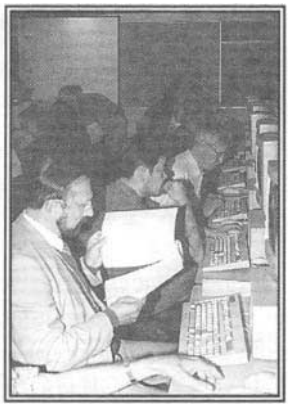
Introducing a new column for APL Quote Quad:

With J

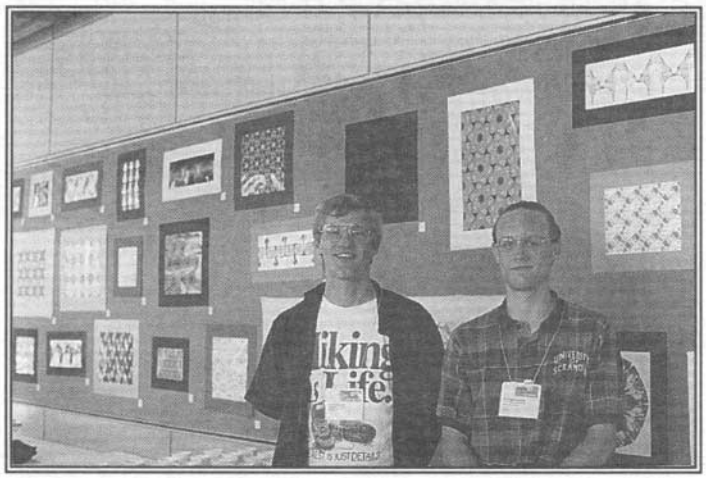
101 Ways to Build a Sierpinski Triangle

*—by **Cliff Reiter***
Department of Mathematics
Lafayette College; Easton, Pennsylvania

At APL97 I ran a workshop on creating images with J. The first exercise involved creating a Sierpinski Triangle. Ken Iverson commented that I seem to start everything I do with the Sierpinski Triangle as an example. I don't think that is true, but it does seem like a good idea. Thus, my first *Quote Quad* column on using J focuses on ways to create Sierpinski Triangles. Of course, this is really an excuse to discuss a bunch of J in a way that results in some interesting images. Many of these ideas may be familiar, but readers are encouraged to look for new points of view. You might note the usefulness of inverse functions, gerunds and extended integers and may even discover new uses of base-three representations. Did you know you don't have to "build" Sierpinski's Triangle? You can decide the value of each pixel from the pixel coordinates.



Cliff Reiter's APL97 workshop



Cliff Reiter and Nathan Carter at the APL97 "Chaos with Symmetry" exhibit

Future columns planned include dot-by-dot creation of one of the fractal types from the APL97 exhibit "Chaos with Symmetry" [1] and another takes a look at creating "music" with J. While I often create fractals in my work with J, my goal for the "With J" column is the illustration of J usage with fun examples from a variety of topics.

I will define what I mean by the Sierpinski Triangle by examples that approximate it. Purists will want to take some sort of limit of the approximation process we will explore. Of course, they won't see their result. We will be able to see our finite approximations. Sometimes these are discrete matrices. Other times, lists of lit pixels or line segments. We won't worry about whether the Sierpinski Triangle will "open" toward the upper right or some other direction; nor will we worry about whether it is equilateral or not. This list of 101 ways is not meant to be complete. There are at least two rather different ways appearing in [8]. Remarkable connections with the tower of Hanoi are described in [13] but won't be considered here. Sierpinski's original paper [12] described a curve that, roughly speaking, intersects itself at every point. Our last "Way" constructs approximations to that curve. We look at matrix- and pixel-based constructions first.

Way I: Juxtaposition

We can readily create a matrix and juxtapose it above or beside itself. We say `m,m` gives `m` adjoined to itself and `m,.m` gives `m` stitched to itself.

```
   ]m=:3+i.2 2
3 4
5 6
   m,m
3 4
5 6
3 4
5 6
   m,.m
3 4 3 4
5 6 5 6
```

Reflex, denoted by ~, allows us to apply any dyadic verb (function) with identical arguments. That is, if `f` is a dyadic verb then `f~ y` is the same as `y f y`. Thus we can stitch a matrix to itself using `,.~ m`. One advantage of using that notation is that we only need to refer to the matrix once. Also observe below that padding with zeros occurs when a matrix is adjoined to another with different row lengths.

```
   ,.~ m
3 4 3 4
5 6 5 6

   m,2 5$1
3 4 0 0 0
5 6 0 0 0
1 1 1 1 1
1 1 1 1 1
```

Two verbs listed in isolation comprise a hook; it results in a composition of the functions. Here `ab=: ,,.~` is a verb that adjoins a thing to the stitch of the thing with itself. The name "`ab`" stands for above and beside.

```
   ab=:,,.~
   ab m
3 4 0 0
5 6 0 0
3 4 3 4
5 6 5 6
```

