CSE 408 Multimedia Information Systems

Phase #3

Group No: 6

Gabriel Crispino
Derek Norman
James Perry
Jake Pruitt
Sreenivas Shenoy

Abstract

Python and OpenCV were used to create a histogram of quantized data from a video. DWT and DCT were applied separately on blocks of each frame from an input video, and the n most significant components were selected. An n-bin histogram is also created by quantizing the differences for each block from the current frame to the next frame. DWT is applied to each whole frame and the most significant wavelet components are found. All of these features output results to a file. A program was also written that takes a video, a frame id, n and m values and matches the ten most similar frames for each of the five features.

Introduction

The goals for this project were to experiment with representing video data, extracting features and retrieving videos from text files. Task I takes a video and divides each frame into eight by eight regions. There are four subtasks for task 1. Two subtasks create n-bin histograms, one is based on gray scale and the other is on the difference from the next frame. The other two take the Y component of the video, and applies DCT and DWT to the blocks.

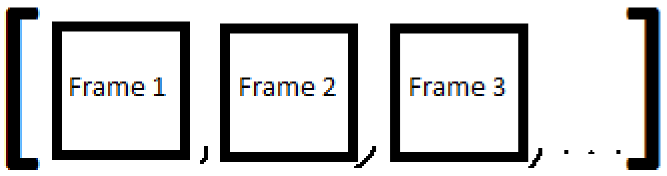
Task two applies DWT to a video frame by frame. An assumption that was made was that the input video is a square. This makes dividing the video frame easier. The final task finds 10 frames that are the most similar to a given frame id after running the first two tasks on a video. Scripts are used to run task 3.

Implementation

Task 1 (A) - Block Quantization and Histogram Representation

Assumptions: Quantization will be done on the block level, so that values per region are binned in n buckets within the range of the region's min and max values.

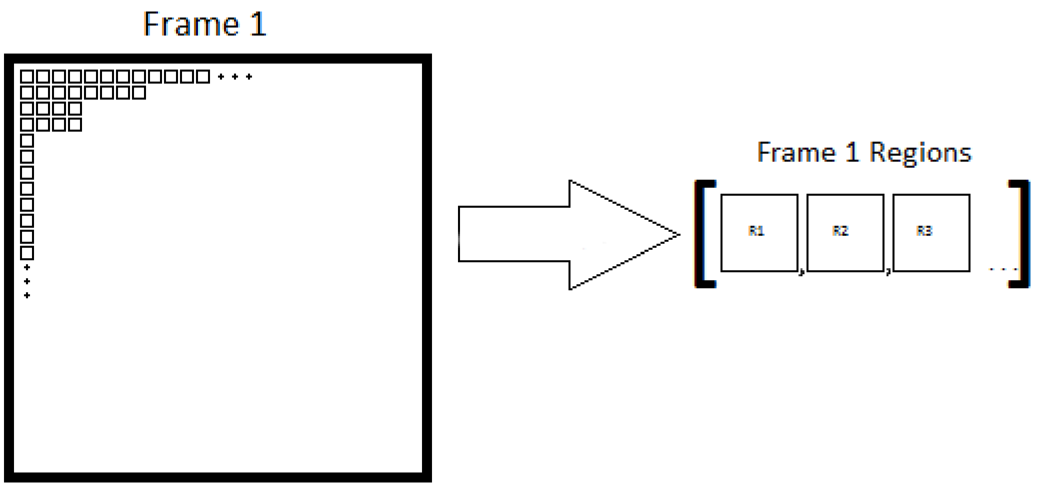
Task 1(a) will prompt a user to select a video for processing, as well as an n value to use for quantization and displaying the histogram. When the video is read into memory it is stored as an array of arrays, where each index is a frame from the video and the length of the array is the total number of frames in the video.



The next step is to work on each of these frames by dividing them into 8x8 blocks and quantizing those blocks. This was done by iterating through the array of frames and working on each frame 1 by 1. In a current iteration the frame is first divided into 8x8 blocks using numpy array slicing:

```
frame[block_x:block_x+8, block_y:block_y+8]
```

which takes the frame and returns the next 8x8 block for that frame. In this format the nth index is the nth block in respect to processing horizontally.



The next step in quantizing is to iterate through this new list of blocks and begin quantizing, and conform to the specific output:

<frame id, block coord, gray instance id, num pixels>

All the data needed for output is available or can be derived inside the q*uantize_block* function. This function, q*uantize_block*, takes the block, number of bins (n value), the frameNumber (given by the index of the frame in the frame array), and the (x,y) coordinates of the block. The first step in this quantization process is to calculate the min and max of the block, and subsequently the range of values for the block given by:

$$valueRange = maxValue - minValue$$

Once this range is calculated the number of values per bin is given by:

$$valuesPerBin = \frac{valueRange}{bins}$$

In the case that the valuesPerBin is < 0, which can happen when the range is smaller than the number of bins, the valuesPerBin is defaulted to 1, which is the smallest number of bins mappable. A dictionary is then created that contains keys for the range of values in order to keep track of the number of occurrences of each value.

After the value is quantized it is used as the key to the dictionary in order to increment the occurrence number. Once all values have been quantized, each key,value pair of the dictionary, ignoring occurrences/values of 0, are iterated and used to build the formatted output. Building the output string(s) is done in the following manner:

```
"< " + str(frameNum) + ", (" + str(x) + ", " + str(y) + ") ," + str(key) + ", " + str(occurrencesInFrame[key]) + " >\n"
```

A histogram is then computed for the current block using all the quantized values and returned with the above formatted output for storing in a block dictionary.

So that the final output is of the form:

<frame id, block coord, gray instance id, num pixels>
and
block_hist

This string contains an entry for each gray instance id for each blocks, and is concatenated for all the blocks of each frame that have been processed. *block_hist* is a histogram that represents the block, one is calculated for each blocks of each frame. The block histograms are stored in *block_hist_dict*, where the key is the (x,y) coordinate of the block, and the value is the histogram for that block. This *block_hist_dict* dictionary is then stored in the *frame_block_dict* dictionary which uses the frame number as they key to access the value which is the *block_hist_dict* for that frame (containing all the block histograms). Finally, when all frames/blocks have been processed this string and frame dictionary are returned, the output file with name format of:

<frame id, block coord, gray instance id, num pixels>

is created using *result* and the *display_histogram* function is called.

Display_histogram uses the quantized values (stored in the frame occurrences dictionary) to display a histogram by adding up all the occurrences for each value then using matplotlib to build the histogram. Once this is done it saves the histogram to a file for later viewing.

Task 1(b) - Discrete Cosine Transform

Task 1(b) will prompt a user to select a video and the number of Discrete Cosine Components which need to be output to the file. The video is split into frames and and each frame is further split into 8 x 8 frame blocks. ‘n’ number of significant components are identified for each of these frame blocks, where n is user input.
The discrete cosine transform helps separate the image into parts of differing importance.
The basic idea of DCT can be depicted with the help of the following figure.

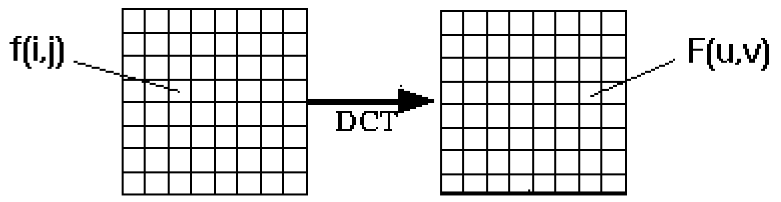


Image Courtesy : http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html

Where f is the input 8x8 pixel block, F is the representation of the block in frequency domain.
The general equation for a 2D (*N* by *M* image) DCT is defined by the equation,

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)\,C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if x is 0, else 1 if } x > 0$$

In our project, the DCT Implementation is done in two ways. The video frames were converted to grayscale and value 128 is subtracted from each of the pixel intensities to normalize the values from -128 to +127.

