Chapter 3:

Fixed Points, Cycles, and Chaos

Goals:

- To characterize the periodic orbits of the logistic map using the Newton-Raphson method.
- To understand the concept of stability of an orbit.
- To understand and to construct Java objects.

In the previous chapter we began to explore some of the behavior present in the logistic map. Now we want to start characterizing quantitatively the various types of behavior displayed by this map. In the simplest behavior the motion settles down to a single value of x. A special value of x which is repeated iteration after iteration is called a fixed point. Mathematically, this special value, x^*, obeys:

$$x^* = f(x^*) \qquad (3.1)$$

where $f(x)$ is the mapping function. Another way of expressing this is to say $F(x^*) = 0$, where $F(x) \equiv x - f(x)$.

One way to find fixed points is by drawing graphs.

Exercise 3.1. How many fixed points are there for the mapping function

$$f(x) \equiv c \sin(x) ?$$

How does this number depend upon c (assume $c \geq 0$)? In particular, for which ranges of c are there 0, 1, 2, or 3 fixed points of $f(x)$ in the region $x \geq 0$? In other words, you want to know the number of solutions of to the equation

$$x = f(x) = c \sin(x).$$

There is a standard way of attacking such a problem. Simply graph x and $f(x)$ and notice how often the graphs cross. The GraphMaker class from the previous chapter could come in handy here.

* * *

Exercise 3.2. How many (real) solutions are there to the equation

$$f(x) \equiv x^7 - x^6 + 6x + 10 = 0 ?$$

Using GraphMaker, get a rough estimate (within 10%) of the value(s) of the real root(s).

We have seen that computing the fixed points of a function f(x) is the same as computing the zeros of the function F(x) = f(x) - x. Graphing the functions allows us to visualize what is going on, but it isn't a very accurate way to determine the zeros. So, we will use a different numerical technique to find the zeros of a differentiable function.

A. The Newton-Raphson Method. Assume we have a smooth function $F(x)$ and that we wish to find a zero of F, i.e. a value X for which

$$F(X) = 0 . \qquad (3.2)$$

We do not know X. Instead, we have a value X_0 which approximates X. Perhaps we obtained this value from studying the graph of $F(x)$, or perhaps it is purely a guess. The value X_0 differs from the correct zero X by some unknown error Δ:

$$X = X_0 + \Delta \qquad (3.3)$$

where we hope Δ is small.

To refine our estimate of X, we expand the function $F(x)$ in a Taylor series about X_0:

$$F(X) = F(X_0 + \Delta) = F(X_0) + \Delta F'(X_0) + O(\Delta^2) = 0 . \qquad (3.4)$$

Here $F'(X_0)$ is the derivative of our function at the known point, X_0, and $O(\Delta^2)$ indicates a term of order Δ^2, which we neglect as (hopefully) small. Solving equation (3.4) gives an estimate for the error Δ:

$$\Delta = -\frac{F(X_0)}{F'(X_0)} \qquad (3.5)$$

* * *

which allows us to improve our estimate of the root to the new value, X_1:

$$X_1 = X_0 + \Delta = X_0 - \frac{F(X_0)}{F'(X_0)} \qquad (3.6)$$

If the $O(\Delta^2)$ terms (which we neglected) were small enough, then X_1 should be closer to the real root X than our original guess X_0. We can then repeat the process, producing another estimate X_2 which improves on X_1, and so on until we're happy with the accuracy of our estimate.

For example, one can find a zero of $F(x) = \cos(x)$ using the applet "Root:"

```
// Root.java
// Uses Newton-Raphson method to find zero of f(x)
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Root extends Applet implements ActionListener
{
        Label trialLabel, resultLabel, functionLabel;        // labels for the text fields
        TextField trial, result, function;

        public void init()                                   //      set up display
        {
                trialLabel = new Label( "Enter trial x value and press return" );
                trial = new TextField( 20 );
                trial.addActionListener( this );
                resultLabel = new Label( "New x value:" );
                result = new TextField( 20 );
                result.setEditable( false );
                functionLabel = new Label ( "f(new x value):" );
                function = new TextField( 20 );
                function.setEditable( false );

                add( trialLabel );                   //installs boxes for input
                add( trial );
```

* * *

```
        add( resultLabel );
        add( result );
        add (functionLabel );
        add ( function );
}

public void actionPerformed( ActionEvent e )
{
        double x, newx, delta;                  //  variables used
        x = new Double (trial.getText() ).doubleValue();   // get trial value
        delta = - f(x)/dfdx(x);                         //  estimated error
        newx = x + delta;                               //  new approximation
        result.setText( fulldouble(newx) ); //  print out new estimate
        function.setText( fulldouble(f(newx)) );    //  print out f(x)
}

double f (double x )   {
        return Math.cos(x);                             //  the function
}

double dfdx (double x ) {
        return -Math.sin(x);                //  the derivative of the function
}
String fulldouble (double x) {              //  converts double to string
        int p[];                            //  with 19 digits of accuracy
        int n;
        String s;
        p = new int [20];
        if (x>0)       {
                p[0] = (int) Math.floor(x);
                s = "" + Integer.toString(p[0]) + ".";
        }
        else           {
                x = - x;
                p[0] = (int) Math.floor(x);
                s = "-" + Integer.toString(p[0]) + ".";
        }
        for (n=1; n<=19; n++) {
```

* * *

```
                x = 10.*(x - p[n-1]);
                p[n] = (int) Math.floor(x);
                s = s + Integer.toString(p[n]);
            }
            return s;
        }
}
```

Program 3.1 Applet that uses the Newton-Raphson method to find a zero of $F(x) = \cos(x)$. Notice that we have replaced the Sun-supplied method 'Double.toString' with our own method 'fulldouble,' which returns a String which contains the decimal representation of a real number to 19 decimal places.

