FRACTAL LAB KIT

By

Ronald Thomas Kneusel

version 3.0

September 9, 1994

Contents

Chapter 1

Introduction

1.1 What is this all about?

The fractal lab kit is a command driven system for generating and investigating fractal images. It is written using Chris Heilman's Pocket Forth , a small Forth interpreter for the Macintosh. Pocket Forth is available via anonymous FTP from *archive.umich.edu* in the directory */mac/development/languages/*.

The kit consists of a set of Forth words that allow the user to easily create fractals based on IFS mappings. Because of it's small size and generality, it should run on virtually all Macintosh computers. By entering simple commands the user can define maps, draw the resultant fractals in a variety of colors, show the orientation of the mappings, measure positions within the image, and zoom in to view details. Fractals using up to seven maps may be generated.

1.2 Why not create a standard Macintosh application for all of this?

I used Forth for two reasons. First, I am still learning the language and thought that this would be a good project towards that end. Secondly, I have used several fractal programs in the past and while they are excellent at what they do, they are rigid and inflexible. Creating a system like this in Forth seemed ideal. Forth is fast and highly interactive. It is also small and very easily extended. Not only does the user have all the fractal commands, but they also have all of Forth still at their disposal. A little programming can quickly extend the existing capabilities. Pocket Forth was my choice for several reasons, primarily, it is free and can be freely distributed, as well as supporting floating point numbers to make life easier.

Kudos to Dr. Bruce Craven, University of Melbourne, Australia, for all his helpful comments and suggestions for improving the program. Kudos also to Mr. Stephen Gard, Australia, for his thorough testing of and enthusiasm for the program!

Fractal Lab Kit is freeware, if you use it and have any suggestions or comments please let me know at the addresses below. I have plans for additional 'modules' for investigating Mandelbrot and Julia sets, biomorph images, and chaos in one and two dimensions. Your feedback will encourage me to continue. Even if you have no specific comment, let me know where you are so I can follow the program. I like postcards, but email is good too.

Ron Kneusel
8725 West Burdick Ave.
Milwaukee, WI 53227
USA

kneusel@msupa.pa.msu.edu
or
rtk@herman.gem.valpo.edu

Chapter 2

Fractals and IFS: an overview

A fractal is a geometric object with a non-integer dimension. We are used to thinking in terms of Euclidean geometry, that is, in terms of points, lines, areas (surfaces), and volumes (solids). Fractals do not easily fit in such a framework and are difficult to comprehend (at least for me). Common geometrical objects have integer dimensions:

Object	**Dimension**
point	0
line	1
plane	2
solid	3

but fractals have non-integer dimensions, like 0.63... etc. How can this be you say? Let's look at the simplest fractal of them all, the Cantor set.

To create the Cantor set imagine a line of length 1. Now, remove the middle third of that line. Remove the middle third of the two remaining lines. Continue removing the middle third an infinite number of times. When you are finished the object you will be left with is a fractal with a dimension that is not 0 (it's not a point) but is less than 1 (a line from 0 to 1 would contain all points, clearly a cantor set does not) and it can be shown that the final dimension is ln2/ln3 = 0.6309297536.....

Derivation of the Cantor Set dimension

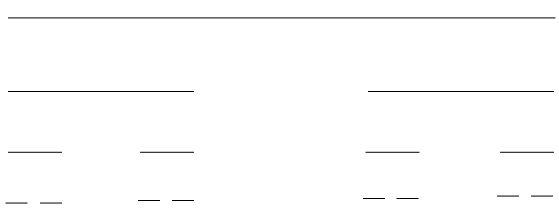
One possible definition of dimension, the self-similar dimension, is the ratio of the natural log of the number of intervals N (in 1-d) of length β needed to completely cover the set to the natural log of the reciprocal of β, taken in the limit that $\beta \rightarrow \infty$. In the case of the Cantor set $\beta = (1/3)^n$ where n is the 'level' of the set. (Check: the second level has two sections of length 1/3 while the next level has four sections of length $1/9 = (1/3)^2$) So the condition $\beta \rightarrow \infty$ becomes $n \rightarrow \infty$. Also, for the Cantor set, the number of intervals to cover the set at the level n is $N = 2^n$. (Check: at the second level there are $2^1 = 2$ intervals, at the third level there are $2^2 = 4$ intervals and so on.) So we write:

$$D = \lim_{n \rightarrow \infty} \ln N / \ln(1/\beta) = \lim_{n \rightarrow \infty} \ln 2^n / \ln(1/(1/3)^n)$$

$$D = \lim_{n \rightarrow \infty} n \ln 2 / n \ln 3 = \ln 2 / \ln 3 = 0.6309297536 \ldots$$

The Cantor set, like other purely mathematical fractals, is completely self-similar. That is, it looks identical on different scales (actually it is independent of scale). Take a piece of paper (you do have scrap paper near the computer, don't you?) and draw the first few lines of the Cantor set, one below the other:

`Cantor set`



If you look at what you have just drawn you will notice that if you magnify a lower level it will look like one above it, the entire set is made up of an infinite number of copies of itself. (Infinity shows up frequently when talking about fractals.) As this is a simple fractal it is not terribly interesting; the program draws more complex fractals where life is a little more exciting.

