- *MATHEMATICS AND COMPUTER EDUCATION* -
Vol. 30, No. 1, Winter 1996

RECURSIVE ALGORITHMS FOR IFS FRACTALS

by Richard D. Neidinger
Davidson College
Davidson, North Carolina

In the world of algorithms for generating fractal graphics, one of the most mathematically satisfying methods remains relatively obscure or unknown. This method uses recursion to generate an image in the sequence of images converging to the fractal. These sequences of images are found throughout the literature and an example is shown in Figure 1. The convergence of such a sequence is a beautiful application of a contraction mapping in a metric space. Still, the usual textbook computer implementations either avoid the sequence (random iteration algorithm or chaos game) or are impractical because of memory requirements (an array entry for each pixel or string entry for each turtle command).

Four recursive algorithms are presented, one in True BASIC and the remaining in pseudo-code that can be adapted to any structured language. The Recursive Drawing Algorithm is an amazingly simple True BASIC program that uses unique language features to directly implement the theoretical definitions. More generic techniques are found in the Map Sequence Algorithm that highlights the role of the recursion by separating it from the calculations and graphics. The Point Sequence and Matrix Sequence Algorithms build on this understanding to produce more efficient programs. In comparing these algorithms, with each other and with alternatives to the recursive approach, we find practical limitations and significant advantages for each of them. Which algorithm is best depends on your goal. Some algorithms allow control of the drawing process (the order in which parts of the image appear) and the coloring pattern of the image.

NOTATION AND BACKGROUND

We seek algorithms that work for a large class of fractal images described by different data codes. One such scheme is Barnsley's very successful IFS description of fractals as a set of contractions. The fractal in Figure 1, $A(7)$, is described by three contraction maps, W_1, W_2, and W_3, that can be seen as the functions mapping $A(0)$ to each of the three images in $A(1)$. In general, an *iterated function system* (IFS) is a

collection of contraction *maps* $W_k: \mathbb{R}^2 \to \mathbb{R}^2$ for $k = 1, 2, \ldots, n$ [1, p. 82]. Let $A(0)$ be any (compact, nonempty) set in the plane. Define

$$A(i) = W_1\big(A(i-1)\big) \cup W_2\big(A(i-1)\big) \cup \cdots \cup W_n\big(A(i-1)\big).$$

The *Fractal Attractor Theorem* states that there exists a limit $A = \lim_{i \to \infty} A(i)$ and, moreover, this limit is the unique set such that $A = W_1(A) \cup W_2(A) \cup \cdots \cup W_n(A)$. Thus, A does not depend on $A(0)$ at all, only on the maps. Many students never cease to be amazed that $A(0)$ can be a single point, a solid screen, or even a stick-man as in Figure 1, and still the same limit set occurs. This is proved, in a metric space of sets, by simply applying the fixed point theorem for contraction mappings in any metric space [2, p. 85]. This type of lively application of abstract analysis makes such material enjoyable for both students and the instructor. Using recursion, the sets $A(i)$ can be displayed on the computer in a way that implements the above definition and demonstrates the Fractal Attractor Theorem.

Most IFS programs, software, and textbook discussions use the Random Iteration Algorithm that does not actually display any set $A(i)$ in the sequence but approximates the limiting set A. The *Random Iteration Algorithm* [1, p. 91] plots a sequence of points given by $P_i = W_{r(i)}(P_{i-1})$ where each $r(i)$ is one map index chosen randomly from $\{1, 2, \ldots, n\}$. If $A(0) = P_0$, then each P_i is one point from the set $A(i)$. This produces an image that materializes, as if on a Star Trek transporter pad. It is easy to program and generates a high-quality image of the fractal in reasonable time.

A *deterministic algorithm* is the usual term for any algorithm that shows the sets $A(i)$ as opposed to the "random" iteration algorithm. If $A(0)$ is a single point, then the difficulty of this task is betrayed by the fact that the set $A(i)$ contains n^i points! Still, as explained above, it is instructive to show different $A(0)$ sets and different levels of $A(i)$, even if we are restricted to small i (such practical restrictions are explored later). To view the "limit," switch to the Random Iteration Algorithm. A deterministic algorithm also allows images from the sequence to be superimposed, an interesting effect that Barnsley calls condensation [1, Section 3.9]. Some commercial fractal software packages have both deterministic and random options. Personally, I like to show the

programming of both, and related issues, along with the analysis. This combination of analysis, programming, and applied linear algebra for the transformations makes for an ideal topic in a "capstone" mathematical sciences course.