Start by setting x to some reasonable value, say 0.5. Then, by the approach outlined above, subsequent iterations should approach a zero of the cosine.

Problem 3.1 . Finding Roots. The cosine has many roots. As a function of the starting value of x, which root is found? When don't you get any root? Answer the same questions for the function $f(x) \equiv x^2 - x - 1$. Also, refine your estimate(s) of the root(s) of $f(x) \equiv x^7 - x^6 + 6x + 10$ (from exercise 3.2).

Menu Project. Redo Problem 3.1 for the complex roots of the function f(z) = z^3 – 7z + 6. The Newton-Raphson method works just the same way for complex numbers as reals. Try to show how the root found depends on the starting value of z. Generate some graphical representation of your answer. (You can see some of the answer on pages 116 and 117 of a book by Peitgen and Richter.[1])

Exercise 3.3 . Precision. It is interesting to see how fast the Newton-Raphson method converges. Please verify that the error in a given step of Newton-Raphson calculation goes as the square of the error in the previous step. Hint: Apply the method to a polynomial function when studying the error.

---

[1] H-O. Peitgen and P.H. Richter The Beauty of Fractals Springer-Verlag Berlin 1986.

* * *

Exercise 3.4 . Finding Fixed Points.

3.4.a. Find two non-negative fixed points x* of the mapping $f(x)=1.5\sin(x)$, with an accuracy of eight decimal digits. Recall that fixed points obey $x^* = f(x^*)$.

3.4.b. Find the fixed point values for the logistic map $f(x)=rx(1-x)$. Find them analytically or, if need be, on the computer.

B. Cycles . A cycle of length N is a set of values generated by a mapping ($x_{j+1} = f(x_j)$) possessing the periodic property

$$x_{j+N} = x_j \qquad (3.7).$$

Last week we saw just such a cycle in the logistic map. Recall Figure 2.1. For r=3.3, the logistic map settles into an oscillation between two values of x. This is a cycle of length 2, because the iterates satisfy $x_{j+2} = x_j$.

A cycle of length N is a fixed point of the mapping

$$x_{j+N} = f^{(N)}(x_j) \qquad (3.8)$$

where $f^{(N)}(x)$ means 'N applications of the mapping f to the initial value x' (NOT the Nth derivative of f). The cycle values are the fixed points of equation (3.8); they satisfy $x^* = f^{(N)}(x^*)$. Clearly, there must be at least N fixed points to the mapping $f^{(N)}(x)$ for the mapping $f(x)$ to have a cycle of length N. If this is not the case, then there is no cycle of length N. Unfortunately, $f^{(N)}(x)$ is generally not very easy to work with (for the logistic map it is a polynomial of order 2N). However, if one can find a good way of calculating $f^{(N)}(x)$ and its first derivative, one could use the Newton-Raphson method to find the elements of the corresponding N-cycle.

In Required Project I, we ask you to examine the elements of the cycle of length two, three, and four for the map $f(x)=rx(1-x)$ for $0<r<4$. To do this calculation numerically, apply the Newton-Raphson method to the mapping function $f^{(2)}(x)$. There are two ways to calculate the necessary derivative. One is to compute it

* * *

analytically, either by writing out the resulting polynomial and differentiating, or by applying the chain rule of differentiation to $f(f(x))$. The second way is to approximate it numerically using the definition of a derivative:

$$\frac{df(x)}{dx} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \qquad (3.9)$$

and using a small but finite δ in the computation. In this method you don't need to know the derivative explicitly; the values of the function are sufficient. When approximating the derivative in this fashion, the Newton-Raphson method becomes known as the secant method.

Exercise 3.5. Accuracy. Compare the two methods of calculating derivatives described above. If one estimates a derivative from the difference formula (eq. 3.9), how does the error in the result depend upon the value of δ? What value of δ would be best for calculating the derivative of $f^{(2)}(x)$?

It is also possible to study the N-cycle maps $f^{(N)}(x)$ using the graphical technique discussed earlier. Simply plot the curves $y = f^{(N)}(x)$ and $y = x$ on the same graph and look for the intersection points. The intersections are elements of the N-cycle.

C. Stability. Last week we saw that the logistic map settles down to a fixed point only when r<3. In Required Project 1 you will show that r=3 is a bifurcation point, where a two-cycle appears in the iterates of the map. At larger values of r there are more bifurcations to longer cycles. The fixed point (1-cycle) solution still exists mathematically for r>3, but we no longer see it. An obvious question to ask at this point is "why do the bifurcations occur?" The answer to this questions lies in the notion of stability. As an example of a system in which stability obviously determines the long-term behavior, consider a pencil balanced on its point. One would like to conclude that in the long run, the steady state of the system is that with the pencil lying on its side. But if the pencil is balanced perfectly on its point, and if there are no perturbations, it will stay balanced on its tip for a long time. Obviously this state is unstable, in the sense that any small perturbation will destroy this state and result in the state with the pencil lying on its side. Thus if we know something about the stability of various states of the system, then we can make predictions about the long-time motion.

* * *

D. Linearization of 1d map. In Required Project 1 you look in detail at the behavior of the logistic map in the vicinity of the bifurcation point r=3. Here we will explore a systematic method for studying the regions near bifurcation points.

