Fractals Visualization and J

4th edition, Part 1

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

My wife Marilyn has filled my life with gifts, including our sons.

This book is dedicated to our sons:
Benjamin Joseph
Zachary Xavier
Nathaniel Levi
Joshua Steven

Published by Lulu
ISBN 978-1-329-92666-0
eBook version

Table of Contents

Preface v
Getting Ready: Addon Files vi
Chapter 1 Introduction to J and Graphics 1
1.1 Some Arithmetic with J 1
1.2 Lists, Arrays and Trigonometric Functions 3
1.3 Experiment: Plotting Polygons 4
1.4 Constructing Arrays 5
1.5 Experiment: Creating a Raster Image 7
1.6 Object versus Raster Graphics 8
1.7 Defining Functions 8
1.8 On Language 12
1.9 Errors and Getting Help 12
1.10 More Nouns and Array Computations 15
1.11 Exercises 18
Chapter 2 Plots, Verbs and First Fractals 23
2.1 Function Composition and Plots 23
2.2 Experiment: Plotting Time Series, Functions and Curves 25
2.3 More Function Composition 26
2.4 Experiment: The Koch Snowflake 30
2.5 Transformations of the Plane and Homogeneous Coordinates 32
2.6 Experiment: Transformations and Animations 34
2.7 Gerunds and Multiplots 36
2.8 Experiment: Collages of Transformations 38
2.9 Simple Verbs 39
2.10 Exercises 41
Chapter 3 Time Series and Fractals 45
3.1 Statistics and Least Squares Fit 45
3.2 Experiment: Plot Driver 47
3.3 Random Walks 47
3.4 Experiment: Observing Trends 50
3.5 R/S Analysis, the Hurst Exponent, and Sunspots 52
3.6 Autocorrelation Functions 55
3.7 Experiment: Random Midpoint Displacement 56
3.8 Experiment: Forecasting via Best Analogs 58
3.9 Exercises 61
Chapter 4 Iterated Function Systems and Raster Fractals 65
4.1 Agenda and the $3x+1$ Function 65
4.2 Experiment: Probabilistic Iterated Function Systems 66
4.3 Remarks on Iterated Function Systems 69
4.4 Weighted Selection of Random Transformations 71
4.5 Experiment: The Chaos Game 72
4.6 Fractal Dimension 75
4.7 Fractal Dimension via Raster Box Counting 76
4.8 Exercises 78
Chapter 5 Color, Contours and Animations 83
5.1 The RGB Color Model 83
5.2 Adverbs and Conjunctions 85
5.3 Experiment: Color Contour Plots 86
5.4 Animations 89
5.5 Plasma Clouds 90

5.6 Experiment: Palettes and Inner Product Fractals 92
5.7 Inverse Iterated Function Systems 96
5.8 Exercises 98
Chapter 6 Complex Dynamics 101
6.1 Experiment: Julia Sets 101
6.2 Experiment: Julia sets for Elliptic Curves 103
6.3 The Mandelbrot Set 104
6.4 The $3x+1$ Function in the Complex Plane 106
6.5 Newton's Method in the Complex Plane 108
6.6 Exercises 112
Chapter 7 Cellular Automata 113
7.1 One Dimensional Automata 113
7.2 Fuzzy Logic and Fuzzy Automata 117
7.3 Experiment: The Game of Life 121
7.4 Majority Rule and Spot Formation 124
7.5 Cyclic Cellular Automata 126
7.6 Experiment: The Hodgepodge Rule 128
7.7 Hexagonal Lattice and the Packard-Wolfram Snowflake 131
7.8 A Snowflake Model Using Intermediate Values 133
7.9 Exercises 134
Bibliography and References 137
Index 139

Preface

The first edition of Fractals, Visualization and J appeared in 1995. The second edition followed in 2000 and the third in 2007. The time has come for a fourth edition. The evolution of J from 6.01 to 8.04 has been substantial. The addons heavily used by the previous two editions are now quite dated and need to be replaced. That task seemed daunting, but I also don't want to wait years for the fourth edition. Thus, I split the fourth edition into two parts. This Part 1 is highly reorganized from the third edition in ways that I believe will make it more attractive for a general reader. There are several new sections and extensive revisions to many other sections. Part 1 contains the most essential part of what has become a visualization and fractals special topics course that I teach about every 4 years. It also contains the core introduction to J in that context.

I remain enthusiastic about sharing with my students the power of visualization through J. I hope the reader also finds joy in the visualization and the expressiveness of J. I am indebted to the students who have used my book over the years. Their insights, questions, and mistakes have helped me improve the book with each edition. I likewise am deeply indebted to my wife Marilyn who has been patient and kind as I have worked on each edition. The J software team has done wonderful work, along with outstanding contributions from other members of the Jforum. I need to mention Eric Iverson, Roger Hui, Chris Burke and Bill Lam and apologize to the other important contributors for my ignorance. Some of the leaders have passed away during the 25 years of J, but the visionary Ken Iverson will always be remembered by those of us who knew him. Because of all these contributors to J, the world is a better place.

January, 2016

Getting Ready: Addon Files

Throughout this book it is assumed that you have available two addons: the *graphics/fvj4* and *media/imagekit* addons. At this time these addons are not a complete replacement of their predecessors in J6.02 (*graphics/fvj3* and *media/image3*), but they are sufficient for working with Part 1. It is intended that the addons be updated as progress on Part 2 is made. However, some features of the predecessor addons will probably not find their way into these addons. The expressions in this book have been tested under J64-8.03 (JQT) and/or J64-8.04 (JQT) using the 64 bit versions.

graphics/fvj4

The *fvj4* addon is analogous to the portions of the J6.02 *fvj3* addon that are used in Part 1 of the fourth edition of this book with some additions.

automata.ijs	gives many functions for creating and visualizing automata
chaotica.ijs	utilities for working with chaos and symmetry
complex_dynamics.ijs	utilities for working with complex dynamics
dwin.ijs	gives a simple window for displaying polygons
*dna_y54g9.ijs*dna	data for experiments with viewing correlations
life_ex2.ijs	gives several sample life configurations
povkit.ijs	gives formatting utilities for POV-Ray scenes
raster.ijs	gives utilities for dealing with arrays of pixels as images
smiles.ijs	gives transformations for an iterated function system
ts_data.ijs	gives time series data
vlife.ijs	runs a version of the Game of Life in color

In addition, there are labs: *FVJ4: dwin.ijt* and *FVJ4: /raster.ijt*

media/imagekit

The *imagekit* addon is a light version of the J6.02 *image3* addon. It resides in *~addons/media/imagekit/* and it contains several scripts. The material used in Part 1 of the text includes the following.

html_gallery.ijs	used to create html galleries
imagekit.ijs	script for displaying, reading and writing raster images
atkiln.jpg	sample image
hy_fly_di.png	sample image

In addition, there are labs: Imagekit: imagekit.*ijt* and *Imagekit Html: imagekit_html.ijt*

Chapter 1 Introduction to J and Graphics

In this chapter we gain a basic understanding of both object based and pixel based graphics. Object based graphics describe a scene in terms of polygons, circles and other geometric objects. Raster images are represented by an array of pixels (picture elements) and the image is described by giving the color of each pixel. We will also develop enough understanding of J to facilitate these preliminary graphics explorations.

1.1 Some Arithmetic with J

Many common J functions consist of a plain ASCII character.

```
   3+8                 plus
11

   3-8                 minus
_5

   3*8                 times
24

   3%8                 divide
0.375

   3^8                 power
6561

  t=: 101              give the value 101 the name "t"

   t^2                 square it
10201
```

All of those illustrations of arithmetic had two arguments for each function and hence were ***dyadic*** uses of the function. Those functions also have a different meaning when used with a single right argument; that usage is the ***monadic*** use of the function.

```
   %5                  reciprocal
0.2

   -3                  negate
_3

   _2-4
_6
```

Notice that the underline is used as a negative sign. This is part of the number, not a function. Another monadic function is the exponential function.

```
   ^1                  exponential base e
2.71828

   1+^1                one plus e
3.71828

   ^1+1                e²
7.38906

   ^2
7.38906
```

```
   1+3*4
13

   3*4+1
15

   (3*4)+1
13
```

J has no hierarchy of functions used to determine the order of evaluation. The order of evaluation is a simple right to left or left to right rule. If you are imagining bottom up evaluation, then the arithmetic is done right to left. Thus, to evaluate `3*4+1` we first compute `4+1` getting `5` and then multiply by `3` to get `15`. On the other hand, if one uses top down evaluation, you may read that expression, `3*4+1`, left to right and think of it as `3` times the sum of `4` and `1`. Of course, the order may be modified with parentheses. Since J has a large number of built in functions, the simplicity of the evaluation rule allows for very concise, but clear, expression of powerful ideas. While the convention may seem unusual for dyadic functions, it is consistent with the common mathematical convention for monadic functions. That is, in common math, if we said or wrote the sine of the log of *t*, then the log would be applied to the *t* before the sine was applied to the result.

Figure 1.1.1 shows a few J expressions and their common mathematics meaning. The expressions are executed below.

J expression	Common Math Expression
`2^10-1`	2^{10-1}
`(2^10)-1`	$2^{10}-1$
`_1+2^10`	$-1+2^{10}$
`-1+2^10`	$-(1+2^{10})$

Figure 1.1.1. J Functions have no Precedence Order

```
   2^10-1
512

   (2^10)-1
1023

   _1+2^10
1023

   -1+2^10
_1025
```

Many J functions are denoted by an ASCII symbol followed by a period or colon.

```
   +:6          double
12

   *:4          square
16

   %:2          square root
1.41421

   ^.2          natural logarithm
0.693147

   2^.8         logarithm base 2
3
```

Common Math Functions and Arithmetic	J Expression
$x+y$	`x+y`
$x-y$	`x-y`
$\frac{x}{y}$	`x%y`
x^y	`x^y`
$\frac{1}{y}$	`%y`
$-y$	`-y`
e^y	`^y`
$2y$	`+:y`
$\frac{y}{2}$	`-:y`
y^2	`*:y`
$\sqrt{y}$	`%:y`
$\sqrt[x]{y}$	`x%:y`
$\ln(y)$	`^.y`
$\log_x(y)$	`x^.y`

Figure 1.1.2. Some Mathematical Functions

1.2 Lists, Arrays and Trigonometric Functions

To prepare for the experiment in the next section, we want to be able to build trigonometric tables. That is, tables of sines and cosines for several simple angles. We first aim to create useful lists of nice angles.

```
   i.5                 a list of indices (integers)
0 1 2 3 4

   2+i.5               add 2 to a list
2 3 4 5 6

   *:i.5               square a list
0 1 4 9 16

   1e6                 one times ten to the 6th
1000000

   1p1                 1 times pi to the first power (result is floating point)
3.14159

   2p1                 2 times pi to the first power
6.28319

  1r2                  one-half; notice the answer is an exact rational number
1r2

   1r2 + 1r4           one-half plus one-quarter; the answer is exact
3r4

   0.1 * 1r2           floating 0.1 times rational 1r2 is floating
0.05

   1r4p1               one-fourth times pi
0.785398

   1r4p1 * i.5         multiples of one-fourth times pi
0 0.785398 1.5708 2.35619 3.14159
```

Notice that the "e", "p" and "r" that appear in the above numbers are part of the constant; they are not functions. Thus `(3)r(5)` makes no sense, while `3%5` is `0.6` and `3r5` is the rational number three-fifths.

Next we introduce the trigonometric functions. We load names for the trigonometric functions from a standard J script and then, alternatively, use the dyadic J circular function denoted `o.` in order to evoke trigonometric functions. This is a convenient form for creating tables.

```
   load 'trig'

   sin 1r4p1 * i.5     the last entry is roundoff error near zero
0 0.707107 1 0.707107 1.22465e_16


   sin                 sin(y) is 1 o. y
1&o.                   (we will discuss bond & in Section 1.7)

   cos                 cos(y) is 2 o. y
2&o.

   1 2 3 +/ 1 2 3      now we create tables; first a plus table
2 3 4
3 4 5
4 5 6
```

```
   1 2 3 */1 2 3
1 2 3
2 4 6
3 6 9
```
a times table

```
   1 2 3 ^/1 2 3
1 1  1
2 4  8
3 9 27
```
a power table

```
   2 1 o./ 1r4p1 * i.5
1 0.707107 6.12303e_17 _0.707107          _1
0 0.707107           1  0.707107 1.22461e_16
```
a table of cosine and sine values (with some round-off error)

```
   |: 2 1 o./ 1r4p1 * i.5
        1             0
 0.707107      0.707107
6.12303e_17           1
  _0.707107    0.707107
         _1 1.22461e_16
```
transpose gives a list of cosine-sine pairs in each row instead of column

Since $\sin^2(t) + \cos^2(t) = 1$ for all angles *t*, those cosine-sine pairs are all on the unit circle. In the next section we will see where those vertices are placed on the circle.

1.3 Experiment: Plotting Polygons

In this section we load the script *~addons/graphics/fvj4/dwin.ijs* (draw window) associated with this text. If the "load" expression given below gives an error, then the addon should be installed using package manager. See the Getting Ready section in the beginning of the book. The script *dwin.ijs* provides some basic utilities for drawing basic objects, namely polygons, in a window. For example, we create a drawing window with coordinates between `_1` and `1` in both the coordinates, and then plot the polygon from the last section as follows.

```
   load '~addons/graphics/fvj4/dwin.ijs'
```
load the drawing window script

```
   _1 _1 1 1 dwin 'hello'
```
open a drawing window

```
   ]p1=: |:2 1 o./ 1r4p1*i.5
          1           0
   0.707107    0.707107
6.12303e_17           1
  _0.707107    0.707107
         _1 1.22461e_16
```
the polygon from the last section

```
   0 255 0 dpoly p1
```
draw the polygon

The graphics window should show a green barn shaped polygon. Notice that its vertices are on the unit circle. The convention we use for describing colors requires giving red-green-blue triples with the intensity of each component being an integer from `0` and `255`. Thus, the color `0 255 0` has no red, full green, and no blue. Eventually we will discuss this color model in detail; for now it suffices to be able to make a few colors for distinguishing between objects. In order to obtain a regular pentagon, we want multiples of the angle `2r5p1`.

```
   dclear ''
```
clear the drawing window

```
   p2=: |:2 1 o./ 2r5p1*i.5
```
define a regular pentagon

```
   0 0 255 dpoly p2
```
display it in blue

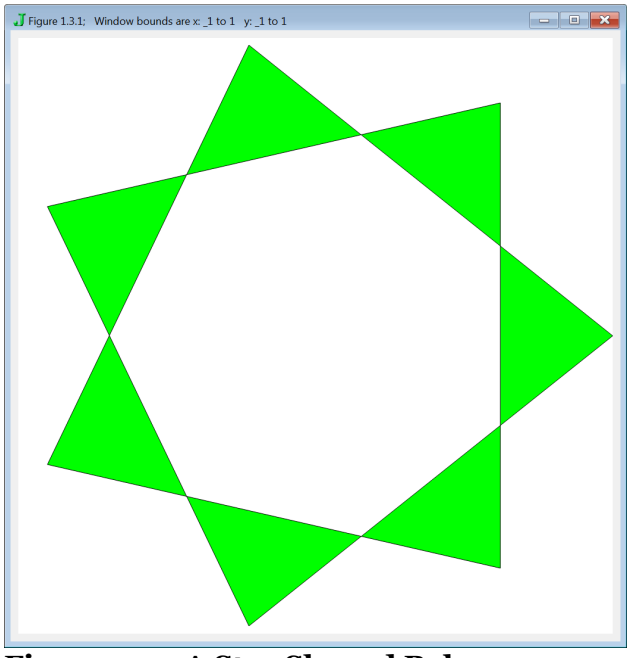


Figure 1.3.1 A Star Shaped Polygon

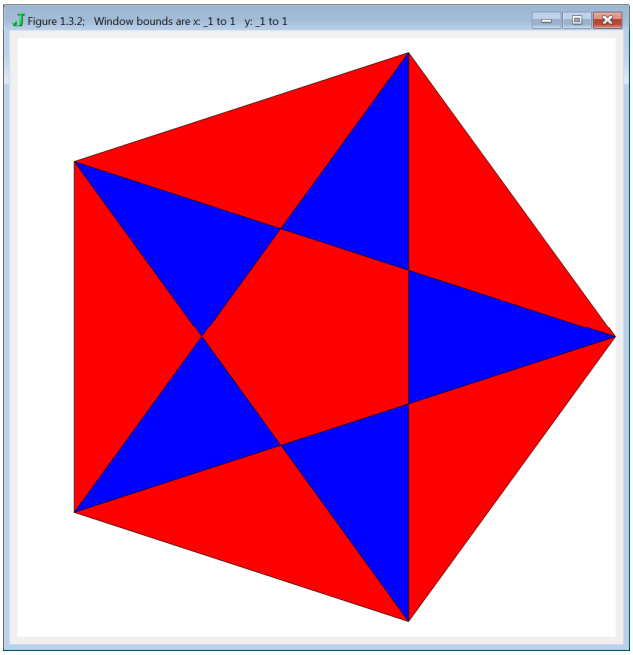


Figure 1.3.2 Two Polygons

Here are experiments you should run in order to get experience with drawing polygons and using colors.

```
dclear ''
p3=: |:2 1 o./ 2r17p1*i.17
0 0 0 dpoly p3
255 0 0 dpoly 0.5*p3
```

```
dclear ''
p4=: |:2 1 o./ 2r9p1*i.9
255 255 0 dpoly p4
p5=: |:2 1 o./ 4r9p1*i.9
0 255 255 dpoly p5
p6=: |:2 1 o./ 8r9p1*i.9
255 0 255 dpoly p6
```

After running those experiments, can you duplicate the polygon(s) in Figure 1.3.1? Figure 1.3.2?

Note that the utilities in *dwin.ijs* are intended to be useful for learning purposes. These simple utilities here have undesirable features. For example, only one graphics window may be accessed at any one time. There are also side effects beyond creating a window. For example, a function `SC` is created to facilitate scaling images that are to be plotted. That can result in a name conflict if we happen to use `SC` for something else. Many more features of graphics programming may be accessed directly with *gl2.ijs* defined functions and many utility functions are defined in the plot script (*plot.ijs*) that will be discussed in later sections. We are content to use the utilities from *dwin.ijs* for now since they allow us to focus on one of the most basic feature of object based graphics: drawing polygons.

1.4 Constructing Arrays

A vector is a list of elements or atoms. A matrix is a list of vectors. Alternatively we can think of it as a two dimensional array of atoms. In general, arrays are lists of items and may have a number of axes. Note that the items of a matrix are its vectors (rows) while the atoms of the matrix are the individual entries. A matrix has two axes, a vector has one axis and an atom has zero axes. We have already seen how to create arrays that have the structure of an addition table, a multiplication table, and the like. We deal here with arrays that contain general data and also consider building arrays by piecing together smaller arrays.

```
   ]p=: 10^i.4
1 10 100 1000
```
recall the index and power functions;
] denotes the identity function and causes display

```
   ]q=: - 2 3 4 5
_2 _3 _4 _5
```
recall the minus function

```
   p+q
_1 7 96 995
```
we can add lists

```
   p*q
_2 _30 _400 _5000
```
or do other arithmetic

```
   p,q
1 10 100 1000 _2 _3 _4 _5
```
we can adjoin two lists

```
   #p,q
8
```
number of items in the result (also called the ***tally***)

```
   $p,q
8
```
the ***shape*** of the result indicates the array has one axis of length 8

We now consider constructing matrices. That is, two dimensional arrays of numbers arranged in rows and columns. First, we can build matrices from two vectors.

```
   p,:q
 1 10 100 1000
_2 _3  _4   _5
```
we can ***laminate*** two lists to form a matrix

```
   p,.q
   1 _2
  10 _3
 100 _4
1000 _5
```
we can ***stitch*** together two vectors

```
   #p,.q
4
```
number of items (rows) in the result

```
   $p,.q
4 2
```
the shape of the result indicates the array has two axes, the first axis has length 4 and the second has length 2.

We can also reshape a list of data into matrix form. Extra data is ignored and data is reused cyclically if necessary.

```
   ]a=: 3 4 $ 9 8 7 6 5 4 3
9 8 7 6
5 4 3 9
8 7 6 5
```
3 by 4 ***reshape*** of the data;
note the reuse of data

```
   ]b=: 10 20,30 40,:50 60
10 20
30 40
50 60
```
we list by rows;
note the trailing laminate

```
   a,b
 9  8 7 6
 5  4 3 9
 8  7 6 5
10 20 0 0
30 40 0 0
50 60 0 0
```
arrays may be adjoined;
note the padding with zeros
on the lower right

```
   a,.b                    matrices may also be stitched side by side when
9 8 7 6 10 20              the number of rows agree
5 4 3 9 30 40
8 7 6 5 50 60
```

We will use arrays in the next section to represent an image.

1.5 Experiment: Creating a Raster Image

A raster image may be thought of as a matrix that specifies the color of corresponding pixels in an image. Instead of giving the red-green-blue intensities for each pixel in the matrix, here we will define a list of possible colors (the palette) and then the matrix entries are indices into the palette.

In the previous section we saw how to adjoin arrays either vertically or horizontally. We can use those constructions to build arrays with a fractal structure. We begin with a 1 by 1 array with the single entry 1. Then we build a sequence of matrices, each built from three copies of the previous matrix, arranged in an L shape with padding by zeros in the upper right.

```
   ]s=: 1 1$1
1

   ]s=: s,s,.s
1 0
1 1

   ]s=: s,s,.s             display the next matrix
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1

   $s=: s,s,.s             display the shape of the new matrix
8 8

   $s=: s,s,.s             and again and again...
16 16
```

Continue this process (use shift-ctrl-up-arrow or cut-and-paste to minimize typing) until you have obtained a 512 by 512 array. Notice that once the arrays get big, it is convenient to not type the identity function `]` and thereby suppress displaying the array. We take the shape to see that the shape of the array is doubling along each axis at each stage as a check that we have not made a typographical error.

We describe two approaches to viewing the matrix `s` as an image. We may use a system utility *viewmat.ijs* or a viewer associated with the *imagekit* addon.

```
   load 'viewmat'          loads the system matrix viewer

   viewmat s               view the matrix (default white on black)

   ]pal=: 255 255 0,:0 0 255   define a palette
255 255   0                    yellow for the zeros
  0   0 255                    blue for the ones

   pal viewmat s           view the matrix blue on yellow
```

Second, we run the script *~addons/media/imagekit/imagekit.ijs*. It defines functions that will convert a palette and an array of indices into an image that can be viewed or saved as a file.

```
   $s                      double check image array size
512 512
```

```
load '~addons/media/imagekit/imagekit.ijs'      load imagekit utilities
view_data s                                     view the array with default palette
(pal;s) write_image jpath '~temp/temp.png'      create an image file
view_image jpath '~temp/temp.png'
```

How does the 1024 by 1024 image created from one more iteration differ in appearance from the 512 by 512 image created above?

1.6 Object versus Raster Graphics

In Sections 1.3 and 1.5 we have seen some preliminary graphics. In the first case, we defined objects that were polygons. Object based graphics have the advantage that they are independent of the screen resolution. That is, if one changes the window size, then the object may be redrawn exactly to that window. This is quite convenient when writing software for an unknown graphics system. The system draws the objects at resolutions appropriate to the monitor (or other output device). This means that software can effectively draw objects on low resolution screens and high resolution screens without the programmer knowing the capabilities of the device. Hence we think of object based graphics as device independent. Moreover, modern graphics systems can rapidly render hundreds of thousands of polygons quickly and thus animations can be created.

Raster graphics require the description of the color of each pixel in an array representing points in the image. In many contexts this requires considerable amounts of memory. This may be done directly by literally listing the red-green-blue components of each pixel or it may be done indirectly by giving indices into a palette of available colors as we did in the previous section. Since most monitors/lcds are raster devices in the sense that the color of each pixel must be specified, raster images may closely fit the particular hardware and hence take advantage of specific hardware capabilities. Computer images like photographs are convenient to store in raster formats. Because of the memory requirements of raster images, a number of methods for compressing them have been developed and this helps explain the large number of raster image formats which include **.bmp*, **.png*, **.gif*, **.tga*, **.jpg* since each met a need when developed. Animation of raster images has even more intensive memory requirements, but such animations are common on modern computers.

1.7 Defining Functions

There are two different types of function definition available in J. Many functions (also called verbs) may be defined tacitly. That means the function is defined without explicit reference to its arguments. One way to accomplish tacit definitions is to ***bond*** one argument of a dyadic function to the function using the ampersand `&` . For example, in order to define the function that is the cubing function, we can define the following.

```
   cube=: ^&3
   cube 4                       cube is a monad; its argument goes on the right
64
   cube 0.5 1 2
0.125 1 8
   cube                         display the function definition (linear view)
^&3
   Sin=: 1&o.                   next define Sin to be the sine function
   Sin 1r4p1
0.707107
```

We may also define functions explicitly by referring to the right variable inside quoted text that gives the formula for the definition. The right argument is denoted by `y` in that context. The number three to

the left of the colon designates that the text to the right of the colon defines a function. Other numbers can be used to define objects that are other parts of speech.

```
   CUBE=: 3 : 'y ^ 3'

   CUBE 0.5 1 2
0.125 1 8

   CUBE
3 : 'y ^ 3'
```

Below we convert from a one line explicit definition into tacit definition using thirteen as the left argument of colon. Note that the tacit result is not identical to the result using bond given above.

```
   13 : 'y ^ 3'
3 ^~ ]
```

In fact, it is an illustration of a fork and passive, which reverses arguments. In the next chapter we will study many types of tacit function composition at which point the meaning of that fork should be clear. As we develop experience with tacit definitions they become natural. They can also be efficient and helpful for implementing powerful programming strategies. However, explicit definitions using `y` need not be avoided.

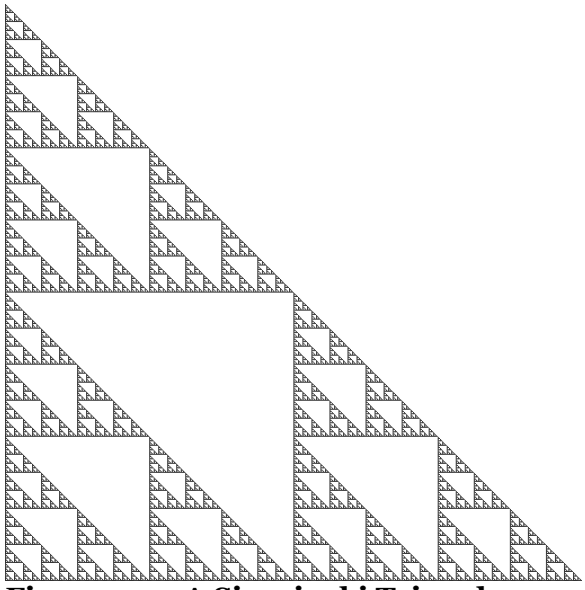


Figure 1.7.1 A Sierpinski Triangle

In many contexts it is convenient to apply a function repeatedly. Below we see this happen with the cubing function by explicitly typing cube repeatedly and also by using function power `^:` for iteration.

```
   cube cube 0.5 1 2
0.00195313 1 512

   cube^:(2) 0.5 1 2
0.00195313 1 512
```

Notice that the number of iterations, `2`, was in parentheses in order to separate the number of iterations from the input data: `0.5 1 2`.

```
   cube^:(i.3) 0.5 1 2
     0.5 1   2
   0.125 1   8
0.00195313 1 512
```

we may apply several different numbers of iterations at once

We return now to the construction of the Sierpinski triangle. In Section 1.4 you might have grown weary of repeatedly typing very similar lines such as the following.

```
   s=: s,s,.s
```

This may be mitigated by using control-shift-up-arrow to capture the previous line for editing, but even that remains tedious. However, once we define a function that performs one step of the process, we need only iterate that function to obtain the desired matrix. Notice that our function is essentially the right hand side of `s=:s,s,.s` with the "input" name changed to `y` in the function definition. Often simple functions in J may be written by doing an experiment, then using the experiment as a template for a general function definition.

```
   sier=: 3 : 'y,y,.y'
```

```
   ]s0=: 1 1$1
1

   sier^:2 s0
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1

   $s=: sier^:9 s0
512 512
```

The result of this image is shown in Figure 1.7.1.

We have seen that many functions have both a monadic and dyadic meaning. This can be true for user defined functions as well. We can define a multiple line function in J using `3 : 0` to begin the definition. The next line begins the definition of the monad; the definition of the monad is followed by a line containing only a colon, which is followed by the definition of the dyad. The final part of the definition is a closing right parenthesis on an isolated line. The right argument is denoted by `y` for both the monad and dyad and `x` denotes the left argument of the dyad. As an illustration we define a function `rpoly` that gives the vertices of a regular polygon with `y` sides whose vertices are a distance `x` from the origin. After showing some examples of how the function may be evoked, we have several additional comments to make about the definition of the function. Round-off error causes many entries that should be zero to be very small numbers. These produce no difficulties for using the polygons.

Usually when entering a multiple line function, we enter the definition in a script. J scripts are text files of J expressions, usually with a name of type **.ijs* and a new temporary script may be created with the file-new menu. Create a new **.ijs* window and enter the definition below (without the comments including or following the `NB.`). Then run the script using the "run"-"load script" menus. Go back to the execution window to use `rpoly` in interactive mode.

```
   rpoly=: 3 : 0
NB. begin monad; y gives number of vertices
1 rpoly y             NB. monad is unit radius polygon
:
NB. begin dyad; x gives the radius
t=.(2p1%y) * i. y     NB. Angles defined locally
p=:|:2 1 o./ t        NB. trig table defined globally
x * p                 NB. the dyadic result
)
```

Now return to the execution window

```
   2 rpoly 3                a triangle
 2        0
_1  1.73205
_1 _1.73205

   1 rpoly 5                a pentagon
        1         0
 0.309017  0.951057
_0.809017  0.587785
_0.809017 _0.587785
 0.309017 _0.951057

   2 rpoly 5                a bigger pentagon
       2        0
0.618034  1.90211
_1.61803  1.17557
```

```
_1.61803 _1.17557
0.618034 _1.90211
```

```
      rpoly 5                    monadic use
        1         0
 0.309017  0.951057
_0.809017  0.587785
_0.809017 _0.587785
 0.309017 _0.951057
```

```
   t                             no global value for t
|value error: t
```

```
   p                             global value for p
        1         0
 0.309017  0.951057
_0.809017  0.587785
_0.809017 _0.587785
 0.309017 _0.951057
```

Notice several things in the definition of `rpoly`. First, comments are preceded by the expression `NB.` (for "nota bene", Latin for "note well"). Second, we could define the monad without reference to the dyad, but since it just gives a default value for the left argument, it is convenient to use a recursive (self-referential) call on the dyad to give the definition of the monad. Third, in the definition of the dyad, we defined two names, `t` and `p`, for the angles and the polygon, respectively. The definition for `t` used `=.` which denotes ***local assignment***. Thus, the value for `t` is not available after the function evaluation is complete. Referring to `t` outside the function gives a value error. Local assignments like this are useful for avoiding name conflicts. The definition of `p` used ***global assignment*** `=:` and hence the value of `p` is available after the function evaluation is complete. However, any previously defined global value for `p` would have been lost. In interactive sessions there is no difference between local and global assignments. However, the distinction is important inside function definitions and usually it is best to use local names to avoid conflicts. Still, global names can be quite useful for debugging and for preserving the result of computations that might be needed later.

Lastly, we remark that if we want to define a function that only has a dyadic meaning, we can have zero lines before the separating colon or we can use `4 : 0` instead of `3 : 0` when beginning multi-line input mode. The function below gives the coordinates of a star with a specified radius and odd number of vertices. ***Half*** `-:` and ***floor*** `<.` are used in its definition and, while only one line, it is only a dyad.

```
   star=: 4 : 'x * |: 2 1 o./ ((2p1%y) * <.-:y) * i.y'
```

```
   1 star 5
        1         0
_0.809017  0.587785
 0.309017 _0.951057
 0.309017  0.951057
_0.809017 _0.587785
```

```
   star 5                        monadic use is not within the defined domain
|domain error: star
|       star 5
```

We can display the star as in Section 1.3.

```
   load '~addons/graphics/fvj4/dwin.ijs'
```

```
   _1 _1 1 1 dwin 'star'
```

```
   255 255 0 dpoly 1 star 5
```

1.8 On Language

In our discussion we have seen many different kinds of J objects. It is traditional to use conventional grammatical terms to describe J objects. First, data are ***nouns***. Thus in the definition below, the vector `3 4r5 1p1` is a noun while `A`, defined below, may be called a ***pronoun***. The addition of `A` and `2` is an action, and hence we call addition a verb. In general, we will think of both monadic and dyadic functions as ***verbs***. Named verbs are called ***proverbs*** (pronounced with a long o).

```
   A=: 2 4r5 1p1

   A+2
4 2.8 5.14159
```

As illustrated below, we saw that the slash `/` modified the behavior of a function (verb), which is specified on its left and the result is another function, called the derived function (verb). The dyadic derived function is a table builder. Thus `*/` is a product table builder. For this reason the slash `/` is called an ***adverb***.

```
   1 2 3 */ 1 2 3
1 2 3
2 4 6
3 6 9
```

When we iterate J functions using function power `^:` we give the function on the left and the power (the number of iterations) on the right. The result is a new function that we can apply to data. Since the function power `^:` requires two arguments to produce a derived verb, it is called a ***conjunction***. We illustrate function power below assuming that `s0` and `sier` are defined as in the previous section.

```
   sier^:2 s0
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1
```

We close this section with some comments on the periods, colons and spaces that need careful reading in J. When a symbol is followed immediately by a period or a colon, the impact is that a new word has been formed. The period or colon form an ***inflection*** of the symbol. However, if there is a space between the symbol and the period or colon, then the colon or period stand alone; for example, a colon with a space to its left (and a `3` or `4` to the left of that) is used to designate function definition.

```
   %:16                    square root
4

   %: 16                   square root
4

   % : 16                  not a proper function definition
|domain error
|       % :16
```

1.9 Errors and Getting Help

We pause to comment on several error messages and remind readers that J has a help system. We can add lists of the same length and a scalar is extended to appropriate length when adding a scalar to a list. However, attempting to add two lists of different length results in a ***length error***.

```
   1 2 3 + 1 2 3
2 4 6
```

Figure 1.9.1 The Official Vocabulary of J

```
   1 + 1 2 3
2 3 4

   2 3 + 1 2 3
|length error
|[-4]
```

Since addition is not defined for strings, it is not a surprise that trying to add a string to a number results in a ***domain error***.

```
   'a'+1
|domain error
|   'a'    +1
|[-0]
```

However, because J recognizes infinity and complex numbers, several arithmetic functions might surprisingly give a result.

```
   %0
_                    infinity
   ^.0
__                   negative infinity
   %: _1
0j1                  the complex number i
```

A J expression that makes no sense or is incomplete often results in a ***syntax error.***

```
   (3 + 4
|syntax error
|   (3+4

   3 %
|syntax error
|       3%
```

Unmatched text quotes result in an ***open quote*** error.

```
   'jjj
|open quote
|   'jjj
|   ^
```

When a number is expressed improperly, an ***ill-formed number*** error may occur.

```
   1p1                          1π1  represents pi.
3.14159

   1p
|ill-formed number
|[-2]
```

It appears at a casual glance that both `a` and `b` below represent one-tenth.

```
   a=: 0.1

   a
0.1

   b=: .1

   b
 .1
```

However, if we try to use them we see they are different.

```
   a+1
1.1

   b+1
|syntax error
|       b+1
|[-2]

   type 'a'
+----+
|noun|
+----+
   type 'b'
+------+
|adverb|
+------+
```

We see that if we want a decimal number, it is important to precede it with a 0 on the left of the decimal point. We will see in Chapter Two that a dot (not surrounded by digits) is a conjunction generalizing the dot product. When it is given one argument, the result is an adverb. The adverb `b` is not meaningful but that fact would not be discovered until it was used.

```
   *: b
|domain error: b
|       *:b
```

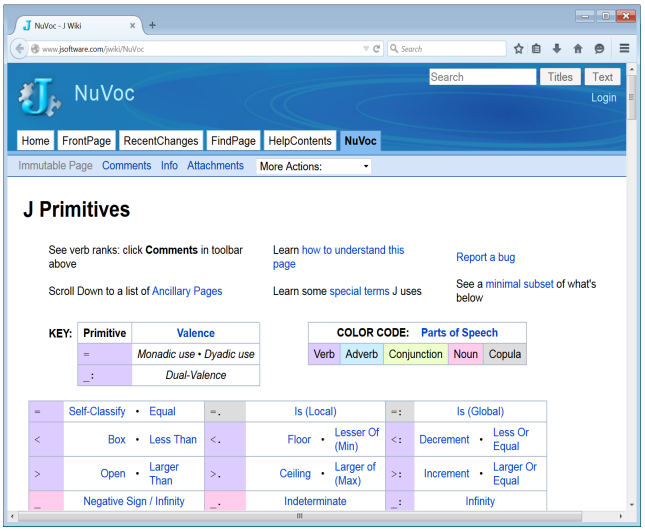


Figure 1.9.2 The NuVoc: Vocabulary of J Designed by Users

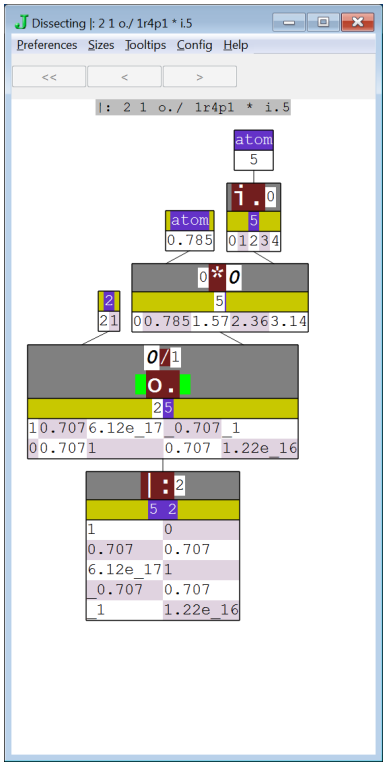


Figure 1.9.3 Dissecting a J Expression

Next we remark that J has a robust help system. However, the language is so rich, that "help" may be overwhelming to beginners. Nonetheless, it is advisable that readers be aware of the help facilities. They may be accessed through the "Help" menu or by using the "F1" or "Shift-F1" keys. "Voc" is the lead page to the official vocabulary of J while "NuVoc" was designed by users with special consideration of new users. A portion the the Voc page appears in Figure 1.9.1. A portion of the NuVoc appears in Figure 1.9.2.

Henry Rich's dissect addon contains facilities for parsing a J expression and displaying intermediate results. Additional information is given by hovering over parts of the display. One can use the package manager (under the "tools" menu) to download the *debug/dissect* addon and experiment as follows.

```
   load '~addons/debug/dissect/dissect.ijs'

   dissect '|: 2 1 o./ 1r4p1 * i.5'
```

The result is shown in Figure 1.9.3.

1.10 More Nouns and Array Computations

We illustrate in this section the fact that powerful facilities are available for the construction and manipulation of very complicated nouns. We begin by considering a three dimensional array of indices. The ***shape*** of the array is a list of the lengths of its axes. Cells are the parts of an array that form lower dimensional parts of the array organized along trailing axes. The ***items*** are the cells of the array that are one dimension lower than the array itself. Thus, the items of an array are "listed" by the first axis. The ***tally*** or ***number*** is the number of items. The array `a` defined below is 3-dimensional. It has 3 axes having lengths 2, 3, and 6. The items are the two matrices. The shape of the array is `2 3 6`. The number of items is 2. The one-dimensional cells are the rows of length 6.

```
   ]a=:i.2 3 6
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17

18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35

   $a
2 3 6

   #a
2
```

The array `b` defined below is 2-dimensional. It has 2 axes having lengths 3 and 6. The items are the three rows. The number of items is 3.

```
   ]b=:i.3 6
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17

   $b
3 6

   #b
3
```

The array `c` defined below is 1-dimensional. It has 1 axis of length 5. The items are the 5 ***atoms***. The number of items is 5.

```
   ]c=:i.5
0 1 2 3 4

   $c
5

   #c
5
```

The array `d` defined below is a 0-dimensional array; that is, it is a scalar. It has 0 axes so it shape is an empty list. The number of items is 1.

```
   ]d=:3
3

   $d


   #d
1
```

We next consider a way to process lists.

```
   +/2 3 4 5                 sum a list
14

   2+3+4+5                   plus inserted between items in a list
14

   */2 3 4 5                 product of a list
120
```

This use of `+/` and `*/` is monadic and is called ***insert*** since the result is the result of inserting the function in between the items of the list. You should be careful to distinguish this monadic insert usage of `+/` from the dyadic table builder that we saw earlier.

```
      b
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17

   +/b                       sum each column of the matrix;
18 21 24 27 30 33            plus inserted between the items (rows)
```

The summation function `+/` applied to a matrix results in inserting the plus sign between the items. That is, one can imagine the first row added to the second row, added to the third row, etc. and this gives the indicated result. Sometimes we want to sum each row. This can be accomplished with the ***rank*** conjunction which is designated with a double quote character. Here `"1` designates that the summation function (i.e., plus insert) should be applied to the 1-dimensional cells (rows) of the argument. That is, summation should be applied to each row.

```
   +/"1 b
15 51 87

   a                         our three dimensional array of indices;
 0  1  2  3  4  5            it forms two planes (matrices)
 6  7  8  9 10 11
12 13 14 15 16 17

18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35

   +/a                       plus insert adds the two matrices together
18 20 22 24 26 28
30 32 34 36 38 40
42 44 46 48 50 52
```

```
   ,/a                adjoin insert gives a 6 by 6 matrix
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35
```

J has complex numbers built in.

```
   _1^0.5             complex number that is the square root of _1
0j1
   0j1^2
_1
   1j1 * 1j2          In common notation: (1+i)(1+2i) = -1+3i
_1j3
```

Character arrays can be restructured in the same ways as numeric arrays.

```
   3 4$'abcde'        an array of characters
abcd
eabc
deab
```

Any array can be ***boxed*** to declare it a scalar.

```
   <i.2 2             an array boxed to produce a scalar
+---+
|0 1|
|2 3|
+---+
   <"2 i.2 3 4        box rank 2 gives a boxed list of the planes (matrices)
+--------+-----------+
|0 1  2  3|12 13 14 15|
|4 5  6  7|16 17 18 19|
|8 9 10 11|20 21 22 23|
+--------+-----------+


   <"1 i.2 3 4        box rank 1 gives a boxed matrix of the rows
+-----------+-----------+-----------+
|0 1 2 3    |4 5 6 7    |8 9 10 11  |
+-----------+-----------+-----------+
|12 13 14 15|16 17 18 19|20 21 22 23|
+-----------+-----------+-----------+
   ]v=: 'hi mom' ; 2 3 ; 2 2$5    mixed type of data ***linked*** together
+------+---+---+
|hi mom|2 3|5 5|
|      |   |5 5|
+------+---+---+
```

A boxed list may be reshaped.

```
   2 2$v
+------+------+
|hi mom|2 3   |
+------+------+
|5 5   |hi mom|
|5 5   |      |
+------+------+
```

1.11 Exercises

1. Write down a common math expression (without simplifying) for each of the following J expressions. Predict the result of each and test your result with J.

(a) `3-4+5` (b) `3*4-5` (c) `3^4-5`
(d) `1^^1+1` (e) `1-_2-3` (f) `%5%5`
(g) `9+%-9` (h) `-%-%5` (i) `1%^1-4-3`
(j) `%:%4` (k) `*:1+1` (l) `%:7+%-1*2-3`

2. Identify each use of every function in the previous exercise as monadic or dyadic.

3. Write J expressions to compute the following:

(a) $3^{\frac{4}{7}}$ (b) $\ln(3)$ (c) $\sin\left(\frac{\pi}{4}\right)$ (d) $\frac{1}{\sqrt{1+e}}$ (e) $\frac{\sqrt{8}-4}{3\sqrt{5}+1}$

4. Write common mathematical notation for the following J arithmetic expressions; do not evaluate or simplify the expressions.

(a) `1 o. 0` (b) `1 o. 1r3p1` (c) `2^3+4`
(d) `1+2*5` (e) `^2*5` (f) `%^.%:3`
(g) `-2+2^5` (h) `_2+2^5` (i) `2+-2^5`
(j) `2+_2^5` (k) `-2+2^5` (l) `_2+2^5`

5. Give J expressions involving `i.5` that generate the following lists.

(a) `0 2 4 6 8`
(b) `0 1 4 9 16`
(c) `0 1 1.41421 1.73205 2`
(d) `1 0.5 0.333333 0.25 0.2`
(e) `4 8 16 32 64`

6. Give J expressions that duplicate the following tables.

(a)
```
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

(b)
```
1 0.5 0.333333 0.25 0.2
2   1 0.666667  0.5 0.4
3 1.5        1 0.75 0.6
```

(c)
```
0 0 0 0
0 1 0 1
0 0 0 0
0 1 0 1
```

7. (a) Plot a regular 11 sided polygon in blue.
(b) Plot an 11-sided star in green.
(c) Plot the polygon in (b) on top of the polygon from (a).

8. Define a polygon that looks like a maple leaf.

9. Run an experiment to determine the geometric shape of the following polygon.

```
p=: #:i.4
```

10. Predict and verify what colors correspond to each of the following red-green-blue triples.

(a) `0 0 0`
(b) `0 0 255`
(c) `0 255 0`
(d) `0 255 255`
(e) `255 0 0`

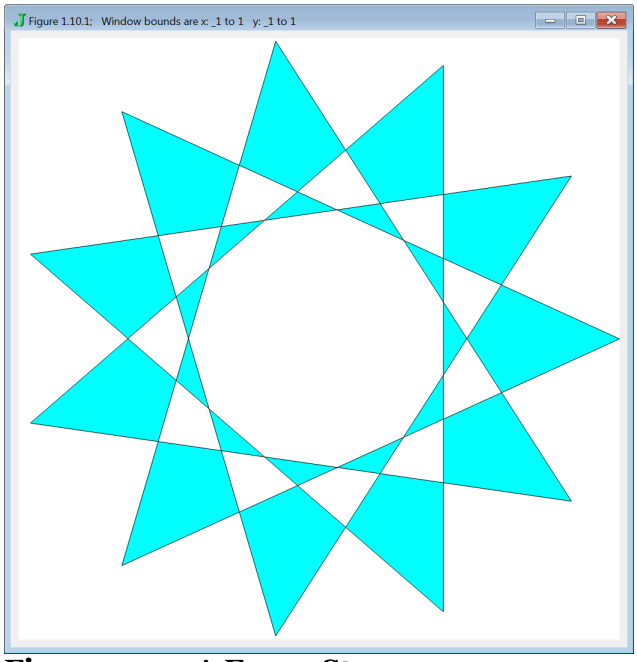


Figure 1.11.1 A Fancy Star

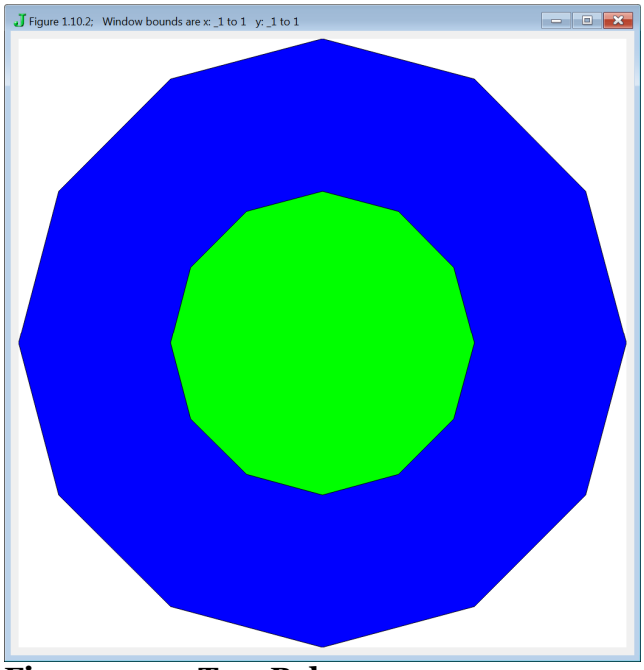


Figure 1.11.2 Two Polygons

(f) `255   0 255`
(g) `255 255   0`
(h) `255 255 255`
(i) `128 128 128`
(j) `255 128   0`

11. (a) Give J expressions to create a polygon that appears like the image in Figure 1.11.1
(b) Give J expressions to create polygons that appear like the image in Figure 1.11.2
(c) Give J expressions to create polygons that appear like the image in Figure 1.11.3
(d) Give J expressions to create polygons that appear like the image in Figure 1.11.4

12. The J function roll `?` selects a random index below its argument. Experiment with the following J expressions. Run each expression a few times.
(a) `?10`
(b) `?5$10`
(c) `?4 5$10`

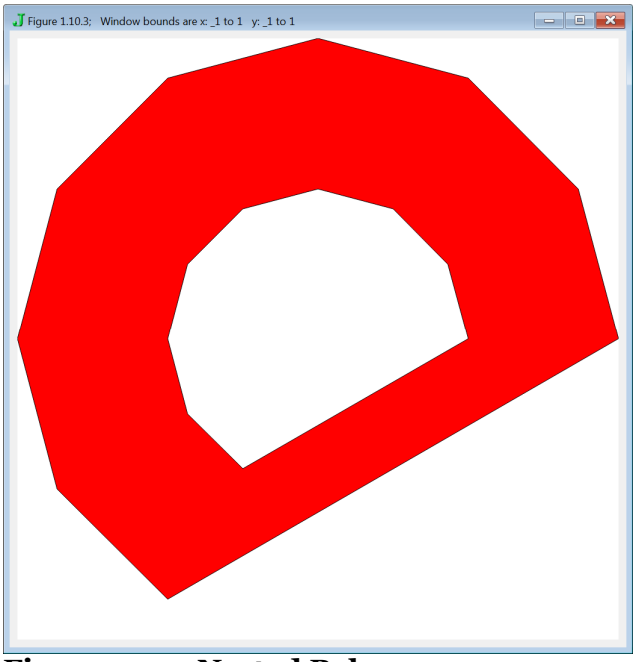


Figure 1.11.3 Nested Polygons

13. The J function random `?` selects a random index below its argument. We can create a random 4-sided polygon, a random color, and plot them as follows.