1. Naive implementation of DCT

A direct implementation of the DCT formula. The pseudocode is provided below,

Input : A frame block of size 8 x 8
Output : Discrete Cosine Components of the frame block

```
for i from 0 to 7:
do :
    for j from 0 to 7:
    do:
        temp = 0.0
            for x from 0 to 7:
            do:
            for y in range(0,8):
            do:
                temp = temp +  cos(pi*i *(2*x+1)/16) * cos(pi*j *(2*y+1)/16) * input[x,y]
              end for
            end for
    if i= 0 and j = 0:
       temp = temp/8
    else if either i == 0 or j==0:
       temp = temp/(4 *sqrt(2))
    else:
       temp = temp/4
    DCT[i][j] = temp
return DCT
```

The naive implementation of DCT algorithm has a time complexity of O(n^4) and therefore extremely slow.

2) Faster implementation of DCT.

A faster implementation of DCT uses precomputed values and two matrix multiplications. Consider the matrix U given below,

$$U = \frac{1}{2}\begin{bmatrix}
\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\
\cos\frac{\pi}{16} & \cos\frac{3\pi}{16} & \cos\frac{5\pi}{16} & \cos\frac{7\pi}{16} & \cos\frac{9\pi}{16} & \cos\frac{11\pi}{16} & \cos\frac{13\pi}{16} & \cos\frac{15\pi}{16} \\
\cos\frac{2\pi}{16} & \cos\frac{6\pi}{16} & \cos\frac{10\pi}{16} & \cos\frac{14\pi}{16} & \cos\frac{18\pi}{16} & \cos\frac{22\pi}{16} & \cos\frac{26\pi}{16} & \cos\frac{30\pi}{16} \\
\cos\frac{3\pi}{16} & \cos\frac{9\pi}{16} & \cos\frac{15\pi}{16} & \cos\frac{21\pi}{16} & \cos\frac{27\pi}{16} & \cos\frac{33\pi}{16} & \cos\frac{39\pi}{16} & \cos\frac{45\pi}{16} \\
\cos\frac{4\pi}{16} & \cos\frac{12\pi}{16} & \cos\frac{20\pi}{16} & \cos\frac{28\pi}{16} & \cos\frac{36\pi}{16} & \cos\frac{44\pi}{16} & \cos\frac{52\pi}{16} & \cos\frac{60\pi}{16} \\
\cos\frac{5\pi}{16} & \cos\frac{15\pi}{16} & \cos\frac{25\pi}{16} & \cos\frac{35\pi}{16} & \cos\frac{45\pi}{16} & \cos\frac{55\pi}{16} & \cos\frac{65\pi}{16} & \cos\frac{75\pi}{16} \\
\cos\frac{6\pi}{16} & \cos\frac{18\pi}{16} & \cos\frac{30\pi}{16} & \cos\frac{42\pi}{16} & \cos\frac{54\pi}{16} & \cos\frac{66\pi}{16} & \cos\frac{78\pi}{16} & \cos\frac{90\pi}{16} \\
\cos\frac{7\pi}{16} & \cos\frac{21\pi}{16} & \cos\frac{35\pi}{16} & \cos\frac{49\pi}{16} & \cos\frac{63\pi}{16} & \cos\frac{77\pi}{16} & \cos\frac{91\pi}{16} & \cos\frac{105\pi}{16}
\end{bmatrix}$$

Let A be the 8 x 8 frame block for which we need to find out the DCT components. The components can be found out easily using the formula, DCT = U A U^T where U^T stands for transpose of the matrix U. This computation makes the DCT components identification faster. Significant components can be chosen in different ways. The components with largest magnitude was chosen at first. Whereas the logic had some flaws because, choosing the components in such a way doesn't help us compare frames present in the video. zig-zag scanning which is usually done in JPEG, orders the DCT coefficients into an efficient manner for the run length coding phase and we have used the same structure to find the significant components
The zig-zag format as shown below,

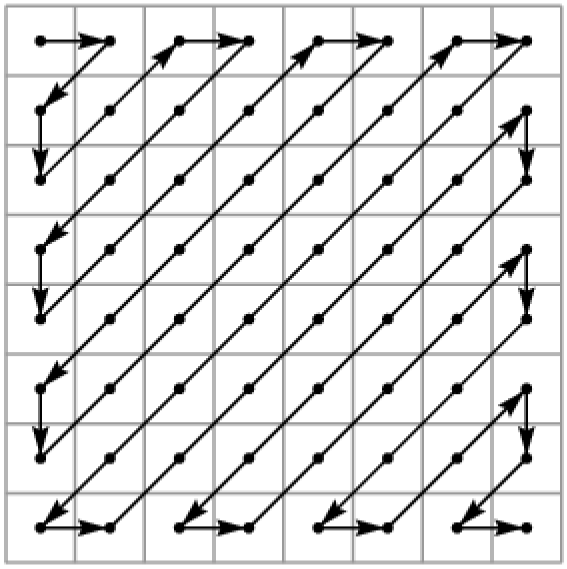


Image Courtesy : Stackoverflow

n- significant components are considered as the first n components which appear after scanning it in the zigzag format shown above. Once the components are identified, the frequency component is output as follows,

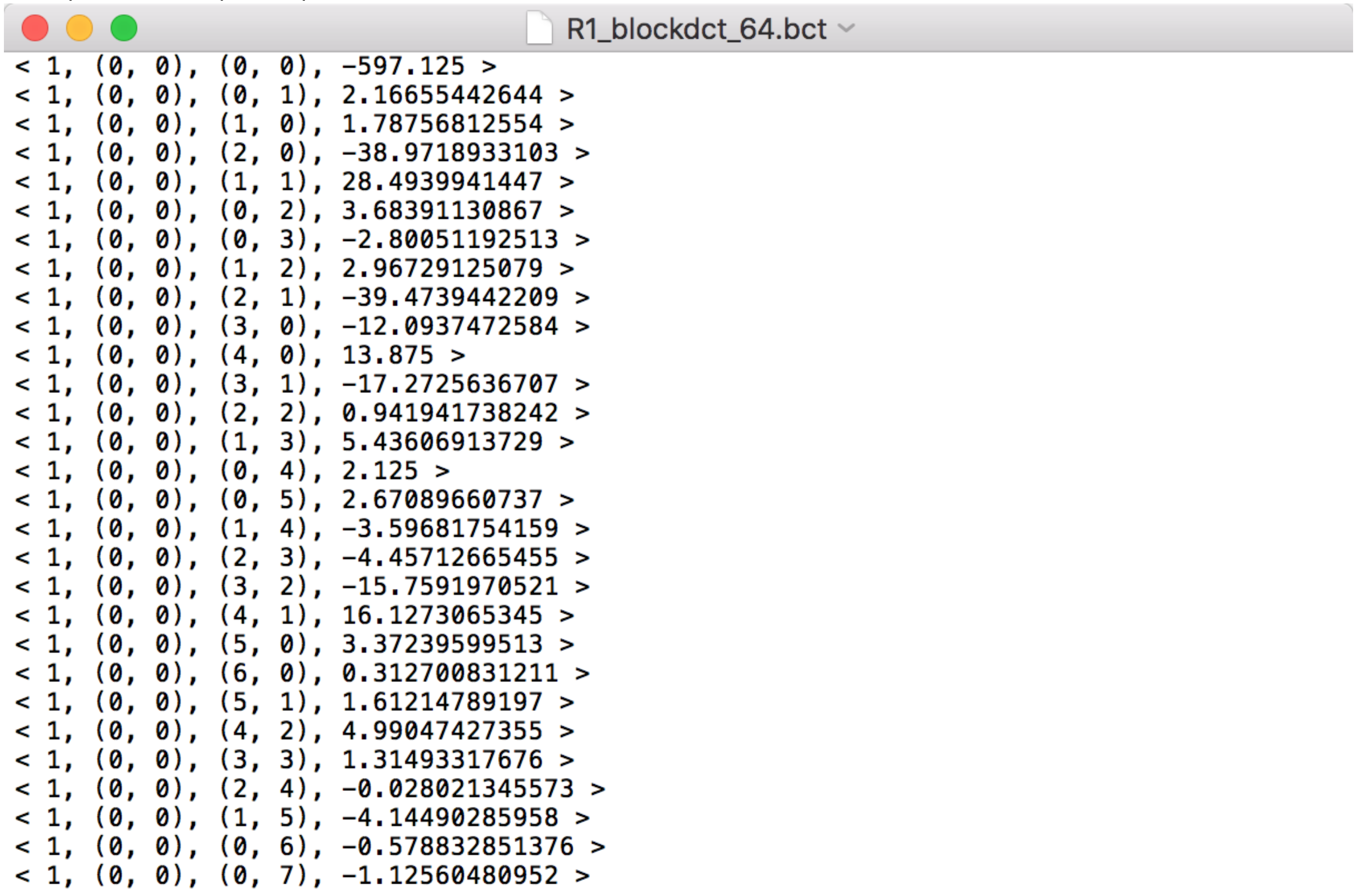
<Frame Number,Frame Block Co-ordinates, Pixel Co-ordinate,Component Value>

