

International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

J.343.1

(11/2014)

SERIES J: CABLE NETWORKS AND TRANSMISSION
OF TELEVISION, SOUND PROGRAMME AND OTHER
MULTIMEDIA SIGNALS

Measurement of the quality of service - Part 3

**Hybrid-NRe objective perceptual video quality
measurement for HDTV and multimedia
IP-based video services in the presence of
encrypted bitstream data**

Recommendation ITU-T J.343.1

ITU-T



Recommendation ITU-T J.343.1

Hybrid-NRe objective perceptual video quality measurement for HDTV and multimedia IP-based video services in the presence of encrypted bitstream data

Summary

Recommendation ITU-T J.343.1 provides hybrid no-reference encrypted (Hybrid-NRe) objective perceptual video quality measurement methods for HDTV and multimedia when encrypted bitstream data are available. The following are example applications that can use this Recommendation:

- potentially real-time, in-service quality monitoring at the headend;
- video television streams over cable/IPTV networks including those transmitted over the Internet using Internet protocol;
- video quality monitoring at the receiver when encrypted bitstream data are available;
- video quality monitoring at measurement nodes located between point of transmission and point of reception when encrypted bitstream data are available;
- quality measurement for monitoring of a transmission system that utilizes video compression and decompression techniques, either a single pass or a concatenation of such techniques;
- lab testing of video transmission systems.

This Recommendation includes an electronic attachment containing test vectors, including video sequences, bitstream files and predicted objective model scores.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T J.343.1	2014-11-29	9	11.1002/1000/12316

* To access the Recommendation, type the URL <http://handle.itu.int/> in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2015

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Table of Contents

	Page
1	Scope..... 1
1.1	Applications..... 2
1.2	Limitations..... 2
2	References..... 3
3	Definitions 3
3.1	Terms defined elsewhere 3
3.2	Terms defined in this Recommendation..... 3
4	Abbreviations and acronyms 3
5	Conventions 4
6	Performance metrics 4
7	Description of the hybrid no-reference methodology..... 4
8	Models 5
Annex A – Hybrid-NRe model RST-V-model 6	
A.1	Packet header data extraction 6
A.2	Extraction of video frame feature statistics 12
A.3	Hybrid core model..... 24
A.4	Additional information 46
Annex B – YHyNRe (Hybrid-NRe model)..... 51	
B.1	Introduction 51
B.2	Hybrid-NRe VQM computation..... 51
Bibliography..... 58	

Electronic attachment: Test vectors, including video sequences, bitstream files and predicted objective model scores.

Recommendation ITU-T J.343.1

Hybrid-NRe objective perceptual video quality measurement for HDTV and multimedia IP-based video services in the presence of encrypted bitstream data

1 Scope

This Recommendation¹ describes algorithmic models for measuring the visual quality of IP-based video services.

The models are hybrid no-reference encrypted (Hybrid-NRe) models, which operate by analysing packet header information and video image data captured at the video player. The models operate without parsing or decoding the packet payload. Thus, these models can be used with encrypted bitstream data as well as non-encrypted bitstream data.

As output, the models provide an estimate of visual quality on the [1,5] mean opinion score (MOS) scale, derived from five-point absolute category rating (ACR) as in [ITU-T P.910]. The models address low-resolution (VGA/WVGA) application areas, including services such as mobile TV, as well as high-resolution (HD) application areas, including services such as IPTV.

This Recommendation is to be used with videos encoded using ITU-T H.264 and media payload encapsulated in RTP/UDP/IP packets for the low resolution and encapsulated in MPEG-TS/RTP/UDP/IP for the high resolution.

The models in this Recommendation measure the visual effect of spatial and temporal degradations as a result of video coding, erroneous transmission or video rescaling. The models may be used for applications such as to monitor the quality of deployed networks to ensure their operational readiness or to benchmark service quality. The models in this Recommendation can also be used for lab testing of video transmission systems.

The models identified in this Recommendation have limited precision. Therefore, directly comparing model results can be misleading. The accuracy of models has to be understood and taken into account (e.g., using [ITU-T J.149]).

The validation test material consisted of video encoded using different implementations of [ITU-T H.264]. It included media transmitted over wired and wireless networks, such as WIFI and 3G mobile networks. The transmission impairments included error conditions such as dropped packets, packet delay, both from simulations and from transmission over commercially operated networks.

The following source reference channel (SRC) conditions were included in the validation test:

- 1080i 60 Hz (29.97 fps);
- 1080p (25 fps);
- 1080i 50 Hz (25 fps);
- 1080p (29.97 fps);
- SRC duration: HD: 10 s, VGA/WVGA: 10 s or 15 s (rebuffering);
- VGA at 25 and 30 fps;
- WVGA at 25 and 30 fps.

¹ This Recommendation includes an electronic attachment containing test vectors, including video sequences, bitstream files and predicted objective model scores.

The following hypothetical reference circuit (HRC) conditions were included in the validation test for each resolution:

Test factors
Video resolution: 1920 × 1080 interlaced and progressive
Video frame rates 29.97 and 25 fps
Video bitrates: 1 to 30 Mbit/s (HD), 100 kbit/s to 3 Mbit/s (VGA/WVGA)
Temporal frame freezing (pausing with skipping) of up to 50% of video duration
Transmission errors with packet loss
Rebuffering (VGQ/WVGA only): up to 50% of SRC
Coding technologies
ITU-T H.264/AVC (MPEG-4 Part 10)
Tandem coding

1.1 Applications

The applications for the estimation model described in this Recommendation include, but are not limited to:

- potentially real-time, in-service quality monitoring at the headend;
- video television streams over cable/IPTV networks including those transmitted over the Internet using Internet protocol;
- video quality monitoring at the receiver when encrypted bitstream data and processed video sequence (PVS) are available;
- video quality monitoring at measurement nodes located between point of transmission and point of reception when encrypted bitstream data and PVS are available;
- quality measurement for monitoring of a transmission system that utilizes video compression and decompression techniques, either a single pass or a concatenation of such techniques;
- lab testing of video transmission systems.

1.2 Limitations

The video quality estimation models described in this Recommendation cannot be used to fully replace subjective testing.

When frame freezing was present, the test conditions had frame-freezing durations up to 50% of SRC duration. The models in this Recommendation were validated for measuring video quality in a rebuffering condition (i.e., video that has a steadily increasing delay or freezing without skipping) only for VGA/WVGA. The models were not tested on other frame rates than those used in TV systems (i.e., 29.97 fps and 25 fps, in interlaced or progressive mode).

If forward error correction techniques are employed, the models in this Recommendation may not be used.

It is important that no additional transmission errors occur between the collection point of the bitstream data and the capture point of the PVS.

It should be noted that in case of new coding and transmission technologies producing artifacts, which were not included in this evaluation, the objective model may produce erroneous results. Here, a subjective evaluation is required.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T H.264] Recommendation ITU-T H.264 (2014), *Advanced video coding for generic audiovisual services*.

[ITU-T J.149] Recommendation ITU-T J.149 (2004), *Method for specifying accuracy and cross-calibration of Video Quality Metrics (VQM)*.

[ITU-T J.343] Recommendation ITU-T J.343 (2014), *Hybrid perceptual bitstream models for objective video quality measurements*.

[ITU-T P.910] Recommendation ITU-T P.910 (2008), *Subjective video quality assessment methods for multimedia applications*.

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

3.1.1 hybrid no reference model [ITU-T J.343]: An objective video quality model that predicts subjective quality using the decoded video frames, packet headers, and video payload. Such models can be deployed in-service but cannot analyse encrypted video.

3.1.2 hybrid no reference encrypted model [ITU-T J.343]: An objective video quality model that predicts subjective quality using the decoded video frames and packet headers. Such models can be deployed in-service and are suitable for use with encrypted video.

3.2 Terms defined in this Recommendation

None.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

CODEC	COder-DECoder
HRC	Hypothetical Reference Circuit
Hybrid-NR	Hybrid No Reference
Hybrid-NRe	Hybrid No Reference encrypted
LUT	Look-Up Table
MOS	Mean Opinion Score
MPEG	Moving Picture Experts Group
NR	No (or Zero) Reference
PES	Packetized Elementary bitStream
PVS	Processed Video Sequence

SRC	Source Reference Channel or Circuit
VQEG	Video Quality Experts Group
VQM	Video Quality Metrics

5 Conventions

None.

6 Performance metrics

A summary of this and other hybrid models may be found in [ITU-T J.343]. See [b-VQEG Hybrid] for a complete analysis of the models included in this Recommendation.

Note that the RST-V-model is referred to as "TVM-Hybrid Encrypted" within [b-VQEG Hybrid].

7 Description of the hybrid no-reference methodology

This Recommendation specifies objective video quality measurement methods which use both processed video sequences and bitstream data. The bitstream data may be provided in the forms of elementary bitstream (ES), packetized elementary bitstream (PES) or packet video (Figure 1).

The Hybrid-NRe models use only PVS and bitstream data, as shown in Figure 1 and Figure 2. While the hybrid no reference (Hybrid-NR) models have access to all of this data, the Hybrid-NRe models do not have access to the video payload. Therefore, these models can be used with encrypted bitstreams.

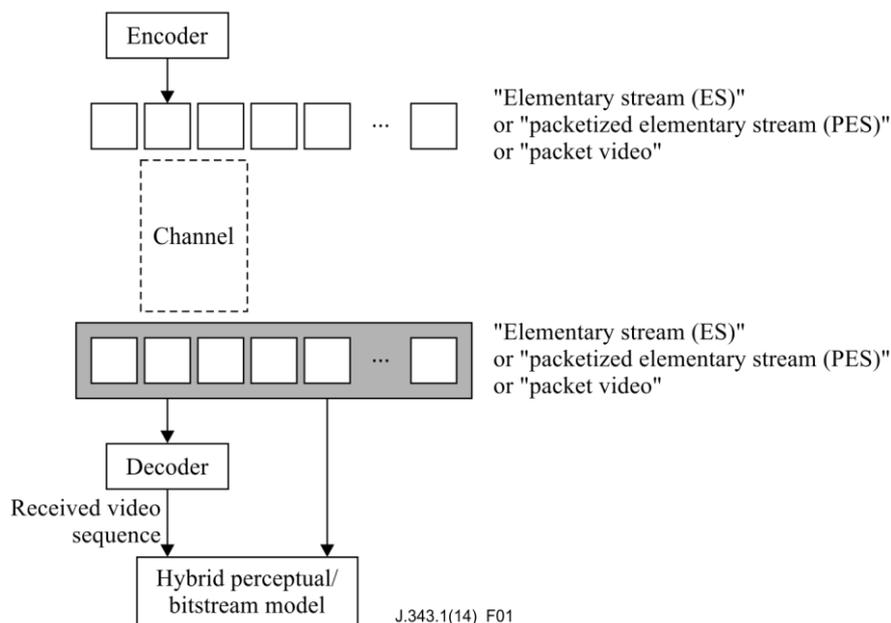


Figure 1 – Block-diagram depicts the core concept of hybrid perceptual bitstream models

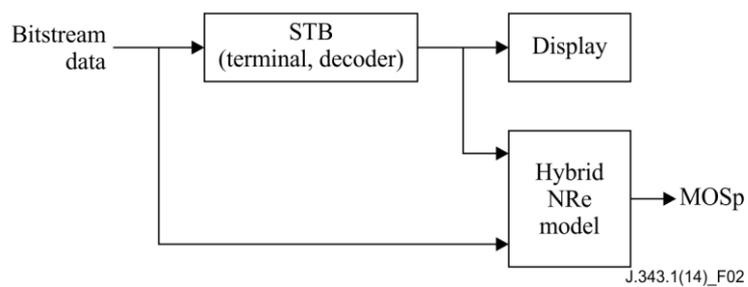


Figure 2 – Block-diagram of the Hybrid-NRe model

8 Models

Annexes A and B contain full disclosures of all models included in this Recommendation. These models are RST-V-model and YHyNRe.

Annex A

Hybrid-NRe model RST-V-model

(This annex forms an integral part of this Recommendation.)

Overview

The RST-V-model is composed of the following three modules:

- 1) packet header data extraction;
- 2) extraction of video frame feature statistics;
alignment of edited PVS to PVS;
- 3) hybrid core model.

Each of these modules is described in the following clauses, together with a clause describing auxiliary functions and containers at the end.

The beginning of each clause contains a high level overview of the module.

The model takes as input the filename of a .pcap file containing the bistream and a filename of an .avi file containing the captured video at the playback side. If the quality should be estimated for the captured video with a pre- and post-roll cut (as it was used for the Video Quality Experts Group (VQEG) evaluation), then additionally, the model takes as input the filename of an .avi file containing the captured video with cut pre- and post-roll.

The modules 1 and 2 are independent. Their output is the input to the hybrid core module 3. The output of the hybrid core module is given by:

`HybridModel.predicted_Quality,`

and denotes the estimated video quality in the range $[1, 5]$. Additional diagnostic information can be retrieved from the model, such as the individual coding and transmission quality estimates.

A.1 Packet header data extraction

Extract the packet header information from a pcap file.

```
MODULE INPUT:
pcap_filename          -- name of the bitstream (pcap) file
```

```
MODULE OUTPUT:
transmitted_frame_height  -- transmitted video frame height
df                        -- data frame (matrix) containing packet header data, one
                           row per video frame
```

Perform the following processing steps:

In case of `rtsp/rtp/udp`, derive the `transmitted_frame_height` from session description protocol (SDP), otherwise set `transmitted_frame_height = 0`.

Next perform

```
df_p = parse_bitstream(pcap_filename)
```

In case of using transport streams (mp2t) do

```
df = per_packet2per_frame_mp2t(df_p)
df = correct_large_frames(df, len_content)
```

else do

```
df = per_packet2per_frame_rtp(df_p)
```

The complete processing is described below in more detail:

```
def parse_bitstream(filename):
    """
    Parse a pcap file and extract information from the packet headers.

    INPUT:
    filename -- pcap file

    OUTPUT:
    df -- DataFrame, of the following form:

        For 'rtsp/rtp/udp' or 'rtp/udp':

            frame.number  frame.time_relative  rtp.seq  rtp.timestamp  rtp.marker  udp.length
    0          5.00          115.00  60794.00  2130617091.00      0.00      841.00
    1          6.00          115.00  60795.00  2130617091.00      0.00     1458.00
    2          7.00          115.00  60796.00  2130617091.00      0.00     1458.00
    3          8.00          115.00  60797.00  2130617091.00      0.00     1233.00

        For 'mp2t/rtp/udp' or 'mp2t/udp':

            frame.number  frame.time_relative  mp2t.pid  mp2t.pusi  mp2t.cc  mp2t.analysis.drops
    0          1.00          0.00      17.00      1.00      0.00          0.00
    1          2.00          0.00       0.00      1.00      0.00          0.00
    2          3.00          0.00    4095.00      1.00      0.00          0.00
    3          4.00          0.00    256.00      1.00      0.00          0.00
```

See <https://www.wireshark.org/docs/dfref/m/mp2t.html> for more information about the field names and their meaning.