```
load '~addons/graphics/fvj4/dwin.ijs'
0 0 99 99 dwin 'random poly'
(?3$256) dpoly ?4 2$100
```

Then five random quadrilaterals may be drawn with the following.

```
(?5 3$256) dpoly ?5 4 2$100
```

Display a hundred random pentagons. Then display a hundred random triangles. What J expressions did you use? How do the displays differ?

14. Suppose `p=: 1 2 3` and `q=: 40 50 60` are defined. Give J expressions in terms of `p` and `q` that create the following arrays.

(a)
```
40 1
50 2
60 3
```

(b)
```
 1  2  3
40 50 60
```

(c)
```
1 40 50 60
2 40 50 60
3 40 50 60
```

15. Give a J expression that creates the matrix below using
(a) reshape and (b) adjoin and laminate (c) stich.

```
1 7 4 5
2 0 6 6
9 3 5 8
```

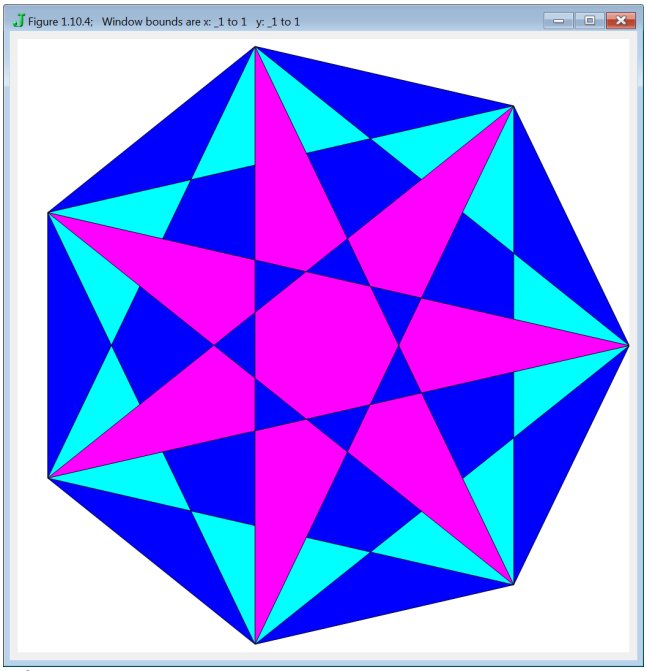


Figure 1.11.4 Polygons

16. Try to figure out the shape and number of items for each of the following arrays without using J. Then use J experiments to verify your answer.

(a) `a=: 3 4$5` (b) `b=: 3 5$2`
(c) `a,b` (d) `a,.b`
(e) `|:a` (f) `|:a,10`
(g) `a,a,.a` (h) `2*b`

17. The Sierpinski carpet is shown in Figure 1.10.5. Create a function in the style of `sier` from Section 1.7 that can be used to build the Sierpinski carpet.

18. A fractal closely related to the Sierpinski triangle is shown in Figure 1.10.6. It was created by modifying `sier` from Section 1.7 by applying the transpose to one of the components. Find the modified function and duplicate the figure.

19. Investigate the J on-line help.
(a) Skim the vocabulary.
(b) Read the preface to the dictionary.
(c) Click on `o.` in the vocabulary and note the circular functions defined there.

20. Write a paragraph comparing object based and raster graphics.

21. Tacitly define the following functions.

(a) $f_1(x) = 10x$ (b) $f_2(x) = 10 + x$ (c) $f_3(x) = 10 - x$
(d) $f_4(x) = x - 10$ (e) $f_5(x) = 10/x$ (f) $f_6(x) = 10^x$

22. Define the following monads explicitly. Recall the argument is denoted by `y` in explicit definitions.

(a) $f_1(x) = 10x$ (b) $f_2(x) = 10 + x$ (c) $f_3(x) = 10 - x$ (d) $f_4(x) = x - 10$
(e) $f_5(x) = 10/x$ (f) $f_6(x) = 10^x$ (g) $f_7(x) = 10x + 10/x$ (h) $f_8(x) = \sin(\sqrt{x}) + \cos(10x)$

23. Define the following dyads explicitly.

(a) $f_1(x, y) = x + 2y$ (b) $f_2(x, y) = x + y + e^{xy}$ (c) $f_3(x, y) = x^2 - y^2$ (d) $f_4(x, y) = \dfrac{1 + e^{xy}}{\sqrt{x + y + \pi}}$

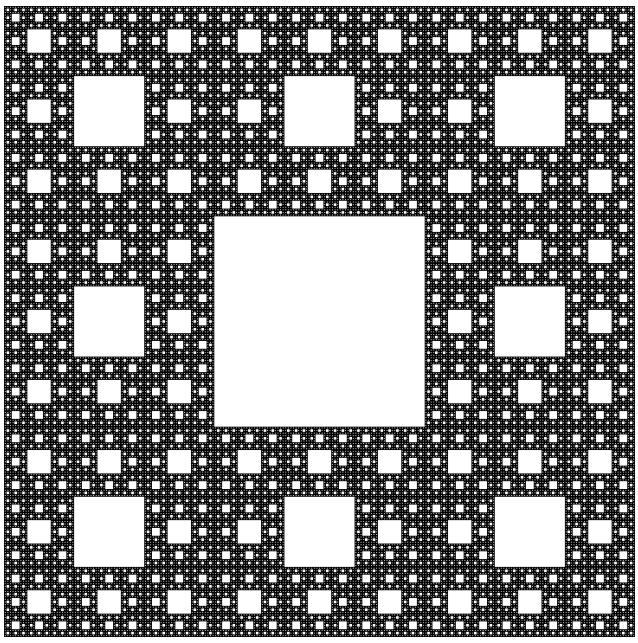


Figure 1.11.5 The Sierpinski Carpet

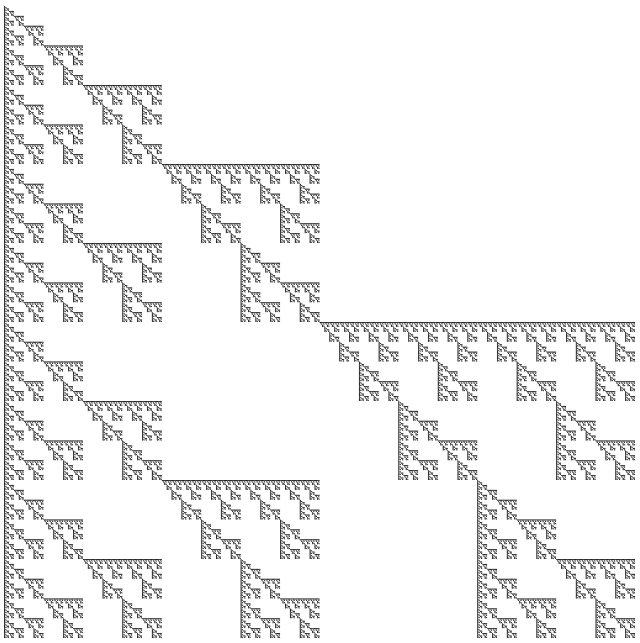


Figure 1.11.6 A Modified Sierpinski Fractal

24. Word formation is shown by `;:` applied to a text string that gives a J expression. Two examples are given below. Identify each word of the expression as a noun, verb (specify monad or dyad), adverb, conjunction or punctuation.

(a)
```
   ;: '3+%*/1+i.5'
+-+-+-+-+--+-+
|3|+|%|*|/|i.|5|
+-+-+-+-+--+-+
```

(b)
```
   ;: '(i.2)*+/^:2 i.2 3'
+-+--+-+-+-+--+-+--+-+
|(|i.|2|)|+|/|^:|2|i.|2|
+-+--+-+-+-+--+-+--+-+
```

25. Predict and check the results of the following.

(a) `]a=: i.3 4`
(b) `b=: i.2 3 4`
(c) `+/a`
(d) `+/b`
(e) `+/"1 a`
(f) `+/"1 b`
(g) `+/"2 a`
(h) `+/"2 b`
(i) `,/ a`
(j) `,/ b`
(k) `,./ a`
(l) `,./ b`
(m) `;/ a`
(n) `;/ b`

26. Predict and check the results of the following.

(a) `a=: 2 3 4;2 2$'a'`
(b) `b=: ;/i.5`
(c) `$a`
(d) `$b`
(e) `#a`
(f) `#b`
(g) `2 2$a`
(h) `2 2$b`
(i) `>2 2$a`
(j) `>2 2$b`
(k) `a;a`
(l) `<b;<b`

27. Write the following functions in common math notation (without simplifying).

(a) `f1=: 3&^`
(b) `f2=: ^&3`
(c) `f3=: 1p1&%`
(d) `f4=: 3 : '1 o. 2 * y'`
(e) `f5=: 3 : '%%:-*:y'`
(f) `f6=: 3 : '(^-(-:*:y))%%:2p1'`
(g) `f7=: 4 : 'x%1+y'`
(h) `f8=: 4 : 'x + 1 o. 3 + 2* 2 o. y ^ x'`

28. Using the definition of `rpoly` is defined as in Section 1.7 as a template, write an explicit monadic function (call it `urpoly` for unit regular polygon) that gives the same result as `1 rpoly y`.

29. Use `urpoly` from the previous exercise, along with the *dwin.ijs* script to define 11 colors (red to black) and 11 pentagons and display them as follows.

```
_1 _1 1 1 dwin 'polys'
$cs=: <.(0.1 *i.-11) */ 255 0 0
$ps=: (0.1 *i.-11) */ urpoly 5
cs dpoly ps
```

(a) Repeat the experiment with 21 seven sided figures.
(b) Repeat (a) with the scales `0.1 *i.-11` replaced by `0.9^i.21`.
(c) Create a nested polygon of your own design.

30. Predict and check the following arithmetic on _ (infinity) __ (negative infinity) and _. (indeterminate).

(a)	`_ + _`	(b)	`-_`	(c)	`%0`
(d)	`_ - _`	(e)	`*: _`	(f)	`% _`
(g)	`__ * __`	(h)	`%: __`	(i)	`% _.`
(j)	`3 + _`	(k)	`1 o. _`	(l)	`^ __`

Chapter 2 Plots, Verbs and First Fractals

In this chapter we will further develop tools for controlling graphics and we will apply those tools to data plotting and displaying two types of fractals. The tools we develop include rich facilities for function composition and we will get experience with geometric transformations of the plane. The first type of fractal that we will consider is based upon refinement of segments of a polygon by sets of smaller segments. Second, we will investigate iterated function systems which are lists of functions that can be used to create remarkable images.

2.1 Function Composition and Plots

Functions in J may be composed by the use of `@` which is called ***atop***. For example, we can define a verb to compute the reciprocal of twice a number via the following.

```
   f=: %@(2&*)

   f 4
0.125

   f 1 4
0.5 0.125
```

If we want the sum of the reciprocals of a list of numbers, we might try the following.

```
   g=: +/@%

   g 1 2 3
1 0.5 0.333333
```

This isn't what we want since the sum of each individual reciprocal is taken, rather than the sum of the entire list of reciprocals. We actually should use `@:` which is called ***at***. This is composition of infinite rank; that is, it creates the entire array resulting from application of the rightmost function and then applies the function on the left. Thus, in our example, it results in the sum of the entire list of reciprocals.

```
   g=: +/@:%

   g 1 2 3
1.83333
```

Functions in J are monads or dyads. When the composition of two functions is a dyad, we sometimes want the first function to be a monad and the second to be a dyad. Other times, it is the other way around. The dyad $f(x,y)=(x+y)^2$ can be conveniently implemented with atop.

```
   f=: *:@+

   3 f 4
49
```

Notice that atop is not the type of composition we want if we want the dyad $g(x,y)=x^2+y^2$ since we want to square both arguments before adding them. This can be accomplished with `&` (***compose***).

```
   g=: +&*:

   3 g 4
25
```

The above forms of compositions are summarized by the diagrams in Figure 2.1.1.

```
u@v y                    u&v y
          u                        u
          |                        |
          v                        v
          |                        |
          y                        y

x u@v y                  x u&v y
          u                        u
          |                      /   \
          v                    v       v
         / \                   |       |
        x   y                  x       y
```

Figure 2.1.1 Composition using Atop and Compose

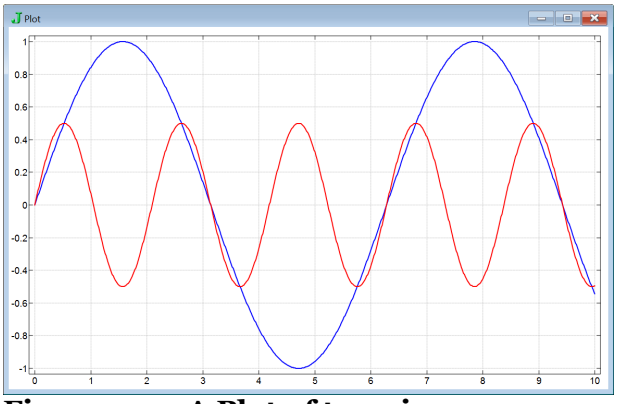


Figure 2.1.2 A Plot of two sines

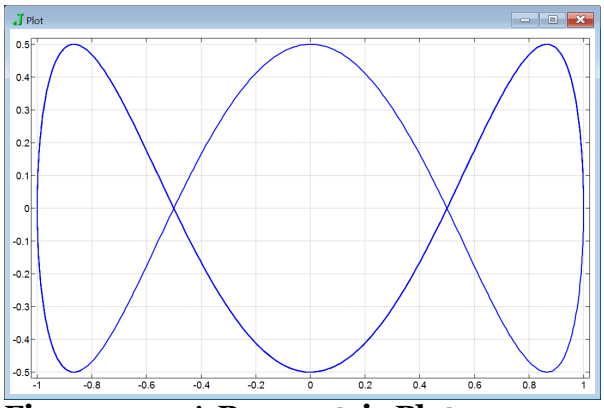


Figure 2.1.3 A Parametric Plot

We turn to plotting some functions resulting from compositions. The plot and numeric scripts contain several utility functions, including `steps`, which creates a uniform sequence of numbers useful as inputs to functions. First we plot the sine function.

```
   load 'plot numeric'

   steps 1 2 5
1 1.2 1.4 1.6 1.8 2

   ]t=: steps 0 10 1000
0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 0.11 0.12....

   sin=: 1&o.

   z=: sin t
```

Now we look at the function using the following.

```
   plot t ; z
```

Notice that the *t*-*z* data points are two lists adjoined via `;` which is link. We now use function composition in J to express the function $f(x) = 1/2 \sin(3x)$. Here we define the function, obtain its values and plot it along with the original sine function using a somewhat larger pen width.

```
   f=: (0.5&*)@sin@(3&*)

   z2=:f t

   'pensize 2' plot t; z,:z2
```

Figure 2.1.2 shows that plot. The plot function provides parametric plots since the lists in `t;z` can really be any values. For example, replacing `t` by `cos t` gives a circle which can be verified with the following.

```
   cos=: 2&o.

   z3=: cos t

   'aspect 1' plot z3;z
```

And we see a more complex parametric figure by plotting `z3` versus `z2`.

```
   'pensize 2' plot z3;z2
```

The result is shown in Figure 2.1.3. More details about the plot function and other plotting facilities can be obtained by running the J lab on plot (in the help-studio-labs menu) or by looking in the plot section of the user manual in the on-line help.

2.2 Experiment: Plotting Time Series, Functions and Curves

Often we have large amounts of data that we want to see at a glance. This may be time series data such as we might find in a list of closing prices of the Dow Jones stock market index or it may be values arising from a function or numerical calculation. Much like the list of function values considered in the previous section, we would like to be able to view the data at a glance. We will find it convenient to investigate a list of numbers generated by the Henon map. First we will discuss selecting items from a list. The dyadic verb `{` is called ***from***.

```
   ]d=: 11*1+i.10      a list of ten items
11 22 33 44 55 66 77 88 99 110

   0{d                 item with index 0 from d
11

   1{d                 item with index 1
22

   4 1{d               items with index 4 and 1
55 22

   10{d                there is no item with index 10
|index error
|   10    {d
|[-8]

   {.d                 the ***head***, or first item
11

   {:d                 the ***tail***, or last item
110

   _1{d                item with index _1 is the tail
110

   _2{d                item with index negative _2 is second to last
99
```

The 1-dimensional Henon map defines a sequence by $t_n = 1 + 0.3t_{n-2} - 1.4t_{n-1}^2$ with $t_0 = t_1 = 0$. We will implement this sequence by creating a function `hen` that applies to one pair of successive points (t_{n-1}, t_{n-2}) and results in the next pair (t_n, t_{n-1}).

```
   load 'plot'
```

```
   hen=: 3 : '({:y),1+(0.3*{.y)+_1.4* *: {:y'

   hen 0 0
0 1

   hen^:(i.5) 0 0
  0       0
  0       1
  1    _0.4
_0.4    1.076
1.076 _0.740886

   $tt=: hen^:(i.100) 0 0
100 2

   't nt'=: |: tt

   $nt
100

    plot t

  'point;pensize 5' plot t;nt
```

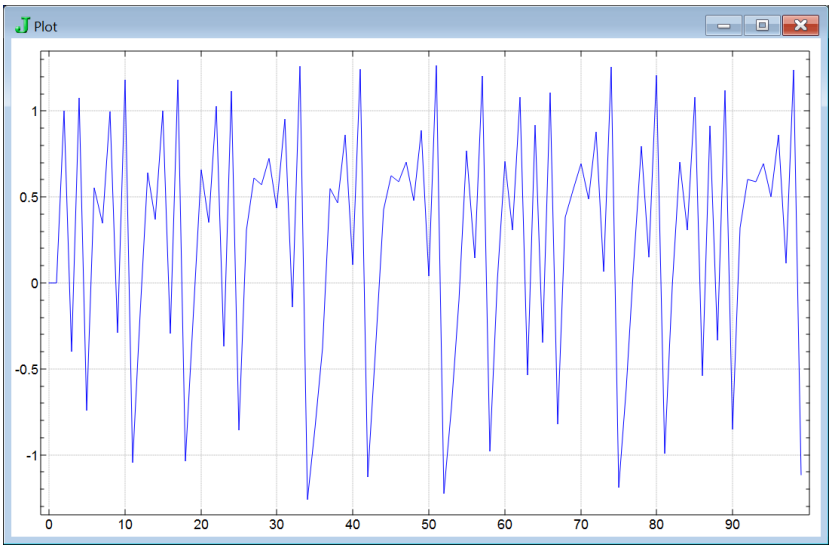


Figure 2.2.1 The Henon Map

We used parallel assignment to define `t` and `nt`. This was done by listing the names in quotes and having the number of names (two) correspond to the number of items in `|: tt`. Figure 2.2.1 shows the result of the first plot. Notice the near randomness in the figure but also notice the intriguing repetition of very similar blips. In the next chapter we will see that we can take advantage of those blips to make forecasts. Figure 2.2.2 shows the result of plotting `t` versus `nt` as seen in the last plot above. Since the result is not random, it is clear there is some correlation between successive entries in the Henon data.

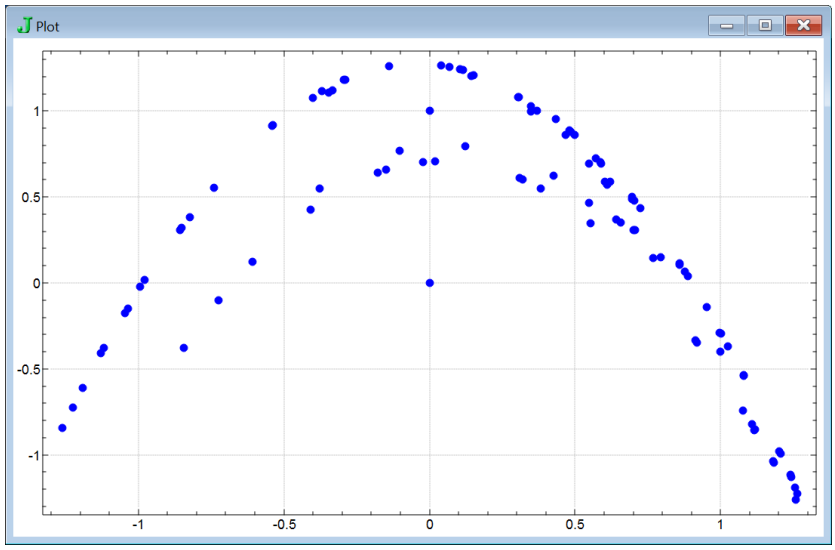


Figure 2.2.2 Henon Data t_k versus t_{k+1}

2.3 More Function Composition

In this section we consider composition of functions by lists of verbs; these are called ***verb trains***. A list of three functions in J is called a ***fork*** and has much the same meaning as expressions such as $f + g$ discussed in many pre-calculus and calculus courses. Namely, that $(f + g)(t)$ is the same as $f(t) + g(t)$. Likewise, we can replace plus by subtract, times, divide or any dyad. First consider an example using plus.

```
   fork=: 3&* + ^&5

   fork 2
38
```

```
   (3*2)+2^5
38
```

We see that the result of the fork is the sum of the results of "multiply by 3" and "raise to the fifth power" functions; this is commonly written as $f(t) = 3t + t^5$. In the next illustration we compute the range of a data set using a fork that adjoins the minimum and the maximum. We create the data using roll with fixed seed, `?.` so that we may exactly duplicate our experiments.

```
   ]data=: ?.10#100          random data
94 56 8 6 85 48 66 96 76 59

   55 <. 44                  minimum of the arguments (***lesser of***)
44

   min=: <./                 minimum insert gives min of a list

   min data
6

   max=: >./                 maximum (***greater of***) insert

   range=: min , max         min adjoin max

   range data
6 96
```

A fork that is very useful is one that gives the average of a list; that is, the sum of a list divided by the number of items in a list.

```
   avg=: +/ % #

   avg 3 14 7
8
```

When forks are used dyadically, the left and right tines are applied to both arguments. In this case it is useful to have left and right identity functions. For example, we can implement a function commonly written as $h(x, y) = x^2 + y^3$ as follows.

```
   3[4                       left argument
3

   3]4                       right argument
4

   h=: ^&2@[ + ^&3@]

   3 h 4
73

   (3^2)+4^3
73
```

Lists of two verbs are called ***hooks***. These also have a monadic and dyadic meaning.

```
   hook=: + *:               add to its square

   hook 3
12
```

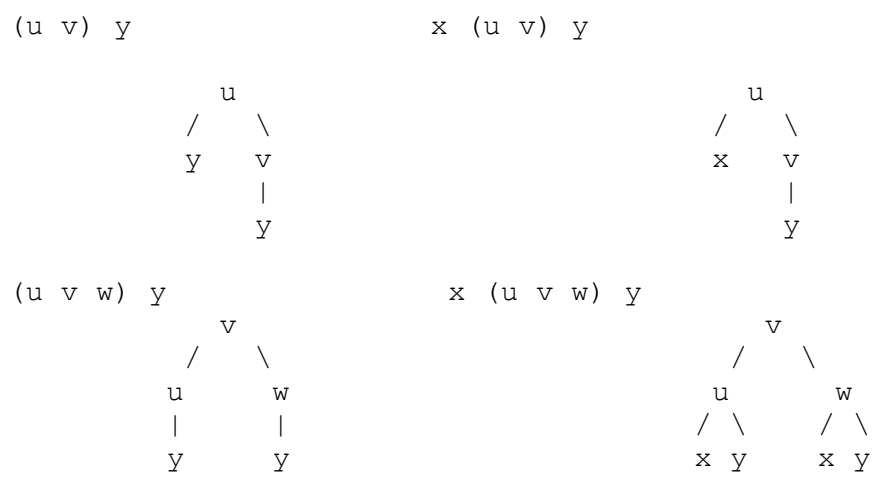


Figure 2.3.1 Diagrams of Hook and Fork Composition

```
   (+ *:) 3                    a hook is two verbs in isolation
12

   (] + *:) 3                  monadic hook is a fork with identity
12
```

The tilde symbol ~ is an adverb called ***reflex***. It puts its argument on both sides of the function. That is, `f~ y` is the same as `y f y`.

```
   *~5                         same as 5*5
25

   -~4                         same as 4-4
0

   ]a=: i.2 2
0 1
2 3

   ,.~a                        the array a stitched to itself
0 1 0 1
2 3 2 3
```

Now we are prepared to discuss a sequence of tacit functions that can be used to implement the step of constructing the Sierpinski triangle that we saw in Section 1.7.

```
   ]s=: 1 1$1
1

   s , s ,. s
1 0
1 1
```

Recall also that `sier=: 3 : 'y , y ,. y'` was the explicit form for the function that we iterated in order to build a raster Sierpinski triangle. Implementing a step tacitly with the argument `y` replaced by the identity function `]` also works.

```
   sier=: ] , ] ,. ]
```

```
   sier^:2 s
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1
```

The leftmost identity function is not necessary as it is implicit in the resulting hook.

```
   sier=: , ] ,. ]

   sier^:2 s
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1
```

The rightmost fork is unnecessary since it is just stitch with self. We are left with a very short tacit definition.

```
   sier=: , ,.~

   sier^:2 s
1 0 0 0
1 1 0 0
1 0 1 0
1 1 1 1
```

The result of iterating that more times may be viewed as we saw.

```
   load '~addons/media/imagekit/imagekit.ijs'

   view_data sier^:9 s
512 512
```

Summary diagrams of the hook and fork compositions are given in Figure 2.3.1. Longer verb trains are resolved right to left as usual. For example, the train of five verbs given by `f g h i j` is the same as a fork with a fork `f g (h i j)`. Indeed, our first tacit form for `sier` was of that type. Another example, assuming `load 'trig'` has been run, can be implemented in order to evaluate $f(x) = \sin(x) + \sin(2x) + \sin(3x)$ as follows.

```
   f=: sin + sin@(2&*) + sin@(3&*)

   f 0 1r4p1
0 2.41421
```

We will see dyadic examples of long trains of verbs in the next section.

We remark that there are constant functions for single digit integers that are the integer followed by a colon. Thus, `3:` is the constant function 3. Any noun can be made a constant verb by applying rank infinity to the noun.

```
    3: 1 2 3 4
3

   three=:3"_

   three 1 2 3 4
3
```

If we want to apply the function $f(x) = 3 + x^2$ to some data, we can do that with a suitable fork.

```
   (3: + *:) 1 2 3 4
4 7 12 19
```

The rules for a fork are slightly relaxed so that if the left tine is a noun, it is treated as a constant verb.

```
   (3 + *:) 1 2 3 4
4 7 12 19
```

Notice there is not a colon on the 3

2.4 Experiment: The Koch Snowflake

In this section we construct a fractal by iteratively refining a polygon. The fundamental step in this process is a refinement that replaces a segment, say from the point `A` to the point `B`, by four sub-segments. The first is a sub-segment that goes from the initial point `A` to a point, `A tri1 B`, that lies one third of the way toward the point `B`. The next sub-segment has the same length, but bulges away from the midpoint such that it would form one side of an equilateral triangle that would have a base on the removed middle third of the original segment. The third sub-segment is the other edge of the equilateral triangle and the final sub-segment consists of the last third of the original segment. See Figure 2.4.1.

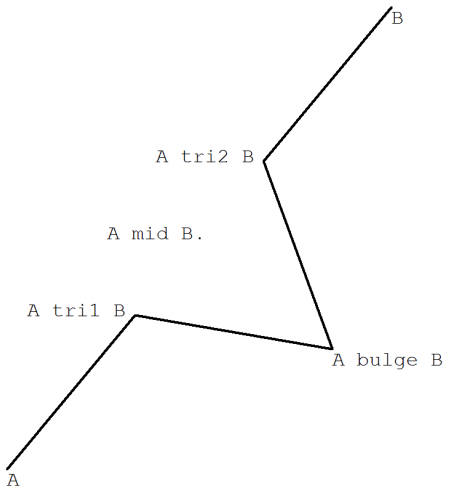


Figure 2.4.1 Refinement of a Segment

```
   mid=: -:@+
```

```
   1 3 mid 2 5
1.5 4
```

the midpoint of a segment

```
   tri1=: 2r3&*@[ + 1r3&*@]
```

the point one third along a segment

```
   1 3 tri1 2 5
4r3 11r3
```

```
   tri2=: 1r3&*@[ + 2r3&*@]
```

the point two thirds along a segment

We can obtain a vector normal to a vector in the plane by reversing the coordinates and changing the sign of the first entry.

```
   |. 2 1
1 2
```

reverse a list

```
   nor=: _1 1&*@|.
```

a direction normal to input

```
   nor 2 1
_1 2
```

```
   bulge=: mid + (%:%12)&*@nor@:-
```

bulge from the midpoint in the normal direction by a suitable factor

```
   1 3 bulge 2 5
2.07735 3.71132
```

The size of the factor used from the bulge is $(\sqrt{3}/2)(1/3) = 1/\sqrt{12}$ which is the relative height of the equilateral triangle times the one third length of its side. Next we put together the vertices of the subsegments in a list.

```
   segdiv=: [ , tri1 , bulge ,: tri2
```

```
   1 3 segdiv 2 5
      1       3
1.33333 3.66667
2.07735 3.71132
1.66667 4.33333
```

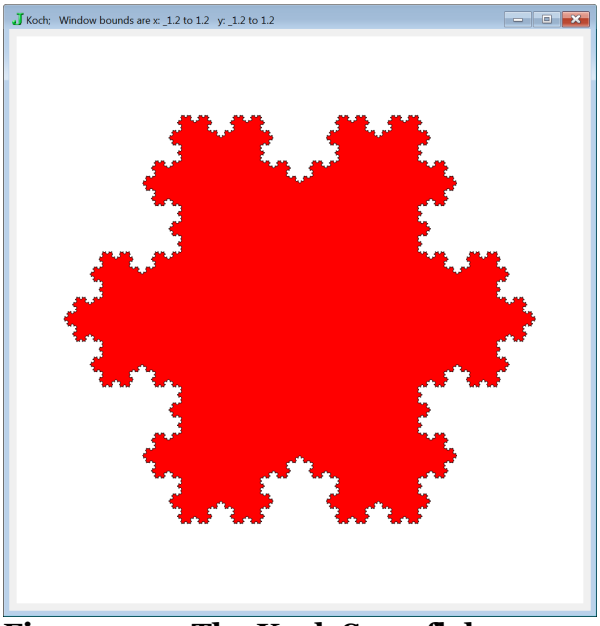


Figure 2.4.2 The Koch Snowflake

Notice `segdiv` is a train of seven verbs. Its left argument is the initial point of the input segment while the right argument is the terminal point. The result is a matrix whose rows are obtained by adjoining and laminating the following: the left argument (the initial point), the `tri1` point, the `bulge` point and the `tri2` point. We do not include the terminal point since it will arise as the first point in the subsequent segment.

```
   ]T=:|: 2 1 o./ 2r3p1*i.3
   1         0
_0.5  0.866025
_0.5 _0.866025
```

a triangle

```
   1&|. T
_0.5  0.866025
_0.5 _0.866025
   1         0
```

vertices ***rotated*** one position

```
   refine=: ,/@(] segdiv"1 (1&|.))
```

The verb `refine` applies `segdiv` to the list of vertices where the right argument is the next vertex. That results in a three dimensional list of lists of vertices. The application of `,/` then creates a single list of vertices.

We can take a look at the twelve vertices of `refine T` and then plot 5 levels of refinement in order to see the Koch snowflake as in Figure 2.4.2.

```
   path=:'~addons/graphics/fvj4/'

   require path,'dwin.ijs'

   _1.2 _1.2 1.2 1.2 dwin 'Koch'

   255 0 0 dpoly refine^:5 T
```

The following lines add less refined versions of the snowflake on top of the highly refined version with contrasting colors. Try them!

```
  255 255 0  dpoly refine^:4 T
   0  255 0  dpoly refine^:3 T
   0 255 255 dpoly refine^:2 T
   0  0  255 dpoly refine^:1 T
  255 0  255 dpoly T
```

2.5 Transformations of the Plane and Homogeneous Coordinates

When manipulating graphics images, it is useful to be able to exert detailed control over the image; in particular, it is useful to be able to geometrically transform the image. That is, we might want to elongate an image, rotate it a few degrees and translate before or after those actions. We first consider representing elongation and rotation with matrix multiplication and then we will see how homogeneous coordinates will allow us to also use matrix multiplication to do translations. We follow the convention that we multiply a point on the left by a matrix on the right in order to evoke the transformation associated with a matrix. This convention is convenient given our lists of points of polygons as items in a matrix. While this convention is common in computer science texts, the reverse convention is most common in mathematics texts so we should be alert to this convention when considering outside references.

If we want to elongate the first coordinate by a factor of r and the second coordinate by a factor of s, then we can use the following map. Here we are using the usual row by column matrix product. In casual use, elongation suggests an uneven expansion; however, we will use the word elongation for describing transformations of this form without regard to the values of the parameters r and s.

$$(x,y)_{new} = (x,y)\begin{pmatrix} r & 0 \\ 0 & s \end{pmatrix} = (rx, sy)$$

Figure 2.5.1 shows a solid rectangle that has been elongated into the dashed rectangle. The following J expression evokes that elongation.

```
   R                    the rectangle
  0 0
1.5 0
1.5 1
  0 1
  0 0

   m                    the elongation matrix
0.5   0
  0 1.5

   mp=: +/ . *          matrix multiplication

   R mp m               apply the elongation
   0   0
0.75   0
0.75 1.5
   0 1.5
   0   0
```

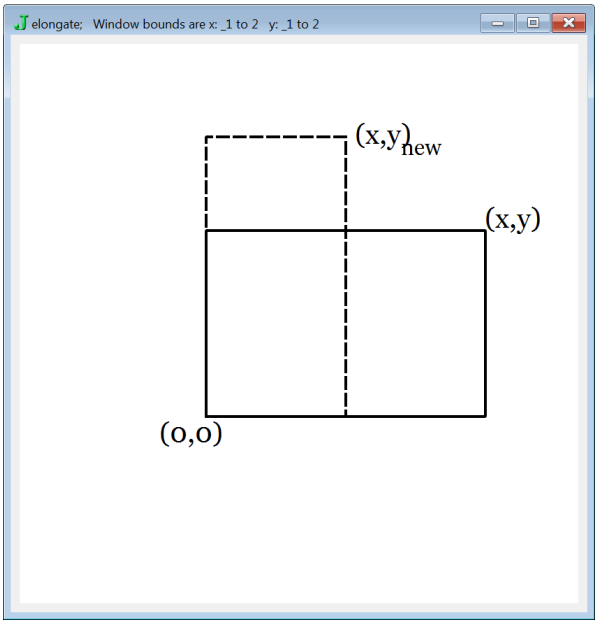


Figure 2.5.1 An Elongation

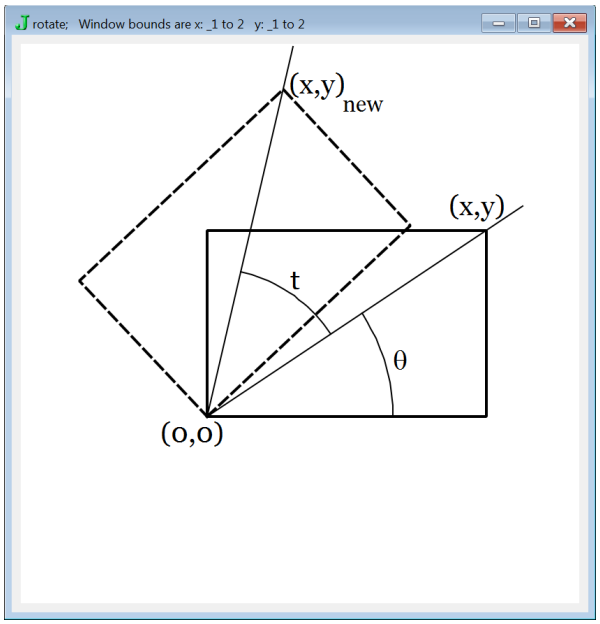


Figure 2.5.2 A Rotation around (0,0)

Notice the elongation matrix has factors of 0.5 and 1.5 and hence the rectangle is compressed by a factor of 0.5 in the horizontal direction and expanded by a factor of 1.5 in the vertical direction. If we want to rotate a point (x, y) counterclockwise around the origin by an angle of *t*, we can use the following transformation.

$$(x, y)_{new} = (x, y)\begin{pmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{pmatrix}$$

Figure 2.5.2 shows an example of the rectangle `R` rotated by an angle `6r25p1` around the origin. We can check that the formula for the new point is correct by observing that if (x, y) has the polar form $(x, y) = (r\cos(\theta), r\sin(\theta))$ then the point $(x, y)_{new}$ has angle $t + \theta$ in its polar form while the distance from the origin, r, is the same. Using trigonometric identities for the sum of angles, we see the following.

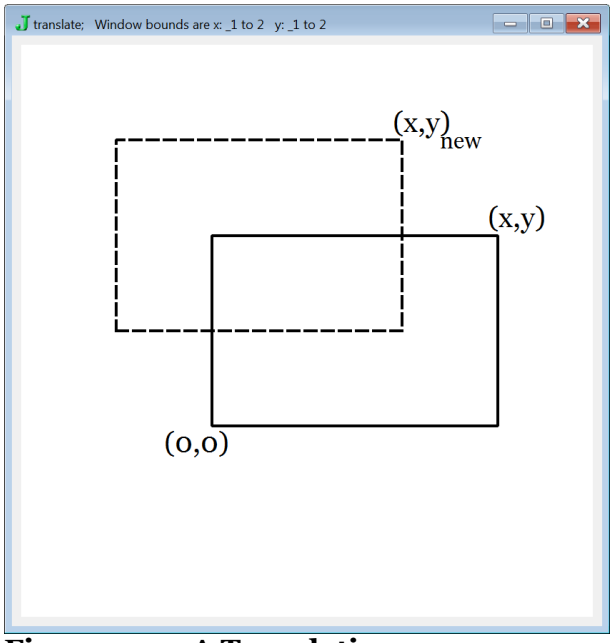


Figure 2.5.3 A Translation

$$(x, y)_{new} = (r\cos(t + \theta), r\sin(t + \theta))$$
$$= (r\cos(t)\cos(\theta) - r\sin(t)\sin(\theta), r\cos(t)\sin(\theta) + r\sin(t)\cos(\theta))$$
$$= (x\cos(t) - y\sin(t), x\sin(t) + y\cos(t))$$
$$= (x, y)\begin{pmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{pmatrix}$$

Next we consider translations. Geometrically and arithmetically the idea is simple. We want to move an object rigidly in the plane by adding some amount to each coordinate. The transformation has the following form.

$$(x, y)_{new} = (x + a, y + b)$$

This transformation can not be directly expressed as the point (x, y) times a matrix. However, if we add a trailing 1 to our points (x, y), then elongation, rotation and translation may all be represented by matrix multiplication. The advantage of this representation is that a sequence of geometric transformation may be applied by multiplying suitable matrices; but applying the sequence to a point requires only multiplication by the product matrix. Thus, $(x, y, 1)$ is defined to be the homogeneous coordinate representation of the point (x, y). The transformations that we have described take the following forms in homogeneous coordinates.

$$(x, y, 1)_{new} = (x, y, 1)\begin{pmatrix} r & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{elongation}$$

$$(x, y, 1)_{new} = (x, y, 1)\begin{pmatrix} \cos(t) & \sin(t) & 0 \\ -\sin(t) & \cos(t) & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{rotation}$$

$$(x,y,1)_{new} = (x,y,1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}$$ translation

In the next section we will use homogeneous coordinates to experiment with these transformations.

2.6 Experiment: Transformations and Animations

In this section we will experiment with using homogeneous coordinates to represent transformations of the plane.

We begin by creating an L shaped polygon and then we convert it to homogeneous coordinates by stitching a 1 onto the array. We assume that *~addons/graphics/fvj4/dwin.ijs* has been loaded.

```
   ]L=: 0 0,2 0,2 1,1 1,1 3,:0 3
0 0
2 0
2 1
1 1
1 3
0 3

   ]H=: L,.1        homogeneous form of L
0 0 1
2 0 1
2 1 1
1 1 1
1 3 1
0 3 1

   _4 _4 4 4 dwin 'H'

   dpoly H
```

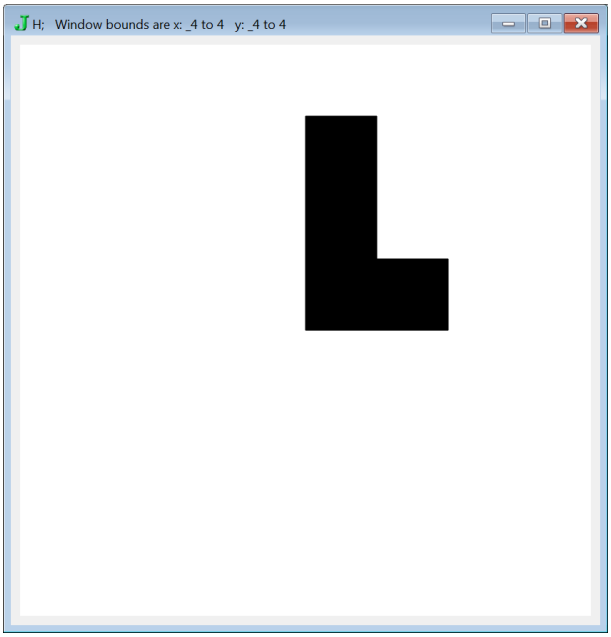


Figure 2.6.1 An L Shaped Polygon

In Figure 2.6.1 we see this polygon plotted. Notice that `dpoly` plots polygons in either Cartesian or homogeneous form. We implement an elongation matrix by first creating a 3 by 3 identity matrix using `=i.3` where the monadic = is self-classify. Then multiplying the items of the identity matrix times the diagonal elements gives the desired diagonal matrix.

```
   mp=: +/ . *

   ]I=: =i.3                    the identity matrix
1 0 0
0 1 0
0 0 1

   ]d=: 0.8 0.6 1 * I           the elongation matrix
0.8   0 0
  0 0.6 0
  0   0 1
```

```
   H mp d
  0   0 1
1.6   0 1
1.6 0.6 1
0.8 0.6 1
0.8 1.8 1
  0 1.8 1
```

We can view multiplication by a matrix as a verb (i.e., a transformation) by bonding the matrix to matrix multiplication.

```
   T=: mp&d
```

```
   T H
  0   0 1
1.6   0 1
1.6 0.6 1
0.8 0.6 1
0.8 1.8 1
  0 1.8 1
```

The constant functions `0:` and `1:` are useful for implementing a function that gives the homogeneous form of a rotation matrix. It is worthwhile to understand this definition. It is a train of 9 verbs with two of those verbs being trains of 5 verbs each. Yet the construction corresponds to adjoining the 9 entries of the desired matrix in a direct fashion.

```
   load 'trig'
```

```
   rotm=:(cos , sin , 0:),(-@sin , cos , 0:),: 0 , 0 , 1:
```

```
   rotm 1r2p1
```

rotation by $\pi/2$

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

```
   rotm 1r4p1
```

rotation by $\pi/4$

```
 0.707107 0.707107 0
_0.707107 0.707107 0
 0        0        1
```

Describe the geometric effect upon the polygon `H` of the homogeneous coordinate transformation associated with the following matrices.

```
   ]m1=: 2 3 1*I
2 0 0
0 3 0
0 0 1
```

```
   ]m2=: _1 1 1*I
_1 0 0
 0 1 0
 0 0 1
```

```
   ]m3=: 0.99 0.99 1*I
0.99    0 0
   0 0.99 0
   0    0 1
```

```
   ]m4=: (=i.2),2 3 1
1 0 0
0 1 0
2 3 1
```

```
   ]m5=: (=i.2),_2 _3 1
 1  0 0
 0  1 0
_2 _3 1
```

```
   ]m6=: 0 1 0,1 0 0,:0 0 1
0 1 0
1 0 0
0 0 1
```

```
   ]m7=: rotm 1r60p1
  0.99863 0.052336 0
_0.052336  0.99863 0
        0        0 1

   m4 mp m8
 0.93358 0.358368 0
_0.358368 0.93358 0
 0.792057 3.51748 1
```

```
   ]m8=: rotm 7r60p1
  0.93358 0.358368 0
_0.358368  0.93358 0
        0        0 1

   m8 mp m4
  0.93358 0.358368 0
_0.358368  0.93358 0
        2        3 1
```

We finish this section by looking at an animation. The left argument of `dawin` is a vector specifying the lower left and upper right corners of the drawing window followed by a delay, in seconds, between frames of the animation. The right argument is an array that gives a list of the polygons giving the frames in the animation.

```
   T=: mp&(m3 mp m7)

   _4 _4 4 4 0.2 dawin 'spin'

   255 0 0 dapoly T^:(i.301) H
```

How would you describe the animation? What happens if the delay is changed to `0`? How about `1`? What would you change in order to slow the shrinkage? How would you increase the number of frames? How would you reverse the process if `(0.05*L),.1` is the starting polygon?

2.7 Gerunds and Multiplots

We have seen that verbs listed in a train have a specific meaning that gives a type of composition of the functions. Sometimes we find it convenient to think of functions as formal objects that may be put into a list. When a verb is treated as a noun, it is known as a ***gerund***. Gerunds in J give us a convenient way to treat functions as objects that can be formally manipulated as data. We can apply every gerund in an array to data or we can select a particular function from the list to evoke in a given situation. Moreover, gerunds allow for the creation of very rich adverbs and conjunctions since any number of functions may be passed to adverbs or conjunctions using gerunds.

```
   g=: -@*:`-`%:        create a gerund

   $g                   it is a list of three atoms
3
   g                    its representation
+----------+-+--+
|+-+------+|-|%:|
||@|+-+--+|| |  |
|| ||-|*:||| |  |
|| |+-+--+|| |  |
|+-+------+| |  |
+----------+-+--+

   (g`:0) 2                 the phrase `:0 evokes each element of a gerund
_4 _2 1.41421

   i. 10
0 1 2 3 4 5 6 7 8 9

   3 | i. 10               modulo 3 remainders of indices
0 1 2 0 1 2 0 1 2 0

   g@.(3&|)"0 i.10          select function according to mod 3 remainders
0 _1 1.41421 _9 _4 2.23607 _36 _7 2.82843 _81
```

```
   2 2$g
+----------+----------+
|+-+------+|-         |
||@|+-+--+||          |
||  ||-|*:|||          |
||  |+-+--+||          |
|+-+------+|          |
+----------+----------+
|%:        |+-+------+|
|          ||@|+-+--+||
|          ||  ||-|*:|||
|          ||  |+-+--+||
|          |+-+------+|
+----------+----------+
```

2 by 2 array of functions

Notice that every third function is the opposite of the square of the argument. The others are negatives or square roots as appropriate. The conjunction `@.` is called ***agenda***.

We can use gerunds to handle plotting the graphs of multiple functions in one plot.

```
   f1=: 1&o.
```
$\sin(x)$

```
   f2=: %&2@f1@(2&*)
```
$1/2\sin(2x)$

```
   f3=: %&4@f1@(8&*)
```
$1/4\sin(8x)$

```
   fs=: f1`f2`f3
```
a list of those three functions

```
   load 'plot numeric'
   t=: steps 0 4p1 1000
```
a plot domain

```
   $fs`:(0) t
3 201
   'pensize 3' plot t;fs`:(0) t
```
plot all three functions

Figure 2.7.1 shows a plot of those functions. This can be automated into a cover function that takes the functions as a left argument and the endpoints of the domain with an optional number of steps as its right argument as follows.

```
   plotfcts=: 4 : 0
t=. steps 3{.y,1000
'pensize 3' plot t;x`:0 t
)

   fs plotfcts 0 4p1
```

While the above dyadic function works fine, we can also use user defined adverbs for constructions like those. User defined adverbs will be discussed in Chapter 5.

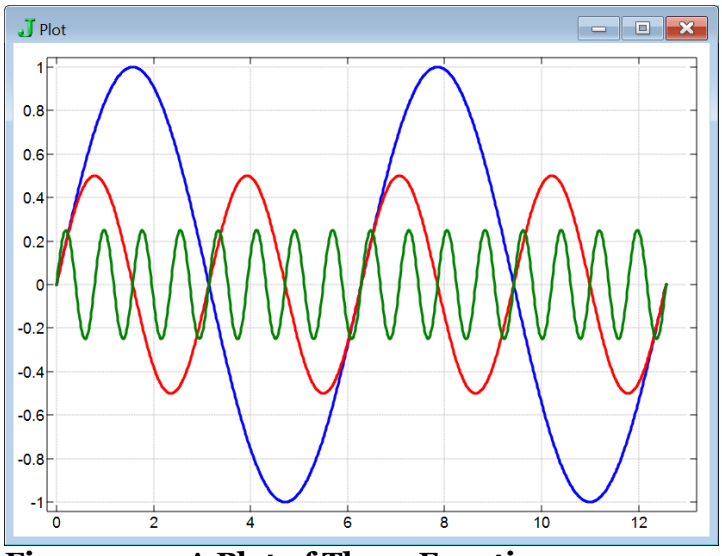


Figure 2.7.1 A Plot of Three Functions

2.8 Experiment: Collages of Transformations

In Section 1.5 we saw that the Sierpinski triangle could be created by a process of juxtaposition. Figure 1.7.1 shows the result. The symmetry appearing in the resulting fractal agrees with the following observations. If we contract the entire fractal by half, we get the lower left portion of the fractal. If we contract by half and add half in the vertical direction, then we obtain the upper left portion of the fractal. If we contract by half and then add half in the horizontal direction, then we obtain the lower right portion of the fractal. The following three matrices and the corresponding transformations implement the three transformations just described.

```
   m0                    m1                    m2
0.5   0 0           0.5   0 0           0.5   0 0
  0 0.5 0             0 0.5 0             0 0.5 0
  0   0 1             0 0.5 1           0.5   0 1

   mp=:+/ . *

   t0=: mp & m0          t1=: mp & m1          t2=: mp & m2
```

We create a gerund from those three transformations and evaluate all three transformations via `` `:0 `` in the function collage below. We define the unit square in homogeneous coordinates and see that the image of the square under the function `collage` is three smaller squares. We can then look at the image of those three squares after applying `collage` again and we get 3 sets of 3 squares (that is, 9 of them), and so on. Figure 2.8.1 shows four iterates of `collage` on the original square. Notice the form of the Sierpinski triangle is beginning to appear. The script *dwin.ijs* should be loaded.

