International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2015): 6.391

Image Compression Techniques Based on Transform Coding

Elhadi Amir Elhadi[1], Dr. Abu Obeida Mohamed Alhassan[2]

Department Of Communication Engineering, Al-Neelain University

Abstract: ***In this paper we attempted to implement basic JPEG compression using only basic Matlab functions. This included going from a basic grayscale bitmap image all the way to a fully encoded file readable by standard image readers. we will show that are have implemented the majority of the paper, including much of the final binary coding. Although we never obtained a fully completed image from our functions, I came very close.***

Keywords:

1. Introduction

Image compression is used to minimize the amount of memory needed to represent an image. Images often require a large number of bits to represent them, and if the image needs to be transmitted or stored, it is impractical to do so without somehow reducing the number of bits. The problem of transmitting or storing an image affects all of us daily. TV and fax machines are both examples of image transmissions, and digital video players and web pictures of Catherine Zeta-Jones are examples of image storage.

Three techniques of image compression that we have discuses later are pixel coding, predictive coding, and transform coding. The idea behind pixel coding is to encode each pixel independently. The pixel values that occur more frequently are assigned shorter code words (fewer bits), and those pixel values that are more rare are assigned longer code words. This makes the average code word length decrease.

Predictive coding is based upon the principle that images are most likely smooth, so if pixel b is physically close to pixel a, the value of pixel b will be similar to the value of pixel a. When compressing an image using predictive coding, quantized past values are used to predict future values, and only the new info (or more specifically, the error between the value of pixels a and b) is coded.

The image compression technique most often used is transform coding. A typical image's energy often varies significantly throughout the image, which makes compressing it in the spatial domain difficult; however, images tend to have a compact representation in the frequency domain packed around the low frequencies, which makes compression in the frequency domain more efficient and effective. Transform coding is an image compression technique that first switches to the frequency domain, then does its compressing. The transform coefficients should be decor related, to reduce redundancy and to have a maximum amount of information stored in the smallest space. These coefficients are then coded as accurately as possible to not lose information. In this project, we will use transform coding.

2. Methodology

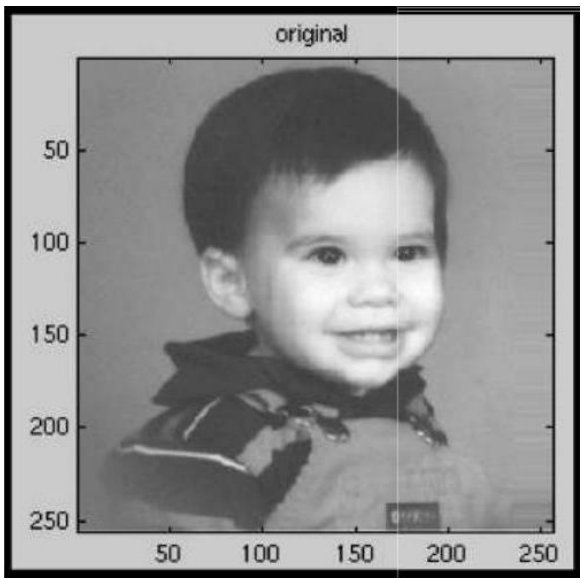

Figure 1: Original image of boy.256

In order to implement the image compression algorithm we c hose, we divided the process into various step s:

- calculate the sums and differences of every row of the image
- calculate the sums and differences of every column of the resulting matrix
- repeat this process until we get down to squares of 16x16
- quantize the final matrix using different bit allocation schemes
- write the quantized matrix out to a binary file

The first two steps are accomplished using simple loops in Matlab. Specifically, we wrote two different functions - row thing for calculating sums and differences of individual rows and clothing for calculating sums and differences of individual columns.

In order to keep the energy of the image the same, we multiplied each sum and difference by a factor of 1/sqrt(2). Performing these two operations once will result in an image that is split into four parts, with the up per left hand quadrant being the sums of sums region (see Figure 2).

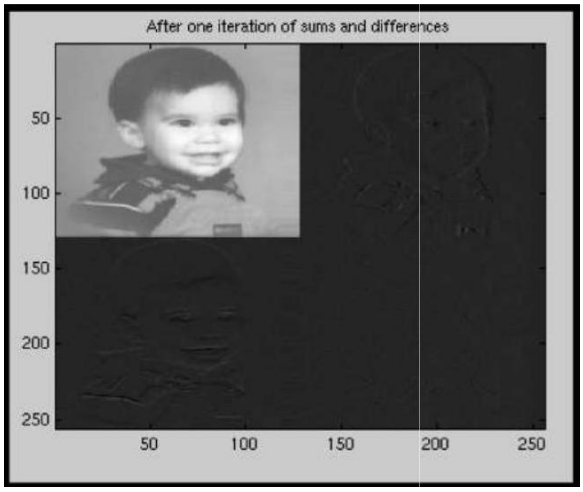

Figure 2: Sums and differences after one iteration

The third step involves repeating the previous two steps until we get down to a small enough final images. The function squishier accomplishes this task. This function performs the sums and differences of rows and columns, in alternating order, until the final sums of sums image is of size 16x16. The resulting image can be seen in the following figure (due to the normalized display, the various quadrants are not too visible, but contain various edge information):

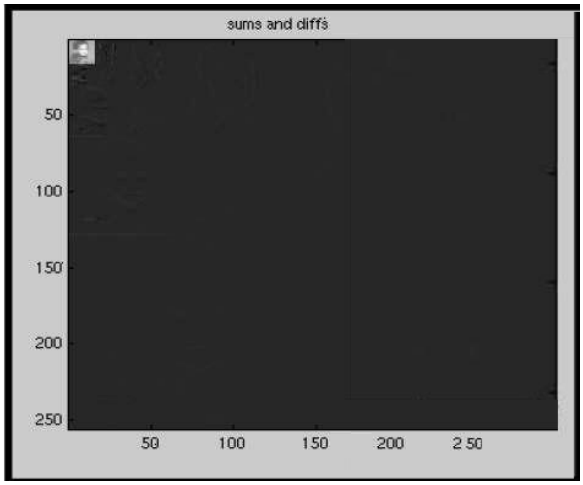

Figure 3: Final sums and differences matrix

The next step is quantization. This is performed during the writing to a binary file (see proceeding discussion of writing to a file); however, we wrote a distinct quantization function to analyze this step statistically (MSE and PS NR -- see the Results section), as well as for our own educational benefit (!). The quantization function, quant, is also called in squishier. Our quantization scheme simply assigns different numbers of bits to different regions, using masks (for example, [b16, b32, b64, b128, b256], where b16 is the quantization level for the upper left 16x16 matrix, b32 is for the next 32x32 matrix surrounding the first one, etc.). We used a number of different bit allocation masks (see next section) in order to determine which scheme is better.

The quantization function takes in as arguments not only the input matrix, but also the mask, i.e. the number of bits to be used to represent each region in the compression scheme. This allows us to modify and test out difference bit allocation algorithms easily. The quantized image looks similar to Figure 3, although depending on the mask used, the level of details may vary.

Once we have generated the compressed matrix, we are ready to transfer it to a binary file for storage (and, in the process, quantize it). We use the write Matlab command, specifying the number of bits with which to quantize. The function bit performs this operation.

Once the file is saved, we can use the file-size information to evaluate the success of the compression technique. Additionally, a further use of the lossless gzip command in Unix can show how thoroughly the image could really be stored. The next section also shows some of these results.

With compression complete, we are ready to decompress and examine the errors introduced by compression. To reverse our compression scheme, we take the following steps:

- reverse-quantize back to original levels
- undo the sums and differences calculation for each column of the matrixundo the sums and differences calculation for each row of the resulting matrix
- repeat this process on successively larger square matrices until we get back to a 256x256 matrix

Quantizing the matrix back to its original levels is easy: just quantize the compressed image back to eight bits, or bit shift 8-(first quantization level). The function iquant performs this operation. The inverse sums and differences calculations are made by adding and subtracting various pairs of numbers in the matrix and dividing by two. These values are renormalized, and placed in their appropriate positions in the target matrix. The two functions row thing and clothing accomplish e this task. These operations are performed first on the columns, and then on the rows of succeedingly larger sections of the matrix until the entire 256x256 matrix have been decompressed. This process is done with the function unsquisher. Once this is done, we have reconstructed our image. Figure 4 shows the reconstructed image using the mask [8 6 4 2 0].

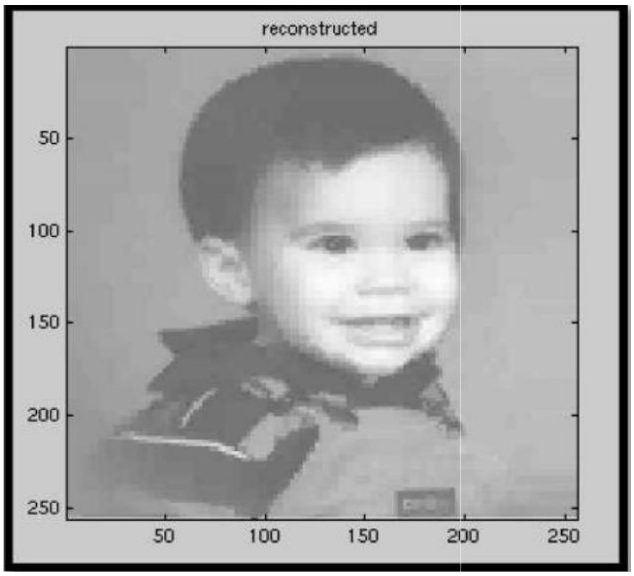

Figure 4: Reconstructed image using mask [8 6 4 2 0]

3. Result

We tested our wavelet transform coder on a few different images to see if some images would compress better (i.e. with less error) than others. In addition to the boy.256 image, which is relatively smooth and does not have too many edges, we also used the following three images:

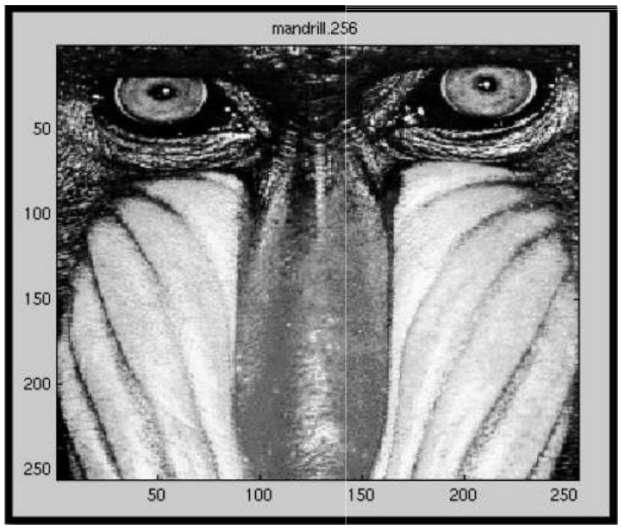

Figure 5: Image with sharp edges and high contrast

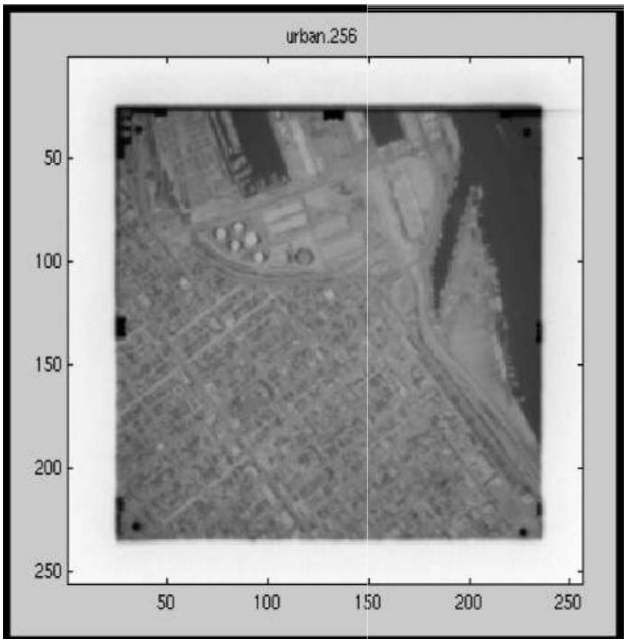

Figure 6: Blurry satellite image

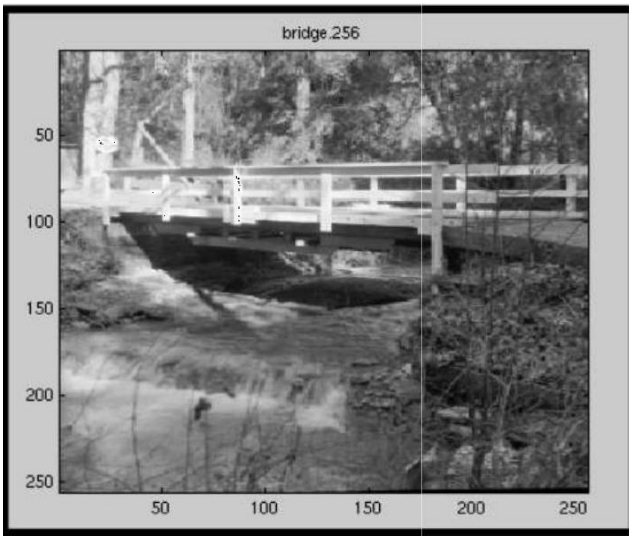

Figure 7: Image with low contrast

We ran our coder with vario us different masks (thus resulting in different numbers of bits per pixel) on each of these images. For each run, we determined the mean-square error (MSE) as well as the peak signal-to-noise ratio (PSNR). The following two graphs show the results of these tests:

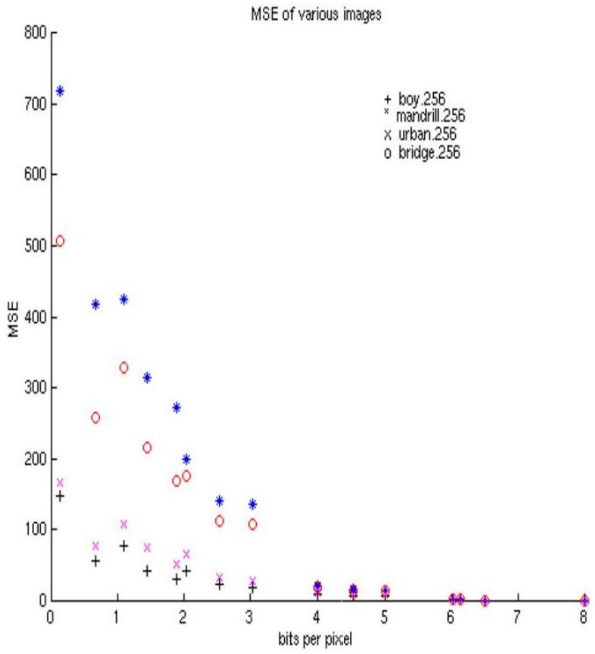

Figure 8: Mean-square error plot

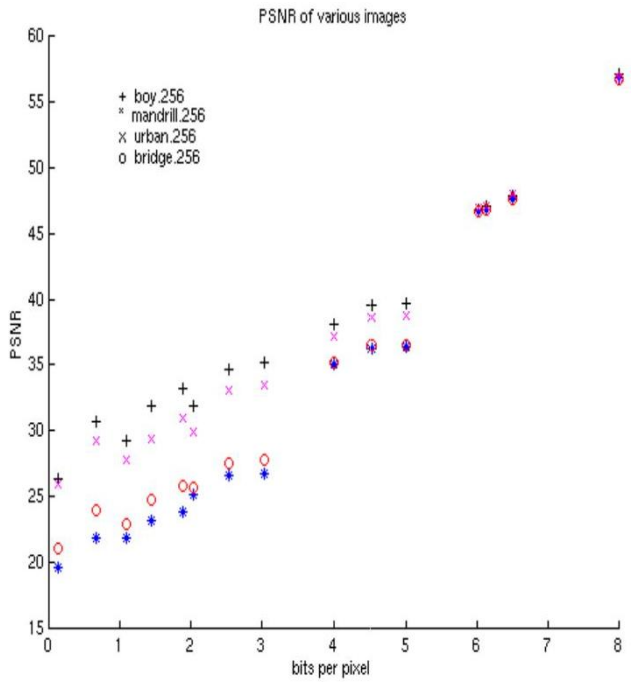

Figure 9: Peak signal-to- noise ratio plot

As we can see, the boy.256 image seem to perform the best out of the four while the madrill.256 image seem to perform the worst. We show a nice linear increase in PSNR with additional bits-per-pixel. Note we get a PSNR of greater than 20 for all images even with less than 1 bit per pixel.

In addition to the MES and PSNR results, we also analyzed the results of the compression (since that is what the assignment is called). For this analysis, we used the mask [8 6 4 2 0] (the same mask used in the representative images above) in the coding scheme because it gave us really good approximations of the original image. For comparison purposes, it is useful to note that the JPEG compression standard has a compression ratio of 40:1.

4. Conclusion

The JPEG algorithm was created to compress photographic images, and it does this very well, with high compression ratios. It also allows a user to choose between high quality output images, or very small output images. The algorithm compresses images in 4 distinct phases, and does so in time, or better. It also inspired many other algorithms that compress images and video, and do so in a fashion very similar to JPEG. Most of the variants of JPEG take the basic concepts of the JPEG algorithm and apply them to more specific problems.

Due to the immense number of JPEG images that exist, this algorithm will probably

(Note: All numbers in units of bytes.) be in use for at least 10 more years. This is despite the fact that better algorithms for compressing images exist, and even better ones than those will be ready in the near future.

Image	Original File	After Our Coder	Compression Ratio	Gzipped	Final Ratio
boy.256	65536	5440	12.1:1	822	79.7:1
mandrill.256	65536	5440	12.1:1	670	79.8:1
bridge.256	65536	5440	12.1:1	972	67.4:1
urban.256	65536	5441	12.0:1	855	76.7:1

References

[1] M. J. Weinberger, G. Seroussi, and G. Sapiro. The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. IEEE Transactions on Image Processing, 2000, 9(8):1309-1324.

[2] Chu, W.C., On lossless and lossy compression of step size matrices in JPEG coding. International Conference on Computing, Networking and Communications, 2013, 103-107.

[3] H. Oh, A. Bilgin, M. Marcellin. Visually Lossless Encoding for JPEG2000. IEEE Transactions on Image Processing, 2013, 22(1):189-201.

[4] K. Srinivasan, J. Dauwels, M. Reddy. Multichannel EEG Compression: Wavelet-Based Image andVolumetric Coding Approach. IEEE Journal of Biomedical and Health Informatics, 2013, 17(1):113-120.

[5] K. Rajakumar, T. Arivoli. Implementation of Multiwavelet Transform coding for lossless image compression. International Conference on Information Comm. and Embedded Systems, 2013, 634-637.

[6] Chiyuan Zhang, Xiaofei He. Image Compression by Learning to Minimize the Total Error. IEEE Transactions on Circuits and Systems for Video Technology, 2013, 23(4): 565-576.

[7] K. Uma, P. Palanisamy, P. Poornachandran. Comparison of image compression using GA, ACO and PSO techniques. International Conference on Recent Trends in Information Technology, 2011, 815-820.

[8] Tzong-Jer Chen, Keh-Shih Chuang. A pseudo lossless image compression method. The 3rd International Congress on Image and Signal Processing, 2010, 2:610-615.