

EDUCATIONAL PEARL

Fractal image compression

C. E. MARTIN

Oxford Brookes University, Oxford OX33 1HX, UK
(e-mail: cemartin@brookes.ac.uk)

S. A. CURTIS

Oxford University, Oxford OX3 7LF, UK
(e-mail: sharon.curtis@ctsuo.ox.ac.uk)

Abstract

This paper describes some experiences of using fractal image compression as the subject of an assignment for a functional programming course using Haskell. The students were fascinated by the reproduction of images from their encodings and engaged well with the exercise which involved only elementary functional programming techniques.

1 Introduction

Fractal image compression (Jacquin, 1989; Barnsley & Hurd, 1992) is a lossy technique that compresses an image by finding a transformation which has a fixpoint closely approximating the original image, then storing (encoding) the parameters of this transformation in a file. The decoding process results in a reconstructed image that is a fractal, hence the name of the technique. This paper, based on our earlier work (Curtis, 2005), describes how a fractal image compression technique formed the basis of an undergraduate programming assignment at Oxford Brookes University, using the Haskell language (Peyton Jones, 2003).

1.1 Context

Many introductory programming courses cover fundamental principles and techniques by introducing basic language features and associated exercises that involve a certain amount of mundane repetition. Although such tasks have a crucial educational role, they can encourage students to look inwards and focus on computer science issues, rather than outwards towards the range of applications that show the flexibility and strength of programming as an activity (Rasala, 2000). One way that some educators have tried to improve student motivation is to introduce computations that involve simple image processing to practise essential techniques such as data abstraction and recursion, for example see Olson (2006).

Fractal images in particular have been used to inspire and motivate students at all levels of education (Peitgen *et al.*, 1991, 1992; Frame *et al.*, 2002) ever since the first attempts to visualise them were made (Mandelbrot, 1977). Their use in teaching remains strong (Fractal Foundation, 2013) even though the concept is no longer novel, partly because students with only very rudimentary skills in algebra and programming can grasp the simplicity of the algorithm to generate fractals like the Mandelbrot set, and students can therefore produce their own fractals and develop a visual understanding of the associated mathematics.

The fractal image compression technique has been used before in education, for example see the textbook by Welstead (1999), which is aimed at advanced undergraduate or beginning postgraduate level. There are also some more theoretical books that describe the underlying theory of the algorithm in depth (Fisher, 1995; Lu, 1997). Published implementations exist in a variety of languages, such as Java and C/C++ (Lu, 1997; Ullrich, 1999; Welstead, 1999; Hafner, 2000; Kanakarakis *et al.*, 2011) as well as functional languages (Thalabard & Zahariade, 2005; Baelde *et al.*, 2011).

This compression technique is seldom used in practice however, because the encoding process is computationally intensive to a degree that renders it less practical for most industrial applications than other lossy image compression standards like JPEG. It does have some significant strengths, including the potential to achieve both a high compression ratio and a high quality of the decoded image. In addition, the reconstructed image can be produced at a magnification independent of the size of the original. There has been much research on improving the efficiency of the encoding process, for example see Saupe (1994) & Lui *et al.* (2007), but the enduring interest in the technique may remain theoretical. There are some exceptions to this, including implementations like the digital encyclopedia Microsoft Encarta (Microsoft, 2009), which used the process to allow fast access to high quality images, and Iterated Systems' software ClearVideo that was involved in a live streaming broadcast of a jazz concert (Kaplan, 1997). More recently, the software Perfect Resize (OnOne Software, 2013) has used a fractal compression algorithm to offer an increase in image size without loss of sharpness or detail; fractal image compression has also been applied to satellite imagery (Ghosh *et al.*, 2004; Vaddella, 2010; Veenadevi, 2011).

One of the aims of this paper is to demonstrate that functional programming is well suited to modelling fractal image compression. It is an inherently functional technique: images and their transformations can be represented in a natural way by using functions (Elliot, 2003), and the decoding process involves iterating a transformation to reach a fixpoint image. In addition, both the compression and decompression processes involve a simple transformation of an input image to produce an output, without need for side effects or mutation.

Fractals have been used previously for educational purposes in functional programming, for example see Hudak (2000) & Jones (2004). However, we are not aware of any other uses of fractal image compression in the teaching of Haskell. We hope that this paper illustrates how students can use functional programming to model a sophisticated image manipulation technique as part of a student assignment that we hope might be considered as “nifty” (Stanford, 2013). One of its pedagogical strengths is that there is a natural separation of concerns between different parts of the algorithm which makes it easy to split it up into

independent tasks that are achievable by most students, thus reinforcing the importance of compositional program construction. In addition, many of the tasks involve higher order functions such as *map*, which abstract over traversal to eliminate the need for recursion, so this encourages students to develop a style of writing code that is concise and readable. One possible drawback of this exercise is that it can be time-consuming for students to develop, because the compression process is computationally intensive.

The paper is structured as follows: The basic fractal image compression concepts are explained in Section 2, and then implemented in Section 3. Testing and debugging is discussed in Section 4, followed by some suggestions for more challenging exercises in Section 5, based upon variants of the basic compression scheme. Finally, Section 6 provides additional pedagogical discussion of the work.

2 Fractal image compression primer

Fractal image compression involves concepts that are potentially challenging to computer science novices, many of whom have no mathematical training beyond a first year undergraduate course in discrete mathematics. The delivery of this assignment therefore included a primer to describe the basic algorithm, which incorporated plenty of visual examples to build students' intuition and ensure that the coursework was sufficiently accessible. We reprise this material in the following section, for the reader's benefit.

2.1 Fractals

Fractals occur throughout nature (Mandelbrot, 1983; Barnsley, 1988), for example, in clouds and coastlines, in ferns and blood vessels, and also in biological spirals like those found in ammonites and cacti. The fractal patterns of many repeated branching processes can be modelled by L-systems, which are formal string rewriting systems introduced in 1968 by botanist Lindenmayer (1968) to model the growth of plants. These provide a link to formal languages and grammars elsewhere in the computer science curriculum.

There are many resources to help students become more familiar with fractals. Online resources include the educators pack from the Fractal Foundation (2013), which contains numerous illustrations along with applications of fractals to medicine and engineering. Barnsley's (2011) website Superfractals has a gallery of recent fractals inspired by nature. Activities to construct fractals include building a Menger sponge out of business cards using Jeannine Mosley's technique (Wertheim, 2006), and drawing a Koch snowflake or Hilbert curve (Peitgen *et al.*, 1992) before writing program code to generate them. Some nice Haskell exercises for producing fractals include creating Mandelbrot sets (Jones, 2004), and snowflake fractals (Hudak, 2000).

2.2 Representing images

The ideas behind fractal image compression originated from Barnsley (1988) and were further developed by Jacquin (1989). In this section, we give a brief overview of Barnsley's work on the *Iterated Function System* (IFS) to show how images can be represented by using the fixpoint of an image transformation. The following section covers Jacquin's work which enables the compression of greyscale images in general.

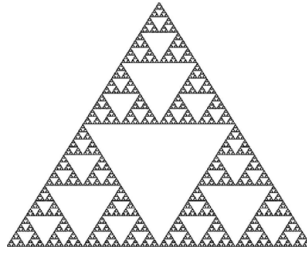


Fig. 1. The Sierpinski triangle.

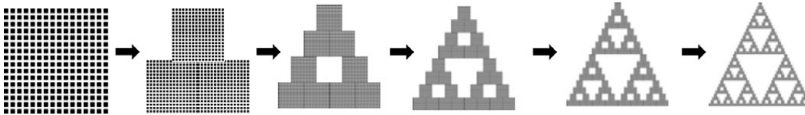


Fig. 2. Iterations of the transformation.

Consider the well-known Sierpinski triangle fractal, illustrated in Figure 1. This image is *self-similar*, because it consists of smaller copies of itself, positioned at top centre, lower left and lower right.

Let t_1 be the transformation that maps the whole image onto the shrunken copy of itself centred at the top of the image, in the style of a photocopying machine loaded with plain white paper, and set to $\times \frac{1}{2}$ magnification, with positioning set to top and centre. Similarly, let t_2 and t_3 be the transformations that map the whole image onto the half-size copies at lower left and lower right, respectively. Now define the image transformation t to be

$$t = t_1 \cup t_2 \cup t_3,$$

where the \cup operation has the effect of overlaying the results of the individual transformations onto one piece of paper. Then the Sierpinski triangle is a fixed point of t because the effect of applying t leaves it unchanged. Furthermore, given any non-blank square image x , the Sierpinski triangle is the limit of the following sequence of images:

$$x, tx, t^2x, \dots,$$

where t^i denotes the composition of t with itself i times. Thus, the Sierpinski triangle can be generated to any required level of detail by repeatedly applying t to any non-blank starting image, as illustrated in Figure 2. In this way, the transformation t can be said to *represent* the Sierpinski triangle. A Haskell implementation of this process can be found in Hudak (2000).

