Fractals and Visualization

for the J User Conference 2000

Clifford A. Reiter
Department of Mathematics
Lafayette College, Easton PA 18042 USA
reiterc@lafayette.edu, http://www.lafayette.edu/~reiterc

Many recent computer experiments have explored fractals. While fractals can be enjoyed for their visual intrigue, we can also discuss their uses. We will see they can be used to model complex behavior and visually identify correlations. Moreover, the richness of techniques available for creating fractals makes them an excellent source for illustrations of techniques for visualizing data and processes. We will also discuss some image processing techniques unrelated to fractals. In particular, we will discuss the following topics.

- an ideal fractal
- some random fractal walks
- plasma clouds
- the chaos game and genes
- chaotic attractors
- playful image processing
- local image processing
- deblurring using fast fourier transforms

Variants of many of these examples appear in *Fractals, Visualization and J*, 2nd edition (FVJ2); those variants sometimes offer further details and generalizations. Also, an article submitted to APL Quote Quad gives further details on the deblurring example. The presentation here is the nub of the conference presentation. However, it is meant to stand on its own, and it offers some new illustrations and points of view.

An Ideal Fractal

We begin with a construction of a Sierpinski triangle using juxaposition of arrays.

```
   ]m=:10*i.3 3
 0 10 20
30 40 50
60 70 80
```

a matrix

```
   (,,.~) m
 0 10 20  0  0  0
30 40 50  0  0  0
60 70 80  0  0  0
 0 10 20  0 10 20
30 40 50 30 40 50
60 70 80 60 70 80
```

the matrix juxaposed above and to the right; zero padding appears in the upper right

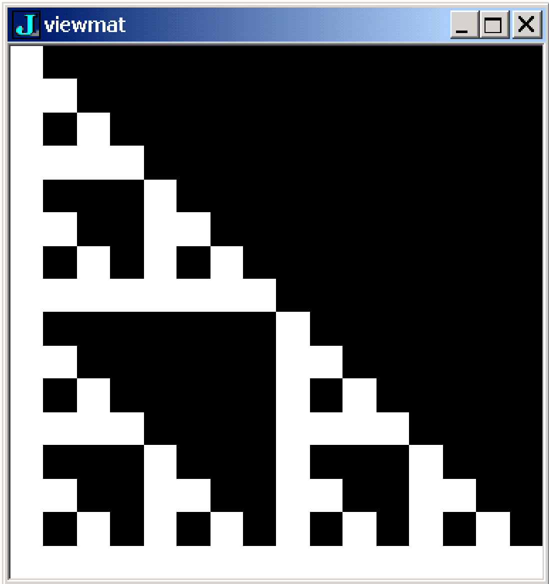

Figure 1. Four iterations toward a Sierpinski Fractal

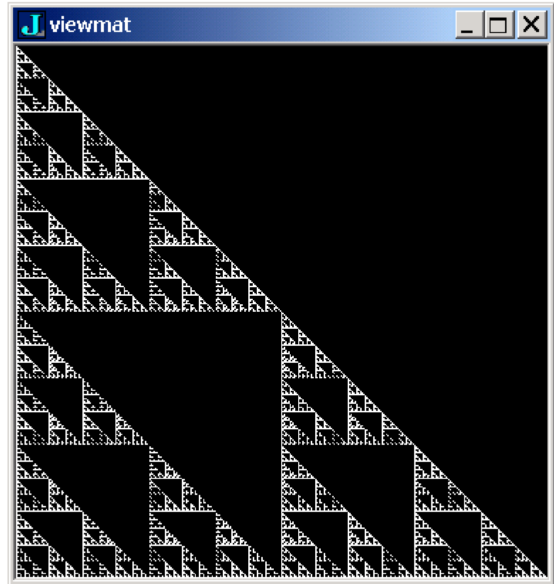

Figure 2. Nine iterations toward a Sierpinski Fractal

```
   (,,.~)^: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
```

juxaposition iterated four times on a singleton gives a Sierpinski fractal pattern

We can view these fractals as follows. Figure 1 shows four iterations of the construction process and Figure 2 shows nine iterations of the process. Notice the symmetry in the form of self-similarity.

```
   load 'graph'

   viewmat (,,.~)^:4 ,1

   viewmat (,,.~)^:9 ,1
```

Random Fractal Walks

In this section we consider some techniques for creating random walks with various Hurst exponents. Hurst was intrigued by the study of such time series while studying the flow of the Nile river. The persistence of flood and drought years caused his intrigue. In general, processes with Hurst exponent between 0.5 and 1 correspond to persistent processes and exponents between 0 and 0.5 correspond to antipersistent processes. We first create a few utilities: two create random numbers with uniform and standard normal distributions. The others interpolate given lists and add random normal numbers to a given array.

```
   load 'trig'

   randunif=: (? % <:)@:($&2147483647) :({.@[ +({:-{.)@[ * $:@])

   randunif 5 5              random 5 by 5 array of numbers
 0.131538   0.755605   0.45865  0.532767 0.218959
0.0470446   0.678865  0.679296  0.934693 0.383502
 0.519416   0.830965 0.0345721 0.0534616   0.5297
 0.671149 0.00769819  0.383416 0.0668422 0.417486
 0.686773   0.588977  0.930436  0.846167 0.526929

   randsn=: cos@+:@o.@randunif * %:@-@+:@^.@randunif

   randsn 10                 random list of 10 standard normal numbers
_1.46889 0.0371147 1.32234 _0.558556 0.00806478 0.732527
_0.279593 0.0520541 _0.454074 1.33969

   interp=: (}. + }:)@:(2: # -:)

   interp 1 3 10             interpolate a list
1 2 3 6.5 10
```

