Video Encoder and Decoder

Jonathan Chamberlain, Jinguang Guo
Jeffrey Lin, Jialiang Shi

Introduction

Our Video Encoder is Modeled after the standard method of encoding that both JPEG Extension and MPEG Extension use.

The process of encoding an individual frame can be broken down into 6 steps:
First, split the image into a series of 16 pixel by 16 pixel Macroblocks, then:

1. Split Macroblock into YCbCr channels
2. Divide into 4 blocks, and subsample Cb, Cr (keeping all blocks in Y channel)
3. Apply Discrete Cosine Transformation (DCT) and Quantization with Quality Factor
4. Zig Zag on resulting Block to <DC,<Run,Level>,EOB> pattern
5. Encode into bitstream using standard MPEG huffman table

Encoding a Frame - Macroblocking and Blocking

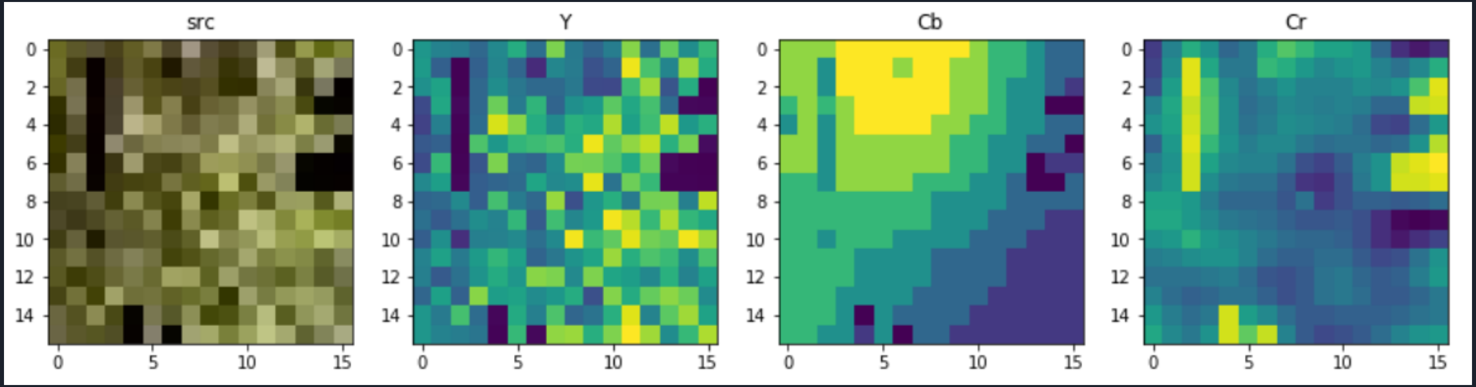
1. Split block into YCbCr channel

Commonly, colors are expressed by their Red, Green, Blue values.

However, human vision is more susceptible to changes in brightness than in hue. Thus we convert the color data to the YCbCr spectrum: Luma (Y), Blue difference Chroma (Cb), Red difference Chroma (Cr).

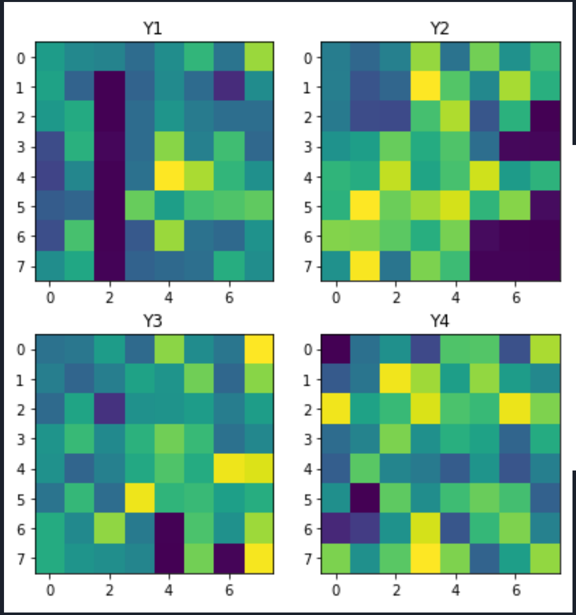
This enables us to more efficiently store data without large loss of quality. We can subsample the Cb and Cr components from each Macroblock, keep one block’s worth of Cb and Cr data and use that to estimate the color of the remaining block, while keeping all 4 blocks of Y data.

