images and record how you created them for future reference. How do the striking constructions you found for base-two appear when base-three is used instead?

Lastly, generalize these experiments to higher bases. Create some images of this type with base-four or higher used to create `M` . Try various combinations of other functions. How do the striking constructions you found for base-two and three appear when base-four is used instead? How about base-five or six?

5.7 Inverse Iterated Function Systems

The iterated function system we associated with the Sierpinski triangle involved three functions, each function involves a contraction by half and may involve a translation. Thus, the inverse maps involve doubling and translation. We divide the unit square into four regions and consider a function defined by suitable inverses (or infinity) in each region. Points typically get large quickly under the iteration of these functions. An image of the number of iterates required for each point shows the escape time and our goal for this section is to create an escape time image for this inverse iterated function system.

As a warm-up, consider the doubling function. We iterate it, storing the successive iterates, until the last iterate is larger than 10.

```
   f=: +:                                 double

   (,f@{:)^:(<&10@{:)^:_ ] 1              iterate until last exceeds 10
1 2 4 8 16
```

The expression `F^:G^:_` applies `F` repeatedly until the Boolean function `G` results in false. The function `F=:,f@{:` adjoins its argument to the result of `f` applied to the last item in its argument. Thus, the expression gives the list of iterates of `f` until the specified size is exceeded. Next we want to generalize this construction to points in the plane. Here our test of largeness is whether the sum of the magnitude of the coordinates is too large.

```
   (,f@{:)^:(<&10@:(+/)@:|@{:)^:_ ,:0.3 1
0.3 1
0.6 2
1.2 4
2.4 8
```

We also can bound the maximal number of items by 3 with the following expression.

```
   (,f@{:)^:(<&3@# *. <&10@:(+/)@:|@{:)^:_ ,:0.3 1
0.3 1
0.6 2
1.2 4
```

Now we implement a general conjunction that performs the iteration illustrated above and results in the number of items required. Note, since the first item is the initial data, the number of iterates is one less than the number of items. We also ensure that the number of iterates (via the number of items) does not exceed some maximum value. In particular, the left argument of the conjunction is the function being iterated and the right argument is a list: the maximal number of allowed iterates and the bound for largeness.

```
   escapet=: 2 : 0
#@((,u@{:)^:(<&({:n)@#*.(<&({.n)@:(+/)@:|@:{:))^:_)@,:f."1
)

   (f escapet 10 100) ,:0.3 1             we saw this gave 4 items (took 3 iterates)
4

   (f escapet 10 3) ,: 0.3 1              we can stop when 3 is exceeded
3
```

We return to our inverse iterated function system. First we define the three inverses and then we define a function that divides the plane into four quadrants, but these are quadrants meeting at (0.5,0.5), not the standard quadrants that meet at the origin.

```
   i0=: +:                      inverse function 0
   i1=: +:@(-&0 0.5)            inverse function 1
   i2=: +:@(-&0.5 0)            inverse function 2
   i3=: _"0                     inverse function 3
   quad=: #.@(>&0.5)            identify one of the special quadrants
   quad 0.1 0.1                 lower left
0
   quad 0.2 0.6                 upper left
1
   quad 0.6 0.2                 lower right
2
   quad 0.6 0.8                 upper right
3
```

Now the inverse functions are applied in the corresponding quadrants.

```
   isier=: i0`i1`i2`i3@.quad        inverse iterated function system
   isier 0.1 0.1                    a point from lower left
0.2 0.2
   isier 0.2 0.6                    a point from upper left
0.4 0.2
   isier 0.6 0.2                    a point from lower right
0.2 0.4
   isier 0.6 0.8                    a point from upper right
_ _
   isier^:(i.4)0.2 0.4              iterates get large in 4 steps
0.2 0.4
0.4 0.8
0.8 0.6
  _   _
   (isier escapet 10 100) ,:0.2 0.4
4
   <"1 |.,"0~/~(i.%<:)5             table of positions in the unit square
+------+---------+--------+---------+------+
|0 1   |0.25 1   |0.5 1   |0.75 1   |1 1   |
+------+---------+--------+---------+------+
|0 0.75|0.25 0.75|0.5 0.75|0.75 0.75|1 0.75|
+------+---------+--------+---------+------+
|0 0.5 |0.25 0.5 |0.5 0.5 |0.75 0.5 |1 0.5 |
+------+---------+--------+---------+------+
|0 0.25|0.25 0.25|0.5 0.25|0.75 0.25|1 0.25|
+------+---------+--------+---------+------+
|0 0   |0.25 0   |0.5 0   |0.75 0   |1 0   |
+------+---------+--------+---------+------+
```