We can iteratively apply this verb. For example, `ab^:3` denotes iterating `ab` three times. This builds binary matrices that form a fractal triangular shape with zeros appearing in the upper right. Below `,1` denotes the vector with a single element `1`. Adding the vector structure to that number is enough to get the matrix building started. Notice that triangular regions of zeros appear. The Sierpinski Triangle can be thought of as the limiting image that appears as we take more iterations.

```
   ab ,1
1 0
1 1

   ab^:2 ,1
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1

   ab^:3 ,1
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1

   ab^:4 ,1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0
1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

We can save a larger version of this array using J bitmap creation utilities. Below we create a palette and a matrix representing the array of indices into the palette giving the image and use the utility `writebmp8` from `raster3.js` used in [8] to save the bitmap. One can also use the utility `writebmp` from `bmp.js` distributed with J in the `\packages\graphics` directory. Readers duplicating these images should run one of those scripts. If `raster3.js` has been run, you should be able to use the commands given here. If `bmp.js` has been run, use the following as a guide to the small changes in syntax required: if `pal` is a palette and `b` is an array of indices giving the color of each pixel, the image file created with `(pal;b) writebmp8 'temp1.bmp'` ought to be equivalent to the file created by `(b{256 256 256#.pal) writebmp 'temp2.bmp'`. The first uses `writebmp8` from `raster3.js` and the second uses `writebmp` from `bmp.js`.

We use a white and black palette when creating the bitmap. The first row of `pal` being `255 255 255` means that the Red Green and Blue components of the corresponding pixels are full on, yielding white. The `0 0 0` in the second row gives full off, yielding black. Hence, zeros in the array will correspond to white in the image and the ones will correspond to black.

```
   ]pal=:255 255 255,:0 0 0
255 255 255
  0   0   0
   b=:ab^:9 ,1
   (pal;b) writebmp8 'sier_tri.bmp'
```

Figure 1 shows the resulting image:

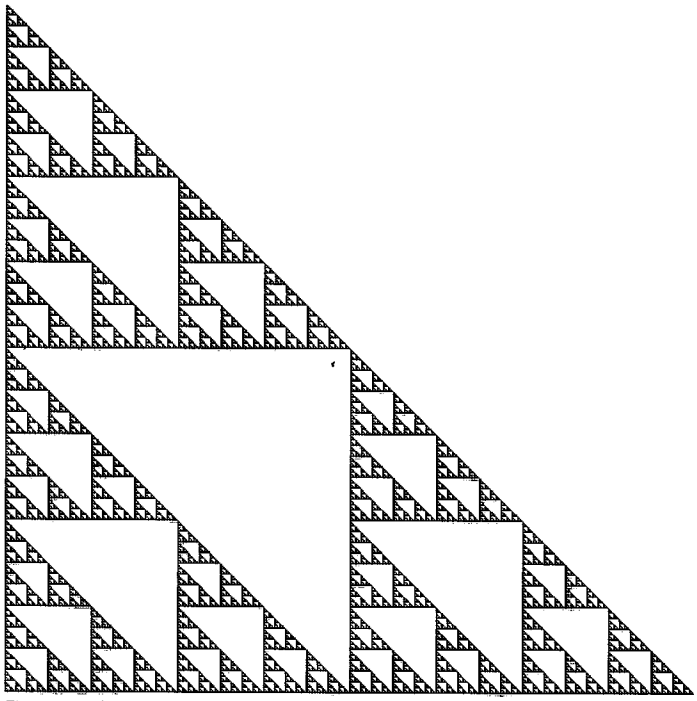


Figure 1: Sierpinski's Triangle

A classic modification of this scheme is to put a block of zeros in the center of a three-by-three juxtaposition of matrix blocks; iteration of this scheme gives the Sierpinski Carpet. This is shown in Figure 2. I like to ask my students to create tacit functions doing the block building required for this process. A spoiler is given below—but another question lurks beyond. The verb `s3` adjoins together three copies of its matrix argument. Do you see why? The verb `s30` does the same thing, but the middle block is made up of zeros. The verb `0"0` is the zero function of

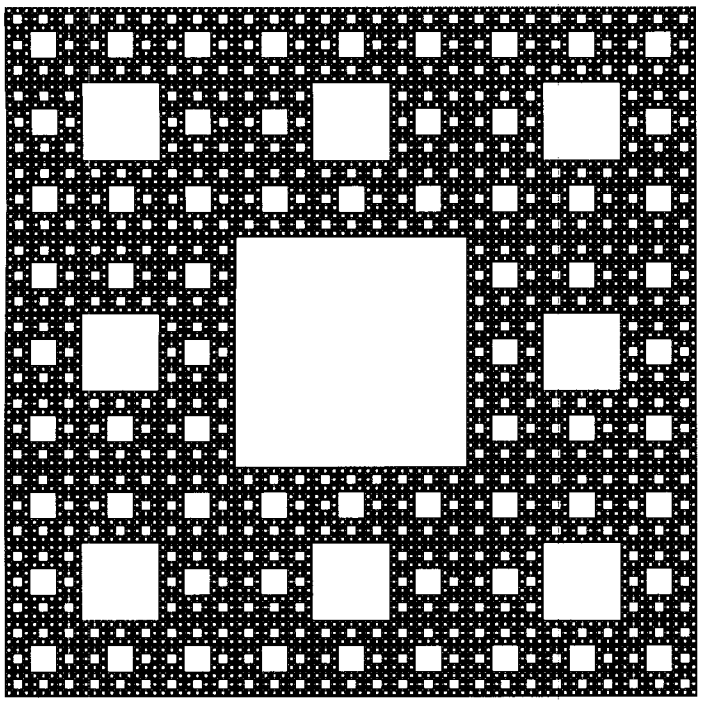


Figure 2: Sierpinski's Carpet

rank 0. That is, every element of its input is replaced by zero. The verb `s30` is created as a train of four verbs. This is a hook of a fork that can be read (left to right) as adjoin the matrix to zero matrix version of the matrix adjoined to the matrix. Note the] denotes the identity function. Lastly, the verb `sc` is a train of five verbs that can be read as stitch `s3` to `s30` to `s3`. Readers new to J would do well to experiment with naming and seeing the result of various pieces of these verbs.

