video_generate_compression_x264_singlefile2

February 8, 2022

1 Compression of generative art videos: h264, h265, VP8/9, AV1

We'll be analyzing a practical use case, video compression of Processing or P5js sketches with FFMPEG and h264 AVC, h265 HEVC, VP8 and VP9, and AV1 codecs.

1.1 Encoder choices

For h264, we'll be using libx264, though we can also use NVenc and kvazaar. For h265 we'll be using libx265, but NVenc is also a possibility. VP8 and VP9 are provided via libvpx and libvpx-vp9 and finally AV1 support is provided via libaom-av1. For h266 we don't have yet FFMPEG support, but we can use the reference h266/VVC encoder and decoders, however FFMPEG has no player support yet, nor metrics analysis support. The state of VVC is too experimental and rough at this stage in order to do a confident analysis of results. MPEG5 EVC has experimental support in a separate FFMPEG fork, but this also requires considerable detours and brings the inability to do a direct analysis of results. This document however will be updated at a later date with MPEG5 EVC tests and VVC/h.266 tests as they evolve.

1.2 Test Content

The creation of relevant good quality meaningful reference material for video compression testing is on itself a challenge due to the incredible number of options, parameters, use cases that need to be covered. What is good quality? Good quality on a 1m diagonal OLED display? Or on your mobile phone? Is the content being streamed? Is latency critical? Or is the content stored offline, i.e, in a disc? Settings that will work great with normal scenes will fail with high motion scenes.

Due to the countless encoding options and the combinatorial explosion of testing them all, we'll be focusing on a narrow subset of options targetting this specific use case - generative art videos which exhibit the following characteristics

- Fast erratic or unpredictable movement
- No motion blur
- High frequency content, in some cases, with aliasing
- High chromatic contrast and color purity
- Cases with overlapping alpha blended unpredictable movement

Together with these test results, notebooks and source code you'll also find the source code for the P5js sketches that were specifically created as reference source material in order to test our use cases on on the github repository that also contains these notebooks.

The sketches were recorded and encoded with NVidia's hevc_nvenc using the GPU to keep the high throughput low latency required to record the processing intensive sketches without the introduction

of lag.

1.3 Reference Material Encoding and Characteristics

The source material was encoded with h265/HEVC lossless profile, intra frames only, with YUV 4:4:4 sampling, high tier mode to have good reference material attending the characteristics outlined above. If needed, you can change the profile to a 10 or even 12bit profile. For these tests we remain with 8bit output. Colorimetry was set to Rec.709 RGB primaries, D65 whitepoint, but with the sRGB IEC-61922-6.1 EOTF.

1.4 Types of Video Quality Metrics

Like with the creation of meaningful content for tests, the metrics used are and have been an area of active and ongoing research for quite some time. Both content and video quality assessment are actives areas of research under several groups, including the VQEG or Video Quality Experts Group. We cannot cover all of this in-depth but we'll provide a brief assessment of the main families of techniques and the ones we'll be using in the tests that we'll be running.

Succintly, there are **objective** and **subjective** quality models. Objective models can be measured free from human interpretation, and can be divided according to what signal we have available to test. Subjective models requires the viewers opinion of the quality of the video being watched and a mean score given.

1.4.1 Full Reference Methods

This method compares the encoded signal with a original reference signal and quantifies the difference according to specific metrics. Since they require both the original and the compressed signals, they cannot be used everywhere and since they measure pixel differences, there is heavy computational cost involved.

1.4.2 Reduced Reference Methods

Similar to the Full Reference Method, except there's no access to the entire original signal, or due to constraints, it's impractical to acess it all.

1.4.3 No Reference Methods

Are used to assign a quality indication of the compressed signal without access to the original material. They are usually lighter to compute. These are divided into * **Pixel Based Methods** - try to quantify the image quality or degradation due to blurring, motion blur, other artifacts * **Parametric or Bitstream Methods** - these have access to encoding data, parameters, motion vectors and other related data and try to access the signal quality according to these. They're not accurate, but provide a good fast approximated indication. * **Hybrid Methods** - combine both the pixel based methods and the parametric or bitstream methods.

1.5 Methodology

1.5.1 Video Quality Metrics Used

Since we have access to the original source material and the compressed material we'll be generating, and performance is not critical, it makes sense to use full reference methods. Of these, the metrics

we'll be using are:

- **PSNR** - Peak Signal to Noise Ratio. Due to the characteristics of the HVS, PSNR is not a reliable indicator of perceived quality.
- **SSIM** - Structural Similarity. SSIM measures perceived change in structural information, including luminance and contrast masking, it takes the HVS characteristics into consideration and is a more reliable indicator of perceived quality than PSNR.
- **VMAF** - Video Multimethod Assessment Fusion, is a metric developed by NetFlix, composed of 4 distinct features which are fused via SVM or Suport Vector Machine, a type of supervised machine learning model used for data classification and regression analysis. These features are:
  - **VIF** or Visual Information Fidelity, which considers information loss in 4 different spatial scales
  - DLM or Detail Loss Metric, measures loss of details
  - MCPD or Mean Co-located Pixel Difference, measures the temporal difference on the luminance component
  - ANSNR or Anti-Noise Signal-to-Noise Ratio, abandoned in new VMAF versions.

Other AI/ML Methods for Video Quality Assessment We'll also be using at a later stage AI/ML for perceptual video quality assessment methods with LPIPS metric - Learned Perceptual Image Patch Similarity, and with the Berkeley-Adobe Perceptual Patch Similarity (BAPPS) dataset, albeit in a randomized set of static frames, as well as PSNR-HVS-M, that is, Peak Signal To Noise Ratio taking into account Contrast Sensitivity Function (CSV) and in-between coefficient contrast masking of DCT basis functions - a PSNR metric that takes into account the HVS response.

1.5.2 Encoding Options Used

As outlined earlier, we'll be restricting the options to the choice of: * presets * tuning * entropy encoding algorithms * rate control types * chroma subsampling

There are countless options to fine-tune the motion vector search algorithms, macroblocks sizes, and so on, but this would lead to thousands of files. It can be done, but in a automated, machine-assisted way that is outside the scope of this report.

These will **not** be used on all codecs since some features are only available or exposed to the end user in some codecs, and it would be a daunting task to establish mappings between most parameters. This being said however, by default most codecs seem to a common setting for presets, determining the amount of resources to allocate to compression and most common options hidden behind a preset configuration. They also have in common the **CRF** or Constant Rate Factor, that adapts the data rate in order to target a quality level. These are in common with x264, x265, vpx (VP8) and vpx-vp9 (VP9), ranging from 0 to 51, where higher values mean higher compression. Some also have exposed a **QP** or *Quantization Parameter* control, that allows one to specify the amount of compression for every macroblock in a frame, with higher values resulting in higher quantization, and therefore more compression.

Control over CRF and QP is also exposed in an experimental VVC/h.266 encoder, as well as on a test implementation of MPEG5 EVC in FFMPEG via the XEVE library but these are left outside

the scope of this document until a later date.

Encoding Entropy Algorithm *(coder)* can be one of

- CABAC - Context Adaptive Binary Arithmetic Coding
- CAVLC - Context Adaptive Variable Length Encoding
- AC - Arithmetic Coding
- VLC - Variable Lenght Encoding

See also this

Rate control, can be one of

- ConstQP - Constant Quantization P mode
- VBR - Variable Bit Rate
- CBR - Constant Bit Rate
- CBR_LD_HQ - Constant Bit Rate with Low Delay, High Quality mode
- CBR_HQ - Constant Bit Rate High Quality Mode
- VBR_HQ - Variable Bit Rate High Quality Mode

Chroma subsampling Can be YUV 4:2:0, 4:2:2, 4:4:4. The libx264 codec has more control over the macroblocks when compared with NVenc. Other codecs support other sampling modes, such as YUV 4:1:1, and quantization modes for 10bit, 12bit, 16bit signals, besides RGB.

Definitions Y versus Y': Relative Luminance Y Relative luminance Y as defined by the CIE refers to a weighted sum of coefficients provided by the color matching functions and the RGB primaries of the color space of the signal. It is a linear quantity or **scene-linear**, in that there is no color component transfer function assigned to it in order to compensate for the characteristics of the human visual system.

Definitions Y versus Y': Luma Y' or Y prime In order to differentiate relative luminance from a signal with color component transfer function attached, such as a simple power function with a *gamma* exponent it was decided to differentiate the luminance Y used in colorimetry from the luminance used in video, by naming it instead **luma**, and assigning it the designation **Y/'** or Y prime, which is also documented under the SMPTE Engineering Guideline Annotated Glossary of Essential Terms for Electronic Production. In real life however, when referring to video, it's rarely seen the use the **Y'** instead of **Y**, but we're referring to the *gamma compressed* **luma Y'** here.

See also YUV and Luminance Considered Harmful

---

[1]:
```
#!ffmpeg -h encoder=libx264
```

1.6 Video resources, configuration

Start by defining a **VIDEO_RESOURCES** environment variable pointing to the location of your original videos, and **VIDEO_COMPRESSED** environment variable pointing to the location to use for storing the resulting compressed videos.

[2]:
```
import media as md
import options as op
import os
```

1.6.1 Create a new Media object

A media class was created accessible here, media.py to encapsulate the options used for each encoder in order to automate the creation of compressed output. You can import the object and check its built-in help and methods in Python in the following snippet:

```
import media as md
media = md.Media()
# help on the class
help(md)
# methods available
dir(md)
```

[3]:
```
media = md.Media()
```

[4]:
```
print(f"source material dir:\n{media.input_dir},\noutput dir:\n{media.
↪output_dir}")
```

```
source material dir:
/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/video
s/original,
output dir:
/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/video
s/compressed
```

We'll glob all the files under the input dir with the following extensions (the following containers)

[5]:
```
print(media.containers())
```

```
['mp4', 'mkv', 'webm']
```

The source material that we'll be using is

[6]:
```
media.glob_media()
media.input_files()
```

[6]:
```
['/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/vid
eos/original/light_orbitals.mkv',
 '/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/vid
eos/original/colored_orbitals.mkv',
```

```
 '/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/vid
eos/original/flowlock.mkv',
 '/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/vid
eos/original/flowfield.mkv',
 '/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/vid
eos/original/noiseywaves.mkv']
```

Add your own directory and set the environment variables, and copy the videos you want to analyze there. Ideally, YUV 4:4:4 chroma sampling, and with the highest quality possible since these are to be the reference sources against with compressed material is going to be compared.

[7]:
```
# same for the output files, but these are the output basenames only, we want␣
 ↪the
# qualified output filenames, with the encoded parameters as part of the␣
 ↪filename
# media.output_files()
```

We can view a video thumbnail, for example, the first video in the list

[8]:
```
import ffmpeg

from ipywidgets import interact
from matplotlib import pyplot as plt
import ipywidgets as widgets
import numpy as np
```

1.6.2 Media information

We'll store the media information in a dictionary to be queried later. Information such as the video stream data, width, height, aspect ratio, pixel format & chroma sampling, color range, color component transfer functions, RGB primaries and whitepoint, frame rate, time base, and so on.

[9]:
```
info = media.info()
```

[10]:
```
# get the width and height of the first media file in the input list
width = media.width(media.input_files()[0])
height = media.height(media.input_files()[0])
print(f"input file {media.input_files()[0]} height is {height}, width is␣
 ↪{width}")
```

```
input file /home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM
/Work/videos/original/light_orbitals.mkv height is 720, width is 1280
```

Viewing the Test Clip(s) Prepare the widget for the thumbnail view with the FFMPEG python bindings and NumPy

[11]:
```
out, err = (
    ffmpeg
```

```
    .input(media.input_files()[0])
    .output("pipe:", format="rawvideo", pix_fmt="rgb24", loglevel="error")
    .run(capture_stdout = True)
)

video = (
    np
    .frombuffer(out, np.uint8)
    .reshape([-1, height, width, 3])
)

@interact(frame=(0, media.number_of_frames(media.input_files()[0])))
def show_frame(frame = 0):
    plt.imshow(video[frame,:,:,:])
```

```
interactive(children=(IntSlider(value=0, description='frame', max=256),␣
↪Output()), _dom_classes=('widget-inter…
```

1.6.3 Available Codecs

We can check codecs with `ffmpeg -codecs` and specific options with `ffmpeg -h encoder=<encoder here>`, example for the h264 case, `ffmpeg -h encoder=libx264`. In our case, we're interested in h264, h264/HEVC, h266/VVC, VP8, VP9, AV1.

[12]:
```
!ffmpeg -hide_banner -loglevel error -codecs|grep -wE␣
↪'av1|vp8|vp9|h264|hevc|vvc'
```

```
 DEV.L. av1                  Alliance for Open Media AV1 (decoders: libdav1d
libaom-av1 av1 ) (encoders: libaom-av1 libsvtav1 )
 DEV.LS h264                 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10
(decoders: h264 h264_v4l2m2m h264_cuvid ) (encoders: libx264 libx264rgb
h264_nvenc h264_v4l2m2m nvenc nvenc_h264 )
 DEV.L. hevc                 H.265 / HEVC (High Efficiency Video Coding)
(decoders: hevc hevc_v4l2m2m hevc_cuvid ) (encoders: libx265 nvenc_hevc
hevc_nvenc hevc_v4l2m2m libkvazaar )
 DEV.L. vp8                  On2 VP8 (decoders: vp8 vp8_v4l2m2m libvpx vp8_cuvid
) (encoders: libvpx vp8_v4l2m2m )
 DEV.L. vp9                  Google VP9 (decoders: vp9 vp9_v4l2m2m libvpx-vp9
vp9_cuvid ) (encoders: libvpx-vp9 )
 ..V.L. vvc                  H.266 / VVC (Versatile Video Coding)
```

1.7 Codec options

We'll then go one by one, starting with

1.7.1 H.264 | MPEG-4 AVC

We need to check the options for each family of codecs, though to keep it manageable, we'll be restricted ourselves to changing the CRF and presets. Further tests will be done later with varying

macroblock sizes, different GOP structures, motion estimation and compensation methods.