Remove duplicate elements in list a, without changing order of the elements.

```
def remove_duplicate(a):
    """
    OUTPUT:
    (b,I) -- pair, b is the list with duplicates removed, I is the list of indices of
           the elements in a, which are in b.
    """
    b = []
    I = []

    for i,x in enumerate(a):
        if x not in b:
            b.append(x)
            I.append(i)

    return (array(b),array(I))
```

```
def supported_protocols():
    return ['rtsp/rtp/udp','rtp/udp','mp2t/rtp/udp','mp2t/udp']
```

The following functions

per_packet2per_frame_rtp, per_packet2per_frame_mp2t

aggregate the packet header data given per-packet to data given per-video frame.

```
def gen_per_frame_mp2t(packet_data):
    """
    INPUT
    packet_data -- DataFrame containing data-rows corresponding to individual packets

    OUTPUT
    generates a tuple summarizing the packet_data corresponding to a video frame
    """
    time_min = -1
    time_max = -1
    size = 0
    lost = 0
    count = 0
    ts_packet_size = 188 - 4

    # take payload start indicator with an offset of 1, i.e. this is the payload end indication, add
    # an additional end indication to the very end:
```

```

payload_end = [packet_data.col('mp2t.pusi')[i] for i in range(1,len(packet_data))] + [1]
for i in range(len(packet_data)):

    count += 1

    if time_min<0:
        time_min = packet_data.col('frame.time_relative')[i]
    else:
        time_min = min(time_min,packet_data.col('frame.time_relative')[i])

    time_max = max(time_max,packet_data.col('frame.time_relative')[i])

    size += ts_packet_size

    # count lost packets
    if i>0:
        diff = packet_data.col('mp2t.cc')[i] - packet_data.col('mp2t.cc')[i-1]
        # compute mod 16
        diff = mod(diff,16)

        if diff>1:
            if not packet_data.col('mp2t.analysis.drops')[i]:
                print('!!! mp2t.analysis.drops says there is no drop !!!')
            lost += diff-1

    #-- if the end of a video frame is reached, 'yield' results:-----

    if payload_end[i]:

        yield (time_min,\
              time_max,\
              size,\
              count,lost)

        time_min = -1
        time_max = -1
        size = 0
        lost = 0
        count = 0

def gen_per_frame_rtp(packet_data,sampling_rate=90000.):
    """
    INPUT
    packet_data -- DataFrame containing data-rows corresponding to individual packets

    OUTPUT
    generates a tuple summarizing the packet_data corresponding to a video frame
    """

    time_max = -1
    seq_nr = -1
    size = 0
    lost = 0
    count = 0

    for i in range(len(packet_data)):

        count += 1
        time_max = max(time_max,packet_data.col('frame.time_relative')[i])

        if seq_nr<0:
            seq_nr = packet_data.col('rtp.seq')[i]

        size += packet_data.col('udp.length')[i]

        is_rebuf = is_rebuffering_start(packet_data.col('rtp.seq'),i)

        rebuf_flag = 0
        if is_rebuf:
            rebuf_flag = 1

        # count lost packets
        if i>0 and not is_rebuf:
            diff = packet_data.col('rtp.seq')[i] - packet_data.col('rtp.seq')[i-1]
            # compute mod 2**16, i.e. diff = 0 - 65535 is no packet loss, but restart

```

```

        # of rtp.seq number:
        diff = mod(diff,2**16)

        if diff>1:
            lost += diff-1

    #-- if the end of a video frame is reached, 'yield' results:-----
    i_next = min(i,len(packet_data)-1)
    ts = packet_data.col('rtp.timestamp')
    time_stamp_diff = ts[i_next] - ts[i]

    is_before_rebuf_start = is_rebuffering_start(packet_data.col('rtp.seq'),i_next)

    if packet_data.col('rtp.marker')[i]==1 or time_stamp_diff>0 or is_before_rebuf_start:
        # note: compute relative time stamp in miliseconds

        yield (time_max,\
              (packet_data.col('rtp.timestamp')[i])/sampling_rate*1000,\
              seq_nr,\
              packet_data.col('rtp.seq')[i],\
              size,\
              count,lost,\
              rebuf_flag)

        time_max = -1
        seq_nr = -1
        size = 0
        lost = 0
        count = 0

def extract_video_stream(packet_data,verbose=False):
    """
    INPUT
    packet_data -- DataFrame containing data-rows corresponding to individual packets of video and
    audio streams

    OUTPUT
    packet_data -- DataFrame containing data-rows corresponding to individual packets of the video
    stream

    """
    stream_id = set(packet_data.col('mp2t.pid'))

    # check for the largest stream
    s = sorted(list(stream_id))
    bins = s + [s[-1]+1]
    count, bins = histogram(packet_data.col('mp2t.pid'),bins=bins)
    m = count.argmax()
    video_stream_id = s[m]

    if verbose:
        print('video stream has id=%d' % video_stream_id)

    # assume, the largest stream is video:
    I = where(packet_data.col('mp2t.pid')==video_stream_id)[0]
    data_video = packet_data.data[I,:]
    pd_video = dfr.DataFrame(data_video,packet_data.col_name)

    return pd_video

def per_packet2per_frame_mp2t(packet_data,verbose=False):
    """
    Convert packet data (mpeg2 ts) to frame data. Remove duplicates.

    INPUT
    packet_data -- DataFrame containing data-rows corresponding to individual packets

    OUTPUT
    frame_data -- DataFrame containing data-rows corresponding to frames
    """

    col_names = ['frame.timestamp_min',\
                 'frame.timestamp_max',\
                 'video_frame_size',\
                 'packet_count',\

```

```

        'lost']

pd_video = extract_video_stream(packet_data,verbose=verbose)

# add +1 for the end indication at the end, see gen_per_frame_mp2t
frame_count = sum((1 for m in pd_video.col('mp2t.pusi') if m>0.5)) + 1

D = []

for i,frame_data in enumerate(gen_per_frame_mp2t(pd_video)):
    D.append(frame_data)

return dfr.DataFrame(array(D),col_names)

def skip_delayed_packets(t_sent,t_received,seq_nr,max_allowed_delay=0.5,sampling_rate=90000.):
    """
    Skip packets with a delay longer than 'max_allowed_delay' (s).

    INPUT:
    t_sent          -- packet rtp time stamp (using sampling_rate)
    t_received      -- captured packet time stamp (in ms)
    seq_nr          -- rtp seq. number
    max_allowed_delay -- float, allowed delay in sec, before packet is considered lost
    sampling_rate   -- corresponding to rtp timestamp

    OUTPUT:
    array(I)  -- ndarray (1-D), index, containing all non-delayed packets
    """

    n = len(t_sent)
    I = [0]
    for i,j in zip(range(n-1),range(1,n)):

        if is_rebuffering_start(seq_nr,j):
            I.append(j)

            elif (t_sent[j]-t_sent[i])/sampling_rate < (t_received[j]-t_received[i])/1000.0 +
max_allowed_delay:
                I.append(j)

    return array(I)

def is_rebuffering_start(seq_nr,i):
    """
    Check if at position i there is a start of rebuffering.
    Note, after 'correct_seq_nr_at_rebuffering', rebuffering is
    determined by a jump of the seq number of 2**16.
    """

    is_rebuf_start = False

    if i>0:
        m = 2**16
        n0 = seq_nr[i-1]
        n1 = seq_nr[i]

        if n1-n0>=m:
            is_rebuf_start = True

    return is_rebuf_start

```

At rebuffering, the sequence number can jump to smaller or any other values. The sequence number is limited to 2^{16} . Then it restarts at 0. Correct the value of the sequence number such that it is increasing, to avoid incorrect re-ordering later. Thus, add 2^{16} to the sequence number at each rebuffering and overflow position.

```

def correct_seq_nr_at_overflow_and_rebuffering(seq_nr,t_received,verbose=False):
    """
    INPUT:
    seq_nr          -- 1-D numpy array, rtp seq number
    t_received      -- 1-D numpy array, captured packet time stamp (in ms)

    OUTPUT:
    seq_nr          -- the corrected seq number

```

```

"""

N = len(seq_nr)

# window size
w = 10

# rebuffering limit, 500 ms
rebuff_limit = 500
seq_nr_diff_at_rebuff = 1000

I_rebuf = []
I_overflow = []

for i in range(1,N):

    # check for overflow (check sequence number in a small window,
    # in case of re-orderings
    max_before = max(seq_nr[max(0,i-w):i])
    min_after = min(seq_nr[i:min(i+w,N)])

    if max_before==2**16-1 and min_after==0:
        I_overflow.append(i)

    # check for rebuffering: a temporal gap in packet arrival and a jump
    # (positive or negative) in seq number
    delta_t = t_received[i] - t_received[i-1]

    if delta_t > rebuff_limit and abs(seq_nr[i-1] - seq_nr[i])> seq_nr_diff_at_rebuff:
        I_rebuf.append(i)

# adjust sequence number:
for i in sorted(set(I_rebuf+I_overflow)):

    if i in I_rebuf and i>0:
        seq_nr[i:] += seq_nr[i-1] - seq_nr[i] + (2**16)

    else:
        seq_nr[i:] += (2**16)

return seq_nr

def per_packet2per_frame_rtp(packet_data,verbose=False):
    """
    Convert packet data (rtp) to frame data. Remove duplicates.

    INPUT
    packet_data -- DataFrame containing data-rows corresponding to individual packets

    OUTPUT
    frame_data -- DataFrame containing data-rows corresponding to frames
    """

    col_names = ['frame.timestamp_max',\
                 'rtp.timestamp',\
                 'rtp.seq_first',\
                 'rtp.seq_last',\
                 'video_frame_size',\
                 'packet_count',\
                 'lost',\
                 'rebuf_start']

    # check for rebuffering

    seq_nr = packet_data.col('rtp.seq')
    seq_nr_cor =
correct_seq_nr_at_overflow_and_rebuffering(seq_nr,packet_data.col('frame.time_relative'))

    # remove duplicates and sort according to rtp.seq number
    seq_nr_no_dup,I = remove_duplicate(seq_nr_cor)
    I_sorted = argsort(seq_nr_no_dup)

    packet_data.data = packet_data.data[I[I_sorted],:]

```

```

    # skip delayed packets
    I_skipped =
skip_delayed_packets(packet_data.col('rtp.timestamp'),packet_data.col('frame.time_relative'),packet_
data.col('rtp.seq'))
    packet_data.data = packet_data.data[I_skipped,:]
    D = []
    for i,frame_data in enumerate(gen_per_frame_rtp(packet_data)):
        D.append(frame_data)

    return dfr.DataFrame(array(D),col_names)

```

A.2 Extraction of video frame feature statistics

This module computes statistics based on features of the video frames.

MODULE INPUT:

```
video_filename -- name of the avi file of the video (PVS)
```

MODULE OUTPUT:

```
video_result -- result list containing the feature statistic values
```

The following features are extracted:

- video frame resolution,
- motion statistics,
- interframe difference statistics,
- spatio-temporal complexity statistics,
- frame display time (inverse of frame rate),
- scene change statistics.

In more detail, the processing is as follows: The video frame features are stored in a dictionary, `video_result`, of the form:

```
{'display_time':[frame0_display_time,frame1_display_time,...],
'complexity':[frame0_complexity,frame1_complexity,...],...}
```

For each frame in the `frame_seq` container the following feature values of the above listed features are kept, explicitly (for later reference), frame resolution:

```
video_result['resolution'] = [(frame_height,frame_width),...],
```

motion statistics

```
video_result['motion_stat'] = [[m.avg_stat(i,0),m.avg_stat(i,1)],...]
video_result['motion_pred_stat'] = [m.motion_prediction_stat(i),...]
```

interframe difference

```
video_result['interframe_diff'] = [m.interframe_diff_stat(i),...]
```

and complexity statistics

```
video_result['complexity_stat'] = [complexity_stat(frame_seq,i),...],
```

where `i` denotes the index running through all frames. As a final step, compute scene changes

```
video_result['scene_stat'] = scene.scene_change_statistic(frame_seq)
video_result['scene_decomposition'] = scene.decompose(frame_seq)
```

and add the display time of each frame (inverse of frame rate) to the result list,

```
video_result['display_time'] = frame_seq.display_time.
```

The main idea to compute the complexity and motion features is to observe 3x3 blocks of the video frame and to compute the similarity of spatially and temporally adjacent blocks and to compute the

predictability of a block, by blocks of the previous frame at the same and spatially adjacent locations (see Figure A.1).

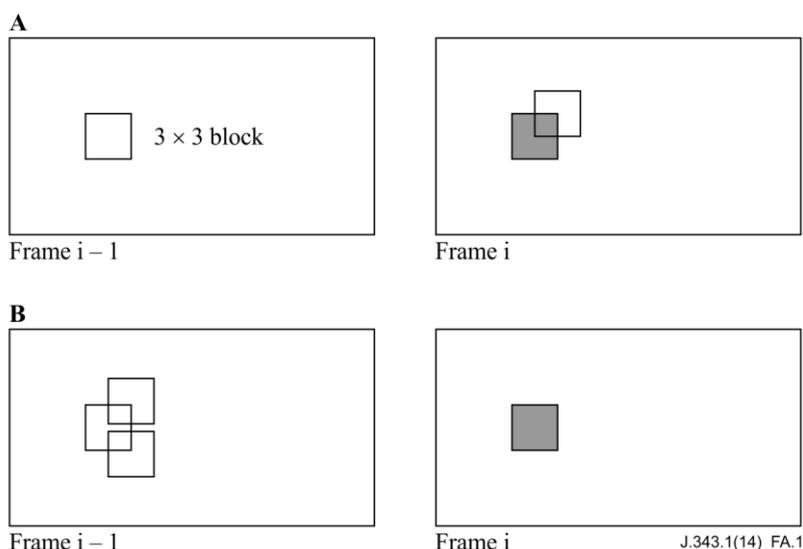


Figure A.1 – Top A: For complexity statistics, similarity of a block (grey) to spatially or temporally adjacent blocks (white) are computed.

Bottom B: For motion statistics, predictability of a block (grey) to spatially and temporally adjacent blocks (white) are computed.

In the next clauses, the details of these computations are described.

Complexity

Compute complexity statistic of frame sequence, by computing local inter-frame and intra-frame dissimilarity of frames i and $i-1$ using the function:

```
complexity_stat(frame_seq,i)

def dissim_stat(S):
    """
    Compute dis-similarity statistics:

    INPUT:
    S -- numpy array (matrix) of local similarity values

    OUTPUT:
    pair containing values, mean dissimilarity and probability of being equal.
    """

    # number of equal values (similarity==1) / number of values
    prob_eq = len(np.where(S>(1.0-1e-3))[0]) / float(np.product(S.shape))

    return (1.0-np.mean(S),prob_eq)

def complexity_stat_inter_intra(A_YCbCr,B_YCbCr,local_sim_par=5.0,verbose=False):
    """
    Compute a complexity statistic for the data corresponding
    to 2 consecutive frames.

    INPUT:
    A_YCbCr,B_YCbCr -- triples of numpy arrays (matrices) containing frame data

    OUTPUT:
    c_stat -- dict, containing inter and intra dict of sim_stat results
    """

    # compute similarity statistics at every 4th location
    h,w = A_YCbCr[0].shape

    if verbose:
        print('complexity stat resolution (h,w)=' ,h,w)
```

```

S_inter = np.zeros((40,40),dtype=np.float32)
S_intra = np.zeros((40,40),dtype=np.float32)

c_stat = dict()
c_stat['inter'] = dissim_stat(S_inter)
c_stat['intra'] = dissim_stat(S_intra)

return c_stat

```

The function `frameProc.local_sim_stat(A,B,S,c)` chooses 40x40 equally spaced points in the frame (ignoring a border of 4 to avoid border problems) and computes at each position (i,j) :

```
S[i,j] = exp(-RMSE(a(i,j),b(i,j))/c),
```

where $a(i,j)$ denotes a 3x3 array in A around (i,j) , and $RMSE$ denotes the root mean square error.

```

def complexity_stat(frame_seq,i,verbose=False):
    """
    Compute complexity stat for frames i,i-1 at all levels of the pyramid.
    Return a trivial result for frame 0.

    INPUT:
    frame_seq -- a FrameSeq instance
    i         -- int, frame number
    verbose   -- bool

    OUTPUT:
    result    -- ndarray, of dim (each level) x (inter/intra) x (mean/prop_eq)
    """

    c_rel = ['inter','intra']
    c_type = ['mean','prob_eq']

    result = np.zeros((frame_seq.common_pyramid_height(),\
                      len(c_rel),\
                      len(c_type)))

    for level in np.arange(frame_seq.common_pyramid_height()):
        # compute complexity stat at this level:
        # create trivial stat
        c_stat = {'inter':(0.0,0.0),'intra':(0.0,0.0)}

        if i>0:
            f = frame_seq
            A_YCbCr = (f.Y[i-1][level],f.Cb[i-1][level],f.Cr[i-1][level])
            B_YCbCr = (f.Y[i][level], f.Cb[i][level], f.Cr[i][level])
            c_stat = complexity_stat_inter_intra(A_YCbCr,B_YCbCr,verbose=verbose)

        result[level,0,:] = c_stat['inter']
        result[level,1,:] = c_stat['intra']

    return result

```

Motion estimation

The following parameters are used for the motion estimation and are retrieved using the function:

```

def get_parameters():
    param = dict()

    # check for motion at this location if mean squared diff is
    # larger than this limit
    param['motion_limit'] = 0.1

    # conversion from dissimilarity values to probabilities:
    # first compute entropy, then parametric conversion
    param['entropy_to_prob'] = 4.0

    # determine motion up to resolution having height < value below.
    # NOTE: check that at these frame sizes the frame data is smoothed.
    param['max_frame_height_for_mEstim'] = 60

    # constant, to convert from rmse to probability value:
    param['similarity_const'] = 10.0

```

```

return param

# utility functions
def shifted_range(N):
    """
    return (array([0,0,1,...,N-2]),array([1,2,...,N-1,N-1]))
    """
    r = np.arange(-1,N-1)
    r[0] = 0
    r1 = np.arange(1,N+1)
    r1[-1] = r1[-2]
    return (r,r1)

```

An array of four displacement vectors dxdy are used in counter clockwise direction,

```

      o-----> dxdy[i+1]
      |
      |
      v
dxdy[i]

```

in total for all four

```

directions counter clock-wise:
|
0: | 1: --> 2: | 3: <--
v          |

```

then, the following quadrants are used for motion estimation:

```

      +
      |
q_2 | q_1
<---o----->
q_3 | q_0
      |
      v

```

```

class Dx Dy:
    def __getitem__(self, key):
        return self.dxdy[mod(key, len(self.dxdy))]

    def __len__(self):
        return len(self.dxdy)

```

The `PDist` container is used to hold the estimated motion values:

```

class PDist:
    """
    Container for estimated motion, estimated separately for each quadrant, together
    with a probability distribution relating to the confidence in estimated motion.

    p_dist[l][:,i,j] is the probability distribution at level l for motion
    of a neighbourhood of (i,j) for each of the 4 quadrants around (i,j).
    The quadrants are defined by the class Dx Dy.

    p_dist -- list of np.array of dim len(Dx Dy) x height x width, list length is pyramid_level

    r_shift -- list of np.array of dim len(Dx Dy) x height x width, list length is pyramid_level
    s_shift -- list of np.array of dim len(Dx Dy) x height x width, list length is pyramid_level

    motion_computed -- list of np.array of dim height x width, points, at which a motion
    estimation was performed.

    dx,dy -- list of np.array of dim height x width, for each pyramid level
    estimated motion at this resolution

    dx_prior,dy_prior -- list of np.array of dim height x width, for each pyramid level, the
    prior displacement used for motion estimation

    |-----> s_shift
    |.....

