Bifurcation Diagrams

Kiyoshi Akima
k_akima@hotmail.com

2006.04.08

Contents

1 Bifurcation Diagrams

Chaos has strange attractions for the mind that can see patterns therein. Some physical systems that exhibit chaotic behavior do so because they are in a sense attracted to such patterns. As a bonus, the patterns themselves are strangely attractive. Some readers may already be aware that the geometric forms underlying choas are called *strange*, or *chaotic* attractors. Strange attractors can be generated with a home computer or even, as we will see in this document, a pocket calculator.

Before setting off on this journey, readers must be equipped with a protective coating of physical intuition. In particular, what is an attractor? Roughly speaking, an attractor is a generalization of the notion of equilibrium; an attractor is what the behavior of the system settles down to, or is attracted to. The pendulum is a simple physical system that illustrates the concept of an attractor. Supose an ordinary pendulum moves under frictional forces that slow it eventually to a standstill. One can describe the pendulum's motion by means of a so-called *phase*, or *state*, diagram in which the angle the pendulum makes with the vertical is graphed against the rate at which the angle changes. The swinging motion of the pendulum is represented by a point circling the origin in the phase diagram; as the pendulum loses energy, the point spirals into the origin, where it ultimately comes to rest. In this case the origin is called an *attractor* because it seems to attract the moving point in the phase diagram. Readers would be correct in thinking there is nothing strange about an attractor consisting of a single point.

A slighly more complicated attractor underlies the motion of a grandfather clock. Here an escapement mechanism feeds energy to a pendulum to keep it from slowing down. If one starts the clock with an overly energetic push of the pendulum, it slows down to the tempo prescribed by the escapement but thereafter slows no further. If the clock is started with a push that is too gentle, however, the pendulum behaves like an ordinary one; it slows to a standstill as before. In the case of the overly energetic push, the pendulum's motion is a phase diagram is a spiral that winds ever more tightly about a circular orbit. Here the attractor is a circular loop. In this context a circle is no stranger than a point.

An ordinary pendulum can be made to show chaotic behavior by introducing a vertical, vibratory motion: If the point of support is moved up and down in a sinusoidal manner by an electric motor, the pendulum may begin to swing crazily, exhibiting no evidence of periodic behavior whatever.

For our journey into chaos, however, we will explore a different physical system. Imagine an arrangement of three amplifiers in which the first amplifier outputs a signal x that is fed to the other two. The second amplifier outputs the signal $1 - x$ in response to x. The third amplifier takes the two signals, x and $1 - x$, as input. It generates the product, $x(1 - x)$, of the two signals and feeds it back to the first amplifier, which also receives a control voltage, r, as input. One additional component, a device that samples its input and delivers the same voltage as output for a short time, completes the circuit; it is inserted

in the output line from the first amplifier. The three-amplifier circuit does a voltage dance that becomes more chaotic as the control voltage r is gradually increased.

When r is less than 3 and x initially has some nonzero setting, the circuit oscillates briefly before it settles down to a specific value of x that remains the same thereafter. This value value constitutes a single point attractor. If the control voltage, r, is now raised to a level just above 3, the circuit flutters between two values of x. At this level of r the circuit is said to be *bistable* and the attractor consists of two points. As r is increased further, the circuit oscillates among four points; yet another increase yields an eight-point attractor. The pattern of doubling and redoubling goes on as the knob controlling r is turned to higher values, until at a setting roughly midway between 3 and 4 the circuit suddenly goes crazy. It hunts endlessly at electronic speed for the simple recurring patterns that marked its earlier existence. Its behavior is now governed by a strange attractor that has a potential infinity of values. The result is chaos.

