DCC 95 Impressions

The most common and recurring theme: <u>multiresolutional/hierarchical</u> analysis/compression of text/images/video. That is, multistage predictor-corrector steals the show. It's time for Grand Unification.

1. Zerotree coding

Zero-tree coding has become very popular. Shapiro and his paper were mentioned in almost every talk about wavelet compression:

J.M.Shapiro, *Embedded image coding using zerotrees of wavelet coefficients*, IEEE Trans. on Signal Processing, 41(12), 3445-3462.

Zero-tree is nothing but a "century-old" quadtree segmentation (still very widely used in computer graphics), only applied to three-trees of wavelet coefficients, rather than quadtrees of image multiscale representations.

The idea: many wavelet coefficients especially after a coarse thresholding are insignificant (zeroed off). Because the wavelet decomposition of an image is so "sparse", it makes sense to represent the entire set of coefficients as a significance bitmap (indicating the location of significant coefficients) plus a sequence of significant values themselves. Zerotree is an efficient way to compactly transmit the significance map to the decoder: if a node is marked as zerotree root, it means that all its children are insignificant, therefore, the corresponding portion of the significance map needn't be transmitted at all.

In the original Shapiro's method the significance map was coded in a so-called dominant pass by telling the decoder the status of the current tree node as being significant positive, significant negative, insignificant with significant children, or a zerotree root. The other bits of significant coefficients were transmitted during a subordinate pass. The three-tree of wavelet coefficients is traversed breadth-first. Progressive thresholding of the three-tree with dyadic thresholds (which are powers of two) is a neat way of sending bits of the wavelet coefficients. A more general approach was mentioned in a talk *Tree structured Vector quantization with significance map for wavelet image coding*, p.33 of the Proceedings: rather than quantize/threshold coefficients, send as many bits of as many coefficients as necessary to achieve a desired bit rate at a given distortion. Note that a loss of one bit in different wavelet coefficients may have different effect on the reconstructed image, therefore, one needs to single out the most "important" /sensitive coefficients/bands and encode them with more bits.

Another idea (which I came up with two years ago but haven't played with): encode the significance maps as a sequence of moves a finite automaton does when it traverses the tree depth-first. Only moves that can't be inferred (like striding over a zero-tree root) should be encoded.

2. Advances in lossless coding and text compression

PPM arithmetic coding with multi-level contexts (and escapes among them) is alive and doing well, *Unbounded Length Contexts for PPM*, p. 52. Most of the improvements are in implementation, like using an efficient data structure to store contexts and count their occurrences: PATRICIA-type tries. The new PPM* method achieves 2.34 bits/char over the Calgary corpus, the best result ever.

New arithmetic coding: CACM++ (The Next Generation of the Comm.ACM 1987 code), available from `ftp://munnari.oz.au/pub/arith_coder/arith_coder.tar.gz`. Major improvements: a better source model based on words along with characters (and using esc-like switching from a word- to a char-based context when a new word shows up; there is also separate contexts for punctuation and streams of non-letters). The coding algorithm is rearranged a little bit to cut one multiplication operation per symbol on encoding and two multiplies per symbol on decoding (at the expense of a very slight decrease in coding efficiency). Another advantage of the rearrangement is significant relaxing of constraints on the code size and the range of the total cumulative counter. The introduced inefficiency can be reduced to be almost negligible (to be below 1%) if one normalizes the code appropriately (when the total cumulative count is scaled not to exceed 2^26 with low/high counters being 32-bit unsigned integers) and one arranges symbols in the way that the last symbol is the most frequently occurring one. The new algorithm also includes a better data structure for updating frequency counts and keeping subtotals, which doesn't require "move-to-the-front", takes only log(n) time, and performs better (on uniform distributions). Move to the front, with O(n) complexity of updating frequency counts, works better on very skewed distributions (coding of a laplacian pyramid may be one of them).

Huffman coding is still alive and used in situations where the coding alphabet is extremely large (e.g., a word-based coding of text) and the source statistics is known in advance (like in compressing bodies of legal documents). Huffman coding offers advantages of being extremely fast (the fastest compression scheme around) and allowing a partial decompression (that is, one doesn't need to decompress the entire corpus to read only few pages of it). To improve the coding efficiency on very large alphabets, it's proposed to group rarely occurred symbols (words) into blocks: it looks like a hierarchical Huffman coding.

Multiple-Dictionary Compression Using Partial Matching, p. 272: a very good source of references for LZxx-type compression, giving many implementation details of LZxx coders.

Interesting paper: The *Effect of Non-Greedy Parsing in Ziv-Lempel Compression Methods*, p. 302. Instead of looking for the longest possible match of a current stream with a dictionary item, they look for a shorter match but such one that could make the next match much longer. This approach is used in gzip compression and proved very useful. The paper has many algorithmic details (and even a pseudo-code).