Consider a one-dimensional map $f(x)$. Suppose the point x* is a fixed point of f, i.e. $f(x^*) = x^*$. Then we are interested in how points near x* behave under iteration. We want to know, specifically, do they move closer to x*, or do they move away? This is a useful thing to know, since if, for example, nearby points tend to move closer to x* and ultimately land on it, then we will know the long term behavior of the map (at least for initial conditions close to x*): after an initial transient period, the orbit is approximately x*,x*,x*. . . . (This is analogous to the situation with the pencil lying on its side).

So how do we decide how a nearby point behaves under iteration? Let us assume that f is smooth, and (for now) that $f'(x^*) \neq 0$. Then we can Taylor expand the iterate of a point x_0, close to x*:

$$x_1 \equiv f(x_0) = f(x^* + (x_0 - x^*)) \approx f(x^*) + (x_0 - x^*) f'(x^*)$$

or,

$$x_1 \approx x^* + (x_0 - x^*) f'(x^*)$$

(since $f(x^*) = x^*$), so that

$$x_1 - x^* \approx (x_0 - x^*) f'(x^*) \ . \qquad (3.10).$$

In other words, multiplying the initial separation, $x_0 - x^*$, by $f'(x^*)$ gives the separation after one iteration, $x_1 - x^*$. So the rule is easy to see: if |f'(x*)| is larger than 1, then the separation increases; the fixed point is called unstable. If the derivative is between -1 and +1, the iterates get closer to the fixed point; in this case, the fixed point is said to be stable. A stable fixed point is sometimes called an attractor, because nearby points move closer to it under iteration. When $f'(x^*) < 0$, the iterates alternate from one side of the fixed point to the other.

* * *

Let us extend this discussion to the two cycle. Label the stable points as x^* and y^*, and consider an expansion of the two-cycle about x^*:

$$x_2 = f^{(2)}(x_0) \approx f^{(2)}(x^*) + \frac{d}{dx}\left(f^{(2)}(x)\right)\Big|_{x=x^*}(x_0 - x^*) = x^* + \frac{d}{dx}\left(f^{(2)}(x)\right)\Big|_{x=x^*}(x_0 - x^*),$$

which implies

$$x_2 - x^* \approx (x_0 - x^*)\frac{d}{dx}\left(f^{(2)}(x)\right)\Big|_{x=x^*}.$$

The term $\frac{d}{dx}\left(f^{(2)}(x)\right)\Big|_{x=x^*}$ can be found by the chain rule

$$\frac{d}{dx}\left(f^{(2)}(x)\right)\Big|_{x=x^*} \equiv \frac{d}{dx} f\left(f(x)\right)\Big|_{x=x^*} = \frac{df(y)}{dy}\Big|_{y=f(x^*)} \frac{dy}{dx}\Big|_{x=x^*} = \frac{df(y)}{dy}\Big|_{y^*} \frac{df(x)}{dx}\Big|_{x^*}.$$

Thus, $\frac{d}{dx}\left(f^{(2)}(x)\right)\Big|_{x=x^*}$ is simply the product of the derivatives of the function at the two stable cycle points.

For higher iterates, we just play the same game: the separation of the Nth iteration from the fixed point is easily seen to be

$$x_N - x^* \approx (x_0 - x^*)\frac{d}{dx}\left(f^{(N)}(x)\right)\Big|_{x=x^*}. \qquad (3.11)$$

The derivative $f'(x^*)$ is often called the Floquet multiplier of the fixed point, and is denoted by Λ. Note that equation (3.11) implies that near the fixed point, the iterates converge to (or diverge away from) the fixed point geometrically. The geometric convergence holds with increasing accuracy for higher iterates because the neglected terms in the Taylor series become more insignificant as the fixed point is approached. However, geometric divergence from the fixed point eventually must break down, because the neglected terms become important and equation (3.10) breaks down.

* * *

So now we have answered the question we started with, and we see that the answer depends on the slope of the map at the fixed point. There are some special cases we still have to consider, namely $f'(x^*) = 0, \pm 1$. But before we do, there is a graphical method of depicting orbits that we should look at.

Consider a map such as the logistic map, shown in figure 3.1. If we also plot the line y = x, then the intersection of this line with

