EC504 Project Final Report: Video Encoder

Jonathan Chamberlain jdchambo@bu.edu
Jialiang Shi markshi@bu.edu
Jinguang Guo gaving@bu.edu
Jeffrey Lin jlin96@bu.edu

I. Project Description and Feature Implementation

A. Project Description

Video sharing is a popular application among computer users. Among other sources, users may be interested in taking a sequence of images and converting them into a continuous video for playback. However, to be effective, such a converter must be able to store the video files efficiently. Rather than simply storing each individual image, we can improve by encoding the image data into a compressed format, which can then later be used for video playback. This project seeks to leverage compression techniques to encode an arbitrary sequence of JPEG image files into a compressed format efficiently (i.e., with the ability to encode 100 images into a file in less than five minutes, and compress an arbitrary amount of images into a file with size no more than the sum of the individual files).

B. High Level description of implementation

After studying the MPEG-1 standard details, we are implementing our project with the reference of MPEG-1 standard Part 2, which exploits perceptual compression methods to significantly reduce the data rate required by a video stream.

"It reduces or completely discards information in certain frequencies and areas of the picture that the human eye has limited ability to fully perceive. It also exploits temporal (over time) and spatial (across a picture) redundancy common in video to achieve better data compression than would be possible otherwise."[1]

a. Intra frame coding

Our video encoder and decoder is modeled after the encoding method that JPEG and MPEG uses. There are six steps that are associated with these image encoding standards:

1. Color Space Transformation
2. Macroblock Creation and Downsampling
3. Block Splitting
4. Discrete Cosine Transformation
5. Quantization
6. Entropy Coding.

The RGB color space is one of the most commonly used color spaces, and is the standard color space used in image display. However, human vision is most receptive to changes in luminosity rather than hue as stated in [10]. Thus, we transform the image from the RGB color space to the

Y'CbCr color space (Y = lumiance, Cb = blue-difference chroma, Cr = red-difference chroma), using the conversion formulae from the JPEG specification as seen in [3]. We can then leverage this fact in the next step to enable space savings.

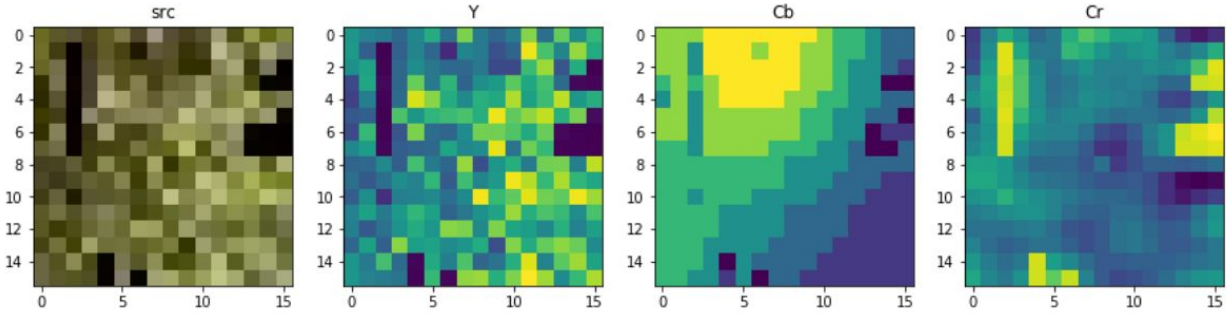


Figure 1: random image divided into Y, Cb, Cr channels. Note that for this and subsequent matrix references, (0,0) is the pixel in the top left corner.

After transforming the color space, we divide the image into 16 pixel by 16 pixel Macroblocks. In order to leverage the properties of the Y'CbCr color space, we keep all of the Y' channel information for each Macroblock, and down sample the Cb and Cr channels by averaging the values of the Cb and Cr values in 4 pixel by 4 pixel subblocks - the end result is then stored in an 8 pixel by 8 pixel block. By downsampling in this manner, we are able to discard 50% of the image data without adversely the data quality. The Y' Macroblock is then split into 4 8 pixel by 8 pixel blocks, labeled Y1 (top left), Y2 (top right), Y3 (bottom left), and Y4 (bottom right).