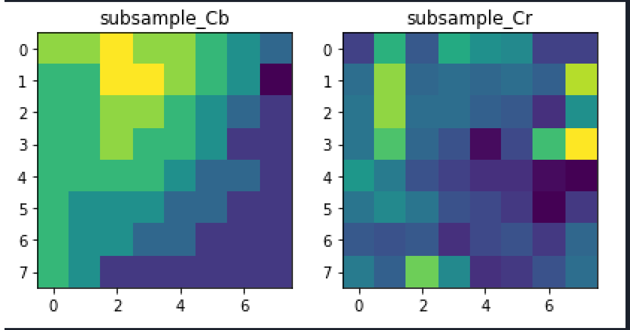
Encoding a Frame - YCbCr Spectrum



Encoding a Frame - Diving and Subsampling

2. Divide Y into 4 blocks, and subsample Cb, Cr - **Discard 50% of original img data.**





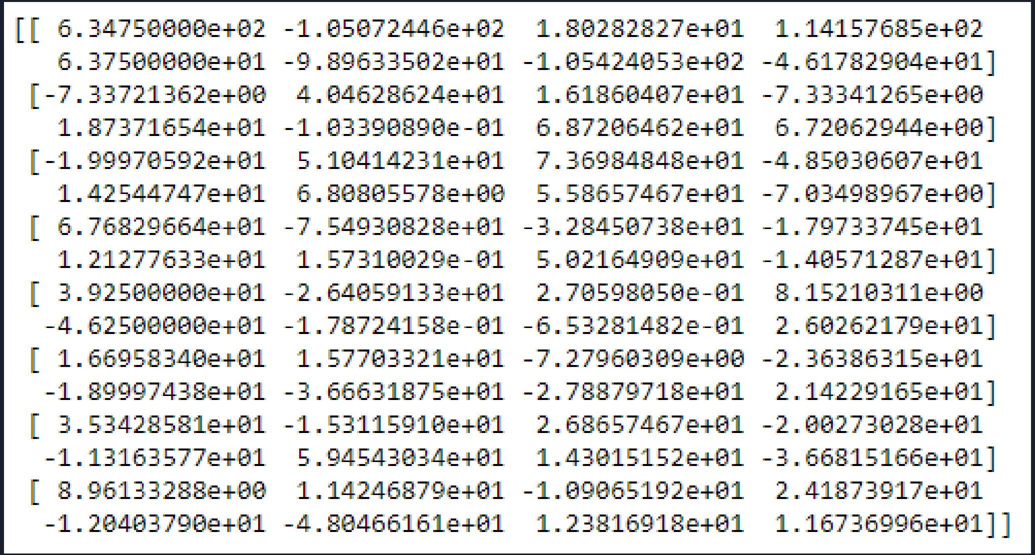
Encoding a Frame - DCT and Quantization

Each of the remaining 6 blocks (the four Y blocks and the single Cb and Cr blocks for each Macroblock) are transformed using the Discrete Cosine Transform to map the frequencies present in each block.

We are able to use this information in order to compress data further - because human vision is less able to discern differences in high frequencies, we can truncate this data by performing an element wise division using a Quantization Matrix, and then saving the floor of the result.

We can further trade off quality for disk space through the use of a Quality Factor (aka Quantization Factor) - the Quantization Matrix utilized for a conversion is given by the QF multiplied by the QM. The larger the QF value, the more disk space conserved at the expense of less data being available to convert afterwards.

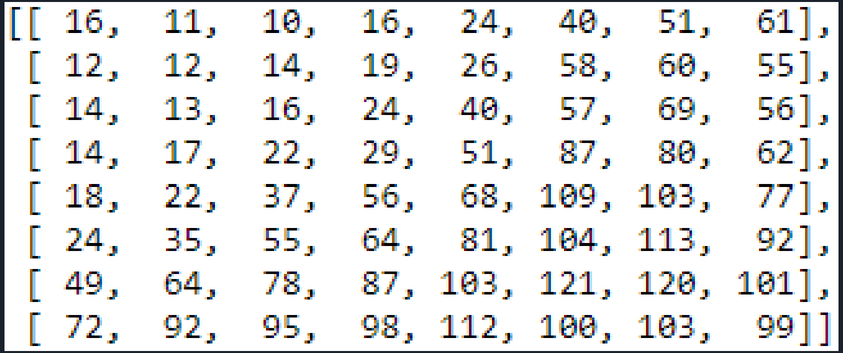
Encoding a Frame - DCT Result



```
[[ 6.34750000e+02 -1.05072446e+02  1.80282827e+01  1.14157685e+02
   6.37500000e+01 -9.89633502e+01 -1.05424053e+02 -4.61782904e+01]
 [-7.33721362e+00  4.04628624e+01  1.61860407e+01 -7.33341265e+00
   1.87371654e+01 -1.03390890e-01  6.87206462e+01  6.72062944e+00]
 [-1.99970592e+01  5.10414231e+01  7.36984848e+01 -4.85030607e+01
   1.42544747e+01  6.80805578e+00  5.58657467e+01 -7.03498967e+00]
 [ 6.76829664e+01 -7.54930828e+01 -3.28450738e+01 -1.79733745e+01
   1.21277633e+01  1.57310029e-01  5.02164909e+01 -1.40571287e+01]
 [ 3.92500000e+01 -2.64059133e+01  2.70598050e-01  8.15210311e+00
  -4.62500000e+01 -1.78724158e-01 -6.53281482e-01  2.60262179e+01]
 [ 1.66958340e+01  1.57703321e+01 -7.27960309e+00 -2.36386315e+01
  -1.89997438e+01 -3.66631875e+01 -2.78879718e+01  2.14229165e+01]
 [ 3.53428581e+01 -1.53115910e+01  2.68657467e+01 -2.00273028e+01
  -1.13163577e+01  5.94543034e+01  1.43015152e+01 -3.66815166e+01]
 [ 8.96133288e+00  1.14246879e+01 -1.09065192e+01  2.41873917e+01
  -1.20403790e+01 -4.80466161e+01  1.23816918e+01  1.16736996e+01]]
```

