CPE645: Final Project

Image Processing and Computer Vision

Ben Mirtchouk

I pledge my honor that I have abided by the
Stevens Honor System.

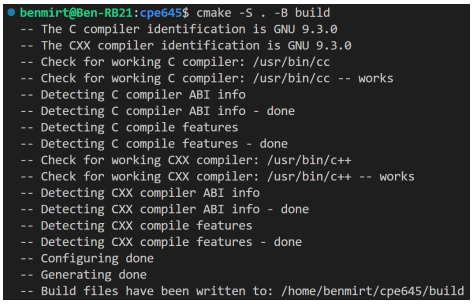
August 25, 2022

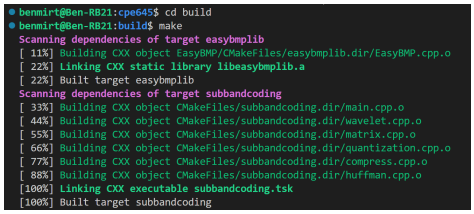
1 Abstract

The objective of this project is to developt an image compression algorithm based on wavelet coding. This compression is accomplished in three phases. First, the image is passed through a wavelet coder, which for simplicity has been limited to the Daubechies 1 and Daubechies 2 wavelets for this project. Next, scalar quantization is performed on the output of the first phase, with decision boundaries and reconstruction levels optimized via the Lloyd algorithm. Finally, Huffman entropy coding is performed to get the raw bytes of output and these are written to a file. To ensure all steps were done as closely to the pure theory as possible (without optimizations for real-world data or unfair pre-training data), all three steps were coded from scratch in C++. The only exception to this is a BMP image reading library (EasyBMP) which was imported to abstract away tedious file manipulation. These steps combined resulted in a very strong compression ratio of 15-25%.

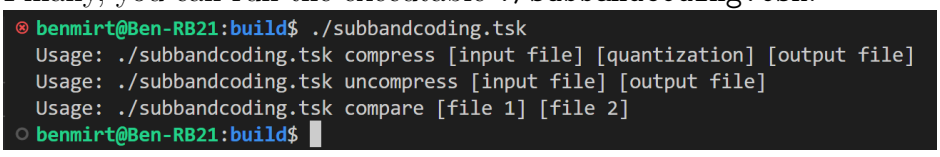
2 Compilation & Running Instructions

The code is integrated with CMake, so building should be very straightforward. Simply run `cmake -S . -B build` to create the build directory and then run `make` from within that directory.





Finally, you can run the executable `./subbandcoding.tsk`.



As shown above, the executable runs in three modes. The first will compress

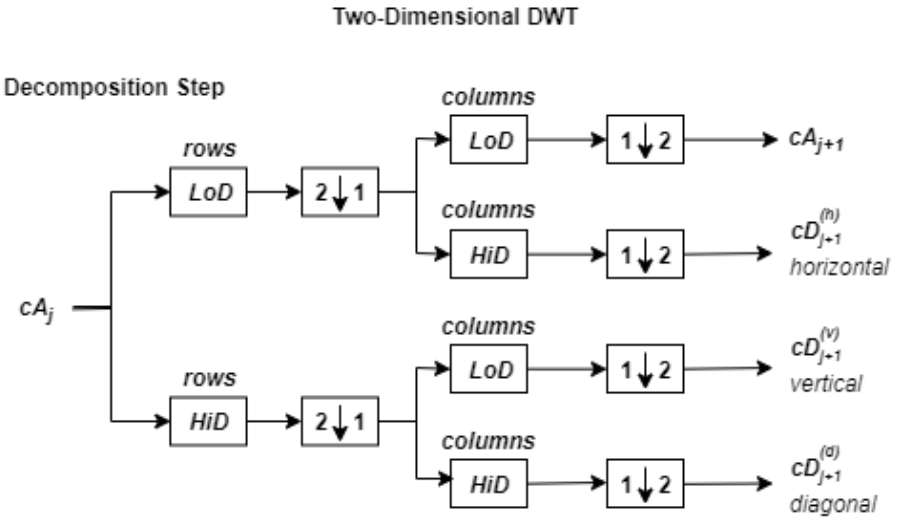
a given input file into a given output file, with a variable quantization. The second will uncompress a given file in a similar way (without the need to specify quantization level). The final operation is to compare two images and compute the Peak-Signal-to-Noise ratio via an Mean-Squared-Error statistic.

```
./subbandcoding.tsk compress ../data/Lenna.bmp 32 compressed.wft2
./subbandcoding.tsk uncompress compressed.wft2 uncompressed.bmp
./subbandcoding.tsk compare ../data/Lenna.bmp uncompressed.bmp
```