They are definitely odd, but fractals do turn up in many places in nature, the nautilus shell (see the SPIRAL), your lungs and circulatory system (see the TREE), etc. Some people even think that the distribution of matter in the universe is a fractal.

2.1 What about fractals and chaos?

Fractals are often mentioned in connection with chaotic behavior of dynamical systems. The link comes from the fact that the final attractor (a strange attractor) of a dissipative dynamical system is a fractal object. If you want to talk more about this, contact me at the above address.

2.2 What is IFS?

IFS (Iterated Function System) is the means by which the program generates images. It was developed by Michael Barnsley. IFS involves defining a number of maps that in some way determine what the final product will look like (more on that later). An initial point is chosen (the origin is nice) and iterated (i.e. put it in - get something out - put that something back in, etc.) . These are maps in the mathematical sense - roughly, a way of transforming a collection of points into another space or back into its own space as is the case here. The map used for each iteration is chosen at random based on the assigned probability, the higher the probability the more likely that map is to be chosen. After each iteration the resulting point is plotted. As this process is continued the fractal image is built. Changing the probability of a map can dramatically affect the resulting image, so it might take a bit to get the picture 'just right'.

In mathematical terms a 2 dimensional map is most easily represented in matrix form. A matrix is similar to the two dimensional arrays used in many programming languages. If we have a starting point (x, y) and we want to find the transformed coordinates, (x', y'), we can write the transformation in this way:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

This is equivalent to writing two equations:

$$\begin{aligned} x' &= ax + by + e \\ y' &= cx + dy + f \end{aligned}$$

The 2x2 matrix controls the reorientation of the initial coordinate system while the vector $\begin{pmatrix} e \\ f \end{pmatrix}$ is an offset to a new origin point for the map.

2.3 Finding the 'magic' numbers for an IFS transformation

By way of example, I will show how to find the matrix values that generate what has become known as the Mandelbrot Dragon. While the dragon is usually generated according to a prescription (like the Cantor set above) it can also be found using two mappings (i.e. two matrices).

Recalling that a fractal is made up of an infinite number of copies of itself, we need only specify the 'first' copy and the IFS algorithm will fill in the rest. Therefore, imagine a square from 0 to 1, this is the starting point as it were. We must transform the points from this square into a different square (or squares), where the number of transformations equals the number of 'parts' that make up the fractal. The dragon consists of two parts: a contracted 45 degree rotation of the coordinate axes and a contracted -45 degree rotation (with a flip). Perhaps the best way to imagine this is to picture an arrow: $\rightharpoonup$ from 0 to 1, the head of the arrow is only a half head so we can see if there is a flip as well as a rotation and translation. After some thought we realize that we need something like the following to generate the dragon:

Again take a piece of paper and draw a 2 inch arrow as above, the head of the arrow is on the right and the barb is pointing north. Now, draw another arrow from the left edge of the first going at a 45 degree angle until it intersects an imaginary line running north to south that passes through the midpoint of the first line. Draw the barb of this arrow on the left side of the right endpoint of this line. Finally, complete the triangle and draw the barb of this arrow on the left side and touching the head of the second arrow. When you are finished you will have something like Figure 2.1. where the directions of the arrow heads are indicated. Finally, complete each of the three squares defined by the arrows as one edge.

Now comes the fun part, how do we change the drawing into the numbers for the maps? To find the maps we need to solve, for each map, two systems of three linear

