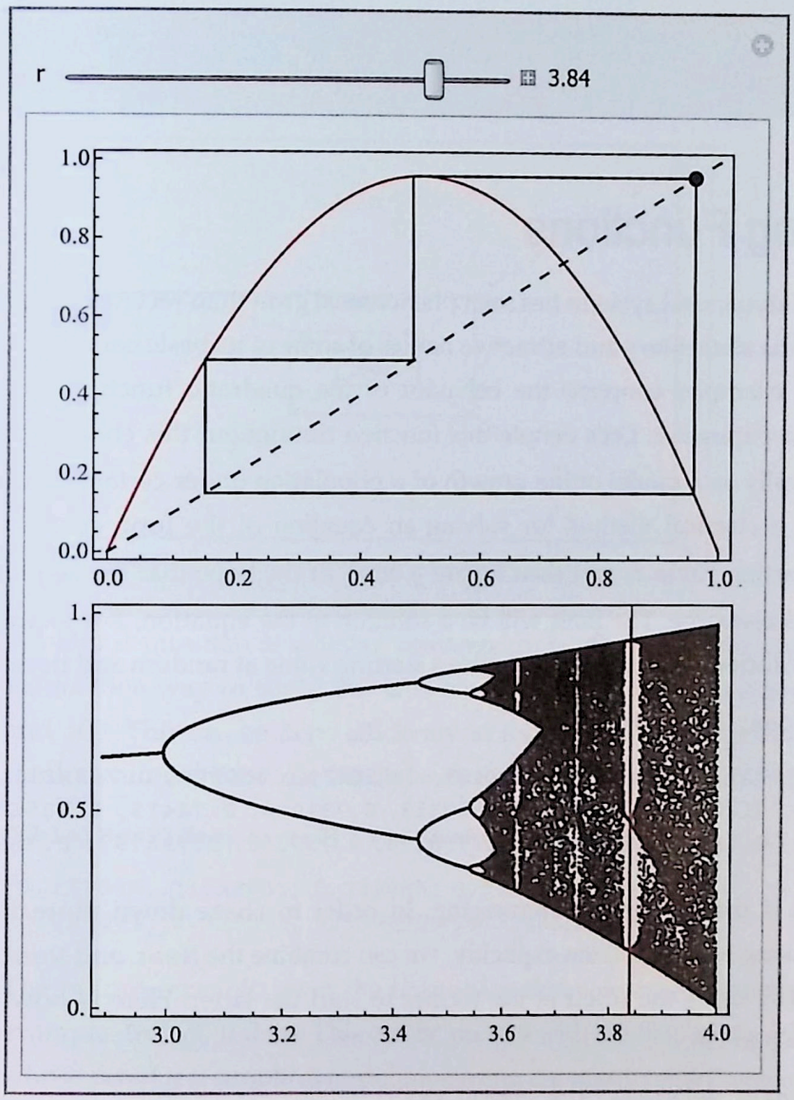
7 The Quadratic Map



There are several ways to examine the dynamics of the quadratic map defined by $f_r(x) = r\,x\,(1-x)$. The image above is from a demonstration that shows the bifurcation diagram for f_r, while allowing the user to move a slider (the red line) which causes the corresponding cobweb plot to appear as well. The example shown has r set to 3.84, for which the limiting behavior is a 3-cycle.

Mathematica allows us to very easily investigate the result of iterating functions, thanks to the built-in functions `Nest` and `NestList`. In this chapter we show how these and other functions can be used to investigate the complicated behavior of $r\,x\,(1-x)$, which is viewed as a function of x that changes as r changes. There are many different ways to visualize the phenomena surrounding this function, so it provides a good context in which to learn graphics programming.

7.1 Iterating Functions

The field of dynamical systems has seen phenomenal growth in recent years, in part because of the elementary and attractive nature of some of its basic concepts. One of the central examples concerns the behavior of the quadratic function $r\,x\,(1-x)$, where r is a parameter. Let's denote this function throughout this chapter by f_r; it arises naturally as a model of the growth of a population under certain conditions. Recall that a classical method for solving an equation of the form $g(x) = x$ is to choose a starting value x_0 and then iterate g on x_0 in the hope that the sequence of iterates will converge. The limit will be a solution to the equation. For example, if we seek a solution of $\cos x = x$, we choose a starting value at random and iterate.

```
NestList[Cos, 1.5, 20]
```

```
{1.5, 0.0707372, 0.997499, 0.542405, 0.85647, 0.655109, 0.792982,
 0.701724, 0.76373, 0.722261, 0.750313, 0.731476, 0.74419, 0.735637,
 0.741403, 0.737522, 0.740137, 0.738376, 0.739563, 0.738763, 0.739302}
```

It looks as if the iterates are converging. In order to chase down more iterates without having to look at them explicitly, we can combine the `Nest` and `NestList` commands by using the result of the former to start the latter. Here is how to see iterations 100 to 110.

```
NestList[Cos, Nest[Cos, 1.5, 100], 10]
```

```
{0.739085, 0.739085, 0.739085, 0.739085, 0.739085,
 0.739085, 0.739085, 0.739085, 0.739085, 0.739085, 0.739085}
```

The limiting value seems to be 0.739085, and it is the desired fixed point.

```
Cos[0.739085]
```

```
0.739085
```

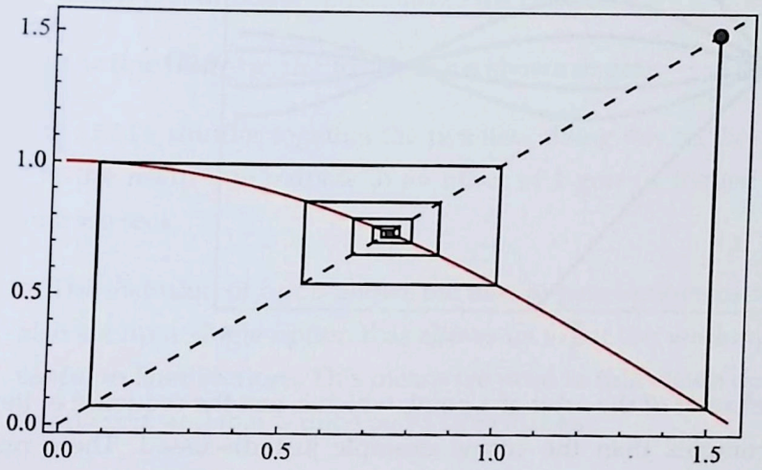
An important point to remember when using `Nest` is to avoid symbolic expressions. If the integer `1` (as opposed to the approximate real `1.`) is used as the starting value, then the result is a symbolic mess, and a crash is likely when the number of

iterations is large.

```
NestList[Cos, 1, 5]
```

```
{1, Cos[1], Cos[Cos[1]], Cos[Cos[Cos[1]]],
 Cos[Cos[Cos[Cos[1]]]], Cos[Cos[Cos[Cos[Cos[1]]]]]}
```

There is a simple graphical interpretation of this procedure. In the following figure, the successive points on the graph of the cosine function are the iterates with starting value 1.5.



A natural question is whether convergence is affected by the choice of a starting value. One way to study this is to try random starting values between, say, -100 and 100. This can be done efficiently as follows where we generate an array of 10 random numbers and use `Table`, with the iterator `c` varying over the random array.

```
Table[Nest[Cos, c, 100], {c, RandomReal[{-100, 100}, {10}]}]
```

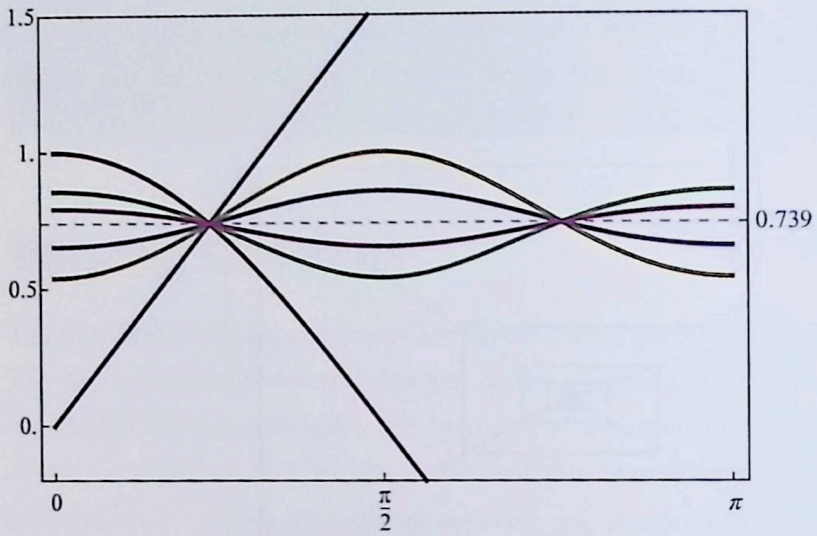
```
{0.739085, 0.739085, 0.739085, 0.739085, 0.739085,
 0.739085, 0.739085, 0.739085, 0.739085, 0.739085}
```

The output certainly gives the impression that convergence always occurs for this example. In fact, it does. This can be proved with the help of the following theorem, whose proof is a simple manipulation using the mean-value theorem of calculus.

THEOREM 1. Suppose f is a differentiable function from $\mathbb{R}$ to $\mathbb{R}$, p is a fixed point of f, and K is a constant less than 1 such that $|f'(x)| < K$ in an interval around p. Then the orbit of any starting value in the interval converges, under iteration of f, to p.

Another way to view the convergence is to plot the iterates of the cosine function using the command that follows. The use of `Evaluate` is important here for a couple reasons. First, one gets more speed. Second, it means that `Plot` sees a list and treats the list as a set of functions from $\mathbb{R}$ to $\mathbb{R}$. If left unevaluated, `Plot` treats this as a function from $\mathbb{R}$ to $\mathbb{R}^5$ and this has implications for the `PlotStyle` settings in that the first style then applies to all of them.

```
Plot[Evaluate[NestList[Cos, x, 5]] {x, 0, π},
 PlotRange → {-0.2, 1.5}, Frame → True,
 Axes → False, PlotStyle → Thick, GridLines → {{}, {{0.739, Dashed}}},
 FrameTicks → {Range[0, π, π/2], Range[-0.5, 2, 0.5], None, {0.739}}]
```



In general, the behavior of the orbit of a point, which is just the sequence of iterates, is much more complex than the cosine example just discussed. There may be convergence for some starting values but not for others; there may be more than one fixed point; an orbit may get locked into a periodic loop; there may be periodic points that are repelling (that is, no point outside the cycle has an orbit that approaches the cycle, no matter how close the starting point is to the cycle), and so on. The importance of the quadratic map is that it displays all these complexities for differing values of r. We will not go into a lot of detail, referring the reader to the lucid discussion of the dynamics of this family in the book by R. Devaney [Dev1] (see also [CE, Dev2]); rather, we will show how *Mathematica* can be used to generate images related to the orbits of the quadratic map, or arbitrary functions.