After apply our DCT on the Y1 block, we are left with a matrix of frequencies

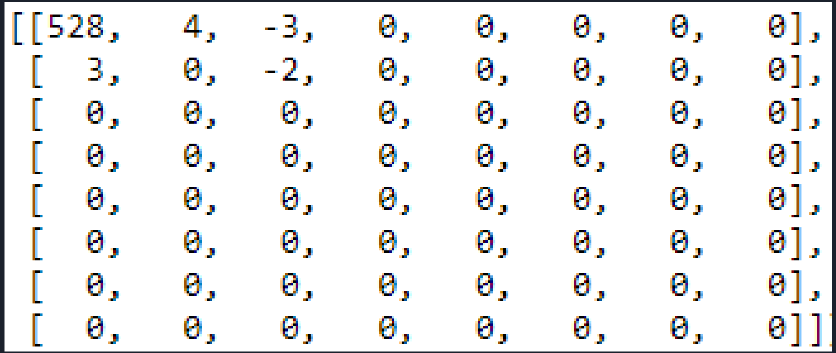
Encoding a Frame - Quantization Matrix



```
[[ 16,  11,  10,  16,  24,  40,  51,  61],
 [ 12,  12,  14,  19,  26,  58,  60,  55],
 [ 14,  13,  16,  24,  40,  57,  69,  56],
 [ 14,  17,  22,  29,  51,  87,  80,  62],
 [ 18,  22,  37,  56,  68, 109, 103,  77],
 [ 24,  35,  55,  64,  81, 104, 113,  92],
 [ 49,  64,  78,  87, 103, 121, 120, 101],
 [ 72,  92,  95,  98, 112, 100, 103,  99]]
```

We multiply this matrix with a Quality Factor and apply element wise division on our resulting DCT matrix to compress the data even more.

Encoding a Frame - Quantization Result

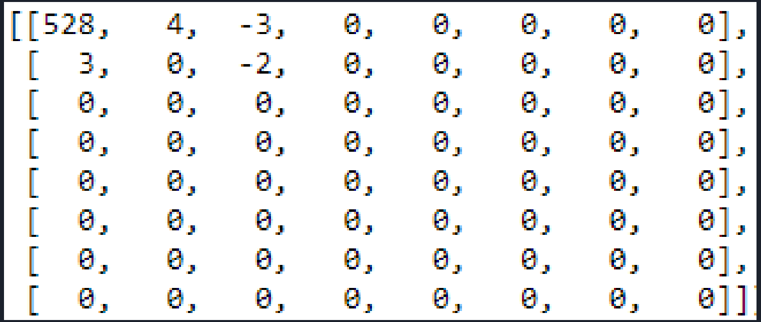


```
[[528,   4,  -3,   0,   0,   0,   0,   0],
 [  3,   0,  -2,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,   0],
 [  0,   0,   0,   0,   0,   0,   0,  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

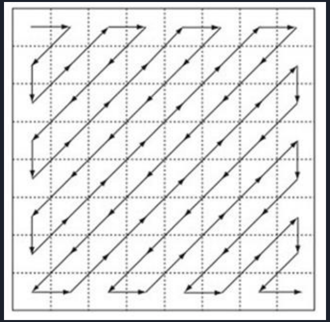
What was once 64 unique Y, Cb, or Cr Values, are now 528,4,3,2, along with zeros.

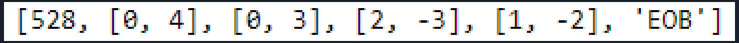
Encoding a Frame- Zig Zag

After Quantization, we can see that the resulting matrix has a large number of 0s. This will be common for many blocks. To reduce the amount of space these 0’s would populate, we can use a Zig Zag to count the 0’s in between non-zero values.



