

# **Functional Fractal Image Compression**

S. A. Curtis and C. E. Martin

Department of Computing, Oxford Brookes University, UK.

## **Abstract**

This paper uses functional programming techniques to model fractal image compression. The language used is Haskell.

## 1 INTRODUCTION

Fractal image compression is a lossy compression technique developed by Barnsley [BH86] and Jacquin [Ja89], in which an image is compressed by storing it as a transformation. As functions are fundamentally involved in the process, the purpose of this work is to use functional programming techniques to model fractal image compression and decompression, and to unify a number of different fractal image compression techniques. The implementation in this paper uses the language Haskell [PJH99].

## 2 FRACTAL IMAGE COMPRESSION

The transformation function used in fractal image compression is chosen in such a way that its unique fixpoint is a close approximation of the input image. Compression occurs because storing the details of the image transform (also known as *encoding*) takes up much less space than the original image. Decompression (or *decoding*) involves applying the transform repeatedly to an arbitrary starting image, to arrive at a (fractal) image that is either the original, or one very similar to it.

To illustrate the concept of representing an image as a transform, consider one of the simplest fractals: the Sierpinski triangle (see Figure 1). This image is *self-similar*, in the sense that it consists of three smaller copies of itself, positioned at lower left, lower right and top middle.

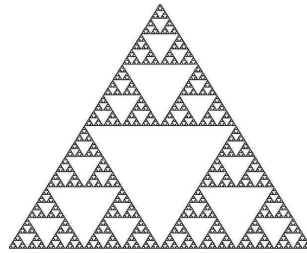


FIGURE 1. The Sierpinski triangle

is the transformation that maps the whole image onto the shrunken copy of itself in the top middle portion of the image, and  $\omega_2$  and  $\omega_3$  respectively map the whole image onto the copies at lower left and lower right, then the Sierpinski triangle is a fixed point of the transformation

$$\omega = \omega_1 \cup \omega_2 \cup \omega_3 \quad (1)$$

because the effect of applying  $\omega$  to the Sierpinski triangle is to leave it unchanged.

According to the *Contractive Mapping Fixed-Point Theorem* [Fi95], applied to a suitable space of images, the Sierpinski triangle is the unique fixpoint of  $\omega$ : the transformation  $\omega$  is *contractive* because all of its transformations  $\omega_i$  are contractive, which means that when applied to any two points it brings them closer together. The theorem states that if  $\omega$  is contractive there is a unique fixpoint (*attractor*) limit of the sequence

$$x, \omega x, \omega^2 x, \dots \quad (2)$$

where  $x$  is any starting image, and  $\omega^i$  denotes the composition of  $\omega$  with itself  $i$  times. So the Sierpinski triangle can be generated to any required degree of accuracy by repeatedly applying  $\omega$  to any starting image, as illustrated in Figure 2, and in this sense, the transformation  $\omega$  represents the Sierpinski triangle. Hudak also illustrates this in [Hu00] by using a Haskell implementation of  $\omega$  to draw the Sierpinski triangle.

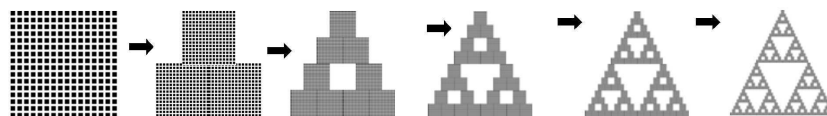
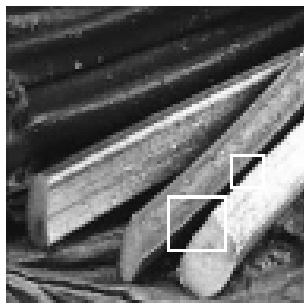


FIGURE 2. Iterations of the transformation

The representation of an image by a union of contractive transforms, like (1), is known as an *Iterated Function System (IFS)*. Although such systems are useful for compressing fractal images like the Sierpinski triangle, most images do not have such obvious self-similarity properties. However many images (particularly photographs) do have areas that are almost self-similar, as illustrated in Figure 3. The similarity of parts of an image provides the inspiration for the concept of a *Partitioned Iterated Function System (PIFS)*, which is similar to an IFS, except that the contractive transforms have restricted domains. In practice, this means that an image to be compressed is partitioned into small regions (*ranges*), each of which is matched as closely as possible to one of several larger regions (*domains*) of the image. The best match for each range yields a contractive transformation that maps part of the original image onto that range block. As with an IFS, the union of the contractive transforms encodes the original image.

