FFMPEG

A BEGINNERS GUIDE

Table of contents

Chapter 1 discusses various fundamental topics about audio and video in FFmpeg. It also shows you how to install FFmpeg on your Linux or Windows system.

Chapter 1

Introduction

The idea for this book arose from my frustration on not finding any organized documentation for learning FFmpeg. Thus, my aim in writing this book has been to provide newbie learners to quickly get up-and-running with FFmpeg.

1.1 Getting started with FFmpeg

1.1.1 What is FFmpeg?

FFmpeg is a command-line tool for *nix and Windows systems that, in its simplest form, provide a facility to decode and an encode media files. With the proliferation of video on the Internet and in our daily lives, users need the ability to transcode (convert) audio and video files from one format to another. For example, a user might have downloaded a video from YouTube and need to convent it to a format playable on an iPod or other media device.

Besides this obvious use, FFmpeg is also capable of a few other fundamental manipulations on the audio and video data. These manipulations include changing the sample rate of the audio and advancing or delaying it with respect to the video, reducing the size of the media file. They also include changing the frame rate of the resulting video, cropping it, resizing it, placing bars left and right and/or top and bottom in order to pad it when necessary, or changing the aspect ratio of the picture. Furthermore, ffmpeg allows importing audio and video from different sources such as a microphone.

The main components of FFmpeg are *libavcodec*, an audio/video codec library, *libavformat*, an audio/video container mux/demux library, and the ffmpeg *command line* program for passing various transcoding options to the main program.

The FFmpeg project was started by Fabrice Bellard, and has been maintained by Michael Niedermayer since 2004. The name of the project comes from the MPEG video standards group, together with "FF" for "fast forward". On March 13, 2011 a group of FFmpeg developers decided to fork the project under the name Libav (http://libav.org/) due to some project management related issues.

FFmpeg is used by many open source and proprietary projects, including ffmpeg2theora, VLC, MPlayer, HandBrake, Blender, Google Chrome, and various others.

1.1.2 Components of FFmpeg

FFmpeg is made of the following main components.

Programs

ffmpeg - a command line tool to convert multimedia files between formats.

ffserver - a multimedia streaming server for live broadcasts.

ffplay - a simple media player based on SDL and the FFmpeg libraries.

ffprobe - a simple multimedia stream analyzer.

Libraries

libavutil - a library containing functions for simplifying programming, including random number generators, data structures, mathematics routines, core multimedia utilities, and much more.

libavcodec - a library containing decoders and encoders for audio/video codecs.

libavformat - a library containing demuxers and muxers for multimedia container formats.

libavdevice - a library containing input and output devices for grabbing from and rendering to many common multimedia input/output software frameworks, including Video4Linux, Video4Linux2, VfW, and ALSA.

libavfilter - a library containing media filters.

libswscale - a library performing highly optimized image scaling and color space/pixel format conversion operations.

In this book we will primarily focus on the *ffmpeg* program, the other programs like *ffserver* are used for video broadcasts and is outside the scope of this book. Among the libraries, the most notable parts of FFmpeg are *libavcodec*, an audio/video codec library, and *libavformat*, an audio/video container mux and demux library

1.1.3 Compression

To be honest, trying to shoehorn the complete details of audio and video in a paragraph or two is plainly ridiculous, as the topic is rather complex. But since this is a beginner’s guide, a few basic overviews will be enough to get you started using ffmpeg properly.

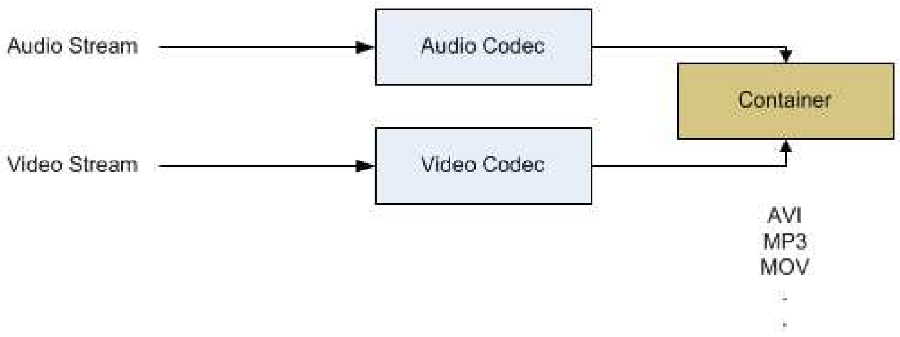
If you are working with audio and video, you are well aware that these files take an inordinate space for storage. You cannot easily work with these files if they were not compressed beforehand. Assuming an NTSC standard video format; a raw (uncompressed) video at 720x480 pixels, 30 frames per second and 24-bit RGB color, would take about 1,036,800 bytes (1 Mb) per frame. That's almost 30MB per second, or over 200GB for a 2-hour movie. And that's just the video. Audio stream also takes additional storage. Something needs to be done so that the movie can be stored on a consumer-grade medium such as a DVD. The data needs to be compressed beforehand.

Conventional, lossless compression algorithms such as ZIP, which everyone uses on a regular basis, don't reduce the size of the data enough, so we need to look into lossy compression for further size reduction. Lossy compression works by discarding some data in the media which results in smaller file sizes. So now you might be thinking what data the compression algorithm discards. Well in general the algorithm does not discard any random data, which would be a disaster. The compression algorithm discards data only if it thinks that the data is redundant. For example in movie frames many times not much changes between successive frames; if the compression software discards some of these frames the viewer will hardly notice any difference, but the storage requirement of those frames have been saved.

Lossy compression is commonly used to compress multimedia data such as audio, video and still images. The only negative aspect of lossy compression is that as some data is removed during compression which can reduce the fidelity of the output.

The algorithms that allow us to encode and decode the data, whether by using lossy or lossless technique are called codecs. Several codecs are enclosed in the libavcodec library supplied with ffmpeg, which enables you to work with a wide variety of video and audio formats.

Once the audio and video streams have been encoded by their respective codecs, this encoded data needs to be put together into a single file. This file is called the ‘*container*’. A graphic of the process is shown below.



1.1.4 Bitrates

A movie is made-up of two main components, Audio and Video. Both "components" produce a separate stream of data that must be decoded by your DVD-player or some program so we can see and hear the video properly.

The bitrate of a movie is the key to the quality of the audio and video of that movie. Also, particular formats specify the bitrate or the maximum bitrate to be used. Bitrate is a measurement of the number of bits that are transmitted over a set length of time. Your overall bitrate is a combination of your video stream and audio stream in your file with the majority coming from your video stream. Bitrate denotes the average number of bits that one second of audio or video data will take up in your compressed bit stream. The overall bitrate of your movie is a combination of your video stream and audio stream in your file with the majority coming from your video stream.

A bit rate is usually measured in some multiple of bits per second - for example, kilobits, or thousands of bits per second (Kbps - for example, kilobits, or thousands of bits per second (Kbps).

Bitrates come in two versions - VBR (Variable Bit Rate encoding) or CBR (Constant Bit Rate encoding). VBR allows a higher bitrate (and therefore more storage space) to be allocated to the more complex segments of media files while less space is allocated to less complex segments. The average of these rates can be calculated to produce an average bitrate for the file. VBR allows you to set a maximum and minimum bitrate. The compression algorithm then tries to efficiently compress the data reducing to the minimum bitrate when there is little or no motion on screen and increasing to the maximum defined rate when the motion is prevalent. This helps to give you a smaller overall file size without compromising the quality of the video.

CBR is used when a predictable flat bit rate is needed. Although the flat bitrate throughout the entire file comes at the price of efficiency for the codec; usually resulting in a larger file, but smoother playback. CBR is useful for streaming multimedia content on limited capacity channels since it is the maximum bit rate that matters, not the average, so CBR would be used to take advantage of all of the capacity. CBR would not be the optimal choice for storage as it would not allocate enough data for complex sections (resulting in degraded quality) while wasting data on simple sections.

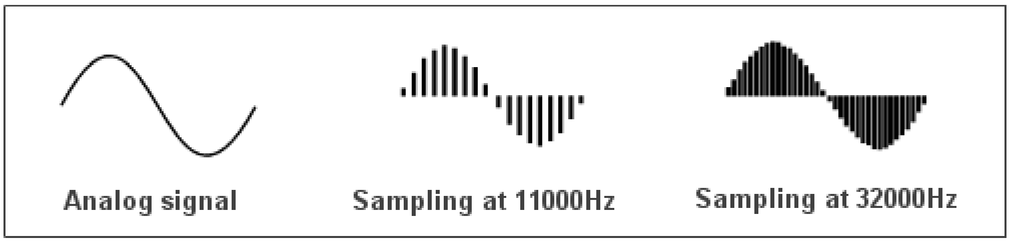
Depending on your video you might want to use a VBR for a streaming playback if the sudden spikes do not exceed your target user's connection speed. For example if there is only one high motion scene in a video, you will be wasting considerable bandwidth on a CBR throughout the entire file and may better serve your user's need by using a VBR. Either way try experimenting with the two settings to find what works best for your video.

Briefly, a bitrate specifies how many kilobits the file may use per second of audio. The following shows the quality for various standard audio bitrates.

64 Kbps	Audio encoded at 64 Kbps have a 15:1 compression ratio. This bitrate is not recommended for digital music but is acceptable for voice-only recordings.
96 Kbps	Audio encoded at 96 Kbps have a 15:1 compression ratio. One minute of music will be about 700KB of disk space.
128 Kbps	Audio encoded at 128 Kbps have an 11:1 compression ratio. One minute of music is takes around 1MB of disk space.
160 Kbps	Audio encoded at 160 Kbps have a 9:1 compression ratio. One minute of music will is about 1.5MB of disk space.
192 Kbps and above	MP3s encoded at this setting take up the most space but have CD quality sound and can take up to 2MB of space per 60 seconds of music. Online music stores or music download services will have at least this high of a bitrate.

1.1.5 Audio Sampling Frequency

The audio sampling frequency is the number of times per second audio is sampled and stored - CD audio is sampled at 44.1 KHz, which means when the sound is converted from analog to digital, 44100 samples per second are taken of the audio signal. The higher the sampling rate the audio has, the wider the frequency range it provides. In other words, higher is better quality. Your lows will be lower; your highs will be higher. For example the following image shows an analog signal on the left converted to a digital representation using two different sampling rates. As you can see the higher sampling will lead to an even more exact reproduction of the original signal.



The sample rate can be thought of as how often or how much the sound is described. CD quality audio has 44,100 of these measurements a second. That's why it's called 44.1 kilohertz (khz).