[13]:
```
!ffmpeg -hide_banner -loglevel error -h encoder=libx264
```

```
Encoder libx264 [libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10]:
    General capabilities: delay threads
    Threading capabilities: other
    Supported pixel formats: yuv420p yuvj420p yuv422p yuvj422p yuv444p yuvj444p
nv12 nv16 nv21 yuv420p10le yuv422p10le yuv444p10le nv20le
libx264 AVOptions:
  -preset            <string>     E..V… Set the encoding preset (cf. x264
--fullhelp) (default "medium")
  -tune              <string>     E..V… Tune the encoding params (cf. x264
--fullhelp)
  -profile           <string>     E..V… Set profile restrictions (cf. x264
--fullhelp)
  -fastfirstpass     <boolean>    E..V… Use fast settings when encoding
first pass (default true)
  -level             <string>     E..V… Specify level (as defined by Annex
A)
  -passlogfile       <string>     E..V… Filename for 2 pass stats
  -wpredp            <string>     E..V… Weighted prediction for P-frames
  -a53cc             <boolean>    E..V… Use A53 Closed Captions (if
available) (default true)
  -x264opts          <string>     E..V… x264 options
  -crf               <float>      E..V… Select the quality for constant
quality mode (from -1 to FLT_MAX) (default -1)
  -crf_max           <float>      E..V… In CRF mode, prevents VBV from
lowering quality beyond this point. (from -1 to FLT_MAX) (default -1)
  -qp                <int>        E..V… Constant quantization parameter
rate control method (from -1 to INT_MAX) (default -1)
  -aq-mode           <int>        E..V… AQ method (from -1 to INT_MAX)
(default -1)
     none            0            E..V…
     variance        1            E..V… Variance AQ (complexity mask)
     autovariance    2            E..V… Auto-variance AQ
     autovariance-biased 3            E..V… Auto-variance AQ with bias to
dark scenes
  -aq-strength       <float>      E..V… AQ strength. Reduces blocking and
blurring in flat and textured areas. (from -1 to FLT_MAX) (default -1)
  -psy               <boolean>    E..V… Use psychovisual optimizations.
(default auto)
  -psy-rd            <string>     E..V… Strength of psychovisual
optimization, in <psy-rd>:<psy-trellis> format.
  -rc-lookahead      <int>        E..V… Number of frames to look ahead for
frametype and ratecontrol (from -1 to INT_MAX) (default -1)
  -weightb           <boolean>    E..V… Weighted prediction for B-frames.
(default auto)
```

```
  -weightp           <int>        E..V… Weighted prediction analysis
method. (from -1 to INT_MAX) (default -1)
     none            0            E..V…
     simple          1            E..V…
     smart           2            E..V…
  -ssim              <boolean>    E..V… Calculate and print SSIM stats.
(default auto)
  -intra-refresh     <boolean>    E..V… Use Periodic Intra Refresh instead
of IDR frames. (default auto)
  -bluray-compat     <boolean>    E..V… Bluray compatibility workarounds.
(default auto)
  -b-bias            <int>        E..V… Influences how often B-frames are
used (from INT_MIN to INT_MAX) (default INT_MIN)
  -b-pyramid         <int>        E..V… Keep some B-frames as references.
(from -1 to INT_MAX) (default -1)
     none            0            E..V…
     strict          1            E..V… Strictly hierarchical pyramid
     normal          2            E..V… Non-strict (not Blu-ray
compatible)
  -mixed-refs        <boolean>    E..V… One reference per partition, as
opposed to one reference per macroblock (default auto)
  -8x8dct            <boolean>    E..V… High profile 8x8 transform.
(default auto)
  -fast-pskip        <boolean>    E..V… (default auto)
  -aud               <boolean>    E..V… Use access unit delimiters.
(default auto)
  -mbtree            <boolean>    E..V… Use macroblock tree ratecontrol.
(default auto)
  -deblock           <string>     E..V… Loop filter parameters, in
<alpha:beta> form.
  -cplxblur          <float>      E..V… Reduce fluctuations in QP (before
curve compression) (from -1 to FLT_MAX) (default -1)
  -partitions        <string>     E..V… A comma-separated list of
partitions to consider. Possible values: p8x8, p4x4, b8x8, i8x8, i4x4, none, all
  -direct-pred       <int>        E..V… Direct MV prediction mode (from -1
to INT_MAX) (default -1)
     none            0            E..V…
     spatial         1            E..V…
     temporal        2            E..V…
     auto            3            E..V…
  -slice-max-size    <int>        E..V… Limit the size of each slice in
bytes (from -1 to INT_MAX) (default -1)
  -stats             <string>     E..V… Filename for 2 pass stats
  -nal-hrd           <int>        E..V… Signal HRD information (requires
vbv-bufsize; cbr not allowed in .mp4) (from -1 to INT_MAX) (default -1)
     none            0            E..V…
     vbr             1            E..V…
     cbr             2            E..V…
```

```
  -avcintra-class    <int>        E..V… AVC-Intra class 50/100/200 (from
-1 to 200) (default -1)
  -me_method         <int>        E..V… Set motion estimation method (from
-1 to 4) (default -1)
     dia             0            E..V…
     hex             1            E..V…
     umh             2            E..V…
     esa             3            E..V…
     tesa            4            E..V…
  -motion-est        <int>        E..V… Set motion estimation method (from
-1 to 4) (default -1)
     dia             0            E..V…
     hex             1            E..V…
     umh             2            E..V…
     esa             3            E..V…
     tesa            4            E..V…
  -forced-idr        <boolean>    E..V… If forcing keyframes, force them
as IDR frames. (default false)
  -coder             <int>        E..V… Coder type (from -1 to 1) (default
default)
     default         -1           E..V…
     cavlc           0            E..V…
     cabac           1            E..V…
     vlc             0            E..V…
     ac              1            E..V…
  -b_strategy        <int>        E..V… Strategy to choose between
I/P/B-frames (from -1 to 2) (default -1)
  -chromaoffset      <int>        E..V… QP difference between chroma and
luma (from INT_MIN to INT_MAX) (default 0)
  -sc_threshold      <int>        E..V… Scene change threshold (from
INT_MIN to INT_MAX) (default -1)
  -noise_reduction   <int>        E..V… Noise reduction (from INT_MIN to
INT_MAX) (default -1)
  -x264-params       <dictionary> E..V… Override the x264 configuration
using a :-separated list of key=value parameters
```

1.7.2 h265 | HEVC options

In this system, we have available libx265, kvazaar, and NVidia's hevc_nvenc. The settings provided by libx265 for h265 encoding are:

[14]:
```
# hevc, libx265, kvazaar, hevc_nvenc
!ffmpeg -hide_banner -loglevel error  -h encoder=libx265
```

```
Encoder libx265 [libx265 H.265 / HEVC]:
    General capabilities: delay threads
    Threading capabilities: other
```

```
    Supported pixel formats: yuv420p yuvj420p yuv422p yuvj422p yuv444p yuvj444p
gbrp yuv420p10le yuv422p10le yuv444p10le gbrp10le yuv420p12le yuv422p12le
yuv444p12le gbrp12le gray gray10le gray12le
libx265 AVOptions:
  -crf               <float>      E..V… set the x265 crf (from -1 to
FLT_MAX) (default -1)
  -qp                <int>        E..V… set the x265 qp (from -1 to
INT_MAX) (default -1)
  -forced-idr        <boolean>    E..V… if forcing keyframes, force them
as IDR frames (default false)
  -preset            <string>     E..V… set the x265 preset
  -tune              <string>     E..V… set the x265 tune parameter
  -profile           <string>     E..V… set the x265 profile
  -x265-params       <dictionary> E..V… set the x265 configuration using a
:-separated list of key=value parameters
```

1.7.3 H.266 | VVC

The h.266/VVC codec is known to FFMPEG, but without a supported encoder or decoder yet.

[15]:
```
!ffmpeg -hide_banner -loglevel error  -h encoder=vvc
```

```
Codec 'vvc' is known to FFmpeg, but no encoders for it are available. FFmpeg
might need to be recompiled with additional external libraries.
```

Though patches exists in development branches, these break video quality assessment modes, so this won't be covered here, yet. However we'll be providing a small guide on how to actually compress a video with the official Fraunhoffer provided reference encoder, VVenc. This implies in some systems, retrieving the source code from the Fraunhoffer github repository and compiling it, installing it into a system visible directory.

[16]:
```
#!ffmpeg -hide_banner -loglevel error  -h encoder=vvc
!vvencapp --help
```

```
vvencapp: Fraunhofer VVC Encoder ver. 1.3.0 [Linux][GCC 10.3.0][64
bit][SIMD=AVX2]

#======== General Options ================
        --help [0]              show default help
        --fullhelp [0]          show full help
  -v,   --verbosity [verbose]   Specifies the level of the verboseness (0:
silent, 1: error, 2: warning, 3: info, 4:
                                notice, 5: verbose, 6: debug)
        --version [0]           show version

#======== Input Options ================
  -i,   --input []              original YUV input file name or '-' for reading
from stdin
```

```
  -s,   --size [1920x1080]      specify input resolution (WidthxHeight)
  -c,   --format [yuv420]       set input format (yuv420, yuv420_10,
yuv420_10_packed)
  -r,   --framerate [60]        temporal rate (framerate numerator) e.g. 25,30,
30000, 50,60, 60000
        --framescale [1]        temporal scale (framerate denominator) e.g. 1,
1001
        --fps [60/1]            Framerate as int or fraction (num/denom)
        --tickspersec [90000]   Ticks Per Second for dts generation,
(1..27000000)
  -f,   --frames [0]            max. frames to encode [all]
  -fs,  --frameskip [0]         Number of frames to skip at start of input YUV
[off]
        --segment [off]         when encoding multiple separate segments,
specify segment position to enable segment concatenation (first, mid, last)
[off]
                                first: first segment
                                mid  : all segments between first and last
segment
                                last : last segment

#======== Output Options ================
  -o,   --output []             Bitstream output file name

#======== Encoder Options ================
        --preset [medium]       select preset for specific encoding setting
(faster, fast, medium, slow, slower)
  -b,   --bitrate [0]           bitrate for rate control (0: constant-QP
encoding without rate control, otherwise
                                bits/second)
  -p,   --passes [-1]           number of rate control passes (1,2)
        --pass [-1]             rate control pass for two-pass rate control
(-1,1,2)
        --rcstatsfile []        rate control statistics file
  -q,   --qp [32]               quantization parameter, QP (0-63)
        --qpa [on]              Enable perceptually motivated QP adaptation,
XPSNR based (0:off, 1:on)
  -t,   --threads [-1]          Number of threads default: [size < 720p: 4, >=
720p: 8]
  -g,   --gopsize [32]          GOP size of temporal structure (16,32)
  -rt,  --refreshtype [cra]     intra refresh type (idr,cra,idr2,cra_cre - CRA
with constrained encoding for RASL
                                pictures)
  -rs,  --refreshsec [1]        Intra period/refresh in seconds
  -ip,  --intraperiod [0]       Intra period in frames (0: use intra period in
seconds (refreshsec), else: n*gopsize)
        --tiles [1x1]           Set number of tile columns and rows
```

```
#======== Profile, Level, Tier ================
        --profile [auto]        select profile (main10, main10_stillpic)
        --level [auto]          Level limit (1.0, 2.0,2.1, 3.0,3.1, 4.0,4.1,
5.0,5.1,5.2, 6.0,6.1,6.2,6.3, 15.5)
        --tier [main]           Tier to use for interpretation of level (main or
high)

#======== HDR and Color Options ================
        --hdr [off]             set HDR mode (+SEI messages) + BT.709 or BT.2020
color space. use: off, pq|hdr10,
                                pq_2020|hdr10_2020, hlg, hlg_2020
```

1.7.4 VP8, VP9, AV1

The Open Media Alliance created VP8, VP9, VP1 to counteract the H.264|MPEG-4 AVC, H.265|HEVC potential licensing woes. H.266/VVC seems to be encumbered by licensing issues as well. VP8 was deprecated, superceded by VP9, which itself was superceded by AV1. FFMPEG provides an interface to VP8, VP9 and AV1. The options for VP9 and AV1 can also be specified by replacing `encoder=vp8` by `encoder=vp9` and `encoder=av1` respectively.

