Fractal Image Compression
Self-Similarity via Locality Sensitive Hashing

Stanford University

Mitchell Douglass

Abstract

In this paper I describe a Haskell implementation of fractal image compression, a lossy image compression technique that leverages self-similarity within an image to produce an encoding. Known for its lengthy encoding time, fractal image encoding implementations require the most cleverness in identifying highly self-similar image regions. In this paper, I describe a simple locality sensitive hash (LSH) used by my implementation to reduce the search time for self-similarity. Though the project is under continued development, I provide details on some preliminary results as well as a discussion of future development and improvements.

I. Introduction

Fractal image compression is a general algorithmic technique for lossy image compression. It was pioneered in the late 1980's and early 1990's, first by Michael Barnsley, founder of the fractal compression company Iterated Systems, but has since been studied by many researchers who have introduced various modifications and improvements to the base algorithm.

Fractal encodings depend upon the discovery of self-similarities within an image, regions within an image which are highly similar with respect to some metric approximating visual distinctness; usually the l_2 norm is used. As such, the main obstacle to the effective fractal compression is the computationally-intensive search for these self-similarities, reflected in the frustratingly slow encoding times of fractal encoding algorithms. The following is a brief overview of the trade-offs of fractal compression.

Pros:
- fast decoding time
- descent compression ratio
- resolution independent

Cons:
- **Slow encoding time**
- susceptible to pathological cases

In what follows, I will touch on three main topics. First, I will discuss the basic method of fractal encoding, giving a general framework for any fractal compression implementation. Next, I will describe a modification to the naive algorithm that I used in my implementation to reduce the search burden of fractal encoding. Thirdly, I will discuss some of the details of my Haskell implementation. I'll then wrap up with a summary of recent results, as well as a discussion of future development and improvements.

II. The Basic Fractal Method

To gain a base understanding of the compression technique, consider the following transformation of the unit square:

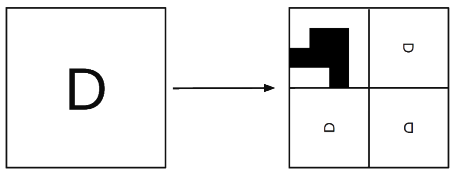

Figure 1: *A basic image transformation*

The transformation is composed of 4 actions: quadrants 1, 3, and 4 of the output are scaled down versions of the entire input domain, while quadrant 2 is the result of a simple shading pattern, applied independent of the input domain. Applying this transformation iteratively to a blank unit square as initial input, the result is a pattern commonly known as a fractal.

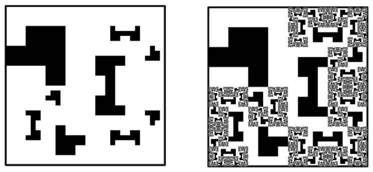
Figure 2: *Iterations 5 (left), and 10 (right)*

To encode an image such as the one produced by 10 iterations of our transformation, it is clear that recording the value of each pixel in the result is unnecessary, and in fact even traditional compression techniques seem like overkill. Instead, one need only store a representation of the simple generating transformation. Notice that it is not necessary to store the number of required iterations, since the fractal is the intrinsic fixed point of this transformation; a decoding algorithm need only iterate the until no change is detected at the desired resolution.

Our first transformation example generalizes to the following model:

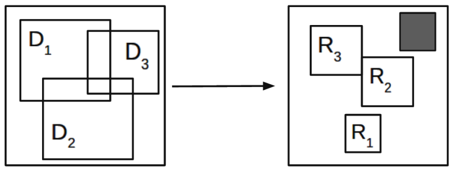

Figure 3: *The general form of an image transformation*

In this model, "image transformations" are represented by a collection of "range transformations". Each range transformation is associated with a particular "range area" of the output; these range areas must partition the output image. A "domain-range transformation" is a range transformation that covers a range area by transforming an associated "domain area", applying scaling, symmetrical transformation (there are 8 on the rectangle), as well as basic brightness and contrast alterations. A "shade-range transformation" is a transformation which simply applies a constant shading to to a range area. In general, range and domain areas need not be rectangular and may undergo transformations beyond basic reflections. However, in this paper this is the case, and indeed this simplified model is quite powerful when applied to encoding general-purpose images.