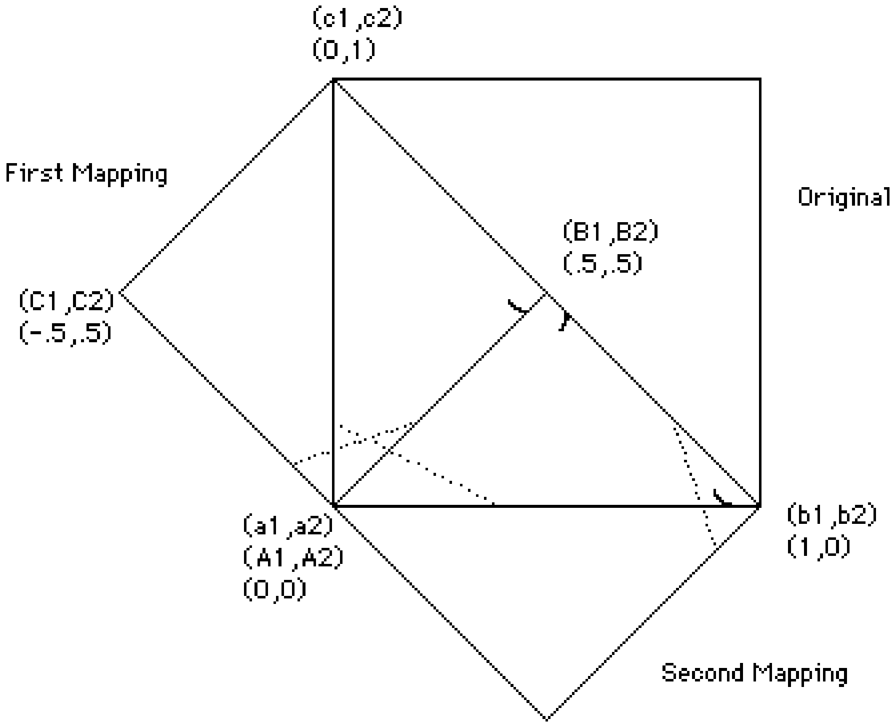


Figure 2.1: The maps necessary to create the Mandelbrot dragon.

equations since there are six numbers to find. To write the six equations we must know where at least three points of the original square (the one from 0 to 1) map to in the new maps. This is where the arrow head directions become important. The arrow determines two points for us, the head and the tail, while the last point can be either of the two corners of the original square and where they map to in the final square. So, to this end, label the tail and head of the first arrow drawn (a_1, a_2) and (b_1, b_2) respectively, and label the corner of the square above (a_1, a_2) as (c_1, c_2). These are the original three points, now label the corresponding points in the second square (the one at a positive 45 degree angle to the first) as (A_1, A_2) for the tail (in this case they are the same point) and (B_1, B_2) for the head. Label the farthest left corner of the square (C_1, C_2).

Once this is done, we can write the following six equations to determine the first map:

$$\begin{aligned} A_1 &= a_1 a + a_2 b + e \\ B_1 &= b_1 a + b_2 b + e \\ C_1 &= c_1 a + c_2 b + e \end{aligned}$$

and

$$A_2 = a_1 c + a_2 d + f$$

$$\begin{aligned} B_2 &= b_1 c + b_2 d + f \\ C_2 &= c_1 c + c_2 d + f \end{aligned}$$

In order to get actual numbers we need to impose a coordinate system. Draw an x and y axis where the first arrow goes from 0 to 1 on the x axis and left edge of the first square goes from 0 to 1 on the y axis. In this coordinate system, $(a_1, a_2) = (0, 0)$; $(b_1, b_2) = (1, 0)$; $(c_1, c_2) = (0, 1)$ and the points for the first map are $(A_1, A_2) = (0, 0)$; $(B_1, B_2) = (0.5, 0.5)$; $(C_1, C_2) = (-0.5, 0.5)$. The values for the matrix and vector are found by solving the two systems. Cramer's rule allows the solutions to be written as:

$$a = \frac{\begin{vmatrix} A_1 & a_2 & 1 \\ B_1 & b_2 & 1 \\ C_1 & c_2 & 1 \end{vmatrix}}{\begin{vmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{vmatrix}}$$

$$b = \frac{\begin{vmatrix} a_1 & A_1 & 1 \\ b_1 & B_1 & 1 \\ c_1 & C_1 & 1 \end{vmatrix}}{\begin{vmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{vmatrix}}$$

$$e = \frac{\begin{vmatrix} a_1 & a_2 & A_1 \\ b_1 & b_2 & B_1 \\ c_1 & c_2 & C_1 \end{vmatrix}}{\begin{vmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{vmatrix}}$$

Where the vertical lines represent the determinant. There exists a corresponding set for the c, d, and f values. With these we find that the transformations are:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0.5 & -0.5 \\ 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -0.5 & -0.5 \\ 0.5 & -0.5 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

I will leave the actual solution as an exercise for the reader.

The last thing to consider is what sort of probability we want to assign to each of these maps. Since there is no reason to favor one map to another we can in this case get away with a probability of 0.5 for each (remember, probabilities should add to 1). This is not always the case, though.

There are many good books on fractals and chaos at all levels, check the local book store or library. The book *Chaos: The Making of a New Science* by John Gleik is a good place to start, though it is lean on the technical aspect.

2.4 Getting Started

The IFS module includes the *Fractal Lab Kit* application and the documentation, demo and maps folders. The program starts in *Interactive mode* (i.e. in Forth). There is a mouse driven shell that you may use to adjust the settings and drawing, to use it enter *shell* at the prompt. See the chapter *Using the Shell* for more information. The examples in the *Tutorial* do not use the shell.

The *Maps* folder contains definitions for several fractals courtesy of Dr.Dale Snider, UW-Milwaukee Department of Physics. Use *Open* under the *File* menu to load the maps. To use, enter the map name and the word `draw`. The *Demo* folder contains a color and black & white version of `Cathedral` which draws a cathedral using fractals. Examine the source code (in Forth) to see how it works. Unfortunately, it works best on large screens... sorry Classic and SE users!

The commands you enter are really just Forth words. In fact, you are really using Pocket Forth with the IFS words predefined, however, you need not be familiar with Forth to use the program.