Having created the 6 blocks associated to the Macroblock, we utilize the 2D Discrete Cosine Transformation to convert the color data into frequencies. The DC term of the transformation is contained within the (0,0) entry of the output matrix, with the associated AC frequencies increasing as the entries move down and to the right to the (7,7). Converting the block entries to the frequency values enables further space savings by taking advantage of the fact that differences at high frequencies are not easily noticeable by human vision, as noted by [1].

The Quantization step capitalizes on this - given a Quantization Matrix Q with fixed values as seen in [5], the matrix of block values B, and a Quality Factor QF, we create a new matrix B', where

$$B'(i,j) = floor\left[\frac{B(i,j)}{QF * Q(i,j)}\right]$$

The matrix Q is arranged such that terms closer to (0,0) are low in value, and terms closer to (7,7) are high in value, thus resulting in the DC term and some low frequency terms preserved, and high frequency terms going to 0, thus allowing space to be saved. The Quality Factor allows the user to trade off saving additional disk space in exchange for decreased quality - raising the QF will result in a larger denominator, which in turn results in more frequency terms evaluating to 0.

Because there are many 0 terms in the matrix, we are able to use entropy coding to represent the matrix in a more compact form:

```
[[528,   4,  -3,   0,   0,   0,   0,   0],
 [  3,   0,  -2,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0]]
```

Figure 2: Matrix following Quantization

```
[528, [0, 4], [0, 3], [2, -3], [1, -2], 'EOB']
```

Figure 3: Matrix following Entropy Coding

As seen in the figures, entropy coding is used to collapse the Matrix into the minimum number of characters necessary to represent it. This is a common technique to conserve data, as described in [6]. As applied here, the first number is the DC term in the (0,0) position. Then, following a zig zag pattern, we count the number of 0's before reaching the next non zero value - in this case, since we first space we visit is (0,1) which is non 0, the second entry of the Entropy Coding [0, 4] represents that there are 0 instances of 0 before encountering the next value, which is 4. We continue in this fashion until we reach the end of the block. Rather than denote the number of zeros following the final non-zero value, we simply insert an 'EOB' character to indicate that there are no further non-zero values in this block.

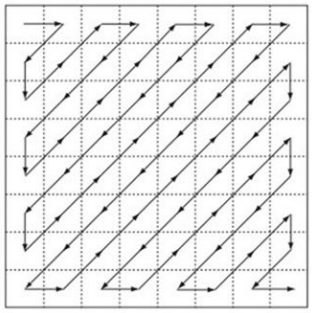


Figure 4: Zig-zag pattern used to encode blocks. The pattern ensures that adjacent blocks in the pattern are close in frequency. (taken from [1])

Huffman coding is then applied to the values in order to compactify the bits used to store the information, the encoding used is taken from the MPEG standard, as we are trying to implement the encoder file MPEG-1 compliant.

