Speeding up Fractal Image Compression

Behnam Bani-Eqbal
Department of Computer Science
University of Manchester
UK.

September 22, 1994

Abstract

Fractal coding of digital images offers many promising qualities. However the coding process suffers from the long search time of the domain block pool. In this paper, we obtain mathematical results on the distance measure used in the search. Then we derive an incremental procedure to bound the domain block pixels. We arrange the domain blocks in a tree structure, and utilise the procedure to direct the search. We show that this method speeds up the coding process by upto 50 times, without noticeable loss of image quality. Our procedure works in conjunction with other methods, such as block classification.

Subject terms: Digital Image compression, iterated function systems, fractal codes.

Address: Department of Computer Science,
University of Manchester,
Oxford Road, Manchester M13 9PL,
UK.

Email: behnam@cs.man.ac.uk

Tel No: +44-61-275-6152

Fax No: +44-61-275-6236

1 Introduction

Recently fractal compression of digital images has attracted much attention. It is based on the mathematical theory of iterated function systems (IFS) developed by Hutchinson [1] and Barnsley [2]. Its use for image compression was proposed by Jacquin in [3], [4] and Barnsley. Other variants of the IFS compression method have been reported in the literature, eg [5]. For a comprehensive survey of the literature on the fractal image compression, we refer the reader to [6]. The fractal theory and some important contributions are explained in a recent book [7] edited by Y. Fisher.

The basic idea of the IFS method is as follows. The image is partitioned into non-overlapping range blocks. For every block a similar but larger domain block is found. The data for the transformation mapping the larger block to the range block is recorded. The compressed image consists of all the transformation data. Decoding proceeds as follows. The transformations are applied to an arbitrary initial image, and the process is repeated. Provided the transformations are contracting, the images converge to a stable image, resembling the original picture.

The fractal method offers high compression ratio, good image quality, and resolution independence of the decoded image. Its disadvantage lies in the long search time for domain blocks. A review paper by Jacquin [8] summarises some recent work to speed up the search. Two types of methods have been investigated. The methods in the first category classify image blocks in some way. A range block is compared against the domain blocks of the same class. Originally, Jacquin [4] classified blocks on their edge content. Later, Jacobs et al. [9] used block brightness orientation. The other methods, eg [5], search a smaller pool of domain blocks, but rely on a richer set of transformations, eg polynomial translation terms.

In this paper, we present a novel scheme for speeding up the search. We arrange the domain block pool in a tree structure to direct the search. We show that it leads to upto 50 times improvement in the search time. Our scheme can be used in conjunction with the block classification methods to get a greater speed-up.

2 Iterated Function Systems

In this section, we explain our fractal compression scheme. Self similarity is the key to the fractal idea. We use the Jacquin method [3], but allow a linear offset term as in [10]. The Jacquin method is investigated further by Jacobs et al. [9].

The image is a digital grey level picture. We cover it by non-overlapping square range blocks of size $n \times n$, where normally $n = 4$. The first block is aligned with the bottom left corner of the picture. If the blocks do not reach upto the right or the top edge, then the margin can be covered by rectangular blocks. We assume that the blocks cover the image exactly. Next, for every range block, a similar but larger domain block is found. the domain blocks are of size $2n \times 2n$, and located anywhere on the image. The x, y coordinates of the lower left corner of the domain block determine its location. This can be on each pixel. However, some authors suggest that the blocks be aligned on a grid size of n or $n/2$. The more domain blocks, the better the decoded image quality, but the longer the compression takes.

The compression process is as follows. For every range block R, we search the domain

pool to find a block D and a transformation T such that $T(D)$ is the best match for R. The closeness is measured by a distance metric. The distance of blocks $A = (a_i)$ to $B = (b_i)$ $0 \leq i < m$ is the root mean squared (RMS) difference:-

$$d_{rms}(A, B) = \sqrt{\frac{1}{m}\sum(a_i - b_i)^2}$$

The transformation T is composed from a contracting map followed by a geometric map followed by a massic map. The contracting map shrinks the domain block to half the size, replacing 2×2 pixel areas by their average. The geometric map is one of the eight flips (or symmetries) of the square. We consider the blocks independently of the image. Some authors consider them in the image plane, so their geometric map involves a translation. For the sake of simplicity, we call the flipped contracted block a domain block. It will be clear from the context which kind of domain block is meant.

The massic map changes the contrast and brightness by a scale factor a_0 and an offset g_{ij}. The domain block (d_{ij}) is mapped to $(a_0 d_{ij} + g_{ij})$. The map T is contracting if $a_0 < 1$. However, Jacobs et al. [9] show that it is not necessary to enforce this condition for all the maps. The offset g_{ij} is a constant in the original paper of Jacquin [4]. Later researchers used more general forms. For instance, [10] suggests a linear term in the indices i, j:-

$$g_{ij} = a_1 + a_2 i + a_3 j$$

Our method works with either form of g. Monro and Dudbridge [5] set g_{ij} to a third degree polynomial in i, j, but used a very restricted set of domain blocks.

The constants in the massic map are chosen to minimise the distance from the (contracted flipped) domain block D to the range block $R = (r_{ij})$. This amounts to minimising the positive definite quadratic form $F = \sum(a_0 d_{ij} + g_{ij} - r_{ij})^2$. Thus a_i are the solutions of the equations:-

$$\partial F / \partial a_i = 0$$

Let us call F_{min} the minimum F obtained from the solutions. The compression algorithm runs as follows. Given a range block, the minimum difference F_{min} with every domain block is calculated. The transformation parameters with the smallest F_{min} are stored as the IFS code. This process is repeated for all the range blocks.

```
For every range block,
    For every domain block,
        Shrink and flip in 8 ways.
        Determine the best a_0, g, and F_min.
    Choose the block giving the smallest F_min,
    Store the transformation.
```

The complexity of this scheme lies in the search for the best domain blocks.

2.1 The Massic Map

We split the computation of the massic map into two steps, the computation of a_0, then the computation of g. Effectively, we decouple the computation of the scale from that of the offset.

We state the problem in terms of a vector space with an inner product (or distance measure). This idea was proposed originally by Oien et al. in [10]. Let $\mathbf{V}$ be the n^2 dimensional space of all $n \times n$ square blocks (with real coefficients). This is a vector space with the usual inner product. Namely, for two blocks $A = (a_{ij})$ and $B = (b_{ij})$, $0 \leq i, j < n$:-

$$A \circ B = \sum_{i,j} a_{ij} b_{ij}, \ |A|^2 = A \circ A$$

The problem of minimising F amounts to finding the minimum distance of R to the subspace generated by D and the offsets. Let $\mathbf{G}$ be the subspace generated by the offsets. Define the blocks A_1, A_2, A_3 with the following entries:-

$$A_1(i,j) = 1, \ A_2(i,j) = i, \ A_3(i,j) = j.$$

If the offsets are constant, then A_1 is a basis of $\mathbf{G}$. If they are linear, then A_1, A_2, A_3 are a basis of $\mathbf{G}$. The problem is to choose a_0 and an offset vector $G \in \mathbf{G}$ such that

$$F = |a_0 D + G - R|^2$$

is minimised. Now, take the orthogonal complement of $\mathbf{G}$ in $\mathbf{V}$, and let D' and R' be the orthogonal projections of D and R onto this subspace. That is, let

$$D = D' + G_1, \ R = R' + G_2, \ D' \circ \mathbf{G} = 0, \ R' \circ \mathbf{G} = 0.$$

D' and R' are computed explicitly below. Putting $G' = G + a_0 G_1 - G_2$, we have :-

$$F = |a_0 D' + G' - R'|^2$$

Now D', R' are orthogonal to G'. Thus F is minimised by $G' = 0$ and $a_0 = D' \circ R' / |D'|^2$, provided $D' \neq 0$. So $G = G_2 - a_0 G_1$, and

$$F_{min} = |R'|^2 - D' \circ R' / |D'|^2$$

If $D' = 0$, then a_0 can be anything and $F_{min} = |R'|^2$. To carry out the F_{min} computation, the blocks are projected and stored. Then $|R'|^2$ and $|D'|^2$ are computed and stored. A comparison of D against R requires $D' \circ R'$. We call it the full distance computation. The compression algorithm requires the distance computation for every domain and range block, a time-consuming operation. Our improvement drastically cuts down on the distance computations, at the additional cost of a tree navigation.
Here are the explicit formulas. In the constant case, we have $D' = D - d_1 A_1$, where

$$d_1 = \frac{1}{n^2} \sum d_{ij}$$

In the linear case, we have $D' = D - d_1 A_1 - d_2 A_2 - d_3 A_3$, where

$$\begin{aligned} n^2(n+1)d_1 &= (7n-5)\sum d_{ij} - 6\sum i d_{ij} - 6\sum j d_{ij} \\ n^2(n^2-1)d_2 &= 12\sum i d_{ij} - 6(n-1)\sum d_{ij} \\ n^2(n^2-1)d_3 &= 12\sum j d_{ij} - 6(n-1)\sum d_{ij} \end{aligned}$$

Similarly for R'.

3 Tree search

The purpose of this section is to use the F_{min} formula above to speed up the search. It turns out that the domain blocks can be arranged in a tree structure so that those with bounded F_{min} can be identified efficiently. In this section only, we consider a block as a one dimensional indexed set of numbers $A = \{a_i;\ i \in I\}$ where I is any finite set, and a_i is a positive or negative integer. In our application, $I = \{(i,j); 0 \le i,j < n\}$ is the pixel positions. The blocks are the projections of the domain and the range blocks as above, where the pixels are rounded to the nearest integer.

For two blocks $A = (a_i)$ and $B = (b_i)$, we define

$$A \circ B = \sum a_i b_i,\ |A|^2 = \sum a_i^2,\ d(A,B) = \min_s \sum (sa_i - b_i)^2$$

The minimum is achieved by $s = A \circ B / |A|^2$ and $d(A,B) = |B|^2 - (A \circ B)^2/|A|^2$, unless $A = 0$. In that case, $d(A,B) = |B|^2$. Note that $d(A,B) \le |B|^2$ and $F_{min} = d(D', R')$.

3.1 Mathematical Results

Let there be a collection of domain blocks $\{A\}$, a fixed range block B and a error $e \le |B|^2$. This section presents results to identify the A such that $d(A,B) \le e$. The proofs of the results are easy and so are omitted.

In lemma 1, for a subset J of I, we define a corresponding sub-block $A' = \{a_j;\ j \in J\}$. We define B' similarly.

Lemma 1 *Let A', B' be sub-blocks of A, B. Then $d(A', B') \le d(A, B)$.*

Lemma 2 bounds the A pixels incrementally.

Lemma 2 *Let $A = (A', a)$, $B = (B', b)$, and $d(A', B') = e'$. Suppose that $A \ne 0$ and $e' \le e \le |B|^2$. If $e \ne |B'|^2$, then define*

$$c = b\frac{A' \circ B'}{|B'|^2 - e},\ \ d = \frac{|A'|\sqrt{(|B|^2 - e)(e - e')}}{||B'|^2 - e|}$$

We have $d(A,B) \le e$ if and only if a satisfies one of the following conditions:-

1. *If $e < |B'|^2$, then $|a - c| \le d$.*
2. *If $e > |B'|^2$, then $|a - c| \ge d$.*
3. *If $e = |B'|^2$, then $2ab(A' \circ B') \ge b^2|A'|^2 - (A' \circ B')^2$.*

Given a partition of the pixels I into disjoint subsets, we construct a new block $\overline{A}$ from A. It has a pixel for each subset, and the pixel value is the average of the A pixels in the subset. We say $\overline{A}$ is averaged from A. Lemma 3 transfers a bound to the averaged blocks.

Lemma 3 *Let $\overline{A}, \overline{B}$ be averaged from blocks A, B. Let k be the minimum size of the partition subsets. Then*

$$d(\overline{A}, \overline{B}) \le \frac{1}{k} d(A,B)$$

3.2 Tree structure

Recall that we have a collection of domain blocks $\{A\}$. As several blocks may have identical pixel values, so we assume that they are identified by a unique tag. We want to arrange the domain blocks in such a way that, given a range block B and an error e, we can enumerate the block tags with $d(A,B) \leq e$ quickly.

The idea is based on the results of the previous section. Lemma 1 says that the bound holds for any sub-block A'. If $A' = (A'', a)$, then Lemma 2 limits the range of a. We order the pixel positions in some arbitrary way, so that the blocks are $A = (a_0, a_1, \cdots, a_{m-1})$. We string them into a tree as follows. Each node (except the root) is labelled by a pixel value. The root has a distinct child for every value of the first pixel a_0. Considering a general node, let the pixel values on the path from the root to it be $a_0, a_1, \cdots, a_{k-1}$. We construct the subset of domain blocks which have these values for the first k pixels, and then look at the $k+1$ pixel of these blocks. The node has a child for each such value, and the child is labelled by it. The tree may be grown in this way, to the depth of m. Each node of the tree defines a sub-block, its pixels are the the values on the path from the root. Finally, a leaf node stores the tags of the blocks it defines.

In our implementation, the tree is grown by inserting the blocks one by one, each as an ordered string of pixels. In the full tree, a node can have as many children as there are pixel values ($-255, \cdots, 255$ in the constant offset case). The branching degree is too great to allocate storage for the pointers to all the children. However, the search algorithm below requires to move from a parent to only one child and then steps to its siblings. So it is sufficient to keep the children in an ordered linked list, and the parent stores a pointer to the head of the list.

3.3 Search Algorithm

Recall the purpose of the algorithm is to home in onto the A block tags with $d(A,B) \leq e$. The blocks are arranged in a tree as above. The tree is traversed in a breadth first manner. That is, at each level, a list of nodes is constructed. The list comprises exactly the nodes whose sub-blocks A' satisfy $d(A', B') \leq e$. The first list consists of all the children of the root (except the node labelled by 0 if $b_0^2 > e$). Once a list at a particular level is constructed, it is examined node by node to construct the next level list. Suppose the node under examination corresponds to the sub-block A'. For convenience, we store the values $|A'|^2, A' \circ B'$ with the node. We compute $e' = d(A', B')$. Lemma 2 specifies the range of the next domain pixel a. We step to the node's list of children and search it for values in the specified range. We add the nodes falling in the range to the end of the next list and we set their stored values to:-

$$|A|^2 = |A'|^2 + a^2; \;\; A \circ B = A' \circ B' + ab.$$

This process is carried out until the list at the depth m is constructed. The block tags stored in the nodes on this list are exactly those needed.

In the implementation , we tighten the search by reducing the error e according to the level. That is, we adjust e to $\alpha_l e$ at the level l. Everythings else runs as before. In the experiments we set the adjustment factor α_l to $\alpha_0 = 0.3$, $\alpha_1 = 0.5$, $\alpha_2 = 0.7$, $\alpha_3 = 1$.

4 Speeding up Compression

In this section we discuss the application of our tree search to the compression process. We project the domain and range blocks onto the orthogonal complement of the offset space. Then we quantise the pixel values, say to integers. Let A or B refer to a domain or a range block after these operations. Given B, we must find a block A with the minimum $d(A,B)$. We know that $d(A,B) \leq |B|^2$, for any block A. However, according to the collage theorem [4], the compression fidelity relies on the fact that some A will give a much smaller minimum difference. In fact, the distribution of the ratio $|B|^2/\min d(A,B)$ relates to the fittness of the collage. We calculated the distribution for several standard pictures, and found that, for over 90% of the range blocks, the ratio is greater than 3. Thus we can write:-

$$\min_A d(A,B) \leq \frac{1}{\alpha}|B|^2$$

Where $\alpha = 3$ works for most range blocks. We use this fact to reduce the number of domain blocks for which the full distance computation is done. We arrange the domain blocks in a tree as above. Then, given B, we set $e = |B|^2/\alpha$, and we traverse the tree to construct the candidate domain list. The distance computations are carried out only for the blocks on the list.

In practice it turns out that this method is still inefficient. The reason is revealed when we examine any particular tree. We construct the tree for the Lena picture, figure 3. We use 4×4 contracted domain blocks, and insert them in the tree to a depth of 16. Table 1 shows the average branching degrees, and the total memory used. The branching degree is the number of children of the nodes on a particular level, on average. The first row shows the degrees for the case where the blocks are flipped 8 ways. In the second row, the blocks are not flipped. As can be seen, nearly all the branches are concentrated on the first three levels. Then, most of the subtrees at level 4 are straight lines to the leaves. These paths waste both memory cells and cpu cycles.

Further improvement requires a reduction of the tree depth. To achieve that, and to keep the maximum pixel information, we average pixel subsets. We partition the pixel positions into disjoint subsets, and then average them. Figure 1 shows the 4×4 partition scheme, where the shading indicates the subsets. Let us call the resulting blocks $\overline{A}$ and $\overline{B}$. Note that every block is partitioned the same way. If A is the best match for B, then lemma 3 and the above bound gives $d(\overline{A},\overline{B}) \leq |B|^2/k\alpha$. We can set e to the right hand side and proceed. However we found empirically that a multiple of $|\overline{B}|^2$ is a better bound. So we use,

$$e = \frac{1}{\beta}|\overline{B}|^2$$

We found that a wide range of β values (from 20 to 100) gives a good image quality. The larger β values narrow the search more sharply, thus speeding the compression, but at some loss of the image quality. We found that $\beta = 100$ gives generally good results. With a large β, the search turns out an empty list occasionally. When this happens, we adjust β down, and try again.

The last row in table 1 shows the branching degrees and the memory requirement under the 4×4 averaging scheme of figure 1. We insert all the 8 way flips of the domain blocks. The average number of block tags stored in a leaf is 2.5. Actually, we quantised

the first two pixels more coarsely, so as to reduce the branching degrees further. We used $x \to 4I(x/4) + 1$, where $I(x)$ is the integer part of x. This produced good match lists.

We also experimented with 6×6 blocks. We used the scheme in figure 2, which is invariant under the 8 flips. It reduces the memory requirement, as only the unflipped blocks need be inserted. Here is the final algorithm.

For every domain block
    Contract, flip, project, average, quantise.
    Insert into tree.
For every range block B
    Project, average, quantise $\to \overline{B}$.
    Set $e = |\overline{B}|^2/\beta$.
    Search tree, produce list of domain blocks.
    while list is empty
        Adjust β down,
        Search tree.
    Search the list to find A with the minimum $d(A, B)$,
    Store its transformation parameters.

The tree algorithm can be modified to work naturally with the block classification schemes. The domain blocks are divided into classes and organised into separate trees. The range blocks are searched for in the trees of the same class.

5 Experimental Results

We have programmed our algorithm in the 'C' language, and tested it on standard images. We show the results for the 'Lena' image, Figure 3, which is 256×256 pixels digitised at 8 bits. We select 4×4 range blocks. The domain blocks are spaced on a 2×2 grid. Figure 4 is decoded from the IFS code using complete search. Then we used the tree technique with $\beta = 20$ and $\beta = 100$. Figures 5 and 6 show the decoded pictures. Table 2 compares the performance of the tree search against the full search method. The second column is the average number of domain blocks used in the full difference computation. The third column is the RMS difference between the original image and the reconstructed image. The fourth column is the run-time of the algorithm in seconds, measured on a SUN Sparcstation 10 model 30. the final column is the speed-up factor against full search. Even with $\beta = 100$, there is no noticeable degradation in image quality, but the search speeds up by a factor of 58. Table 3 shows the performance of the tree search for some other images.

6 Conclusions

We have presented a novel technique of speeding up the fractal compression and showed that it leads to an order of magnitude speedup gain against the full search method. Our method can be used also in conjunction with other speed-up methods to lead to even better gain in the compression time.

References

[1] J. E. Hutchinson 'Fractals and Self-Similarity', *Indiana Univ Math Journal, Vol 35, 1981, p5.*

[2] M. F. Barnsley, *'Fractals Everywhere'*, Academic Press, San Diego, CA, 1988.

[3] A. E. Jacquin, 'A fractal theory of iterated Markov operations, with applications to digital image coding', *Ph.D. Thesis, Department of Mathematics, Georgia Institute of Technology, 1989.*

[4] A. E. Jacquin, 'Image Coding Based on a fractal Theory of Iterated Contractive Image transformations', *IEEE Transactions on Image processing, Vol 1, No 1, 1992.*

[5] D. M. Munro, F. Dudbridge, 'Fractal Block Coding of Images', *Electronics Letters, Vol 28, No 11, May 1992.*

[6] D. Saupe, R. Hamzaoui, 'A Guided tour of the Fractal Image Compression Literature' *ACM SIGGRAPH 94 course notes. Available by ftp from fidji.informatik.uni-freiburg.de under /papers/fractal/Guide.ps.Z*

[7] Y. Fisher, Editor, *Fractal Image Compression: Theory and Applications to Digital Images*, Springer Verlag, 1994.

[8] A. E. Jacquin, 'Fractal Image Coding: A Review', *Proceedings of the IEEE, Vol 81, No 10, October 1993.*

[9] E. W. Jacobs, Y. Fisher, R. D. Boss 'Image Compression: A Study of the iterated transform method' *Signal Processing, Vol 29, 1992, pp 251 - 263.*

[10] G. E. Oien, S. Lepsoy, T. A. Ramstad, 'An inner product space approach to image coding by contractive transformations', *in Proc. of ICASSP-91, 1991, pp 2773-2776.*

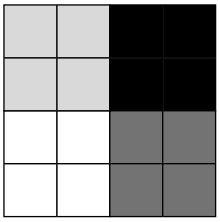

Fig 1. 4×4 square

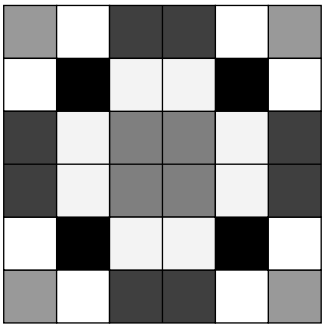

Fig 2. 6×6 square

Fig 3. Lena image $256 \times 256 \times 8$.

Fig 4. Decoded Lena, full search.

Fig 5. Decoded Lena, tree search $\beta = 20$.

Fig 6. Decoded Lena, tree search $\beta = 100$.

Domain blocks flipped?	Average deg Level 1	Level 2	Level 3	Level 4	All other levels	Memory Used
Yes	314	50	3	1.5	1	16MB
No	215	20	2	1	1	2MB
Yes, Averaged	56	35	9	2	-	1.5MB

Table 1: The tree branching data for the Lena image.

Lena	d-blocks	rms	time(sec)	speedup
full search	125000	6.7	8750	1
tree $\beta = 20$	1954	7.8	340	25
tree $\beta = 100$	306	8.7	150	58

Table 2: The performance data for the full search and the tree search for Lena.

picture	d-blocks	rms	best rms	time(sec)
Ape $\beta = 40$	675	26	22.6	215
Ape $\beta = 100$	206	27.8	-	207
Salad $\beta = 20$	2123	9.1	7.4	465
Salad $\beta = 100$	295	11.1	-	128
Hayes $\beta = 100$	280	5.8	-	94
Tools $\beta = 100$	118	4.2	-	52
Clown $\beta = 100$	580	8.5	-	187

Table 3: The tree search data for other images at $256 \times 256 \times 8$.

Figure 1: 4×4 square.
Figure 2: 6×6 square.
Figure 3: Lena image $256 \times 256 \times 8$.
Figure 4: Decoded Lena, full search.
Figure 5: Decoded Lena, tree search $\beta = 20$.
Figure 6: Decoded Lena, tree search $\beta = 100$.
Table 1: The tree branching data for the Lena image.
Table 2: The performance data for the full search and the tree search for Lena.
Table 3: The tree search data for other images at $256 \times 256 \times 8$.