## C.11 From Classification to Multi-dimensional Keys

*D. Saupe*

Fractal image compression allows fast decoding but tends to suffer from long encoding times. This project introduces a new twist for the encoding process. During encoding a large domain pool has to be repeatedly searched, which by far dominates all other computations in the encoding process. If the number of domains in the pool is $N$, then the time spent for each search is linear in $N$, $O(N)$. Previous attempts to reduce the computation times employ classification schemes for the domains based on image features such as edges or bright spots (from 3 in [48] up to 72 in [27]). Thus, in each search only domains from a particular class need to be examined. However, this approach reduces only the factor of proportionality in the $O(N)$ complexity.

In this project, we replace the domain classification by a small set of real-valued multi-dimensional keys for each domain. These keys are carefully constructed such that the domain pool search can be restricted to the nearest neighbors of a query point. Thus, we may substitute the sequential search in the domain pool (or in one of its classes) by multi-dimensional nearest neighbor searching. There are well known data structures and algorithms for this task which operate in logarithmic time $O(\log N)$, a definite advantage over the $O(N)$ complexity of the sequential search. These time savings may provide a considerable acceleration of the encoding and, moreover, facilitate an enlargement of the domain pool, potentially yielding improved image fidelity.

For simplicity, we present the approach for a special one-dimensional case and generalize later. We consider a set of $N$ vectors $x^{(1)}, \ldots, x^{(N)} \in \mathbb{R}^d$ (representing $d$ pixel values in each of the $N$ domains of the pool) and a point $z \in \mathbb{R}^d$ (represending a range with $d$ pixels). We let $E(x^{(i)}, z)$ denote the smallest possible least squares error of an approximation of the range data $z$ by an affine transformation of the domain data $x^{(i)}$. In terms of a formula, this is $E(x^{(i)}, z) = \min_{a,b\in\mathbb{R}} \|z - (ae + bx^{(i)})\|^2$, where $e = \frac{1}{\sqrt{d}}(1, \ldots, 1) \in \mathbb{R}^d$ is a unit length vector with equal components. Computing the optimal $a, b$ and the error $E(x^{(i)}, z)$ is a costly procedure, performed for all of the domain vectors $x^{(1)}, \ldots, x^{(N)}$ in order to arrive at the minimum error $\min_{1\le i\le N} E(x^{(i)}, z)$. This staggered minimization problem needs to be solved for many query points $z$ in the encoding process (i.e., for all ranges). We now show that this search can be done in logarithmic time $O(\log N)$. The following lemma provides the mathematical foundation for the solution. We use the notation $d(\cdot,\cdot)$ for the Euclidean distance and $\langle\cdot,\cdot\rangle$ for the inner product in $\mathbb{R}^d$, thus, $\|x\| = d(x, 0) = \sqrt{\langle x, x\rangle}$.

**Lemma C.1** *Let $d \ge 2$, $e = \frac{1}{\sqrt{d}}(1, \ldots, 1) \in \mathbb{R}^d$ and $X = \mathbb{R}^d \backslash \{re \mid r \in \mathbb{R}\}$. Define the normalized projection operator $\phi : X \to X$ and the function $D : X \times X \to [0, \sqrt{2}]$ by*

$$\phi(x) = \frac{x - \langle x, e\rangle e}{\|x - \langle x, e\rangle e\|} \quad \textit{and} \quad D(x, z) = \min(d(\phi(x), \phi(z)), d(-\phi(x), \phi(z))).$$

*For $x, z \in X$ the least squares error $E(x, z) = \min_{a,b\in\mathbb{R}} \|z - (ae + bx)\|^2$ is given by*

$$E(x, z) = \langle z, \phi(z)\rangle^2 g(D(x, z)) \quad \textit{where} \quad g(D) = D^2(1 - D^2/4).$$