```
[[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]]
```

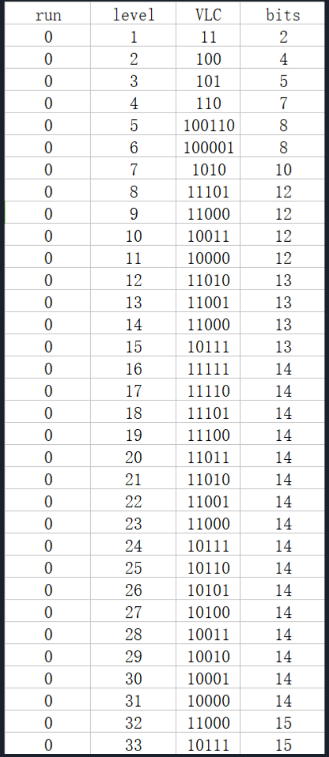




```
[528, [0, 4], [0, 3], [2, -3], [1, -2], 'EOB']
```

Resulting Array After Zig Zag

Encoding a Frame- bitstream



run	level	VLC	bits
0	1	11	2
0	2	100	4
0	3	101	5
0	4	110	7
0	5	100110	8
0	6	100001	8
0	7	1010	10
0	8	11101	12
0	9	11000	12
0	10	10011	12
0	11	10000	12
0	12	11010	13
0	13	11001	13
0	14	11000	13
0	15	10111	13
0	16	11111	14
0	17	11110	14
0	18	11101	14
0	19	11100	14
0	20	11011	14
0	21	11010	14
0	22	11001	14
0	23	11000	14
0	24	10111	14
0	25	10110	14
0	26	10101	14
0	27	10100	14
0	28	10011	14
0	29	10010	14
0	30	10001	14
0	31	10000	14
0	32	11000	15
0	33	10111	15

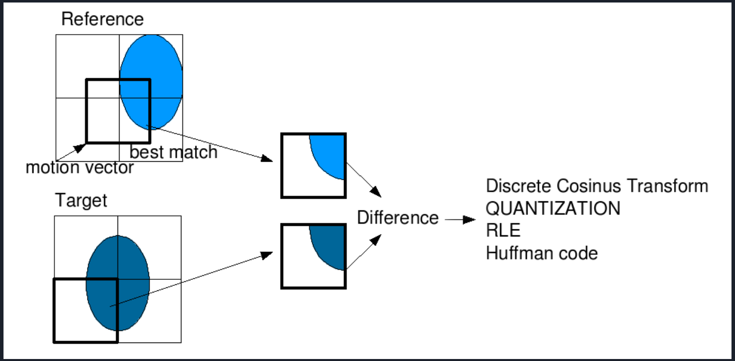
Then we encode each <DC, <Run,Level>, 'EOB'> to bitstream using the Huffman table provided by MPEG-1 (3) standard.

(The number of horizontal and vertical metablocks and the original QF are encoded in a metadata section of the binary file.)

Interframe compression

Goal: To Reduce Temporal Redundancy

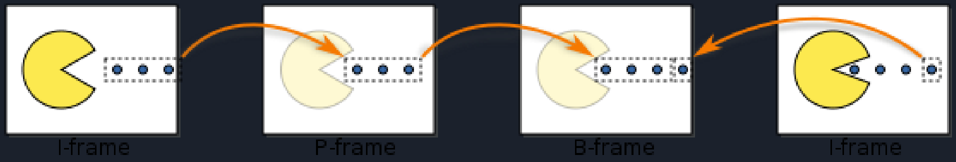
- **Frame-difference:**
  - Simply compare the adjacent frames, and check whether there are differences.
- **Block-matching:**