[17]:
```
!ffmpeg -hide_banner -loglevel error  -h encoder=vp8
```

```
Encoder libvpx [libvpx VP8]:
    General capabilities: delay threads
    Threading capabilities: other
    Supported pixel formats: yuv420p yuva420p
libvpx-vp8 encoder AVOptions:
  -lag-in-frames     <int>        E..V… Number of frames to look ahead for
alternate reference frame selection (from -1 to INT_MAX) (default -1)
  -arnr-maxframes    <int>        E..V… altref noise reduction max frame
count (from -1 to INT_MAX) (default -1)
  -arnr-strength     <int>        E..V… altref noise reduction filter
strength (from -1 to INT_MAX) (default -1)
  -arnr-type         <int>        E..V… altref noise reduction filter type
(from -1 to INT_MAX) (default -1)
     backward        1            E..V…
     forward         2            E..V…
     centered        3            E..V…
  -tune              <int>        E..V… Tune the encoding to a specific
scenario (from -1 to INT_MAX) (default -1)
     psnr            0            E..V…
     ssim            1            E..V…
  -deadline          <int>        E..V… Time to spend encoding, in
microseconds. (from INT_MIN to INT_MAX) (default good)
     best            0            E..V…
     good            1000000      E..V…
     realtime        1            E..V…
  -error-resilient   <flags>      E..V… Error resilience configuration
```

```
(default 0)
     default                     E..V… Improve resiliency against losses
of whole frames
     partitions                  E..V… The frame partitions are
independently decodable by the bool decoder, meaning that partitions can be
decoded even though earlier partitions have been lost. Note that intra
prediction is still done over the partition boundary.
  -max-intra-rate    <int>       E..V… Maximum I-frame bitrate (pct)
0=unlimited (from -1 to INT_MAX) (default -1)
  -crf               <int>       E..V… Select the quality for constant
quality mode (from -1 to 63) (default -1)
  -static-thresh     <int>       E..V… A change threshold on blocks below
which they will be skipped by the encoder (from 0 to INT_MAX) (default 0)
  -drop-threshold    <int>       E..V… Frame drop threshold (from INT_MIN
to INT_MAX) (default 0)
  -noise-sensitivity <int>       E..V… Noise sensitivity (from 0 to 4)
(default 0)
  -undershoot-pct    <int>       E..V… Datarate undershoot (min) target
(%) (from -1 to 100) (default -1)
  -overshoot-pct     <int>       E..V… Datarate overshoot (max) target
(%) (from -1 to 1000) (default -1)
  -ts-parameters     <dictionary> E..V… Temporal scaling configuration
using a :-separated list of key=value parameters
  -auto-alt-ref      <int>       E..V… Enable use of alternate reference
frames (2-pass only) (from -1 to 2) (default -1)
  -cpu-used          <int>       E..V… Quality/Speed ratio modifier (from
-16 to 16) (default 1)
  -speed             <int>       E..V…  (from -16 to 16) (default 1)
  -quality           <int>       E..V…  (from INT_MIN to INT_MAX)
(default good)
     best            0           E..V…
     good            1000000     E..V…
     realtime        1           E..V…
  -vp8flags          <flags>     E..V…  (default 0)
     error_resilient             E..V… enable error resilience
     altref                      E..V… enable use of alternate reference
frames (VP8/2-pass only)
  -arnr_max_frames   <int>       E..V… altref noise reduction max frame
count (from 0 to 15) (default 0)
  -arnr_strength     <int>       E..V… altref noise reduction filter
strength (from 0 to 6) (default 3)
  -arnr_type         <int>       E..V… altref noise reduction filter type
(from 1 to 3) (default 3)
  -rc_lookahead      <int>       E..V… Number of frames to look ahead for
alternate reference frame selection (from 0 to 25) (default 25)
  -sharpness         <int>       E..V… Increase sharpness at the expense
of lower PSNR (from -1 to 7) (default -1)
```

```
Encoder vp8_v4l2m2m [V4L2 mem2mem VP8 encoder wrapper]:
    General capabilities: delay hardware
    Threading capabilities: none
vp8_v4l2m2m_encoder AVOptions:
  -num_output_buffers <int>        E..V… Number of buffers in the output
context (from 6 to INT_MAX) (default 16)
  -num_capture_buffers <int>        E..V… Number of buffers in the capture
context (from 4 to INT_MAX) (default 4)
```

1.7.5 MPEG-5 EVC

MPEG-5 EVC support in FFMPEG is not native. Testing support for FFMPEG will results in the same message as the previous VVC case. However, experimental support is provided in a fork of FFMPEG via XEVE. Since development is detached from the main FFMPEG development, the video quality assessment methods we use are not yet available, but other methods might be employed at a later date. We won't be covering MPEG-5 EVC for now, but will do so later.

1.8 H.264 Compression: x264 options:

From the list of presets, *ultra/super/veryfast, faster, fast, medium, slow, slower, veryslow, placebo*, we will be using the tuning presets: film, animation, grain, stillimage, psnr, ssim, fastdecode, zerolatency.

Since hardware constraints, network or streaming limitations aren't considerations, the tuning will be restricted to *film, animation, grain*.

Chroma subsampling will be restricted to **YUV 4:2:0, YUV 4:2:2, YUV 4:4:4** with 8bit output. With higher bitdepth source content we can use 10bit support,i.e, `yuv422p10le`, see `x264 --fullhelp` for an exhaustive list of options for the source material.

Range is set to *pc*, so 8bit material is in `[0,255]` range, instead of `[16,235]` All these options can be changed via the provided Options.py Python class via their associated methods, which can be consulted via `dir(Options)`.

Motion estimation an take the values:

- **dia** - Diamond search, checks motion vectors one pixel up, left, down and right choosing the best candidate and repeated until no better motion vector is found.
- **hex** - Hexagon like the Diamond search, but searches instead in a radius of two pixels for six adjacent points. It's a good default since it's more efficient than the Diamond search and is very efficient.
- **umh** - Uneven Multi-Hexagonal search, has a `--me-range` parameter, defaulting to 16, to control the search radius, and is heavier than the regular Hexagonal search.
- **esa** - Exhaustive search, an optimized search of the complete motion vector space within the `--me-range` parameter, which defaults to 16.
- **tesa** - Transformed Exhaustive search, attempts to approximate the effect of running a Hadamard transform comparison at each motion vector.

Subpixel motion estimation will be left unchanged to mode 7, that is, Rate Distortion Optimization mode decision for all frames (I, P, B). Mode 9 and 10 will be tested later, that is, rate distortion refinement for all frames, and quarter pixel RD, which requires Trellis RD quantization mode 2 via the `--trellis` option.

Use `--slow-firstpass` with multipass approaches to keep good quality analysis settings before the actual encoding takes place on the 2nd pass.

Use `--b-adapt` with value 1=*Fast*, 2=*Optimal*, default being one, for the adaptive B-frame decision method.

Use `--keyint` with an integer for the maximum GOP size, 250 being the default.

Use `--qp` from 0 to 81, 0 being lossless, and `--crf` for quality based VBR, from -12 to 51, the default being 23.0.

Adaptive quantization method, `--aq-mode` 0 = Disabled, 1 = Variance AQ (complexity mask), 2 = Auto-Variance AQ, 3 = Auto-variance AQ with bias to dark values.

P-frame weighted prediction is controlled with `--weightp`, taking the values 0 = disabled, 1 = enable only weighted references, 2 = enable both weighted references and duplicates.

Quantization matrices will be left unchanged to the default H.264/AVC quantization matrices, *flat*.

[18]:
```
# Create an options container
options = op.Options()
```

Encode options are the codec specific encoding options passed to FFMPEG.

Common options are the global/general FFMPEG encoding options.

Encoding sets are the sets of codec specific or global encoding options that will be iterated in order to generate compressed tests with multiple parameters. Example, if you want to use *Constant Rate Factor* rate control mode to encode a H.264 video with libx264, and generate 4 outputs each with different CRF quality in order to evaluate the perceptual video quality metrics on each, you would store this in a `key:value` pair in the encoding sets, i.e: `{"crf" : [12, 28, 40],...}`

[19]:
```
#print(options.encode_options())
#print(options.common_options())
print(options.encoding_sets())
```

```
{'crf': [18, 27, 36], 'preset': ['veryfast', 'medium'], 'tune': ['film',
'grain'], 'motion-est': ['esa', 'umh'], 'aq-mode': [2, 3], 'weightp': [0, 1, 2],
'pix_fmt': ['yuv420p', 'yuv422p', 'yuv444p']}
```

Create an encoder object (from the file encoder.py you can find here), and this will be our interface between the videos, the compressors and their options, and FFMPEG.

[20]:
```
import encoder as enc
```

[21]:
```
encoder = enc.Encoder(media, options)
```

1.9 Compressed Output

A full combinatorial analysis, besides a naif brute force approach, would be a daunting task. Smarter approaches exist to use neural networks to identify the type of content and start exploring settings optimized for that family. In our case we'll do tests for individual sets of parameters, run perceptual video quality metrics on them, and the other metrics outlined earlier, compare resulting metrics versus file size, and plot them.

We'll start by constant rate factor, presets, tuning presets, chroma subsampling methods, and motion estimation methods.

The CRF rate control can be capped, also known as **cap-constrained CRF** and this will be calculated according to the maximum target size we'll allow. Example, from a 50MiB file the desired target bitrate will be calculated, with the joining maximum rate and buffer size to keep the rate control constrained within a maximum 200% maximum rate.

1.10 Compressing, Calculating Metrics, Interpreting Results

We'll calculate the PSNR, SSIM, VMAF per frame metrics, store them into a Pandas dataframe and compute the mean and moving averages, plotting them next. A brief overview of the metrics ensues then.

1.10.1 SSIM and MS-SSIM

Structured Similarity Index and Multi-Scale Structural Similarity (source code here) is one of the most well-known image quality evaluation algorithms and computes relative quality scores between the reference and distorted images by comparing details across resolutions, providing high performance for learning-based image codecs. The MS-SSIM is more flexible than single-scale methods such as SSIM by including variations of image resolution and viewing conditions. Also, the MS-SSIM metric introduces an image synthesis-based approach to calibrate the parameters that weight the relative importance between different scales. A high score expresses better image quality.

1.10.2 PSNR

Peak Signal to Noise Ratio. Peak Signal-to-Noise Ratio is widely used as a video quality metric or performance indicator. Some studies have indicated that it correlates poorly with subjective quality, whilst others have used it on the basis that it provides a good correlation with subjective data.

1.10.3 VIF

The Visual Information Fidelity (source code here) measures the loss of human perceived information in some degradation process, e.g. image compression. VIF exploits the natural scene statistics to evaluate information fidelity and is related to the Shannon mutual information between the degraded and original pristine image. The VIF metric operates in the wavelet domain and many experiments found that the metric values agree well with the human response, which also occurs for learning-based image codecs. A high score expresses better image quality.

1.10.4 VMAF

The Video Multimethod Assessment Fusion metric and score, developed by Netflix is focused on artifacts created by compression and rescaling and estimates the quality score by computing scores from several quality assessment algorithms and fusing them with a support vector machine (SVM, *a supervized ML method for data classification and regression*). Even if this metric is specific for videos, it can also be used to evaluate the quality of single images and has been proved that performs reasonably well for learning-based image codecs. Since the metric takes as input raw images in the YUV color space format, the PNG (RGB color space) images are converted to the YUV 4:4:4 format using FFMPEG (BT.709 primaries). A higher score of this metric indicates better image quality. Built-in support in FFMPEG is provided via libvmaf.

VMAF Score The VMAF score will be using the model we've chosen to provides us a score, so what it's actually telling us will change a bit depending on the model. For our model, its predicting quality of the video on a 1080p display in a *living room environment*, and presuming that the persons viewing distance is 3x the height of the screen (25cm monitor/TV height = 75cm viewing distance). We consider a score of 20 to be very bad, and a score of 100 to be excellent (flawless). I would consider anything above 80 to be quite good, and above 90 to very close perfect/indistinguishable from the ref.

1.10.5 Metrics interface

The interface to the video quality metrics is provided by a Python library called FFMPEG Quality Metrics. This is one of the dependencies of the project, the full list of which will be listed at the end. We'll be importing it, and use our encoder object, as well as Pandas dataframes for statistical containers and statistical analysis, and this module with the metrics outlined above for each of the videos encoded with the options we defined in our options object.

[22]:
```
from ffmpeg_quality_metrics import FfmpegQualityMetrics as ffqm
import pandas as pd
```

Example, with a given test video and given options:

[23]:
```
display(media.input_files()[0])
display(options.common_options())
display(options.encode_options())
```

```
'/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/
 ↪videos/original/light_orbitals.mkv'
```

```
{'c:v': 'libx264',
 'f': 'mp4',
 'coder': 'cabac',
 'color_range': 'pc',
 'loglevel': 'error',
 'export_side_data': 'venc_params'}
```

```
{'crf': 23,
 'preset': 'veryfast',
 'tune': 'film',
```

```
'motion-est': 'esa',
'aq-mode': -1,
'weightp': -1,
'pix_fmt': 'yuv420p'}
```

[24]:
```
encoder.encode_video(
    media.input_files()[0],
    media.output_files()[0],
    {
        **options.common_options(),
        **options.encode_options()
    },
    debug=False)
```

We'll be calculating the structural similarity index for this given video, and as a result we should have a dictionary with the metrics computed, each containing an array of dictionaries, each with the respective metrics and auxiliary values for each frame n

[25]:
```
metrics = ffqm(
    media.input_files()[0],
    media.output_files()[0],
    progress = True)
```

[26]:
```
metrics_data = metrics.calc(["ssim"])
```

```
ssim:   0%|                                                  | 0/100 [00:00<?,
?it/s]
ssim:   0%|                                                  | 0/100 [00:00<?,
?it/s]
ssim:  40%|                              | 40/100 [00:00<00:00,
82.03it/s]
ssim:  80%|                         | 80/100 [00:01<00:00,
80.48it/s]
ssim: 100%|                    | 100/100 [00:01<00:00,
75.80it/s]
```

Now we convert to a Pandas DataFrame for quick monitoring. This will allow us to have easy data access besides performing common time series options, and plotting.

Note that with a huge series of files to test and more metrics, these steps might take a bit of time and it would therefore make sense to save the metrics dictionaries output as JSON files, then at a later date to plot them. That is, just load the saved JSON files into a Pandas DataFrame, and do your Pandas operations there before plotting them. Converting to a Pandas DataFrame before saving would make it harder to flatten as a JSON file.

[27]:
```
encoder.encode_video(
    media.input_files()[0],
    media.output_files()[0],
```

```
    {
        **options.common_options(),
        **options.encode_options()
    },
    debug=False)
```

For the short given the SSIM score across the Y, u and v channels as well as the average are given below from the Pandas dataframe.

[28]:
```
# print(ssim)
df = pd.DataFrame.from_dict(metrics_data["ssim"])
df.tail()
```

[28]:

	n	ssim_y	ssim_u	ssim_v	ssim_avg
252	253	0.950	0.955	0.929	0.944
253	254	0.950	0.955	0.927	0.944
254	255	0.951	0.956	0.931	0.946
255	256	0.950	0.956	0.932	0.946
256	257	0.949	0.956	0.931	0.945

1.10.6 Processing the Results

Now we use the Pandas functionality to clean, filter, organize our dataframe.

- Rename the *n* column as *frame*
- Set the new *frame* column as the index
- Fill the NaN values at row 0 with 0.

Computing the mean, harmonic mean, and simple or exponential moving averages (i.e, per second (25 frames) or 2 seconds (50 frames)) smoothes the data on a temporal basis, and gives some insights on metric variation, since the per-frame statistics might be too noise, reflecting sudden changes in frame, scene or shot content, motion, image data.

[29]:
```
if "n" in df.columns:
    df.rename(columns={"n" : "frame"}, inplace = True)
    df.set_index("frame", inplace = True)

#df.fillna(0, inplace = True)
```

The structural similarity index is meaningful on the luma Y' channel, and we can use the dataframe `describe()` method to get its arithmetic mean, minimum, maximum, and standard deviation from the mean, as well as the percentiles.