**Proof:** The least squares approximation of $z$ by a vector of the form $ae + bx$ (resp. $ae + b\phi(x)$) is given by the projection

$$\mathrm{Proj}(z) = \langle z, e\rangle e + \langle z, \phi(x)\rangle \phi(x) = \langle z, e\rangle e + \langle z, \phi(z)\rangle \langle \phi(x), \phi(z)\rangle \phi(x),$$

where the last equation is derived from $z = \langle z, e\rangle e + \langle z, \phi(z)\rangle \phi(z)$ (see Figure C.3). The least squares error in this approximation is calculated as

$$E(x, z) = \|z - \mathrm{Proj}(z)\|^2 = \langle z, \phi(z)\rangle^2 (1 - \langle \phi(x), \phi(z)\rangle^2).$$

Since $d(\pm\phi(x), \phi(z)) = \sqrt{2(1 \mp \langle \phi(x), \phi(z)\rangle)}$ we have $D(x, z) = \sqrt{2(1 - |\langle \phi(x), \phi(z)\rangle|)}$. Solving the last equation for $|\langle \phi(x), \phi(z)\rangle|$ and inserting the square of the result in the formula for $E(x, z)$ completes the proof. ∎

The lemma states that the least squares error $E(x, z)$ is proportional to the simple function $g$ of the Euclidean distance $D$ between the projections $\phi(x)$ and $\phi(z)$ (or $-\phi(x)$ and $\phi(z)$). Since $g(D)$ is a monotonically increasing function for $0 \le D \le \sqrt{2}$ we conclude that *the minimization*

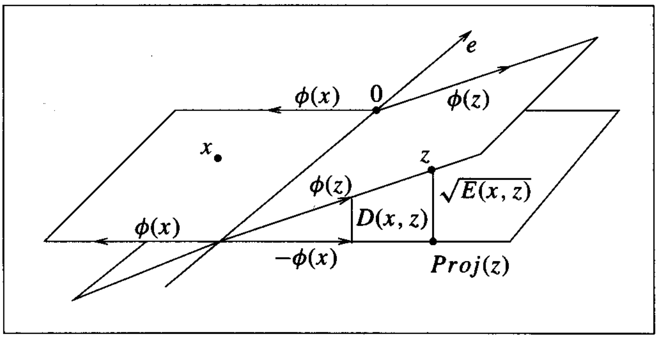

Figure C.3: Illustration of the geometry underlying the lemma.

*of the least squares errors $E(x^{(i)}, z)$ for $i = 1, \ldots, N$ is equivalent to the minimization of the distance expressions $D(x^{(i)}, z)$.* Thus, we may replace the computation and minimization of $N$ least squares errors $E(x^{(i)}, z)$ by the search for the nearest neighbor of $\phi(z) \in \mathbb{R}^d$ in the set of $2N$ vectors $\pm\phi(x^{(i)}) \in \mathbb{R}^d$.

The problem of finding closest neighbors in Euclidean spaces has been thoroughly studied in computer science. For example, a method using $k$-d trees that runs in expected logarithmic time is presented in [31] together with pseudo code. After a preprocessing step to set up the required $k$-d tree, which takes $O(N \log N)$ steps, any search for the nearest neighbors of query points can be completed in expected logarithmic time $O(\log N)$. An even more efficient method that produces a list of so-called approximate nearest neighbors is presented in [1].

We conclude with some remarks on generalizations and implications.

1. In the above we have made the assumption that the number of components in all vectors is the same, namely $d$. This is not the case in practice, where small and large ranges need to be covered by domains that can be of a variety of sizes. To cope with this difficulty, we settle for a compromise and proceed as follows. We down-filter all ranges and domains to some prescribed dimension of moderate size, for example, $d = 8$ or $16$. This allows the processing of arbitrary domains and ranges, however, with the implication that the formula of the lemma is no longer exact but only approximate.

