Mathura Shivakaran
1006879184

ECE 462 - Lab 4 Week 2: Code Explanation

forward_dct_quant:

```
function Output = forward_dct_quant(Input, QP, Q_matrix)
    % Perform 2D forward DCT on the input, with the in-built MATLAB function as I was unable to complete Lab #2
    T = dct2(Input);
    % Perform quantization on T, which holds the DCT coefficient matrix, given the formula found in the lab prep document
    Output = round(T./(QP.*Q_matrix));
end
```

This code runs a forward DCT quantization, given an input of 8x8 pixels, some quantization parameter, and whatever Q_matrix, either Q_intra or Q_inter, provided in the lab documentation. We first computed a 2D forward DCT, using the in-built dct2 MATLAB function, as I was unable to complete Lab 2. We save the DCT coefficients in T. We then perform a quantization on these DCT coefficients with the given formula 1 in the Lab 4 preparatory document. We save and return the final quantized DCT coefficients in the variable "Output".

inverse_dct_quant:

```
function Output = inverse_dct_quant(Input, QP, Q_matrix)
    % Perform inverse quantization, given the formula found in the lab prep document
    tHat = Input.*QP.*Q_matrix;
    % Perform 2D inverse DCT on the T Hat, with the in-built MATLAB function as I was unable to complete Lab #2
    Output = idct2(tHat);
end
```

This code runs an inverse DCT quantization, given an input of 8x8 block of quantized DCT coefficients, some quantization parameter, and whatever Q_matrix, either Q_intra or Q_inter, provided in the lab documentation. We first computed an inverse quantization on the input with the given formula 2 in the Lab 4 preparatory document. We save this in tHat. We then computed the 2D inverse DCT on tHat, using the in-built idct2 MATLAB function, as I was unable to complete Lab 2. We save and return the 8x8 block of reconstructed pixels in the variable "Output".

Lab4Week2Run:

```
% Loading the sequence.mat file
sequence = load('sequence.mat');

% Loading the Q_matrices.mat file
Q_matrices = load('Q_matrices.mat');

% Extracting the necessary functions
Qintra = Q_matrices.Q_intra;
Qinter = Q_matrices.Q_inter;
sequence = sequence.sequence;

% Setting variables
QP = 2;
numFrames = size(sequence, 3);
sequence = double(sequence);
```

We first load in the sequence which was provided in sequence.mat and save it into the variable "sequence". We also load Q_matrices.mat into the variable "Q_matrices". We then extract all the necessary information, such as Q_intra, Q_inter, and sequence and save them in variables. A constant for QP has been set to 2, as enlisted in the lab document. A constant for the number of frames is also set, which should be 30 as enlisted in the lab document. Finally, we convert sequence to double to ensure consistency in the types of the variables.

Mathura Shivakaran
1006879184

I-Encoder:

```
function constructedImage = Iencoder(currentFrame, QP, Qintra)
    % Retrieving the number of rows and columns of the input frame
    [M, N] = size(currentFrame);
    % Initializing the output to have the same dimensions as the input frame
    constructedImage = zeros(M, N);

    % Looping through every macroblock, using a step of 16, as each MB is 16x16
    for x = 1:16:M
        for y = 1:16:N
            % Initializing the final encoded MB to have the correct dimensions
            encodedMB = zeros(16, 16);
            % Extracting the macroblock from the current input frame
            MB = currentFrame(x:x+15,y:y+15);

            % Looping through 8x8 areas of the macroblock with a step of 8 to perform encoding
            for i = 1:8:16
                for j = 1:8:16
                % Performing a forward DCT quantization on the MB the iteration is currently on
                fDCTq = forward_dct_quant(MB(i:i+7,j:j+7), QP, Qintra);
                % Performing an inverse DCT quantization on the forward DCT quantization output
                iDCTq = inverse_dct_quant(fDCTq, QP, Qintra);
                % Saving the inverse DCT quantization output to the variable holding the final encoded MB that the iteration is currently on
                encodedMB(i:i+7, j:j+7) = iDCTq;
                end
            end
            % Adding the encoded macroblock to its respective area in the final constructed image
            constructedImage(x:x+15, y:y+15) = encodedMB;
        end
    end
end
```

This code does I-encoding on a given frame. The inputs are whatever frame, some quantization parameter, and the provided Q_intra. We start off by retrieving the size of the input frame for iteration purposes. We also initialize the output to have the same dimensions as the input. We then have a nested for loop which loops through every macroblock in the input frame. We use a step of 16 to get the correct dimensions of each macroblock, as they are 16x16 in size. We initialize a 16x16 block to save the final encoding of that particular MB. We then extract the appropriate MB from the input frame. We use another nested loop to iterate through this extracted MB. We use a step of 8 to correctly compute 8x8 blocks. Thus, we will perform forward DCT and inverse DCT four times on each MB. We then call our previously written forward_dct_quant function passing in the extracted MB. We then use the output of this function as an input parameter for the following call of the inverse_dct_quant function. We then save the output of iDCTq into the variable that holds the final encoded MB. We then save this encodedMB into its appropriate region in the output of the entire function. We repeat this process until all MBs are encoded and have a final constructedImage via I-encoding.