2.5 Setting up for Different Sized Monitors

Upon startup *Fractal Lab Kit* will get the size of the main monitor's screen and set the application window accordingly. However, the drawing parameters will be preset only if the screen is `512 x 342`, `512 x 384`, or `640 x 480` pixels. If you use a different sized monitor you will need to adjust the screen origin to make the most use of it as *Fractal Lab Kit* will set up for a `512 x 384` pixel screen if it does not recognize one of the above monitor sizes.

If you are using a monitor other than the main monitor you will also need to adjust the screen origin with the word `screen` to move the drawing region onto the secondary monitor. By default, the main monitor is used.

Chapter 3

Tutorial

3.1 A Sample Session

In this sample session and what follows, things the user types are indicated in bold while the computer's response is in plain text.

after double-clicking the Fractal Lab Kit icon
ok _ **open** *return*
choose the IFS-Maps file from the standard Mac open dialog
ok _ **fern green color on outlines draw**
program loads the maps for the fern, draws the outlines of the maps, waits for a keypress, and draws the fern in green until the user presses a key
ok _ **mouse**
program shows the true coordinates of the mouse pointer until the user presses a key
ok _ **settings**

```
Current plot origin ( 0.00000 , 0.00000 )
Current screen origin ( 100 , 330 )
Current x & y scale is 1.0000 :  1.0000
Axes are currently OFF
Draw Outlines is currently ON
Current number of maps = 5
```

ok _ **off outlines 0.178 0.034 origin .25 range cdraw**
program zooms in to the second leaflet and generates a magnified image using a different color for each map until a keypress
ok _ **bye**
program exits

3.2 Basic commands

A list of the basic, and most interactive, commands are presented along with an example of their use. These are the minimum commands necessary to use the program:

```
draw
Draw a fractal based on the current maps.  Press a key to stop.
E.g. green color fern draw

cdraw
Plots each map in a different color.
E.g. spiral cdraw

<n> edit
Edit the n-th map.  Enter a new value or press return to leave the
existing value as is.
E.g. 2 edit
  a = 0.5 ?-0.5 (new value)
  b = -0.5 ?<return>   etc.

zero-maps
Erase all twelve maps.
E.g. zero-maps

<color.name> color
Set the current drawing color to the color named.  Valid colors are
black, wite, red, green, blue, yellow, cyan, magenta
E.g. magenta color

<n> maps
Set the number of maps to use to <n>.
E.g. 4 maps

<x> <y> origin
Set the origin to (x,y).  X and Y are floating point numbers.
E.g. -0.354 .789 origin

<u> <v> screen
Set the screen origin to (u,v) (pixels).
E.g. 120 220 screen

<r> range
Set the range to <r> (floating point).  The viewing window is a
square with the lower left corner as the origin and side length
as range.
E.g. 0.5 range

<x> <y> scale
Set the x-axis and y-axis scales to the floating point values given.
The default scale is 1.0 for a full screen image.  Changing the scale
to a value less than one shrinks the image, greater than one expands
the image.
E.g. 2.0 2.0 scale

mouse
```

When issued, mouse will translate the position of the pointer into an x,y coordinate allowing the user to 'see' where certain parts of the image are. Clicking the mouse button will select that point as the new origin and decrease the range by a factor of two.

on|off axes
Turn the coordinate axes (really a mark on the origin) ON or OFF.
E.g. off axes

on|off outlines
Set showing the map outlines on or off, press a key to continue after viewing the outlines.
E.g. on outlines

settings
Show a list of the current origin, screen origin, range, scale, number of maps and whether the axes and outlines are on or off.
E.g. settings

findmap
Allows the user to enter three initial coordinates and three image coordinates and calculates the map for those values.
E.g. findmap

make
Puts the most recent values from findmap on the stack in order for set. The user needs to add the probability and map number before calling set.
E.g. make .333 1 set

<a> <b> <c> <d> <e> <f> <p> <n> set
Sets the parameters for a map. The letters a-f correspond to the values for the matrix and offset vector, <p> is the probability for the map and <n> is the map number. All values except <n> are to be floating point numbers.
E.g. 0.5 -0.5 0.5 0.5 0.0 0.0 0.5 1 set

<m1> <m2> copy
Copy map number <m1> to <m2> without disturbing <m1>.
E.g. 2 5 copy

<m> delete
Delete map number <m> and move any other maps up in memory.
E.g. 3 delete
<m> insert
Insert a blank map before map <m>.
E.g. 1 insert

cls
Clear the window.

```
E.g. cls

bye
Exit Fractal Lab Kit.
E.g. bye
```

3.3 Using the FindMap command

`FindMap` allows the user to find the parameters for a map by entering the coordinates and where they map to. This example will use the FindMap command to calculate the maps for the Mandelbrot Dragon.

