Fractals, Bitmaps, and APL

Jeffrey Shallit

ABSTRACT

In this article, we show how to generate approximations to fractals and fractal-like patterns using operations on boolean matrices. The pictures may be displayed on a bitmapped device like the Apple Macintosh.

The methods illustrate the beauty and power of APL, as well as the suggestiveness of the notation.

1. The Sierpiński carpet.

The Sierpiński carpet is a fractal in the plane which is the limit of the sequence of pictures in Figure 1.

In the limit, this carpet covers zero area. The "removal of middle thirds" used to generate it is reminiscent of the method for constructing the Cantor set [Man, p. 144].

We can represent the sets in Figure 1 in APL as square boolean matrices, where 1 represents a black square, or *pixel*, and 0 represents a white pixel. For example, the first three pictures in Figure 1 could be represented by the three arrays

```
  1 1 1
1 , 1 0 1 ,
  1 1 1

1 1 1 1 1 1 1 1 1
1 0 1 1 0 1 1 0 1
1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1
1 0 1 0 0 0 1 1 1
1 1 1 0 0 0 1 1 1
1 1 1 1 1 1 1 1 1
1 0 1 1 0 1 1 0 1
1 1 1 1 1 1 1 1 1
```

Such matrices represent bitmap images which can be attractively displayed on a raster device like the Apple Macintosh.[1] STSC's APL*PLUS[2] on the Mac supports display of bitmaps with their built-in system function ⎕*PUTBITS*. The left argument to ⎕*PUTBITS* is a boolean matrix representing an image; the right argument is a vector of length 2 or 4. If the right argument is of length 2, then it specifies the screen coordinates of the upper left-hand-corner of the image. If it is of length 4, then it specifies the coordinates of both the upper-left and lower-right-hand corner of a rectangle which is the target of the image. The image is automatically shrunk or expanded to fit this rectangle.

[1] Macintosh is a trademark of Apple computers.

[2] APL*PLUS is a trademark of STSC. It may be ordered from them at 2115 E. Jefferson St., Rockville, MD 20852. Phone: (301) 984-5000.

How can we efficiently generate these boolean matrices in APL? In the case of the Sierpinski carpet, we could use the following recursive solution, due to David Salzman:

```
CARPET0:  (T,T,T),[0](T,((⍴T)⍴0),T),[0]
          T,T,T←CARPET0 ⍵-1 : ⍵=0 : 1 1⍴1
```

(Note: this function, like all others in this article, assumes origin-0.)

This direct definition[3] function takes as its right argument a non-negative integer *N* indicating the "generation number" of the result. The result is a matrix of dimension `2⍴3*N` giving the bitmap image of the *N*-th generation of the carpet. We can generate a 243×243 matrix in only 14 seconds on a Macintosh Plus! (See Figure 2.)

We can obtain a different solution to the carpet-generation problem by noting that subsequent generations are a sort of "Kronecker product" of matrices:

$$\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix} \otimes \begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix} = \begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}$$
$$\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}$$
$$\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}$$

We can easily perform the Kronecker product in APL using outer product, as in the following function:

```
KAND:  ((⍴⍺)×⍴⍵)⍴ 0 2 1 3 ⌹ ⍺ ∘.∧⍵
```

Now we can define a function for the *iterated* Kronecker product:

```
IKAND: ⍺ KAND ⍺ IKAND ⍵-1: ⍵=0: 1 1⍴1
```

The left argument to *IKAND* is a "generator matrix" *M*. The result is the iterated Kronecker product performed *N* times, where *N* is the right argument. Now we can easily write a new carpet function:

```
CARPET1: (3 3⍴ 4≠⍳9) IKAND ⍵
```

CARPET1 exhibits several advantages over *CARPET0*.[4] For one thing, it suggests using matrices other than `3 3⍴4≠⍳9` as generators. Further, it suggests replacing the ∧ function with one of the other logical functions. Let us take a moment to pursue the first of these ideas, postponing the

[3] In this article we freely use the "direct definition" form of functions. For an interpreter that converts these definitions to ordinary APL functions, see [May] or [Ive].

[4] Speed, unfortunately, does not seem to be one of them. *CARPET0* executes faster than *CARPET1* on all of the systems tried.

second until later. If we use `(2 2⍴ 1 1 1 0)` as the left argument to *IKAND*, then we get a repeating triangle pattern which is essentially Pascal's triangle considered modulo 2 (see [Goe]). This image appears in Figure 3.

The reader is encouraged to do some experimenting at this point. Many other interesting patterns can be generated. For example, using the 7×7 matrices

```
1 1 1 1 1 1 1      1 1 1 1 1 1 1
1 1 1 1 1 1 1      1 0 1 0 1 0 1
1 1 0 0 0 1 1      1 1 1 1 1 1 1
1 1 0 0 0 1 1  or  1 0 1 0 1 0 1
1 1 0 0 0 1 1      1 1 1 1 1 1 1
1 1 1 1 1 1 1      1 0 1 0 1 0 1
1 1 1 1 1 1 1      1 1 1 1 1 1 1
```

we can generate the pictures which appear on page 318 of [Man].