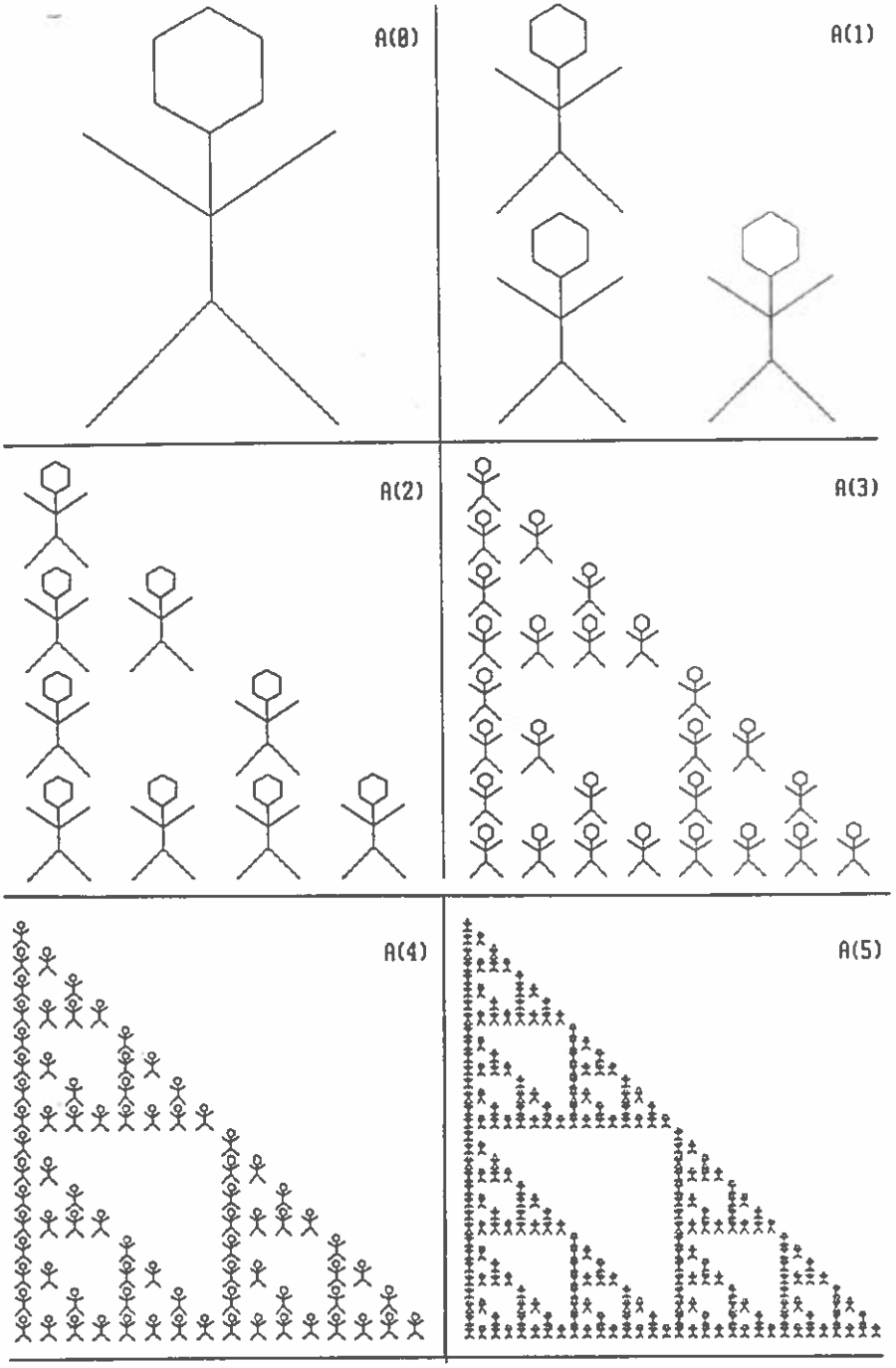

Figure 1 (continued on next page)

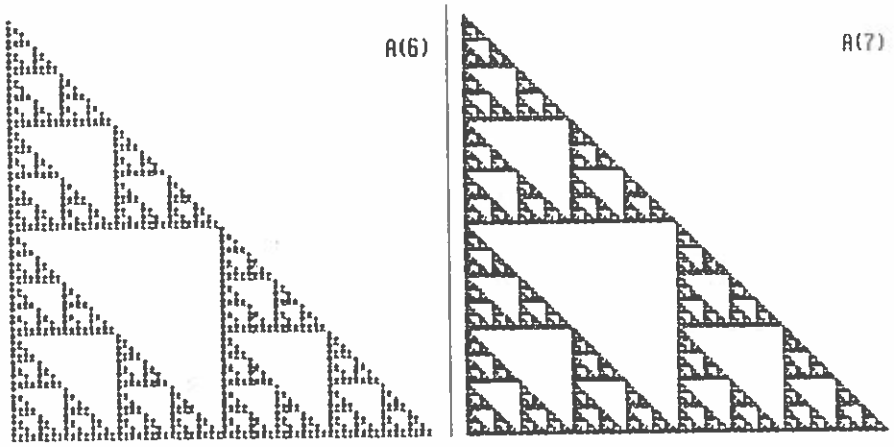

Figure 1

THE RECURSIVE DRAWING ALGORITHM

True BASIC enables a very simple and naive implementation of the recursive definition $A(i) = \bigcup_{k=1}^{n} W_k\big(A(i-1)\big)$. If a procedure (subroutine) has graphic output, then "call procedure with W" will transform whatever the procedure would have displayed by the geometric transformation W. Actually, the True BASIC syntax uses PICTURE (instead of PROCEDURE or SUB) to introduce a procedure with graphic output, and DRAW (instead of CALL) to call it. We recursively define PICTURE A(i) by calling "DRAW A(i-1) WITH W ", for each map W in the IFS. The transformation W is given by a transformation matrix (in the form described in [4]) or a combination of the supplied transformations SCALE, SHIFT, ROTATE, and SHEAR. In particular, SCALE(r,r)*SHIFT(sx,sy) will shrink (assuming $r<1$) points toward the origin by factor r, add sx to x-coordinates and add sy to y-coordinates.