So what is the relationship between bitrate and sampling frequency? Bitrate simply specifies the number of bits per second that are used to encode the audio stream. The uncompressed bitrate for CD audio is 16 bits x 44100 samples x 2 channels = 1411200bps, or approximately 1411kbps. When audio is stored in an uncompressed format, the bitrate is a linear function of the sample rate; i.e. doubling the sample rate doubles the bitrate.

With uncompressed audio, there is a direct relationship between the sample rate and the bitrate. A 44.1kHz 16-bit stereo signal takes 1411.2 kbps, or approximately 10.4Mb per minute to record. A 44.1kHz 16-bit mono file would take half of this, as would a 44.1kHz 8-bit stereo file or a 22.05kHz 16-bit stereo file.

But now formats like Ogg Vorbis and MP3, compress audio by making calculated guesses about the sounds humans aren't likely to hear and then discard these sound samples. As part of this process, such formats allow us to make some of the decisions by deciding how much to throw away, or to put it more simply, how much data to use to represent the original sound. So, using our 44.1kHz stereo sample, we can choose to use as little as 48kbps or as much as approx 500kbps to store this sound. At 500kbps, more of the original sound fidelity is preserved than at 48kbps.

Calculating values

An audio file's bit rate can be easily calculated when given sufficient information.
Bit rate = (sampling rate) x (bit depth) x (number of channels)

e.g., a recording with a 44.1 kHz sampling rate, a 16 bit depth, and 2 channels:
44100 x 16 x 2 = 1411200 bits per second, or 1411.2 kbit/s

The file size of an audio recording can also be calculated using a similar formula:
File Size (Bytes) = (sampling rate) x (bit depth) x (total channels) x (seconds) / 8

e.g. a 70 minutes long CD quality recording will take up 740MB:
44100 x 16 x 2 x 4200 / 8 = 740880000 Bytes

Some standard sampling frequencies with their applications is given below.

Sampling Rate	Use
8,000 Hz	Telephone, walkie-talkie, wireless intercom and wireless microphone transmission; adequate for human speech.
11,025 Hz	used for lower-quality PCM, MPEG
22,050 Hz	One half the sampling rate of audio CDs; used for lower-quality PCM and MPEG
32,000 Hz	miniDV digital video camcorder, video tapes with extra channels of, DAT, High-quality digital wireless microphones, digitizing FM radio.
44,100 Hz	Audio CD, also most commonly used with MPEG-1 audio (VCD, SVCD, MP3). Most professional audio equipment uses 44.1 kHz sampling and above.
48,000 Hz	The standard audio sampling rate used by professional digital video equipment such as tape recorders, video servers, vision mixers and so on. Also used for sound with consumer video formats like DV, digital TV, DVD, and films.
96,000 Hz	DVD-Audio, some LPCM DVD tracks, Blu-ray Disc audio tracks, HD DVD High-Definition DVD) audio tracks.

1.1.6 Frame rate

The frame rate is how many unique consecutive images are displayed per second in the video to give the illusion of movement; each image thus is called a ‘frame’. The human brain perceives a smooth continuous motion if shown around 24 frames per second. If the frames are less than this magic number, you will see a jerky motion rather than a smooth one. Most video creators use this frame rate.

This is not a standard of course, if your video is a screen cast you can get to frame rates as low as 5fps. Television standards such as PAL (common in Europe and some parts of Asia) uses 25fps, while NTSC standard (used in the US and Japan) uses 29.97fps. Generally you should never exceed the frame rate of the source video. Obviously, the best results will be achieved if the frame rate is kept the same as your original source.

1.1.7 Containers

A container file is used to identify and combine different data types. Simpler container formats can contain different types of audio formats, while more advanced container formats can support multiple audio and video streams, subtitles and meta-data — along with the synchronization information needed to play back the various streams together. In most cases, the file header and most of the metadata are specified by the container format. For example, container formats exist for optimized, low-quality, internet video streaming which differs from high-quality DVD streaming requirements.

The video file formats we’re familiar with, such as Quicktime movies (.mov), .avi are media container formats. Some container formats just contain audio, like WAV file fro Windows, MP3 music files or AIFF files for Macs. Others contain audio and video, such as ASF files for Windows, which contain audio compressed with the WAV codec and video compressed with the WMV codec. There are dozens of these container formats. If you’re uploading a video to an online site, check to see what formats the site supports. Sometimes this can be confusing because the list of accepted formats may have both compression formats like MPEG-4 and container formats like .mov listed.

1.2 Installing FFmpeg

1.2.1 Installing FFmpeg?

FFmpeg is developed under GNU/Linux, but it can be compiled under most operating systems, including Mac OS X, Microsoft Windows, AmigaOS. In most of the Linux distros, you can directly install ffmpeg using their respective package managers. But in case you are looking for installing the latest version or want to customize the installation, you might need direct installation from the source code too, but as it is an involved and tricky procedure, I'm not discussing it here.

Installing FFmpeg in Ubuntu

Run the following command in the terminal to install FFmpeg.

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

Installing FFmpeg in Fedora

FFmpeg can be directly installed from the repos using the following command.

```
$ su -c 'yum install ffmpeg'
```

Installing FFmpeg on CentOS

FFmpeg can be directly installed from the repos using the following command.

```
$ yum install ffmpeg ffmpeg-devel
```

Installing FFmpeg on Windows

By far the easiest way to start using FFmpeg is to get a precompiled binary. Zeranoe.com has pre-built binaries for windows, which makes it easier to install ffmpeg. So if you are using Windows you can get up and running FFmpeg in no time. Go ahead and grab the binaries from the below link.

http://ffmpeg.zeranoe.com/builds/

Once installed use the following command to get the ffmpeg version and the versions of the codecs installed.

```
C:\ffmpeg>ffmpeg  -version
```

On my Windows machine it returns the following; of course this may be different on your system, depending on the version of FFmpeg installed:

```
ffmpeg version N-31100-g9251942, Copyright (c) 2000-2011 the FFmpeg
developers
 built on Jun 30 2011 21:17:59 with gcc 4.5.3
  libavutil    51. 11. 0 / 51. 11. 0
  libavcodec   53.  7. 0 / 53.  7. 0
  libavformat  53.  4. 0 / 53.  4. 0
  libavdevice  53.  2. 0 / 53.  2. 0
  libavfilter   2. 24. 0 /  2. 24. 0
  libswscale    2.  0. 0 /  2.  0. 0
  libpostproc  51.  2. 0 / 51.  2. 0
ffmpeg N-31100-g9251942
libavutil    51. 11. 0 / 51. 11. 0
libavcodec   53.  7. 0 / 53.  7. 0
libavformat  53.  4. 0 / 53.  4. 0
libavdevice  53.  2. 0 / 53.  2. 0
libavfilter   2. 24. 0 /  2. 24. 0
libswscale    2.  0. 0 /  2.  0. 0
libpostproc  51.  2. 0 / 51.  2. 0
```

1.2.2 FFmpeg Command syntax

Adhering to the UNIX culture, FFmpeg relies on a plethora of command-line options to do its work. The generic syntax of an FFmpeg command is shown below.

```
ffmpeg [[infile options][`-i' infile]]...{[outfile options]
outfile}...
```

Each section of the command is explained below.

ffmpeg - The first is the FFmpeg executable file name.

i*nfile option* - This is where you put options for your input video or audio file. This tells FFmpeg to apply any options give here to the input file before processing starts. This section is not as widely used as the '*outfile options*'.

-*i infile* - This is the actual video or audio file you use for processing, and also the directory of where it is located.
e.g /home/george/media/myvideo.flv. You will always need to include the `-i` option before your file name.

outfile options - This is where you will put the various options that are required which you want to be applied to the video or audio you will be creating.

o*utfile* – The name of the output file you want to create, and also the directory path if it not the same as your input file directory.
e.g is /home/george/media/out.flv

Now that we have FFmpeg installed, we can move to chapter 2 which discusses audio processing.

Chapter 2 discusses various fundamental tasks you can accomplish with the audio stream in FFmpeg.

Chapter 2

Audio Processing

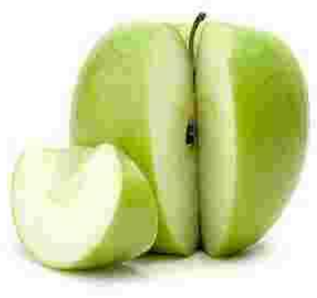
2.1 Transcoding audio files

2.1.1 Introduction to Transcoding

One of the basic tasks you can perform on an audio track in FFmpeg is to convert it into another format. This process known as *Transcoding*, is the direct digital-to-digital conversion of one stream encoding to another, whether video or audio. Transcoding is usually done in cases where a target device – media player such as iPod, iPAD, DVD players or a software application, does not support the format or has limited storage capacity that requires a condensed file size. Transcoding can also be used to convert an incompatible or obsolete format to a better-supported format.

Transcoding is generally a “lossy process” - a data encoding method which compresses data by discarding (losing) some of it to minimize the amount of data that need to be stored in a file; however, transcoding can also be “lossless” if the input is losslessly compressed and the output is either losslessly compressed or stored in a uncompressed state. Although compression can reduce file size consideberaly, repeatedly performing transcoding on a single file using lossy compression can create a ‘generation loss’ – a reduction in the quality of the audio when copying, which would cause further reduction in quality on making a copy of the copy. So you need to keep this in mind while repeatedly transcoding between various formats.

Although I could not show you here the difference between an original and lossy audio compression (due to the limitation of the media of course), the following shows an example of a lossy compression in an image. The original JPG image is on the left and a lossy image of the same after repeated compression is shown on the right. As we lose precious information forever during compression, we cannot get back the original image using the compressed image.



2.1.2 Audio compression

Audio compression is a form of data compression designed to reduce the transmission bandwidth and storage requirement of a digital audio stream. Audio compression algorithms are implemented in software as audio codec's, - which is a software program or library capable of encoding/decoding a digital audio stream.

Audio compression is either lossy or lossless as discussed earlier. Lossless audio compression produces a version of digital audio that can be decoded to an exact digital duplicate of the original audio stream. This is in contrast to the irreversible changes upon playback from lossy compression techniques such as Vorbis and MP3.

Bitrates are explained in Chapter 1.

The whole idea behind audio compression in FFmpeg is to lower the audio bitrate (96kbps, 128kbps, 192 kbps etc.), this effectively also reduces the fidelity or quality of the audio. So you want to keep in mind that, a high bitrate audio file confirms a better sound quality, so by lowering its bitrate you are actually degrading the quality.

For normal computer use, the 128kbs rate produces a quality equal to that of an audio CD. But in the case of an MP3 use, it is necessary to use a 256kbs bitrate to reach an identical result to that of the CD quality sound.

2.1.3 Getting your audio file ready

Now that we have gone through a short introduction to compression, we will now work on the process of transcoding audio files.