The arithmetic mean would at first seem to be a good estimate of a video quality, but it might hide difficult to encode frames. We can have a better grasp of the compression results by extracting the 25%, 50%, 75% percentiles as well, and by adding the simple and exponential moving averages as well and plotting everything (a popular approach in econometrics, time-series forecasting and stochastic analysis, just to name a few examples).

[30]:
```
df.describe()
```

[30]:

	ssim_y	ssim_u	ssim_v	ssim_avg
count	257.000000	257.000000	257.000000	257.000000
mean	0.952257	0.955183	0.926304	0.944584
std	0.010840	0.005449	0.009118	0.008070
min	0.938000	0.944000	0.909000	0.932000
25%	0.942000	0.951000	0.918000	0.938000
50%	0.949000	0.956000	0.927000	0.944000
75%	0.961000	0.959000	0.934000	0.952000
max	0.979000	0.968000	0.948000	0.963000

The window for a SMA would be for instance, two seconds at the media frame rate, i.e, 50 (for 25 progressive frames media). Structured Similarity Index Matrix is applied generally over the luma channel only, but the dataframe has the values for the U and V channels as well as their average. We'll be applied the SMA(2*framerate) and EMA(framerate) over the* ssim_y* channel only.

[31]:
```
media.framerate(media.input_files()[0])
```

[31]: 25

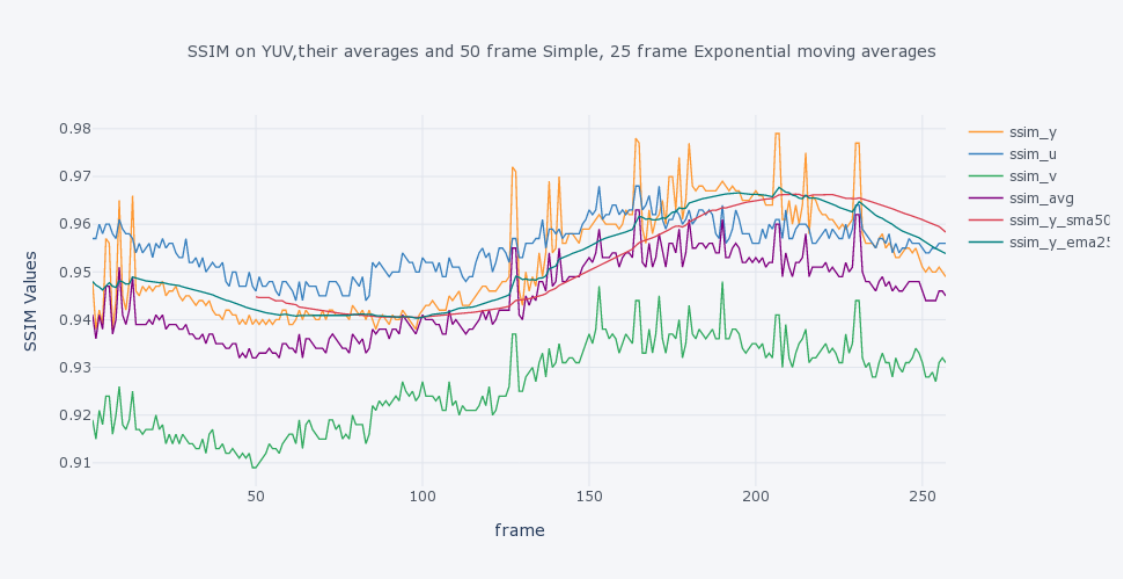
[32]:
```
#df.rolling(window=media.framerate(media.input_files()[0])).mean()
#df.ewm(span=media.framerate(media.input_files()[0])).mean()

framerate = media.framerate(media.input_files()[0])

df[f"ssim_y_sma{2*framerate}"] = df["ssim_y"].rolling(window=2*framerate,␣
↪min_periods=2*framerate).mean()
df[f"ssim_y_ema{framerate}"] = df["ssim_y"].ewm(span=framerate, adjust=False).
↪mean()
```

Plotting Now we import the Python cufflinks module in order to override the built-in Pandas `plot()` functions, which make use of MatplotLib, by Plotly, which has a more dynamic functionality well suited to data exploration and analysis.

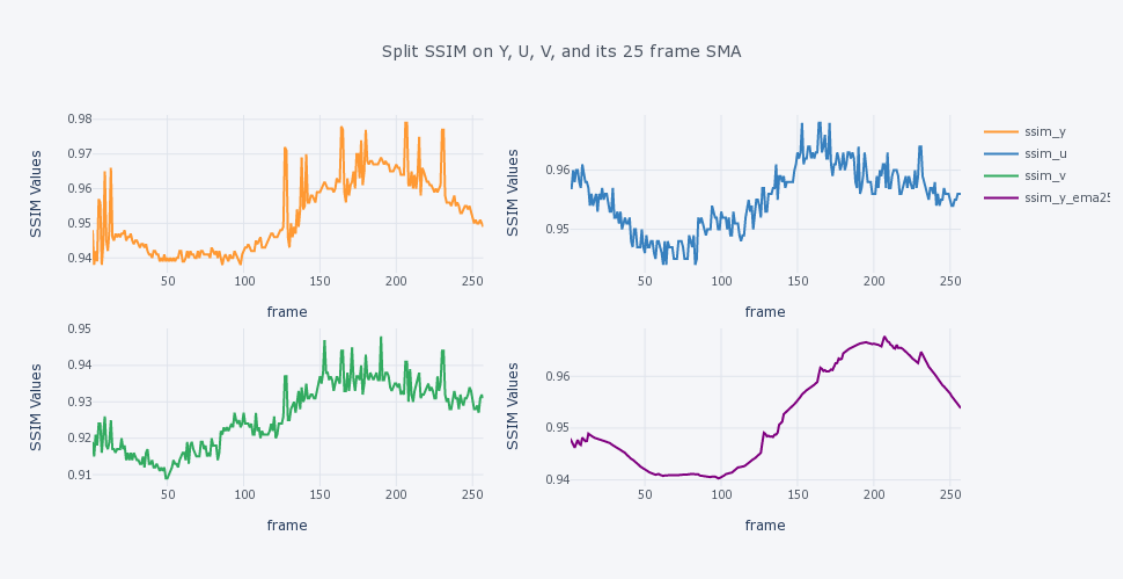
[33]:
```
import cufflinks as cf
```

[34]:
```
#  ['ggplot', 'pearl', 'solar', 'space', 'white', 'polar', 'henanigans']
cf.set_config_file(theme="pearl",sharing="private",offline=True)
```

[35]:
```
fig = df.iplot(title="SSIM on YUV,their averages and 50 frame Simple, 25 frame␣
↪Exponential moving averages", asFigure=True)
fig.update_layout(title_font=dict(size=14),
                  xaxis_title="frame",
                  yaxis_title="SSIM Values",
                  width=900, height=500, title_x=0.5)
```



[36]:
```
fig = df.iplot(y=["ssim_y", "ssim_u", "ssim_v", "ssim_y_ema25"],subplots=True,␣
↪width=2.0, asFigure=True)
fig.update_layout(
                  title=dict(text="Split SSIM on Y, U, V, and its 25 frame␣
↪SMA", font=dict(size=14)),
                  xaxis_title="frame", yaxis_title="SSIM Values",
                  xaxis2_title="frame", yaxis2_title="SSIM Values",
                  xaxis3_title="frame", yaxis3_title="SSIM Values",
                  xaxis4_title="frame", yaxis4_title="SSIM Values",␣
↪font=dict(size=10),
                  width=900, height=500, title_x=0.5,
                 )
```



Initial Observations For the given clip in this notebook which you can view here we can see is that in general the SSIM values over the luma Y' channel are above 0.94 for the entire length of the clip and spike to near 0.98 around the second half of the clip, as the moving averages also show, falling again.

Since we had a constant quality rate factor for this test- therefore instead of using a fixed value quantization matrix we allowed it to vary as long as it is bounded by the crf value - compression was more efficient and retained more details compared with the original on the second half of the clip.

Another observation is that the the V channel had the same fluctuations but in general had much lower SSIM values than the U channel which was consistently above the V channel SSIM value. Noteworthy is also the more subdued impact of the adaptive quantization of the CRF mode on the U channel, since its variation is much less than the luma Y' and V channel - around 0.005 standard deviations from the mean, compared with the V channel approx. 0.009.

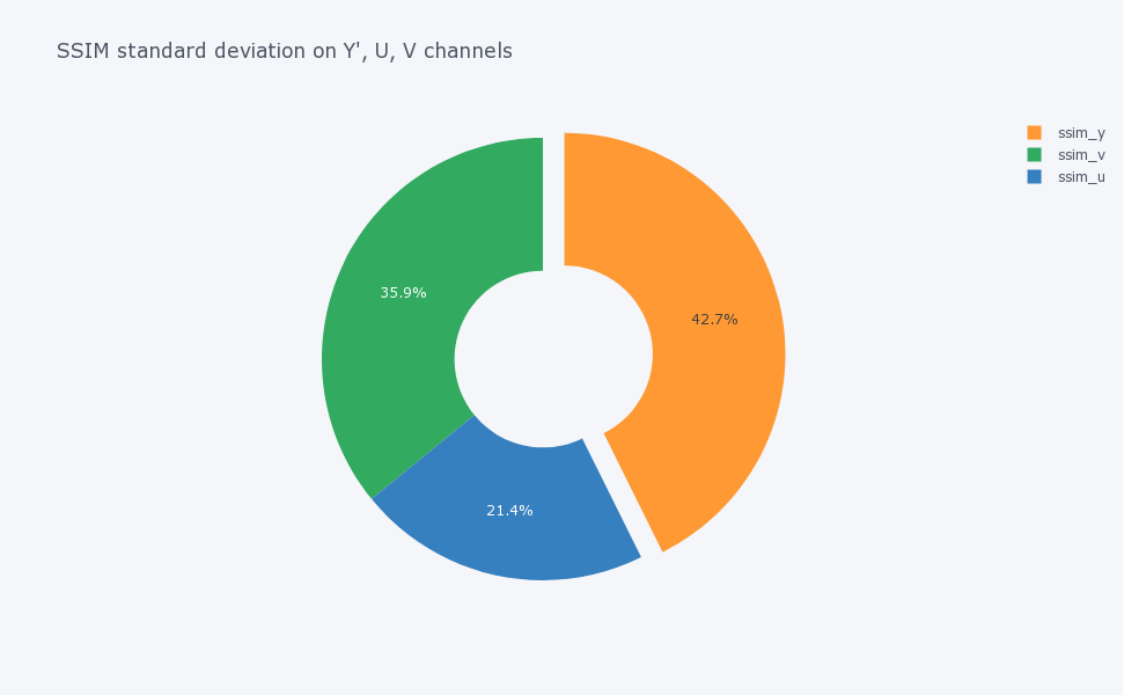
[37]:
```
# make a copy of the dataframe, transpose, prepare for a pie chart to compare␣
↪the
# standard deviation to the mean of the SSIM for the Y, U, V channels.

df2 = (df.describe()).copy(deep=True)
df2.drop(columns=["ssim_y_ema25", "ssim_y_sma50", "ssim_avg"], inplace=True)
df2 = df2.transpose()

df2.reset_index(inplace = True)
df2.head()
```

[37]:

	index	count	mean	std	min	25%	50%	75%	max
0	ssim_y	257.0	0.952257	0.010840	0.938	0.942	0.949	0.961	0.979
1	ssim_u	257.0	0.955183	0.005449	0.944	0.951	0.956	0.959	0.968
2	ssim_v	257.0	0.926304	0.009118	0.909	0.918	0.927	0.934	0.948

[38]:
```
df2.rename(columns={"index" : "ssim"}, inplace=True)
```

[39]:
```
fig = df2.iplot(kind="pie", labels="ssim", values="std", hole=0.4,
          pull=[0.1, 0, 0], title="SSIM standard deviation on Y\', U, V␣
 ↪channels", asFigure=True)
fig.update_layout(width=900, height=600)
```



It seems very clear that compression fluctuations are predominent, as expected, on the luma Y' channel. The U and V channels fluctuate a lot less but this is expected to vary according to the scene content, albeit in a more or less decorrelated way in regard to Y'.

Test and Export to JSON Before proceeding with the full compression suite and analysis we'll be running the PSNR, SSIM, VMAF and VIF metrics on a reference video, export the data to a file as a JSON file, then load it and convert it to a Pandas DataFrame to avoid having to recompute everything for each compressed video.

The next step will compute the metrics for a reference video.

[40]:
```
metrics_data = metrics.calc(["psnr", "ssim", "vmaf", "vif"])
```

```
psnr, ssim, vmaf, vif:   0%|                                | 0/100 [00:00<?,
?it/s]
psnr, ssim, vmaf, vif:   0%|                                | 0/100 [00:00<?,
?it/s]
psnr, ssim, vmaf, vif:   0%|                                | 0/100 [00:00<?,
?it/s]
psnr, ssim, vmaf, vif:   0%|                                | 0/100 [00:01<?,
?it/s]
psnr, ssim, vmaf, vif:   0%|                                | 0/100 [00:01<?,
?it/s]
psnr, ssim, vmaf, vif:  10%|                  | 10/100 [00:02<00:04,
18.44it/s]
psnr, ssim, vmaf, vif:  20%|                 | 20/100 [00:03<00:09,
8.55it/s]
psnr, ssim, vmaf, vif:  29%|                | 29/100 [00:05<00:10,
7.07it/s]
psnr, ssim, vmaf, vif:  30%|                | 30/100 [00:05<00:11,
6.04it/s]
psnr, ssim, vmaf, vif:  39%|               | 39/100 [00:06<00:08,
7.00it/s]
psnr, ssim, vmaf, vif:  40%|               | 40/100 [00:07<00:10,
5.97it/s]
psnr, ssim, vmaf, vif:  50%|              | 50/100 [00:09<00:08,
6.11it/s]
psnr, ssim, vmaf, vif:  59%|             | 59/100 [00:10<00:06,
6.82it/s]
psnr, ssim, vmaf, vif:  60%|             | 60/100 [00:10<00:06,
5.86it/s]
psnr, ssim, vmaf, vif:  69%|            | 69/100 [00:11<00:04,
6.69it/s]
psnr, ssim, vmaf, vif:  70%|            | 70/100 [00:12<00:05,
5.81it/s]
psnr, ssim, vmaf, vif:  79%|           | 79/100 [00:13<00:03,
6.86it/s]
psnr, ssim, vmaf, vif:  80%|           | 80/100 [00:13<00:03,
5.88it/s]
psnr, ssim, vmaf, vif:  89%|          | 89/100 [00:14<00:01,
6.99it/s]
psnr, ssim, vmaf, vif:  90%|          | 90/100 [00:15<00:01,
5.94it/s]
psnr, ssim, vmaf, vif:  99%|         | 99/100 [00:16<00:00,
6.89it/s]
psnr, ssim, vmaf, vif: 100%|         | 100/100 [00:17<00:00,
5.86it/s]
```

[41]:
```
import ffmpeg, json, os
```

```
import encoder as enc
import media as md
import options as op

from ipywidgets import interact
from matplotlib import pyplot as plt
import ipywidgets as widgets
import numpy as np

# FFQM metrics
from ffmpeg_quality_metrics import FfmpegQualityMetrics as ffqm
# Pandas
import pandas as pd

# plotly cufflinks and configuration
import cufflinks as cf
#  ['ggplot', 'pearl', 'solar', 'space', 'white', 'polar', 'henanigans']
cf.set_config_file(theme="pearl",sharing="private",offline=True)
```

Saving a DataFrame as JSON Export the metrics data into a JSON file for later reuse, with parameter qualification.

