EQ2330 Image and Video Processing
Project 3

Chenting Zhang chzha@kth.se
Lukas Rapp lrapp@kth.se
Shaotian Wu shaotian@kth.se

February 23, 2023

1 Introduction

In this project, we will investigate three different kinds of video coders and compare their performances when compressing two videos. The first video coder is the Intra-Frame Video Coder which only exploits spatial redundancy in each independent frame. The second video coder is Conditional Replenishment Video Coder which is the extension of Intra-frame video coder by adding block-based conditional replenishment, thus it exploits temporal redundancy between frames. The last one is Video Coder with Motion Compensation which is the extension of conditional replenishment video coder by tracking object movement and compensating for it during the prediction and differencing process.

2 System Description

2.1 Intra-Frame Video Coder

The intra-frame video coder encodes the video inside a single frame. In other words, it takes each single frame as a single image, and processes the video by each frame.

In this case, we use the a kind of discrete cosine transform, DCT-II, which is an orthonormal transform and a unitary transform to process each single frame. DCT is a well-known compression transform, which can easily transform a normal image to an image that is fit for compression. It can also work as a transform implemented in an intra-frame video coder, where each frame is independently encoded. DCT of a size $M \times M$ block x can be implemented by the following method:

$$y = AxA^T \tag{1}$$

where y is the result block, and A is a $M \times M$ matrix whose elements are

$$a_{ij} = \alpha_i \cos(\frac{(2k+1)i\pi}{2M}) \tag{2}$$

with

$$\alpha_i = \begin{cases} \sqrt{\frac{1}{M}}, i = 0 \\ \sqrt{\frac{2}{M}}, \forall i > 0 \end{cases} \tag{3}$$

In the reconstruction process, we use IDCT, the inverse transform of DCT. It can be implemented by the following method:

$$x = A^T yA \tag{4}$$

We are going to divide each frame to 16×16 blocks, and apply four 8×8 DCTs to each block.

2.2 Conditional Replenishiment Video Coder

The conditional replenishiment video coder considers temporal redundancy between different frames. For each block in each frame, there are two modes for the encoder to decide. Intra mode is elaborated in the previous section. For copy mode, the current 16×16 block is directly copied from the co-located block in the previous frame. Since the decoder decodes one frame after each other, the previous reconstructed frame can be used to as template to copy from. Therefore, the encoder does not need to store any coefficients for this block despite the mode selection.

The mode selection must stored for every block and increases the data rate of each block by 1 bit. Nevertheless, copy mode can provide a significant compression due to the fact that most blocks in successive frames are highly dependent and even identical in the video. For example, in a video conference with a static camera, the background usually does not change and can be copied.

For each block, the encoder will make a decision based on the Lagrangian Cost Function:

$$J_n = D_n + \lambda R_n. \tag{5}$$

D_n is the mean square error (MSE) between the reconstructed block $\tilde{f}_n(x, y, t)$ and the block of the original video stream $f(x, y, t)$. The rate R_n is given by

$$R_n = 1\,\text{bit} + \begin{cases} R_{\text{DCT}} & n = \text{code mode}, \\ 0 & n = \text{copy mode}, \end{cases}$$

where R_{DCT} is the rate to encode the DCT coefficients in code mode using a variable length code (VLC). Section 2.4 describes how the rate is calculated. λ is the Lagrangian multiplier balancing the weight between distortion and bit rate. It is scaled by the step size q: $\lambda = 0.2 \times q^2$.

Remark Since the decoder in copy mode copies frames from the reconstructed video, the distortion $D_{\text{copy mode}}$ must also be calculated between the previous block of the reconstructed video and of the current block of the original video. Hence, during the encoding process the video is decoded in parallel providing the needed reconstructed video stream.

2.3 Video Coder with Motion Compensation

Successive frames in a video sequence are often very similar, so temporal redundancy can be exploited like copying a block as seen in the previous section. However, when a sequence of frames contains moving objects, the similarity between blocks of successive frames is reduced, and compression is affected negatively. Thus, motion compensation is introduced to track movements and compensate for it.

For each block of the current frame, a motion compensation encoder searches for the most similar block of the previous frame. The motion is described by a horizontal and vertical displacement d_x and d_y for the "most likely" position [1]. To keep the search feasible, the search range is limited by $d_x, d_y \in [-10, 10]$.

To find the optimal motion vector, the sum of squared difference (SSD) between the current block $f(x, y, t)$ and the shifted blocks $\tilde{f}(x, y, t-1)$ of the previous frame is minimized:

$$\text{SSD}(d_x, d_y) = \sum_{x=0}^{m-1} \sum_{y=0}^{m-1} \Big(f(x, y, t) - \tilde{f}(x + d_x, y + d_y, t-1) \Big)^2. \tag{6}$$

We neglect motion vectors that point out of the frame. Furthermore, as discussed in the last section, the decoder only has access to the reconstructed video stream. Therefore, in order to minimize the distortion of the decoded video, the encoder also uses the reconstructed frame $\tilde{f}$ for its search. The implementation of this search can be found in section A.2.3.

After calculating the optimal motion vector, we use it to generate the predicted block which is a shifted version of previous encoded frame $\hat{f}(x, y, t) = \tilde{f}(x + d_x, y + d_y, t-1)$. Then, by subtracting the predicted block from the original block, we could obtain the residual of each block:

$$r(x, y, t) = f(x, y, t) - \hat{f}(x, y, t).$$

This residual is encoded by quantized DCT coefficients like the blocks in code mode. Since the residual only stores the prediction error, its values are usually significantly smaller than the ones of the original image. For the same distortion, the VLC can therefore compress the residual more efficiently than the original image resulting in a data rate reduction. In addition to the residual, the motion vectors are encoded and stored. For the reconstruction, the decoder decodes the motion vectors and determines the predicted block in the previous reconstructed frame. Then, the residual is decoded by a IDCT and added to predicted block.