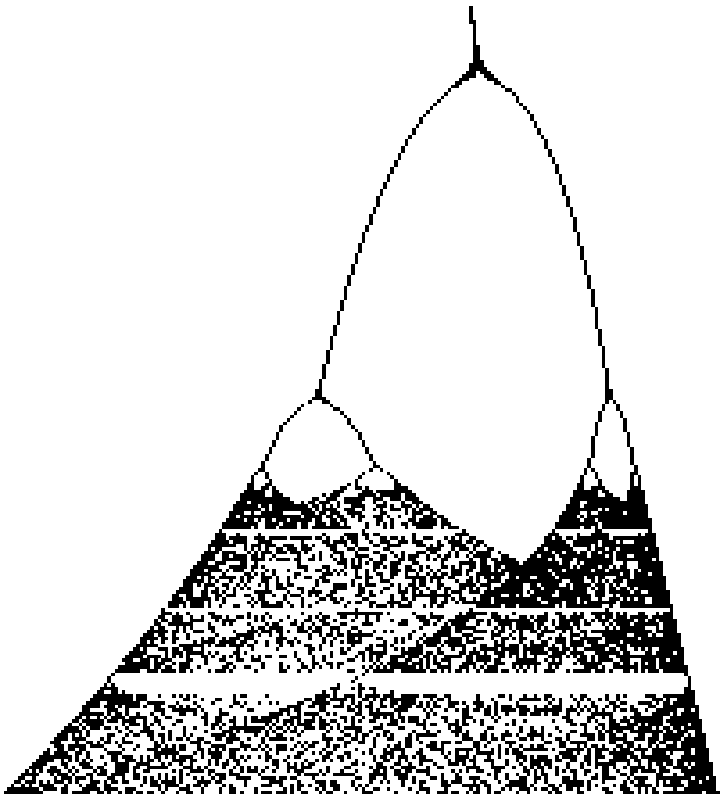
Electronically ilterate readers may be tempted to build such a circuit. The rest of us may simulate it on a computer of any size, viewing the dance with great clarity on the display screen. To do so, we merely need to write a program that computes the iterated equation $x \leftarrow rx(1-x)$. The program has a core that consists of six steps:

$x \leftarrow .5$
for $i \leftarrow 1$ to 100
    $x \leftarrow rx(1-x)$
for $i \leftarrow 1$ to 300
    $x \leftarrow rx(1-x)$
    plot(r, x)

The variable x starts at the value .5 The program then enters a loop that iterates the basic equation 100 times to allow transients to die away. The transients are inherent in the equation itself, not in imprecise arithmetic. The program then enters a new loop that iterates the equation 300 more times, plotting the value of x on each occasion.

A complete picture of the behavior of the simple amplifier circuit emerges if the program computes a raft of plots, each plot below the previous one. The plots results from a succession of r values that run from 2.9 to 4.0 in, say, 600 steps from the top of the screen to the bottom. A more elegant picture emerges if more steps are used, but in this case the diagram will probably not fit on your screen and will have to be scrolled to be seen.

For values of r less than 3.56 (a more precise value is 3.56994571869) the attractors of the simple dynamical system embodied in the iterated equation $x \leftarrow rx(1-x)$ consists of a few points. These points, which represent nonchaotic behavior, are arranged in three large bands and an infinity of smaller ones. The attractors become strange as r approaches 3.56. Here chaos sets in as the hiterto smoothly bifurcating lines suddenly fall into a pepper-and-salt madness. Strangely enough, the chaotic regime vanishes from time to time as r continues its inexorable march to 4.



The entire plot is called a *bifurcation diagram*. The plot is embellished by curves and attractively shaped folds. The reasons for ornamental details are mysteries that can be explained only by the theory of chaos.

Much of the structure of the bifurcation diagram has been analyzed by chaos theorists. The boundaries of the chaotic regions are set by the minimum and maximum values of the iterates of $x = .5$. The curves followed by the minima and maxima, as well as those followed by the "veils" that hang so strangely in the chaotic regions, are all simple polynomials in r. At the places where the shading is densest one finds the highest concentration of points in the strange attractors taht cross them. In the empty bands mentioned above chaos gives way to order. Theory tells us that for every whole number there is a band (however narrow) with orbits of precisely that size. Finally, it will come as no surprise to readers familiar with chaos that strange attractors, even in the humble system just explored, have a fractal nature; an infinite number of points show interesting detail at all levels of magnification, like the famous Mandelbrot set.

2 User Programming for the HP 48

A bifurcation diagram can be generated on just about any computer with a graphical output device. The device we'll use in this document is not a computer but the HP 48 handheld graphics calculator.