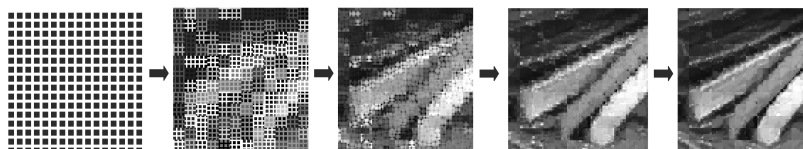
To illustrate, an easy way to partition an image for a PIFS is to divide



**FIGURE 3. Self-similarity between large and small portions of an image**

the image into a grid of small squares for the range regions (blocks), and use larger square regions within the image for the domain blocks. Each range block is then compared with all the domain blocks to find the one that matches it most closely. As well as being reduced in size, each domain block can be rotated or reflected in order to best match the range block. It can also have its contrast and brightness adjusted. For example, Figure 3 illustrates a small square range block, 8-by-8 pixels, along with a larger 16-by-16 pixels domain block. This domain block, in the unrotated and unreflected orientation, is a good match for the range block, needing only a very small contrast and brightness adjustment, as the colours already match closely.

The reproduction of the original image from the PIFS transform is illustrated in Figure 4. The decoding process involves repeatedly applying the transform until it converges to an image, which closely approximates the original:



**FIGURE 4. Decoding**

Note: the quality of the fractal image compression in these example pictures is poor, because for small images, there are fewer domain blocks to choose from and thus the picture quality is poorer. For larger images the quality of the compression is much better, particularly when more sophisticated compression is used rather than the basic block encoding scheme

outlined here.

### 3 IMAGES

#### 3.1 Image Types

There are several possible ways to model images. For example in [Co03], images are represented as functions on the real plane, using a datatype such as

```
type ImageFn a = Point2D -> a
type Point2D = (Float, Float)
```

This representation is polymorphic, so for example, the type `ImageFn Colour` can be used to represent colour images, and the type `ImageFn Greyscale` can be used for greyscale images, for some suitable types `Colour` and `Greyscale`.

This technique of modelling images as functions is elegant, but for fractal image compression the individual pixels must be examined, and so the domain of the function should be finite and discrete. In addition, efficiency is an issue during compression, so we prefer to represent images as arrays, which perform the same role as such functions, but have faster access to pixel data:

```
type Image      = Array Point Greyscale
type Point      = (Int, Int)
type Greyscale  = Int
type Pixels     = [Greyscale]
```

We represent shades of grey by a linear scale from 0 (black) to 255 (white), hence the choice of type for `Greyscale`. Note that in this paper we consider only greyscale images: colour images are easily dealt with by splitting a picture into its hue, saturation and lightness components, and compressing each component using the same technique as for greyscale images. [Fi95]

#### 3.2 Image Comparison

It is necessary to have a measure of “distance” between images, for comparing how closely two blocks match, and for using the Contractive Mapping Fixed-Point Theorem.

There are many methods available [Fi95] for measuring how similar two images (or portions of an image) are; we use the root mean square

(rms) metric. If  $rs = [r_1, r_2, \dots, r_n]$  and  $ds = [d_1, d_2, \dots, d_n]$  are lists of greyscale pixels (i.e. integers) then the rms distance between them is defined by

$$\text{basic\_rms } rs \ ds = \sqrt{\sum (d_i - r_i)^2}$$

where  $\sum$  is an abbreviation for  $\sum_{i=1}^n$ . If  $ds$  represents a domain block which is to have contrast and brightness adjustments  $c$  and  $b$  respectively, then the distance between the adjusted block and the range block  $rs$  is

$$\text{adjusted\_rms } rs \ ds \ c \ b = \sqrt{\sum (c * d_i + b - r_i)^2}$$

The values of  $c$  and  $b$  that minimise this value will provide the best match between the blocks  $rs$  and  $ds$ . This minimum occurs when the partial derivatives with respect to  $c$  and  $b$  are zero, which is when

$$c = \frac{n \sum d_i r_i - \sum d_i \sum r_i}{n \sum d_i^2 - (\sum d_i)^2}$$

$$b = \frac{1}{n} (\sum r_i - c \sum d_i)$$

unless the denominator of  $c$  is zero, in which case  $c = 0$  too. The rms distance between the pixels is then  $\sqrt{r}$ , where

$$r = \frac{1}{n} [\sum r_i^2 + c(c \sum d_i^2 - 2 \sum d_i r_i) + 2b \sum d_i + b(n b - 2 \sum r_i)]$$