We will first show how to define a function, `CobwebPlot`, that accepts as input an expression (not a function: `Cos` is a function, `Cos[x]` is an expression), a plotting interval, a starting value, and a number of iterations and returns the graph of the function together with the cobweb of lines that illustrates what happens to the orbit. Actually, it is convenient to allow the number of iterations to be a single integer, in which case that many iterations are carried out, or a pair, $\{n_0, n\}$, in which case the first n_0 iterations are not shown, but then n iterations beyond that are shown.

We do this by using two cases, with the single-integer case (the `_Integer` added to `n` is essential so that this case doesn't respond when the fourth argument is a pair) simply calling the pair case with `{0, n}`.

Here is a guide to the code that follows.

1. The line `fcn = Compile[x, f]` sets up a compiled function. This speeds up computations quite a bit. As a further reminder, note that `Cos[x]` is an expression, while `Cos` and `Compile[x, Cos[x]]` are functions. Commands such as `Plot` or `Integrate` want expressions as a first argument; `Nest` or `NestList` take a function as the first argument. Because we allow for higher precision, an `If` and `Function` construction is used to make a wp-precision version of `f`.

2. The `data` definition line finds the orbit; the `start` definition phrase sets `start` to be the x-coordinate of the start of the cobweb.

3. The line from `{a, a}` to `{b, b}` is shown in gray.

4. `Riffle` shuffles together the two lists; doing this on the orbit and then partitioning the result into pairs with an offset of 1 gets us the list of points in the broken line we seek.

5. The inclusion of `opts` allows the user to pass options suitable for `Graphics`. We also set up a single option that allows us to set the working precision; this will be useful in later sections. This means we need to thin down `opts` to those suitable for `Graphics`, and that is done by `FilterRules`.

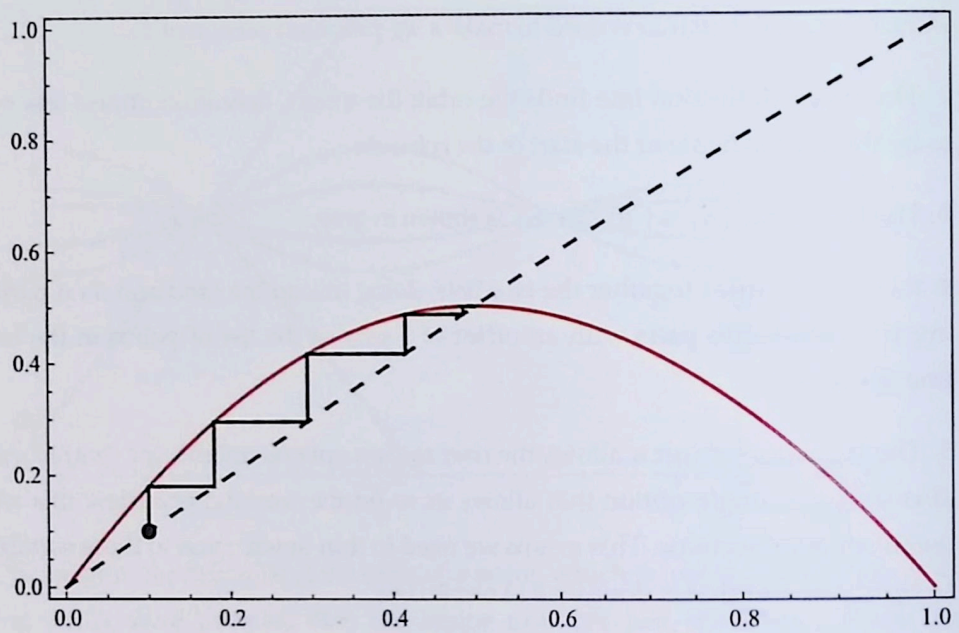
```
Options[CobwebPlot] = {WorkingPrecision → MachinePrecision};

CobwebPlot[f_, {x_, a_, b_}, x0_, n_Integer, opts___] :=
  CobwebPlot[f, {x, a, b}, x0, {0, n}, opts];

CobwebPlot[f_, {x_, a_, b_}, x0_, {n0_, n_}, opts___] :=
 Module[{fcn, data, start, wp},
  wp = WorkingPrecision /. {opts} /. Options[CobwebPlot];
  fcn = If[wp === MachinePrecision,
    Compile[x, f], Function[{x}, N[f, wp]]];
  start = Nest[fcn, x0, n0];
  data = NestList[fcn, start, n - 1];
  Show[Plot[f, {x, a, b},
    Evaluate[Sequence @@ FilterRules[{opts}, Options[Plot]]],
    PlotStyle → {Red, Thickness[0.004]}],
    Graphics[{{Thickness[0.004], Black,
      Line[Partition[Riffle[data, data], 2, 1]]},
     {PointSize[Large], Point[{start, start}]},
     {Dashing[{0.013, 0.023}],
      Thickness[0.004], Line[{{a, a}, {b, b}}]}}],
   Sequence @@ FilterRules[{opts}, Options[Graphics]],
   PlotRange → All, Frame → True, Axes → False,
   FrameTicks → {Automatic, Automatic, False, False}]]
```

We define the quadratic map for use throughout this chapter as follows, using a subscript for the parameter r. One could also use `f[r_][x_]`. Either of these is pre-

ferable to `f[r_, x_]` since that would require the unwieldy `f[r, #] &` in order to get f_r as a function.

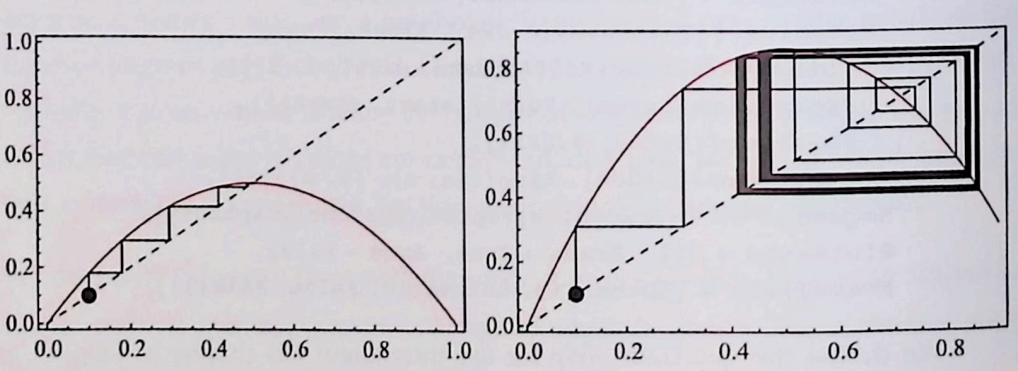
```
f_r_[x_] := r x (1 - x)
```

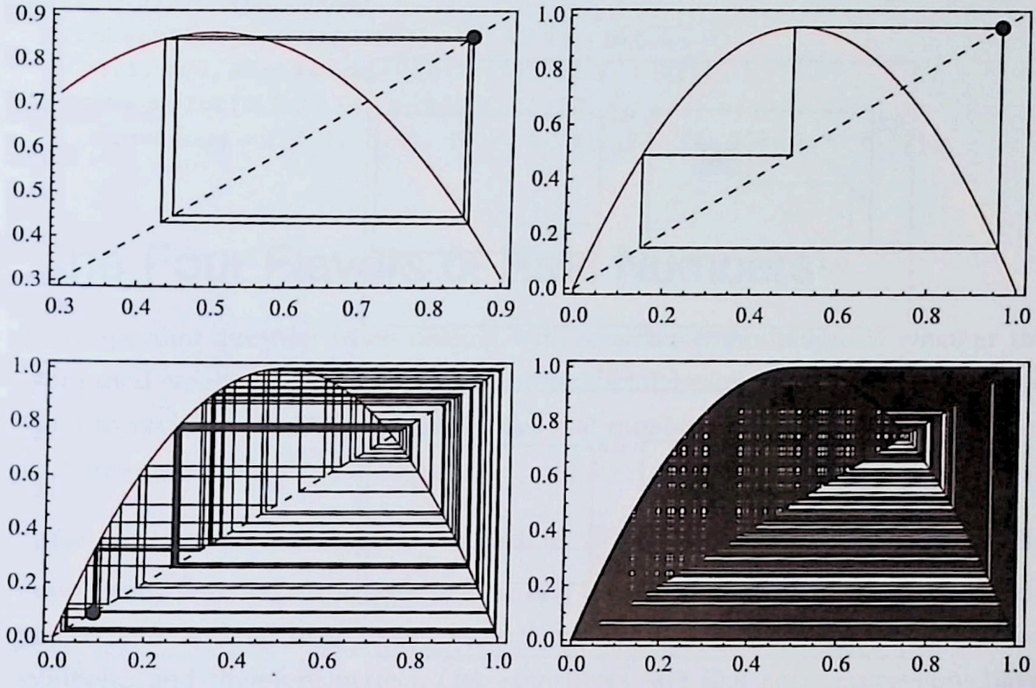
```
CobwebPlot[f_2[x], {x, 0, 1}, 0.1, 10]
```