As mentioned earlier, an encoding of this type is meaningful in the sense that it encodes a fixed point under iterated application. To ensure that such a fixed point exists, the image transformation must be a contraction in the space of all images, meaning the transformation, applied to two distinct input images, must produce output images which are more similar (under the l_2 norm) than these inputs, by a constant factor less than 1. Sparing the details, a theorem of Real Analysis called the Contraction Mapping Theorem states that any contraction mapping produces a unique fixed-point under iteration, independent of choice of initial point (a.k.a. image). In reference to our transformation model, our transformations are contractions when (1) domain areas are larger, in both dimensions, than their corresponding range areas, and (2) the contrast is effectively reduced under domain-range transformations. Shade-range transformations are also contractions. This gives a simple criteria for valid image transformations.

The existence and uniqueness provided by the Contraction Mapping Theorem guarantee us that any image transformation which is a contraction is a valid encoding of its fixed point. Yet the question remains: can a general image be well-represented by the fixed point of an image transformation of this form, and if so, how can these transformations be constructed? The answer to the first part is straight forward: yes. If we want an image transformation which encodes image A as a fixed point, we must find a transformation which alters A as little as possible; that is we must find domain areas which, under transformation, are almost identical to their corresponding range areas. For those ranges that are not-well approximated by larger domains, or that are best approximated by a constant shade, we may apply shade-range transformations. As it turns out, general images contain an abundance of of these pairs (e.g. Figure 4). The main challenge of fractal image compression is finding them.

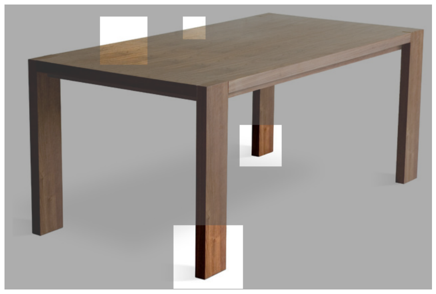

Figure 4: *Self similarity in a coffee table*

Fractal image encoding involves finding a sufficient number of similar domain area-range area pairs such that the range areas partition the image. Compression is achieved for each matched range area covering more than just a few rows and columns, since storing a domain-range or shade-range transformation requires the equivalent disk space of a small number of pixels. The decoding process involves iterating the stored image transformation to an arbitrary base image, stopping when no changes occur at the desired resolution. An important question, impacting decoding time, is whether a fixed point will be achieved quickly. The answer is yes, provided that all domain-range transformations uniformly involve a scaling of a factor $k < 1$. If $k \leq 1/2$, as is the case in my implementation, then a fixed point is achieved within $\log_2 d$ iterations, where d is the largest dimension of the output image.

Following is a pseudo-code for a fractal encoding algorithm. Here, `regionlists` is a list of lists of regions, where each element region list contains image regions of the same size, and where the size of regions in each list is strictly decreasing in the high-level list. The function `find_best_trans` must iterate through all larger image regions, finding the domain which is a best l_2 approximation under some domain-range transformation, or else produce a shade-range transformation if this is the best choice.

```
-- candidate regions for a domain-
-- range transformation.
larger_regions = []

-- the result of the computation
image_transform = []

foreach regionlist in regionlists:
  -- transformations that are good
  -- enough to be accepted in into
  -- the image transform this level
  good_lvl_trans = []

  -- collect possible transformations
  foreach region in regionlist:
    region.best = find_best_trans(
        region, larger_regions)
    if is_good_fit(region.best):
      good_lvl_trans.add(region.best)

  -- find maximal set of good
  -- transforms whose ranges do not
  -- intersect
  new_trans = best_nonintersecting(
      good_lvl_trans, transforms,
      image_transform)
  image_transforms.add(new_trans)

  larger_regions.add_all(regions)
```

Figure 5: *A pseudo-code for a fractal encoding algorithm*

III. Locality Sensitive Hashing

As mentioned earlier, the most computationally intensive part of the fractal encoding algorithm is the `find_best_trans` function in the pseudo-code of figure 5. This is due to the fact that most of the self-similarity within images manifests itself on a small scale; that is, the median appropriate range area is very small. As such, the algorithm outlined above must search a massive set of `larger_regions` for each range area, and a comparison with any particular larger area involves computing an l_2 distance on high dimensional data. This computation becomes very expensive very quickly. On large scales, a brute force approach is impractical.