The above illustrates the concept of an IFS, which consists of a collection of contractive image transformations. A transformation is *contractive* if its application to any two points brings them closer together (in this context, “closer” means with respect to Euclidean distance). An IFS represents the image that is a fixpoint of the union of its transforms. Thus, the above collection $\{t_1, t_2, t_3\}$ of transforms is an IFS representing the Sierpinski triangle. Self-similar images can be described compactly by using an IFS. For further technical details about contractive transforms and the existence of unique fixpoints, see Section 2.5.



Fig. 3. Self-similarity between large and small portions of an image (enlarged).

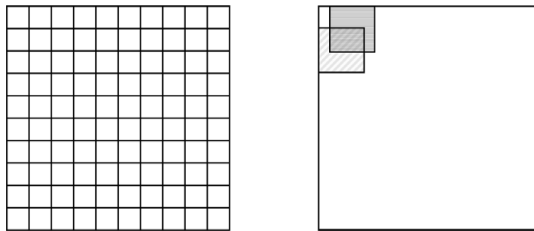


Fig. 4. Partitioning an image into range regions (left), and two possible domain regions (right).

2.3 Partitioned iterated function systems

Not many images are as conveniently self-similar as the Sierpinski triangle, so how can more general images be represented? We now consider greyscale images, many of which – especially photographs – do have areas that are almost self-similar, as illustrated in Figure 3. This leads to the concept of a *Partitioned Iterated Function System* (PIFS), which, like an IFS, consists of a collection of contractive transformations, representing the image that is a fixpoint of the union of the transforms. However, unlike an IFS, the transforms in a PIFS each operate on a separate part of the image, not the whole. This works by first partitioning the image into small regions (known as *ranges*), then matching each as closely as possible to one of several larger regions (*domains*), using a contractive transformation that includes contrast and brightness adjustments. These transformations collectively form the PIFS representing the image.

To illustrate, one way to partition an image for a PIFS is to divide it into a grid of small squares to form the range regions, as illustrated on the left-hand side of Figure 4. The domain regions are not as restricted: they can overlap, and need not cover the whole image, although in practice spreading them evenly across the whole image may result in a more varied selection for matching against range regions. However, the domain regions should be larger than the range regions (this helps produce detail in the compressed image), and shaped so that they map onto the same shape as the range regions under the chosen transforms.

For example, if range regions are small squares, then possible domain regions could be all the squares twice as wide, such as those on the right-hand side of Figure 4. To find the transforms, each range region is compared with all domain regions to seek a best match (for details, see Section 2.4). As well as shrinking the domain blocks, they are also considered in

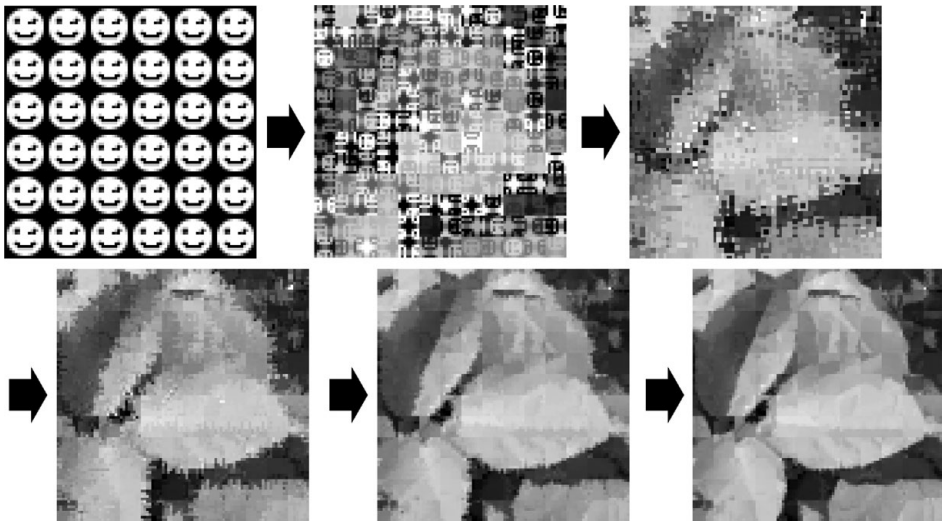


Fig. 5. Steps of the decoding process.

rotated and reflected orientations, and contrast and brightness can be adjusted. For example, Figure 3 illustrates a square range region of size 8×8 pixels in the lower centre of the image, along with a larger 16×16 -pixel domain region at upper left. This domain region, unrotated and unreflected, is a reasonably good match for the range region, needing only a modest contrast and brightness adjustment.

Labelling the range regions for the image as p_1, p_2, \dots, p_n , let w_i be the transform for the range region p_i , acting on the domain region that it best matches. The resulting PIFS then consists of the collection $\{w_1, w_2, \dots, w_n\}$, and the original image is then represented by the fixed point of the transformation

$$w = w_1 \cup w_2 \cup \dots \cup w_n.$$

The encoding (and thus the compressed form) of the image consists of the details of the chosen transforms of the PIFS, including their domains and ranges. The search for the best matching transforms is the most time-consuming stage in the fractal image compression process, and the time taken is strongly dependent on how many domain regions there are to compare against range regions. However, for compression techniques in general, spending some time achieving an accurate highly compressed image is a reasonable trade-off for many purposes. For example, the speed of viewing images on a web page depends on the size of the image file and the speed of the image decoding process, but not on the time that the website author spent encoding the image file in the first place.

It takes a relatively short time to decode the compressed image, and this process is illustrated in Figure 5, using an encoding of the image from Figure 3, with range and domain regions as illustrated in Figure 4. Starting with an arbitrary image of the same size as the original, the decoding process repeatedly applies the transform w , until the sequence converges. The resulting image approximates the original, but note that for this example, the quality of the fractal image compression is poor because the original image is only 96×96 pixels, and for small images there are fewer domain regions to choose from, leading

to poorer picture quality. For larger images the quality of the compression is much better, particularly when more sophisticated variants of the fractal image compression technique are used, rather than the basic scheme outlined in this paper.

2.4 Region comparison

This section gives details of how to compare range and domain regions in order to find the best match, by using a “distance” metric to measure how closely two image regions match. There are many different metrics available for comparing images (Fisher, 1995), and we used a standard method, the *root mean square (rms)* metric: If $[r_1, r_2, \dots, r_n]$ and $[s_1, s_2, \dots, s_n]$ are the numerical greyscale values corresponding to the pixels of two image regions, then the *rms* distance between them is $\sqrt{\sum_{i=1}^n (s_i - r_i)^2 / n}$.

When this metric is used to compare a range region with a domain region, contrast and brightness adjustments for the domain region pixels need to be considered, along with possible transforms to map the domain region onto the smaller range region. For example, if the regions are square blocks, then there are eight different rotation/reflection combinations that can be used to transform each domain region (illustrated in Figure 8, Section 3.3).

All the different possible transforms are considered during the encoding process, and the best are chosen. Each transform is applied to the domain region under consideration to yield pixels $[d_1, d_2, \dots, d_n]$ corresponding to the pixels $[r_1, r_2, \dots, r_n]$ of the range region. Applying a contrast c and brightness b adjustment to each pixel d_i then results in a transformed pixel $cd_i + b$. The *rms* calculation above now gives us a distance of $\sqrt{\sum_{i=1}^n (cd_i + b - r_i)^2 / n}$ between the range region and the transformed domain region. The values of c and b that minimise this value will provide the best match. 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.

An example of a range block and its best matching domain block are shown in Figure 6, which uses an image with greyscale values from 0 (black) to 255 (white). The range block is 8×8 pixels in size, and the possible domain blocks were 16×16 pixels, spaced 4 pixels apart across the image. The winning domain block has the best match with this range block by a reflection about its diagonal axis, along with a contrast scaling of -0.89 and a brightness offset of 204.4.

2.5 Contractive transforms

Although we do not discuss the following issue with the students, the alert reader may well be wondering whether the fixpoint of the transformation function for the encoded image is unique or not. After all, if the fixpoint is not unique, then a student decoding an encoded image might end up with an image that looks nothing like the original, which would not contribute to the credibility of the assignment!



Fig. 6. A range block (lower centre) and its best matching domain block (lower right).

The following discussion summarises the results from Fisher (1995) that provide reassurance; these are based on a standard theorem from topology, namely the *Contractive Mapping Fixed-Point Theorem*. This states that a mapping in a metric space has a unique fixpoint, provided that it is contractive with respect to that metric.

Black and white images, such as the Sierpinski triangle, may be represented as a non-empty set of points in a bounded area of the real plane (analogous to the toner dots produced by a photocopying machine on a piece of paper), with distance between sets/images being measured by the well-known Hausdorff metric. This image model can be used to show the uniqueness of IFS fixpoint images: If the transformations comprising an IFS are all based on mappings of image points that are contractive with respect to Euclidean distance, then the Collage Theorem ensures that the union of those transformations is contractive with respect to the Hausdorff metric, and therefore the Contractive Mapping Fixed-Point Theorem applies. Thus, the Sierpinski triangle has a unique fixpoint in this space of images, as its transform t comprises three individual transforms, all of which use mappings that halve the distance between image points.