With the program running enter `findmap`. You will see be asked to enter the coordinates of three points from the original map. The origin, the lower right and upper left corners of the initial box (remember, it goes from 0..1 in both x and y) make good starting places. Therefore, enter 0 for x_1 and 0 for y_1, (1,0) for (x_2, y_2), and (0,1) for (x_3, y_3). These are the default values. You do not need to enter a decimal point with each number in this case, pressing return uses the value displayed. For the image points, enter (x'_1, y'_1) as (0,0), (x'_2, y'_2) as (0.5,0.5) and (x'_3, y'_3) as (-0.5,0.5). The program will calculate the appropriate map and display its values. At the prompt, enter `make 0.5 1 set` to make this newly calculated map the first map. Enter `findmap` again and press return for each of the original points since we will use the same ones as before. For the image points enter (x'_1, y'_1) as (1,0), (x'_2, y'_2) as (0.5,0.5) and (x'_3, y'_3) as (0.5,-0.5). Then enter `make 0.5 2 set` to fix this as the second map. Lastly, enter `2 maps 120 220 screen cdraw` to use two maps, adjust the screen origin so the image will fit, and draw using color.

3.4 Writing your own fractals to disk

Unfortunately, *Fractal Lab Kit* is unable to write fractal maps to disk. Therefore, when you use `FindMap` to create a new map you must write it down by hand. When you are finished designing a new fractal you can use any text editor, like *TeachText*, to create a file to load the map from disk. Simply enter the skeleton code on the next few lines, substituting your values for the letters indicated. See the structure of the included fractals as an example.

```
: myfractal  ( give it a name, remember, comments in parentheses )
   a b c d e f p 1 set         ( enter the values for map 1 )
   a' b' c' d' e' f' p' 2 set  ( enter the values for map 2, etc. )
   n maps ;    ( 'n' is the number of maps you are entering )
```

Once loaded from disk, you need to enter the name of the fractal to load the maps and you are ready to draw.

Chapter 4

Advanced Features

4.1 A very (very) brief introduction to programming in Forth

Forth is an interpreted, stack based programming language known for its speed and extensibility. This is not an attempt to completely teach Forth so much as to teach a little about Forth so that the user who is unfamiliar with Forth can make some use of the language.

Forth is a stack based language, data is manipulated using a stack that works in a way very similar to the lunch trays in a cafeteria. The last tray in the stack is the first one out. Because of this, all mathematical operations are in postfix format, i.e., instead of typing `4 + 7` one would type `4 7 +` which would leave the value `11` on the top of the stack. This illustrates an important thing to remember about using Forth, anything that is entered is interpreted as either a word in the dictionary (more on that later) or a number to be pushed on the stack, so entering `4 7 +` told Forth to push a `4` on the stack followed by a `7` and the `+` word adds the top two stack items. To see the value stored on the top of the stack use the `.` word. Note that this is a destructive operation, it prints the value at the top of the stack and removes it from the stack as well. To see the top stack value but *not* remove it you need to enter `dup .` to first duplicate the top item and then print it. By default, Forth only operates on 16-bit integers but Pocket Forth supports real numbers as well. Forth will interpret a value as a real number `only` if it contains a decimal point! It is therefore important to enter a decimal point for every number that should be a real number and to `not` use one on numbers that should be integers.

Forth supports the standard arithmetic operations: `+`, `-`, `*`, `/` (integers) and `f+`, `f-`, `f*`, `f/` (real numbers). Use `f.` to print the top of stack as a real number. fdup duplicates the real number at the top of the stack while `fswap` will switch the top two real numbers on the stack. These few words will allow for using Forth as a simple calculator. Forth, typically, does not support higher mathematical functions, though *Pocket Forth* does.

Forth derives its extensibility from the way in which programs are written. As Forth interprets tokens from the input line (anything surrounded by spaces is a token) it either pushes it on the stack as a number or looks it up as a word in its dictionary. A

Forth program, therefore, consists of adding definitions to the dictionary. Definitions are begin with the : word and end with a ; word. Once defined, the word can be used in subsequent definitions. Parameters are passed via the stack. Forth does allow for the use of variables and constants, though these are slower than the stack. Use the word `variable` (or `fvariable`) followed by the name for the variable to create one. Use *value* `constant` (or `fconstant`) *name* to define a constant. Examples: typing `fvariable stddev` will create room in the dictionary for a floating point variable named `stddev` while typing `3.141592 fconstant PI` will create a constant for `pi`. Forth is case-insensitive. Constants are really special words that push the value on the stack so that typing `pi` will cause the value to be pushed on the stack. However, entering the name of a variable will NOT place its value on the stack, but rather, the address where the variable is stored will be placed on the stack. To get the value of a variable a two word combination must be used: `stddev f@` will 'fetch' the floating point number stored at the address that `stddev` places on the stack. Similarly, the value of an integer variable is found using `@` instead of `f@`. To store a value in a variable, use `!` or `f!`, `3 age !` or `1.414 sqr2 f!`.