The HP 48 has a 131×64 display, small by modern computer standards but adequate for our immediate needs. The calculator is user-programmable from the keyboard, in a language called *User RPL*. This language is descended from the *RPN* language used by earlier Hewlett-Packard calculators, with additional elements of *Forth* and *LISP*.

There are additional programming schemes available, one of which we will discuss later. But for now, let's start off by writing a program in User RPL. This program takes no input from the user and draws a bifurcation diagram filling the screen from left to right.

It's a relatively straighforward process to implement the pseudocode on page 1. To make it a little faster, we will only iterate 70 times to dampen the initial transients. And, due to the small size of the display, we will only plot 30 points. These numbers can be easily changed (they're the numbers before the START s). Due to the shape of the screen, we will also turn the plot on its side.

BI 254.5 Bytes Checksum 2180h

```
«
    PICT PURGE ( #0 #0 ) PVIEW
    (2.9,0) PMIN (4,1) PMAX
    0 130 FOR i
        i 130 / 1.1 * 2.9 + .5
        1 70 START
            1 OVER - * OVER *
        NEXT
        1 30 START
            1 OVER - * OVER *
            DUP2 R→C PIXON
        NEXT
        DROP2
    NEXT
    7 FREEZE
»
```

Even with the reduction in the number of iterations, this program takes about eight minutes to run on an HP 48GX. Saying that the plot crawls across the screen might be generous. Can we do better than this?

We can and we will, by switching to System RPL.

3 System Programming for the HP 48

The next step from User RPL is *System RPL*. This is an extension to User RPL, but requires external tools. Usually these tools are run on a computer and the resultant programs downloaded to the calculator.

There are many development environments available for the task. The one we will use is the original set of tools published by Hewlett-Packard. This set consists of an RPL compiler (`RPLCOMP`), a Saturn assembler (`SASM`), and a Saturn linker (`SLOAD`). These tools run on an IBM-compatible PC running DOS or Windows. Similar tools are available for the Mac and Unix. This document is not intended as a tutorial on System RPL, but rather as a simple illustration.

3.1 Initial Translation

To get our feet wet with System RPL, let's just try a straightfoward translation of the User RPL program.

5a ⟨*bi1.s* 5a⟩≡

```
⟨prologue 5b⟩
::
    ⟨bi1: initialize 5c⟩
    ⟨bi1: for 131 columns 6a⟩
        ⟨bi1: calculate r 6b⟩
        ⟨bi1: loop 70 times to dampen transients 6d⟩
        ⟨bi1: plot 30 points 6f⟩
    LOOP
    ⟨bi1: freeze display 7a⟩
;
```

3.1.1 Prologue

All System RPL programs require a prologue. This assembles a header that tells the calculator how to handle the program when it receives it from the computer.

5b ⟨*prologue* 5b⟩≡ (5a 8a 11a 13a 16a)

```
ASSEMBLE
    NIBASC  /HPHP48-D/
RPL
```

3.1.2 Initialization

The first step is to initialize the graphics display. We have to work with physical coordinates, so there is no logical-to-physical mapping involved.

5c ⟨*bi1: initialize* 5c⟩≡ (5a)

```
TOGDISP ZEROZERO SIXTYFOUR XHI MAKEGROB XYGROBDISP TURNMENUOFF
```

3.1.3 Main Loop

Then comes the main loop. Note that the loop limits are reversed from User RPL.

6a ⟨*bi1: for 131 columns* 6a⟩≡ (5a)

```
XHI ZERO_DO (DO)
```

3.1.4 Calculate r

Within the main loop, we're going to need the value of r.

6b ⟨*bi1: calculate r* 6b⟩≡ (5a) 6c▷

```
INDEX@ UNCOERCE 130. %/ 1.1 %* 2.9 %+
```

We also need the starting value for x.

6c ⟨*bi1: calculate r* 6b⟩+≡ (5a) ◁6b

```
%.5
```

3.1.5 Dampen Transients

We repeat the basic iteration 70 times to dampen the initial transients.

6d ⟨*bi1: loop 70 times to dampen transients* 6d⟩≡ (5a)

```
SEVENTY ZERO_DO (DO)
    ⟨iterate 6e⟩
LOOP
```

The basic iteration is the same as in the User RPL program, merely translated to System RPL.