A simple example program draws the image in Figure 2 without the shading. (Shading or coloring will be discussed in the next section.) This image is the set $A(7)$ where $A(0)$ is a right triangle with negative-slope hypotenuse and the IFS converges to a Sierpinski triangle with positive-slope hypotenuse. The following program is complete, ready to run on any Mac or DOS machine with True BASIC. An exclamation point indicates that the remainder of the line is comment only. Using default coordinates, the screen is the unit square in the first quadrant.

```
! Recursive Drawing Algorithm in True BASIC
PICTURE A(i)
    IF i = 0 THEN
        PLOT 0,0; 1,0; 0,1; 0,0
    ELSE
        DRAW A(i-1) WITH SCALE(.5, .5)
        DRAW A(i-1) WITH SCALE(.5, .5) * SHIFT(.5, 0)
        DRAW A(i-1) WITH SCALE(.5, .5) * SHIFT(.5, .5)
    END IF
END PICTURE
! beginning of main program
DRAW A(7)
END
```

This simple program is easily adapted to any $A(0)$ and any IFS. In fact, the case $i = 0$ can use any standard graphics command, such as flooding regions with color. This makes $A(0)$ more flexible than in the following sections where $A(0)$ is a one-color stick-figure. The general IFS is accomplished with a FOR loop that calls A(i-1) with each transformation matrix.

The important and unique feature is that this transforms a procedure and not an array. True BASIC has a transparent and fairly efficient way of internally storing the necessary information from one recursive level to the next. To implement a version of this idea in most languages would require significant memory hassles in saving the picture $A(i-1)$ in a huge array. Such algorithms are discussed later in the section "The Pixel Array Algorithm and L-systems". We now pursue a language-independent algorithm that avoids these huge arrays. Surprisingly, a True BASIC program that implements the pseudo-code of the next section is about twice as fast as the above.

THE MAP SEQUENCE ALGORITHM

The key to the following recursive algorithms is to abandon the idea of finishing $A(i-1)$ before starting $A(i)$. Instead, consider $A(m)$ to be the union of many copies of $A(0)$, each determined by a sequence of transformation maps. Formally, $A(m) = \bigcup W_{k_m} W_{k_{m-1}} \cdots W_{k_2} W_{k_1}(A(0))$ where the union is over all m-tuples $[k_1, k_2, \ldots, k_m]$ where each $k_i \in \{1, 2, \ldots, n\}$. We call such an m-tuple a *map sequence* (of length m).

Consider the example in Figure 1 where $n = 3$ (three maps in the IFS)

and the goal is producing $A(7)$. The map sequence [1, 2 ,3 ,1 ,2 ,3 ,1] indicates that the contraction W_1 will shrink the stick-man image in $A(0)$ to one of the three stick-men in $A(1)$, then W_2 shrinks this to one of the nine images in $A(2)$, then W_3 shrinks the image again, etc. The result $W_1 \circ W_3 \circ W_2 \circ W_1 \circ W_3 \circ W_2 \circ W_1$ ($A(0)$) is only one small copy of the stick-man in $A(7)$ but, together, all such map sequences form $A(7)$. To get all of the map sequences, all you need is seven nested FOR loops. However, when m is variable, recursion is the solution.

The following recursive procedure loops through all map sequences and isolates all of the graphics and transformations in the procedure TransformA0, discussed below. The main program should call DoAllMapSeq (MapSeq, 1, m) to display $A(m)$. To test the recursion, replace the TransformA0 call with a printout of the MapSeq. We will consistently use i and j to specify levels from 0 to m while k specifies a map from 1 to n.

```
Map Sequence Algorithm in pseudo-code
n = number of maps
m = level to be displayed
MapSeq: ARRAY [1..m] OF integer

PROCEDURE DoAllMapSeq (MapSeq, i, m)
{Given MapSeq from index 1 to i-1, generate all sequences from index i to m.
Display a copy of A0 transformed by each resulting MapSeq.}
LOCAL k  {Different local variable k for each call.}
    IF i = m+1 THEN
        TransformA0 (MapSeq)
    ELSE
        FOR k = 1 TO n
            LET MapSeq(i) = k
            CALL DoAllMapSeq (MapSeq, i+1, m)
        NEXT k
    END IF
END PROCEDURE
```

To implement TransformA0, arrays are used to store the A0 image, copies of it, and the IFS transformations. Each map W_k is an affine transformation given by the six parameters that map (x, y) to $(ax + by + e, cx + dy + f)$. For each W_k, these parameters are stored in a matrix and the transformation is captured in a matrix multiplication of the form (see [3, Chapter 7])

$$[x \quad y \quad 1]\begin{bmatrix} a & c & 0 \\ b & d & 0 \\ e & f & 1 \end{bmatrix}$$

The IFS is stored in a list W of n different 3×3 matrices, so that W(k) will return the matrix for the kth affine transformation. The array A0 has p rows, each of the form $[x, y, 1]$, listing points to be connected for some stick-figure. Assume a pseudo-statement "plot matrix A0" that will ignore the column of ones and connect the points on a graphics screen. Now, TransformA0 (MapSeq) is an iterative loop of matrix multiplications.