3 Implementation

3.1 Discrete Wavelet Transform

Figure 1: 2-D Discrete Wavelet Transform



The first phase of the compression is a 2-D Discrete Wavelet Transform. This is performed as per Figure 1. First, the rows of the image are passed through the low-pass and high-pass decomposition filters (LoD, HiD). Each of the outputs is then downsampled at a rate of 2:1, and combined into one image again by concatenating the resulting signals (row-by-row). See Figure 2a for an example of the output. Next, we perform the same operation, except this

time on the columns, and again downsample and concatenate the results. See Figure 2b for an example of the output. Note that the output of this step is an image with the same exact dimensions as what we input, because of the 2:1 downsampling at each step.

(a) After processing rows

(b) After processing rows and columns

Figure 2: 2-D Discrete Wavelet Transform Output

That being said, the resulting "image" (really no more than a 2-D matrix of intensities) is no longer constrained to the $[0, 255]$ intensity levels we started with due to the filters we applied. This is why the Figures 2a and 2b have had their intensities point-wise linearly scaled to the [0,255] range so that we may display them comfortably.

An important optimzation is carried out during this step. To avoid unnecessary copying of data, all operations are performed in place when possible. This is done by adding a layer of abstraction above our matrix of data called a **Span**. The Span internally stores a pointer to the full matrix, but supports indexing operations in a single direction (row or column) along with an offset and upscale factor. This state information is persisted in the Span and used to determine the "real" position of a requested element when indexing into the Span. As seen from the implementation, this operation remains $O(1)$ and means we can convolve with elements of our matrix efficiently despite the data we need often not being contiguous in the matrix.

```
const double Span::operator()(int i) const {
  if (i % d_upsample != 0) return 0;
  i = i / d_upsample + d_offset;

  if (d_isRow) return (*d_matrix)(d_index, i);
  return (*d_matrix)(i, d_index);
}
```

3.2 Scalar Quantization

The next step of the compression is a scalar quantization. We desire a quantization output $\{y_i^*\}_{i=0}^{Q-1}$ and $\{b_i^*\}_{i=0}^{Q}$, where y_i^* are the reconstruction levels and b_i^* are the decision boundaries. Q is the quantization parameter which specifies the number of output levels we would like. A smaller Q will lead to better compression, but worse a Peak-Signal-to-Noise ratio. This is to say, we will transform any intensity $b_i^* \leq I < b_{i+1}^*$ to the intensity y_i^*. We optimize our reconstruction levels and decision boundaries according to the Lloyd algorithm.

Let X be the multiset of intensities in our input. Let $f_I(I)$ be the probability density function of these intensities. Since we are working on a discrete input, $f_I(I) = |\{x \in X | x = I\}|$.

First, we initialize:

$$y_i^{(0)} = \frac{(i+1)(M-N)}{Q+1} + N$$

where $M = \max\{I\}$ and $N = \min\{I\}$ for the intensities I in our input. Then we continuously update

$$b_i^{(k)} = \frac{y_{i-1}^k + y_i^k}{2} \tag{1}$$

$$D^{(k)} = \sum_{i=0}^{Q-1} \left(\sum_{x=b_i^{(k)}}^{b_{i+1}^{(k)}-1} \left(x - y_i^{(k)} \right)^2 f_I(x) \right) \tag{2}$$

$$y_i^{(k+1)} = \frac{\sum_{x=b_i^{(k)}}^{b_{i+1}^{(k)}-1} x f_I(x)}{\sum_{x=b_i^{(k)}}^{b_{i+1}^{(k)}-1} f_I(x)} \tag{3}$$

Upon evaluating equation (2), we check if $|D^{(k)} - D^{(k-1)}| < \epsilon$ and terminate if so. This signals that our distortion has converged and we will not see any further improvement from future iterations.