To run the example commands in this section, you will need an audio file in a **.wav** or an **.mp3** format. You can get hold of a wav file by ripping an audio track from a music CD or downloading an mp3 file from the Internet. Call the resulting file 'myaudio.mp3'. For this section I used the 'Solo Piano 7' Opening file from http://www.archive.org/details/solo-piano-7.

Next, we will get ffmpeg to identify the file. This will tell us the various details of the audio file. The simple way to get this information is to just tell ffmpeg to use it for input. For this we need to use the *–i* option. Enter the following command at your prompt.

```
ffmpeg  -i  myaudio.mp3
```

The exact output on my PC is shown below; which may differ from yours depending on the version of ffmpeg you are using.

```
D:\ffmpeg>ffmpeg -i myaudio.mp3

ffmpeg version N-31100-g9251942, Copyright (c) 2000-2011 the FFmpeg
developers
Input #0, mp3, from 'myaudio.mp3':
  Metadata:
    album           : solo piano 7
    artist          : Torley
    album_artist    : Torley
    composer        : Torley
    genre           : Piano
    track           : 001/176
    title           : 001 – Openings
    date            : 2008
  Duration: 00:01:39.50, start: 0.000000, bitrate: 193 kb/s
  Stream #0.0: Audio: mp3, 44100 Hz, stereo, s16, 192 kb/s
At least one output file must be specified
```

There is a lot of information we can gather from the output - the track is 1 minute 39.50 seconds long, the bitrate is 193kb/s, the audio is encoded in mp3 format at 44100Hz (44.1KHz) and has two channels (stereo). All this information will come in handy during a transcoding process.

2.1.4 Transcoding to a different format

Let us now convert the downloaded file to a simple wav format. Notice that we have not specified any format option or flag, just the complete output filename. FFmpeg automatically guesses which encoders to use by noticing the format of the input and output files, this can be a big help if you keep forgetting the option name or are just being lazy. If you are not going to specify the encoder format, make sure you mention the full filename, along with the appropriate format extension.

```
ffmpeg -i myaudio.mp3 myaudio.wav
```

The output of the command is shown below.

```
ffmpeg version N-31100-g9251942, Copyright (c) 2000-2011 the FFmpeg
developers
Input #0, mp3, from 'myaudio.mp3':
  Metadata:
    album           : solo piano 7
    artist          : Torley
    album_artist    : Torley
    composer        : Torley
    genre           : Piano
    track           : 001/176
    title           : 001 - Openings
    date            : 2008
  Duration: 00:01:39.50, start: 0.000000, bitrate: 193 kb/s
    Stream #0.0: Audio: mp3, 44100 Hz, stereo, s16, 192 kb/s
File 'myaudio.wav' already exists. Overwrite ? [y/N] y
Output #0, wav, to 'myaudio.wav':
  Metadata:
    album           : solo piano 7
    artist          : Torley
    album_artist    : Torley
    composer        : Torley
    genre           : Piano
    track           : 001/176
    title           : 001 - Openings
    date            : 2008
    encoder         : Lavf53.4.0
    Stream #0.0: Audio: pcm_s16le, 44100 Hz, stereo, s16, 1411 kb/s
Stream mapping:
  Stream #0.0 -> #0.0
Press [q] to stop, [?] for help
size=   17141kB time=00:01:39.50 bitrate=1411.2kbits/s
video:0kB audio:17141kB global headers:0kB muxing overhead 0.000251%
```

Notice how large the resulting wav file is (17 Mb) as compared to the original mp3 format (2.1 Mb). This being for the reason that the wav file is not compressed like its mp3 counterpart. Incidentally, the audio format of the wav is Pulse-code modulation (PCM), technically *PCM signed 16 bit little-endian* format.

As you can see from the screenshot above the output of an ffmpeg command is quite large, so from here on I'll just specify the command and do away with the output screen unless it is required for explanation.

2.1.5 Changing the bitrate of the audio

As we learned in Chapter 1, bitrates control the file size and the quality of an audio or video stream. Lowering the bitrate will result not only in a reduced file size but also diminish the quality of the final output. This can be required if you have a high quality audio recording and need to lower the quality for a reduced file size to stream over the Web. For example the following command will set the bitrate of the mp3 file to 64kb/s. This uses the *–ab* option to the job.

-ab <value>

```
ffmpeg  -i  myaudio.mp3  -ab 64k  out.mp3
```

The higher the value the better is the audio quality. This is one of the important factors responsible for the audio quality. But that doesn't mean you can make a poor audio file sound better by increasing its bitrate. The resultant file will just be of bigger size.

Another example - to transcode an mp3 file to an AAC format, with a bitrate of 128K, we can use the following.

```
ffmpeg  -i  myaudio.mp3  -ab 128k  myaudio.aac
```

As we saw earlier the original audio track has 2 channels (stereo). Many times it is not necessary to have 2 channels, like in a speech recording, where its really doesn't matter.. In such cases you can further reduce the file size by setting the audio channels to mono or '1'. For output streams it is set by default to the number of input audio channels.

-ac <value>

```
ffmpeg  -i  myaudio.mp3  -ac 1 out.mp3
```

Note that once you convert a stereo channel to a mono, you cannot convert it back to a stereo channel audio. That information is lost forever. The same thing happens with bitrates. Once you reduce a bitrate of an audio file, you cannot just increase the bitrate back again to get the original quality. That information is already gone. So as a precaution, *never work* with your original media files. Make a copy of the original and work with the copy.

The other important audio option is *–acodec***.** This option lets you choose the type of audio codec you want to use. e.g. if you are using ffmpeg on a mp3 file, then it will need the audio codec libmp3lame. You can specify it using *-acodec* libmp3lame. Although, by default, ffmpeg takes care of the codecs you need (by guessing it from the output file format) but if you need anything different, then go for this option. FFmpeg uses a default encoder for each audio stream, using the output file extension to guess the encoder to use. This option lets you force FFmpeg to use a specific audio encoder rather than the default. The following for example will extract the audio stream from a .flv video and save it as an .mp3 file using the *libmp3lame* encoder.

-acodec <encoder/decoder>

```
ffmpeg  -i  myvideo.flv   -acodec  libmp3lame  myaudio.mp3
```

Sometime you FFmpeg may be unable to correctly decode the input file, giving the error something like the following.

```
Error while decoding stream #0.0
```

In such cases you can force FFmpeg to use a particular decoder to decode the input file. The following example will force FFmpeg to use the mp3 codec toe decode the input file audio.

```
ffmpeg  -acodec  libmp3lame  -i  myvideo.flv    myaudio.mp3
```

Note that the *–acodec* option comes before the *–i* option when we want the codec to apply to the input stream and comes after the *–i* option when we want the codec to apply to the output stream. To see what codecs are available on your system, issue the following command.

```
ffmpeg  -codecs
```

Sometimes you may want to completely disable the audio recording for which we can use the *–an* option. This can be used to strip out an audio stream from a video file. When you use this option, all the other audio related attributes are cancelled out, which is fine, as they would not matter without the audio. So for example you are want to disable the audio from a video file and only copy the video stream, you can use the following.

```
ffmpeg  -i  myvideo.flv  -an out.flv
```

Another important option is *–ar*, the audio sampling frequency. This lets you set the maximum sampling frequency of the audio stream. Audio sampling was discussed in Chapter 1. You can use the option to reduce the sampling frequency to a lower value to reduce file storage or Internet bandwidth capacity. The default value is set at 44100Hz. The value is given in Hz. So the following will resample the input audio to 11025Hz with a single channel (mono).

-ar <value in Hz>

```
ffmpeg  -i  myaudio.mp3  -ar 11025  -ac 1 myaudio.mp3
```

Note that once you have reduced the sampling frequency some of the audio data is lost. You cannot again resample it to a higher value and expect increase in the audio quality.

2.1.6 Audio grabbing

Until now we have looked into how to transform existing audio stream into other formats. FFmpeg can also grab audio from external devices such as a microphone. This can be useful if you need to record from your desktop microphone or create a screencast. Note that the following command will not work on a Windows machine. You need to have a Linux machine to correctly grab the mic audio. Enter the following command at your Linux prompt.

```
ffmpeg  -f oss  -i  /dev/dsp  ./audio.wav
```

This will start recording the input audio from the mic to the 'audio.wav' file. Once started you will need to press 'q' to stop the recording. We will now look into the various options given above.

The option –f denotes the format to be used for the input stream. There are various formats FFmpeg supports; you can find the complete list by issuing the following command.

```
ffmpeg  -formats
```

Here we are using the 'oss' format, which stands for Open Sound System input device. The Open Sound System (OSS) is an interface for making and capturing sound in Unix or Unix-like operating systems. In the Linux kernel, there have historically been two uniform sound APIs. One is OSS; the other is ALSA (*Advanced Linux Sound Architecture*). ALSA is available for Linux only.

The device ‘/dev/dsp’ is the default audio input device in the Linux system. It's connected to the main speakers and the primary recording source such as a microphone. The system administrator can set /dev/dsp to be a symbolic link to the desired default device.

The ‘audio.wav’ file is where the recorded audio will be saved.

Another example - the following will record the mic audio to the file ‘rec.flac’ in the current directory, this is a flac format file.

```
ffmpeg  -f alsa  -ar 48000  -i front  ./rec.flac
```

2.2 Some popular audio formats

.AAC Advanced Audio Coding File - declared the new audio-file standard in 1997, designed to replace its predecessor, MP3. It provides better quality at lower bit rates, and its Apple's standard iTunes and iPod audio format.

.AIF(F) Audio Interchange File Format - developed by Electronic Arts and Apple back in the '80s. AIFF files contain uncompressed audio, resulting in large file sizes.

.m4a Apple Lossless - This file format uses lossless compressions for digital music.

.MP3 MPEG Layer 3 - the most popular digital-audio music format, designed by a team of European engineers in 1991 to conserve the quality of a song while storing it in a small, compact file.

.OGG Ogg Vorbis - one of the most popular license-free, open-source audio-compression formats. It's efficient for streaming and compression because it creates smaller files than MP3 while maintaining audio quality.

.**RA(M)** Real Audio Media - developed by RealNetworks in 1995. It has a wide variety of uses, from videos to music, but is mainly used for streaming audio such as that from Internet radio stations.

.WAV Windows WAVE - IBM and Microsoft-developed format popular audio format among PC computer users; it can hold both compressed and uncompressed audio.

.WMA Windows Media Audio - designed by Microsoft to be an MP3 competitor, but with the introduction of iTunes and iPods, it's fallen far behind MP3 in popularity.