For eg, the first entry on the output file for a video in the format <1,(0,0),00,361.875> means,
1 - Represents the frame number
(0,0) - Represents the frame block.
(0,0) - Represents the frequency component id
361.875 - Represents the frequency component value.

Frequency Component ids are referred in the following format,

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,3)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

A snapshot of sample output is as shown below,



```
< 1, (0, 0), (0, 0), -597.125 >
< 1, (0, 0), (0, 1), 2.16655442644 >
< 1, (0, 0), (1, 0), 1.78756812554 >
< 1, (0, 0), (2, 0), -38.9718933103 >
< 1, (0, 0), (1, 1), 28.4939941447 >
< 1, (0, 0), (0, 2), 3.68391130867 >
< 1, (0, 0), (0, 3), -2.80051192513 >
< 1, (0, 0), (1, 2), 2.96729125079 >
< 1, (0, 0), (2, 1), -39.4739442209 >
< 1, (0, 0), (3, 0), -12.0937472584 >
< 1, (0, 0), (4, 0), 13.875 >
< 1, (0, 0), (3, 1), -17.2725636707 >
< 1, (0, 0), (2, 2), 0.941941738242 >
< 1, (0, 0), (1, 3), 5.43606913729 >
< 1, (0, 0), (0, 4), 2.125 >
< 1, (0, 0), (0, 5), 2.67089660737 >
< 1, (0, 0), (1, 4), -3.59681754159 >
< 1, (0, 0), (2, 3), -4.45712665455 >
< 1, (0, 0), (3, 2), -15.7591970521 >
< 1, (0, 0), (4, 1), 16.1273065345 >
< 1, (0, 0), (5, 0), 3.37239599513 >
< 1, (0, 0), (6, 0), 0.312700831211 >
< 1, (0, 0), (5, 1), 1.61214789197 >
< 1, (0, 0), (4, 2), 4.99047427355 >
< 1, (0, 0), (3, 3), 1.31493317676 >
< 1, (0, 0), (2, 4), -0.028021345573 >
< 1, (0, 0), (1, 5), -4.14490285958 >
< 1, (0, 0), (0, 6), -0.578832851376 >
< 1, (0, 0), (0, 7), -1.12560480952 >
```

As we can see, the significant components are chosen in the zigzag format once the components are obtained.

Task 1(c) - Block-level DWT

The block-level DWT algorithm (implemented in "block_dwt.py") extracts the most important information of each block and uses that concentrated information to efficiently compare different frames. Through successive matrix operations on each 8x8 region, multiple resolutions of the image are created and the most important signals are aggregated, while the least important signals are truncated.

The implementation of DWT begins by reading the entire video frame data into memory, building a 3-dimensional array of Y values for every pixel in the video. This frame data matrix is the same frame data matrix used in other parts of the project. After DWT is performed, the resulting matrix has the same shape and size as this matrix, only each 8x8 block is transformed into its wavelet component coefficients.

As described in other sections, each frame is divided into 8x8 regions, known as *frame blocks* that are passed through the DWT algorithm individually. The DWT algorithm runs 3 stages of the DWT using the 2D Haar wavelet transform and returns the wavelet coefficients in an 8x8 matrix.

In order to see how this process works, let's take the following 8x8 matrix and perform the steps of the three-stage DWT Haar wavelet transform on it:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	63	127	127	63	0	0
0	0	127	255	255	127	0	0
0	0	127	255	255	127	0	0
0	0	63	127	127	63	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

This array is then run through a "Low pass filter" and "High pass filter" to produce two 8x4 arrays - one of the inter-column averages (low pass) and one of the inter-column differences (high pass). These two arrays (when put side-by-side, with L on the left and H on the right) would be:
L concat H:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	95	95	0	0	-32	32	0
0	191	191	0	0	-64	64	0
0	191	127	0	0	-64	64	0
0	95	95	0	0	-32	32	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

The code to perform this action is the following:

```
vertical_lowpass = (block[:, ::2] + block[:, 1::2])/2.0
vertical_highpass = (block[:, ::2] - block[:, 1::2])/2.0
vertical_dwt = np.hstack((vertical_lowpass, vertical_highpass))
```

The same operation is then performed again on that array, only horizontally. In this case, each row is averaged and diffed to produce all four quadrants of LL, LH, HL and HH, as seen below:

0	0	0	0	0	0	0	0
0	143	143	0	0	-48	48	0
0	143	143	0	0	-48	48	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	-48	-48	0	0	16	-16	0
0	48	48	0	0	-16	16	0
0	0	0	0	0	0	0	0

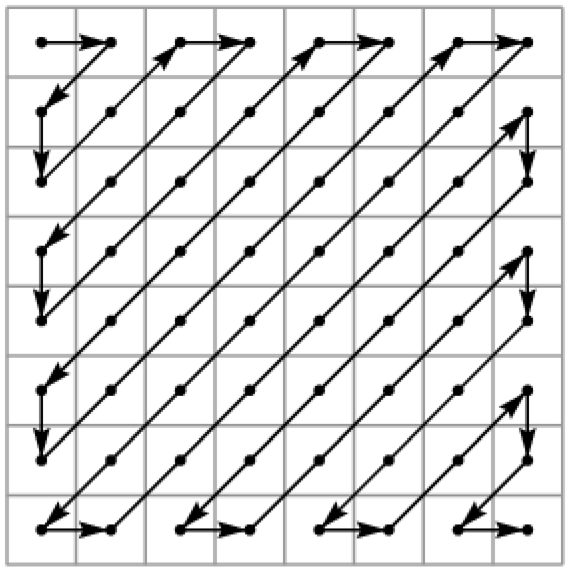
The code for performing this operation is here:

```
horizontal_lowpass = (vertical_dwt[::2] + vertical_dwt[1::2])/2.0
horizontal_highpass = (vertical_dwt[::2] - vertical_dwt[1::2])/2.0
dwt = np.vstack((horizontal_lowpass, horizontal_highpass))
```

This 2D Haar wavelet transform is then performed again on the top-left hand 4x4 corner of the array, and then again on the top 2x2 corner of that array, producing the three-iteration Haar wavelet 2D transform:

35.75	0	-35.75	35.75	0	0	0	0
0	0	-35.75	35.75	0	-48	48	0
-35.75	-35.75	35.75	-35.75	0	-48	48	0
35.75	35.75	-35.75	35.75	0	0	0	0
0	0	0	0	0	0	0	0
0	-48	-48	0	0	16	-16	0
0	48	48	0	0	-16	16	0
0	0	0	0	0	0	0	0

Depending on the value *n* supplied by the user, the top *n* significant wavelets are selected and written to the file. The significance of the wavelets is determined in the exact same way as DCT significance, which comes from the JPEG standard for compression.