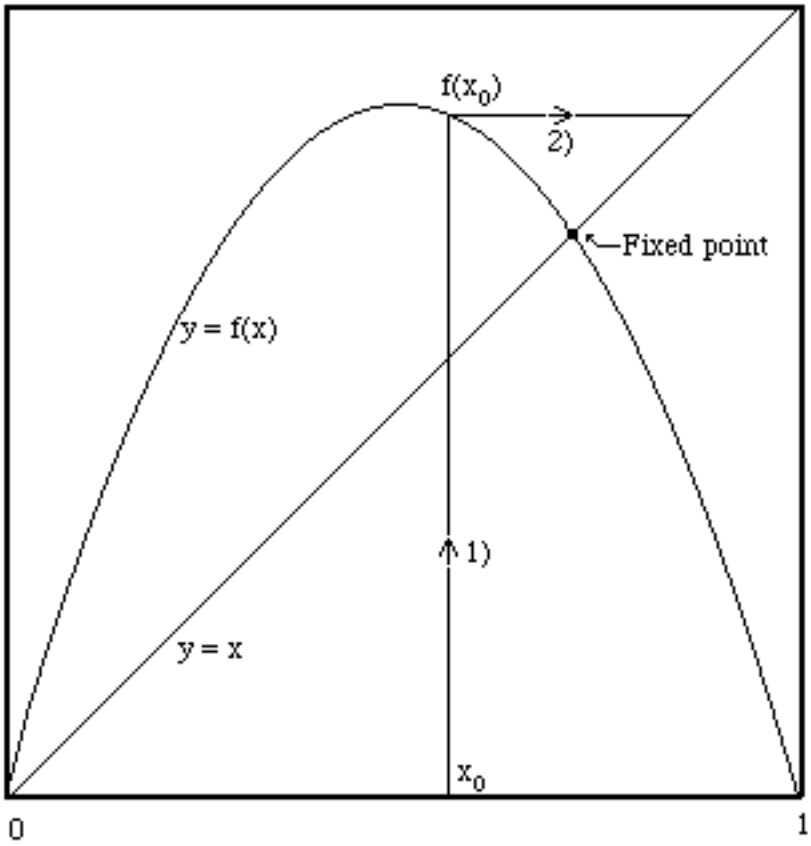

Figure 3.1. Graphical Method of Depicting Orbits

the curve of our map is a fixed point. To determine how a point such as the point x_0 iterates under the map, we do the following:

1) From the point x_0 on the horizontal axis, go up vertically until you hit the curve.

2) From this point, go horizontally (either left or right) to the line $y = x$. The value of x here is the iterate, $x_1 = f(x_0)$.

* * *

3) From this point, move vertically to the curve, then horizontally to the line, etc. In this way, a succession of iterates is determined.

In figure 3.2, this procedure is employed for the logistic map with r=3.2. As you can see, the two cycle is stable (i.e. attracting).

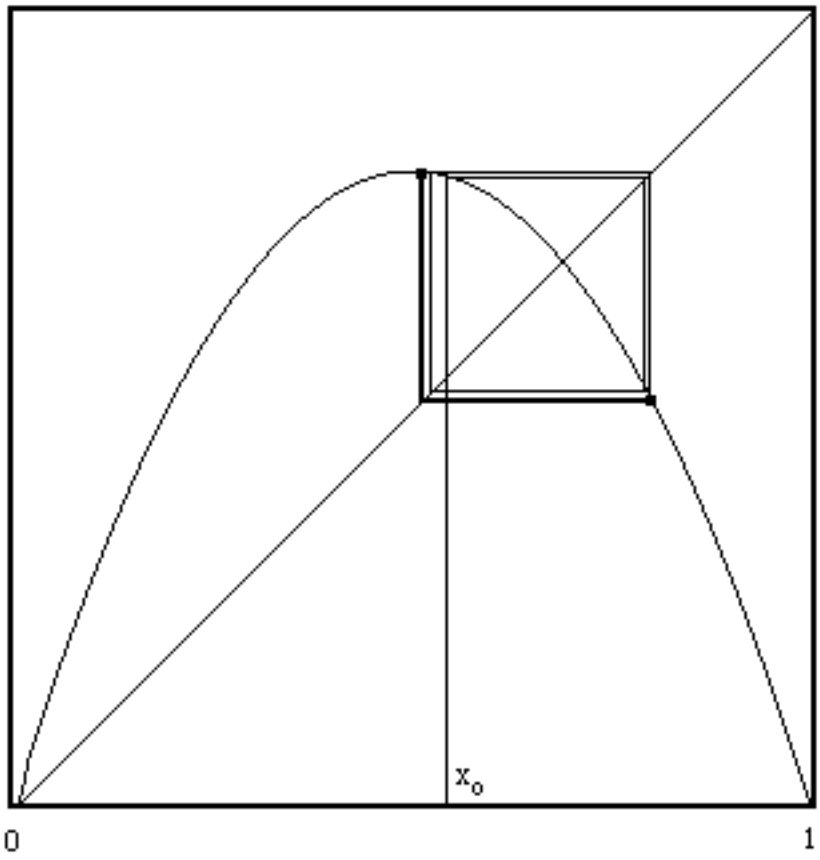

Figure 3.2. Graphical Method Depicting a Two-Cycle.

We return now to the special cases which are not included in our analysis. First, suppose $f'(x^*) = 0$. Then equation (3.10) breaks down, and we must expand to second order, to find

$$x_1 - x^* \approx \frac{1}{2}(x_0 - x^*)^2 f''(x^*). \qquad (3.12)$$

* * *

Exercise 3.8. Convergence. Equation (3.12) says that $(x_n - x^*) \approx k(x_{n-1} - x^*)^2$, where $k \equiv \frac{1}{2} f''(x^*)$. From this, derive an expression for $(x_n - x^*)$ in terms of k and $(x_0 - x^*)$.

In this case, the fixed point is called superstable; the convergence is faster than geometric.

When $f'(x^*) = \pm 1$, equation (3.10) predicts that the separation remains the same. What really happens depends on the second derivative of f at the fixed point. For a map like the logistic map where the second derivative is negative, the orbits converge to the fixed point, but slower than for $|f'(x^*)| < 1$. This case is called marginally stable. For maps with a positive second derivative, the orbits slowly diverge from the fixed point.

So now we have classified all possible behaviors for a smooth one-dimensional map near a fixed point. The important idea is that the classification depends on local properties of the map at the fixed point, and were obtained by linearization (i.e. first order Taylor series). We will turn to the somewhat more complicated case of two dimensional systems in a later chapter.