For each block in each frame, there are three modes for the encoder to decide: code mode, copy mode, and inter mode. The first two modes are elaborated in the previous section. For the inter mode, the coded residual coefficients and motion information are encoded. The decision is made by minimizing the Lagrangian cost function (5) of all three modes. However, in this case, there are three modes to select from which costs 2 bit per block resulting in the following rates:

$$R_n = 2\,\text{bit} + \begin{cases} R_{\text{DCT, frame}} & n = \text{code mode}, \\ 0 & n = \text{copy mode}, \\ R_{\text{DCT, residual}} + 9\,\text{bit} & n = \text{inter mode}, \end{cases}$$

For code and inter mode, the encoded dct coefficients of the original image and residual contribute to the data rate by $R_{\text{DCT, frame}}$ and $R_{\text{DCT, residual}}$, respectively (see section 2.4). For inter mode, the $N = 20^2 = 400$ motion vectors must be additionally encoded resulting in $\lceil \log_2(N) \rceil = 9\,\text{bit}$ per block.

2.4 Rate calculation

The DCT coefficients of the frames or residuals in code mode or inter mode are encoded by a VLC. The encoder uses an individual VLC for each coefficient in a 8×8 DCT block. These VLCs are used for all blocks and all frames. Furthermore, individual VLCs are used to encode the coefficients in code mode and the residuals in inter mode. These VLCs are generated based on the statistics of the coefficients of all blocks of the video. For this, all blocks are encoded by code or inter mode, respectively. Section A.2.4 lists the code to calculate the statistics and the resulting rates:

First, the probability $P(m, n, c)$ of the quantized DCT coefficient c is calculated based on the quantized DCT blocks $\text{coeff}^{(\ell)}(m, n, t)$, where m, n denote the position in the DCT block, ℓ is the index of the block and t is the frame number:

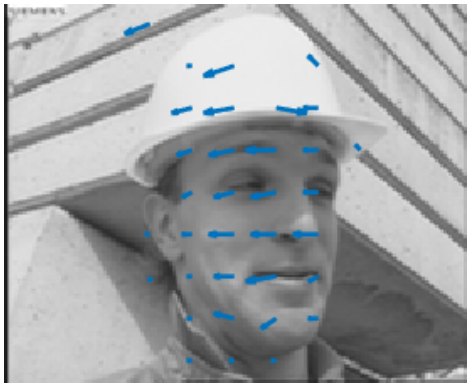
$$P(m, n, c) = \frac{1}{N_{\text{blocks}} N_{\text{frames}}} \sum_{\ell=1}^{N_{\text{blocks}}} \sum_{t=1}^{N_{\text{frames}}} \mathbb{1}\left\{ \text{coeff}^{(\ell)}(m, n, t) = c \right\},$$

where N_{blocks} is the number of blocks in one frame and N_{frames} is the number of frames in the video and $\mathbb{1}(\text{cond})$ is the indicator function which is 1 if the condition "cond" is fulfilled and zero otherwise.

During the real encoding process, these probabilities are used to calculate the bits which are needed to encode each coefficient. For instance, the coefficient c at position (m, n) in a DCT block would need approximately $-\log_2(P(m, n, c))$ bits. Summing all bit rates in a block results in the bit rate of this block.

Note that the number of bits per coefficient are not integers because this calculation is only an approximation of the bit rate that a real VLC would have.[1] Furthermore, As pointed out in the project description, the estimation of the probabilities is suboptimal because in reality all coder modes are mixed resulting in slightly different statistics. Hence, during the encoding process, some coefficients c could be encoded whose probability $P(m, n, c)$ is 0 because they have not occurred in the in the only code mode or only inter mode encoding. The bit rate of these coefficients is not definied because they are not part of the VLC. Hence, a large penalty bit rate is used for them: $\log_2(N_{\text{coeff},m,n}$, where $N_{\text{coeff},m,n}$ is the number of quantized coefficients that are encoded by the VLC.

[1] This approximation is used because the construction of a VLC was not part of the project. However, the analyze could be easily extended by a VLC construction.



(a) Motion vectors for inter mode.

2	2	2	2	2	2	2	3	3	2	3
2	2	2	3	2	2	2	3	3	3	3
2	2	2	2	3	3	3	3	3	2	3
2	2	2	3	3	3	3	3	3	2	2
2	3	2	3	3	3	3	1	2	3	2
2	2	2	3	3	3	3	3	2	3	3
2	2	2	3	3	3	1	3	2	3	3
2	2	2	2	3	3	3	3	2	2	2
2	2	2	2	2	3	3	2	2	2	2

(b) Selected mode

Figure 1: Encoding of frame 4 of *Foreman* with video coder 3. Number 1 represents Code mode, Number 2 represents Copy mode, and number 3 represents Intra mode. (For visualization, the arrows of the motion vectors are scaled.)

3 Results

We analyzed the performance of all three video coders using the the *Mother-Daughter* video and on the *Foreman* video. For both videos, the first 50 frames have been encoded.

3.1 Motion vector

Figure 1a illustrates the optimal motion vectors of frame 4 of the Foreman video. We could conclude that the motion vector of most blocks is $[0, 0]$ which indicates that they do not contain motion information and are stationary. The motion vectors at the head of the man are pointing to the left indicating that the head of the man is waving to the right. Note that the motion vectors at the helm of the man are sometimes $[0, 0]$ or pointing in the wrong direction even though the helm is moving as well. These mistakes occur because the helm is completely white without texture so an exact estimation of the movement is not possible. This is however no issue for the motion compensation because the exact motion does not matter as long as predicted blocks with a high similarity can be found.