And here are six cobweb plots. The first shows several iterates of f_2; there is convergence. The second, reading across the top, shows the first 100 iterates of $f_{3.45}$. The third shows iterates 300 to 350 of $f_{3.45}$, and the 4-cycle becomes clear. The fourth shows iterates 150 to 200 for $r = 3.839$ and we see a 3-cycle. The last two show lots of iterates of f_4, for which the result is chaotic.

```
Grid[Partition[{
    CobwebPlot[f_2[x], {x, 0, 1}, 0.1, 10],
    CobwebPlot[f_3.45[x], {x, 0, 0.9}, 0.1, 100],
    CobwebPlot[f_3.45[x], {x, 0.3, 0.9}, 0.1, {300, 50}],
    CobwebPlot[f_3.839[x], {x, 0, 1}, 0.1, {150, 50}],
    CobwebPlot[f_4[x], {x, 0, 1}, 0.1, {200, 75}],
    CobwebPlot[f_4[x], {x, 0, 1}, 0.1, 600]}, 2]]
```

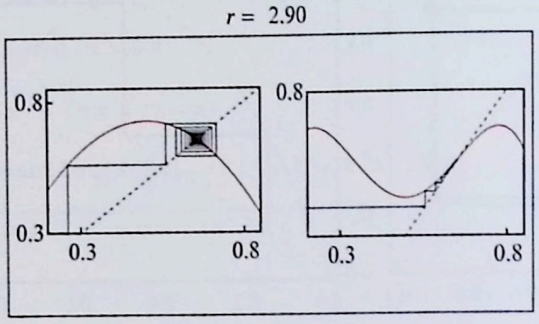


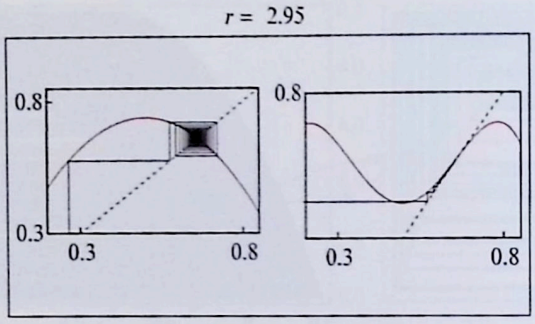


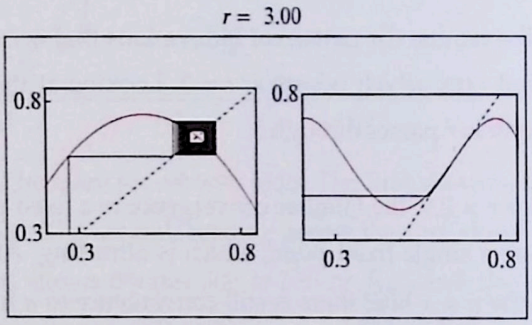
Later in this chapter we will examine the pattern of bifurcations that occur, but it is instructive to look at the first one, which occurs at $r = 3$. Looking at the graph of $f_r(f_r(x))$ clarifies what happens as r passes through 3.

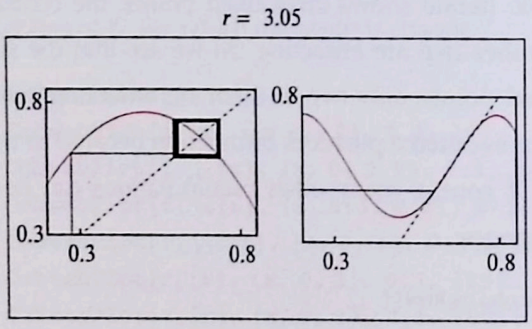
In the chart below we see, for $r = 2.9$, the familiar convergence to a fixed point. And the graph of $f_r(f_r(x))$ also has a single fixed point, which is attracting. At $r = 3$ the double iterate is tangent to the $y = x$ line; there is still convergence to a fixed point. As r rises past 3 the double iterate shows three fixed points: the central one has become repelling and the other two are attracting. So we see that the single fixed point has become three fixed points; only two of them are attracting, and we have the first bifurcation, sometimes called a pitchfork bifurcation because of the underlying three-point behavior. Of course, many other manipulations can be set up to watch what happens as r changes.

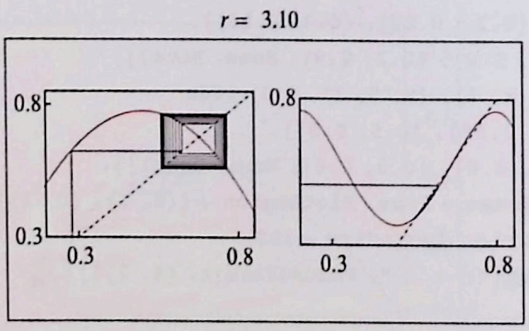
```
GraphicsColumn[Table[GraphicsRow[{
    CobwebPlot[fr[x], {x, 0, 1}, 0.1,
     100, PlotRange → {{0.2, 0.85}, {0.3, 0.85}},
     FrameTicks → {{0.3, 0.8}, {0.3, 0.8}, None, None}],
    CobwebPlot[Nest[fr, x, 2], {x, 0, 1}, 0.1, 100,
     PlotRange → {{0.2, 0.85}, {0.5, 0.8}},
     FrameTicks → {{0.3, 0.8}, {0.3, 0.8}, None, None}]},
   AspectRatio → 1 / 2, Frame → True, PlotRegion → {{0, 1}, {0, 1}},
   ImagePadding → Automatic, ImageSize → 300,
   PlotLabel → StringForm["r = ``", PaddedForm[r, {4, 2}]]],
  {r, 2.9, 3.1, 0.05}]]
```











For a live manipulation of this phenomenon use the following code.

```
Manipulate[GraphicsRow[
  {CobwebPlot[Nest[f_r, x, 2], {x, 0, 1},
    0.1, 100, PlotRange → {{0.2, 1}, {0.5, 0.8}}],
   CobwebPlot[f_r[x], {x, 0, 1}, 0.1, 100,
    PlotRange → {{0.2, 0.8}, {0.3, 0.75}}]}], {r, 2.9, 3.3, 0.01}]
```

7.2 The Four Flavors of Real Numbers

An important question when dealing with sensitive computations is whether the computed results are correct. To delve into this interesting and important area we need to review how *Mathematica* handles real numbers. There are four data types that are relevant:

1. Rational or symbolic expressions such as $\frac{20}{49}$ or `Sin[10^50]`: these are symbolic expressions with no error whatsoever. If a computation can be done with such objects and without ever using decimal approximations then the result will be symbolic, and therefore perfect. One sometimes says that such expressions have infinite precision. In the following example, the result is perfect.

```
20/49 + 31/17
```

$$\frac{1859}{833}$$

2. Machine reals: the approximate real numbers with about 16 digits of precision that arise when a decimal point or the `N[]` operator is used. For such numbers, no attempt is made to keep track of how precision is lost (or gained) through a computation. In the following example the information contained in the trailing 5s is totally lost when the addition is performed.

```
1.3 + 1.12345678955555 10^-7 // InputForm
```

```
1.300000112345679
```

Asking for the precision of the number we learn that it is the abstract object `MachinePrecision` (which can vary between platforms).

```
Precision[%]
```

```
MachinePrecision
```

Understanding this data type makes one understand, if not appreciate, the error that occurs in the following computation. There is not enough room in machine precision to store both the leading 1 and the 1 lying seventeen places to the right of the decimal point. So the latter is lost; this is typical subtractive cancellation.

```
1. + 1/10^17 - 1
```

```
0.
```

Without the decimal point the result of the arithmetic is a rational, and taking the numerical value of that gives a numerically correct answer.

```
N[1 + 1/10^17 - 1]
```

```
1. × 10^-17
```

3. Software reals. These arise when one uses the N operator with a second argument to force a certain precision. The number then comes with a precision attached.

```
(a = N[π, 30]) // InputForm
```

```
3.14159265358979323846264338327950288419716767885484628792 81`30.
```

The 30 at the tail end indicates the precision; this is generally a pessimistic view as it typically happens that more than 30 digits are correct in such a situation. If we raise the 30-digit number to the 100 000th power, the precision goes down to 25.

```
Precision[a^100 000]
```

```
25.
```

If we iterate the quadratic map 30 times on a 30-digit number, the result has only 12 digits of precision, which we can see as an appendage to the software real when using InputForm.

```
Nest[f4, N[1 / 10, 30], 30] // InputForm
```

```
0.32034249381837718430096359663`12.443814812333398
```

In fact, all digits except the final four, 9663, are correct, behavior that is indicative of the overly pessimistic precision estimates.

While precision is often lost, it can also be gained. In the following example the argument to sine is near a maximum, and so the precision of the result is quite a bit greater than the 30-digit precision of the input.

```
a = N[π/2 + 1/10^7, 30];
Precision[Sin[a]]
```

```
36.8039
```

Note that the precision can dip below 16, down all the way to 0.

4. Another way of dealing with real numbers is to view them as intervals that are guaranteed to contain the abstract real number one has in mind. Such interval objects are built into *Mathematica* and can be used both to obtain guaranteed correct results of numerical computations and to design algorithms (see §13.4).

```
Sin[Interval[{-π, π}]]
```

```
Interval[{-1, 1}]
```

```
g[x_] := Sin[x] + Cos[x];
g[Interval[{-π, π}]]
```

```
Interval[{-2, 2}]
```

This is overly pessimistic, the reason being that interval operations do not know that the same number is being fed to sine and cosine; it is just assumed that the interval $[-1, 1]$ is being added to itself. Of course, we know that a tight answer to this question is $\left[-\sqrt{2}, \sqrt{2}\right]$. But software arithmetic is pessimistic also. The point of interval work is that the final interval is guaranteed to contain the answer. A downside to the use of intervals is that they do not work beyond the basic functions, so if expressions involve the Riemann ζ-function or the gamma function, intervals cannot be used.

When we use intervals we can start with `Interval[s]`, which gets transformed to a small interval around s. We learn here that the answer is consistent with what was given when we used software arithmetic.

```
Nest[f4, Interval[N[1 / 10, 30]], 30]
```

```
Interval[{0.320342493798, 0.320342493839}]
```

Another important idea related to software reals is the concept of adaptive precision. If one has a symbolic expression and asks for d digits of precision, that might require evaluating the expression to much more than d digits of precision.