6e ⟨*iterate* 6e⟩≡ (6 9a 14c)

```
%1 OVER %- %* OVER %*
```

3.1.6 Plot Points

With the initial transients taken care of, we plot 30 points.

6f ⟨*bi1: plot 30 points* 6f⟩≡ (5a) 6h▷

```
THIRTY ZERO_DO  (DO)
    ⟨iterate 6e⟩
    ⟨bi1: plot one point 6g⟩
LOOP
```

Plotting a point requires a little more work since we have to convert from our logical coordinates to physical coordinates.

6g ⟨*bi1: plot one point* 6g⟩≡ (6f)

```
JINDEX@ 63. 3PICKOVER %* %- COERCE PIXON3
```

When we're done plotting, we need to drop the x and r values from the stack.

6h ⟨*bi1: plot 30 points* 6f⟩+≡ (5a) ◁6f

```
2DROP
```

3.1.7 Freeze Display

Once we've created the plot, we want to freeze the display so it remains there until the user presses a key.

7a ⟨*bi1: freeze display* 7a⟩≡ (5a)

```
SetDAsTemp
```

3.2 Building the Program

As mentioned earlier, this program cannot be keyed directly into the HP 48 but must first be built on a computer and then transferred to the calculator. A brief overview of the process is given here.

3.2.1 Tangling

This program and subsequent ones are written in a style called *Literate Programming* (see Appendix A for more information). The first step is to extract the program source from the document, a process called *tangling*.

This command will extract the source for this program from the document:

```
notangle -Rbi1.s -t4 bifurc48.nw >bi1.s
```

3.2.2 Building

The RPL compiler, the Saturn assembler, and the Saturn linker must all be run on the program source. The following command lines will do this:

```
rplcomp bi1.s bi1.a
sasm bi1.a
sload -H bi1.m
```

The linker needs a command file telling it what to do. The following command file can be extracted with this command line:

```
notangle -Rbi1.m -t4 bifurc48.nw >bi1.m
```

7b ⟨*bi1.m* 7b⟩≡

```
LL BI1.LR
OU BI1
RE BI1.O
SE C:\HP48\LIB\ENTRIES.O
SU XR
```

For further details on the use of the tools, please consult the appropriate documentation.

3.3 Running the Program

Once the program has been built and downloaded to the calculator, it can be stored in a variable and run like any other program. Press the ON key to terminate.

The program is 159.5 bytes in size and runs in about 135 seconds. A 35% reduction in size and a whopping 70% reduction in runtime compared to the User RPL version. Not bad for a straighforward translation.

4 Adding Scrolling

We could speed up the program even more by coding in assembler, but there's only 28% of the original runtime remaining. Instead, let's try adding some features, even at the expense of some speed.

As a first stab, let's try creating a larger picture. Since the picture already fills the entire screen, we'll have to implement a scheme for the user to scroll the image.

We don't want to make the picture too big or it'll take too long to draw, wiping out our speed gains. Let's turn the image upright, make it 221 pixels high to cover the range of r from 2.9 to 4 and 201 pixels wide.

Due to the higher resolution, we'll need more iterations to dampen out the initial transients so we'll repeat the loop 80 times. And due to the additional real estate, we'll plot 140 points instead of a mere 30.

8a ⟨*bi2.s* 8a⟩≡

```
INCLUDE C:\HP48\LIB\KEYDEFS.H
⟨prologue 5b⟩
::
    ⟨bi2: initialize 8b⟩
    221 ZERO_DO (DO)
        ⟨bi2: draw picture 9a⟩
    LOOP
    ⟨bi2: allow scrolling 9b⟩
    ⟨bi2: terminate 10b⟩
;
```

4.1 Initialization

This time we're going to follow the programming guidelines and use the stack display instead of the graphics display.

8b ⟨*bi2: initialize* 8b⟩≡ (8a 11b)

```
RECLAIMDISP ClrDA1IsStat
ZEROZERO 221 201 MAKEGROB XYGROBDISP TURNMENUOFF
```

4.2 Drawing

Since you've already seen how to do this, we'll present this part of the program without much commentary. The main difference, aside from the loop controls, is in the plotting, which has been turned on its side.