3.3 Entropy Encoding

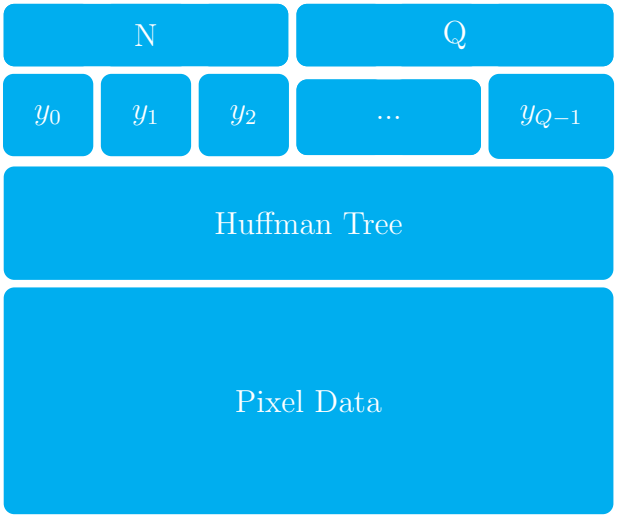
The final step in compression is entropy encoding via Huffman coding. By cleverly defining the Huffman tree's Node object, we can leverage the built-in priority queue in C++ to do the bulk of the algorithm in just a few deceptively simple lines of code:

```
priority_queue<Node*, vector<Node*>, NodeCompare> pq;
for (const auto& p : freq) {
  pq.push(new Node(p.value, p.frequency));
}

while (pq.size() != 1) {
  Node* a = pq.pop();
  Node* b = pq.pop();
  pq.push(new Node(a, b));
}

Node* root = pq.pop();
```

The main burden of this entropy encoding is then actually in coming up with a compact representation of this book-keeping data structure. This process is detailed in the next section.

3.4 Custom File Format

It is important to note that while the final output of the previously outlined three steps is simply a stream of bits, we must actually save much more additional information in our final output file. This is because we cannot assume that the quantization levels stay constant between different images (in fact they certainly do not). Similarly, our Huffman encoding is also dependent on the input image, so we must similarly save this mapping. The $wft2$ file format that I have used for this assignment (which I have made up and does not really exist), consists of the following blocks:



} 8 bytes

} $4Q$ bytes

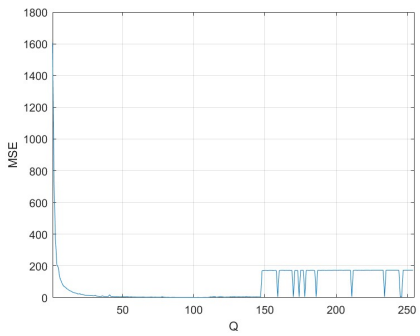
} (nodes $+ 8 *$ leaves) bits

The first section contains the integers N (the width & height of the square image) and Q (the quantization parameter). This is followed by the Q quantization levels y_i. Note that we do not need to save the decision boundaries b_i because we encode the raw indicies into y when we perform quantization. Next, we encode the Huffman tree itself. This is done via a pre-order traversal in which an internal node is encoded as the single bit 0, and a leaf node is encoded as the 9-bit sequence consisting of the bit 1 followed by the 8-bit quantization index. Finally, we record the encoded pixel values. Since Huffman coding outputs a variable length code per token, it is unclear how long this section ends up. That being said, we can estimate the length by noting that the average length of these tokens should be less than $\lceil \log_2(Q) \rceil$ since that is what we could get with naive encoding. So we have that this section's total size is less than $n^2 \lceil \log_2(Q) \rceil$ bits long.

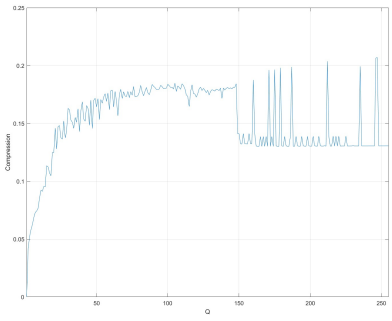
4 Results

Figures 3a and 3b show the results obtained for each given quantization parameter. As we would expect, the Mean Squared Error goes down when we use more quantization levels, since we can more accurately represent the original data. This also causes the size to increase as we have more levels to encode.

One anomaly is that the code does not function well for $Q > 148$. This is a quirk of running Lloyd's algorithm with integers (rather than floating-point numbers) as at a certain point, rounding error causes many levels to collapse into oneanother. However, the speedup we get from using integers is well worth this downside since at $Q = 148$, the image quality is very good and thus there is no reason to go higher. We can see that at this highest quantization level, we get an MSE of around 4, meaning each pixel deviates by around 2 contrast levels out of 256. As we know from Weber's Law of Just Noticeable Differences, a $\frac{\Delta I}{I}$ of around 0.02 is the limit of human perception, so this result is comfortably in the range of "good enough to not be noticeable." At even this highest quality, we get a compression ratio of 0.184 meaning that we have compressed the image to less than one-fifth of its original size!