```
   s3=:,,~
   zero=:0"0
   zero m
0 0
0 0
   s30=: , zero , ]
   s30 m
3 4
5 6
0 0
0 0
3 4
5 6
   sc=:s3 ,. s30 ,. s3
   sc^:2 ,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 0 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

   b=:sc^:6 ,1
   (pal;b) writebmp8 'sier_car.bmp'
```

Once students have created a working tacit function, like `sc`, I often ask them to see how short a version they can give. I have seen some remarkable solutions. Can you give a shorter definition for zero that will work for this construction?

Way 2: Binomial Coefficient Function Tables

The dyad `!` can be used for computing binomial coefficients. When these are placed in an array using table building (also known as "outer product") via `!/`, Pascal's triangle appears. We may want to commute the arguments of `!` as in `!~/` to transpose the function table so that each row is the sum of the previous row with a shifted version of the previous row; this is closer to the traditional visual display of Pascal's triangle.

```
   3 ! 5
10
   0 1 2 3 4 5 !/ 0 1 2 3 4 5
1 1 1 1 1  1
0 1 2 3 4  5
0 0 1 3 6 10
0 0 0 1 4 10
0 0 0 0 1  5
0 0 0 0 0  1

   !~/~i.8
1 0  0  0  0  0 0 0
1 1  0  0  0  0 0 0
1 2  1  0  0  0 0 0
1 3  3  1  0  0 0 0
1 4  6  4  1  0 0 0
1 5 10 10  5  1 0 0
1 6 15 20 15  6 1 0
1 7 21 35 35 21 7 1
```

Modulo two versions of Pascal's triangle give our Sierpinski Triangle matrix.

```
   2|!~/~i.8
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1
```

Sadly, if one tries to create a 64-by-64 version of the Sierpinski Triangle this way, incorrect results appear because the coefficients become too large for floating precision. Happily, we can compute the binomial coeffecients to arbitrary precision by using extended integers in J. These extended integers can be input using an "x" suffix on the integer. Other integers are "promoted" to extended integers, as need be, in order to preserve the exact integers. Thus, `2|!~/~i.64x` would give a 64-by-64 correct version the Sierpinski Triangle. Newer versions of J may not show the extended integer suffix x on output.

```
   30 ! 63
8.60778e17
   30 ! 63x
86077800559424706 9x
   2|30 ! 63
0
   2|30 ! 63x
1x
```

How might these images be generalized? A natural choice is to use other bases. A small example modulo 3 is given below and Figure 3 gives an image from the 81-by-81 case. Creating that figure took advantage of these exact integers and used gray scale for color coding. Can you duplicate the image?