9a ⟨*bi2: draw picture* 9a⟩≡ (8a 12a)

```
⟨bi2: break 10a⟩
INDEX@ UNCOERCE 200. %/ 2.9 %+ %.5
EIGHTY ZERO_DO  (DO)
    ⟨iterate 6e⟩
LOOP
140 ZERO_DO (DO)
    ⟨iterate 6e⟩
    200. OVER %* COERCE JINDEX@ PIXON
LOOP
2DROP
```

4.3 Scroll

This scrolling code is taken directly from the `SCROLL` example included with the HP tools. It uses a *Parameterized Outer Loop* to handle the keyboard input. The arrow keys scroll the display one pixel in the indicated direction, while the right-shifted arrow keys scroll to the edge of the picture in that direction. The ON key terminates the program and restores the display.

9b ⟨*bi2: allow scrolling* 9b⟩≡ (8a 11a 13a 16a)

```
FALSE { LAM Exit } BIND ' NOP
' ::
    kpNoShift #=casedrop ::
        kcUpArrow ?CaseKeyDef       :: TakeOver SCROLLUP ;
        kcLeftArrow ?CaseKeyDef     :: TakeOver SCROLLLEFT ;
        kcDownArrow ?CaseKeyDef     :: TakeOver SCROLLDOWN ;
        kcRightArrow ?CaseKeyDef    :: TakeOver SCROLLRIGHT ;
        kcOn ?CaseKeyDef            :: TakeOver TRUE ' LAM Exit STO ;
        kcRightShift #=casedrpfls
        DROP 'DoBadKeyT
    ;
    kpRightShift #=casedrop ::
        kcUpArrow ?CaseKeyDef       :: TakeOver JUMPTOP ;
        kcLeftArrow ?CaseKeyDef     :: TakeOver JUMPLEFT ;
        kcDownArrow ?CaseKeyDef     :: TakeOver JUMPBOT ;
        kcRightArrow ?CaseKeyDef    :: TakeOver JUMPRIGHT ;
        kcRightShift #=casedrpfls
        DROP 'DoBadKeyT
    ;
    2DROP 'DoBadKeyT
;
TrueTrue NULL{} ONEFALSE ' LAM Exit ' ERRJMP ParOuterLoop
```

4.4 Interrupting the Program

If you've run the previous program, you might have noticed that you couldn't terminate the program until it had finished creating the diagram. This isn't very polite on the part of such a long-running program. So, let's allow the ON key to interrupt the drawing process and terminate the program.

This code is inserted at the top of the main loop. It will terminate the main loop and drop through into the scrolling code, which will also see that the ON key has been pressed and terminate the program.

10a ⟨*bi2: break* 10a⟩≡ (9a 14c)

```
ATTN? IT :: ZERO ISTOPSTO ;
```

4.5 Termination

The termination differs because we're using the stack display.

10b ⟨*bi2: terminate* 10b⟩≡ (8a 11a 14e)

```
TURNMENUON RECLAIMDISP ClrDAsOK
```

4.6 Linker Command File

As before, this program needs a linker command file.

10c ⟨*bi2.m* 10c⟩≡

```
LL BI2.LR
OU BI2
RE BI2.O
SE C:\HP48\LIB\ENTRIES.O
SU XR
```

4.7 Running the Program

This program, which is 424 bytes in length, now takes more than nine-and-a-half minutes to produce the diagram. Even though it's actually slower than the original User RPL program, it generates a much larger and more detailed image. Once the hourglass goes away, you can use the arrow keys to scroll around the image. The right-shifted arrow keys will jump you to the edge of the image. The image is detailed enough to display some of the thinner bands of non-chaotic behavior.

When you're done playing with it, the ON key will terminate the program.

5 Indicating Progress

It's a little hard to tell how far the previous program has progressed. The screen is blank for half a minute before the first point shows up at the right edge. And once the single limb disappears off the bottom edge, there's nothing to indicate how close the diagram is to being completed. The only thing to indicate completion is the disappearance of the hourglass.

There must be a better way.

We could put a progress meter on the screen but that would be kind of boring. We would much rather show the user the diagram being generated on-screen. And we can do just that. If you jump to the right edge of the picture and scroll top-to-bottom, you can see that a portion of the diagram is always visible. And the dots approach the right edge as we go from top to bottom. So, if we show the right edge of the diagram instead of the left, and scroll as we generate it...