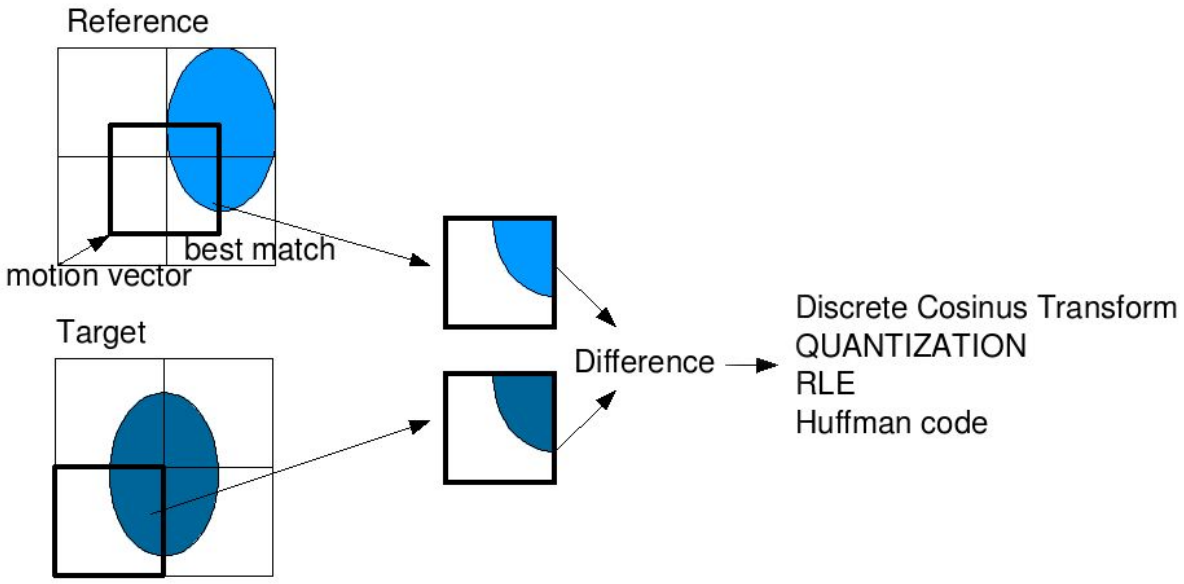
b. Inter frame compression

In order to reduce temporal redundancy, inter frame is exploited as a frame in the video compression stream which is expressed in terms of one or more neighboring frames. We use inter frame prediction to take advantage from temporal redundancy between neighboring frames to enable higher compression rates.

In our plan, we defined our own simple **Group of Pictures (GOP)** structure:
I-frame, the Intra-frame, is the starter of GOP. It can be decoded independently of any other frames and has the best quality as well as the fastest compression but if only use I-frame it will creates relatively large files (although lossless theoretically).
P-frame, the forward-predicted-frame, contains motion-compensated difference info relative to previously decoded pictures. It use inter frame prediction to exploit its reference frame to calculate the frame difference.It can improve compression rate, but increase delay on decoding.
~~**B-frame**~~, the bidirectional-frame, make predictions using both the previous and future frames. But it is not implemented as it requires large data buffers and increase delay on both decoding and encoding.
And our experimental Inter frame compression took the simplest GOP structure for testing:
I P I P I P I P I P ...

The inter frame prediction of P-frame can be implemented in two methods:
Frame-differencing, which simply compares the adjacent frames, use the former one as the reference frame (Intra-frame), and compare the frames in block level. The later ones will only keep the difference data between the blocks from the Inter-frame with those from the reference frame. When decoding, the decoder will identify the kind of Inter-frames and combine it with the reference frame to get the original frame back after the reference frame decoding is done. This method is not strictly a frame prediction method, which means it is not effective enough when it comes to the compression rate, especially when dealing with motion of objects in the video.
Block-matching, which search and track the motion of certain macroblocks and utilize motion vectors matrix to greatly reduce encoded file size. When decoding, the decoder will wrap up the anchor frame with the matrix of motion vector which is in a scale of macro block number of the frame after the reference frame decoding is finished. This method saves bits by sending encoded difference images, which have less entropy as opposed to sending a fully coded frame. But the most computationally expensive and resource extensive operation in compression process is also the motion estimation, which may take time if not handled well in the detailed implementation of algorithm.

If the encoder succeeds in finding a matching block on a reference frame, it will obtain a motion vector pointing to the matched block and a prediction error. Using both elements, the decoder will be able to recover the raw pixels of the block.