Greyscale images need a more versatile representation, and a common model for pixelated images of width l and height h is an $l \times h$ matrix containing pixel values drawn from the interval $[0, 1]$, representing a scale of greys from black to white. The choice of metric, however, is not without problems. It is simpler in the theory to guarantee contractivity for a PIFS transform w with respect to the *supremum* metric, which defines the distance between two images as the largest difference between the greyscale values of corresponding pixels. A well-formed PIFS transform w is contractive with respect to this metric if each of the individual transforms $\{w_1, w_2, \dots, w_n\}$ comprising w have their contrast scaling factors c_i all satisfy $c_i < 1$. Note that contractivity with respect to Euclidean distance in the (x, y) direction of the image, achievable by having range regions smaller than domain regions, is not required to ensure a unique fixpoint image, but such contractivity is still desirable as this is how detail is created during the decoding process.

However, it is much more convenient to use the *rms* metric to find the PIFS transforms for an image: not only is it straightforward to find the contrast and brightness adjustments that produce the closest match (see Section 2.4), but an *rms* comparison corresponds reasonably well with human visual perception of the similarity between two images. Once a PIFS has been found with contrast scaling factors all satisfying $c_i < 1$, the existence of a unique fixpoint image is guaranteed by its contractivity with respect to the supremum

metric. Furthermore, empirical observation shows that values $c_i < 1.2$ are usually safe and can provide slightly better image quality.

An aside: The Sierpinski triangle transformation t does not have a unique fixpoint in the space of greyscale images, as a blank white image is also a fixpoint: imagine the results of feeding a blank piece of paper into the photocopier. As the machine preserves the full range of contrast between black and white, some of its contrast scaling factors do not satisfy $c_i < 1$, thereby illustrating the potential dangers of a non-contractive transform.

3 Implementation

The assignment consists of a series of tasks culminating in students' implementation of a basic fractal image compression scheme for greyscale images. Students supply definitions for all the functions described in this section, except those where we explicitly state otherwise. A sample completion of the fractal image compression code in Haskell, including the code supplied to students, can be found as supplementary material online at <http://dx.doi.org/10.1017/S095679681300021X>.

The assignment is split into two parts, to allow students to get feedback and sample answers on the initial tasks, before attempting the more difficult ones. The first part covers simple image manipulations, extraction of blocks from images and transformations on images and blocks (see Sections 3.1 to 3.3); the second part covers the encoding (compression) and decoding of images (see Sections 3.4 to 3.7).

3.1 Images

The assignment starts with some simple functions to create and manipulate images, and to access pixel data. Some simple types are provided for students to use, including one to represent greyscale images as arrays associating points – x, y coordinates – with pixel values:

```
type Image = Array Point Pixel
type Point = (Int, Int)
type Pixel = Int
```

Here, pixels are denoted by greyscale values within the range 0 (black) to 255 (white). The following function creates an image of given dimensions that is a uniform shade of grey:

```
type Dims = (Int, Int)
blankImage :: Dims → Int → Image
blankImage (w, h) g
  = array ((0,0), (w-1, h-1)) [((x, y), g) | x ← [0..w-1], y ← [0..h-1]]
```

This function can be used later on to produce an arbitrary starting image for the decoding process (see Section 3.7).

Images can be saved using the Portable GreyMap (PGM) image format, which stores an image in plain ASCII text, making it easy for students to inspect the pixel data. An example PGM image is shown in Figure 7, where the P2 on the first line of the file indicates that this is a greyscale image in the PGM format, the # indicates a comment line, the following line gives the size of the image, the line with 255 indicates that the pixels are in the range 0

```

P2
# arrow picture
5 5
255
255 255 0 255 255
195 255 0 255 195
0 195 0 195 0
255 0 0 0 255
255 255 0 255 255

```




Fig. 7. An example PGM image file (left), together with the image it represents.

(black) to 255 (white), and the following numbers are the pixel data from left-to-right and top-to-bottom.

Students are provided with input/output wrapper functions for reading and writing images to and from PGM files during the assignment along with some sample images. They can also convert files from more common formats like JPEG and PNG to PGM using free tools such as Gimp (GIMP, 2012) and IrfanView (Skiljan, 2012) as well as the Linux utilities `jpegtopnm` and `convert`, the latter being a part of the cross-platform ImageMagick suite (Still, 2005).

3.2 Extracting blocks

For simplicity, this fractal image compression scheme uses square blocks for the domain and range regions. A block is represented by the position of its top left-hand corner together with a listing of its pixels,

```

type Pixels = [Pixel]
type Block = (Point, Pixels)

```

and the following function

$$\text{getBlock} :: \text{Image} \rightarrow \text{Dims} \rightarrow \text{Point} \rightarrow \text{Block}$$

extracts a block of a given size and position from an image. The pixel values in a block are listed from left-to-right and top-to-bottom to match the way that they are listed in the PGM image format. The rationale for this choice of block representation is discussed in Section 6.

The range blocks are specified to be 8×8 -pixel squares, partitioning the input image in a grid fashion, as illustrated in Figure 4. The domain blocks are also square, twice as wide as the range blocks, but spaced half as far apart.

```

rangeBlockSize      = 8
domainScalingFactor = 2
domainBlockSpacing  = rangeBlockSize 'div' 2

```

These block sizes achieve a reasonable compromise between speed and compression quality: the smaller the range blocks and more numerous the domain blocks, the more faithful the encoded image, but encoding takes longer. Students are directed to use input images with dimensions that are multiples of the range block width, cropping where necessary, to avoid having to cope with awkward image sizes.

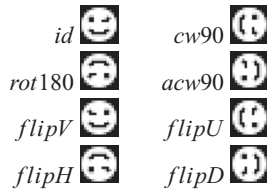


Fig. 8. The eight possible rotation/reflection combinations for square blocks.

The following function uses the above *getBlock* function to extract all blocks of a given size from an image at the given regular spacing:

$$getBlock :: Image \rightarrow Dims \rightarrow Dims \rightarrow [Block]$$

Then it is easy to extract all the range blocks from a given image:

$$\begin{aligned} rangeBlocks &:: Image \rightarrow [Block] \\ rangeBlocks\ im &= getBlocks\ im\ (r, r)\ (r, r) \\ &\text{where } r = rangeBlockSize \end{aligned}$$

However, it is inefficient to extract domain blocks from an image in the same way as range blocks, as discussed in the following section.

3.3 Image and block transformations

Since the algorithm must shrink the domain blocks down to the size of range blocks (as well as possibly rotating or reflecting them) before making any comparisons, it is more efficient to scale the input image first before extracting the domain blocks. The following function shrinks an image by a given scaling factor by averaging the pixel values:

$$shrink :: Int \rightarrow Image \rightarrow Image$$

The *domainBlocks* function for extracting all the shrunken domain blocks of an image uses both *shrink* and the *getBlock* function:

$$\begin{aligned} domainBlocks &:: Image \rightarrow [Block] \\ domainBlocks\ im &= map\ (scalePos\ d)\ (getBlock\ (shrink\ d\ im)\ (r, r)\ (s, s)) \\ &\text{where } r = rangeBlockSize \\ &\quad d = domainScalingFactor \\ &\quad s = domainBlockSpacing\ 'div'\ d \end{aligned}$$

where *scalePos* $:: Int \rightarrow Block \rightarrow Block$ relabels an individual block so that its position refers to the original image, not the shrunken one. Note that this efficiency improvement relies on the spacing of the domain blocks being a multiple of the scaling factor.

Besides scaling, the other transformations required are all the possible rotations and reflections on square blocks, as illustrated in Figure 8. The transformation *rot180* is simply the *reverse* function, but the other rotations and reflections provide an opportunity to use

common higher order list operators, for example:

$$\begin{aligned} \text{flipD} &:: \text{Pixels} \rightarrow \text{Pixels} \\ \text{flipD} &= \text{concat} \cdot \text{transpose} \cdot \text{toRows} \\ \text{toRows} &:: [a] \rightarrow [[a]] \\ \text{toRows} [] &= [] \\ \text{toRows } xs &= \text{take } r \text{ } xs : \text{toRows } (\text{drop } r \text{ } xs) \\ &\quad \text{where } r = \text{rangeBlockSize} \\ \text{transpose} &= \text{foldr } (\text{zipWith } (:)) (\text{repeat } []) \end{aligned}$$

For the encoding part of the assignment, students are supplied with the following list of the above eight transforms:

$$\begin{aligned} \text{transforms} &:: [\text{Pixels} \rightarrow \text{Pixels}] \\ \text{transforms} &= [\text{id}, \text{cw90}, \text{rot180}, \text{acw90}, \text{flipV}, \text{flipU}, \text{flipH}, \text{flipD}] \end{aligned}$$

This is because the indices of the transforms in the above list are used later to record which transforms produce the best block matches, and having all students using the same standardised list helps to avoid problems.