Implementations of fractal image compression must employ some method of minimizing the search for acceptable transformations. Some methods involve classifying ranges by a short list of measurable properties, such as luminance gain vs. average luminance, maximum pixel variation, and limiting the search for good domain areas to only those in the same or similar category. Other solutions involve much more complicated feature detection techniques that are beyond the scope of this paper. I have chosen to implement a simple locality-sensitive hash algorithm to map image regions into low-dimensional space, and I use proximity of regions hashes in low dimension to identify likely candidate domain areas.

A locality-sensitive hash (LSH) is a function that maps high-dimensional data (in our case regions of an image) into low-dimensional space (in this case $\mathbb{R}^4$), such that "similar" high-dimensional input data are mapped to "close" low-dimensional points.

Here is how the LSH works in my implementation: Let X be an image region, let q_1, q_2, q_3, and q_4 be functions from rectangular image regions to rectangular image regions corresponding to the four standard, equally-sized quadrants of their inputs (e.g. $q_1(X)$ represents the top-right quadrant of X, $q_2(X)$ the top-left, etc). Let avg be a function giving the average luminance of an image region. Our LHS, call it lsh, is defined inductively as follows:

$$lsh(X) = proj(X) + \frac{1}{8} \sum_{1 \le i \le 4} lsh(q_i(X))$$

where $proj(X)$ is a vector in $\mathbb{R}^4$ satisfying

$$(proj(X))_i = \frac{avg(q_i(X)) - \mu}{\sigma}$$

where μ and σ are the mean and population standard deviation of $avg(q_1(X)), \ldots, avg(q_4(X))$. Image regions that are single pixels have the zero vector of $\mathbb{R}^4$ as lsh value.

As you can see, this LSH attempts to capture the high-level features of an image region with the $proj(X)$ function. The features of each component quadrant are captured by adding a scaled-down version of their lsh value. Due to properties of normalization, the $proj$ function produces vectors of magnitude 2, where the important point is that $proj(X)$ has constant magnitude. The factor of $1/8$ ensures that the size of the vector produced by the sum term of lsh is no greater than $1 = 1/2 \cdot 2$. Therefore, lsh obeys a sort of limiting property: if X is an image region of size $2^n x 2^n$ and v is lsh vector which results from scaling down X to a region of size $2^m x 2^m$ by averaging pixels in blocks of size $2^{n-m} x 2^{n-m}$, then $||lhs(X) - v|| < 2^{(1-m)} - 2^{(1-n)}$, which is quite small for values of n, m larger than 3 or 4. This indicates that regions with identical features (i.e. when a domain is itself scaled down) have lhs values that are very close, especially at reasonably-high dimension. Some other nice properties of this LSH are

- Images are reduced to the same low-dimension, $\mathbb{R}^4$, appropriate for search in kd-trees.
- Images with low l_2 distance are close under this LSH. However, due to the arbitrary combination of quadrant lhs values, images may have close lsh values when they are not similar.
- Due to properties of the normalization of $proj$, the lsh value of a region is invariant under brightness and contrast changes. This property is useful since good transformations requiring changes to brightness and contrast can be identified by a single hash.
- Due to properties of symmetries of the rectangle, the lsh vectors of regions which vary only by a symmetry of the rectangle are themselves only permutations of each other. Thus, only a single hash is required to identify candidate symmetric transformations.
- The lsh vectors can be computed dynamically; i.e. by a "table-filling" technique.

Computing averages of smaller regions before larger regions, the *lsh* value of all regions of size 2^k may be computed in time linear in the number of pixels in the image.

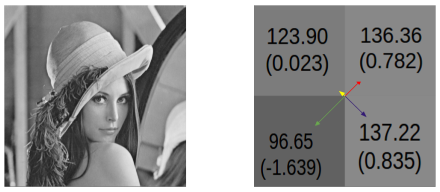

Figure 6: *The Lena image, showing proj(lena), normalized values in parentheses*

The modification to the general fractal encoding algorithm of Figure 5 is straightforward. When a range area is considered for a transformation, only candidate domain areas with hash values sufficiently close to the hash value of the range need be considered, and they can be considered in order of nearest hash value. As mentioned in the previous list, a domain area with a close hash vector need not necessarily be a good approximation of the query range area, so an exhaustive search of near neighbors is still necessary to verify the quality of a transformation.