Inter-frame prediction process. In this case, there is an illumination change between the block at the reference frame and the block which is being encoded, representing prediction error to this block.[8]

We implemented our block-matching algorithm as a simplified **Three Step Search(TSS)**, which was also based on finding the minimum cost function, and get the Motion Vector matrix, but has limited precision of 4 pixels instead of 1.
Our model runs as follows:

1. Start with search location at center
2. Set step size as half-macroblock-size 4 and search scope size as 7
3. Search 8 locations +/- 4 pixels around location (0,0) and the location (0,0)
4. Pick the one with minimum cost function among the 9 locations searched
5. Set the new search origin to the above picked location

Notice that when Exhaustive Search evaluates cost for 225 macro-blocks, TSS evaluates only for 25 macro blocks, and reduce running time by a factor of 9. While our implementation is even simpler, will be faster in exchange of precision.

Although theoretically our implementation should be fast enough to fulfill the minimum requirement and greatly increase the compression rate. However, the intra frame coding is already consuming quite some time and the block matching also takes time: it takes around 2.2 to 2.5 seconds to encode one P-frame. That means when the inter frame compression is implemented, the total encoding time will no longer satisfy the minimum requirement of the project. The reason may lies on the fact that our implementation is recursively invoking some function which can be really slow in python. Maybe using another language or simply utilize tools like cython can solve this problem, however, we do not have more time for such experiments, and we have to take down the inter frame compression parts temporarily. Instead, our current implementation is I-frame only.

c. Decoding and Viewing

To decode, we essentially run the above steps in reverse - after using the Huffman coding to recover the data, we use the zig-zag pattern to reconstruct the blocks, then use the quality factor originally specified to apply the Inverse Discrete Cosine Transformation to recover Y, Cb, Cr data. Because the Cb, Cr data was subsampled, each pixel in the reconstructed block is used to represent 4 pixels worth of data in the reconstructed Macroblock. We then take the Y, Cb, Cr information, reassemble the image, and finally call ffmpeg once the images have been reconstructed to create the video file.

d. User Interface

To enable users to specify their options for encoding and decoding, a Command Line Interface was created. By calling encode.py, users are able to encode a directory of pictures, or a list of images. Users may optionally customize the output file name and the quality factor through the use of output flags. Similarly, by calling view.py, users are able to decode a binary file created using our encoder into the associated mp4 file. Users may optionally customize the output file name and frames per second rate of the film through the use of output flags.

To facilitate ease of use of the encoder and decoder for the user, we created a simple to use Graphical User Interface that essentially serves as a wrapper for the Command Line Interface calls - users are able to list the files they want to encode or decode using a combo box at the top of the GUI. In addition, they are able to use sliders to adjust the quality factor for encoding, and the fps for decoding, and input an output file name in a provided text box.

C. Features implemented from original proposal and midterm report

Project Requirements:

- The encoder is able to encode 100 JPEG files in under five minutes, with compression rate of between 6-40% depending on user specified inputs.
- The encoder is able to generate an binary file; playback speed varies depending on options selected by user, but 10 frames per second is achievable.
- A Command Line interface has been created. However, the implementation of argparse is such that any optional inputs, including the output flag, have to be specified first. Any images are compressed/reassembled in the order passed in.

Selected Extensions:

- MPEG-1 compliance - elected to not pursue this as noted in the Midterm report, as determined to be more difficult than feasible within the time allotted, instead opted to implement the below extensions.
- Develop GUI -completed, with the options available in the gui matching those able to be input in the command line interface.
- Allow user to adjust parameters on movie creation in order to trade disk space for quality - this has been accomplished by allowing the user to specify a Quality Factor of between 0.1 and 1.5 when passing in the image files.
- Provide three "interesting" real-time video effects during playback - as our decoder hasn't been able to playback real-time video so far, we were unable to determine how to implement this within our current code base.
- Ability to store arbitrary binary files - the encode routine allows users to specify non JPEG image files. However, no testing was done on non JPEG images, and we were unable to make sufficient progress on encoding truly arbitrary files.