3.4 Block comparisons

The *rms* metric described in Section 2.4 is used to compare how close the matches are between the range blocks and the transformed domain blocks. For efficiency, the sums of the pixel values and their squares are obtained in advance for all the domain and range blocks extracted from the image, since they appear multiple times in the expression to be calculated. Students are given the following datatype for this purpose,

$$\text{type BlockSums} = (\text{Block}, \text{Int}, \text{Int})$$

The following function sums the pixel values and their squares for a block:

$$\begin{aligned} \text{sumBlock} &:: \text{Block} \rightarrow \text{BlockSums} \\ \text{sumBlock } (d, p) &= ((d, p), \text{sum } p, \text{sum } (\text{map } (^2) p)) \end{aligned}$$

Recall from Section 2.4 that the *rms* calculation comparing the pixels $[d_1, d_2, \dots, d_n]$ of the transformed domain block with the range block pixels $[r_1, r_2, \dots, r_n]$ results in three values, specifically:

- the *rms* distance $\sqrt{\sum_{i=1}^n (cd_i + b - r_i)^2 / n}$,
- the corresponding brightness adjustment b for the domain block, and
- the corresponding contrast adjustment c .

However, it is not necessary to calculate square roots: the value $\sum_{i=1}^n (cd_i + b - r_i)^2$ suffices to compare blocks, as the $\sqrt{\quad}$ function is monotonic, and so too is division by n , as the range block sizes are all the same. Thus, the following function produces only the value $(\sum_{i=1}^n (cd_i + b - r_i)^2, b, c)$ for the *rms* calculation:

$$\text{match} :: \text{BlockSums} \rightarrow \text{BlockSums} \rightarrow (\text{Float}, \text{Float}, \text{Float})$$

Here the calculation of the brightness and contrast values is a direct implementation of the formulae given in Section 2.4, and the value of the *rms* distance comes from a straightforward rearrangement of the expression $\sum_{i=1}^n (cd_i + b - r_i)^2$ to make use of the previously calculated sums of pixel values and their squares. We supply the definition of the *match* function to students to save the time involved with its tedious conversion of integral pixel values to floating point numbers; however, this can be easily set as an exercise for students with more time available.

3.5 Representing transforms

At this point, almost all of the ingredients required to implement the encoding are in place, but it remains to supply a means of representing the transforms recording the relationships between the range blocks and the matching domain blocks. The following type is used to hold this information:

$$\text{type } \textit{Trans} = (\textit{Point}, \textit{Point}, \textit{Int}, \textit{Float}, \textit{Float}, \textit{Float})$$

Here the tuple values are the position of a range block, the position of the matching domain block, the index of the transform used, the brightness and contrast adjustments, and the *rms* value from the calculation above, respectively. For example, Figure 6 illustrates a range block at position (48, 80) along with a matched domain block at (64, 60), using the reflection *flipD*, and the tuple ((48, 80), (64, 60), 7, 204.4, -0.89, 10240) of type *Trans* corresponds to this transform.

The function *storePifs* takes a list of transforms and produces a string suitable for saving in a file:

$$\textit{storePifs} :: [\textit{Trans}] \rightarrow \textit{String}$$

This definition is supplied to students so that the results of the compression algorithm can be saved easily in a standardised format. The output of the compression algorithm is a list of transforms corresponding to the best matches between range and domain blocks, comprising the PIFS that collectively represents the original image. An example string produced by *storePifs* is illustrated in Figure 9.

Although the general aim of fractal image compression is to store images using as few bytes as possible, we use a somewhat wasteful plain text format for the saved compressed image, including a header string to provide a reminder about which number is in each column. This is because the assignment is designed for education rather than code performance, and the readable format for the compressed image helps with debugging (see Section 4) and understanding how the algorithm works.

Indeed, if maximum compression is the goal, then there is no need to save the *rms* distance information, as this is only used for comparing blocks, and not for decoding. Also, with a fixed-grid partitioning scheme for range regions, there is no need to store the range block positions, as they can be inferred from the dimensions of the image and the position of the transform details in the saved file. Further suggestions for efficient storage are given in Fisher (1995); for example, just three bits can be used to store the index of the transform.

3.6 Encoding

Students are supplied with a summarized description of the compression algorithm to illustrate how the various pieces of code fit together: .6

- 1) Generate all of the domain blocks in the shrunken image.
- 2) For each domain block:
 - Generate all eight of its transformations (i.e. rotations/reflections).
- 3) Generate all of the range blocks in the image.
- 4) For each range block:
 - Calculate the rms distance to each transformed domain block, using optimal contrast and brightness.
 - Select the best matching domain block and store the resulting transform.

Here is the *encode* function itself:

```

encode    :: Image → [Trans]
encode im = map (bestMatch tdoms) rans
            where doms = map sumBlock (domainBlocks im)
                  tdoms = concatMap applyTransforms doms
                  rans = map sumBlock (rangeBlocks im)
  
```

The definition of *encode* can be supplied to students to help them keep to an efficient algorithmic structure as well as applying the (*map sumBlock*) function to the range blocks and shrunk domain blocks, the domain blocks need to have each of their eight rotation/reflection transforms pre-applied, to prevent their application being repeated unnecessarily during the encoding. The function *applyTransforms* produces these transformed domain blocks, labelling them with the index number of the rotation/reflection used from the *transforms* list.

```

applyTransforms :: BlockSums → [(BlockSums, Int)]
  
```

The *bestMatch* function, given a list of transformed domain blocks, finds the best match for a given range block. This can be implemented with a simple map/fold combination, for example:

```

bestMatch    :: [(BlockSums, Int)] → BlockSums → Trans
bestMatch ts r = foldr1 goCompare (map (getTrans r) ts)
  
```

where *getTrans* returns the transformation corresponding to a given range and domain block, using the function *match* from Section 3.4, and *goCompare* compares two transforms and returns the better match

```

getTrans    :: BlockSums → (BlockSums, Int) → Trans
goCompare :: Trans → Trans → Trans
  
```

The method used by *goCompare* to decide the better match of two transforms is not entirely trivial. Choosing the transform with a smaller *rms* distance results in better quality of the compressed image, but for greater safety in ensuring that the resulting PIFS has a unique

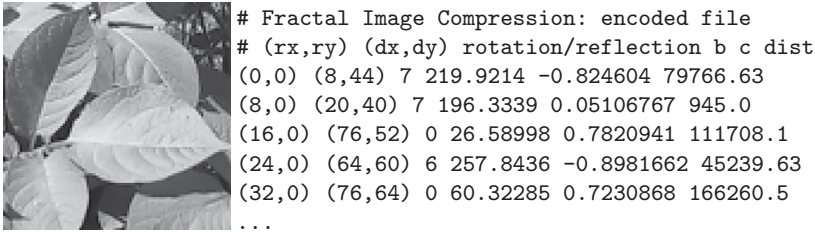


Fig. 9. An example 96×96 -pixel image, together with the first few lines of the encoded file produced.

fixpoint (see the discussion in Section 2.5), transforms with contrast multipliers less than a suitable *contrastBound* can be preferred, but otherwise the choice is based on the *rms* distance.

```
contrastBound :: Float
contrastBound = 1.2
```

This completes the code for the encoding; we also provide students with an I/O wrapper for the *encode* function to assist them with loading images from PGM files and saving the results of the compression. Figure 9 illustrates an example result of an image encoding.

The performance of the encoding algorithm is discussed in Section 3.8. In practice, if students are being slowed down by waiting for results, there are several possible ways to speed up the computation. One way is to use smaller images: even 96×96 -pixel images can give a reasonable image quality.

Another alternative is to use an error tolerance: instead of searching through all of the domain blocks to find the best match, the *bestMatch* function can be replaced with an alternative function that stops when it finds an acceptable match (with suitably low contrast and *rms* distance), and if none is found, then the best match available is selected, as before.

3.7 Decoding

The decoding process takes as input an arbitrary starting image, along with the list of transforms (PIFS) produced by the *encode* function, and then repeatedly carries out the whole-image transformation that the individual transforms collectively represent, as illustrated in Figure 5. The *decode* function, supplied to students, is structured to take a fixed quantity of steps, rather than iterating the image transformation until a fixpoint image is reached, with the resulting burden of detecting termination:

```
decode :: Int → Image → [Trans] → Image
decode i im ts = (iterate (decodeStep ts) im) !! i
```

Note that this definition provides an opportunity to illustrate to students the effects of Haskell's lazy evaluation: the first few reduction steps of *(iterate (decodeStep ts) im) !! n* lead to a simple repeated application of *decodeStep*, which is more memory-efficient than the use of the *!!* indexing operation might suggest.