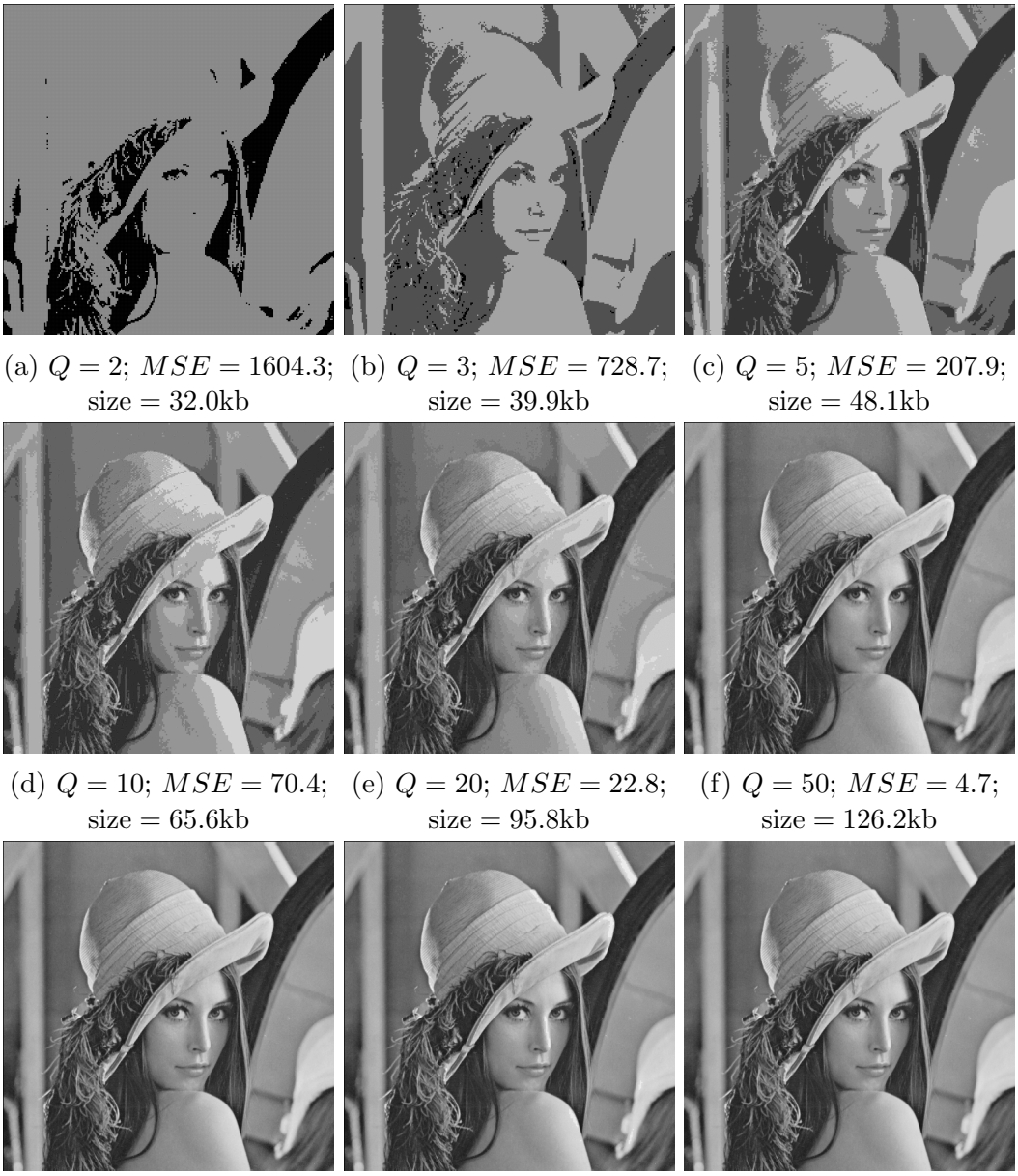
(a) MSE by Quantization



(b) Compression by Quantization

Figure 3: Result Graphs

Figure 4 shows the output with various quantization parameters.



(a) $Q = 2$; $MSE = 1604.3$; size = 32.0kb (b) $Q = 3$; $MSE = 728.7$; size = 39.9kb (c) $Q = 5$; $MSE = 207.9$; size = 48.1kb

(d) $Q = 10$; $MSE = 70.4$; size = 65.6kb (e) $Q = 20$; $MSE = 22.8$; size = 95.8kb (f) $Q = 50$; $MSE = 4.7$; size = 126.2kb

(g) $Q = 100$; $MSE = 2.0$; size = 141.2kb (h) $Q = 148$; $MSE = 4.4$; size = 141.5kb (i) Original image; size = 768.1kb

Figure 4: Result of compression followed by uncompression