Mathematica's `N` command handles this in a way that is transparent to the user, and delivers the required precision by automatically figuring out how many extra digits are required. In the next example the first answer, computed with machine precision, is quite wrong, since machine precision is nowhere near accurate enough to compute the sine of 10^{50}.

```
N[Sin[10^50]]
```

```
-0.4805
```

But adaptive precision gets the correct answer, quite different than the preceding.

```
N[Sin[10^50], 20]
```

```
-0.78967249342931008271
```

Note that we can use interval arithmetic to obtain certified digits. The use of 75-digit precision is enough to get the result we seek; using, say, 20 instead of 75 gets the correct, but useless, result `Interval[{-1, 1}]`.

```
Sin[Interval[N[10, 75]]^50]
```

```
Interval[{-0.78967249342931008271030, -0.78967249342931008271028}]
```

For more difficult examples the default setting of `$MaxExtraPrecision`, which is 50, might not be enough. For the following we must reset this system variable.

```
$MaxExtraPrecision = 500;
N[Sin[10^100], 20]
```

```
-0.37237612366127668826
```

We can now apply our understanding to the quadratic map. Using only machine precision, we get an answer very quickly.

```
Nest[f4, 0.1, 100]
```

```
0.372447
```

The traditional method of checking accuracy would be to run it with more digits. Using 20 or 50 is not enough as the result has zero precision. Using 70 gives us something useful, and it shows that the sensitivity issue here is a serious one.

```
Nest[f4, N[1 / 10, 70], 100]
```

```
0.9301089742
```

Note that we cannot use the adaptive precision idea here because that requires computing the rational form of the result first. The numerator of this rational is gigantic — over a million digits — even for 21 iterations.

```
Log[10., Numerator[Nest[f4, 1 / 10, 21]]]
```

```
1.46585 × 10^6
```

We can use intervals, which confirms correctness of the 70-digit computation.

```
Nest[f4, Interval[N[1 / 10, 100]], 100]
```

```
Interval[{0.93010897418655471559156762324361396511 60,
  0.93010897418655471559156762324361396512 99}]
```

We can monitor the loss of accuracy in the case of f_4 as follows where we use `Precision` to track the loss. But the result of this experiment — a loss of 59 digits in 100 iterations — is not the true story since the precision tracking will be overly pessimistic. Another approach is to monitor the growth in the spread of a small interval about x_0; that will be pursued in the next section.

```
70 - Round[Precision /@ NestList[f4, N[1/10, 70], 100]]
```

```
{0, 0, 0, 1, 1, 2, 3, 4, 4, 4, 6, 6, 6, 10, 10, 10, 10, 10, 10, 11, 11,
 12, 12, 14, 14, 14, 16, 16, 16, 16, 18, 18, 19, 19, 21, 21, 21,
 21, 24, 24, 24, 24, 24, 25, 26, 26, 27, 29, 29, 29, 29, 30, 32,
 32, 32, 32, 33, 33, 34, 35, 36, 36, 37, 37, 38, 38, 39, 39, 40,
 41, 41, 42, 44, 44, 44, 44, 46, 46, 46, 48, 48, 48, 49, 49, 50,
 50, 51, 52, 52, 53, 54, 54, 54, 55, 56, 57, 57, 57, 59, 59, 59}
```

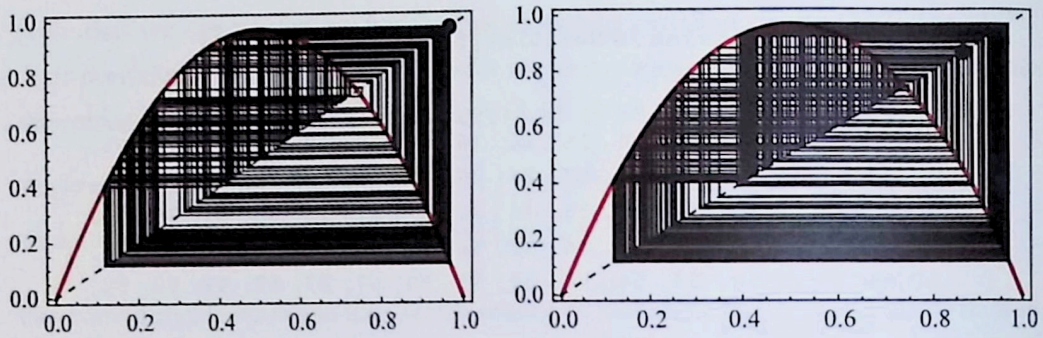
The precision loss is less for other values of r and, indeed, when there is convergence there is precision gain. Moreover, as we shall see in the next section, the case of $r = 4$ is the worst case for precision loss. However, let us perform one quick computation to see the *exact* loss of accuracy. We do that by starting with the interval $\left[\frac{1}{10} - \frac{1}{2}10^{-150}, \frac{1}{10} + \frac{1}{2}10^{-150}\right]$, which captures the uncertainty in a 150-digit approximation to $\frac{1}{10}$, and see how that uncertainty expands as we iterate f_4, using base-10 logarithms of the expansion. From this computation we learn that 30 digits were lost in 100 iterations.

```
δ = 10^-150; x0 = 1/10;
Table[Δ = Abs[Nest[f4, N[x0 - δ/2, 200], n] - Nest[f4, N[x0 + δ/2, 200], n]];
  Round[Log[10, Δ/δ]], {n, 0, 100}]
```

```
{0, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 2, 3, 3, 4, 4, 5, 6, 6, 6, 7, 6, 7, 7, 7,
 8, 8, 9, 9, 9, 9, 10, 9, 10, 11, 11, 10, 11, 11, 12, 13, 13, 13, 13, 14,
 13, 14, 14, 15, 15, 15, 15, 16, 16, 17, 17, 17, 18, 18, 18, 18, 19, 19, 19,
 20, 20, 20, 21, 21, 21, 21, 22, 22, 22, 22, 23, 23, 23, 23, 24, 24, 25,
 25, 25, 26, 26, 26, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 30, 30, 30}
```

Now that we know how to get perfectly accurate results, is it worth it? It turns out that the general behavior one sees when using machine precision is qualitatively the same as when using more accurate methods. Here we use a high working precision to generate an accurate cobweb plot; we see that the accurate plot differs little from the machine precision plot.

```
r = 31 / 8;
Row[{CobwebPlot[fr[x], {x, 0, 1}, 1 / 10, {100, 300},
    PlotStyle → {Red, Thick}, ImageSize → 180], Spacer[20],
      CobwebPlot[fr[x], {x, 0, 1}, 1 / 10, {100, 300}, WorkingPrecision →
     1000,   PlotStyle → {Red, Thick}, ImageSize → 180]}]
```



But this similarity between machine precision and absolute truth must not be carried too far. There are only finitely many machine reals, so any iterative process must eventually cycle. On my computer (Macintosh MacBook Pro with Intel processor) I found that any iteration of $4x(1-x)$ fell into one of six cycles, having periods 1, 10210156, 14632801, 5638349, 2625633, and 2441806. The 1 is easy to understand, since if an iteration ever gets close to 0.5, then the next iteration is 1.0, and all the ones after that are 0.0. The other five are somewhat machine-dependent, but the basic reason they arise is related to the birthday paradox on repeated elements in a list. A lucid explanation of this cycling behavior can be found in [Sau].

7.3 Attracting and Repelling Cycles

Leaving numerical intricacies aside, note the interesting behavior of the quadratic map that occurs for $r \geq 1$. (The behavior for $r < 1$ is easily handled by Theorem 1.) For $1 < r < 4$, it is always the case that negative starting values have orbits converging to negative infinity, as do starting values greater than 1. And a starting value of either 0 or 1 converges immediately to 0. Moreover, if $r > 4$, all starting values, except for a Cantor set, have orbits that converge to negative infinity (see [Dev1, §1.5]). Thus throughout this chapter we shall consider only values of r between 1 and 4 and starting values in the open unit interval. Note that the nonzero fixed point of f_r is simply $1 - \frac{1}{r}$.

```
Clear[r]; x /. Solve[f_r[x] == x, x]
```

$$\left\{0, \frac{-1+r}{r}\right\}$$

Some treatments of this subject (e.g., [CE]) use the function $1 - \mu x^2$ instead of f_r, with the parameter μ varying between 0 and 2; this version maps the interval $[-1, 1]$ into itself, whereas f_r maps the interval $\left[1 - \frac{r}{4}, \frac{r}{4}\right]$ into itself when r is between 2 and 4. Still another important quadratic family is given by $x^2 + c$ with c between -2 and $\frac{1}{4}$.

This last family is the correct one to look at in order to relate the quadratic map to the quadratic map $z^2 + c$ in the complex plane, which underlies the definition of the Mandelbrot set (the complex map is discussed further in Chapter 10). In any event, these three families are entirely equivalent as far as orbit behavior goes [Dev1, §1.7]; we will focus on f_r.

The values of r in the examples at the end of §7.1 yield functions with radically different orbit structure. For $r = 2$ there are two fixed points, 0 and $\frac{1}{2}$, and any starting value in (0, 1) converges to $\frac{1}{2}$; that is, points arbitrarily close to 0 move away from 0, whence 0 is a repelling fixed point. For $r = 3.45$, the starting value 0.1 leads to a 4-cycle. Do all starting values lead to this 4-cycle? A quick way to check is with the following command, which shows that for ten random starting values in [0, 1], the 500th iteration is very close to one of the four periodic points 0.445968, 0.852428, 0.433992, 0.847468. Of course, to be certain one should double-check using high precision, as indicated earlier.

```
Table[Sort[NestList[f3.45, Nest[f3.45, Random[], 500], 3]], {10}]
```

```
{{0.435794, 0.444013, 0.848278, 0.851686},
 {0.433021, 0.447048, 0.847017, 0.852827},
 {0.433015, 0.447056, 0.84702, 0.852829},
 {0.434634, 0.445263, 0.847762, 0.852163},
 {0.433007, 0.447064, 0.847016, 0.852832},
 {0.433088, 0.446973, 0.847054, 0.852799},
 {0.43853, 0.441162, 0.849464, 0.850557},
 {0.433118, 0.446928, 0.847067, 0.852783},
 {0.432991, 0.447068, 0.847009, 0.852834},
 {0.433007, 0.447064, 0.847016, 0.852832}}
```

If you want to see the exact values of the 4-cycle, just use `FindRoot` with the seeds from the preceding computation as follows.