In order to obtain walks with nice properties, we follow the same construction as in FVJ2 and use the following for the original standard deviation perturbation on an array. Then the standard deviation for a general array is the product of that with the reciprocal of the size of the array raised to the Hurst exponent power. The verb `randadd` applies that random addition to an input array.

```
   osz=:%:@-.@(2&^)@+:@<:@[

   0.9 osz 1                 original size standard deviation for
0.359791                     random additions with this Hurst exponent

   0.1 osz 1
0.84429
```

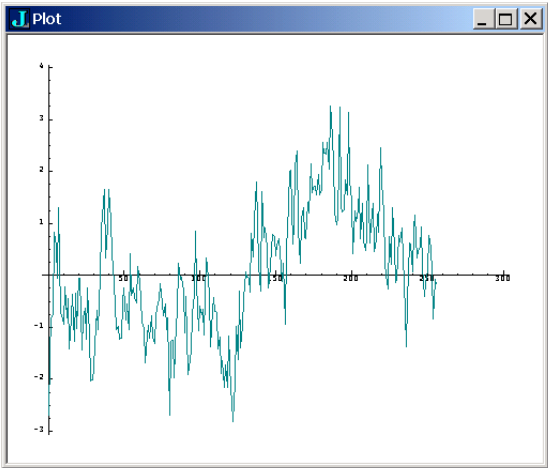

Figure 3. A Random Walk with Hurst Exponent 0.1.

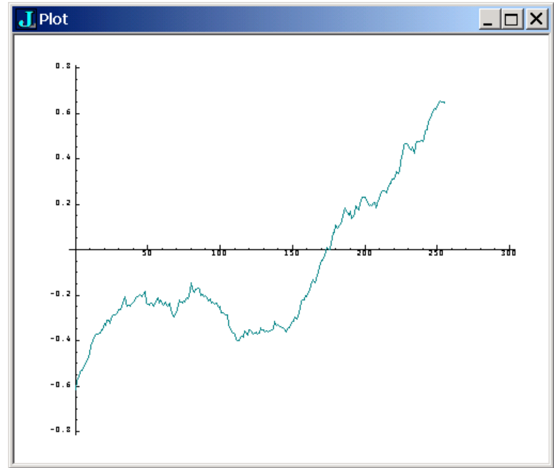

Figure 4. A Random Walk with Hurst Exponent 0.7.

```
sz=:osz * %@+:@<:@#@] ^ [

randadd=: ] + sz*randsn@$@]
```

The random walks are created by iterating interpolation and random additions upon an initial vector with two elements. Figure 3 shows the result when a Hurst exponent of 0.1 is used and Figure 4 shows the result when a Hurst exponent of 0.7 is used. Figure 5 shows Sunspot data. What do you think the Hurst exponent of the sunspot data is?

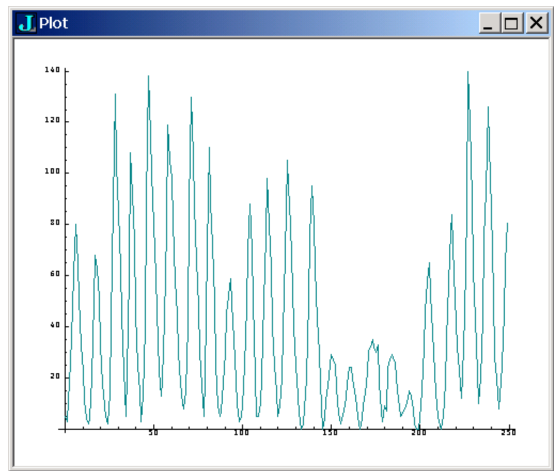

Figure 5. Some Sunspot Dat.

```
hwalk=: 4 : 'x. ([randadd interp@])^:y. (x. osz 1)*randsn 2 '

load 'plot'

plot 0.1 hwalk 8                create Figure 3

plot 0.9 hwalk 8

plot 0.7 hwalk 8                create Figure 4

load 'fvj2\ts_data'

plot 250{.sunspots              create Figure 5
```

Plasma Clouds

It is interesting that only very small changes to the 1-dimensional random walk functions need to be made in order to create a 2-dimensional random walk (also called a plasma cloud). In particular, we need to interpolate along both axes and begin with an initial matrix rather than a list. We will show the resulting arrays with false coloring by hues (that is, colors ranging from red to magenta correspond to the height of the walk. In order to make the image suitable for saving as an 8-bit raster array, it is convenient to make those heights discrete. We accomplish that with the `cile` function.

```
   interp2=:interp"1@:interp
```
interpolate rows and columns

```
   hwalk2=:4 : 'x.([randadd interp2@])^:y.(x. osz 1)*randsn 2 2'
```

```
   0.1 hwalk2 2
  0.694055  0.965148  0.813761    1.7371  3.68347
  0.264332 _0.578903   1.30879  0.183548 0.804614
   _1.1384   _1.70865 _0.206643  0.15884 0.183516
  0.166768  0.496437 _0.760414  _1.79932  1.18048
_0.0255202 _0.312702 _0.513639 _0.642307 _0.99058
```
a 2-dimensional random walk

```
   load 'fvj2\raster4 fvj2\povkit'
```
load raster utilities

```
   lv=: 3 : 0
wd 'winexec "c:\win_apps\lview\lview31.exe ',y.,'"'
)
```
a function to run a viewer

```
   p=:hue 5r6*(i.%<:)256
```
a 256 color palette