Map Sequence Algorithm Continued

```
p = number of points in A0 array
A0, COPYA0:  ARRAY [1..p, 1..3] OF real  {points to be connected}
W:  ARRAY [1..n] OF ARRAY [1..3, 1..3] OF real  {a matrix for each affine
transformation in the IFS}

PROCEDURE TransformA0 (MapSeq)
     LET COPYA0 = A0
     FOR j = 1 TO m DO
          matrix COPYA0 = COPYA0 * W(MapSeq(j))
     NEXT j
     plot matrix COPYA0
END PROCEDURE
{example main program}
CALL ReadW (n, W, xmin, xmax, ymin, ymax, "IFSfile")
CALL ReadA0 (p, A0, "A0file")
INPUT m  {level to be displayed}
SET WINDOW xmin, xmax, ymin, ymax  {initialize graphics screen, scaled to IFS}
CALL DoAllMapSeq (MapSeq, 1, m)
END
```

The example main program calls procedures to read the IFS maps and A0 set from files and displays the one set $A(m)$. To display the sequence $A(0)$, $A(1)$,..., $A(m)$, place a FOR loop around the call to DoAllMapSeq. The appendix contains example data files; math.ifs is used in producing Figure 3 and stickman.a0 used in producing Figure 1. It is a simple exercise to define the other IFSfiles and A0files used in producing the figures. Many authors (e.g., [2]) don't include screen scaling parameters in IFS files since they always assume the screen to be the unit square $[0,1]\times[0,1]$. Since this is not always consistent or desirable, listing the scale can be extremely helpful. The industrious programmer can transform the A0 set to match the IFS scale before starting the recursion.

Remember that it is the IFS that determines the attractor, regardless of the scale of the initial A0. The math.ifs maps are designed to fill the unit square. Although Figure 3 uses the unit square as the A0 set, the scale *ymin* = - 0.5 and *ymax* = 1.5 reshapes the square into a rectangle that circumscribes the figure. To produce Figure 3, the plot matrix pseudo-statement was modified to fill the rectangle given by COPYA0.

The pseudo-code is readily translated into structured languages, such as True BASIC or Pascal. The True BASIC language includes all of the matrix manipulations routines, including a command for plot matrix (if the column of 1's is moved to the middle). It also allows dynamic sizing of the arrays so that *n*, *p*, and *m* can be read and used to resize W, A0 and MapSeq. Although True BASIC allows the three-dimensional array W, it, unfortunately, can't reference a slice W(k) and use it as a matrix. This requires copying the data into a matrix by a procedure such as

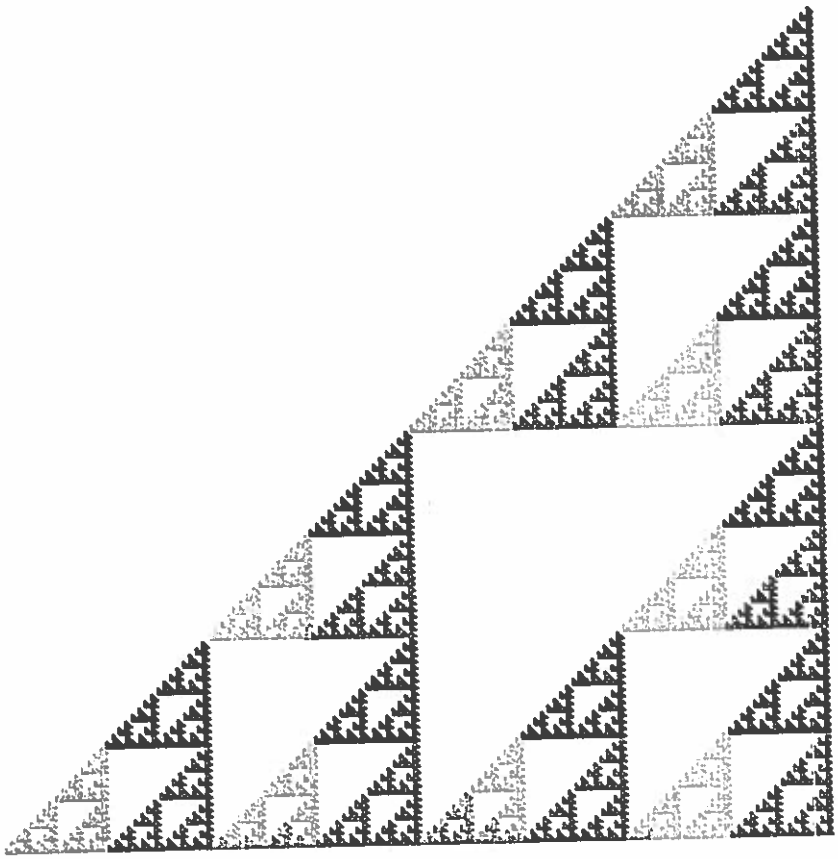

Figure 2

Assign(Matrix, W, k). Most Pascal compilers would have the opposite problems - using W would be easy but the matrix commands would have to be programmed. Actually, programming matrix commands is helpful since they can be customized for efficiency.