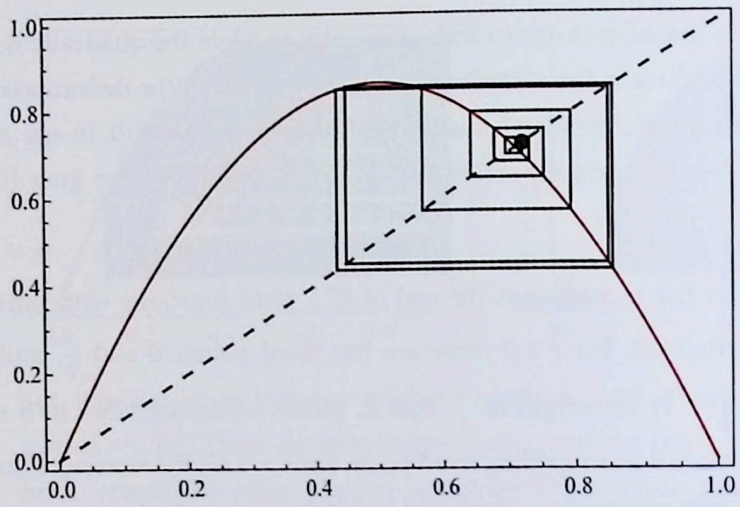
```
(x /. FindRoot[Nest[f3.45, x, 4] == x, {x, #}, AccuracyGoal → 12] &) /@ %[[1]]
```

```
{0.433992, 0.445968, 0.847468, 0.852428}
```

These computations might lead one to believe that all starting values in the open unit interval have orbits that are attracted to the 4-cycle. But this is not true! There are infinitely many points that fail to be attracted to the 4-cycle. For example, there is the nonzero fixed point of $f_{3.45}$, which is $1 - \frac{1}{r} = \frac{49}{69} \approx 0.70144927$.

This is a repelling fixed point because the derivative at the point is greater than 1. Therefore the orbit of any finite-digit approximation to $\frac{49}{69}$ will eventually drift away, and it is not hard to see that it will be sucked into the 4-cycle.

```
CobwebPlot[f3.45[x], {x, 0, 1}, 49/69 + 0.01, 100]
```



Of course, if perfect rationals are used, then the repelling fixed point is visible.

```
f345/100[49/69]
```

$$\frac{49}{69}$$

There are still more atypical points. The rational $\frac{20}{69}$ has the property that $f_{3.45}\left(\frac{20}{69}\right) = \frac{49}{69}$; the rational $\frac{20}{69}$ can be discovered using `Solve`.

```
Solve[f345/100[x] == 49/69]
```

$$\left\{\left\{x \to \frac{20}{69}\right\}, \left\{x \to \frac{49}{69}\right\}\right\}$$

We can keep working backward in this way — there will be two inverse images of $\frac{20}{69}$ — to come up with an infinite sequence of points whose orbits end up at $\frac{49}{69}$. This is related to the notion of the Julia set of a complex function, which will be discussed further in Chapter 11.

EXERCISE 1. Find some more of the points of the inverse orbit of $\frac{49}{69}$ and generate a graph that illustrates them.

EXERCISE 2. Prove that all starting values between 0 and 1, with the exception of the points in the inverse orbit of $\frac{49}{69}$, have orbits that converge to the 4-cycle. Hint: Let f denote $f_{3.45}$ and plot the two functions $h(x) = f(f(f(f(x))))$ and x; observe that the fourfold composition of f has six fixed points, four of which correspond to the four period-4 points of f. To examine the graph closely, it may be better to plot $h(x) - x$ on a small domain surrounding the fixed points. Now convergence for any x can be proved by looking at the graph of $h(x)$ in pieces and using Theorem 1.

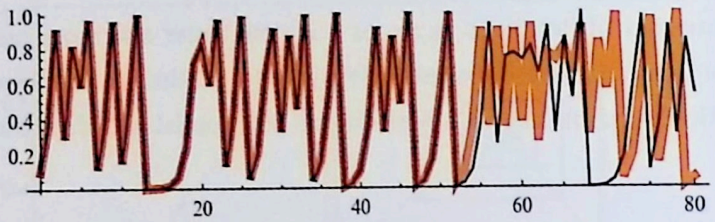
Note that as the number of iterations increases it is both faster and more numerically stable to plot iterates using `Plot[Nest[f, x, 4], ...]` than to first evaluate the iteration function symbolically as a high-degree polynomial and then plot the polynomial.

The case of $r = 3.839$ seems to be similar to $r = 3.45$ in that there is an attracting 3-cycle. Here too there is a repelling fixed point, this time at 0.7395155... . But in fact this case is dramatically different from the preceding case of a 4-cycle. A consequence of a remarkable theorem due to A. N. Sarkovskii states that if f is a continuous function $f : \mathbb{R} \to \mathbb{R}$ that has a periodic point whose period is 3, then for each positive integer n, f has a periodic point of period n. The proof of this theorem is not too difficult and can be found, together with the general Sarkovskii theorem for other periods, in [Dev1]; see also [ASY]. So, $f_{3.839}$ has periodic points of all orders in the unit interval. Moreover, these points are all repelling [Dev1, Cor. 11.10 and §1.13].

EXERCISE 3. Plot `Nest[f, x, 3]` and `x` (where `f` is $f_{3.839}$) to determine the approximate positions of the three repelling period-3 points. Then use `Roots[Nest[f, x, 3] == x, x]` to locate them with more precision. Because we are dealing with polynomials here, `Roots` can be used instead of `FindRoot`, which would be the tool to use if the underlying function involved has, say, a sine term.

The case $r = 4$ illustrated at the end of §7.1 has an even richer set of periodic points than the preceding cases: the set of periodic points is dense in the unit interval and there are period-n points for every n. But all these periodic points are repelling. Moreover, given any two intervals U and V, there is some positive integer k such that $f^k(U) \cap V \neq \emptyset$; in short, f is *topologically transitive*. This is illustrated by the all-over-the-place behavior of the orbit diagrams. Finally, the function depends sensitively on the starting value, as we have already seen, and can see again by looking at orbits of nearby points. First let's try looking at 80 iterations, perturbing the starting value by 10^{-17}; we use high precision so that the data shown is fully accurate.

```
n = 80;
orbit = Transpose[{Range[0, n], NestList[f4, N[1/10, n], n]}];
orbit1 = Transpose[{Range[0, n], NestList[f4, N[1/10 + 1/10^17, n], n]}];
ListLinePlot[{orbit, orbit1},
 PlotStyle → {{Thickness[0.013], Orange}, {Thickness[0.004], Black}},
 AspectRatio → 0.25]
```



A function, such as f_4, is said to be *chaotic* if it has three properties: a dense set of periodic points, sensitivity to small variations in the starting values, and topological transitivity. Part of the surge of interest in chaos and chaotic functions is due to the fact that a function as seemingly simple as a quadratic with small integer coefficients can exhibit the complexity of chaotic behavior (see [Gle]).

The case $r = 4$ is somewhat special because there is a nifty closed form for the iterated function. It is not hard to show that the nth iterate of f_4 on x_0 is given by the following expression [ASY, Exer. 1.15].

```
TrigIteration[n_, x0_] := 1/2 (1 - Cos[2^n ArcCos[1 - 2 x0]])
```

```
N[{TrigIteration[5, 1/10], Nest[f4, 1/10, 5]}]
```

```
{0.585421, 0.585421}
```

Indeed, `RSolve` is capable of deducing this formula.

```
(x /. RSolve[{x[n + 1] == 4 x[n] (1 - x[n]), x[0] == x0}, x, n][[1]])[n] //
 Quiet
```

$$\frac{1}{2}\left(1 - \text{Cos}\left[2^n \text{ArcCos}[1 - 2\,x0]\right]\right)$$

The proof is easy since the trig iteration is the identity when $n = 0$ and satisfies the correct recursive formula.

```
TrigExpand[TrigIteration[n + 1, x] ==
  4 TrigIteration[n, x] (1 - TrigIteration[n, x])]
```

```
True
```

This formula allows us to interpret f_4 as a bit-shifting map (see [Whi] for more on these ideas). The following code shows that if one starts with a number v between 0 and $\frac{1}{2}$ then one can pass to an x-value, feed that to f_4, and then invert the passage to a transformed v-value which turns out to be simply $2\,v$.

```
fval = f4[Sin[π v/2]^2];
```

```
Simplify[2/π ArcSin[Sqrt[fval]], 0 ≤ v ≤ 1/2]
```

```
2 v
```

And it is similar when $\frac{1}{2} \le v \le 1$.

```
PowerExpand[Simplify[2/π ArcSin[Sqrt[fval]], 1/2 < v ≤ 1],
 Assumptions → 1/2 < v ≤ 1]
```

```
2 - 2 v
```

The implication of this is that if one starts with $v = 0.b_1\,b_2\,b_3\ldots$ in base 2 where $b_1 = 0$, forms x to be $\sin^2\left(\frac{\pi}{2}v\right)$, applies f_4, and then inverts via $\frac{2}{\pi}\arcsin\sqrt{x}$, the resulting transform of v has binary representation $0.b_2\,b_3\ldots$. If $b_1 = 1$ the relationship is a little different: the leading bit (b_1) is dropped and the rest of the bits are flipped. So if one defines the tent map

$$T(v) = \begin{cases} 2v & \text{if } 0 \le v \le \frac{1}{2} \\ 2-2v & \text{if } \frac{1}{2} < v \le 1 \end{cases},$$

then f_4 is conjugate to T (i.e., $f_4 = F \circ T \circ F^{-1}$, where F denotes the $\sin^2\left(\frac{\pi}{2}v\right)$ function). Therefore $f_4^{(n)} = F \circ T^{(n)} \circ F^{-1}$.