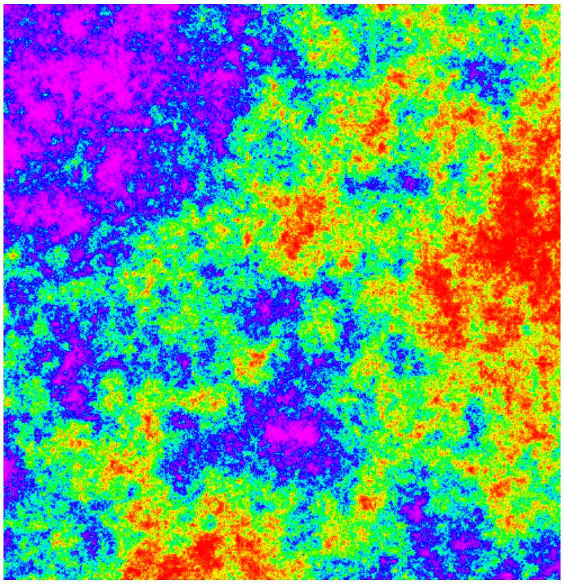

Figure 6. A Plasma Cloud with Hurst Exponent 0.2.

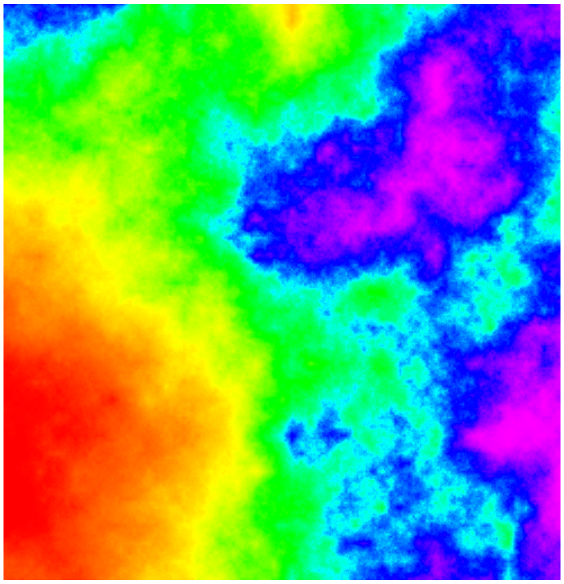

Figure 7. A Plasma Cloud with Hurst Exponent 0.7.

```
   (p; 256 cile 0.2 hwalk2 9) writebmp8 'fig6.bmp'

   lv 'fig6.bmp'
0
   (p; 256 cile 0.7 hwalk2 9) writebmp8 'fig7.bmp'

   lv 'fig7.bmp'
0
```

Figures 6 and 7 show the result of a plasma cloud with Hurst exponent 0.2 and 0.7. Notice that both clouds are coherent, but that the variation is much slower with the higher Hurst exponent.

The Chaos Game and Genes

The chaos game is an image building technique that plots a sequence of points. Each point is halfway between the previous point and some randomly selected vertex of a fixed polygon. We will restrict our attention to the unit square, with its four vertices, as our fixed polygon.

```
   ]sq=:#:0 1 3 2                 the four vertices of the unit square
0 0
0 1
1 1
1 0

   mid=:-:@+                      function for computing midpoints

   0.2 0.2 mid 0 1                a midpoint computed
0.1 0.6

   ]r=:(?.4#4){sq                 four random vertices
0 0
1 0
0 1
1 1

   mid/\.r                        the chaos game on those four points
0.375 0.25                        starting from the bottom
 0.75  0.5
  0.5    1
    1    1
```

The previous computation uses suffix scan in order to do the midpoint inserts efficiently. Now we apply and plot the results for 300,000 randomly selected vertices. The result is shown in Figure 8. Notice the result is a uniformly filled square (up to randomness). Figure 9 shows the result when only the first three vertices are chosen. Notice the Sierpinski Triangle results. Figure 10 shows the result when all four vertices are allowed, but a bias in favor the first three is used.

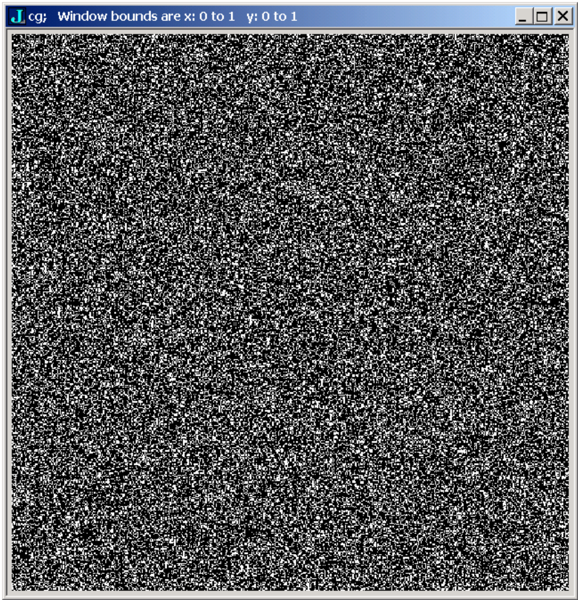

Figure 8. The Chaos Game Using Four Vertices.