```

```

        |.....
        V
        r_shift
"""

def __len__(self):
    return len(self.p_dist)

def __getitem__(self, key):
    return self.p_dist[key]

def __setitem__(self, key, item):
    self.p_dist[key] = item

def avg_stat(self, level):
    """
    Compute average statistics of motion

    INPUT
    level -- int, resolution level, at which the statistics will
            be computed

    OUTPUT
    (xfrac,yfrac,mx,my) -- tuple, 4 float values,
                        xfrac/yfrac, relative number of locations, at which
                        estimated x-/y-motion is larger than 0.1
                        mx/my, mean absolute motion in x-/y-direction at the
                        positions with motion larger than 0.1
    """

    dx = self.dx[level]
    dy = self.dy[level]

    ix = np.where(abs(dx)>0.1)
    iy = np.where(abs(dy)>0.1)
    xfrac,mx = 0.0,0.0
    if len(ix[0])>0:
        xfrac = len(ix[0])/float(np.prod(dx.shape))
        mx = np.mean(abs(dx[ix]))
    yfrac,my = 0.0,0.0
    if len(iy[0])>0:
        yfrac = len(iy[0])/float(np.prod(dx.shape))
        my = np.mean(abs(dy[iy]))
    return (round(xfrac,3), round(yfrac,3), round(mx,4), round(my,4))

def motion_prediction_stat(self):
    """
    OUTPUT
    (frac_motion,mean_sim) -- pair of fraction of 'moving' local regions, and
                            mean 'confidence'(=similarity) in prediction.
    """
    p = self.p_dist[-1] # level of highest resolution
    p_max = p.max(axis=0) # maximum among p values of different motion directions

    m = self.motion_computed[-1]
    # fraction, where no motion estimate is computed
    frac_motion = sum(m)/float(prod(m.shape))

    I = where(m>0.5)
    mean_sim = 1.0
    if len(I[0])>0:
        mean_sim = mean(p_max[I])
    return (round(frac_motion,4), round(mean_sim,4))

```

The function

```
def expect(level,bRound)
```

computes the expected motion as arrays (dx,dy) at the level `level` of the pyramid together with a confidence value. The estimate is given by

$$dx,dy = \sum (p_dist[m] * (r[m]*dxdy[m] + s[m]*dxdy[m+1])),$$

where the sum runs over the four quadrants indexed by `m`. Furthermore, the confidence value `p` is computed as

```
p[i,j] = <p_dist[:,i,j] , log(p_dist[:,i,j])>
```

where `<.,.>` denotes the inner product. If `bRound=True` the estimates of (dx, dy) are rounded to an integer value. def `expect(self,level,bRound=True)`:

```
def next_level_prior(self,level,dx,dy,dx_prior,dy_prior):
    """
    At level 'level' the motion prior is (dx_prior,dy_prior) and
    the estimated update is (dx,dy). Compute from these a motion
    prior at level 'level+1' (higher resolution).
    """

    if self.__len__()>level+1:
        I = np.where(self.motion_computed[level]==1)
        dx[I] = dx[I] + 2*dx_prior[I]
        dy[I] = dy[I] + 2*dy_prior[I]

        # upsample to next level:
        dim = self.p_dist[level+1][0,:].shape
        dx_up = np.zeros(dim)
        dy_up = np.zeros(dim)
        frameProc.upsample_by2_smooth(dx,dx_up)
        frameProc.upsample_by2_smooth(dy,dy_up)

        dx_up = dx_up.round()
        dy_up = dy_up.round()

        # store computed prior:
        self.dx_prior[level+1] = dx_up
        self.dy_prior[level+1] = dy_up

    return (dx_up,dy_up)
```

The motion container and its methods are used for performing the motion estimation. See the main method

```
Motion.estimate()
```

for the entry point.

```
class Motion:
    """
    Motion estimation for a FrameSeq object:

    dx, dy -- list of matrices, estimated motion vectors
    p -- list of matrix, probability value, confidence in estimate
    p_dist -- list of objects of type PDist

    motion_estimated -- int, motion is estimated up to this frame
    max_topDown_level -- max resolution used for motion estimation
    min_frame_time_diff -- int, min temp difference of two frames (in ms) for motion estimation

    d_max -- int, max motion at given resolution
    """

    # max displacement for search
    d_max = 1 # nothing else supported for PDist!

    # dx,dy measure displacement with respect to delta time (in ms):
    dt_for_motion = 40.0
    # motion is estimated between two frames with a
    # temporal difference of min_frame_time_diff or larger
    min_frame_time_diff = 30

    motion_estim_start_level = 0

    motion_half_region_size = 1
    frames_for_motion = list()

    def determine_max_level(self):
        """
        determine pyramid level corresponding to a resolution
        with frame width smaller
        than max_frame_height_for_mEstim (skip highest
        resolution of Y, in any case, as Cb, Cr planes do not have this).
        """
```

```

y_pyramid = self.frame_seq.Y[0]
self.max_topDown_level = 0
for i in range(len(y_pyramid)-1): # -1, as Cb, Cr planes have 1 level less
    if y_pyramid[i].shape[0] < get_parameters()['max_frame_height_for_mEstim']:
        self.max_topDown_level = i
    else:
        break

def display_time(self):
    """
    Determine the display time of each frame.
    The display time of the last frame is 0.
    """
    # time = self.frame_seq.time_stamp

    # disp_time = [time[i+1]-time[i] for i in range(len(time)-1)]
    # disp_time.append(0) # display time of last frame

    # self.disp_time = disp_time
    # self.time = time
    self.disp_time = self.frame_seq.display_time
    t = [0]
    for d in self.disp_time:
        t.append(t[-1]+d)
    self.time = array(t[:-1])

def append_zero_motion_estim(self):
    """
    Skip the first few frames for motion estimation, but append
    'zero-arrays', such that all lists have the same length.
    """
    dim = self.frame_seq.Y[0][self.max_topDown_level].shape
    self.dx.append(np.zeros(dim,dtype=np.float32))
    self.dy.append(np.zeros(dim,dtype=np.float32))
    self.p.append(np.zeros(dim,dtype=np.float32))
    self.frames_for_motion.append((0,0))
    pyramid_dim = self.frame_seq.Y[0].get_dim()[self.max_topDown_level+1]
    self.p_dist.append(PDist(pyramid_dim))

def estimate(self,estim_up_to_frame=5):
    """
    Perform motion estimation for the video sequence self.frame_seq.
    """
    estim_up_to_frame = min(estim_up_to_frame,len(self.frame_seq)-1)
    if self.motion_estimated < len(self.frame_seq)+1:
        dx,dy = self.dx,self.dy
        p_dist = self.p_dist

        # loop over all frames up to frame_end
        for t in range(self.motion_estimated,estim_up_to_frame+1):

            # do not perform motion estimation for first few frames
            # (until self.min_frame_time_diff)
            if sum(self.disp_time[:t]) <= self.min_frame_time_diff:
                self.append_zero_motion_estim()
                continue

            # consider Y pyramid at time t0 and t ...
            # determine t0,such that frames are > than min_frame_time_diff apart
            t0 = t-1
            while self.time[t]-self.time[t0]<= self.min_frame_time_diff:
                t0 -= 1

            self.frames_for_motion.append((t0,t))
            Y0 = self.frame_seq.Y[t0]
            Y1 = self.frame_seq.Y[t]
            # ... and analogously Cb, Cr
            Cb0,Cb1 = self.frame_seq.Cb[t0],self.frame_seq.Cb[t]
            Cr0,Cr1 = self.frame_seq.Cr[t0],self.frame_seq.Cr[t]

            m0 = self.motion_estim_start_level

            shift_dx,shift_dy = (0,0)

            Dx = np.zeros(Y0[m0].shape,dtype=np.float32)+shift_dx # matrix of dim of first
element in Y pyramid

```

```

        Dy = np.zeros(Y0[m0].shape,dtype=np.float32)+shift_dy # matrix of dim of first
        element in Y pyramid

        # init probability pyramid (of same dim as Y0)

        pyramid_dim = self.frame_seq.Y[0].get_dim()[:self.max_topDown_level+1]
        p_dist_per_frame = PDist(pyramid_dim)

        # loop over resolution pyramid, from top down.
        # stop at level self.max_topDown_level
        for r in range(self.motion_estim_start_level,self.max_topDown_level+1):

            # determine global motion
            shift_dx,shift_dy = (0,0)

            Dx_prior = Dx + shift_dx
            Dy_prior = Dy + shift_dy

            YCbCr0 = (Y0[r],Cb0[r],Cr0[r])
            YCbCr1 = (Y1[r],Cb1[r],Cr1[r])

            delta = 1 # half region size used for motion estimation

        self.estimate_per_frame(YCbCr0,YCbCr1,Dx_prior,Dy_prior,p_dist_per_frame.dxdy,\
                                p_dist_per_frame[r],\
                                p_dist_per_frame.motion_computed[r],\
        p_dist_per_frame.r_shift[r],p_dist_per_frame.s_shift[r])
            bRound = False
            (Dx,Dy,P) = p_dist_per_frame.expect(r,bRound)

            # prepare prior for next higher resolution
            if r<self.max_topDown_level:
                Dx,Dy = p_dist_per_frame.next_level_prior(r,Dx,Dy,Dx_prior,Dy_prior)

            else:
                Dx = Dx + Dx_prior
                Dy = Dy + Dy_prior

            # normalise Dx,Dy with respect to dt_for_motion
            time_fac = self.dt_for_motion/(self.time[t]-self.time[t0])
            Dx = Dx * time_fac
            Dy = Dy * time_fac

            # append motion estimates at time t to list
            dx.append(Dx)
            dy.append(Dy)
            self.p.append(P)
            self.p_dist.append(p_dist_per_frame)

        self.motion_estimated = t+1

    def
    estimate_per_frame(self,YCbCr0,YCbCr1,Dx_prior,Dy_prior,dxdy,p_dist,motion_computed,r_shift,s_shift,
    sim_par=10):
        """
        Estimate motion at given time and resolution.

        INPUT:
        YCbCr0, YCbCr1      -- triple of numpy arrays, representing consecutive frames
        Dx_prior, Dy_prior -- numpy array (matrix) of prior values for displacement

        p_dist              -- write here prob distribution of motion estimation
        motion_computed    -- write here at which positions motion was estimated
        r_shift            -- write here estimated shift in given direction, value in (0,1)

        OUTPUT:
        (dx,dy,p) -- triple of numpy arrays, representing motion vectors
                   at corresponding location and their probability of
                   being correct

        """

```

```

Y0,Cb0,Cr0 = YCbCr0
Y1,Cb1,Cr1 = YCbCr1
bd = 2
delta = 1
for i in range(bd,Y0.shape[0]-bd-1):
    for j in range(bd,Y0.shape[1]-bd-1):

        dx_prior = int(Dx_prior[i,j])
        dy_prior = int(Dy_prior[i,j])
        ri = i + np.arange(dx_prior-delta, dx_prior+delta+1)
        rj = j + np.arange(dy_prior-delta, dy_prior+delta+1)

        a = Y0[np.ix_(ri,rj)]
        b = Y1[np.ix_(ri,rj)]
        d_ab = b - a

        # check for enough motion
        msqd = np.mean(d_ab*d_ab)

        if msqd > get_parameters()['motion_limit']:

            motion_computed[i,j] = 1
            # loop over possible displacements
            number_dxdy = len(dxdy)
            for m in arange(number_dxdy):

                dxm,dym = dxdy[m]
                dxn,dyn = dxdy[m+1]

                # -dxm, -dym, i.e. the minus is set, such that an object
                # moving downwards (in +x direction), has a high value
                # in p_dist in +x direction
                d0 = Y0[ix_(ri-dxm,rj-dym)] - a
                d1 = Y0[ix_(ri-dxn,rj-dyn)] - a

                # Determine r,s as the least square estimates of the following
                # equations:
                # |<d0,d0> <d1,d0>| |r| = |<d_ab,d0>|
                # |<d0,d1> <d1,d1>| |s| = |<d_ab,d1>|

                # the implementation is straight forward

                err = r*d0+s*d1-d_ab
                p_dist[m,i,j] = np.exp(-np.mean(err*err)/sim_par)

self.normalise(p_dist)

def interframe_diff_stat(self,frame_nb):
    """
    Compute interframe difference (difference to previous frame) at a low resolution level.
    Return the mean and max interframe difference.

    OUTPUT:

    (if_diff_avg,if_diff_max) -- pair of float, average and max interframe
    difference.
    """
    if_diff_avg = 0.0
    if_diff_max = 0.0

    n = min(1,len(self.frame_seq.Y)-1) # select level 1 of pyramid

    if frame_nb>0 and len(self.frame_seq.Y)>1:

        Y0 = self.frame_seq.Y[frame_nb-1][1]
        Y1 = self.frame_seq.Y[frame_nb][1]
        d = (Y1-Y1.mean()) - (Y0-Y0.mean())
        d = d*d

        if_diff_avg = sqrt(d.mean())
        if_diff_max = sqrt(d.max())

    return (round(if_diff_avg,2),round(if_diff_max,2))

def avg_stat(self,frame_nb,level):

```

```

        return self.p_dist[frame_nb].avg_stat(level)

def motion_prediction_stat(self, frame_nb):
    return self.p_dist[frame_nb].motion_prediction_stat()

def normalise(self, p_dist):
    """
    normalise p_dist, such that sum(p_dist)<=1.
    It could be that locally, an occlusion situation, or another image change happens,
    which is not a translation, therefore, if at a point p all values in p_dist are
    very small, keep them small, i.e. sum(p_dist[:, :, px, py]) < 1.
    Practically, use the constant 'max_norm_const'.
    """
    #
    max_norm_const = 0.2

    for i in range(p_dist.shape[1]):
        for j in range(p_dist.shape[2]):

            p_local = p_dist[:, i, j].squeeze()
            # make sum(p_local)<=1
            p_local = p_local/np.maximum(max_norm_const, np.sum(p_local))
            p_dist[:, i, j] = p_local

```

Scene decomposition of video sequence

Decomposition of a frame sequence into scenes. See the main method `decompose` below. The following parameters are used for the scene decomposition and retrieved using the function:

```

def get_parameters():
    """
    Return dict of parameters used for scene decomposition
    """
    param = dict()

    # minimal resolution (height) of frame used for scene detection
    param['frame_height_min'] = 30

    # number of frames used for scene change detection:
    # use more than two frames, to avoid scene change detection e.g. for a
    # scene containing a flash light
    param['number_frames'] = 5

    # number of tiles used for scene change detection
    # i.e. n x n tiles will be used
    param['number_tiles'] = 3

    # minimal time interval between scenes in ms
    param['min_scene_duration'] = 600

    # increase factor for scene detection, of dis-similarity at current
    # position in time with respect to average dis-similarity over
    # current scene
    param['scene_detect_fac'] = 5
    # minimal dis-similarity for scene change:
    param['scene_detect_thresh'] = 0.001

    return param

def pyramid_level(frame_seq, height_min):
    """
    Return the level l of the pyramid, such that for the height of
    the frame

    heigth(l-1) < height_min <= height(l)

    """
    dim = frame_seq.Y[0].get_dim()

    ind = [i for i in reversed(range(len(dim))) if dim[i][0] >= height_min]

    return ind[-1]

def frame_index_given_by_duration(frame_seq, i, duration):
    """
    Return the frame index j, such that either j=0 or j is displayed around

```

```

'duration' before i.
"""
j = 0
while sum(frame_seq.display_time[j:i])>duration:
    j = j+1

return j

def frame_statistic(H0,H1,verbose=False):
    """
    Return a dis-similarity of the frames, in [0,1+epsilon], (epsilon depend on
    numerical accuracy).
    The dis-similarity is the median of local dis-similarity values, computed
    by comparing regularly spaced tiles.
    For efficiency, instead of the frames/frame-tiles, pre-computed histograms
    per tile are given:

    INPUT:
    H0,H1 -- histogram of tiled frames, numpy array of dim m x n x hist_size

    OUTPUT:
    float, dissimilarity

    """
    m,n = H0.shape[:2]

    # dissim will contain n x n values of local dissimilarities
    dissim = zeros((m,n))

    # n_bins = 10
    bins = [0,50,75,100,125,150,175,200,255]
    n_bins = len(bins)-1

    for i in arange(m):
        for j in arange(n):

            h0 = H0[i,j,:]
            h1 = H1[i,j,:]

            # note: h0/sum(h0) is a vector in the n_bins-dimensional
            # unit cube
            d = sqrt(mean((h0-h1)*(h0-h1)))

            dissim[i,j] = d/sqrt(n_bins)
    # allow for some large changes in some tiles, thus
    # return the median dis-similarity, as the frame statistic:
    return median(sorted(dissim.flatten()))

def get_frame_seq_tile_hist(frame_seq):
    """
    Cut n x n tiles out of the frame sequence and compute a histogram over the tile.

    INPUT:
    frame_seq -- frame sequence

    OUTPUT:
    tile_hist -- numpy array of dimension n x n x len(frame_seq) x n_bins
    """
    param = get_parameters()

    l = pyramid_level(frame_seq,param['frame_height_min'])

    bins = [0,50,75,100,125,150,175,200,255]
    n_bins = len(bins)-1

    # init tiles:
    n = get_parameters()['number_tiles']

    h,w = frame_seq.Y[0].P[1].shape
    len_h = h/n
    len_w = w/n

    # n x n tiles
    tile_hist = zeros((n,n,len(frame_seq),n_bins))

    for i in arange(n):