```
   collage=: t0`t1`t2`:0

   sq=: 0 0,1 0,1 1,:0 1

   ]sq=: sq,.1
0 0 1
1 0 1
1 1 1
0 1 1

   collage sq
  0   0 1
0.5   0 1
0.5 0.5 1
  0 0.5 1

  0 0.5 1
0.5 0.5 1
0.5   1 1
  0   1 1

0.5   0 1
  1   0 1
  1 0.5 1
0.5 0.5 1

   $collage^:2 sq
3 3 4 3

   dwin 'collage 4'

   0 255 0 dpoly collage^:4 sq
```

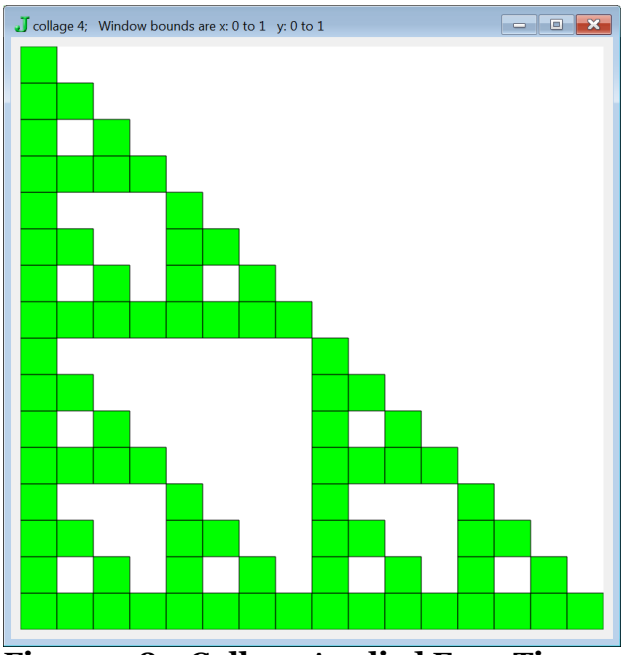


Figure 2.8.1 Collage Applied Four Times

What happens when you run six or seven iterates of `collage`? What does the fractal look like when the transformation `t2` is replaced by the transformation associated with one of the following matrices? The function `rotm` from Section 2.6 is assumed to be defined.

```
   ]m3=: m0 mp (rotm 1r60p1) mp (=i.2),0.5 0 1
 0.499315 0.026168 0
_0.026168 0.499315 0
      0.5        0 1

   ]m4=: m0 mp (rotm 1r6p1) mp (=i.2),0.5 0 1
0.433013     0.25 0
   _0.25 0.433013 0
     0.5        0 1

   ]m5=: m0 mp (rotm 1r2p1) mp (=i.2),1 0 1
   0 0.5 0
_0.5   0 0
   1   0 1
```

The list of functions we have given in collage are known as iterated function systems. There is a rich theory of these systems and we will return to further consideration of them in Section 4.4.

2.9 Simple Verbs

In Section 1.10 we saw that we can create very rich nouns; this includes character and boxed arrays. Verbs and higher order parts of speech are also available in J, but it is important to be comfortable with very basic verbs. This includes constant and identity functions. These verbs may seem almost trivial, and in some sense they are very simple; however, their use can also be subtle. We have already seen that their inclusion in the language makes it easy to express some functions very gracefully.

First we recall some of the basic verbs that we have already used briefly. Recall that the dyad `[` denotes left and `]` denotes right. Thus we can implement the dyad $h(x,y) = \sin(x) + \sin(y) + \sin\left(\sqrt{x^2 + y^2}\right)$ as follows.

```
   sin=: 1&o.

   h=: sin@[ + sin@] + sin@%:@+&*:

   4 h 6
_0.235845
```

We see that left and right can almost be thought of as standing for the "dummy" variables *x* and *y* in standard math notation. For example, we could have implemented $\sqrt{x^2 + y^2}$ that way.

```
   s=: %:@(*:@[ + *:@])

   3 s 4
5
```

In Section 2.6 we saw that constant functions for `0` and `1` were denoted `0:` and `1:` respectively. The notation for constant functions using a colon suffix is available for the integers between `_9` and `9`. However, we can change any noun into a verb of suitable rank by giving the noun as a left argument to rank. Recall that _ denotes infinity, so "_ denotes rank infinity. That is, it applies to the entire noun.

```
   3: i.5                    the verb 3: always results in 3
3

   (3: % 8:) i.5             a fork giving constant function three-eighths
0.375
```

```
   (3 % 8)"_ i. 5            a constant verb
0.375

   (3 % 8)"0 i. 5            gives three-eighths for each input atom
0.375 0.375 0.375 0.375 0.375

   'hi mom'"1 i.2 3          phrase is the result for each row in the argument
hi mom
hi mom
```

Using these ideas we can implement the quadratic function $f(x) = -1.4 + 3.1x + 6.5x^2$ as below.

```
   f=: _1.4"_ + (3.1"_ * ]) + (6.5"_ * ] ^ 2:)

   f 2
30.8
```

Recall the fact that tines of a verb train must be a list of verbs is slightly relaxed so that when the left tine is a noun `n`, the train treats it as `n"_` which allows for simplifications.

```
   fx=: _1.4 + (3.1 * ]) + 6.5 * ] ^ 2:

   fx 2
30.8
```

However, it actually is easier to take advantage of the built-in polynomial function `p.` if we need to evaluate a polynomial.

```
   g=: _1.4 3.1 6.5&p.

   g 2
30.8
```

In some situations we want to apply a monad at the end of a verb train. This can always be done using "atop" or "at" applied to the left-most dyad. However, because the verbs that are at an odd position away from the right end of a train are always dyads, this means the left tine of a fork may appear to the left of that last composition. That tends to hide the last composition and often requires extra parentheses. The ***cap*** `[:` facility helps clarify the writing of long trains. Cap appearing to the left of a dyad in a verb train eliminates the left argument that would come from that left position and evokes the monadic use of the verb to its right. Consider the function $h(x) = \sqrt{x^2 + \sin(x) + 1/x}$.

```
   h1=: *: %:@+ 1&o. + %

   h1 2
2.32579

   h2=: [: %: *: + 1&o. + %

   h2 2
2.32579
```

While neither `h1` nor `h2` are difficult to read, the fact that the square root is the outermost function is probably more apparent in `h2`.

Our next example uses floor, the greatest integer less than or equal to its argument, after addition of half, in order to round numbers to the nearest integer.

```
   <.2.1 2.5 2.9 3 3.1       floor
2 2 2 3 3

   r1=: <.@(0.5&+)           round
```

```
   r1 2.1 2.5 2.9 3 3.1
2 3 3 3 3
```

```
   r2=: [: <. 0.5&+                    round
```

```
   r2 2.1 2.5 2.9 3 3.1
2 3 3 3 3
```

Notice we avoided parentheses in `r2` although both solutions are straightforward.

2.10 Exercises

1. Write tacit verbs for computing the following functions.

(a) $f_1(x) = 1/(x+3)$ (b) $f_2(x) = \frac{1}{x} + 3$

(c) $f_3(x) = \sum_i x_i^2$ where x_i denotes the element in the list *x* at position *i*.

(d) $f_4(x) = \left(\sum_i x_i\right)^2$ where x_i denotes the element in the list *x* at position *i*.

(e) $f_5(x) = e^{-x^2}$ (f) $f_6(x) = \left(e^{-x}\right)^2$

(g) $f_7(x,y) = e^{x-y}$ (h) $f_8(x,y) = e^x - e^y$

2. Write the following in common math notation. For (a) and (b) give both (i) the monadic and (ii) the dyadic meaning. For (c) give the monadic meaning when the function is applied to a list.
(a) `f1=: ^@%` (b) `f2=: ^&%` (c) `f3=: (*/)@:(-&1)`

3. Plot 1000 points of the Henon map from Section 2.2. Do the blips persist that long?

4. We can create a random walk as follows.

```
   ]r=: _1+2*?.10#2                    random _1 or 1
_1 1 _1 1 _1 _1 1 1 1 _1
```

```
   ]ps=: +/\r                          partial sums of the data
_1 0 _1 0 _1 _2 _1 0 1 0
```

Create a random walk of that type with a thousand data points and plot it as a time series

5. Create a plot of the functions
(a) 1, x, x^2, ..., x^9 together on the interval $0 \le x \le 1$.
(b) 1, $1 - x^2/2!$, $1 - x^2/2! + x^4/4!$, and $1 - x^2/2! + x^4/4! - x^6/6!$ together on the interval $-6 \le x \le 6$.

6. Plot the polar curves below using the fact that polar curves can be viewed as parametric curves using $x = r\cos(\theta)$ and $y = r\sin(\theta)$ to write x and y in terms of a single parameter θ.

(a) $r = \cos(2\theta)$ (b) $r = \cos(3\theta)$ (c) $r = \cos(17\theta)$
(d) $r = \theta$ (e) $r = 2 + 3\cos(\theta)$ (f) $r = 2 + 3\cos(5\theta)$
(g) $r = 1 + \cos(\theta)$ (h) $r = 3\cos(3\theta/2)$ (i) $r = \cos(5\theta/3)$

7. Write tacit definitions for the following functions.

(a) $f_1(x) = x^2 + e^x$

(b) $f_2(x) = \sin(x) + \cos(x) + 1$

(c) $f_3(x, y) = e^{xy} - x + 2y$

(d) $f_4(x, y) = \dfrac{\sqrt{x + y}}{\sin(x + y)}$

8. Write the following monads in common math notation.

(a)
```
f1=: ^ + ^.
```
(b)
```
f2=: 0: >. ]
```
(c)
```
f3=: 1&o. + 2&o. + -
```
(d)
```
f4=: [: ^. ] + 1:
```

9. Write the following dyads in common math notation.

(a)
```
g1=: ^ + ^.
```
(b)
```
g2=: - >. +
```
(c)
```
g3=: [ ^ 2: + ] ^ 2:
```
(d)
```
g4=: ([ ^ 2:) + ] ^ 2:
```

10. What part of speech is reflex ~ ?

11. Apply the Koch refinement process to a regular (i) square and (ii) pentagon.

12. (a) Create the polygons shown in Figure 2.10.1.
(b) Apply the Koch snowflake refinement process to that figure.

13. Modify the Koch bump process from Section 2.4. Experiment with the result of using the factors `0.33`, `0.5`, `0.15`, and `_0.2` instead of `%%:12`.

14. In light of the previous exercise and the experiment at the end of Section 2.4:
(a) duplicate Figure 2.10.2. (b) duplicate Figure 2.10.3.

15. Implement the Henon map from Section 2.2 with a tacit definition.

16. The Fibonacci sequence is defined by $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. Implement a J function in the style of the Henon map from Section 2.2 that can be iterated to give the Fibonacci sequence.

17. A Tribonacci sequence is defined by $T_0 = 0$, $T_1 = 0$, $T_2 = 1$ and $T_n = T_{n-1} + T_{n-2} + T_{n-3}$ for $n \geq 3$. Implement a J function in the style of the Henon map from Section 2.2 that can be iterated to give the Tribonacci sequence.

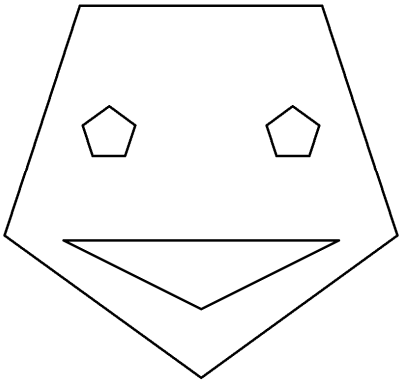


Figure 2.10.1 Polygons Smile

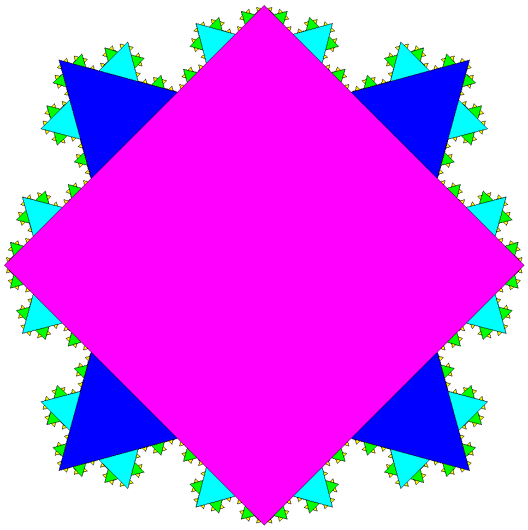


Figure 2.10.2 Mystery Fractal I

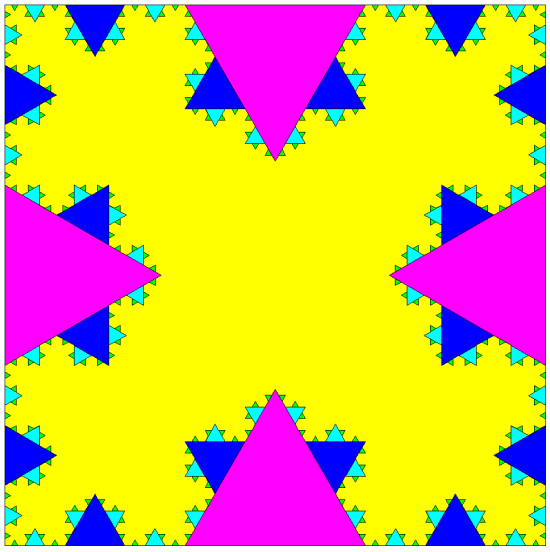


Figure 2.10.3 Mystery Fractal II

18. Describe in words the geometric effect of the transformation in homogeneous coordinates associated with the following matrices.

```
   mp=: +/ . *

   a0                  a1                  a2                  a3
1   0 0             1  0 0              1 0 0                0 1 0
0 0.5 0             0 _1 0              0 0 0               _1 0 0
0   0 1             0  0 1              0 0 1                0 0 1

   a4                  a5                     a1 mp a3            a3 mp a1
0 1 0                1 0 0              0 1 0                0 _1 0
1 0 0                0 1 0              1 0 0               _1  0 0
0 0 1              0.5 0 1              0 0 1                0  0 1

      a6                        a5 mp a6                  a6 mp a5
 0.707107 0.707107 0        0.707107 0.707107 0       0.707107 0.707107 0
_0.707107 0.707107 0       _0.707107 0.707107 0      _0.707107 0.707107 0
        0        0 1        0.353553 0.353553 1       0.5      0        1
```

19. Give the matrix in homogeneous coordinates that represents the following transformations.
(a) an elongation by a factor of 2 in the first coordinate and factor of one third in the second coordinate
(b) a reflection across the *y*-axis
(c) a rotation by 60 degrees clockwise
(d) a translation by 2 3
(e) a translation by 2 3 followed by a rotation by 60 degree clockwise
(f) a rotation by 60 degrees clockwise followed by a translation by 2 3

20. Suppose that we are given matrices `m1` and `m2` representing transformations of the plane in homogeneous coordinates. Let

```
   mp=: +/ . *
   t1=: mp&m1
   t2=: mp&m2
   c1=: t1@:t2
   c2=: t2@:t1
   p1=: mp&(m1 mp m2)
   p2=: mp&(m2 mp m1)
```

How are `c1` and `c2` related to `p1` and `p2`?

21. Give J expressions in the style of Section 2.6 to create an animation that
(a) translates `H` off the screen
(b) rotates `H` without changing its size
(c) contracts the image vertically until it disappears, while leaving the *x*-coordinate intact.
(d) like (c), but go further so that the last image is the reflection of `H` through the *x*-axis.
(e) spins a polygon around its center while the center point moves around a circle

22. Make a plot of the functions $e^{-x}\sin(kx)$ for $k = 1, 2, 3, 4$ on the same graph.

23. Make a plot of the functions $1/k \sin(kx)$ for $k = 1, 2, 3, 4$ on the same graph for $-3\pi \le x \le 3\pi$.

24. In light of your experience in plotting functions in Section 2.7, approximately duplicate Figure 2.10.4.

25. Try to predict the shapes indicated in the following J expressions. Test your answers.

```
      g1=: +`-`*:
      g2=: ,`,.
(a)   $g1
(b)   $g1,g2
(c)   $g1,:g2
(d)   $g1`:(0) 1 2 3
(e)   $g2 `:(0) 1 2 3
```

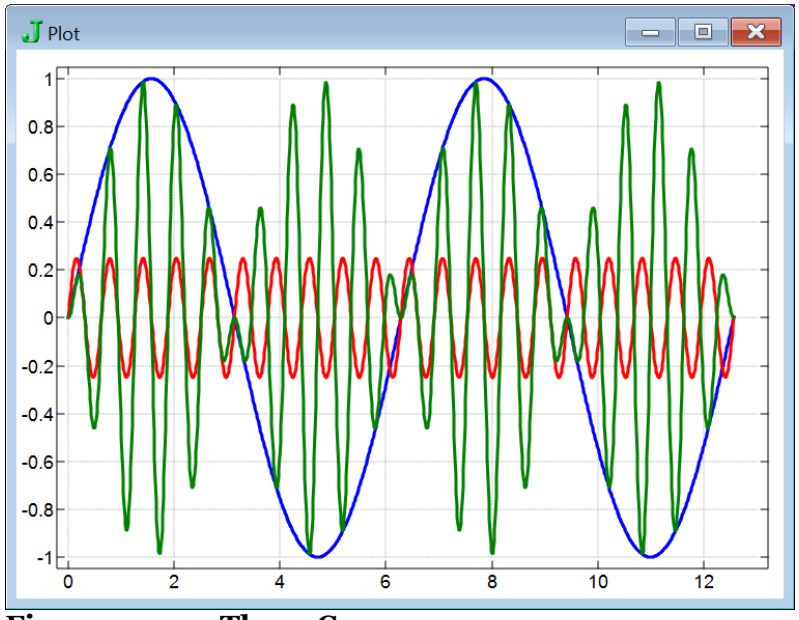


Figure 2.10.4. Three Curves

26. What happens when you iterate the collage function from Section 2.8 if you use a five pointed star for the initial polygon?

27. Create the fractal arising from iterating the collage function using the transformations associated with the following four matrices. What shape does it have after applying `collage` from Section 2.8 several times? Note: you will be able to give a much clearer answer in later sections.

```
   fm0                fm1                fm2                fm3
   0     0 0          0.85    _0.04 0      0.2    0.23 0     _0.15    0.26 0
   0 0.16 0           0.04     0.85 0   _0.26    0.22 0      0.28    0.24 0
0.25     0 1         0.0375   0.17 1      0.2  0.1025 1   0.2875  _0.021 1
```

28. (a) Identify seven one-third size transformations that take the fractal in Figure 2.10.5 to one-third size versions of itself.
(b) Recreate the fractal in the style of Section 2.8 by making a suitable collage based upon the transformations from (a).

29. Redefine the function `refine` from Section 2.4 using cap.

30. Predict the values of floor and ceiling in the following. Test your answers.

```
(a)   <. _1.1 _0.9 0 0.9 1.1
(b)   >. _1.1 _0.9 0 0.9 1.1
```

31. J labs provide introductions to many topics using interactive J. You may access J labs via the help-studio-labs menu. The labs associated with FVJ4 are titled in lower case, and come near the end. If they do not appear on the list, you can browse to the appropriate addon directory.
(a) Run the *dwin* lab, *dwin.ijt*, associated with the *graphics/fvj4* addon.
(b) Run the *imagekit* addon lab, *imagekit.ijt*, associated with the *media/image* addon.

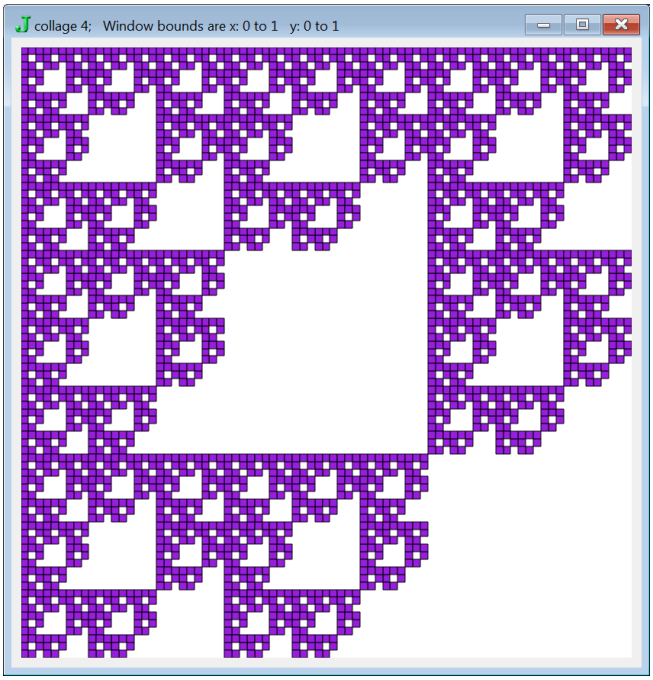


Figure 2.10.5 A Symmetric Fractal

Chapter 3 Time Series and Fractals

Time series may be thought of as data resulting from measurements of some phenomenon at regular time intervals. Time series may also be thought of as a list of random variables that could give rise to the data. The analysis of time series data requires interpretation of graphs and data analysis. Creating time series graphics is relatively direct visualization but we will also discuss more advanced graphics utilities. The previous chapters introduced most of the J facilities we need to do our analysis, but we will introduce a few additional J primitives that will be helpful. We will be able to see trends in data, identify correlated data, get experience with time series having a wide range of fractal dimensions and recognize situations where chaotic behavior might be playing a role in physical data. We will also consider a forecasting method that attempts to take advantage of underlying chaotic, but deterministic, behavior that at first glance might have appeared random.

3.1 Statistics and Least Squares Fit

We begin by recalling and implementing techniques for computing some elementary sample statistics. We also will see how to construct the line of best least squares fit to data. These will be basic utilities for our further work in this chapter.

```
   data=: 4 2 4 4 7 8

   min=: <./                    minimum

   min data
2

   max=: >./                    maximum

   max data
8

   range=: min , max            range

   range data
2 8

   +/data                       sum of the data
29
```

If we have a collection of data given by x_1, x_2, ..., x_n then the average of that data is $\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$ and the sample standard deviation is $s = \sqrt{\left(\sum (x_k - \bar{x})^2\right)/(n-1)}$. We can implement the numerator of the radicand in that formula relatively easily using verb trains including the use of cap, `[:` , to apply the desired monads.

```
   avg=: +/ % #                 average

   avg data
4.83333

   ()-avg) data                 subtract the average from the data
_0.833333 _2.83333 _0.833333 _0.833333 2.16667 3.16667

   ([:*: )-avg) data            squares of that
0.694444 8.02778 0.694444 0.694444 4.69444 10.0278

   ([:+/ [:*: )-avg) data       sum of the squares of that
24.8333
```

```
   #data                          number of data items
6

   <:@# data                      predecessor of the number of data items
5
```

The square root of the quotient gives the desired formula for standard deviation.

```
   sd=: [: %: ([: +/ [: *: ] - avg) % <:@#

   sd data
2.2286
```

When we have data that seems to fall into a nearly linear pattern, we want to be able to find the line that is the best fit. The most common type of best fit is given by seeking the line for which the sum of the squares of the differences of the data from the linear model is least.

We create some data that is reasonably close to being on the line $y = 12 + 2x$. We use `?.` which gives ***random*** numbers with fixed seed (so we can easily duplicate our "random" experiment). We then order the data using `/:~` which is "***sort up***".

```
   ?.10$100
94 56 8 6 85 48 66 96 76 59

   ]x=: /:~?.10$100                ordered data
6 8 48 56 59 66 76 85 94 96

   ]y=: (_11+?.10#21)+12+2*x       perturbed linear data
20 35 111 117 125 150 164 176 210 195
```

You can plot this data using segments or points as follows.

```
   load 'plot'

   plot x;y

   'point;pensize 4' plot x;y
```

Next, we seek the line $y = a_0 + a_1 x$ of best fit, that is, we are seeking the least square solution to the system at the right.

$$\begin{pmatrix} 1 & 6 \\ 1 & 8 \\ 1 & 48 \\ 1 & 56 \\ 1 & 59 \\ 1 & 66 \\ 1 & 76 \\ 1 & 85 \\ 1 & 94 \\ 1 & 96 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 20 \\ 35 \\ 111 \\ 117 \\ 125 \\ 150 \\ 164 \\ 176 \\ 210 \\ 195 \end{pmatrix}$$

There is no exact solution, but the least squares solution may be computed in J. In general, `b %. A` gives the least square solution, z, to the linear system $Az = b$. In our example we get the following least square solutions and use polynomial, `p.`, to create a function that gives the result of the least squares fit.

```
   ]coef=: y %. 1,.x
13.1366 1.97245

   f=: coef & p.

   f 0 6 96
13.1366 24.9713 202.492
```

Hence we see $y = 13.1366 + 1.97245x$ is approximately the best fit line to the data. Since the actual line that we randomized was $y = 12 + 2x$, this is not an unreasonable answer. We can plot the best fit line and the data as follows.

```
plot x;y,: f x
```

However, it would be much nicer to use points for the points and a line for the best fit line. This is easily done using plot driver commands, rather than making a single plot. In the next section we will do that.

3.2 Experiment: Plot Driver

In this section we give some illustrations of using the plot driver to create plots with a variety of information on them. The J plot utilities are loaded by running the plot script.

We assume the `x` and `y` data vectors and the function `f` that gives the least squares linear fit as defined in the previous section. We saw how to plot the line data and fit on the same graph using `plot`.

We want to plot both the data as points and the best fit line as a line, together on the same plot. This is easy using `pd` (plot driver) commands. Figure 3.2.1 shows the result of the following.

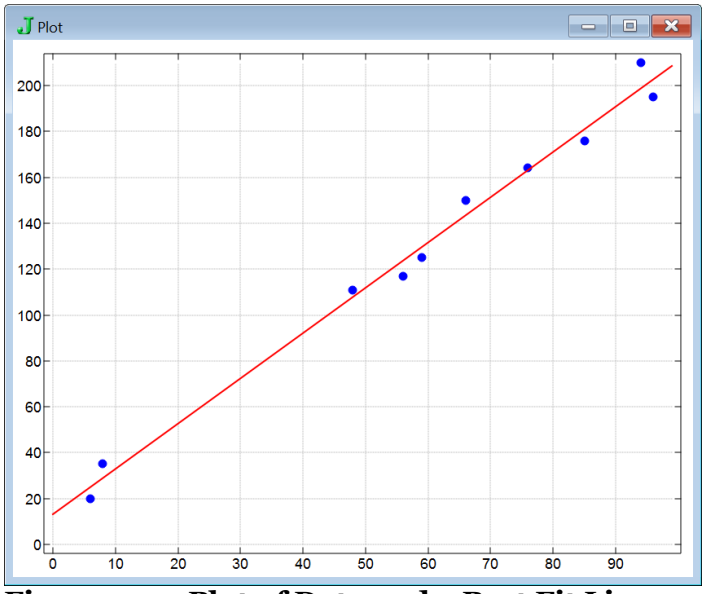


Figure 3.2.1 Plot of Data and a Best Fit Line

```
require 'plot'            loads the plot utilities if not already loaded
pd 'reset;'               clears the screen
pd 'type point'           sets the graphics type to display points
pd 'pensize 5'            sets the size of the points
pd 'itemcolor 0 0 255'    sets the color to blue
pd  x;y                   plots the data points
pd 'type line'            sets the graphics type to line
pd 'pensize 2'            sets the line width
pd 'itemcolor 255 0 0'    sets the color to red
pd 0 99;f 0 99            plot the least squares line
pd 'show'                 show the resulting graphics
```

The user manual in the on-line help contains the details of the commands available in the plot driver. Many illustrations using the plot driver utilities can be found in the plot demos and laboratories.

Now create a plot of a new data set along with the line of best least squares fit on the same plot. Use the following data.

```
x=: /:~?.100$1000
y=: (_15+?.100$31)+120-3*x
```

3.3 Random Walks

We begin our examples of time series by creating some simulations of random walks. Several types of random data will be created. These provide a convenient source of data for our illustrations and provide

experience with some random time series phenomena.

We begin with an example of a random walk where we make a sequence of steps of plus or minus one and consider our position at time *t* to be the sum of the steps up to index *t*.

```
   ?. 7 $ 2                    random choice of 0 or 1
0 0 0 0 1 0 0

   (?. 7 $ 2){_1 1             random choice of _1 or 1
_1 _1 _1 _1 1 _1 _1

   ]d=: _1 1 {~ ?. 7 $ 2       using ***commute***: x{y is y{~x
_1 _1 _1 _1 1 _1 _1
```

Now we use the adverb \ to create prefixes and length 5 infixes. The list of partial sums results from plus insert on the prefixes.

```
   <\i.5                       ***prefixes*** on indices
+-+---+-----+-------+---------+
|0|0 1|0 1 2|0 1 2 3|0 1 2 3 4|
+-+---+-----+-------+---------+

   <\d                         prefixes on our random data
+--+-----+-------+----------+------------+---------------+------------------+
|_1|_1 _1|_1 _1 _1|_1 _1 _1 _1|_1 _1 _1 _1 1|_1 _1 _1 _1 1 _1|_1 _1 _1 _1 1 _1 _1|
+--+-----+-------+----------+------------+---------------+------------------+

   5 <\ d                      ***infixes*** of width 5
+-------------+-------------+-------------+
|_1 _1 _1 _1 1|_1 _1 _1 1 _1|_1 _1 1 _1 _1|
+-------------+-------------+-------------+

   +/\ d                       the partial sums
_1 _2 _3 _4 _3 _4 _5

   load 'plot'

   'pensize 3' plot +/\_1 1{~?. 300 $ 2
```

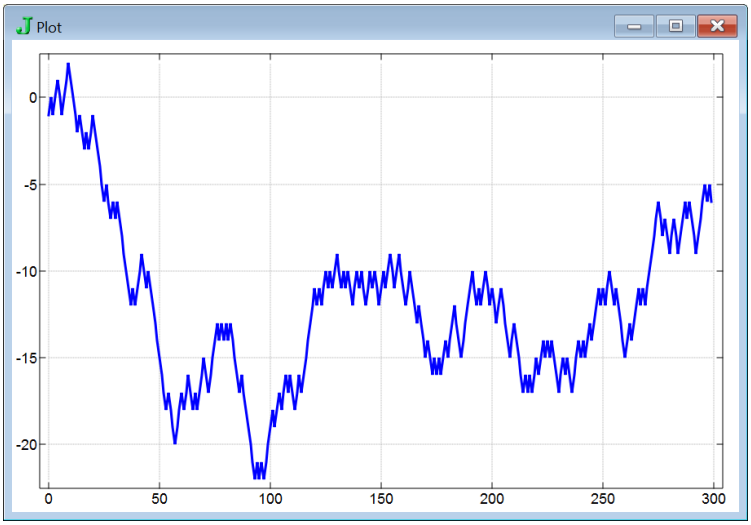


Figure 3.3.1 A Random Plus or Minus One Walk

Figure 3.3.1 shows the result of that 300 step random walk.

Next we aim to create an example of a walk with steps from a random uniform distribution of numbers between `_1` and `1`. That is, our steps can have any value between `_1` and `1` with equal likelihood. We begin by creating arrays with entries between 0 and 1.

```
   ?. 0
0.038363

   ?.@($&0) 2 3
0.038363  0.329284 0.335644
0.985972 0.0583756 0.282326
```

Thus the above function creates an array of pseudo-random real numbers between `0` and `1` with the specified shape. The function `randunif` defined below has that function as its monadic definition while the dyadic use (the verb to the right of the isolated colon) maps the random numbers into an interval specified by the left argument. Note the dyad uses `$:` which is ***self-reference*** and ***random*** `?` (without the period) which uses the current seed rather than the default seed. Thus, repetitions of experiments with `randunif` will vary.

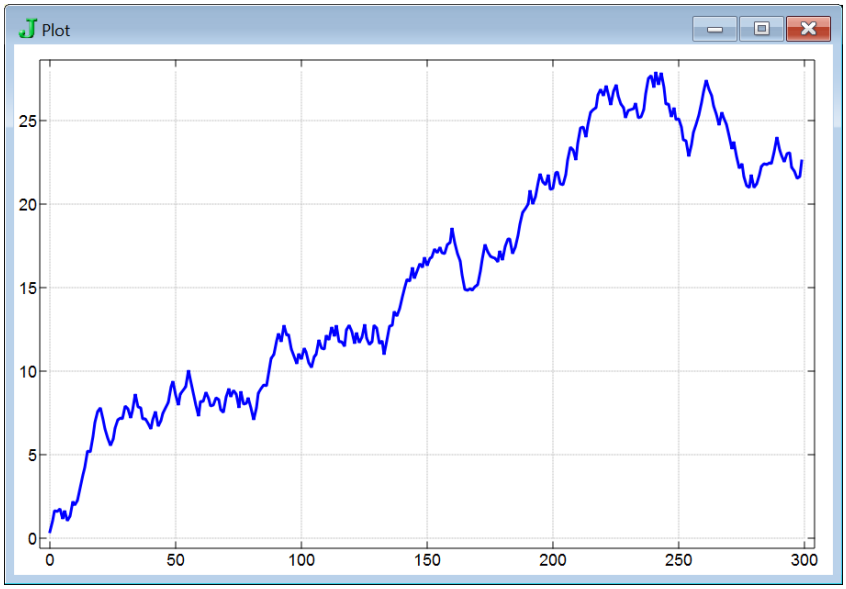


Figure 3.3.2 A Random Uniform Walk

```
   randunif=: ?@($&0) : ({.@[+({:-{.)@[*$:@])

   randunif 2 3                    array with entries between 0 and 1;
0.622471 0.324707 0.907825         results vary with random seed
0.0631566  0.38662 0.338598

   _1 2 randunif 2 3               array with entries between _1 and 2
_0.712699  0.804436 _0.143027
  1.78015 _0.100978  0.252813
```

We can do a random walk of 300 uniformly distributed steps using the following. An example is shown in Figure 3.3.2 (results will vary). The walk seems a little smoother than the walk seen in Figure 3.3.1 but is fairly similar.

```
   'pensize 3' plot +/\_1 1 randunif 300
```

Next we consider the case of random walks with normally distributed steps. We use the fact that if u and v are uniformly distributed on the interval from 0 to 1, then $\cos(2\pi u)\sqrt{-2\ln(v)}$ has a standard normal distribution. We assume `avg` and `sd` are defined as in Section 3.1 and that *plot.ijs* has been loaded so that `cos` is defined. Note `o.` is the "***pi times***" function.

```
   o. 0 1 2
0 3.14159 6.28319

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

   randsn 2 3                      array of standard normal values
```

```
_0.208032 0.677887 _1.71958
 0.250948 _1.04687 _1.48519
```

```
   d=: randsn 10000          10,000 standard normal values
```

```
   avg d                     average is near 0 (results will vary)
0.0168546
```

```
   sd  d                     standard deviation is near 1
1.00462
```

```
   'pensize 3' plot +/\randsn 300
```

We see that the sample average and standard deviation are quite close to the expected values and the plot of a 300 step random walk with standard normal steps looks similar to the random walks we have already seen. These walks are also called Gaussian walks. An example is shown in Figure 3.3.3. In the next section we will see the data from a random standard normal walk along with a trend curve.

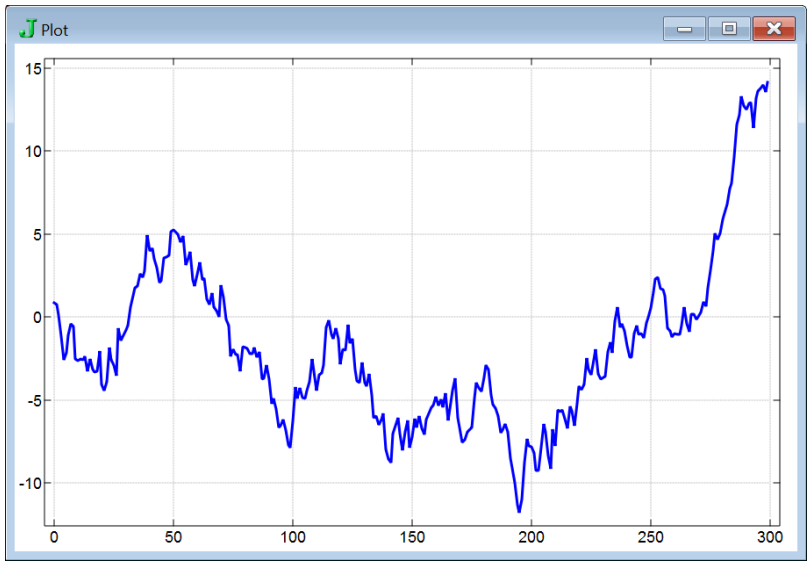


Figure 3.3.3 A Random Gaussian Walk

3.4 Experiment: Observing Trends

A simple way to identify trends is via a moving average. We associate with each time period an average over some subinterval of times near the point. First we will apply moving averages of equal weights and then we will implement Spencer's 15 point moving average. We begin with some simple data. We assume `avg` from Section 3.1.1 has been defined.

```
   ]d=: ?.7$100              seven random data points
94 56 8 6 85 48 66
```

```
   3 <\ d                    contiguous sets of size 3 using infixes
+-------+------+------+-------+--------+
|94 56 8|56 8 6|8 6 85|6 85 48|85 48 66|
+-------+------+------+-------+--------+
```

```
   3 avg\ d                  average the triples
52.6667 23.3333 33 46.3333 66.3333
```

Since we only get 5 triples, the number of data elements and number of averages are different. If we extend our data suitably before applying the averages, we can compare the data to the averages. Before we extend our data that way, we pause to further discuss taking and dropping elements from a list.

```
   3 {. d                    ***take*** the first 3 items
94 56 8
```

```
   _3 {. d                   take the last 3
85 48 66
```

```
   3 }. d                    ***drop*** the first 3
6 85 48 66
```

```
   {. d
94
```
head: the first item

```
   }. d
56 8 6 85 48 66
```
behead: drop the first

```
   {: d
66
```
tail: the last item

```
   }: d
94 56 8 6 85 4
```
curtail: drop the last

We can extend the data in a constant manner at both ends using the verb train: the head adjoin the identity function adjoin the tail. Then we can compare averages with the original data.

```
   ]xd=:({.,],{:) d
94 94 56 8 6 85 48 66 66
```
extend data on the ends

```
   d,:3 avg\ xd
  94       56        8  6       85       48 66
81.3333 52.6667 23.3333 33 46.3333 66.3333 60
```
compare data with moving averages

```
   ((3#{.),],3#{:) d
94 94 94 94 56 8 6 85 48 66 66 66 66
```
extend data on the ends three times

In practice we might want to use averages over longer intervals and we most likely would want to weight the values near the center more than the values far away from the point.

The Spencer 15 point average uses neighborhoods that are 15 elements wide with coefficients chosen carefully so that cubic and lower degree patterns will not be destroyed by the averaging process. The function `locspen` (local Spencer) applies the process to size 15 neighborhoods. It uses the ordinary dot product `+/ . *` in order to take the appropriate sum of weights times the data entries. The function `spencer` extends the data by 7 points on each end (repeating the endpoints) and then applies the Spencer averaging to the size 15 infixes.

```
   3 4 5 6 * 1 2 2 1
3 8 10 6
```
pairwise products

```
   +/ 3 4 5 6 * 1 2 2 1
27
```
sum of pairwise products

```
    3 4 5 6 +/ . * 1 2 2 1
27
```
sum of pairwise products using `+/ . *`

```
   ]wts=: (|.,}.)74 67 46 21 3 _5 _6 _3%320
_0.009375 _0.01875 _0.015625 0.009375 0.065625 0.14375
0.209375 0.23125 0.209375 0.14375 0.065625 0.009375 _0.015625 _0.01875
_0.009375
```

```
   locspen=: (+/ . *)&wts
```

```
   locspen 1 14#1 0
_0.009375
```

```
   spencer=: 15 locspen\ (7 # {.) , ] , 7 # {:
```

```
   walk=: +/\randsn 100
```

```
   pd 'reset;'
   pd 'type line'
   pd 'pensize 2'
   pd 'itemcolor 255 0 0'
```

```
pd walk
pd 'type line'
pd 'pensize 3'
pd 'itemcolor 0 0 255'
pd spencer walk
pd 'show'
```

Figure 3.4.1 shows the Spencer average applied to that random standard normal walk of length 100.

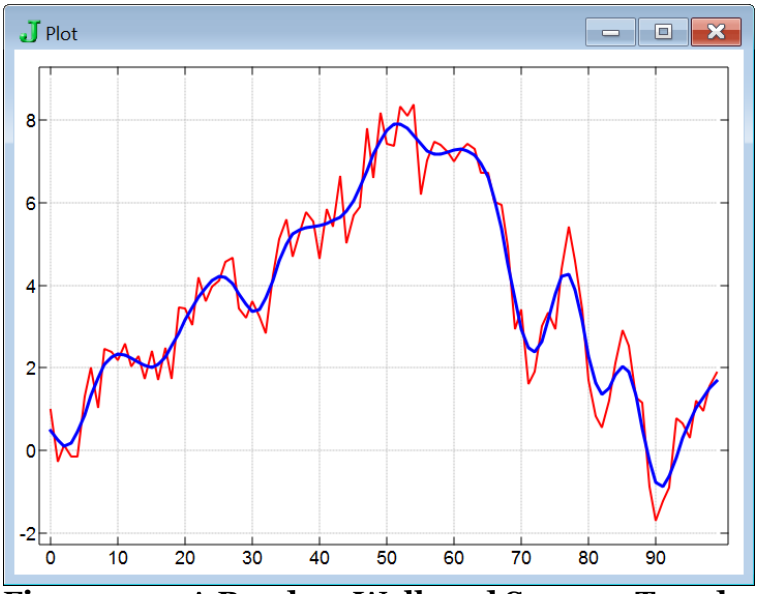


Figure 3.4.1 A Random Walk and Spencer Trend

A few sets of data are defined by *~addons/graphics/fvj4/ts_data.ijs* One is the Lake Huron levels, called `huron` while another is `sunspots`. Figure 3.4.2 shows the Spencer average applied to data giving the levels of Lake Huron. Duplicate Figure 3.4.2. Then apply Spencer averaging to the sunspots data and plot both the data and the trend on the same plot.

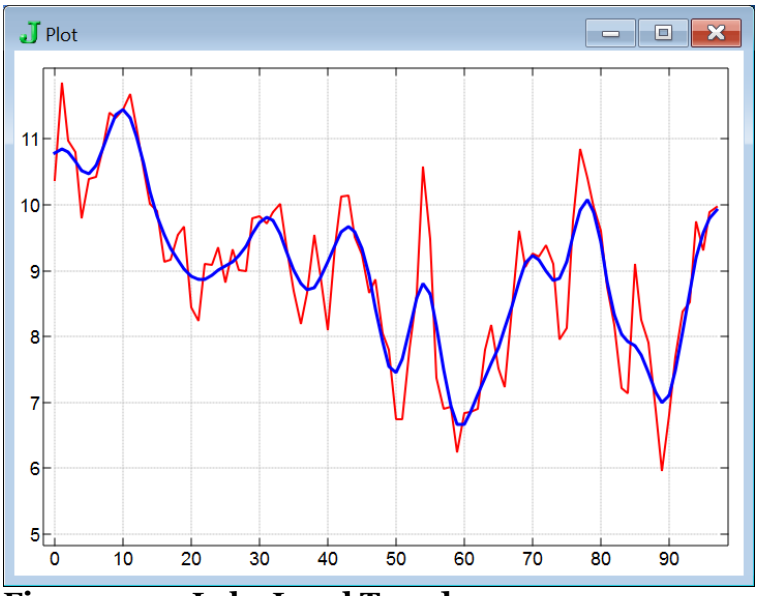


Figure 3.4.2 Lake Level Trend

3.5 R/S Analysis, the Hurst Exponent, and Sunspots

Rescaled range analysis was developed by Hurst in order to measure the persistence of time series. He was studying the annual Nile river discharge and noticed that a rise was often followed by an additional rise and likewise a drop was usually followed by a further drop. We will illustrate this technique with random walks and annual sunspot activity data.

Let x_t denote our time series data at time t and let $\bar{x}_{t,N}$ denote the average on a contiguous block of data of length N beginning at time t and let $S_{t,N}$ denote the sample standard deviation on the same block. We define the range $R_{t,N}$ on the block by the following.

$$R_{t,N} = \max_{1 \le k \le N} \left\{ \sum_{i=1}^{k} (x_{t+i} - \bar{x}_{t,N}) \right\} - \min_{1 \le k \le N} \left\{ \sum_{i=1}^{k} (x_{t+i} - \bar{x}_{t,N}) \right\}$$

Then we average the ratios $R_{t,N}/S_{t,N}$ over all the successive non-overlapping subintervals of length N (discarding trailing fragments). We obtain the average quotient which we denote $\overline{Q}_{t,N}$. The idea is that $R/S \approx \overline{Q}_{t,N} \approx cN^H$ would be a generalization of the special case when the time series has Gaussian distributed steps in which case it is known that the exponent is $H = 0.5$. We can estimate the Hurst exponent using a log-log plot. In particular, the slope of the least squares fit line to the data

$\left(\ln(N), \ln(\overline{Q}_{t,N})\right)$, where $N = 2,3,\ldots,T/2$ and T is the length of the entire time series, gives an estimate for the Hurst exponent.

A time series that has $H = 0.5$ corresponds to the steps in a random Gaussian walk. Moreover, the range of Hurst exponents $0.5 < H \leq 1$ corresponds to a persistent walk. That is, a move in one direction tends to be followed by a move in the same direction. The range of Hurst exponents $0 < H < 0.5$ corresponds to an anti-persistent walk. That is, a move in one direction tends to be followed by a move in the opposite direction.

We first estimate the Hurst exponent for a sequence of random, normally distributed random numbers. We assume that `avg`, `min` and `max` from Section 3.1 have been defined and `randsn` from Section 3.3 has been defined. In order to be able to replicate our experiment, we will set the seed used by the J random number generator. This is done using the foreign conjunction with arguments `9` and `1`. The set of sums in our definition of $R_{t,N}$ forms a cumulative centered distribution which we will compute using the function `ccd`. The function `Q` computes $\overline{Q}_{t,N}$ on the block which is its argument. The standard deviation is estimated by `S`, which is a variant of `sd` from Section 3.1 with denominator given by `#` instead of `<:@#`.

```
   setseed=: 9!:1                    foreign conjunction to set random seed
   setseed 7^5                       set it to its default initial value
   ]s=: randsn 5
_0.514477 1.23645 _0.353373 _0.522193 1.23505
   ccd=: [: +/\ ] - avg              cumulative centered distribution
   ccd s
_0.730768 0.289392 _0.280273 _1.01876 2.22045e_16
   R=: (max-min)@:ccd                range
   R s
1.30815
   S=: [: %: ([: +/ [: *: ] - avg) % #       standard deviation
   Q=: R % S                                 rescaled range
   Q s
1.56746
```

Now we turn to applying `Q` on various blocks and averaging the result. It is convenient to use the ***cut*** conjunction in order to obtain the desired subintervals. The right argument specifies the type of cut; we will use `_3` which gives suitable tesselations. The left argument of the resulting function specifies the relative movement vector and the size of the tesselation. For us, we will use movement vectors and sizes that are both the same scalar which will result in applying our function to non-overlapping blocks. The use of cut is a convenience since infixes

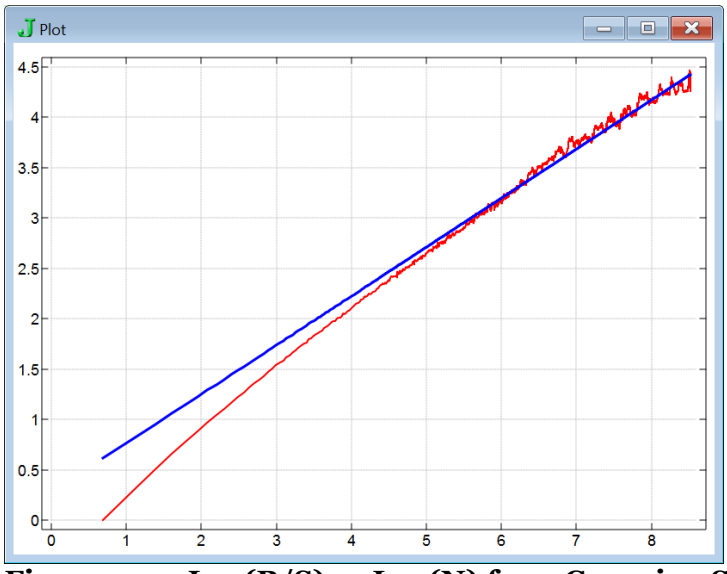


Figure 3.5.1 Log(R/S) vs Log(N) for a Gaussian Series

with negative left arguments give the same result except they sometimes include fragments at the end which we do not want.