So these calculations can be used to find the domain block that best matches each range block, as well as the contrast and brightness adjustments for that block.

If each contrast value  $c$  satisfies the constraint  $c < 1$ , then the corresponding transformation is guaranteed to be contractive, and so the sequence of approximations towards the final image will converge. In practice, experiments have shown that the constraint  $c < 1.2$  is also sufficient.

### 3.3 Blocks

Rectangular image regions will be used for both range and domain blocks. Since the compression is very computationally intensive, there is an inevitable trade off between elegance and efficiency. The first concession to efficiency is in the declaration of the datatype `Block`, which stores information about each block, as well as its pixel data.

```
type BlockInfo = (Int, Int, Int, Int, Classification)
type Block = (BlockInfo, Pixels)
```

Information stored about the block includes its width and height, its position in the image, and also some further information to classify the block during more sophisticated versions of the encoding, which are described briefly in Section 7.2.

### 3.4 Transforms

The natural type for a transform is obviously a function, but unfortunately this leads to implementation problems because transforms must be saved in a file to encode the compressed image.

The required block transforms are easily described with a few parameters, so we use the type:

```
type Orientation = Int
type Transform = (Float, Float, Orientation, Float)
```

Respectively, these are the brightness adjustment, the contrast adjustment (a scaling factor for the grey level), an integer describing one of the 8 possible affine transformations for rectangular blocks, and the rms error. The rms error is needed for determining the closest block match, but is not saved in the encoding.

The types needed for a PIFS are as follows:

```
type PIFSTransform = (BlockInfo, Transform, BlockInfo)
type Pifs = (Int, Int, [PIFSTransform])
```

The type `PIFSTransform` stores the information about each range block, together with the closest matching domain block and corresponding transform. The type `Pifs` also stores the width and height of the original image, because during decompression it will be necessary to know what size the decoded image should be.

## 4 ENCODING

This section outlines the encoding process. The basic encoding function first gets details of all the domain blocks including all rotations and reflections of each domain block (for efficiency purposes), and then uses that information to find a suitable PIFS:

```
encode :: Image -> Pifs
encode im = (imageWidth im, imageHeight im,
            findPifs im (getAllDomBlocks im))
```

```
getAllDomBlocks :: Image -> [(Orientation,BlockInfo)]
```

In the simple block encoding scheme, we chose to use square range blocks of size 8 by 8 pixels to partition the image (for simplicity, images had dimensions divisible by 8), and domain blocks of size 16 by 16 pixels. Using all possible domain blocks results in a lengthy encoding, so instead, a representative sample can be used: we chose to use domain blocks laid out in a grid fashion, spaced 4 pixels apart.

The function `findPifs` takes an input image and list of domain blocks and finds the best transform for each range block:

```
findPifs :: Image -> [(Orientation,BlockInfo)]
           -> [PIFSTransform]
findPifs im domBlocks
  = map (findTransform im domBlocks)
        (getAllRangeBlocks im)
```

```
getAllRangeBlocks :: Image -> [BlockInfo]
```

The best transform for a given range block is calculated by `findTransform`, which goes through all possible domain blocks to find the closest match:

```
findTransform :: Image -> [(Orientation,BlockInfo)]
              -> BlockInfo -> PIFSTransform
findTransform im doms b
  = bestOf (map (blockMatch ran im) doms)
    where pixels = getBlock b im
          s      = sum pixels
          ssq    = sum (map square pixels)
          ran    = (b,pixels,s,ssq)
```