While in theory, the decoding can be carried out at whatever level of magnification is desired, it is simpler to use a starting image of the same dimensions as the original for this programming assignment. Then the domain and range blocks are of the same size as in the

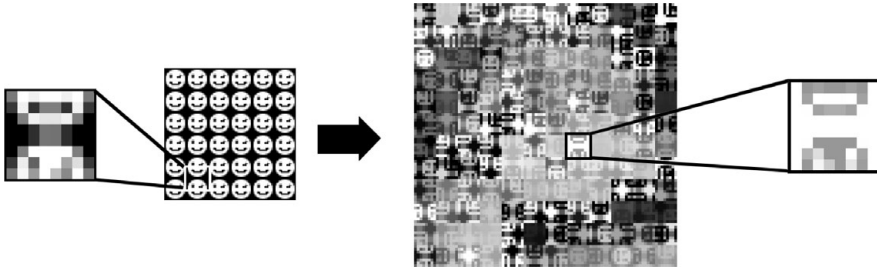


Fig. 10. The effects of the transform $((56, 48), (16, 72), 4, 356.0677, -0.8026732, _)$: the shrunk domain block with original position $(16, 72)$, on the left, is reflected about the y axis (the transform with index 4), and then a contrast scaling factor of -0.8026732 is applied to the pixel values, followed by a brightness adjustment of 356 to produce the range block at position $(56, 48)$. Note the $-$ sign on the contrast multiplier, which inverts the pixel shades.

encoding, and the output of the decoding can be easily compared with the original image to examine the quality of the compression. For a starting image, students can either use the image produced by the *blankImage* function in Section 3.1, or a suitably sized image from a file, such as the “smiley faces” image used as a starting point in Figure 5. For ease of input, there is an I/O wrapper for the *decode* function that also reads the encoded file and the starting image.

A single step of the decoding can be implemented as follows:

$$\begin{aligned} \text{decodeStep} &:: [\text{Trans}] \rightarrow \text{Image} \rightarrow \text{Image} \\ \text{decodeStep } ts \text{ im} &= \text{glue im (map (makeRangeBlock im') ts)} \\ &\text{where } \text{im}' = \text{shrink domainScalingFactor im} \end{aligned}$$

Here the function *decodeStep* applies one PIFS transformation, preshrinking the input image by the domain/range block scaling factor for efficiency purposes.

The function *makeRangeBlock* returns a range block produced by the effect of a single domain block transform, and a function *glue* assembles all the resulting range blocks together into an array of coordinate/pixel associations to form an image of the same size as before. Assistance is given to students when defining *glue* by providing a suggested type of a subsidiary function that they could use

$$\begin{aligned} \text{makeRangeBlock} &:: \text{Image} \rightarrow \text{Trans} \rightarrow \text{Block} \\ \text{glue} &:: \text{Image} \rightarrow [\text{Block}] \rightarrow \text{Image} \end{aligned}$$

Figure 10 illustrates the effect of the *makeRangeBlock* function. In its implementation, care needs to be taken to ensure that the domain block positions refer to the correct place in the preshrunk image. Also, when applying the contrast and brightness transformation to the rotated/reflected pixels of the domain block,

$$\begin{aligned} \text{adjustCB} &:: \text{Float} \rightarrow \text{Float} \rightarrow \text{Pixel} \rightarrow \text{Pixel} \\ \text{adjustCB } c \text{ b } p &= \text{greyscale } (c * \text{fromIntegral } p + b) \end{aligned}$$

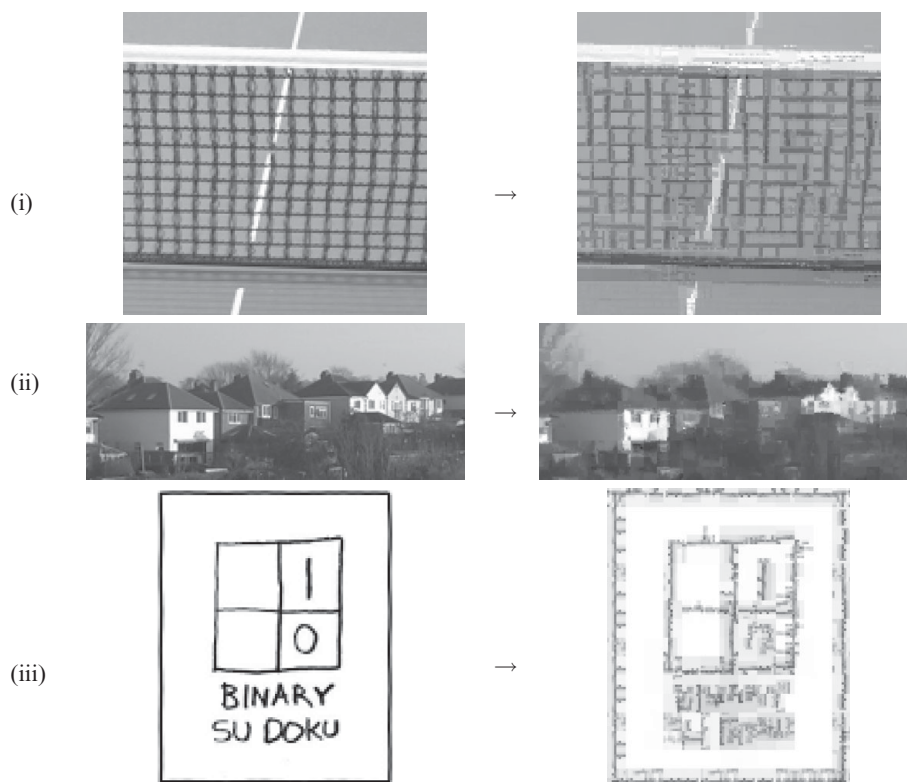


Fig. 11. Sample results of image compression. (i) A 144×144 -pixel photograph of a table tennis net. (ii) A 232×96 -pixel photograph of some houses. (iii) An *xkcd* cartoon (Munroe, 2006) at 104×128 -pixel size.

the pixel values are kept within the 0–255 range by using the following function:

$$\begin{aligned}
 & \text{greyscale} :: \text{Float} \rightarrow \text{Pixel} \\
 & \text{greyscale } p \\
 & \quad | p < 0 \quad = 0 \\
 & \quad | p > 255 = 255 \\
 & \quad | \text{otherwise} = \text{truncate } p
 \end{aligned}$$

After completing the code for the decoding, students can then explore the results. It is interesting and instructive to look individually at the first few decoding steps using *decode 1*, *decode 2*, etc., and also decoding several times using different starting images, as it illustrates how the iteration converges on the same fixpoint image.

Figure 11 illustrates some sample results of decoding some small images encoded with fractal image compression.

3.8 Performance

This section analyses the performance of the encoding and decoding functions in three ways: image quality, running time, and compression ratio.

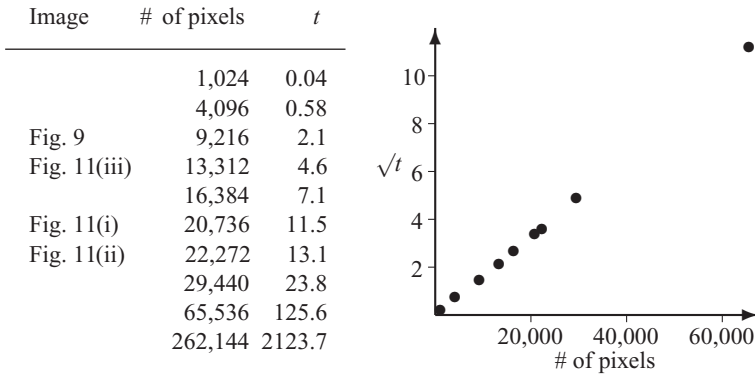


Fig. 12. Left: Encoding times for 10 images of varied sizes, where t is the time taken in seconds. Right: A plot of \sqrt{t} , compared with image sizes (all but the largest image shown).

Image quality

From empirical observation, we find that photographs tend to result in decoded images that are suitably faithful to the originals even if they do not possess obvious self-similarity at different scales, for example, image (i) from Figure 11. However, fine detail in images does not encode well in this basic compression scheme, especially for small images with their smaller selection of domain blocks to choose from. This is illustrated in the poor reproduction of the house windows in Figure 11(ii), and the blurring of the line drawing in Figure 11(iii). Nevertheless, the quality of the decompressed images was still enough to impress students.

Running time analysis

If we consider the block sizes, spacing and domain block scaling factors as constants, then the encoding algorithm takes time quadratic in the size of the image (denoted by n , the number of pixels, in the explanation that follows.)

Retrieving the input image from a file takes linear time since array construction in Haskell is linear. Then building the lists of domain and range blocks and annotating with sums and sums of squares of pixel values also take linear time, since there are $O(n)$ blocks, and each block is retrieved using constant time array access. Similarly, applying the affine transforms to the domain blocks takes $O(n)$. It is the matching of each of the $n/64$ range blocks to a best-suited domain block that takes the time, and as the number of domain blocks is linear in the size of the image, overall the matching takes $O(n^2)$ time,

$$\begin{aligned}
 \text{encode } im &= \text{map } (\text{bestMatch } tdoms) \text{ rans} && O(n^2) \\
 \text{where } doms &= \text{map } \text{sumBlock } (\text{domainBlocks } im) && O(n) \\
 tdoms &= \text{concatMap } \text{applyTransforms } doms && O(n) \\
 rans &= \text{map } \text{sumBlock } (\text{rangeBlocks } im) && O(n)
 \end{aligned}$$

Empirically, we tested the performance by running the encoding on assorted images (including those shown in this paper, and in its supplementary materials) varying in size from 32×32 to 512×512 pixels, the results of which are shown in Figure 12. All test runs were executed on a machine running Windows XP Professional version 2002 with Service Pack 3 on an Intel Core 2 Duo 2.40 Ghz with 2-GB RAM. The programs were compiled with

the Glasgow Haskell Compiler (GHC), version 6.10.1, and execution times (in seconds) were read from the profiling information generated using the *-prof* compiler flag.