```
   ?.10#10                    some random numbers
4 6 8 6 5 8 6 6 6 9

   (5,:5) <;._3 ?.10#10       subintervals of size 5
+---------+---------+
|4 6 8 6 5|8 6 6 6 9|
+---------+---------+

   (4,:4) <;._3 ?.10#10       subintervals of size 4
+-------+-------+
|4 6 8 6|5 8 6 6|
+-------+-------+

  (4,:4) Q;._3 ?.10#10        apply Q on subintervals of size 4
1.41421 1.60591

   4 (,:~@[ Q;._3 ]) ?.10#10           as a dyad
1.41421 1.60591

   4 ([: avg ,:~@[ Q;._3 ]) ?.10#10    average the result
1.51006

   RS=: ([: avg ,:~@[ Q;._3 ] )"0 1

   2 3 4 RS s
1 1.40931 1.72483
```

Thus RS estimates the R/S statistic on subintervals of size given as its left argument. We now create 1000 random numbers and show the plot of $\left(\ln(N), \ln(\overline{Q}_{t,N})\right)$ along with the line of least squares fit in Figure 3.5.1. Below, N gives the block size lengths.

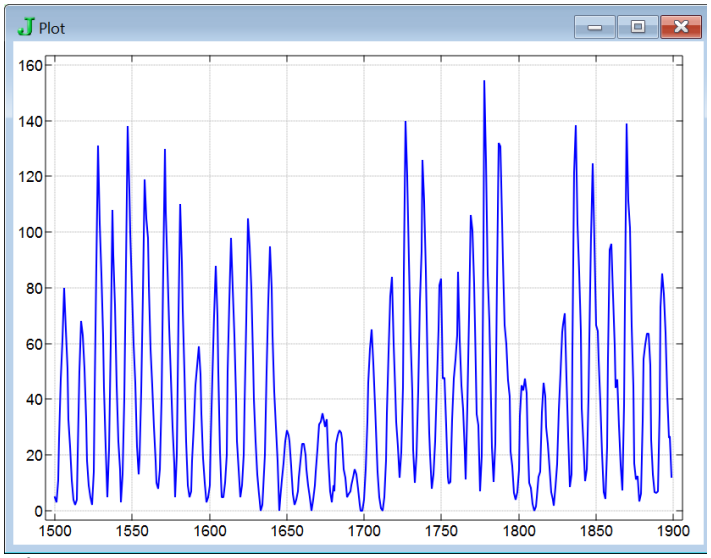


Figure 3.5.2 Sunspot Data

```
   setseed 7^5

   x=: randsn 10000

   N=: 2+i.<.-:#x

   rs=: N RS x

   ((^.rs)%.1,.^.N)     coefficients for the line of best fit
0.275168 0.487128
```

We see that the Hurst exponent is around 0.49 which is near the expected 0.5. Next we apply rescaled range analysis to sunspot data. Loading the script *ts_data.ijs* will load several time series data sets. This includes sunspots which gives the average number of sunspots observed for each of the 400 years from 1500 until 1899. The data is shown in Figure 3.5.2. Notice that the data seems quite periodic but there is no precise period or amplitude.

```
   load '~addons/graphics/fvj4/ts_data.ijs'

   $x=: sunspots                  sunspot data
398
```

```
   N=: 2+i.<. 0.5*#x              subinterval test lengths
   rs=: N RS x
   plot (^.N);^.rs
```

If we do a linear fit to the data we get an estimated Hurst Exponent near 0.79 which is indicative of considerable persistence. This suggests that there may be some hope of using weak underlying chaotic behavior for forecasting.

```
   ((^.rs)%.1,.^.N)
_0.530876 0.791816
```

The log-log plot and the line of best fit is shown in Figure 3.5.3.

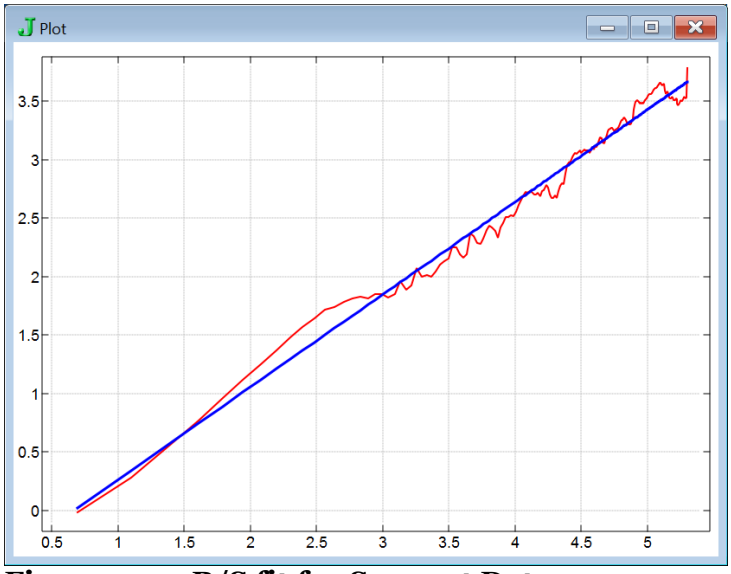


Figure 3.5.3 R/S fit for Sunspot Data

3.6 Autocorrelation Functions

Another technique for seeking evidence of long term memory in data is to look at autocorrelation coefficients. Informally speaking, a time series, x_t, for $t = 1,2,3,\ldots,T$ is called stationary if the statistical properties of the series are similar to those of any time shifted version of the series. Once long term trends have been removed from a time series, it may appear that the result is random noise. If the noise is normally distributed with mean 0 and variance 1, then the sample autocorrelations ought to be normally distributed with mean 0 and variance $1/T$. Thus we would expect approximately 95% of the autocorrelation coefficients to be in the interval $\pm 2/\sqrt{T}$. Thus, if the autocorrelation coefficients significantly deviate from that expectation, we suspect that the data is not pure noise and contains underlying correlations.

The sample autocovariance function is defined by $\hat{\gamma}(t) = \frac{1}{T}\sum_{i=1}^{T-t}(x_{t+i} - \bar{x})(x_i - \bar{x})$ where $\bar{x}$ is the sample average and we may think of *t* as the length of a delay. The sample autocorrelation function is defined by $\hat{\rho}(t) = \hat{\gamma}(t)/\hat{\gamma}(0)$. We can readily implement and experiment with these functions. Both the autocovariance function `acov` and the autocorrelation function `acor` take the delay as left argument and the data as right argument.

```
   avg=: +/ % #
   acov=: ([: +/ [ ((-@[ }. ]) * }.)(- avg)@] ) % #@]
   acor=: (acov % 0 acov ])"0 1
   2 acor 1 2 1 2 1 2
0.666667
```

We assume we have `randsn` and `randunif` from Section 3.3 and that plot and *ts_data.ijs* have been loaded as in the previous section. We will compute the autocorrelation coefficients for the steps of a random Gaussian walk and the sunspot data time series. Since these each have approximately 400 data points, we would expect approximately 95% of the coefficients to lie within the interval if the data is from a random walk. Since we look at 40 coefficients with delay greater than 0, we expect typically two or three of these coefficients might be out of range if the time series is truly random. Figure 3.6.1 shows

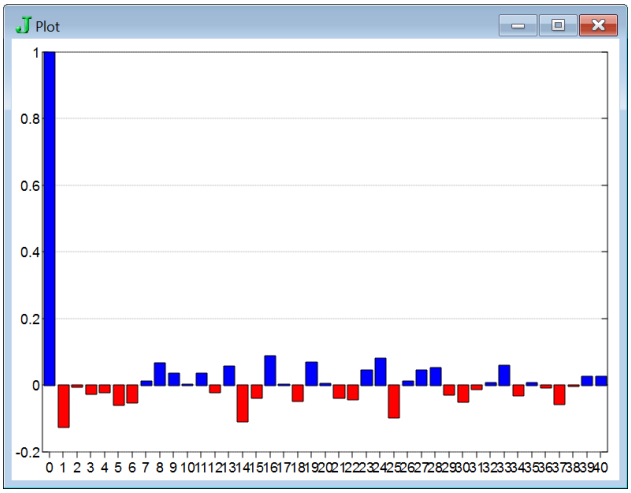


Figure 3.6.1 Autocorrelation of Gaussian Data

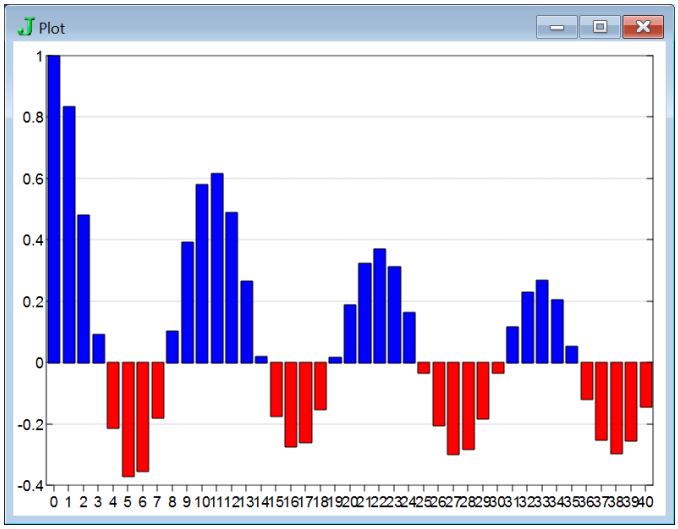


Figure 3.6.2 Autocorrelation for Sunspots

the autocorrelation coefficients for randomly distributed time series data while Figure 3.6.2 shows the autocorrelation coefficients for the sunspot data.

```
('bar;xlabel ',":i.41) plot (i.41) acor randsn 400
'bar' plot (i.41) acor sunspots
```

Notice that the autocorrelation coefficients of the random time series seem well bounded by ± 0.1 while the autocorrelation coefficients of the second differences of the sunspot time series clearly falls outside that range more frequently than expected. In fact, the first positive peak of the correlation coefficients is around the 11 years; given the approximately 11 year cycle of sunspots, this is consistent with expectations.

3.7 Experiment: Random Midpoint Displacement

In this section we look at creating fractal time series in an iterative manner that allows us to randomly produce time series with a variety of Hurst exponents. In fact, it is known that a Hurst exponent of *H* corresponds to a fractal dimension of 2-*H*. We will discuss fractal dimension in Section 4.6. We can view these experiments as giving us random curves with different fractal dimensions. The construction we do in this section will be useful in later chapters when we want to create plasma clouds and fractal mountains.

The basic idea is that we will take some series, interpolate to intersperse the data with midpoints and then we perturb the data by suitable random amounts. Following Peitgen, Jürgens, and Saupe (1992) we begin with a series that is 0 at one end and perturbed by a random normal amount with a certain standard deviation, $\sqrt{1-2^{2H-2}}$, at the other end. We then interpolate using midpoints and perturb each entry by a random normal amount with a standard deviation $1/2^{H}$ times the standard deviation at the previous level. We will perturb all the points while the reference we mentioned only perturbs the new midpoints. We repeat this process, at each stage interpolating to create new midpoints and then making random adjustments whose standard deviation is reduced at each iteration.

Our interpolation function works by making two copies of one-half times the data and then adds the behead to the curtail. Every other term gives the original data back and the ones in between are the averages. Our "original size" function, `osz`, computes $\sqrt{1-2^{2H-2}}$ while our "size" function, `sz`, computes $1/2^{(k+1)H}\sqrt{1-2^{2H-2}}$ where *k* denotes the number of levels of refinement (however, we have used some algebraic simplification in our implementation). Our functions `osz` and `sz` take the Hurst exponent as their left argument and the data as their right argument.

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

   interp 1 2 5                    interpolate some data
1 1.5 2 3.5 5

   osz=: %:@-.@(2&^)@+:@<:@[       get starting size standard deviation

   0.5 osz 1 1 1                   here the original size is 1/√2
0.707107

   0.1 osz 1 1 1
0.84429

   sz=: osz * %@+:@<:@#@] ^ [      get the size given the list

   0.5 sz 1 1                      here the size is 1/√4
0.5

   0.5 sz 1 1 1                    here the size is 1/√8
0.353553

   0.1 sz 1 1 1
0.734997
```

Next we implement the random addition by adding to any given data the random normal data of the desired size. The function `randsn` was defined in Section 3.3.

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

   0.5 randadd 0 1 2 3 4
_0.128619 1.30911 1.91166 2.86945 4.30876

   0.9 randadd 0 1 2 3 4           notice the smaller changes
_0.0838968 0.942347 2.01761 2.99631 4.0119
```

Lastly we iterate the process of random addition and interpolation upon the beginning sequence `0` and an original size random perturbation with the function `hwalk` defined below. The left argument of `hwalk` is the Hurst exponent while the right argument is the number of levels of refinement to be applied. Figure 3.7.1 shows this process with Hurst exponent 0.1 while Figure 3.7.2 shows this process with Hurst exponent 0.9.

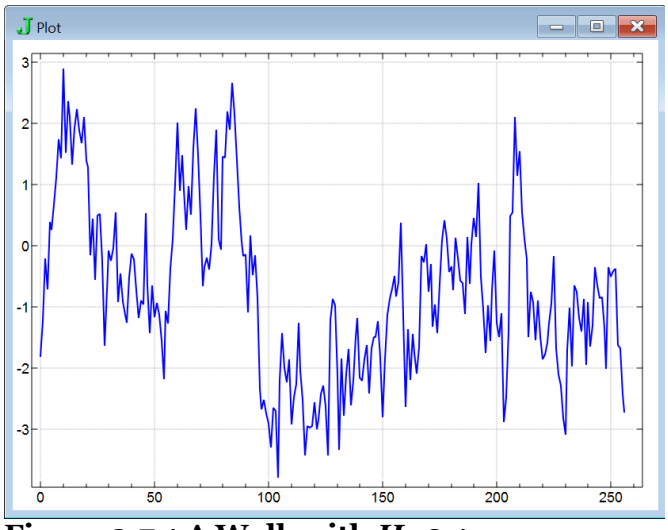


Figure 3.7.1 A Walk with *H*=0.1

Figure 3.7.2 A Walk with *H*=0.9

```
hwalk=: 4 : 'x([randadd interp@])^:(y) 0,(x osz 1)*randsn 1'
setseed 7^5                          as defined in Section 3.5
'pensize 2' plot 0.1 hwalk 8
'pensize 2' plot 0.9 hwalk 8
```

Create eleven such random walks corresponding to $H = 0, 0.1, 0.2, ..., 1.0$.

3.8 Experiment: Forecasting via Best Analogs

In this section we experiment with some forecasts of data based upon matching given patterns with the best analogs that can be made to historical reference patterns. This is sometimes called fractal forecasting because it offers very intriguing possibilities for taking advantage of hidden chaotic/fractal behavior that appears random. We will consider best analog forecasting of the Henon map and the sunspot data. The general strategy we use follows references Balkin (1996) and Casdagli (1992) but we will take the prediction based upon the single best fit analog, rather than using regression on serveral close fits.

We first define a function `dist` that can be used to find the distance between points. That is, it computes the square root of the sum of the squares of the differences between the points. We use the Henon map from Section 2.2 in order to generate our first set of data.

```
dist=: %:@(+/)@:*:@:-"1
1 2 3 dist 3 4 2                     distance between two points
3
1 2 3 dist 3 5 5,:3 4 2              distance between a point and a list of points
4.12311 3
```

The basic strategy is to take the historical data set and divide it up into blocks of fixed length *M* called *M*-histories: they provide a reference list of *M*-histories whose future behaviors are known. In practice, we take windows of size *M+1* and consider the first *M* points to be the "reference" pattern and the last point to be the next "result" in the sequence. Thus, below we take size *M+1* infixes of the Henon data time series where we are using $M = 2$.

```
hen=: 3 : '({:y),1+(0.3*{.y)+_1.4* *: {:y'
$hd=:{.|: 10}.hen^:(i. 510) 0 0       Henon data
500
$ref_res=:3 ]\ 450{.hd                reference and result lists combined
448 3
$ref=:}:"1 ref_res                    reference list
448 2
$res=:{:"1 ref_res                    next result list
448
_4{.ref                               last 4 pairs in the reference set
  1.04842 _0.669762
_0.669762  0.686513
 0.686513  0.139251
 0.139251   1.17881
_4{.res                               next terms in each sequence (the results)
0.686513 0.139251 1.17881 _0.903645
```

```
   $fhd=:450}.hd                    future Henon data (it is not in the reference set)
50

   ]X=: 2{.fhd                      a starting pattern for our predictions
0.210438 0.666909
```

Grade up gives the indices of a list in increasing order. Thus the first element of the grade up gives the index of the *M*-history closest to our data.

```
   /:13 2 53 1 77 99                grade up gives indices of increasing order
3 1 0 2 4 5

  {./:13 2 53 1 77 99               index of the smallest item
3

   3{13 2 53 1 77 99                the smallest item
1
```

We see below that the index of the closest pattern to `X` is 241 and when we compare that pattern to `X`, we see they are quite close. Looking at the data at index 241 in `res` is our prediction for the next point in the sequence.

```
   {.@/: ref dist X
241

   241{ref
0.209871 0.668865

   X
0.210438 0.666909

   241{res
0.436629
```

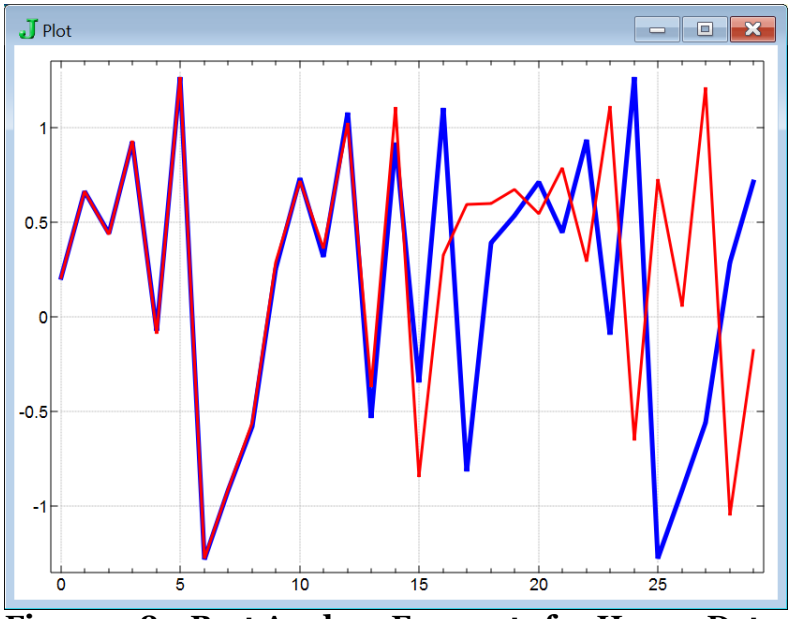


Figure 3.8.1 Best Analogs Forecasts for Henon Data

We define a function `fpred_step` which puts all these steps together. Note, it presumes that `ref` and `res` have been correctly defined. The result is the subsequent pattern using the prediction. This makes the function suited for iteration.

```
   fpred_step=:3 : 0
(}.y),({.@/:ref dist y){res
)

   fpred_step X
0.666909 0.436629

   fpred_step^:(i.4) X
0.210438   0.666909
0.666909   0.436629
0.436629   0.933757
0.933757 _0.0896736
```

The following records the actual 30 steps of the Henon map and computes a 30 step forecast.

```
      $a=: 30{.fhd
30
```

```
      $b=:{."1 fpred_step^:(i.30) X
30

   require 'plot'

   plot ((i.30) ; a ,: b)
```

The following plot driver commands show a plot of the actual Henon data in blue, and the forecasts using `fpred_step` in red. The result is shown in Figure 3.8.1.

```
   pd 'reset;'
   pd 'type line'
   pd 'pensize 5'
   pd 'itemcolor 0 0 255'
   pd  a
   pd 'itemcolor 255 0 0'
   pd 'pensize 3'
   pd  b
   pd 'show'
```

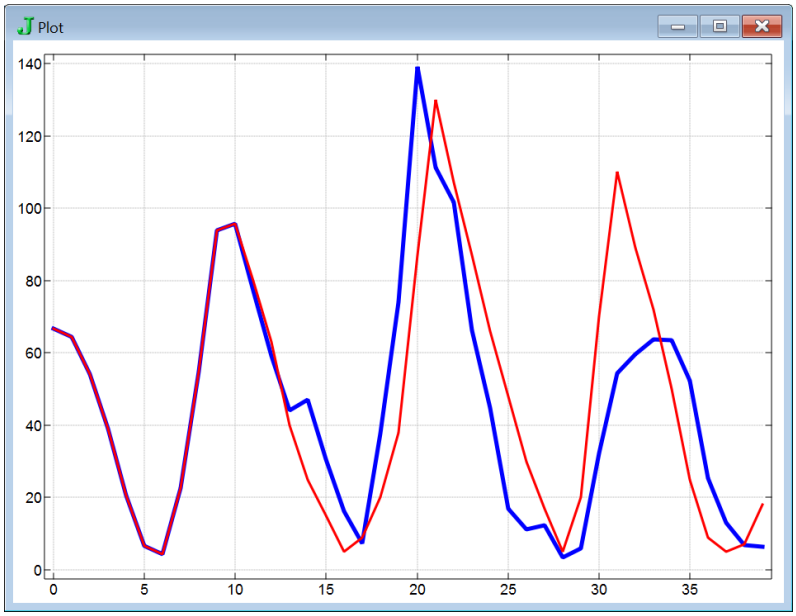


Figure 3.8.2 Best Analogs Forecasts for Sunspot Data

Next we repeat the same experiment for our `sunspots` data from *ts_data.ijs*. This time we use the nearest point to 11-histories for our estimates and attempt a 40 year forecast.

```
   load '~addons/graphics/fvj4/ts_data.ijs'

   $ref_res=:12 ]\ 350{.sunspots
339 12

   $ref=:}:"1 ref_res
339 11

   $res=:{:"1 ref_res
339

   fsd=:350}.sunspots

   ]X=: 11{.fsd
66.6 64.5 54.1 39 20.6 6.7 4.3 22.7 54.8 93.8 95.8

   $a=: 40{.fsd
40

   $b=:{."1 fpred_step^:(i.40) X
40

   plot ((i.40) ; a ,: b)
```

The result is shown in Figure 3.8.2.

3.9 Exercises

1. (a) Implement the sample average and sample standard deviation functions using explicit definitions.
(b) Implement the sample standard deviation tacitly without using cap.

2. Define a J function that gives the coefficients of the line of best linear fit to the *x*-*y* data given as its left and right arguments respectively.

3. Decide the result of the following uses of prefix and infix without using J. Then verify your answers.

(a) `<\i.5`	(c) `#\i.5`	(e) `+/\i.5`	(g) `(,+/%#)\i.5`
(b) `3 <\i.5`	(d) `3 #\i.5`	(f) `3 +/\i.5`	(h) `3(,+/%#)\i.5`

4. (a) Write a J function that computes the maximum value on prefixes of its argument.
(b) Write a J function that computes the maximum value on blocks of the right argument whose size is given by the left argument.
(c) Write a J function that is (a) as a monad and (b) as a dyad.

5. (a) Consider random selection of *n* items uniformly from the interval [0,1]. The mean is 0.5 and the standard deviation is $1/\sqrt{12}$. Run an experiment on 1000 random uniform points selected from the interval [0,1] and compute the sample mean and standard deviation of the result.
(b) Compute the sample average and standard deviation for the first thousand points in the Henon sequence defined in Section 2.2. Does the data from the Henon map appear to be uniform random?
(c) Compute the sample average and standard deviation for the first thousand points generated by the special logistic map given by $L(x) = 4x(1-x)$ beginning at 0.99. Does the data from the map appear to be uniform random?
(d) Compute the sample average and standard deviation for the first thousand points generated by the special logistic map given by $L(x) = 3.62x(1-x)$ beginning at 0.99. Does the data from the map appear to be uniform random?

6. (a) Compute the best least squares linear fit line to the Huron lake data defined in the *graphics/fvj4/ts_data.ijs* script.
(b) Plot the data and the line on the same plot.
(c) Plot the differences of the data from the value of the best fit line.

7. Create a random walk where at each step a move by `_1`, `0` or `1` occurs with equal likelihood. How does this compare to Figure 3.3.1?

8. Write a J function that has `randsn` from Section 3.3 as its monadic use and generates arrays of random normal numbers with mean and standard deviation specified by the left argument when used as a dyad.

9. Several time series may be shown on the same plot. Experiment with the following plot commands after defining `randsn` as in Section 3.3.

```
plot randsn 2 200
plot +/\"1 randsn 2 200
plot (1000+2*i.200);+/\"1 randsn 2 200
```

10. Standard normal Gaussian walks of length *T* are expected to wander between $\pm\sqrt{T}$. Create a plot of 1000 standard normal random Gaussian walks of length 200 on the same plot. Is the result consistent with the above fact? Hint: see the previous exercise.

11. (a) Run the plot package lab (from the studio-labs menu)
(b) Read the section in the on-line user manual on plot commands to see how to save **.pdf* and **.bmp* versions of plot images. Use some of those utilities to save a plot of a random standard normal Gaussian walk to a file.

12. Try to predict the result of the following commands assuming `avg` and `sd` are defined as in Section 3.1 and `randsn` is defined as in Section 3.3. Check your answer.

```
avg 1+randsn 1000
sd  1+randsn 1000
avg 3*randsn 1000
sd  3*randsn 1000
```

13. As in Section 3.3, create several random `_1 1` walks and then create several different `0 1` random walks. How are they different in appearance?

14. (a) Write a J function that implements random `_1 1` walks of length specified by the right argument.
(b) Write a J function that implements random uniform (from the interval [0,1]) walks of length specified by the right argument.
(c) Write a J function that implements random uniform from (b) as its monadic use while the dyadic use specifies the interval as its left argument.
(d) Write a J function that implements random standard normal walks of length specified by the right argument.

15. (a) Write a J function that uses `1 2 4 8 4 2 1%22` as the weights in a moving average function.
(b) Use the function from (a) on the Huron lake data huron from *~addons/graphics/fvj4/ts_data.ijs* to obtain a trend series; plot the trend and the data on the same plot.

16. Load the lake levels `huron` by loading the script *~addons/graphics/fvj4/ts_data.ijs*. Experiment with the appearance of the trend when Spencer averaging, as in Section 3.4 is applied repeatedly, as in `spencer^:3 huron` . Make a plot of the data, 3-fold, 10-fold and 50-fold repetitions of `spencer` on the same plot.

17. Predict the result of each of the following. Check your answers.

(a) `{. i.7`	(b) `}. i.7`	(c) `{: i.7`	(d) `}: i.7`
(e) `3 {. i.7`	(f) `3 }. i.7`	(g) `_3 {. i.7`	(h) `_3 }. i.7`

18. Predict the result of each of the following. Check your answers.

(a) `{. i. 7 7`	(b) `}. i. 7 7`	(c) `{: i. 7 7`	(d) `}: i. 7 7`
(e) `3 {. i. 7 7`	(f) `3 }. i. 7 7`	(g) `_3{. i. 7 7`	(h) `_3}. i. 7 7`
(i) `2 3{. i. 7 7`	(j) `2 3}. i. 7 7`	(k) `_2 3{.i.7 7`	(l) `_2 3}.i. 7 7`

19. Apply R/S analysis to the following time series. In each case use a suitable plot to determine a range of *N* values to use for linear fit to the log-log plot and estimate the Hurst exponent.
(a) Lake Huron data `huron` from *ts_data.ijs*
(b) Difference of Lake Huron data from its best linear fit.
(c) 400 points from the Henon map defined in Section 2.2.
(d) 400 points from the logistic map $L(x) = 4x(1-x)$
(e) 400 points from the logistic map $L(x) = 3.62x(1-x)$

20. Make an autocorrelation plot for the Huron data `huron` from *ts_data.ijs*.

21. Make an autocorrelation plot for the wheat price data `wheat` from *ts_data.ijs*.

22. Experiment with the following uses of cut 0, 3 and _3.

(a) `(4,:4) <;. 0 i.10`	(b) `(4,:3) <;. 0 i.10`	(c) `(4,:_4) <;. 0 i.10`
(d) `(4 ,: 4)<;._3 i.10`	(e) `(4 ,: 3)<;._3 i.10`	(f) `(4 ,:_4)<;._3 i.10`
(g) `(4 ,: 4)<;.3 i.10`	(h) `(4 ,: 3)<;.3 i.10`	(i) `(4 ,:_4)<;.3 i.10`

23. Experiment with the following other uses of cut _1, 1,_2 and 2.
Define `b=:1 0 0 1 0 0 0 0 1 0`
(a) `b<;.1 i.10` (b) `b<;._1 i.10`
(c) `b<;.2 i.10` (d) `b<;._2 i.10`

24. Estimate the Hurst exponent of the `sunspots` data from *ts_data.ijs* after applying the $f(x) = \log(1 + x)$ Box-Cox transformation to the original sunspot data. Note: Mandelbrot and Wallis (1969) obtained roughly 0.93.

25. Estimate the Hurst exponent of the random series created at the end of Section 3.7 with $H = 0.7$ using R/S log-log plots.

26. As in Exercise 3.10, plot multiple random walks with specified Hurst exponent. How does the result compare with the results of Exercise 3.10?
(a) $H = 0.1$ (b) $H = 0.9$

27. Create images of the fractal prediction technique used to make figures analogous to Figure 3.8.1 using the same Henon data but using reference windows of size (a) $M = 3$, (b) $M = 4$, (c) $M = 5$.

28. Apply best-analog forecasting to the following sets:
(a) a set of random standard normal data.
(b) the `wheat` data from *ts_data.ijs*.
(c) 400 points from the logistic map $L(x) = 4x(1 - x)$.
(d) 400 points from the logistic map $L(x) = 3.62x(1 - x)$.

29. Instructors may offer classes grading schemes where a certain number of quizzes are dropped. Write a J verb `quizdt` that takes a list (of quiz scores) as its right argument and results in the sum of the scores after the lowest `x` scores are dropped, where `x` is specified by the left argument. The monad should drop the single lowest quiz score from the total.

Chapter 4 Iterated Function Systems and Raster Fractals

In this chapter we will take a deeper look at iterated function systems, considering them in general, and implementing them probabilistically. We will implement the chaos game which both creates fractals in a simple way and is applicable to visually identifying nonrandom data. We will also discuss fractal dimension and explore different techniques for computing the dimension.

4.1 Agenda and the 3*x*+1 Function

In preparation for discussing probabilistic iterated function systems in the next section, we will consider a simple example of using agenda, `@.`, to select which function, from among various possibilities, to apply. It is also sensible to consider using control words for the same purpose. Agenda can be quite useful for defining functions whose definition breaks into cases based upon a simple choice of an index for the function to apply. We will illustrate with a function that has remarkable properties. It is a (slight variant of) a function that is often called the 3*x*+1 function. It takes a positive integer as argument and results in half the number if the integer is even and results in half one plus three times the number if the argument is odd. Thus, we will define the 3*x*+1 function as follows.

$$t(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ (3x+1)/2 & \text{if } x \text{ is odd} \end{cases}$$

First we will use a construction of the form `f0`f1@.test y` to implement and apply this function. This function results in `f0 y` if `test y` is `0` and results in `f1 y` if `test y` is `1`. In general, `g@.test y` applies the function in the gerund `g` whose index is the result of `test y`. We will use the remainder modulo 2 to specify the test for evenness or oddness.

```
   10 | 55 61 29          remainders after division by 10
5 1 9

   2 | 0 1 2 3 4          remainders modulo 2 give a test for oddness
0 1 0 1 0

   f0=: -:                half

   f1=: -:@(1 3&p.)       half the function 1+3x

   t=: f0`f1@.(2&|)       the 3x+1 function

   t 7
11

   t 14
7

   t^:(i.15) 7
7 11 17 26 13 20 10 5 8 4 2 1 2 1 2     notice the repetition
```

While we are for now interested in the 3*x*+1 function for purposes of illustrating the use of agenda, we can not resist a short aside to illustrate its intriguing behavior. First observe that the function values were drawn to the cycle `1 2` when applied to `7`. Further experimentation suggests that this always happens when starting with a positive integer. However, that fact has not been proven; see [Lagarias 1985]. The function `tpath` defined below finds the path to `1`. It uses an iterate to convergence function of the type `f^:test^:_` that iterates `f` while the `test` is true.

```
   (, t@{:) 7
7 11

   (, t@{:)^:(5) 7
7 11 17 26 13 20
```

```
   tpath=: (,t@{:)^:({:~:1:)^:_

   tpath 7
7 11 17 26 13 20 10 5 8 4 2 1

   tpath 27
27 41 62 31 47 71 107 161 242 121 182 91 137 206 103...
```

Figure 4.1.1 shows the result of plotting the sequence `tpath 27`. It takes 70 steps to reach `1` and in between it nearly reaches a few thousand.

How long is the path to `1` for `255875336134000063x` ? How large does the sequence get?

We close this section by discussing an alternate implementation that uses control words instead of agenda and the iteration.

```
   s=: 3 : 'if. 0=2|y do. y%2 else. (1+3 * y)%2 end.'

   spath=: 3 : 0
while. 1 ~: {: y do.
  y=.y,s {:y
  end.
)

   spath 7
7 11 17 26 13 20 10 5 8 4 2 1
```

Thus we see that `spath` gives the same result as `tpath`. Using whichever suits you is fine. However, we can test how much time and space each of these require. The result of `ts` below gives the time in seconds and the space in bytes required to evaluate the expression.

```
   ts=: 6!:2 , 7!:2@]

   ts 'tpath 27'
0.000226217 14976

   ts 'spath 27'
0.00056319 7680
```

We see that `tpath` is faster while `spath` uses less space. Usually such differences are not significant, but in some circumstances performance differences are worth investigating.

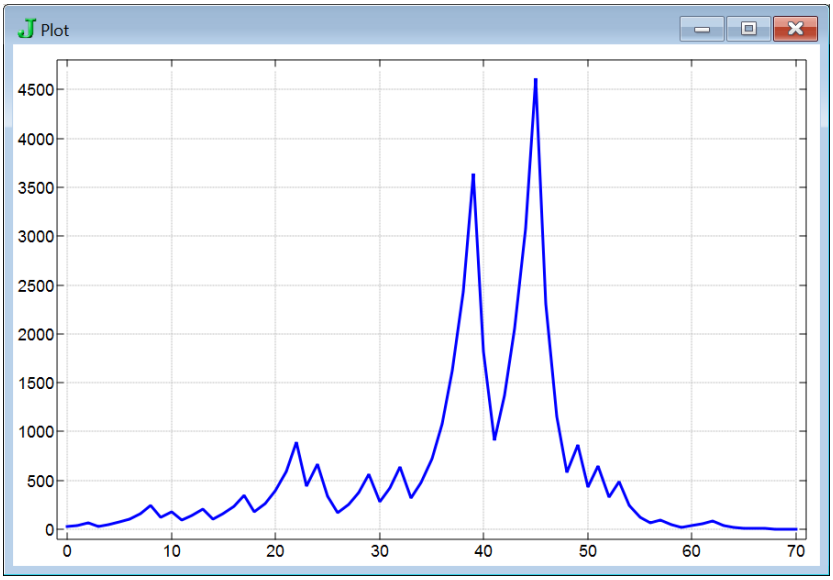


Figure 4.1.1 The 3*x*+1 Sequence on 27

4.2 Experiment: Probabilistic Iterated Function Systems

In Section 2.8 we saw that lists of functions, created using gerunds and called iterated function systems (IFS), could be used to create fractals. Applying the list of transformations to a polygon created a list of polygons, called the collage of the original polygon under the iterated function system. We then iterated the process of taking collages and obtained the fractal image. Remarkably, if we apply a randomly selected transformation, instead of applying all of the transformations, we also create the fractal image. In the next three sections we will investigate that style of constructing fractals. This is known as the probabilistic construction of the attractor of the iterated function system. For our first example, we define the matrices and transformations that we used in Section 2.8.

```
   ]m0=: (0.5*=i.2),0 0 1
0.5   0 0
  0 0.5 0
  0   0 1

   ]m1=: (0.5*=i.2),0.5 0 1
0.5   0 0
  0 0.5 0
0.5   0 1

   ]m2=: (0.5*=i.2),0 0.5 1
0.5   0 0
  0 0.5 0
  0 0.5 1

   mp=: +/ . *

   t0=: mp&m0

   t1=: mp&m1

   t2=: mp&m2

  ?@3: 1.111 2.345              random index from i.3 ; the argument is irrelevant
1

   ?@3: 1.111 2.345
1

   ?@3: 1.111 2.345
2
```

We put the transformations and random index selection together in a function that uses adgenda `@.` to apply one of the randomly chosen transformations. We denote the function by `rt` for random transformation. It selects one of the three transformations `t0`, `t1` or `t2` and applies it to the argument.

```
   rt=:t0`t1`t2@.(?@3:)          random transformation function

   rt 0.1 0.1 1                  the verb t1 was chosen at random
0.55 0.05 1

   rt 0.1 0.1 1                  the verb t2 was chosen at random
0.05 0.55 1

   rt^:(100+i.5) 0.1 0.1 1       five random iterates
0.116649 0.750047 1
0.0583244 0.875023 1
0.0291622 0.437512 1
 0.514581 0.218756 1
 0.257291 0.609378 1
```

Next we want to display the result of this transformation in a graphics window. We will do this by using some utilities for virtual raster array plotting that are defined by running the script *raster.ijs*.

```
   require '~addons/graphics/fvj4/raster.ijs'

   vwin 'fractal'                          open a blank raster window;
                                           default corners are 0 0 and 1 1
   vpixel rt^:(100+i.10000) 0.1 0.1 1      plot 10,000 points

   vshow ''                                show the virtual raster array in the window
```

```
   $VRA              the image is stored in the global array VRA
500 500

   +/,VRA            we can directly count the number of pixels that are lit
8289
```

Notice the image resembles a fuzzy version of the Sierpinski triangle. We can run 100,000 points and get a great plot. However, it is more fun to watch the image materialize; though on modern computers the images are created quickly. Here we define a function that plots the result of iterating `rt` on a seed value. The right argument specifies the number of points to plot before updating the screen while the left argument specifies how many times you want repeat that process. Loading *raster.ijs* defines the function `rtiter` as below, but you must define `rt` as above.

```
   rtiter=: 3 : 0            do not type in; the function is loaded by raster.ijs
100 rtiter y
:
k=.0                         number of times through the loop
vwin 'IFS'                   open a virtual raster window
seed=.rt^:(100) 0.1 0.3 1    a random starting point near the attractor
while. k<x do.
  d=.rt^:(i.y)seed           get random transformation data
  seed=.{:d                  update the seed
  vpixel d                   update the virtual raster array with the data
  vshow ''                   show the updated array
  k=.k+1                     update the counter
end.
)

   rtiter 1000               show 1,000 more points plotted repeatedly
```

Now suppose we redefine the matrix `m2` and then redefine the corresponding transformation `t2`.

```
   ]m2=: (0.5*=i.2),0.25 0.5 1
0.5    0 0
  0  0.5 0
0.25 0.5 1

   t2=: mp&m2

   rtiter 1000
```

Try it! You should think about the geometric effect of the new and old `m2` and how that affects the resulting image. What does the new image look like?

Next consider the transformations resulting from the following four matrices. The first three of these may be thought of as an elongation followed by a translation.

```
   e0            e1            e2
0.8   0 0     0.8   0 0     0.5   0 0
  0 0.2 0       0 0.2 0       0 0.2 0
0.2   0 1     0.2 0.8 1     0.2 0.4 1
```

The fourth matrix can be thought of as a vertical contraction, followed by a rotation of 90 degrees, followed by a translation. We express that using `rotm` from Section 2.6, although we could more easily enter the matrix directly.

```
   load 'trig'

   rotm=:(cos , sin , 0:),(-@sin , cos , 0:),: 0: , 0: , 1:
```

```
   ]e3=: (1 0.2 1 *=i.3)mp(rotm 1r2p1)mp(=i.2),0.2 0 1
   0 1 0
_0.2 0 0
 0.2 0 1
```

Now we define `t0`, `t1`, `t2` and `t3` suitably and update the definition of `rt` to account for the fact that there are now four transformations. The resulting image is shown in Figure 4.2.1. You can view the fractal "E" that appears there as made up of a vertical stroke of width 0.2 and height 1 on the left and three horizontal strokes of height 0.2 and width 0.8 or 0.5. Can you see which matrices correspond to which strokes?

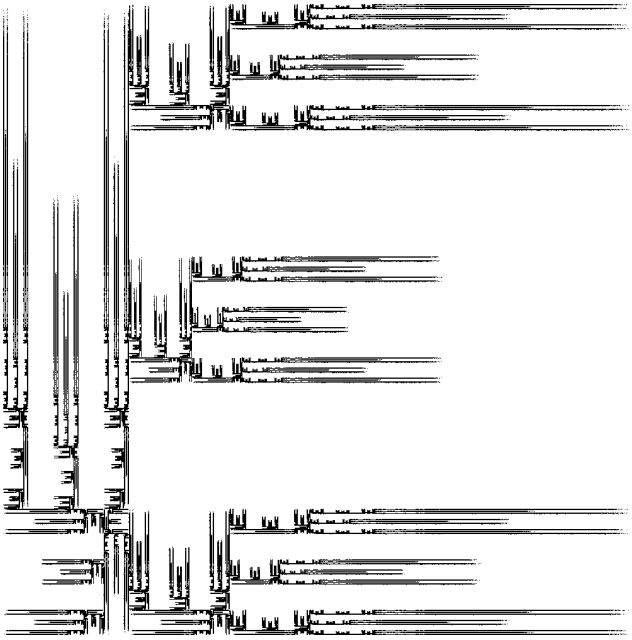


Figure 4.2.1 A Fractal E of E's

It is fun to create a fractal version of an entire word. Figure 4.2.2 shows a fractal version of the word "COKE" which is a variant of an image from [Barnsley, 1993] that appeared in [Reiter, B et al, 1998]. Can you see what 14 strokes create the image? Can you approximately duplicate that image? Can you create an image of that type that gives your name?

Next consider the four transformations associated with the matrices `fm0`, `fm1`, `fm2`, and `fm3` that appeared in the homework for Chapter 2 and which are repeated below. Create the function `rt` corresponding to those four transformations and run `rtiter`. Can you see what the image is supposed to look like? In Section 4.4 we will see how to balance the image by choosing from among the transformations with appropriate probabilities.

```
   fm0                fm1                fm2                fm3
   0    0 0          0.85 _0.04 0        0.2   0.23 0      _0.15   0.26 0
   0 0.16 0          0.04  0.85 0       _0.26  0.22 0       0.28   0.24 0
0.25    0 1        0.0375  0.17 1         0.2 0.1025 1     0.2875 _0.021 1
```

4.3 Remarks on Iterated Function Systems

Remember that an iterated function system is a list of functions. In general we use functions that are contractions to make interesting images. Indeed, there is a highly developed theory of iterated function systems [Barnsley, 1993] that describes what conditions on the functions guarantee a unique attractor (the resulting image). Furthermore, any image can be well

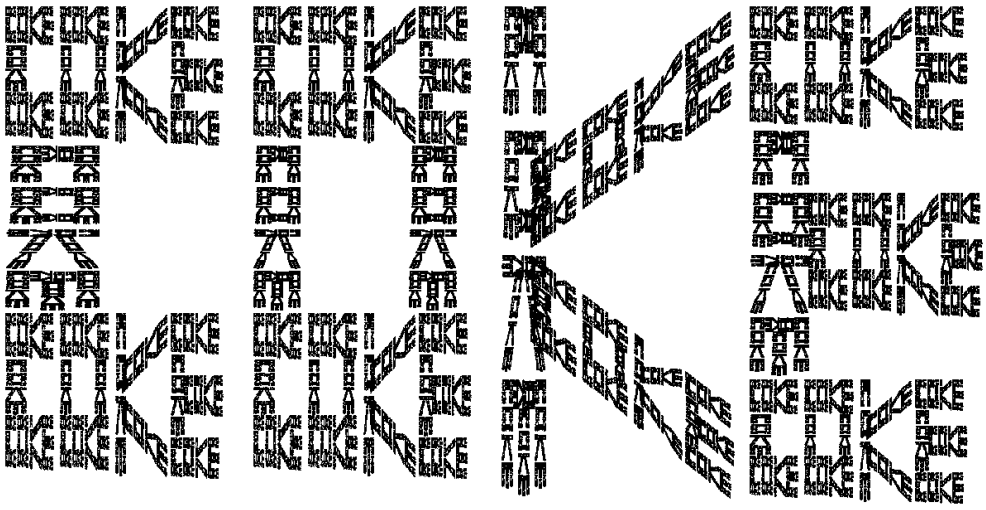


Figure 4.2.2 A Fractal Coke of Coke's

approximated by such an iterated function. The resulting iterated function system typically offers a significantly more compact description of the image than direct raster array storage of the image [Barnsley and Hurd 1993]. Hence, iterated function systems have attracted considerable attention as a method for compressing data, especially photographic images. Of course, such images contain more color information than the black and white images that we are studying at the moment, but the basic principles remain the same.

In Chapter 2 we saw that we could create fractal images by taking repeated collages of an iterated function system. So long as sufficiently many iterations were used, the resulting image did not seem to depend upon the shape of the original polygon; indeed, it could be as small as a pixel. In the previous section, we saw that we could randomly select a function from an iterated function system and by marking all the positions visited, create an image of the same attractor.

Consider the following remarks about the probabilistic approach to locating the attractor. First, before we plotted any points in our probabilistic method, we applied `rt` one hundred times to obtain our seed value (look back at the code for `rtiter` in Section 4.2 if you had not studied that before). Hence, the point we found after the 100 steps was a point that would have been on the hundredth iteration of the collage method. With the collage method we only applied up to seven iterations. Thus, up to our computational precision, the seed is on the attractor much more accurately than some points shown by the collage method. It is true that if the transformations contract very slowly, the points may not be accurate, but recall that in our first example, which gave the Sierpinski triangle, each transformation was a contraction by half and hence 10 steps would have sufficed to reach the attractor within a pixel on a 1000 by 1000 plot. If the contraction factor is near one, convergence is slow, but we can also choose to iterate more than a hundred times to create the seed when necessary.

Now that we see that the seed value is likely to be on the attractor, the question arises as to why we expect to see all the points on the attractor with the probabilistic method. Consider some point that was on the 5th iteration of the collage to be our target point. We need to see why we will eventually visit that point with the probabilistic method. Recall that even when we started with a single pixel, the process of taking collages led to an image of the attractor. Thus, we can consider our target point to be on the 5th iterate of our collage making process on our seed (or any other point). Thus, if we happened to pick the right sequence of 5 transformations, we would end up plotting the target point. Regardless of where we are in our probabilistic iteration method, the same facts are true: we are at (near) a point on the attractor and there is a sequence of 5 transformations that would take us to the target point. Thus, if we keep iterating the process, eventually we will happen to pick that sequence of five points and mark the target point. Is it reasonable to expect this to converge in a reasonable amount of time? First notice that if there are just three transformations, then there are just 243=3^5 sequences of five choices of the transformations; thus we would expect to mark all the pixels quickly.

However, our suggestion of 5 steps sufficing is an exaggeration. Even for the Sierpinski triangle where contractions are by half, getting to within one pixel's resolution on a 1000 by 1000 image might be expected to take 10 steps. Still, 59049 = 3^10 so this would be a very tractable way to create that image. However, one might use a very large number of transformations or use contraction factors that are near one. In that case convergence can be slow. Notice the E of E's in Figure 4.4.1 is not crisp, in part because many of those transformations had one coordinate in which they were contractions by a factor of 0.6 or 0.8 or worse: one was not a contraction. Thus, it can be difficult under the worse circumstances to get rapid convergence of the probabilistic method. However, in practice it tends to work reasonably well; if one uses the techniques of the next section where we bias the choice of the transformations then the technique tends to work very well indeed.

4.4 Weighted Selection of Random Transformations

Figure 4.4.1 An IFS Fern

At the end of Section 4.2 we considered the attractor based upon the transformations associated with the four matrices: `fm0`, `fm1`, `fm2` and `fm3`. It turns out that we will get a much better image if we bias our choice of transformations roughly by the ratios of the absolute value of the determinants of the associated matrices. We assume those matrices have been defined.

The determinant is given by `-/ . *` and it is rank 2; that is, it applies to matrices. We can not actually take the probability of any transformation to be zero which is why we make sure every entry is at least slightly positive.

```
   det=: -/ . *

   det fm0,fm1,fm2,:fm3
0 0.7241 0.1038 _0.1088

   ]w=:0.001 >. |det fm0,fm1,fm2,:fm3
0.001 0.7241 0.1038 0.1088
```

Next we divide by the sum, resulting in probabilities weighted by the modified determinants. Then we produce the cumulative weights based on those ratios which are convenient for selecting random indices with the given probabilities.

```
   (%+/) w
0.00106644 0.772209 0.110696 0.116029

   ]cw=: +/\(%+/)w
0.00106644 0.773275 0.883971 1
```

```
  ?@0: 5                                  random scalar
0.622471

   wri=: +/@(cw&<)@?@0:

   wri"0 i.10                             weighted random indices
1 3 1 1 1 1 1 1 3 1

   mp=: +/ . *
   t0=: mp&fm0                            update the transformations
   t1=: mp&fm1
   t2=: mp&fm2
   t3=: mp&fm3

   rt=: t0`t1`t2`t3@.wri                  update rt
```

Now running `rtiter` from Section 4.2 by loading *raster.ijs* will result in the image shown in Figure 4.4.1. This is a variant of the Barnsley fern [Barnsley, 1993]. We see that the image is well balanced.

When the number of transformations is small, the technique described above is convenient. However, if the number of transformation is large, it may be easier to refer directly to the matrices, rather than the transformations. Thus, we next consider the dyad `rmt` that takes a list of matrices as it argument and applies the transformation associated with a random matrix to a point. To be efficient, we want to compute the probabilities only once, and use `withw` to augment the matrices, as a boxed list, with their probabilities.

```
   withw=: 3 : 'y;+/\(%+/)0.001 >.|det y'
   rmt=: 4 : 'y mp (+/(>{:x)<?0){>{.x'
   rt=: (withw fm0,fm1,fm2,:fm3)&rmt
   rtiter 1000
```

This approach avoids retyping the list of matrix names, but runs slightly slower than the transformation based approach.

4.5 Experiment: The Chaos Game

The chaos game is a simple experiment that yields a way of producing probabilistic fractals. While these are in some sense very special fractals compared to those produced by iterated function systems, we will see they also offer a dramatic visual technique for observing non-randomness in some data.

The basic idea of the chaos game is that we begin with a set of vertices and an initial position. We move from our current position halfway toward one of the vertices. The vertex is chosen at random. We then repeat the process, each time marking the position we visited on our image. We begin with an illustration of this process on an equilateral triangle. We assume *~addons/graphics/fvj4/raster.ijs* has been loaded.