2.3 Audio processing recipes

MP3 to AAC High Quality Stereo

```
ffmpeg -i in.mp3  -acodec aac -ac 2 -ar 48000 -ab 192k out.aac
```

MP3 to AAC High Quality 5.1

```
ffmpeg -i in.mp3  -acodec aac -ac 6 -ar 48000 -ab 448k out.aac
```

Convert to low quality mp3 to preserve storage

```
ffmpeg  -i in.mp3 -ab 64K out.mp3
```

MP3 to Vorbis OGG (can be played in HTML 5)

```
ffmpeg  -i in.mp3 -acodec vorbis -aq 50 out.ogg
```

Chapter 3 discuses video processing – conversion between various formats, video splitting, thumbnail generation from videos and other miscellaneous video processing tasks.

Chapter 3

Video Processing

3.1 Transcoding video files

3.1.1 Video transcoding introduction

Among users, video conversion between various formats is doubtless the widest and most popular use of FFmpeg. Before starting we need to have a sample video handy. Let's start by grabbing a Flash Video .flv file from YouTube and name it ‘myvideo.flv’. If you have an FLV video on your PC, well and good. You can use http://keepvid.com/ do download a video from YouTube. Now that you a video ready let us see what ffmpeg can tell us about it.

```
ffmpeg  -i  myvideo.flv
```

The output of the command is shown below (Box 1.). This shows that the audio stream is a mono track sampled at 22050 Hz and MP3-encoded. The audio bitrate is 64kbs. The video stream was encoded using the "flv" (Flash Video) codec, "yuv420p" is how the color is encoded, the picture resolution is 320x240 pixels and the frame rate is 25 frames per second.

Getting to know the details before you start any processing any video is essential to getting correct and optimal results.

```
(some output removed)
  Duration: 00:01:40.07, start: 0.000000, bitrate: 307 kb/s
    Stream #0.0: Video: flv, yuv420p, 320x240, 243 kb/s, 25 tbr, 1k tbn,
1k tbc
    Stream #0.1: Audio: mp3, 22050 Hz, mono, s16, 64 kb/s
At least one output file must be specified
```

Box .1

Let’s start by transcoding the FLV we downloaded to an AVI format. If you do not specify any options FFmpeg automatically selects the proper output encoder by reading the extension of the output file.

```
ffmpeg  -i  myvideo.flv  output.avi
```

Sometimes you might want to force FFmpeg to use a particular type of codec in case it fails to correctly use a particular codec to convert to a required output format. You can easily do it using the -*vcodec* flag. In the below example we are asking FFmpeg to use the 'mpeg4' codec to transcode the source video.

-vcodec *codec*

```
ffmpeg  -i  myvideo.flv  –vcodec mpeg4  output.mp4
```

The above showed a barebones video transcoding, but it rarely ends with this. Users have various other requirements while converting between formats. User may need to reduce the quality of the audio stream to reduce the video size, increase the resolution of the output video or remove some sections of the video. For this FFmpeg gives you various options to work with. The following sections show some of the essential options FFmpeg provides.

3.1.2 Setting the resolution or frame size of a video

The resolution of our original video is 320x240 pixels. What if we want to increase the resolution to 640x480 i.e. double the size of the video. You may also need to change the resolution if you are converting to format which has a different aspect ratio than the source video. We can use the –*s* option for this exact purpose.

-s *size*

```
ffmpeg  -i  myvideo.flv  –s 640x480  out.flv
```

Instead of specifying the size in numbers you can use an abbreviation.

```
ffmpeg  -i  myvideo.flv  –s vga  out.flv
```

The complete list of resolutions and their respective abbreviations is given in the following table.

`sqcif' 128x96	`uxga' 1600x1200	`wqsxga' 3200x2048
`qcif' 176x144	`qxga' 2048x1536	`wquxga' 3840x2400
`cif' 352x288	`sxga' 1280x1024	`whsxga' 6400x4096
`4cif' 704x576	`qsxga' 2560x2048	`whuxga' 7680x4800
`16cif' 1408x1152	`hsxga' 5120x4096	`cga' 320x200
`qqvga' 160x120	`wvga' 852x480	`ega' 640x350
`qvga' 320x240	`wxga' 1366x768	`hd480' 852x480
`vga' 640x480	`wsxga' 1600x1024	`hd720' 1280x720
`svga' 800x600	`wuxga' 1920x1200	`hd1080' 1920x1080
`xga' 1024x768	`woxga' 2560x1600	

Let us now convert a high resolution MOV video to a low resolution FLV. The original video has the following specification.

Format = H.264
Resolution = 847x478
Video Bitrate = 700kb/s
Audio Bitrate = 1255 kb/s
Audio = 44 KHz, stereo

This takes about 90Mb of storage. To convert it to a lower resolution FLV I used the following settings.

Format = FLV
Resolution = 480x268
Video Bitrate = 300kb/s
Audio = 22 KHz, stereo

We have halved the audio sampling rate and halved the video size; we will also reduce the video bitrate. The following is the final command. After completion the final file was at a size of 35Mb, half of the original. Of course we lost a lot of quality in the process, but that was our original goal.

```
ffmpeg  -i  high-res.mov  -ar 22050  -b 300k -s 480x268  -vcodec flv out.flv
```

If you would like to do away with all the options but still instruct FFmpeg to use the same quality for the output video, as the input video, you can use the *–sameq* option which uses the same quality factors in the encoder as in the decoder which allows almost lossless encoding.

```
ffmpeg  -i  myvideo.flv –sameq out.mpeg
```

3.2 Extracting images from a video

Let us say you have a ton of videos and need to identify each one by an image taken from the video itself. For that you need to extract a frame from the respective video and save it as an image or rather call it a 'thumbnail'. FFmpeg lets you easily accomplice this task. You can extract a single image at a specific position in the video or multiple images from multiple positions.

The most basic command to get the task done is shown below.

```
ffmpeg -i myvideo.flv  -r 1 –vframes 1 image-%d.jpeg
```

The command will capture a single frame from the start of the video and save it as a jpeg file. The various options used are listed below.

-r	Used to set the frame rate of video. i.e. no. of frames to be extracted into images per second. The default value is 25, but that will return a large number of images, and with little difference between subsequent images in a single second. So we will set this to 1.
-vframes	The number of video frames to record.
image-%d.jpeg	The output images file names. Image-%02d.jpeg means that the images will be saved in the following format. image-01.jpeg image-02.jpeg image-12.jpeg etc. You can change the format as per your liking and use case.

You may be asking why use the *–r* and the *–vframes* option; even if we remove the –r option, we will still get 1 frame. True, but if you set the *–vframes* option to maybe 12 and remove the *–r* option, then you will capture 12 images from the "same 1 second" position rather then 1 image from each new second. So if we give a new command like the following, we will capture 12 images; 6 from each second.

```
ffmpeg -i myvideo.flv  -r 6 –vframes 12  image-%d.jpeg
```

You do not have to start capturing images from the start of the video, rather you have the option to capture from a particular location in the video. For example to capture a single frame at location 00:01:30 - i.e. at 1minute 30 seconds in the video, we can use the following command.

```
ffmpeg -i myvideo.flv  -ss 00:01:30 -r 1 –vframes 1 image-%d.jpeg
```

The –ss option details are given below.

-ss	You can either specify the duration as a whole number of seconds (eg: "-ss 90" if you want 1½ minutes of video) or you can use hh:mm:ss.sss notation (eg: "-ss 00:01:30" for 1 minutes and 30 seconds of video).

So, to extract 12 images starting from the 00:01:30 location, 1 image for each second of video, we can use the following.

```
ffmpeg -i myvideo.flv  -ss 00:01:30 -r 1 –vframes 12 image-%d.jpeg
```

You can also specify the position time in seconds rather than the HH:MM:SS format. So the following will extract 12 images starting from 90 seconds in the video.

```
ffmpeg -i myvideo.flv  -ss 90 -r 1 –vframes 12 image-%d.jpeg
```

If you need to resize the output images to a particular size you can use the -*s* option.

```
ffmpeg -i myvideo.flv  -ss 90 -r 1 –vframes 1 –s 100x100 image-%d.jpeg
```

3.2.1 Creating a video from images

In the last section we saw how you could extract images from a video for thumbnail creation purposes. You can also work the other way and create a video from a sequence of images. The following for example will take a bunch of jpeg images named as *image-01.jpeg*, *image-02.jpeg* etc and convert it to an flv video. The *–f* option instructs FFmpeg to use a particular container, here 'image2'.

```
ffmpeg -f image2 -i image-%2d.jpeg out.flv
```

You can further ask FFmpeg to use a particular video size for the output video using the *–s* option.

```
ffmpeg -f image2 -i image-%2d.jpeg –s 480x240  out.flv
```

3.3 Misc. video processing tasks

Below are some other miscellaneous video tasks you can accomplish using FFmpeg.

3.3.1 Extract a video segment

Extracting a small segment of a video can be accomplished using the following command. This will extract the starting 30 seconds of the video.

```
ffmpeg -i myvideo.flv  -t 00:00:30 out.flv
```

If you would like to extract a video segment from a certain position in time rather than from the start, you will have to use the –ss flag along with the –t flag. The following for example will extract a 30 second video starting from the position 00:01:30; i.e. 1 minute 30 seconds from the start.

```
ffmpeg -i myvideo.flv  -ss 00:01:30 -t 00:00:30 out.flv
```

The same can be accomplished as below, using the absolute 'seconds' format.

```
ffmpeg -i myvideo.flv  -ss 90 -t 30 out.flv
```

3.3.2 Splitting a video

In some cases you will want to split a huge video into two or multiple parts. Lets us say for example that you want to split a 1 minute 40 second video into a 1 minute video and another of 40 seconds. We can do that in two steps. First you will extract the starting 1 minute of the video and save it in 'out1.flv'.

```
ffmpeg -i myvideo.flv -vcodec copy -acodec copy -ss 00:00:00 -t 60 out1.flv
```

We will then extract the remaining 40 seconds of the video starting from the 1 minute position and save it to 'out2.flv'.

```
ffmpeg -i myvideo.flv -vcodec copy -acodec copy -ss 00:01:00 –t 40 out2.flv
```

We are using the special 'copy' value of the –*vcodec* and –*acodec* options. We could have also only specified the following command but I wanted to highlight the use of the 'copy' value.

```
ffmpeg -i myvideo.flv -ss 00:01:00 -t 40 out2.flv
```

As we are extracting the complete second part of the video, we could eliminate the –t option altogether, which will instruct FFmpeg to extract all of the remaining video after 1 minute.

```
ffmpeg -i myvideo.flv -vcodec copy -acodec copy -ss 00:01:00  out2.flv
```

3.3.2 Combining or stitching videos

Although you cannot merge video files using only FFmpeg, a few multimedia containers like MPEG-1, MPEG-2 PS allows you to join video files by merely concatenating them. For example if you have two video files video1.mpg and video2.mpg, you can use Linux cat command as below.

```
cat video1.mpg video2.mpg > merged.mpg
```

On Windows you can use the following.

```
copy /b video1.mpg video2.mpg > merged.mpg
```

Note that this method does not work with all containers, so be sure that you check the final output video.