The shading in Figure 2 is a good example of the coloring patterns possible with the Map Sequence Algorithm. This figure colors (in gray-scale) every copy of A0 according to the map number used in the third from last transformation in the corresponding MapSeq. For any copy of A0 that is plotted, the last transformation MapSeq(m) determines which third of the overall image it appears in. MapSeq(m–1) determines which third of that third contains the copy, etc. Thus, we can color the components of any level by inserting set color MapSeq(m+1-colorlevel) immediately before plot matrix COPYA0. Figure 2 uses *colorlevel* = 3 and, thus, alternates the $n = 3$ colors on every 1/27th of the figure.

The Map Sequence Algorithm also allows control over the order in which pieces are drawn. Figure 3 can be produced by completing all of the smallest *M*'s first, then all the smallest *A*'s, etc; a drawing order that we'll call a *scattered plot*. The figure can also be produced by completing the largest *M* with all it's detail before starting the largest *A*. This drawing order seems to "grow out" of the lower-left corner and will be called a *contiguous plot*.

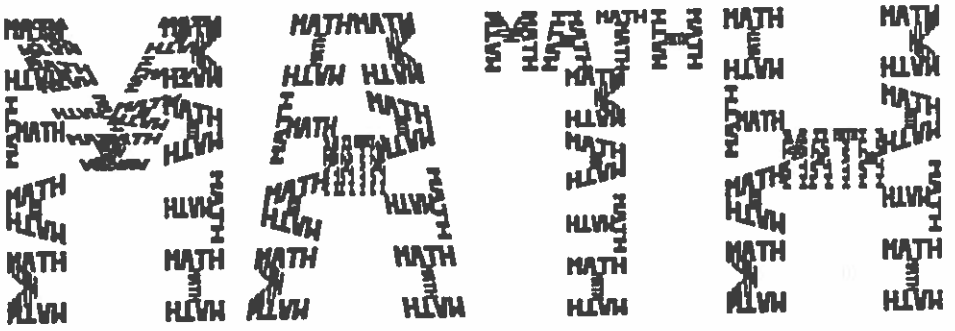

Figure 3

The Map Sequence Algorithm as stated above produces a scattered plot. Figure 3 consists of $m = 3$ levels using $n = 12$ different maps, one for each stick in the word MATH. In the recursion, the first twelve MapSeq's produced are $[1,1,1]$, $[1,1,2]$, $[1,1,3]$, ... ,$[1,1,12]$. When

TransformA0 is called with these MapSeq's, the first two maps in each MapSeq form the same small rectangle and the last map, as it varies, scatters this small rectangle into a corner of each of the twelve large sticks.

To produce a contiguous plot, we can alter TransformA0 to use MapSeq in reverse order by changing the FOR loop to FOR j = m DOWNTO 1 DO. With this alteration, each of the 12 maps is applied before the identical contractions in the first twelve MapSeq vectors. This takes the sticks that form the word MATH, image $A(1)$, and shrinks them into the corner of the image. This way the word MATH is written over and over as the larger image appears. After processing every MapSeq with MapSeq(1) = 1, an entire large stick is filled with the word MATH made from smaller copies of the word; this is a copy of image $A(2)$. So, the contiguous plot does complete copies of $A(i-1)$ in order to form $A(i)$. Indeed, this is the drawing order performed by the Recursive Drawing Algorithm of the previous section.

THE POINT SEQUENCE ALGORITHM

When using the Map Sequence Algorithm to produce a scattered plot, unnecessary computation can be avoided by storing intermediate results. If $m = 7$, MapSeq $[1,1,1,1,1,1,1]$ is immediately followed by MapSeq $[1,1,1,1,1,1,2]$ and the same first six transformations are performed over again. Such repetition can be eliminated by performing the transformation, and storing the result, at each level of the recursion. Each recursive call needs a local array for the transformed copy of A0 at that level. The vector MapSeq is not necessary, since the transformations are performed as they are chosen. This streamlines the algorithm and earns a new name.