```
   ]T=: |: 2 1 o./ _1r6p1+2r3p1*i.3     an equilateral triangle
   0.866025 _0.5
6.12303e_17    1
  _0.866025 _0.5
   mid=: -:@+
   1 1 mid 2 5                          compute midpoints
1.5 3
   rv=: {&T@?@((#T)"_)                  get a random vertex of T
   rv ''                                the middle vertex of T
6.12303e_17 1
   cg=: mid rv                          step of the chaos game
   cg 0.8 0.9                           move toward a random vertex
0.4 0.95
   cg 0.8 0.9                           move toward a random vertex
_0.0330127 0.2
```

A slight variation on `rtiter` from Section 4.2 displays the chaos game. The result is shown in Figure 4.5.1.

```
   cgiter=: 3 : 0
1000 cgiter y
:
k=.0
_1 _1 1 1 vwin 'Chaos Game'
seed=.cg^:(100) 0.1 0.3
while. k<x do.
  seed=.{: d=.cg^:(i.y)seed
  vpixel d
  vshow ''
  k=.k+1
end.
)
```

```
   cgiter 1000        run it
```

Now run the process on a regular pentagon. Be sure to update the definition of `rv`. Then run the process on a square. Our understanding of the remainder of this experiment depends upon that experience.

If you ran the experiment on the square, you should have gotten a boring image. We can take advantage of that in order to test for randomness. The basic idea is that if we have data, we can decide which vertex to move toward using the data rather than deciding randomly. Then, if the resulting image differs from the uniform filled square, we know the data was not random. Indeed, with some effort, one can even figure out what sequences are being forbidden by the process. See [Jeffery, 1992] for further discussion of this technique.

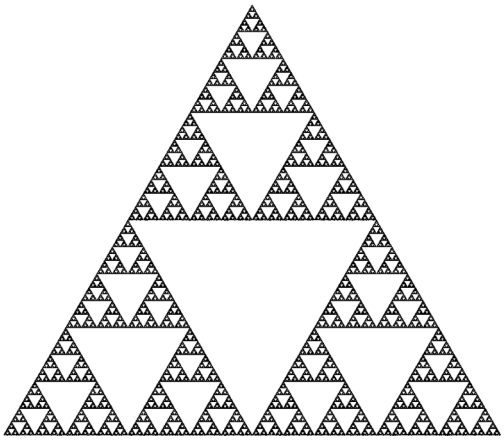


Figure 4.5.1 The Chaos Game on a Triangle

A deterministic version of the chaos game, which we will play on the square, can be implemented as follows. Note that `#:` is ***antibase two*** which gives a convenient way to define the square. The adverb `f.` is ***fix*** which replaces the names in the definition of `dcg` by their definitions.

```
   ]T=: #:0 1 3 2                    unit square
0 0
0 1
1 1
1 0
   gv=: {&T                          get vertices
   dcg=: mid/\.@:(gv@[ , ]) f.       deterministic chaos game
```

There we used ***suffix*** insert for efficiency, as we will explain. Suffix scan can do the same computations as prefix scan with reversals, although since we are doing "random" movements, the order will not matter and hence we did not use reversal in our definition of `dcg`. We illustrate the suffix and prefix scan and describe testing the performance of our constructions before automating the application of `dcg`.

```
   <\i.5                             prefixes
+-+---+-----+-------+---------+
|0|0 1|0 1 2|0 1 2 3|0 1 2 3 4|
+-+---+-----+-------+---------+
   <\.i.5                            suffixes
+---------+-------+-----+---+-+
|0 1 2 3 4|1 2 3 4|2 3 4|3 4|4|
+---------+-------+-----+---+-+
   mid/\ i.5                         midpoint prefix scan
0 0.5 0.75 0.875 0.9375
   |. mid /@:|.\. |. i.5             same using suffix
0 0.5 0.75 0.875 0.9375
   mid/\. i.5                        just using suffix
0.9375 1.875 2.75 3.5 4
```

The efficiency issue is rooted in the fact that given the order of evaluation in J, prefix inserts must fully compute each element of the result while the suffix inserts can take advantage of the insert computed to the right. We use `6!:2` and `7!:2` to measure the time in seconds and space in bytes required to compute a prefix and suffix scan. This illustrates the efficiency of the suffix scan.

```
   ts=: 6!:2 , 7!:2@]        time and space utility

   ts 'mid/\ i.5000'         prefix scan takes about 4.2 seconds
4.55721 567616

   ts 'mid/\. i.5000'        suffix scan takes about 0.0026 seconds
0.00183454 846592
```

Now we turn back to discussing our deterministic chaos game function `dcg` that was implemented with suffix scan above. Notice when we apply `dcg` below, the last point is the right argument and the three points above that are successive steps halfway toward vertex `0` and the top two points are successive steps halfway toward vertex `1`.

```
   1 1 0 0 0 dcg 0.5 0.5
0.015625 0.765625
 0.03125  0.53125
  0.0625   0.0625
   0.125    0.125          (2) moved halfway toward origin again
    0.25     0.25          (1) moved halfway toward origin
     0.5      0.5          (0) initial point

   detcg=: 3 : 0
vwin 'Deterministic Chaos Game'
seed=.{.(10#0) dcg 0.1 0.3
d=.y dcg seed
vpixel d
vshow ''
)
```

Next we use some DNA data to determine the direction of the motions. The list `Y` gives a list of 4 directions based on the DNA data. Running the deterministic chaos game on that data results in the image in Figure 4.5.2. It is clear that the DNA data is not random.

```
   open '~addons/graphics/fvj4/dna_y54g9.ijs'

   load '~addons/graphics/fvj4/dna_y54g9.ijs'

   $X=:(Y54G9 e. 'acgt')#Y54G9
292967

   30{.X
tggaataaacgttcgatcaacgtatagaaa

   Y=:'acgt' i. X

   detcg Y
```

What happens if you use `Y=:?100000#4` for your data? How about using `Y=:?100000#3` for your data?

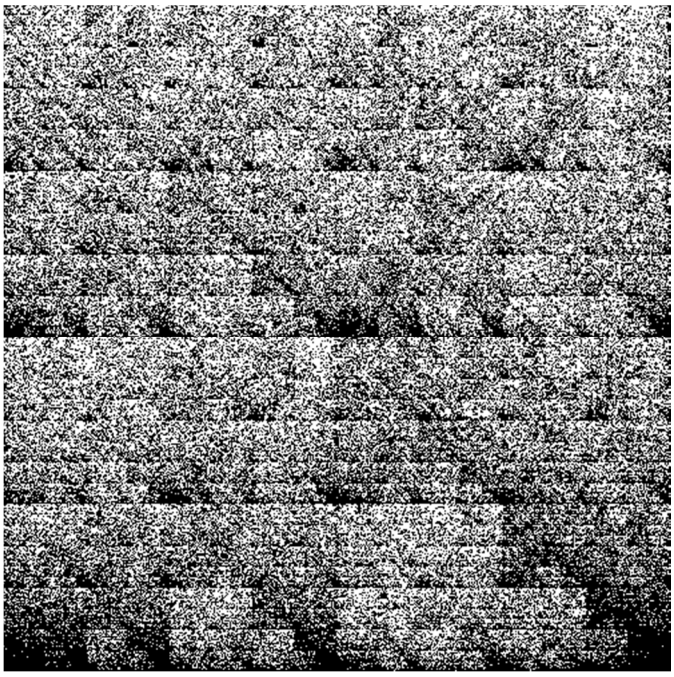


Figure 4.5.2 The Chaos Game Using DNA Data for Directions

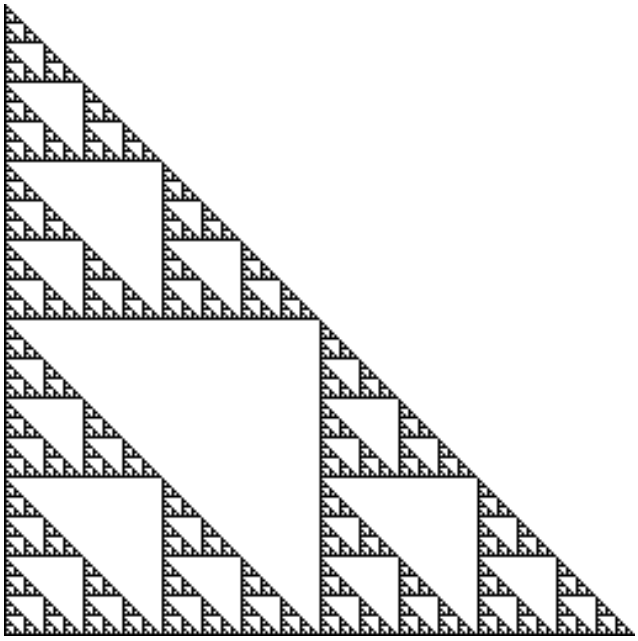


Figure 4.6.1 Skew Sierpinski Triangle

4.6 Fractal Dimension

In general we are content to leave the definition of "fractal" informal. Nonetheless, it is useful to be able to quantify fractal dimension in many circumstances. In practice this may be somewhat unreliable, yet it often seems to match our qualitative expectations and hence appears to be a valuable technique. In particular, we can define the fractal (box counting) dimension by saying it is the exponent that describes how quickly the number of boxes needed to cover our fractal object grows as the size of the boxes diminishes. In particular, if $N(\varepsilon)$ denotes the number of boxes of side size ε required to cover an object, then the fractal dimension of the object is defined to be $d = \lim_{\varepsilon \to 0^+} \frac{\ln(N(\varepsilon))}{\ln(1/\varepsilon)}$, provided the limit exists. Note this corresponds to saying that the number of boxes required grows roughly as $N(\varepsilon) \approx (1/\varepsilon)^d$.

Consider our example of the skew Sierpinski triangle. Figure 4.6.1 reminds us of its structure that we imagine sitting in a unit square. If we use boxes of side size 1/2, we need 3 to cover the set. If we use boxes of size 1/4, we need just 9. Table 4.6.I records those facts along with the generalization. We see that it appears, at least for sizes which are powers of 1/2, that we have $N((1/2)^n) = 3^n$ and hence the fractal dimension (if it exists) is $d = \lim \ln(3^n)/\ln(2^n) = \ln(3)/\ln(2) \approx 1.58496$.

ε	$N(\varepsilon)$
$(1/2)$	3
$(1/2)^2 = 1/4$	$3^2 = 9$
$(1/2)^3 = 1/8$	$3^3 = 27$
⋮	⋮
$(1/2)^n$	3^n

Table 4.6.I Number of Boxes to cover the Sierpinski Triangle

As a second illustration, we consider the Sierpinski carpet shown in Figure 1.10.2. A table of covering values is shown in Table 4.6.II. We have $N((1/3^n) = 8^n$ and hence the fractal dimension is $d = \ln(8)/\ln(3) \approx 1.89279$. Notice that this is consistent with our sense that the Sierpinski carpet is denser than the Sierpinski triangle.

As a third example, consider the fractal curve given by the Koch snowflake. That is, consider the edge of Figure 2.4.2. At each step in the construction, we replace each segment by 4 segments that are one third as long. Thus, we get the facts

ε	$N(\varepsilon)$
$(1/3)$	8
$(1/3)^2 = 1/9$	$8^2 = 64$
$(1/3)^3 = 1/27$	$8^3 = 512$
$\vdots$	$\vdots$
$(1/3)^n$	8^n

Table 4.6.II Number of Boxes to cover the Sierpinski Carpet

ε	$N(\varepsilon)$
$2(1/3)^0 = 2$	3
$2(1/3)^1 = 2/3$	$3(4^1) = 12$
$2(1/3)^2 = 2/9$	$3(4^2) = 48$
$\vdots$	$\vdots$
$2(1/3)^n$	$3(4^n)$

Table 4.6.III Number of Boxes to cover the Koch Snowflake

given in Table 4.6.III. We have $N(2(1/3^n) = 3(4^n)$ and by taking the appropriate limit, the fractal dimension is $d = \ln(4)/\ln(3) \approx 1.26186$.

In general, finding the rate at which the number of boxes in a covering grows is difficult. In the next section we will avoid that problem by running experiments. However, there is also a nice theory that can be applied to find the fractal dimension of the attractor arising from an iterated function system made up of non-overlapping (or just touching) similitudes. A similitude is an affine map that when applied to a shape results in a similar shape. So any rescaling is the same in all directions. Suppose we have a non-overlapping iterated function system made up of *k* similitudes with contraction factors $s_1, s_2, ..., s_k$. Then the fractal dimension, *d*, of the attractor of the iterated function system satisfies the equation $s_1^d + s_2^d + ... + s_k^d = 1$. For example, the iterated function system giving the Sierpinski triangle contains three contractions by half, thus the fractal dimension satisfies $(1/2)^d + (1/2)^d + (1/2)^d = 1$ and hence $d = \ln(3)/\ln(2)$ as we saw above.

Consider the iterated function system associated with the image shown in Figure 4.6.2. It is created using 7 non-overlapping affine maps. One of those maps (associated with the center) has a contraction factor of $1/3$ and 6 have contraction factors of $1/4$. Thus the fractal dimension satisfies $(1/3)^d + 6(1/4)^d = 1$. That does not have a closed form solution, but we can find a numeric solution fairly easily by trial and error.

```
   sop=: 1r3&^ + 6 * 0.25&^

   sop 1 1.5 2
1.83333 0.94245 0.486111

   sop 1.45536
1
```

Thus we see that the fractal dimension is near 1.45.

4.7 Fractal Dimension via Raster Box Counting

Given an image, we may not be able to theoretically find the number of boxes covering the fractal, but we can count the number of raster boxes of each size that contain part of the image. For a first example, consider a tiny version of the Sierpinski triangle that was an alternate experiment in Section 4.2. We then illustrate using cut negative three `;._3` to create non-overlapping 2 by 2 tesselations of the array `b`. These tesselations are a two dimensional analog of the cut construction that we saw in Section 3.5.

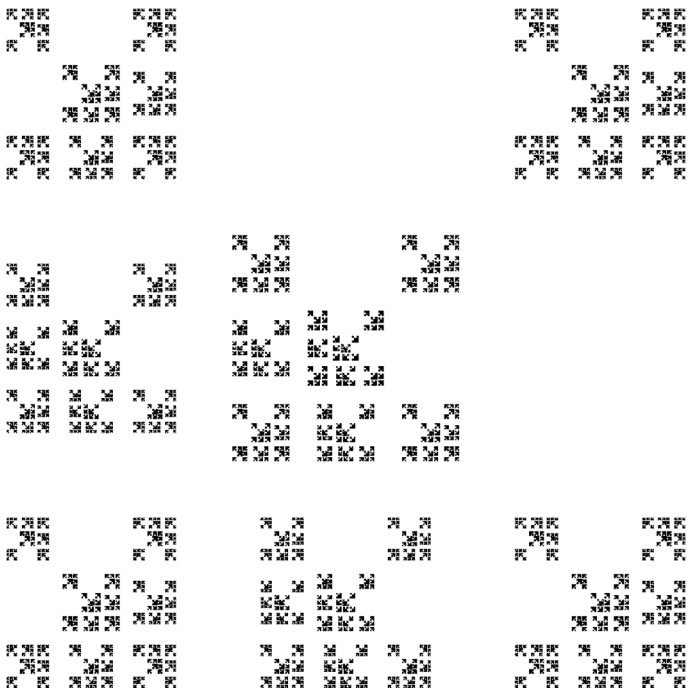


Figure 4.6.2 IFS of Similitudes

```
   m0=: (0.5*=i.2),0 0 1
   m1=: (0.5*=i.2),0.5 0   1
   m2=: (0.5*=i.2),0.25 0.5 1
   mp=: +/ . *
   t0=: mp&m0
   t1=: mp&m1
   t2=: mp&m2
   rt=:t0`t1`t2@.(?@3:)
   require '~addons/graphics/fvj4/raster.ijs'
   VRAWH=:13 13
   rtiter 100
100
   ]b=:|.VRA
0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 0 0
0 0 0 0 1 1 0 1 1 0 0 0 0
0 0 0 1 1 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 1 1 0 0 0
0 0 1 1 0 0 0 0 0 1 1 0 0
0 0 1 1 1 0 0 0 1 1 1 0 0
0 1 1 1 1 0 0 0 1 1 1 1 0
0 1 1 0 1 1 0 1 1 0 1 1 0
1 1 1 1 1 1 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
   (2 2,:2 2) <;._3 b
+---+---+---+---+---+---+
|0 0|0 0|0 0|1 0|0 0|0 0|
|0 0|0 0|0 1|1 1|0 0|0 0|
+---+---+---+---+---+---+
|0 0|0 0|0 1|1 1|0 0|0 0|
|0 0|0 0|1 1|1 1|1 0|0 0|
+---+---+---+---+---+---+
|0 0|0 0|1 1|0 1|1 0|0 0|
|0 0|0 1|1 1|1 1|1 1|0 0|
+---+---+---+---+---+---+
|0 0|0 1|1 1|1 1|1 1|0 0|
|0 0|1 1|0 0|0 0|0 1|1 0|
+---+---+---+---+---+---+
|0 0|1 1|1 0|0 0|1 1|1 0|
|0 1|1 1|1 0|0 0|1 1|1 1|
+---+---+---+---+---+---+
|0 1|1 0|1 1|0 1|1 0|1 1|
|1 1|1 1|1 1|0 1|1 1|1 1|
+---+---+---+---+---+---+
   1 e. 1 2 3 4
```

test if one is ***in*** (a ***member*** of) the list

```
1
```

```
   1 e. 0 2 3 4
0
```

```
   onein=: 1 e. ,
```
test if one is in the array

```
   (2 2,:2 2) onein;._3 b
```
apply `onein` to the tesselation;

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

```
   N=: [: +/@, (,:~@,~)@[ onein;._3 ]
```

```
   2 N b
25
```

```
   eps=: % #
```
epsilon

```
   2 eps b
0.153846
```

```
   fd=: (N %&^. %@eps)"0 2
```
estimate based upon this tesselation

```
   2 fd b
1.71967
```

Next we consider a larger version of the Sierpinski triangle.

```
   VRAWH=:1000 1000
```

```
   rtiter 10000
100
```

```
   $b=:|.VRA
1000 1000
```

```
   1 2 3 4 5 6 7 8 9 10 fd b
1.64822 1.65963 1.66552 1.67062 1.66397 1.67517 1.67752 1.68224 1.68849
1.67705
```

The fractal dimension is computed to be near 1.68 which is not too far from the theoretic value near 1.58. Larger versions of the image give slightly more accurate estimates.

As a second example, consider the fern shown in Figure 4.4.1. If `b` denotes the 1000 by 1000 binary matrix giving that image, we compute the following estimates of fractal dimension.

```
   1 2 3 4 5 6 7 8 9 10 fd b
1.73172 1.72922 1.72449 1.71946 1.71589 1.71194 1.7081 1.70432 1.70172
1.69914
```

It appears that the fractal dimension of that image is near 1.7.

4.8 Exercises

1. Implement the following functions using agenda to handle the cases. The domain is real numbers except (a) has integer domain.

(a) $f(x) = \begin{cases} x^2 & if\ x\ is\ even \\ 1-5x & if\ x\ is\ odd \end{cases}$ (b) $g(x) = \begin{cases} x^2 & if\ x \le 3 \\ 1-5x & if\ 3 < x \end{cases}$ (c) $g(x) = \begin{cases} x^2 & if\ x \le 3 \\ 1+x^2 & if\ 3 < x \le 7 \\ 2+x^2 & if\ 7 < x \end{cases}$

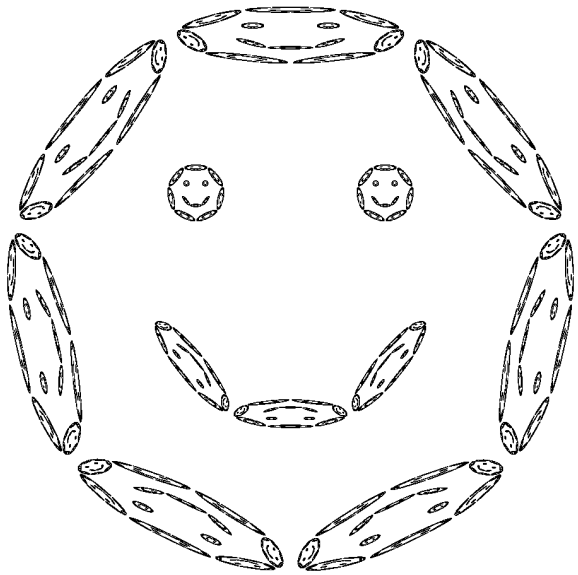


Figure 4.8.1 Smile of Smiles

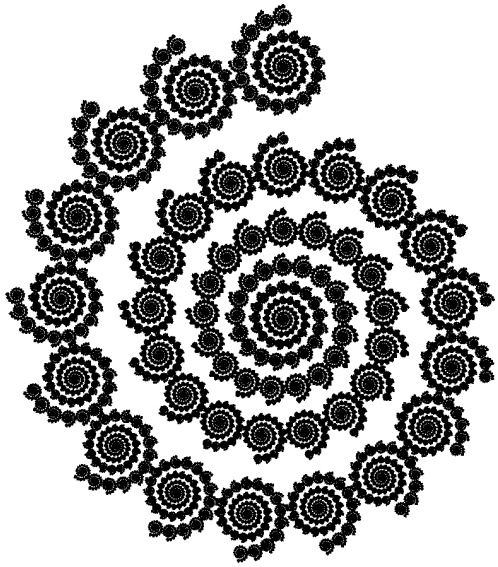


Figure 4.8.2 A Spiral of Spirals

2. Implement the functions in the previous exercise using explicit control words.

3. Consider the 3*x*+1 function *t*(*x*) from Section 4.1.
(a) Plot the path to 1 for each of the integers from 100 and 109.
(b) Implement a function `tn` which gives the number of elements in the path to 1 for a given number. Plot `tn` for all the integers from 1 to 200.

4. The classic 3*x*+1 is defined as follows: $h(x) = \begin{cases} x/2 & if\ x\ is\ even \\ 3x+1 & if\ x\ is\ odd \end{cases}$.
Implement that function and create an analog of Figure 4.1.1 for the path to 1 generated by *h*(*x*) on 27.

5. Load the script *~addons/graphics/fvj4/smiles.ijs*. It defines 12 matrices and transformations. Duplicate the smiles of smiles image in Figure 4.8.1 by defining a random transformation function `rt` in the following ways.
(a) Select the random transformations with equal probability.
(b) Find weights to bias the selection so that the image is balanced.

6. A spiral of spirals as in Figure 4.8.2 can be created using the transformations associated with `m0` and `m1` .

```
   m0                          m1
0.2  0    0                   0.904498 0.350404  0
0    0.2  0                  _0.350404 0.904498  0
0.4  0.8  1                   0.222953 _0.12745   1
```

(a) Create an image of the iterated function system selecting those transformations with equal probability.
(b) Find weights to bias the selection so that the image is balanced.

7. Identify transformations needed to approximately replicate the spiral of smiles image shown in Figure 4.8.3. Use the transformations from the previous exercise and the smile transformations in Exercise 5 to build the ones you need.

8. Identify transformations that allow you to approximately duplicate Figure 4.2.2.

9. We are not restricted using affine maps in iterated function systems. Figure 4.10.3 illustrates a fractal version of the word CHAOS from [Reiter, B et al, 1998] which uses curved strokes for parts of some of the letters.
(a) Approximately duplicate Figure 4.8.4 using transformations of your own design.
(b) Create a curved stroke fractal version of the word "WORD".

10. Randomly chosen iterated function systems do not have the same appearance as ones that use hand crafted affine transformations. Figure 4.8.5 shows a fractal resulting from a randomly selected set of six affine

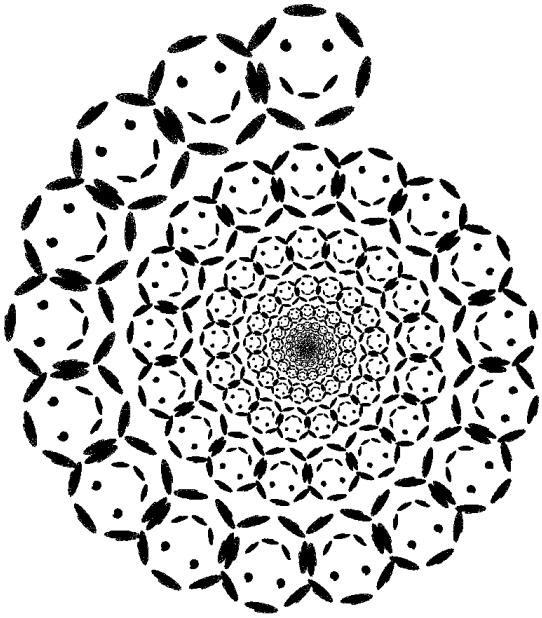


Figure 4.8.3 A Spiral of Smiles of Same

transformations. The six coefficients of the matrix in the left two columns were selected at random from between -1 and 1. Such a matrix was acceptable if it mapped the unit square back into the unit square. Six acceptable matrices were used to create the random transformation function `rt`.
(a) Implement that strategy for creating random fractals. Find several different images that way.
(b) Use box counting to estimate the fractal dimension of the fractals you produced in (a).

11. In Section 4.4 we created a weighted random index function which was used to define the function `rt` based upon transformation and also an alternate approach that used matrices.
(a) Write a tacit version of the function `rmt` from the matrix approach.
(b) Use the timing facilities (`6!:2`) to order the time required by (i) the transformation based approach in Section 4.4 (ii) the matrix based approach in Section 4.4 (iii) approach (ii) modified to use the new `rmt` from (a).

12. Apply the chaos game (a) to an isosceles right triangle and (b) to a triangle with a large obtuse angle.

13. (a) Create a variant of the chaos game where you move two thirds of the way toward the selected vertex. What does the image look like when this is applied to a triangle?
(b) Create a variant of the chaos game where you move either one third or two thirds (selected at random) of the way toward the selected vertex. What does the image look like when applied to a triangle?

14. (a) Find the fractal dimension of the X fractal appearing in Figure 4.8.6.
(b) Find the fractal dimension of the J fractal appearing in Figure 4.8.7.

15. Use the sum of powers equation at the end of Section 4.6 to compute the fractal dimension of the image in Figure 4.8.8.

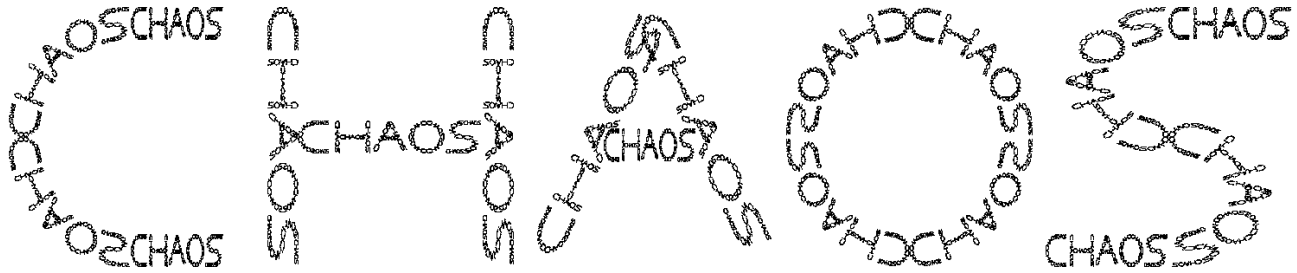


Figure 4.8.4 Chaos within Chaos

Figure 4.8.5 A Random IFS

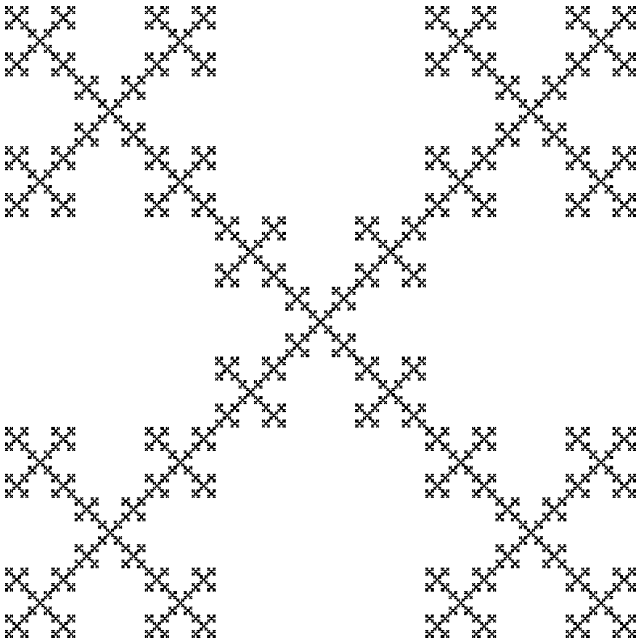


Figure 4.8.6 A Fractal X

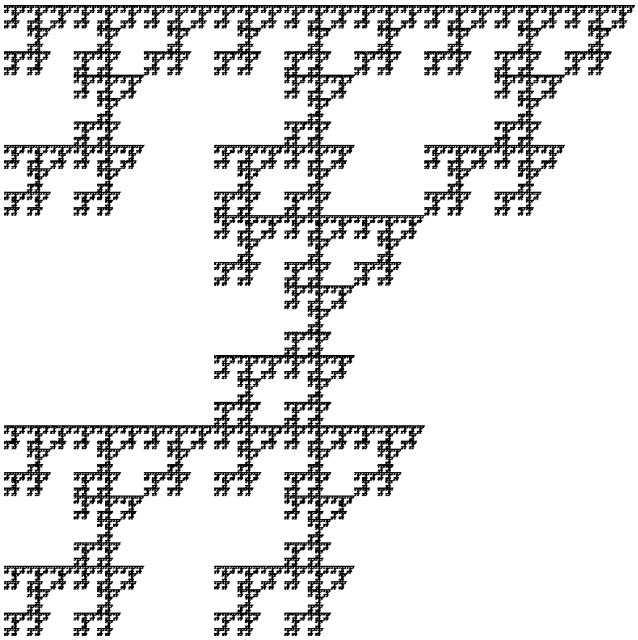


Figure 4.8.7 A Fractal J

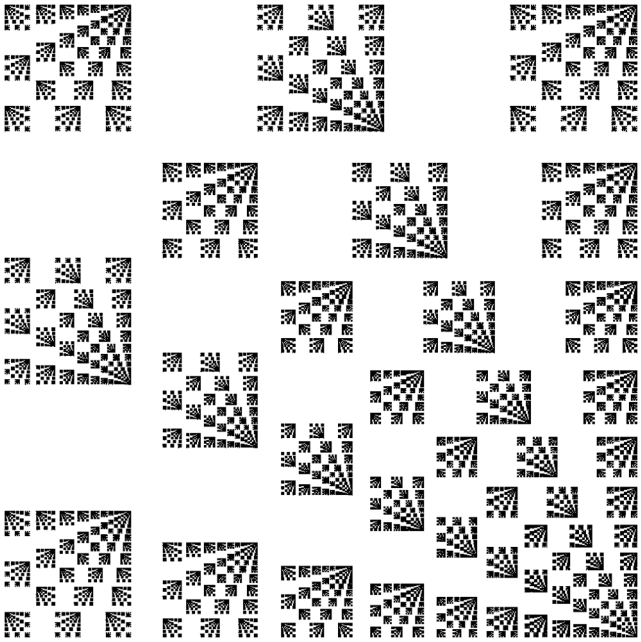


Figure 4.8.8 A Fractal from Similitudes

16. Consider the tessellations
`(3 3,:3 3)<;._3 i.6 6` and
`(1 1,:3 3)<;._3 i.6 6`
(a) How would you describe the second one?
(b) Can you now predict the result of
`(1 2,:3 3)<;._3 i.6 6`

17. In Section 4.7 non-overlapping tesselations were used to estimate fractal dimension. Use the constructions in the previous exercise to guide estimation of fractal dimension using overlapping tesselations.

Chapter 5 Color, Contours and Animations

This chapter develops methods for visualizing data that is essentially three or four dimensional by using color to express one dimension and animations are also created which adds another dimension. We also introduce user defined adverbs and conjunctions. This lays the groundwork for many of the remaining chapters. The most generic form of using color is to use it to represent the height of a function of two variables. We also apply these techniques to some fractal constructions. We begin by considering the RGB color model in more detail.

5.1 The RGB Color Model

We have seen that colors may be represented by specifying the intensities of the red, green and blue components of the color. This is known as the RGB color model. Other color models are convenient for other purposes. For example, a printer may have cyan-magenta-yellow and black inks available and hence use a model with those colors. Other color models are useful for compressing photographic data by taking advantage of the particular sensitivities of human vision. However, most computer displays are based on red, green and blue light, so we will use the RGB model for now. In Part 2 we will consider some other color spaces. Figure 5.1.1 shows the edges and main diagonal of the RGB color cube. Notice that one corner is black and the opposite diagonal corner is white. The diagonal between them corresponds to shades of gray. The three corners nearest black are red, green and blue while the three corners nearest white are yellow, cyan and magenta. When we move along an edge not involving black or white, we maintain fully saturated, intense colors. These distinct intense colors are called hues. Typically as a parameter varies from 0 to 1, the corresponding hues run through red, yellow, green, cyan, blue, magenta and back to red. We will illustrate these facts with some experiments. The images resulting from these experiments are shown in Figure 5.1.2.

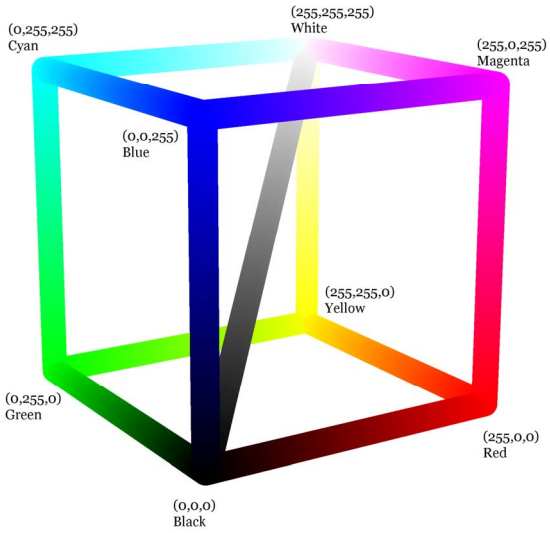


Figure 5.1.1. The RGB Color Cube

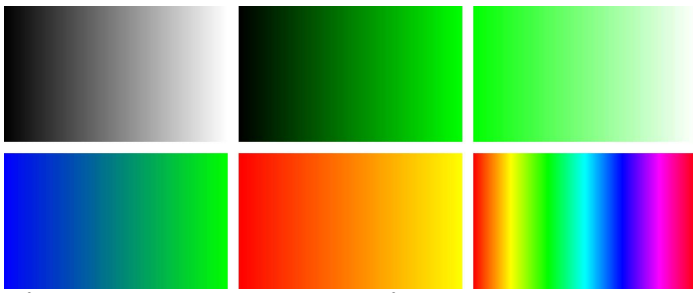


Figure 5.1.2 Palette Blending Colors

We will create a sample bitmap containing 300 rows each of which run from 0 to 255 (two copies of each index). Then we can create various 256 color palettes as follows. The viewing utility `view_image` defined in *imagekit.ijs* can be used to display images described in various formats: palette-indices, RGB triples, or just via an image filename.

```
load '~addons/media/imagekit/imagekit.ijs'          load viewing utilities
i=: i.256                                           256 indices
b=: 300 #,: 2 # i                                   create a bitmap that contains 256 colors
fnl=: 4&{. ; _4&{.                                  show first and last 4 items
```

```
   fnl p0=: 3#"0 i
```
grayscale appears in upper left of Plate 4.1.2

```
+-----+-----------+
|0 0 0|252 252 252|
|1 1 1|253 253 253|
|2 2 2|254 254 254|
|3 3 3|255 255 255|
+-----+-----------+
```

```
   view_image p0;b
```
view the palette-indices image

```
   $b{p0
```
array of RGB triples

```
   view_image b{p0
```
view the image of triples

```
   (p0;b) write_image jpath '~temp/temp0.png'
```
save the image as a file

```
   view_image jpath '~temp/temp0.png'
```
view the image from a file

```
   fnl p1=: 0,.i,.0
```
black to green palette

```
+-----+-------+
|0 0 0|0 252 0|
|0 1 0|0 253 0|
|0 2 0|0 254 0|
|0 3 0|0 255 0|
+-----+-------+
```

```
   fnl p2=: i,.255,.i
```
green to white palette

```
+-------+-----------+
|0 255 0|252 255 252|
|1 255 1|253 255 253|
|2 255 2|254 255 254|
|3 255 3|255 255 255|
+-------+-----------+
```

```
   fnl p3=: 0,.i,.|.i
```
blue to green palette

```
+-------+-------+
|0 0 255|0 252 3|
|0 1 254|0 253 2|
|0 2 253|0 254 1|
|0 3 252|0 255 0|
+-------+-------+
```

```
   fnl p4=: 255,.i,.0
```
red to yellow palette

```
+-------+---------+
|255 0 0|255 252 0|
|255 1 0|255 253 0|
|255 2 0|255 254 0|
|255 3 0|255 255 0|
+-------+---------+
```

```
   (i.%<:) 5
```
list of 5 numbers from 0 to 1

```
0 0.25 0.5 0.75 1
```

```
   fnl p5=: Hue (i.%<:)256        red to red via hues palette
+--------+--------+
|255  0 0|255 0 18|
|255  6 0|255 0 12|
|255 12 0|255 0  6|
|255 18 0|255 0  0|
+--------+--------+
```

In the next section we will see examples of user defined adverbs and conjunctions. Then in Section 5.3 we will begin to use color palettes to highlight information in images.

5.2 Adverbs and Conjunctions

We have seen that J makes use of higher order parts of speech known as adverbs and conjunctions. Adverbs take a single argument and conjunctions take two. Ordinarily we think of these as resulting in a verb although that need not be the case. Recall some examples. Insert is denoted `/` and takes a single argument as in `+/` to result in the verb that sums. Hence insert is an adverb. Iteration or function power, denoted `^:` takes two arguments as in `f^:3` which results in a verb that applies `f` three times. Thus, function power is a conjunction.

We turn to creating some illustrations of adverbs and conjunctions. The left and right arguments of explicitly defined conjunctions are designated `m` and `n` if the arguments are nouns; they are designated `u` and `v` if the arguments are verbs. Adverbs are similar except they have only left arguments. The arguments of the derived function are specified by `x` and `y` as we have previously done. While `3 : 0` is used for multi-line definition of a verb, `1 : 0` is used for defining multi-line adverbs and `2 : 0` is used for defining multi-line conjunctions. Also, nouns in the form of multi-line text can be defined using `0 : 0`. Note: remembering which number goes with which type object is relatively easy: nouns have no arguments and result from `0 : 0`, adverbs have one argument and result from `1 : 0`, conjunctions have two arguments and result from `2 : 0`, verbs have one or two arguments and result from `3 : 0`.

In Section 2.1 we looked at making plots using the utility `plot`. If we want to be able to pass the function as an argument to our plot routine, we want to define an adverb. Below we define such an adverb, `Plotf`. The function is the adverb argument, the right argument is the plot interval and the number of sample steps, which defaults to 100 if left out. The left argument is any option for the plot function. The default is to plot in black.

```
   Plotf=: 1 : 0                  define an adverb
'itemcolor 0 0 0 ' u Plotf y      monad draws in black
:
require 'plot numeric'            make sure the plot script is loaded
X=.steps 3{.y,100                 sample points
Y=.u"0 X                          function values on sample points
x plot X;Y                        plot the function with options
)

   1&o. Plotf 0 6p1               plot the sine function in black
```

We can easily make a wider plot, use the default `plot` color, and thicken the plot as follows.

```
   'aspect 0.2;pensize 3' 1&o. Plotf 0 6p1
```

Next consider computing the Fourier sine series given the coefficients. That is, given a vector `m` of data, the series is the function defined by $fss_m(x) = \sum_{1 \le k \le n} m_{k-1} \sin(kx)$ where the sum is from `1` to `n`, the number of elements in `m`. We implement an adverb `fss` that computes that series. Notice that the result of the adverb with a data argument is a tacitly defined function. Thus, this construction technique offers a convenient way to define tacit (preparsed) functions that can be efficiently applied.

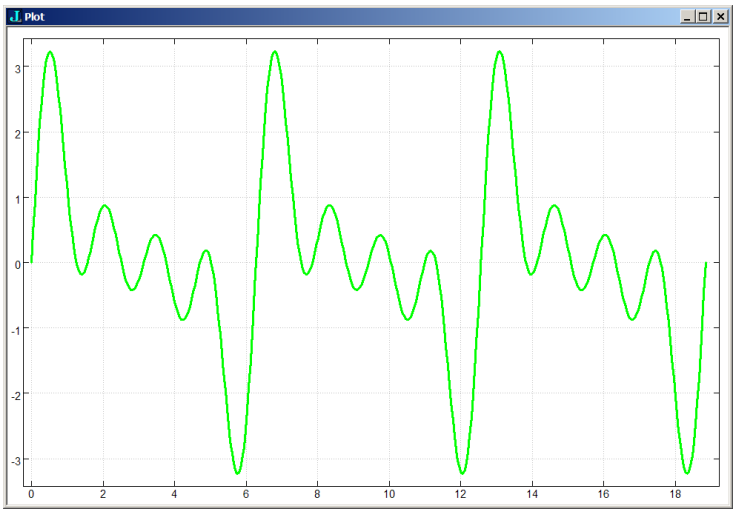


Figure 5.2.1 A Fourier Sine Series

```
   fss=: 1 : 0
m +/ . * 1 o. *&(>: i. #m)
)
```

Fourier sine series

```
   1 1 1 1 fss
```

four terms with coefficient 1

```
+-------+----------+--------------------+
|1 1 1 1|+-----+-+-+|+-+--+-------------+|
|       ||+-+-+|.|*|||1|o.|+-+-+-------+||
|       |||+|/||  |  |||  |   ||*|&|1 2 3 4|||
|       ||+-+-+|  |  |||  |   |+-+-+-------+||
|       |+-----+-+-+|+-+--+-------------+|
+-------+----------+--------------------+
```

```
   1 1 1 1 fss Plotf 0 6p1 400
```

The resulting plot is shown in Figure 5.2.1. What happens if you plot `1 1r2 1r3 1r4 fss`?

Next we consider defining a conjunction that allows us to have easy access to a family of functions. Consider functions of the form $f(t) = e^{mt}\cos(nt)$. That family is defined below as the conjunction `expcos`. Then three functions of that form are plotted.

```
   expcos=:2 : '^@(m&*) * 2 o. n&*'
```

```
   _1 expcos 3
^@(_1&*) * 2 o. 3&*
```

```
   t=:steps 0 4p1 1000
```

```
   plot t;(_0.3 expcos 2 t),(_0.5 expcos 2 t),:(_0.3 expcos 8 t)z
```

The plot is shown in Figure 5.2.2. Can you decide which function corresponds to which curve in the figure?

5.3 Experiment: Color Contour Plots

We begin by considering the function $f_1(x, y) = \sin(x) + \sin(y)$. This function is the interference pattern of two perpendicular oscillations and results in an egg carton like pattern. We load the script *raster.ijs* and use some of the utilities from that script. We will also create some new utilities in order to organize and view our data.

```
   load '~addons/graphics/fvj4/raster.ijs'
```

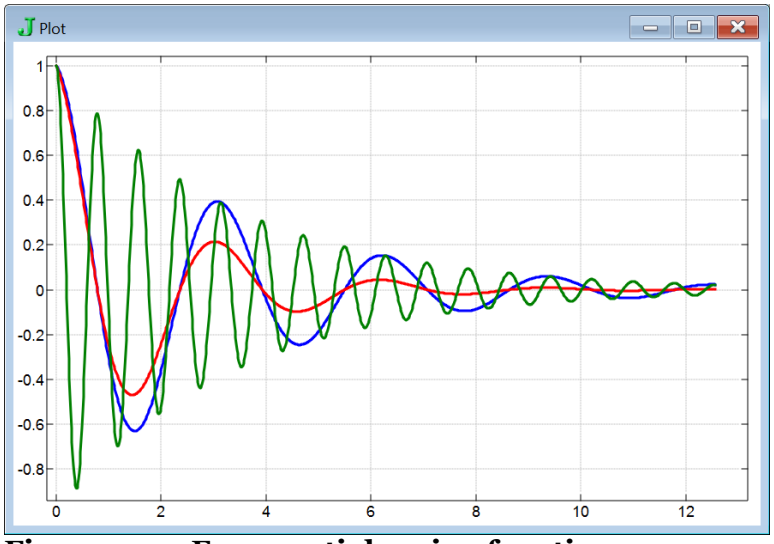


Figure 5.2.2 Exponential cosine functions.

```
   sin=: 1&o.                          the sine function

   f1=: +&sin"0                        sum of the sine of its two arguments

   clean=: * * |                       remove fuzz

   clean f1/~ 2p1*(i.%<:) 5            the function on some points; notice the oscillations
 0 1  0 _1  0
 1 2  1  0  1
 0 1  0 _1  0
_1 0 _1 _2 _1
 0 1  0 _1  0
```

We want to be able to put the function values into discrete categories that will correspond to colors. We will use `cile` that allows us to select the number of levels. We are content to use it as a utility; however, the core idea is that /:@/: gives rank order.

```
   cile=: $@] $ ((/:@/:@] <.@:* (%#)),)        defined by raster.ijs

   5 cile f1/~ 2p1*(i.%<:) 10                  scale array into 5 discrete categories
2 3 4 3 3 1 1 0 1 2
3 4 4 4 4 3 2 1 2 3
4 4 4 4 4 3 2 2 3 4
4 4 4 4 4 3 2 2 2 3
3 4 4 4 3 2 1 1 1 3
1 3 3 3 2 1 0 0 0 1
1 2 2 2 1 0 0 0 0 0
0 1 2 2 1 0 0 0 0 0
1 2 3 2 1 0 0 0 0 1
2 3 4 3 3 1 1 0 1 2

   b=: f1/~ 4p1*(i.%<:) 1000                   sample on a 1000 by 1000 array

   B=: 256 cile b                              use 256 levels

   view_image BW256;B                          view with a gray level palette


   H=:Hue 5r6*(i.%<:)256

   view_image H;B                              view with red to magenta palette
```

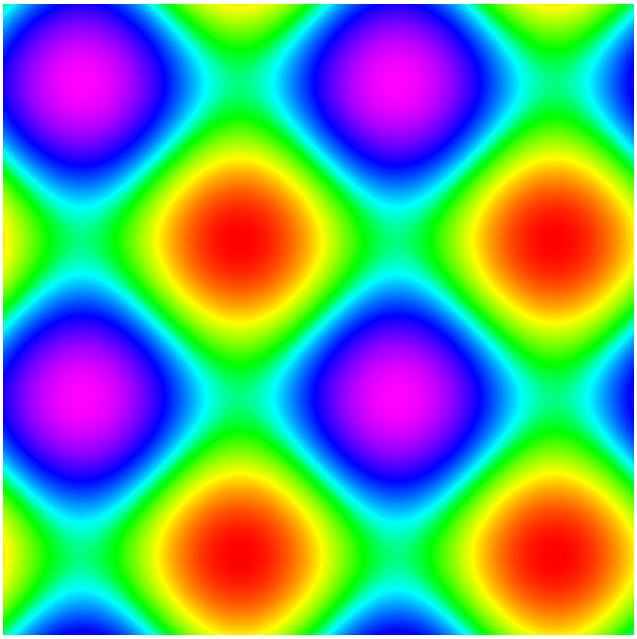


Figure 5.3.1 A Sum of Two Sines

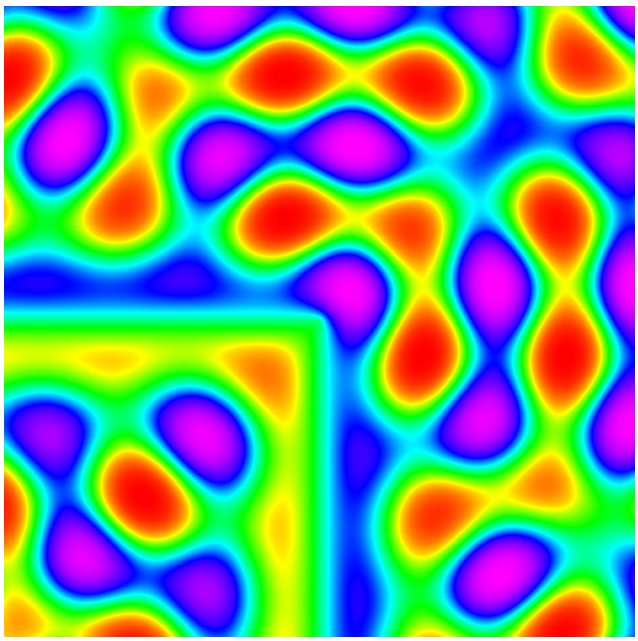


Figure 5.3.2 A Sum of Three Sines

Figure 5.3.1 shows the image created with the red (low) to magenta (high) palette. Notice that it is slightly easier to see the differing shapes of the contours when using the hue palette. We next turn to defining an adverb that automatically displays contours of its function argument. The left argument of the derived verb is the number of levels and the right argument is a list specifying the lower left and upper right corners of the region to be sampled. The function `f2` defined below uses ***under*** `&.` which post applies the inverse of its right adverb argument. That is, `x u&.v y` is `v^:_1 (v x) u (v y) .`

```
   contours=: 1 : 0
256 u contours y
:
y vwin 'contours'
'a c b d'=.y
xx=.a+(b-a)*(i.%<:){.VRAWH
yy=.c+(d-c)*(i.%<:){:VRAWH
VRA=:|:x cile xx u/ yy
vshow Hue 5r6*(i.%<:) x                palette with x hues
)

   f2=: sin@(+&.*:)                    the function sin √(x^2 + y^2)

   f3=: (f1 + f2)"0 f.                 the function sin(x) + sin(y) + sin √(x^2 + y^2)

   VRAWH=:1000 1000

   f3 contours _14 _14 14 14

   (H;|.VRA) write_image 'temp/3sines.png'
```

Figure 5.3.2 shows the contour plot of that function. Notice that there is a surprising right-angle of contours opening toward the lower left. Can you explain those contours? What does a contour plot of `f2` look like? What do level plots of these three functions look like when restricted to just two levels? The `write_image` expression shows how the result may be saved.

5.4 Animations

As a first example of an animation we will create a movie showing the evolution of the contour plot for `f1` from Section 5.3 into the contour plot for `f3=:f1+f2`. We define an adverb `F` below that allows us to vary from `f1` to `f3` as the adverb argument runs from 0 to 1. Then we use that adverb inside another adverb `mk_fseq_contours` that produces a sequence of images. The adverb argument to `mk_fseq_contours` is the bounds for the contour plots. The left argument is the values that the adverb argument to `F` should take. The right argument is the file name for the first frame of the movie. It should have a numeric suffix. The make directory utility `mkdir` assumes the directory does not exist but that its parent does.