Forth uses several standard control structures: `if else then`, `do loop` or `+loop`, `begin until`, and `begin while repeat`. The phrase

```
count @ 100 < if ." Yes, there is room" cr
              else ." No, there is no room." cr then
```

will check whether the current value of `count` is less than 100 or not. Forth supports <, >, and = for comparing integer values. Pocket Forth has a single word for comparing floating point numbers, `fcompare`, which returns a `-1` if $f1 < f2$, `0` if $f1 = f2$, and `+1` if $f1 > f2$, where $f1$ and $f2$ are the top two stack numbers (assumed to be floating point). It is important to note that unlike most other Forth words, `fcompare` does *not* remove the top two floating point numbers. The `do` loop is similar to the `for` loops in other languages. As might be expected, the syntax is

hi lo `do` *body* `loop`

where the index (pushed on the stack by the word `r`) will go from *lo* to *hi*`-1`. A variation is to use `+loop` instead of `loop` to jump by the value on the top of the stack (which must be positive). `Begin until` and `begin while repeat` are for bottom tested and top tested conditional loops. The condition is the same as for the `if` statement:

```
0 begin ." Hello" cr 1+ dup 99 > until
```

will print the word 'Hello' 100 times. Similarly, this fragment will also print 'Hello' 100 times:

```
0 begin dup 100 < while ." Hello" cr 1+ repeat
```

4.2 Putting it all together

The following examples will illustrate the creation of simple Forth words.

1. Averaging four numbers

```
:  ave4  ( a b c d -- average )  + + + 4 / ;
```

Using integer arithmetic, sum the top four stack items and divide the result by 4. Illustrates comments which are anything surrounded by '()', note the space after the '('. The comment given is known as a *stack effect comment* and shows the word's effect on the stack. Initial stack items are on the left of the '`--`' and the result is on the right.

2. Averaging N floating point numbers

```
:  averageN ( a1. ... aN. N -- average. )
dup >r          ( save N on the return stack )
1- 0 do f+ loop ( adjust N and add the values )
r>          ( get N off the return stack )
0 d>f f/ ;      ( make it real and divide to find average )
```

This example illustrates use of the return stack. The return stack is the place where Forth places addresses to return to when the current word is done executing. While a word is executing it is possible to use the return stack for temporary storage, but one must be careful to make sure that all values placed on the stack by `>r` are removed using `r>` before the word is done, otherwise Forth will attempt to return to who knows where and will very likely crash. Integers are transformed into real numbers by the two word sequence `0 d>f`. This transforms the integer into a double length integer and then into a real number.

3. Evaluating a function: $y = 3.4e^x$

```
fvariable x
: sqr  ( x. -- x.*x. )  fdup f* ;
: cube  ( x. -- x.^3  )  fdup fdup f* f* ;
: expf  ( x. -- exp[x.] )  ( use Taylor series approx. )
   fdup x f! 1.0 f+ x f@ sqr 2.0 f/ f+ x f@ cube 6.0 f/ f+ ;
: Y  ( x. --  Y[x.] )  expf 3.4 f* ; ( keep x < 1 for accuracy )
```

This is an example of *factoring*, the code for the square and cube could easily have been left in the definition of `expf` but factoring them out made the definition shorter and easier to read. In theory, according to some, a properly factored Forth program combined with well chosen word names and stack effect comments should be nearly self-documenting. By the way, Forth already has a word for calculating the exponential, `expt`. Compare `expf` with `expt` to see how large x can be before the error introduced by truncating the Taylor series at x^3 becomes too great.

These examples are brief, but hopefully should be sufficient, especially when combined with a list of Forth words, to allow you to write simple words to extend the power of the program.

4.3 The Sierpinski triangle, an example

The Sierpinski Triangle is a commonly seen fractal consisting of a triangle made of triangles. This will serve as an example of how the program can be extended by adding words to the Forth dictionary. The triangle is made up of three maps that divide the region 0..1 in x and 0..1 in y into three equal squares. We will develop a Forth word that will find the maps for the triangle and then generate the fractal. First, we must determine the maps. The findmap command's interactive nature is unsuited to our task, fortunately, there are three 'primitive' (i.e. non-interactive) words that will perform the same task: `initial`, `image`, and `solve`. These words operate as follows:

x1 y1 x2 y2 x3 y3 `initial`
Sets the initial points for finding a map, $(x_1, y_1), (x_2, y_2), (x_3, y_3)$.

X1 Y1 X2 Y2 X3 Y3 `image`
Sets the image points for finding a map,$(X_1, Y_1), (X_2, Y_2), (X_3, Y_3)$.

`solve`
Finds the values that will map the initial points to the image points.