Zig-zag order of significance

The zig-zag order is the true order of most significant wavelets for DWT, since the top-left-hand wavelets have the most important and significant information, while the bottom-right-hand wavelets have the least significant information. The top left hand pixel is essentially a 1x1 resolution summary of the entire block. The top left hand 4 pixels can be used to create a 2x2 resolution summary of the entire block, and so if the number of wavelets chosen increases, a higher resolution block can be recreated.

The top n of the zigzag format are chosen and written to the .bwt text file. The format of each component entry is the following:

<Frame Number, Frame Block Coordinates, Pixel Coordinate, Component Value>

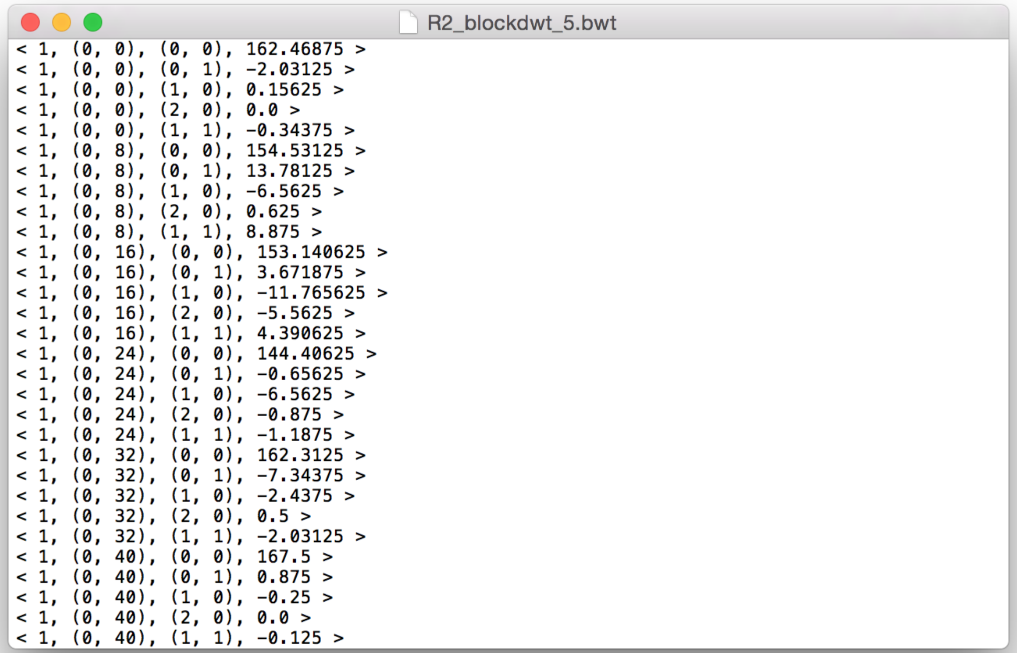
The “Frame Number” refers to the index of the frame in the video, starting with 1 and incrementing by 1 for each successive frame. The “Frame Block Coordinates” refers to the top-left-hand coordinate of the 8x8 block that the DWT algorithm was performed on. For example, the top-left block will have the Frame Block Coordinates of “(0, 0)”, the block to the right of that will be “(0, 8)”, followed by “(0, 16)”. After wrapping past the right-hand side of the frame, the next Frame Block Coordinates would be “(8, 0)”, “(8, 8)” and so on.

The “Pixel Coordinate” refers to the location of the wavelet in the block. Since the same wavelets are always chosen in our given significance test, this information is not necessary for comparison or reconstruction. We could know which value corresponded to which wavelet based solely on the order of the component value. In our case, the pixel coordinates for each frame block will always be “(0, 0)”, “(0, 1)”, “(1, 0)”, and so on in the zig-zag format.

The Component Value is the signed floating point value associated with that wavelet component. As an example, the first line in a .bwt file could be

< 1, (0, 0), (0, 0), 35.75 >

Meaning the component in the first frame, in the (0,0) frame block at the (0, 0) position has a component value of 35.75. The final output of the algorithm looks like the following bwt file:


R2_blockdwt_5.bwt

```
< 1, (0, 0), (0, 0), 162.46875 >
< 1, (0, 0), (0, 1), -2.03125 >
< 1, (0, 0), (1, 0), 0.15625 >
< 1, (0, 0), (2, 0), 0.0 >
< 1, (0, 0), (1, 1), -0.34375 >
< 1, (0, 8), (0, 0), 154.53125 >
< 1, (0, 8), (0, 1), 13.78125 >
< 1, (0, 8), (1, 0), -6.5625 >
< 1, (0, 8), (2, 0), 0.625 >
< 1, (0, 8), (1, 1), 8.875 >
< 1, (0, 16), (0, 0), 153.140625 >
< 1, (0, 16), (0, 1), 3.671875 >
< 1, (0, 16), (1, 0), -11.765625 >
< 1, (0, 16), (2, 0), -5.5625 >
< 1, (0, 16), (1, 1), 4.390625 >
< 1, (0, 24), (0, 0), 144.40625 >
< 1, (0, 24), (0, 1), -0.65625 >
< 1, (0, 24), (1, 0), -6.5625 >
< 1, (0, 24), (2, 0), -0.875 >
< 1, (0, 24), (1, 1), -1.1875 >
< 1, (0, 32), (0, 0), 162.3125 >
< 1, (0, 32), (0, 1), -7.34375 >
< 1, (0, 32), (1, 0), -2.4375 >
< 1, (0, 32), (2, 0), 0.5 >
< 1, (0, 32), (1, 1), -2.03125 >
< 1, (0, 40), (0, 0), 167.5 >
< 1, (0, 40), (0, 1), 0.875 >
< 1, (0, 40), (1, 0), -0.25 >
< 1, (0, 40), (2, 0), 0.0 >
< 1, (0, 40), (1, 1), -0.125 >
```

Task 1 (d) - Block differences quantization and histogram representation

On task 1-a, a video was read and then its frames were broken into 8x8 block regions, and after that quantized using *n* bins. On task 1-d, the goal is very similar. The main difference is that instead of quantizing the pixel values, we quantize the difference between these pixel values, on a block-level manner.

In more details, the program works in the following way:

First, you can either execute the program without any argument or with 2 command liine arguments: the value of *n* and the absolute path of the input video. Both ways are shown below:

```
$ python diff_quantize.py
```

or:

```
$ python diff_quantize.py n ../path/to/file.mp4
```

If no arguments are passed, the user will be prompted to provide the absolute path to the directory where the video is located, and then choose one of the files located on this directory. Otherwise, the program will try to open the video located on the provided absolute path(using the command line arguments), and if it is a valid path and file, its processing will start.

The program then takes each frame and puts it into an array, using the *getContent()* function, located on the auxiliary *util* module. Then, the numpy library built-in *diff* function is used to calculate the differences between the pixel values in each frame stored in the array. After that, using the same quantization methods used on task 1-a, for each frame, the video is broken into 8x8 regions and then gets quantized based on the difference values stored in these regions.