Let us return to the function *IKAND*. Notice that in *KAND* we reshape and transpose, and this function is called repeatedly by *IKAND*. This seems inefficient--could we somehow group all the dyadic transposes together and perform them once, at the end?

The answer is yes. We use the function *IOPAND* to perform an "iterated outer product" which results in a (2×*N*) dimensional array. This array is then transposed using the "perfect shuffle" permutation `⍋⍋(2×N)⍴ 0 1`.

```
IOPAND:  (⍺ IOPAND ⍵-1)∘.∧⍺ : ⍵=0: 1

IK2AND:  ((⍴⍺)*⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                              ⍉⍺ IOPAND ⍵

CARPET2: (3 3⍴4≠⍳9)IK2AND ⍵
```

We could dispense entirely with the recursion in *IOPAND* at this point by (a) using an APL interpreter that has a "power operator" or (b) using an APL interpreter that allows reduction with the outer product operator or (c) "kludging" the iterated outer product using ⍎.

For example, Roger Hui has provided the following function using option (a):

```
CARPET3: ((⍴T)*⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                    ⍉T∘.∧.⍵ T←3 3⍴4≠⍳9
```

Unfortunately, the versions of APL currently available on the Mac do not allow such pyrotechnics... but we can hope!

Another non-recursive, non-iterative version of *CARPET* can be defined using the *strong relationship* between the Sierpiński carpet and *expansion in base-3*. Using this hint, can the reader come up with such a function? (One solution is given at the end of this article.)

2. Iterated Kronecker products with other functions.

Let us return to the function *IK2AND* and *CARPET2*. Their definitions strongly suggest replacing the ∧ function with one of the other boolean functions. Using this idea, many delightful patterns can be obtained.

(What we really want, of course, is a user-defined *operator* that performs this iterated Kronecker product, given a function as an argument. But such facilities are not presently available on the Macintosh.)

For example, we could define the following functions:

```
IOPXOR: (⍺ IOPXOR ⍵-1)∘.≠⍺: ⍵=0: 0

IKXOR:  ((⍴⍺)*⍵)⍴(⍋⍋(2×⍵)⍴0 1)
                              ⍉⍺ IOPXOR ⍵
```

The left argument to *IKXOR* is a generator matrix. The right argument is a generation number.

For example, see Figure 4.

The matrices

```
(2 2⍴ 0 0 0 1) IKXOR N
```

are interesting because of their relationship to combinatorial objects called *Hadamard matrices* (see [Rys]). Define

```
HAD: ¯1 * (2 2⍴ 0 0 0 1) IKXOR ⍵
```

Then *HAD N* is a matrix of dimension `2⍴2*N` such that

```
(HAD N)+.×⍉HAD N ↔ (2*N)×ID 2*N
```

where

```
ID: (⍳⍵)∘.=⍳⍵
```

is the identity matrix function.

Figure 5 shows the result of *IKXOR* with a different generator.

Still wilder patterns can be obtained by using the boolean functions *NAND* and *NOR*. Since these functions, unlike ∧ and ≠, are not associative, we must define an order for their associated Kronecker products: either right-to-left, or left-to-right. For example:

```
IOPNORLR: (⍺ IOPNORLR ⍵-1)∘.⍱⍺: ⍵=1: ⍺
IKNORLR:  ((⍴⍺)*⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                          ⍉ ⍺ IOPNORLR ⍵

IOPNORRL: ⍺∘.⍱⍺ IOPNORRL ⍵-1: ⍵=1: ⍺
IKNORRL:  ((⍴⍺)*⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                          ⍉ ⍺ IOPNORRL ⍵
```

Many other patterns can be obtained using different generators and the relational functions <, ≤, >, and ≥. The author has explored only a small subset of this space of patterns.

3. Iterated morphisms.

There is still another way to generate the Sierpiński carpet, and it generalizes in a different direction. We define a map, ϕ, which takes 0 to the 3×3 matrix of symbols

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and 1 to the matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Then by iterating ϕ we get the successive generations of the carpet.[5]

More generally, we could take an alphabet of A symbols; say 0,1,2,...,$A-1$. We define ϕ by providing an array R of dimension (A, M, M) such that $R[0;;]$ gives the value of $\phi(0)$; $R[1;;]$ gives the value of $\phi(1)$, etc. This idea leads to the following functions:

```
APPLY: ((⍴⍵)×¯2+⍴⍺)⍴ 0 2 1 3 ⌹⍺[⍵;;]
IM:    ⍺ APPLY ⍺ IM ⍵-1: ⍵=0: 1 1 ⍴ 1
```

The left argument to *APPLY* is an array of dimension (A, M, M) of integers in ιA that specifies the map ϕ. (Remember: 0-origin.) The right argument is a matrix of values B, and the result is $\phi(B)$.