Now we create a 1024 by 1024 array of escape times and plot them with a carefully chosen palette. Notice that we allow no more than 100 iterates. Since each point is in the square, the smallest possible escape time is 2. Thus we chose a palette with two white entries that never occur, 9 hues corresponding to the 9 lowest occurring escape times, and then the rest of the palette entries are black.

```
    xy=: |.,"0~/~(i.%<:)1024
    b=: isier escapet 10 100 xy
    h=:Hue 5r6*(i.%<:)9
    pal=: 255,255,h,90 3$0
    view_image c=:pal;b
```

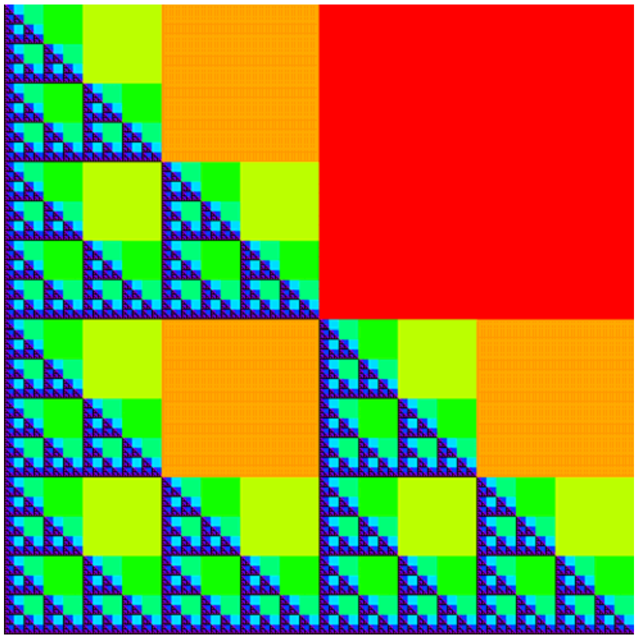
Figure 5.7.1 Inverse IFS

Figure 5.7.1 shows the resulting image. This plate illustrates the construction of the Sierpinski triangle in a natural manner. Namely, the Sierpinski triangle is the result of the process which begins with the removal of the upper right quarter of the square; that is, the red square is removed. Then the upper right corner of each remaining square is removed; that is, the orange squares are removed. Then the upper right corner of each remaining square is removed; that is, the yellow-green squares are removed. This continues until just the Sierpinski triangle, rendered as the black portion (almost indistinguishable from the adjacent violet region) remains.

5.8 Exercises

1. Create a 256 color palette that
(a) blends from red to white
(b) blends from red to black
(c) blends from red to magenta
(d) blends from blue to red
(e) has white as its first color, black as its last color, and that blends from cyan to blue in the middle 254 colors.
(f) has white as its first color, black as its last color, and that runs through the hues from red to red in the middle 254 colors.
(g) has white as its first color, black as its last color, and that runs through the hues from red to magenta in the middle 254 colors.

2. Decide the result of the following expressions and verify your answer with J.
(a) `'abcdef' i. 'dad'`
(b) `'abcdef' i. 'daddy'`
(c) `1 1 0 1 i. 1`
(d) `1 1 0 1 i. 0`
(e) `#: 7 10`
(f) `3 3 3 #: 7 10`
(g) `#. 1 1 0 1`
(h) `3 3 3 3 #. 1 1 0 1`

3. Two of the constructions suggested in Section 5.6 are described again as follows.
Let `M=:#:i.2^8` and consider the bitmaps constructed by the following.

```
(I)    $b=: M #.&|: . * |: M
       pal=: 0,Hue 5r6*(i.%<:)255
(II)   $b=: M i.&1"1&|: . * |: M
       pal=: 0,~Hue 5r6*(i.%<:)8
```

(a) generalize (I) to base-three
(b) generalize (II) to base-three
(c) generalize (I) to base-four
(d) generalize (II) to base-four