In contrast, the decoding is much quicker, with each step of the decode taking $O(n)$. This is because the shrinking of the input image is linear in the image size, the construction of the new range blocks is linear because array accesses take constant time, and constructing the new image is also linear. The number of decoding steps can vary from one image to the next, depending on how fast the shrunken domain blocks can transfer detail to individual pixels, but the vast majority of images we tried reached a stable state after just seven iterations. For i decoding steps, the time taken is $O(ni)$, and in practice, the decoding takes a very short time, with even the 512×512 -pixel image taking only 3.7 seconds to decode over 10 iterations.

One possible exercise is to set students the task of doing some timing experiments to see if their implementations are also quadratic in the size of the images.

Compression ratio

Students may like to consider the compression rate achieved with their encodings. This is not directly obvious from a simple comparison of file sizes, as the encodings are human-readable and stored in plain text files, not optimized for efficient storage.

However, it makes a good exercise to calculate the compression ratio, making reasonable assumptions on the number of bytes used for storage. For example, the image in Figure 9 takes approximately 9.2 Kb to store uncompressed, as can be seen directly if the PGM image is stored in “raw” format rather than ASCII. For this image, if the encoded range block transformations were stored using raw bytes without extraneous information, then the encoded file would occupy just over 1.4 Kb, giving a compression ratio of approximately 6:1. Alternatively, students can zip their *.fic* files to get a sense of how much compression their encodings achieve.

In general, compression ratios produced by this basic fractal image compression scheme are no better than those achieved for the JPEG compression standard for similar image quality. However, more sophisticated versions of the technique (e.g. see Section 5.3) can outperform JPEG, and produce results of comparable quality to widely used wavelet-based compression schemes (Fisher, 1995).

4 Testing and debugging

This assignment can seem daunting to students, as there are many component functions that are assembled into the substantial compression and decompression algorithms. It is important to test all the functions carefully, step-by-step, before attempting the encoding and decoding, so the instructions and code given to students come with plentiful support for testing and debugging. Note that functions providing support for testing are included with the supplementary code provided with this paper.

4.1 Component functions

The assignment instructions include sample test cases to help students validate all of the component functions. Some of these tests use small hard-coded blocks or images and state

explicitly what happens to the pixel values, and others display observable changes to larger images. For example, we supply tiny images like the arrow picture in Figure 7, hard-coded as a value of type *Image*, as well as similar examples of type *Block*. These allow pixel values to be inspected in detail for a manageable size of input, when examining the output of functions producing block positions, blocks, and blocks annotated with sums and sums of squares of pixel values. The tests for the transform and block extraction functions are more visual, and use supplied I/O wrapper functions that facilitate the use of PGM images for testing so that both input and output can be viewed as an image.

4.2 Encoding & decoding

Clearly, the ultimate test is to see whether encoding an image and then decoding it returns an approximation of the original, but such tests cannot be carried out until the code for both encoding and decoding is complete. This is unsatisfactory for testing purposes, as if such a test fails, it is not then clear whether the fault lies in the encoding, or decoding, or both. Each part needs to be tested separately.

Encoding

One way to help students debug their encoding functions is to supply sample encodings of given images so that they can compare their output numbers with encodings known to be good. Note that students may not necessarily get exactly the same encoding as the supplied example however, because they may have chosen a different way of breaking ties when choosing a minimum transform using the *goCompare* function. However, almost all of the numbers should be the same so that a student's correct results should closely match the sample encoding.

If a student's compression of a sample image looks nothing like the known encoding, then the text file containing the results needs to be examined in detail. We find that student errors tend to be either in mislabelling, or in selecting the wrong transform, and here are some pertinent questions to assist in debugging:

- Is there precisely one entry for every range block?
- Could the (x,y) coordinates of the domain or range blocks have been switched around accidentally? For example, try using a short but wide image, like the 232×96 -pixel image of Figure 11(ii); domain blocks are expected in the encoding at positions like (108, 72), but not (72, 108).
- Inspect the original image, and choose a specific range block that looks distinctive visually (pixel coordinate information available in typical image editing software should make it easier to identify individual blocks). Look at the encoding to see which domain block has been selected as the closest match for that range block, with what rotation/reflection, and what contrast and brightness values? Does that seem to match visually in the image (remember that a negative contrast value will invert the colours)? For example, see Figure 13, where the darker shape within the matching domain block and a contrast multiplier of 0.317 corresponds to a fainter similar shape in the range block.

It can also help to test the encode function using an alternative I/O wrapper that writes results to the screen, instead of saving them in a file. Laziness will ensure that matches

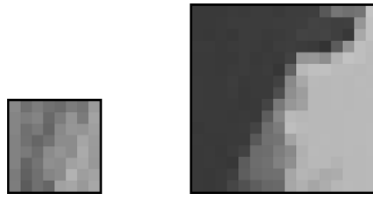


Fig. 13. On the left: The range block from the top left of the image in Figure 11(ii). Its closest match is the domain block at position (132,28), with reflection 5 (*flipU*), and adjustments for brightness 98.5 and contrast 0.317. On the right: The domain block reflected for comparison without any contrast/brightness adjustment.

for range blocks get displayed as the code finds them. This means that instead of having to wait for an encoding to finish, students can start examining whether the results seem reasonable as soon as the first range block has been produced. One way to exploit this is to choose an image with a distinctive range block at the top-left, such as in Figure 13.

Decoding

There is a case for designing the assignment so that students implement the decoding before the encoding: the decoding is more straightforward, and gives an understanding of what is required for the encoding before it is attempted. In any case, students can test their decoding independently of the encoding by using known encoded images supplied by the instructor. The following suggestions may help to debug the decoding function if the results seem visually strange:

- Try using a uniformly grey image as input, for just one iteration. The range blocks should be clearly visible in the decoded image as a neat grid of squares in assorted shades of grey: if not, either the *glue* function is not assembling the blocks correctly, or the *makeRangeBlock* function is producing blocks with strange positions or sizes. Also, each range block should be a uniform colour, with no surprising colours (e.g., a dark block in an area where the original image is very light). Any surprising colours suggest errors in the contrast/brightness calculation.
- Try using an input image made up of repeating distinctive range blocks with a black and white asymmetrical pattern, such as the smiley faces image in Figure 5, and do just one iteration. From a visual inspection, it should be possible to see whether the domain blocks are being shrunk properly, whether they are being rotated/reflected correctly, and whether the contrast and brightness adjustment seems to be consistent with the values given in the encoding.

5 Further exercises

This section contains suggestions for further programming exercises based on fractal image compression; they are not student-tested due to teaching time constraints. The tasks are suitable for more advanced students with time available, and perhaps may suggest avenues for student projects.

In addition, the techniques outlined in Section 5.3 provide the opportunity for students to use different functional programming techniques: compared to the list comprehensions

and map operators used in Section 3, the flexible partitioning schemes described below make use of recursion and tree structures.

5.1 Colour images

Students can also use the above techniques to compress colour images. The method is similar to that used for greyscale images: the pixel data is split into different colour channels, and then each channel is encoded separately. However, the standard RGB colour model does not give good results: as human perception is more sensitive to brightness than colour, a model based on brightness (or luminance), hue, and saturation works better. Then the channels encoding hue and saturation can be compressed more than the brightness channel, typically at half-resolution. Results vary depending on which HSV model is used; chapter 2 of Fisher (1995) recommends the YIQ model, which is used in the NTSC television broadcasting standard:

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}.$$

Here Y is the luminance, I is the hue, and Q is the saturation. The reverse conversion is as follows:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.273 & -0.647 \\ 1.000 & -1.104 & 1.701 \end{pmatrix} \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}.$$

Colour images can be stored in human-readable form using the Portable PixMap (PPM) image format that is similar to the PGM format used for greyscale images. Tools for converting images to and from Portable PixMap formats are listed at the end of Section 3.1.

5.2 Zoomable images

One of the advantages of the fractal image compression method is that the decoding is resolution-independent: for example, the resulting image can be produced at half or twice the resolution of the original. Note that this magnification does not mean that additional detail can be seen: magnifying an image of a snowman will not show the individual snowflakes. The additional detail is an artificial side effect of the chosen domain blocks. However, images magnified in this way can appear more aesthetically pleasing than other interpolations.

The implementation given in Section 3 does not feature resolution independence, but the decoding can be altered to provide this facility. The user supplies the desired magnification/reduction factor, either explicitly or through the input of a starting image of the required size.