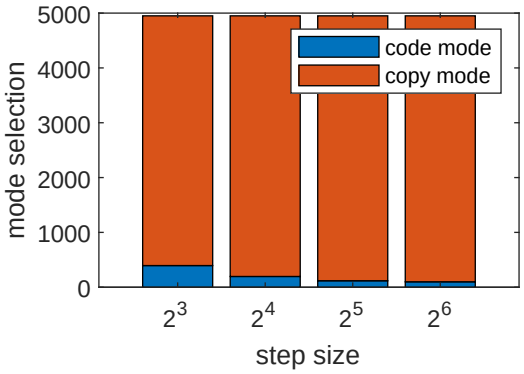
3.2 Block selection

The figure 2 shows the frequency with which each mode is selected for video coder2 (code + copy mode) and coder3 (code + copy + inter mode) in the videos *Foreman* and *Mother-Daughter*.

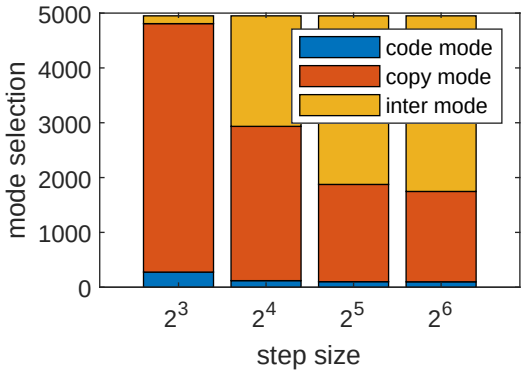
In general, we see that copy and inter mode are mostly used. Especially for large step sizes, code mode is only rarely used. Both modes have compared to code mode a smaller bit rate but introduce more distortion. However, for large step sizes the quantization noise increases for code mode. Therefore, the distortion that copy and inter mode introduce is comparable with code mode and copy and inter mode are favoured due to its reduced bit rate. We assume that only few complex blocks which cannot be predicted from the previous frame (even with motion compensation) choose code mode. Furthermore, the first frame is always encoded with code mode.

As for two diagrams of video coder 3, the choice between copy mode and inter mode is largely related to the "trade off" parameter λ, by virtue of the fact that motion compensated prediction decreases the distortion at the cost of higher bit rate compared with copy mode. This assumption is supported by our experiments that have shown that copy mode is preferred over inter mode for a larger λ.

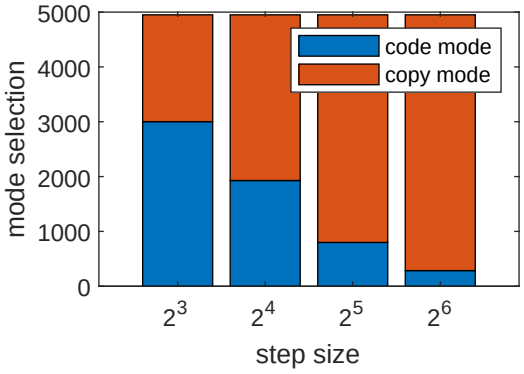
Especially, when plotting the heat map of mode selection of encoder 3 in a particular frame(figure 1b), we could conclude that most blocks choose the latter two modes, the blocks representing moving object(foreign man) choose motion compensated prediction(inter mode) while the blocks representing stationary background choose the copy mode which coincides with our previous assumption.