  - Search and Track the motion of certain blocks.
  - Utilize motion vectors to reduce encoded file size.



Interframe compression

GOP(group of pictures) structure:

- **I-frame:**
  - Intra-frame; starter of GOP; can be decoded independently of any other frames; best quality fastest compression but creates large files (lossless, in theory).
- **P-frame:**
  - (Forward-)predicted-frame; contains motion-compensated difference info relative to previously decoded pictures; use motion vectors on each macroblock of its anchor frame to calculate the frame difference; improve compression but increase delay on decoding
- **B-frame** (not implemented):
  - bidirectional-frame; make predictions using both the previous and future frames; requires large data buffers; increase delay on both decoding and encoding.

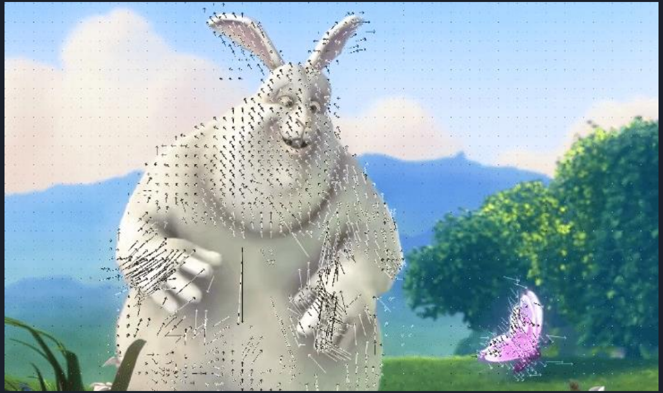


Interframe compression

Motion Vector:

Save 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.

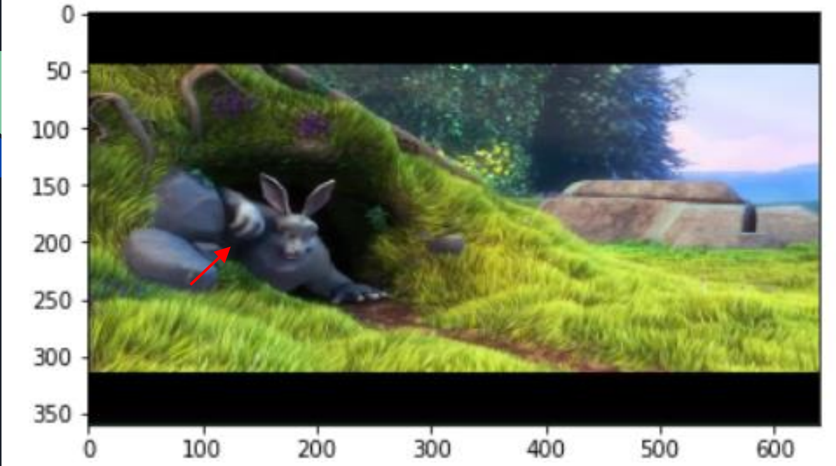
- Here we implement our block-matching algorithm as a classic *Three Step Search*(TSS), which based on finding the minimum cost function, and get the Motion Vector matrix.



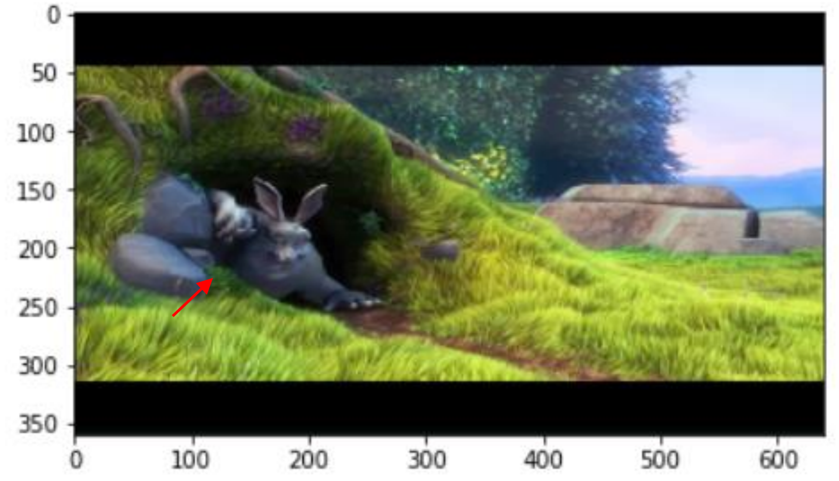