E. Exact Solutions. The logistic map is exactly solvable at the two opposite limits of the range of r-values under consideration. For r = 0, there is the trivial solution in which $x_i = 0$ for all $i > 0$ regardless of the initial point x_0.

The case r = 4 is more interesting. To solve that case we make a change of variables from x_j to θ_j defined by

$$x_j = \frac{1}{2}\left(1 - \cos 2\pi\theta_j\right) \tag{3.13}$$

Notice that θ_j is not uniquely defined. Any one of the changes

$$\theta_j \rightarrow \pm\theta_j + n \tag{3.14}$$

for any integer n leaves x_j unchanged. Now set r = 4 and substitute definition (3.13) into the recursion equation $x_{j+1} = 4x_j\left(1 - x_j\right)$. After a bit of fussing with trigonometric identities, the result becomes

* * *

$$\left(1-\cos 2\pi\theta_{j+1}\right)=\left(1-\cos 4\pi\theta_j\right)\ , \tag{3.15}$$

which then has a solution

$$\theta_j = 2^j\theta_0\ . \tag{3.16}$$

This rather simple solution enables one to understand some aspects of the chaos which arises at r = 4 rather precisely. We ask you to explore this in Required Project I.

F. More Java: Objects and Classes. In this course we will be examining several systems with different rules for how they evolve in time. So we'd really like to have some way to define something called a DynamicalSystem, which embodies many different evolution rules with different numbers of variables and parameters. One type of DynamicalSystem could be LogisticMap; another type could be, say, DampedPendulum. But in all cases our applet could ask the DynamicalSystem to calculate its own evolution (and even maybe find its own periodic orbits and plot its own graphs). Java enables us to do exactly this by defining classes of objects.[2] Indeed, in a few weeks we will define and use a DynamicalSystem class.

We start here by discussing how every applet that we have written uses objects. Next we present a class Complex that contains operations for complex numbers (which you might well find useful for the Menu Project described in this chapter). Then finally we will look at the classes Dataset, GraphMaker and Util from Chapter 2.

One way to think about objects is that they are packages that contain both data and methods that operate on the data. Each object typically has many methods in it. So far, we have added methods to pre-existing objects and put these objects together to perform the tasks in our programs. For example, an applet is an object that automatically calls the methods `init` and `paint`, among other things. In the applet "FirstMap," when we declare:

```
public class FirstMap extends Applet
```

---

[2]A note about Java: Some of you may have heard that Java is "object-oriented" (like some other computer languages such as C++) rather than "procedure-oriented" (like Fortran, Basic, C, and Pascal). The objects we're discussing here are the reason for the term "object-oriented".

* * *

our applet knows about ("inherits") all the methods that Sun has already put in the Applet class. We will discuss inheritance in more detail in Chapter 6. So “FirstMap” automatically calls init and paint. To get “FirstMap” to actually do something, we define our own versions of init and paint that get executed instead of the Sun-supplied Applet versions (which do nothing). We also defined the new methods f, fn, ifromn, and jfromx that we added to the class “FirstMap.”

We used more objects when we read in the value of r from the TextField on the applet. (Unfortunately, in Java reading in a double number from a TextField involves fairly sophisticated use of objects. But if you understand this process, you are well on the way to being object-oriented.) The TextField itself is an object (or, more specifically, we define inputr to be an object in the TextField class), which comes packaged with a method getText that takes whatever is on the TextField and converts it to a String:

```
s = inputr.getText();      //  read string in TextField inputr
```

We then convert the String s into the double variable r :

```
r = new Double(s).doubleValue();      //  convert string to double variable r
```

This statement first converts the String s into a Double object (not a double number!) and then calls the method doubleValue() in the Double class, whose output is the double number r. You might find it easier to understand the more verbose three-step process:

```
Double temp;                 //  declare temp as a Double object
temp = new Double(s);        //  allocate temp; define its value as the Double
                             //  conversion of String s
r = temp.doubleValue();      //  call doubleValue method to convert Double object
                             //  temp into double value r
```

The methods in the class Double are all documented at the web page http://java.sun.com/products/jdk/1.1/api/java.lang.Double.html. The CodeWarrior help also has documentation for the Sun-supplied class libraries.

Now we create our own class Complex. The behavior of objects in this class should remind you of that of complex numbers $z = x + iy$, with x and y real, and $i = \sqrt{-1}$. We write two Java files, one with the class Complex itself, and second with an applet “ComplexTest” which demonstrates use of the class Complex.

* * *

```
//   Complex.java
//   definition of class Complex        (for complex numbers)

public class Complex  {
    private double real;        //  real and imag are the instance variables of the class
    private double imag;        //  private means real and imag are invisible outside
                                //  the class Complex

    Complex(double x, double y) {    //  The constructor method, called each time a
                                     //  new instance of the class is set up.
        real = x;
        imag = y;
    }
    //  now define the methods for the class

    //     Get real part
    public double realpart() { return real; }

    //     Get imaginary part
    public double imagpart(){ return imag; }

    //     Add two Complex numbers
    public Complex plus(Complex c2)    {
        return new Complex(real + c2.real, imag + c2.imag);
    }

    public static Complex plus(Complex c1, Complex c2)   {
        return new Complex(c1.real + c2.real, c1.imag + c2.imag);
    }

    //     Subtract two Complex numbers
    public Complex minus(Complex c2) {
        return new Complex(real - c2.real, imag - c2.imag);
    }

    //     Multiply two Complex numbers
    public Complex times(Complex c2)  {
```

