264 --enable-libx265 --enable-libfontconfig --enable-
libfreetype --enable-frei0r --enable-libass --disable-libjack --disable-
indev=jack --enable-opencl --enable-videotoolbox --disable-htmlpages
  libavutil      56. 31.100 / 56. 31.100
  libavcodec     58. 54.100 / 58. 54.100
  libavformat    58. 29.100 / 58. 29.100
  libavdevice    58.  8.100 / 58.  8.100
  libavfilter     7. 57.100 /  7. 57.100
  libswscale      5.  5.100 /  5.  5.100
  libswresample   3.  5.100 /  3.  5.100
  libpostproc    55.  5.100 / 55.  5.100
Input #0, matroska,webm, from 'grant.webm':
  Metadata:
    ENCODER         : Lavf58.29.100
  Duration: 00:05:50.50, start: -0.007000, bitrate: 1957 kb/s
    Stream #0:0(eng): Video: vp9 (Profile 0), yuv420p(tv, bt709), 1920x1080,
SAR 1:1 DAR 16:9, 23.98 fps, 23.98 tbr, 1k tbn, 1k tbc (default)
    Metadata:
      DURATION        : 00:05:50.474000000
    Stream #0:1(eng): Audio: opus, 48000 Hz, stereo, fltp (default)
    Metadata:
      DURATION        : 00:05:50.501000000
```

Stream #0:1 (eng): is the Audio Track of the video, in Opus format.

In this case to extract the audio from the Webm video you will type:

```
ffmpeg -i YOUR_VIDEO.webm -vn -c:a copy
audio.opus
```

Replace Audio of a Video

If you have a video and wants to replace the audio, with another audio, you can do it simply using FFMPEG's **`-map`** option.

An example formula will be as follow:

```
ffmpeg -i your_source_video.mp4 -i
your_source_audio.mp4 -map 0:v -map 1:a -c
copy final.mp4
```

`-i your_source_video.mp4`: this is an example video input file

`-i your_source_audio.mp4`: this is the audio example file

`-map 0:v`: this will specify that the first input is the video source (FFMPEG starts counting from 0)

`-map 1:a`: this will specify that the second input is the audio source

-c copy: this command will avoid to re-encode the source files

`final.mp4`: this is an output example file

Batch Convert Audio Files to a specific format

Consider the following scenario.

You have a folder full of WAV files, stereo at 24 Bit/96 KHz resolution and you want to convert it to the FLAC Lossless format (**F**ree **L**ossless **A**udio **C**odec) and keep the original metadata informations, such as Artist, Title, etc., of each file.

This will be performed with the following BASH script.

Open Terminal and go to your working directory.
For example you have your "WAV" Folder named "WAV" inside within your Desktop.

You will need to type:

```
cd ~/Destkop/WAV
```

Then create a new file with nano. Let's call it "`wav.sh`"

```
nano wav.sh
```

In the NANO window type the following code:

```
mkdir Processed
for i in *.wav
do ffmpeg -i "$i" -nostdin -map_metadata:s:a
0:s:a
-c:a flac "./Processed/${i%.*}.flac"
done
```

Save the document and exit.

Then type the following command and hit enter:

```
source wav.sh
```

Let's breakdown the script:

`mkdir Processed`

This will create a folder within the working directory called "Processed" (you can name it as you wish)

`for i in *.wav`

Find every input in the working directory with the *.wav extension

```
  do ffmpeg -i "$i" -nostdin
-map_metadata:s:a 0:s:a
  -c:a flac
"./Processed/${i%.*}.flac"
done
```

This will perform the FFMPEG conversion from WAV format to Free Lossless Audio Codec (FLAC) and will preserve the original metadata (information) of the source file and will copy any video information contained in the WAV file.

The final files will have the required `.flac` extension and they will be putted into the "`Processed`" folder previously created. Everything will stop after each WAV file will be converted.

Batch Convert Audio Files in Multiple Formats

Let's take the previous example and let's examine a similar case scenario. You have a folder containing lots of files in **`.wav`** format. This time you want to batch convert all the WAVs in multiple formats, for example in AIFF, Mp3 @64Kbps, Mp3 @128Kbps, AAC @192 Kbps and in FLAC format.

In this case you'll have to create a new Bash script.

Let's call it "`multiaudio.sh`"

```
nano multiaudio.sh
```

Now let's write the following script:

```
#!/bin/bash
mkdir aiff mp3-64 mp3-128 aac-192 flac
for i in *.wav; do ffmpeg -i "$i" -nostdin
-map_metadata:s:a 0:s:a "./aiff/$
{i%.*}.aiff" -b:a 64k "./mp3-64/$
{i%.*}.mp3" -b:a 128k "./mp3-128/$
{i%.*}.mp3" -b:a 192k "./aac-192/$
{i%.*}.aac" "./flac/${i%.*}.flac"
done
```

Let's breakdown this script.

```
#!/bin/bash
```

Most versions of Unix make this a part of the operating system's command execution mechanism. If the first line of a script begins with the two characters '#!', the remainder of the line specifies an interpreter for the program. Thus, you can specify Bash, or some other interpreter and write the rest of the script file in that language.[62]

```
mkdir aiff mp3-64 mp3-128 aac-192 flac
```

this will create 5 folders named respectively *aiff*, *mp3-64*, *mp3-128*, *aac-192* and *flac*.

```
for i in *.wav; do ffmpeg -i "$i" -nostdin
```

This will check for every .WAV file in the working directory and then it will process it thru FFMPEG. You can modify this input extension, with your desired extension or you can add another extension if you wish. (E.g.: *.aiff or

62 https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Syntax

*.wav *.aiff *.wma, etc.)

The **`nostdin`** option tells FFMPEG to ignore the console input while performing this script. FFMPEG normally checks the console input, for entries like "**`q`**" to stop and "**`?`**" to give help, while performing operations. FFMPEG does not have a way of detecting when it is running as a background task. When it checks the console input, that can cause the process running ffmpeg in the background to suspend. To prevent those input checks, allowing ffmpeg to run as a background task, use the **`-nostdin`** option in the ffmpeg invocation[63].

```
  -map_metadata:s:a 0:s:a
```

This will specify to copy all the available metadata on the first audio stream available on the input file, across every output file.