Here the function `getBlock` simply retrieves the required pixels from the image, and the function `blockMatch` does the rms calculations between the range block and the domain block under consideration, as detailed in Section 3.2, after orientating and shrinking the domain block. The function `bestOf` simply selects the transform with the smallest rms error. However there is a way to make this more efficient: if, on searching through all possible transformations of domain blocks, a transformation is found that is “good enough” in some way, for example its rms error is sufficiently within a certain tolerance and its contrast scaling factor is not too high

```
goodEnough :: PIFSTransform -> Bool
```

```

goodEnough (rangeBlock,
            (brightness, contrast, rotreflect, error),
            domainBlock)
            = (error < 640) && (contrast < 1.2)

```

then instead of using the function `bestOf`, we can define and use in its place a function `closeEnough`, using `goodEnough`, which will return the first transform found to be good enough, or if no such block exists, the best matching of the available transforms.

A sample output of the basic block encoding can be seen in Figure 4, from the original image displayed in Figure 3.

## 5 DECODING

The encoding of an image as the fixpoint of a PIFS transform suggests a decoding of the form

```

decode :: Pifs -> Image
decode (width,height,ts)
    = until converged (applyTransforms ts) startImage

```

However, detecting when the decoding has finished is a little awkward: the current image has to be compared with the image produced at the previous step, and such an implementation slows the iteration down and is fiddly to code. In practice, this decoding function, which produces the steps of the iteration of the image transform (2), is useful:

```

decodeN :: Int -> Pifs -> Image
decodeN n (width,height,ts)
    = (iterate (applyTransforms ts) startImage) !! n
      where startImage = blankImage width height 128

```

where the function `blankImage` simply creates an image of the desired width and height, with all pixels the given shade of grey.

The function `applyTransforms` is defined

```

applyTransforms
    :: [PIFSTransform] -> Image -> Image
applyTransforms ts im
    = combineRanges im (map (applyTransform im) ts)

```

which involves applying each of the transforms in the list `ts` to the image, in a straightforward fashion. The function `applyTransform` fetches the

pixel data for the required domain block, then transforms the data to produce a range block. Combining the resulting range blocks is done with the `combineRanges` function, which updates the values of the pixel array for each range block.

```
applyTransform :: Image -> PIFSTransform -> Block
combineRanges :: Image -> [Block] -> Image
```

## 6 IO

For the input of pixel data from images, we chose to use the PGM (Portable GreyMap) image format, which stores data simply in text files. Conversion functions to and from the PGM format were implemented: Additionally, after encoding the images, the resulting PIFS data had to be stored. Since the aim of this work was to model fractal image compression rather than to compress images, the PIFS data was stored simply in text files. Given suitable input and output conversion functions for the PGM files and text files storing the image encoding, the encoding and decoding functions were implemented as

```
encodeFile :: String -> String
encodeFile = toEncoding . encode . fromPGMtoImage

decodeFile :: Int -> String -> String
decodeFile n = imageToPGM . decodeN n . fromEncoding
```

which were used with a straightforward IO wrapper function for file input and output.

## 7 OPTIMIZATION AND IMAGE QUALITY

### 7.1 Range Selection

A weakness of the grid square range partitioning scheme is that it does not take into account variability of the amount of detail in different image regions. A better compression quality might therefore be obtained by varying the choice of range blocks. We model two of the many partitioning methods described in [Fi95].

### 7.1.1 Quadtree Partitioning

Typically more detailed areas of the original image need smaller size range blocks, in order to be matched accurately with a domain block. Because of this, quadtree partitioning uses square range blocks that can vary in size. If a large range block does not have a suitably accurate match with some domain block, using the standard block encoding as described in Section 4, then it is divided into four quadrants and the process is repeated. Two integer parameters specify the maximum and minimum possible sizes for range blocks.