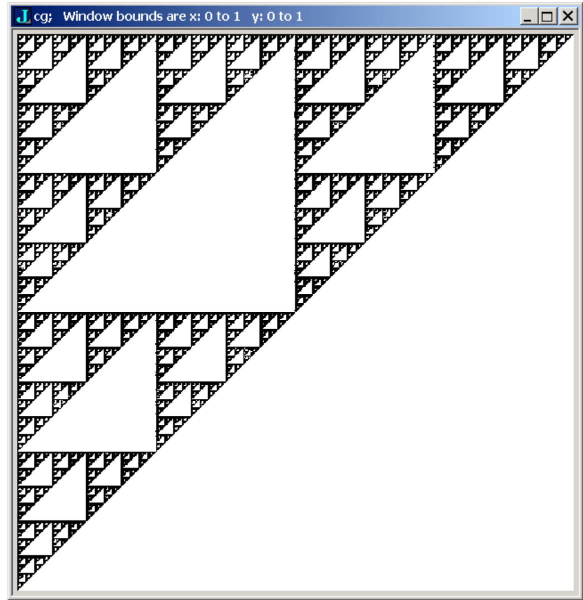

Figure 9. The Chaos Game Using Three Vertices.

Notice the square is filled, but shadows of the Sierpinski triangle are visible. The main point is that deviations from the randomly filled square correspond to bias in the data.

```
load 'fvj2\raster4'

vwin 'cg'[wd 'reset;'

vshow vpixel mid/\. sq{~ ?300000$4          create Figure 8

vwin 'cg'[wd 'reset;'

vshow vpixel mid/\. sq{~ ?300000$3          create Figure 9

vwin 'cg'[wd 'reset;'

vshow vpixel mid/\. sq{~ 4|?300000$11       create Figure 10
```

We now turn to using DNA data to select the order of our vertices. Figure 11 shows some sample DNA data. We create our list of vertices by only keeping the entries in `'cagt'` from the DNA file text.

```
open 'c:\j\405b\user\dna_y54g9.ijs'         shown in Figure 11

load 'c:\j\405b\user\dna_y54g9.ijs'

dna=:(-.~.@(-.&'cagt'))Y54G9                the c-a-g-t gene sequence
```

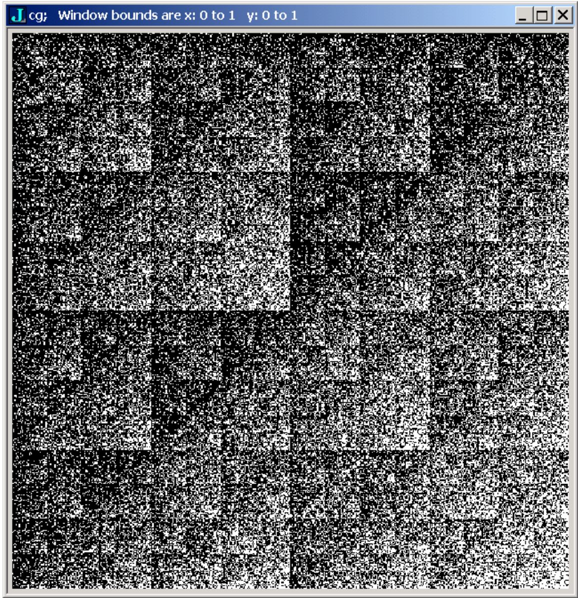

Figure 10. The Chaos Game Using Four Vertices with a Bias Toward Three.

```
vwin 'cg'[wd 'reset;'

vshow vpixel mid/\. sq{~ 'cagt' i. dna
```
create Figure 12

Notice dark upper left and lower right corners. They show that sequences of a's and t's seem to appear more often than expected if they were random.

dna_y54g9.ijs

```
NB. http://nt-salkoff.wustl.edu/Highend.htm
NB. K+ and Ca++ Channels in C. elegans

Y54G9=:0 : 0
        1 tggaataaac gttcgatcaa cgtatagaaa ggaaacgatt tctttttccg gcaaatcggc
       61 aaattgccgg aattgaaaat ttcgggcaaa tcggcagatt gccggaatta aaaatttcag
      121 gaaaatcagc aaattgccgg aattgcaaat ttccggcaaa tcggcaaatt gcgggatttg
      181 aaaatttccg gcaaaccggc aaattgccgg aattgaaatt tccagcaaat ttgcaaattg
      241 ccggaattga aatttccagc aaatttgcaa attgccggaa ttgaaaattt ccagcaaatt
      301 tgcaaattgc cggaattgaa aatttccggc aaatcggcaa acacgaagtt ttcatttttt
      361 ggcaaattgc cgacttgcca gaaattttca ttttcgtcat attgccgatt tgccggaaaa
      421 aatcaacact tccggcgaac ggcaattcag caaattgccg aaaatcaaaa tttttccgga
      481 actgaaattt ccggcaaatt ggcaaaccgg caaatttcgg gcaaatctgc aaattgccgg
      541 aattgcaaat ttcagcccaa gaacaccgat ttcccaatta gtcaacaatt tccaatttcc
      601 gagtcacact ccgcgtgttt tttcttgtat cggaaaaccg gcaatttgca gatttgccga
      661 actgcaattg ccccccgccc actcctgagt cagacacttc ttctaaaagt gaacgtttgc
      721 ccaatttttg actcgaacat gatgttctgt caaaaatcgt ttcgtcgccg cttaaacatc
      781 actgattgta cgtgtaaatg gaagctaatc acactaaaca cactgccaaa gagatagtgg
      841 tgtagtggtg atctgtggtt tgtcttacaa actccaacga tcagaagtta cgggtgggtg
      901 tcaaacgatt ttttccggca aattggcaag ttgctagaat taaaaatttc cggcaaatcg
      961 gcaaaccggc aaattctcgg aattgaaatt tccggcaagc cggcaatttg ccaaaaaatg
     1021 aaattttcca gcaaatcagc aaatcaacaa atttccgaga ttgaaaaatt tccggcaatt
     1081 cggcaaacct gcaatttgcc gcatttatcg ataaaaacgt tgccaattgt cgcccacctc
     1141 tgatatatac aagtgcaagt ttcaataaat gtttctgcac cggctccctg attaaggctg
```