The left argument to *IM* (iterated map) is an array defining ϕ. The right argument is a generation number N. The result is $\phi(\phi(\ldots\phi(1)))=\phi^N(1)$.

This gives us still another carpet function:

```
CARPET4: (0,[¯0.5] 3 3⍴ 4≠⍳9) IM ⍵
```

Many pleasant examples can be constructed with the function *MORPHISM*. See Figures 8 and 9.

It is perhaps somewhat surprising that we can even generate some space-filling curves using this scheme. In this article we give just one such example. The curious reader can find two more in [Sha].

The function *PCM* below is an encoding of a particular map ϕ. *SFC* (space-filling curve) applies this map repeatedly and then applies a certain coding given by the function *CODE*. The argument to *SFC* is a generation number N. The result is a $(3*N)\times(3*N)$ matrix giving an image of the N-th generation of the space-filling curve of Peano [Pea].

To understand this function it may be useful to realize that *PCM* has been defined to map small sections of the curve to larger sections of the curve. For example,

```
CODE [PCM APPLY 1 1⍴5]
0 0 0
1 1 0
0 1 0

CODE [PCM APPLY PCM APPLY 1 1⍴5]
0 0 0 0 0 0 0 0 0
1 1 0 0 1 1 1 1 0
0 1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 1 0
0 1 1 1 1 0 0 1 0
0 0 0 0 0 0 0 1 0

PCM: 15 3 3⍴⍳(9⍴15)⊤ 393331468
  393331446    5842325861   5842325861
  10645653343 10645653343   33881768946
  6070134986   6070134986   3388177232⍺
  33881772324  6070895246   6070895246
  33881768968   33881768968

CODE: 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1

SFC: CODE [PCM IM ⍵]
```

4. Parting words.

We have only touched the surface in this article. We hope the reader will be encouraged and enthusiastic enough to explore

(a) morphisms on bigger alphabets;

(b) morphisms on higher-dimensional results;

(c) pictures on *color* bitmap displays (see (a) above);

(d) the Menger sponge (3-dimensional generalization of the Sierpinski carpet), etc.

5. Acknowledgments.

Roger Hui and Norman Thomson read a preliminary version of this manuscript and made many useful suggestions.

Thanks to Howard Karloff for pointing out the connection with Hadamard matrices.

6. Solution to the problem posed at the end of section 1.

```
CARPET5: (⌹T)∧.∨T←1≠(⍵⍴3)⊤⍳3*⍵
```

Summary of Functions.

```
CARPET0:  (T,T,T),[0](T,((⍴T)⍴0)T,T,T←
                    CARPET0 ⍵-1: ⍵=0: 1 1⍴1

KAND:     ((⍴⍺)×⍴⍵)⍴ 0 2 1 3 ⌹ ⍺ ∘ .∧⍵
        ⍝ KRONECKER PRODUCT WITH ∧ FUNCTION

IKAND:    ⍺ KAND ⍺ IKAND ⍵-1: ⍵=0: 1 1⍴1
```

---

[5] Iterations of *one*-dimensional maps (as opposed to the two-dimensional ones we consider here) have well-known relationships to fractals and fractal-like patterns. See the beautiful article of Dekking, Mendes France, and van der Poorten for many examples [DMP].

Iterations of *two*-dimensional maps (and, more generally, N dimensions) are less well understood. See the recent article of Salon [Sal].

```
          ⍝ ITERATED OUTER PRODUCT WITH ∧

CARPET1:  (3 3⍴ 4≠⍳9) IKAND ⍵

IOPAND:   (⍺ IOPAND ⍵-1)∘.∧⍺ : ⍵=0: 1
          ⍝ ITERATED OUTER PRODUCT WITH ∧

IK2AND:   ((⍴⍺)⍴⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                              ⍉⍺ IOPAND ⍵

CARPET2:  (3 3⍴4≠⍳9)IK2AND ⍵

IOPXOR:   (⍺ IOPXOR ⍵-1)∘.≠⍺: ⍵=0: 0
          ⍝ ITERATED OUTER PRODUCT WITH ≠



IKXOR:    ((⍴⍺)⍴⍵)⍴(⍋⍋(2×⍵)⍴0 1)
                              ⍉⍺ IOPXOR ⍵

HAD:      ¯1 * (2 2⍴ 0 0 0 1) IKXOR ⍵
          ⍝ HADAMARD MATRIX OF SIZE 2*⍵

ID:       (⍳⍵)∘.=⍳⍵

IOPNORLR: (⍺ IOPNORLR ⍵-1)∘.⍱⍺: ⍵=1: ⍺

IKNORLR:  ((⍴⍺)⍴⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                              ⍉ ⍺ IOPNORLR ⍵

IOPNORRL: ⍺∘.⍱⍵ IOPNORRL ⍵-1: ⍵=1: ⍺

IKNORRL:  ((⍴⍺)⍴⍵)⍴(⍋⍋(2×⍵)⍴ 0 1)
                              ⍉ ⍺ IOPNORRL ⍵

APPLY:    ((⍴⍵)×¯2↑⍴⍺)⍴ 0 2 1 3 ⍉⍺[⍵;;]
          ⍝ APPLY MAP TO MATRIX RIGHT ARG

IM:       ⍺ APPLY ⍺ IM ⍵-1: ⍵=0: 1 1 ⍴ 1
          ⍝ ITERATED MAP APPLICATION

CARPET4:  (0,[¯0.5] 3 3⍴ 4≠⍳9) IM ⍵

PCM:      15 3 3⍴⍉(9⍴15)⊤ 393331468
  393331446    5842325861   5842325861   10645653343
10645653343   33881768946   6070134986   6070134986
33881772324   33881772324   6070895246   6070895246
33881768968   33881768968

CODE:     0 1 0 1 0 1 0 0 1 0 1 0 1 0 1

SFC:      CODE [PCM IM ⍵]

CARPET5:  (⍉T)∧.∨T←1≠(⍵⍴3)⊤⍳3*⍵
```