```

```

    for j in arange(n):

        # cut a rectangular tile from the frames
        r_h = arange(i*len_h, (i+1)*len_h)
        r_w = arange(i*len_w, (i+1)*len_w)

        for k in arange(len(frame_seq)):

            M = frame_seq.Y[k].P[l]
            v = M[ix_(r_h,r_w)].flatten()
            h,b = histogram(v,bins,density=True)
            tile_hist[i,j,k,:] = h

    return tile_hist

def scene_change_statistic(frame_seq,verbose=False):
    """
    Compute a statistic, having high values at scene changes

    INPUT:
    frame_seq -- frame sequence

    OUTPUT:
    scene_stat -- array (1-D) of float, scene change statistic per frame
    """
    param = get_parameters()

    scene_stat = zeros(len(frame_seq))

    # compute histogram per tile
    tile_hist = get_frame_seq_tile_hist(frame_seq)

    for i in range(1,len(frame_seq)):

        # compute dissimilarity between frame i and i-1,i-2,...
        # then store the minimal dissimilarity
        frame_stat = []

        for j in range(min(i,param['number_frames'])):

            s = frame_statistic(tile_hist[:, :, i, :].squeeze(), tile_hist[:, :, i-j-1, :].squeeze())
            frame_stat.append(s)
            scene_stat[i] = min(frame_stat)

    return scene_stat

def decompose(frame_seq,verbose=False,scene_stat=zeros((0))):
    """
    Detect scene changes in frame sequence

    INPUT:
    frame_seq -- a frame sequence

    OUTPUT:
    (scene_start,scene_end) -- pair of lists of frame numbers of first frames of scenes
    and last frames of scenes, both have same length
    """
    param = get_parameters()

    if len(scene_stat)==0:
        scene_stat = scene_change_statistic(frame_seq)

    # by definition, there is a scene start at the beginning
    scene_start = [0]

    # decompose sequence according to scene statistic:
    i_last = frame_index_given_by_duration(frame_seq,len(frame_seq)-1,2*param['min_scene_duration'])
    for i in arange(i_last):

        # check if minimal scene duration has exceeded:
        scene_duration = sum(frame_seq.display_time[scene_start[-1]:i])
        if scene_duration > param['min_scene_duration']:

```

```

j = frame_index_given_by_duration(frame_seq,i,2*param['min_scene_duration'])

# median dissimilarity over last part of current scene
m = median(scene_stat[max(j,scene_start[-1]):i])
if scene_stat[i] > m * param['scene_detect_fac'] and\
scene_stat[i] > param['scene_detect_thresh']:
    # scene start detected
    # for fading:
    # check if dissimilarity increases, take the scene_start at the max position,
    # but check for increase only up to half 'min_scene_duration' after time of frame i:
    ii = i
    while scene_stat[ii+1]>scene_stat[i] and\
i>frame_index_given_by_duration(frame_seq,ii,0.5*param['min_scene_duration']):
        ii += 1
    scene_start.append(ii)

# if nothing is skipped:
scene_end = [i-1 for i in scene_start[1:]] + [len(frame_seq)-1]

return (scene_start,scene_end)

```

A.2.1 Alignment of edited PVS to PVS

As the video sequence (edited PVS), for which the quality is predicted, is only a part of the complete PVS, an alignment step is needed, which determines the start and end frame of the edited PVS.

MODULE INPUT:

```

video_pvs          -- the video sequence (complete)
video_edited_pvs   -- the edited (shorter) video sequence

```

MODULE OUTPUT:

```

frame_start,frame_end -- indices in longer sequence of the start/end of shorter sequence.

```

The start and end frames are determined as follows:

First take the middle frame k in the longer sequence L .
Match it to all frames in the shorter sequence S .

If it matches exactly one frame, say r , then $k-r$ is the start of the shorter sequence in L , and $k-r+\text{length}(S)$ is the end.

Else, loop over all frames f_r in shorter sequence S :
match frame r of S , f_r with all frames in longer sequence L .

If exactly one match can be found, say with frame k in L , then the position of S in L is determined as above.

Else if for each frame f_r more than one matches are possible, store for each f_r the set start position of S in L corresponding to each match,
and take the intersection of the possible start positions.
Select the first element of this intersection as start position of S in L .

A.3 Hybrid core model

Hybrid model core module

MODULE INPUT:

```

video_result          -- video frame feature statistics, from process_video module 2
bitstream_data        -- packet header data from pcap_extractor
transmitted_frame_height -- from pcap_extractor
frame_start           -- first frame and...
frame_end             -- ...last frame of edited PVS in PVS

```

MODULE OUTPUT:

```

HybridModel.q -- the estimated quality in the range [1,5]

```

All processing is performed by calling

```

HybridModel.process(video_result,bitstream_data,transmitted_frame_height,frame_start,frame_end),

```

which calls the estimation of coding and transmission quality

```
HybridModel.coding_quality()
HybridModel.transmission_quality()
```

The final quality estimate is the product of coding and transmission quality in the [0,1] range, re-scaled to the [1,5] MOS range, explicitly

$$q = 4 * q_{cod} * q_{trans} + 1.$$

The estimation of coding quality is based mainly on complexity and motion statistics – derived from the video frames – and on the total frame-size per scene – derived partly from the packet headers of the bitstream.

On the other hand, the estimation of transmission quality is based heavily on the bitstream. Still, statistics from video frames are needed, e.g., in case of repeated frames, to link rebuffering to packet loss.

In both cases, some correspondence of the packet data to the video frames is needed; this is estimated in the function

```
align_bitstream_content()
```

by matching scene changes to large frames in bitstream (see Figure A.2):

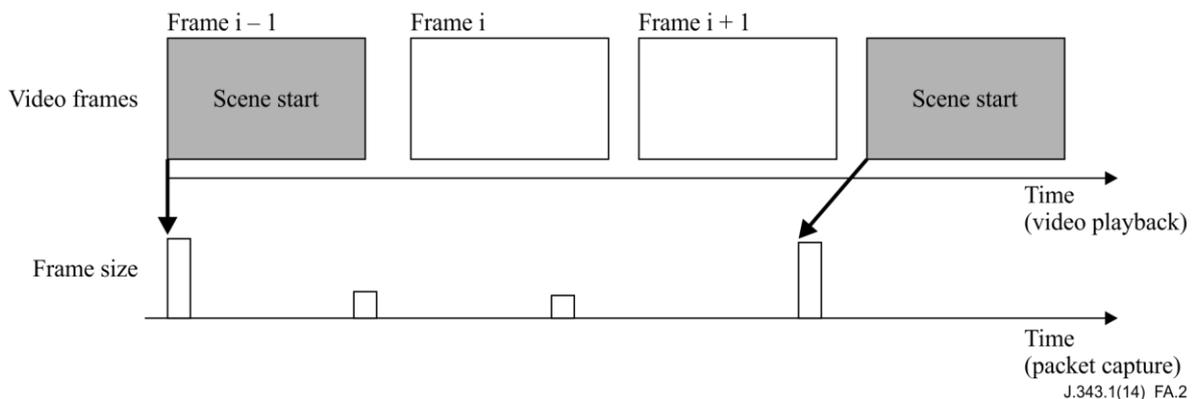


Figure A.2 – The bitstream and the video frames have their own time scale. The video frames are aligned by matching scene changes to large frames in bitstream

All the remaining details of the hybrid core model are following:

Model parameters for the core model are retrieved using the following function:

```
def get_hybrid_parameters():
    param = dict()

    # fade out time constant for temporal processing of per-frame degradations
    param['deg_temp_fade_out'] = 1.0
    # time interval used for temporal smoothing of degradations
    param['deg_temp_smooth_dt'] = 0.08

    # content to bitstream frame alignment: search in bitstream at
    # content scene start location +- 'bitstr_content_delta_time' for scene start I-frame
    param['bitstr_content_delta_time'] = 0.3 # in seconds

    # content to bitstream frame alignment: minimisation tradeoff between frame size
    # and mis-alignment
    param['misalign_tradeoff'] = 1.0/(4.0*param['bitstr_content_delta_time'])

    # map content to bitstream frames, max. offset at beginning (i.e. frames not captured)
    param['start_offset_max'] = 10

    param['unrepeated_frame_limit'] = 0.5

    # set a priori maximum motion masking effect (reduction of degradation)
    param['max_motion_masking'] = 0.5
```

```

# max length of B-frame interval
param['max_BFrame_interval'] = 3
return param

```

Hybrid model core methods and container are described in detail below:

```
class HybridModel:
```

The main function is `process`, which performs initial processing and calls the quality estimation functions, which are

```

coding_quality()
transmission_quality()

```

computing the quality factors, which are finally multiplied to yield overall quality.

```

def process(video_result,bitstream_data,transmitted_frame_height,frame_start,frame_end)
    v = video_result
    self.frame_start = frame_start
    self.frame_end = frame_end

    self.scene = v['scene_decomposition']
    self.scene_stat = v['scene_stat']

    # complexity statistics
    complex_stat = array(v['complexity_stat'])
    self.cplx_inter_avg = complex_stat[:, -1, 0, 0]
    self.cplx_inter_eq = complex_stat[:, -1, 0, 1]
    self.cplx_intra_avg = complex_stat[:, -1, 1, 0]
    self.cplx_intra_eq = complex_stat[:, -1, 1, 1]

    # motion statistics
    motion_pred = v['motion_pred_stat']
    self.frac_motion = array([ m[0] for m in motion_pred]) # fraction of moving blocks
    self.conf_motion = array([ m[1] for m in motion_pred]) # confidence in motion estimate

    motion_stat = v['motion_stat']
    low_res = 0 # motion stat at lowest resolution
    self.frac_motion_dx = array([m[low_res][0] for m in motion_stat])
    self.frac_motion_dy = array([m[low_res][1] for m in motion_stat])
    self.motion_dx = array([m[low_res][2] for m in motion_stat])
    self.motion_dy = array([m[low_res][3] for m in motion_stat])

    self.frame_height = v['resolution'][0][0]
    self.frame_width = v['resolution'][0][1]

    # interframe difference
    f_diff = array(v['interframe_diff'])
    self.frame_diff = f_diff[:, 0]
    self.frame_diff_max = f_diff[:, 1]

    self.dt = array(v['display_time'])/1000.0 # store display time in seconds

    # get bitstream data
    N = len(self.dt)

    self.transmitted_frame_height = transmitted_frame_height
    if self.transmitted_frame_height==0:
        self.transmitted_frame_height = self.frame_height

    N_b = len(bitstream_data)

    self.loss_b = bitstream_data.col('lost')
    self.frame_size_b = scale_frame_size(bitstream_data.col('video_frame_size'))
    self.packet_count_b = bitstream_data.col('packet_count')

    t_b = bitstream_data.col('frame.timestamp_max')/1000.0 # store time stamp in seconds
    t_b -= min(t_b)

    # add additional time stamp at end
    dt_mean = mean(t_b[1:]-t_b[:-1])
    t_b = array( list(t_b) + [t_b[-1]+dt_mean] )

    self.frame_t_b = t_b

    # add rebuffering start flag

```

```

self.rebuf_b = zeros(len(bitstream_data))
if 'rebuf_start' in bitstream_data.col_name:
    self.rebuf_b = bitstream_data.col('rebuf_start')

# add rtp timestamp
t_sent_b = arange(len(t_b))*dt_mean

if 'rtp.timestamp' in bitstream_data.col_name:
    t_sent_b = bitstream_data.col('rtp.timestamp')/1000.0 # store time stamp in seconds
    t_sent_b = correct_sent_timestamp(t_sent_b,self.rebuf_b)

    # add additional time stamp at end
    dt_sent_mean = mean(t_sent_b[1:]-t_sent_b[:-1])
    t_sent_b = array( list(t_sent_b) + [t_sent_b[-1]+dt_sent_mean] )

self.frame_t_sent_b = t_sent_b

# estimate frame types
ret_val =
get_frame_type_bstr(self.frame_size_b,self.frac_motion,self.conf_motion,sum(self.dt))

prob_key_frame_b = ret_val[0]
self.prob_key_frame_b_no_adj = prob_key_frame_b
self.prob_b_frame_b = ret_val[1]

# adjust frame type estimation using knowledge from content
self.prob_key_frame_b =
adjust_frame_type_at_scene_change(self.frame_size_b,self.loss_b,self.scene,self.dt,prob_key_frame_b)

# key frames are frames with prob(of key frame)>0.5
self.key_frame_b = where(self.prob_key_frame_b>0.5)[0]

# estimate loss
self.eff_loss_b,self.eff_loss_key_b =
get_eff_loss_bstr(self.loss_b,self.prob_key_frame_b,self.prob_b_frame_b,self.frame_size_b,self.packed_count_b,self.is_mp2t)

# for jerkiness,freezing, use knowledge from bitstream:
self.rep_frame = jrk.repeated_frame(self.frame_diff_max,self.dt)

# correct video display time (e.g. for offline simulation, in case of skipping without freezing)
self.dt_no_corr = self.dt
self.dt,self.dt_b =
correct_content_duration(self.dt,self.rep_frame,self.frame_t_sent_b,self.loss_b,verbose=False)

self.delay_b = check_delay_in_packet_arrival(self.frame_t_b,self.frame_t_sent_b,self.dt)

self.jerk_motion_weight_prior = jrk.get_motion_weighting(self.frame_diff_max)
self.jerk_motion_weight =
link_freezing_to_bitstream_stat(self.loss_b,self.delay_b,self.rebuf_b,self.dt_b,self.rep_frame,self.dt,self.jerk_motion_weight_prior)

self.jerk =
jrk.jerkiness(self.dt,self.rep_frame,self.jerk_motion_weight,self.get_viewing_distance2height())

self.estimate_quality()

```

The function `estimate_quality` combines coding and transmission quality. Note that the value factors `q_cod` and `q_trans` are on a `[0,1]` scale and have to be re-scaled to the `[1,5]` scale,

$$Q = 4 * q_{cod} * q_{trans} + 1.$$

The value `Q` is the final output of the model.

```

def estimate_quality(self):
    self.coding_quality()
    self.transmission_quality()

    self.q = self.q_cod*self.q_trans

    self.predicted_Quality = 4*self.q + 1

```

Coding quality estimates the perceived degradations due to encoding of the video and resizing.

```

def coding_quality(self):
    # do not count repeated frames
    lim = get_hybrid_parameters()['unrepeated_frame_limit']
    non_rep = [1 if self.rep_frame[i]<lim else 0 for i in range(len(self.rep_frame))]

    nr_frame_s = sum_per_scene(non_rep, self.scene)
    self.nr_frame_s = nr_frame_s

    scene_start = self.scene[0]
    scene_end = self.scene[1]

    dt = dt_no_rep(self.rep_frame, self.dt)

    f_s = average_per_scene(self.frac_motion, scene_start, scene_end, dt)
    c_s = average_per_scene(self.conf_motion, scene_start, scene_end, dt)
    self.frac_motion_s = f_s

    p_s = average_per_scene(self.conf_motion, scene_start, scene_end, dt)
    self.conf_motion_s = p_s
    p_s = self.get_conf_motion_estim(f_s, p_s)

    dt_s = sum_per_scene(self.dt, self.scene)
    self.dt_s = dt_s

    b_s, fsize_eff_b =
get_bitrate_per_scene(self.frame_size_b, self.key_frame_b, scene_start, scene_end, dt_s)

    self.frame_size_eff_b = fsize_eff_b
    nr_pixels = self.frame_height*self.frame_width

    # check if transmitted frame resolution was different:
    upscale_fac = 1.0

    if self.transmitted_frame_height > 0:
        upscale_fac = self.frame_height/float(self.transmitted_frame_height)
        upscale_fac_lim = 2.25
        upscale_fac = min(upscale_fac_lim, max(1/upscale_fac_lim, upscale_fac))

        nr_pixels /= upscale_fac*upscale_fac

    b_s_unscaled = b_s
    b_s = b_s * (480*852) / nr_pixels

    self.bitrate_per_scene = b_s

    cpx_s = average_per_scene(self.cplx_intra_avg, scene_start, scene_end, dt)
    cpx_inter_s = average_per_scene(self.cplx_inter_avg, scene_start, scene_end, dt)
    self.cpx_intra_s = cpx_s
    self.cpx_inter_s = cpx_inter_s

    # avoid division by 0 for exceptional cases, i.e. freezing of a whole scene
    # of either first scene (first frame complexity is not computed) or first
    # frame of scene is homogeneous
    acc = 0.01
    I_cpx = [i for i in range(len(cpx_s)) if cpx_s[i]<acc and f_s[i]<acc]

    if len(I_cpx)>0:
        cpx_s[I_cpx] = 1.0

    c = 0.5

    v_s = ((1-c)+c*(1-p_s)*cpx_s)

    bbar_s = b_s * dt_s / (cpx_s+nr_frame_s*f_s* v_s)**0.95

    self.bbar_s = bbar_s
    self.cpx_s = cpx_s

    jerk_s = ustat.trimmed_mean(self.jerk, (0.25, 0.25)) # trimmed mean jerkiness
    q_cod_s = strans.STransform((0.3, 0.3, 1.0)).map(bbar_s/100.0)

    # adjust visibility: distortions in high motion scenes are less visible
    self.motion_mask_fac = self.motion_masking()