2. For a given range, not all domains from the pool are admissible. There are restrictions on the resulting number $b$, the contraction factor of the affine transformation. This is necessary in order to ensure convergence of the iteration in the image decoding. Also, one imposes bounds on the size of the domain. To take that into consideration, we search the *entire* domain pool not only for the nearest neighbor of the given query point but also for, say, the next 10 or 20 nearest neighbors (this is called the all-nearest-neighbors problem and can still be solved in logarithmic time using a priority queue). The non-admissible domains are discarded from this set, and the remaining domains are compared using the ordinary least squares approach. This also alleviates the problem mentioned in the previous remark: that the estimate by the lemma is only approximate. Some preliminary empirical tests (using a $256 \times 256$ image) have shown that although the best domain for a given range is often not the first entry in the priority queue, it usually is among the first six or so. Thus, the result is identical to a search through the *entire*

domain pool.
3. Our technique for encoding one-dimensional image data readily carries over to (two dimensional) images. There are two possible approaches. The first is simply to represent an image by a one-dimensional scan, for example, by the common scan-line method or (better) using a Hilbert or Peano-scan. However, in most implementations, image domains and ranges are squares or rectangles. In that case image data from a rectangle, for example, can first be transformed to a square, then down-filtered to, say, an array of $3 \times 3$ or $4 \times 4$ intensity values. After applying the normalized projection operator $\phi$ we can use the result as a multi-dimensional key in the exact same fashion as described before.
4. We make two technical remarks concerning memory requirements for the $k$-d tree. First, it is not necessary to create the tree for the full set of $2N$ keys in the domain pool. We need to keep only one multi-dimensional key per domain if we require that this key has a non-negative first component (multiply the key by $-1$ if necessary). In this set-up a $k$-d tree of all $2N$ vectors has two symmetric main branches (separated by a coordinate hyperplane). Thus, it suffices to store only one of them. Second, there is some freedom in the choice of the geometric transformation that maps a domain onto a range. A square, for example, may undergo any of the 8 transformations of its symmetry group. This will create 7 additional entries in the $k$-d tree, enlarging the size of the tree. However, we can get away without this tree expansion. To see this, just note that we may instead consider the 8 transformations of the range (or just some of them) and search the original tree for nearest neighbors of each one of them.
5. The $O(N \log N)$ preprocessing time required to create the data structure for the multidimensional search is not a limitation of the method. To see that, observe that the number of the ranges to be considered is of the same order $O(N)$ as the number of domains. Thus, the sum of all search times, including the preprocessing, is $O(N \log N)$, to be compared with $O(N^2)$ for the method using sequential search.
6. Another approach that uses several real numbers as keys for domains and ranges is given in terms of so-called Rademacher labellings in [9]. These labels are used to make a pre-selection of the domain blocks much in the spirit of the usual classification. Another concept using a similarity measure related to the above lemma is presented in Section 9.3.

## C.12 Polygonal Partitioning

The scheme described in Chapter 6 partitions ranges into rectangles. It is possible to generalize this to a method that is much more versatile, but not particularly memory intensive. Rather than being partitioned only horizontally or vertically, the image may be partitioned along lines that run at angles of 0, 45, 90, or 135 degrees, as in [83]. This still leads to a tree-structured partitioning of the image, and only one extra bit is required per node to specify the orientation of the partition.

## C.13 Decoding by Pixel Chasing

Consider a modification of the notation of Chapter 11 in which $x \in \mathbb{Z}^2$ represents a coordinate in a matrix representing an image. Also, let $f(x)$ represent the value of the matrix element, or pixel, at position $x$. When the domain decimation consists of subsampling, a decoding iteration consists of computing

$$f(x) = s(x)f(m(x)) + o(x)$$

for each $x$. Note that when this equation is satisfied, then we have found the fixed point. Rather than computing this equation once for each pixel and repeating, we can compute it repeatedly for each pixel. That is, fix $x$, and choose some random initial image $f_0$. Now "chase" the pixel values through the action of $m$. Compute

$$f_1(x) = s(x)f_0(m(x)) + o(x).$$

We really want to know the fixed point $f_\infty$, but if $m(x) = y$ and $f_1(y)$ is known, then we can write the fixed point equation