* * *

```
        return new Complex(real*c2.real - imag*c2.imag,
                           real*c2.imag + imag*c2.real);
    }

    public static Complex times(Complex c1, Complex c2) {
        return new Complex(c1.real*c2.real - c1.imag*c2.imag,
                           c1.real*c2.imag + c1.imag*c2.real);
    }

    //  Divide two Complex numbers
    public Complex divideby(Complex c2)   {
        double denom;
        denom = c2.real*c2.real + c2.imag*c2.imag;
        if(denom == 0)   {
            return new Complex(Double.NaN, Double.NaN);
        }                          //  return Not-a-Number if dividing by zero
        else    {
            return new Complex ((real*c2.real + imag*c2.imag)/denom,
                        (imag*c2.real - real*c2.imag)/denom);
        }
    }                          //  end of divideby method

    //  Converts Complex to String (needed to display on applet)
    public String toString() {
        return "(" + real + ", " + imag + ")";
    }
}
```

Program 3.2 Class Complex, for complex numbers.

The first method in the class definition, whose name is the same as the name of the class, is called the constructor. It is called when a new instance of the class is declared in an applet. In this case the constructor just initializes the instance variables real and imag, giving them the values of the first and second arguments with which the constructor is called. The other methods define various standard operations, including adding, subtracting, multiplying, and dividing.

Here is an applet that uses the Complex class:

* * *

```
// ComplexTest.java
// applet that puts the class Complex through some of its paces
import java.awt.*;
import java.applet.Applet;

public class ComplexTest extends Applet {
    private Complex a, b;

    public void init() {
        a = new Complex ( 5.5, 9.3 );
        b = new Complex ( 21.3, 15.0 );
    }

    public void paint(Graphics g)    {
        g.drawString("a = " + a, 25, 25 );
        g.drawString( "b = " + b, 25, 40 );

        g.drawString( "a + b = " + a.add( b ), 25, 70 );
        g.drawString( "a - b = " + a.subtract( b ), 25, 85);
    }
}
```

Exercise 3.9 . What happens when you try to access one of the private instance variables in Complex from the applet "ComplexTest"?

Exercise 3.10. What happens if the method toString is not defined in the class Complex?

Exercise 3.11. How would you modify the class Complex to be able to calculate the modulus and phase of a complex number?

G. The classes GraphMaker, Dataset, and Util. Finally we show you the contents of the classes we used in the previous chapter to draw graphs. We start with the class Dataset, which bundles together two arrays into a single object.

```
// file Dataset.java
class Dataset {
```

```
    private double[] xdata;                 //  instance variables
    private double[] ydata;
    private int npts;

// constructor for DataSet class:
    public Dataset(double[] dx, double[] dy, int n)  {
        npts=n;                             //  initialize instance variable and arrays
        xdata = new double[ n ];            //  set size of arrays
        ydata = new double[ n ];
        setXdata(dx, n);                    //  set values of arrays
        setYdata(dy, n);
    }

    public void setXdata(double[] dx, int n) {          //  sets xdata array
        for (int m=0; m<n; m++) {
            xdata[m]=dx[m];
        }
    }

    public void setYdata(double[] dy, int n) {          //  sets ydata array
        for (int m=0; m<n; m++) {
            ydata[m] = dy[m];
        }
    }

    public double[] getXdata()    {return  xdata; }
    public double[] getYdata()    { return ydata; }
    public int length() {return npts;}

    // methods to find the maximum and minimum values of the x
    // and y coordinates

    public double xmax() {return Util.max(xdata, npts);}
    public double xmin() {return Util.min(xdata, npts);}
    public double ymax() {return Util.max(ydata, npts);}
    public double ymin() {return Util.min(ydata, npts);}
}
```

* * *

Now you may be wondering why we have bothered to define a Dataset object when we could have defined a two-dimensional array with both sets of coordinates. But suppose that some of the data came with additional information, such as who took it, when it was taken, or even axis labels. All this information can all be added into a Dataset, and moreover it is possible to do this while keeping all the parts used by our current version of GraphMaker unchanged. Thus, changes to Dataset do not propagate through the entire program.

Problem 3.2. Class Dataset. Modify the class Dataset so that it also keeps track of a dataTaker and a dataDate. Write an applet that creates a Dataset, uses GraphMaker to make a graph of the Dataset, and also prints out the dataTaker and dataDate onto the applet.

Now we show you the class GraphMaker, which takes either one or two Datasets and plots them on a graph. This class is long, but each piece is rather straightforward.

```
//  file GraphMaker.java
// class to make a graph
//  can plot either one or two datasets

import java.awt.*;
public class GraphMaker extends Canvas {    //  GraphMaker is subclass of Sun-supplied
                                            //  class java.awt.Canvas

    private int height;                 //  height of canvas
    private int width;                  //  its width
    private final int offi = 0;         // the graph starts at this x-coordinate on canvas
    private final int  offj = 40;       // the graph starts at this y-coordinate on canvas

    private final double bordl = 0.15;    // these constants give borders around graph
    private final double bordr = 0.1;     //  expressed as a portion of the entire picture
    private final double bordt = 0.1;
    private final double bordb = 0.2;

        // the user might well wish to change the constants defined above

    private int i0, j0, i1, j1;     // positions of bottom-left and top-right of plot area
```