You could instead use MEncoder to do the stitching, but that is an altogether different program, which will not be covered here. Still a command is shown below that will help you merge two videos.

```
mencoder -ovc lavc video1.mpg video2.mpg -o merged.mpg
```

3.3.3 Recording a screencast

You can also use FFmpeg to take screencasts of your desktop screen. Screencasts enable you to record your dynamic video screen to a video file. Most people use such screencasts to make video tutorials. The following is a simple command to record a screencast.

```
ffmpeg  -f  x11grab  -r 25  -s xvga  -sameq  -i :0.0 out.mpg
```

The ‘x11grab’ is the input source from where we are going to get eh video data. Note that this is only available in Linux and not Windows. We will set the frame rate to 25 with the *–r* option. The *-s* options specifies the size of the output video, try to match this with your screen resolution. **‘-i :0.0’** is the display screen number of your X11 server and we will use ***–sameq*** **option to keep the** video quality equal to the source.

While recoding a screencast we can also add a audio background to the same. The following for example will start grabbing video from *x11grab* and simultaneously add the audio from the file 'music.mp3'.

```
ffmpeg  -f  x11grab  -r 25  -s xvga  -sameq  -i :0.0  -i music.mp3  out.mpg
```

3.4 Video processing recipes

Get info about a video file.

```
ffmpeg -i myvideo.flv
```

Extract sound from a video, and save it as Mp3.

```
ffmpeg -i myvideo.avi -vn -ar 44100 -ac 2 -ab 192 -f mp3 out.mp3
```

Encode video for the PSP

```
ffmpeg -i myvideo.avi -b 300 -s 320x240 -vcodec xvid -ab 32 -ar 24000
-acodec aac out.mp4
```

Convert .flv to animated gif(uncompressed). Watch the gif file size.

```
ffmpeg -i myaudio.flv  out.gif
```

Convert .avi to mpeg for dvd players

```
ffmpeg -i myvideo.flv -target pal-dvd -aspect 16:9 out.mpeg
```

Compress .mov to VCD mpeg2

```
ffmpeg -i myvideo.mov  -target pal-vcd  out.mpg
```

Convert flv video for iPhone

```
ffmpeg -i myvideo.flv -vcodec mpeg4 -b 700k -acodec aac -ab 96 -f mp4
-s 320x240 -r 25 out.mp4
```

3.5 Some popular video formats

.FLV Flash Video Format - developed by Macromedia, and is widely used to deliver video over the Internet (it's the format used by YouTube) and requires the Adobe Flash Player for viewing.

H.264 MPEG-4 - is a digital video codec standard based on MPEG-4 used to get high data compression while maintaining good image quality.

.MOV QuickTime - a multimedia container file that contain audio and video tracks. The video and audio may be encoded using one of several different QuickTime-supported codecs.

.MP4, **.M4V** MPEG Layer 4 - a popular video format defined by the Moving Picture Experts Group. It became a standard in 2000 and was included in QuickTime in 2000.

.MPV MPEG Layer 1 - developed in 1993 by the Moving Picture Experts Group. It's often used with Video CDs (VCDs).

.WMV Windows Media Video - a format developed by Microsoft. Other Windows Media Video formats include .ASF, .AVI, or .MKV.

Chapter 4 discusses various video filters
and their different uses.

Chapter 4

Filters

4.1 Applying video filters

Applying filters to videos is one of the most common tasks users carry out using FFmpeg. There are a number of video filters you can choose from to perform a variety of effects. I'ye limited the filters discussed here to the common ones a beginner can use.

Filters are specified by the *–vf* option and is specified as follows, where "filter" is the name of the particular filter you are going to use.

-vf <filter>

```
ffmpeg -i myvideo.flv  -vf "filter"  out.flv
```

Not every FFmpeg installation has all the filters present. If you want to find which filters your version of FFmpeg supports use the following command.

```
ffmpeg  -filters
```

Let us start out filter section with one of the common filters you will be going to use, – *crop*.

4.1.1 Cropping videos

Sometimes we need to make a video appear widescreen for aesthetic purposes. Resizing the video directly makes it appear to be squashed; instead we need to crop the video to make it appear anamorphic. You may also sometimes require removing some area from the video; e.g. removing the top area containing a logo from a video. We can accomplice this tasks using the 'crop' filter. The crop filter lets you cut a rectangular portion of a video frame; the filter takes four parameters in the following format:

width:height:x:y

width	= output width of the crop area
height	= output height of the crop area
x	= the x position of the crop rectangle
y	= the y position of the crop rectangle

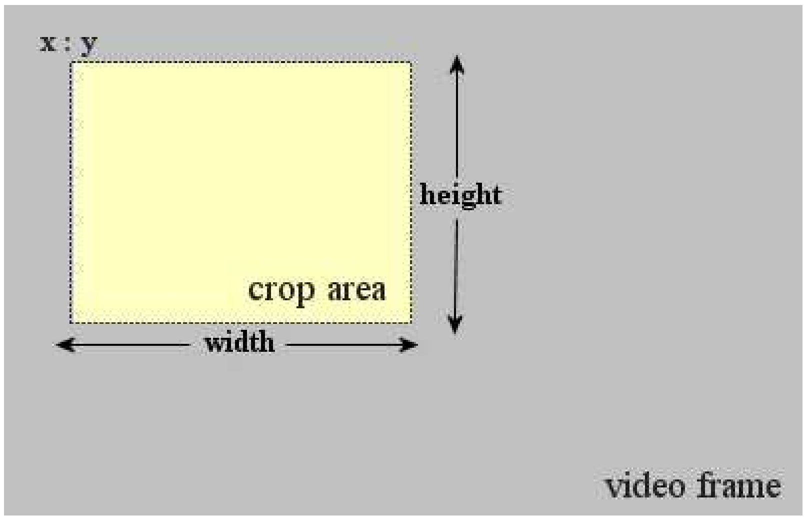


Fig. 4.1

Lets us take the 'myvideo.flv' from the previous chapter and crop it to 320x180 pixels. Note that the *–crop* option has been deprecated in the newer versions of FFmpeg; you now require to use the 'crop' filter instead.

```
ffmpeg -i myvideo.flv  -vf crop=320:200  myvideo-cropped.flv
```

As we have not provided the x:y position for the crop rectangle, this will crop the central 320x200 pixel area from the video, which is the default setting. You can instead set the x:y position at which the crop will occur. For example, to crop the video at 320x200 pixels starting from the left corner of the frame as shown in Fig 4.2, you can use the following command.

```
ffmpeg -i myvideo.flv  -vf crop=320:200:0:0  myvideo-cropped.flv
```

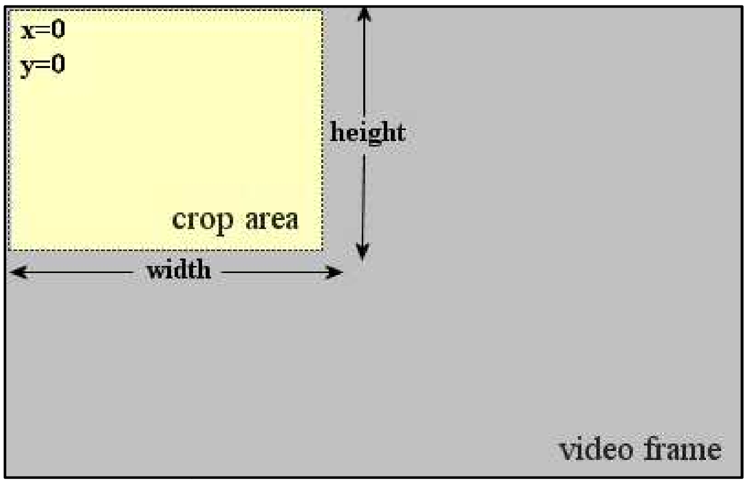


Fig 4.2

4.1.2 Expressions and constants

Besides taking numbers, the ‘crop’ filter and other filters, can also take parameters containing arithmetical expressions and the following constants:

Constant	Description
E, PI, PHI	the corresponding mathematical approximated values for e (euler number), pi (greek PI), PHI (golden ratio)
x , y	the computed values for x and y. They are evaluated for each new frame
in_w , in_h	the input width and height
iw , ih	same as in_w and in_h
out_w , out_h	the output (cropped) width and height
ow , oh	same as out_w and out_h
n	the number of input frame, starting from 0
pos	the position in the file of the input frame, NAN if unknown
t	timestamp expressed in seconds, NAN if the input timestamp is unknown

For example the following will crop the video with the size equal to that of the input frame, in other words it will keep the output size the same as the input, no surprises there.

```
ffmpeg -i myvideo.flv  -vf crop=in_w:in_h  out.flv
```

The following command will only keep the bottom right quarter of the input video.

```
ffmpeg -i myvideo.flv  -vf  crop=in_w/2:in_h/2:in_w/2:in_h/2 out.flv
```

Various arithmetical expressions and functions that are allowed in FFmpeg filters are given below.

Expression	Description
+, -, *, /, ^	Binary operators allowed
+, -	Unary operators allowed

sinh(x)
cosh(x)
tanh(x)
sin(x)
cos(x)
tan(x)
atan(x)
asin(x)
acos(x)
exp(x)
log(x)
abs(x)
gauss(x)
mod(x, y)
max(x, y)
min(x, y)
eq(x, y)
gte(x, y)
gt(x, y)
lte(x, y)
lt(x, y)
	Functions

So to crop the output keeping the height with accordance to PHI (the Golden Ratio) we can use the following.

```
ffmpeg -i myvideo.flv  -vf crop=in_w:1/PHI * in_w  out.flv
```

Here is a complex crop command. The following creates an oscillatory moving video using the ‘sin’ and ‘cos’ functions and various arithmetical expressions.

```
ffmpeg -i myvideo.flv  -vf "crop=in_w/2:in_h/2:(in_w-out_w)/2+((in_w-out_w)/2)*sin(n/10):(in_h-out_h)/2 +((in_h-out_h)/2)*cos(n/6)"  out.flv
```

Taking another example, suppose you have a video with an aspect ratio of 4:3 with size of 320x240 and you need to turn it into a widescreen MPEG movie that can be used on a DVD player. The trouble is that as its aspect ratio is 4:3, which is narrower than the 16:9 format of a DVD. If we want to give it a widescreen aspect ratio, its height should be 320/(16/9)=180 pixels instead of 240. If we simply resize the image in order to squeeze it vertically, the picture will appear deformed. The only thing we can do is to trim off parts of it, or to "crop" it. We need to crop 30 pixels from the top and bottom (60 pixels total) and then convert the result to a 16:9 NTSC DVD.

```
ffmpeg -i myvideo.flv -vf "crop=in_w:in_h-(2*30)"  -target ntsc-dvd -aspect
16:9 out.mpg
```

4.1.3 Padding videos

You might need padding when you want to burn a video into a DVD widescreen format say 16:9 or any other video format. While watching videos you may have noticed many times the 2 black bars at the top and bottom on a DVD video, these are the padding bars.