Point Sequence Algorithm in pseudo-code

```
m = level to be displayed
n = number of maps
p = number of points in A0
A0: ARRAY [1..p,1..3] OF real
W: ARRAY [1..n] OF ARRAY[1..3,1..3] OF real

PROCEDURE DoAllMapSeq (COPYA0, seqlength)
{Transform COPYA0 by every possible sequence of maps of length seqlength; plot
each result.}
```

```
LOCAL k: integer,
LOCAL NEWA0: ARRAY [1..p,1..3] OF real
    IF seqlength = 0 THEN
        plot matrix COPYA0
    ELSE
        FOR k = 1 TO n
            matrix NEWA0 = COPYA0 * W(k)
            CALL DoAllMapSeq (NEWA0, seqlength-1)
        NEXT k
    END IF
END PROC

{example main program}
CALL ReadW (n, W, xmin, xmax, ymin, ymax, "IFSfile")
CALL ReadA0 (p, A0, "A0file")
INPUT m {level to be displayed}
SET WINDOW xmin, xmax, ymin, ymax {initialize graphics screen, scaled to IFS}
CALL DoAllMapSeq (A0, m)
END
```

The LOCAL variable declarations call attention to memory allocations that are necessary for the recursion to work properly. The algorithm works with a "call by reference" for the array parameter COPYA0 (required for array parameters in True BASIC). If allowed, "call by value" could be used to allocate the memory by directly entering the matrix multiplication as the actual parameter.

The "Multiple Reduction Copy Machine" algorithm found in [6, p. 294] performs computations that are essentially equivalent to the Point Sequence Algorithm. However, the algorithm in [6] is unstructured. The recursion is not explicit but is accomplished by a nesting of GOTO line number statements.

THE MATRIX SEQUENCE ALGORITHM

While the Point Sequence Algorithm significantly reduced the computation of the Map Sequence Algorithm, the Point Sequence Algorithm cannot produce a contiguous plot. How can such efficiencies be used to generate a contiguous plot when the very first map to be applied varies between consecutive map sequences? The answer is to accumulate a transformation matrix TM at each level that represents the effect of several maps that will be performed after remaining maps are determined. Each recursive call can have a local copy of TM for that level. For example, when MapSeq $[1, 2, 3, 1, 2, 3, 1]$ is immediately followed by MapSeq

$[1, 2, 3, 1, 2, 3, 2]$, the sixth level will have created a matrix TM corresponding to $W_1 \circ W_2 \circ W_3 \circ W_1 \circ W_2 \circ W_3$. The recursive call for the remaining map will effectively insert the matrix W(k) between A0 and TM, for each k.

```
Matrix Sequence Procedure in pseudo-code
PROCEDURE DoAllMapSeq (TM, seqlength)
{Form transformation matrices for all map sequences of length seqlength, followed
by the transformation in the given TM; transform and plot resulting copies of A0.}
LOCAL k: integer,
LOCAL NEWTM: ARRAY [1..3,1..3] OF real
    IF seqlength = 0 THEN
        plot matrix A0 * TM
    ELSE
        FOR k = 1 TO n
            matrix NEWTM = W(k) * TM
            CALL DoAllMapSeq (NEWTM, seqlength-1)
        NEXT k
    END IF
END PROCEDURE
```

The main program calls this procedure with an identity matrix for argument TM and seqlength $= m$. To avoid unnecessary multiplication by an identity matrix, the main program could include a FOR loop that would CALL DoAllMapSeq (W(k), m–1) for each k.

The Matrix Sequence Algorithm can produce coloring patterns and control over the drawing order, as discussed for the Map Sequence Algorithm. To color a level, insert IF seqlength = m+1-colorlevel THEN set color k just inside the FOR loop. This coloring scheme also works in the Point Sequence Algorithm. The Matrix Sequence Algorithm can also produce a scattered plot by simply switching the order of multiplication, so that the line reads matrix NEWTM = TM * W(k).

THE PIXEL ARRAY ALGORITHM AND L-SYSTEMS

There is a frightening observation about all that we've done. In all four algorithms, the computation time grows exponentially, proportional to n^m. Actually, any pixel-free graphics algorithm is doomed to exponential growth. The number of copies of $A(0)$ in $A(m)$ is exactly n^m. In pixel-free graphics, we have to know the points to be plotted; whether or not there is a unique pixel for this point must be a display problem, not reflected in the algorithm. Luckily the memory costs of

these recursive algorithms are insignificant (proportional to m). This space efficiency makes these algorithms useful for reasonably low m values. Before considering how useful, we consider the common pixel-based deterministic algorithm that avoids exponential growth. It is very different from the preceding algorithms.

The idea of the Pixel Array Algorithm is to iteratively (no recursion here!) finish $A(i-1)$ and carry all of that information to level i by storing the screen image in an array. Barnsley calls this the deterministic algorithm [1, p.88]. Every pixel on the computer screen is represented by an array entry which stores on or off (or the pixel color). Actually, we need two arrays; OldArray represents the screen for $A(i-1)$ and NewArray represents the screen for $A(i)$. For each "on" entry in OldArray, transform it by all n of the maps in the IFS and "light" the resulting entry in NewArray. (By only transforming "on" entries, you error on the side of keeping information; otherwise, whole lines can disappear in one step!) Replace old with new and repeat. Of course, rounding can map several old entries to the same new entry, whenever the distance between the image of points is smaller than the screen resolution. This "discretization" phenomena is common in computer graphics and allows the practical convergence of all fractal images.