Much of the program is the same as before, so we can just reuse the code previously developed. We just have one additional step in the initialization, and we want to scroll the diagram as we create it.

11a ⟨*bi3.s* 11a⟩≡

```
INCLUDE C:\HP48\LIB\KEYDEFS.H
⟨prologue 5b⟩
::
    ⟨bi3: initialize 11b⟩
    ⟨bi3: draw picture 11d⟩
    ⟨bi2: allow scrolling 9b⟩
    ⟨bi2: terminate 10b⟩
;
```

5.1 Initialization

Since the image is the same size as in the previous program, we can use the same code to initialize it.

11b ⟨*bi3: initialize* 11b⟩≡ (11a) 11c▷

```
⟨bi2: initialize 8b⟩
```

But before we draw anything, we want to move to the right edge of the image.

11c ⟨*bi3: initialize* 11b⟩+≡ (11a) ◁11b

```
JUMPRIGHT
```

5.2 Drawing

The only difference in the drawing code is that we want to scroll the screen one pixel after we've done the first 64 rows.

11d ⟨*bi3: draw picture* 11d⟩≡ (11a) 12a▷

```
221 ZERO_DO (DO)
    SIXTYTHREE INDEX@ #>?SKIP SCROLLDOWN
```

The remainder of the drawing code is identical to the previous program.

12a ⟨*bi3: draw picture* 11d⟩+≡ (11a) ◁11d
⟨*bi2: draw picture* 9a⟩
```
LOOP
```

5.3 Linker Command File

As before, this program needs a linker command file.

12b ⟨*bi3.m* 12b⟩≡
```
LL BI3.LR
OU BI3
RE BI3.O
SE C:\HP48\LIB\ENTRIES.O
SU XR
```

5.4 Running the Program

The program is now 436.5 bytes in size and runs in nearly the same time as the previous one, taking perhaps a second longer because of the scrolling. But now we can see the bifurcation diagram being generated. And once the diagram has been fully drawn, we can scroll it around as before.

And as before, the ON key will terminate the program.

6 Program Arguments

All of the programs presented thus far take no arguments. The size of the diagram and the number of points to be plotted are coded directly in the programs. Now it's time to let these values be specified at runtime.

13a ⟨*bi4.s* 13a⟩≡

```
INCLUDE C:\HP48\LIB\KEYDEFS.H
ASSEMBLE
    CLRLIST ALL
    SETLIST ALL
RPL
⟨prologue 5b⟩
::
    ⟨bi4: initialize 13b⟩
    ⟨bi4: process arguments 13c⟩
    ⟨bi4: draw picture 14c⟩
    ⟨bi2: allow scrolling 9b⟩
    ⟨bi4: terminate 14d⟩
;
```

6.1 Initialization

The first thing we need to do is to clear the saved command name and then ensure that there are three real arguments on the stack.

13b ⟨*bi4: initialize* 13b⟩≡ (13a)

```
0LASTOWDOB! CK3NOLASTWD CK&DISPATCH1 3REAL ::
```

6.2 Argument Processing

Now that we've got the proper number and type of arguments, let's store them away for future use.

13c ⟨*bi4: process arguments* 13c⟩≡ (13a 16a) 13d▷

```
{ NULLLAM NULLLAM NULLLAM } BIND
```

Then let's initialize the display. It's the same process as before, except that the size is taken from the program arguments instead of being coded directly into the program.

As before, we'll show the right edge of the diagram as it's being generated. Note that if the image is *too* big, nothing may appear on the screen for a while.

13d ⟨*bi4: process arguments* 13c⟩+≡ (13a 16a) ◁13c 13e▷

```
RECLAIMDISP ClrDA1IsStat ZEROZERO
2GETLAM 3GETLAM COERCE2 MAKEGROB XYGROBDISP TURNMENUOFF JUMPRIGHT
```

Now we need the upper limit for the main loop. This value will be left on the stack.

13e ⟨*bi4: process arguments* 13c⟩+≡ (13a 16a) ◁13d 14a▷

```
2GETLAM COERCE
```

Then we need to compute some plotting parameters.