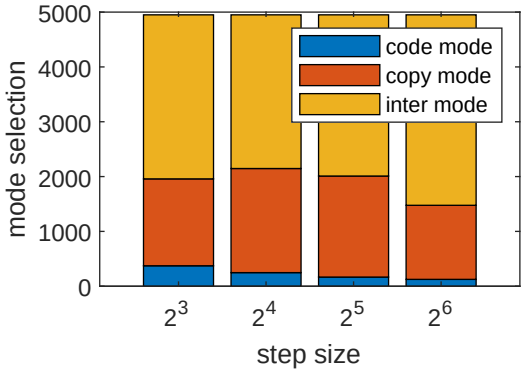
(a) Video Coder 2: Mother-Daughter



(b) Video Coder 3: Mother-Daughter



(c) Video Coder 2: Foreman



(d) Video Coder 3: Foreman

Figure 2: Mode selection frequency for encoder 2 and 3

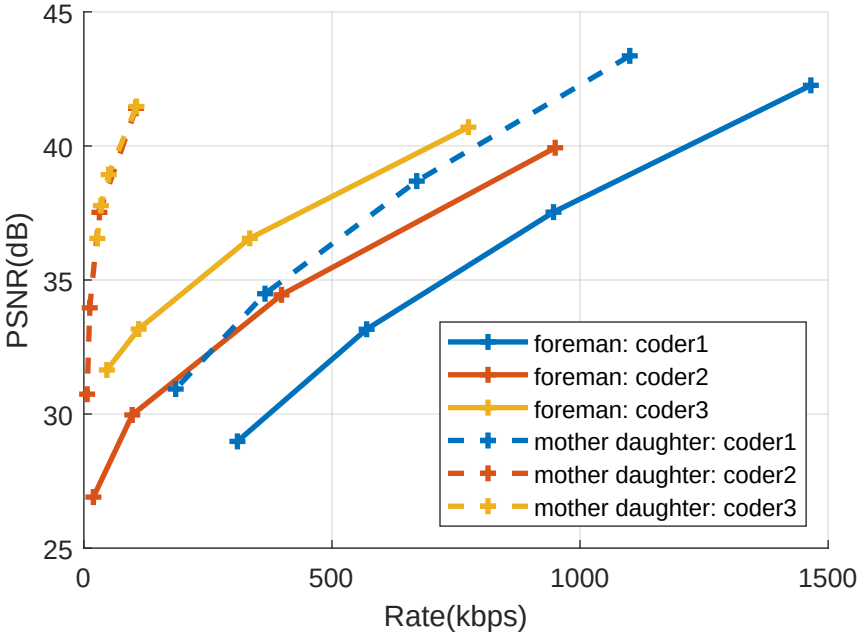


Figure 3: Rate distortion curves

3.3 Performance analysis

First, the bit rate without compression is analyzed: For each frame with size 176×144 pixels, every pixel value needs to be encoded with 8 bit. The resulting bit rate (in kbps) is given by

$$\mathrm{R} = 30 \times (176 \times 144 \times 8) = 6117 \text{ kbps}$$

Figure 3 shows the rate-distortion curves of the three decoders for both videos. We can conclude that the bit rate largely decreases in all three coders(compared with 6117kbps). The compression performance can be ranked as coder3 > coder2 > coder1. This result was expected: Since each coder3 and coder2 use an additional mode compared to the coder2 and coder1, respectively, their performance must necessarily be at least as good as the performance of the previous one.

The compression performance of the *Mother-Daughter* video for all decoders is significantly better than of the *Foreman* video. The *Mother-Daughter* video has a static camera and two people are only slowly moving their heads. In addition, a large part of the background has only a constant color. The constant background can be easily encoded by copy mode. The compression of the background in code mode is also large since only the DC coefficient is not zero. Both facts result in this large compression compared to the other video.

Interestingly, the performance of coder3 is almost the same as for coder2 for the *Mother-Daughter* video. The reason for this might that there is only a small area in the video which is moving and hence, the gain of using inter mode is only small.

For the *Foreman* video on the other hand, their is a compression gain of 200 kbps between coder3 and coder2 for 35 dB PSNR. In the *foreman* video a man is heavily moving his head in front of a textured background. Furthermore, the camera is sometimes moving slowly. Hence, inter mode can encode these motions which in coder2 has to be encoded by code mode. Figure 2 supports this assumption: The frequency of the usage of code mode is significantly reduced from coder 2 to coder 3 which is not the case for the *Mother-Daughter* video

3.4 Mode selection

Lastly, we analyze the mode selection of video encoder 3, where we use frame 4 of the *Foreman* video which we already have used to analyze the motion prediction (see figure 1). In the figure, We see that the areas with large motion, which is the head of the man which is moving, are primarily

encoded with inter mode as expected. Since the camera is static in this frame, the background is not moving and is therefore encoded with copy mode. This corresponds to the analysis of the mode selection when we build our encoders.

4 Conclusions

In this project, three video coder were implemented. The coders compress the video by using code, copy and inter mode which exploit spatial and temporal redundancy. Spatial redundancy is exploit by applying a DCT transformation on small blocks of the video whose coefficients are encoded by a VLC. Especially, for a video with a constant background, we see that this transformation allows a large compression since likely low frequency components can be encoded with shorter codewords. The temporal redundancy is exploit by copy mode and motion mode. Copy mode copies previous blocks. We have seen that this mode allows a large compression of constant backgrounds and is therefore especially for video conference systems important. Motion mode deals with motion in the video by searching for similar blocks in the previous frame and encoding only the residual. In our example videos, we saw that this mode allows the efficient encoding of moving head since only their position but not the pixels are changing. This mode avoids the expensive encoding of moving parts by intra mode and should therefore be part of any efficient video encoder. For the mode selection, a Lagrangian loss function weighting the compression rate and distortion was implemented. Our experiments have shown that the weight parameter is an important parameter which needs to be optimized for the requirements of the specific application to get the best trade-off between rate and distortion.

A Appendix

A.1 Who Did What

- Chenting Zhang and Lukas Rapp: Conditional Replenishment video coder and motion compensated video coder
- Shaotian Wu: Intra-Frame video coder

A.2 MatLab code

In the following, the code for the video encoder is listed. The script in section A.2.1 executes all three encoders and analyzes the results. The following sections list the functions needed for this script.

A.2.1 Video Encoder

```
% without any compression:
% frame size: 176*144 pixels , 8 bits/pixel , 30 frames/s
% (176*144*8*30) bits/s = 6117kbps

%% initialization
clear;
close all;
num_frames = 50;
size_y = 176;
size_x = 144;           % size of the frame in the video

shift_x_list = -10:10; % -2:2;
shift_y_list = -10:10; %-2:2;
step_size_list = [2^3,2^4,2^5,2^6];