```
   load '~addons/graphics/fvj4/raster.ijs'

   sin=: 1&o.

   f1=: +&sin"0

   NB. cile=: $@] $ ((/:@/:@] <.@:* (%#)),)

   H=:Hue 5r6*(i.%<:)256

   f2=: sin@(+&.*:)

   F=: 1 : '(f1 + m * f2)"0 f.'

   mk_fseq_contours=: 1 : 0
:
'a c b d'=.m
xx=.a+(b-a)*(i.%<:){.VRAWH
yy=.c+(d-c)*(i.%<:){:VRAWH
fn=.y
for_X. x do.
  VRA=:|:256 cile xx X F/ yy
  (H;VRA) write_image fn
  fn=.nx_fn fn
end.
#x
)

   mkdir=:1!:5@<

   mkdir path=:'d:\temp\fs1\'
1

   (0.01*i.101) _14 _14 14 14 mk_fseq_contours path,'con000.png'
101
```

Now we load the file sequence into a movie utility. For example, using Quicktime Player™ one can use the “file”-“open image sequence” to select those frames. Then an appropriate frame rate can be chosen (15 frames per second works well for the above sequence). The resulting movie can then be saved as a single animation file in various possible formats.

As a second illustration we will create a zoom into an image file. We begin with selecting a square portion of a sample image of Ken Iverson at Kiln Farm. The function `mk_zoom` takes the image as its adverb argument, the movie size as its left argument and an initial image filename for the first frame.

```
   load '~addons/media/imagekit/imagekit.ijs'

   $pic=: read_image jpath '~addons/graphics/fvj4/atkiln.jpg'
700 468 3

   view_image b=:400 400{.0 68}.pic
```

```
   mk_pic_zoom=:1 : 0
:
fn=.y
'w h'=.x
for_k. i.n=.<.-:h-~#m do.
  ((h,w) resize_image m)write_image fn
  fn=.nx_fn fn
  m=.1 1}._1 _1}.m
end.
n
)

   mkdir path=:'d:\temp\zoom\'
1

   200 200 b mk_pic_zoom path,'z000.jpg'
```

The files can then be assembled into a movie as described in the previous example.

5.5 Plasma Clouds

Plasma clouds are phenomena that have both coherence and randomness. We use a type of midpoint interpolation with random variations in order to construct examples of plasma clouds. Projections of data created this way will produce the fractal mountains that we will consider in Part 2. Remarkably, only very small augmentations of the expressions giving random walks in Section 3.7 are required. We first duplicate functions from that section below that are directly useful. These functions are also defined in the script *~addons/graphics/fvj4/povkit.ijs* so you may save typing by loading that script. We also assume the script *raster.ijs* has been loaded.

```
   randunif=: ?@($&0) : ({.@[+({:-{.)@[*$:@])

   cos=: 2&o.

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

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

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

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

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

To apply the random additions technique to a matrix, we need to interpolate 2-dimensional arrays. See `interp2` below. The plasma field builder `plasma` is the same as `hwalk` from Section 3.7 except that the initial array to which we apply the process is a 2 by 2 matrix and we use `interp2` instead of `interp` for the interpolation.

```
   interp2=: interp"1@:interp          interpolate in two directions

   plasma=: 4 : 0
x ([randadd interp2@])^:y (x osz 1)*randsn 2 2
)

   setseed=:9!:1

   setseed 7^5                         reset random seed
```

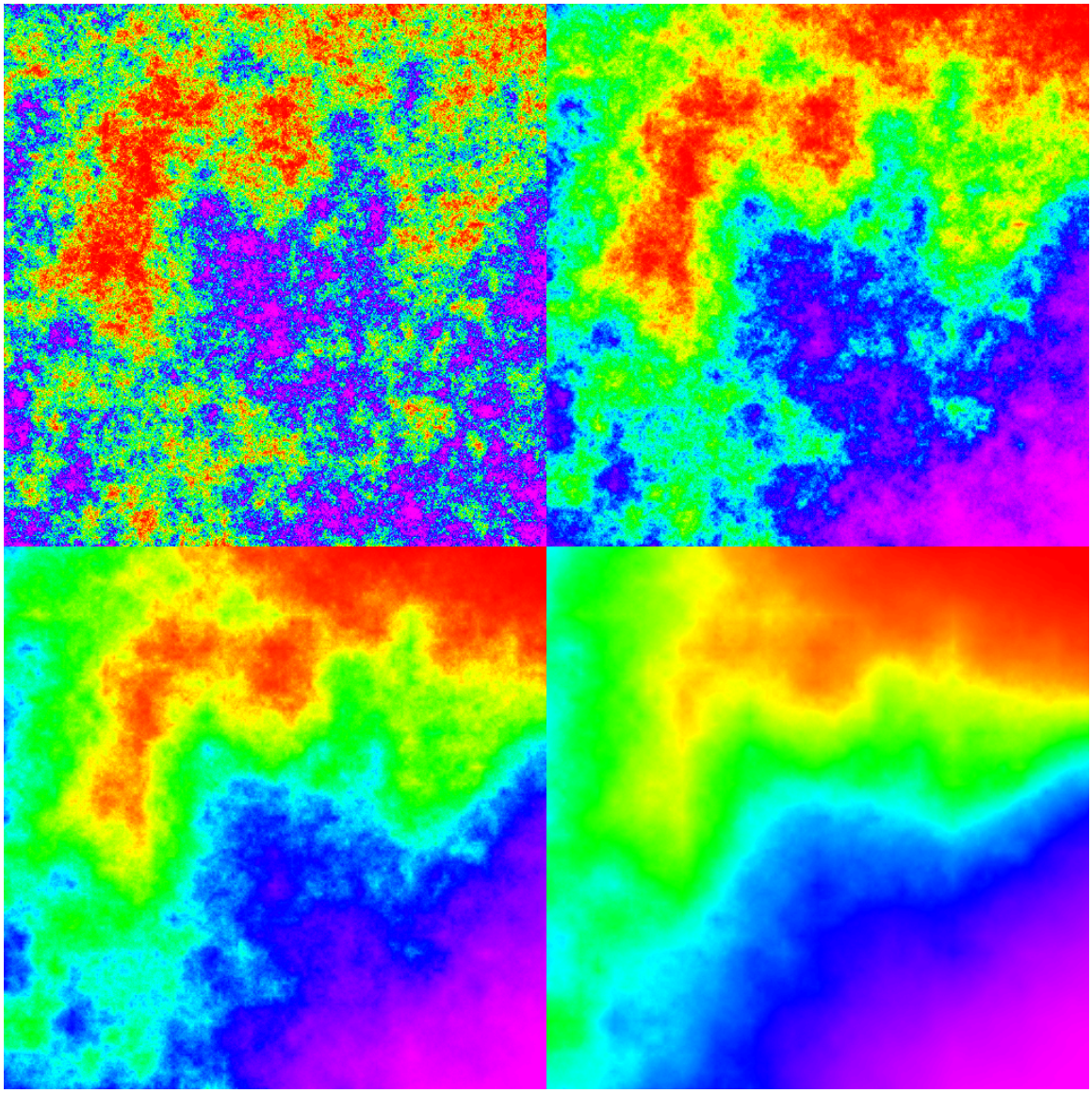


Figure 5.5.1 Plasma Clouds with Hurst Exponent 0.1, 0.4, 0.6, and 0.9.

```
   3 cile 0.6 plasma 3
2 1 1 0 1 1 1 1 2
1 1 0 1 1 0 1 2 2
1 1 2 0 0 0 0 0 1
1 2 2 1 0 0 0 0 0
2 1 1 2 2 1 1 0 0
1 2 2 2 2 1 1 0 0
2 2 2 1 2 0 1 0 0
2 2 2 2 0 1 2 0 0
2 2 2 2 1 0 0 0 0
```

three levels of interpolation and random addition; notice the randomness; notice the coherence

```
   B=: 256 cile 0.1 plasma 10
```

Hurst exponent 0.1

```
   view_image P256;B
```

view the image

Recall that the left argument is the Hurst exponent and the right argument is the number of iterations to apply. The result of the above computation with Hurst exponent 0.1, 0.4, 0.6 and 0.9 is shown in Figure 5.5.1 using the red to magenta palette `H` from Section 5.2. We reset the random seed to its default value before each plasma cloud was created so the only variation was the Hurst exponent.

5.6 Experiment: Palettes and Inner Product Fractals

Our goal for this experiment is to gain experience with using color images to represent matrices that contain interesting fractal structure. These images can be created with simple matrix arithmetic that may involve Boolean arithmetic, relational arithmetic and other arithmetic functions. The key part of the construction will be a generalized matrix product of indices listed in binary and other bases.

We begin by giving a construction of the Sierpinski triangle using matrix arithmetic. Recall that binary representations express a number as a sum of 0 or 1 times powers of 2. For example, since $13 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$, `13` has the binary representation `1 1 0 1`. Consider the matrix created by applying the antibase function `#:`, which we saw gives binary representations.

```
   #: 13                binary representation of 13
1 1 0 1

   ]M=: #:i.2^3         binary representations of 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
```

Thus, we get all of the three digit binary representations. Recall that the ordinary dot product of vectors is denoted `+/ . *` and it gives the sum of the pairwise products of its vector arguments. The matrix product is also denoted `+/ . *` and it gives the dot product using each row of its left argument with each column of its right argument. In general it takes dot products using the trailing axis of its left argument and leading axis of its right argument. Now consider the matrix product of `M` with its transpose.

```
   ]b=: M +/ . * |: M
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 2 0 1 1 2
0 0 0 0 1 1 1 1
0 1 0 1 1 2 1 2
0 0 1 1 1 1 2 2
0 1 1 2 1 2 2 3
```

We are interested in the pattern of zeros in `b`. We can highlight the pattern of zeros by comparing each entry to `0`. Try the following.

```
   0 = b                mark the zeros in b
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 see the pattern better by creating an image from a larger example. Redefine `M` to give all the `9` digit binary representations. Define the new `b` corresponding to that `M` and then create an image of the pattern given by `0 = b` as follows.

```
   $M=: #:i.2^9
512 9

   $b=: M +/ . * |: M
512 512

   ]pal=: 255 255 255,: 0 0 0      black and white palette
255 255 255
  0   0   0

   $B=: 0=b                        the arrray B is a 0-1 matrix
512 512
```

We will consider some methods for creating images from `b` and `B`. Since `B` contains only `0` and `1` entries, it is the simplest. In order to create an image for `B`, we defined a black and white palette above; the first row of pal is `255 255 255` which is white; the second row corresponds to black. The matrix `B` is a matrix of zeros and ones; we want an image of `B` with the color of each pixel chosen according to the palette. We can view that matrix with that palette as follows.

```
   load '~addons/media/imagekit/imagekit.ijs'

   view_image pal;B
512 512
```

We can see the pattern of zeros and ones in the 512 by 512 matrix `B`. This is an image of the (skew) Sierpinski triangle that we constructed in Sections 1.5 and 1.7.

We saw that the matrix `b` had entries other than `0` and `1`. If we want to view all the distinct entries as different colors, one approach is as follows. We use the nub, `~.` , which removes duplicate items, to see what atoms appear in the array.

```
   ~.6 6 7 0 5 0 5 0 7             nub removes duplicates
6 7 0 5

   ~.,b                            our matrix has entries between 0 and 9
0 1 2 3 4 5 6 7 8 9
```

We would like to highlight the zeros by making them black and then create nine pure hues for the colors of the other entries.

```
   5r6*(i.%<:) 9                   list of nine numbers from 0 to 5r6
0 0.104167 0.208333 0.3125 0.416667 0.520833 0.625 0.729167 0.833333

   ]pal=: 0,Hue 5r6*(i.%<:)9       palette of black adjoined to nine pure hues
  0   0   0
255   0   0
255 159   0
191 255   0
 32 255   0
  0 255 128
  0 223 255
  0  64 255
 96   0 255
255   0 255

 view_image pal;b
```

We will want to eventually become comfortable with controlling the details of our palettes, but `view_data` function, loaded with the script *view_m.ijs*, often picks a reasonable palette as does `viewmat` from the system *viewmat* script.

```
    view_data b          these auto-palette the image
512 512

   view_data B
512 512
```

View and compare the three different images: the "black and white" image, the "black and hue" image and the "`view_data`" image. Notice each has its own merits.

J has many other primitive dyads and we can investigate them relatively easily using "tables" of small values. The utility `table` is loaded automatically in the default J setup. Try the following.

```
   + table 0 1 2 3          plus table
   * table 0 1 2 3          times table
   <. table 0 1 2 3         minimum table
   >. table 0 1 2 3         maximum table
```

J uses `0` for false and `1` for true. We can see various relational tables and logical tables. Try the following.

```
   < table 0 1 2 3          less than table
   <: table 0 1 2 3         less than or equal table
   = table 0 1 2 3          equal table
   +. table 0 1             logical "or"
   *. table 0 1             logical "and"
```

Run some experiments to see how logical "and" and "or" are generalized to non-Boolean (not just 0 and 1) integers. Hint: look at tables on `i.11`. These are familiar functions.

Above we created a color image of the ordinary matrix product of the matrix with its transpose via the following.

```
   b=: M +/ . * |: M
```

Recall that the matrix product gives sums (plus insert) of pairwise products. Plus and times can be replaced by any dyads. Thus if you want to take alternating differences (the minus insert) of pairwise minimums, you can use the following. However, because some of the entries are negative, some extra care is necessary to select an index set and palette. We assume `M` is the 9-digit base 2 integers as above.

```
   b=: M -/ . <. |: M

   ]nb=:~.,b                the nub
0 1 _1 2 _2 3 _3 4 _4 5

   nb i. _2 5               indices of _2 and 5 relative to the nub
4 9                         using dyadic i. which is "index of"

   pal=:Hue 5r6*(i.%<:)#nb

   view_image pal;nb i. b
512 512

   (pal;nb i. b) write_image jpath '~temp/ipf.png'
```

Try various other combinations of functions in the place of plus and times in the construction of `b`. Keep any striking images and record how you created them for future reference. Two interesting variants worth viewing are given by the following and shown in Figures 5.6.1 and Figure 5.6.2.

```
M=: #:i.2^8

b=: M #.&|: . * |: M

pal=: 0,Hue 5r6*(i.%<:)255

b=: M i.&1"1&|: . * |: M

pal=: 0,~Hue 5r6*(i.%<:)8
```

The function, used above, `#.` is called ***base***. Next we turn to generalizing our experiments to a base-three generated matrix `M`. We saw that monadic `#:` gives binary representations. If a left argument is given, it specifies a list of the bases used for the corresponding digits and the function is called antibase. The following gives all the two digit base-three representations of 0 to 8.

```
   3 3#:i.3^2
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

We can also check that `3 3 3#:i.3^3` gives all the three digit base-three representations of `0` to `26`. Create a matrix `M` that gives all the 6-digit base-three representations. When generalized beyond Boolean values, `+.` gives *gcd* (***greatest common divisor***) and `*.` gives the *lcm* (***least common multiple***); perhaps you figured that out earlier. Create an image of *gcd* inserts over the list of pairwise *lcm*'s. This is the Sierpinski carpet. Try various combinations of other functions. Keep any striking

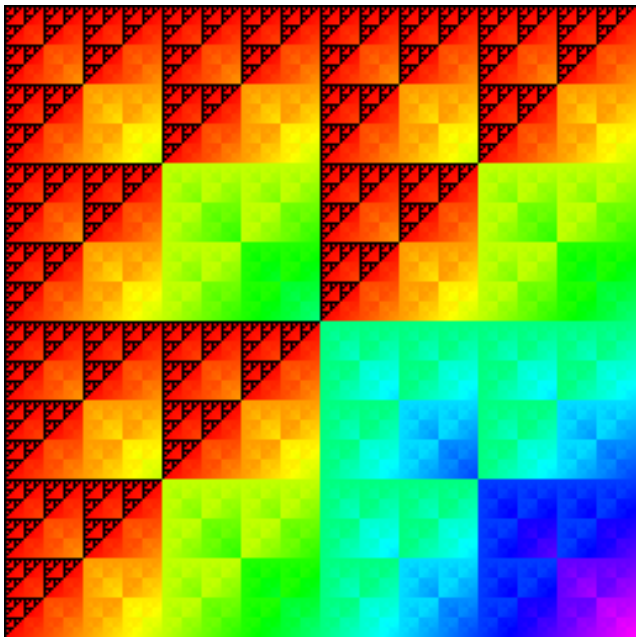


Figure 5.6.1 Inner Product Fractal I

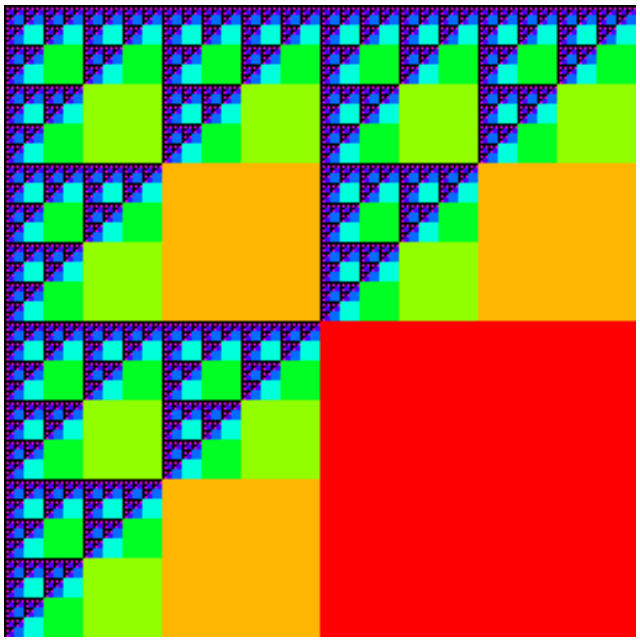


Figure 5.6.2 Inner Product Fractal II

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
```

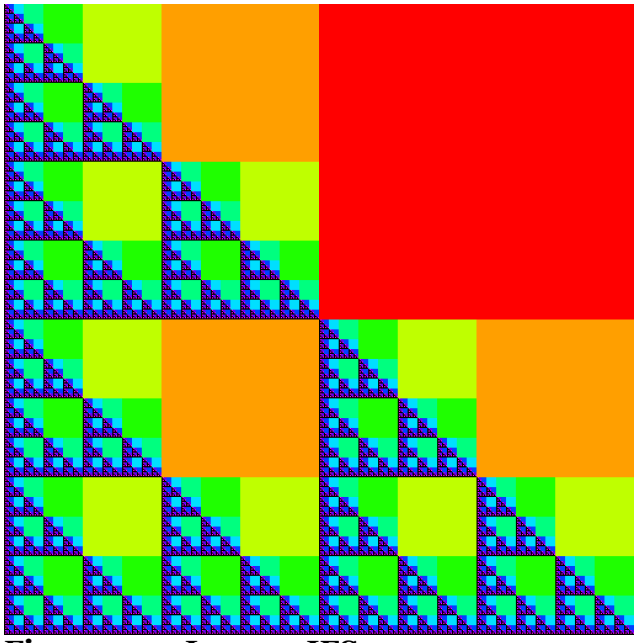


Figure 5.7.1 Inverse IFS

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

4. Create the Kimberly Corbett funky fractal: this is produced in the style of the fractals in Section 4.2 using `+./ . *` on three digit base 7 indices.

5. Create contour plots of the following functions using the approach of Section 5.3.

(a) $f_1(x,y) = \dfrac{x^2 - y^2}{x^2 + y^2}$ (b) $f_2(x,y) = \dfrac{2xy^2}{x^2 + y^4}$

(c) $f_3(x,y) = \sin(xy)$ (d) $f_4(x,y) = (x^2 + y^2)e^{-(x^2+y^2)}$

6. You can use the density plot option of the plot function to create contour plots. Consider the following.

```
load 'plot'
sin=: 1&o.
f1=: +&sin"0
b=: f1/~ 4p1*(i.%<:) 100
'density' plot b
```

Use the density plot utility to create contours plots for the functions given in Exercise 5.

7. Write a variant of the function `contours` from Section 5.3 which creates grayscale contours.

8. Write a variant of `contours` that creates a white and black image showing the level contours.
Use a black and white palette with several white entries between a couple black entries. Use that facility to create contour plots for the functions given in Exercise 5.

9. Using the construction in Exercise 6, `'contour' plot b` creates a contour plot showing contours as curves. Use this facility to create contour plots for the functions given in Exercise 5.

10. Create nine plasma fields with Hurst exponents given by `0.1*1+i.9`.

11. Use the utilities in Section 5.4 to create an animation of the evolution of the contour plots of family of functions beginning at $\dfrac{x^2}{x^2 + y^2}$ and ending at $\dfrac{x^2 - y^2}{x^2 + y^2}$.

12. In a manner similar to the example in Section 5.4, create an animation of a zoom into a feature of your choice in an image of your choice.

13. Create an animation of plasma clouds with 99 Hurst exponents between 0 and 1. Reset the random seed to the same value before creating each frame in the animation.

14. Develop the plasma cloud associated with a biased input array. Use an input array that appears to have five dots arranged in the form of the five dots on a die as your biased input.

15. Modify the definition of `isier` in Section 5.7 to replace the verb `i3` by `i2`. Produce the corresponding image. How is it different from Figure 5.7.1?

16. Create an inverse iterated function system for the Sierpinski carpet. See Figure 1.11.5.

17. Create an adverb that takes a function as argument and results in an escape time image of iterating the function. The arguments of the derived function should specify the domain, give the sense of large, and bound the number of iterates.

Chapter 6 Complex Dynamics

6.1 Experiment: Julia Sets

Next we consider the iteration of "simple" complex valued functions. We will see that while these functions have a simple algebraic form, they do not need to have simple escape time images. Our complex escape time conjunction, `escapetc`, is similar to the conjunction `escapet` from Section 5.7 except that the complex magnitude is used to measure largeness and the inputs are complex, rather than pairs. Our first example is to investigate the escape time of $f(z) = z^2 - 0.2 + 0.8i$ as a function of a complex variable. The function `escapetc` is loaded by running *complex_dynamics.ijs;* however, to follow along the reader needs to define `f`. As before, the right conjunction argument is a pair of numbers giving the sense of escape and the maximum number of iterations.

```
   load '~addons/graphics/fvj4/complex_dynamics.ijs'

   *: 1j2                    square a complex number
_3j4

   f=: +&_0.2j0.8@*:         the function adds _0.2j0.8 to the square

   f^:(i.4) 1.5              some iterates
1.5 2.05j0.8 3.3625j4.08 _5.53999j28.238

   escapetc                  view the definition
2 : '#@((,u@{:)^:(<&({:n)@# *. (<&({.n)@|@{:))^:_) f."0'

   f escapetc (4 100) 1.5    magnitude larger than 4 after 3 steps
3
```

The escape time pictures can have large regions where the function remains finite. That region is shown in white (up to our computational precision). The white region is known as the filled Julia set for the function; see [Devaney 1990] for a more detailed discussion of the behavior of functions such as these. The boundary of the white region is the Julia set. A simple example of a Julia set is the Julia set for the function that gives the square of a complex argument. Its Julia set is the unit circle. Figure 6.1.1 shows the escape time image for that function. Figure 6.1.2 shows the filled Julia set and escape time for the iteration of the function `f=:+&_0.2j0.8@*:` which can be constructed as follows.

We display the utility `zl_clur` that was loaded when *complex_dynamics.ijs* was loaded. It can be used to create convenient arrays of complex inputs. In particular, the left argument of `zl_clur` gives the approximate number of horizontal steps to use while the vertical (imaginary direction) steps are chosen to maintain aspect ratio; the right argument is a pair of numbers, the center left point in the sample array and the upper right point. Note that `zl_clur` uses the circle functions 9 (real part) and 11 (imaginary part).

```
   zl_clur                   view the definition
4 : 0
w=.-~/9 o.y
h=.-~/11 o.y
xs=.({.y)+w*(i.%<:)1+x
ys=.h*(i:%j.)<.0.5+x*h%w
ys +/ xs
)
```

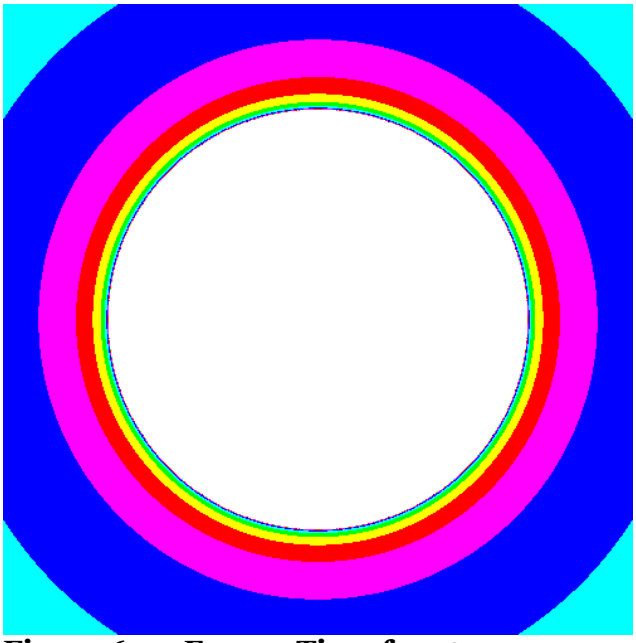


Figure 6.1.1 Escape Time for z^2

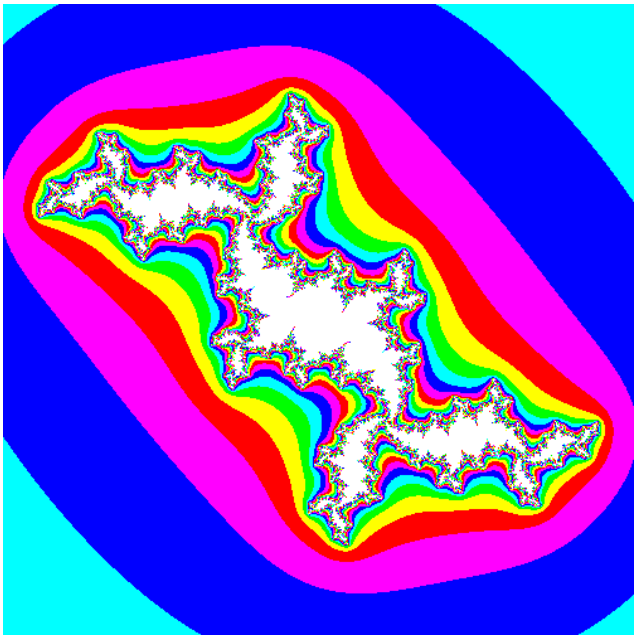


Figure 6.1.2 Escape Time for z^2-0.2+0.8i

```
   4 zl_clur _2 2j2
 _2j2  _1j2  0j2  1j2  2j2
 _2j1  _1j1  0j1  1j1  2j1
   _2    _1    0    1    2
_2j_1 _1j_1 0j_1 1j_1 2j_1
_2j_2 _1j_2 0j_2 1j_2 2j_2
```

gives a complex array of inputs to `escapetc`
notice the center left and upper right entries in bold

```
   4 zl_clur _2 2j1
 _2j1  _1j1  0j1  1j1  2j1
   _2    _1    0    1    2
_2j_1 _1j_1 0j_1 1j_1 2j_1
```

```
   b=: f escapetc (10 255) 500 zl_clur _1.5 1.5j1.5

   pal=: 255,~Hue 5r6*(i.%<:)255

   view_image pal;b
```

Next try the following palette which darkens the hues, depending on the index modulo 3. This enhances the contrast of the regions with different escape times.

```
   pal2=: 255,~0,<.(254$1 0.8 0.6)*Hue 5r6*(i.%<:)254

   view_image pal2;b
```

Lastly, consider the palette which stripes the escape time mod 6 by hue.

```
   pal3=: 255,~255$ Hue 5r6*(i.%<:)6
   view_image pal3;b
```

That palette is the one used both for Figures 6.1.1 and 6.1.2. Now create escape time images for the following functions.

```
f0=: *:

f1=: +&_0.1j0.8@*:

f2=: +&0.1j0.6@*:

f3=: +&0.1j0.61@*:

f4=: +&0.1j0.62@*:
```

You should see that the results can depend on the constants in a very delicate manner. In Section 6.3 we investigate the Mandelbrot set, which is a kind of roadmap to the Julia sets. In particular, in that section we will see an example of the fact that the Julia sets share features with related regions of the Mandelbrot set.

6.2 Experiment: Julia sets for Elliptic Curves

Curves of the form $y^2 = a_0 + a_1 x + a_2 x^2 + x^3$ are known as elliptic curves. We will explore the escape time of iterating $f(x) = \sqrt{a_0 + a_1 x + a_2 x^2 + x^3}$ in the complex plane. We assume *complex_dynamics.ijs* has been loaded and that `pal2` from Section 6.1 is available. We begin with an illustration and then proceed to consider how to experiment in order to find choices for the parameters that give interesting images.

```
f=:[: %: 0 0.4j0.1 0 1&p.

b=:f escapetc (10 255) 500 zl_clur _1.5 1.5j1.5

view_image pal2;b
```

The result is shown in Figure 6.2.1. We next turn to trying to find interesting examples by randomly choosing the coefficients and seeing whether the result is interesting. We trim our search by creating small images and then demanding that some number of pixels are white and that there is a decent variety of escape times. The function `mk_gis` displayed below does that. It was loaded by *complex_dynamics.ijs*. Its right argument has two bounds, the minimal number of distinct escape times and the minimal number of white pixels. If the small image created passes those tests, then a larger escape time array is computed. The left argument gives a scaling factor that can be used to bias the search. The result is the number of distinct escape times in the larger image and a display of the image. The global results include `coef` which gives the coefficients of the radicand, the function `f` that was iterated, and `b`, the array of indices.

Figure 6.2.1 Escape Time for an Elliptic Curve

```
load 'numeric'

randomize ''
```

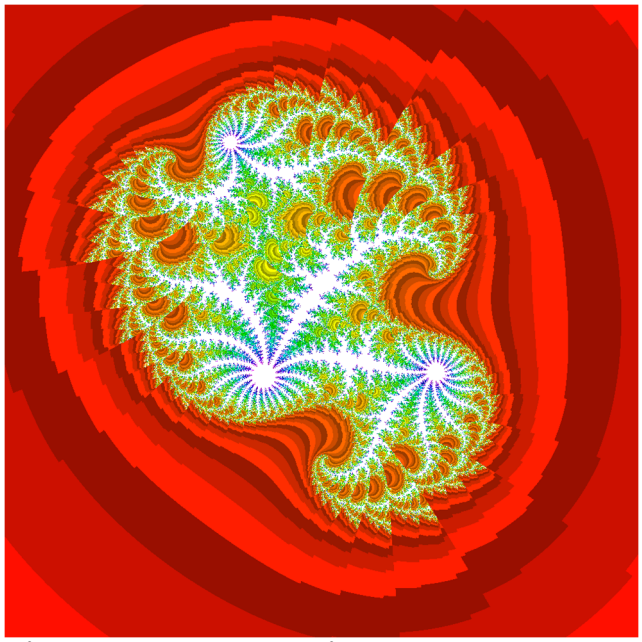


Figure 6.2.2 Escape Time for a Random Elliptic Curve

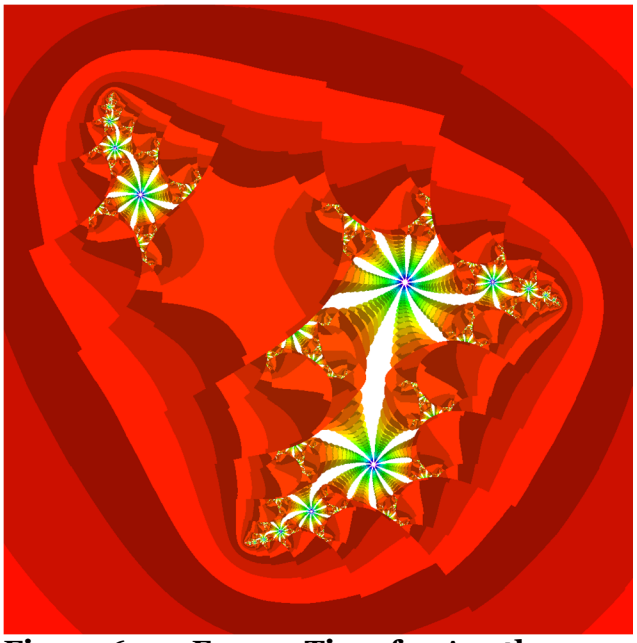


Figure 6.2.3 Escape Time for Another Random Elliptic Curve

```
   mk_gjs                    view the definition
3 : 0
1 mk_gjs y
:
'nnb rqw'=.y
whilst. (nnb>#~.,b) +. rqw>+/255=,b do.
  coef=: 1,~x*j./-/?2 2 3$0
  f=:[: %: coef&p.
  b=:f escapetc (10 255) 100 zl_clur _1.5 1.5j1.5
end.
b=: f escapetc (10 255) 500 zl_clur _1.5 1.5j1.5
view_image pal2;b
#~.,b
)
```

```
   0.8 mk_gjs 100 10

   view_image pal2;b
```

We repeated computations a couple dozen times, and 3 of the more interesting results appears in Figure 6.2.2, 6.2.3, and 6.2.4. There are also other ways to find interesting parameters as we will see in the next section.

6.3 The Mandelbrot Set

The Mandelbrot set is an escape time image of a different type. The escape time associated with a position `c` in the complex plane is the escape time of `0` under the iteration of the function $c + z^2$. Unlike Julia sets, position in the plane corresponds to different functions. There is a sense in which the Mandelbrot set gives a kind of road map for Julia sets.

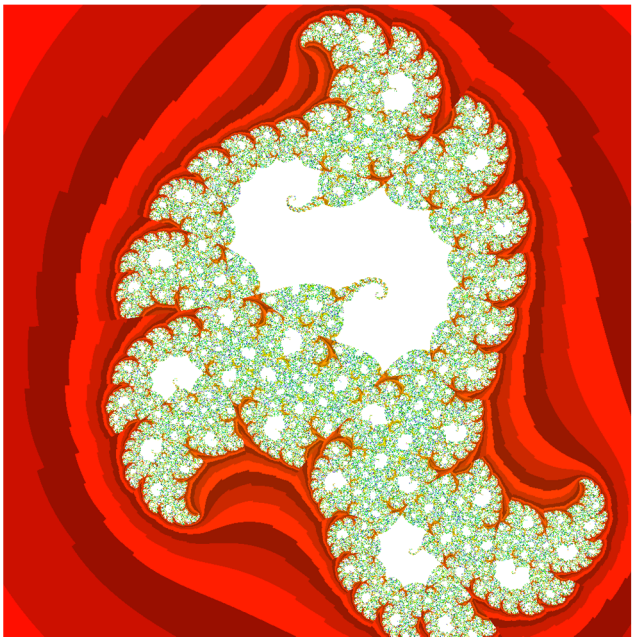


Figure 6.2.4 Escape Time of Another Random Elliptic Curve

```
   load '~addons/graphics/fvj4/complex_dynamics.ijs'

   mandelt=: (3 : 'y&+@*: escapetc (10 255) 0')"0

    4 zl_clur _2 1j1.5
  _2j1.5   _1.25j1.5   _0.5j1.5   0.25j1.5   1j1.5
 _2j0.75  _1.25j0.75  _0.5j0.75  0.25j0.75  1j0.75
      _2       _1.25       _0.5       0.25       1
_2j_0.75 _1.25j_0.75 _0.5j_0.75 0.25j_0.75 1j_0.75
 _2j_1.5  _1.25j_1.5  _0.5j_1.5  0.25j_1.5  1j_1.5

   mandelt 14 zl_clur _2 1j1.5
  4   4   4   4   5   5   5   5   5   5   5   5 4 4 4
  4   4   4   5   5   5   5   5   6   6   5   5 5 4 4
  4   5   5   5   5   5   6   6   7   8   6   6 5 5 4
  5   5   5   5   6   6   7   7  14  28   8   6 6 5 5
  5   5   6   6   6   7   8  17 218 255  38 255 6 5 5
  5   6   6   8   9   9  12 255 255 255 255  58 7 5 5
  6   6   7  10 255 255 255 255 255 255 255 255 7 6 5
255 255 255 255 255 255 255 255 255 255 255  11 7 6 5
  6   6   7  10 255 255 255 255 255 255 255 255 7 6 5
  5   6   6   8   9   9  12 255 255 255 255  58 7 5 5
  5   5   6   6   6   7   8  17 218 255  38 255 6 5 5
  5   5   5   5   6   6   7   7  14  28   8   6 6 5 5
  4   5   5   5   5   5   6   6   7   8   6   6 5 5 4
  4   4   4   5   5   5   5   5   6   6   5   5 5 4 4
  4   4   4   4   5   5   5   5   5   5   5   5 4 4 4
```

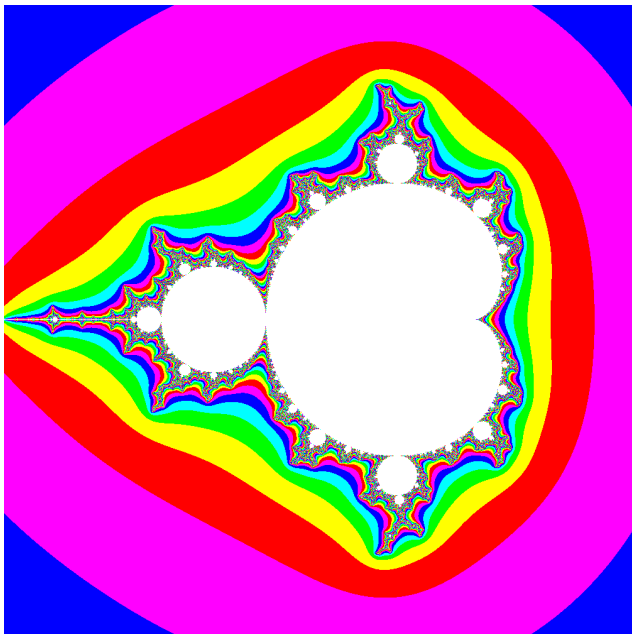


Figure 6.3.1 The Mandelbrot Set

Notice that the values 255 form a complicated central region. That region is known as the Mandelbrot set. Figure 6.3.1 shows a higher resolution version of that image using `pal3` from Section 6.1. We can also zoom into regions of interest. For example, we saw that the function `f=:+&_0.2j0.8@*:` had a moderately interesting Julia set showing regions with three spirals nearly touching. We will look at the Mandelbrot set around the point `_0.2j0.8` but first we will view a utility for creating input complex arrays based upon central point and the center of the right hand edge. The function `zl_cccr` was defined when *complex_dynamics* was loaded.

```
   zl_cccr
4 : 0
w=.--/y
({.y)+w*((i:%j.) +/ (i:%]))) <.-:x
)
```

```
   3 zl_cccr _0.2j0.8 _0.05j0.8        points around _0.2j0.8
_0.35j0.95 _0.2j0.95 _0.05j0.95
 _0.35j0.8  _0.2j0.8  _0.05j0.8
_0.35j0.65 _0.2j0.65 _0.05j0.65

   b=: mandelt 500 zl_cccr _0.2j0.8 _0.15j0.8

   pal=: 255,~<.(255$1 0.6)*Hue
5r6*(i.%<:)255

   view_image pal;b
```

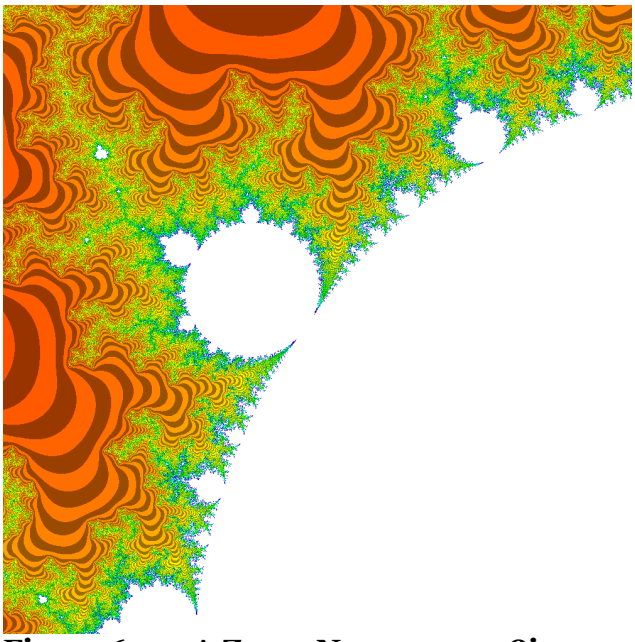


Figure 6.3.2 A Zoom Near -0.2+0.8i

Figure 6.3.2 shows the Mandelbrot set in that region using a palette that uses hue to distinguish escape time, but alternates intensity so that the contours are readily visible. We see that the triple spirals have similar structure to the spirals we saw for the corresponding Julia set. It is in that way that the Mandelbrot set is a roadmap for Julia sets.

Many remarkably fast implementations of Mandelbrot drawing algorithms have been written and *fractint* is one such that can be easily found on the web. We will explore some additional regions and palettes in the exercises, but recommend specialized software, such as *fractint*, for computationally exploring the details of the Mandelbrot set.

6.4 The 3*x*+1 Function in the Complex Plane

Recall that in Section 4.1 we introduced the $3x+1$ function, defined on integers, as follows.

$$t(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ (3x+1)/2 & \text{if } x \text{ is odd} \end{cases}$$

We also saw the conjecture that iteration of that function on any positive integer leads eventually to the cycle 2 1. It turns out there is a very nice idea [Terras, 1976] for expressing those cases together in a natural way. In particular,

$$T(x) = \frac{1}{2}\left(\text{mod}_2(x) + x3^{\text{mod}_2(x)}\right)$$

where $\text{mod}_2(x)$ is a function that is `0` for the even integers and `1` for the odd integers. Since there are several ways to define the function $\text{mod}_2(x)$ with smooth complex valued functions, we consider $T(x)$ to be a generalization of $t(x)$ to complex arguments. Here we use $\text{mod}_2(x) = \frac{1}{2}\left(1 - e^{i\pi x}\right)$. We assume that *complex_dynamics.ijs* is loaded. Notice below that `mod2` and `T` behave as desired on integers.

```
   mod2=: -:@-.@^@(0j1p1&*)

   mod2 0 1 2 3 4
0 1 0 1 0

   T=: [: -: mod2 + ] * 3 ^ mod2

   T 3 8
5 4
```

However, since `T` is a kind of doubly exponential function, even modest values may lead to the function being unable to evaluate. Consider the following.

```
   T^:(11) _2j1
_15.3616j_18.0779

   T T^:(11) _2j1
|limit error: T
|       T T^:(11)_2j1
|[-0]
```

We handle this by using adverse, as in `T :: _:` which applies the right hand function, here infinity, whenever an error occurs. We can produce a small example computation as follows.

```
   T :: _: escapetc (1e4 253) 6 zl_clur _2 2j1
 13  11 253 253 253 253 253
253 253 253   8 253 253 253
 53 253 253 253 253 253 253
253 253 253 253   4 253  23
253   3   3 253   3   3 253
```

Next we define a much larger example (that may take a few minutes to execute) and create a special palette. A slightly larger version of this computation is shown in Figure 6.4.1.

```
   b=:T :: _: escapetc (1e4 255) 800 zl_clur _5 15j2.5

   $pal3x=:255,0,~254{.<.,/( >:-:(i.%-)22) */(Hue (i.%])12)
256 3

   view_image pal3x;b
```

Notice the wild plumes of rapid escape, but large regions that do not escape (remain black). Of course, we know that the positive integers in this picture converge to the cycle `1 2` and hence are black. Interestingly, it is not obvious where those integers fall. While the complex dynamics we see does not offer any new insights into the behavior of the integer $3x+1$ function, this example is a good illustration of how processes that might be viewed as having only an integer domain, can often be generalized to much broader domains, such as the complex numbers, and the complicated image we produced makes it clear that this function has very rich dynamics.

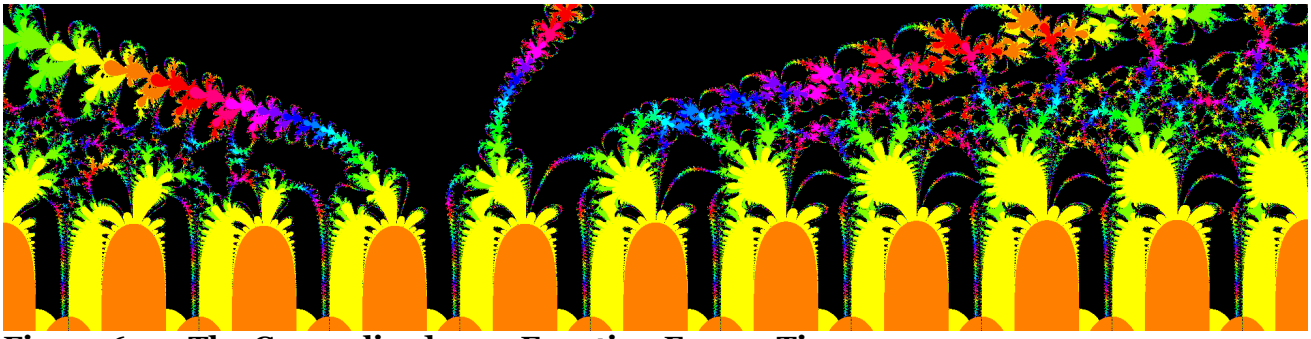


Figure 6.4.1 The Generalized 3*x*+1 Function Escape Time

6.5 Newton's Method in the Complex Plane

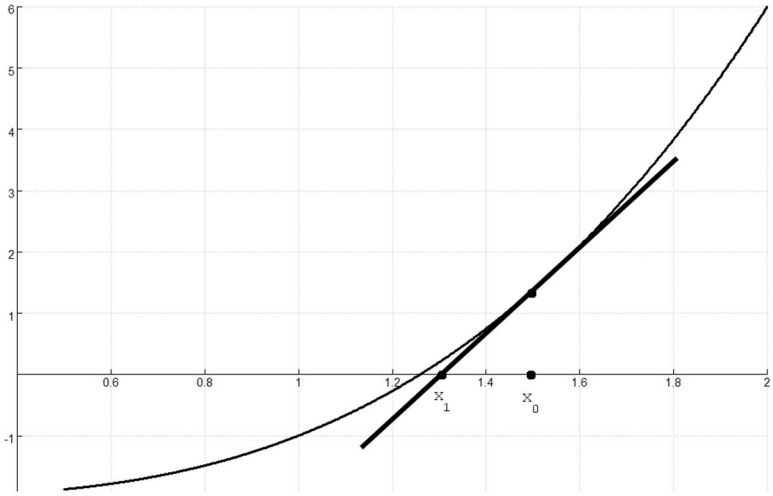


Figure 6.5.1 One Step of Newton's Method on a Real Valued Function

Newton's method is a technique for approximating (complex) roots of smooth functions. It uses tangent line approximations to obtain successive estimates for the roots. Locally it converges wonderfully for typical roots. That is, with an initial guess close to the root, it tends to converge quadratically, which means that the number of correct digits doubles for every iteration. Globally, the process is unpredictable. The set of initial points that converge to a specific root is called the basin of attraction of the root. The process is unpredictable in the sense that the basins of attraction for roots can be incredibly intertwined in intricate and complicated ways.

Newton's method for finding a root to $f(x)$ begins with an initial estimate x_0. Successive estimates are computed by the following formula.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The derivative of the function gives the slope of the tangent line to the function at the estimated root. If a function is smooth and the estimated root is close to the actual root, the root of the tangent line is a good choice for a better estimate of the root. That is precisely what Newton's method computes. Figure 6.5.1 shows a function and a tangent line at an initial estimate, x_0, along with x_1 which is the root of the tangent line and which gives the first Newton's method estimate.

Consider the function $x^5 - 2$ which has the fifth root of 2 as a root. It has four other complex roots. These five roots are arranged on a circle around the origin.

```
load '~addons/graphics/fvj4/complex_dynamics.ijs'

f=: _2: + ^&5                    the function
```

```
   newt=: - f % f D. 1            one step of Newton
```

In that construction `f D. 1` denotes the first derivative of `f`.

```
   (f D. 1) 1.2                   derivative at a point
10.368

   5*1.2^4                        derivative of f is 5x^4
10.368

   newt 1.2                       one step of the process
1.1529

   newt^:(i.5) 1.2                several iterates
1.2 1.1529 1.14873 1.1487 1.1487
```

The following function computes the sequence until an entry duplicates a previous entry. Recall from Section 4.7 that `x e. y` tests whether `x` is an item of `y`. We bound the number of iterates at 254 to avoid infinite loops.

```
   newton=: (,newt@{:)^:({: -.@e. }:)^:_

   newton 1.2
1.2 1.1529 1.14873 1.1487 1.1487 1.1487

   2^1r5                          we found the root
1.1487

      newton 0                    problem at 0
|NaN error: newt
|       newton 0
```

We may be interested in both the number of steps until convergence and which root is found. There are several reasons that errors may occur including the error above, so we give the number of steps as `3` and the basin as _ in that case.