5.3 Adaptive partitioning

There has been much research over the years to improve the original fractal image compression method invented by Barnsley (1988) and Jacquin (1989) by speeding up the

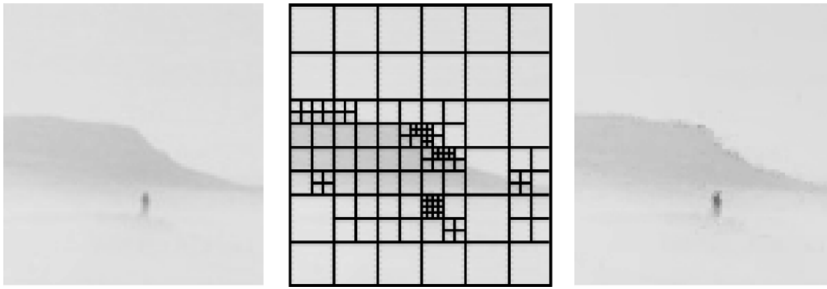


Fig. 14. Original image (left), quadtree range block partitions (centre), and decoded image (right).

computation and improving the quality of the compressed image. The search for matching domain blocks is the most time-consuming part of the compression algorithm as well as block classification and indexing schemes (Jacquin, 1989). To reduce the size of domain block searches, there exist sophisticated schemes to find matching image portions such as nearest-neighbour searches (Saupé, 1994; Cardinal, 1999), which have similarities to the techniques used in PatchMatch (Barnes *et al.*, 2009).

A variety of adaptive partitioning schemes exist that address image quality by choosing range and domain regions that vary in size and shape. Some partitioning schemes use triangles and more general polygons (Davoine *et al.*, 1997); here we take a brief look at two methods that use rectangular blocks, and feature the opportunity to use more varied functional programming techniques, including recursion and tree datatypes.

Quadtree partitioning

This method uses square range blocks that can vary in size, as illustrated in Figure 14. The encoding starts using a grid of square range blocks, similar to before, but when a range block does not have a suitably accurate match with any of the domain blocks with respect to some error tolerance, it is divided into four quadrants and the process is repeated. This recursion continues until a good enough match is found, or a specified minimum block size is reached. Figure 14 illustrates how the range blocks are much smaller in the more detailed areas of the image. Further details of the partitioning technique using quadtrees can be found in chapter 3 of Fisher (1995), or chapter 3.3 of Welstead (1999).

HV partitioning

This method recursively divides range blocks until a good enough match is found in a similar way to the quadtree partitioning scheme, but it is more flexible in how it carries out the division, as illustrated in Figure 15. Here range blocks without a suitably accurate domain block match are subdivided into two blocks, either horizontally or vertically (hence, “HV” partitioning), and the tree so formed is thus a k -d tree of dimension 2 (Bentley, 1975). This division is made in a smart way: the choice of where to divide a block is made according to where the biggest differences in neighbouring row/column pixels can be found.

Similar to before, there is a minimum range block width and height, and varying sizes of domain blocks can be used, although it is certainly convenient if the domain blocks under consideration have widths and heights that are multiples of the particular range block being considered. As before, the partitioning scheme ends up using smaller range blocks in the

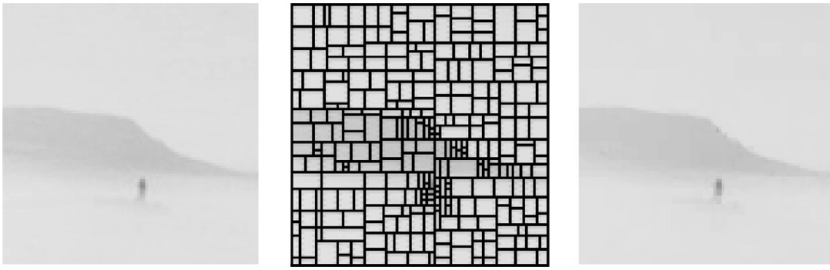


Fig. 15. Original image (left), HV range block partitions (centre), and decoded image (right).

more detailed areas of the image (see Figure 15), but HV partitioning is more flexible, and thus usually results in better image quality. Further details of HV partitioning can be found in chapter 6 of Fisher (1995).

Both quadtree and HV partitioning require adjustments to the basic compression code discussed in Section 3 in order to keep it reasonably simple for students. As well as restructuring the top-level part of the encoding process to use recursion, the other main changes needed are as follows:

- The *Block* and *Trans* datatypes need to store width/height information for blocks, and as a result several functions would need to be updated.
- The pre-shrinking of images for domain blocks would need to be reorganised, both for encoding and decoding.
- The *rms* calculation can no longer take the shortcut of omitting the $/n$ step because distances need to be averaged per pixel in order to consistently evaluate blocks of differing sizes against an error tolerance.

6 Discussion

Background and context

The inspiration for this functional implementation of fractal image compression originally came from the treatment of images in the Haskell library Pan (Elliot, 2003). In Pan, images are represented as polymorphic functions from infinite, continuous two-dimensional space to some pixel type. This representation lends itself very well to manipulation by affine transformations of the kind used for fractal image compression, as such conversions are simply higher order functions which can be glued together in a variety of ways.

Initially, we set a simpler functional programming assignment focusing on manipulations of colour images, including affine transformations and colour changes. In that instance, the use of images gave valuable visual feedback to students, aiding their code testing. The success of that assignment gave us the confidence to go on to a more challenging subject of fractal image compression.

Subsequently, we set two slightly different variations of this fractal image compression assignment for undergraduates, first in 2007 and then again in 2011. In 2007, the students took functional programming as an optional module during the second or third year of their degree after previously studying imperative programming and data structures. By 2011, however, the curriculum had been restructured, and the students learnt to use

Haskell as part of a more general declarative programming module, which also covered logic programming. Consequently, this second group was less practised with functional programming techniques, having only studied them for five weeks. The short syllabus for this later run included basic concepts such as recursion, list comprehensions, currying, higher order functions, abstract data types, type classes, lazy evaluation, and infinite lists, but did not include monads. However, even this smaller range of topics was sufficient to allow students to tackle this assignment.

Technological choices

GHC is the recommended choice in terms of speed, but since some of our students used the Hugs interpreter (Hugs, 2006), we did check that our code was runnable in this environment without problems. In either case, we would recommend increasing the size of the heap (using the `-H` option in GHC or `-h` for Hugs.)

Concerning datatypes, although it is arguably simpler to use just one data type for data storage, for pedagogical reasons we consciously chose to use both lists and arrays: lists of pixels for image blocks, and arrays for entire images. Array processing is a useful concept for functional programmers to learn, with its efficient $O(1)$ access and $O(n)$ construction times; indeed without the use of arrays in our code, the block extraction and image reconstruction would have been far less efficient and more complicated. List manipulation is also a fundamental topic in most traditional introductory functional programming textbooks, for example, Hutton (2007) & Thompson (2011), which tend to omit arrays. So lists of pixels were chosen for the block representation even though they were not necessarily the most efficient option. They have the advantage that they give students a chance to demonstrate their ability to write list-processing functions and to use common higher order functions to develop concise code, as required for the rotation and reflection transformations on blocks, for example. The application of these skills was an essential learning outcome of the assignment.

Support for students

Although the implementation of the fractal image compression algorithm does not rely on any sophisticated functional programming techniques, it is still a challenging concept for novices. As well as a verbal presentation of the fractal image compression concept with plenty of visual examples, we took further measures to help support students.

We structured the assignment carefully: As the testing of later functions relied on the correctness of earlier written functions, we split the assignment into two parts. The earlier part involved basic functions and datatypes concerning images, blocks, and affine transformations, and we provided sample answers for this part before students attempted the latter part containing the encoding and decoding. Also, as there are many pieces to the compression algorithm, we outlined the structure of the code for students, providing several function types as a guide, to make the tasks straightforward for students to understand. In addition, we supplied some difficult and/or tedious functions for students in order to match the difficulty level of the exercises to the students' abilities and the time available.

In addition to some sample images, full code for PGM image input and output was supplied to students, as I/O was beyond the scope of the course. Also, we wanted students

to concentrate on the functions directly concerned with the fractal image compression, rather than parsing or writing PGM files.

Results

Overall, both groups of students enjoyed the assignment, with many of them completing the algorithm and managing to compress their own images. However, there were some issues that arose on the first run of the assignment, and were subsequently addressed.

The first time we set the assignment, during the full functional programming module, 90% submitted work for the first part of the assignment and received a passing grade for it. However, even though the average mark for the second part was 80%, only 75% of the students submitted work for the second part. Feedback from students indicated that some of the more able students felt that the design of the assignment was too prescriptive, and they thought they should have been given more scope to design the whole algorithm from scratch. It was difficult to pinpoint accurately why so many students did not complete the second part of the assignment, but our impression was that they lacked the confidence to attempt the rest of the exercises, possibly due to lack of opportunities for testing of their functions so that they could see the results visually.

We modified the assignment on the second run to take these issues into account, and also to reflect the lesser exposure of this group of students to Haskell. We made the exercises more visual, and included some extra test functions so that students could more easily see the domain and range blocks that they had extracted, and the results of their image transformations. We gave students some pseudocode to outline how the encoding and decoding worked, and supplied Haskell code for the top-level encoding and decoding so that the students could see where to fill in the gaps. Finally, we included some questions with extra challenges, to offer a little more scope for creativity.