7. References.

[DMP] Dekking, M., Mendes France, M. and van der Poorten, A., *Folds!*, Mathematical Intelligencer 4 (1982) 130-138; 173-195.

[Goe] Goetgheluck, P., *Computing binomial coefficients*, Amer. Math. Monthly 94 (1987) 360-365.

[Ive] Iverson, K.E., *Programming style in APL*, in An APL User's Meeting, I. P. Sharp Associates, 1978, pp. 200-224.

[Man] Mandelbrot, B., *The fractal geometry of nature*, W. H. Freeman, 1983.

[May] Mayforth, R., *An alpha-omega compiler*, APL News #3 (1977).

[Pea] Peano, G., *Sur une courbe, qui remplit toute une aire plane*, Math. Annalen 36 (1890) 157-160.

[Rys] Ryser, H., *Combinatorial mathematics*, Math. Association of America, 1963.

[Sal] Salon, O., *Suites automatiques a multi-indices*, Seminaire de theorie des nombres de Bordeaux, Expose #4, 1986-7.

[Sha] Shallit, J., *Two methods for the generation of fractal images*, University of Chicago Department of Computer Science Technical Report 87-010 (June, 1987).

Department of Computer Science
University of Chicago
1100 E. 58th St.
Chicago, IL 60637
USA

(Figures 1 to 10 appear on the next 5 pages)