In lieu of a list of larger domains, the algorithm employs a kd-tree to store the hash values of each image area as it is considered. Just as in the pseudo-code, the kd-tree in the modified algorithm contains domain area hashes of image areas that are strictly larger than the range area under consideration, so near points truly represent valid candidate domain areas.

IV. Haskell Implementation

I now give a tour of this basic infrastructure, as implemented in Haskell. Here is a list of some of the technologies I used in my project:

- **repa package:** high-performance arrays for unboxed numerical data, used for storing and transforming image pixel luminance data.
- **unm-hip package:** image processing library, used for intermediate image storage, image utility functions, as well as image IO.
- **kdt package:** kd-tree library for storing image region approximations, and performing nearest neighbor searches.
- **pqueue package:** priority queues to choose the best-fit non-intersecting transformations for each region size.
- **fingertree package:** Implements interval storage and search, used for quickly identifying overlapping range regions.

We have already described the general type of an image transformation, which I represent in the following way in Haskell:

```haskell
type ImageTransform
       = [RangeTransform]

data RangeTransform =
  DomainRangeTransform {
      domainArea   :: ImageArea,
      rangeArea    :: ImageArea,
      symmetry     :: D8,
      invertLum    :: Bool,
      contrastFac  :: Double,
      brightFac    :: Double
  } |
  ShadeRangeTransform {
      rangeArea  :: ImageArea,
      shade      :: Double
  } deriving (Eq, Show)

data ImageArea = ImageArea {
    areaYOffset :: Rational,
    areaXOffset :: Rational,
    areaHeight  :: Rational,
    areaWidth   :: Rational
} deriving (Eq, Show)

-- 8 symmetries of the rect.
data D8 = ID | R1 | R2 | R3 |
          S  | RS | SR | R2S
          deriving (Eq, Show)
```

The following Haskell data data types represents the dynamically built, low-dimensional data structure that allows image regions to be embedded into $\mathbb{R}^4$.

```
type LowDimPnt = V.Vector Double

data AreaInfo =
  AreaInfo {
    -- a 4-dim. approximation
    approx :: LowDimPnt,
    proj   :: LowDimPnt,
    avgLum :: Double,
    maxLum :: Double,
    minLum :: Double,
    pxArea :: PixelArea
  } |
  NoArea

type ImageArray =
  R.Array U DIM2 Double

type ImageAreaTable =
  Array (Int, Int) AreaInfo

type ImageAreaTableList =
  [ImageAreaTable]
```

Figure 7: *Pseudo-code for specialized data structures.*

As seen here, image pixel data is represented by efficient arrays from the repa package, while the low-dimensional approximations are represented by vectors from the standard vector package.

The `ImageAreaTable` and `ImageAreaTaleList` is a data structure utilized to store the `AreaInfo` structures, which hold meta-data about image regions. As of the current state of the implementation, only blocks of size 2^k are stored in these tables. However, all possible blocks of size 2^k are dynamically computed in these tables, not solely those which are "well" aligned. The result is that the the implementation currently considers a very large number, on the hundreds of thousands, of image regions as candidate domain regions for small range regions.

The following data structure is utilized during the encoding process to track the "state" of affairs (although the data structure is not mutable, it is treated as a sort of accumulator).

```
type LvlPQueue =
        PQ.MinPQueue Double
                 RangeTransform

data ImageEncodingState =
  ImageEncodingState {
    imgWidth         :: Int,
    imgHeight        :: Int,
    imgArray
      :: ImageArray,
    areaInfoTblLst
      :: ImageAreaTableList,
    depth            :: Int,
    rangeTransforms
      :: [RangeTransform],
    xIntrMap
      :: IntrMap.IntervalMap
           Int (Int, Int),
    yIntrMap
      :: IntrMap.IntervalMap
           Int (Int, Int),
    kdTree
      :: KdMap Double
               LowDimPnt
               AreaInfo,
    pixelsCovered    :: Int
}
```

To start, the `depth` field determines the number of "levels" of computation that have been performed, where each level represents evaluating ranges of increasingly smaller size. The `imgHeight` and `imgWidth` fields provide a context for the size image being encoded. `PixelsCovered` tracks the number of pixels which are contained in a transformation in `rangeTransforms`, allowing the algorithm to short circuit when full coverage has been achieved.