II. Code

The project code can be found at https://github.com/jefflin96/EC504-Video-Encoder/tree/master/markshi/code. Because the project code is written in Python to take advantage of existing packages for image processing, there is no need to compile code, just simply download to a local folder. Refer to section III for the list of existing packages utilized, and instructions on how to install any you do not currently possess.

To run the code, there are two options - the Command Line Interface, or the Graphical User Interface:

- Command Line Interface:
    - To run from the command line, navigate to the folder the Python code was downloaded to.
    - To see help, run
        python encode.py -h
        python decode.py -h
    - To encode, run
        python encode.py [--output] [--qf] input
        - output is the name of the output file
        - qf is the Quality Factor (between 0.1 and 1.5 in 0.1 increments)
        - Input is either a directory containing image files, or a list of image files
    - To decode the video to python view, run
        python view.py [--output] [--fps] input
        - output is the name of the output file
        - fps is the frames per second of the output video (5, 10, 15, or 20 fps)
        - Input is the binary file representing the compressed video
- Graphical User Interface
    - Navigate to the folder the Python code was downloaded to, and run python gui.py to launch the GUI
    - To encode:
        - Click Add File(s) and navigate to the folder containing the images you wish to encode into a video - select all images you wish to add and click Open.
        - To remove any files, highlight the file and click Remove File
        - To remove all files, click the Clear All button
        - Adjust the QF slider to the QF value you wish to specify
        - (optional) Specify an output name in the Output field
        - Click Encode
        - The progress bar will indicate the number of images already encoded. A pop up alert will open to indicate when the encoding has been completed.

    - To decode:
        - Click Add File(s) and navigate to the folder containing the binary file you wish to decode into a video. Select the binary file and click Open.
        - To remove the file, highlight the file name and click Remove File.
        - To remove all file(s) entered in the selection window, click Clear All.
        - Adjust the FPS of Decoded Video slider to the Frames per Second of the constructed video.
        - (optional) Specify an output name in the Output field

- Notes:
    - The code uses Python 3. Linux installs, or any others which have multiple versions of Python installed, may need to specify python3 rather than python when calling the code.
    - When using the command line interface, optional inputs must be flagged first, otherwise argparse will ignore these.
    - The default output file name for encoding is out.bin
    - If the user does not specify the encoded output as a .bin file type, the .bin extension is automatically added to the user supplied name
    - The default QF setting is 0.8
    - The default output file name for decoding is decoded_movie.mp4
    - If the user does not specify the decoded output as a .mp4 file type, the .mp4 extension is automatically added to the user supplied name
    - The default FPS setting is 10
    - Currently, the video playback is not launched automatically upon completion of decoding. The mp4 output file will be deposited into the folder that contains the Python code. The mp4 can then be played back using a video player application.
    - The encoder/decoder is not fully MPEG compliant - the Decoder checks for the input ‘EC504’ as the first entry in the binary file. This input is used to flag binary files as having been created using this project’s encoder. If ‘EC504’ is not the first input found in the binary file, the decoder will exit immediately with a message indicating that only files encoded with this project can be passed into the decoder. This is so we can assume a consistent metadata header containing the information to reassemble the blocks in the binary file.

III. Supporting Libraries

A. Libraries required to run code

The following Python libraries are required to run the code:

- openCV
- numpy
- time
- math
- scipy
- Matplotlib
- os
- pillow
- argparse
- sys
- Imghdr
- tkinter
- pickle
- ffmpeg

These libraries may be installed using the Python Installer Package for Python 3 by running pip3 install <package name> from a shell terminal. PIP may be found at: https://pip.pypa.io/en/stable/installing/, if not already installed with your python install.