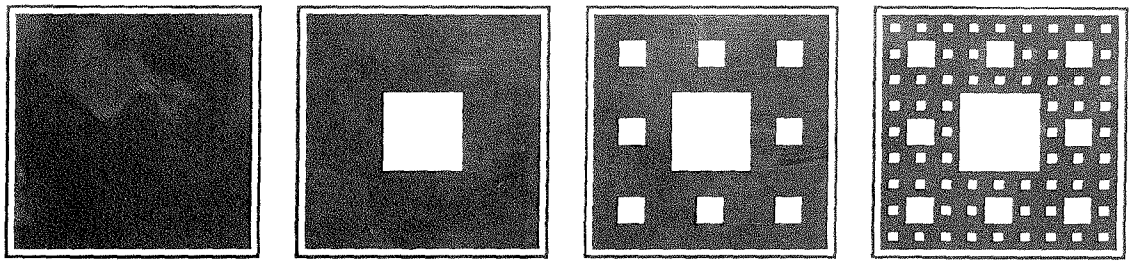

Figure 1

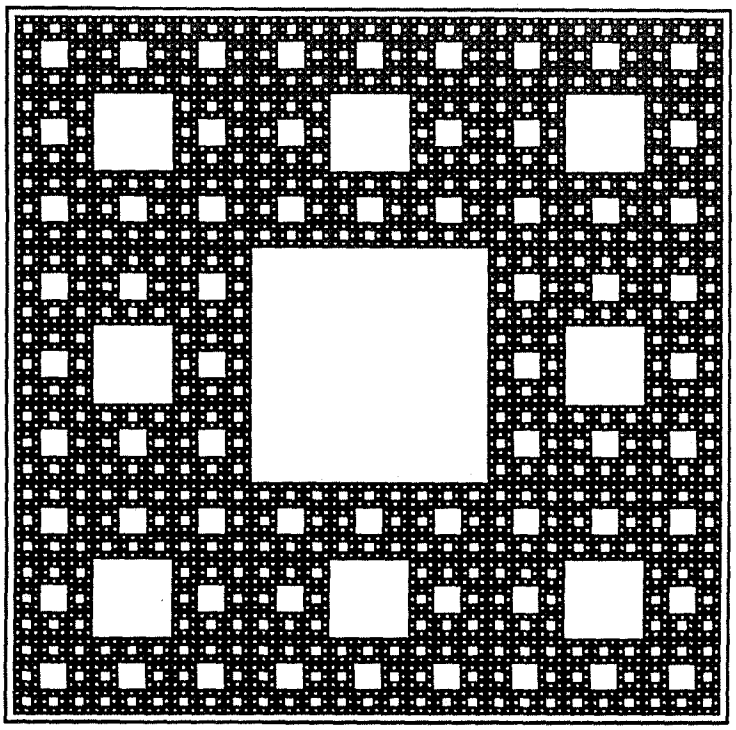

Figure 2: *CARPET*0 5

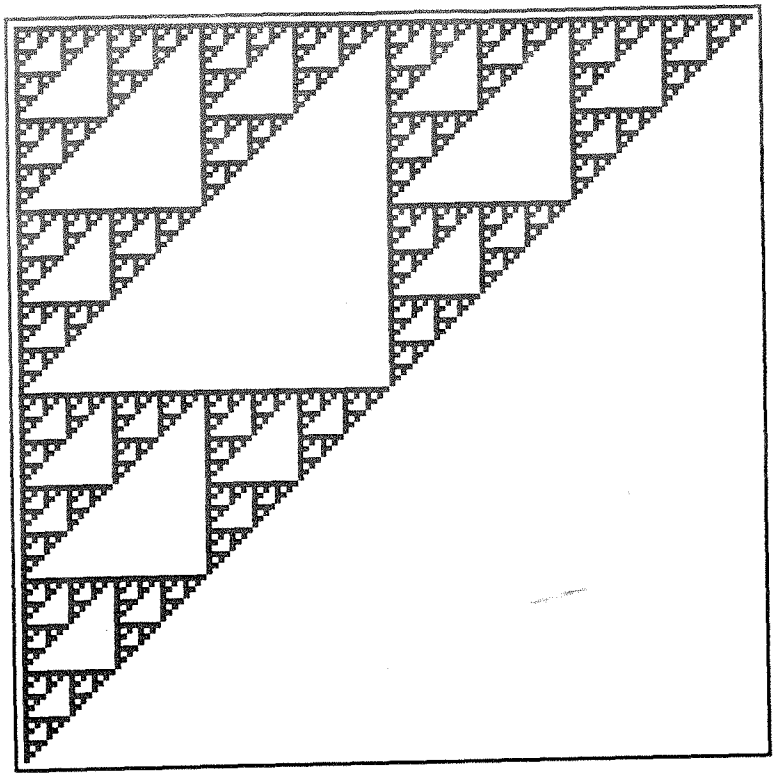

Figure 3: `(2 2⍴ 1 1 1 0) IKAND 7`