The implementation requires an adjustment. Using a function

```
getQuadtreeBlocks :: Image -> [BlockInfo]
```

which returns a list of maximum size range blocks, `findPifs'` is adjusted to use a recursive function `findMatches`:

```
findPifs' :: Image -> [(Orientation,BlockInfo)]
           -> [PIFSTransform]
findPifs' im d =
    concat (map (findMatches im d)
              (getQuadtreeBlocks im))
```

The function `findMatches` operates on a single block to return the list containing a single transform, if a good enough match can be found for that block. Otherwise, it returns a list of transforms found for the range blocks it gets divided into:

```
findMatches im d b
  | goodEnough t = [t]
  | tooSmall b   = [t]
  | otherwise    = splitQ
  where t        = findTransform' im d b
        bs       = quarter b im
        splitQ   = concat (map (findMatches im d) bs)
```

Here the predicate `tooSmall` tests whether the size of the block is within the allowed range, `quarter` splits a block into a list of its four quadrant blocks, and `findTransform'` is very similar to the `findTransform` function used before, except that it first filters the list of domain blocks to remove those that are the wrong size:

```
findTransform' :: Image -> [(Orientation,BlockInfo)]
               -> BlockInfo -> PIFSTransform
```

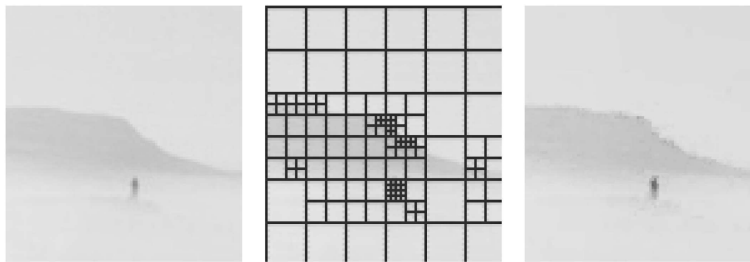
```

findTransform' im doms b
  = closeEnough (map (blockMatch ran im)
                   (filter (suitable b) doms))
  where pixels = getBlock b im
        s      = sum pixels
        ssq    = sum (map square pixels)
        ran    = (b,pixels,s,ssq)

suitable (_,_,ranWidth,ranHeight,_)
  (_, (_,_, domWidth, domHeight, _))
  = (ranHeight*2 == domWidth)

```

A sample result of using quadtree partitioning is illustrated in Figure 5. Notice how the blocks are much smaller in the more detailed areas of the image. Further details of the quadtrees implementation are omitted for



**FIGURE 5. Quadtree partitioning: original (left), partitions (centre), decoded image (right)**

reasons of space.

### 7.1.2 HV Partitioning

The HV partitioning scheme is similar to the quadtree scheme, but more flexible: range blocks without a suitably accurate match are subdivided into two blocks, either horizontally or vertically. This division is made in a smart way, choosing where to partition a block according to differences in neighbouring pixels, aiming to produce subdivided blocks that are as different as possible.

Like quadtree partitioning, HV partitioning repeats recursively until a given error tolerance is reached, but the HV scheme is more flexible because the position of the partition is variable. Apart from the subdivision of range blocks, the implementation is similar to that for quadtrees, except

that instead of starting with a list of maximum size domain blocks, it starts with the whole image:

```

findPifs'' :: Image -> [(Orientation,BlockInfo)]
              -> [PIFSTransform]
findPifs'' im d = findMatches' im d (0,0,w,h,0)
                where (w,h) = getWidthHeight im

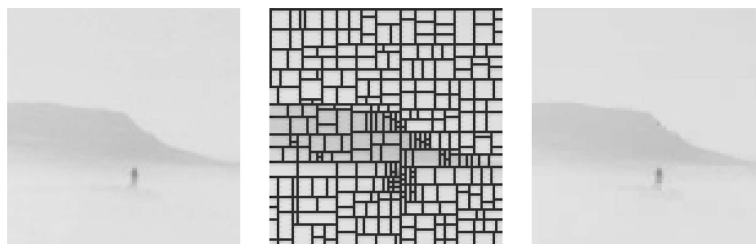
findMatches' im d b
  | tooBig b      = splitHV
  | goodEnough t = [t]
  | tooSmall b   = [t]
  | otherwise    = splitHV
  where
    bs      = splitBlockHV b im
    t       = findTransform'' im d b
    splitHV = concat (map (findMatches' im d) bs)

```

Here if the examined block is too big to be a domain block, it is split either horizontally or vertically, according to the split which makes the biggest difference between pixels either side of the split (giving slightly more weight to splits nearer the middle of the block).

The function `findTransform''` is the same as before, except that a different suitability predicate is needed, to check that the dimensions of a domain block are multiples of the dimensions of the range block it is being compared with.