Mathura Shivakaran
1006879184

P-Encoder:

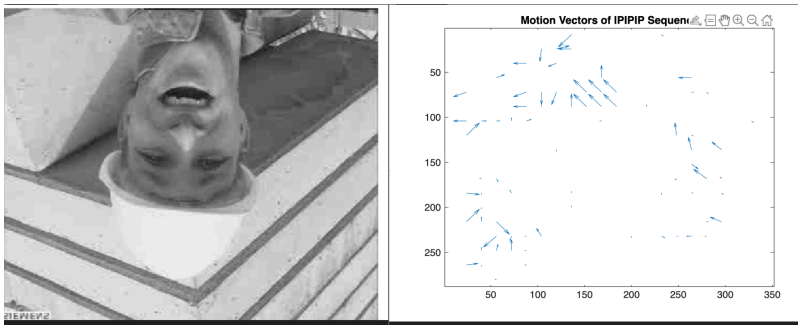
```
function constructedImage = Pencoder(referenceFrame, currentFrame, QP, Qintra, Qinter)
    % Retrieving the number of rows and columns of the input frame
    [M, N] = size(currentFrame);
    % Initializing the output to have the same dimensions as the input frame
    constructedImage = zeros(M, N);

    % Running a sequential motion search with the inputs; NB: Lab 4 sequential motion search function has errors I could not figure out
    [predictedImage, motionVectors, Flag] = Sequential_MotionSearch(currentFrame, referenceFrame);
    % Counter to keep track of which MB we are on
    counter = 1;

    % Looping through every macroblock, using a step of 16, as each MB is 16x16
    for x = 1:16:M
        for y = 1:16:N
            % If the flag is one, this indicates a successful sequential motion search
            if (Flag(counter) == 1)
                % Initializing the final encoded MB to have the correct dimensions
                encodedMB = zeros(16, 16);
                % Calculating the prediction error to use as input for the forward DCT quantization; current MB minus the predicted MB
                MB = currentFrame(x:x+15, y:y+15) - predictedImage(x:x+15, y:y+15);
                % Looping through 8x8 areas of the macroblock with a step of 8 to perform encoding
                for i = 1:8:16
                    for j = 1:8:16
                        % Performing a forward DCT quantization on the MB the iteration is currently on with Qinter for successful motion search
                        fDCTq = forward_dct_quant(MB(i:i+7, j:j+7), QP, Qinter);
                        % Performing an inverse DCT quantization on the MB the iteration is currently on with Qinter for successful motion search
                        iDCTq = inverse_dct_quant(fDCTq, QP, Qinter);
                        % Saving the inverse DCT quantization output to the variable holding the final encoded MB that the iteration is currently on
                        encodedMB(i:i+7, j:j+7) = iDCTq;
                    end
                end
                % Adding the encoded macroblock and predicted image to its respective area in the final constructed image
                constructedImage(x:x+15, y:y+15) = encodedMB + predictedImage(x:x+15, y:y+15);
                % Incrementing the counter to move to next MB and correctly retrieve the flag value
                counter = counter + 1;
            else
                % Initializing the final encoded MB to have the correct dimensions
                encodedMB = zeros(16, 16);
                % Using the current MB as input for the forward DCT quantization
                MB = currentFrame(x:x+15, y:y+15);
                % Looping through 8x8 areas of the macroblock with a step of 8 to perform encoding
                for i = 1:8:16
                    for j = 1:8:16
                        % Performing a forward DCT quantization on the MB the iteration is currently on with Qintra for unsuccessful motion search
                        fDCTq = forward_dct_quant(MB(i:i+7, j:j+7), QP, Qintra);
                        % Performing an inverse DCT quantization on the MB the iteration is currently on with Qintra for unsuccessful motion search
                        iDCTq = inverse_dct_quant(fDCTq, QP, Qintra);
                        % Saving the inverse DCT quantization output to the variable holding the final encoded MB that the iteration is currently on
                        encodedMB(i:i+7, j:j+7) = iDCTq;
                    end
                end
                % Adding the encoded macroblock to its respective area in the final constructed image
                constructedImage(x:x+15, y:y+15) = encodedMB;
                % Incrementing the counter to move to next MB and correctly retrieve the flag value
                counter = counter + 1;
            end
        end
    end
end
```