$$f_1(x) = s(x)f_1(y) + o(x).$$

We are then done for $f(x)$, and must go on to the next value of $x$. If we don't know $f_1(m(x))$, which is the case initially, we compute

$$f_1(y) = f_1(m(x)) = s(m(x))f_0(m(m(x))) + o(m(x)),$$

so that

$$\begin{aligned} f_2(x) &= s(x)f_1(y) + o(x) \\ &= s(x)\,[s(m(x))f_0(m(m(x))) + o(m(x))] + o(x) \\ &= s(x)s(m(x))f_0(m(m(x))) + s(x)o(m(x)) + o(x), \end{aligned}$$

and so on. Again, if we know $f_2(m(m(x)))$, we can substitute this for $f_0(m(m(x)))$ and we will be done. (Eventually we will find a loop, leading to a system that can be solved, but this is not the point here. This is discussed, however, in [59].) Even though each $f_0(m^{\circ n}(x))$ is not well known, we can find an excellent estimate of the fixed point $f_\infty(x)$. We know (empirically, for example) that 10 iterations are sufficient to decode the image to a close approximation of the fixed point, so that $f_{10}(x)$ will be nearly equal to $f_\infty(x)$. As the algorithm progresses, the first pixels will require computation of $f_{10}(x)$, but eventually, the image will get filled up so that it will be sufficient to compute $f_n(x)$ for $n < 10$. On average, for example, when half the image is computed, there is a 50% chance that the $f_1(x)$ will be nearly exact for an $x$ which is not yet computed. This means that there is some reduction in the number of computations that must be carried out.

We can compute how many operations this will take. Assume that the proportion of undecided pixels is $p$ and that the action of $m$ is random. Then the proportion of pixels whose value will be determined through one reference of $m$ is $p$. The proportion that will require two references through $m$ is $p(1-p)$; three references requires $p(1-p)^2$, and so on. But since we allow a maximum of 10 references, the proportion of the pixels requiring 10 references is $(1-p)^9$. So the average pixel will require $10(1-p)^9 \sum_{n=1}^{9} n\, p(1-p)^{n-1}$. Since the proportion ranges from 0 to 1, we can estimate the average number of references by

$$\int_0^1 \sum_{n=1}^{10} n\, p^n \approx 3.$$

That means that on average, each pixel will require 3 references, not 10. Each reference is equivalent to an iteration, so that the computational burden of this scheme is comparable to doing just 3 iterations in a normal iterative decoding method. However, this model just accounts for two sorts of pixels – "known," with 10 references or more, and "unknown," whose number of references is considered 0. The number of operations can be significantly reduced if we keep track of populations of pixels with 0, 1, ...,9 ,10 computed references. With these improvements, the decoding scheme should be significantly improved. It should also be pointed out that most of the pixels will have values that are equivalent to doing more than 10 references, since they will be iterates of pixels that have a minimum of 10 references.

Pg. 59

## C.14 Second Iterate Collaging

Given an image $f$, the Collage Theorem motivated the minimization of $d(f,\ W(f))$. However, what we really want to minimize is $d(f,\ x_W)$, where $x_W$ is the fixed point of $W$. Unfortunately, this is not usually possible. However, it may be possible to minimize $d(f,\ W(W(f)))$. Since $W(W(f))$ is closer to the fixed point than $W(f)$, this should improve the encoding. One approach might be to find proper domains and ranges using the collage minimization, but to find the optimal scaling and offset values using the second iterate condition.