After that, the program outputs all the processed and quantized data to a file, and shows a histogram based on the whole frame video difference values.

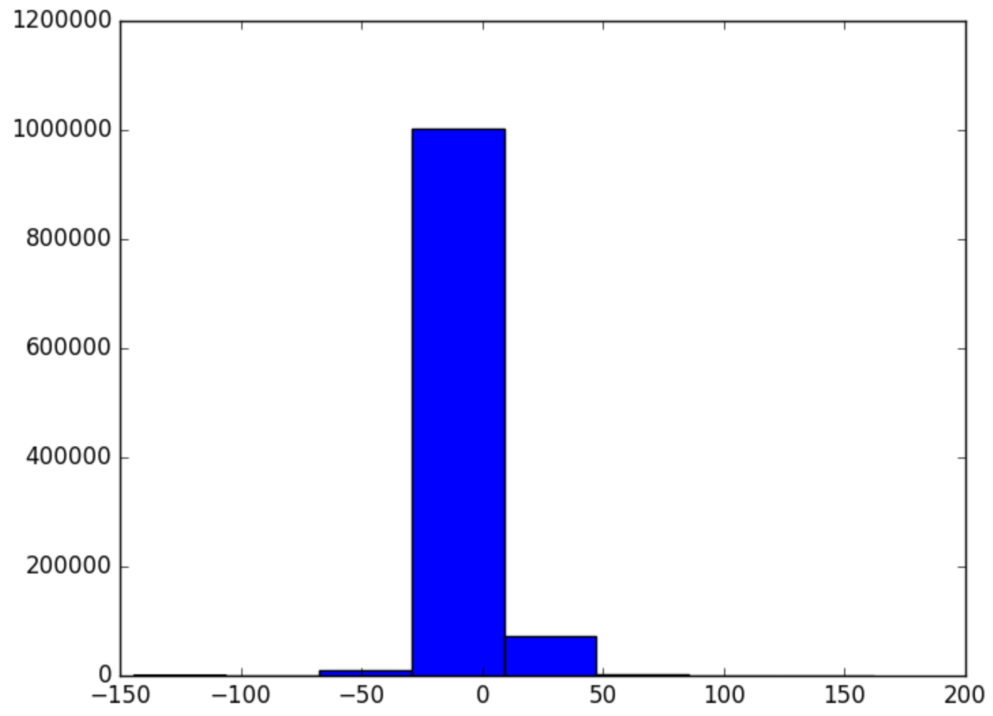
The output is formatted based on the following form:

<*frame_id*, *block_coord*, *diff_comp_id*, *pixelcount*>

Where *frame_id* identifies the particular frame, *block_coord* represents the coordinates for a 8x8 block region, *diff_comp_id* contains the quantized difference values, and *pixelcount* stores the count of how many times the value in *diff_comp_id* appeared on the frame represented by *frame_id*, on the particular block identified by the *block_coord* coordinates. Below, an image illustrates part of an output file, generated after the processing of the “R1.mp4” sample video, using 8 bins:

```
< 1, (0, 0), -5.125, 2 >
< 1, (0, 0), -3.375, 9 >
< 1, (0, 0), 3.625, 5 >
< 1, (0, 0), 7.125, 4 >
< 1, (0, 0), 0.125, 12 >
< 1, (0, 0), 5.375, 3 >
< 1, (0, 0), -1.625, 10 >
< 1, (0, 0), 1.875, 19 >
< 1, (0, 8), 0.3125, 35 >
< 1, (0, 8), 0.9375, 16 >
< 1, (0, 8), 4.6875, 1 >
< 1, (0, 8), 3.4375, 0 >
< 1, (0, 8), 4.0625, 4 >
< 1, (0, 8), 1.5625, 0 >
< 1, (0, 8), 2.1875, 5 >
< 1, (0, 8), 2.8125, 3 >
```

The histogram representation, as quantization, is implemented in the same way as in task 1-a, using its same function. Thus, it basically gets all the difference values of the video and displays a histogram showing how many times these values appear on the whole video. You can see, below, a image of a histogram generating by the processing of video “R1.mp4” using 8 bins:



After showing the histogram, the program saves it into a .png file, along with the .dhc output file, inside the path the user entered on the beginning of the execution.

Task 2

The frame-level DWT (implemented in “frame_dwt.py”) performs the same 2D Haar wavelet transform that the block-level DWT algorithm performs, only this time the entire frame is used as input rather than 8x8 blocks of the frame. Since the size of the frame is more than 8x8, there are more iterations of DWT that can be performed on the top-left-hand corner of the image, compressing whatever resolution of the frame to a 1x1 pixel resolution summary image.

The number of iterations of DWT depends on the width and height of the image. Assuming the video has an equal height and width of 2^n, for some n > 0, the number of iterations of DWT is equal to n, or $\log_2$(width). For example, if a video has resolution 64x64, the number of iterations would be 6, since $2^6 = 64$:

1st iteration: whole 64x64 frame
2nd iteration: top-left 32x32 pixels
3rd iteration: top-left 16x16 pixels
4th iteration: top-left 8x8 pixels
5th iteration: top-left 4x4 pixels
6th iteration: top-left 2x2 pixels

This produces a matrix of wavelets where the top-left pixel is the lowest resolution image, and the surrounding pixels can be used to reconstruct higher and higher resolution versions of each frame.

Choosing the m most significant wavelets is done in the same way as DCT and frame-block level DWT - in the zig-zag format from the top-left to the bottom right (See above for more details).

Once the top m components are chosen from the zig-zag ordering for each frame, they are written to the .fwt text file in the following format:

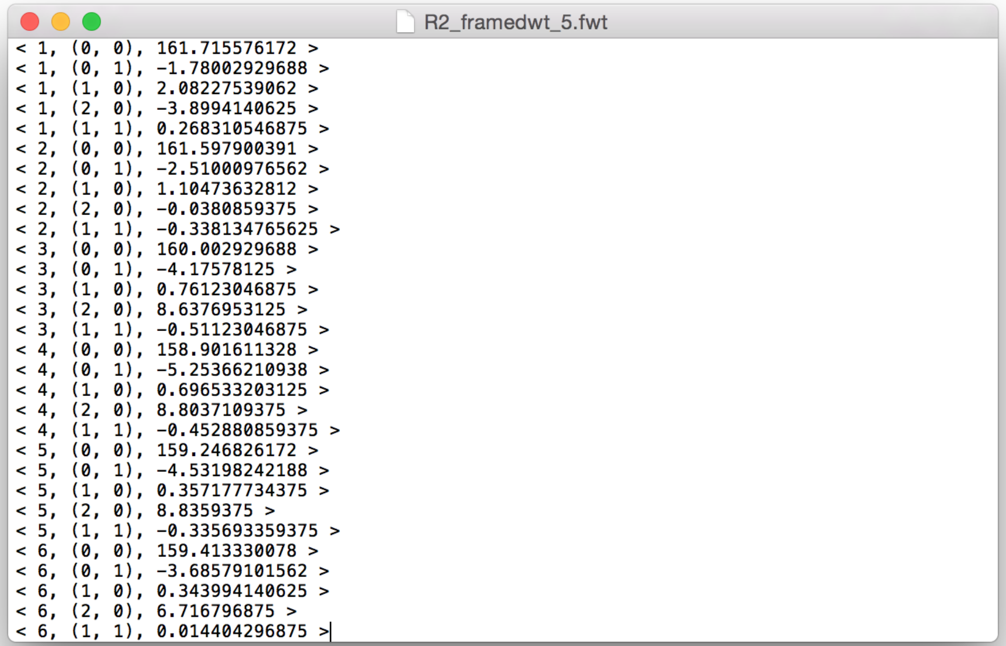
<Frame Number, Pixel Coordinate, Component Value>

“Frame Number” represents the index of the frame number, starting with 1 and going through the number of frames in the video. The “Pixel Coordinate” is the (row, column) index of the wavelet component. This information is not necessary for comparison or reconstruction, since we know the order of wavelets will always stay the same for each frame - in the zig-zag ordering. The “Pixel Coordinate” values for each frame will therefore be “(0, 0)”, “(0, 1)”, “(1, 0)” and so on in the zig-zag ordering. The “Component Value” is the signed floating-point coefficient of the wavelet component.