```

```

q_cod_s = 1-(1-q_cod_s)*self.motion_mask_fac
self.q_cod_s = q_cod_s

dt_s = sum_per_scene(self.dt,self.scene)
q_cod = mean_over_scene(q_cod_s,dt_s)

# adjust for viewing conditions:
# 1. adjustment for better degradation visibility (artifacts are larger)
if upscale_fac>1.0:
    q_cod *= q_cod**(upscale_fac*1.5)
# 2. adjustments for blurriness
q_cod *= self.get_view_condition_adjustment()

self.q_cod = q_cod

def dt_with_cutted_prepost_roll(self):

    dt = self.dt.copy()
    if self.frame_end>0:
        dt[self.frame_end+1:] = 0.0001
    if self.frame_start>0:
        dt[:self.frame_start] = 0.0001

    return dt

def get_conf_motion_estim(self,frac_motion,conf_motion):
    e_vec = array([.5,1])
    e_vec /= sqrt(sum(e_vec*e_vec))
    z = (e_vec[0]*frac_motion + e_vec[1]*conf_motion)/sqrt(2)
    return z

def get_viewing_distance2height(self):

    vd_fac = 3
    if self.frame_height<500:
        vd_fac = 5

    return vd_fac

def get_view_condition_adjustment(self):

    vertical_resolution = self.transmitted_frame_height

    # if transmitted frame height is not known:
    if vertical_resolution==0:
        vertical_resolution = self.frame_height

    # viewing_distance, in multiples of display height
    viewing_distance = self.get_viewing_distance2height()

    return self.get_viewing_condition_factor(vertical_resolution,viewing_distance)

def get_viewing_condition_factor(self,vertical_resolution,viewing_distance):
    """
    INPUT:
    vertical_resolution -- float, vertical resolution (transmitted)
    viewing_distance -- float, multiples of display height (use viewing_distance=5 for WVGA and
3 for HD)

    OUTPUT:
    max_score -- float in [0,1], max score of a video having 'vertical_resolution' under
    given viewing conditions.
    """

    viewing_angle = arctan(0.5/viewing_distance)*2
    pix_per_arc = vertical_resolution/viewing_angle
    s = strans.STransform([1,0.5,0.4])
    max_score = s.map(pix_per_arc/1000.0)

    return max_score

```

Observation: degradations in high motion areas are less visible. Therefore, determine fraction of blocks of high motion, and their average displacement. Using these values, determine a correction factor.

```
def motion_masking(self):
    """
    OUTPUT:
    fac    -- float, motion masking correction factor
    """
    s_frac = strans.STransform([0.1,0.3,6])
    s_motion = strans.STransform([0.2,0.1,4])

    scene_start = self.scene[0]
    scene_end = self.scene[1]

    dt = dt_no_rep(self.rep_frame,self.dt)
    f_dx_s = average_per_scene(self.frac_motion_dx,scene_start,scene_end,dt)
    f_dy_s = average_per_scene(self.frac_motion_dy,scene_start,scene_end,dt)
    m_dx_s = average_per_scene(self.motion_dx,scene_start,scene_end,dt)
    m_dy_s = average_per_scene(self.motion_dy,scene_start,scene_end,dt)

    fac_dx = s_frac.map(f_dx_s) * s_motion.map(m_dx_s)
    fac_dy = s_frac.map(f_dy_s) * s_motion.map(m_dy_s)

    c = get_hybrid_parameters()['max_motion_masking']
    return 1.0-c*maximum(fac_dx,fac_dy)
```

Transmission quality estimates the perceived degradations due to transmission errors, such as packet loss. It is mainly based on the estimation of effective loss derived from packet header information, together with jerkiness estimation, derived from the video frames.

```
def transmission_quality(self):
    N_b = len(self.dt_b)

    Eloss = stepf.StepFunc(self.dt_b,self. eff_loss_b)
    eff_loss = stepf.resample_step_func(Eloss,self.dt).y

    Eloss_key = stepf.StepFunc(self.dt_b,self. eff_loss_key_b)
    eff_loss_key = stepf.resample_step_func(Eloss_key,self.dt).y

    c_s = 0.5
    p = 1.0
    # slicing degradation
    slicing = (1.0 - exp(-eff_loss*p)) * (c_s + (1-c_s)*self.frac_motion)

    # add slicing at key frames (no motion correction term for slicing in key frames)
    q_slicing = (1.0-slicing)*exp(-eff_loss_key*p)

    self.q_slicing = q_slicing

    q_loss = (1-self.jerk)*q_slicing

    dt = self.dt_with_cutted_prepost_roll()

    q_loss = 1.0 - deg_temp_fade_out(1.0-q_loss,dt)
    self.q_loss = q_loss

    q_trans = sum(q_loss*dt/sum(dt))

    self.q_trans = self.q_trans

def mean_over_scene(x,duration):
    """
    Calculate feature mean over scenes

    INPUT
    x -- array (1-D), feature values per sample and per scene
    duration -- array (1-D), duration of scenes

    OUTPUT
    ms -- float, mean of x per scene
```

```

"""
xT = x
mx = sum(xT*duration)/sum(duration)
return mx

def non_rep_frame_index(rep_frame):
    return where(rep_frame<get_hybrid_parameters()['unrepeated_frame_limit'])[0]

def dt_no_rep(rep_frame,dt):
    """
    Compute a display time, such that repeated frames have display time equal to 0, and the
    non-repeated frames have display time as the sum of all subsequent repetitions.
    """
    # index of non-repeated frames
    I = non_rep_frame_index(rep_frame)

    # index of next non-repeated frames
    # e.g. for 5 frames with non-repeated frames [0,2,4] there is:
    # I = [0,2,4], J = [2,4,5]
    J = array(list(I[1:])+[len(rep_frame)])

    dt_no_rep = zeros(len(dt))
    dt_no_rep[I] = array([sum(dt[i:j]) for i,j in zip(I,J)])

    return dt_no_rep

def average_per_scene(x,scene_start,scene_end,dt):
    """
    Compute the mean feature value of x per scene over un-repeated frames.

    INPUT:
    x          -- 1-D ndarray of feature values
    scene_start -- list of scene starts
    scene_end  -- list of scene ends
    dt         -- 1-D ndarray of display time

    OUTPUT:
    am         -- array with len(am)=len(scene_start), of mean values
    """
    m = []

    for start,end in zip(scene_start,scene_end):

        x_s = x[start:end+1]
        dt_s = dt[start:end+1]

        m.append(sum(x_s*dt_s)/sum(dt_s))

    return array(m)

def correct_large_frames(df,len_content,verbose=False):
    """
    If the PES packet size is limited, a video frame can be in more than one
    PES packet. Check if frames are within the limit of PES packet size, and check
    if more frames in bitstream are present than in content, and thus, join frames
    together.
    """
    # check for columns in df:
    # if columns changed: do nothing:
    if not df.col_name == ['frame.timestamp_min', 'frame.timestamp_max', 'video_frame_size',
'packet_count', 'lost']:
        if verbose:
            print('correct_large_frames : no correction due to non-consistent col-names')
        return df

    pes_limit_min = 65500
    pes_limit_max = 65750

    nr_diff = len(df)-len_content

```

```

if nr_diff>0:

    f_size = df.col('video_frame_size')

    I_larger_min = where(f_size>pes_limit_min)[0]
    I_smaller_max = where(f_size<pes_limit_max)[0]

    I_very_large = where(f_size>=pes_limit_max)[0]

    I_large = intersect1d(I_larger_min,I_smaller_max)

    # only make adjustments, if some large frames within PES limit
    # are detected, but no frame is substantially larger (which
    # indicates that PES packet size is not limited)
    if len(I_large)>0 and len(I_very_large)==0:

        if verbose:
            print('correct frames with index: ',I_large)
            print('but correct max. ',nr_diff,' frames')

        # join the first nr_diff many frames
        I_large = I_large[:nr_diff]

        data_cor = []

        for i in arange(len(df)):

            data_row = df.data[i,:].copy()

            if i-1 in I_large:
                # modify last data_row
                last_row = data_cor[-1]
                last_row[1] = data_row[1]
                last_row[2] += data_row[2]
                last_row[3] += data_row[3]

            else:
                data_cor.append(data_row)

        df_cor = dfr.DataFrame(array(data_cor),df.col_name)

        return df_cor

return df

def scale_frame_size(fs):
    """
    Convert from bytes to kbits
    """
    return fs/1000.0*8

def get_bitrate_per_scene(frame_size_b,key_frame_b,scene_start,scene_end,duration,verbose=False):
    """
    compute the kbit/s of data (from bitstream) per scene and return it in a list.
    INPUT:
    frame_size_b -- arrays (1-D), containing bitstream frame size
    key_frame_b -- array (1-D) of indices of key frames
    scene_start, scene_end -- arrays (1-D) of same length containing scene start/end
    content frame numbers.
    scene_duration -- arrays (1-D) of same length containing duration of scenes (in s)

    OUTPUT:
    (b,fsize_eff_b) -- pair of array (1-D), len(b)== number scenes, average bitrate per scene and
    array (1-D), frame size, with additional size of 'redundant' key frames
removed
    """
    b = []

    scene_start_bstr,scene_end_bstr =
align_bitstream_content(frame_size_b,scene_start,scene_end,duration)

    # correct frame size by removing size of key frames, which are not at scene changes, as these
    # are for robustness only:
    fsize_eff_b =
get_frame_size_non_redundant(frame_size_b,key_frame_b,scene_start_bstr,scene_end_bstr)

    if verbose:
        print('scene_start=',scene_start)

```

```

        print('scene_start_bstr=',scene_start_bstr)

for i in range(len(scene_start_bstr)):
    start = scene_start_bstr[i]
    end = scene_end_bstr[i]

    # frame size of frames in this scene:
    fs = fsize_eff_b[start:end+1]

    #
    bitrate = 0

    if duration[i]>0:
        bitrate = sum(fs) /duration[i]

    b.append(bitrate)

if verbose:
    print('bitrate per scene=',b)

return (array(b),fsize_eff_b)

def sum_of_vec_per_scene(x,scene):
    """
    Compute the mean per scene.

    INPUT:
    x -- numpy array of floats
    scene_start_end -- pair of arrays containing scene start, end

    OUTPUT:
    x_sum -- array, mean per scene
    """

    x_sum = [sum(x[start:end+1]) for start,end in zip(scene[0],scene[1])]

    return array(x_sum)

def sum_per_scene(x,scene):
    if isinstance(x[0],ndarray):
        x_sum = []

        for i in range(len(x)):

            x_sum_i = sum_of_vec_per_scene(x[i],scene[i])
            x_sum.append(x_sum_i)

        return array(x_sum)
    else:
        return sum_of_vec_per_scene(x,scene)

def get_scene_align_range(n,scene_start,seq_length,delta):

    s = scene_start[n] - delta
    e = scene_start[n] + delta

    s = max(0,s)
    e = min(seq_length-1,e)

    if n>0:
        d = scene_start[n] - scene_start[n-1]
        s = max(scene_start[n-1]+d/2,s)

    if n<len(scene_start)-1:
        d = scene_start[n+1] - scene_start[n]
        e = min(scene_start[n]+d/2,e)

    return range(s,e+1)

def check_transmission_rate(t,t_b):
    """
    Check, if the stream is streamed in real-time, or faster (can be used in simulations)

    INPUT:

```

```

t      -- ndarray (1-D), video content frame time stamp
t_b    -- ndarray (1-D), bitstream frame time stamp (max per video frame)

OUTPUT:
t_scale -- float, time rescale factor, to rescale t_b to the scale of t.
"""

t_scale = mean(t)/mean(t_b-min(t_b))

return t_scale

def get_index_of_frac(frac,t):
    """
    frac -- float in [0,1)
    t     -- ndarray (1-D) of monotone increasing values

    i     -- index, such that t[i]/t[-1]<=frac<t[i+1]/t[-1]
    """
    i = 0
    frac = min(1.0,max(0.0,frac))

    for t_i in t[1:]:
        if t_i/t[-1]>frac or t_i/t[-1]==1.0:
            break
        i += 1

    return i

def correct_sent_timestamp(t_sent,rebuf_start):
    """
    Adjust sent timestamp, such that it is monotonically increasing, and starts
    at 0.

    INPUT:
    t_sent      -- ndarray (1-D), time stamp
    rebuf_start -- ndarray (1-D), indicator for rebuffering start

    OUTPUT:
    t_sent_corr -- ndarray (1-D), time stamp
    """

    t_sent_corr = t_sent - min(t_sent)

    I_rebuf_start = where(rebuf_start>0.5)[0]

    # if rebuffering
    if len(I_rebuf_start)>0:
        I_rebuf_start = [0]+list(I_rebuf_start)+[len(rebuf_start)]

        # loop over rebuf start positions (including position 0,len(rebuf_start))
        for i,j in zip(I_rebuf_start[:-1],I_rebuf_start[1:]):
            t_sent_corr[i:j] -= min(t_sent[i:j])
            if i>0:
                t_sent_corr[i:j] += t_sent[i-1]

    return t_sent_corr

def correct_content_duration(dt,rep_frame,frame_t_b,loss_b,verbose=False):
    """
    If duration of video < duration of bitstream:
    Correct video time stamps, by introducing freezing at 'skip positions'.
    Skip positions are estimated by loss positions in bitstream:

    |-----x-----xx-----|      bitstream duration, x : loss positions

    |-----|                        video duration

    |-----|||-----|||-----|    corrected video duration, by introduction
of |||
        >|..|< correction at first loss
            |..| delta due to first loss
            >|.....|< correction for second loss

```

```

Note: sum_of_corrections(||||..) = duration_bitstream - duration_video
"""
# compute expected number of frames:
nr_frames_b = len(loss_b)

# count frames in video, but removing repeated frames due to frame rate reduction
dt_nr = dt_no_rep(rep_frame,dt)
dt_m = ustat.trimmed_mean(dt_nr[dt_nr>0.0],(0.1,0.1))

nr_frames = len(dt)
# better estimate, i.e. skipping repeated frames due to frame rate reduction:
if dt_m > 0.01:
    nr_frames = sum(dt)/dt_m

missing_frames = nr_frames_b - nr_frames

dt_cor = dt
dt_b = ones(nr_frames_b)*sum(dt)/float(nr_frames_b)

if missing_frames>3:

    # estimate fps of content, and estimate duration of video as
    # number of frames in bitstream * estimated_fps:
    duration = sum(dt)

    duration_b = ustat.trimmed_mean(dt,(0.3,0.3))*nr_frames_b

    # duration_b > duration-epsilon
    # otherwise, it might be that it was streamed faster?
    if duration/duration_b>1.1:
        if verbose:
            print('duration=%1.2f,duration_b=%1.2f --PVS longer than bitstream' %
(duration,duration_b))

        duration_b *= max(1.0,duration/duration_b)

    # due to skipping, bitstream duration can be longer than video duration:
    delta_duration = duration_b-duration
    # re-distribute difference delta_duration among lost packets
    t_cor = stepf.cumulative_t(dt)

    # make only correction if bitstream duration longer than content and loss occurred
    if delta_duration>0.0 and sum(loss_b)>0.5:

        # compute de_per_loss, i.e. delta of display time, which will be added at each
        # loss position
        d_per_loss = delta_duration/sum(loss_b)
        # compute corrected time stamps (putting freezing into skip positions)
        n_b = len(loss_b)

        # assume that up to first loss, frames in bitstream and frames in content
        # are aligned:

        # start at the first loss
        i_first_loss_b = n_b-1
        I_loss = where(loss_b>0.0)[0]

        if len(I_loss)>0:
            i_first_loss_b = I_loss[0]

        if i_first_loss_b<len(t_cor):

            for i,nr_loss in enumerate(loss_b):
                if nr_loss>0 and i>=i_first_loss_b:

                    frac = (i-i_first_loss_b) / float(n_b-i_first_loss_b)

                    j = get_index_of_frac(frac,t_cor[i_first_loss_b:])

                    t_cor[i_first_loss_b+j+1:] += nr_loss*d_per_loss

            dt_cor = t_cor[1:]-t_cor[:-1]
            dt_b = ones(nr_frames_b)*sum(dt_cor)/float(nr_frames_b)

    return (dt_cor,dt_b)

def align_bitstream_content(frame_size_b,scene_start,scene_end,scene_duration,verbose=False):

```

```

"""
Match scene changes to large frames in bitstream:

content: scene start

|.....|.....|.....

bitstream: frame size

|           |           |           |
|.....|.....|.....|.....|.....
-----> time

for all scenes n and i in [k-delta,k+delta]:

minimise  $f_j/f_i + c * \text{abs}(j - i)$ 

where  $f_i$  : size of bitstream frame i
      j : 'scene start prior' in bitstream of scene n
      i : position (of frames) in bitstream

at the moment, these will be optimised independently for each scene.

INPUT:
frame_size_b -- arrays (1-D), containing bitstream frame size
scene_start, scene_end -- arrays (1-D) of same length containing scene start/end
                    content frame numbers.
scene_duration -- arrays (1-D) of same length containing duration of scenes (in s)

OUTPUT:
scene_s -- array (1-D) of bitstream frame numbers of scene start (corresponding
                    to scene start in content

"""
# scene start in bitstream:
nr_frames_content = scene_end[-1] - scene_start[0]
nr_frames_bitstr = len(frame_size_b)

d = nr_frames_bitstr - nr_frames_content

duration = sum(scene_duration)

# init scene start in bitstream
scene_s_b = [int(nr_frames_bitstr*sum(scene_duration[:i])/duration) for i,si in
enumerate(scene_start)]
# get optimisation parameters
fps = nr_frames_content/duration
delta = int(get_hybrid_parameters()['bitstr_content_delta_time'] * fps + 0.5)
c = get_hybrid_parameters()['misalign_tradeoff'] /fps

for n in range(len(scene_s_b)):

    j_range = get_scene_align_range(n,scene_s_b,nr_frames_bitstr,delta)

    z = zeros(len(j_range))

    i = scene_s_b[n]

    for k,j in enumerate(j_range):

        z[k] = c*abs(i-j) + frame_size_b[i]/frame_size_b[j]

    scene_s_b[n] = j_range[z.argmin()]

    if verbose:
        print('scene[n]=%d' % i)
        print('j_range=',j_range)
        print('frame_size_b=',frame_size_b[j_range])
        print('z=',z)

    scene_start_bitstr = scene_s_b
    scene_end_bitstr = [nr_frames_bitstr-1]
    if len(scene_start_bitstr)>1:
        scene_end_bitstr = list(array(scene_s_b[1:])-1) + scene_end_bitstr

return (array(scene_start_bitstr),array(scene_end_bitstr))

```

```

def get_key_frame_in_content(dt,dt_b,key_frame_b):
    """
    Map the estimated key frames from the bitstream time scale to the content
    time scale

    INPUT:
    dt -- ndarray (1-D), content display time
    dt_b -- ndarray (1-D), bitstream display time
    key_frame_b -- ndarray (1-D), key frames in bitstream

    OUTPUT:
    key_frame -- ndarray (1-D), key frames in content
    """

    kf_b = stepf.StepFunc(dt_b,key_frame_b)
    key_frame = stepf.resample_step_func(kf_b,dt).y

    return key_frame

def scale_peak_fsize(fsize):
    """
    Filter to enhance peaks.

    fsize -- array (1-D), bitstream frame size
    """
    # use asymmetric filter: put more weight on right side, as by scene changes
    # frame size of right side determines statistics of current scene
    filt = array([-0.5,-1,-1.5,-2,15,-4,-3,-2,-1],dtype=float)
    filt = filt/sqrt(sum(filt*filt))
    filt = flipud(filt)

    return maximum(0.0,convolve(fsize,filt,'same'))

def smooth_with_gaussian(x):
    """
    Smooth vector x using gaussian smoothing filter
    """
    if len(x)>=3:

        n = min(len(x)/3,10)
        filt = exp(-(arange(n+1)**2/(n*n/2.0))
        filt = concatenate((flipud(filt[1:]),filt))
        filt = filt/sum(filt)

        y = convolve(x,filt,'valid')
        z = concatenate((ones(n)*y[0],y,ones(n)*y[-1]))
    else:
        print(mean(x))

        z = ones(len(x))*mean(x)

    return z

def get_sliding_window_indices(vector_length,window_size):
    """
    Determine start and end indices of sliding window. The window size is constant,
    and is centered in the middle, and it is on the right/left at the beginning/end.

    OUTPUT:
    (I_start,I_end) -- pair of 1-D numpy arrays with start,end index
    """
    d = min(vector_length/2,window_size/2)

    I_start = maximum(0,arange(vector_length)-d)
    I_end = minimum(vector_length,I_start+2*d+1)
    I_start = I_end-2*d

    return (I_start,I_end)

def kbit_per_I_frame(fsize_b,frac_motion,conf_motion,duration):
    """
    Estimate I-frame size using a simplified formula, similar in spirit to the

```

one used for coding quality estimation.
 Use features within a sliding window of temporal size 'sliding_window_temp_size'.

```

INPUT:
fsize_b -- vector (numpy array), bistream frame size in kbit
frac_motion -- vector (numpy array),
conf_motion -- vector (numpy array),
duration -- total video duration in seconds

OUTPUT:
bbar      -- float, estimate of I frame size
"""

# sliding window size in sec:
sliding_window_temp_size = 4.0

N = len(frac_motion)
N_b = len(fsize_b)

# determine window size
n = N * sliding_window_temp_size / duration
n_b = N_b * sliding_window_temp_size / duration

start,end = get_sliding_window_indices(N,n)
start_b,end_b = get_sliding_window_indices(N_b,n_b)

frac_motion_m = array([mean(frac_motion[i:j]) for i,j in zip(start,end)])
conf_motion_m = array([mean(conf_motion[i:j]) for i,j in zip(start,end)])

# resample to length of bitstream:
FF = stepf.StepFunc(ones(N)*duration/N,frac_motion_m)
frac_motion_m_b = stepf.resample_step_func(FF,ones(N_b)*duration/N_b).y

FC = stepf.StepFunc(ones(N)*duration/N,conf_motion_m)
conf_motion_m_b = stepf.resample_step_func(FC,ones(N_b)*duration/N_b).y

# compute mean frame size (non-redundant) over sliding window):
fsize_b_m = []
tm_all = []
for i,j in zip(start_b,end_b):

    # compute mean non-redundant frame size: mean of frame_size max (an I-frame) and
    # trimmed mean of frame size, ignoring the largest frames (redundant I-frames):
    tm = ustat.trimmed_mean(fsize_b[i:j],[0.0,1.0/25.0])

    tm_all.append(tm)

    fsize_b_m.append( max(fsize_b[i:j]) + (n_b-1)*tm )

fsize_b_m = array(fsize_b_m)

# estimate I-frame size using formula similar to coding quality (simplified)
c = 0.5
bbar = fsize_b_m / (1+n_b*frac_motion_m_b*((1-c)+c*(1-mean(conf_motion_m_b))))
return bbar

```

```

def next_element_larger(i,J):
    """
    Return the first element in J, that is larger than i.
    """
    J_larger = [j for j in J if j>i]
    return J_larger[0]

```

```

def convert_to_prob_partial_lin(x,p0,p1):
    """
    Convert x to values in [0,1] using partial linear function, such that
    x<=p0 : y = 1
    x>=p1 : y = 0
    p0<x<p1 : linear
    """
    acc = 1.0e-4
    y = zeros(len(x))
    I = where((p1-p0)>acc)

    if len(I)>0:
        y[I] = (p1[I]-x[I])/(p1[I]-p0[I])

    y[x<=p0] = 1

```

```

y[x>=pl] = 0

return y

def get_prob_of_B_frame(frame_size_b,P_size_b,B_size_b):
    """
    Compute for each frame a probability of being NOT a B-frame.

    INPUT:
    frame_size_b -- ndarray (1-D), bitstream frame size
    P_size_b -- ndarray (1-D), local mean of larger frames
    B_size_b -- ndarray (1-D), local mean of small frames

    OUTPUT:
    prob_b -- ndarray (1-D), prob of being b frame
    """
    n = len(P_size_b)
    prob_b = zeros(n)
    I = where(B_size_b < P_size_b/2.0)

    if len(I)>0:
        PB_mean = (P_size_b+B_size_b)/2.0
        prob_b[I] = convert_to_prob_partial_lin(frame_size_b[I],B_size_b[I],PB_mean[I])

    return prob_b

def compute_local_trimmed_mean(fsize_b,q_min,q_max>window_size=10):
    """
    Compute local trimmed mean

    INPUT:
    fsize_b -- ndarray (1-D)

    OUTPUT:
    fs_low -- ndarray (1-D), local trimmed mean
    """
    N = len(fsize_b)
    bf = zeros(N)

    # use sliding window to determine trimmed mean
    d = min(N/2>window_size/2)
    I_start = maximum(0,arange(N)-d)
    I_end = minimum(N,I_start+2*d+1)
    I_start = I_end-2*d

    # compute trimmed mean in window [i:j]
    fs_low = array([ ustat.trimmed_mean(fsize_b[i:j],(q_min,q_max)) for i,j in zip(I_start,I_end) ])

    return fs_low

def adjust_frame_type_at_scene_change(frame_size_b,loss_b,scene,dt,prob_key_frame_b):
    """
    Adjust frame type estimation (performed using mainly bitstream), using content scene changes.

    INPUT:
    fsize_b          -- vector(numpy array), frame size in bitstream
    loss_b           -- vector(numpy array), number of lost packets per frame
    scene            -- pair of vector(numpy array), scene start/scene end
    dt               -- vector(numpy array), display time
    prob_key_frame_b -- vector(numpy array), probability of being a key frame

    OUTPUT:
    prob_key_frame_b -- vector(numpy array), adjusted prob of key frame
    """
    prob_key_adj_b = prob_key_frame_b.copy()

    scene_start,scene_end = scene
    # duration per scene
    duration_s = sum_per_scene(dt,scene)

    scene_start_bstr,scene_end_bstr =
align_bitstream_content(frame_size_b,scene_start,scene_end,duration_s)

    # for scene start, if there is no loss (strong loss can make false scene start estimation

```

```

for s in scene_start_bstr:
    if loss_b[s]==0:
        # estimated scene start is at index s (in bitstream)
        # thus, prob of key frame at s should be equal to 1
        # is prob of key frame at s very low, maybe something with the alignment
        # went wrong, therefore, do not change too much
        # i.e. linear from [0,0.1],
        prob_key_adj_b[s] = min(10*prob_key_frame_b[s],1.0)

return prob_key_adj_b

def detect_regular_key_frames(frame_size_b,prob_key_frame_b,verbose=False):
    """
    INPUT:
    fsize_b -- vector(numpy array), frame size in bitstream
    scene_start_b -- vector(numpy array), scene start indices in bitstream
    prob_key_frame_b -- vector(numpy array), probability of being a key frame

    OUTPUT:
    prob_key_frame_b,scene_start_bitstr) -- vector(numpy array), adjusted prob of key frame
    """
    prob_key_adj_b = prob_key_frame_b.copy()

    P_size = compute_local_trimmed_mean(frame_size_b,q_min=0.6,q_max=0.1>window_size=12)

    peak_index = where(frame_size_b > 2*P_size)[0]

    if verbose:
        print('peak_index=',peak_index)

    if len(peak_index) > 1:
        inter_peak_dist = peak_index[1:] - peak_index[:-1]

        max_key_dist = ustat.trimmed_mean(inter_peak_dist,[.2,.2])

        # count how many times inter peak distance is close to estimated max_key_dist
        sum_close = sum([1 for d in inter_peak_dist if abs(d-max_key_dist)<2])

        if verbose:
            print('max_key_dist=',max_key_dist)
            print('sum_close=',sum_close)
            print('len(inter_peak_dist)',len(inter_peak_dist))

        # only make adjustment, if many peaks are around 'max_key_dist'
        if sum_close > len(inter_peak_dist)/2:

            # if estimated key frames have a realistic value
            if max_key_dist > 5:

                key_index = array([pind for i,pind in enumerate(peak_index[1:]) if peak_index[i]-
                peak_index[i-1] >= (max_key_dist-1) ])

                if len(key_index)>0:

                    prob_key_adj_b[key_index] += 1.0
                    prob_key_adj_b = minimum(1.0,prob_key_adj_b)

    return prob_key_adj_b

def get_frame_type_bstr(fsize_b,frac_motion,conf_motion,duration,verbose=False):
    """
    INPUT:
    fsize_b -- vector(numpy array),
    frac_motion -- vector(numpy array),
    conf_motion -- vector(numpy array),
    duration -- float,total video duration in seconds
    verbose -- bool
    """
    # number frames in bitstream
    N = len(fsize_b)
    dt = duration/float(N)

    # rough content-dependent estimation of I-frame size
    I_size_content = kbit_per_I_frame(fsize_b,frac_motion,conf_motion,duration)
    # in case the estimation is completely wrong, limit estimation by largest frame

```

```

I_size_content = minimum(max(fsize_b),I_size_content)

P_size = compute_local_trimmed_mean(fsize_b,q_min=0.6,q_max=0.1>window_size=12)
B_size = compute_local_trimmed_mean(fsize_b,q_min=0.0,q_max=0.6>window_size=5)

# compute relative frame size, i.e. subtract frame size of small frames in neighbourhood
rel_fsize = maximum(fsize_b - B_size,0.0) #scale_peak_fsize(fsize_b)
# compute relative I_size_content:
I_size_content -= mean(B_size)
I_size_content = minimum(max(rel_fsize),I_size_content)

I_size_limit = zeros(len(rel_fsize))

prob_b_frame = get_prob_of_B_frame(fsize_b,P_size,B_size)

I_size_min_limit = I_size_content/4.0
I_size_med_limit = I_size_content*3/4.0

# define lower limit (on the relative scale) of I-frames by:
low_lim = maximum(I_size_min_limit,minimum( I_size_med_limit, (2*(P_size-
B_size)+I_size_med_limit)/3.0 ))

I_size_limit[0] = max(I_size_min_limit[0],rel_fsize[0])

for i in range(1,len(fsize_b)):

    # if last frame was above limit, take its size
    I_size_limit[i] = max(rel_fsize[i-1],I_size_limit[i-1])

    # exponential decay of size limit
    # constant chosen, such that in 1 second limit drops
    # to about 0.5 of initial value
    I_size_limit[i] *= exp(-dt/1.44)

    # avoid decay to value too small, by imposing to be above low_lim
    I_size_limit[i] = max(low_lim[i],I_size_limit[i])

prob_key_frame = zeros(N)
prob_key_frame = minimum(1.0,maximum(0.0,rel_fsize-I_size_limit)/low_lim)
prob_key_frame[0] = 1.0

# make adjustment in case of regular key frames
prob_key_frame = detect_regular_key_frames(fsize_b,prob_key_frame)

return
(prob_key_frame,prob_b_frame,rel_fsize,I_size_limit,I_size_content,I_size_min_limit,I_size_med_limit
,P_size,B_size)

def get_frame_size_non_redundant(fsize_b,key_frame_b,scene_start_b,scene_end_b):
    """
    Estimate frame sizes excluding redundant key frames.

    INPUT:
    fsize_b -- vector(numpy array),
    key_frame_b -- array, indices of key frames
    scene_start_b -- array, indices of scene starts
    scene_end_b -- array, same length as scene_start_b, indices of scene ends

    OUTPUT:
    fsize_eff_b -- vector, equal to fsize_b, except at positions of redundant key frames.
    """
    fsize_eff_b = fsize_b.copy()

    for s,e in zip(scene_start_b,scene_end_b):
        # determine key frames in scene [s,e]
        kf_scene = [i for i in key_frame_b if i>=s and i<=e]
        # list of redundant key frames are all key frames per scene except the first
        kf_red = kf_scene[1:]

        if len(kf_red)>0:

            for k in kf_red:

                # indices of neighbouring non-key frames
                I = [i for i in range(k-4,k+4) if i>0 and i<len(fsize_b) and (not i in key_frame_b)]

                if len(I)>0:
                    fsize_eff_b[k] = max(fsize_b[I])

```

```

return fsize_eff_b

def
get_eff_loss_bstr(loss_count_b,prob_key_frame_b,prob_b_frame_b,frame_size_b,packet_count_b,is_mp2t,v
erbose=False):
    """
    INPUT:
    loss_b -- vector(numpy array),
    prob_key_frame_b -- vector(numpy array),
    prob_b_frame_b -- vector(numpy array),
    verbose -- bool

    OUTPUT:
    eff_loss -- vector(numpy array), estimated effective loss
    eff_loss_key -- vector(numpy array), estimated effective loss, that occurs in key frames

    """
    # number frames in bitstream
    N = len(loss_count_b)

    loss_b = loss_count_b

    eff_loss = zeros(N)
    eff_loss_key = zeros(N)

    # add additional probability of 'hypothetical' key frame at end, used for later processing:
    prob_key_frame_b = array(list(prob_key_frame_b)+[1.0])

    prob_nonB = 1.0 - prob_b_frame_b
    delta_B = get_hybrid_parameters()['max_BFrame_interval']

    for i in range(N):

        if loss_b[i]>0:

            # check if loss is likely to occur in B-frame, then loss is likely stop at next P-frame:
            next_nonB = i
            p = 0.0

            if prob_b_frame_b[i] > 0.001:

                next_nonB = i+1
                p = 1
                if len(prob_nonB[i+1:i+delta_B])>0:
                    next_nonB = argmax(prob_nonB[i+1:i+delta_B]) + (i+1)

                # p is the probability of frame next_nonB is not a B-frame:
                p = prob_nonB[next_nonB]

                eff_loss[i:next_nonB] += prob_b_frame_b[i] * p*loss_b[i]

                if verbose:
                    print('loss at b-frame %d up to %d with prob %1.2f' %
(i,next_nonB,prob_b_frame_b[i]*p))

            # otherwise, loss is likely to propagate until next I-frame:
            prob_next_key_frame = 1.0-prob_b_frame_b[i]

            # add a hypothetical key frame at end, for easier processing in loop
            I_key_f = where(prob_key_frame_b[i+1:]>0.01)[0] + (i+1)

            if verbose:
                print(I_key_f)

            # loop over all possible next key frames and add loss
            for k in I_key_f:

                if prob_next_key_frame < 0.01:
                    break

                # p is prob that loss propagates from frame i to frame k
                p = prob_next_key_frame * prob_key_frame_b[k]

                # if loss starts in key frame
                if prob_key_frame_b[i]>0.5:
                    eff_loss_key[i:k] += p * loss_b[i]

                else:
                    eff_loss[i:k] += p * loss_b[i]

```

```

        prob_next_key_frame *= (1-prob_key_frame_b[k])

    return (eff_loss,eff_loss_key)

def deg_temp_fade_out(v,dt):
    """
    Calculate a temporal (perceptually motivated) degradation fade out and smoothing.
    Such that:
    * the relative impact of a degradation (local in time) is smaller if it
      occurs after a previous degradation
    * degradations occurring for very short time are attenuated (due to a
      temporal smoothing

    INPUT:
    v -- numpy array (vector) of per frame degradations
    dt -- numpy array (vector) display time (in s)

    OUTPUT:
    w -- array (vector), of 'faded out' degradation values

    """

    w = zeros(len(v))

    t_const = get_hybrid_parameters()['deg_temp_smooth_dt']
    par_dt = get_hybrid_parameters()['deg_temp_fade_out']

    fv = stepf.StepFunc(dt,v)

    for i in arange(1,len(v)):

        a = exp(-dt[i-1]/par_dt)

        # compute the average of v over the last t_const seconds:
        v_avg = stepf.eval_step_func(fv, (fv.t[i+1]-t_const, fv.t[i+1]))

        # note: w[i] depends on v[i] only through the average v_avg.
        # therefore, very short very strong degradations are attenuated.
        w[i] = max(v_avg,a * w[i-1] + (1-a) * v_avg)

    return w

def check_delay_in_packet_arrival(frame_t_b,frame_t_sent_b,video_dt):
    """
    Check for delays in packet arrival, which could result in
    discarding the packets by the video player, or which could
    result in rebuffering.

    INPUT:
    frame_t_b -- bitstream time stamp of arrival of last packet of a frame
                NOTE: reception time of the first packet of first frame is set to 0.
    frame_t_sent_b -- bitstream sent time stamp of last packet of a frame
                    NOTE: sent time of the first packet of the first frame is set to 0.
    video_dt -- display time of video frame (e.g. for a 25fps video this is
                a vector with entries 0.04)

    OUTPUT:
    delay -- 1-D numpy array, delay

    """

    delay = zeros(len(frame_t_b)-1)

    return delay

```

Link freezings to bitstream statistics, either to delay or to packet loss. Remove freezings (longer than those due to frame rate reduction) in case no degradation of the bitstream can be observed.

Overview of types of frame repetitions:

- low frame rate (min 8 fps, i.e., max 4 equal consecutive frames);
- repetition in source (still image, cartoon);
- freezing due to packet loss/delay.

Thus, frame repetitions larger than 4 frames are a degradation in case of packet loss/delay.

Short frame repetitions should be penalized if not present in bitstream.

```
def link_freezing_to_bitstream_stat(lost_b,delay_b,rebuf_b,dt_b,rep_frame,dt,jerk_weight_prior,verbose=F
else):
    """
    INPUT:

    lost_b    -- 1-D numpy array, lost frames
    delay_b   -- 1-D numpy array, lost frames
    rebuf_b   -- 1-D numpy array, lost frames
    dt_b      -- 1-D numpy array, bitstream dt, corrected, to match video content dt
    rep_frame -- 1-D numpy array, repeated frames
    dt        -- 1-D numpy array, frame display time
    jerk_weight_prior -- 1-D numpy array, prior values for jerkiness weight in [0,1], has
    len(rep_frame)+1

    OUTPUT:
    jerk_weight -- 1-D numpy array, weightings, used for jerkiness calculation
    """
    jerk_weight = jerk_weight_prior.copy()

    I_non_rep = non_rep_frame_index(rep_frame)
    I_non_rep = array(list(I_non_rep)+[len(rep_frame)])

    t = stepf.cumulative_t(dt)
    t_b = stepf.cumulative_t(dt_b)

    # loop over start,end of repetition intervals (note: max(j)==len(rep_frame))
    for i,j in zip(I_non_rep[:-1],I_non_rep[1:]):

        # determine indices (i_b,j_b) in bitstream corresponding to indices (i,j) in content
        i_b = argmin(array([abs(tk-t[i]) for tk in t_b]))
        j_b = argmin(array([abs(tk-t[j]) for tk in t_b]))

        j_lim = min(j,len(rep_frame)-1)
        j_b_lim = min(j_b,len(lost_b)-1)

        if verbose:
            if j-i>1:
                print('i=%d,j=%d,i_b=%d,j_b=%d' % (i,j,i_b,j_b))

        # in the error free case
        if sum(lost_b[i_b:j_b_lim])<0.001 and sum(delay_b[i_b:j_b_lim])<0.001 and sum(rebuf_b)<0.5:

            if verbose and j-i>1:
                print('no loss case')

            jerk_weight[i+1:j_lim+1] /= max(1,j_b-i_b)

        else:

            # saturate linearly, i.e. [0.100,0.200] --> [prior_value,1], delta --> weight
            delta = t[j]-t[i]

            if verbose and j-i>1:
                print('delta=',delta)
                print('jerk_weight[j_lim]=%1.3f' % jerk_weight[j_lim])

            if delta>=0.2:
                jerk_weight[i:j_lim+1] = 1.0

            elif delta>0.1:
                jerk_weight[i:j_lim+1] = jerk_weight[j_lim]+(1-jerk_weight[j])*(delta-0.1)/0.1

    return jerk_weight
```

Jerkiness

Compute perceived jerkiness, using the details explained below

```
def is_new_frame(frame_diff,frame_diff_prior):
    """
    Partial linear function taking values in [0,1], the probability of a new frame.
    """

    new_frame = zeros(len(frame_diff))
```

```

I1 = where(frame_diff>frame_diff_prior*3.0/2.0)[0]
I0 = where(frame_diff<=frame_diff_prior*3.0/2.0)[0]

if len(I1)>0:
    new_frame[I1] = 1.0

if len(I0)>0:
    new_frame[I0] = (frame_diff[I0]-frame_diff_prior[I0]/2.0)/frame_diff_prior[I0]
    new_frame = maximum(0.0,new_frame)

# the first frame is always a new frame
new_frame[0] = 1.0

return new_frame

def get_interframe_prior(frame_diff,display_time,time_const = 0.05,frame_diff_max=10.0):
    """
    Determine a frame-difference prior, used for detection of repeated frames.
    """
    N = len(frame_diff)

    acc = 0.001
    f_diff_prior = zeros(N)+acc

    for i in range(1,N):

        d = min(frame_diff[i-1],frame_diff_max)
        # divide by 4.0, such that for constant motion, the prior is at half the motion * exp(...)
        d_prior = (f_diff_prior[i-1]+d)/4.0
        f_diff_prior[i] = d_prior * exp(-display_time[i]/time_const)

    return f_diff_prior

def repeated_frame(frame_diff,display_time):
    """
    Probabilistic computation of repeated frames.
    """
    f_diff_prior = get_interframe_prior(frame_diff,display_time)
    new_frame = is_new_frame(frame_diff,f_diff_prior)

    return 1.0-new_frame

def get_motion_weighting(motion_intensity):
    """
    INPUT:
    motion_intensity      -- numpy array (1-D), global motion intensity, such that
                           motion_intensity[i] corresponds to motion from frame i-1 to i.

    OUTPUT:
    motion_weight         -- numpy array (1-D), has length(motion_intensity)+1, rescaled
    motion_intensity to [0,1]

    """
    # parameters for motion part
    p_mu = (2, 0.5, 0.5)
    mu = strans.STransform(p_mu)

    # add a motion intensity value to end
    motion_intensity = array(list(motion_intensity)+[motion_intensity.max()])

    motion_weight = mu.map(motion_intensity)

    return motion_weight

def jerkiness(display_time,rep_frame,jerk_motion_weight,viewing_distance2height=3,verbose=False):
    """
    Compute perceived jerkiness.

    INPUT:
    display_time          -- numpy array (1-D), in seconds(!)
    rep_frame             -- numpy array (1-D)
    jerk_motion_weight    -- numpy array (1-D), weighting, taking motion into account
    viewing_distance2height -- relation between viewing distance and display height of video
    """

```

```

(=3 for std. HD viewing settings)

OUTPUT:
jerkiness          -- numpy array (1-D). Note: add jerkiness to whole repetition
                    interval, to easier create per-frame quality value

"""

# viewing angle factor
c = 2.54 * 3.0/viewing_distance2height

# parameters for time part
p_tau = (0.12/c,0.05,1.5*c)

tau = strans.STransform(p_tau)

# skip computations with very low probability
p_low = 0.01

N = len(display_time)

jerk = zeros(N)

# add a no repetition value to end
rep_frame = array(list(rep_frame)+[0.0])
new_frame = 1-rep_frame

if verbose:
    print('new_frame=',new_frame)

#look for frame repetitions starting at position j
for j in range(N):
    if new_frame[j]>p_low:

        # look for frame repetition intervals (=~ display time) of length i
        for i in range(1,N-j+1):

            # if one of the repeated frames in the interval has a very low prob.
            # of being repeated, skip interval
            if i>1:
                if min(rep_frame[j+1:j+i])<p_low:
                    break

            # if at the end of the repetition interval, the probability of a new frame
            # is very low, skip computation
            if new_frame[i+j]<p_low:
                continue

            # compute probability of frame repetition interval
            prob = new_frame[j] * new_frame[j+i]

            if i>1:
                prob *= prod(rep_frame[j+1:j+i])

            if prob<p_low:
                continue

            # display time of frame repetition
            dt = sum(display_time[j:j+i])

            # jerk[j:j+i] += prob * dt * tau.map(dt) * mu.map(motion_intensity[j+i])
            jerk[j:j+i] += prob * tau.map(dt) * jerk_motion_weight[j+i]

return jerk

```

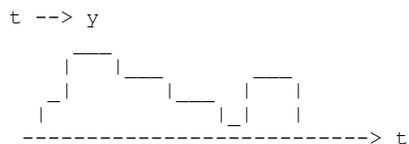
A.4 Additional information

For ease of presentation, this detailed description uses for the algorithmic parts the syntax of the python programming language. In addition, to avoid repeating the description of basic functions and containers, this description references those of (www.python.org) and the numerical python library (www.numpy.org).

In the remaining part of this clause, functions and containers are presented that are used in the model description above.

Step functions

Step functions are functions of the form



i.e., constant over an interval $dt = [t_0, t_1]$.

```
def cumulative_t(dt):
    """
    Compute cumulative t

    INPUT:
    dt -- vector (numpy array)

    OUTPUT:
    t -- vector with len(t)=len(dt)+1
    """
    t = [0]
    for d in dt:
        t.append(t[-1]+d)
    return array(t)

class StepFunc:
    """
    A step function is defined by a pair of vectors (dt,y) of interval length dt and step values y,
    or a pair of vectors (t,y) of interval start/end positions and step values, where
    len(t)=len(y)+1.

    """
    def __init__(self,dt,y):
        """
        INPUT:
        dt -- numpy array with len(dt)==len(y), of t intervals of constant values of step function.
        y -- numpy array of step values
        """
        self.dt = array(dt)
        self.t = self.cumulative_t(dt)
        self.y = array(y)

    def __len__(self):
        return len(self.y)

    def cumulative_t(self,dt):
        """
        Compute cumulative t

        INPUT:
        dt -- vector (numpy array)

        OUTPUT:
        t -- vector with len(t)=len(dt)+1
        """
        return cumulative_t(dt)

def eval_step_func(f_step,t_interval,verbose=False):
    """
    Evaluate step function given by (y,dt) at
    position [t0,t1].
    Make a zero-padding of f_step, if t0 or t1 are outside of f_step.t

    INPUT:
    f_step -- StepFunc
    t_interval -- pair of float, defining interval [t0,t1]

    OUTPUT:
    y -- value of the step function with sampling positions (t0,t1)
    """
```

```

"""
t0,t1 = t_interval

delta_t = t1-t0

t0 = max(t0,f_step.t[0])
t1 = min(t1,f_step.t[-1])

i = 0
t = f_step.t

while f_step.t[i+1]<=t0:
    i += 1
j = i
while f_step.t[j+1]<t1:
    j += 1

if verbose:
    print('t interval start %1.2f' % f_step.t[i])
    print('t interval stop %1.2f' % f_step.t[j])

y = 0
for k in range(i,j+1):
    dt = min(t1,f_step.t[k+1])-max(t0,f_step.t[k])
    y += dt/delta_t*f_step.y[k]

return y

def resample_step_func(f_step,dt_new):
    """
    Given a step function f_step=(dt,y) compute a new step function given by (dt_new,y_new)
    using the intervals of constant value given by dt_new.

    INPUT:
    f_step -- StepFunc
    dt_new -- numpy array (vector) with len(dt_new)==len(f_step)

    OUTPUT:
    g_step -- StepFunc with g_step.dt == dt_new
    """

    t_new = cumulative_t(dt_new)
    y_new = []

    for i in range(len(dt_new)):
        t0,t1 = t_new[i],t_new[i+1]
        y_new.append(eval_step_func(f_step, (t0,t1)))

    return StepFunc(dt_new,array(y_new))

```

Frame pyramid and frame sequence

Video sequences are stored in frame sequences, where each frame is kept in a multi-resolution pyramid. The details are described below:

```

class FramePyramid:
    """
    self.P -- a list of subsampled frames (numpy arrays)
    self.height -- a list of the frame height
    self.width -- a list of the frame width
    """

    def __init__(self,F,empty=False,skip_level=0):
        """
        construct the multi-scale pyramid starting
        from F.

        INPUT:
        F -- numpy array (matrix) of frame data
        empty -- boolean, if true, create a pyramid of zeros.
        skip_level -- positive int, the number of high-res level of the pyramid to skip

        """
        self.P = list()
        self.height = list()
        self.width = list()

        self.smooth_height_max = 60

```

```

count_level = 0

F_sub = F.astype(np.float32)
h,w = F_sub.shape

while min(h,w)>=10:

    if count_level >= skip_level:
        if empty:
            F_sub = np.zeros((h,w),dtype=np.float32)

            self.P.append(F_sub)
            self.height.append(h)
            self.width.append(w)

        if empty:
            h,w = ((h+1)/2, (w+1)/2)
        else:
            # smooth frame data before subsampling, but for low resolution
            # store the smoothed data (as this will be used for motion
            # estimation), i.e. smooth after subsampling.
            # In addition, for larger formats, do not smooth at the first iteration
            if h>self.smooth_height_max and h<F.shape[0] :
                F_sub = self.smooth(F_sub)

            F_sub = self.subsample(F_sub)

            h,w = F_sub.shape
            if h<=self.smooth_height_max:
                F_sub = self.smooth(F_sub)

        count_level += 1

    # make low resolution first
    self.height.reverse()
    self.width.reverse()
    self.P.reverse()

def __getitem__(self, key):
    return self.P[key]

def __setitem__(self, key, item):
    self.P[key] = item

def __len__(self):
    return len(self.P)

def get_dim(self):
    """
    return list of tuples, the height x width of the array
    at each level.
    """
    return [self.P[i].shape for i in range(len(self.P))]

def subsample(self, F):
    """
    Subsample frame F by 2x2.
    """
    return F[::2,::2].copy() # use copy!

def smooth(self, F):
    # Filter the array F using the filter
    # filt = np.array([1,2,1])/4.0
    # and using padding of the array borders.
    # Return the filtered array.

class FrameSeq:
    """
    self.Y -- list of frame pyramid
    self.Cb -- list of frame pyramid
    self.Cr -- list of frame pyramid
    self.display_time -- list of frame display times (in ms)
    self.name -- name of the sequence, if read from file, the filename

```

```

"""
def __len__(self):
    return len(self.Y)

def __iter__(self):
    return self

def next(self):
    if self.iter_index >= len(self.Y)-1:
        raise StopIteration
    self.iter_index += 1
    iter_index = self.iter_index
    return (self.Y[iter_index],self.Cb[iter_index],self.Cr[iter_index])

def common_pyramid_height(self):
    """
    Return the height of the pyramid of Cb,Cr (note, Y has one additional level)
    """
    return self.Cb[0].__len__()

def from_avi(self,fn,frame_start=0,frame_end=-1,frame_step=1,skip_level=0):
    """
    Read frames from avi-file into lists of FramePyramids

    INPUT:

    fn          -- filename
    frame_start -- start frame
    frame_end   -- end frame, if -1 read to end
    skip_level  -- positive int, the number of high-res level of the pyramid to skip

    OUTPUT:

    nb_frames -- int,number frames in file
    """
    # The implementation of this method is straight-forward, but may depend on
    # additional libraries to read the video frames. For each read video frame F
    # the method append_frame(F) is called.

def diff(self):
    """
    Compute frame difference between consecutive frames
    """
    for l in range(len(self.Y[0])):
        for i in reversed(range(1,len(self.Y))):
            self.Y[i][l] = 128.0+0.5*np.abs(self.Y[i][l]-self.Y[i-1][l])
            self.Y[0][l] = np.zeros(self.Y[0][l].shape,dtype=np.float32)

def append_frame(self,frame_YCbCr,display_time=40,skip_level=0):
    """
    Convert frame data in triple frame_YCbCr into
    FramePyramid's and append to lists.
    """
    self.Y.append(FramePyramid(frame_YCbCr[0],skip_level=skip_level))
    self.Cb.append(FramePyramid(frame_YCbCr[1],skip_level=skip_level))
    self.Cr.append(FramePyramid(frame_YCbCr[2],skip_level=skip_level))

    self.display_time.append(display_time)

```

Annex B

YHyNRe (Hybrid-NRe model)

(This annex forms an integral part of this Recommendation.)

B.1 Introduction

The YHyNR model first computes a video quality metrics (VQM) value using the total number of packets and number of packet loss using a pre-defined look-up table (LUT). Then, post-processing is applied to reflect the various impairments due to transmission errors.

B.2 Hybrid-NRe VQM computation

B.2.1 Feature computation

B.2.1.1 Total number of packets and number of packet loss

The total number of packets (*TotalPacket*) is computed as follows:

$$TotalPacket = \begin{cases} TS \text{ packet number} & \text{if TS protocol} \\ \frac{1}{180} RTP \text{ PayloadSize} & \text{otherwise} \end{cases}$$

The number of packet loss (*TotalPacket_{loss}*) is computed as follows:

$$TotalPacket_{Loss} = \begin{cases} TS \text{ loss packet number} & \text{if TS protocol} \\ \frac{1}{180} (RTP \text{ loss packet number} \times \text{Average RTP: packet size}) & \text{otherwise} \end{cases}$$

Then two features (X_{enc} , Y_{enc}) are computed by taking a log function as follows:

$$X_{enc} = \log_{10}(TotalPacket)$$
$$Y_{enc} = \log_{10}(TotalPacket_{LOSS} + 1)$$

B.2.1.2 Green block feature

Some videos may contain mono-color blocks due to severe transmission errors. A feature (*Greenblk*) reflecting this impairment is computed as follows:

$$U_{zero_pixel}(i, j, k) = \begin{cases} 1 & U(i, j, k) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$V_{zero_pixel}(i, j, k) = \begin{cases} 1 & V(i, j, k) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$U_{zero_line}(j, k) = \sum_{i=1}^{width} U_{zero_pixel}(i, j, k)$$

$$V_{zero_line}(j, k) = \sum_{i=1}^{width} V_{zero_pixel}(i, j, k)$$

$$U_{zero_flag}(j, k) = \begin{cases} 1 & U_{zero_line}(j, k) > width/8 \\ 0 & \text{otherwise} \end{cases}$$

$$V_{zero_flag}(j, k) = \begin{cases} 1 & V_{zero_line}(j, k) > width/8 \\ 0 & \text{otherwise} \end{cases}$$

$$U_{zero} = \sum_{k=1}^{NumFrameheight} \sum_{j=1}^{NumFrameheight} U_{zero_flag}(j, k)$$

$$V_{zero} = \sum_{k=1}^{NumFrameheight} \sum_{j=1}^{NumFrameheight} V_{zero_flag}(j, k)$$

$$Greenblk = \frac{1}{NumFrame} (U_{zero} + V_{zero})$$

Here, U is the u channel and V is the v channel in the yuv video format.

B.2.1.3 Freeze feature

To compute a freeze feature (FRZ_{total}), the frame difference is calculated using the luminance channel as follows:

$$FrameDiff(k) = \frac{1}{FramePixelSize} \sum_{(i,j)} |Y(i, j, k) - Y(i, j, k-1)|$$

$$FreezeFlag(k) = \begin{cases} 1 & \text{if } FrameDiff(k) < Th_{frz} \\ 0 & \text{otherwise} \end{cases}$$

$$FRZ_{total} = \sum_k FreezeFlag(k)$$

B.2.1.4 Features for blocking, blur metric, freeze and repeating blocks

B.2.1.4.1 Blocking metric

To compute the blocking metric, the absolute horizontal difference is first computed as follows:

$$d_h[j, k] = |Avg_L - Avg_R|$$

where $Avg_L = \frac{1}{2} \sum_{p=-1}^0 Frame[j + p, k]$, $Avg_R = \frac{1}{2} \sum_{p=1}^2 Frame[j + p, k]$.

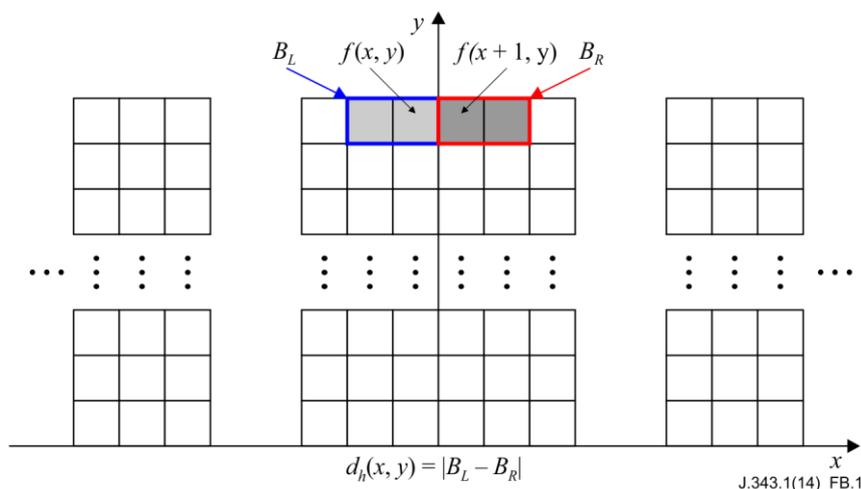


Figure B.1 – Computing the absolute horizontal difference ($d_h[j, k]$)

Then, the sum of horizontal blockiness (SB_h) at position j is computed as follows:

$$SB_h[j] = \left(\sum_{1 \leq k \leq height} (|Frame[j, k] - Frame[j + 1, k]| \times u(d_h[j, k] - \Phi(Avg_L))) \right)^2$$

where $u(\cdot)$ represents the unit step function and

$$\Phi(s) = \begin{cases} 17 \left(1 - \sqrt{\frac{s}{127}}\right) + 3 & \text{if } s \leq 127 \\ \frac{3(s-127)}{128} + 3 & \text{otherwise} \end{cases}$$

After repeating the procedure for the entire frames, the frame horizontal blockiness (FB_h) is computed as follows:

$$FB_h = \left(\sum_{\substack{1 \leq j \leq \text{width} \\ j \equiv 0 \pmod{8}}} SB_h[j] \right)^{\frac{1}{2}}$$

For each frame, the column difference (NFB_h) excluding every 8-th column is computed as follows:

$$NFB_h = \frac{1}{7} \sum_{l=1}^7 \left(\sum_{\substack{1 \leq j \leq \text{width} \\ j \equiv l \pmod{8}}} \left(\sum_{1 \leq k \leq \text{height}} (|Frame[j, k] - Frame[j+1, k]| \times u(d_h[j, k] - \Phi(Avg_L))) \right)^2 \right)^{\frac{1}{2}}$$

Then, the final horizontal blocking feature (BLK_H) is computed as follows:

$$BLK_H = \ln(FB_h/NFB_h)$$

The vertical blocking feature (BLK_V) is computed similarly. For interlaced video sequences, the vertical blocking feature is computed in the field sequence. The i -th frame blocking score is computed as follows:

$$FrameBLK[i] = 0.5 \times BLK_H + 0.5 \times BLK_V$$

The final blocking score ($BlockingScore$) is computed by averaging the top 10% frame blocking scores.

B.2.1.4.2 Blurring metric

While blurring artifacts are not always visible in flat (homogeneous) regions, they are mostly recognizable in edge areas. Based on this observation, the frames are divided into a number of blocks and each block is classified as a flat or edge block. Then, the blur radius is computed for the edge blocks. To classify each block as a flat (homogeneous) or edge block, the variance is computed at each pixel position (x, y) as follows:

$$v(x, y) = \frac{1}{LN} \sum_{j=-N/2}^{N/2} \sum_{i=-L/2}^{L/2} (f(x+i, y+j) - E)^2$$

where $v(x, y)$ represents the variance value at (x, y) , L is the width of the window, N is the height of the window, and E is the mean of the window. Then each pixel is classified using the following equations:

$$Pixel\ type = \begin{cases} Flat, & v(x, y) \leq th \\ Edge, & th < v(x, y) \end{cases}, \quad th = 400.$$

A block is classified as flat or edge block by pixel classification results. If there is at least one edge pixel in a block, the block is classified as an edge block. Otherwise, the block is classified as a flat block. An edge $e(x)$ is modeled as a step function:

$$e(x) = \begin{cases} A + B, & x \geq 0 \\ B, & x < 0 \end{cases}$$

where A is an amplitude and B is an offset. When the edge is blurred with an unknown Gaussian blur radius σ , the blurred edge is modeled as follows:

$$b(x) = \begin{cases} \frac{A}{2} \left(1 + \sum_{n=-x}^x g(n, \sigma) \right) + B, & x \geq 0 \\ \frac{A}{2} \left(1 - \sum_{n=x+1}^{-x-1} g(n, \sigma) \right) + B, & x < 0 \end{cases}$$

where $g(n, \sigma)$ represents a normalized Gaussian kernel ($g(n, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{n^2}{2\sigma^2}}$, $n \in Z$).

Two re-blurred edges ($b_a(x)$, $b_b(x)$) are obtained with two blur radii (σ_a and σ_b ($\sigma_a < \sigma_b$)). Then, the difference $r(x)$ is calculated as follows:

$$r(x) = \frac{b(x) - b_a(x)}{b_a(x) - b_b(x)}$$

The blur radius σ is estimated as follows:

$$\sigma \approx \frac{\sigma_a \cdot \sigma_b}{(\sigma_b - \sigma_a) \cdot r(x)_{max} + \sigma_b}$$

where σ_a and σ_b are empirically set to 1 and 4. The blur radius σ is calculated only for the edge blocks and F_{BLR} is obtained as follows:

$$F_{BLR} = \begin{cases} \frac{1}{N_B} \sum_i \sigma_i, & N_B > 0 \\ 1, & N_B = 0 \end{cases}$$

where σ_i is the blur radius of the i_{th} block and N_B is the total number of edge blocks in the frame.

The i_{th} frame blurring score is computed as follows:

$$FrameBLR[i] = F_{BLR}$$

The final blurring score (*BlurringScore*) is computed by averaging the frame blurring scores.

B.2.1.4.3 Freezing metric

To detect irregular frame freezing, all frozen frames are detected as follows:

$$Diff(i) = \frac{1}{W \times H} \sum_{x=0}^W \sum_{y=0}^H u(|f_i(x, y) - f_{i-1}(x, y)| - T)$$

$$FreezeFlag(i) = \begin{cases} 1 & \text{if } Diff(i) > 0.99 \\ 0 & \text{otherwise} \end{cases}$$

where $Diff(i)$ represents a portion of pixels that have the same value between adjacent frames, i is the frame index, W is the width of the frame, H is the height of the frame, and $u(t - T)$ is the shifted unit step function for a thresholding operation.

In order to measure irregular frame freezing, the regularly repeated frame freezing is first detected as follows:

$$Repeat[i \bmod M] ++, \text{ if } (FreezeFlag(i) = 1) \quad \{i | 1 \leq i \leq N\}$$

where $Repeat[k]$ represents the histogram of the frozen frames, and M represents the initial assumed freeze duration. The repeated freeze duration and the freeze flag are modified as follows:

$$FreezeFlag(i) = 0, \text{ if } (Repeat[i \bmod M] > \bar{m} \text{ and } v > 4.0), \quad \{i | 1 \leq i \leq N\}$$

$$\bar{m} = \frac{1}{M} \sum_{k=1}^M Repeat[k]$$

$$v = \frac{1}{M} \sum_{k=1}^M (Repeat[k] - \bar{m})^2$$

The final freezing score ($FreezingScore$) is computed as follows:

$$FreezingScore = \begin{cases} \frac{1}{N} \times (30 \times \log_{30}(1 + x_k)), & 0 \leq x_k \leq 29 \\ \frac{1}{N} \times 30, & x_k \geq 30 \end{cases}$$

$$x_k = \sum_{i=1}^N FreezeFlag(i)$$

where N represents the total number of frames in the video sequence.

B.2.1.4.4 Repeating block metric

Compressed video sequences may contain repeating blocks when the video sequence is highly compressed. To count the number of repeating blocks, the pixel difference between adjacent frames is computed as follows:

$$PixelFlag(bx, by) = \begin{cases} 1 & \text{if } |B_{i-1}(bx, by) - B_i(bx, by)| = 0 \\ 0 & \text{otherwise} \end{cases}$$

where bx and by represent the index value of the 4×4 block and $B(bx, by)$ is the 4×4 block. Then the homogeneous block (HB) is defined as follows:

$$HB \equiv \sum_{bx=1}^4 \sum_{by=1}^4 PixelFlag(bx, by) = 0$$

In each frame, the portion of homogeneous blocks (P_{HB}) is calculated as follows:

$$P_{HB} = \frac{NHB}{TB}$$

where NHB represents the total number of repeating blocks whose averaging luminance ranges from 60 to 200 and TB represents the total number of 4×4 blocks. The final repeating block score ($SameblockScore$) is computed by averaging the top 10% of P_{HB} .

B.2.2 VQM Computation

B.2.2.1 VQM computation using encrypted bitstream data and LUT

Using LUT, $HNRI_{enc}$ is computed as follows:

$$HNRI_{enc} = LUT_{enc}(X_{enc}, Y_{enc})$$

The function (*LUTenc*) uses the bilinear interpolation. The LUT for HD and the LUT for VGA/WVGA are provided electronically in the Excel file attached to this Recommendation.

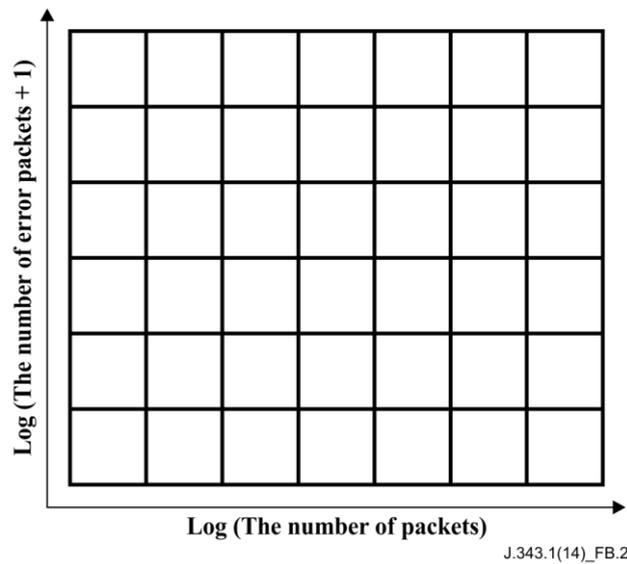


Figure B.2 – A look-up table for VQM computation

B.2.2.2 PVS VQM

The PVS VQM (*PVQM*) is computed as a weighted summation of the blocking score, blurring score, freezing score and repeating block score as follows:

$$PVQM = -3.09977 \times BlockingScore - 0.9942 \times BlurringScore - 1.83274 \times FreezingScore - 0.72623 \times SameblockScore + 4.175739$$

For HD, the following adjustment is made to $HNR1_{enc}$:

if (Resolution = HD and PVQM > HNR_{enc} + 0.5 and PVQM > 2.7)

$$HNR1_{enc} = \frac{1}{2}(HNR1_{enc} + PVQM)$$

B.2.3 Post-processing

The VQM value computed using encrypted bitstream data and LUT ($HNR1_{enc}$) is used as input (vqm_{in}) for the post-processing in this clause.

First, the green block impairment is reflected as follows:

$$vqm_1 = \begin{cases} MIN(vqm_{in}, 1.6) & \text{if } Greenblk > 1.0 \\ MIN(vqm_{in}, 2.5) & \text{if } Greenblk > 0.0 \\ vqm_{in} & \text{otherwise} \end{cases}$$

Second, the frame rate is considered for VGA/WVGA as follows:

$$vqm_2 = \begin{cases} MIN(vqm_1, 3.2) & \text{if } fps < 6 \\ MIN(vqm_1, 3.5) & \text{if } fps < 10. \\ vqm_1 & \text{otherwise} \end{cases}$$

Finally, the freeze impairment is reflected as follows:

$$FRZ_{temp} = \begin{cases} FRZ_{total} - \left(1 - \frac{fps}{fps_{original}}\right) \times fps_{original} \times VideoSec & \text{if } Resolution = VGA \text{ or } Resolution = WVGA \\ & otherwise \end{cases}$$

$$FRZ_{log} = MIN(\log_{10}(FRZ_{temp} + 1.0), 2.3)$$

$$vqm_{out} = \begin{cases} MIN(vqm_2, 4 - \log_{10}(FRZ_{log} - 0.3) \times 3.8) & \text{if } FRZ_{log} > 1.3 \\ vqm_2 & otherwise \end{cases}$$

This vqm_{out} value is outputted as the final VQM.

Bibliography

- [b-VQEG Hybrid] Video Quality Experts Group (2014), *Hybrid Perceptual/Bitstream Validation Test Final Report*.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	Languages and general software aspects for telecommunication systems