* * *

```
private final int ngridsi = 4;          // number of grid ticks in x direction
private final int ngridsj = 4;          // number of grid ticks in y direction
private final int ticklength = 12;      // length of tick lines

private double scalefx, scalefy, offx, offy;
    /*  these variables are scale factors and offsets describing the relation
    //  between x,y and i,j coordinates.
    */
private double maxx, maxy, minx, miny;          // limits on x and y values in graph
private double xrange, yrange;              // ranges of x and y values after rounding
private double intx, inty;                      // intervals between ticks

private Dataset mydata1, mydata2;               //  Datasets to be plotted
private int ncurve;                             // number of curves to plot

// constructor for GraphMaker also sets size:
public GraphMaker( int w, int h)    {
    super();                            //  call Canvas constructor
    width=w;                            //  width of canvas
    height=h;                           //  height of canvas
    this.setSize(width, height);        // set canvas size
    this.setBackground( Color.white );  //  set white background
    ijset();                            // set i,j coords of graph corners
    ncurve = 0;                         // no curves until setData sets the Datasets
}

private void ijset()   {
    i0 = (int) Math.round(bordl*width);
    j0 = height - (int) Math.round(bordb*height);     // lower lh corner of graph
    i1 = (int) Math.round((1 - bordr) * width);
    j1 = height - (int) Math.round((1 - bordt) * height);     // upper rh corner
}

public void setData(Dataset d1)  {
    mydata1 = d1;
    ncurve = 1;
}
```

* * *

```
    public void setData(Dataset d1, Dataset d2) {
        mydata1 = d1;
        mydata2 = d2;
        ncurve = 2;
    }

    private int ifromx( double x)    {
    //  converts drawing variable x into screen variable i
        return i0 + (int) Math.round(scalefx * (x - offx));
    }

    private int jfromy(double y){
    //  converts drawing variable y into screen variable j
        return j0 - (int) Math.round(scalefy * (y - offy));
    }

    private void drawCurve(Graphics g, Dataset d)  {
        double dax[], day[];
        int n;
        n = d.length();
        dax = new double[n];
        day = new double[n];
        int m;      // loop variable

        dax = d.getXdata();
        day = d.getYdata();

        for(m=0; m<n-1; m++) {
            g.drawLine(ifromx(dax[m]), jfromy(day[m]),
                       ifromx(dax[m+1]), jfromy(day[m+1]));
        }
    }

    private void drawPoints(Graphics g, Dataset d) {        //  draws data points
        int SQSIZE=1;
        double dax[], day[];
        int n;
```

* * *

```
        n = d.length();
        dax = new double[n];
        day = new double[n];
        int m;      // loop variable

        dax = d.getXdata();
        day = d.getYdata();

        for (m=0; m<n; m++)   {
            g.drawRect(ifromx(dax[m])-SQSIZE, jfromy(day[m])-SQSIZE,
                        2*SQSIZE, 2*SQSIZE);
        }
    }

    private void setXgrid(Dataset d)     {                  //  calculate x tick locations
        minx = d.xmin();
        maxx = d.xmax();
        intx = RoundUp((maxx - minx) / ngridsi);          // size of intervals in x
        xrange = intx*ngridsi;                            // total range of x
        minx = intx * Math.floor(minx/intx);      // resets convenient minimum for x
    }

    private void setXgrid(Dataset d1, Dataset d2)   {      //  calculate x tick locations
        minx = Math.min(d1.xmin(), d2.xmin());
        maxx = Math.max(d1.xmax(), d2.xmax());
        intx = RoundUp((maxx - minx) / ngridsi);          //  size of intervals in x
        xrange = intx*ngridsi;                            // total range of x
        minx = intx * Math.floor(minx/intx);      // resets convenient minimum for x
    }

    private void drawXgrid(Graphics g) {    //  put x ticks on graph
        int k;                              // loop variable for putting down grid ticks
        double xt;                          // x-value for tick
        int i;                              // x coordinate of tick in screen variables

        for (k=0; k<=ngridsi; k++) {
            xt = minx + k*intx;                     // x values for ticks
            i=ifromx(xt);
```

* * *

```
            g.drawLine(i, j0, i, j0 - ticklength);    // draws ticks
            g.drawString(Double.toString(xt), i-5, j0+15);       //  puts on numbers
        }
    }

    private void setYgrid(Dataset d)      {           //   calculate y tick locations
        miny = d.ymin();
        maxy = d.ymax();
        inty = RoundUp((maxy - miny) / ngridsj);        // size of intervals in y
        yrange = inty*ngridsj;                          //  total range of y
        miny = inty * Math.floor(miny/inty);       // resets convenient minimum for y
    }

    private void setYgrid(Dataset d1, Dataset d2)   {       //  calculate y tick locations
        miny = Math.min(d1.ymin(), d2.ymin());
        maxy = Math.max(d1.ymax(), d2.ymax());
        inty = RoundUp((maxy - miny) / ngridsj);        // size of intervals in y
        yrange = inty*ngridsj;                          // total range of y
        miny = inty * Math.floor(miny/inty);       // resets convenient minimum for y
    }

    private void drawYgrid(Graphics g) {    // put y ticks on graph
        int k;                              //  loop variable for putting down grid ticks
        double yt;                          // y-value for tick
        int j;                              // y coordinate of tick in screen variables

        for (k=0; k<=ngridsj; k++) {
            yt = miny + k*inty;                     // y values for ticks
            j=jfromy(yt);
            g.drawLine(i0, j, i0 + ticklength, j);    // draws ticks

            g.drawString(Double.toString(yt), i0-35, j+5);       // puts on numbers
        }
    }

    private void setscale()    {    // sets scale for x and y axes and connects
                                    // them with i,j coordinates
        offy = miny;
```