As an example, the first line of a .fwt file could be:

< 1, (0, 0), 161.715576172 >

Where “1” represents the first frame, “(0, 0)” represents the wavelet at row 0 and column 0, and 161.715576172 is the coefficient of that wavelet. The whole fwt file looks like the following:


R2_framedwt_5.fwt

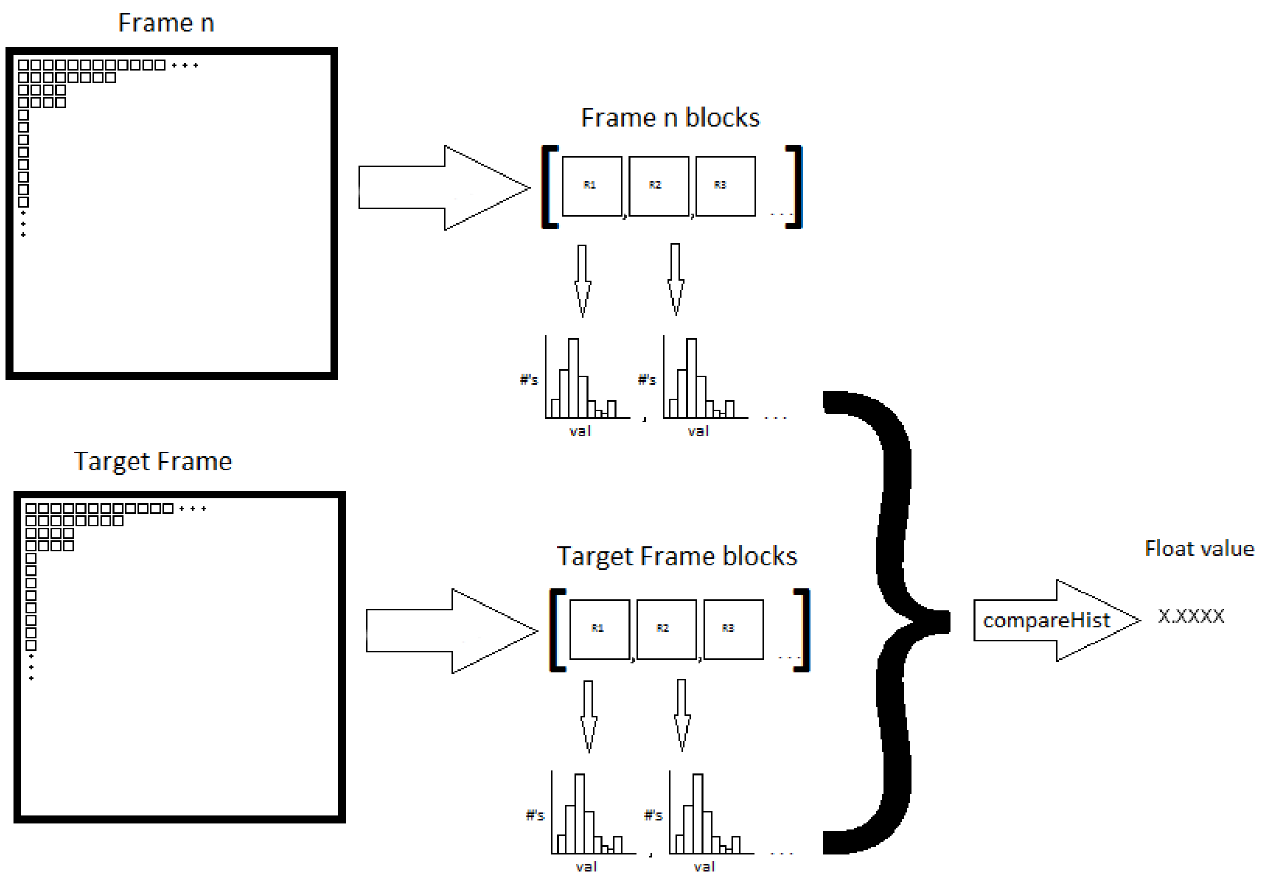
```
< 1, (0, 0), 161.715576172 >
< 1, (0, 1), -1.78002929688 >
< 1, (1, 0), 2.08227539062 >
< 1, (2, 0), -3.8994140625 >
< 1, (1, 1), 0.268310546875 >
< 2, (0, 0), 161.597900391 >
< 2, (0, 1), -2.51000976562 >
< 2, (1, 0), 1.10473632812 >
< 2, (2, 0), -0.0380859375 >
< 2, (1, 1), -0.338134765625 >
< 3, (0, 0), 160.002929688 >
< 3, (0, 1), -4.17578125 >
< 3, (1, 0), 0.76123046875 >
< 3, (2, 0), 8.6376953125 >
< 3, (1, 1), -0.51123046875 >
< 4, (0, 0), 158.901611328 >
< 4, (0, 1), -5.25366210938 >
< 4, (1, 0), 0.696533203125 >
< 4, (2, 0), 8.8037109375 >
< 4, (1, 1), -0.452880859375 >
< 5, (0, 0), 159.246826172 >
< 5, (0, 1), -4.53198242188 >
< 5, (1, 0), 0.357177734375 >
< 5, (2, 0), 8.8359375 >
< 5, (1, 1), -0.335693359375 >
< 6, (0, 0), 159.413330078 >
< 6, (0, 1), -3.68579101562 >
< 6, (1, 0), 0.343994140625 >
< 6, (2, 0), 6.716796875 >
< 6, (1, 1), 0.014404296875 >
```

Task 3

The frame matching algorithm (implemented in “match_frame.py”) finds the 10 best frames for each of the 5 different features, block histograms, block DCT, block DWT, frame-difference block histograms, and frame DWT. Each algorithm is run on the video, and the target frame’s features are compared with the features of every other frame, producing a score for each frame. The frames with 10 best scores are then displayed in grayscale along with their corresponding scores.

The comparison calculation depends on which feature is being compared. For the histogram features (task 1a and task 1d) the show_ten_quantized_closest is used to compare the shapes of the histograms, while all of the wavelet and frequency component features (task 1b, task 1c, and task 2) use show_ten_closest to calculate the difference between significant components.

show_ten_quantized_closest finds the best 10 matching quantized frames compared to the selected target frame. The provided target frame is extracted and displayed to the user, and the specific quantization algorithm is run on the entire video - either diff quantization or frame block quantization. The target frame’s histograms are extracted from the resulting histograms, and the frames of the entire video are iterated through, each comparing against the target frame’s histograms.



The histograms are simply arrays of length n (user provided) that count the number of pixels that fall in each of the n bins. This effectively provides a shape that can be compared against the shapes of other histograms. Since the histograms do not describe the values associated with each bin, and the range of the histograms are block-dependent (see Task 1a description), the shape must be the aspect compared between the blocks of different frames.

This comparison yields a float value, and depending on the comparison method selected from the OpenCV compareHist function that compares histograms. This table describes the different similarity indexes.