Let us consider the typical scenario – we need to convert a video into widescreen 16:9 format, the international standard format of HDTV, Full HD and non-HD digital television. Suppose the video format we have is 1280x720 and now we want to pad it with bars on top and bottom of the frame to get the final aspect ratio as 4:3 i.e. a resolution of 1280x960. Incidentally 4:3 format is used in the standard television since television's origins and many computer monitors employ the same aspect ratio. In order to achieve this we will have to increase the height of the video by 240 pixels. We can accomplice this by using the ‘pad filter. The pad filter takes five parameters in the following format:

width : height : x : y:color

width, height	The size of the output image with the padding's added. If the value for *width* or *height* is 0, the corresponding input size is used for the output. The default value of *width* and *height* is 0.
x, y	The offsets where to place the input frame in the padded area with respect to the top/left border of the output frame. The default value of *x* and *y* is 0.
color	The color of the padded area, it can be the name of a color (case insensitive match) or a 0xRRGGBB sequence like in HTML & CSS. The default value of *color* is "black".

The following for example increases the height of the input video by 240 pixels, adding a padding of 120 pixels at the top and 120 pixels at the bottom. The videos *y* position is set at 120 pixels and *x* position at 0. The padding color is set to 'green' to make us clearly see where we have added the padding. Fig. 4.3 shows it visually.

```
ffmpeg -i myvideo.flv -vf pad=0:in_h+240:0:120:green  out.flv
```

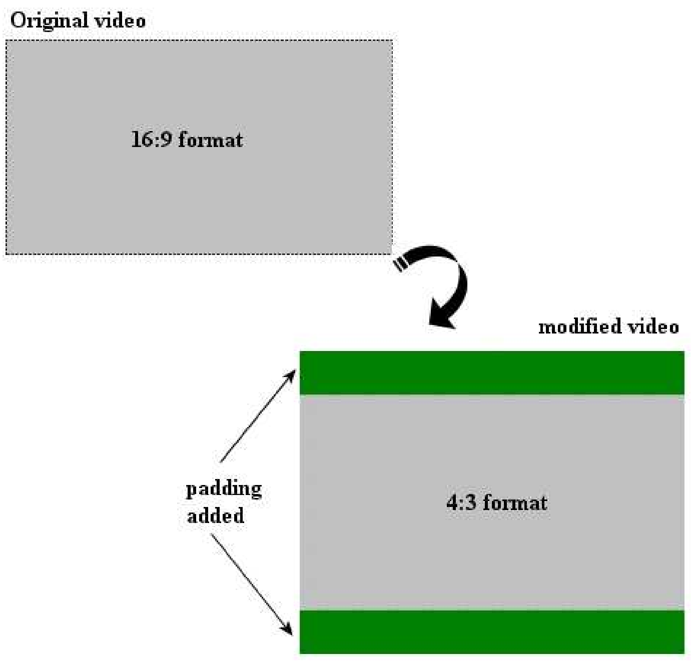


Fig. 4.3

The following example will pad the input to get an output with dimensions increased by 3/2, and put the input video at the center of the padded area.

```
ffmpeg –i myvideo.flv –vf  pad="3/2*iw:3/2*ih:(ow-iw)/2:(oh-ih)/2" out.flv
```

4.1.4 Flipping videos

You can flip a video horizontally or vertically. Frankly I've no idea why someone will want to flip a video horizontally unless of course the video at some stage during processing came out wrong. To flip a video horizontally you run the following command.

```
ffmpeg -i myvideo.flv -vf “hflip”  out.flv
```

Likewise to flip a video vertically you can use the following.

```
ffmpeg -i myvideo.flv -vf “vflip”  out.flv
```

4.1.5 Draw Box

The *–drawbox* filter will enable you to draw a color box around a frame. The filter accepts the following parameters.

drawbox=x:y:width:height:color

width, height	Specify the width and height of the box, if set to 0 they are interpreted as the input width and height. Default is 0.
x, y	Specify the top left corner coordinates of the box. Default is 0.
color	Specify the color of the box, it can be the name of a color (case insensitive match) or a 0xRRGGBB[AA] sequence.

So to draw a blue box around the video with the default size settings, use the following.

```
ffmpeg -i myvideo.flv -vf drawbox=0:0:0:0:blue  out.flv
```

Draw a box with color red and opacity of 60%.

```
ffmpeg -i myvideo.flv -vf  “drawbox=10:20:200:60:red@0.6" out.flv
```

4.1.6 Scale

The *–scale* filter takes the following format:

scale=<*width*:*height*>

A few examples are shown below.

Scale the input video to a size of 200x100.

```
ffmpeg -i myvideo.flv -vf  “scale=200:100” out.flv
```

Scale the input to 2x

```
ffmpeg -i myvideo.flv -vf  “scale=2*iw:2*ih”  out.flv
```

Scale the input to half size

```
ffmpeg -i myvideo.flv -vf  “scale=iw/2:ih/2”  out.flv
```

Increase the height, and set the width to 3/2 of the height

```
ffmpeg -i myvideo.flv -vf  “scale=3/2*oh:3/5*ih”  out.flv
```

4.1.7 Overlay - Watermarking your videos

One of the most common tasks during video processing is that of adding a watermark to your videos. Typically a watermark is used to protect ownership of the video or to provide video credit by adding a logo. One of the common areas where watermarks appear is the bottom right hand or the top right hand corner of a video.

To accomplish this purpose we need to use the 'overlay' filter. This filter is used to overlay one video or image on top of another. It accepts the following parameters and constants.

x, y	x and y is the position of overlaid video or image on the main video.
	You can also use the following constants.
main_w, main_h	main input width and height
W, H	same as *main_w* and *main_h*
overlay_w, overlay_h	overlay input width and height
w, h	same as *overlay_w* and *overlay_h*

So for example to position a watermark 10 pixel to the right and 10 pixels down from the top left corner of the main video, we would use the following.

```
ffmpeg –i myvideo.flv -vf "movie=logo.png [watermark]; [in][watermark]
overlay=10:10 [out]" out.flv
```

To position the same at the bottom left hand corner we use the following.

```
ffmpeg –i myvideo.flv -vf "movie=logo.png [watermark]; [in][watermark]
overlay=10:H-h-10 [out]" out.flv
```

To position at the top right hand corner you use the following.

```
ffmpeg –i myvideo.flv -vf "movie=logo.png [watermark]; [in][watermark]
overlay=W-w-10:10 [out]" out.flv
```

Now you make be thinking about the new strings in the above commands – [watermark], [in] and [out]. These are the source and sinks. The '[in]' is a source stream from where ffmpeg takes the source data. Similarly the '[out]' sink is where ffmpeg send the processed stream. Sink and Source are used specially when chaining filters as above.

In the above watermark example, the 'movie' filter takes the 'logo.png' file and sends it to the '[watermark]' sink. You can name the label whatever you want, but keep it descriptive. The next filter in the chain, 'overlay', takes the input from the '[watermark]' source and overlays it on the [in] stream, processes it and send it to the '[out]' sink, which is finally saved to a file.

As an example we have added the FFmpeg logo at the bottom left corner of the following image.

The above example also includes a new filter – m*ovie*. We did not discuss the filter before because it makes more sense to discuss it where it can be neatly used, as in the current example. The filter takes the following format.

movie=movie_name[:options]

Where *movie_name* is the name of the resource to read, image or a video (not necessarily a file but also a device or a stream accessed through some protocol), and *options* is an optional sequence of *key=value* pairs, separated by ":". The list of the accepted options is given below.

‘format_name, f’	Specifies the format assumed for the movie to read, and can be either the name of a container or an input device. If not specified the format is guessed from movie_name or by probing.
‘seek_point, sp’	Specifies the seek point in seconds, the frames will be output starting from this seek point, the parameter is evaluated with av_strtod so the numerical value may be suffixed by an IS postfix. Default value is "0".
‘stream_index, si’	Specifies the index of the video stream to read. If the value is -1, the best suited video stream will be automatically selected. Default value is "-1".

The *movie* filter also lets us overlay one video on top of another. In the example before we overlaid a plain image on the video. In the next example we will overlay a video on a video.

```
ffmpeg -i myvideo.flv -vf  "movie=vd1.mpg, scale=50:50 [w]; [in][w]
overlay=10:H-h-10 [out]" out.flv
```

‘vd1.mpg’ is a 320x240 video, which we scaled to a 50x50 video and overlaid on the original ‘myvideo.flv.

The output is shown below.

The process of the above command is shown in figure 4.3 below.

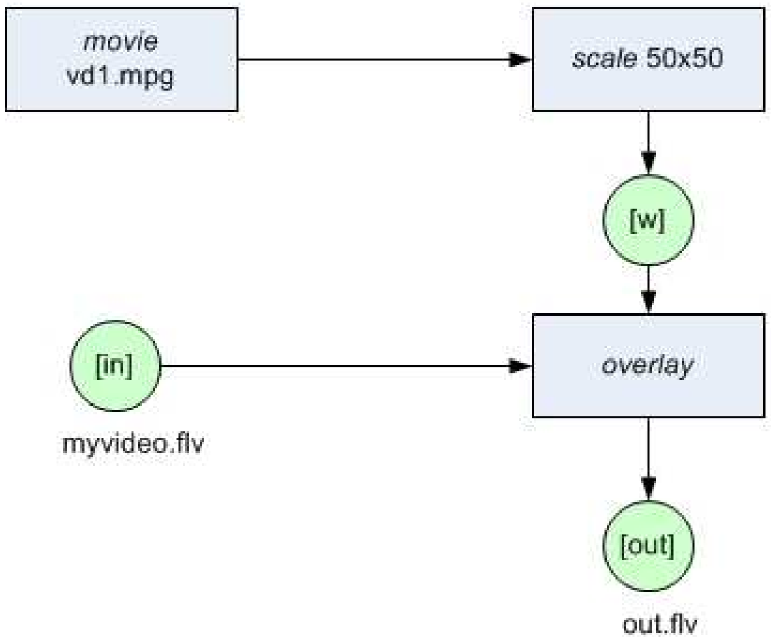


Fig. 3

4.1.8 Sharpening and Blurring videos

Another filter that is commonly used that to Sharpen or Blur a video. The filter takes the following parameters.

unsharp=luma_msize_x:*luma_msize_y*:*luma_amount*:*chroma_msize_ x*:*chroma_msize_y*:*chroma_amount*

luma_msize_x	the luma matrix horizontal size. It can be an integer between 3 and 13, default value is 5.
luma_msize_y	the luma matrix vertical size. It can be an integer between 3 and 13, default value is 5.
luma_amount	the luma effect strength. It can be a float number between -2.0 and 5.0, default value is 1.0.

chroma_msize_x	the chroma matrix horizontal size. It can be an integer between 3 and 13, default value is 0.
chroma_msize_y'	the chroma matrix vertical size. It can be an integer between 3 and 13, default value is 0.
chroma_amount	the chroma effect strength. It can be a float number between -2.0 and 5.0, default value is 0.0.