Though conceptually straight-forward, the Pixel Array Algorithm runs into serious memory problems and significant, but linear, computation time cost. Barnsley's example implementation uses a 100×100 array which is only about one-sixth of a crude CGA screen. If you use only one byte per array entry, a 320×200 CGA screen uses 64K – the maximum allowed by some DOS based languages. The catch-22 is that, as computer memory grows, the number of screen pixels usually also increases, so that allocating such arrays may always be a problem. One way to overcome this problem is to use a color scheme to store both OldArray and NewArray in the hardware video memory [5]. Still, processing every pixel is a time-consuming process. However, every step takes about the same amount of computation time, independent of the level i! While it may take patience to reach $A(5)$, it's not any harder to go on to $A(10)$. Indeed, you can eventually reach a true steady-state.

There is a pixel-free version of the iterative idea – carrying all of the $A(i-1)$ information to level i. Instead of actually "plotting" a figure, simply list all of the points in an array NewPoints, starting with the array A0. For each step in the sequence of images $A(i)$, let NewPoints become OldPoints and, for each IFS map, transform all of the OldPoints into the

NewPoints array. Of course, this multiplies the array size by n (the number of maps) on each step. Thus, both time and memory costs grow exponentially. Since the Point Sequence Algorithm produces similar time efficiency without any of the memory problems, this idea will be dropped. However, the idea is related to the method known in the literature as L-systems.

Whereas an IFS encodes fractals by matrix transformations on points, an *L-system* encodes fractals by transformation rules on strings of turtle graphics commands (pen movements left, right, forward, etc.). In the usual implementation of L-systems, the goal is to produce a string of all of the turtle graphics commands necessary to draw $A(m)$. An iterative algorithm, such as [7, p. 275], repeatedly builds a NewString from the OldString by transformation rules. While the transformation rules are an efficient coding, the iterative algorithm that stores the entire resulting string has serious space problems. This algorithm has exponential growth in time and memory costs as described just above. Actually, the L-system notation is not tied to this algorithm any more than the IFS notation is tied to the Pixel Array. In [6, Section 7.7], a recursive algorithm for L-systems avoids actually computing the string. We will not pursue L-systems further, although recursive implementation would parallel this article and show the efficiency of recursion whenever the iterative alternative is to build an exponentially growing array.

EFFICIENCY AND FEASIBILITY

Because of the exponential time-cost, all of the recursive algorithms are feasible only for relatively low values of m, the desired level, but they can be very useful for these values. The usual purpose of a deterministic algorithm is to view these early steps where the recursion excels. Also, there is a tendency for a usual "time limit" of most people's patience to correspond to the "graphics limit" where higher values of m no longer produce significant visible improvement in the image. Here is where a contiguous plot has an important payoff - you get to see a corner of the finished image and can decide if you want to abort and try a higher or lower m value. For the Sierpinski triangle of Figure 1, the recursive algorithms require patience around $m = 7$, the last image displayed in the sequence. For the MATH logo in Figure 3, patience is needed for this level $m = 3$ but it is more appealing than even higher levels where the detail is illegible. The difference in the two examples is because a higher number of IFS maps takes longer but usually implies more contraction

(smaller factor) in each map so that the sequence converges faster. On the other hand, if your IFS includes a map with a large contraction factor (very little contraction), then your patience may run out before the image "converges." An example of this problem is Barnsley's famous fern IFS with only $n = 4$ maps but where the large top image of the fern is only a 0.85 contraction of the whole [1, p. 104 and 40].

Of course, the different recursive algorithms are not all the same speed. The Recursive Drawing and Map Sequence algorithms are good expository versions, that can be used for low m, but they do take longer than the other two algorithms. The Point Sequence Algorithm nests m loops of p×3 times 3×3 matrices, where we may assume the number of points (in A0) $p \geq 3$. The Matrix Sequence Algorithm nests m loops of 3×3 matrix products, which seems better until you realize that there still remains a p×3 times 3×3 product in the innermost loop. Analysis below shows that the Point Sequence Algorithm requires fewer multiplications than the Matrix Sequence Algorithm if $p < 3n$. This is the case for simple A0 sets, as in Figures 2 or 3. The Matrix Sequence Algorithm is faster for the stick-man images in Figure 1.

To compare algorithms, we count the number of *row-multiplications*, meaning, for each matrix multiplication, count the number of rows of the left matrix when it is multiplied times a 3×3 matrix on the right. In a general matrix product, such a row-multiplication requires 9 floating-point multiplications. However, the third column of every matrix consists of one's or zero's, so that customized products only require 4 multiplications per row-multiplication. Graphics commands can be more time consuming than these multiplications, but they don't significantly vary between algorithms.

The algorithms differ in row-multiplication counts because of the different strategies for computing $A0*W(k_1)*W(k_2)*\cdots*W(k_{m-1})*W(k_m)$ for each map sequence. The Map Sequence Algorithm performs all of the matrix products, left to right, for all n^m map sequences for a total of $(mp)n^m$ row-multiplications. The Point Sequence algorithm nests loops of products from left to right. The first matrix product is performed n times, the second is done n^2 times (n times for each of the first products), etc. Thus, the algorithm performs $s_m = n + n^2 + \cdots + n^m = \left(\frac{n}{n-1}\right)(n^m - 1) < 2n^m$ matrix products. We conclude that the Point Sequence Algorithm performs $(p)s_m < (2p)n^m$ row-multiplications. The Matrix Sequence Algorithm associates the loops of products from right to left. The right

product is performed n^2 times (n times for each of the n right matrices), the next is done n^3 times, etc. (If the identity matrix is used to start the process, it adds n products - relatively insignificant). Thus, the algorithm performs $n^2+n^3+\cdots+n^m = ns_{m-1}$ products between 3×3 matrices and an additional n^m matrix products with a p×3 matrix for a total of $(3n)s_{m-1}+(p)n^m < (6+p)n^m$ row-multiplications. Since the Point Sequence count can be written $(p)s_{m-1}+(p)n^m$, we conclude that it beats the Matrix Sequence if $p < 3n$.