Figure 11. A Script File Containing the Amino Acid Sequence for a Gene.

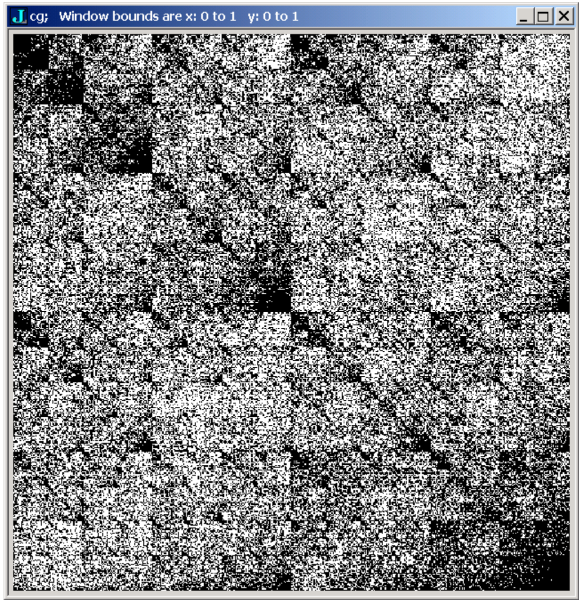

Figure 12. The Chaos Game Showing the Bias in a DNA sequence.

Chaotic Attractors

Chapter 5 of FVJ2 is devoted to creating chaotic attractors. This includes attractors with no special symmetry and those with a great variety of symmetry types. These include cyclic symmetry, frieze pattern symmetry (repetition of a motif along a line) and all of the symmetries of the planar crystallographic groups (wallpaper symmetries). Some hyperbolic symmetry groups are considered as well. Examples from FVJ2 illustrating chaotic attractors include Figure 13, which has frieze symmetry, Figure 14, which has crystallographic symmetry with a kind of 3-fold rotational symmetry on a hexagonal lattice, and Figure 15 which has hyperbolic symmetry arising from an iterated function system.

We will illustrate the constructions via an example with p4g symmetry. That crystallographic group has 4-fold rotational symmetry along with glide reflections. We begin by constructing some random functions in the plane. The adverb `DF` constructs a double (2-variable) Fourier series with given coefficients. The function `mkrandf` creates a random function with coefficients selected in the specified range.

```
   load 'fvj2\chaotica trig'

   four=: 1: , sin , cos , sin@+: , cos@+:

   four 1r3p1                          Fourier sequence at a point
1 0.866025 0.5 0.866025 _0.5

   DF=:1 : '1: | ((m."_ +/ . * {: )+/ . * {.)@:(four"0)@(2p1&*)'

   mkrandDF=:2 : '((m.*_1 1) randunif 2 5 5) DF'
```

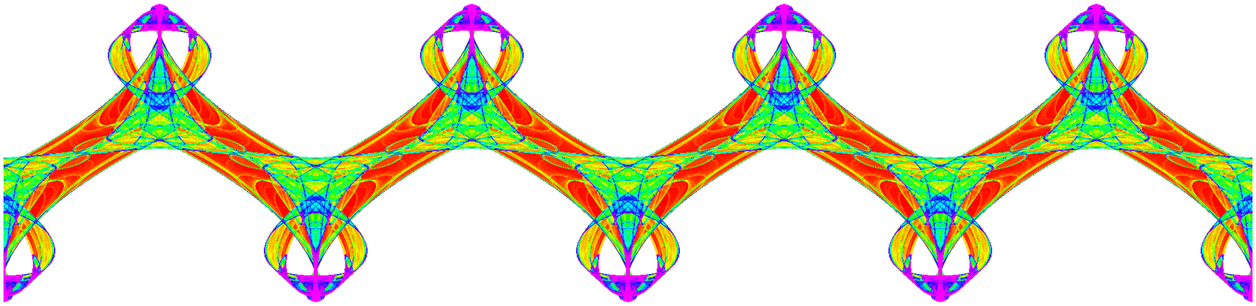

Figure 13. A Chaotic Attractor with Frieze Symmetry.

```
   mkrandf=:0.14 mkrandDF

   f=:'' mkrandf                          a random double Fourier series function

   f^:(i.5) 0.1 0.4                       its iterates have no special structure
      0.1       0.4
 0.502265 0.739905
0.0989961  0.92448
 0.997826 0.558391
 0.792996 0.135455
```