These words, when combined with `make` and `set`, will allow us to create a single Forth word to find all three maps at once. At this point, then, we can write:

```
: sierpinski  ( generates the Sierpinski Triangle)
( Set up initial values, for all maps )
0.0 0.0 1.0 0.0 0.0 1.0 initial
( First map )
0.0 0.0 0.5 0.0 0.0 0.5 image  solve
make 0.333 1 set
( Second map )
0.5 0.0 1.0 0.0 0.5 0.5 image  solve
make 0.333 2 set
( Third map )
0.25 0.5 0.75 0.5 0.25 1.0 image  solve
make 0.333 3 set
```

Now before viewing the fractal show the maps, reset the program, show the settings, and draw the outlines:

```
( Show the maps )
page ." The Sierpinski maps: " cr showmaps
reset  settings
key drop  ( wait for a key press )
( Show outlines when drawing )
on outlines
( Reset the program and draw the fractal )
cdraw  ;
```

Try this word and see what happens.

4.4 Basic Forth words

Table 4.1 lists of some basic Forth words and their use. With these it should be possible to define your own words for use with the program.

Table 4.1: Basic Forth words. Words in **bold** are specific to *Pocket Forth*. The `save` command makes a permanent change in the application and should only be used on a copy. It is equivalent to selecting *Save Dictionary* from the *File* menu.

Word	Stack effect	Use
swap	(a b – b a)	Switch top two stack items
dup	(a – a a)	Duplicate top of stack
over	(a b – a b a)	Bring 2nd to top
rot	(a b c – b c a)	Rotate stack items
variable	(–)	Make a variable of next token
constant	(a –)	Make a constant of next token
+, -, *, /	(a b – a.b)	Math, where . is an operation
mod	(a b – a mod b)	Remainder after dividing
drop	(a –)	Drop the top stack item
cr	(–)	Print a return character
space	(–)	Print a space (ASCII 32)
emit	(a –)	Print the character whose code is on the stack
."	(–)	Print text until a " found (definitions only)
.	(a –)	Print the top of stack
key	(– a)	Get a key, ASCII code on stack
bye	(–)	Exit from Forth
?terminal	(– b)	Has a key been pressed?
(	(–)	Start a comment (remember space)
!pen	(x y –)	Move the pen to (x,y) (pixels)
-to	(x y –)	Line from current to (x,y)
page	(–)	Clear the screen
-->	(–)	Load filename, no spaces!
open	(–)	Load file chosen in Mac dialog
?button	(– t)	Mouse button down?
@mouse	(– x y)	Push mouse position on stack
save	(–)	Save the current dictionary

See the reference section for more words that are program specific. Words in bold text are special to Pocket Forth and may not be available on other Forth systems, though there will likely be something similar. Those interested in seriously learning Forth (some swear that it is the best computer language there is) should get a hold of the book *Starting Forth* by Leo Brodie (2nd ed. 1987), it is an excellent and entertaining introduction.

Chapter 5

Using the Shell

Fractal Lab Kit contains a mouse driven shell. The shell is a single Forth word (named *shell*) that allows you to use the mouse to set many of the parameters used in generating fractals. The shell makes it easy to change settings, colors, edit and calculate maps, and draw. The only thing the shell cannot do is load fractals from disk. The use of the shell is straightforward, simply click on the command you want to execute. Numerical data is entered at the bottom of the screen. When more than one item of information needs to be entered you must press the 'return' key after each item. The application menu bar is disabled when the shell is running, use the *Forth* or *Quit* commands to exit the shell or program.

The shell uses much of the memory available to Forth. This might cause a problem if you are creating more sophisticated programs. You can eliminate the shell code by entering `forget shellcode` when in interactive mode. This is exactly what the `Cathedral` demo program does. The code is erased from Forth's memory and will be permanently erased should the word `save` be entered after the code is cleared. Alternatively, one could select *Save Dictionary* from the *File* menu instead of using the word `save`. This will write the current Forth dictionary into the application itself. This should never be a problem since everyone runs from backup copies only, right? This is also a way to create a version of *Fractal Lab Kit* that has more memory for your own programs.

If you are designing a new fractal the shell will be handy since it lets you change values quickly. Write your maps down though since *Fractal Lab Kit* has no ability to save maps to disk! If you are manipulating a fractal stored on disk you must load it in interactive mode (i.e. go to *Forth*), enter its name to load the maps, and then enter the shell. For example, the following lines in interactive mode will load the fern maps from disk and enter the shell to allow you to work with the fractal:

```
--> :maps:fern                (load the maps from disk)
fern shell                    (load the maps & enter the shell)
```

Try switching the signs of the `a` and `c` values in the first fern map to change the fern into a weed. Use the *Help* command for a quick summary of the commands available in the shell. Note that the help screen in interactive mode is different than the one in shell mode.

Appendix A

Reference

A.1 Commands

draw
Draw a fractal based on the current maps. Press a key to stop.
E.g. green color fern draw

cdraw
Except for plotting each map in a different color, it is the same as draw.
E.g. spiral cdraw

<n> idraw or <n> icdraw
Same as draw and cdraw respectively except for iterating through <n>*500 points. Useful for drawing a fractal and stopping without user interaction.