Name	**Arg #**	**Float Value**	**Description**
Correlation	**0**	**Larger = higher similarity**	Computes the correlation between the two histograms.
Chi-Squared	**1**	**Smaller = higher similarity**	Applies the Chi-Squared distance to the histograms.
Intersection	**2**	**Larger = higher similarity**	Calculates the intersection between two histograms.
Bhattacharyya	**3**	**Smaller = higher similarity**	Bhattacharyya distance, used to measure the "overlap" between the two histograms.

Rosebrock, 1

For our program Intersection argument number 2 in the compareHist function, is used. When each blocks histogram is compared the returned float value is summed up for the entire frame and stored in a list as a tuple with the frame number and its score. Once all frames compared values have been calculated the list is sorted and then reversed in order to have the highest comparison values first. The first 10 are then taken from this list and using the stored frame number, taken from the video data and displayed. Below you will see that not all frames are visually similar even though their scores indicate a high similarity. This is due to the fact that relatively sparse histograms and aliasing (from quantization) causes the coordinates of non-zero histogram bins so shift slightly (Histograms, 1). This is especially apparent in smaller videos, such as those provided for testing which have a a small amount of regions and a very small resolution. However, Intersection produced the best performance in terms of image similarity in our tests.

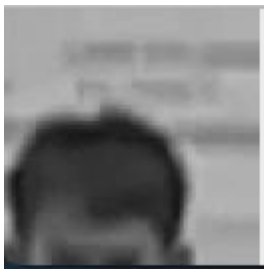
For the wavelet comparison function, show_ten_closest, a similar operation takes place. The features for each frame-block are calculated using the functions from Task 1b, 1c, and 2. The coefficients for all of the components in each block of the target frame is then extracted and stored in a target_features dictionary, where the keys are block and wavelet ids, and the values are the coefficients of the wavelets. The function then iterates through all of the frames, subtracting the coefficients of corresponding wavelets, and adding together the absolute values of those differences. The frames with the lowest sums are the closest to the target frame, and are displayed alongside their scores.

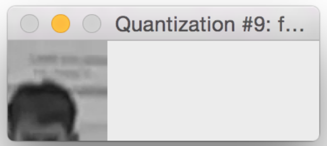
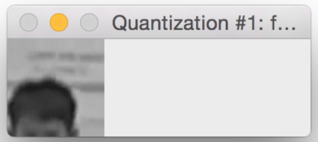
Example:

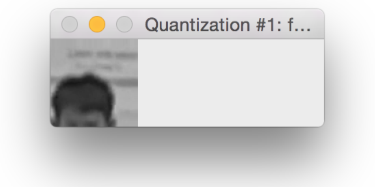
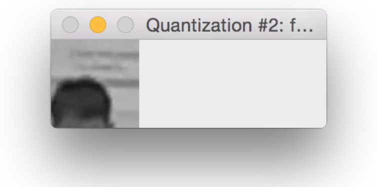
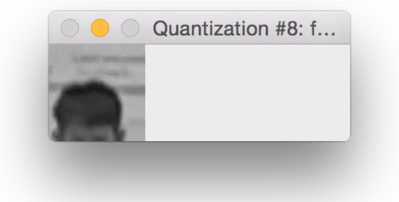
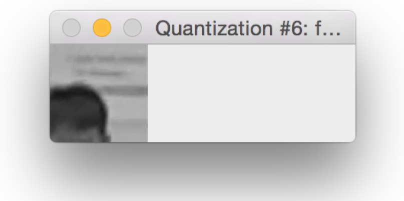
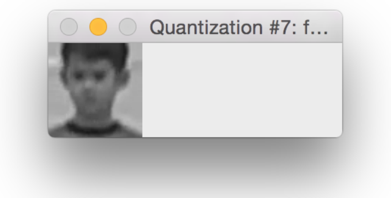
Match frame 150 of video R2.mp4 using n=5 and m=5:

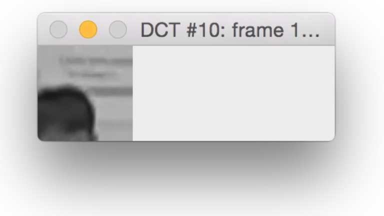
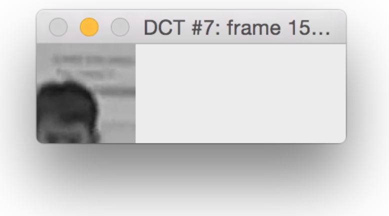
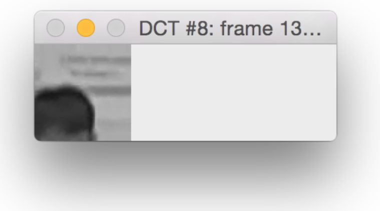
```
$ python match_frame.py 150 5 5 R2.mp4
```

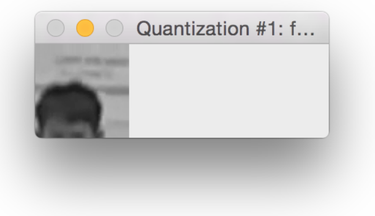
Original frame

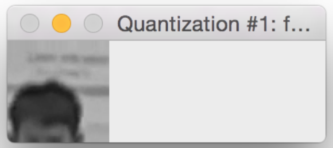
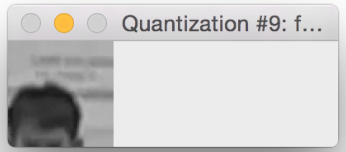
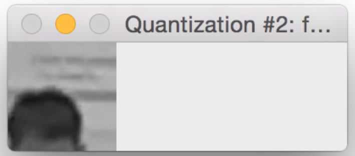
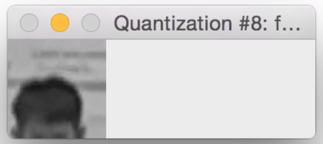


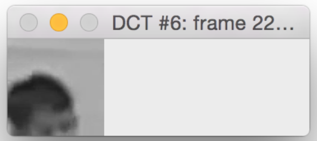
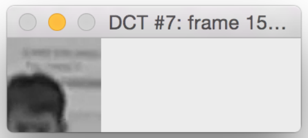
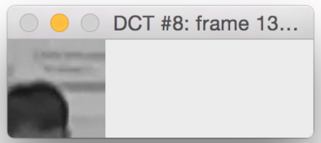
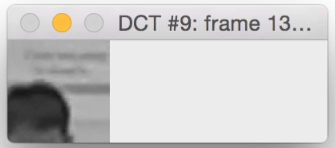
Block Quantized Histogram	
 Quantization #9: f...	frame 151. Score: 3077.0
 Quantization #1: f...	frame 149. Score: 3042.0

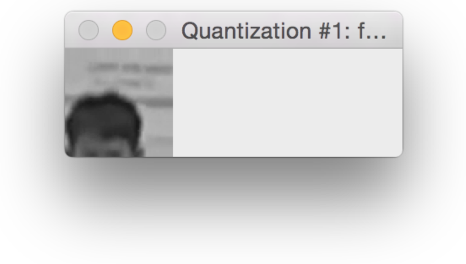
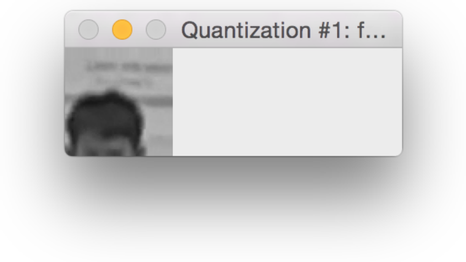
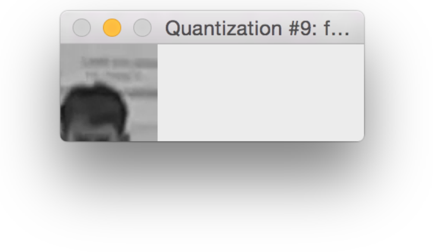
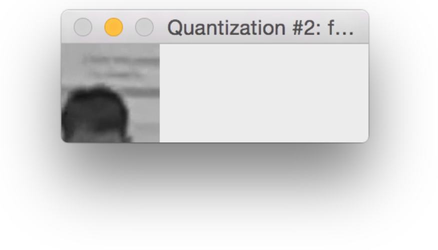
 Quantization #1: f...	frame 148. Score: 2970.0
 Quantization #2: f...	frame 140. Score: 2791.0
 Quantization #8: f...	frame 144. Score: 2657.0
 Quantization #6: f...	frame 138. Score: 2590.0
 Quantization #7: f...	frame 179. Score: 2588.0