The main chaotic attractor search function defined in *chaotica.ijs* is `ca_create`. It requires several global definitions. First, a verb that gives the window bounds for the plot window is required. Here the plot window will always be the unit square. Second, some measure of the fullness of the window is required. Here we check whether every row and column has been visited (and append the number visited).

```
   winsq=: 0 0 1 1"_

   winbd=: winsq                          window bounds are the unit square

   fullsq=: 3 : '(([:*./#=+/),+/)(+./,.+./"1)*y.'

   fullness=:fullsq                       tests if all rows and columns are visited
```

Now we turn to the construction of the symmetries of p4g. The generators of the symmetry group may be found in the International Tables of Crystallography. First we consider two matrices that specify symmetries in homogeneous coordinates. The first is a central inversion (which is a sign change in both coordinates). The second is a 4-fold rotation around the origin. We then take matrix products of the generators repeatedly until no new elements are found. (This computes the group closure).

```
   ]m0=:_1 0 0,0 _1 0 ,:0 0 1
_1  0 0
 0 _1 0
 0  0 1

   ]m1=:0 _1 0,1 0 0,:0 0 1
0 _1 0
1  0 0
0  0 1
```

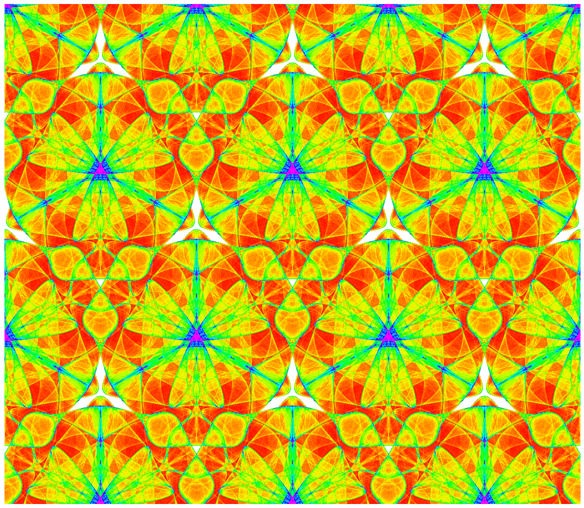

Figure 14. A Chaotic Attractor with Crystallographic Symmetry

```
   prods=:\:~@~.@(**|)@([,(,/)@:((+/ . *)"2/))
```

```
   prods~ m0,:m1
 1  0 0
 0  1 0
 0  0 1

 0  1 0
_1  0 0
 0  0 1

 0 _1 0
 1  0 0
 0  0 1

_1  0 0
 0 _1 0
 0  0 1
```

we get
four matrices

```
   prods^:_~ m0,:m1
 1  0 0
 0  1 0
 0  0 1

 0  1 0
_1  0 0
 0  0 1

 0 _1 0
 1  0 0
 0  0 1

_1  0 0
 0 _1 0
 0  0 1
```

same four allowing
more iterations

The third generator for the p4g symmetry group is a glide reflection. That is, a reflection through the first variable and translation by half a cell along a diagonal. This is given by the matrix `m2`. Since iterating matrix products of that matrix with itself would lead to arbitrarily long translations, we reduce those translations mod 1 (using `tr`) in the function `Prods`.

```
   ]m2=: _1 0 0, 0 1 0,:_0.5 0.5 1
  _1    0 0
   0    1 0
_0.5  0.5 1
```

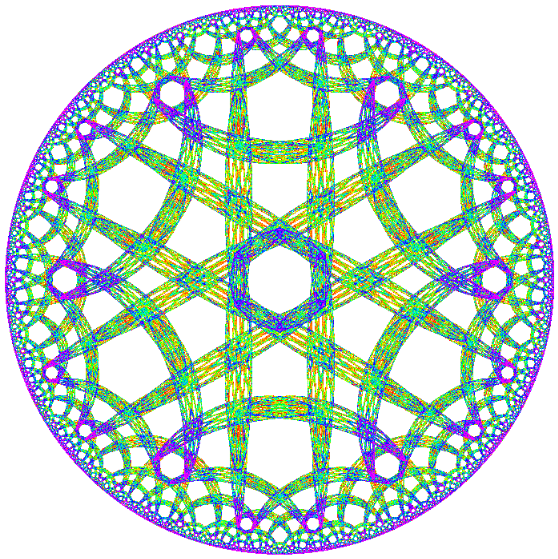

Figure 15. An Attractor with Hyperbolic Symmetry Coming from an Iterated Function System.

```
   ]tr=:0,0,:1 1 0
0 0 0
0 0 0
1 1 0

   Prods=:[: ~. tr"_ |"2 prods

   $p4g=:Prods^:_~ m0,m1,:m2
8 3 3
```

there are eight elements of the symmetry group (modulo the mod 1 reduction)

```
   2 4$<"2 p4g
+-----+----------+----------+-------+
|1 0 0|  1    0 0|  0    1 0|  0 1 0|
|0 1 0|  0   _1 0|  1    0 0| _1 0 0|
|0 0 1|0.5  0.5 1|0.5  0.5 1|  0 0 1|
+-----+----------+----------+-------+
|0 _1 0|  0   _1 0| _1    0 0|_1  0 0|
|1  0 0| _1    0 0|  0    1 0| 0 _1 0|
|0  0 1|0.5  0.5 1|0.5  0.5 1| 0  0 1|
+------+----------+----------+-------+
```