You don't have to worry what each parameter means, just remember that negative values for the amount will blur the input video, while positive values will sharpen. All parameters are optional and default to the equivalent of the string '5:5:1.0:0:0:0.0'.

For example the following will blur the input video considerably.

```
ffmpeg -i myvideo.flv -vf “unsharp=3:3:-2:3:3:-2” out.flv
```

Note that the 'unsharpen' filter can take some time to process a video compared to other filters.

4.1.9 Transpose

You can transpose input video easily with this filter. It takes the following parameters.

'0' : Rotate by 90 degrees counterclockwise and vertically flip (default).
'1' : Rotate by 90 degrees clockwise.
'2' : Rotate by 90 degrees counterclockwise.
'3' : Rotate by 90 degrees clockwise and vertically flip.

For example the following will rotate the video 90 degrees clockwise.

```
ffmpeg -i myvideo.flv –vf  “transpose=1”  out.flv
```

4.1.10 Fade

This option applies a fade-in/fade-out effect to the input video. It takes the following parameters.

type:*start_frame*:*nb_frames*

type specifies the effect type, which can be either "in" for fade-in, or "out" for a fade-out effect.

start_frame specifies the number of the start frame from where the effect is applied.

nb_frames specifies the number of frames for which the fade effect has to last. At the end of the fade-in effect the output video will have the same intensity as the input video; at the end of the fade-out effect the output video will be completely black.

To apply an effect for a number of seconds, multiply the framerate of the video with the effect duration. For example if the framerate is 25 and you want the effect to last 6 seconds starting from the 0 frame, then nb_frames will be: 25 * 6 = 125.

```
ffmpeg –I myvideo.flv  -vf “fade=in:0:125”  out.flv
```

A few other examples are given below. Assume that the framerate of the input video is 25.

Fade out last 45 frames of a 2500-frame video.

```
ffmpeg -i myvideo.flv –vf  “fade=out:2455:45 ”  out.flv
```

Fade-in first 45 frames and fade-out last 45 frames of a 2500-frame video

```
ffmpeg -i myvideo.flv –vf  “fade=in:0:45, fade=out: 2455:45”  out.flv
```

Make first 50 frames black, and then fade in from frame 50 to a total length of 50 frames. Note that 50 frames equals 2 seconds of video.

```
ffmpeg -i myvideo.flv –vf  “fade=in:50:50”  out.flv
```

4.1.11 Drawtext

The *-drawtext* option enables you to draw a text string or text from a specified file on top of a video.

The filter accepts parameters as a list of key=value pairs, separated by ":", the complete list is shown below. A simple example is shown below. It draws the text ‘Hello World!’ at the upper left corner of the video using the default settings.

```
ffmpeg -i myvideo.flv -vf drawtext="fontfile=arial.ttf: text='Hello World!'”
out.flv
```

The various settings for the *-drawtext* filter are given below.

‘fontfile'	The font file to be used for drawing text. Path must be included. This parameter is mandatory.
\`text'	The text string to be drawn. The text must be a sequence of UTF-8 encoded characters. This parameter is mandatory if no file is specified with the parameter textfile.
\`textfile'	A text file containing text to be drawn. The text must be a sequence of UTF-8 encoded characters. This parameter is mandatory if no text string is specified with the parameter text. If both text and textfile are specified, an error is thrown.
\`x, y'	The offsets where text will be drawn within the video frame. Relative to the top/left border of the output image. The default value of x and y is 0.
\`fontsize'	The font size to be used for drawing text. The default value of fontsize is 16.
\`fontcolor'	The color to be used for drawing fonts. Either a string (e.g. "red") or in 0xRRGGBB[AA] format (e.g. "0xff000033"), possibly followed by an alpha specifier. The default value of fontcolor is "black".
\`boxcolor'	The color to be used for drawing box around text. Either a string (e.g. "yellow") or in 0xRRGGBB[AA] format (e.g. "0xff00ff"), possibly followed by an alpha specifier. The default value of boxcolor is "white".
\`box'	Used to draw a box around text using background color. Value should be either 1 (enable) or 0 (disable). The default value of box is 0.
\`shadowx, shadowy'	The x and y offsets for the text shadow position with respect to the position of the text. They can be either positive or negative values. Default value for both is "0".
\`shadowcolor'	The color to be used for drawing a shadow behind the drawn text. It can be a color name (e.g. "yellow") or a string in the 0xRRGGBB[AA] form (e.g. "0xff00ff"), possibly followed by an alpha specifier. The default value of shadowcolor is

	"black".
`ft_load_flags'	Flags to be used for loading the fonts. The flags map the corresponding flags supported by libfreetype, and are a combination of the following values: default , no_scale , no_hinting , render , no_bitmap , vertical_layout , force_autohint , crop_bitmap , pedantic , ignore_global_advance_width , no_recurse , ignore_transform , monochrome , linear_design , no_autohint , end table Default value is "render". For more information consult the documentation for the FT_LOAD_* libfreetype flags.
`tabsize'	The size in number of spaces to use for rendering the tab. Default value is 4.

Here is a more compete example. The following will draw the text 'Hello World!' with font ‘Arial’ of size 16 at position x=5 and y=5 (counting from the top-left corner of the screen), text is black with a red box around it. Text has a opacity of 0% (i.e no opacity) and the box has an opacity of 50%.

```
ffmpeg -i myvideo.flv -vf drawtext="fontfile=arial.ttf: text='Hello World!':x=5:
y=5: fontsize=16: fontcolor=0x000000@1: box=1: boxcolor=red@0.5" out.flv
```

4.2 Chaining filters

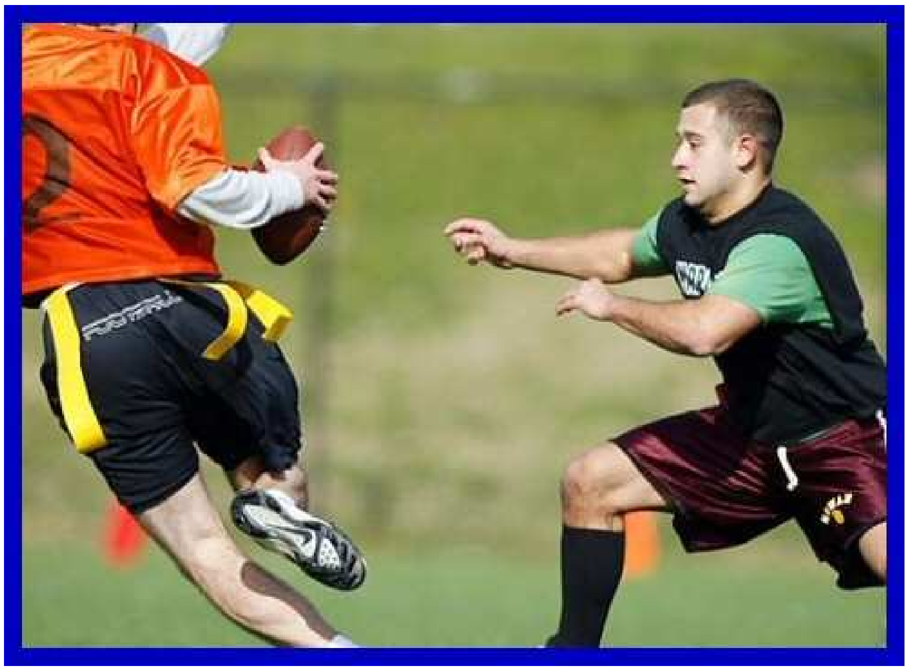
Until now we have applied individual filters to the videos. But the real power of filters can be realized when we chain multiple filters together, which is called a ‘filterchain’. A filterchain consists of a sequence of filters, each one connected to the previous one in the sequence separated by a comma. The format is shown below.

ffmpeg –vf “filter1, filter2, filter3”

For example let us chain together the ‘crop’ and the ‘drawbox’ filters. An example is given below.

```
ffmpeg -i myvideo.flv -vf crop=300:200:10:10,drawbox=0:0:0:0:blue  out.flv
```

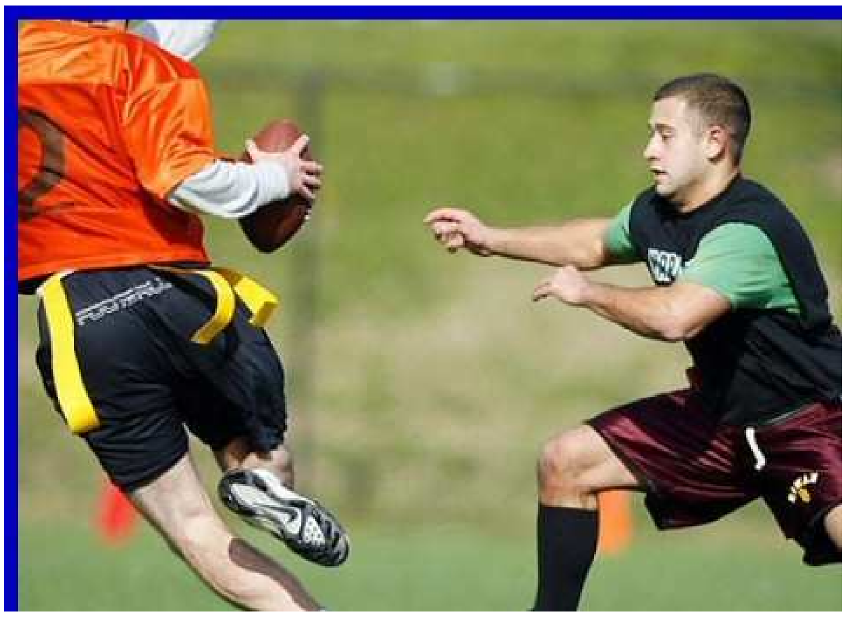
In the example the filters are applied in their respective order. First the ‘crop’ filter is applied to the frame, after which the ‘drawbox’ filter is applied. You can see the effect in the following picture.