Interframe compression

Three Step Search(TSS):

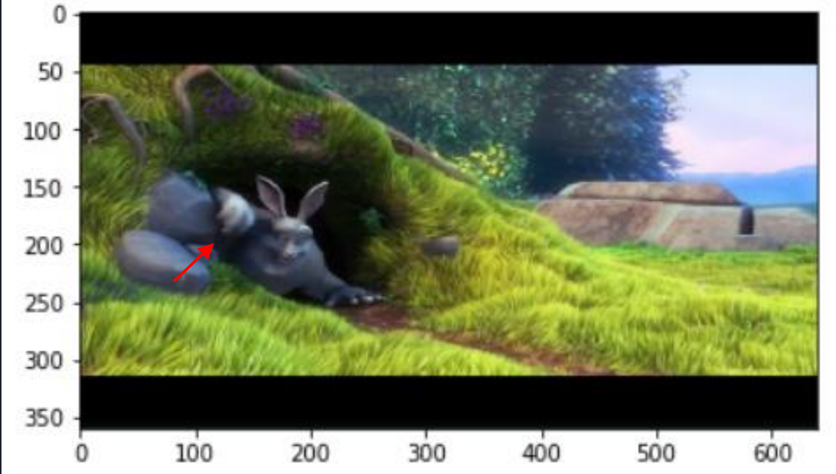
- Our implementation can be described as follows
    1. Start with search location at center
    2. Set step size as half-macroblock-size 4 , search scope size 7
    3. Search 8 locations +/- 4 pixels around location (0,0) and the location (0,0)
    4. Pick the one with minimum` cost function among the 9 locations searched
    5. Set the new search origin to the above picked location
    6. Set the new step size as 2
    7. Repeat the search procedure until step size = 1
- While Exhaustive Search evaluates cost for 225 macro-blocks, TSS evaluates only for 25 macro blocks, it reduce running time by a factor of 9.



Frame 1(Original)



Wrapped up Frame 2 (from P-frame)



Frame 2(Original)

Interframe compression

Drawback:

I-, P- (and B-) frame sequences can give very high compression, but also increase the coding/decoding delay significantly

Our current implementation of motion vector searching takes:

- P-frame: Encoding less than 2.5 s/frame

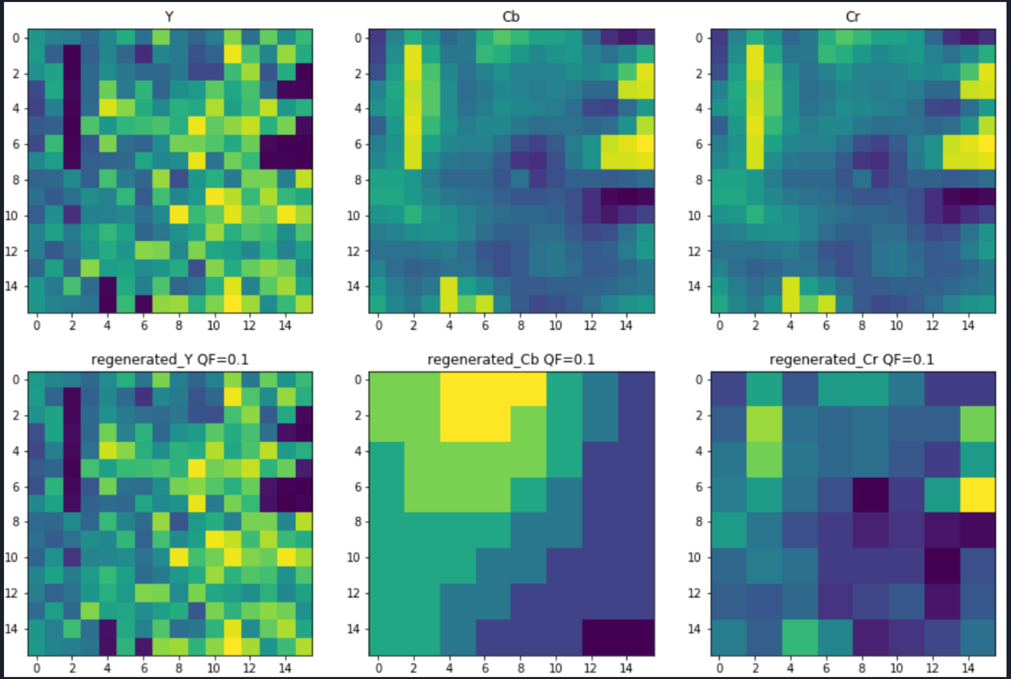
For now, we built our proto-mpeg system with I-frames only, better interframe compression requires better optimized algorithm.