In particular this allows us to easily construct periodic points for f_4. For a quick example consider the binary number 0.0100100100...; we may as well program the string operation, working on lists of bits.

```
stringOp[bits_] := If[bits[[1]] == 0, Rest[bits], 1 - Rest[bits]]
Nest[stringOp, Flatten[Table[{0, 1, 0}, {10}]], 3]
```

```
{0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0}
```

Let v be the real corresponding to the 3-periodic string, which is simply $\frac{2}{7}$.

```
v = FromDigits[{{{0, 1, 0}}, 0}, 2]
```

$$\frac{2}{7}$$

Now we code the transformation and get the x-value.

```
ToX[v_] := Sin[π v/2]^2;
```

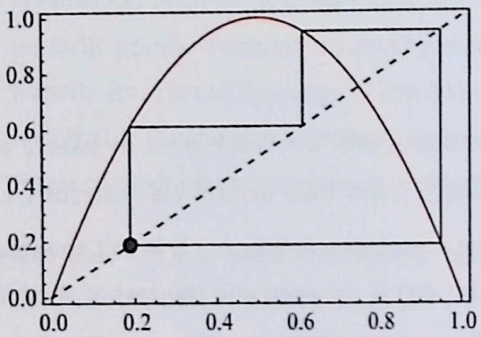
```
FromX[x_] := 2/π ArcSin[Sqrt[x]];
```

```
x0 = ToX[v]
```

$$\sin\left[\frac{\pi}{7}\right]^2$$

The reader can investigate the iterates of x_0 and the bits when one translates back to v. Here we show only the cobweb plot, which is quite different than the usual chaos one expects. We see here that x0 is a period-3 point, since the binary sequence we started with had period 3.

```
CobwebPlot[f4[x], {x, 0, 1}, x0, 10]
```



Note also that if v has a finite binary expansion then this process converges to 0 as the bits disappear one by one.

```
v = 31 / 64;
x0 = ToX[v]
```

$$\text{Sin}\left[\frac{31\,\pi}{128}\right]^2$$

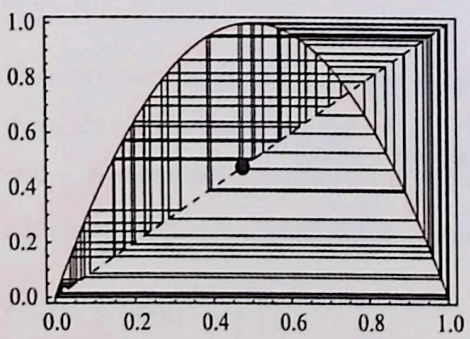
It is easier to now proceed numerically, using high precision.

```
N[NestList[f4, N[x0, 20], 7], 6]
```

$$\{0.475466,\ 0.997592,\ 0.00960736,\ 0.0380602,\ 0.146447,\ 0.500000,\ 1.00000,\ 0.\times 10^{-17}\}$$

But the attraction to 0 here (and also the periodic behavior shown earlier) is very special; if one starts just a teeny bit away from the special point, it is back to chaos. In other words, 0 is a repelling fixed point of f_4.

```
CobwebPlot[f4[x], {x, 0, 1}, x0 + 10^-10, 80, WorkingPrecision → 200]
```



7.4 Measuring Instability: The Lyapunov Exponent

One way to quantify the loss of precision of an iterated function is through the concept of the Lyapunov exponent, which we denote by λ. This number summarizes the speed with which a function separates nearby values. For the functions f_r the behavior depends on r of course, but also on the number of iterations n. Fixing r, n, and x_0, the value of λ is obtained by taking the limit as $\delta \to 0$ of the spread one gets by starting with an interval of size δ around x_0 and following it with the n-fold iterate of f_r. This spread is considered as a ratio Δ / δ and natural logarithms are taken, with the final result divided by n. This final division comes from thinking of n as time; the parameter λ can be viewed as a measure of the separation as a function of n or, after dividing by log 10, the number of decimal digits lost per iteration. A positive value of λ indicates exponential separation (hence exponential buildup of roundoff error when working with approximate reals), while a negative value indicates stability.

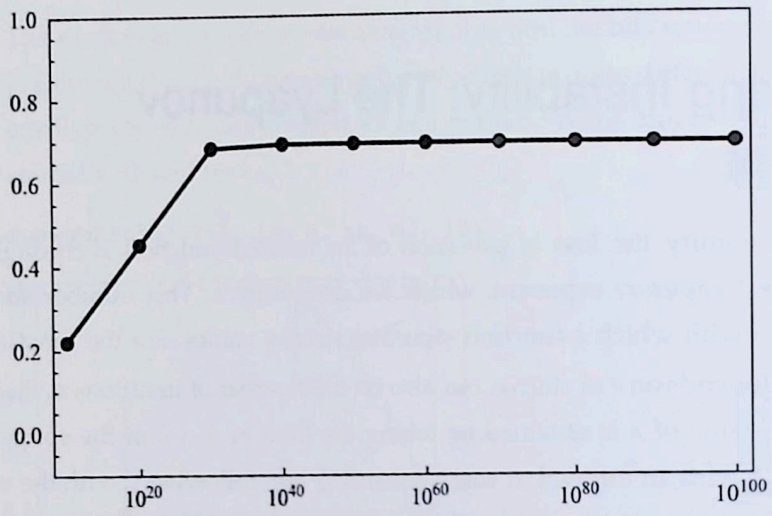
Here is a routine to compute this parameter, where we use information gained in the preceding section to define a precision high enough so that the computations are not themselves subject to numerical error. It is critical that the four inputs be symbolic expressions, since the routine will take high-precision values of them; using, say, 0.1 for x0 will cause this to fail.

```
Lyapunovλ[r_, x0_, n_, δ_] :=
 With[{prec = Max[2 n, -Log[10, δ] 2 + 20]},
   Δ = Abs[Nest[f_r, N[x0 - δ/2, prec], n] - Nest[f_r, N[x0 + δ/2, prec], n]];
   N[1/n Log[Δ/δ], 4]] /; Union[Precision /@ {r, x0, n, δ}] == {∞}
```

```
f4[0.1]
```

```
0.36
```

```
ListLinePlot[
 data = Table[{d, Lyapunovλ[4, 1/10, 100, 10^-d]}, {d, 10, 100, 10}],
 Frame → True, Axes → False, PlotStyle → {Thick, Black},
 PlotRange → {-0.1, 1}, Epilog → {PointSize[0.02], Point[data]},
 FrameTicks → {{#, HoldForm[10^#]} & /@ Range[20, 200, 20],
   Automatic, None, None}]
```



We see that there appears to be convergence and that $\delta = 10^{-100}$ is adequate to learn the limit. Moreover the convergence is sufficiently rapid that $n = 100$ gives a good approximation to the limit. One can also get these values symbolically by interpreting λ as a derivative: it is $\frac{d}{dx} f_r^{(n)}(x)$ where the superscript indicates the nth iteration. We give an example, but the method is limited by the complexity of the derivative as n increases, so we omit further discussion of this approach.

```
λByDer[r_, x0_, n_] := 1/n Log[Abs[∂x Nest[fr, x, n]]] /. x → x0
```

```
{λByDer[4, 0.1, 10], Lyapunovλ[4, 1/10, 10, 1/10^10]}
```

```
{0.709963, 0.7100}
```

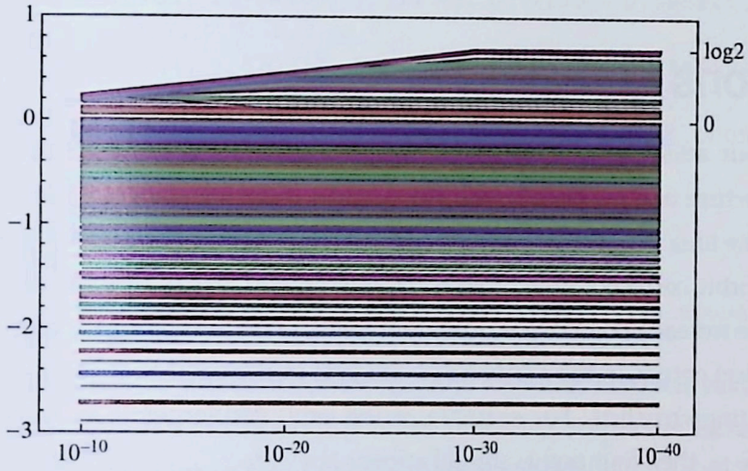
It appears that λ does not vary as x_0 changes. Moreover, it appears that λ for f_4 is exactly $\log 2$. In fact, one can prove this last assertion by making use of the bit-shift interpretation of the iterates given in §7.3.

The following demonstration shows the behavior of λ as d changes. There is no result when the convergence is too fast, as with $r = 2$ in which case $\lambda = -\infty$. Note that the iterator `r` varies over reals, but `r1` is defined to be a rational approximation.

```
Manipulate[r1 = Rationalize[r]; ListLinePlot[
  Table[{d, Lyapunovλ[r1, 1 / 10, 100, 10^-d]}, {d, 10, 100, 10}],
  Frame → True, Axes → False, PlotStyle → {Thick, Black},
  FrameTicks → {{#, HoldForm[10^#]} & /@ Range[20, 200, 20],
    Automatic, None, None}, PlotRange → {{5, 105}, {-3, 1}},
  PlotLabel → StringForm["r = ``", N[r, 3]]], {{r, 4}, 2, 4, 1. / 100}]
```

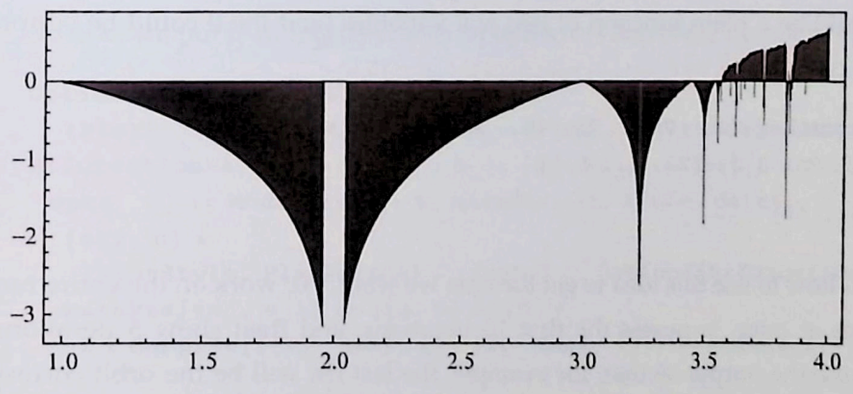
The following graph shows the evolution of λ as r goes from 2 to 4 in steps of $\frac{1}{100}$; this shows that $d = 10^{-40}$ is good enough for convergence when n is 100. The lowest line corresponds to $r = 2$ and the uppermost to $r = 4$.

```
ListLinePlot[
 Table[Table[{d, Lyapunovλ[r, 1/10, 100, 10^-d]}, {d, 10, 40, 10}],
  {r, 2, 4, 1/100}], Frame → True, Axes → False,
 PlotStyle → {Thick, Black}, FrameTicks →
  {({#1, HoldForm[10^-#1]} &) /@ Range[10, 40, 10], Automatic,
   None, {0, {Log[2], "log 2"}}}, PlotRange → {{8, 42}, {-3, 1}}]
```



So now we can generate a plot of λ as r changes but x_0, n, and δ are held at $\frac{1}{10}$, 300, and 10^{-100}, respectively. We use a `ProgressIndicator` so that it is easy to monitor the time the computation takes, which is a minute or so since 20 000 values of λ are computed.

```
Monitor[ListLinePlot[
  Table[{r, Lyapunovλ[r, 1/10, 300, 10^-100]}, {r, 1, 4, 3/20 000 100}],
  Frame → True, PlotStyle → {Thickness[0.0001], Black},
  PlotRange → All, Filling → {1 → {0, Red}}],
 ProgressIndicator[r, {1, 4}]]
```



Quoting David Campbell [Cam], "That such a filigree of interwoven regions of periodic and chaotic motion can be produced by a simple quadratically nonlinear map is indeed remarkable." The main point to take away from this diagram is that the behavior is complicated with intermixed regions where λ is positive and negative. Positive values imply sensitivity, negative ones imply stability or convergence. And the fact that the largest value occurs at 4 indicates, at least experimentally, that $r = 4$ is the worst parameter as far as sensitivity goes.