If you change the sequence of the filter as below, you will get a different output. Here, first the ‘drawbox’ filter will be applied followed by the ‘crop’ filter.

```
ffmpeg -i myvideo.flv -vf drawbox=0:0:0:0:blue,crop=300:200:10:10 out.flv
```

Notice that some of the border in the video frame below has been cropped.



Chapter 5 discusses a simple way to integrate FFmpeg with PHP.

Chapter 5

PHP

5.1 Integrating FFmpeg with PHP

5.1.1 Calling executable programs from PHP

If you are a web developer and need to process videos uploaded by the user on the website, FFmpeg is an indispensible tool. And since it is broadly available on most platforms, integration with languages like PHP is relatively easy. One of the common methods to integrate FFmpeg with PHP is to use the 'exec()' function.

So to use the example command given in chapter 2, you can use the following code. Note that until the FFmpeg command is running, PHP statements after the exec() will not be executed. i.e. the 'echo' below will only executes after ffmpeg completes its processing.

```
<?php

$audio_source = "myaudio.mp3";
$audio_dest = "myaudio.wav";

$command = "ffmpeg  -i  $audio_source  $audio_dest";
exec($command);
echo "done";

?>
```

The above code assumes that FFmpeg executable is stored in the current directory or is set in your path variable. If it is not stored in your path or in your current directory than you have to explicitly mention the full path to your FFmpeg. For example in Windows, if the path to FFmpeg is 'c:\ffmpeg\bin', then the above code should be adjusted as below. Note the forward slashes as the path separator.

```
<?php

$ffmpeg = "c:/ffmpeg/bin/ffmpeg.exe;
$audio_source = "myaudio.mp3";
$audio_dest = "myaudio.wav";

$command = "$ffmpeg  -i  $audio_source  $audio_dest";
exec($command);

?>
```

Many times you may also need to look at the output of the executable to find if any errors occurred during processing or for debugging purposes.

The following changes to the script will let you capture the output of the 'exec()' function. Note the redirection operators (2>&1) suffixed to the $command string. The output messages of the FFmpeg processing are stored in the $output variable, and the $ret variable stores a Boolean value, indicating success or failure of the execution.

```
<?php

$ffmpeg = "c:/ffmpeg/bin/ffmpeg.exe;
$audio_source = "myaudio.mp3";
$audio_dest = "myaudio.wav";

$command = "$ffmpeg  -i  $audio_source  $audio_dest  2>&1";
$ret = exec($command, $output, $ret);

if(!$ret) {
   echo "command executed successfully";
} else {
   echo "error";
}

/* If you need to see the messages returned by FFmpeg */
print_r($output);

?>
```

As in all software programs errors can occur during execution. Below are some of the few common problems encountered when executing FFmpeg via PHP.

The most common problem user's encounter is that the PHP script cannot read an input file or create the output file. This commonly occurs in Linux systems due to file permissions. If the video or audio file you are reading is owned by someone and the PHP script is owned by another person, Linux will deny the script from accessing the input file. Also while writing the output file, make sure that you have write access and permission to the directory where the output file will be written.

We will be not dwelling further into PHP as I wanted to keep the focus on the pure FFmpeg command tool. Probably I'll be adding or expanding on this in the next version of the book if readers request additional details.

A. FFmpeg Options

A.1 Generic options

These options are shared among all the ff* tools.

`-L'
Show license.

`-h, -?, -help, --help'
Show help.

`-version'
Show version.

`-formats'
Show available formats. The fields preceding the format names have the following meanings:

`D'
Decoding available

`E'
Encoding available

`-codecs'
Show available codecs. The fields preceding the codec names have the following meanings:

`D'
Decoding available

`E'
Encoding available

`V/A/S'
Video/audio/subtitle codec

`S'
Codec supports slices

`D'
Codec supports direct rendering

`T'
Codec can handle input truncated at random locations instead of only at frame boundaries

`-bsfs'
Show available bitstream filters.

`-protocols'
Show available protocols.

`-filters'
Show available libavfilter filters.

`-pix_fmts'
Show available pixel formats.

`-loglevel loglevel'
Set the logging level used by the library. loglevel is a number or a string containing one of the following values:

`quiet'
`panic'
`fatal'
`error'
`warning'
`info'
`verbose'
`debug'

By default the program logs to *stderr*, if coloring is supported by the terminal, colors are used to mark errors and warnings. Log coloring can be disabled setting the environment variable @env{FFMPEG_FORCE_NOCOLOR} or @env{NO_COLOR}, or can be forced setting the environment variable @env{FFMPEG_FORCE_COLOR}. The use of the environment variable @env{NO_COLOR} is deprecated and will be dropped in a following FFmpeg version.

A.2 Main options

`-f fmt'
Force format.

`-i filename'
input file name

`-y'
Overwrite output files.

`-t duration'
Restrict the transcoded/captured video sequence to the duration specified in seconds. hh:mm:ss[.xxx] syntax is also supported.

`-fs limit_size'
Set the file size limit.

`-ss position'
Seek to given time position in seconds. hh:mm:ss[.xxx] syntax is also supported.

`-itsoffset offset'
Set the input time offset in seconds. [-]hh:mm:ss[.xxx] syntax is also supported. This option affects all the input files that follow it. The offset is added to the timestamps of the input files. Specifying a positive offset means that the corresponding streams are delayed by 'offset' seconds.

`-timestamp time'
Set the recording timestamp in the container. The syntax for time is:

now|([(YYYY-MM-DD|YYYYMMDD)[T|t|]]((HH[:MM[:SS[.m...]]])|(HH[MM[SS[.m...]]]))[Z|z])

If the value is "now" it takes the current time. Time is local time unless 'Z' or 'z' is appended, in which case it is interpreted as UTC. If the year-month-day part is not specified it takes the current year-month-day.

`-metadata key=value'
Set a metadata key/value pair. For example, for setting the title in the output file:

```
ffmpeg -i in.avi -metadata title="my title" out.flv
```

`-v number'
Set the logging verbosity level.

`-target type'
Specify target file type ("vcd", "svcd", "dvd", "dv", "dv50", "pal-vcd", "ntsc-svcd", ...). All the format options (bitrate, codecs, buffer sizes) are then set automatically. You can just type:

```
ffmpeg -i myfile.avi -target vcd /tmp/vcd.mpg
```

Nevertheless you can specify additional options as long as you know they do not conflict with the standard, as in:

```
ffmpeg -i myfile.avi -target vcd -bf 2 /tmp/vcd.mpg
```

`-dframes number'
Set the number of data frames to record.

`-scodec codec'
Force subtitle codec ('copy' to copy stream).

`-newsubtitle'
Add a new subtitle stream to the current output stream.

`-slang code'
Set the ISO 639 language code (3 letters) of the current subtitle stream.

A.3 Video options

`-b bitrate'
Set the video bitrate in bit/s (default = 200 kb/s).

`-vframes number'
Set the number of video frames to record.

`-r fps'
Set frame rate (Hz value, fraction or abbreviation), (default = 25).

`-s size'
Set frame size. The format is `wxh' (ffserver default = 160x128, ffmpeg default = same as source). Abbreviations are given in the text.

`-aspect aspect'
Set aspect ratio (4:3, 16:9 or 1.3333, 1.7777).

`-croptop size'
`-cropbottom size'
`-cropleft size'
`-cropright size'
All the crop options have been removed. Use -vf crop=width:height:x:y instead.

`-padtop size'
`-padbottom size'
`-padleft size'
`-padright size'
`-padcolor hex_color'
All the pad options have been removed. Use -vf pad=width:height:x:y:color instead.

`-vn'
Disable video recording.

`-bt tolerance'
Set video bitrate tolerance (in bits, default 4000k). Has a minimum value of: (target_bitrate/target_framerate). In 1-pass mode, bitrate tolerance specifies how far ratecontrol is willing to deviate from the target average bitrate value. This is not related to min/max bitrate. Lowering tolerance too much has an adverse effect on quality.

`-maxrate bitrate'
Set max video bitrate (in bit/s). Requires -bufsize to be set.

`-minrate bitrate'
Set min video bitrate (in bit/s). Most useful in setting up a CBR encode:

```
ffmpeg -i myfile.avi -b 4000k -minrate 4000k -maxrate 4000k -bufsize 1835k out.m2v
```

It is of little use elsewise.

`-bufsize size'
Set video buffer verifier buffer size (in bits).

`-vcodec codec'
Force video codec to codec. Use the copy special value to tell that the raw codec data must be copied as is.

`-sameq'
Use same video quality as source (implies VBR).

`-pass n'
Select the pass number (1 or 2). It is used to do two-pass video encoding. The statistics of the video are recorded in the first pass into a log file (see also the option -passlogfile), and in the second pass that log file is used to generate the video at the exact requested bitrate. On pass 1, you may just deactivate audio and set output to null, examples for Windows and Unix:

```
ffmpeg -i foo.mov -vcodec libxvid -pass 1 -an -f rawvideo -y NUL
```

```
ffmpeg -i foo.mov -vcodec libxvid -pass 1 -an -f rawvideo -y /dev/null
```

`-passlogfile prefix'
Set two-pass log file name prefix to prefix, the default file name prefix is "ffmpeg2pass". The complete file name will be `PREFIX-N.log', where N is a number specific to the output stream.

`-newvideo'
Add a new video stream to the current output stream.

`-vlang code'
Set the ISO 639 language code (3 letters) of the current video stream.

`-vf filter_graph'
filter_graph is a description of the filter graph to apply to the input video. Use the option "-filters" to show all the available filters (including also sources and sinks).

A.4 Audio options

`-aframes number'
Set the number of audio frames to record.

`-ar freq'
Set the audio sampling frequency (default = 44100 Hz).

`-ab bitrate'
Set the audio bitrate in bit/s (default = 64k).

`-aq q'
Set the audio quality (codec-specific, VBR).

`-ac channels'
Set the number of audio channels. For input streams it is set by default to 1, for output streams it is set by default to the same number of audio channels in input. If the input file has audio streams with different channel count, the behaviour is undefined.

`-an'
Disable audio recording.

`-acodec codec'
Force audio codec to codec. Use the copy special value to specify that the raw codec data must be copied as is.

`-newaudio'
Add a new audio track to the output file. If you want to specify parameters, do so before -newaudio (-acodec, -ab, etc..). Mapping will be done automatically, if the number of output streams is equal to the number of input streams, else it will pick the first one that matches. You can override the mapping using -map as usual. Example:

```
ffmpeg -i file.mpg -vcodec copy -acodec ac3 -ab 384k test.mpg -acodec mp2 -ab 192k -newaudio
```

`-alang code'
Set the ISO 639 language code (3 letters) of the current audio stream.