Decoding and Viewing

To Decode and view the video, we essentially perform the steps for Encoding in reverse

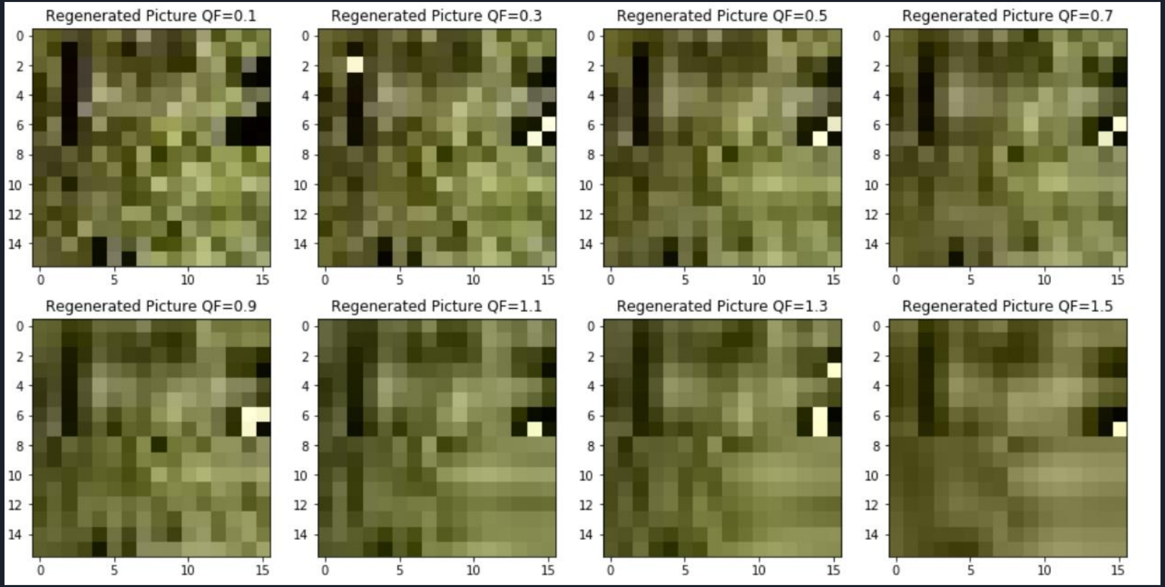
- To reconstruct the images from the encoded binary file, we must:
  - UnZigZag the Array into a 8x8 Matrix
  - Reverse the Quantization with the QF stored in a header
  - Apply the Inverse DCT
  - Reassemble image using YCbCr blocks
- Assembling these images into a video file which can then be played back.

Decoding and Viewing



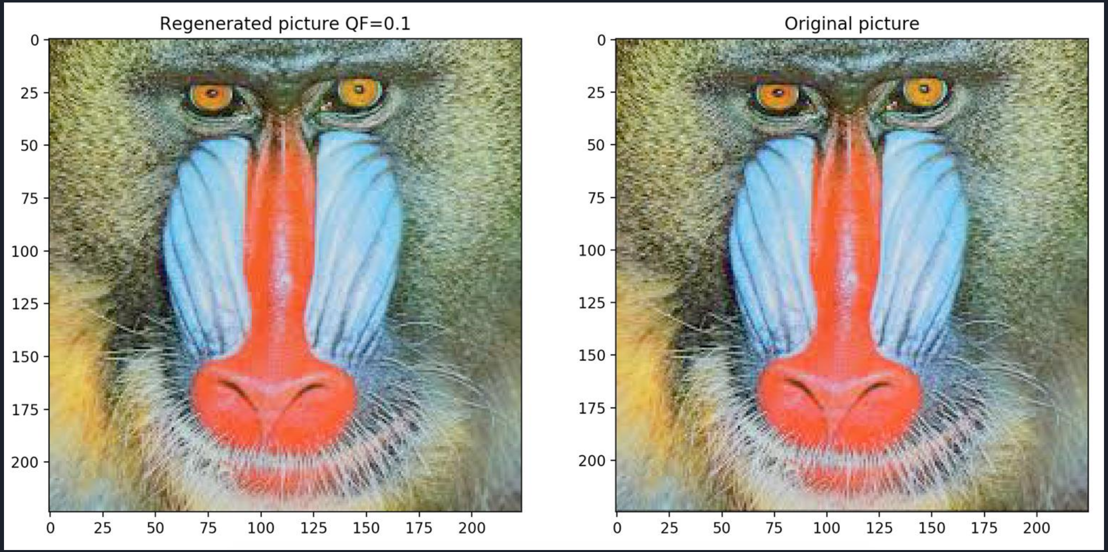
Compare result of regenerated YCbCr channels with the original

Decoding and Viewing



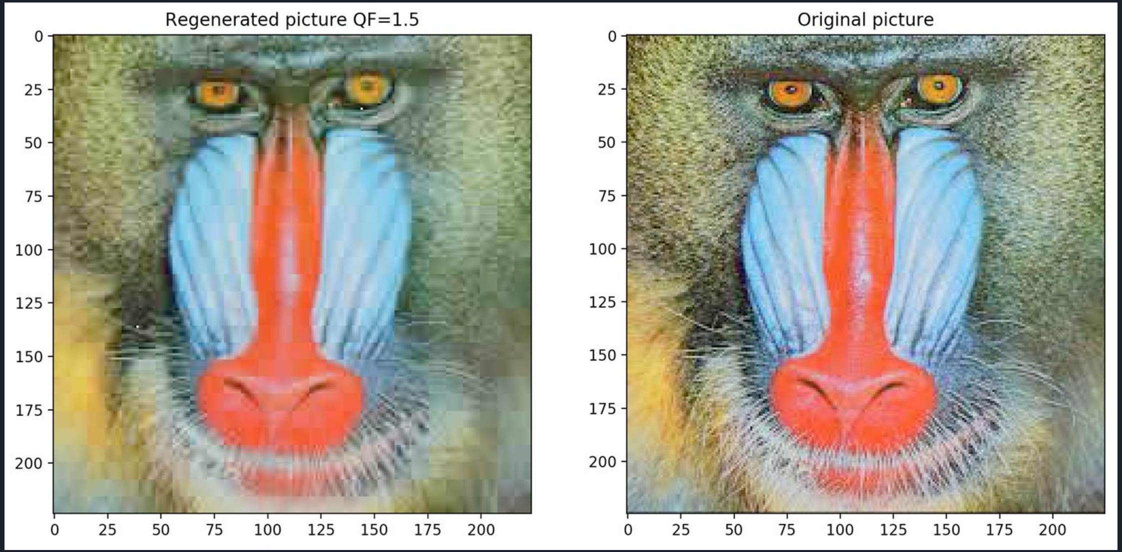
Compare result of regenerated block with the original block using different Quality Factors

Decoding and Viewing



Compare result of regenerated images with the original image using different Quality Factors

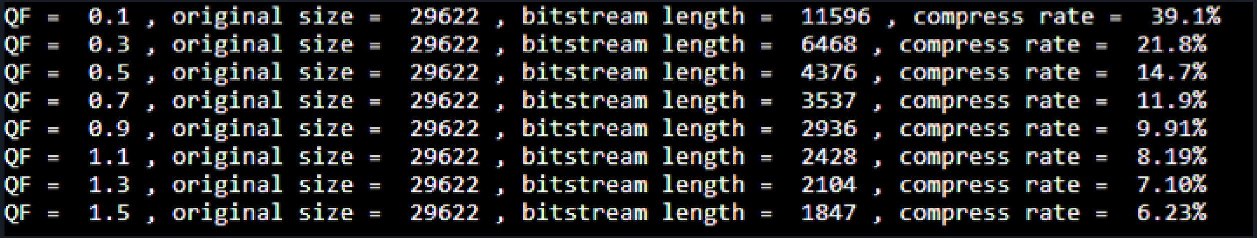
Decoding and Viewing



Compare result of regenerated images with the original image using different Quality Factors

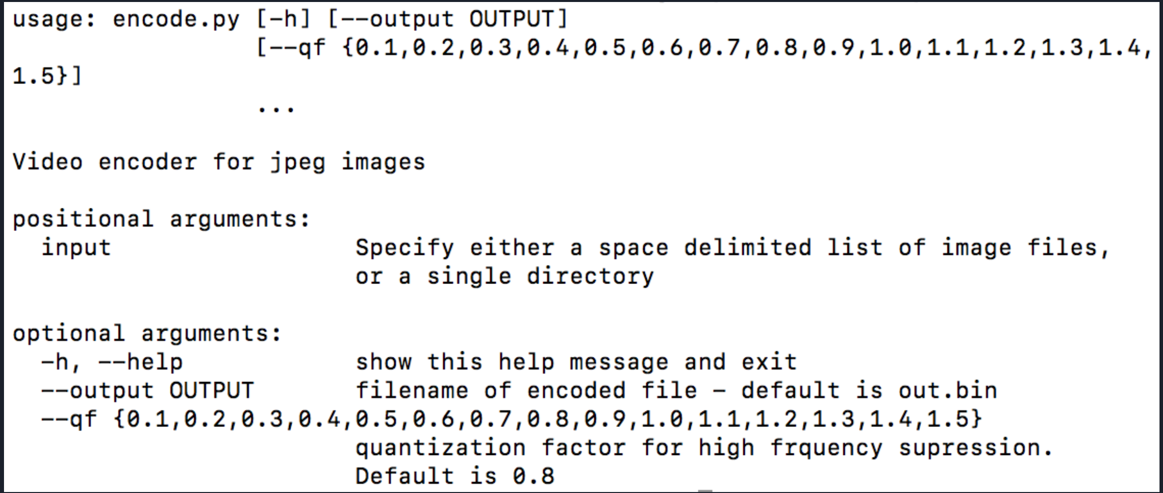
Efficiency Results

- Average encoding time is 2.25s/img, QF=0.1, 640x360
- Average decoding time is 2.21s/img, QF=0.1, 640x360
- Compress rate