```
   3|!~/~i.9
1 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0
1 2 1 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0
1 1 0 1 1 0 0 0 0
1 2 1 1 2 1 0 0 0
1 0 0 2 0 0 1 0 0
1 1 0 2 2 0 1 1 0
1 2 1 2 1 2 1 2 1
```

The fractal dimension of Pascal's triangle modulo p is investigated in Reiter's paper [4] (no relation to me). Other generalizations and examples can be found in [14]. In particular, creating these images for combinatorial numbers, such as Stirling Numbers [2], is a worthwhile and interesting exercise.

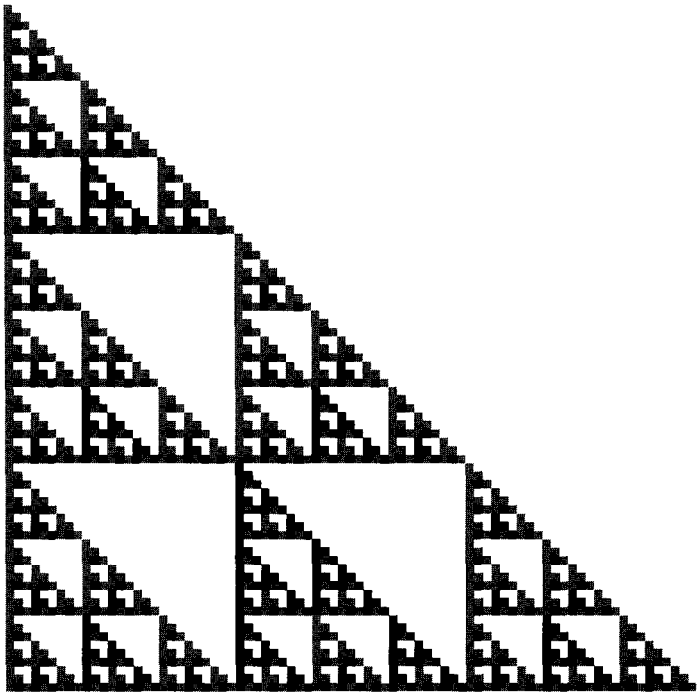


Figure 3: Binomial Coefficients Modulo 3

Way 10: Binary Matrix Arithmetic

We can generate all the three-"digit" binary numbers using #: which denotes antibase. Each row in the result corresponds to a number. Transpose is denoted with |:. The logical functions "or" and "and" are denoted +. and *. respectively. Thus +./ . *. denotes a logical matrix product that for each row and column determines whether there is a common lit position.

```
   ]X=:#: i. 8
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
   |: X
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
   X +./ . *. |: X
0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 1 1 1 0 1 1 1
0 0 0 0 1 1 1 1
0 1 0 1 1 1 1 1
0 0 1 1 1 1 1 1
0 1 1 1 1 1 1 1
```

The logical matrix product given above gives the logical "not" of our Sierpinski Triangle opening to the lower right. Note no iterative process was used. The entries were computed directly from the binary form of their row and column indices.

This suggests that it might be interesting to try base three forms for the indices. The matrix below shows the holes in the Sierpinski Carpet arising from the ones in that computation. For other variations of this, refer to [5,6,10,11] or try your own ideas. Of course, using color in the images helps when there are more than two different array entries.

```
   Y=:3 3#:i.9

   Y +./ . *. |: Y
0 0 0 0 0 0 0 0 0
0 1 2 0 1 2 0 1 2
0 2 2 0 2 2 0 2 2
0 0 0 1 1 1 2 2 2
0 1 2 1 1 1 2 1 2
0 2 2 1 1 1 2 2 2
0 0 0 2 2 2 2 2 2
0 1 2 2 1 2 2 1 2
0 2 2 2 2 2 2 2 2
```

```
pal2=:255 0 0,255 255 255,:0 255 0
b=:Y +./ . *. |: Y=: (5#3)#:i.3^5
(pal2;b) writebmp8 'siercar2.bmp'
```

Note that "or" generalizes to gcd (greatest common divisor) and "and" generalizes to lcm (least common multiple) so that these are meaningful for nonbinary numbers. Base three also keeps the length of "101 Ways" to do something bearably short.

Way I1: One dimensional Finite Automata

We next consider the function that tests whether or not an entry is equal to its neighbor. The function `_1&|.` rotates a vector one element to the right wrapping the last item around to the first position. If we use this with not-equals `~:` in a hook we have a function that tests whether each element is not-equal to its left neighbor. Such a function is considered a finite automaton since the result at any position depends only on two neighboring values in the argument.

```
   ]v=:1 1 1 1 0 0 1 1
1 1 1 1 0 0 1 1

   _1&|.v
1 1 1 1 1 0 0 1

   v ~: _1&|. v
0 0 0 0 1 0 1 0

   auto=:~: _1&|.

   auto v
0 0 0 0 1 0 1 0

   auto^:(i.8) 0=i.8
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1
```

We see we get the Sierpinski Triangle from iterating this function. Finite automata have been the subject of much recent study because of their complex behavior. For example, Figure 4 shows a three-neighbor finite automata (Rule 135 from [8,9]) that has an interesting mix of structure and randomness. The array giving that image can be recreated with the following:

```
perext=:{: , ] , {.
AUTO=:{~ 3&(#.\)@perext
b=:(#:135)&AUTO^:(i.128) ?. 128$2
```

We leave the details of the J code for automata of that type for the interested reader to pursue.

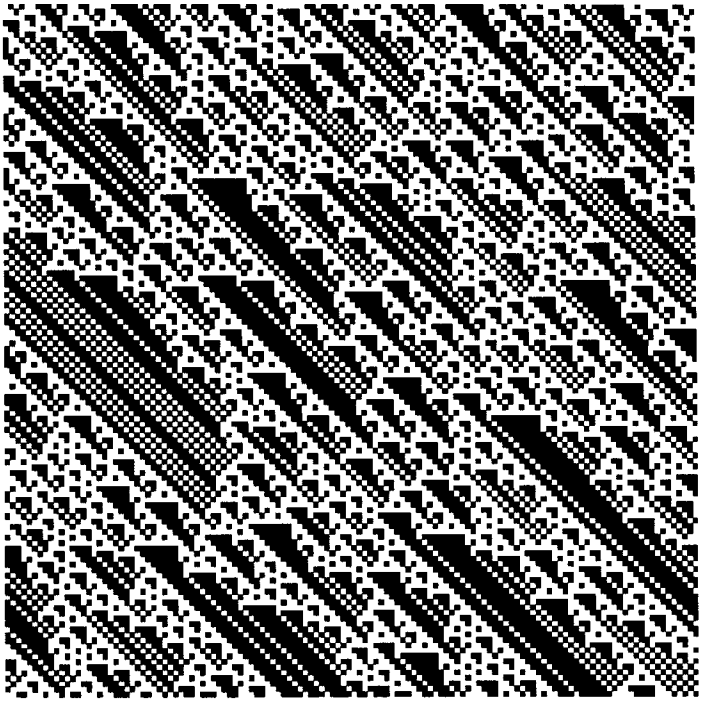


Figure 4: An automaton with complex behavior

Way I2: Not-Equals Scan

Gerald Langlet [3] was a not-equals scan enthusiast. This function can be thought of as giving the partial sums modulo 2. Iterating the not-equals scan function on a vector of ones gives the Sierpinski Triangle.

```
   v
1 1 1 1 0 0 1 1

   nes=:~:/\

   nes v
1 0 1 0 0 0 1 0

   nes^:(i.8) 8$1
1 1 1 1 1 1 1 1
1 0 1 0 1 0 1 0
1 1 0 0 1 1 0 0
1 0 0 0 1 0 0 0
1 1 1 1 0 0 0 0
1 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
```

We can flip this matrix if we so desire in order to obtain the Sierpinski Triangle opening toward the upper right, but we can also run the process backward. Rather remarkably, the negative iterates can be computed since J knows the inverse of not-equals scan.

```
   8{.1
1 0 0 0 0 0 0 0
   nes^:(-i.8) 8{.1
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1
```

Way 20: Deterministic Chaos Game

The chaos game is usually described as an iterative process of moving from a current position midway toward a vertex randomly selected from a fixed finite list of vertices. While we will consider the probabilistic version in the next Way, in this Way we consider a deterministic version where we keep a list of all possible current positions. We consider a list of three vertices T and compute a table of all midpoints of points of T with points of T.

```
   ]T=:0 0,0 1,:1 0
0 0
0 1
1 0
   mid=:-:@+"1
   T mid/ T
  0   0
  0 0.5
0.5   0

  0 0.5
  0   1
0.5 0.5

0.5   0
0.5 0.5
  1   0
```

We define dcg to be a function that creates that table, then uses adjoin insert , / to collapse the first axis, then uses nub ~. to remove duplicates, and then uses /:~ to sort the list of points for convenience.

```
   dcg=:/:~@:~.@:(,/)@:(mid/)
   T dcg^:2 T
   0    0
   0 0.25
   0  0.5
   0 0.75
   0    1
0.25    0
0.25 0.25
0.25  0.5
0.25 0.75
 0.5    0
 0.5 0.25
 0.5  0.5
0.75    0
0.75 0.25
   1    0
```

Figure 5: The deterministic Chaos Game

Figure 5 shows several iterates of this process using supersize pixels. One can create a bitmap array of these positions using the virtual raster array utilities in vra3.js from [8]. The following lines indicate how that might be done where the palette is the white and black palette defined in Way 1.

```
vspixel 0.05+0.9* T dcg^:5 T
vrashow ''
(pal;vra) writebmp8 'dcgvra.bmp'
```

In the above, we rescale points by 0.9 since vspixel doesn't allow coordinates to exactly equal 1 and add 0.05 to give a margin. The command vrashow '' should display the image in the graphics window opened when vra3.js was run and the last line saved the image with given file name.

The image shows dots arranged in the general form of the Sierpinski Triangle and indeed that is what it approximates. However, if this image were turned into a binary array, it is natural to use an odd number of rows and columns so that this really is a different approximation than the binary array approximations discussed in all the previous Ways.

Way 21: Probabilistic Chaos Game

The probabilistic version of the chaos game proceeds by randomly selecting one of the three vertices and then moving halfway toward that vertex. In particular, {&T can be used to select a vertex from T given an index and ?@3: can be used to randomly select the index. If this is only run a few thousand times, a shadowy version of the Sierpinski Triangle results. Here we show a few iterates of a point. Note that because of the random selections, the two experiments aren't duplicates. Hence, you shouldn't expect to duplicate the data given below.