[42]:
```
fname = media.output_files()[0].split(".")[0] + ".json"

metrics_json = json.dumps(metrics_data, indent = 4)
#print(metrics_json)
media.output_dir
#
with open(fname, "w") as f:
    f.write(metrics_json)
```

Load the JSON file Now we can load the JSON file again, and create an array of Pandas DataFrames, each one for each metric, as well as computing a simple moving average with a window of 25 frames in order to smooth the data and have a better overview of the general tendency of the results.

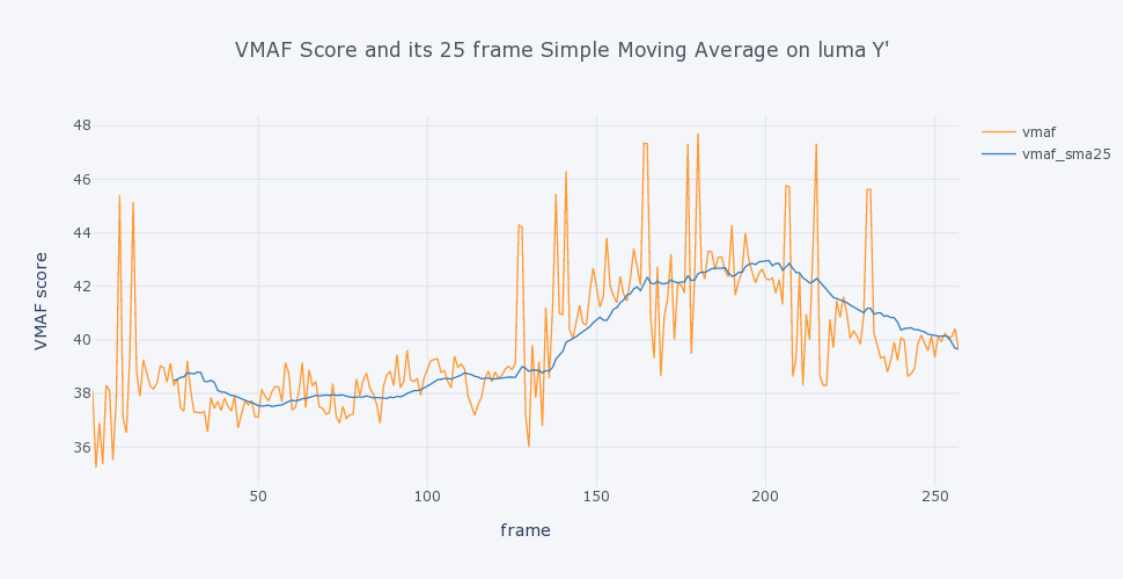
[43]:
```
metrics_dict = {}

with open(fname, "r") as f:
    metrics_dict = json.load(f)
```

[44]:
```
vmetrics = ["psnr", "ssim", "vmaf", "vif"]
framerate = 25

dfs = {}

for metric in vmetrics:
```

```
    df = pd.DataFrame.from_dict(metrics_dict[metric])

    if "n" in df.columns:
        df.rename(columns={"n" : "frame"}, inplace = True)
        df.set_index("frame", inplace = True)

    if metric == "psnr":
        df["psnr_y_sma25"] = df["psnr_y"].rolling(window=framerate,␣
↪min_periods=framerate).mean()
        df["mse_y_sma25"] = df["mse_y"].rolling(window=framerate,␣
↪min_periods=framerate).mean()

    elif metric == "ssim":
        df["psnr_y_sma25"] = df["ssim_y"].rolling(window=framerate,␣
↪min_periods=framerate).mean()

    elif metric == "vmaf":
        df["vmaf_sma25"] = df["vmaf"].rolling(window=framerate,␣
↪min_periods=framerate).mean()
        df["ms_ssim_sma25"] = df["ms_ssim"].rolling(window=framerate,␣
↪min_periods=framerate).mean()
        df["ssim_sma25"] = df["ssim"].rolling(window=framerate,␣
↪min_periods=framerate).mean()
        df["psnr_sma25"] = df["psnr"].rolling(window=framerate,␣
↪min_periods=framerate).mean()

    else:
        pass

    dfs[metric] = df.copy(deep = True)  # must be deep copy
```

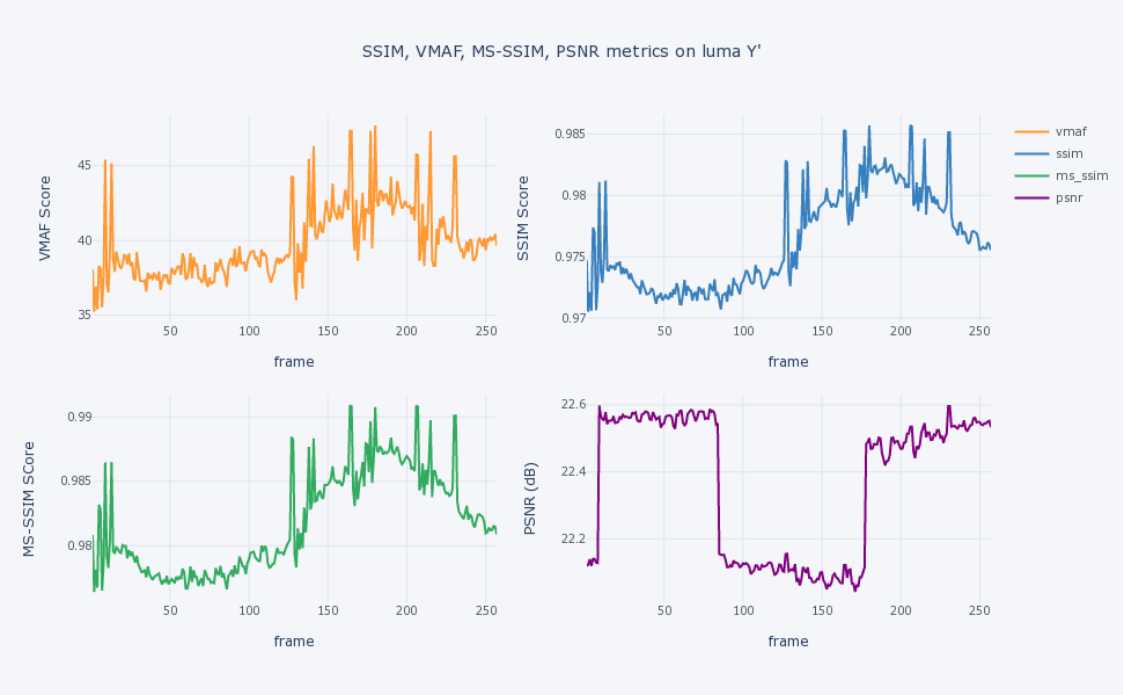
Starting by the VMAF metric and its 25 frames (1s) simple moving average.

[45]:
```
fig = dfs["vmaf"].iplot(y=["vmaf", "vmaf_sma25"],
                        title="VMAF Score and its 25 frame Simple Moving␣
↪Average on luma Y'", asFigure=True)

fig.update_xaxes(title="frame")
fig.update_yaxes(title="VMAF score")
fig.update_layout(width=900, height=500, title_x=0.5)
```



Initial Analysis For this test video, the VMAF score seems consistent with the earlier SSIM plots. The video stars with relatively low quality, has a peak in perceptual quality, showing a increase in effiency of the compression, then decreases and is centered around a score of 37 roughly until halfway, and then starts increasing in quality roughly halfway with irregular fluctuations, but an overall increasing tendency, which means that less bits were needed to have a result perceptually closer to the original source material, than in the first half, where the characteristics of the scene are showing a lower SSIM and VMAF. Multi-scale SSIM might also sched some light, as well as the other objective non-perceptual metrics.

[46]:
```
fig = dfs["vmaf"].iplot(y=["vmaf", "ssim", "ms_ssim", "psnr"],subplots=True,␣
↪width=2.0, asFigure=True)

fig.update
fig.update_layout(title="SSIM, VMAF, MS-SSIM, PSNR metrics on luma Y'",
                  xaxis_title="frame", yaxis_title="VMAF Score",
                  xaxis2_title="frame", yaxis2_title="SSIM Score",
                  xaxis3_title="frame", yaxis3_title="MS-SSIM SCore",
                  xaxis4_title="frame", yaxis4_title="PSNR (dB)",␣
↪font=dict(size=10),
                  width=900, height=600, title_x=0.5,
                 )
```



Regarding VMAF, the overall score follows the same pattern as the structured similarity and multi-scale structured similarity metrics. The PSNR metric however is a result which is relatively difficult to relate with the perceptual metrics, specially when it comes to moving images and the characteristics of the HVS, however in general a higher value reflects a higher quality reconstruction of the signal. Here it seems a value of 22.6dB is achieved until around frame 90, then drops to around 20dB, and peaks again to around 25dB from frame 180 onwards. There seems to be some detachment from the PSNR values and the perceptual video but there's some expectation for this general detachment - that is - a change in the PSNR value doesn't necessarily imply this variation will be noticeable by the HVS. For 8bit material, a value between 30dB and 50dB is considered lossy. Since PSNR has a mean squared error term in the denomimator of one of the terms of its equation, for two identical frames the error is 0, and the PSNR unbounded/undefined, but a higher PSNR points to higher quality.

1.11 Video Sources

Python classes were created to automate as much as possible encoding of sources with the variations presets. Next we'll be encoding all the input videos with the parameter sets that will be defined and commented as we go. Check the MediaTest class file for more information.

1.11.1 Encoding

[47]:
```
from mediatests import MediaTests
```

[48]:
```
mt = MediaTests()
```

Prepare the container with all the source material under the input directory described earlier in

the media class and its methods, and assemble the fully qualified names after setting the encoding options that will be used. Have in mind that, a large number of files, and encoding methods will take time to process and storage will also be used.

[49]:
```
# Shows the default encoding sets
mt.options.encoding_sets()
```

[49]:
```
{'crf': [18, 27, 36],
 'preset': ['veryfast', 'medium'],
 'tune': ['film', 'grain'],
 'motion-est': ['esa', 'umh'],
 'aq-mode': [2, 3],
 'weightp': [0, 1, 2],
 'pix_fmt': ['yuv420p', 'yuv422p', 'yuv444p']}
```

We'll be using h264, a CRF of 18, 25, and 36, exhaustive and multi-hexagonal searchs for motion estimation since our clip has erratic movement. Adaptive quantization based on variance (complexity mask), automatic, and biased towards dark values is commented out to keep the number of files low. See the libx264 encoding options here We'll be using diferent chroma subsampling methods as well.

[50]:
```
h264_encode_set = {
    "preset" : ["veryfast"],
    "crf" : [18, 25, 36],
    "motion-est" : ["esa", "umh"],
#    "aq-mode" : [1, 2, 3],
    "pix_fmt" : ["yuv420p", "yuv422p", "yuv444p"]}
```

[51]:
```
mt.options.insert_encoding_sets(h264_encode_set, replace=True)
```

[52]:
```
mt.options.encoding_sets()
```

[52]:
```
{'preset': ['veryfast'],
 'crf': [18, 25, 36],
 'motion-est': ['esa', 'umh'],
 'pix_fmt': ['yuv420p', 'yuv422p', 'yuv444p']}
```

Now prepare the structure, feeding in the source material, and building the fully qualified output filenames for each variation of the encoding set, to be stored in the compressed material structure. We'll be calling our Encoder() methods but via our MediaContainer() class to iterate over this and process the videos. Once compressed we'll iterate over this to generate the video quality assessment in regard to the original file. Methods are provided to, once done, filter easily by codec family, parameter sets, filenames, and by video quality metrics, so that one can plot in regard to these group criteria.

[53]:
```
# we can also pass a ```options.py``` instance to this method, optionally.
mt.prepare_media()
```

[54]:
```
# show the main data structure (it's deeply nested)
mt.media_container.keys()
```

[54]:
```
dict_keys(['inputdir', 'inputdata', 'outputdir', 'outputdata'])
```

[55]:
```
mt.options.encoding_sets()
```

[55]:
```
{'preset': ['veryfast'],
 'crf': [18, 25, 36],
 'motion-est': ['esa', 'umh'],
 'pix_fmt': ['yuv420p', 'yuv422p', 'yuv444p']}
```

1.11.2 Batch Encoding and Tests

Accessing via `mt.media_container["inputdir"]` will show you the directory for the ingested material, with `["inputdata"]`containing the found footage there, and the analysis of the source material - basic statistics such as frame rate, size, and so on. The keys `["outputdir"]` point to the destination for the compressed files, and `["outputdata"]` shows a very deeply nested structure containing the variations. You can query it in order to have an idea of what's going to be generated in the encoding step next. Beware of combinatorial explosion.