The purpose of the `IntervalMaps` is to store a ledger of which range regions have already been covered by transformations. Using these interval maps for interval search, determining a region that is already covered becomes an efficient process.

The kd-tree is used for storing approximations of regions that are larger than any previously considered regions. The `AreaInfoTblLst` corresponds to the `ImageAreaTableList` data type, and stores the computed region information.

V. Results

While this implementation is still under development, there have already been promising results to suggest that the technique worth further exploration and improvement. In what follows, claimed compression ratios are based on conservative pen-and-paper calculation, taking into consideration the number of range-transformations required to encode an image with respect to the pixel dimension of the image.

The first success of this implementation is to compress images of simple geometry, for instance the circle. Early attempts at the circle were abysmal, while quite entertaining.

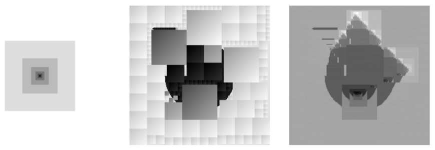

Figure 8: *Early attempts at the circle*

After some streamlining, I was able to achieve an implementation which did infact encode the circle (and other simple geometry) quite well, with a compression ratio of approx. 90:1). However, these algorithms were not very good at encoding general images.

Figure 9: *The circle on the left, an attempt to decode lena on the right*

After further streamlining, specifically incorporating a sufficiently broad l_2 search of nearest neighbours in the kd-tree, I was able to achieve an algorithm which produces interesting results for both simple geometry and general images. In the case of the circle, I was able to achieve 42:1 image compression. Unfortunately, the decompression of the circle produces artefacts in the form of blurred edges and a gray hue which is not present in the input image. In the case of lena, I was able to achieve only a 3:2 compression ratio. However, the quality of the decompressed lena is quite good, with few distracting artefacts.

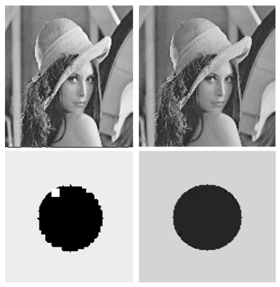

Figure 10: *From top-left to bottom-left clockwise: lena original, lena full decompressed at 3:2, circle first iteration of decompression, circle full decompression at 42:1*

In terms of efficiency, this implementation has not yet been sufficiently optimized for performance. Encoding the 128x128 circle image above required 15.12 seconds, while encoding the 128x128 lena image required 47.0 seconds. These deficiencies likely are the result of suboptimal memory use in the functional setting of Haskell. There are certain algorithmic improvements to be made as well.

VI. Future Development and Improvement

The primary goals for development on this implementation in the future are

- Use of criterion and other profiling / benchmarking libraries to improve this baseline resource use, and performance, of the existing algorithm.

- Begin writing a robust test suite for the library to track and maintain well-beardedness of the various components of the algorithm.

- Begin writing a utilities library to aid in agile development and testing.

- Provide a robust disk serialization of image encoding such that image transformations may be written and read from disk correctly, and efficiently.

- Leverage the parallel / mutable state capabilities of the Repa, and other package to improve performance.

- Currently only considers blocks of size 2^k, which improves encoding efficiency, but is a lost opportunity in terms of compression. Incorporate blocks of arbitrary size and non uniform aspect ratio.

- Currently blindly partitions regions only into 4 equal quadrants: modify partitioning scheme to split based on feature detection.

References

[1] Curtis, S.E. and C.E. Martin *Functional fractal image compression*, Proceedings of the 6th Symposium on Trends in Funcional Programming, TFP 2005 pp: 383-398, 2005.

[2] Fisher, Y. *Fractal Image Compression*, SIGGRAPH'92 course notes, 1992.

[3] Saupe, D. and Hamzaoui, R. and Hartenstein, H. *Fractal Image Compression - An Introductory Overview*, Albert-Ludwigs University at Freiburg, TFP 2005 pp: 383-398, 1997.