```
   pcg=:mid {&T@(?@3:)
   pcg^:(i.6) 1 1
    1       1
    1     0.5
    1    0.25
  0.5   0.125
 0.75  0.0625
0.875 0.03125
   pcg^:(i.6) 1 1
      1       1
    0.5     0.5
   0.75    0.25
  0.375   0.625
 0.1875  0.8125
0.09375 0.40625
```

Way 22: Deterministic Iterated Function Systems

Iterated function systems consist of lists of functions. When discussing generation of fractals with these, the functions involved are maps that take the fractal onto some subpart of the fractal. Consider our Sierpinski Triangle in Figure 1 where we imagine the corners being the vertices in T. We can see that the map that halves both coordinates, -:, would map the entire fractal onto the half size "triangular part" of the fractal in the lower left. Note that the holes would line up. Likewise we could add one half to the x or y coordinates after halving and obtain the other two triangular parts appearing in the other corners. Thus we think of the Sierpinski Triangle as the union (or collage) of its image under those three maps. We create a list of those three maps as a gerund using tie ` and apply all three functions to given data with evoke gerund ` : 0. The list of those three functions and the boxed form for that gerund are given in Table I.

```
   ]ifs=:-:`(0 0.5&+@:-:)`(0.5 0&+@:-:)
```

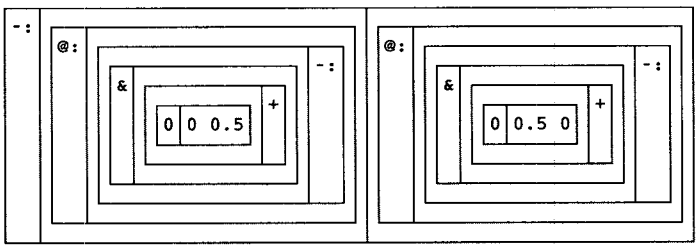


Table I. An iterated function system with three functions

```
   difs=:ifs`:0"1

   difs T
  0    0
  0  0.5
0.5    0

  0  0.5
  0    1
0.5  0.5

0.5    0
0.5  0.5
  1    0

   difsx=: ~.@:(/:~)@:(,/)@:difs
```

The function difsx can be used like dcg in Way 20 to create the Sierpinski Triangle.

Way 100: Probabilistic Iterated Function Systems

While Way 22 gives all the points arising from the initial triangle T under the transformations in the iterated function system, it is adequate to just look at random iterates of any single general point. Below we can choose a random index and use agenda @. to apply just that function. As with our previous probabilistic method, we don't expect to be able to duplicate experiments. Plotting tens of thousands of points generated this way gives the Sierpinski Triangle.

```
   pifs=:ifs@.(?@3:)

   pifs 0.5 0.5
0.25 0.25
   pifs 0.5 0.5
0.75 0.25
   pifs^:(i.4) 0.5 0.5
   0.5    0.5
  0.25   0.25
 0.125  0.625
0.0625 0.8125
```

Try it.

Way 101: Fractal Curve

We noted in the introduction that the Sierpinski Triangle was originally described as a curve. The curve results from the iteration of a refinement scheme: each segment is replaced at each refinement by five segments according to the pattern in Figure 6. In that figure, the segment at top is replaced by the segments below. If we start with an equilateral triangle and refine each segment into five subsegments and repeat the process, the Sierpinski Triangle results as we will see.

While the curve described here is different, the general construction style we will use for creating the curve follows [7,8]. First we define the monad "nor" that gives the left normal vector to a given vector. The function bump gives a point displaced from the midpoint of a segment a suitable distance in the normal direction. The suitable distance is one so that equilateral triangles are formed. The functions bump1 and bump2 give the two vertices off the original segment that appear in the bottom of Figure 6. The matrix tri gives the vertices of a roughly equilateral initial triangle.