```
   newton_sb=: (#,{:)@:newton"0 f. :: (3 _"_)"0

   newton_sb 1.2 1.5 0            results in number of steps and the basin (root)
6 1.1487
8 1.1487
3      _
```

Notice that for each possible initial point, the result is the number of iterations required for convergence and the last item, which is the root in these examples, or the error result. We can apply this to an input array of complex numbers as follows. We separate the number of iterations from the roots and truncate for display. The function `tru` defined below uses ***under*** `&.` monadically which post applies the inverse of its right adverb argument. That is, `u&.v y` is `v^:_1 u v y .`

```
   tru=: <.&.(1000&*)             truncation function

   tru 0.333333j0.777777          truncate a complex number
0.333j0.778
```

```
   <"2 tru 0 1|: newton_sb 3 zl_cccr 0 1.75
+--------+-------------------------------------+
|10 10 14| _0.93j0.675  0.355j1.092  0.355j1.092|
|26  3  9|       1.148                   1.148|
|10 10 14|_0.93j_0.675 0.355j_1.093 0.355j_1.093|
+--------+-------------------------------------+
```

Thus we see the number of steps and basin tables. The following expressions result in Figure 6.5.2. The data may take a minute to compute. The first step creates the number of steps and basin arrays. There are five roots. We will use the palette to distinguish the basins of attraction and to give some sense of the number of iterations used. For this example we need 5 colors for the 5 basins for the 5 roots, but we select 6 hues since there is a stray point, namely 0, which will be mapped to the sixth palette color. Then we repeat the palette reducing the color intensity each time. Decreases in intensity will correspond to increasing numbers of iterations (mod 3). Changes in color correspond to changes in roots. We encode information about the basin and the convergence time mod 3 in `bb` defined below. It is worthwhile to study this technique for paletting. It is a powerful way to put information about a process into an image.

```
   zs=: 512 zl_cccr 0 1.75

   's b'=: 0 1|: newton_sb zs

   B=: (~.,b)i. b                    basin number in nub of roots

   P6=: Hue (i.%])6                  six color palette

   P18=: <. ,/ 0.4 0.7 1 */P6        eighteen color striped palette

   B18=: 3 6#.0|:(3|s),:B            mix information from both arrays

   view_image P18;B18
```

While the above illustration is delightfully simple and works, there can be many other complications. Round-off and overflow errors are problematic, and both can cause errors for Newton's method, so our next example will serve as an illustration of some steps we can take to deal with those difficulties.

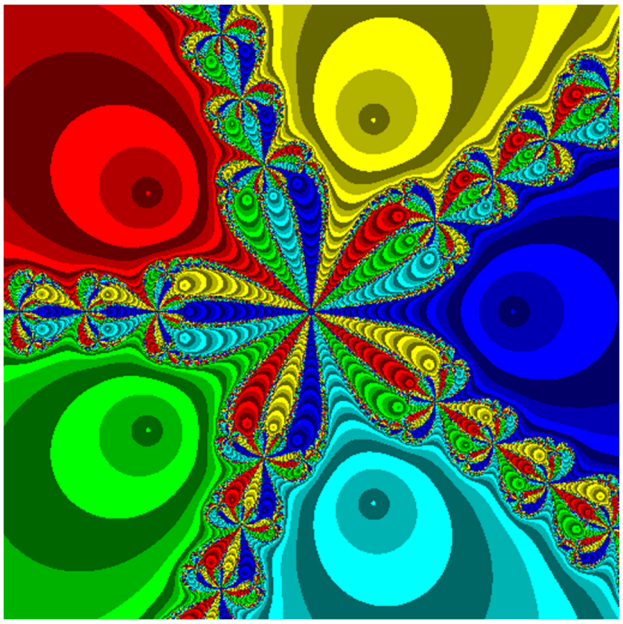


Figure 6.5.2 Newton's Method on z^5-2

We will use Newton's method to find the roots of the $3x+1$ function from Section 6.4. Note that `newton_sb` must be redefined since `f` has changed. Be forewarned that at this writing the main computation took around 3.5 hours.

```
   mod2=: -:@-.@^@(0j1p1&*)

   T=: [: -: mod2 + ] * 3: ^ mod2

   f=:T              the 3x+1 function

   sb=:newton_sb=: (#,{:)@:newton"0 f. :: (3 _"_)"0

   $'s b'=: 0 1|:  sb
2 257 513
```

Figure 6.5.3 Newton's Method on the 3*x*+1 Function

```
   |2 4{.b
4.91727e_10 6.30171e_11 3.77147e_12 1.45581e_10
9.35484e_11 1.91182e_10 1.91867e_12 1.38897e_10
```

There are many entries near zero, so we look to make them zero before we look more closely at the basins of attraction.

```
   zclean=: 1 : '(**|*u&<@|)&.+.'

   $'nub freq'=:|:({.,#)/.~ ,zb=:1e_8 zclean b
2 1678

   10{.\:~freq
44922 13918 12328 11879 11286 5827 2301 1317 1295 1239
```

We see there are over 1600 basins of attraction, but most occur very infrequently. We will limit our basins to the most frequent 6, plus a seventh category for everything else.

```
   rnub=:nub #~ freq>5000

   tru rnub
0 _0.488j_0.104 1.984j_0.309 3.991j_0.379 5.993j_0.417 7.995j_0.444
```

Notice one basin is attracted to 0, another with a small negative real part, (off the left of our image rectangle), and others near 2, 4, 6, and 8. As in the previous example, it is quite nice to use a palette that uses information both from the convergence time and the basin.

```
   B21=:3 7 #.0|:(3|s),:rnub i. zb

   P21=: <.,/0.4 0.7 1 */ 0,~Hue 0.7*(i.%<:)6

   view_image P21;B21
```

The result is shown in Figure 6.5.3. Notice the red portion is attracted to 0, the orange are attracted to the root with the negative real part, and the basins near 2, 4, 6, and 8 are fairly apparent. Black corresponds to low frequency basins, or errors in computation along the way.

6.6 Exercises

1. Create a Julia set for the function: (a) $f(z) = z^2 + 0.36 + 0.1i$, (b) $f(z) = z^2 + 0.41029 + 0.15345i$.

2. Write a function that creates and displays the Julia set for $f(z) = z^2 + c$ for a specified argument `c`.

3. Create an animation of the Julia sets corresponding to
(a) as `c` runs from `0.1j0.5` to `0.1j0.7` (b) as `c` runs from `_0.2j0.8` to `_0.2j1.0`.

4. Zoom into the region near the upper rightmost white region in the Mandelbrot set. Explore the Julia sets with constant values from one of the white bulbs of the Mandelbrot set. Create and compare that image of the Mandelbrot set with the corresponding Julia set.

5. Create an animation of the Mandelbrot set zooming in toward the center value `0.41029j0.15345` with width decreasing to `0.02`.

6. Write a function analogous to `mk_gjs` in Section 6.2 and use it to find 3 interesting Julia sets for functions of the form: (a) $c + z^2$ (b) $c + z^3$.

7. The Binet formula $F(z) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^z - \left(\frac{1-\sqrt{5}}{2}\right)^z\right)$ gives a complex generalization of the Fibonacci numbers.

(a) Implement the Binet formula and create an image that shows its escape time for $-6 \le \mathrm{Re}(z), \mathrm{Im}(z) \le 6$.

(b) Repeat (a) for $-0.5 \le \mathrm{Re}(z), \mathrm{Im}(z) \le 0.5$.

(c) Create an animation of the escape time for this function as you zoom in toward the origin.

8. Create an animation of a zoom into the center point 17 using the 3*x*+1 escape time algorithm from Section 6.4.

9. Create an image of 3*x*+1 escape time where $\mathrm{mod}_2(z) = \sin^2(\frac{\pi}{2}z)$.

10. The total stopping time for the 3*x*+1 function from is Section 6.4 is defined to be the number of steps required to iterate a positive integer until 1 is reached (and infinity if that doesn't happen in a finite number of steps). One way to generalize this to the complex function T(*x*) is to ask how many steps it takes until a value with magnitude strictly less than 2 is reached.
(a) create an image of the total stopping time using the function T(*x*) defined in Section 6.4.
(b) create an image of the total stopping time of the variant function T(*x*) using $\mathrm{mod}_2(z)$ from the previous exercise.

11. Create an adverb that creates images of the basins of attraction of Newton's method for the given function argument. Use the adverb to draw images of the basins of attraction for the following functions.

(a)	$z^3 - 1$	(b)	$z^4 - 1$	(c)	$z^5 - 1$	(d)	$z^6 - 1$
(e)	$z(z^4 - 1)$	(f)	$z(z^4 - 4)$	(g)	$z(z^4 - 0.25)$	(h)	$(z^5 - 1)(z^2 - 1)$

12. Create an image for the boundaries of attraction for $\sin(z)$ with $-3\pi \le \mathrm{Re}(z) \le 3\pi$ and $-\pi \le \mathrm{Im}(z) \le \pi$.

13. (a) Create an animation of the basins of attraction for Newton's method on $z^5 - 1$ as the region of interest zooms into the origin.
(b) Create an animation of the basins of attraction for Newton's method on $z^n - 1$ for *n* between 3 and 6, including fractional exponents.
(c) Create an animation of the basins of attraction for Newton's method on $\sin(z)$ as the region of interest zooms into $z = \pi/2$.

14. Implement Halley's method, which is described by the formula: $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n) - \dfrac{f''(x_n)f(x_n)}{2f'(x_n)}}$.

Like Newton's method, it can be used to approximate roots. Make an image of the basins of attraction for Halley's method applied to the following functions.

(a)	$z^3 - 1$	(b)	$z^4 - 1$	(c)	$z^5 - 1$	(d)	$z^6 - 1$
(e)	$z(z^4 - 1)$	(f)	$z(z^4 - 4)$	(g)	$z(z^4 - 0.25)$	(h)	$(z^5 - 1)(z^2 - 1)$

Chapter 7 Cellular Automata

Cellular automata give simple models for formalizing the notion of computation. They also give rise to intriguing behavior and are useful for applications. A cellular automaton (or simply an automaton) consists of an infinite array of cells, each cell is in one of a finite number of states at each time step; each cell evolves from step to step according to local rules. That is, the state of a cell at a subsequent time may only depend on some finite region around that point at the previous time step and the same rule is applied at each location. In practice, the arrays of cells are taken to be finite. We will first consider Boolean (2-state) automata, but we will also consider automata that use several discrete values or real values for their entries. In Part 2 we will see that several image processing techniques can be considered automata in this sense.

7.1 One Dimensional Automata

We begin by considering a one-dimensional Boolean automaton whose local rules involve just three cells: the cell and its left and right neighbors. There are eight possible configurations of the two states of those three cells. The particular automaton is specified by specifying the resulting output on each of the possible input triples. We consider one such automaton below. Recall that `#:` (***antibase***) is used to obtain other base representations; the inverse is `#.` (***base***) which gives the integer associated with a base representation.

```
   in=: 2 2 2 #: i. 8            three digit binary representations

   ]out=: (8#2)#:72              eight digit representation of 72
0 1 0 0 1 0 0 0

   (' in';'out'),:in;' ',.":,.out
+-----+---+
| in  |out|
+-----+---+
|0 0 0| 0 |
|0 0 1| 1 |
|0 1 0| 0 |
|0 1 1| 0 |
|1 0 0| 1 |
|1 0 1| 0 |
|1 1 0| 0 |
|1 1 1| 0 |
+-----+---+
```

Thus we see this automaton results in a `1` if and only if the list of the three relevant states is `0 0 1` or `1 0 0`. Since there are eight binary results, there are a total of 256 different rules of this type and we can associate each binary list of outputs with the number it represents in Boolean. In that way, this automaton can be referred to as Rule 72. All the functions in this section are defined by loading the following script.

```
   load '~addons/graphics/fvj4/automata.ijs'

   rule=: (8#2)&#:               loaded by automata.ijs

   rule 72
0 1 0 0 1 0 0 0
```

We saw in Section 3.4, where Spencer averaging was defined, that infixes could be used to apply a verb on all possible subintervals of a specified width. Thus, in order to apply the automaton, we first concentrate upon defining the local rule.

```
   lauto=: {~ #.                 local automaton
```

```
   (rule 72) lauto 1 1 0                    apply it to a triple
0

   (rule 72) lauto 1 0 0                    another triple
1
```

Now in order to apply the local rule to an entire array of states, we first periodically extend the array, and then apply infixes of width 3 using an adverb construction. While it is not necessary to extend the data, extending it is convenient for viewing the time evolution of the automaton since at each time step there will be the same number of cells. The data can be extended in a variety of ways other than periodically; for example, the boundaries can be extended in a constant or zero manner; however, the periodic extension provides a natural extension that tends to introduce fewer artifacts.

```
   perext=: {: , ] , {.

   perext 0 0 0 1 1 1                       periodically extend the list
1 0 0 0 1 1 1 0

   auto=: 4 : '(3: x&lauto\ perext) y'      automaton builder

   (rule 72) auto 0 0 0 1 1 1               Rule 72 on an input array
1 0 1 0 0 0

   ?.10#2                                   random data
0 0 0 0 1 0 0 0 0 1

   (rule 72) auto^:(i.10) ?. 10#2           Rule 72 iterated on random data
0 0 0 0 1 0 0 0 0 1
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
```

In order to see the evolution of an automaton, we will often want to iterate the automaton the same number of steps as the length of its input. The tacitly defined adverb `autoevo` accomplishes that evolution.

```
   autoevo=: auto ^:(i.@#@])                automaton evolution

   (rule 72) autoevo ?.10#2
0 0 0 0 1 0 0 0 0 1
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 1 0 0
1 0 0 1 0 1 0 0 1 0
```

We find it convenient to store these arrays as images with "super-pixels" for easy viewing. The function `spix` creates the super-pixels by copying each entry in a square pattern.

```
   spix=:[ # #"_1
```

```
   2 spix i.2 3                          two repetitions of each entry; 2 by 2 super-pixels
0 0 1 1 2 2
0 0 1 1 2 2
3 3 4 4 5 5
3 3 4 4 5 5

   wb=: 255,:0 0 0                       a white-black palette

   $b=: (rule 72) autoevo ?.64$2         evolution of 64 random values
64 64

   $6 spix b                             6 by 6 super-pixels
384 384

   view_image wb ; 6 spix b

   ((|.BW256);6 spix 255*b) write_image 'c:/temp/r072.png'
3232
```

We can use `view_image` in order view the image. In order to save the image as a file it is best to use a 256 color palette as illustrated. This assumes the directory *c:/temp/* exists. Figure 7.1.1 shows the result of six of the 256 automata of this type on the random initial configuration used above.

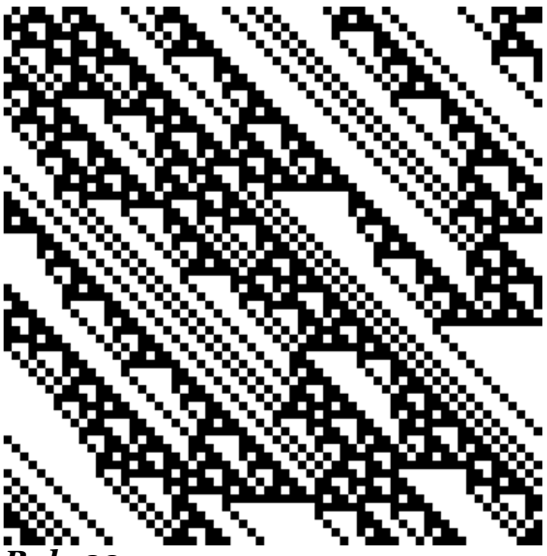
Notice that the automaton given by Rule 5 extinguishes rapidly. Rule 30 seems to have relatively complex behavior, Rule 72 has a region of chaotic triangles that extinguishes, Rule 102 appears to have chaotic triangles that synchronize at the last step, Rule 154 seems chaotic with an embedded region with structure and Rule 193 seems to generate different kinds of simple repeating behavior. With just 256 automata of this type it is fairly straightforward to create examples of them all as follows.

```
   path=: 'c:/temp/auto/'

   1!:5< path                            create path if necessary
1

   mk_auto=:3 : 0                        defined by automata.ijs
b=. (rule y) autoevo ?.64$2
((|.BW256);6 spix 255*b) write_image path,'auto', (nfmt y),'.png'
)

   mk_auto"0 i.256
1515 1529 1537 1551 1546 1552 1582 1606 2808 2796 3077...

   5 mk_html_gallery 192 192 'auto*.png' files_in path
34673

   open_html path,'index.html'
0
```

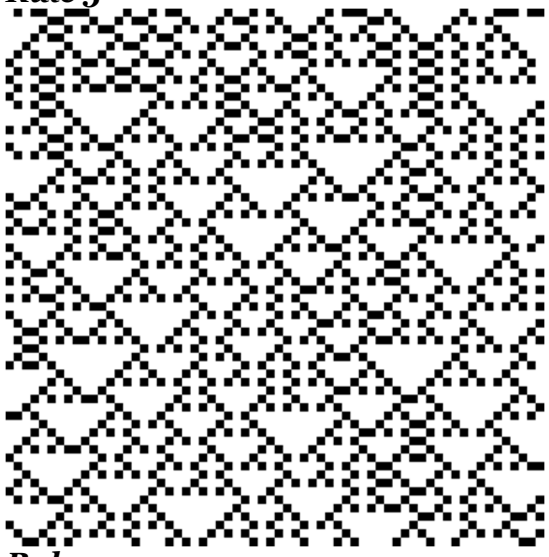
Clicking on an image will display a larger version of the image. Lastly, we remark that some authors use the reverse convention for rule numbers.

```
   rrule =: |.@rule

   rule 30
0 0 0 1 1 1 1 0

   rrule 120                             rule 30 is reverse rule 120
0 0 0 1 1 1 1 0
```

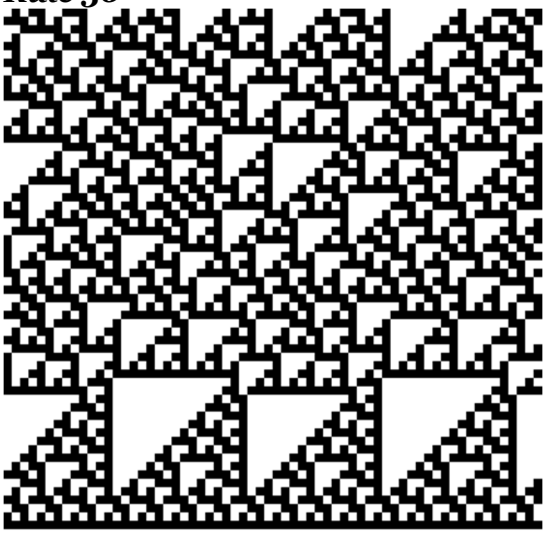
Rule 5



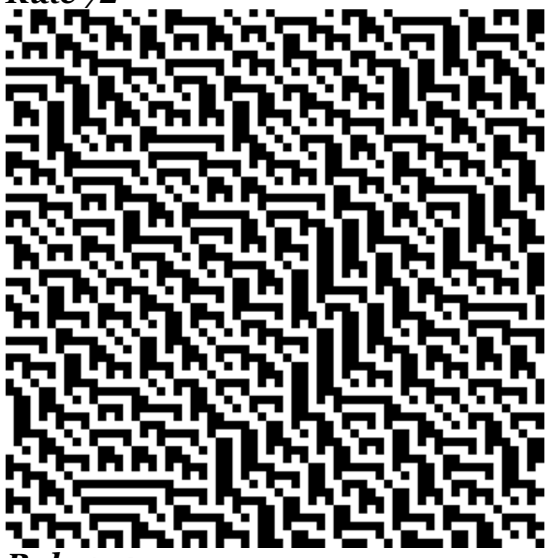
Rule 30



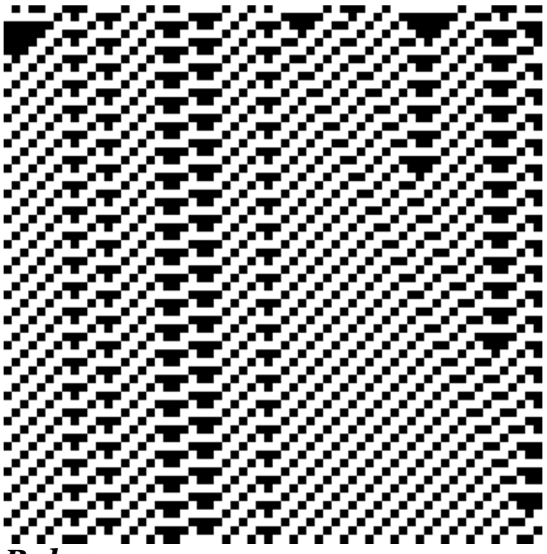
Rule 72



Rule 102



Rule 154



Rule 193

Figure 7.1.1 Evolution of Booleans Automata from Several Rules

7.2 Fuzzy Logic and Fuzzy Automata

Fuzzy logic has been developed in an attempt to quantify uncertainty as an aid to decision making. In general, one imagines values between 0 and 1 specifying the tendency toward truth and a level of confidence in that tendency. In this setting, a value of 0.9 might be interpreted as probably true. Since 0.5 is exactly between true and false, it would be completely ambiguous. There are many different models for fuzzy logic; some using only three possible states 0, 0.5 and 1. These are called trivalent fuzzy logics. Others use more values or can be viewed as functions on the interval [0,1]. In each case, one wants the functions giving logical "negation", "and" and "or" to be consistent with the Boolean case and one may want some additional properties to hold. Many book-size discussions of fuzzy logic are available; see [Dubois and Prade 1980] and [Klir et al 1997] in general and [Cattaneo et al 1997] for a discussion of fuzzy automata. However, the basic idea is very simple and we will see it is easy to generalize Boolean automata to fuzzy automata.

It is natural to take the logical negation of `x` to be `-. x`, even when `x` has a fuzzy value. However, there seem to be a great variety of choices for analogs to "or" and "and". In this section we will consider two choices for fuzzy "or" and "and" functions defined on the interval [0,1] and create fuzzy automata with them. First recall the Boolean "or" and "and" truth tables. We then will consider the max/min and the probabilistic fuzzy logics. All the functions defined in this section are loaded by the script.

```
   load '~addons/graphics/fvj4/automata.ijs'
```

```
   +. table i.2
```
Boolean "or" table

```
+--+---+
|+.|0 1|
+--+---+
|0 |0 1|
|1 |1 1|
+--+---+
```

```
   *. table i.2
```
Boolean "and" table

```
+--+---+
|*.|0 1|
+--+---+
|0 |0 0|
|1 |0 1|
+--+---+
```

```
   >. table (i.%<:)5
```
maximum generalizes "or"

```
+----+--------------------+
|>.  |   0 0.25  0.5 0.75 1|
+----+--------------------+
|   0|   0 0.25  0.5 0.75 1|
|0.25|0.25 0.25  0.5 0.75 1|
| 0.5| 0.5  0.5  0.5 0.75 1|
|0.75|0.75 0.75 0.75 0.75 1|
|   1|   1    1    1    1 1|
+----+--------------------+
```

```
   <. table (i.%<:)5
```
minimum generalizes "and"

```
+----+--------------------+
|<.  |0 0.25  0.5 0.75    1|
+----+--------------------+
|   0|0    0    0    0    0|
|0.25|0 0.25 0.25 0.25 0.25|
| 0.5|0 0.25  0.5  0.5  0.5|
|0.75|0 0.25  0.5 0.75 0.75|
|   1|0 0.25  0.5 0.75    1|
+----+--------------------+
```

The probabilistic generalization of "or" is the sum minus the product: `x+y-x*y`. The generalization of "and" is the ordinary product.

```
   por=: (+ - *)"0          sum minus the product; loaded by automata.ijs

   pand=: *                 product


   por table (i.%<:)5                the function por generalizes "or"
+----+-------------------------+
|por |   0   0.25   0.5   0.75 1|
+----+-------------------------+
|   0|   0   0.25   0.5   0.75 1|
|0.25|0.25 0.4375 0.625 0.8125 1|
| 0.5| 0.5  0.625  0.75  0.875 1|
|0.75|0.75 0.8125 0.875 0.9375 1|
|   1|   1      1     1      1 1|
+----+-------------------------+

   pand table (i.%<:)5               the function pand generalizes "and"
+----+-------------------------+
|pand|0   0.25   0.5   0.75    1|
+----+-------------------------+
|   0|0      0     0      0    0|
|0.25|0 0.0625 0.125 0.1875 0.25|
| 0.5|0  0.125  0.25  0.375  0.5|
|0.75|0 0.1875 0.375 0.5625 0.75|
|   1|0   0.25   0.5   0.75    1|
+----+-------------------------+
```

Consider the following Boolean expression and its generalizations to slightly uncertain truth values using the probabilistic fuzzy logic.

```
   x=: 0                          false

   y=: 1                          true

   (x +. y) *. (x +. -.x)         the Boolean expression is true
1

   x=: 0.1                        probably false

   y=: 0.9                        probably true

   (x por y) pand (x por -.x)     probably true; but less certain
0.8281
```

Now we are prepared to discuss fuzzy generalizations to the Boolean automata that we discussed in the previous section. Recall the local definition of Rule 72. Using `in` and `out` from that section, we see that the Boolean rule results in truth exactly in the two situations where the three input states are `0 0 1` or `1 0 0`.

```
   out # in                       inputs resulting in truth
0 0 1
1 0 0
```

If we imagine the three Boolean input states to be named `x`, `y` and `z`, then the result of the Rule 72 Boolean automaton is given by the following formula.

```
   ((-.x)*.(-.y)*.z)+.(x*.(-.y)*.(-.z))
```

This is a disjunctive normal form for the Boolean formula defining the automaton. Notice we need to take the "or" insert of the "and" insert of either truth values or their negation. The `out#in` array shown above has a zero in positions corresponding to the need for negation. It turns out that the dyad `|@-` is useful for implementing the desired computation. Notice that `0|@-y` is the same as `y` while `1|@-y` is the same as `1-y`. Thus, we can create a suitable array of input values or their negation by using `out#in` as a left argument. Then we need only take the "or" insert of the "and" insert of the result. We can readily use fuzzy logical functions in place of "or" and "and". The following conjunction takes the sense of "or" and "and" as its conjunction arguments; the rule to be used is the left argument of the derived function. Its right argument is the array of states.

```
   lfauto=: 2 : 0                          local fuzzy automaton;
:                                          defined by automata.ijs
u/ (-.x##:i.8) v/ . (|@-) y
)

   (rule 72) +. lfauto *. 0 1 1            traditional rule 72 on 0 1 1
0

   (rule 72) +. lfauto *. 0 0 1            traditional rule 72 on 0 0 1
1

  (rule 72) por lfauto pand 0.1 0.1 0.9    fuzzy rule 72
0.731439
```

Now we can generalize this to global behavior by using the size 3 infixes.

```
   perext=: {: , ] , {.                    periodic extension

   fauto=: 2 : 0                           fuzzy automaton
:
3 (x&(u lfauto v)\) perext y
)

   fautoevo=: 2 : 0                        fuzzy automaton evolution
:
x(u fauto v)^:(i.@#@]) y
)

   (rule 72) +. fautoevo *. ?.5#2          traditional rule 72
0 0 0 0 1
1 0 0 1 0
0 1 1 0 0
1 0 0 1 0
0 1 1 0 0
```

Next we implement the probabilistic Rule 72 on a fuzzy, but nearby argument.

```
   8j5 ": (rule 72) por fautoevo pand (?.5$2){0.01 0.99
0.01000 0.01000 0.01000 0.01000 0.99000
 0.97030 0.01951 0.01951 0.97030 0.00020
 0.00058 0.93286 0.93286 0.00058 0.05679
 0.87982 0.06260 0.06260 0.87982 0.00110
 0.00764 0.77471 0.77471 0.00764 0.20008
```

Notice that the probabilistic rule shows a great deal of similarity to the traditional rule. This is true only for a short time in general, but is apparent in that example. We used ***format*** `": ` with a left argument to

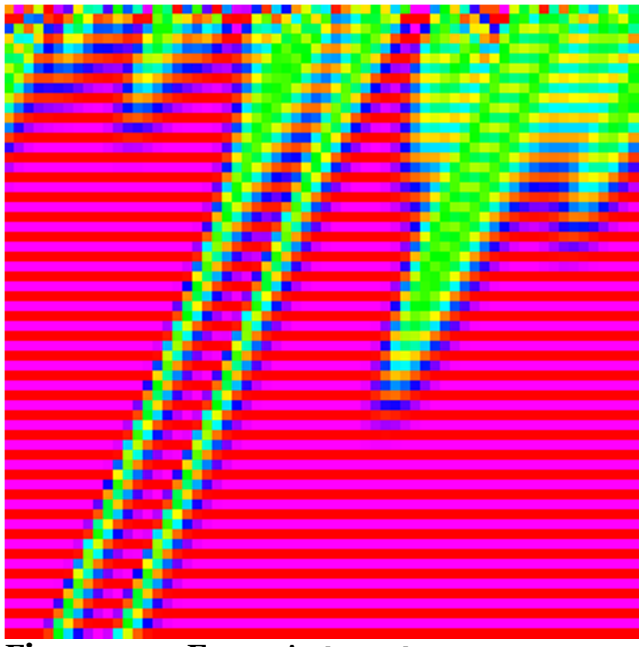


Figure 7.2.1 Fuzzy Automata: Probabilistic Rule 136

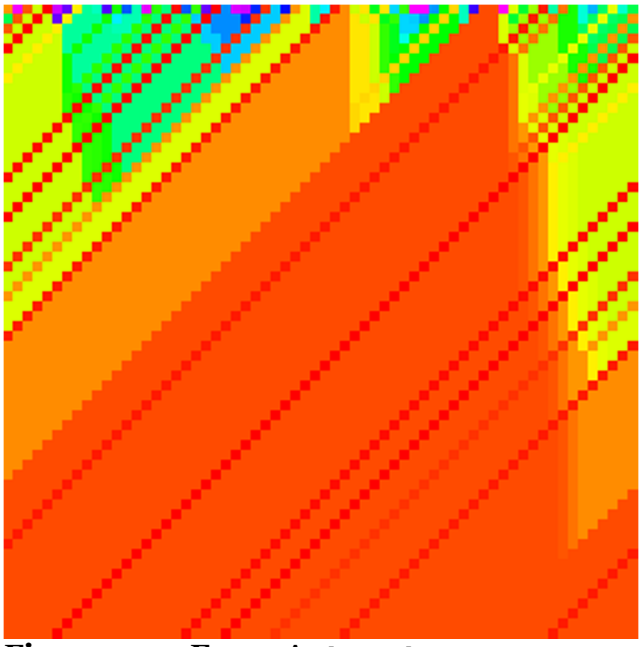


Figure 7.2.2 Fuzzy Automata: Min/Max Rule 21

align the decimal points in the result. The left argument `8j5` specifies a format field width of 8 with 5 places to the right of the decimal point.

In order to view the time evolution of these fuzzy automata, we will map the result to discrete values, use our `P256` palette from *raster.ijs* which is loaded by *automata.ijs*, and create a bitmap. The function `lin256` defined below maps fuzzy values to the integers between 0 and 255. The array `b` gives the time evolution of the automaton. Finally, we create an image file with pixels replicated for easy viewing using `spix` from Section 7.1. Here we look at rule 136 with the `por` and `pand` fuzzy functions.

```
   lin256=: <.@(255.99&*)

   b=: lin256 (rule 136) por fautoevo pand 0.01*?.64$101

   view_image P256; 6 spix b
```

Figure 7.2.1 shows that image and Figure 7.2.2 shows the image resulting from Rule 21 using min/max logical functions. It is easy to build a utility function that makes it easy to make a gallery of all the (nonzero) rules for any fuzzy logical system. This is defined in *automata.ijs*.

```
   path=:'c:/temp/fz/'              be sure to create the directory if necessary

   mk_fz_auto=:3 : 0
b=: 6 spix lin256 (rule y) por fautoevo pand 0.01*?.64$101
(P256;b) write_image path,'p',(nfmt y),'.png'
)

   mk_fz_auto"0 >:i.255
1988 2121 2225 2180 2326 2356 2513 6447 6782 8264...

   (path,'index_p.html')5 mk_html_gallery 192 192 'p*.png' files_in path
32238

   open_html path,'index_p.html'
```

7.3 Experiment: The Game of Life

The Game of Life is a 2-dimensional automaton that updates a Boolean array of cells according to simple rules. For an early discussion of the automaton, see [Berlekamp, Conway, Guy, 1982]. Its remarkable, complex and rich behavior makes it a popular programming exercise and, despite its almost random behavior, has led to serious investigations of its behavior. In particular, it is known to be a universal computer. While amazing implementations may readily be found on the web, we will offer a straightforward implementation and consider a few examples.

The new state of each cell is determined by the 3 by 3 neighborhood surrounding the cell. If the cell's current state is 1 and exactly two or three of its eight neighbors are also ones, then the cell's new state is 1. Also, if the cell's current state is 0 and exactly three of its eight neighbors are ones, then the cell's new state is 1. Otherwise, the cell's new state is 0.

We implement the local rule by multiplying a 3 by 3 Boolean array by a mask matrix that has 9 in the center and ones everywhere else. The sum of the products determines whether a cell will be alive at the next step; it will be if the result is 3, 9+2, or 9+3; that is 3, 11 or 12. We can test membership in a list using `e.` as illustrated below.

```
   ]L=: 1,1 9 1,:1                        the mask for local life
1 1 1
1 9 1
1 1 1

   3 e. 3 11 12                           it is in the list
1

   10 e. 3 11 12                          it is not in the list
0

   llife=: +/@:,@(L&*) e. 3 11 12"_       local life

   ]n0=: 0 1 1,0 0 0,:0 0 0               a neighborhood
0 1 1
0 0 0
0 0 0

   llife n0                               local life on the neighborhood
0

   ]n1=: 0 1 1,0 1 0,:0 0 0               another neighborhood
0 1 1
0 1 0
0 0 0

   llife n1                               local life on the neighborhood
1

   ]n2=: 0 1 1,0 1 0,:1 0 0               another neighborhood
0 1 1
0 1 0
1 0 0

   llife n2                               local life on the neighborhood
1
```

As in Section 4.7, where we computed fractal dimension by counting the number of zero and nonzero tesselations, it is convenient to use `_3` cut in order to apply the local rule to tesselations. The input array will be extended periodically in two dimensions before we apply the local life rule to the tesselation.

```
   perext=: {: , ] , {.
```

```
   perext2=: perext"1@:perext          2-dimensional periodic extension
   perext2 n2
0 1 0 0 1
1 0 1 1 0
0 0 1 0 0
0 1 0 0 1
1 0 1 1 0
   3 3 <;._3 perext2 n2
+-----+-----+-----+
|0 1 0|1 0 0|0 0 1|
|1 0 1|0 1 1|1 1 0|
|0 0 1|0 1 0|1 0 0|
+-----+-----+-----+
|1 0 1|0 1 1|1 1 0|
|0 0 1|0 1 0|1 0 0|
|0 1 0|1 0 0|0 0 1|
+-----+-----+-----+
|0 0 1|0 1 0|1 0 0|
|0 1 0|1 0 0|0 0 1|
|1 0 1|0 1 1|1 1 0|
+-----+-----+-----+
   life=: 3 3&(llife;._3)@ perext2
   life n2
0 1 1
0 1 0
1 0 0
```

In order to investigate more complex configurations, we define an adverb that visually displays the time evolution of the Game of Life in an animation.

```
   load '~addons/graphics/fvj4/automata.ijs'
   spix=: [ # #"_1
   show_auto=: 1 : 0                   defined by automata.ijs
100 0.5 u show_auto y
:
'maxit delay'=.x
m0=.<.<./VRAWH%|.$y
VRAWH=:m0*|.$y
vwin 'auto'
y=.|.y
for. i. maxit do.
  VRA=:m0 spix <.255*y
  vfshow ''
  y=.u y
  6!:3 delay
end.
)
```

The use of `y=.|.y` in the definition of `show_auto` corresponds to the fact that the coordinates for plotting polygons in `dwin` and the coordinates for indexing matrices are different. Using `show_auto`, we can watch the evolution of the Game of Life on a random initial configuration.

```
   200 0.2 life show_auto ?.100 100$2
```

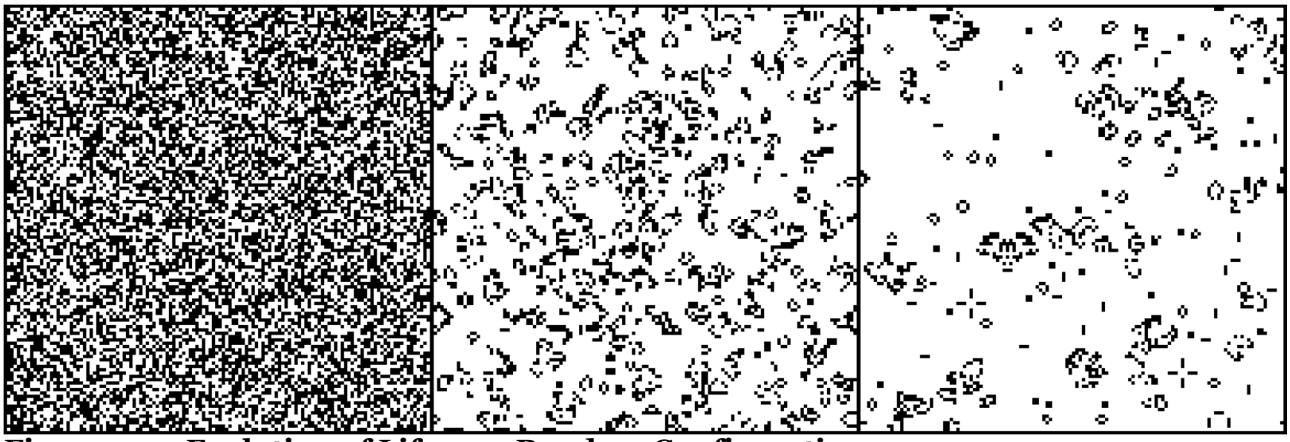


Figure 7.3.1 Evolution of Life on a Random Configuration

The left argument of `show_auto` specifies the number of iterations and the delay in seconds between frames. Figure 7.3.1 shows that random initial configuration and the configuration after 50 and 200 iterations. Notice the evolution simplifies the configurations yet they remain complex. While it is not apparent from our figures, during the animation you will probably notice stable and periodic configurations and you might notice there seems to be occasional local four-fold symmetry in the configurations that are an artifact of the square lattice on which the Game of Life is defined.

Next we look at some special configurations.

```
load '~addons/graphics/fvj4/life_ex2.ijs'
life show_auto A1                    view configuration A1

151 0.2 life show_auto A2            view configuration A2
```

The configurations in `A1` illustrate some stable patterns, some periodic patterns (periods 2 and 6) and a glider which replicates itself translated until it interferes with other configurations. The remarkable configuration in `A2` gives rise to an infinite stream (if there is no interference) of gliders. The configuration with five gliders visible is shown in Figure 7.3.2.

As we have noted, there are many implementations of the Game of life. We mention two other J implementations. First is Ewart Shaw's signature that he uses on the J forum. It is a one line definition of the algorithm and its application to an initial configuration.

```
   3 ((4&({*.(=+/))++/=3:)@([:,/0&,^:(i.3)@|:"2^:2))&.>@]^:(i.@[) <#:3 6 2
+-----+---------+-------------+
|0 1 1|0 0 0 0 0|0 0 0 0 0 0 0|
|1 1 0|0 1 1 1 0|0 0 0 1 0 0 0|
|0 1 0|0 1 0 0 0|0 0 1 1 0 0 0|
|     |0 1 1 0 0|0 1 0 0 1 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 0 0 0|
+-----+---------+-------------+
```

Second, we mention the script *vlife.ijs* implements life with a GUI and shows each alive cell according to how many neighbors it has. It can be started via the folowing.

```
load '~addons/graphics/fvj4/vlife.ijs'
```

Figure 7.3.2 Evolution of Life on a Glider Gun

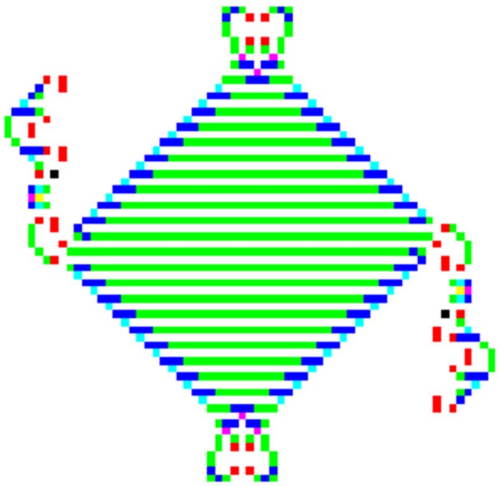


Figure 7.3.3 Evolution of Life on *max.lif* using Color

The evolution of the *max.lif* pattern from Al Hensel's pattern collection, *lifep.zip*, is shown in Figure 7.3.3.

7.4 Majority Rule and Spot Formation

While the Game of Life is an intriguing example of complex behavior, it is primarily a recreational example. However, 2-dimensional automata are capable of modeling many more realistic behaviors and in Chapter 8 we will see that many image processing techniques can be viewed as a type of cellular automata.

In this section we look at majority rule automata that can simulate spot formation. We begin with a simple example: the majority rule on 3 by 3 neighborhoods. We assume that *automata.ijs* has been loaded.

When applied to a Boolean input, the function tests whether more than half of the entries are ones. Then we use the same 3 by 3 tessellations that we used for Life in order to apply the majority rule locally throughout the array. The first image in Figure 7.4.1 shows the result of that process on a random 200 by 200 array with periodic boundary conditions.

```
   lmajor=: (+/ > -:@#)@:,
```

defined by *automata.ijs*

```
   ]n=: 1 1 0,1 0 0,:0 1 1
1 1 0
1 0 0
0 1 1

   lmajor n
1

   major=: 3 3&(lmajor;._3)@ perext2

   VRAWH=:720 720

   20 0.5 major show_auto ?.200 200$2
0.5
```

Figure 7.4.1 shows the result after 5, 20 and 150 iterations. Next, we see we can look at 5 by 5 neighborhoods but first need to enhance our ability to extend periodically more than one cell.

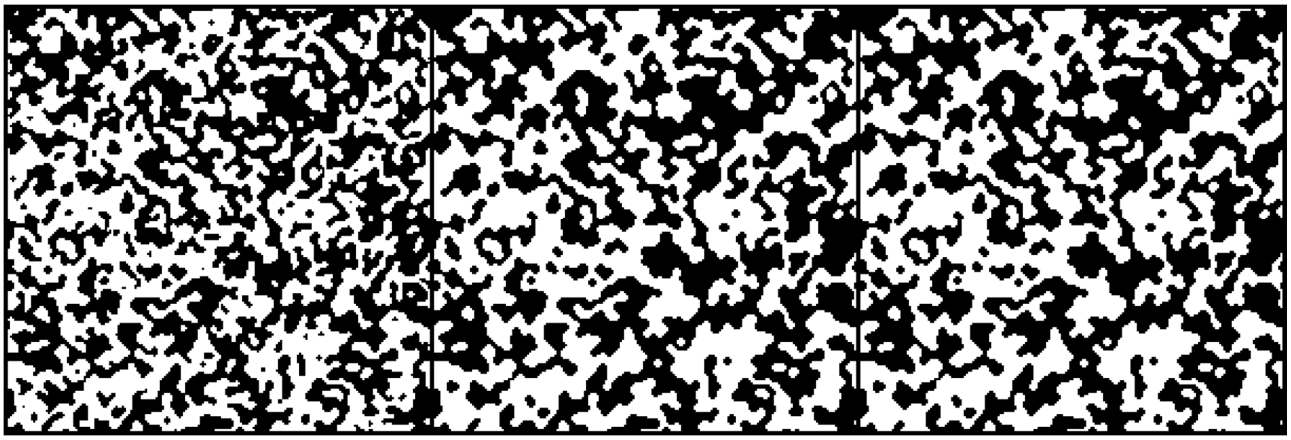


Figure 7.4.1 Evolution of a 3 by 3 Majority Rule

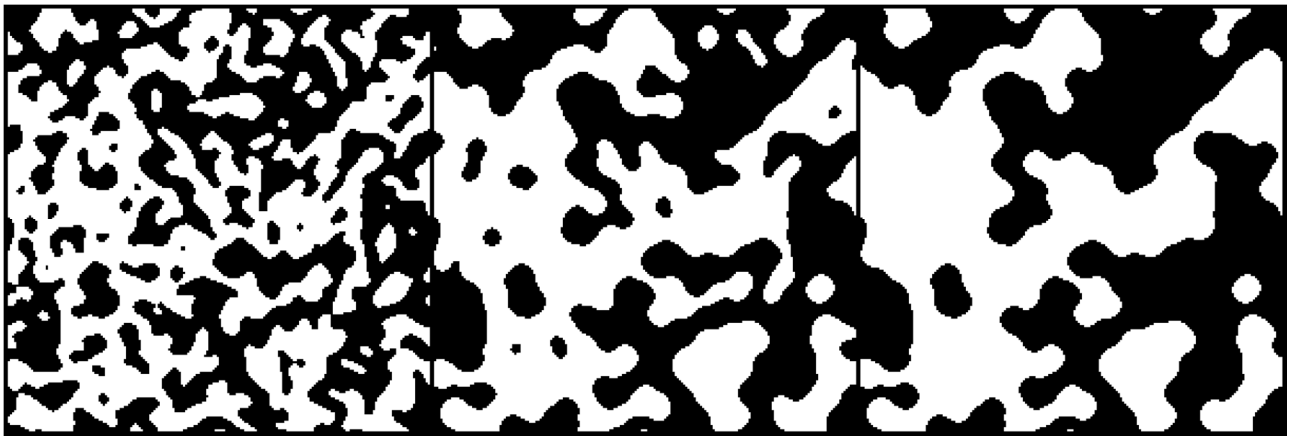


Figure 7.4.2 Evolution of a 5 by 5 Majority Rule

```
   nperext=: 1 : '(-m)&{. , ] , m&{.'

   nperext2=: 1 : '({:m) nperext"1@:(({.m) nperext)'

   2 nperext2 i.4 5
13 14 10 11 12 13 14 10 11
18 19 15 16 17 18 19 15 16
 3  4  0  1  2  3  4  0  1
 8  9  5  6  7  8  9  5  6
13 14 10 11 12 13 14 10 11
18 19 15 16 17 18 19 15 16
 3  4  0  1  2  3  4  0  1
 8  9  5  6  7  8  9  5  6
```

Now we can see the result of the majority rule on larger neighborhoods and, by selecting the middle three rows of a 5 by 5 neighborhood, we can bias the direction of the edges that eventually form. Iterations 5, 20 and 150 for experiments appear in Figures 7.4.2 and Figure 7.4.3.

```
   major2=:5 5&(lmajor;._3)@(2 nperext2)

   150 0.3 major2 show_auto ?.200 200$2

   major3=:5 5&(lmajor@(1 2 3&{);._3)@(2 nperext2)

   150 0.3 major3 show_auto ?.200 200$2
```

Many other variants on the majority rule can easily be tried. One can multiply the neighborhoods by weights, and sum the results, and result in one when some threshold is passed. Negative weights can be used. Also, twisted rules, can be considered. For example, one could count the number of ones in a 3 by

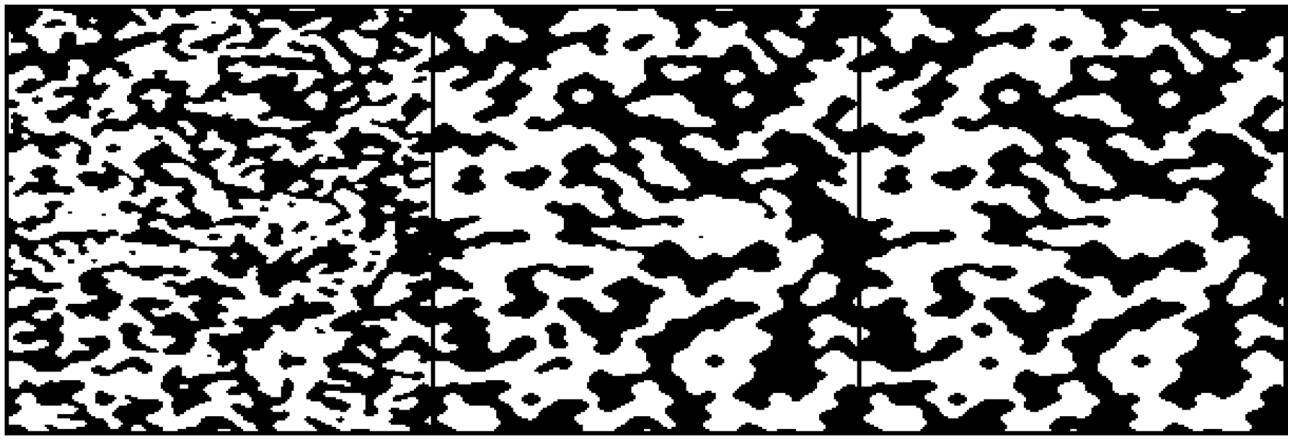


Figure 7.4.3 Evolution of a Biased Majority Rule

3 neighborhood, and result in a one when the sum is 4 or greater than 5. Also, we need not limit ourselves to Boolean arrays.

7.5 Cyclic Cellular Automata

The basic idea of the rule is that each cell is in one of several possible states 0, 1, ..., *N-1* and at the next step a cell increases its state by 1 (mod *N*) is it has a neighbor in that state. As an example, we consider a 17 state automata with Moore neighborhoods. In each 3 by 3 patch the center has index 4 after the patch is raveled and the other 8 entries are neighbors.

```
   ns=:17

   cen=:4

   nei=:(i.9)-.cen

   lcca=:(ns|cen&{ + (ns|1+cen&{)e. nei&{)@,

   cca=:3 3&( lcca;._3)@perext2

   ]a=:?.5 5$ns
10  3  8 13  8
16  1  6 14  2
 5  4 15 13  4
 2  3 13 13  0
 2 14  5 12 12

   cca a
10  3  8 14  8
16  1  6 15  2
 5  5 15 14  5
 3  4 14 13  0
 3 14  5 13 13
```

Notice that in the first row no entries changed, but the 14 in the second row became a 15 because there was a 15 to the southwest of the 14. We now run 300 steps of the automata on a random initial configuration. The last frame is then shown with a slightly nicer palette. Notice that to be compatible with the range expectations of `show_auto`, we divide each entry by the number of states and run `cca` under multiplication by the number of states.