```
  "./aiff/${i%.*}.aiff" -b:a 64k "./mp3-64/$
{i%.*}.mp3" -b:a 128k "./mp3-128/$
{i%.*}.mp3" -b:a 192k "./aac-192/$
{i%.*}.aac" "./flac/${i%.*}.flac"
```

[63] https://ffmpeg.org/faq.html#toc-How-do-I-run-ffmpeg-as-a-background-task_003f

```
=
```

This will tell FFMPEG to place the output file on the respective directory, and also will give to FFMPEG all the parameters for each format.

```
done
```

This will end the script once completed or in case of an error.

Audio Loudness Normalization for TV Broadcast

This formula takes the previous one, and will apply a standard audio normalization filter. The Audio Loudnorm Normalization Filter which is a standard in the broadcast field, as per standard EBU 128R Specification[64].

```
#!/bin/bash
mkdir aiff mp3-64 mp3-128 aac-192 flac
for i in *.wav; do ffmpeg -i "$i" -nostdin 
-map_metadata:s:a 0:s:a -af loudnorm "./
aiff/${i%.*}.aiff" -b:a 64k "./mp3-64/$
{i%.*}.mp3" -b:a 128k "./mp3-128/${i%.*}.m$
done
```

The above script is just like the previous formula, but it will apply the LoudNorm Audio Filter **`-af loudnorm`** on the input, so to produce multiple audio conversions with loudness normalization according to the standard EBU R 128 standard.

64 https://tech.ebu.ch/docs/r/r128.pdf

Audio Loudness Normalization for Amazon Alexa and Google Assistant (Audiobooks/Podcasts)

LUFS (Loudness Units relative to Full Scale) is a standard that measures the perceived loudness[65] of an audio file. Volume Normalization is the application of a constant amount of gain to an audio recording so to bring the amplitude to a target level, the norm, across many genres and production styles.

LUFS is a complicated algorithm based on perceived loudness of human hearing at a comfortable listening volume and lets audio producers avoid jumps in amplitude that would require users to constantly adjust volume. LUFS is also referred as LKFS
(Loudness, K-weighted, relative to Full Scale)[66].

Amazon Alexa, Google Home and every modern vocal assistant requires specific loudness ranges in order to accept audio materials from content providers.

---

[65] A subjective perception of sound pressure (https://www.gcaudio.com/tips-tricks/the-relationship-of-voltage-loudness-power-and-decibels/)

[66] https://developers.google.com/assistant/tools/audio-loudness

Amazon Alexa requires the following parameters:

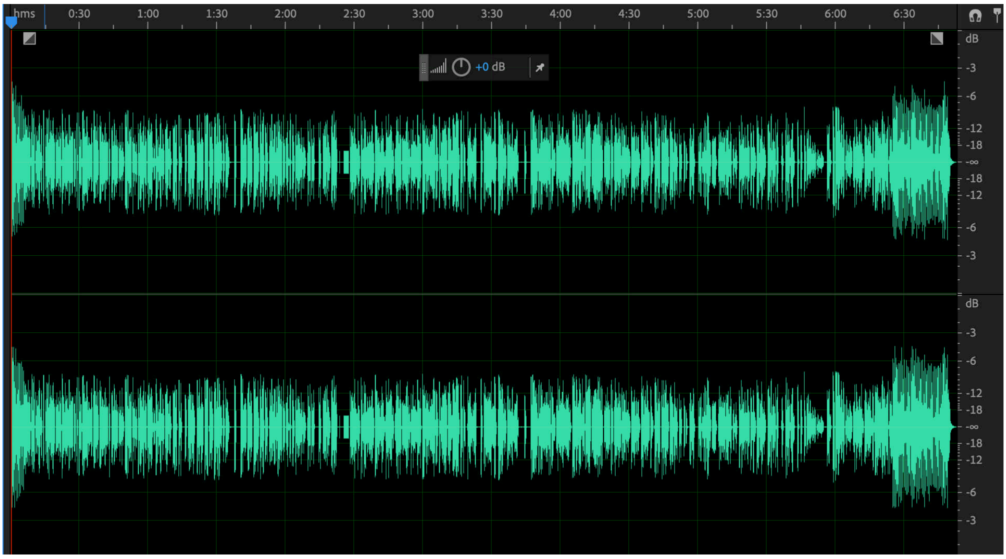
"Program loudness for Alexa should average -14 dB LUFS/LKFS."

So in order to produce standard audio files for Alexa Skills, you will need to produce a standard audio with average of -14dB LUFS.

In order to make FFMPEG process an audio file and process it according to the standard required, you will need to perform 2 steps:

1) measure your current Loudness, or the intensity of your sound;

2) adjust the measured audio parameters so to meet the Amazon requirements.



test.wav - Original Speech Audio Waveform

Let's take an audio file ("`test.wav`") and let's measure the current loudness:

```
ffmpeg -i test.wav
-af
loudnorm=I=-14:TP=-3:LRA=11:print_format=sum
mary -f null -
```

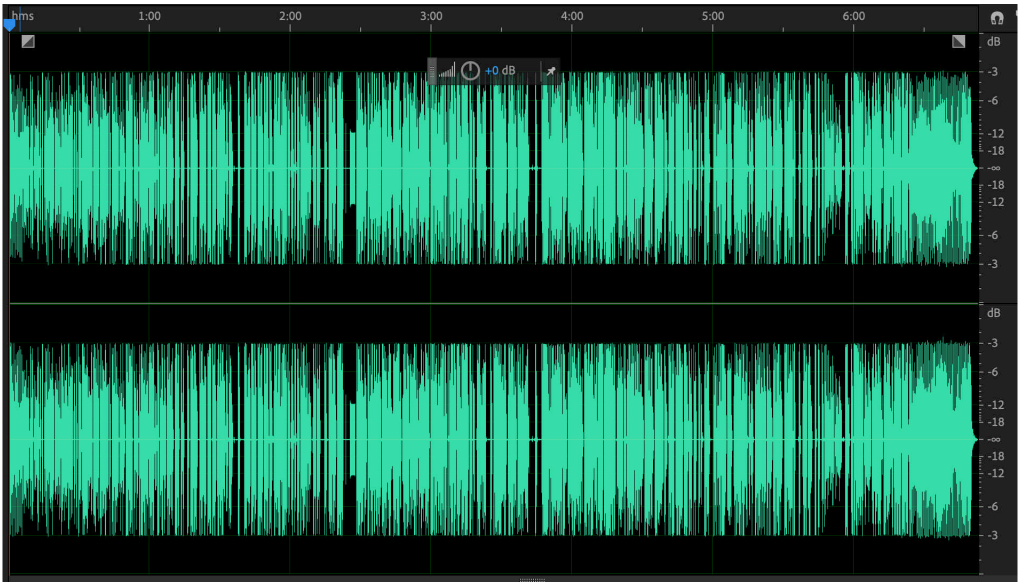
This will output a print summary of the measured loudness of the "test.wav" file, such as this one:

```
Input Integrated:    -21.9 LUFS
Input True Peak:      -4.2 dBTP
Input LRA:             8.1 LU
Input Threshold:     -32.5 LUFS

Output Integrated:   -14.7 LUFS
Output True Peak:     -3.0 dBTP
Output LRA:            5.1 LU
Output Threshold:    -25.3 LUFS

Normalization Type:   Dynamic
Target Offset:        +0.7 LU
```

In this example we will need to adjust the values, so to match the values required by Amazon Alexa, and we will include the values measured in the first pass of the previous FFMPEG command, and this time producing a final output of the file.

```
ffmpeg -i test.wav -af
loudnorm=I=-14:TP=-3:LRA=11:measured_I=-21.9:measured
_TP=-4.2:measured_LRA=8.1:measured_thresh=-32.5:offse
t=+0.7:linear=true:print_format=summary test-
amazon.wav
```

Running again the first command, by measuring the output file, in this audio example the Terminal will output the following values:



test.wav - Processed Audio

```
Input Integrated:    -15.1 LUFS
Input True Peak:      -2.4 dBTP
Input LRA:             5.5 LU
Input Threshold:     -25.8 LUFS

Output Integrated:   -14.2 LUFS
Output True Peak:     -3.0 dBTP
Output LRA:            5.0 LU
Output Threshold:    -24.8 LUFS

Normalization Type:   Dynamic
Target Offset:        +0.2 LU
```

For Google Assistant, the requirements are described in the Google Developers Website:

Measure and adjust the loudness of your audio so that it has an integrated average loudness of about -16 LUFS (or -19 LUFS if the content is mono).

Taking the previous 'test.wav' audio file, we will then produce a file that has -16 dB LUFS (please note that as per Google Reccomendations you will have to replace the value of -16 to the value of -19 if you are working with mono files).

```
ffmpeg -i test.wav -af 
loudnorm=I=-16:TP=-3:LRA=11:measured_I=-21.9
:measured_TP=-4.2:measured_LRA=8.1:measured_
thresh=-32.5:offset=+0.7:linear=true:print_f
ormat=summary test-google.wav
```

To wrap-up: a standard Audio Loud Normalization for Amazon Alexa and Google Assistants needs two passes, one for audio measurement and the other for the actual processing.

Batch Audio Loudness Normalization for Amazon Alexa (AudioBooks/Podcasts)

If you have tens or thousands of audio files that needs audio loudness normalization you will need to automate both the steps of audio measurements and the actual audio processing. In order to do so, there is a great Python program that can solve our batch needings. The name is **ffmpeg-normalize.**

To install it you will need first to install Pyhton 3 for Ubuntu. Just type the following command:

```
sudo apt update
sudo apt install python3-pip
```

When done, just type the following command:

```
pip3 install ffmpeg-normalize
```

Please note that FFMPEG-Normalize **will not install FFMPEG**, but it will use your existing FFMPEG installation.

If you don't have FFMPEG installed already, please refer to Chapter 1 in order to go thru the installation process.

In order to convert a file that meets the Amazon Alexa requirements, we can use this single line command example: (Note: don't use quotation "" symbol on MacOSX)

```
ffmpeg-normalize [INPUT].wav -nt ebu -t
"-14" -lrt "11" -tp "-3" -ar 44100 -v -o
[OUTPUT].wav
```

Let's breakdown this command:

`[INPUT].wav`

this is your complete file's path

`-nt ebu`

Normalization type (`ebu` is the default option, but you can choose between `ebu`, `rms` or `peak`)

`-t "-14"`

Normalization target level in dB/LUFS. For EBU normalization, it corresponds to Integrated Loudness Target in LUFS. Amazon reccomends an integrated value of

-14, wihle Google reccomends a value of -16 for stereo files and a value of -19 for mono files.

```
-lrt "11"
```

Loudness Range Target. EBU Loudness Range Target in LUFS (default: 7.0). Amazon Alexa and Google reccomends a value of 11.

```
-tp "-3"
```

True Peak Value. EBU Maximum True Peak in dBTP. Amazon reccomends a value of -3 dB, while Google reccomends a value of -1.5 dB.

```
-ar 44.100
```

That's the audio rate specifier. (By default FFMPEG-Normalizer will output the audio file at 192KHz).

```
-v
```

Verbose: meaning that FFMPEG-Normalize will output in text format every step accomplished, on your shell.

```
-o [OUTPUT].wav
```

This is your output's path.

Please note: FFMPEG-Normalize has a "Dual Mono" option. This will treat mono input files as "dual-mono".

If a mono file is intended for playback on a stereo system, its EBU R128 measurement will be perceptually incorrect. If set, this option will compensate for this effect. Multi-channel input files are not affected by this option.

For automating this process (e.g.: you need to loudness normalize thousands of WAV files, with different levels and measurements), you can easily use FFMPEG-Normalize with a single command, without using necessarily a BASH script.

Let's make an example, with the following command:

```
ffmpeg-normalize *.wav -nt ebu -t "-14" -lrt
"11" -tp "-3" -ar 44100 -v -ext .wav
```

With the above command, FFMPEG-Normalize will automatically create a folder entitled "*normalized*" containing all the processed WAV files according to the specifics of Amazon Alexa. You can specify the output file extension with the option **`-ext`** (in this example *-ext.wav),* otherwise FFMPEG-Normalize will use the MKV container by default, as the file extension.

You might want to output a different format than the original one: in this case you just have to specifiy che **`-c:a`** (codec audio) argument, with your desired parameters, such as the example below:

```
ffmpeg-normalize *.wav -nt ebu -t "-14" -lrt
"11" -tp "-3" -c:a aac -ar 192000 -v -o
output.aac
```

You can also apply additional FFMPEG arguments, just like if you were using the regular FFMPEG (FFMPEG-

Normalize is using your existing installation of FFMPEG in order to work).

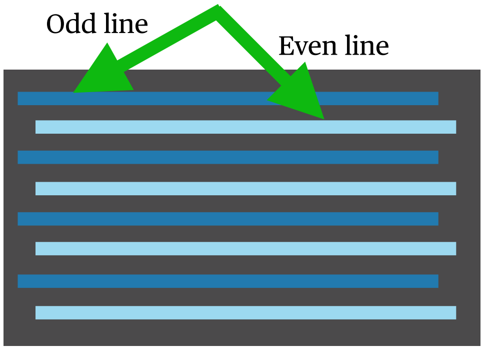
Of course FFMPEG-Normalize will work also with video inputs: that means that you can process and normalize the audio of a video, according to the same standards above mentioned, without re-process the video itself.

In this example a video file called "`video.mp4`" will be processed with the Amazon Alexa requirements, without process the video itself (thus using **`-c:v copy`**) and by converting the existing audio stream to the AAC format at 128 Kbps, 44.100 KHz.