```

```
16 block_size = 16;
17 num_blocks_x = size_x / block_size;
18 num_blocks_y = size_y / block_size;
19 % 9*11 blocks per frame, 4950 blocks per video
20 file = yuv_import_y('mother-daughter_qcif.yuv', [size_y, size_x],
       ↪ num_frames);      % extract the luminance component of YUV
       ↪ sequence
21 %file = yuv_import_y('foreman_qcif.yuv', [size_y, size_x], num_frames);
       ↪       % extract the luminance component of YUV sequence
22
23 video_original = zeros(size_x,size_y,num_frames);
24 for frame_idx = 1:num_frames
25     tmp = file(frame_idx);
26     video_original(:,:,frame_idx) = tmp{1};
27 end
28
29 %% intra-frame coder
30 % generate frame_DCT
31 video_dct = DCT_video(video_original, num_blocks_x, num_blocks_y,
       ↪ block_size);
32
33 % generate quantized_DCT
34 video_dct_quantized = zeros(size_x,size_y,num_frames,length(
       ↪ step_size_list));
35 video_dct_quantized_int = zeros(size_x,size_y,num_frames,length(
       ↪ step_size_list));
36 for q = 1:length(step_size_list)
37     video_dct_quantized_int(:, :, :, q) = sign(video_dct) .* floor(abs(
           ↪ video_dct)./step_size_list(q)+1/2);
38     video_dct_quantized(:, :, :, q) = step_size_list(q)*
           ↪ video_dct_quantized_int(:, :, :, q);
39 end
40
41 number_of_bits_per_block_intra = calc_number_of_bits_per_block(
       ↪ video_dct_quantized_int, step_size_list, num_blocks_x,
       ↪ num_blocks_y, block_size);
42
43 % IDCT reconstruction
44 video_reconst_coder1 = IDCT_video(video_dct_quantized, num_blocks_x,
       ↪ num_blocks_y, block_size);
45
46 PSNR_coder1 = calc_PSNR(video_original, video_reconst_coder1);
47
48 % Rate intra decoder kbits per second
49 bitr_coder1 = squeeze(sum(number_of_bits_per_block_intra, [1, 2, 3]))
       ↪ /1000/(5/3);
50
51 %% conditional replenishment coder
52 [selection_coder2, bitr_coder2, PSNR_coder2, video_reconst_coder2] =
       ↪ encode_video(size_x, size_y, num_blocks_x, ...
53     num_blocks_y, num_frames, step_size_list,
           ↪ number_of_bits_per_block_intra, ...
54     video_original, video_reconst_coder1, block_size, ...
55     0, 0, false, shift_x_list, shift_y_list);
56
57 %% motion-compensated coder
```

```
58 [vlc_values_motion_residual, vlc_probs_motion_residual] =
       ↪ calc_vlc_statistics_motion_residuals(size_x, size_y,
       ↪ num_blocks_x, ...
59     num_blocks_y, num_frames, step_size_list, video_original,
           ↪ block_size, video_reconst_coder1, shift_x_list, shift_y_list
           ↪ );
60
61 [selection_coder3, bitr_coder3, PSNR_coder3, video_reconst_coder3] =
       ↪ encode_video(size_x, size_y, num_blocks_x, ...
62     num_blocks_y, num_frames, step_size_list,
           ↪ number_of_bits_per_block_intra, ...
63     video_original, video_reconst_coder1, block_size, ...
64     vlc_values_motion_residual, vlc_probs_motion_residual, true,
           ↪ shift_x_list, shift_y_list);
65
66 save("video_encoding_results_mother-daughter", "bitr_coder1", "
       ↪ bitr_coder2", "bitr_coder3","PSNR_coder1", "PSNR_coder2", "
       ↪ PSNR_coder3", "selection_coder2", ...
67     "selection_coder3", "video_reconst_coder1", "video_reconst_coder2",
           ↪  "video_reconst_coder3", "video_original");
68
69
70 function [selection, bitr, PSNR, video_reconst] = encode_video(size_x,
       ↪ size_y, num_blocks_x, ...
71     num_blocks_y, num_frames, step_size_list,
           ↪ number_of_bits_per_block_intra, ...
72     video_original, video_reconst_coder1, block_size, ...
73     vlc_values_motion_residual, vlc_probs_motion_residual,
           ↪ use_motion_mode,...
74     shift_x_list, shift_y_list)
75     % for each block, select one mode
76     selection = zeros(num_blocks_x, num_blocks_y, num_frames,length(
           ↪ step_size_list));
77     % for the first frame, always mode1
78     selection(:,:,1,:) = 1;
79
80     % add bits for the first frame (bits for intra frame encoding plus
           ↪ 2 bit
81     % per block to encode mode)
82     % todo: Discussion: Bits for mode selection in first frame
           ↪ necessary?
83     number_of_bits = reshape(sum(number_of_bits_per_block_intra(:, :,
           ↪ 1, :), [1, 2]), 1, []); % + 2 * num_blocks_x * num_blocks_y;
84
85     video_reconst = zeros(size_x,size_y,num_frames,length(
           ↪ step_size_list));
86     % for the first frame, always intra-mode
87     video_reconst(:,:,1,:) = video_reconst_coder1(:,:,1,:);
88
89     if use_motion_mode
90         % 2 bit for mode selection (3 mode)
91         num_bits_mode_selection = 2;
92     else
93         % 1 bit for mode selection (3 mode)
94         num_bits_mode_selection = 1;
95     end
```

```
96
97      for q = 1:length(step_size_list)
98          for frame_idx = 2:num_frames
99              if use_motion_mode
100                 optimal_shift_list = get_shifts(video_original(:,:,
                        ↪ frame_idx), video_reconst(:,:,frame_idx-1),
                        ↪ block_size, num_blocks_x, num_blocks_y,
                        ↪ shift_x_list, shift_y_list);
101             end
102
103             disp(frame_idx);
104             for block_idx_x = 1:num_blocks_x
105                 for block_idx_y=1:num_blocks_y
106                     x = (block_idx_x-1)*block_size+1;
107                     y = (block_idx_y-1)*block_size+1;
108                     block_original = video_original(x:x+block_size-1, y
                            ↪ :y+block_size-1,frame_idx);
109
110                     block_mode1 = video_reconst_coder1(x:x+block_size
                            ↪ -1, y:y+block_size-1,frame_idx,q);
111                     rate_mode1 = num_bits_mode_selection +
                            ↪ number_of_bits_per_block_intra(block_idx_x,
                            ↪ block_idx_y, frame_idx, q);
112                     distortion_mode1 = sum((block_original-block_mode1)
                            ↪ .^2,'all');
113                     L_mode1 = lagrangian(distortion_mode1, rate_mode1,
                            ↪ step_size_list(q));
114
115                     block_mode2 = video_reconst_coder1(x:x+block_size
                            ↪ -1, y:y+block_size-1,frame_idx-1,q);
116                     rate_mode2 = num_bits_mode_selection;
117                     distortion_mode2 = sum((block_original-block_mode2)
                            ↪ .^2,'all');
118                     L_mode2 = lagrangian(distortion_mode2, rate_mode2,
                            ↪ step_size_list(q));
119
120                     if use_motion_mode
121                         [block_mode3, rate_residual] =
                                ↪ calc_motion_block(video_reconst,
                                ↪ frame_idx, x, y, block_idx_x,
                                ↪ block_idx_y, optimal_shift_list, ...
122                                 block_original, vlc_values_motion_residual,
                                    ↪ vlc_probs_motion_residual,
                                    ↪ block_size, q, step_size_list);
123                         rate_mode3 = num_bits_mode_selection +
                                ↪ rate_residual + 9;
124                         distortion_mode3 = sum((block_original-
                                ↪ block_mode3).^2,'all');
125                         L_mode3 = lagrangian(distortion_mode3,
                                ↪ rate_mode3, step_size_list(q));
126                     end
127
128                     if use_motion_mode
129                         [~, mode_index] = min([L_mode1,L_mode2,L_mode3
                                ↪ ],[],'all');
130                     else
```

```
131                             [~, mode_index] = min([L_mode1,L_mode2],[],'all
                                    ↪ ');
132                         end
133
134                         if (mode_index == 1)
135                             selection(block_idx_x, block_idx_y, frame_idx,
                                    ↪ q) = 1;
136                             video_reconst(x:x+block_size-1, y:y+block_size
                                    ↪ -1,frame_idx,q) = block_mode1;
137                             number_of_bits(q) = number_of_bits(q) +
                                    ↪ rate_mode1;
138                         elseif (mode_index == 2)
139                             selection(block_idx_x, block_idx_y, frame_idx,
                                    ↪ q) = 2;
140                             video_reconst(x:x+block_size-1, y:y+block_size
                                    ↪ -1,frame_idx,q) = block_mode2;
141                             number_of_bits(q) = number_of_bits(q) +
                                    ↪ rate_mode2;
142                         else
143                             selection(block_idx_x, block_idx_y, frame_idx,
                                    ↪ q) = 3;
144                             video_reconst(x:x+block_size-1, y:y+block_size
                                    ↪ -1,frame_idx,q) = block_mode3;
145                             number_of_bits(q) = number_of_bits(q) +
                                    ↪ rate_mode3;
146                         end
147                     end
148                 end
149             end
150         end
151         bitr = (number_of_bits./1000)./(5/3);
152         PSNR = calc_PSNR(video_original, video_reconst);
153     end
154
155     function L = lagrangian(Distortion, rate, q)    % for each block
156         lambda = 0.2.*(q^2);
157         L = Distortion + lambda.*rate;
158     end
159
160     function PSNR=calc_PSNR(video_original, video_reconst)
161         % performance analysis(intra-coded)
162         % measure the PSNR of each reconstructed frame, average PSNR over
                ↪ 50 frames
163         distortion = zeros(size(video_original, 3), size(video_reconst, 4))
                ↪ ;
164         for q = 1:size(video_reconst, 4)
165             distortion(:,q) = sum((video_reconst(:,:,:,q)-video_original)
                    ↪ .^2,[1, 2]);
166         end
167
168         distortion_per_pixel = distortion/(size(video_original, 1)*size(
                ↪ video_original, 2));
169         PSNR_per_frame = 10.*log10(255^2./distortion_per_pixel);
170         PSNR = sum(PSNR_per_frame,1)./50;
171     end
```