7.5 Bifurcations

We now turn our attention to producing the now-famous diagram that shows the entire orbit structure of the quadratic map as r varies. We will focus on the interval from 2.8 to 4. The idea is to use 100 or more equispaced values of r on the horizontal axis, form the orbit corresponding to each parameter, and place a point above the parameter value for each point in the orbit. Moreover, one will want to suppress the first 100 (or more) entries in the orbit so that the part of the orbit that is shown will reflect the limiting structure. For example, if the orbit converges to a 4-cycle for parameter value r_0, then four points should appear above r_0.

In order to get good speed we will compile functions when we can; this restricts us to machine precision, which means that in some cases (such as f_4) the results may be only qualitatively correct, as opposed to perfectly accurate. This distinction matters little in this application and the reader can use the ideas of §7.2 to compute the perfectly correct numbers if desired.

The basic compile construction, `Compile[{x}, f]`, defines a function from $\mathbb{R}$ to $\mathbb{R}$ But *Mathematica* can compile functions on lists, and that can really speed things up. First, an example. In the code that follows, `cf` is a function that takes two arguments: `r` and `x`, each of which is a list of reals. The `1` in the type specification indicates that `r` is to be an object of tensor rank 1 (i.e., a list). If 0 were used instead, `cf` would be a plain function of two real variables (and the 0 could be suppressed in that case).

```
cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, r + x];
cf[{1., 2., 3.}, {2., 3., 4.}]
```

```
{3., 5., 7.}
```

Here is how to use this idea to get the data we want. We work on the entire range of r-values at once, suppress the first 10 iterations, and then show 5 iterations. We transpose the output so that, for example, the last list will be the orbit corresponding to $r = 4$.

```
cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, r x (1 - x)];
rVals = Range[2., 4, (4 - 2)/(10 - 1)];
Transpose[NestList[cf[rVals, #1] &,
  Nest[cf[rVals, #1] &, (0.1 &) /@ rVals, 10], 4]]
```

```
{{0.5, 0.5, 0.5, 0.5, 0.5}, {0.550001, 0.55, 0.55, 0.55, 0.55},
 {0.590894, 0.590916, 0.590906, 0.59091, 0.590909},
 {0.622448, 0.626684, 0.62387, 0.62575, 0.624499},
 {0.627918, 0.674951, 0.633799, 0.670505, 0.638237},
 {0.582761, 0.756469, 0.573141, 0.761135, 0.565627},
 {0.7, 0.7, 0.7, 0.7, 0.7},
 {0.882004, 0.370037, 0.828834, 0.504421, 0.888819},
 {0.516488, 0.943417, 0.201661, 0.6082, 0.900217},
 {0.147837, 0.503924, 0.999938, 0.000246305, 0.000984976}}
```

We can compare the last list with the actual iteration.

```
NestList[f4, Nest[f4, 0.1, 10], 4]
```

```
{0.147837, 0.503924, 0.999938, 0.000246305, 0.000984976}
```

Finally, we attach the r-values in a way that makes it easy to turn the data into points. Each of the ten lists in the following output can be sent to `makePts`, which will turn it into a set of pairs over the r-value.

```
Column[Transpose[
  {rVals, Transpose[NestList[cf[rVals, #] &, Nest[cf[rVals, #] &,
    Array[ 0.1 &, 10] , 10], 4]]}] ]
```

```
{2., {0.5, 0.5, 0.5, 0.5, 0.5}}
{2.22222, {0.550001, 0.55, 0.55, 0.55, 0.55}}
{2.44444, {0.590894, 0.590916, 0.590906, 0.59091, 0.590909}}
{2.66667, {0.622448, 0.626684, 0.62387, 0.62575, 0.624499}}
{2.88889, {0.627918, 0.674951, 0.633799, 0.670505, 0.638237}}
{3.11111, {0.582761, 0.756469, 0.573141, 0.761135, 0.565627}}
{3.33333, {0.7, 0.7, 0.7, 0.7, 0.7}}
{3.55556, {0.882004, 0.370037, 0.828834, 0.504421, 0.888819}}
{3.77778, {0.516488, 0.943417, 0.201661, 0.6082, 0.900217}}
{4., {0.147837, 0.503924, 0.999938, 0.000246305, 0.000984976}}
```

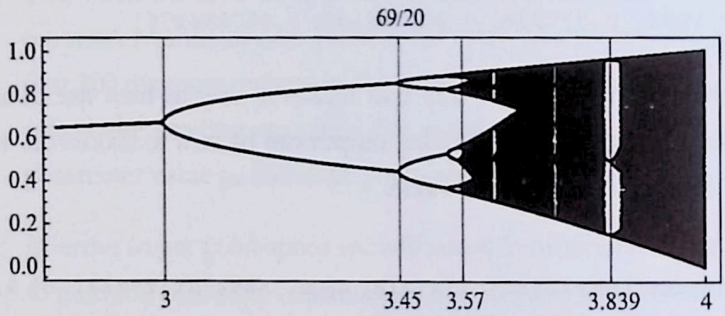
So now we put it all together, with an option to control the number of r-values.

```
Options[BifurcationPlot] =
  {PlotPoints → 2000, PlotStyle → {Black, PointSize[0.001]}};
BifurcationPlot[f_, {r_, a_, b_}, {x_, x0_}, {iter0_, iterShow_},
 opts___] := Module[{sty, n, makePts, cf, rVals, data},
  {sty, n} =
   {PlotStyle, PlotPoints} /. {opts} /. Options[BifurcationPlot];
  makePts[{s_, v_}] := ({s, #} &) /@ v;
  cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, Evaluate[f]];
```

```
rVals = Range[N[a], b, (b - a)/(n - 1)];
  data = Transpose[{rVals, Transpose[NestList[cf[rVals, #] &,
        Nest[cf[rVals, #1] &, Array[x0 &, n], iter0], iterShow]]}];
  Graphics[Append[Flatten[{sty}], Point[Flatten[makePts /@ data, 1]]],
   Sequence @@ FilterRules[{opts}, Options[Graphics]],
   AspectRatio → 1 / 3, Frame → True, FrameTicks →
    {Automatic, Range[0, 1, 0.5], None, None}, Axes → None]];
```

And here is the result with 2000 r-values; it is remarkable that this complicated graphic can be generated in a fraction of a second. In this image 100 points are shown above each r-value, after 500 iterations are run to eliminate the transient behavior.

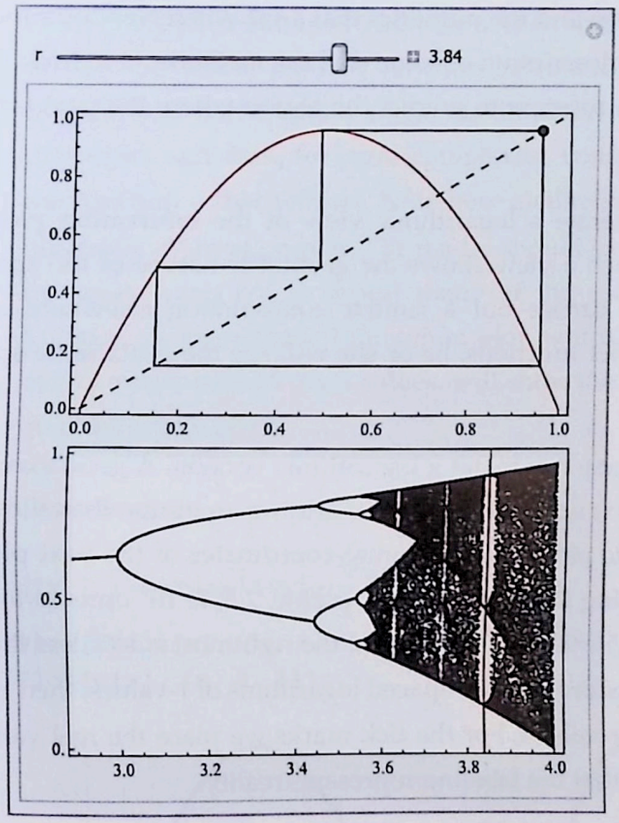
```
BifurcationPlot[f_r[x], {r, 2.8, 4}, {x, 0.5}, {500, 100}, FrameTicks →
  {{3, 3.45, 3.57, 3.839, 4}, Automatic, {{69 / 20, "69/20"}}, None},
 GridLines → {{3, 3.45, 3.57, 3.839}, None}]
```