Attention: By default encoding videos is set to print the output, unless you pass the optional parameter debug as False. Have in mind that encoding the videos takes some time. Beware of using too many options, options sets, combinations.

[56]:
```
# the default if you call mt.encode_videos(), print the output files
#mt.encode_videos(debug=True)

# REALLY encode now. GPU encoders do their job but it still takes time
#mt.encode_videos(debug=False, progress=True)
```

We now have the h264 variations for the options set above. We need to compute the video quality metrics for each of them, and store them into respective json files. Presently the automated test feeding in MediaTests() and VideoQualityTests() is not yet complete, but the full automated encoding is, as well as the actual test metrics. We'll proceed to filter some files by a criteria to compare the dataframes.

[57]:
```
# make a deep copy of the data since we'll be filtering data in-place
import copy as cp
mt2 = cp.deepcopy(mt)
```

Filtering by filename first, we'll narrow to a file with random chaotic motion, then to the `libx264` codec, then to the `crf` metric.

[58]:
```
# Filter by input file
mt.outputdata = mt.by_file("light_orbitals.mkv")
```

[59]:
```
# Filter by codec
mt.outputdata = mt.by_codec("libx264")
```

[60]:
```
# Filter by encoding parameter in the parameter sets, in this case, CRF
mt.outputdata = mt.by_paramset("crf")
```

[61]:
```
mt.outputdata
```

[61]:
```
{'light_orbitals.mkv': {'libx264': {'crf': ['light_orbitals_-
_preset_veryfast__crf_18__motion-est_esa__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_esa__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_esa__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_esa__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_umh__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_umh__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_umh__pix_fmt_yuv444p.mkv']}}}
```

[62]:
```
# Filter by fully qualified output name, FQN
```

```
filteredfile = mt.
 ↪by_fqn_output("light_orbitals_-_preset_veryfast__crf_18__motion-est_esa__pix_fmt_yuv422p.
 ↪mkv")
filteredfile
```

[62]:
```
{'light_orbitals.mkv': {'libx264': {'crf': ['light_orbitals_-
_preset_veryfast__crf_18__motion-est_esa__pix_fmt_yuv422p.mkv']}}}
```

[63]:
```
mt.outputdata
```

[63]:
```
{'light_orbitals.mkv': {'libx264': {'crf': ['light_orbitals_-
_preset_veryfast__crf_18__motion-est_esa__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_esa__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_esa__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_esa__pix_fmt_yuv444p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_umh__pix_fmt_yuv420p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_umh__pix_fmt_yuv422p.mkv',
    'light_orbitals_-_preset_veryfast__crf_36__motion-
est_umh__pix_fmt_yuv444p.mkv']}}}
```

[64]:
```
testfiles = []

for k, v in mt.outputdata.items():
    for i, j in v.items():
        for x, y in j.items():
            testfiles.extend(
                [os.path.join(mt.media_container["outputdir"], x) for x in y]
            )
```

[65]:
```
srcfiles = []
for k, v in mt.outputdata.items():
    srcfiles.append(os.path.join(mt.media_container["inputdir"], k))
```

[66]:
```
srcfiles
```

[66]:
```
['/home/cgwork/Downloads/Masters/Tecnologias_Comunicacao_Multimedia_TCM/Work/videos/original/light_orbitals.mkv']
```

[67]:
```
from ffmpeg_quality_metrics import FfmpegQualityMetrics as ffqm
import pandas as pd
```

[68]:
```
# prep storage and metrics
# testio = {srcfiles[0] : testfiles}
testresults = []
tests = ["vmaf", "ssim", "psnr"]
```

[69]:
```
# only one source file, so no need to iterate over the entire footage list
inputfile = srcfiles[0]

# Finally run the tests, store into the array, and save as JSON files to
# avoid having to run expensive tests again
# True to run the tests, False to load the JSON files
run_slow_ffqm_tests = False

if run_slow_ffqm_tests:
    for compressedfile in testfiles:
        metrics = ffqm(inputfile, compressedfile, progress = False)
        metrics_data = metrics.calc(tests)
        fname, ext = os.path.splitext(compressedfile)
        fname = fname + ".json"
        mt.videoqtests.save_json(metrics_data, fname)
        testresults.append(metrics_data)
```

1.11.3 Building a DataFrame to Plot

Now we have an array with the test results for the footage and we can start seeing the impact of the options choosen in the perceptual and objective measures. Load the JSON for illustration purposes, build a tuple of 2 or 3 elements, element 0 for the input file if needed, 1 for test file, 2 for the test results dictionary with the metrics used, in this test, **SSIM**, **VMAF**, **PSNR**

[70]:
```
# Load JSON from results
testresults = []

for compressedfile in testfiles:
    fname, ext = os.path.splitext(compressedfile)
    fname = fname + ".json"
    testresults.append(mt.videoqtests.load_json(fname))
```

[71]:
```
# f = [os.path.basename(x) for x in testfiles]
framerate,_ = mt.media_container["inputdata"][srcfiles[0]]["r_frame_rate"].
 ↪split("/")
framerate=int(framerate)
dfs = {}
```

Build a nested dictionary to convert to a Pandas multi-index dataframe for easy data operations and plotting.

[72]:
```
dfs = {}

for i, c in enumerate(testfiles):
    df = pd.DataFrame.from_dict(testresults[i]["vmaf"])
    if "n" in df.columns:
        df.rename(columns={"n" : "frame"}, inplace=True)
        df.set_index("frame", inplace=True)
    dfs[os.path.basename(c)] = df.copy(deep=True)

reformed_dict = {}
for outerk, innerd in dfs.items():
    for innerk, values in innerd.items():
        reformed_dict[(outerk, innerk)] = values

mdf = pd.DataFrame(reformed_dict)
```

[73]:
```
# file to access for test
nfile =␣
 ↪"light_orbitals_-_preset_veryfast__crf_18__motion-est_esa__pix_fmt_yuv420p.
 ↪mkv"
```

[74]:
```
mdf2 = mdf.copy(deep = True)
```

[75]:
```
mdf2[nfile]
```

[75]:

frame	psnr	integer_motion2	integer_motion	integer_adm2	\
1	22.173126	0.000000	0.000000	0.822708	
2	22.176135	2.143594	2.143594	0.804705	
3	22.186353	2.174164	2.229993	0.809113	
4	22.169560	2.141280	2.174164	0.803830	
5	22.173912	0.000000	2.141280	0.816986	
…	…	…	…	…	
253	22.596779	2.130053	2.130053	0.817084	
254	22.594971	2.106853	2.349652	0.816119	
255	22.599585	2.068774	2.106853	0.817749	
256	22.601394	2.068774	2.068774	0.817251	
257	22.585603	2.320569	2.320569	0.814164	

frame	integer_adm_scale0	integer_adm_scale1	integer_adm_scale2	\
1	0.889201	0.827274	0.790936	
2	0.877725	0.810931	0.770919	
3	0.881307	0.816285	0.772766	
4	0.877931	0.811109	0.769427	
5	0.888836	0.825354	0.782443	
…	…	…	…	
253	0.885460	0.819984	0.785397	
254	0.885890	0.822219	0.782842	
255	0.886496	0.824338	0.784062	
256	0.886347	0.823406	0.783542	
257	0.885670	0.821235	0.780613	

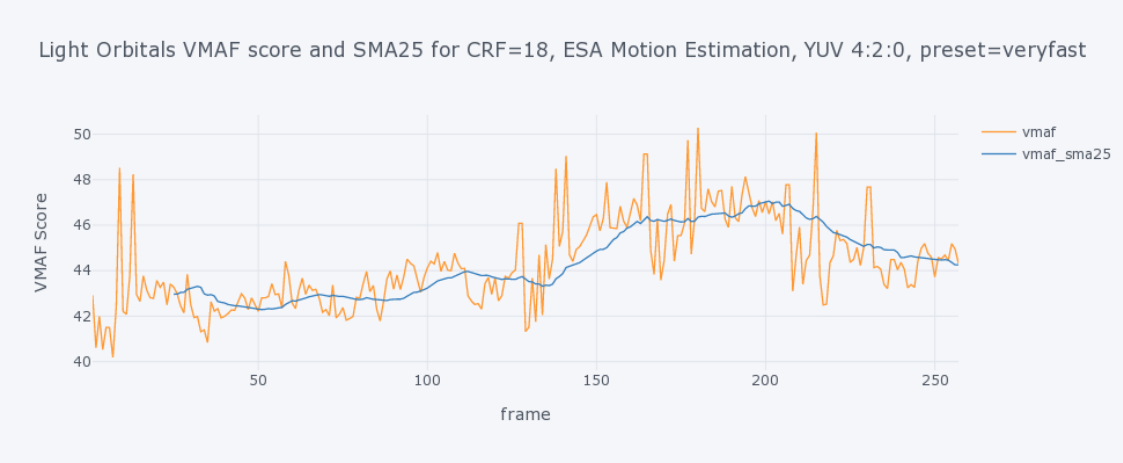
frame	integer_adm_scale3	ssim	integer_vif_scale0	integer_vif_scale1	\
1	0.776726	0.981752	0.659667	0.828004	
2	0.751866	0.978929	0.616752	0.809357	
3	0.756687	0.979879	0.630125	0.813490	
4	0.750504	0.979165	0.620198	0.812802	
5	0.768642	0.981941	0.688541	0.831151	
…	…	…	…	…	
253	0.771735	0.982546	0.654402	0.836419	
254	0.768972	0.982392	0.655843	0.834218	
255	0.770092	0.982834	0.655904	0.836654	
256	0.769823	0.982660	0.653512	0.835708	
257	0.765988	0.982251	0.650706	0.833324	

frame	integer_vif_scale2	integer_vif_scale3	ms_ssim	vmaf
1	0.841630	0.850765	0.987508	42.908559
2	0.825326	0.837393	0.984524	40.608449
3	0.830120	0.843222	0.985550	41.995908

```
4                0.828150             0.837903  0.984864  40.530280
5                0.841455             0.848130  0.987614  41.508192
…                   …                    …         …         …
253              0.848781             0.853865  0.987648  44.687808
254              0.847135             0.854174  0.987610  44.444005
255              0.849826             0.859685  0.987958  45.186619
256              0.848971             0.857975  0.987794  44.948962
257              0.846452             0.854535  0.987363  44.321558

[257 rows x 15 columns]
```

Example file, `light_orbitals.mkv`, with random chaotic movement, light saturated cycling colors. With 4:2:0 chroma sampling we should expect to see degradation since we only have 2 chroma pixels per 4 luma pixels for every other odd or even line.

[76]:
```
# Compute a 25 frames (1s) simple moving average
dfs[nfile]["vmaf_sma25"] = dfs[nfile]["vmaf"].rolling(25).mean()

fig = dfs[nfile].iplot(y=["vmaf", "vmaf_sma25"],
                       title = dict(text = "Light Orbitals VMAF score and SMA25␣
↪for CRF=18, ESA Motion Estimation, YUV 4:2:0, preset=veryfast",
                                    font = dict(size = 8)),
                  legend="foo", xaxis_title="frame", yaxis_title="VMAF Score",␣
↪asFigure=True)

fig.update_layout(width=900, height=400, title_x = 0.5)
```



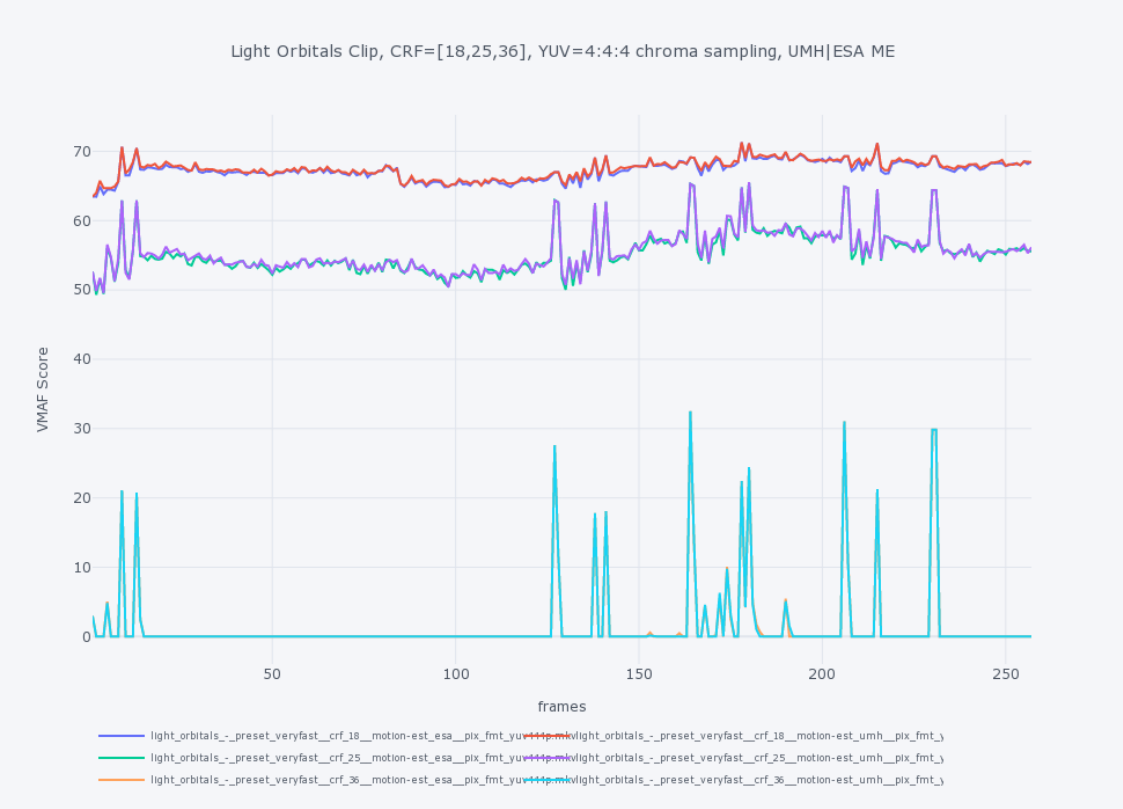
[77]:
```
# we need figure objects
import plotly.graph_objects as go
```

1.11.4 Filtering For Plots

It would be advantageous to filter by chroma sampling since in these cases the difference is dramatic. We'll see below all the files encoded with constant rate factors, that is, the quantization matrix was allowed to vary as long as the quality factor remained the same. Clicking in one of the trace legends twice rapidly will hide all traces, then one can enable them one by one. This will allow one to for instance, group the traces by chroma sampling, in order to have a better appreciation of the impact of the motion estimation.