A.2.2 Intra Encoder

```
function frame_DCT=DCT_video(video, num_blocks_x, num_blocks_y,
    ↪ block_size)
    frame_DCT = zeros(size(video));
    for frame_idx = 1:size(video, 3)
        current_frame = video(:, :, frame_idx);
        for block_idx_x = 1:num_blocks_x
            for block_idx_y=1:num_blocks_y
                x = (block_idx_x-1)*block_size+1;
                y = (block_idx_y-1)*block_size+1;
                frame_DCT(x:x+block_size-1, y:y+block_size-1,frame_idx)
                    ↪  = DCT(current_frame(x:x+block_size-1, y:y+
                    ↪ block_size-1), block_size, 8);
            end
        end
    end
end

function im=DCT(im,im_size,block_size)   % block_size here refer to 8*8
    ↪  DCT(8)!!!
if mod(im_size,block_size)
    error("Wrong block size");
end
s=im_size/block_size;
for i=0:s-1
    for k=0:s-1
        C=im(block_size*i+1:block_size*(i+1),block_size*k+1:block_size
            ↪ *(k+1));
        D=dct2(C,[block_size block_size]);
        im(block_size*i+1:block_size*(i+1),block_size*k+1:block_size*(k
            ↪ +1))=D;
    end
end
end

function reconstr_video=IDCT_video(video_dct, num_blocks_x,
    ↪ num_blocks_y, block_size)
    reconstr_video = zeros(size(video_dct));
    for q = 1:size(video_dct, 4)
        for frame_idx = 1:size(video_dct, 3)
            quantized_frame = video_dct(:,:,frame_idx,q);
            for block_idx_x = 1:num_blocks_x
                for block_idx_y=1:num_blocks_y
                    x = (block_idx_x-1)*block_size+1;
                    y = (block_idx_y-1)*block_size+1;
                    reconstr_video(x:x+block_size-1, y:y+block_size-1,
                        ↪ frame_idx,q) = IDCT(quantized_frame(x:x+
                        ↪ block_size-1, y:y+block_size-1), block_size,
                        ↪  8);
                end
            end
        end
    end
end