Participation was much better the second time round: all of the students submitted answers to both parts, and they all passed, with an average mark of 81% overall. Students generally enjoyed the assignment, and the feedback was positive, for example: "I liked the way the coursework was set out, especially how the image compression algorithm was mostly written for us and we just had to complete the undefined functions – since this way we were able to concentrate on understanding how to program declaratively."

Next time

We conclude that the changes made for the second run of the assignment were successful in solving some of the problems, but there is still room for improvement. If we were to set this assignment again with more time available, we would keep the step-by-step approach for the basic algorithm, but would offer students greater scope for extra challenges in order to achieve a broader spread of marks and to stretch the stronger students, for example, using one of the variations described in Section 5, and calculating compression ratios for their own images.

We would also include more introductory material on fractals as described in Section 2.1 (some of which was suggested by an anonymous referee) to help build students' intuition.

Another change to help students would be to ask them to implement the decoding before the encoding in the second part of the assignment, providing them with some compressed images to decode. We estimate that this would be more straightforward for students, and it

should help students see what they are aiming for when writing the functions to carry out the encoding.

Reflections

Overall, we were very happy with using the topic of fractal image compression as an assignment for students. Novice functional programmers were able to understand a sophisticated technique for image compression and implement it, and had the satisfaction of decoding an encoded image to produce an approximate version of the original.

Also, the assignment had a good coverage of basic functional programming techniques, including list comprehensions, several standard higher order functions, and even lazy evaluation and infinite lists, in the use of the *iterate* function to do the decoding. It also illustrated the benefits of compositional program design. We felt that this assignment offered students a good chance to see how this algorithm could be represented compactly using a functional programming language.

Acknowledgments

Thanks go to Ian Bayley, who also taught on the course and provided several of the I/O wrapper functions for testing purposes. Thanks also go to the anonymous referees who provided numerous suggestions that helped improve this paper.

References

- Baelde, D. & Mimram, S. (2011) Fractal compressor. Accessed March 26, 2013. Available at: <http://fractcompr.sourceforge.net/>.
- Barnes, C., Shechtman, E., Finkelstein, A. & Goldman., D. B. (2009) *PatchMatch: a Randomized Correspondence Algorithm for Structural Image Editing*. ACM Trans. Graph. **28**, 3 (Article 24).
- Barnsley, M. F. (1988) *Fractals Everywhere*. Waltham, MA: Academic Press.
- Barnsley, M. F. (2011) *Superfractals*. Waltham, MA: Academic Press. Accessed March 20, 2013. Available at: <http://www.superfractals.com/>.
- Barnsley, M. F. & Hurd, L. P. (1992) *Fractal Image Compression*. Natick, MA: AK Peters.
- Bentley, J. (1975) Multidimensional binary search trees used for associative searching. *Commun. ACM* **18**(9), 509–517.
- Cardinal, J. (1999) Faster fractal image coding using similarity search in a KL-transformed feature space. In *Fractals: Theory and Applications in Engineering*, Dekking, M. (eds). New York, NY: Springer, pp. 293–306.
- Curtis, S. A. & Martin, C. E. (2005) Functional fractal image compression. In *Proceedings of the 6th Symposium on Trends in Functional Programming (TFP 2005)*, Tallinn, Estonia, pp. 393–398.
- Davoine, J. (1997) How to improve pixel-based fractal image coding with adaptive partitions. In *Fractals in Engineering: From Theory to Industrial Applications*, Lévy Véhel, J., Lutton, E. and Tricot, C. (eds). Berlin, Germany: Springer-Verlag, pp. 292–306.
- Elliot, C. (2003) Functional images. In *The Fun of Programming*, Gibbons, J. & de Moor, O. (eds). Sydney Australia: Palgrave Macmillan, pp. 131–150.
- Fisher, Y. (1995) *Fractal Image Compression. Theory and Application*. Nerw York, NY: Springer.
- Fractal Foundation. (2011) Accessed March 14, 2013. Available at: <http://www.fractalfoundation.org>.

- Frame, M. & Mandelbrot, B. B. (2002) *Fractals, Graphics, and Mathematics Education*, Mathematical Association of America Notes, vol. 58. Cambridge, UK: Cambridge University Press.
- Ghosh, S. K., Mukhopadhyay, J., Chowdary, V. M. & Jeyaram, A. (2002) Relative fractal coding and its application in satellite image compression. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP)*, India.
- GIMP (2012) *The GNU Image Manipulation Program* (version 2.8.2). Accessed October 23, 2013. Available at: <http://www.gimp.org>.
- Hafner, U. (2000). Fiasco. Accessed February 21, 2013. Available at: <http://github.com/megatherion/Fiasco/>.
- Hudak, P. (2000) *The Haskell School of Expression*. Cambridge, UK: Cambridge University Press.
- Hugs (2006) *The Hugs 98 System*. Available at: <http://haskell.org/hugs/>.
- Hutton, G. (2007) *Programming in Haskell*. Cambridge, UK: Cambridge University Press.
- Jacquin, A. (1989) *A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding*. PhD thesis, Georgia Institute of Technology, Atlanta, GA.
- Jones, M. P. (2004) Composing Fractals. *J. Funct. Program.* **14**(6), 715–725.
- Kanakarakis, I., Ntanasis, P. & Sarbinowski, P. (2011) Fractal image compression. Accessed March 26, 2013. Available at: <http://github.com/c00kiemon5ter/Fractal-Image-Compression>
- Kaplan, K. (1997) Fractals are emerging as a shape of things to come. *LA Times* May 12, 1997. Accessed October 23, 2013. Available at: http://articles.latimes.com/1997-05-12/business/fi-58093_1_fractal-compression.
- Lindenmayer, A. (1968) Mathematical models for cellular interaction in development, Parts I and II. *J. Theor. Biol.* **18**, 280–315.
- Liu, D. & Jimack, P. K. (2007) A survey of parallel algorithms for fractal image compression. *J. Algorithms Comput. Technol.* **1**, 171–186.
- Lu, N. (1997) *Fractal Imaging*. Waltham, MA: Academic Press.
- Mandelbrot, B. B. (1977) *Fractals, Form, Chance and Dimension*. New York, NY: W. H. Freeman.
- Mandelbrot, B. B. (1983) *The Fractal Geometry of Nature*. New York, NY: W. H. Freeman.
- Microsoft (2009) Microsoft encarta. Accessed March 21, 2013. Available at: <http://microsoft.com/uk/encarta/>
- Munroe, R. (2006) Su Doku. *xkcd*. Accessed October 23, 2013. Available at: <http://xkcd.com/74/>
- Olson, C. F. (2006) Encouraging the development of undergraduate researchers in computer vision. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. New York, NY: ACM, pp. 255–259.
- OnOne Software (2013) Perfect resize. Accessed February 21, 2013. Available at: <http://www.ononesoftware.com/products/perfect-resize/>
- Peitgen, H.-O., Jürgens, H. & Saupe, D. (1991) *Fractals for the Classroom, Part One*. New York, NY: Springer-Verlag.
- Peitgen, H.-O., Jürgens, H. & Saupe, D. (1992) *Fractals for the Classroom, Part Two*. New York, NY: Springer-Verlag.
- Peyton Jones, S. (ed) (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge, UK: Cambridge University Press.
- Rasala, R. (2000, March) Toolkits in first year computer science: A pedagogical imperative. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, 2000, (SIGCSE '00)* Austin, Texas, Vol. 32 Issue 1. New York, NY: ACM, pp. 185–191.

- Saupe, D. (1994) *Breaking the Time Complexity of Fractal Image Compression*. Technical Report 53, Institut für Informatik Freiburg, Freiburg, Germany.
- Skiljan, I. (2012) *IrfanView* (version 4.33). Accessed October 23, 2013. Available at: <http://www.irfanview.com>
- Stanford (2013) Nifty assignments. Accessed February 21, 2013 Available at: <http://nifty.stanford.edu/>
- Still, M. (2005) *The Definitive Guide to ImageMagick*. New York, NY: Apress.
- Thalabard, S. & Zahariade, G. (2005) *Compression Fractale d'Images*. Lyon, France: TIPE, ENS.
- Thompson, S. (2011) *The Craft of Functional Programming*, 3rd ed. Boston, MA: Addison-Wesley.
- Ullrich, H. (1999) *Low Bit-rate Image and Video Coding with Weighted Finite Automata*. Berlin, Germany: Mensch & Buch.
- Vaddella, V. R. P. & Inampudi, R. B. (2010) Fast fractal compression of satellite and medical images based on domain-range entropy. *J. Appl. Comput. Sci. Math.* **9**(4), 21–26.
- Veenadevi, S. V. & Ananth, A. G. (2011) Fractal image compression of satellite imageries. *Int. J. Comput. Appl.* **30**(3), 33–36.
- Welstead, S. (1999) *Fractal and Wavelet Image Compression Techniques*. Bellingham, WA: SPIE Press.
- Wertheim, M. (2006) *Field Guide to the Business Card Menger Sponge*. Los Angeles, CA: The Institute for Figuring.