[78]:
```
# from plotly get a figure object, for the preset layout
fig = dfs[nfile].iplot(y=["vmaf"], asFigure=True)
data, flayout = fig["data"], fig["layout"]
```

[79]:
```
fig = go.Figure()
fig["layout"] = flayout

# Plot the files with CRF=18 only
for k, v in dfs.items():
    #fig.add_trace(go.Scatter(x=["frame"], y=["psnr"]))
    if "yuv444p" in k:
        fig.add_trace(go.Scatter(x=dfs[k].index.values, y=dfs[k]["vmaf"],␣
↪name=str(k)))
    #print(k)

fig.update_xaxes(title=dict(font=dict(size=12), text="frames"))
fig.update_yaxes(title=dict(font=dict(size=12), text="VMAF Score"))
fig.update_layout(title=dict(font=dict(size=14),
                             text="Light Orbitals Clip, CRF=[18,25,36], YUV=4:4:
↪4 chroma sampling, UMH|ESA ME"),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=40,
                              tracegroupgap=5,
                              font=dict(size=8),),
                  width=900, height=700,
                  title_x = 0.5,
                 )
```



From the initial glance we can see that the files encoded with a CRF value of 36 have very bad VMAF scores. Inspecting the file shows **unnaceptable** degradation in image quality. Visible macroblocks, excessive quantization.

If you hide the CRF=36 plot traces shows more nuanced results (click on the trace legend color in the plot to toggle its visibility).

With a CRF of 25, the VMAF scores fluctuates as we saw earlier in the first plots. Higher complexity and lower quality in the first half, then spikes in the VMAF score and a smoother climb and progression in the 2nd half.

The **esa** (exhaustive search adaptive) motion estimation method is a *smarter* brute-force method searching the entire motion space within the motion estimation window. It still underperformed consistently, albeit not by much, the **umh** multi-hexagen pattern method motion estimation method. This method was created for harder to find motion vectors and it seems to have benefits.

Going to the CRF=18 trace plots, the difference between the motion estimation methods is stronger, and **umh** seems to be advantageous.

From this we can immediatly discard anything higher than CRF=25 in this clip. The arithmetic means of VMAF in the YUV 4:4:4, 4:2:0 clips are listed next, however, there is a catch.

[80]:
```
for k, v in dfs.items():
    if "crf_36" in k:
        continue
    print(f"Clip: {k} VMAF mean = {v['vmaf'].mean()}")
```

```
Clip: light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv420p.mkv VMAF mean = 44.30000864202334
Clip: light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv422p.mkv VMAF mean = 44.23868481322957
Clip: light_orbitals_-_preset_veryfast__crf_18__motion-
est_esa__pix_fmt_yuv444p.mkv VMAF mean = 67.27321896887159
Clip: light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv420p.mkv VMAF mean = 44.759980926070035
Clip: light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv422p.mkv VMAF mean = 44.689372976653694
Clip: light_orbitals_-_preset_veryfast__crf_18__motion-
est_umh__pix_fmt_yuv444p.mkv VMAF mean = 67.49915161867705
Clip: light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv420p.mkv VMAF mean = 36.2753383385214
Clip: light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv422p.mkv VMAF mean = 36.01070726070039
Clip: light_orbitals_-_preset_veryfast__crf_25__motion-
est_esa__pix_fmt_yuv444p.mkv VMAF mean = 55.3118496536965
Clip: light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv420p.mkv VMAF mean = 36.47667958754864
Clip: light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv422p.mkv VMAF mean = 36.24430987159533
Clip: light_orbitals_-_preset_veryfast__crf_25__motion-
est_umh__pix_fmt_yuv444p.mkv VMAF mean = 55.514778715953305
```

1.11.5 Reading the VMAF Score

As mentioned earlier and referred to in the NetFlix Technical Blog

> VMAF scores range from 0 to 100, with 0 indicating the lowest quality, and 100 the highest. A good way to think about a VMAF score is to linearly map it to the human opinion scale under which condition the training scores are obtained. As an example, the default model v0.6.1 is trained using scores collected by the Absolute Category Rating (ACR) methodology using a 1080p display with viewing distance of 3H. Viewers voted the video quality on the scale of "bad", "poor", "fair", "good" and "excellent", and roughly speaking, "bad" is mapped to the VMAF scale 20 and "excellent" to 100. Thus, a VMAF score of 70 can be interpreted as a vote between "good" and "fair" by an average viewer under the 1080p and 3H condition. Another factor to consider is that the best and the worst votes a viewer can give is calibrated by the highest- and lowest-quality videos in the entire set (during training, and before the actual test starts, subjects are typically accustomed to the experiment's scale). In the case of the default model v0.6.1, the best and the worst videos are the ones compressed at 1080p with a low quantization parameter (QP) and the ones at 240p with a high QP, respectively.

According to this criteria, anything but the YUV 4:4:4 sampled clip with a CRF of 18 is discarded. However according to Twitter Engineering one shouldn't give too much credence to the mean since it might hide difficult to encode frames, choosing percentiles instead.

> aggregating VMAF scores of frames by averaging over the entire sequence may hide the impact of difficult-to-encode frames (if these frames occur infrequently). An optimal way to pool frames is an open problem
>
> By definition, the 5th percentile gives us a VMAF score of the worst 5% frames while the 50th percentile is the median. The intuition here is that instead of weighing all frames equally and getting one score, we rank frames according to their complexity and look at how a particular encoder setting performs across these different ranks. We want to prioritize improving quality on frames in the order of their VMAF scores, from lowest to highest. Frames with a high VMAF score already look great and improving quality on them won't matter as much

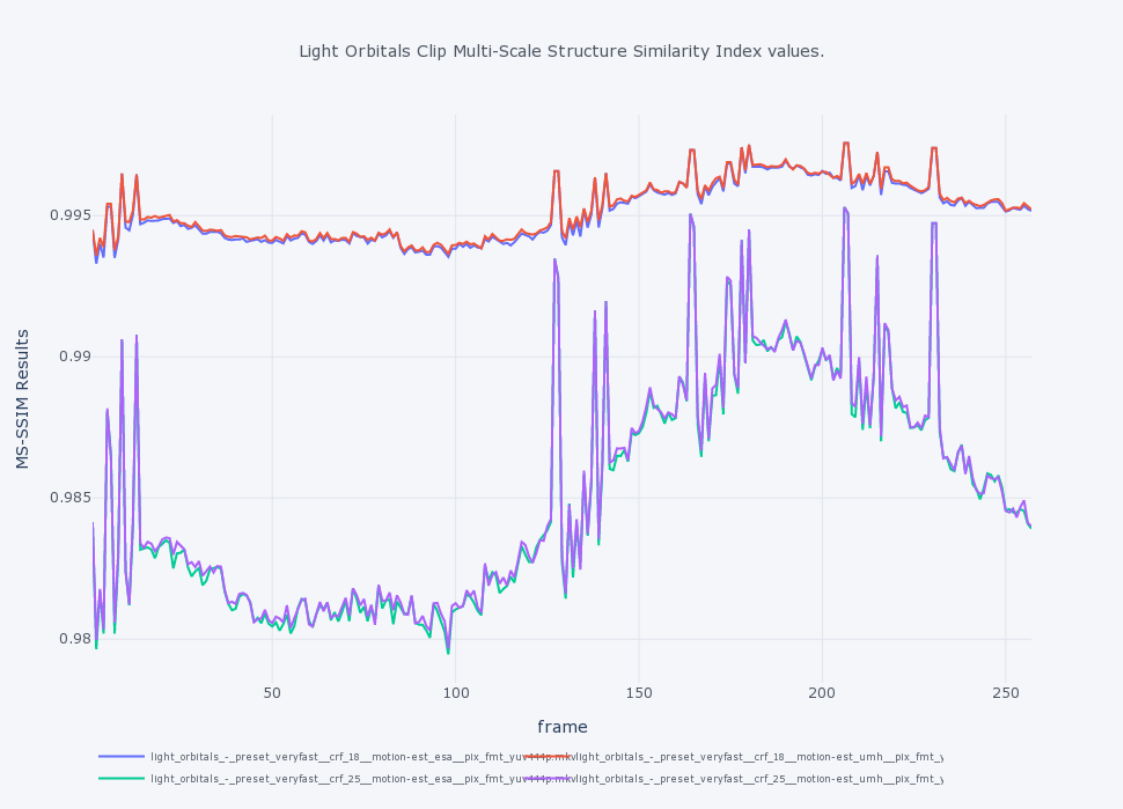
Let's inspect the percentiles then, as well as standard deviation besides the arithmetic mean.

[81]:
```
#for k, v in dfs.items():
#    if ("crf_18" in k or "crf_25" in k) and "yuv444p" in k:
#        #print(f"VMAF overall statistics: {dfs[k]['vmaf'].describe()}")
#        dfs[k]["vmaf"].describe().iplot(kind="bar")

fig = go.Figure()
fig["layout"] = flayout

for k, v in dfs.items():
    if ("crf_18" in k or "crf_25" in k) and "yuv444p" in k:
        fig.add_trace(
            go.Bar(x=["count", "mean", "stddev", "25% pct", "50% pct", "75%␣
 ↪pct", "max"], y=dfs[k]["vmaf"].describe(),
                   name=str(k),
))

fig.update_xaxes(type="category", title="VMAF statistics")
fig.update_yaxes(title="VMAF statistics values")
fig.update_layout(title=dict(font=dict(size=14),
                             text="Clip Aggregate VMAF Statistics for YUV 4:4:4␣
 ↪sampling, ESA|UMH ME methods"),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=40,
                              tracegroupgap=5,
                              font=dict(size=8),),
                  width=900, height=700,
                  title_x = 0.5,
                 )
```

The multi-scale structural similarity index shows consistent results. A value of 1.0 is only reached in two identical data sets, and as such, indicates perfect structural similarity.

[82]:
```
fig = go.Figure()
fig["layout"] = flayout

for k, v in dfs.items():
    if ("crf_18" in k or "crf_25" in k) and "yuv444p" in k:
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["ms_ssim"], name=str(k)))

fig.update_yaxes(title="MS-SSIM Results")
fig.update_xaxes(title="frame")
fig.update_layout(title=dict(font=dict(size=14),
                             text="Light Orbitals Clip Multi-Scale Structure␣
 ↪Similarity Index values."),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=40,
                              tracegroupgap=5,
                              font=dict(size=8),),
                  width=900, height=700,
```

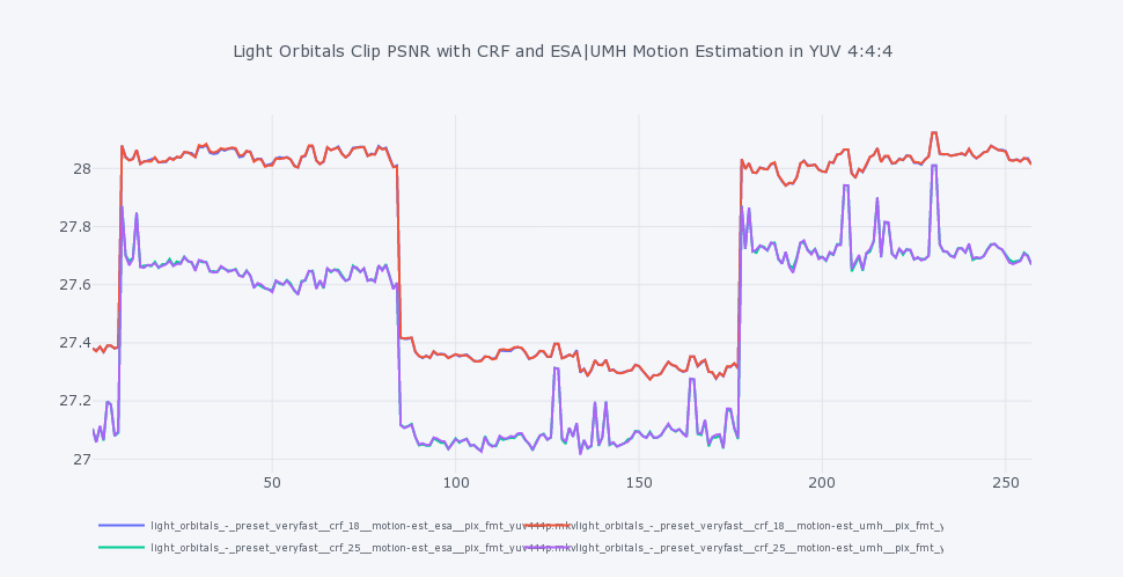
```
                    title_x = 0.5,
                   )
```



The PSNR values are show next.

[83]:
```
fig = go.Figure()
fig["layout"] = flayout

for k, v in dfs.items():
    if ("crf_18" in k or "crf_25" in k) and "yuv444p" in k:
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["psnr"], name=str(k)))

fig.update_layout(title=dict(font=dict(size=14),
                             text="Light Orbitals Clip PSNR with CRF and␣
 ↪ESA|UMH Motion Estimation in YUV 4:4:4"),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=40,
                              tracegroupgap=5,
                              font=dict(size=8),),
                  width=900, height=500,
```

```
                title_x = 0.5,
               )
```



Typical values are between 30dB and 50dB for 8bit sources, which is the case. Since PSNR is $PSNR = 10 \cdot \log_{10} \frac{MAX_i^2}{MSE}$, with MAX_i the maximum pixel code value of the image, two identical images would result in a error of 0, a denominator of 0, and $\lim_{mse \to 0} \left[10 \cdot \log_{10} \frac{MAX_i^2}{MSE}\right] = \infty$ It shows values below 30dB, but unlike PSNR-HVS-M it doesn't take into account the characteristics of the HVS.