## C.15 Rectangular IFS Partitioning

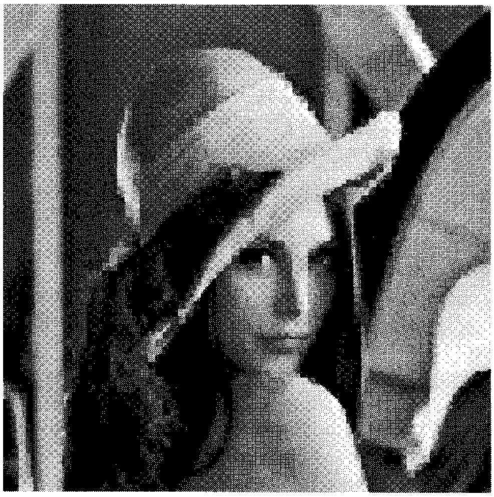

Figure C.4: An encoding of 256 × 256 Lenna using a simple IFS on 8 × 8 blocks.

Figure C.4 shows an encoding of 256 × 256 Lenna using a simplification of Dudbridge's scheme in [21]. The original image was partitioned into 8 × 8 blocks, each of which was encoded using a simple square IFS as in Figure 12.1 with an affine transformation of the grey levels. This is equivalent to finding a fixed point for the operator

$$\tau(f) = s(x)f(m(x)) + o(x),$$

as in Chapter 11, with $s(x)$ and $o(x)$ piecewise constant over the quadrants of the block and with $m(x) = 2x \bmod 1$. In this notation, $x = (u, v)$ represents a point in $\mathbb{R}^2$, so that

$$m \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 2u \bmod 1 \\ 2v \bmod 1 \end{bmatrix}.$$

Notice that edges in the image that run at 0, 45, and 90 degrees are well encoded, but that the other edges are not. This is because the map $m(x) = 2x \bmod 1$ has a natural affinity for such things. For example, if two of the diagonal $s(x)$ values are zero, then the attractor for the set will naturally be a line along the other diagonal of the block. If edges at other angles are also well encoded, the scheme may be viable.

To do this, we can consider a more general $m(x)$. For example, if

$$m \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 3u \bmod 1 \\ 2v \bmod 1 \end{bmatrix},$$

then we should be able to encode edges with slopes of $-2/3$ and $2/3$ well. In this case, $s(x)$ and $o(x)$ would be piecewise constant over six rectangles, not four as before. To encode an image, a small set of possible $m(x)$-functions can be tested, with the optimal one used for the encoding.

## C.16 Hexagonal Partitioning

Consider the following process, shown in Figure C.5, which constructs a hexagonal tiling of the plane. Begin with a subset of the plane $A_0$ which is a filled hexagon; surround $A_0$ with six copies of itself, laid adjacent to $A_0$; scale the new subset to exactly fit in $A_0$ and call this $A_1$. Now repeat. The limit is a fractal that forms a tiling of the plane and which is built up of seven copies of itself.

Just as the square is a basic tile for the quadtree partition, we can make a hept-tree partition based on this shape. As shown in Figure C.6, if the pixels on every other row are shifted one half of a pixel width, the centers of the pixels now lie on a roughly hexagonal tile; the aspect ratio is not 1, but the connectivity is the same. There are several obvious difficulties; it is easier to think of pixels as squares which fit nicely to form larger squares than it is to think of them as hexagonal tiles.

The potential advantages of this scheme are:

- The overlap between ranges can be used to minimize artifacts along range boundaries. (This follows the long tradition of claiming bugs as features.)

- Since the range boundaries are not straight edges, artifacts will be less distracting.

- The domain-range map can take on 12 orientations. This breaks away from the strict vertical and horizontal self-similarity of the quadtree scheme.

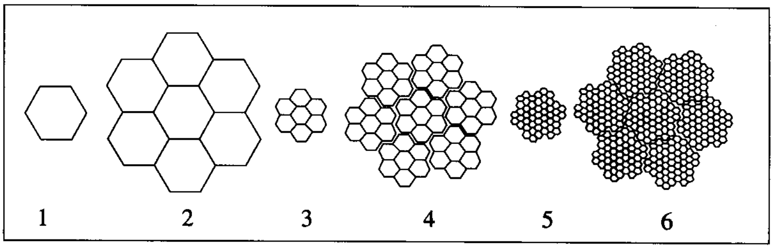

Figure C.5: The first steps in building a hexagonal tile of the plane. The tiles are displayed slightly separated to show how they fit together. The steps are: (1) an initial hexagon $A_0$, (2) seven hexagons, (3) the seven hexagons reduced to form $A_1$, (4) seven copies of $A_1$, (5) the seven copies of $A_1$ reduced to form $A_2$, and (6) seven copies of $A_2$.