An example of HV partitioning is given in Figure 6.



**FIGURE 6.** HV partitioning: original (left), partitions (centre), decoded image (right)

Further details of the HV partitioning implementation are omitted for reasons of space.

## 7.2 Block Classification

The encoding process will be lengthy irrespective of what partitioning scheme is used, because each range must be compared with a large number of domains. One way to speed this up is to adopt a *classification scheme* [JBF90, Ja89] whereby all domain and range blocks are assigned a classification based on pixel values. Then each range need only be compared with those domain blocks that have the same classification, significantly reducing the amount of computation involved.

One possible scheme is to divide blocks into four quadrants and classify blocks based on which quadrant is the lightest in colour, second lightest in colour, and darkest. This produces twenty four classes because that is the number of permutations of four values. The changes to the implementation involve the assignment of a classification number when gathering information for domain blocks (the last number in tuples of type `BlockInfo`), and the filtration of blocks in the function `findTransform` to only consider domain blocks that have the same classification as the range block currently being matched. A number of more sophisticated schemes are described in [Fi95].

## 8 DISCUSSION

This paper has shown how functions can be used to model the fractal image compression technique in which an image is represented as a transform from which an approximation to the original image can be generated as a fractal. The implementations of three partitioning schemes have been outlined and briefly discussed: simple grid squares, quadtrees and HV partitioning. Some benefits of using a classification scheme to improve efficiency have also been described.

The original motivation for this functional implementation of fractal image compression came from seeing how beautifully affine transformations have been expressed in the Haskell library Pan [Co03], where images are modelled as functions from the real plane, as described in Section 3.1. Since affine transformations are the building blocks of the fractal image compression technique, it seemed natural to attempt a similar implementation of it. In practice, efficiency considerations have led to a slightly different model, where images have been modelled as arrays rather than functions, and storage of encoded images has used numbers and lists, rather than the actual functions which encode images. However, this approach has still produced some elegant functions which have made effective use

of the higher order functions of Haskell. In the encoding process for example, the `findTransform` function, which finds the best transform for a given range block, uses a `filter` to select only the domain blocks that match the size and classification of a given range block, followed by a `map` to calculate the rms errors of each of the selected blocks.

In the decoding process, the `applyTransform` function, which converts a transform back into a range block, is mapped over the list of transformations and the resulting blocks are folded together to produce an image by the function `combineRanges`. The same decoding functions work for all encoding methods. So an economy of code is achieved that would not be possible without the use of the `map` and `fold` functions. The main `decodeN` function also has a very clear and succinct expression via the higher order function `iterate`. We feel that this modelling in a functional style helps to illuminate the concepts of fractal image compression in a clear way.

Whilst this implementation utilises Haskell's higher order functions, it makes very little use of polymorphism. The various encoding methods are very similar, and would seem to lend themselves to a more generic programming style. In practice the elegance of expressing the encoding with a single function (with different parameters according to encoding method) has been sacrificed for clarity of code.

It would also be interesting to experiment with some of the more recent developments in graphical libraries for Haskell, to see whether they offer any advantages in expression or performance.

## REFERENCES

- [BH86] M. F. Barnsley and L. P. Hurd. *Fractal Image Compression* AK Peters Ltd, Wellesley, Ma, 1992
- [Co03] Conal Elliot. Functional Images *The Fun of Programming* Palgrave, 2003.
- [Fi95] Yuval Fisher. Editor *Fractal Image Compression. Theory and Application* Springer, 1995.
- [Hu00] Paul Hudak *The Haskell School of Expression* Cambridge University Press, 2000.
- [PJH99] S. Peyton Jones . Editor *Haskell 98 Language and Libraries: The Revised Report* . Cambridge University Press, 2003.
- [JBF90] E. W. Jacobs, R.D. Bos and Y. Fisher. Fractal-based image compression ii. Technical Report 1362, Naval Ocean Systems Center, San Diego, CA, June 1990.
- [Ja89] A. Jacquin *A Fractal Theory of iterated Markov Operators with Applications to Digital Image Coding* . PhD thesis, Georgia Institute of Technology, August 1989.
- [Re03] Alistair Reid. HGL Graphics Library  
<http://www.haskell.org/graphics/>