```
QF =  0.1 , original size =  29622 , bitstream length =  11596 , compress rate =  39.1%
QF =  0.3 , original size =  29622 , bitstream length =  6468 , compress rate =  21.8%
QF =  0.5 , original size =  29622 , bitstream length =  4376 , compress rate =  14.7%
QF =  0.7 , original size =  29622 , bitstream length =  3537 , compress rate =  11.9%
QF =  0.9 , original size =  29622 , bitstream length =  2936 , compress rate =  9.91%
QF =  1.1 , original size =  29622 , bitstream length =  2428 , compress rate =  8.19%
QF =  1.3 , original size =  29622 , bitstream length =  2104 , compress rate =  7.10%
QF =  1.5 , original size =  29622 , bitstream length =  1847 , compress rate =  6.23%
```

CLI / GUI



```
usage: encode.py [-h] [--output OUTPUT]
                 [--qf {0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.1,1.2,1.3,1.4,
1.5}]
                 ...

Video encoder for jpeg images

positional arguments:
  input                 Specify either a space delimited list of image files,
                        or a single directory

optional arguments:
  -h, --help            show this help message and exit
  --output OUTPUT       filename of encoded file - default is out.bin
  --qf {0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.1,1.2,1.3,1.4,1.5}
                        quantization factor for high frquency supression.
                        Default is 0.8
```

Typical usage:
python encode.py --output output.bin --qf 0.2 ./example_pics

CLI / GUI

Using the tkinter package, we created a rudimentary GUI to facilitate user friendly interaction with the encoder and decoder.

In addition, encode and view exist as routines that wrap around the encoder and decoder functions, to allow the user to enter the inputs directly through the command line shell.

DEMO

Future Improvements

1. Better Motion Vector searching algorithm and GOP implementation to compress the files in a higher rate
2. Adding more interesting effects such as filters
3. Convert Python Code To C/C++ code to Reduce Running Time
4. Implement real-time playback after Reducing Running Time

Questions?

Thank You