```
nor=:1 _1&*@:|.
bump=:mid + (%:3r4)&*@nor@:-
bump1=:[ bump mid
bump2=:mid bump ]
tri=:0 0, 1000 0,: 500 866
```

Each segment is divided into new segments: the original left vertex, the midpoint, the first bump, the second bump, and then

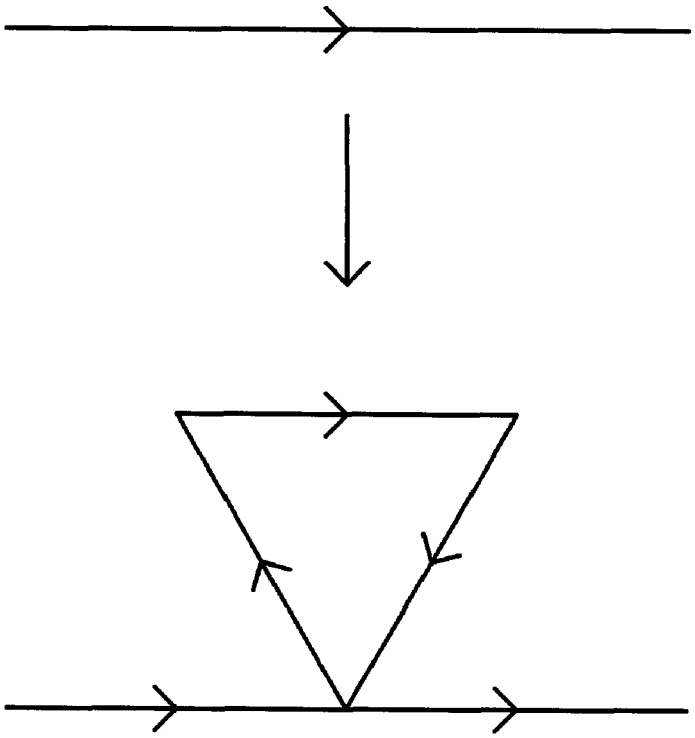


Figure 6: Refinement of a segment into five segments

back to the midpoint. We skip the last vertex since it will be produced from the refinement of the subsequent segment. The verb `refine` applies the segment division to each neighboring pair of vertices of a given polygon and adjoins the items on the first axis yielding a single polygon with five times the number of vertices.

```
   segdiv=:[ , mid , bump1 , bump2 ,: mid
   0 0 segdiv 1 0
   0        0
 0.5        0
0.25 0.433013
0.75 0.433013
 0.5        0
   refine=:,/@(] segdiv"1 (1&|.))
```

One can see the refinement of the triangle with `refine tri`.

We can plot that polygon using `gwin3.js` from [8] and the commands below.