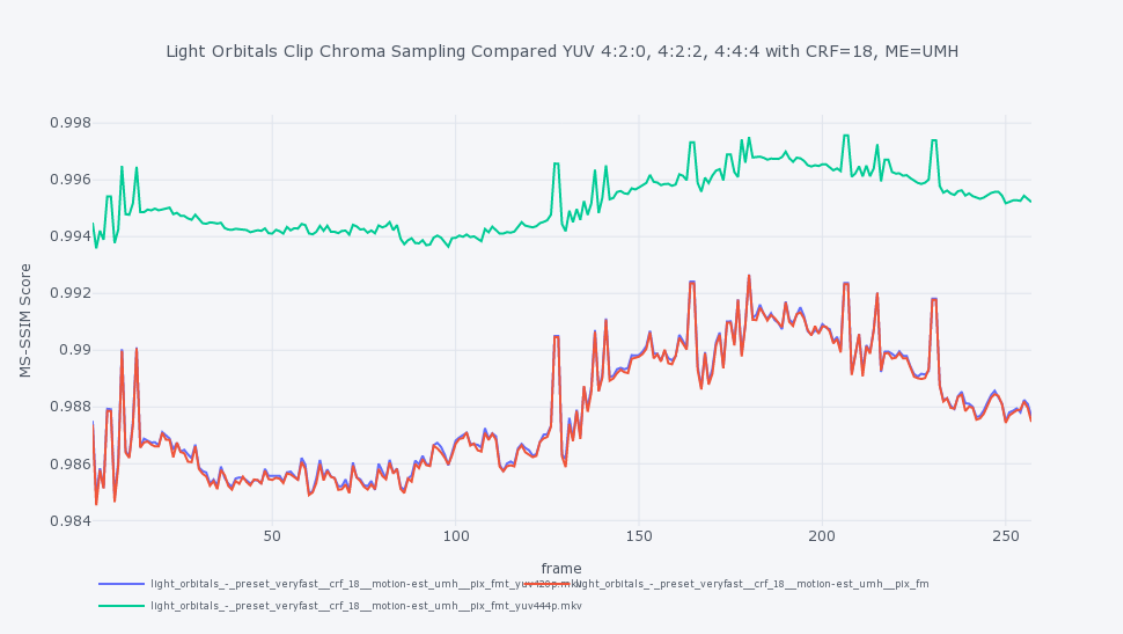
[84]:
```
dfs[nfile].columns
```

[84]:
```
Index(['psnr', 'integer_motion2', 'integer_motion', 'integer_adm2',
       'integer_adm_scale0', 'integer_adm_scale1', 'integer_adm_scale2',
       'integer_adm_scale3', 'ssim', 'integer_vif_scale0',
       'integer_vif_scale1', 'integer_vif_scale2', 'integer_vif_scale3',
       'ms_ssim', 'vmaf', 'vmaf_sma25'],
      dtype='object')
```

Since UMH motion estimation and as expected the lowest constant rate factor gave us the best results, how can we compare the chroma sampling in perceptual terms?

[85]:
```
fig = go.Figure()
fig["layout"] = flayout

for k, v in dfs.items():
    if "crf_18" in k and "umh" in k:
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["ms_ssim"], name=str(k)))
```

```
fig.update_layout(title=dict(font=dict(size=14),
                             text="Light Orbitals Clip Chroma Sampling Compared␣
 ↪YUV 4:2:0, 4:2:2, 4:4:4 with CRF=18, ME=UMH"),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=40,
                              tracegroupgap=5,
                              font=dict(size=8),),
                  xaxis=dict(title=dict(font=dict(size=12), text="frame")),
                  yaxis=dict(title=dict(font=dict(size=12), text="MS-SSIM␣
 ↪Score")),
                  width=900, height=550,
                  title_x = 0.5,
                 )
```



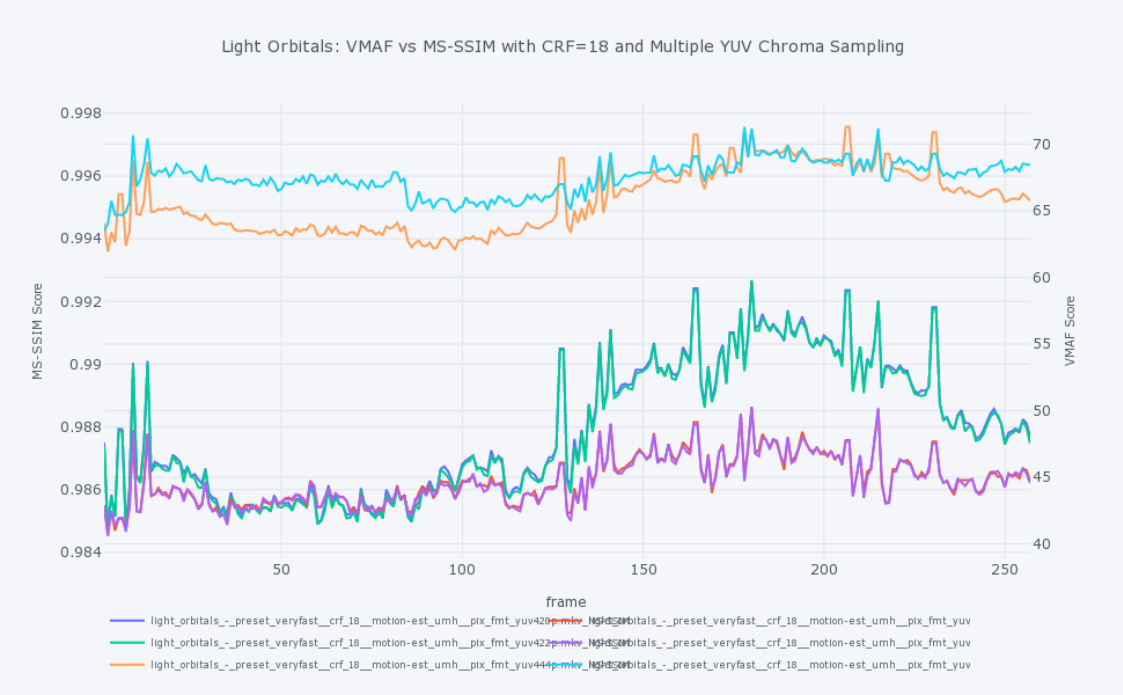
The MS-SSIM difference between the YUV 4:4:4 and 4:2:2, 4:2:0 chroma sampling seems more subtle than the difference between the VMAF scores. Considering the fact that the source material has highly saturated colors and that 4:2:0 chroma sampling for example, will inevitably show degradation of these characteristics, it's good to keep within the metrics tools the VMAF metric, besides SSIM, PSNR.

[86]:
```
from plotly.subplots import make_subplots
# custom layout for subplots
custom_subplot_layout = {
    'legend': {'bgcolor': '#F5F6F9', 'font': {'color': '#4D5663'}},
```

```
    'paper_bgcolor': '#F5F6F9',
    'plot_bgcolor': '#F5F6F9',
    'title': {'font': {'color': '#4D5663'}},
    'xaxis': {'gridcolor': '#E1E5ED',
              'showgrid': True,
              'tickfont': {'color': '#4D5663'},
              'title': {'font': {'color': '#4D5663'}, },
              'zerolinecolor': '#E1E5ED'},
    'yaxis': {'gridcolor': '#E1E5ED',
              'showgrid': True,
              'tickfont': {'color': '#4D5663'},
              'title': {'font': {'color': '#4D5663'},},
              'zerolinecolor': '#E1E5ED'},
    'yaxis2': {'gridcolor': '#E1E5ED',
              'showgrid': True,
              'tickfont': {'color': '#4D5663'},
              'title': {'font': {'color': '#4D5663'},},
              'zerolinecolor': '#E1E5ED'}
}
```

[87]:
```
fig = go.Figure()
# make a secondary Y plot to keep 2 metrics in the plot with different Y axis␣
 ↪scales on the left and right
fig = make_subplots(specs=[[{"secondary_y" : True}]])
fig.update_layout(**custom_subplot_layout)

for k, v in dfs.items():
    # narrow down to CRF=18, hexagonal pattern search motion estimation
    if "crf_18" in k and "umh" in k:
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["ms_ssim"],␣
 ↪name=str(k)+"_MS-SSIM"),
            secondary_y = False)
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["vmaf"],␣
 ↪name=str(k)+"_VMAF"),
            secondary_y = True)


fig.update_layout(title=dict(font=dict(size=14),
                             text="Light Orbitals: VMAF vs MS-SSIM with CRF=18␣
 ↪and Multiple YUV Chroma Sampling"),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=30,
                              tracegroupgap=5,
                              font=dict(size=8),),
```

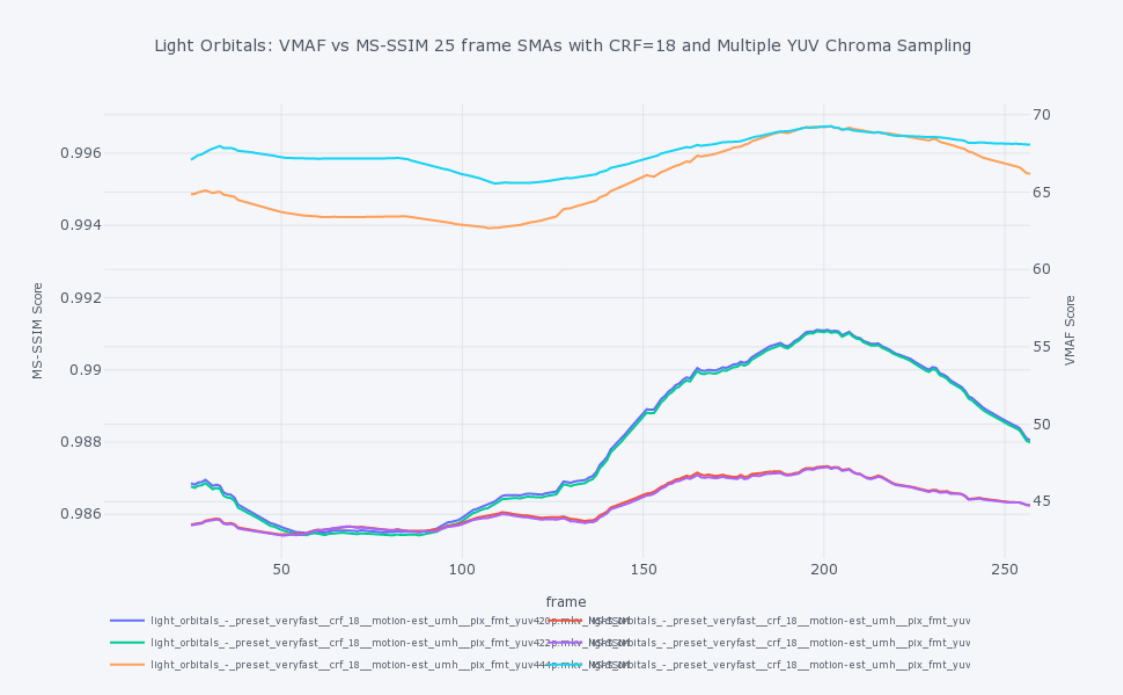
```
                  width=900, height=600,
                  yaxis1=dict(title=dict(font=dict(size=10), text="MS-SSIM␣
 ↪Score")),
                  yaxis2=dict(title=dict(font=dict(size=10), text="VMAF␣
 ↪Score"), anchor="x", side="right", position=0.15),
                  xaxis=dict(title=dict(font=dict(size=12), text="frame")),
                  title_x = 0.5,
                  margin=dict(l=90, r=30, b=20, t=90, pad=1),
                 )
#fig.update_xaxes(zeroline=False, showgrid=False)
#fig.update_yaxes(zeroline=False, showgrid=False)
fig.show()
```



Choosing one example, the YUV 4:2:0 sampling and the MS-SSIM, VMAF traces, we can see the plots overlap more or less, with some minor fluctuations, but overall the VMAF score is lower, in the 40 to 50 range, meaning a noticeable loss of quality according to the viewing conditions outlined earlier. The MS-SSIM score seems to oscillate between 0.985 and 0.992, so there is some noticeable difference even here.

[88]:
```
fig = go.Figure()
# make a secondary Y plot to keep 2 metrics in the plot with different Y axis␣
 ↪scales on the left and right
fig = make_subplots(specs=[[{"secondary_y" : True}]])
fig.update_layout(**custom_subplot_layout)
```

```
for k, v in dfs.items():
    # narrow down to CRF=18, hexagonal pattern search motion estimation
    if "crf_18" in k and "umh" in k:
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["ms_ssim"].
 ↪rolling(framerate).mean(), name=str(k)+"_MS-SSIM"),
            secondary_y = False)
        fig.add_trace(
            go.Scatter(x=dfs[k].index.values, y=dfs[k]["vmaf"].
 ↪rolling(framerate).mean(), name=str(k)+"_VMAF"),
            secondary_y = True)


fig.update_layout(title=dict(font=dict(size=14),
                             text="Light Orbitals: VMAF vs MS-SSIM 25 frame␣
 ↪SMAs with CRF=18 and Multiple YUV Chroma Sampling"),
                  legend=dict(orientation="h",
                              itemsizing="trace",
                              itemwidth=30,
                              tracegroupgap=5,
                              font=dict(size=8),),
                  width=900, height=600,
                  yaxis1=dict(title=dict(font=dict(size=10), text="MS-SSIM␣
 ↪Score")),
                  yaxis2=dict(title=dict(font=dict(size=10), text="VMAF␣
 ↪Score"), anchor="x", side="right", position=0.15),
                  xaxis=dict(title=dict(font=dict(size=12), text="frame")),
                  title_x = 0.5,
                  margin=dict(l=90, r=30, b=20, t=90, pad=1),
                 )
#fig.update_xaxes(zeroline=False, showgrid=False)
#fig.update_yaxes(zeroline=False, showgrid=False)
fig.show()
```



Filtering the results with a one second (25 frames in this case) simple moving average shows the same fundamental scenario and the difference in scores between the 4:2:2 and 4:2:0 sampling is not very significant. Again, we can see that in the 4:4:4 case, the results are significantly higher and sustained across the frames.

For this test clip at least, but expected generally, keeping the chroma sampling at 4:4:4 seems sensible.

[ ]: