

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.

Downloaded from

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



Documents

Document Sharing Hub

Rec. ITU-T J.343.1 (11/2014)

3

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.

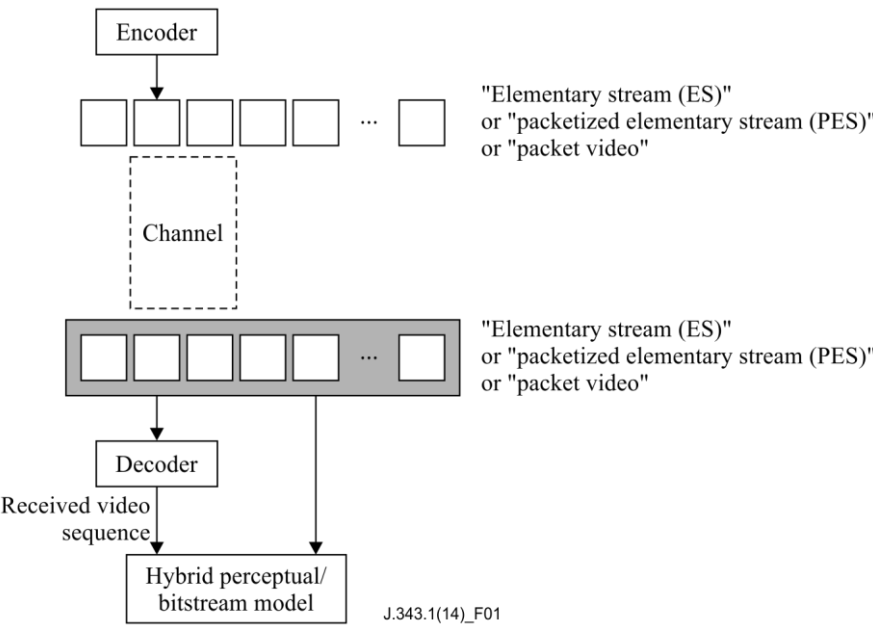


Figure 1 – Block-diagram depicts the core concept of hybrid perceptual bitstream models

Copyright International Telecommunication Union/ITU-T J.343.1(14)_F01

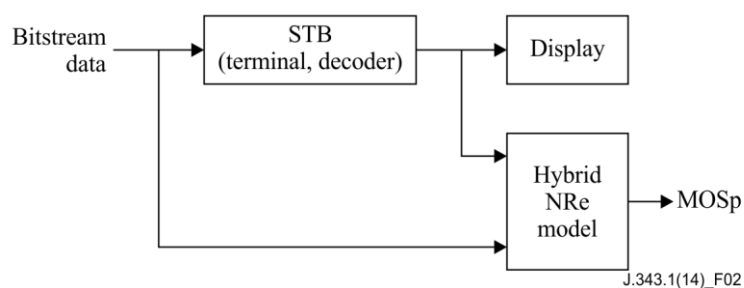


Figure 2 – Block-diagram of the Hybrid-NRe model

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 bistream (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/rtsp/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

```



```

        '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).

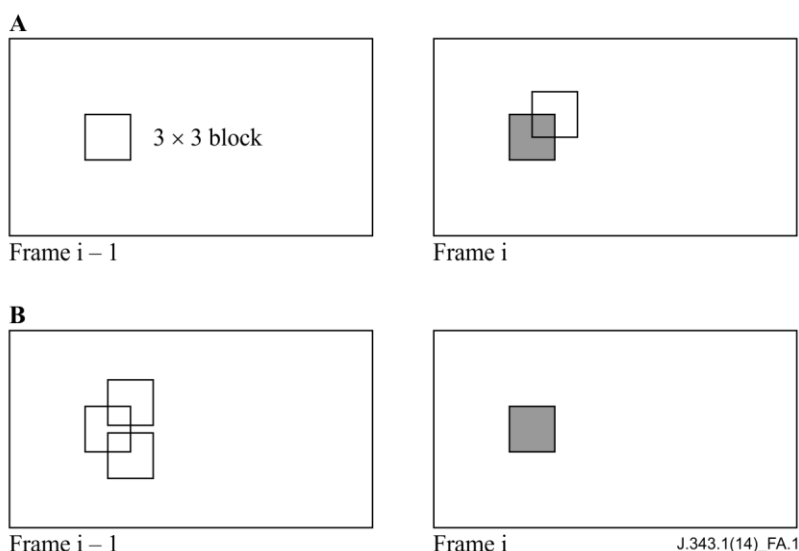


Figure A.1 – Top A: For complexity statistics, similarity of a block (grey) to spatially or temporally adjacent blocks (white) are computed.
Bottom B: For motion statistics, predictability of a block (grey) to spatially and temporally adjacent blocks (white) are computed.

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

```