14a ⟨*bi4: process arguments* 13c⟩+≡ (13a 16a) ◁13e 14b▷
```
3GETLAM %1- 3PUTLAM
2GETLAM %1- 1.1 %/ 2PUTLAM
```

And the upper limit for the plotting loop.

14b ⟨*bi4: process arguments* 13c⟩+≡ (13a 16a) ◁14a
```
1GETLAM COERCE 1PUTLAM
```

6.3 Drawing

The drawing code follows the pattern set by the previous programs, the main difference being that various parameters are taken from the temporary environment rather being wired directly into the program.

14c ⟨*bi4: draw picture* 14c⟩≡ (13a 16a)
```
ZERO_DO (DO)
    ⟨bi2: break 10a⟩
    SIXTYTHREE INDEX@ #>?SKIP SCROLLDOWN
    INDEX@ UNCOERCE 2GETLAM %/ 2.9 %+ %.5
    EIGHTY ZERO_DO  (DO)
        ⟨iterate 6e⟩
    LOOP
    1GETLAM ZERO_DO (DO)
        ⟨iterate 6e⟩
        3GETLAM OVER %* COERCE JINDEX@ PIXON
    LOOP
    2DROP
LOOP
```

6.4 Termination

We need to dispose of the temporary environment before terminating the program. This requires only a single instruction.

14d ⟨*bi4: terminate* 14d⟩≡ (13a 16a) 14e▷
```
ABND
```

Most of the rest of the termination process is the same as before.

14e ⟨*bi4: terminate* 14d⟩+≡ (13a 16a) ◁14d 14f▷
```
⟨bi2: terminate 10b⟩
```

Then we need an additional `SEMI` to close the argument dispatch.

14f ⟨*bi4: terminate* 14d⟩+≡ (13a 16a) ◁14e
```
;
```

6.5 Linker Command File

As before, this program needs a linker command file.

15 ⟨*bi4.m* 15⟩≡

```
LL BI4.LR
OU BI4
RE BI4.O
SE C:\HP48\LIB\ENTRIES.O
SU XR
```

6.6 Running the Program

The program is now 483.5 bytes in size, nearly double that of the original User RPL program. The runtime varies on the size of the diagram and the depth of the plotting loop. With the arguments to duplicate the previous program:

201 ENTER 221 ENTER 140 BI4

it's a little slower, but still generates the diagram in under ten minutes. But now we're not limited to fixed sizes nor to fixed iteration depths: You can explore the bifurcation diagram to your heart's content.

Of course, we're still limited by the calculator's memory and its relative slowness. However, running this program on an emulator on a faster computer will get around one of the drawbacks. The other drawback will require a system with a more capable display.

7 Input Form

Let's take the evolution of this program one step further. The previous version required the user to provide arguments on the stack, in a specific order. Put them in the wrong order and, while the program won't necessarily crash, the image will not be as desired.

One way to avoid this is to make the program ask the user for the image parameters. And the easiest way to do this, at least from the user's viewpoint, is through an *input form*. Unfortunately, the input form was introduced with the `G` and `GX` versions of the calculator; if you have the `S` or `SX`, you can skip the rest of this section. For those of you with the `G` or `GX`, most of the program remains unchanged from the previous version.

16a ⟨*bi5.s* 16a⟩≡

```
INCLUDE C:\HP48\LIB\KEYDEFS.H
ASSEMBLE
    CLRLIST ALL
    SETLIST ALL
RPL
⟨prologue 5b⟩
::
    ⟨bi5: get user input 16b⟩
    ⟨bi4: process arguments 13c⟩
    ⟨bi4: draw picture 14c⟩
    ⟨bi2: allow scrolling 9b⟩
    ⟨bi4: terminate 14d⟩
;
```

7.1 Using an Input Form

Creating an input form requires a lot of parameters and we won't try to describe them all. Please consult the appropriate documentation for further details.

16b ⟨*bi5: get user input* 16b⟩≡ (16a)

```
⟨bi5: form label specifiers 16c⟩
⟨bi5: form field specifiers 17a⟩
THREE THREE 'DROPFALSE "BIFURCATION DIAGRAM"
DoInputForm case ::
```

7.1.1 Field Labels

We'll label the fields so the user knows what's expected.