```

```
function im=IDCT(im,im_size,block_size)
if mod(im_size,block_size)
    error("Wrong block size");
end
s=im_size/block_size;
for i=0:s-1
    for k=0:s-1
        C=im(block_size*i+1:block_size*(i+1),block_size*k+1:block_size
            ↪ *(k+1));
        D=idct2(C,[block_size block_size]);
        im(block_size*i+1:block_size*(i+1),block_size*k+1:block_size*(k
            ↪ +1))=D;
    end
end
end
```

A.2.3 Motion compensation

```
function optimal_shift_list = get_shifts(current_frame, previous_frame,
    ↪ block_size, num_blocks_x, num_blocks_y, shift_x_list,
    ↪ shift_y_list)
    optimal_shift_list = zeros(num_blocks_x, num_blocks_y, 2);
    optimal_loss_list = zeros(num_blocks_x, num_blocks_y);

    for block_idx_x = 1:num_blocks_x
        for block_idx_y=1:num_blocks_y
            x = (block_idx_x-1)*block_size+1;
            y = (block_idx_y-1)*block_size+1;

            current_block = current_frame(x:x+block_size-1, y:y+
                ↪ block_size-1);

            loss_list = 1000000000*ones(length(shift_x_list), length(
                ↪ shift_y_list));
            for shift_x_idx = 1:length(shift_x_list)
                shift_x = shift_x_list(shift_x_idx);
                x_start = x + shift_x;
                x_end = x + block_size -1 + shift_x;
                if x_start < 1 || x_end > size(current_frame, 1)
                    continue;
                end

                for shift_y_idx = 1:length(shift_y_list)
                    shift_y = shift_y_list(shift_y_idx);
                    y_start = y + shift_y;
                    y_end = y + block_size -1 + shift_y;
                    if y_start < 1 || y_end > size(current_frame, 2)
                        continue;
                    end
                    previous_block = previous_frame(x_start:x_end,
                        ↪ y_start:y_end);

                    loss_list(shift_x_idx, shift_y_idx) = sum((
                        ↪ current_block - previous_block).^2, 'all');
                end
```

```
            end

            [val, opt_idx] = min(loss_list, [], 'all');
            [x_opt_idx, y_opt_idx] =ind2sub(size(loss_list), opt_idx);
            optimal_shift_list(block_idx_x, block_idx_y, 1) =
                ↪ shift_x_list(x_opt_idx);
            optimal_shift_list(block_idx_x, block_idx_y, 2) =
                ↪ shift_y_list(y_opt_idx);
            optimal_loss_list(block_idx_x, block_idx_y) = val;
        end
    end
end

function [block_mode3, rate_residual] = calc_motion_block(
    ↪ video_reconst_coder3, ...
    frame_idx, x, y, block_idx_x, block_idx_y, optimal_shift_list,
        ↪ block_original, ...
    vlc_values_motion_residual, vlc_probs_motion_residual, block_size,
        ↪ q, step_size_list)

    shift_x = optimal_shift_list(block_idx_x, block_idx_y, 1);
    shift_y = optimal_shift_list(block_idx_x, block_idx_y, 2);

    optimal_block_mode3 = video_reconst_coder3(x+shift_x:x+block_size
        ↪ -1+shift_x, y+shift_y:y+block_size-1+shift_y,frame_idx-1);
    residual_mode3 = block_original - optimal_block_mode3;
    residual_dct = DCT(residual_mode3, block_size, 8);
    residual_dct_quantized_int = sign(residual_dct) .* floor(abs(
        ↪ residual_dct)./step_size_list(q)+1/2);
    residual_dct_quantized = step_size_list(q) *
        ↪ residual_dct_quantized_int;
    rate_residual = calc_number_of_bits_of_block(
        ↪ residual_dct_quantized_int, vlc_values_motion_residual,
        ↪ vlc_probs_motion_residual, q);
    block_mode3 = IDCT(residual_dct_quantized, block_size, 8) +
        ↪ optimal_block_mode3;
end
```