The video file itself is assembled from the decoded images using ffmpeg, which can be installed from: https://www.ffmpeg.org/

B. Examples of how to use the encoder/decoder

The folders “rocket_images” and “sample_images” contain test images which can be used to run the encoder/decoder. Copy the folder contents to a local folder, and then follow the steps in section II to run the encoder.

To test decoding, take the binary files that were output in the previous step, and again follow the directions in section II to run the decoder. Upon decoding, an mp4 file is created, which can then be viewed using most video playback software.

C. Testing patterns

Unit testing for the encode method was accomplished by the following:

- Passing in no input
- Specifying bad inputs
- Passing in a non image file as only input
- Including a non image file in a list of images
- Passing optional parameter flags after images
- Passing in directory containing images
- Passing in directory containing both images and non images
- Calling the -h flag to bring up the help menu.
- Calling without specifying QF or output file name
- Specifying output file name without .bin extension.

Unit testing for the view method was accomplished by the following:

- Passing in no input
- Specifying invalid input types
- Passing in a non-binary file type
- Passing in a binary file not generated with the correct metadata
- Passing in a directory
- Calling without specifying FPS or output file name
- Specifying output file name without .mp4 extension

Testing of full encoder was performed on the images in the rocket_images and sample_images folder, passing them into the CLI or GUI encoder, and then passing the result into the CLI or GUI decoder/viewer, and viewing the result to check that it matched known examples (up to FPS differences)

IV. Breakdown of Work

The following work on the project was performed by each group member -

Jonathan:

- High Level design and initial research into methods
- Command Line Interface routines
- Added metadata header terms to encoder, decoder.py
- Midterm report clarification of methods being used
- Final report - Sections I except IC portion related to entropy coding, II, IIIA, IIIB, and IIIC portions related to CLI

Jinguang:

- Studied MPEG papers and focus on the interframe compression and GOP structure
- Designed and implemented the block matching function
- Tested the inter-frame(P-frame) compression effectiveness, compared to the certain project requirement.

Jeffrey:

- Initial research into methods for encoding of single image
- Assisted with naive implementation of encoding
- Created GUI with Tkinter interfacing with CLI

Jialiang:

- Study MPEG papers, understand how image encoding and video compression works.
- Design and implement Frame ClassK
- Implement encoder and decoder, analyze intermediate result, and modify each process so that everything works correct
- Implement analyzer.py, comparer.py to show our efficiency result and quality of encoding & decoding in terms of different QF

V. References, Background materials

[1] MPEG-1 Standard
https://en.wikipedia.org/wiki/MPEG-1#Part_5:_Reference_software

[2] JPEG Compression Overview
http://www.massey.ac.nz/~mjjohnso/notes/59731/presentations/jpeg.pdf

[3] Definition of YCbCr:
https://www.pcmag.com/encyclopedia/term/55147/ycbcr

[4] Formulae for RGB to YCbCr transformation:
https://en.wikipedia.org/wiki/YCbCr#JPEG_conversion

[5] Discrete Cosine Transform, and applications in image processing (as archived by the Internet Archive, as the original URL is no longer accessible):
https://web.archive.org/web/20150711105353/http://wisnet.seecs.nust.edu.pk/publications/tech_reports/DCT_TR802.pdf

[6] Quantization Matrix
https://en.wikipedia.org/wiki/MPEG-1#Quantization

[7] Entropy Encoding
http://www.pcs-ip.eu/index.php/main/edu/8

[8] Details of inter frame
https://en.wikipedia.org/wiki/Inter_frame

[9] Ways to implement block-matching algorithm
https://en.wikipedia.org/wiki/Block-matching_algorithm

[8] Past EC504 Encoder Project - Leveraged Readme to aid in high level design and understanding project specifications. Leveraged command line entries as examples of how argparse works, how to make calls on directories. Leveraged their implementation of Huffman encoding within our project.
https://github.com/appletonbrian/EC504-video-encoder

[10] Overview of Chroma Subsampling.
https://en.wikipedia.org/wiki/Chroma_subsampling