We obtain a function with the desired symmetries by taking the identity function plus a sum of conjugates of a given function modulo 1. Conjugates are like J's under conjunction, since the inverse symmetry is applied to the given function which was applied to the symmetry. There are a variety of details (see FVJ2 for more) since the function takes pairs as input and applying the matrix symmetries requires homogeneous coordinates.

```
   crycon=: 2 : 0
x=.+/ . *
1: | ] + }:@(+/)@(] x"1 2 (%.n.)"_)@:((,1:)@u.@}:"1)@(] x"1 2
n."_)@(,1:) f.                        (run-on line continued)
)

   mkrandcry=: 2 : ' m. mkrandDF 0 crycon n.'

   mkrandf=: 1 : '0.03 mkrandcry p4g'

   f=:'' mkrandf                      function with p4g symmetry

   s=:] +/ . *"_ 2 p4g"_              we can observe the symmetry

   1|s (,1:) f 0.1 0.4                   f"1 }:"1 s 0.1 0.4 1
0.23864 0.26136 0                     0.23864 0.26136
0.73864 0.23864 0                     0.73864 0.23864
0.76136 0.73864 0                     0.76136 0.73864
0.73864 0.23864 0                     0.73864 0.23864
0.26136 0.76136 0                     0.26136 0.76136
0.23864 0.26136 0                     0.23864 0.26136
0.26136 0.76136 0                     0.26136 0.76136
0.76136 0.73864 0                     0.76136 0.73864
```

We can use `ca_create` to create functions of this type and it results in low quality sample images. We create 3 images below (setting the random seed so the experiment can be replicated).

```
   (9!:1) 2050148813                  set the random seed

   3 ca_create '\temp\5\pgma'
k: 0 ful: 186 186 L: 0.536602 _0.452262
k: 1 ful: 500 500 L: 0.218792 _0.064359
k: 1 ful: 81 81 L: 0.306514 _0.48405
k: 2 ful: 500 500 L: 0.562975 _0.697111
k: 2 ful: 306 306 L: 0.199107 _0.761448
k: 3 ful: 500 500 L: 0.375233 _0.0573264
3
```

Upon viewing the created files, it appears that *p4g001.bmp* might be the most interesting. Thus we create a higher resolution version as follows. We then use the function `tilebmp` to tile the high resolution version.