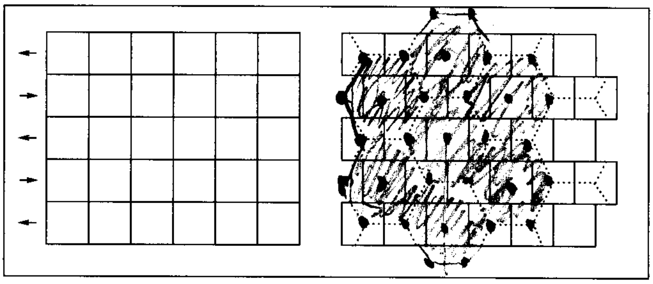

Figure C.6: A way of generating hexagonal tiling on a square lattice.

## C.17 Parallel Processing

While computationally demanding and not well suited to vectorization, the compression methods discussed in this book can fortunately be readily parallelized: they can be implemented in parallel on separate processors sharing the same memory. The most straightforward method simply assigns a separate piece of the image to each processor.

A better approach might incorporate some form of pipelining. For example, a master processor could determine ranges and deal with input/output, while each of several sub-processors compares a range to some subset of the domain pool. Each subprocessor would find the best domain in its subset, and return it to the master processor, which would find the optimal domain and send the next range for matching.

## C.18 Non-contractive IFSs

The image in Figure C.7a shows an expansive IFS which is eventually contractive. One of the transformations, shown by the larger rectangle, is expansive. Therefore, the $W$ map is not

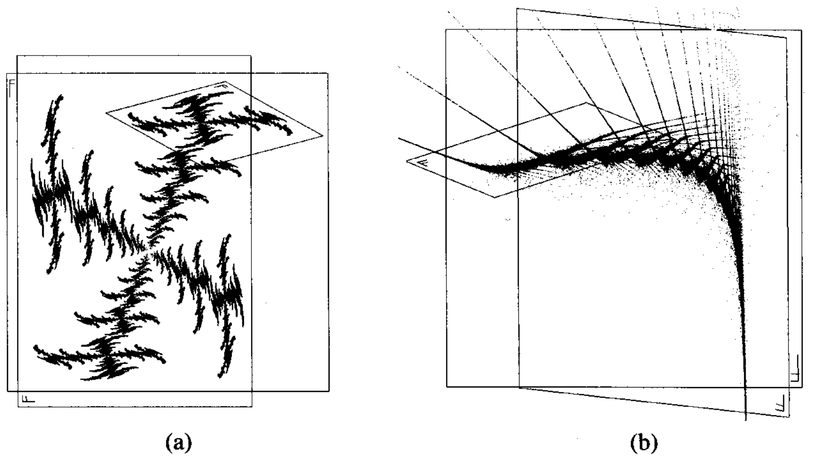

Figure C.7: (a) An expansive IFS which is eventually contractive. (b) An expansive IFS which is not eventually contractive, but still has an interesting attractor. Both IFS's are shown as the image of a square (marked to indicate orientation).

contractive in the Hausdorff metric. However, the second iterate of $W$

$$W^2 = w_1 \circ w_1 \cup w_1 \circ w_2 \cup w_2 \circ w_1 \cup w_2 \circ w_2$$

is contractive.

Figure C.7b shows another expansive IFS. In this case, the attractor is not bounded (so the figure shows only a part of the attractor, and not well). Nevertheless, the attractor exists and has interesting properties.

There should be interesting theory of non-contractive IFS's.

Pg. 37