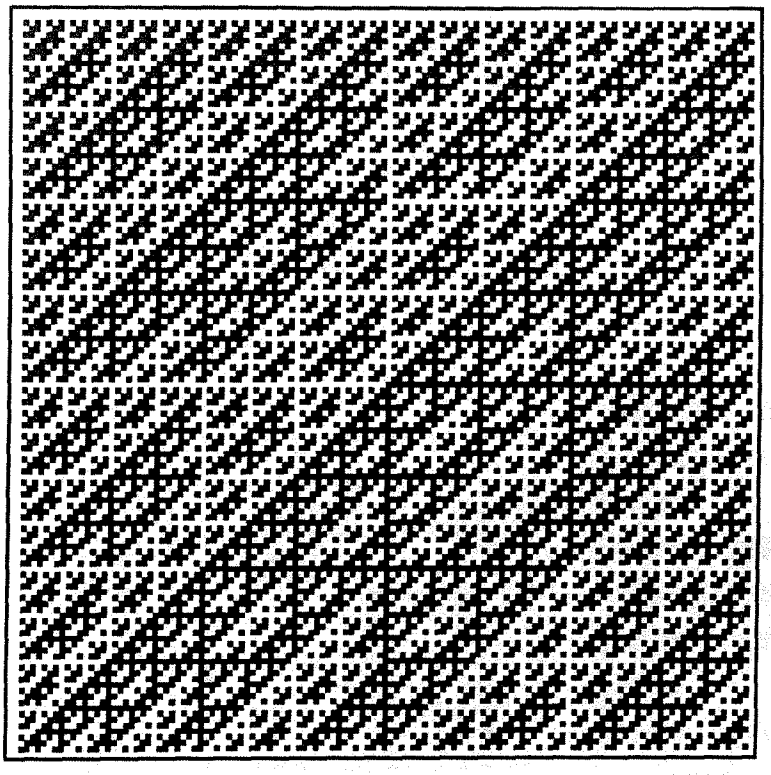

Figure 4: `(2 2⍴ 0 0 0 1)IKXOR 7`

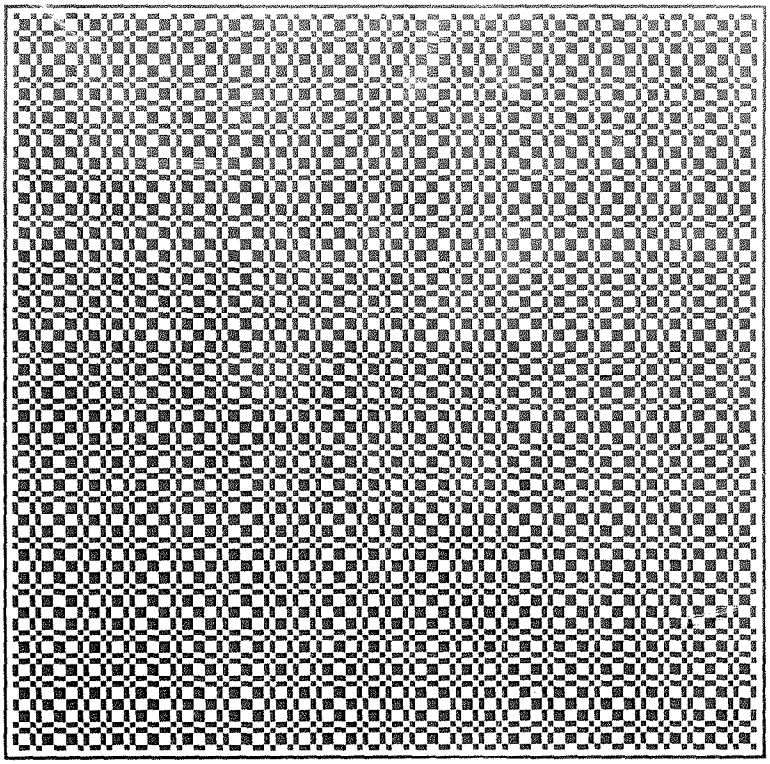

Figure 5: `(2 2⍴ 1 0 0 1)` *IKXOR* `7`

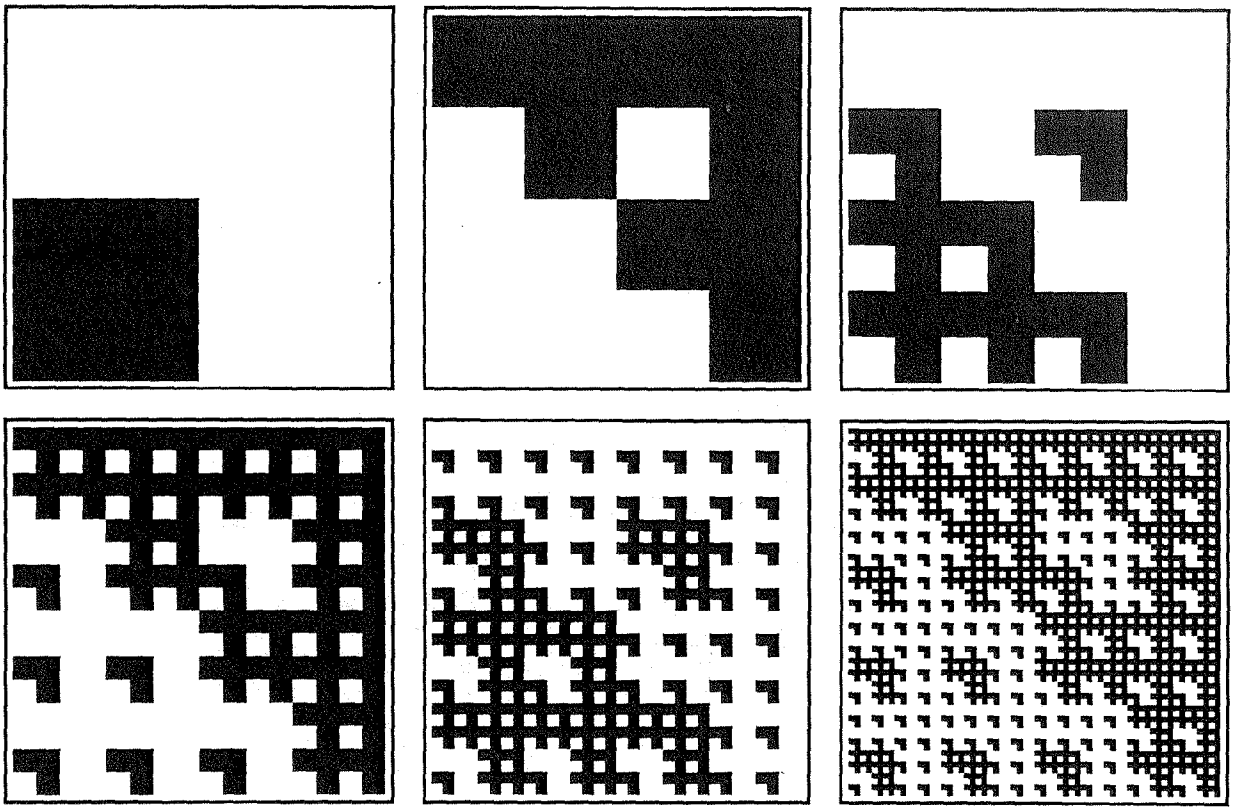