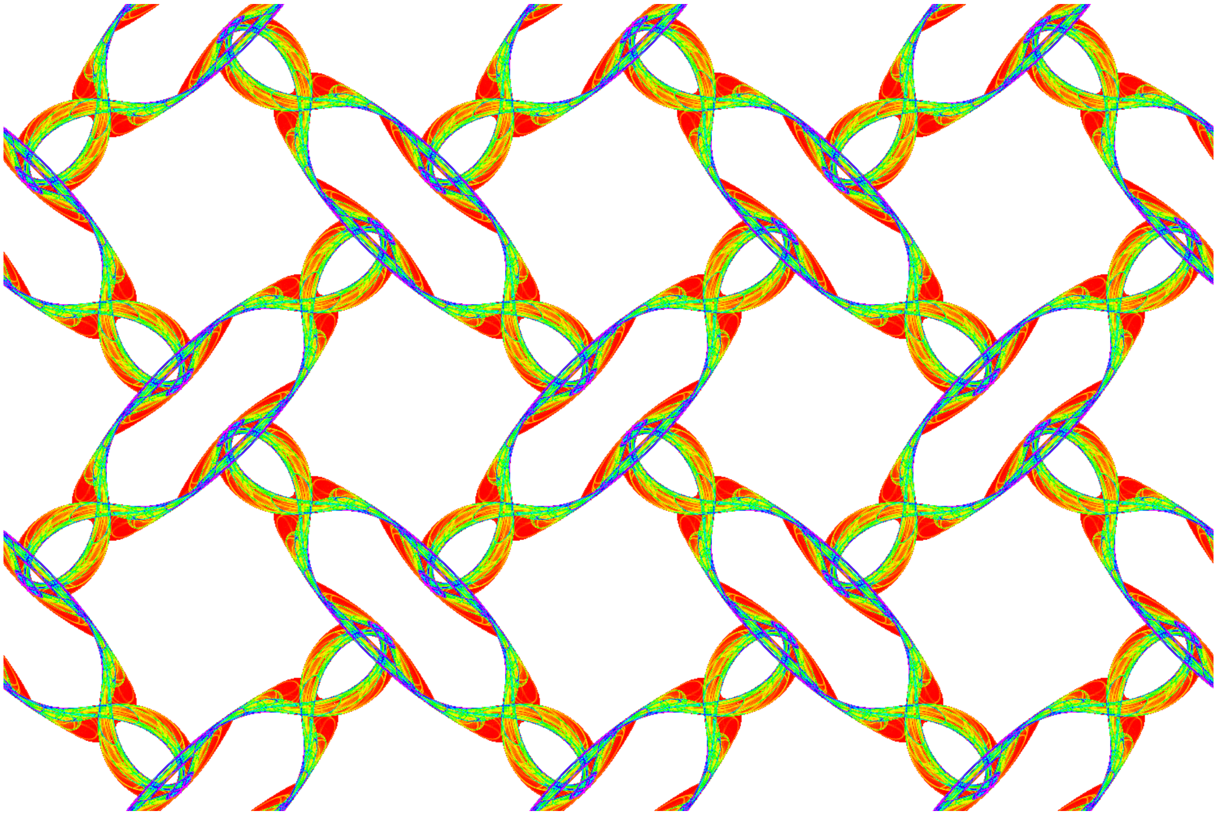

Figure 16. A Chaotic Attractor with p4g Crystallographic Symmetry.

```
   f=:fp4g001                           set global f to be the desired function

   ca_hr 'p4g_h'                        begin high resolution iteration

   (10#,:20 200000) ca_hr_add 'p4g_h'      add 40,000,000 iterates

   tilebmp=:3 : 0"1
2 2 tilebmp y.
:
'p b'=.readbmp8 y.
'r s'=.<.x.*$b
b=.s$"1 r$b
(p;b) writebmp8 y.
)

   2 3 tilebmp 'p4g_h.bmp'
```

The resulting image is shown in Figure 16.

Figure 17. The Stone Building Image.

Playful Image Processing

We use the image *stones.bmp* from FVJ2 for some simple and fairly direct image processing. Figure 17 shows the original image. Figure 18 shows the images after being grayscaled, Figure 19 shows the negative image, Figure 20 shows the color planes permuted, and Figure 21 shows the image with the red-green and blue color planes rotated slightly out of synchronization.

```
   lv '\j\404a\fvj2\stones.bmp'
0

   $B=: readbmp24 'fvj2\stones.bmp'        read the stones image as an array
373 562 3

   (3&#@<.@(+/%#)"1 B) writebmp24 'fig18.bmp'

  lv 'fig18.bmp'
0

   (255-B) writebmp24 'fig19.bmp'
   lv 'fig19.bmp'
0

   (1 2 0&{"1 B) writebmp24 'fig20.bmp'
   lv 'fig20.bmp'
0

   (|: 10 0 _10 |."0 2 |:B) writebmp24 'fig21.bmp'
   lv 'fig21.bmp'
0
```

Figure 18. The Grayscale Image.

Figure 19. The Negative Stone Building Image.

Figure 20. The Stone Building Image with Colors Permuted.

Figure 21. The Stone Building Image with Color Planes Rotated.

Local Image Processing

As our last example of image processing with the *stones.bmp* image, we locally average the red, green, blue triples using _3 cuts on 5 by 5 tessellations. This gives a blurred image, but removes some artifacts: most of the glare reflections from the back of the vertical stones has been removed. The result is shown in Figure 22.

```
   3 3 <;._3 i. 5 5
```

boxing a 3 by 3 tessellation

```
┌────────┬────────┬────────┐
│ 0  1  2│ 1  2  3│ 2  3  4│
│ 5  6  7│ 6  7  8│ 7  8  9│
│10 11 12│11 12 13│12 13 14│
├────────┼────────┼────────┤
│ 5  6  7│ 6  7  8│ 7  8  9│
│10 11 12│11 12 13│12 13 14│
│15 16 17│16 17 18│17 18 19│
├────────┼────────┼────────┤
│10 11 12│11 12 13│12 13 14│
│15 16 17│16 17 18│17 18 19│
│20 21 22│21 22 23│22 23 24│
└────────┴────────┴────────┘
```

```
   lavg=:<.@(+/ % #)@(,/)
```

apply this averaging on a 5 by 5 tessellation

```
   (,/"2]5 5 lavg ;._3 B) writebmp24 'fig22.bmp'

   lv 'fig22.bmp'
0
```

Figure 22. The Stone Building With Local Averaging.

Deblurring Using Fast Fourier Transforms

Our last illustration uses fast Fourier transforms to remove motion blur. The basic idea is that we compute the Fourier transform of the blurred image and do a modified divide by the Fourier transform of a line segment representing the blur. The magnitude of the inverse transform of that quotient is the deblurred image. In practice, we also need to recenter the image.

```
   lv '\j\404a\fvj2\blur.bmp'          view the blurred image
0

   'p b'=:readbmp8 '\j\404a\fvj2\blur.bmp'

   $b
834 834

   lv '\j\404a\fvj2\line.bmp'
0
   'p lin'=:readbmp8 '\j\404a\fvj2\line.bmp'

   load 'addons\fftw\fftw'             load fast fourier transform package

   fb=: fftw b                         transform of the blurred image

   fl=: fftw lin                       transform of the line

   fi=:(fb * + fl)%(*:|fl)+10*255^2

   i=:fftw^:_1 fi

   (p;<.255*(]%>./@,)|i) writebmp8 'temp.bmp'

   lv 'temp.bmp'                       deblurred, but incorrectly centered
0

   hr=:-:@# |. ]                       recenter an axis

   HR=:hr"1@hr                         recenter a matrix

   (p;HR <.255*(]%>./@,)|i) writebmp8 'fig25.bmp'

   lv 'fig25.bmp'
0
```

Figure 23. The Motion Blurred Image.

Figure 24. The Line of the Blur.

Figure 25. The Deblurred Image.