16c ⟨*bi5: form label specifiers* 16c⟩≡ (16b)

```
"WIDTH:"    ONE NINETEEN
"HEIGHT:"   ONE TWENTYEIGHT
"POINTS:"   ONE THIRTYSEVEN
```

7.1.2 Field Specifiers

The fields are where the data go. For default values we'll use the same values we used two programs ago.

17a ⟨*bi5: form field specifiers* 17a⟩≡ (16b) 17c▷
```
'DROPFALSE FORTYFIVE SEVENTEEN
⟨bi5: form field size and type 17b⟩
"IMAGE WIDTH" MINUSONE MINUSONE 201. 201.
```

17b ⟨*bi5: form field size and type* 17b⟩≡ (17)
```
THIRTYFIVE NINE ONE { ZERO } FOUR
```

The other two fields are specified similarly.

17c ⟨*bi5: form field specifiers* 17a⟩+≡ (16b) ◁17a
```
'DROPFALSE FORTYFIVE TWENTYSIX
⟨bi5: form field size and type 17b⟩
"IMAGE HEIGHT" MINUSONE MINUSONE 221. 221.

'DROPFALSE FORTYFIVE THIRTYFIVE
⟨bi5: form field size and type 17b⟩
"# OF POINTS TO PLOT" MINUSONE MINUSONE 140. 140.
```

7.2 Running the Program

The program is now 770 bytes in size, half again the size of the previous program and triple the size of the original User RPL program. Many people would consider this one more user friendly, though power users might prefer the previous stack-based one.

7.3 Linker Command File

As before, this program needs a linker command file.

17d ⟨*bi5.m* 17d⟩≡
```
LL BI5.LR
OU BI5
RE BI5.O
SE C:\HP48\LIB\ENTRIES.O
SU XR
```

8 Some Final(?) Comments

We've taken quite a journey through the chaotic world of bifurcation diagrams and the somewhat more ordered world of System RPL programming. But we've only scratched the surface of both worlds. Interested readers are urged to surf the Web for more details. Your local library also probably has books on chaos (alas, you probably won't find many books on HP 48 System RPL programming at your library).

A Literate Programming

This document not only describes the implementation of these programs, it *is* the implementation. The noweb system for "literate programming" generates both the document and the program code from a single source. This source consists of interleaved prose and labelled *Code Fragments*. The fragments are written in the order that best suits describing the program, namely the order you see in this document, not the order dictated by the programming language. The program noweave accepts the source and produces the document's typescript, which includes all of the code and all of the text. The program notangle extracts all of the code, in the proper order for compilation.

Fragments contain source code and references to other fragments. Fragment definitions are preceded by their labels in angle brackets. Several fragments may have the same name; notangle concatenates their definitions to produce a single fragment. noweave identifies this concatenation by using $+\equiv$ instead of $\equiv$ in continued definitions:

Fragment definitions are like macro definitions; notangle extracts a program by expanding one fragment. If its definition refers to other fragments, they themselves are expanded, and so on.

Fragment definitions include aids to help readers navigate among them. Each fragment name ends with the number of the page on which the fragment's definition begins and a letter giving its sequence within that page. If there is only one fragment on a page then there is no letter. This is also shown in the left margin. Each continued definition also shows the previous definition, and the next continued definition, if there is one. ◁ 7b is an example of a previous definition that appears on page 7, and 11 ▷ says the definition is continued on page 11. These annotations form a double linked list of definitions; the left arrow points to the previous definition in the list and the right arrow points to the next one. The previous link on the first definition is omitted, and the next link on the last definition is omitted. These lists are complete: If some of a fragment's definition appears on the same page with each other, the links refer to the page on which they appear.

Most fragments also show a list of pages on which the fragment is used. These unadorned use lists are omitted for root fragments, which define modules.

Of course, the simple fact that the documentation can be placed right next to the code doesn't necessary mean that the documentation and the code match, nor that either is any good. Still, it's a good start...

For more information about the noweb system of literate programming, please refer to `http://www.eecs.harvard.edu/~nr/noweb`.

B Index of Code Fragments

Underlined entries are to the definition of the Code Fragment. In many cases, the definition of a fragment can be continued from one piece to another.