We can compare the row-multiplication count for the Pixel Array Algorithm, though it must be remembered that the real advantage of the recursive algorithms is in space-efficiency. In one step (from $A(i-1)$ to $A(i)$) of the Pixel Array Algorithm, each "on" pixel requires n row-multiplications to transform it to the n copies at the next level. If we make the crude assumption that, say, about one one-hundredth of 640×480 pixels are "on" at each level, then this algorithm requires (3072)n row-multiplications per step or (3072n)m row-multiplications to display $A(m)$. For example, if $n=3$ and $p=3$ (as in a classic Sierpinski triangle), the Pixel Array Algorithm uses (9216)m row-multiplications while the Point Sequence Algorithm uses less than (6)3^m row-multiplications. This Point Sequence count is significantly smaller through $m=8$, even though the exponential growth will greatly exceed the linear growth of the Pixel Array count for higher values of m. This may seem like an unfair comparison since the Pixel Array displays the whole sequence of images $A(0)$, $A(1)$, ... , $A(m)$ while the Point Sequence directly displays $A(m)$ only. To display the sequence of images, the recursive algorithms must start from scratch for each image; but this does not make that much difference in the exponential growth. In the example, the Point Sequence Algorithm can generate the sequence of images with less than $6(3+3^2+\cdots+3^m) < 9(3^m)$ row-multiplications.

CONCLUSION

Each of the algorithms has a purpose. The Recursive Drawing Algorithm and Map Sequence Algorithms are excellent, simple examples of recursion. The former directly implements the Fractal Attractor Theorem and the latter yields insight into how the more technical recursive methods work. All of the deterministic algorithms provide an

entertaining demonstration of functional iteration to a fixed-point in a more general metric space than usually envisioned. They also clearly show the action of the IFS maps. For software to be used rather than just studied, it's wise to go with the most efficient method. If the number of points in A0 is small compared to the number of IFS maps, this would mean the Point Sequence Algorithm. On the other hand, the Matrix Sequence Algorithm is sometimes more efficient and has the significant advantage that it can produce a contiguous plot. It can be very helpful to see a finished corner of your image in order to decide if you want to invest the time necessary to get the full image. Of course, if you want to go beyond the first few levels, try the Pixel Array Algorithm or just give up on the deterministic approach and use the Random Iteration Algorithm.

We gratefully acknowledge the assistance of our former students John Annen and Barbara Defenbaugh and colleague Stephen Davis.

APPENDIX

MATH.IFS, an example IFS file containing
n (the number of IFS maps)
xmin, *xmax*, *ymin*, *ymax* (scale plots to these screen coordinates)
a, *b*, *c*, *d*, *e*, *f* (matrix entries for each IFS map)

```
12
0,      1,      -.5,    1.5
0,      -.62,   1,      0,      .062,   0
.084,   .062,   -.5,    0,      0,      1
.084,   -.062,  .5,     0,      .147,   .5
0,      .062,   -1,     0,      .169,   1
.053,   -.062,  1,      0,      .319,   0
.053,   .062,   -1,     0,      .372,   1
.063,   0,      0,      .193,   .34,    .404
.231,   0,      0,      .193,   .513,   .807
0,      .062,   -.807,  0,      .597,   .807
0,      -.062,  1,      0,      .831,   0
.106,   0,      0,      .193,   .831,   .404
0,      .062,   -1,     0,      .938,   1
```

STICKMAN.A0, an example A0file containing points to be connected.

```
16
.2, 0
.5, .3
.8, 0
.5, .3
.5, .5
.2, .7
.5, .5
.8, .7
.5, .5
.5, .7
.63, .775
.63, .925
.5, 1
.37, .925
.37, .775
.5, .7
```

REFERENCES

1. M. F. Barnsley, *Fractals Everywhere*, First edition, Academic Press, Boston, MA (1988).
2. R. M. Crownover, *Introduction to Fractals and Chaos*, Jones & Bartlett, Boston, MA (1995).
3. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA (1982).
4. J. G. Kemeny and T. E. Kurtz, *True BASIC Reference Manual*, version 3.0, True BASIC (1990).
5. L. Noble, Meredith College Student Project, contact Jo Guglielmi, Meredith College (1995).
6. H. O. Peitgen, H. Jürgens, and D. Saupe, *Chaos and Fractals, New Frontiers of Science*, Springer-Verlag, NY (1992).
7. H. O. Peitgen and D. Saupe, *The Science of Fractal Images*, Springer-Verlag, NY (1988).