* * *

```
        scalefy = (height * (1 - bordt - bordb)) / yrange;
        scalefx = (width * (1 - bordl - bordr)) / xrange;
        offx = minx;
    }

    private void drawAxes(Graphics g)  {                    //  draws axes on graph
        g.drawLine(i0,j0,i1,j0);
        g.drawLine(i0,j0,i0,j1);
    }

    public void paint(Graphics g)    {                      // display graph on canvas
        drawAxes(g);

        if(ncurve == 1)  {
            setXgrid(mydata1);
            setYgrid(mydata1);
        }
        else if (ncurve == 2){
            setXgrid(mydata1, mydata2);
            setYgrid(mydata1, mydata2);
        }
        setscale();

        drawXgrid(g);
        drawYgrid(g);
        drawCurve(g, mydata1);
//      drawPoints(g, mydata1);

        if (ncurve == 2) {
            g.setColor(Color.magenta);
            drawCurve(g, mydata2);
//          drawPoints(g, mydata2);
            g.setColor(Color.black);
        }
    }

    //  method from Leigh Brookshaw's graph package, available at
    //  http://www.sci.usq.edu.au/staff/leighb/graph
```

* * *

```
    private double RoundUp ( double val )    {
    //  rounds up val to a NICE value
    //  used for figuring out where to put ticks on graphs
        int exponent;
        int i;
        exponent = (int) (Math.floor ( Util.log10(val)));
    //  loop to strip off zeros and get to the significant digits
        if (exponent < 0)  {
            for (i=exponent; i<0; i++) { val *=10.0; }
        }
        else    {
            for (i=0; i<exponent; i++)   { val /= 10.0; }
        }

        if (val > 5.0)  val = 10.0;
        else
        if (val > 2.5)  val = 5.0;
        else
        if (val > 2.0)  val = 2.5;
        else
        if (val > 1.0 ) val = 2.0;
        else            val = 1.0;

    //  loop to reconstruct original order of magnitude of val
        if (exponent < 0)  {
            for (i = exponent; i<0; i++) { val /= 10.0; }
        }
        else    {
            for (i=0; i< exponent; i++)  {val *=10.0;}
        }
        return val;
    }
}
```

Finally we present the class Util, which is a collection of some utility methods that can be used by any class. Util is an abstract class, which means that, unlike Complex, Dataset, and GraphMaker,

you cannot make any objects of the Util class. (An abstract class in Java is exactly analogous to a collection of subroutines in Pascal, C or Fortan.)

```
// Util.java
/* Declaration of Util class--abstract class of utility methods.  Its main purpose is to
calculate the maximum and minimum of a double array.  It also contains a simple-
minded method that calculates base-10 logarithms.
*/
public abstract class Util {
    public static double max(double[] data, int n)
    {          /*  calculate maximum of first n elements in array data    */
        int k;              // loop variable
        double d, mt;       // d stores the immediately needed array element
                            // mt stores a temporary variable which is the
                            // maximum data element found so far
        mt = data[0];
            // at start largest element found so far is first element in array
        for (k=0; k<=n-1; k++)
        {
            d = data[k];
            if (d > mt)     // if new element is larger than maximum found so
                            // far, then:
            {
                mt = d;     // new data replaces maximum-up-to-now.
            }
        }                   // end of loop
        return mt;          // maximum evaluated
        }                   // end max

    public static double min(double[] data, int n)   {
        /*  calculate minimum of first n elements in array data
            works just the same as max
        */
        int k;              // loop variable
        double d, mt;       // d stores the immediately needed array element
                            // mt stores a temporary variable which is the
                            // minimum data element found so far
        mt = data[0];
            // at start smallest element found so far is first element in array
```

* * *

```
    for (k=0; k<=n-1; k++){
        d = data[k];
        if (d < mt){        // if new element is smaller than minimum found so
                            // far, then:
            mt = d;     // new data replaces minimum-up-to-now.
        }
    }                       //  end of loop
    return mt;              // minimum evaluated
}                           // end min
    public static int max(int[] data, int n)       {
            /*  calculate maximum of first n elements in integer array    */
    int k;                  // loop variable
    int d, mt;              // d stores the immediately needed array element
                            // mt stores a temporary variable which is the
                            // maximum data element found so far
    mt = data[0];
        // at start largest element found so far is first element in array
    for (k=0; k<=n-1; k++){
        d = data[k];
        if (d > mt){        // if new element is larger than maximum found so
                            // far, then:
            mt = d;         // new data replaces maximum-up-to-now.
        }
    }                       // end of loop
    return mt;              // maximum evaluated
}                           // end max

public static int min(int[] data, int n)   {
    /*  calculate minimum of first n elements in array data
        works just the same as max
    */
    int k;              // loop variable
    int d, mt;          // d stores the immediately needed array element
                        // mt stores a temporary variable which is the
                        // minimum data element found so far
    mt = data[0];
        // at start smallest element found so far is first element in array
    for (k=0; k<=n-1; k++){
```

* * *

```
            d = data[k];
            if (d < mt){        // if new element is smaller than minimum found so
                                    // far, then:
                mt = d;         // new data replaces minimum-up-to-now.
            }
        }                       //  end of loop
        return mt;              // minimum evaluated
    }                           // end min

    public static double log10( double x )    {
        //  returns log base 10 of x
        return Math.log(x)/2.30258509299404568401;
    }
}
```

* * *