```
ffmpeg-normalize video.mp4 -nt ebu -t "-14"
-lrt "11" -tp "-3" -c:v copy -c:a aac -b:a
128k -ar 44100 -v -o output.mp4
```

Once again: please note that the default output container used by FFMPEG-Normalize is the MKV container, unless you specify a container with the **`-o`** sing the **`-ext`** option, you can supply a different output extension common to all output files.

De-Interlacing Filter - 13 FFMPEG solutions



The RCA Victor Engineer Randall Ballard invented the technique of interlacing pictures in 1932. It's a way to display a full picture by dividing it into 2 different set of lines: the even and the odd lines.
The even lines are called "even field", while the odd lines are called the "odd field".

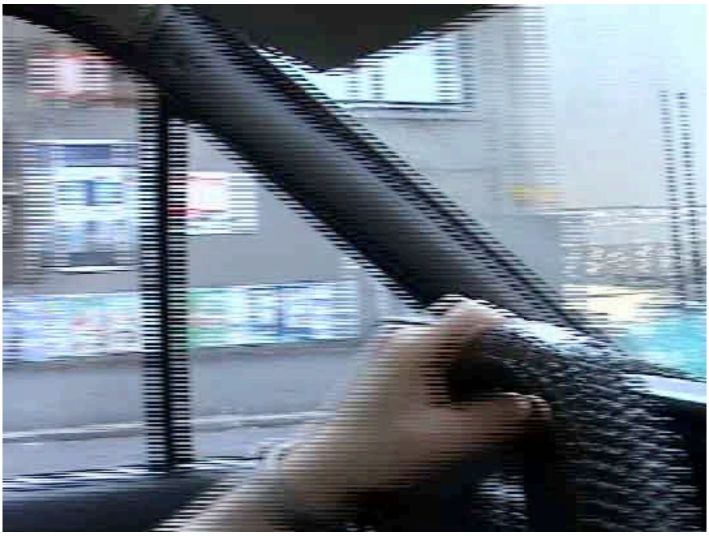
The even lines are displayed on the screen, then the odd lines are displayed on the screen, each one every 1/60th of a second: both of these, even and odd fields, make up one video frame.

The numbers 24, 25, 29.97, 30, 50, 59.94 and 60 represent how many frames are displayed per each second. You may see also letters near the fps value: 24i, 50i or 25p.

These letters shows if the video is displayed using an interlaced (i) or a progressive (p) format.

Progressive format refers to video that displays both the even and odd lines (the entire video frame) at the same time.

When you have to deal with TV broadcast footage (or any interlaced format) on a computer screen, you will usually see artifacts or blurred lines, especially during camera movements, such as the sample picture below:



In order to convert an interlaced source into a progressive format, FFMPEG offers 13 different approaches or solutions.

Each of one it's called by the ***-vf*** parameter (video filter) along with the name of the desired de-interlacing algorithm.

The solutions (in no particular order) are:

The YADIF Method

• **YADIF**: **Y**et **A**nother **D**e-**I**nterlace **F**ilter

(Reference: https://ffmpeg.org/ffmpeg-filters.html#yadif-1)

```
ffmpeg -i [YOUR_INTERLACED_VIDEO] -vf yadif
[OUTPUT_RESULT]
```

The BWDIF Method

• **BWDIF**: **B**ob **W**eaver **D**e-**I**nterlacing **F**ilter

(Reference: https://ffmpeg.org/ffmpeg-filters.html#bwdif)

```
ffmpeg -i [YOUR_INTERLACED_VIDEO] -vf bwdif
[OUTPUT_RESULT]
```

The Kerndeint Method

• **KERNDEINT:** a kernel (image processing calculation) created by Donald Graft (http://www.volter.net/avisynth_en/externalfilters/kerneldeint.htm)

(Reference: https://ffmpeg.org/ffmpeg-filters.html#kerndeint)

```
ffmpeg -i [YOUR_INTERLACED_VIDEO] -vf
kerndeint [OUTPUT_RESULT]
```

The MCDEINT Method

- **MCDEINT**: Motion Compensation De-Interlacing

(Reference: https://ffmpeg.org/ffmpeg-filters.html#mcdeint)

This filter works in conjunction with the YADIF filter, thus you will need to call the YADIF filter before applying the MCDEINT filter, although this solution can become painfully slow. An example code is here below:

```
ffmpeg -i [YOUR_INTERLACED_VIDEO] -vf
yadif=1:-1:0,mcdeint=2:1:10 [OUTPUT_RESULT]
```

The NNEDI Method

- **NNEDI**: Neural Networks Edge Directed Interpolation

(Reference: https://ffmpeg.org/ffmpeg-filters.html#nnedi)

This approach takes every single frame, throws away one field, and then interpolates the missing pixels using only information from the remaining field.

It is also good for enlarging images by powers of two.

In order to use NNEDI you must this particular library from Github: **https://github.com/dubhater/vapoursynth-nnedi3/**

The PP Method

• PP: Post-Processing Subfilters

(Reference: https://ffmpeg.org/ffmpeg-filters.html#pp)

This is an approach that use chained filters and has 6 different methods of de-interlacing.

The actual code is executed by using one of the following **`-vf pp`** options below. For example:

```
ffmpeg -i [YOUR_INTERLACED_VIDEO] -vf pp=lb/
linblenddeint [OUTPUT]
```

The 6 options are:

```
-vf pp=lb/linblenddeint
-vf pp=li/linipoldeint
-vf pp=ci/cubicipoldeint
-vf pp=md/mediandeint
-vf pp=fd/ffmpegdeint
-vf pp=l5/lowpass5
```

The tech specs of each filter is explained on the PP section of the FFMPEG guide.

The W3DFIF Method

- **W3DFIF**: Weston 3 Field Deinterlacing Filter

Based on the process described by Martin Weston for the Research & Development department of the British Broadcasting Company, and implemented based on the de-interlace algorithm written by Jim Easterbrook for BBC R&D, the Weston 3 field deinterlacing filter uses filter coefficients calculated by BBC R&D.

(Reference: https://ffmpeg.org/ffmpeg-filters.html#w3fdif)

```
ffmpeg -i [YOUR_INTERLACED_VIDEO] -vf w3fdif [OUTPUT]
```

How to make a high-quality GIF from a video

These are the steps required to produce an high quality GIF from a video.

1. Download your desired YouTube Video with YouTube-dl

```
youtube-dl YOUR-YOUTUBE-VIDEO-URL
```

2. Cut the desired portion of the downloaded video with FFMPEG by using the -SS before the -i and by using the -t option after the input. In this example we will cut the video at 29 minutes and 18 seconds, and we will keep 10 seconds after that. Example:

```
ffmpeg -ss 00:29:18 -i youtubevideo.mkv -t 
00:00:10 -avoid_negative_ts 1 output.mp4
```

2B. SUGGESTED: Resize the original video from HD to SD (or less such 640x360)

```
ffmpeg -i output.mp4 -vf scale=640x360 -c:v
h264 -crf 18 -c:a aac outputscaled.mp4
```

3. Create an SRT Subtitle File with NANO

```
nano mycaptions.srt
```

Use the following syntax in order to produce a standard SRT file and make sure to sync your text with the video:

```
1
00:00:00,00 --> 00:00:03,00
This could be the first sentence

2
00:00:03,00 --> 00:00:06,00
and this could be the second sentence.

3
00:00:06,00 --> 00:00:09,00
Maybe you should need to write a third sentence,
```

Please note carefully the above example and pay attention at the spaces between each line, between the timecode, pay attention to the commas in the timecode and create the arrow --> by typing twice the dash button on your keyboard and the ">" symbol once, as above.

An error on the syntax will make a non-standard SRT file, producing errors on the next steps.

4. Save the SRT File

5. Convert the SRT to ASS Format

```
ffmpeg -i mycaptions.srt mycaptions.ass
```

6. Open the output `.ass` format with Nano (or your desider Text editor) and modify the size and/or the font as desired. The following example avoids overlapping lines and create a gray overlay boxed in the background:

```
Style: 
Default,Avenir,20,&H00FFFFFF,&H000000FF,&H80
000000,&H80000000,-1,0,0,0,100,100,0,0,4,0,0
,2,10,10,10,1
```

7. Burn the `.ass` subtitle into the video

```
ffmpeg -i outputscaled.mp4 -filter_complex 
"subtitles=sub.ass" videosubtitled.mp4
```

8. Create PNG Conversion for GIF Elaboration (with Gifski)

```
ffmpeg -i videosubtitled.mp4 frame%04d.png
```

9. Make the GIF (High Quality with Gifski)

```
gifski -o myvideo.gif frame*.png
```

9B. OPTIONAL: use FFMPEG (Low Quality Output)

```
ffmpeg -i videosubtitled.mp4 mygif.gif
```

How to add an Overlay Banner and burn subtitles onto a video

Step 1: Create your desired banner with your favourite graphic software. Set the size. 1920x220 would be an exmple size for placing a header onto a Full-HD video.

Step 2: Export the graphic in PNG format

Step 3: Overlay the graphic on top of your video. In this example the desired overlay graphic will have a time of 10 seconds before disappearing.

```
ffmpeg -i videoinput.mp4 -i imageinput.png
-filter_complex "[0:v][1:v]
overlay=0:0:enable='between(t,0,10)'"
-pix_fmt yuv420p -c:a copy output.mp4
```

In the previous Formula we generated a subtitle, and burned it onto the video. We might then want to add the graphic overlay we generated in this example. So the formula for achieving this will be:

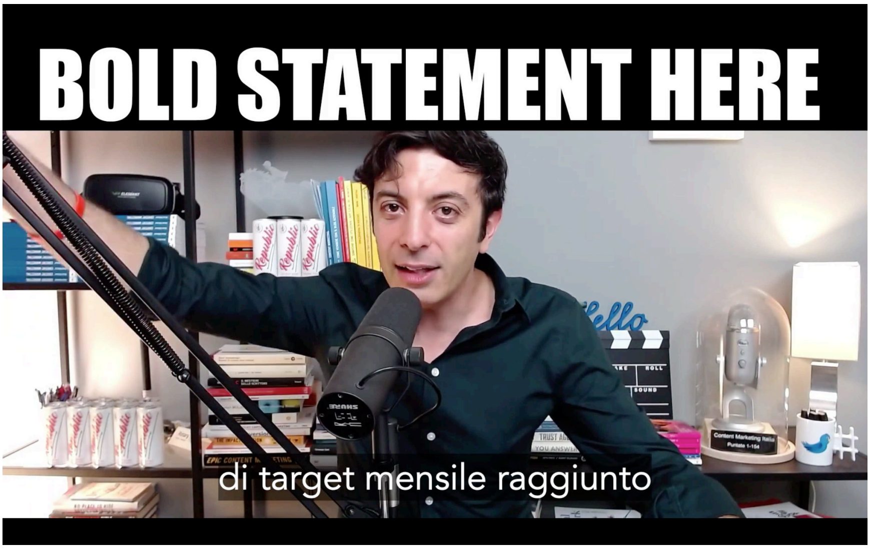
```
ffmpeg -i videoinput.mp4 -i 
yourgraphicheader.png -filter_complex "[0:v]
[1:v] 
overlay=0:0:enable='between(t,0,10)',subtitl
es=sub.ass" -pix_fmt yuv420p -c:a copy 
output.mp4
```

Again: you can remove the following code if you want to have the graphic overlay across the entire duration of your video. The code will be:

```
ffmpeg -i videoinput.mp4 -i 
yourgraphicheader.png -filter_complex "[0:v]
[1:v] overlay=0:0,subtitles=sub.ass"
-pix_fmt yuv420p -c:a copy output.mp4
```

BOLD STATEMENT HERE

A Banner Example



Final Composition with burned subtitles

Picture Courtesy of Alessio Beltrami - Copyright © 2020 Alessio Beltrami.

How to extract VTT files (Web Video Text Track) and burn it onto a video as a subtitle

Step 1: Download the VTT fil, typically multiple VTT files inside an `m3u8` playlist:

```
ffmpeg -i “https://URL-PATH/subtitles/italian/index.m3u8" subtitles.vtt
```

Step 2: Check the syntax of the VTT File with Nano or your favourite text Editor. You might want to check for words such as “`WEBVTT`” or “`X-TIMESTAMP-MAP=LOCAL:00:00:00.000,MPEGTS:120000`” or any additional extra code. You need to remove this extra data, otherwise it will be displayed onto your final subtitles:

```
nano out.vtt
```

Step 3: Convert the VTT file into .SRT format

```
ffmpeg -i subtitles.vtt subtitles.srt
```

Step 4: Burn your subtitle .srt file with FFMPEg

```
ffmpeg -i yourvideo.mp4 -filter_complex
“subtitles=subtitles.srt" -c:a copy
videosubtitled.mp4
```

Note: you may want to avoid re-encoding the audio stream of your source video, in order to speed up things while burning your subtitles, that's why i use **`-c:a copy`** in the above example. Alternatively you might want to convert your input with a specific video codec and a bitrate.

The above example will produce a standard h264 file, with FFMPEG's default settings for h264 format.

Change the style of your subtitles

To get a more professional look on the subtitling style you may want to change the font and perhaps adding a boxed-overlay as a background.

In this case i suggest you to convert your original VTT subtitle file, in the SSA format (that stands for Sub Station Alpha Format).

To convert from **`.vtt`** to **`.ass`** format, just type:

```
ffmpeg -i source-subtitles.vtt subtitles.ass
```

Now open the output **.ass** file with Nano or your desired Text editor, and modify the size and/or the font as desired in the "Style" section of it.

The following example avoids overlapping lines and create a gray overlay boxed in the background:

```
Style: Default,Gill
Sans,20,&H00FFFFFF,&H000000FF,&H80000000,&H8
0000000,-1,0,0,0,100,100,0,0,4,0,0,2,10,10,1
0,1
```

Save the file and close Nano.

Now you can type the following command:

```
ffmpeg -i inputvideo.mp4 -vf
"ass=inputsubtitle.ass" -c:a copy output.mp4
```

UBUNTU: In Ubuntu Linux, font files are installed in

`/usr/lib/share/fonts` or `/usr/share/fonts`.

The former directory is recommended in the case of manual installation.

To manually install a downloaded font you can create a directory such as "myfonts", like this:

```
sudo mkdir /usr/local/share/fonts/myfonts
```

Now you can move all your downloaded fonts into this new created directory.

Then you will need to set permissions to the new created directory, in order to read, write and execute the fonts, as follows:

```
sudo chown root:staff /usr/local/share/
fonts/myfonts -R
```

```
sudo chmod 644 /usr/local/share/fonts/
myfonts/* -R
```

```
sudo chmod 755 /usr/local/share/fonts/
myfonts
```

And finally builds font information caches for apps using fontconfig for their font handling, with the following command:

```
sudo fc-cache -fv
```

Now open the **`.ass`** subtitle file and insert your desired font:

```
Style: Default,Gill
Sans,20,&H00FFFFFF,&H000000FF,&H80000000,&H8
0000000,-1,0,0,0,100,100,0,0,4,0,0,2,10,10,1
0,1
```

And, finally, execute the FFMPEG command in order to burn the subtitles with your desired style and boxed overlay:

```
ffmpeg -i inputvideo.mp4 -vf
"ass=inputsubtitle.ass" -c:a copy output.mp4
```

As mentioned previously, the above command will produce a standard h264 file: you might want to change the video codec settings and the audio codec settings, accordingly to your needs.

Automatic Transcriptions and Subtitles

When you finish to upload a video on YouTube, using YouTube Studio, several process are performed such as conversion in multiple format and resolutions, compression in multiple formats and so forth.

One thing that might be unknown is that YouTube is also using algorithms for speech recognition of the language of your video. YouTube call this "ASR": Automatic Speech Recognition. The feature is available in *English*, *Dutch*, *French*, *German*, *Italian*, *Japanese*, *Korean*, *Portuguese*, *Russian*, and *Spanish*. Google also states that ASR is not available for all videos[67].

Please note that a complete transcription of a video could be available several hours later, after the upload completion.

In order to download the complete automatic transcription of your video, by accessing the YouTube Studio dashboard with a PC, you will find a section called

67 https://support.google.com/youtube/answer/7296221?hl=en

"Subtitles" which displays the subtitles available for each of your videos. YouTube will display a "Published - Automatic" option. By clicking next to this, you will have 3 options: "*Modify on the Classic Version of YouTube*", "*Download*" or "*Delete*".

By pressing "*Download*" you will be able to download the complete SBV file (SubViewer Format[68]) of the automatic transcription of your video, recognized by YouTube.

SBV or SUB both stands for “SubViewer.”

This is a very simple YouTube caption file format that doesn’t recognize style markup.

It’s a text format that is very similar to SRT (SubRip Subtitle File).

From time to time, unfortunately, the SBV file provided by YouTube seems to have timecode issues, producing overlapping subtitles: thus you will need first to select "Modify on the Classic Version of YouTube". In this way you will be also able to fix errors and more generally look at the automatic transcription, to see if everything has been understood by YouTube the way it's meant to be.

68 https://wiki.videolan.org/SubViewer

When you will finish to edit the subtitle in the "Classic Version of YouTube", you will be able to save the resulting file in different file formats: the original format, or in `.vtt` format, or in `.srt` format or in `.sbv` format.

For this formula i suggest you to download the caption file in the SRT file format.

Now that you have your clean and corrected SRT file you can then perform 2 useful things:

1) Burn the subtitle on the video, so to use it across different social media;

2) Create a Text file containing the full transcription of the video for blog or news articles, books and other useful purposes.

"Burn" the subtitle:

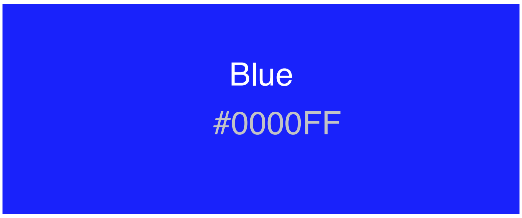
```
ffmpeg -i yourvideo.mp4 -filter_complex
“subtitles=captions.srt" -c:a copy
videosubtitled.mp4
```

You might want to change the default style of subtitles. In this case you can adjust the parameters with an example code like this one:

```
ffmpeg -i input.mp4 -filter_complex
"subtitles=captions.srt:force_style='Fontnam
e=Lato,Fontsize=28,PrimaryColour=&HFF0000&'"
-c:a copy output.mp4
```

PrimaryColour is expressed in hexadecimal[69] or HEX Color, in **B**lue **G**reen **R**ed order. Note that this is the *opposite* order of HTML color codes. Color codes must always start with `&H` and end with `&`.

Example: the HEX color code **`0000FF`** (for Blue), must be written like this: **`&HFF0000&`**



69 The Hexadecimal numeric format uses the decimal numbers and six extra symbols

Create a text file from the SRT file

This can be useful in case you want to re-use the audio of your speech for other destinations, in the form of written words.

To achieve this you will need Sublime Text, or any other Text Editor that can perform Regex[70] operations.

The sequence of actions will be:

1) Download the subtitle in .srt format from your YouTube Studio;

2) Wipe the SRT formatting, the unwanted timecodes, and the numeric symbols with Sublime Text or other Text Editor that supports RegEx;

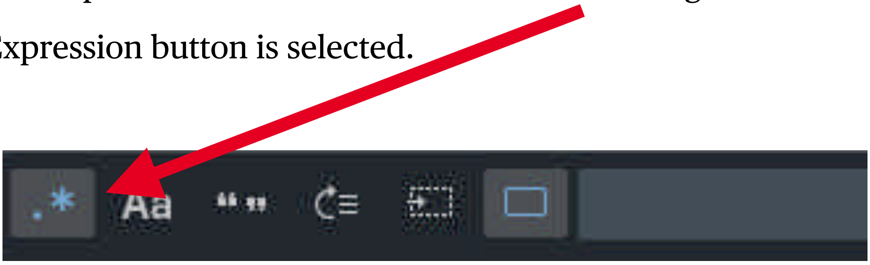
3) Format the text for human reading with Sublime and paste it on your desired Text Editor or on your Webpages, Blog Articles, etc.

[70] A **Reg**ular **Ex**pression (shortened as regex or regexp) is a sequence of characters that define a search pattern in a text file. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

Sublime Text on MacOS X

Open the .srt file with Sublime Text

The steps number 2 and 3 are achieved with Sublime's Text Regex options, more specifically by selecting the "Find and Replace" function and make sure that the Regular Expression button is selected.

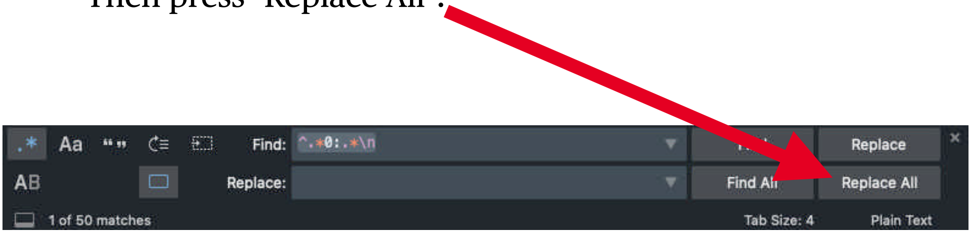


In the search bar you can type the code accordingly to the action you need to perform, more specifically:

Remove TimeCode from SRT:

`^.*0:.*\n`

Then press "Replace All".



Remove sequential numbers:

```
^\d+$
```

Then press "Replace All".

Delete empy lines:

```
^\n
```

Then press "Replace All".

Select from the "Edit" Menu the "Wrap Paragraph at 80 carachters" option.

Remove line breaks:

```
\n
```

Sublime Text for Windows

For this platform, the Sublime Text's commands are slightly different.

First of all you will select the "Find and Replace" function, then:

Remove timecode:

```
^00.*$
```

Then press "Replace All".

Remove numbers:

```
\n\d\d*
```

Then press "Replace All".

Remove new lines:

```
\n$\n
```

Then press "Replace All".

Delete Line Breaks:

```
\n
```

Then press "Replace All".

Select the "Wordwrap at 80 carachters" function.

With the above sequence of commands you will end up with a complete written transcription of your video, ready to be cleaned, corrected or modified and then ready to be published on the destination of your choice.

Additional Notes and Syntax Definitions

This section is about the explanation of external scripts and commands used in this book, that are not specifically referred to FFMPEG, or the other tools described in this text.

Installing Homebrew: Syntax Explanation

```
/bin/bash -c "$(curl -fsSL https://
raw.githubusercontent.com/Homebrew/install/
master/install.sh)"
```

`/bin/bash`

this will tell your shell to execute the command by using the BASH language

`-c`

stands for "command". This will instruct the computer to to read and execute a command.

"

" Enclosing characters in double quotes ('"') preserves the literal value of all characters within the quotes.

`$`

means to expand a parameter enclosed in braces

`(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)`

`curl` is a tool to transfer data from or to a server, using several different supported protocols.

`-fsSL`

It's a combination of more options of **`curl`**.
Literally "**`-f`** (fail silently), **`-s`** (don't show progress meters), **`-S`** (show errors if it fails) in the **`-L`** (location, the URL address).

https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"

=

The location of the script to be executed.

FFPROBE KeyFrames Command: Syntax Explained

```
ffprobe -loglevel error -skip_frame nokey
-select_streams v:0 -show_entries
frame=pkt_pts_time -of csv=print_section=0
YOUR_FILE.mp4
```

`ffprobe`: this will launch the program ffprobe

`-loglevel error`: a type of log that will displays all errors (if any), including ones which can be recovered from.

`-skip_frame nokey`: ffprobe will discard all frames excepts keyframes.

`-select_streams v:0`: this will select the first video stream of the file (FFMPEG starts counting from 0).

`-show_entries frame=pkt_pts_time`: FFPROBE will output the Presentation Time Stamp (PTS) value only.
See the below section "PTS and DTS Explained" for more informations.

`-of csv=print_section=0`: this will output just the PTS values, value after value, without any other names, or additional text

---

PTS and DTS Explained[71]

Fortunately, both the audio and video streams have the information about how fast and when you are supposed to play them inside of them. Audio streams have a *sample rate*, and the video streams have a *frames per second* value. However, if we simply synced the video by just counting frames and multiplying by frame rate, there is a chance that it will go out of sync with the audio. Instead, packets from the stream might have what is called a Decoding Time Stamp (DTS) and a Presentation Time Stamp (PTS).

To understand these two values, you need to know about the way movies are stored. Some formats, like MPEG, use

[71] http://dranger.com/ffmpeg/tutorial05.html

what they call "B" frames (B stands for "bidirectional"). The two other kinds of frames are called "I" frames and "P" frames ("I" for "intra" and "P" for "predicted"). I-frames contain a full image. P-frames depend upon previous I-frames and P-frames and are like diffs or deltas.

B-frames are the same as P-frames, but depend upon information found in frames that are displayed both before and after them!

So let's say we had a movie, and the frames were displayed like: **I B B P**.

Now, we need to know the information in **P** before we can display either **B** frame. Because of this, the frames might be stored like this: **I P B B**.

This is why FFMPEG has a Decoding Time Stamp and a Presentation Time Stamp on each frame.

The decoding timestamp tells us when we need to decode something, and the presentation time stamp tells us when we need to display something. Thus in this case, our stream might look like this:

```
   PTS: 1 4 2 3
   DTS: 1 2 3 4
Stream: I P B B
```

Generally the PTS and DTS will only differ when the stream we are playing has B frames in it.

Batch Processing for DASH and HLS Delivery: Syntax Explanation

`#! /bin/bash:` Means "Run this script in BASH"

```
for file in *.mp4;
do
mkdir "${file%.*}"
```

This script instruct the computer to select any Mp4 file contained in a specific directory, then it further instruct to create a new directory titled as the input file's name.

The dollar symbol $ specifies that the directory's name it's a variable value.

`%` (percentage): In this case the percent sign (%) is used in the pattern ${variable%.*} for matching the input file name and deleting the extension of the file.

In the command `mkdir "${file%.*}"` you are creating a directory titled as the input file, deleting with the

% symbol everything that follows the percentage symbol, meaning the extension of the input file.

```
  ffmpeg -i "$file" [OPTIONS] "${file%.*}/$
{file%.*}_1920.mp4"
```

This part of the script specify to select the above mentioned .mp4 file taken from the working directory, to process it with specific options, and then to place the final output with the suffix `_1920.mp4`, inside the previously created folder, titled as the input file. The percentage symbol (%) is used to delete every character that follows that symbol, until the closing curly bracket }

```
  mp4fragment "${file%.*}/${file%.*}
_1080".mp4 "${file%.*}/${file%.*}_1080-
f".mp4
```

This part of the BASH script will instruct the shell to open the Bento4's Utility *Mp4Fragment* and to process the file previously processed with FFMPEG and to save the output inside the same directory, with the suffix
`_1080-f.mp4`

```
  mp4dash --hls --output-dir="${file%.*}/$
{file%.*}_output" --use-segment-list "$
{file%.*}/${file%.*}_1080-f".mp4 "$
{file%.*}/${file%.*}_720-f".mp4 "${file%.*}/
${file%.*}_480-f".mp4 "${file%.*}/${file%.*}
_360-f".mp4 "${file%.*}/${file%.*}_240-
f".mp4
  mv "$file" "${file%.*}"
  done
```

This final part of the BASH script will instruct the shell to process the segmented files with another Bento4's utility called *Mp4Dash* in order to create both a DASH and HLS Playlists, using as the output directory a new folder named as the input filename along with the suffix `_output.`

The script further instruct the shell, with the `--use-segment-list` option, to use a specific list of file to be processed, which are the previously segmented files processed with Bento4's *Mp4Fragment*.

Finally the script, with the option `-mv "$file" "${file%.*}"` will move the original input file from the existing directory, to the newly created directory.

The last command, `done`, ends the script.

Bibliography

Jan Ozer

Learn to produce video with FFMPEG in 30 minutes or less

(ISBN: 978-0998453019)

Jan Ozer

Producing Streaming Video for Multiple Screen Delivery

(ISBN: 978-0976259541)

Frantisek Korbel

FFmpeg Basics: Multimedia handling with a fast audio and video encoder

(ISBN: 978-1479327836)

Cameron Newham

Learning the Bash Shell: Unix Shell Programming (In a Nutshell)

(ISBN: 978-0596009656)

Free Software Foundation

Command Line Introduction

(ISBN: 978-1-882114-04-7)

Free Software Foundation

The GNU BASH Reference Manual

Recommended Resources

FFMPEG Documentation

http://ffmpeg.org/documentation.html

FFMPEG Bug Tracker and Wiki

http://trac.ffmpeg.org/wiki/WikiStart

MPEG - Moving Picture Experts Group

https://mpeg.chiariglione.org

SMPTE - Society of Motion Picture and Television Engineers

https://www.smpte.org

Google Developers - VP9

https://developers.google.com/media/vp9

WebM Project

https://www.webmproject.org/

Netflix Tech Blog

https://netflixtechblog.com/

BitMovin Blog

https://bitmovin.com/blog/

Alliance for Open Media

https://aomedia.org/

Rav1e - AV1 Encoder

https://github.com/xiph/rav1e

videolan.org - The Dav1d Project (AV1 Decoder)

https://www.videolan.org/projects/dav1d.html

Command Line Magic

https://twitter.com/climagic

nixCraft

https://twitter.com/nixcraft

Streaming Learning Center

https://streaminglearningcenter.com/

Jan Ozer

https://twitter.com/janozer

Andy Lombardi

https://twitter.com/nd_lombardi

Paul B Mahol

https://twitter.com/BMahol

Julia Evans

https://twitter.com/b0rk

FFMPEG User Questions - Mailing List

https://ffmpeg.org/mailman/listinfo/ffmpeg-user

All questions, comments, thoughts, feedbacks, or errors, typos and bugs reports are very welcomed. Please feel free to write me directly at:

nick.ferrando@ffmpegfromzerotohero.com

Or on Twitter:

https://www.twitter.com/nfer81

About Me

I'm an author, music composer and video producer born in Brescia (Italy), from american mother and italian father. I've started my music career at the age of 14, studying contemporary jazz in Los Angeles and at 15 I've started to work as a studio engineer and then as a music producer in one of the biggest indipendent label in Europe, Time Records. I've produced and composed more than 120 records licensed worldwide, and in 2007 I've founded one of the very first digital content distribution company in Italy, named *Wondermark*, producing also hundreds of videos and content for authors, books and music publishers across Europe. I'm a Mac user since 1993.

Alphabetical Index