Hierarchical dictionary compression method (I guess was invented by some guy from Princeton: overheard from a private conversation) uses a set of dictionaries. Say, one dictionary contains only 8 strings (so it takes only 3 bits to code their indices plus one bit for the dictionary selector), another dictionary has, say, at most 32 entries (so it takes 5 bits plus, say, two bits of the selector to code indices), etc. The encoder tries to find the longest possible match from the current point forward with some string in the dictionaries (starting from the 1st level dictionary). When a match is found, the index of the corresponding dictionary string is transmitted. At the same time, the frequency counter of the string is incremented and, if it’s high enough, the string is moved to a higher-level dictionary (pushing a less frequently occurred string to a lower-level dictionary). In a sense, it looks like the “move-to-the-front” strategy of the CACM Arithmetic coder, or like a set of multi-level “memory caches”. The method has a clear advantage over the conventional LZxx in that it accounts not only for substring occurrence in the past text but also for their frequency, and assigns shortest codes to the most frequently occurred strings (rather than most recently occurred ones). The guy said that the method performed very well.

3. Fractal image compression

The main stress is on making it faster. One way (pretty straightforward) is to perform a classification of range/domain blocks and do search only among blocks of a similar (the same) class, and use a fast nearest-neighbor search (a recently developed technique of fast approximate matching in R^d). In *Accelerating Fractal Image Compression by Multi-dimensional Nearest Neighbor Search*, p. 222. How to classify blocks (4x4 blocks in the paper): perform the DCT on them and use the few largest AC coefficients as a “shape code” or block “signature” (also called “feature vectors”). Matching signatures using few DCT coefficients is faster than matching 16 pixels (especially in cases when one need to decide which of several imperfect matches is “better”). The DC components, that is, an average brightness, don’t count in matching as they always can be "equalized" by an affine transform. To accelerate the search, domain blocks are placed into a specially designed tree with their "shape code" (feature vector) as a key. Domain/range blocks are generated by Yuval Fisher's adaptive quadtree partitioning algorithm. Note a paper, Saupe D., Hamzaoui R., *A review of the fractal image compression literature*, ACM Computer Graphics, 28, 4 (Nov 1994) 268-276.

Another approach (*Self-Quantized Wavelet Subtrees: A Wavelet-Based Theory for Fractal Image Compression*, p. 232) is a multiresolutional matching of domain/range block. The paper considers the fractal transform as a Residual (that is, multiresolutional) VQ of an image with the codebook being an image itself at a reduced resolution. A lower band of the wavelet transform (among other possible choices) can serve as this "reduced-resolution" image view. In this formulation, the corrector-predictor nature of fractal compression is apparent, as its relation to wavelet transforms. The ideas in the talk (especially the idea of a "smart" expansion, that is, predicting a finer-resolution layer of the decomposition based on similarities among the previous levels) are very close to what I talked about last year. Geoffrey Davis, the author, reports that his implementation of the "multiresolutional" fractal transform performs no better than a good wavelet decomposition. He admitted that he got his program running and obtained the first compression results merely a week or so before the conference, so this isn't by any means a final conclusion. The final version of his paper is

```
http://www.cs.dartmouth.edu/~gdavis
```

It turns out that he was the guy who sent me e-mail asking for my DCC'94 paper. So he must've read my paper and my thesis.

The best image compression ever is given by a Finite State Image compression by Karel Culik. It offers 100:1 and even 300:1 compression with almost no noticeable image degradation. The method is explained in papers:

K.Culik II and J.Kari, *Image Compression Using Weighted Finite Automata*, Computer and Graphics, 17, 3, 305-313 (1993)

K.Culic II and J.Kari, *Image-Data Compression Using Edge-Optimizing Algorithm for WFA Inference*, Journal of Information Processing and Management, 30, 829-838 (1994).

The algorithm belongs to the family of fractal image compressors. Unlike Iterated Function Systems method, the search for self-similarity is constrained within relatives of the same quadtree branch. That's why the algorithm is much faster than a typical IFS compression. In a sense, the method is a variation of the quadtree image segmentation, only segmentation criterion is much more sophisticated (not a simple uniformity criterion). Culik's algorithm can also be called a "multiresolutional" fractal compression. Unfortunately, the language of finite state machines isn’t very well understood in the signal processing and image compression community. I think that the algorithm can be better explained in terms of the multiresolutional fractal compression as follows.