We can turn this into a manipulation but then there are some efficiencies necessary to cut down the memory requirements of the bifurcation diagram. When $r < 3$ we can use a formula to get the one or two points we want. And `ListPlot` with a `PerformanceGoal` option set to `"Speed"` cuts down on memory usage. So here we retreat from list compilation to just compiling for reals r and x, and we use a quadratic formula for the 2-cycle, again compiling for speed.

```
Options[BifurcationPlotEfficient] =
  {PlotPoints → 2000, PlotStyle → {Black, PointSize[0.001]}};
BifurcationPlotEfficient[f_, {r_, a_, b_}, {x_, x0_},
  {iter0_, iterShow_}, opts___] := Module[{ n, cf, rVals, data },

{sty, n} =
  {PlotStyle, PlotPoints} /. {opts} /. Options[BifurcationPlot];
makePts[{s_, v_}] := ({s, #} &) /@ v;
cf = Compile[{r, x}, Evaluate[f]];

com1 = Compile[{rr}, (1 + rr + Sqrt[-3 - 2 rr + rr^2])/(2 rr)];

com2 = Compile[{rr}, (1 + rr - Sqrt[-3 - 2 rr + rr^2])/(2 rr)];
```

```
iter[rr_] := Which[rr < 3, {1 - 1 / rr},
      rr < 3 + 45 / 100, {com1[rr], com2[rr]}, True,
      NestList[cf[rr, #] &, Nest[cf[rr, #] &, x0, iter0], iterShow]];
rVals = Range[N[a], b, (b - a)/(n - 1)];
  data = {#, iter[#]} & /@ rVals;
  ListPlot[Flatten[makePts /@ data, 1],
   Sequence @@ FilterRules[{opts}, Options[Graphics]],
   Frame → True, FrameTicks → {Automatic, Range[0, 1, 0.5], None,
     None}, Axes → None, PerformanceGoal → "Speed", PlotStyle → sty]];
bifPlot = BifurcationPlotEfficient[f_r[x], {r, 2.9, 4},
   {x, 0.5}, {500, 32}];
Manipulate[Column[{CobwebPlot[f_r[x], {x, 0, 1},
    0.1, {300, 20}, ImageSize → 200],
   Show[bifPlot, GridLines → {{{r, Red}}, {}}, ImageSize → 200]}],
 {{r, 3.84}, 2.9, 4, Appearance → "Labeled"}]
```



A most remarkable aspect of the bifurcation diagram was discovered by Mitchell Feigenbaum in 1975 (and later proved by O. Lanford, P. Collet, and J.-P. Eckmann; see [CE]). The visible sequence of bifurcations from a fixed point to a 2-cycle to a 4-cycle to an 8-cycle to a 16-cycle continues through all the powers of 2 until the first chaotic r-value, 3.569945... is reached. The bifurcating values are at $r = 3$, 3.44949, 3.54409, 3.56441, 3.56876, 3.56969, 3.56989, 3.569943, 3.5699451, 3.569945557,.... These values increase at a uniform rate in that the distance between each bifurcat-

ing r-value and the limiting value 3.569945... is approximately 1 / 4.669 times the distance between the preceding bifurcation values and the limit. And more surprising still, this same speed of convergence occurs wherever bifurcations occur for the quadratic map. And the same convergence constant shows up whenever iterates of a sufficiently smooth function exhibit period-doubling behavior! Thus the constant 4.669... has come to be known as the *Feigenbaum number*. See chapter 3 of [Gle] for more on the discovery of the complexity of the orbits of the quadratic map. Chapter 1 of [CE] contains a lucid survey of these ideas, and later chapters have rigorous proofs.

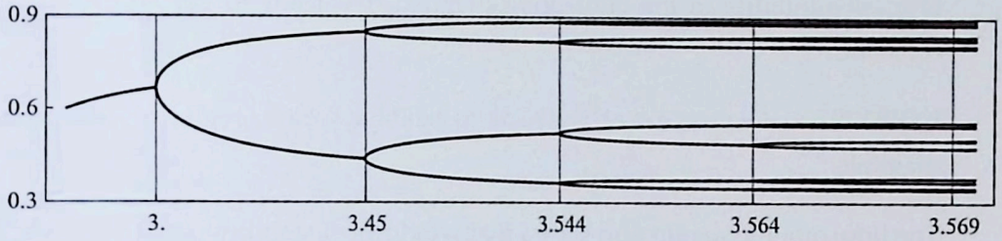
There are many computations that can be done to illustrate and investigate these phenomena further. For starters, one can work on finding the bifurcation points for the various bifurcation regimes that occur as r varies up to 4. This can be done by using root-finding techniques, but there are subtleties, in both the symbolic setting up of the functions in question and the numerics that arise when Newton's method is attempted. See [GG] for a description of some of these methods. One trick is that one can use the Feigenbaum constant to predict the region where the next bifurcation should occur.

Following [CE] we can generate a logarithmic view of the bifurcation plot as r varies from 2.5 to 3.5699; such a view shows the geometric nature of the speed of convergence. If the reader carries out a similar computation elsewhere in the domain of interest or for other functions, he or she will see the exact same spacing occurring.

We will revise `BifurcationPlot` to get a logarithmic version. A good exercise is to incorporate all this into `BifurcationPlot` by adding an option that allows the user to specify a logarithmic plot. The horizontal coordinates of the next plot run from roughly 0 to 3, indicating that the leftmost r-value, 2.5, is 10^0 units away from the limit point (`limitr` = 3.569945557391440) and the rightmost is 10^{-3} less than the limit. We use `rValsLog` to store equally spaced logarithms of r-values; then `rVals` is the set of corresponding r-values. For the tick marks we place the real values at the logarithmic positions so that the labeling represents reality.

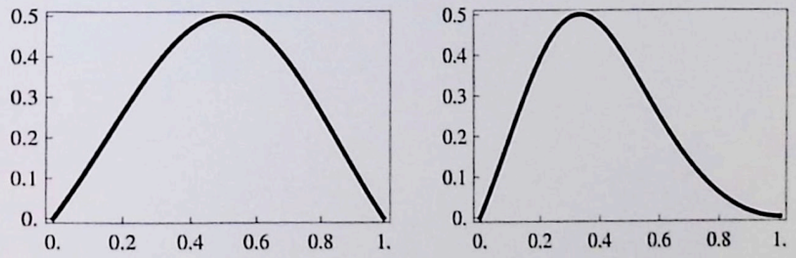
```
bifPts = {3, 3.45, 3.54409, 3.56441,
   3.56876, 3.56969, 3.56989, 3.569943, 3.5699451};
limitr = 3.569945557391439;
logMin = Log[10, limitr - 2.5];
n = 2000; x0 = 0.1; iter0 = 1000; iterShow = 64;
makePts[{s_, v_}] := ({s, #1} &) /@ v;

cf = Compile[{{r, _Real, 1}, {x, _Real, 1}}, Evaluate[f_r[x]]];

rValsLog = Range[logMin, -3, (-3 - logMin)/(n - 1)];
```

```
rVals = limitr - 10^rValsLog;
data = Transpose[{-rValsLog, Transpose[NestList[cf[rVals, #] &,
       Nest[cf[rVals, #] &, Array[x0 &, n], iter0], iterShow]]}];

Graphics[{PointSize[0.0013], Point[Flatten[makePts /@ data, 1]]},
 Frame → True, AxesOrigin → {0, 0.3},
 GridLines → {-Log[10, limitr - bifPts], None}, FrameTicks →
  {({-#, NumberForm[limitr - 10^#, 4]} &) /@ Log[10, limitr - bifPts],
   {0.3, 0.6, 0.9}, None, None}, PlotRange → {0.3, 0.9}]
```



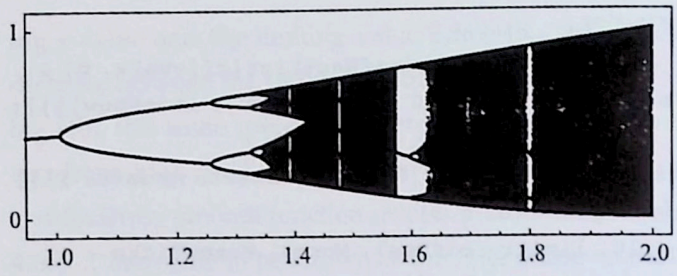
Note that `BifurcationPlot` was written so that we can use any function in place of $x(1-x)$, where the new function has the same general shape: it maps the unit interval into itself, with a single maximum. If we modify e^{-x^2} to have the desired properties and then, for extra complexity, compose it with a sine function, we get the function g that follows. Note how qualitatively similar the plot is to the classic quadratic bifurcation plot. The reader should look at the graphs of this function and its iterates and try to repeat many of the computations of this section for this function. A logarithmic bifurcation plot, which requires the preliminary computation of various bifurcation values, will show the same speed of convergence that the quadratic map does.

```
eFunc[x_] := 1/2 e/(e - 1) (e^-(2 x-1)^2 - 1/e);
g[x_] := eFunc[Sin[π x/2]];
Plot[eFunc[x], {x, 0, 1}]
Plot[g[x], {x, 0, 1}]
```



And here is a bifurcation plot for this new mapping.

```
BifurcationPlot[r g[x], {r, 0.95, 2}, {x, 1/10}, {1000, 128},
 PlotPoints → 1000, PlotRangePadding → {{0, 0.03}, {0.1, 0.1}},
 FrameTicks → {Automatic, {0, 1, 2}, None, None}]
```



We have barely scratched the surface of the dynamics of the quadratic map. There is a self-similarity in the bifurcation diagram that leads to yet another universal constant.

EXERCISE 4. Use `BifurcationPlot` to examine various regions of the preceding diagram.

The bifurcation diagram also shows that windows of stability (regions where f_r has an attracting n-cycle) are intermixed with chaotic regions. This leads to the still unsolved problem of clarifying the nature of the set of chaotic parameters. It has been conjectured [CE, p. 31] that each subinterval of [2, 4] contains a stable interval (that is, the nonchaotic parameters are dense in [2, 4]), but this has not been proved. On the other hand, the related conjecture that the set of chaotic parameters has positive measure has been proved (M. V. Jacobson, 1981). Finally, we mention that the orbit behavior of the quadratic map is related to iterations of quadratic maps in the complex plane and the Mandelbrot set (see [Dev1, §3.8; Dev2, Chap. 8]).

The field of real and complex dynamical systems is very reliant on computation, both numerical and graphical. Some of those computations, especially those that generate high-resolution color images requiring lots of computation for each pixel, require a mainframe computer and a faster programming language than *Mathematica*. Yet *Mathematica*'s ease of programming and combination of numerical and graphics abilities make it a good tool for preliminary investigations, and the enhanced compilation features can be used to improve speed.