Figure 6: `(2 2⍴ 0 0 1 0)`*IKNORLR N*
for *N*=1,2,3,4,5,6

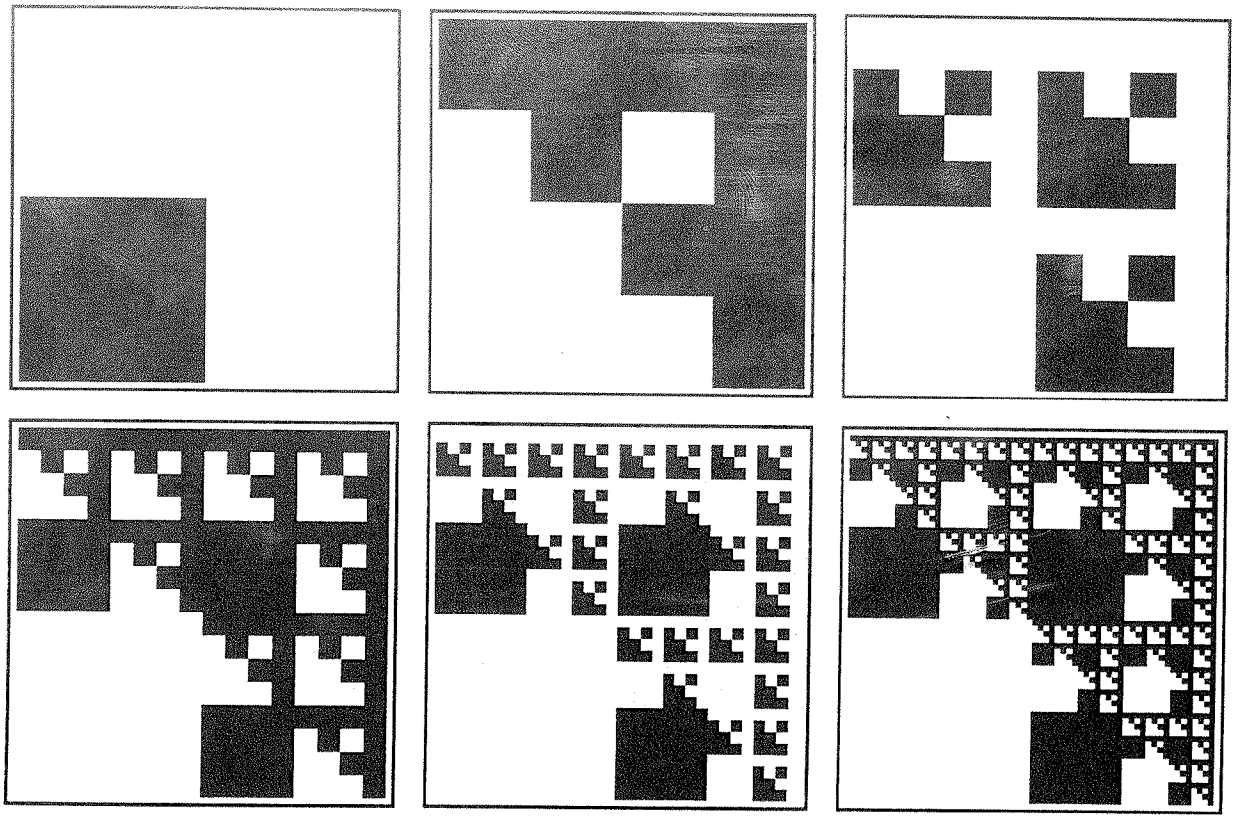

Figure 7: `(2 2⍴ 0 0 1 0) IKNORRL N`
for `N=1,2,3,4,5,6`

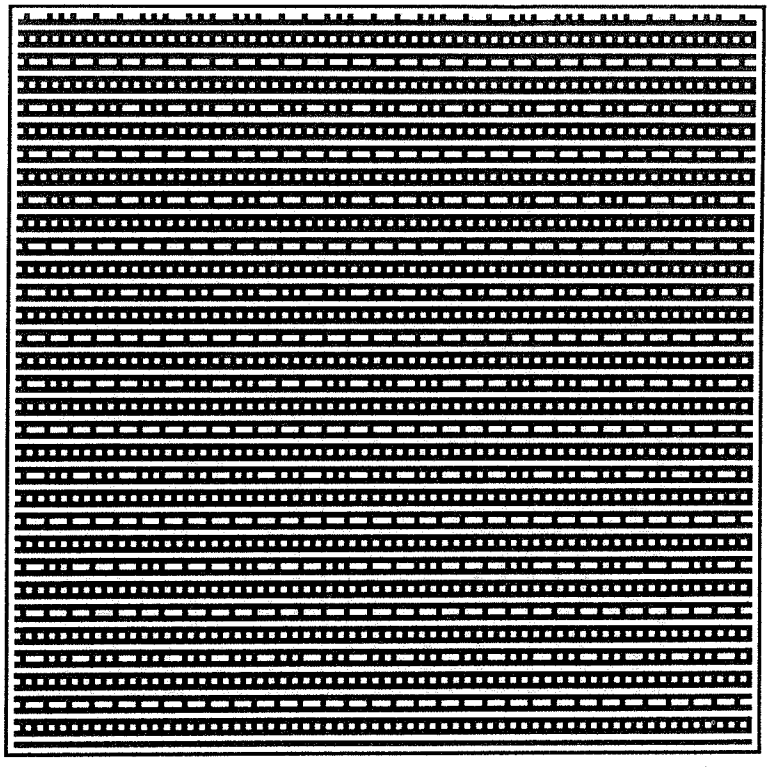