Consider the following simple image

1	2	3	4	5	6	7	15
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
0	0	9	10	11	12	13	14
0	0	10	11	12	13	14	15

Fig. 1. An original image

and downsample it by picking a median pixel from each 2x2 square:

2	4	6	8
4	6	8	10
6	8	10	12
0	10	12	14

Fig 2. A lower resolution view of the image, Fig. 1

Downsampling one more time gives:

4	8
8	12

Fig. 3. Image Fig. 1 at resolution 4:1

Obviously there is a relation between an entire image on Fig. 3 and quadrants of a bigger picture, Fig. 2. We will look for a simple relationship of the form

quadrant = A * low-res-image + B

and use a primitive least-square estimate of coefficients A and B. In the example above, the formula of “self-similarity” is

quadrant #0 of a high-res image is equal to 1/2 * low-res image + 0
quadrant #1 of a high-res image is equal to 1/2 * low-res image + 4
quadrant #2 of a high-res image is equal to 1/2 * low-res image + 8 (1)
quadrant #3 of a high-res image is equal to 1/2 * low-res image + 4

where quadrants are numbered anti-clockwise starting from the upper-left corner. Here I harken back to my “formula of self-similarity”; indeed, it looks exactly the same, the only difference is in addition of the “intercept” term (and no rotation of the hi-res image). Now we can reconstruct Fig. 2 from Fig. 1 based on the formula above:

2	4	6	8
4	6	8	10
6	8	10	12
8	10	12	14

Fig 4. A reconstructed Fig. 2 using the formula of self-similarity

This looks very similar to the original picture, Fig. 2, but it's not identical. However, we disregard a single (point) error at this time. Thus, we can encode the picture on Fig.4 as Fig.3 plus the transformation matrix

T0	T1
T1	T3

Fig.5 Transformation matrix to encode Fig. 4

Where T0 means "take a 2x2 square and half the value of all the pixels", T1 means "half the value of all the pixels and then add 4", etc. We can apply the formula of the self-similarity again to Fig. 4 to obtain

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15

Fig. 6. Self-similar extrapolation of Fig. 4

Note that Fig.6 can be obtained from the initial small picture, Fig. 3, by using the following transformation matrix:

T0T0	T0T1	T1T0	T1T0
T0T1	T0T3	T1T1	T1T3
T1T0	T1T1	T3T0	T3T1
T1T1	T1T3	T3T1	T3T3

Fig. 7. Transformation matrix to encode Fig. 6

That is, if one applies transform T0 twice to the image on Fig. 3 (that is, divides all pixel values of Fig. 3 by four), one gets the upper-left 2x2 square of Fig 6. The next 2x2 square in a row is obtained by halving pixels of Fig. 3, adding 4, and halving again (which is exactly what the composition of transforms T0T1 stands for), etc. Fig.6 is very similar to the original picture, but not identical. The major difference is that the lower-left corner of Fig.1 is zero. We can modify the transformation matrix, Fig. 7, to read

T0T0	T0T1	T1T0	T1T0
T0T1	T0T3	T1T1	T1T3
T1T0	T1T1	T3T0	T3T1
T4T1	T1T3	T3T1	T3T3

Fig. 8. Transformation matrix to encode Fig. 9

which, applied to Fig. 3, gives

```
1  2  3  4  5  6  7  8
2  3  4  5  6  7  8  9
3  4  5  6  7  8  9 10
4  5  6  7  8  9 10 11
5  6  7  8  9 10 11 12
6  7  8  9 10 11 12 13
0  0  9 10 11 12 13 14
0  0 10 11 12 13 14 15
```

Fig. 9. Self-similar extrapolation of Fig. 3

This image is almost identical to the original picture, Figure 1 (like the IFS fractal compression, Culik's compression is almost always lossy, but the loss can be made "single point" one (uncorrelated)). On Fig. 8, transformation T4 means "set all pixels of the 2x2 square to zero". Obviously, Fig.8 has a very regular structure (it is nothing but the transformation matrix of Fig. 5 applied recursively to itself, with a small correction). Therefore, we don't need to transmit the matrix on Fig.8 in its entirety, it's enough to send only the correction to the decoder:

T4			

Fig. 10. Correction to the Transformation matrix to encode Fig. 5

In other words, we need to transmit only a significance map (binary map indicating where correction is needed) plus the correction itself. If it looks similar to the zerotree wavelet transform of Shapiro, it indeed is! In a nutshell, Culik's algorithm is a zerotree coding of "A*square+B"-type operations on image.

Of course, producing a low-resolution view of an image can be done in many different ways other than downsampling or picking a median value. For example, one can use a lower band of a wavelet transform of an image.

Note that Culik’s method can easily compress color pictures (in YIQ space), exploring similarities not only across the multiple resolutions but also across the color planes. Also note similarities of the method with the multistage predictor-corrector and the dynamic Markov model.

4. Vector Quantization

VQ is used a lot in compressing video and still imagery. Hierarchical VQ is the most promising way of reducing complexity of encoding /decoding, since table lookups are very fast. Of course one still needs to prepare these lookup tables (and here is where the NP-complexity of VQ lies). Also, the tables take a lot of space. A talk *Hierarchical Vector Quantization of Perceptually Weighted Block Transforms*, p.3, uses two adjacent pixels in a row as a 16-bit index in a look-up table yielding 8-bits of the “compressed” data. At the next stage, two vertically adjacent “compressed” pixels (outputs from the first stage) index a 16-bit-to-8-bit lookup table. Thus, after the two stages, a 2x2 square block of the image is represented by an 8-bit index, yielding 4:1 compression. The lookup tables for that transformation occupy 4x64K bytes. The lookup indices can be grouped into squares and compressed again using the look-up tables of the next level, giving 16:1 compression. It’s also possible to apply this hierarchical VQ technique to a preprocessed image, say, to a Haar, Hadamard or DCT transform of the original image. Still, examples given in the paper for a 16:1 compression of lenna aren’t that great: blockiness is apparent, even JPEG codes better (quantitatively speaking, at 0.5 bpp, JPEG gives almost 5db of PSNR gain compared to any of the hierarchical VQs proposed in the paper).

Indeed, VQ codes non-overlapping blocks of an image, each block being coded (approximated) regardless of others. Therefore, the reconstructed image is always blocky, with a noticeable tile-effect.

A few talks mentioned a residual vector quantizer, which vector quantizes the residual (prediction error) of a previous vector quantization stage. Once again, the idea of a multiresolutional, multistage predictor-corrector is manifested clearly.

5. Video

Some documents and even freeware implementations of a H.263 standard (low bitrate video-conferencing) can be downloaded from homer.nt.com/h.263

I saw a demo of QCIF (174x148?) 7.5 frames/sec, 10 kbs rate, “miss America” sequence. Looks very good! It uses DCT with SIM3 motion detection (16x16 block, +/- 15 pixel full search, then repeat for 8x8 subblocks and 4x4 subblocks if necessary). On a SunSparc 10, it takes 1 sec to do DCT and 14 sec/frame for motion prediction.

Motion estimation is easily (and better) done on a subband level

6. Matching pursuit

There were two talks about the matching pursuit, a greedy method of decomposing a vector (frame, image) into an overcomplete basis. In essence, the method picks up the “closest” basis function, finds the remainder, and keeps doing that. Still, it’s very time consuming. If the overcomplete basis has a property of being multi-resolutional (many of them **are** in practical applications), then the multiresolutional decomposition is faster. Moreover, it can even be done in parallel. I talked to both guys who used the matching pursuit (both of them from Berkeley), and they agreed with me. Ralph Neff gave me some time estimates: it takes 3-4 min per QCIF frame on a SunSparc 10 for matching pursuit. He admitted the algorithm is very far from being online.

7. Misc notes

Overcomplete bases: *Quantization of Overcomplete Expansions*, p. 13. The abstract says that the goal of using overcomplete sets of vectors (redundant bases of frames) is “to retain the computational simplicity of transform coding while adding flexibility like adaptation to signal statistics.” This is exactly what I said two years ago, in “Laplacian pyramid decompositions: A New Look”. This theoretical paper gives some justification to my claims. No practical results for the image compression. The paper uses the matching pursuit in vain (see above). Its consistency quantization condition is nothing but the consistency condition of the Laplacian pyramid

(quantize(decompose(compose(quantize(decompose(image)))) = quantize(decompose(image))

The intent of a reversible basis, *CREW:Compression with Reversible Embedded Wavelets*, p. 212 is to allow a wavelet decomposition using only integer arithmetic. The approach is almost identical to mine, the only difference is that I keep bits I lose while rounding in a separate band; he shifts them in into the high-pass band. I think my method is better (better for quantizing), as both low- and high-pass bands are consistently normalized in my method. His filter, two-six transform (up to normalization):

low-pass [1 1]
high-pass [-1 -1 8 -8 1 1]

looks very promising! It requires **no** multiplication/divisions.

Midterm talk by Michael Orchard, Beckman Institute, University of Illinois at Urbana-Champaign: Image coding is an experimental science. Therefore, experimental comparison (with the results of similar methods) is an important part of the research. Specifically, every paper on image compression that claims some improvement must illustrate the improvement by comparing compression results (numerically and visually) for some "standard" pictures.

PSNR! The DCC conference has definitely a signal-processing flavor: the Peak Signal-to-Noise Ratio was used almost by everybody, instead of MSE (mean-square error). MSE has more sense mathematically since it shows the approximation quality, that is, how well the reconstructed image approximates the original one in L_2 norm.

PSNR of a few compression schemes of a monochrome Lenna image

		PSNR (db)	
Compr ratio	Bits per pixel	JPEG	Plain VQ
8:1	1	37.7	32.5
	0.796	34.9	
16:1	0.5	34.7	30.5