```
sogwin 'curve'
spoly refine^:4 tri
```

Figure 7 shows that curve.

Alternately, one can do the same plotting with `plot.js` from the J `\packages\graphics` directory using `plot ;/|: (,{.) refine^: 4 tri`. See the tutorial/lab for how to vary the plot.

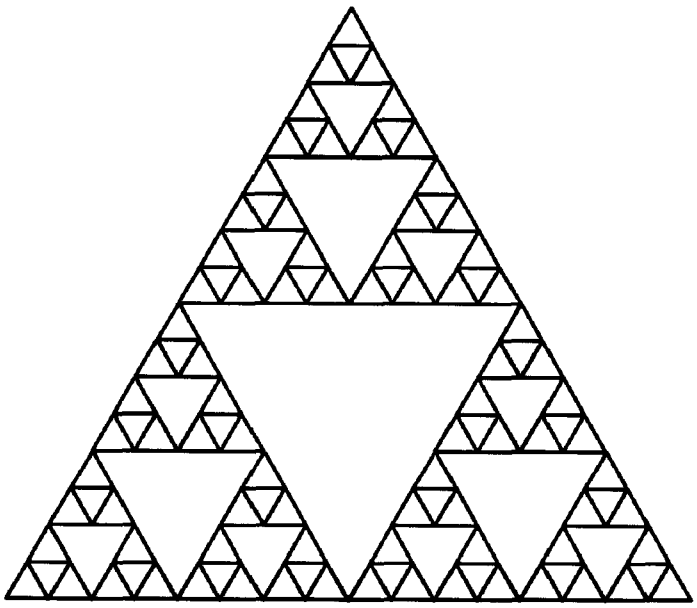


Figure 7: The Sierpinski Triangle as a curve

Since the result of this refinement is a polygon, it is amusing to consider the "inside" of the polygon. If `0 0 0` is given as the left argument of `spoly`, the polygon will be filled with black, as in Figure 8. How many vertices does the polygon you plotted have? What happens if you displace the bumps less? More? ■

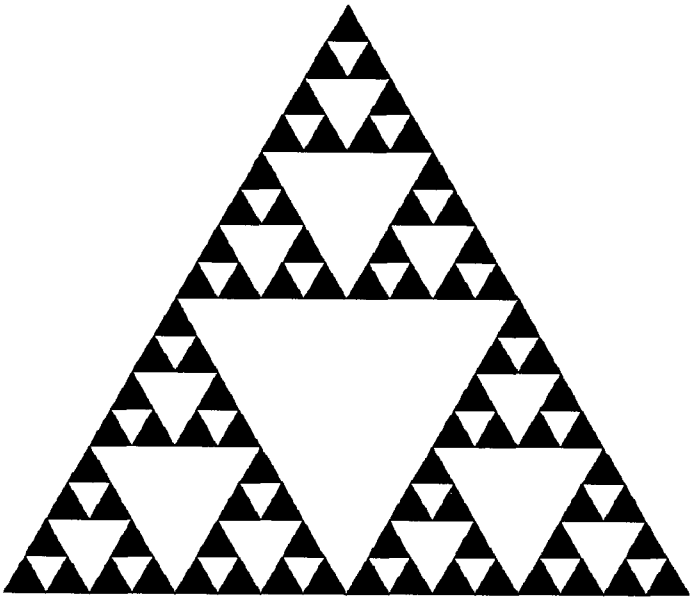


Figure 8: The interior of the Sierpinski Triangle

Acknowledgements

The support of Lafayette College and the provision of work and living space by the University of Waterloo and K. E. Iverson during the sabbatical leave during which this was written are greatly appreciated.

References

[1] N. Carter, R. Eagles, S. Grimes, A. Hahn, and C. Reiter, Chaos with Symmetry: Reflections on an Exhibition, Proceedings of the APL97 International Conference on APL, CD-ROM, Toronto, Ontario, August 17-20, (1997).

[2] K. Iverson, Concrete Math Companion, Iverson Software Inc, Toronto (1995).

[3] G. Langlet, The APL Theory of Human Vision, APL Quote Quad, 25 1 (1994) 105-121.

[4] A. Reiter, Determining the Dimension of Fractals Generated by Pascal's Triangle, The Fibonacci Quarterly, 31 2 (1993) 112-120.

[5] C. Reiter, Fractals and Generalized Inner Products, Chaos, Solitons & Fractals, 3 6 (1993) 695-713.

[6] C. Reiter, Sierpinski Fractals and GCDs, Computers & Graphics, 18 6 (1994) 885-891.

[7] C. Reiter, "Fractals RYIJ", Vector, 11 2 (1994) 86-104.

[8] C. Reiter, Fractals, Visualization and J, Iverson Software, Inc., Toronto (1995).

[9] C. Reiter, Infix, Cut and Finite Automata, APL Quote Quad, 25 4 (1995) 162-170.

[10] J. Shallit, Fractals, Recreation, and APL, APL Quote Quad 18 3 (1988) 24-32.

[11] J. Shallit and J. Stolfi, Two Methods for Generating Fractals, Computers & Graphics, 13 2 (1989) 185-191.

[12] W. Sierpinski (1915), Sur une Courbe Cantorienne dont tout Point est un Point de Ramification, Comptes Rendus Académie des Sciences, 160 (1915) 302-305.

[13] I. Stewart, Four Encounters with Sierpinski's Gasket, The Mathematical Intelligencer, 17 1 (1995) 52-64.

[14] M. Sved, Divisibility—With Visibility, The Mathematical Intelligencer, 10 2 (1988) 56-64.

***Cliff Reiter** teaches math at Lafayette College when he isn't wandering around the Adirondacks. He can be reached at "reiterc@lafayette.edu".*

We want *your* thoughts!

Inside the wrapper of this issue is a *Feedback Form*, asking for your comments about APL Quote Quad and SIGAPL. What did you *like* about this issue? What did you *dislike?* What else should we be working on? We'd really like to hear from you. Can you please take a minute and fill out this form? **If you no longer have the form,** just mail your comments to the Executive Editor of *APL Quote Quad* (or via e-mail, to "Polivka@ACM.org"). ... *Thank you!*

COVER STORY

Reviews of the APL97 Conference

THE APL97 CONFERENCE, entitled *Share Knowledge/Share Success*, was held on 17-20 August 1997 in Toronto, Canada, and was hosted by the Toronto APL Special Interest Group. The venue was a combination of presentations, tutorials, and workshops. For the first time at an APL conference, the proceedings and related materials were on a CD-ROM. There was no printed copy. A copy of the proceedings CD-ROM is available from The Toronto APL Special Interest Group (http://www.torontoapl.org).

We asked several people for reviews of the conference, in order to get balanced viewpoints. Following this overview of the conference are four separate reviews, from:

- **Eugene McDonnell** (from San Jose, California)
- **Timo Laurmaa** (from Basle, Switzerland)
- **Roy C. Willitts** (from Clarksburg, New Jersey)
- **Bob Brown** (from Thetford Center, Vermont)

We hope that you find the reviews to be as interesting as we did.

We extend our thanks to Richard Procter and his staff in Toronto, for not only making the conference a success, but also for supplying each of the photos of APL97 used in these reviews.

And for the benefit of those of you who were not able to attend the conference, we have also included a list of the papers that were presented at the conference. As you can see, it was a busy week!

APL97 wrap-up message

Thanks for Coming!

Once again the worldwide APL community has shown how to *Share Knowledge/Share Success*, as our conference theme suggested.

APL97 was an overwhelming success on many fronts. Total attendance was about 260 persons. Our plenary sessions, featuring Ian Sharp, founder of former software/timesharing giant I.P. Sharp Associates; and John Heinmiller, of actuarial software firm SS&C/Chalke, proved that APL has been and continues to be the secret weapon for success in the information technology industry.

—Richard Procter
Conference Chair

Attendance

APL97 Conference Registrations by Country:

Austria	2	Russia	2
Canada	110	Saudi Arabia	2
Denmark	12	South Africa	1
Finland	10	Sweden	1
France	5	Switzerland	3
Germany	6	UK	9
Italy	3	USA	82
Japan	7		
Netherlands	7	Total	263
Portugal	1		