```
   VRAWH=:600 600
```

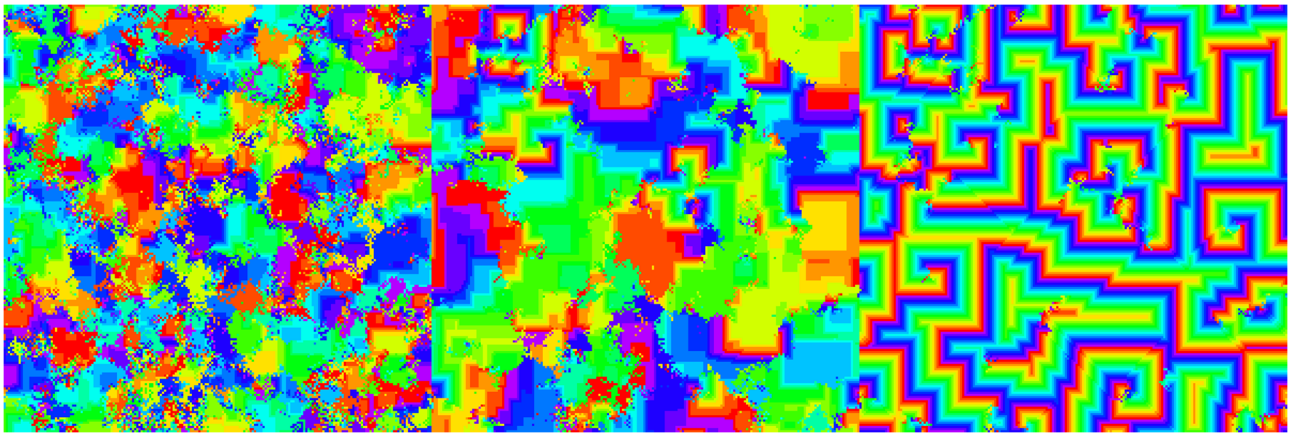


Figure 7.5.1 Evolution of a Moore Neighborhood Cyclic Cellular Automaton

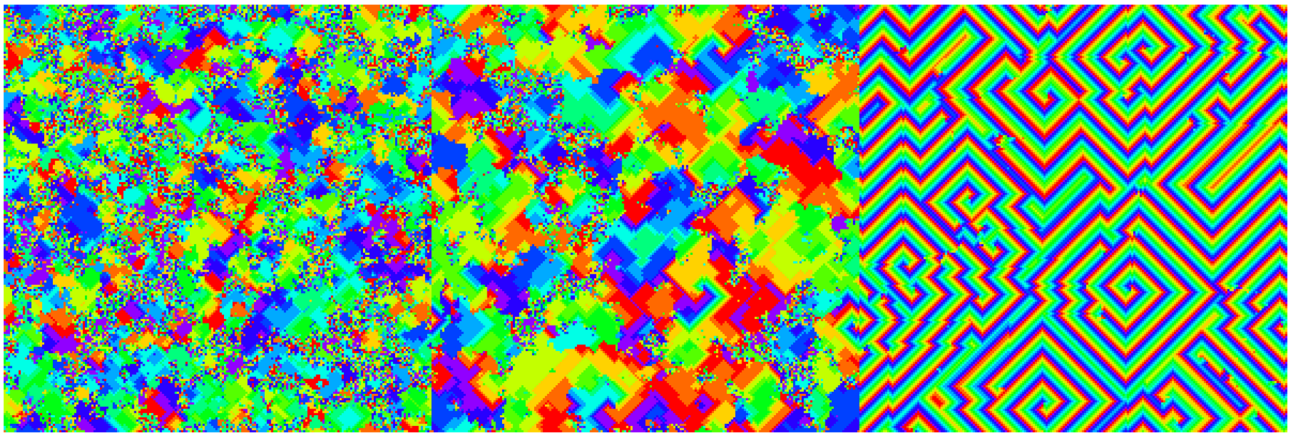


Figure 7.5.2 Evolution of Von Neumann Neighborhood Cyclic Cellular Automaton

```
   300 0.01 cca&.(*&ns) show_auto %&ns ?.200 200$ns
0.01

   view_image P256;|.VRA
```

Notice that droplets of one color form and that eventually self-organizing spirals dominate. Figure 7.5.1 shows this automaton after 40, 60 and 300 iterations. We next consider Von Neumann neighborhoods with 12 states.

```
   ns=:12

   nei=:1 3 5 7

   lcca=:(ns|cen&{ + (ns|1+cen&{)e. nei&{)@,

   200 0.01 cca&.(*&ns) show_auto %&ns ?.200 200$ns
```

Again we see that the automaton self-organizes into spirals, although these are oriented differently. Figure 7.5.2 shows the configurations after 40, 65, and 200 iterations. If we consider an asymmetric neighborhood pattern `nei=:0 1 5 7` with `ns=:8` we still observe spirals self-organize. Figure 7.5.3 shows that automaton after 10, 17 and 60 iterations.

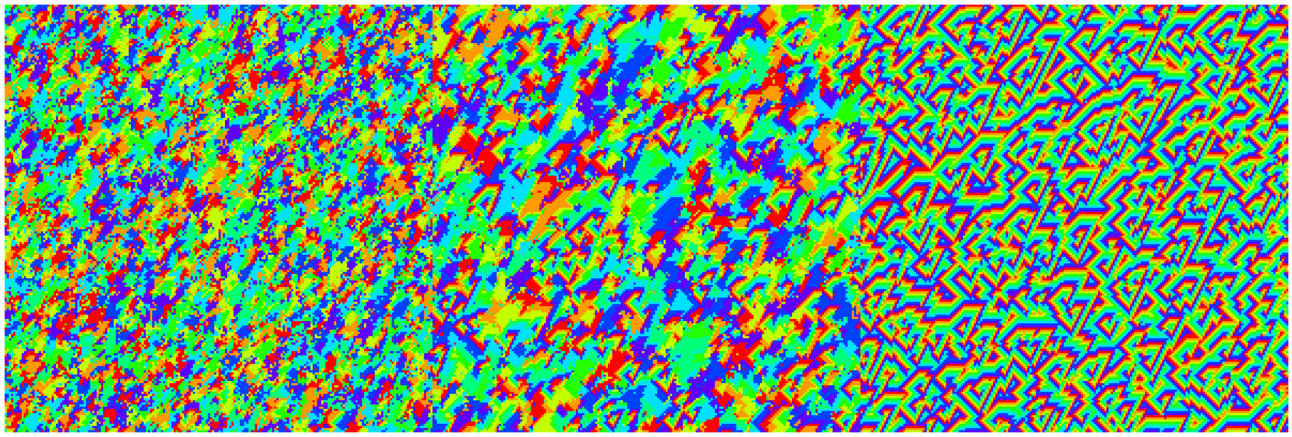


Figure 7.5.3 Evolution of a Asymmetric Neighborhood Cyclic Cellular Automaton

7.6 Experiment: The Hodgepodge Rule

In this section we look at rule that is of a type used to model the spread of illnesses and which is similar to rules used to describe chemical reactions involving catalysts. While this Hodgepodge rule is traditionally viewed as being based upon cells in one of many states (say 100), we will look at a slight variant that uses fuzzy values for the intermediate states.

The basic idea of the rule is that each cell is either healthy (value 0), ill (value 1) or infected (intermediate between 0 and 1). A healthy cell becomes infected if a threshold of ill or infected neighbors occur. An infected cell, moves toward being ill by a fixed amount from an average of its neighbors. An ill cell becomes healthy on the next generation. We will apply these rules on 3 by 3 neighborhoods, but we will ravel them before processing, so we can view them as length 9 vectors. Below we generate a matrix `X` that contains 3 neighborhoods, one in each row, containing one each of healthy, infected and ill center cells. The function `case` determines state of the center cell in each neighborhood, resulting in 0, 1, or 2 depending whether the cell is healthy, infected or ill.

```
   ]X=:(t=1)>.(0=t=:?.3 9$5)*(?.3 9$0)
0        1 0 1 0.0583756 0 1 1        1
0        0 0 0         1 0 0 0 0.119496
0 0.392216 0 0         0 0 0 0        0

   case=: (~:&0 + =&1)@(4&{)

   case 0{X            infected
1

   case 1{X            ill
2

   case 2{X            healthy
0
```

The evolution of the rule depends upon four parameters, `a`, `b`, `c` and `N`. Healthy cells become infected if the number of infected or ill neighbors passes a threshold, the level of infection is determined by the number of times the thresholds are exceeded, added together and scaled back by `N` to remain between `0` and `1`.

```
   ill=: #~ =&1

   inf=: #~ ~:&0 *. ~:&1

   ill 0{X             ill part of the neighborhood
1 1 1 1 1
```

```
   inf 0{X              infected part of the neighborhood
0.0583756

   'a b c N'=.2 3 0.17 100

   forh1=. <.@:(*&(%a))@#@ill

   forh2=. <.@:(*&(%b))@#@inf

   forhea =. [: %&N forh1 + forh2

   forhea 0{nei              this healthy cell stays healthy
0
```

Next, infected cells tend to increase in level of infection, toward illness, as the average of infected neighbors plus an additional infection amount, `c`, and the total not exceeding one.

```
   avg=:+/ % #

   forinf=: 1 <. avg@inf + c"_

   forinf 2{nei
0.228376
```

The ill cells immediately revert to healthy. We can put all of this into an adverb for creating a local Hodgepodge rule.

```
   lhodge=: 1 : 0           defined by automata.ijs
'a b c N'=.m
ill=. #~ =&1
inf=. #~ ~:&0 *. ~:&1
case=. (~:&0 + =&1)@(4&{)
avg=.+/ % #
forh1=. <.@:(*&(%a))@#@ill
forh2=. <.@:(*&(%b))@#@inf
forhea =. [: %&N forh1 + forh2
forinf=. 1 <. avg@inf + c"_
forill=. 0:
forhea`forinf`forill@.case@:, f.
)

   2 3 0.17 100 lhodge 0{X
0.228376

   2 3 0.17 100 lhodge 1{X
0

   2 3 0.17 100 lhodge 2{X
0
```

Now we can make this into a global rule and use `show_auto` to display the evolution on a random 200 by 200 array of cells.

```
   VRAWH=:600 600

   b=: (c=1)>.(0=c=:?.200 200$20)*(?.200 200$0)

   hodge=: 1 : '3 3&(m lhodge;._3)@ perext2'
```

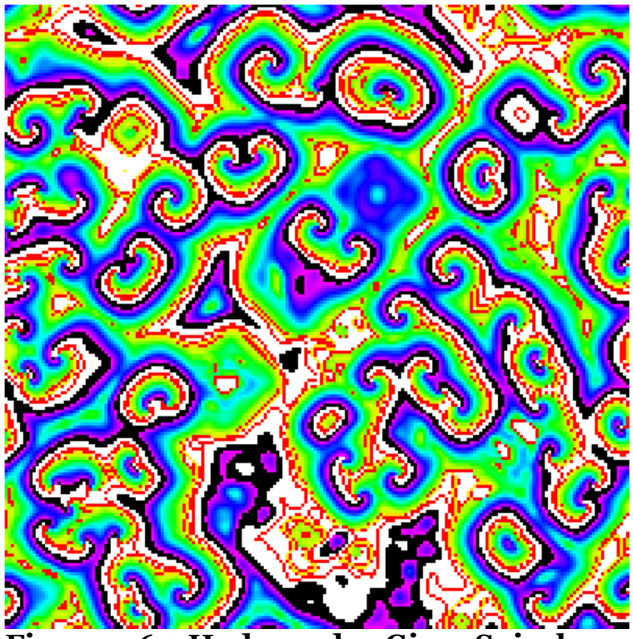


Figure 7.6.1 Hodgepodge Gives Spirals

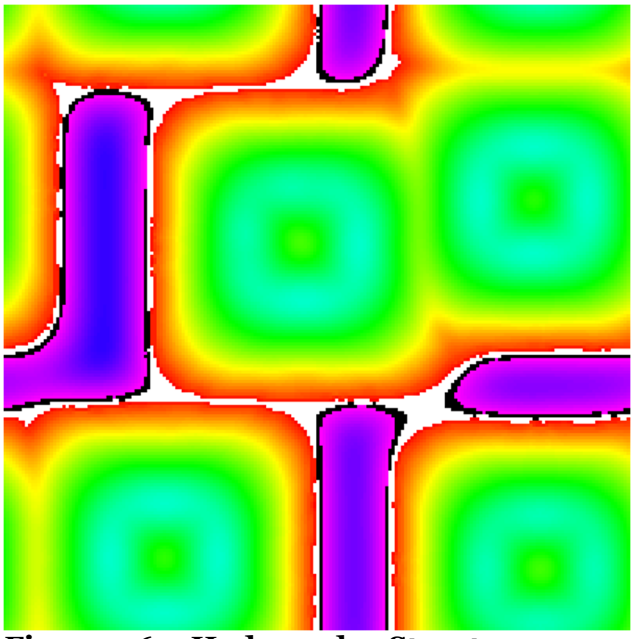


Figure 7.6.2 Hodgepodge Structures

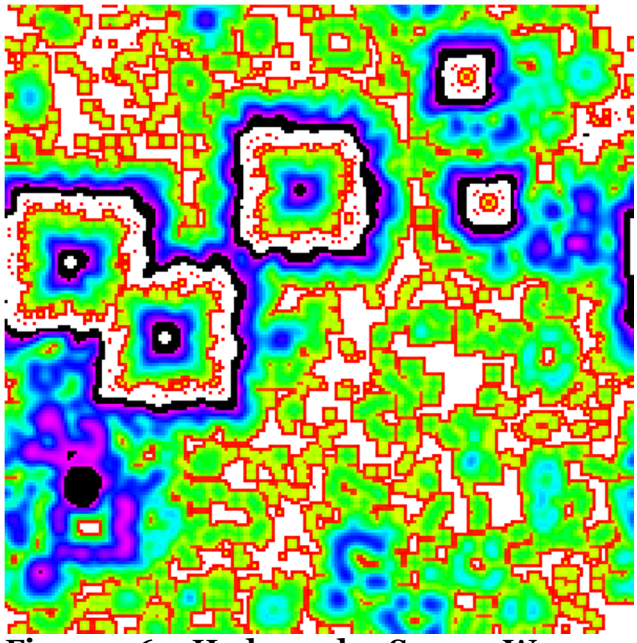


Figure 7.6.3 Hodgepodge Square Waves

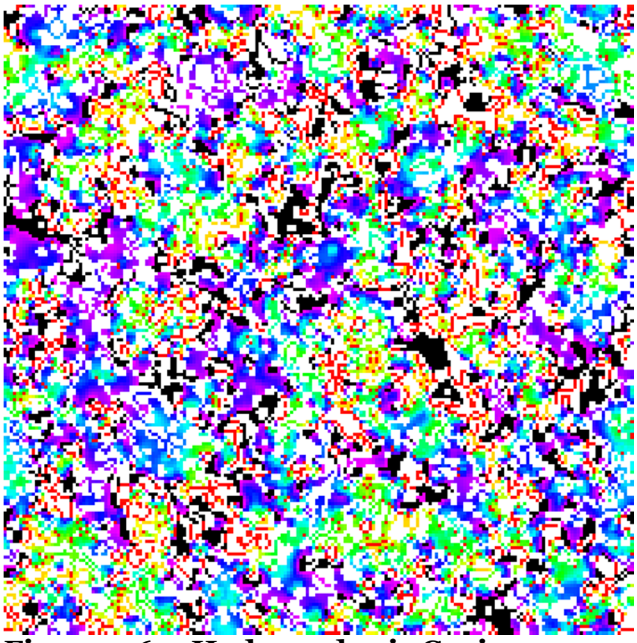


Figure 7.6.4 Hodgepodge is Grainy

```
   200 0.1 (2 3 0.17 100 hodge) show_auto b
0.1
```

Watching the evolution shows moving wave-fronts. White corresponds to healthy cell, hues to infections, and black as ill. The last step of that evolution is shown in Figure 7.6.1. Notice that the ends of the wave-fronts exhibit some dramatic spiraling arms. Watching these develop for more than 200 steps can be fun too. Figure 7.6.2 shows a highly structured configuration resulting from parameters `6 1 0.02 100`. Figure 7.6.3 shows very rich structured configuration resulting from the parameters `4 1 0.2 100`. Watching that hodgepodge machine develop for more than 200 iterations is worthwhile too. Figure 7.6.4 shows very fine patterns resulting from parameters `2 6 0.16 100`. The Hodgepodge rules provide models where a variety of self-organizing structures develop.

7.7 Hexagonal Lattice and the Packard-Wolfram Snowflake

We can consider automata on cellular arrangements that are not rectangular. This includes hexagonal or more general arrangements. As a first example we look at a simple automaton on a hexagonal lattice that has been popularized by Wolfram (2002) but had precedent due to Packard (1986). We begin by organizing and initializing the hexagonal arrangement in a way that generalizes to other sorts of cellular arrangements.

A hexagonal arrangement of cells can be represented by a rectangular array where we imagine offsetting alternate rows, and we need to take that view for display purposes; nonetheless, we will think of our cells as a single list with another list of giving the neighbors for corresponding cells. This view is relatively efficient and easily generalizes to more general arrangements of cells.

The function `hx_init` creates several global arrays related to a hexagonal arrangement of the specified size. These functions are loaded by *automata.ijs*. The size is given by `hxSZ` while the number of vertices is `hxNV` and a blank array `hxA` of suitable size is created. The index of a cell near the center is given by `hxcen` and, most importantly, the list of neighbors is given by `hxN`. We also create a utility that lets us see the index positions offset in their hexagonal arrangement.

```
   hx_init=: 3 : 0                          defined by automata.ijs
hxSZ=: y
hxNV=:*: hxSZ
hxA=:hxNV$0
hxcen=:<:-:hxNV-hxSZ
nr0=. _1 1 |."0 _ i=. i. hxSZ
nr0=.nr0 ,_1 0 |."0 _ i+hxSZ
nr0=.nr0 ,_1 0 |."0 _ i-hxSZ
nr1=. _1 1 |."0 _ i=.i + hxSZ
nr1=.nr1 ,0 1|."0 _ i+hxSZ
nr1=.nr1 ,0 1|."0 _ i-hxSZ
$hxN=:hxNV|,/(hxSZ*i.&.-: hxSZ)+"0 _|:nr0,.nr1
)
```

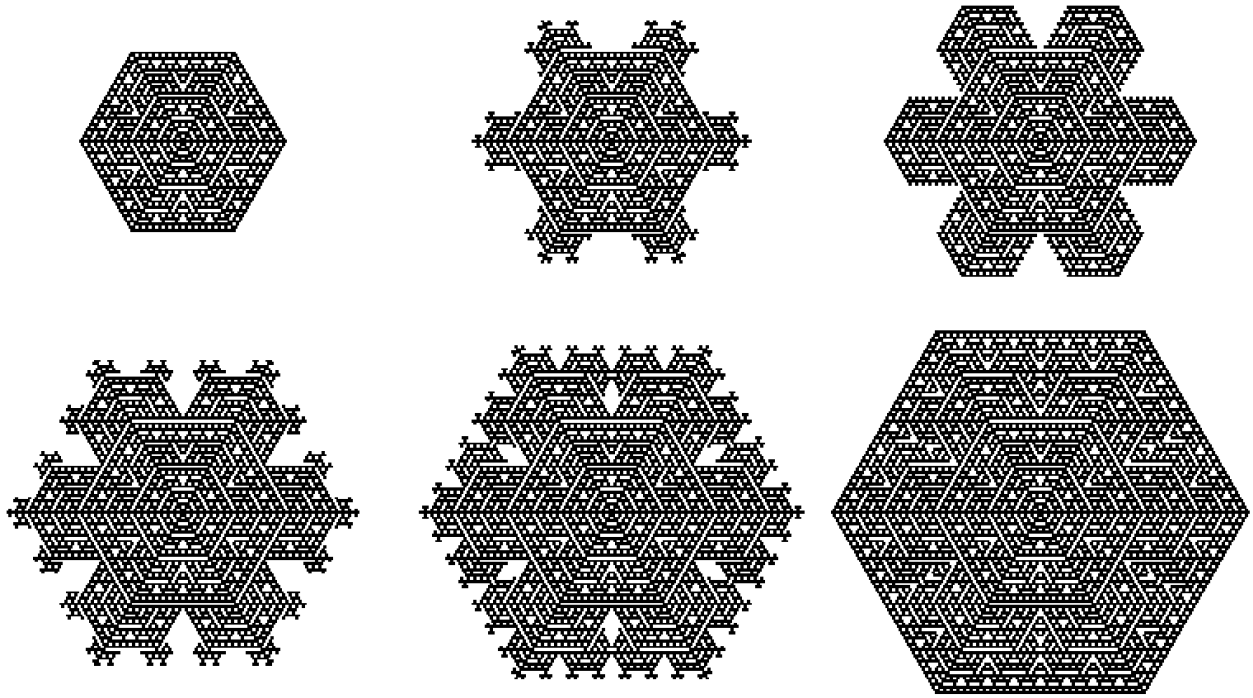


Figure 7.7.1 Some Iterates of the Packard-Wolfram Model

```
   hxv=: 3 : '((# $ 2 0"_) |."0 1]) ''  '' ,"(1) 4j0 ": y'

   hx_init 6
36 6

   hxv i. 6 6
   0   1   2   3   4   5
     6   7   8   9  10  11
  12  13  14  15  16  17
    18  19  20  21  22  23
  24  25  26  27  28  29
    30  31  32  33  34  35

   19{hxN
18 20 25 26 13 14
```

So above we can see that position 19 has positions 18, 20, 25, 26, 13, and 14 as its neighbors. To see neighbors near the edges, we periodically extend using functions from Section 6.3 and we need a slight variant for the display function since the periodic extension switches the even-odd row offset.

```
   hxvx=:3 : '((# $ 0 2"_) |."0 1]) ''  '' ,"(1) 4j0 ": y'

   hxvx perext2 i. 6 6
    35  30  31  32  33  34  35  30
   5   0   1   2   3   4   5   0
    11   6   7   8   9  10  11   6
  17  12  13  14  15  16  17  12
    23  18  19  20  21  22  23  18
  29  24  25  26  27  28  29  24
    35  30  31  32  33  34  35  30
   5   0   1   2   3   4   5   0

   3{hxN
2 4 8 9 32 33
```

So we see that position 3 has neighbors 2, 4, 8, 9, 32 and 33.

Now we are ready to discuss the Packard-Wolfram automaton. This automaton operates on a Boolean valued hexagonal arrangement of cells. It leaves any cell with value one as one, and a zero cell becomes one if exactly one of its neighbors is a one. Ordinarily, we begin with a single cell set to one. As a model for snowflake growth, a one corresponds to solid ice. The function `packwolf` defined below takes an array of length `hxNV` that represents an array of cells as its argument and results in the configuration updated by one time step. It uses the global array of lists of neighbors to get the information about neighbors. Here we see the result of one step of the function on an array that had a single cell ice.

```
   packwolf=: 3 : 'y+.1=+/"1 hxN{y'

   hxv (2#hxSZ)$ packwolf 1 hxcen}hxA
   0   0   0   0   0   0
     0   1   1   0   0   0
   0   1   1   1   0   0
     0   1   1   0   0   0
   0   0   0   0   0   0
     0   0   0   0   0   0
```

Of course we want to use a much larger array and watch the updates of the iteration process. Viewing a raster image of a hexagonal arrangement requires a little effort. We accomplish the offsets by making 2 by 2 blocks of pixels of the rectangular array and shifting by one pixel alternate pairs of rows. Then select rows are taken to achieve the correct aspect ratio. Since those viewing requirements will be true for any hexagonal automaton, we create a utility similar to the one we used for viewing rectangular lattice automata.

```
   spix=: [ # #"_1

   show_hx_auto=: 1 : 0                          defined by automata.ijs
(0.1,~_2+<.-:%:#y) u show_hx_auto y
:
'maxiter delay'=.x
k=.0
szh=.<.+:hxSZ* 1 o. 2r3p1
rows=.<.(+:hxSZ%szh)*i.szh
VRAWH=:+:(+:hxSZ),szh
rot=.(1 1 0 0 $~ +:hxSZ)"_ |."0 1 ]
vwin 'Hex Auto'
while. k<maxiter do.
  y=. u y
  VRA=:2 spix rows{rot 2 spix (2#hxSZ)$ <.y*255
  vfshow ''
  6!:3 delay
  k=.k+1
end.
hxSZ
)

   hx_init 230
52900 6

   110 0.1 packwolf show_hx_auto 1 hxcen}hxA
230
```

Several steps of this automaton are shown in Figure 7.7.1.

7.8 A Snowflake Model Using Intermediate Values

Next we turn to a 2-dimensional model of snowflake growth where we imagine cells as containing either bound or free water. Each cell will contain a value between zero and one where zero indicates no water and one designates completely bound ice. Intermediate values indicate an intermediate amount of water. However, we consider a cell receptive (to ice formation) if it is either already ice (value one) or if an immediate neighbor has value one. Thus, we imagine separating the material into receptive and non-receptive arrangements of material. At the subsequent step, each cell is the sum of two terms: (a) the receptive cells have a constant gamma added to their values, giving the first term; (b) the average of the non-receptive material (padded with zeros in receptive locations) is taken over each neighborhood, giving the second term.

Since snowflakes are often considered to have six fold symmetry, we will implement this automaton on a hexagonal array. The necessary utilities and functions below are loaded by *automata.ijs*. The first line of the dyad `cryst` determines (0/1) the cells that are receptive. Notice that the values of the neighbors are given by the rows of `hxN{y` as appears on the first line of the dyad. The second line gives the non-receptive values, padding with zeros in the receptive locations. The last line gives the sum of the two terms described above and puts an upper bound of one on the final result.

```
   cryst=: 3 : 0                                 defined by automata.ijs
0.001 cryst y
:
r=. +./"(1) 1=y,.hxN{y
nrv=.y*-.r
1 <. (avg"1 nrv,.hxN{nrv)+r*y+x
)

   hx_init 230
```

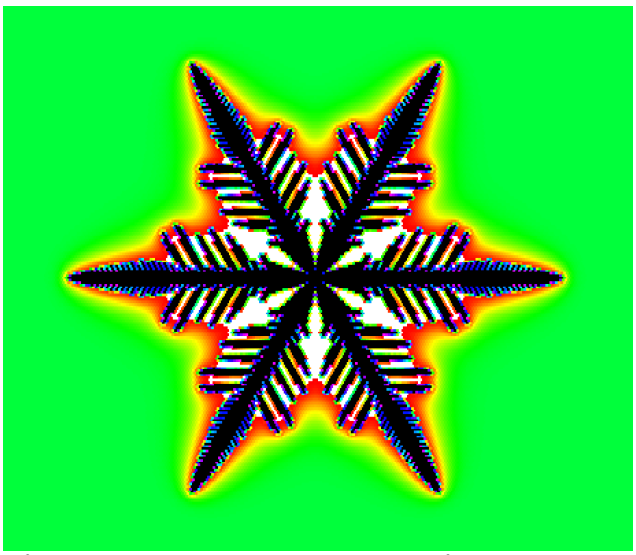


Figure 7.8.1 A Crystal Formation Automaton

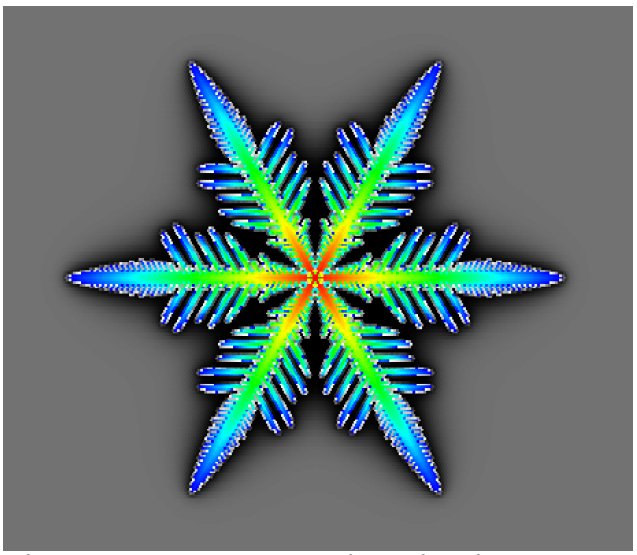


Figure 7.8.2 Another Visualization of the Crystal Formation

```
800 0 cryst show_hx_auto 1 hxcen}hxA+0.45
```

The above expressions create Figure 7.8.1. Of course, the generic adverb `show_hx_auto` does not necessarily show the detail that we would desire for visualizing the evolution of the crystal growth described above. In particular, that experiment highlights the non-receptive material and all the ice is shown in one color. The exercises explore a variant where `show_hx_auto` is modified to record the time-steps at which ice forms in any position. Figure 7.8.2 shows a way of combining the values of the non-receptive cells with the time steps that were required until ice formation. This shows the depletion of the receptive material while showing the accumulation of ice over time.

7.9 Exercises

1. Write a function and create raster images of all 255 nonzero Boolean, 3-cell automata, as in Section 7.1;
(a) however, the automata should be applied to `?.256#2`.
(b) however, the automata should be applied to `128=i.256`.

2. (a) Implement a general function giving a Boolean 5-cell automaton (that is, the new state of a cell depends upon state of the cell and the states of its two left and two right neighbors.
(b) How many rules of this type are there?
(c) Create 6 raster images of the evolution of rules of this type, on argument `?.64#2`, and that result in visually different types of behavior.

3. (a) Implement a general 3-state, 3-cell automaton; that is, the possible states are 0,1,2 and new state of a cell depends upon state of the cell and the states of its left and right neighbors
(b) How many rules of this type are there?
(c) Create 6 raster images of the evolution of rules of this type, on argument `?.81#3`, and that result in visually different types of behavior.

4. Write a conjunction that creates raster images of the 255 rules numbered 1 to 255 for specified fuzzy logics. Apply the result to the following logics and arguments.
(a) Max/Min on `0.1*?.64#11`
(b) Probabilistic on `0.01+0.98*?.64#2`
(c) The gcd/lcm fuzzy logic where "or" is `+.`, and "and" is `*.`, on `-:?.64#3`
(d) MV fuzzy logic where "or" is min(1, $x+y$) and "and" is max(0, $x+y-1$) on `0.01+0.98*?.64#2`
(e) CFMQVS fuzzy logic where "or" is min(1, $x+y$) and "and" is the product $x*y$ on `0.01+0.98*?.64#2`

5.(a) Create an animation of the evolution of the Game of Life on a random initial configuration.
(b) Create an animation evolution of the majority rule using 3 by 3 neighborhoods on a random initial configuration.

6. Identify the Game of Life with a Boolean automaton and use that viewpoint to create a fuzzy version of the Game of Life using the probabilistic fuzzy logic. Create a function that allows the viewing of the time evolution of that automaton on `0.01+0.98*?.50 50$2`.

7. Create an animation of the fuzzy Game of Life described in the previous exercise.

8. Iterate rules based on the following on a random Boolean initial configuration.
(a) the majority rule for the five cells on the upper right edge of a 3 by 3 neighborhood
(b) the majority rule for the top three rows of a 5 by 5 neighborhood
(c) the majority rule for the middle three rows of a 9 by 9 neighborhood?

9. What happens to a random initial configuration with any fuzzy value between zero and one under iteration of a rule if it is based upon a local rule that gives
(a) the average value in a 3 by 3 neighborhood
(b) the median value appearing in a 3 by 3 neighborhood

10. Implement the behavior of a cyclic cellular automaton with the following neighborhood patterns. Explore the results for various choices of number of states.
(a) `nei=: 0 1 2 3 5 6 7`
(b) `nei=: 1 2 3 5 6 7`
(c) `nei=: 1 2 3 5 7`
(d) `nei=: 0 2 7`
(e) `nei=: 0 2 6 8`
(f) `nei=: 0 3 5`

11. Explore the behavior of the Hodgepodge automata
(a) Create a movie of the first 200 iterations of the automaton with parameters `2 3 0.17 100`.
(b) Create a movie of the first 200 iterations of the automaton with parameters `2 3 0.13 100`.
(c) Find images after 200 iterations as `a` varies for parameters `a, 3 0.17 100`.
(d) Find images after 200 iterations as `b` varies for parameters `2, b, 0.17 100`.
(e) Find images after 200 iterations as `c` varies for parameters `2 3, c, 100`.
(f) Find images after 200 iterations as `N` varies for parameters `2 3 0.17, N`.

12. Implement an analog of the Packard-Wolfram on a rectangular array of cells. What sort of images are produced as the process is implemented?

13. Implement cyclic cellular automata on a hexagonal lattice. Explore the results for various choices of number of states.

14. The crystallization function used in Section 7.8 can be viewed as depending on the background level used for all the cells except the central 1 and also the value of added to the receptive cells.
(a) Run the crystallization process with background level 0.35 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$.
(b) Run the crystallization process with background level 0.4 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$.
(c) Run the crystallization process with background level 0.45 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$.
(d) Run the crystallization process with background level 0.95 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$.
(e) Find 5 beta-gamma pairs that give significantly different behavior.
(f) Create a modified version of `show_hx_auto` that runs until ice has reached within 5 cells of the top edge or until 10,000 iterations have been done.
(g) Use (f) to make a table of the results for background levels `0.05*i.20` and all γ as in (a).

15. Create a modified version of `show_hx_auto` and apply it to the crystallization process so that it
(i) runs until ice has reached within 5 cells of the top edge or until 10,000 iterations have been done
(ii) defines a global value `when` that keeps track of the iteration time when any cell becomes ice
(iii) defines a global value `what` that gives the values in all the cells
(iv) defines a palette, half grayscale and half hue by the following:

```
$BWH=: (3#"0 +:i.128),Hue |.2r3*(i.%<:) 128
```

(v) creates images such that ice cells are shown in a hue corresponding to `when` and non-ice cells are shown in a grayscale according to `what`
(vi) offers an option or variant to save every frame
(a) use the function to duplicate Figure 7.8.2
(b) create a movie of the evolution as the figure in (a) is created

(c) create movies of the evolution when background level is 0.35, 0.4, 0.45, 0.5 and 0.55 and where $\gamma = 0$
(d) create a movie of the final frame evolving for $\gamma = 0$ as the background level moves from 0 to 0.99
(e) create a movie of the final frame evolving for background level 0.45 as γ moves from 0 to 1 (but use a nonlinear sampling that samples more near 0).

16. (a) Implement the crystallization process on a rectangular arrangement of cell.
(b) Run the crystallization process with background level 0.35 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$.
(c) Run the crystallization process with background level 0.4 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$.
(d) Run the crystallization process with background level 0.45 and $\gamma = 0, 0.001, 0.002, 0.01, 0.1, 0.5$

Bibliography and References

T. W. Anderson (1994), *The Statistical Analysis of Time Series*, John Wiley & Sons.
S. D. Balkin (1996), *Fractal and Chaotic Time Series Analysis*, Master of Arts Paper, The Pennsylvania State University.
M. Barnsley (1993), *Fractals Everywhere, 2nd Edition*, Academic Press.
M. Barnsley, L. Hurd (1993), *Fractal Image Compression*, AK Peters, Ltd.
H. E. Benzinger, S. A. Burns and J. I. Palmore (1987), Chaotic complex dynamics and Newton's method. *Physics Letters A*, **119**, 441-446.
E. R. Berlekamp, J. H. Conway, and R. K. Guy (1982), *Winning Ways for Your Mathematical Plays*, Academic Press.
B. A. Bondarenko (1993), *Generalized Pascal Triangles and Pyramids*, The Fibonacci Association.
P. J. Brockwell and R. A. Davis (1996), *Introduction to Time Series and Forecasting*, Springer.
F. Budden (1985), Cayley Graphs for Some Well-Known Groups, *The Mathematical Gazette*, **69**, 271-278.
C. Burke, R. K. W. Hui, K. E. Iverson, E. E. McDonnell, D. B. McIntyre (1998), *Phrases, Introduction and Dictionary*, Iverson Software Inc.
M. Casdagli (1992), Chaos and Deterministic versus Stochastic Non-linear Modelling, *Journal of the Royal Statistical Society, Series B*, **54** 2, 303-328.
G. Cattaneo, P. Flocchini, G. Mauri, C. Quaranta Vogliotti, and N. Santoro (1997), Cellular automata in fuzzy backgrounds, *Physica D*, **105**, 105-120.
Cayley (1879), The Newton-Fourier Imaginary Problem, *American Journal of Mathematics*, **2**, 97.
Chen Ning, C. A. Reiter (2007a), Generalized Binet Dynamics, *Computers & Graphics* **31**, 301-307.
Chen Ning, C. A. Reiter (2007b), A Cellular Model for 3-Dimensional Snow Crystallization, *Computers & Graphics* 31, 668-677.
P. Chidyagwai, C. A. Reiter (2005), A Local Cellular Model for Growth on Quasicrystals, *Chaos, Solitons & Fractals*, **24**, 803-812.
A. M. Coxe, C. A. Reiter (2003), Fuzzy Hexagonal Automata and Snowflakes, *Computers & Graphics* **27**, 447-454.
P. J. Davis (1993), *Spirals, From Theodorus to Chaos*, A K Peters.
R. L. Devaney (1987), *An Introduction to Chaotic Dynamical Systems*, Addison Wesley.
R. L. Devaney (1990), *Chaos, Fractals and Dynamics*, Addison Wesley.
R. L. Devaney (1992), A First Course in Chaotic Dynamical Systems, Addison-Wesley.
A. K. Dewdney (1988), *The Armchair Universe*, W. H. Freeman and Co.
A. K. Dewdney (1993), *The Tinkertoy Computer*, W. H. Freeman and Co.
D. Dubois, H. Prade (1980), *Fuzzy Sets and Systems*, Academic Press.
D. J. Dunham (1986a), *Creating Hyperbolic Escher Patterns, in M. C. Escher: Art and Science*, H. M. S. Coxeter et al (Editors), Elsevier Science Publishers.
G. A. Edgar, editor (1993), *Classics on Fractals*, Addison-Wesley Publishing Co.
J. Feder (1988), *Fractals*, Plenum Press.
R. M. Friedhoff, W. Benzon (1989), *Visualization, the Second Computer Revolution*, Harry N. Abrams, Inc.
J. Gleick (1988), *Chaos, Making a New Science*, Penguin Books.
R. C. Gonzalez, R. E. Woods (1992), *Digital Image Processing*, Addison-Wesley Publishing Company.
I. Grossman, W. Magnus (1964), *Groups and Their Graphs*, Random House.
D. Gulick (1992), *Encounters with Chaos*, McGraw-Hill.
R. K. W. Hui (1992), *An Implementation of J*, Iverson Software Inc.
R. K. W. Hui, K. E. Iverson (1998), *J Dictionary*, Iverson Software Inc.
J. Hutchinson (1981), Fractals and Self-similarity, *Indiana University Journal of Mathematics*, **30**, 713-747.
H. J. Jeffrey (1990), Chaos Game Representation of Genetic Sequences, *Nucleic Acid Research*, **18** 8, 2163-2170.
H. J. Jeffrey (1992), *Chaos Game Visualization of Sequences*, Computers & Graphics, **16** 1, 25-33.
E. B. Iverson (1998), *J Primer*, Iverson Software Inc.

K. E. Iverson (1990), *The Dictionary of J*, *Vector*, **7** 2.
K. E. Iverson (1993), *Calculus*, Iverson Software Inc.
K. E. Iverson (1995), *Concrete Math Companion*, Iverson Software Inc.
Y. Kayama, M. Tabuse, H. Nishimura, T. Horiguchi (1993), Characteristic Parameters and Classification of One-dimensional Cellular Automata, *Chaos, Solitons & Fractals*, **3** 6, 651-665.
G. J. Klir, U. H. St. Clair, B. Yuan (1997), *Fuzzy Set Theory*, Prentice Hall.
A. Lakhtakia, D. Passoja (1993), On Congruence of Binary Patterns Generated by Modular Arithmetic on a Parent Array, *Computers & Graphics*, **17** 5, 613-617.
J. Lagarias (1985) The 3*x*+1 problem and its generalizations, *American Mathematical Monthly*, **92** 1, 3-23.
H. Levinson (1990), Cayley Diagrams, *Mathematical Vistas: Papers from the Mathematics Section*, Annals of the New York Academy of Sciences v. **607**.
B. Mandlebrot (1983), *The Fractal Geometry of Nature*, W.H. Freeman and Company.
B. Mandelbrot (1997), Fractals and Scaling in Finance; discontinuity, concentration, risk, Springer-Verlag.
B. B. Mandelbrot, J. R. Wallis (1969), *Water Resource Research*, **5** 2, 321-340.
M. A. Motyka, C. A. Reiter (1990), Chaos and Newton's Method on Systems, *Computers & Graphics*, **14** 1, 131-134.
N. H. Packard (1986), Lattice models for solidification and aggregation, *Theory and Applications of Cellular Automata*, S. Wolfram, ed, World Scientific Publishing, 305-310.
H.-O. Peitgen, D. Saupe, F. v. Haeseler, (1984), Cayley's Problem and Julia Sets, *Mathematical Intelligencer*, **6**, 11-20.
H.-O. Peitgen, D. Saupe, editors (1988), *The Science of Fractal Images*, Springer-Verlag.
H.-O. Peitgen, H. Jürgens, D. Saupe (1992), *Chaos and Fractals, New Frontiers of Science*, Springer-Verlag.
E. E. Peters (1991), Chaos and Order in the Capital Markets, Wiley.
E. E. Peters (1994), *Fractal Market Analysis*, Wiley.
C. A. Pickover (1990), *Computers Pattern Chaos and Beauty*, St. Martin's Press.
C. A. Pickover (1992), *Mazes for the Mind*, St. Martin's Press.
B. J. Reiter, C. A. Reiter, Z. X. Reiter, (1998) Word of Words from Iterated Function Systems, *Vector*, **15** 2, 104-120.
C. A. Reiter (1993), Fractals and Generalized Inner Products, *Chaos, Solitons and Fractals*, **3**, 695-713.
C. A. Reiter (1994a), Sierpinski Fractals and GCDs, *Computers & Graphics*, **18** 6, 885-891.
C. A. Reiter (1994b), Fractals RYIJ, *Vector*, **11** 2, 86-104.
C. A. Reiter (1995), Infix, Cut and Finite Automata, APL95 Conference Proceedings, *APL Quote Quad*, **25** 4, 162-169.
C. A. Reiter (1996), Mathematical Languages, *Proceedings of the J User Conference*, Iverson Software, Inc.
C. Reiter (2003), With J: Fractal Forecasting, *APL Quote Quad*, **34** 1, 15-18.
C. Reiter (2004), With J: The Hodgepodge Machine, *APL Quote Quad*, **34** 4, 2-7.
C. A. Reiter (2005), A Local Model for Snow Crystal Growth, *Chaos, Solitons & Fractals*, **23** 4, 1111-1119.
C. A. Reiter (2010), Medley of Spirals from Cyclic Cellular Automata, *Computers & Graphics*, **34**, 72-76
A. Salomaa (1985), *Computation and Automata*, Cambridge University Press.
D. J. Schove (1983), *Sunspot Cycles*, Hutchinson Ross Publishing.
W. Sierpinski (1915), Sur une Courbe Cantorienne dont tout Point est un Point de Ramification, *Comptes Rendus Académie des Sciences*, **160**, 302-305.
W. Sierpinski (1916), Sur une Courbe Cantorienne qui Contient une Image Biunivoquet et Continue Detoute Courbe Donnée, *Comptes Rendus Académie des Sciences*, **162**, 629-632.
R. Terras (1976), A stopping time problem on the positive integers, *Acta Arithmetica*, **XXX**, 241-252.
T. Vicsek (1992), *Fractal Growth Phenomena, 2nd edition*, World Scientific.
T. Wegner and M. Peterson (1991), *Fractal Creations*, Waite Group Press.
S. Wolfram (1986), *Theory and Applications of Cellular Automata*, World Scientific.
S. Wolfram (2002), *A New Kind of Science*, Wolfram Media.

Index

3*x*+1 function, 65, 106
Adjoin, 6
Adverb, 12, 85
Adverse, 107
Agenda, 37, 65
Al Hensel's Life patterns, 124
Animations, 36, 89
Antibase, 73, 92, 95, 113
Anti-persistent walk, 53
Argument, 8
Arrays, 5
Assigment, 1
Assignment, 11, 26
At, 23
Atom, 15
Atop, 23
Autocorrelation, 55
Automata, 113
automata.ijs, 113, 115
Average, 45
Axis, 6, 15
Barnsley fern, 71
Base, 95, 113
Behead, 51
Best analogs, 58
Bond, 8
Boolean automaton, 113
Box, 17
Box counting dimension, 75
Bulge, 30
Cap, 40
Cell, 15
Cellular automata, 113
Chaos game, 72
Characters, 17
`cile`, 87
Circular function, 3
COKE IFS, 69
Collage, 38
Color contours, 86
Color models, 83
Commute, 48
Complex number, 13, 17, 101
complex_dynamics.ijs, 101
Compose, 23
Composition, 26
Conjunction, 12, 85
Constant function, 39
Contour plot, 86, 99
Conway's Game of Life, 121
Cosine, 3
Curtail, 51
Cut, 53, 62
Cyclic cellular automata, 126
Density plot, 99
Derivative, 109
Determinant, 71
Deterministic chaos game, 74
Divide, 1
DNA, 74
Domain error, 13
Dot product, 51
Double, 2
Drawing window, 4
Drop, 50, 62
dwin.ijs, 4
Dyad, 8, 11
Elliptic curves, 103
Elongate, 32
Escape time, 96, 101
Evoke gerund, 36
Ewart Shaw's signature, 123
Exact rational, 3
Explicit function, 8
Exponential, 1
Extend data, 51
Fern IFS, 71
Fibonacci sequence, 42
Fix, 73
Floor, 11, 40
Foreign conjunction, 53
Fork, 26
Fourier series, 86
Fractal dimension, 75, 78
Fractal forecasting, 58
Fractal time series, 56
From, 25
Function definition, 8
Function power, 9, 12
Fuzz removal, 87
Fuzzy logic, 117
Game of Life, 121
Gaussian distribution, 49
Gaussian walks, 50
Gerund, 36
Glider gun, 123
Global assignment, 11
Grade up, 59
Greater of, 27
Greatest common divisor, 95
Half, 11
Head, 25, 51

Henon map, 25, 58
Hexagonal lattice, 131
Hierarchy, 2
Hodgepodge rule, 128
Homogeneous coordinates, 32
Hook, 27
Hues, 85
Hurst exponent, 53, 56, 91
Identity function, 6, 7, 27
Identity matrix, 34
Ill-formed number, 14
Image file, 8
imagekit.ijs, 7
In, 77
Indeterminate, 22
Index, 25
Index of, 94
Indices, 3
Infected neighbors, 128
Infinity, 13, 22
Infinity function, 107
Infixes, 48
Inflection, 12
Inner product fractals, 92
Insert, 16
Integers, 3
Interference pattern, 86
Interpolate, 57
Interpolation, 56, 90, 91
Inverse IFS, 96
Item, 15
Iterate until, 96
Iterated function system, 66, 69
Iteration, 9
J scripts, 10
Julia set, 101, 103
Koch snowflake, 31, 42, 75
Lake Huron, 52
Laminate, 6
Least common multiple, 95
Least squares fit, 45, 47
Left, 27, 39
Length error, 12
Lesser of, 27
Life, 121
Linear model, 46
Linear system, 46
Link, 17, 24
Local assignment, 11
Local rule, 114
Logarithm, 2
Logistic map, 61, 63
Majority rule, 124
Mandelbrot set, 104
Matrix, 5
Matrix product, 32, 92
Maximum, 27, 45
Member, 77, 109
Midpoint, 30
Midpoint displacement, 56
Minimum, 27, 45
Minus, 1
Modulo, 65
Monad, 8
Moving average, 50
Multiple line function, 10
Natural logarithm, 2
Negate, 1
Newton's method, 108
Nile river, 52
Normal distribution, 49
Noun, 12
Nub, 93
Number, 15
Number of items, 6
Object graphics, 8
One-dimensional automata, 113
Open quote, 13
Packard, 131
Palette, 83
Parallel assignment, 26
Parametric plot, 25
Partial sums, 48
Performance measures, 66
Periodic extension, 114, 122, 124
Persistent walk, 53
Pi, 3
Pi times, 49
Pixel, 7
Plasma clouds, 90
Plot, 24
Plot driver, 47
Plus, 1
Plus insert, 16
Polar curves, 41
Polygons, 4
Polynomial, 40, 46
Power, 1
Prefix scan, 73
Prefixes, 48
Probabilistic fuzzy logic, 118
Probabilistic IFS, 66, 71
Pronoun, 12
Proverb, 12
Random, 46, 49
Random addition, 56, 91
Random IFS, 79
Random normal, 49
Random seed, 53
Random walk, 48

Range, 27, 45
Rank, 16, 39
Rank infinity, 39
Raster graphics, 8
Raster image, 7
raster.ijs, 67
Rational, 3
Reciprocal, 1
Regular pentagon, 4
Regular polygon, 10
Remainder, 65
Rescaled range analysis, 52
Reshape, 6
Reverse, 30
Reverse rule, 115
RGB color model, 83
Right, 27, 39
Root finding, 108
Rotate, 31, 32
Round, 40
Scripts, 10
Self-classify, 34
Self-organization, 127, 130
Self-reference, 49
Shape, 6, 15
Sierpinski carpet, 21, 75, 78, 95
Sierpinski triangle, 7, 9, 28, 38, 68, 75, 92, 98
Sine, 3
Smile, 79
Snowflake, 131, 133
Sort, 46
Space to execute, 66
Spencer average, 52
Spiral, 79, 127
Spiral waves, 130
Spot formation, 124
Square, 2
Square root, 2, 17
Standard deviation, 45
Standard normal distribution, 49
Stationary, 55
Statistics, 45
Stitch, 6, 28
Subintervals, 54
Suffix scan, 73
Sum, 16, 45
Sunspots, 54, 60
Super-pixels, 114
Syntax error, 13
Table, 4, 94
Tacit functions, 8
Tail, 25, 51
Take, 50, 62
Tally, 6, 15
Tesselations, 53, 76
Time series, 25, 45, 48
Time to execute, 66
Times, 1
Times table, 4
Transformation, 32, 33, 43
Translate, 32
Transpose, 4
Trends, 50
Tribonacci sequence, 42
Trigonometric functions, 3
Two-dimensional automata, 121
Under, 88, 109
Uniform distribution, 49
Verb trains, 26
Verbs, 12
viewmat.ijs, 7
Word formation, 21