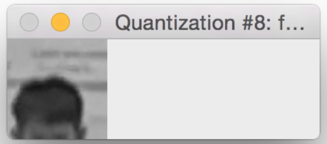
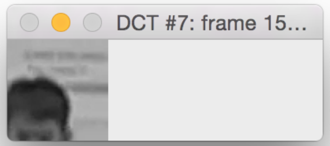
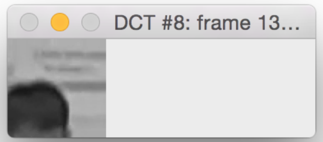
	frame 137. Score: 2577.0
	frame 152. Score: 2576.0
	frame 136. Score: 2570.0

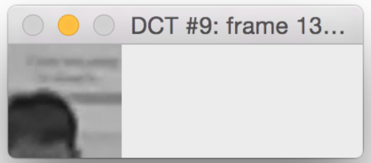
DCT	
	frame 149. Score: 4001.85489768

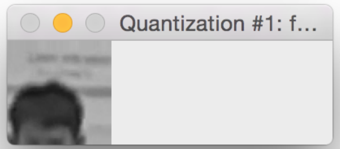
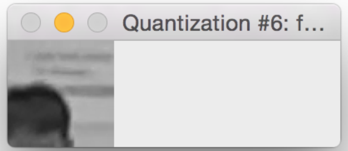
 Quantization #1: f…	frame 148. Score: 4484.96788723
 Quantization #9: f…	frame 151. Score: 6685.33798289
 Quantization #2: f…	frame 140. Score: 10774.4574018
 Quantization #8: f…	frame 144. Score: 14522.2384452

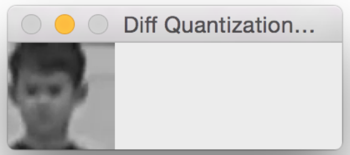
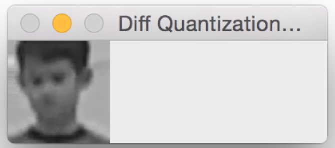
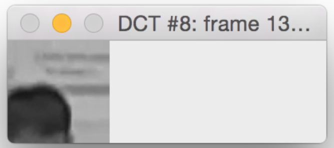
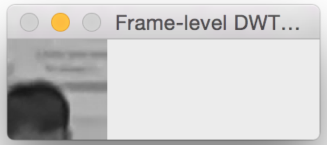
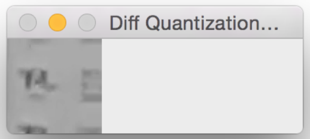
	frame 228. Score: 16159.0997339
	frame 152. Score: 16585.5425093
	frame 136. Score: 17168.9372832
	frame 139. Score: 17464.6408265
	frame 137. Score: 17467.3959966

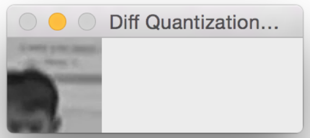
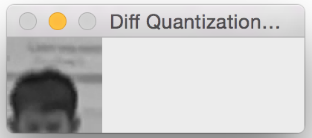
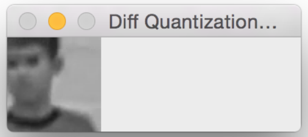
Block-level DWT	
	frame 149. Score: 528.96875
	frame 148. Score: 588.21875
	frame 151. Score: 885.59375
	frame 140. Score: 1372.40625

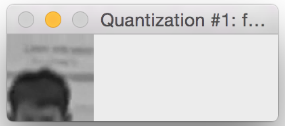
 Quantization #8: f...	frame 144. Score: 1754.53125
DCT #6: frame 22...	frame 228. Score: 1926.625
 DCT #7: frame 15...	frame 152. Score: 2047.46875
 DCT #8: frame 13...	frame 136. Score: 2101.375
DCT #10: frame 1...	frame 137. Score: 2134.84375

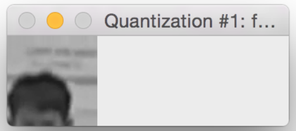
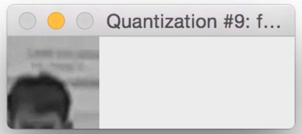
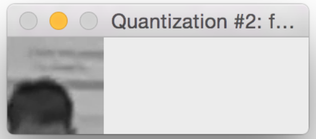
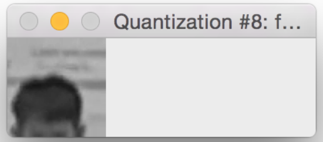
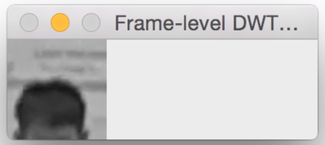
	frame 139. Score: 2139.625

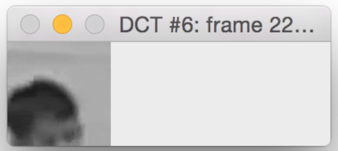
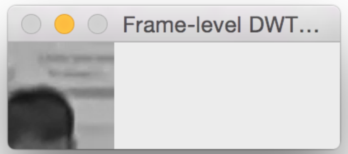
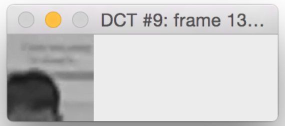
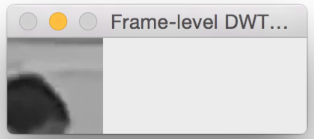
Diff histogram	
	frame 148. Score: 1360.0
	frame 138. Score: 1229.0

	frame 182. Score: 1113.0
	frame 174. Score: 1112.0
	frame 136. Score: 1110.0
	frame 135. Score: 1105.0
	frame 9. Score: 1098.0

 Diff Quantization...	frame 154. Score: 1097.0
 Diff Quantization...	frame 147. Score: 1088.0
 Diff Quantization...	frame 208. Score: 1086.0

Frame-level DWT	
 Quantization #1: f...	frame 148. Score: 1.8701171875

 Quantization #1: f...	frame 149. Score: 2.38232421875
 Quantization #9: f...	frame 151. Score: 6.36767578125
 Quantization #2: f...	frame 140. Score: 16.3354492188
 Quantization #8: f...	frame 144. Score: 17.3090820312
 Frame-level DWT...	frame 142. Score: 18.083984375

	frame 228. Score: 19.9375
	frame 135. Score: 22.259765625
	frame 139. Score: 24.42578125
	frame 81. Score: 24.9838867188

References

"Histograms." Histograms — OpenCV 2.4.12.0 Documentation. OpenCV, n.d. Web. 28 Nov. 2015.

Rosebrock, Adrian. "How-To: 3 Ways to Compare Histograms Using OpenCV and Python - PyImageSearch." PyImageSearch. PyImageSearch, 14 July 2014. Web. 28 Nov. 2015.