This code does P-encoding on a given frame. The inputs are whatever frame, some quantization parameter, and both the provided Q_intra and Q_inter. We start off by retrieving the size of the input frame for iteration purposes. We also initialize the output to have the same dimensions as the input. We then run a sequential motion search to determine whether the motion searches for each MB was successful or unsuccessful using the flag. A counter has been defined to keep track of which MB we are on to ensure the correct flag value is taken. We then have a nested for loop which loops through every macroblock in the input frame. We use a step of 16 to get the correct dimensions of each macroblock, as they are 16x16 in size. If the motion search is successful, we must use Q_inter as input for fDCTq and iDCTq. However, if the motion search is unsuccessful, we must use Q_intra. An if-else statement has been implemented to check whether the flag is 1 (successful, hence Q_inter), or 0 (unsuccessful, hence Q_intra). Accordingly, we initialize a 16x16 block to save the final encoding of that particular MB. We then extract the appropriate MB from the input frame. We use another nested loop to iterate through this extracted MB. We use a step of 8 to correctly compute 8x8 blocks. Thus, we will perform forward DCT and inverse DCT four times on each MB. We then call our previously written forward_dct_quant function passing in the extracted MB and either Q_inter or Q_intra, accordingly. We then use the output of this function as an input parameter for the following call of the inverse_dct_quant function, again using Q_inter or Q_intra accordingly. We then save the output of iDCTq into the variable that holds the final encoded MB. We then save this encodedMB into its appropriate region in the output of the entire function. We increment the counter for the next MB flag value, and then repeat this process until all MBs are encoded and have a final constructedImage via P-encoding.

Mathura Shivakaran
1006879184

IPIPIP Sequence Script + Outputs in Lab4Week2Run.m:

```
%-------------------------- SCRIPT FOR IPIPIP --------------------------%

% Initializing the output to have the correct dimensions
IPoutput = zeros(288, 352, 30);

% Initializing the first reference frame to be the first frame of input sequence
referenceFrame = sequence(:, :, 1);

% Looping through the total number of frames, ie. 30
for i = 1:numFrames
    % If the frame index is odd, we conduct I-encoding
    if mod(i, 2) == 1
        % I-encoding where we pass in whichever odd-index sequence frame the iteration is on
        IPoutput(:, :, i) = Iencoder(sequence(:, :, i), QP, Qintra);
    % If the frame index is even, we conduct P-encoding
    else
        % Resetting reference frame to be the previously encoded frame
        referenceFrame = IPoutput(:, :, i - 1);
        % P-encoding where we pass in the reference frame and whichever even-index sequence frame the iteration is on
        IPoutput(:, :, i) = Pencoder(referenceFrame, sequence(:, :, i), QP, Qintra, Qinter);
    end
end

% Playing the reconstructed sequence
play_frames(IPoutput, 5);

% Running a sequential motion search on the last frame and displaying the motion vectors
[predictedIP, motionVectorsIP, flagIP] = Sequential_MotionSearch(sequence(:, :, end), IPoutput(:, :, end));
show_mv(motionVectorsIP);
title('Motion Vectors of IPIPIP Sequence');

% Calculating and displaying the PSNR of the reconstructed picture
ipPSNR = psnrCalc(sequence(:, :, end), IPoutput(:, :, end));
disp(['The PSNR of the IPIPIP sequence reconstructed picture is: ' num2str(ipPSNR)]);
```



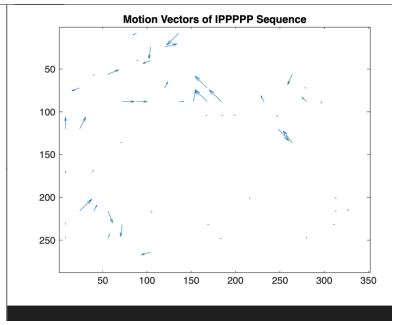

```
The PSNR of the IPIPIP sequence reconstructed picture is: 32.5679
```