Figure 8: `(2 2 2⍴ 0 1 1 1 0 0 1 1) IM 7`

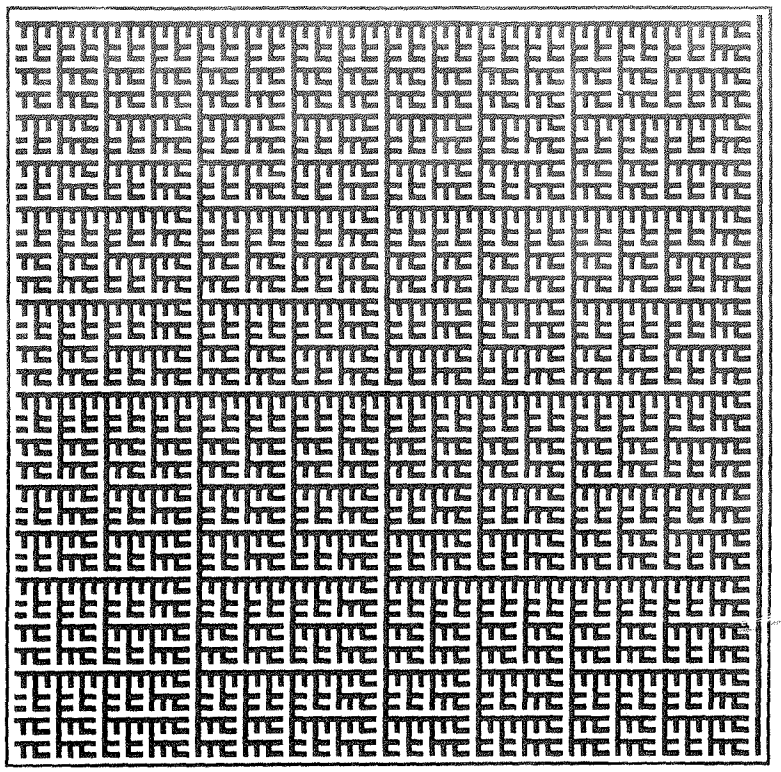

Figure 9: `(2 2 2⍴ 0 0 1 1 0 1 0 1) IM 7`

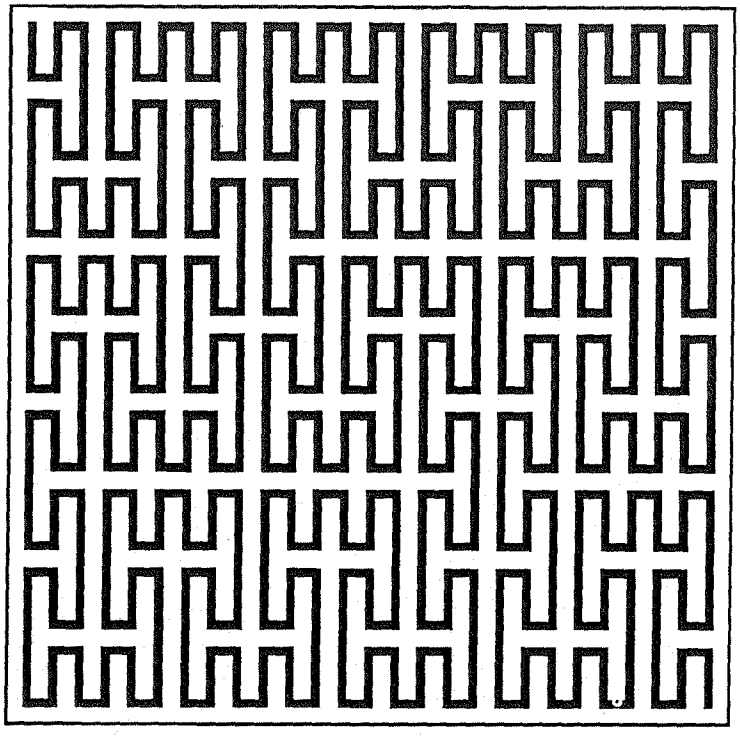

Figure 10: `SFC 4`