<n> edit
Edit the n-th map. Enter a new value or press return to leave the existing value as is.
E.g. 2 edit
  a = 0.5 ?-0.5 (new value)
  b = -0.5 ?<return>   etc.

zero-maps
Erase all twelve maps.
E.g. zero-maps

<color.name> color
Set the current drawing color to the color named. Valid colors are black, wite, red, green, blue, yellow, cyan, magenta
E.g. magenta color

<n> maps
Set the number of maps to use to <n>.

E.g. 4 maps

<x> <y> origin
Set the origin to (x,y). X and Y are floating point numbers.
E.g. -0.354 .789 origin

<u> <v> screen
Set the screen origin to (u,v) (pixels).
E.g. 120 220 screen

<r> range
Set the range to <r> (floating point). The viewing window is a square with the lower left corner as the origin and side length as range.
E.g. 0.5 range

<x> <y> scale
Set the x-axis and y-axis scales to the floating point values given. The default scale is 1.0 for a full screen image. Changing the scale to a value less than one shrinks the image, greater than one expands the image.
E.g. 2.0 2.0 scale

mouse
When issued, mouse will translate the position of the pointer into an x,y coordinate allowing the user to 'see' where certain parts of the image are. Clicking the mouse button will select that point as the new origin and decrease the range by a factor of two.

on|off clear
Turn clearing of page before drawing on and off. Default is on.

on|off axes
Turn the coordinate axes (really a mark on the origin) ON or OFF.
E.g. off axes

on|off outlines
Set showing the map outlines on or off, press a key to continue after viewing the outlines.
E.g. on outlines

settings
Show a list of the current origin, screen origin, range, scale, number of maps and whether the axes and outlines are on or off.
E.g. settings

findmap
Allows the user to enter three initial coordinates and three image coordinates and calculates the map for those values.

E.g. findmap

make
Puts the most recent values from findmap on the stack in order for set. The user needs to add the probability and map number before calling set.
E.g. make .333 1 set

<a> <b> <c> <d> <e> <f> <p> <n> set
Sets the parameters for a map. The letters a-f correspond to the values for the matrix and offset vector, <p> is the probability for the map and <n> is the map number. All values except <n> are to be floating point numbers.
E.g. 0.5 -0.5 0.5 0.5 0.0 0.0 0.5 1 set

<m1> <m2> copy
Copy map number <m1> to <m2> without disturbing <m1>.
E.g. 2 5 copy

<m> delete
Delete map number <m> and move any other maps up in memory.
E.g. 3 delete

<m> insert
Insert a blank map before map <m>.
E.g. 1 insert

cls
Clear the window.
E.g. cls

bye
Exit Fractal Lab Kit.
E.g. bye

mem
Show the available dictionary space. (Forth only has 32k)
E.g. mem

?origin, ?screen, ?scale, ?range, ?maps, ?axes, ?outline, ?clear
Show individual settings. settings calls each of these.

<n> show
Show the values of map <n>.
E.g. 3 show

A.2 Primitive commands

These commands (words) are 'primitive' in the sense that they are called by the regular command words. As they are standard Forth words they are available to the user as well.

```
input   ( -- a )
Get a 16-bit integer on the stack.

finput  ( -- f b )
Get a floating point number on the stack and a boolean value that is
false if the user pressed the return key only, in which case the
number is 0.0.

#map->addr  ( a -- addr )
Convert a number for a map into an address to the map location in
memory.

get   ( offset map# -- value )
Get a value for a particular map.  The offset is a branch into the
map, each value is 10 bytes long.  The constants a,b,c,d,e,f and p
are defined to give the proper offset:  c 3 get  returns the c value
of the third map.

update  ( value offset map# -- )
Put the value in the numbered map at the offset (use  a-f or p).

print  ( addr -- )
Print the map starting at addr.

wsize   ( h v -- )
Resize the window to h pixels high and v pixels across.

dot   ( u v -- )
Draw a dot on the screen at (u,v) (pixels).

plot   ( x. y. -- )
Plot the point (x,y) on the screen.

plotto   ( x. y. -- )
Draw a line from the last plotted point to (x,y).

determinant  ( -- d )
Find the determinant of the 3x3 matrix whose values are stored
in the floating point variables d1 through d9.  Values in the form:
[ [d1,d2,d3],[d4,d5,d6].[d7,d8,d9]].

x->d  ( -- )
Copy the values in x1,y1 .. x3,y3 to the matrix d.  Used to setup for
```

finding a map.

xy->uv (x. y. -- u v)
Change real coordinates (x,y) into screen coordinates (u,v). Call factor first.

factor (--)
Calculates redundant factors for xy->uv to speed drawing.

firstpoints (--)
Get the initial points for a map, interactive.

solve3x3 (-- f. d. c. e. b. a.)
Solve for a map, calculated values on stack. Call either firstpoints and imagepoints or initial and image before calling solve3x3.

outputmap (f. d. c. e. b. a. --)
Display map values on the stack on the screen.