We initialize the IPIPIP sequence output to have the same dimensions as the input sequence. We also initialize a variable called referenceFrame, which will be used in P-encoding. We loop through each of the frames, using numFrames to ensure we loop through every frame of the sequence. To determine whether the frame should be I-encoded or P-encoded, an if-else statement was used. In IPIPIP, we can see that every odd frame should be I-encoded and every even frame should be P-encoded. Thus, we use the modulus to determine this. If the remainder is 1, this indicates the iteration is on an odd frame. As such, we call our previously written I-encoder function, passing in the frame of the sequence we are currently looping on. Whatever is returned by the I-encoder is saved into IPoutput and its appropriate frame. If the remainder is 0, this indicates the iteration is on an even frame. As such, we call our previously written P-encoder function, passing in the frame of the sequence we are currently looping on. We also save in referenceFrame the previously encoded frame, which we determine with IPoutput (:, :, i -1). Whatever is returned by the P-encoder is saved into IPoutput and its appropriate frame. Once all the frames are looped through and saved in IPoutput, we call the play_frames function with the parameters given in the lab document to output the final video. We also run a sequential motion search on the final frame and show the motion vectors through show_mv.m. Unfortunately, I am unable to figure out what was wrong with my Sequential_MotionSearch function, so the video and motion vectors displayed are wrong. Finally, I used the function I wrote in Lab 1 to calculate the PSNR value.

Mathura Shivakaran
1006879184

IPPPPP Sequence Script + Outputs in Lab4Week2Run.m:

```
%------------------------ SCRIPT FOR IPPPPP ------------------------%

% Initializing the output to have the correct dimensions
PPoutput = zeros(288, 352, 30);

% Initializing the first reference frame to be the first frame of input sequence
referenceFrame = sequence(:, :, 1);

% Looping through the total number of frames, ie. 30
for i = 1:numFrames
    % If the remainder is 1, this indicates that the current frame is the first of a set of 6 frames, thus we perform I-encoding
    if mod(i, 6) == 1
        % I-encoding where we pass in the 1st of each set of 6 frames
        PPoutput(:, :, i) = Iencoder(sequence(:, :, i), QP, Qintra);
    % If the remainder is anything else, this indicates that the current frame is one of the other 5 frames of the set of 6, thus we perform P-encoding
    else
        referenceFrame = PPoutput(:, :, i - 1);
        % P-encoding where we pass in the 2nd to 6th of each set of 6 frames and the reference frame
        PPoutput(:, :, i) = Pencoder(referenceFrame, sequence(:, :, i), QP, Qintra, Qinter);
    end
end

% Playing the reconstructed sequence
play_frames(PPoutput, 5);

% Running a sequential motion search on the last frame and displaying the motion vectors
[predictedPP, motionVectorsPP, flagPP] = Sequential_MotionSearch(sequence(:, :, end), PPoutput(:, :, end));
show_mv(motionVectorsPP);
title('Motion Vectors of IPPPPP Sequence');

% Calculating and displaying the PSNR of the reconstructed picture
ppPSNR = psnrCalc(sequence(:, :, end), PPoutput(:, :, end));
disp(['The PSNR of the IPPPPP sequence reconstructed picture is: ' num2str(ppPSNR)]);
```




```
The PSNR of the IPPPPP sequence reconstructed picture is: 32.3809
```

We initialize the IPPPPP sequence output to have the same dimensions as the input sequence. We also initialize a variable called referenceFrame, which will be used in P-encoding. We loop through each of the frames, using numFrames to ensure we loop through every frame of the sequence. To determine whether the frame should be I-encoded or P-encoded, an if-else statement was used. In IPPPPP, we can see that in every set of 6 frames, the 1st should be I-encoded, and the next 5 should be P-encoded. Thus, we use the modulus to determine this. If the remainder is 1, this indicates the iteration is on the first of 6 frames. As such, we call our previously written I-encoder function, passing in the frame of the sequence we are currently looping on. Whatever is returned by the I-encoder is saved into IPoutput and its appropriate frame. If the remainder is anything else, this indicates the iteration is on the last 5 frames of the set. As such, we call our previously written P-encoder function, passing in the frame of the sequence we are currently looping on. We also save in referenceFrame the previously encoded frame, which we determine with IPoutput (:, :, i -1). Whatever is returned by the P-encoder is saved into IPoutput and its appropriate frame. Once all the frames are looped through and saved in IPoutput, we call the play_frames function with the parameters given in the lab document to output the final video. We also run a sequential motion search on the final frame and show the motion vectors through show_mv.m. Unfortunately, I am unable to figure out what was wrong with my Sequential_MotionSearch function, so the video and motion vectors displayed are wrong. Finally, I used the function I wrote in Lab 1 to calculate the PSNR value.