A.2.4 Rate calculation

```
function [vlc_values_motion_residual, vlc_probs_motion_residual] =
    ↪ calc_vlc_statistics_motion_residuals(size_x, size_y,
    ↪ num_blocks_x, ...
    num_blocks_y, num_frames, step_size_list, video_original,
        ↪ block_size, video_reconst_coder1, shift_x_list, shift_y_list
        ↪ )
    % Encode using only motion compensation
    residual_dct_quantized_int = zeros(size_x, size_y, num_frames - 1,
        ↪ length(step_size_list));
    video_reconst = zeros(size_x,size_y,num_frames,length(
        ↪ step_size_list));
    video_reconst(:,:,1,:) = video_reconst_coder1(:,:,1,:);
    for q = 1:length(step_size_list)
        for frame_idx = 2:num_frames
            optimal_shift_list = get_shifts(video_original(:,:,
                ↪ frame_idx), video_reconst(:,:,frame_idx-1),
```

```
                ↪ block_size, num_blocks_x, num_blocks_y, shift_x_list
                ↪ , shift_y_list);
10              for block_idx_x = 1:num_blocks_x
11                  for block_idx_y=1:num_blocks_y
12                      x = (block_idx_x-1)*block_size+1;
13                      y = (block_idx_y-1)*block_size+1;
14
15                      block_original = video_original(x:x+block_size-1, y
                            ↪ :y+block_size-1,frame_idx);
16
17                      shift_x = optimal_shift_list(block_idx_x,
                            ↪ block_idx_y, 1);
18                      shift_y = optimal_shift_list(block_idx_x,
                            ↪ block_idx_y, 2);
19
20                      optimal_block_mode3 = video_original(x+shift_x:x+
                            ↪ block_size-1+shift_x, y+shift_y:y+block_size
                            ↪ -1+shift_y,frame_idx-1);
21                      residual_mode3 = block_original -
                            ↪ optimal_block_mode3;
22
23                      residual_dct = DCT(residual_mode3, block_size, 8);
24                      residual_dct_quantized_int(x:x+block_size-1, y:y+
                            ↪ block_size-1, frame_idx - 1, q) = sign(
                            ↪ residual_dct) .* floor(abs(residual_dct)./
                            ↪ step_size_list(q)+1/2);
25                      residual_dct_quantized = step_size_list(q) *
                            ↪ residual_dct_quantized_int(x:x+block_size-1,
                            ↪  y:y+block_size-1, frame_idx - 1, q);
26
27                      video_reconst(x:x+block_size-1, y:y+block_size-1,
                            ↪ frame_idx,q) = IDCT(residual_dct_quantized,
                            ↪ block_size, 8) + optimal_block_mode3;
28                  end
29              end
30          end
31      end
32      [vlc_values_motion_residual, vlc_probs_motion_residual] =
            ↪ calc_vlc_statistics(residual_dct_quantized_int,
            ↪ step_size_list);
33  end
34
35  function [values, probs] = calc_vlc_statistics(quantized_DCT_int,
        ↪ step_size_list)
36      values = cell(8, 8, length(step_size_list));
37      probs = cell(8, 8, length(step_size_list));
38      % generate quantized_DCT
39      for q = 1:length(step_size_list)
40          % Calculate statitistics for every DCT coefficient
41          % Stores number of bits needed to store each coefficient
42          for m = 1:8
43              for n = 1:8
44                  coefficients_m_n = quantized_DCT_int(m:8:end, n:8:end,
                        ↪ :, q);
45                  [numbers, values{m, n, q}] = groupcounts(reshape(
                        ↪ coefficients_m_n, [], 1));
```

```
46                  probs{m, n, q} = numbers / sum(numbers);
47              end
48          end
49      end
50  end
51
52  function num_bits=calc_number_of_bits_of_block(block, vlc_values,
        ↪ vlc_probs, q)
53      num_bits = 0;
54      for sub_block_x = [0, 1]
55          for sub_block_y = [0, 1]
56              for m = 1:8
57                  for n = 1:8
58                      probs = vlc_probs{m, n, q};
59                      indicies = find(vlc_values{m, n, q} == block(m +
                            ↪ sub_block_x * 8, n + sub_block_y * 8));
60                      if length(indicies) >= 1
61                          num_bits = num_bits - log2(probs(indicies(1)));
62                      else
63                          % penalty for decoding a coefficient which is
                                ↪ not
64                          % in the statistics
65                          num_bits = num_bits + log2(length(probs));
66                      end
67                  end
68              end
69          end
70      end
71  end
72
73  function number_of_bits_per_block = calc_number_of_bits_per_block(
        ↪ quantized_DCT_int, step_size_list, num_blocks_x, num_blocks_y,
        ↪ block_size)
74      number_of_bits_per_block = zeros(num_blocks_x, num_blocks_y, size(
            ↪ quantized_DCT_int, 3), length(step_size_list));
75      % generate quantized_DCT
76      for q = 1:length(step_size_list)
77          % Calculate statitistics for every DCT coefficient
78          % Stores number of bits needed to store each coefficient
79          number_of_bits_per_coefficient = zeros(size(quantized_DCT_int,
                ↪ 1), size(quantized_DCT_int, 2), size(quantized_DCT_int,
                ↪ 3));
80          for m = 1:8
81              for n = 1:8
82                  coefficients_m_n = quantized_DCT_int(m:8:end, n:8:end,
                        ↪ :, q);
83                  [numbers, values] = groupcounts(reshape(
                        ↪ coefficients_m_n, [], 1));
84                  probs = numbers / sum(numbers);
85
86                  data_tmp = zeros(size(coefficients_m_n));
87                  for value_idx = 1:length(values)
88                      data_tmp(coefficients_m_n == values(value_idx)) = -
                            ↪ log2(probs(value_idx));
89                  end
90                  number_of_bits_per_coefficient(m:8:end, n:8:end, :, :)
```

```
                    ↪ = data_tmp;
91              end
92          end
93
94          % Sum number of bits per block
95          for block_idx_x = 1:num_blocks_x
96              for block_idx_y=1:num_blocks_y
97                  x = (block_idx_x-1)*block_size+1;
98                  y = (block_idx_y-1)*block_size+1;
99
100                 current_block = number_of_bits_per_coefficient(x:x+
                        ↪ block_size-1, y:y+block_size-1, :);
101                 number_of_bits_per_block(block_idx_x, block_idx_y, :, q
                        ↪ ) = sum(current_block, [1, 2]);
102             end
103         end
104     end
105 end
```

References

[1] Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing*, Prentice Hall, 2nd ed., 2002