Chapter 2: From Maps to Chaos

Goals:

- To use graphics to reveal the range of behaviors exhibited by a simple dynamical system, the logistic map
- To be able to use Java arrays, methods, and classes.

A. Introduction. Undergraduate mechanics courses start from Newton's laws. Given the velocity and position of a particle at some instant of time, these laws enable the prediction of the velocity and position at all future times. From one point of view, these laws are the simplest description of mechanics. If you want to predict when a cannonball will hit the ground, then you need to know both its velocity when it was shot out of the cannon as well as whether the cannon was on top of a hill. However, if one looks at Newton's laws as a set of rules for predicting the velocities and positions at a later time given their values at some earlier time, then it is natural to consider systems described by different rules.

Indeed, one can imagine a system represented by a single variable x. For example, in a problem involving population growth, x might be a population or a ratio of the population to some reference value. We might give x a label corresponding to the population (x) at some particular time (t). If we call the label 'j,' then x_j would represent the state of the system at some particular time t_j. (Then x_0 would be the initial population, at time t_0, followed a bit later by a population x_1 at time t_1, and so on.) Then one has a rule that says how the state of the system at the later time depends upon that at the earlier one. Symbolically this law can be expressed as[1]

$$x_{j+1} = f(x_j) \ . \qquad (2.1)$$

---

[1]For a further exposition of this kind of time development see, for example, Leo P. Kadanoff, Roads to Chaos, Physics Today, p. 46 (December, 1983). A parallel discussion is provided in Gould and Tobochnik, second edition, Chapter 6.

* * *

Here, the nature of the dynamics is encoded in the function f(x). This type of relationship is described by saying that the function f 'maps' the population x_j at time t_j into the population x_{j+1} at time t_{j+1}.

In this chapter and the next few, we will be examining a very simple mapping (called the "logistic map"), where $x_{j+1} = f(x_j)$, with

$$f(x) = r*x*(1-x) \tag{2.2}$$

in which r is a parameter. If we continue to interpret Eq. (2.2) as a population model, then x could be the ratio of a population to some maximum possible population for a biological species at some particular time (hence x lies between zero and 1). All the biological facts about the viability of the species in question are hidden in the parameter r in Eq. (2.2). The value of r depends upon the species and its environment. To interpret this parameter, notice that if the population is small, $x << 1$, the population grows by a factor of r during one time period. Hence r is simply the natural growth factor for the population during one time period. However, as x approaches it maximum value of 1, the quadratic term becomes important and forces the population to decrease (presumably due to some form of natural competition).

We shall be examining the sequence of values $\{x_1, x_2, x_3, ...\}$ that are generated by the model defined by Eqs. (2.1) and (2.2), with a particular focus on how the sequence of population values depends upon r (we keep r fixed within any one sequence of populations). We shall see that the qualitative properties of the sequence will be very different in different ranges of r. In particular, for some ranges we shall see quite orderly behavior and for others very chaotic behavior.

This interesting behavior leads us to ask many questions, including: What is it about the logistic map that leads to its chaotic behavior? Do systems described by other mappings (in particular, those described by Newton's laws) have similar properties? What do we mean when we say that the behavior of two different systems is similar? We will answer these questions over the next several chapters.

* * *

To get started, we will learn just a little more Java and then write a simple program to study the qualitative behavior of the logistic map.

B. Methods. By putting together what you have learned last week with the material in the text you probably know enough to do some rough-and-ready program development in Java. We want to mention one more concept and then get started doing some numerical work.

Methods are sets of instructions which are combined into a single command for use in a program. We have already worked with the supplied methods Math.sin and Math.cos. They are used so that the statement

```
x = Math.sin (y);
```

has the result, when x and y are reals, of assigning to x the value of sine of y (y is in radians). One can equally well define one's own method within a program. Below, we have written a program with a method that implements the map we just described; it calculates x_1 as a function of x_0. Please write this program for yourself, and **Run** it.

```
// Iterate.java
import java.awt.Graphics;
import java.applet.Applet;
public class Iterate extends Applet {
    double x, r;
    int applet_width, applet_height;

    public void init() {
        applet_width=300;
        applet_height=600;
        setSize(applet_width, applet_height);      // make room for many iterates
        r = 0.5;                                   // (though this program only
        x = 0.7;                                   //   iterates once)
    }

    public void paint(Graphics g)    {
        int xpos, ypos;        // coordinates of output on applet
```

```
        xpos = 10;
        ypos = 20;
        g.drawString("x = " + x, xpos, ypos);
        x = f(x);
        g.drawString("f(x) = " + x, xpos+70, ypos);
    }

    public double f( double y )   {   //  we declare a real method of a real variable
        return  r * y * (1 - y);          //  this line defines the value of f
    }                                     //  en of the method
}   //  end of applet
```

Program 2.1. We define a method within a program. The method depends upon a dummy (or formal) variable y. When we use the method we must replace y by a number, or by a constant, or by a variable. In any case, this replacement must take on a double value (or an integer value that will automatically be converted to a double value).

To see the method in action, we now want to iterate it many times. One can do this by inserting a for(){} (or a while(){}) loop to iterate the function and display the current value of x over and over (being sure to increment ypos and to define the total number of iterations). You will do this in the first Required Project.

Producing the Iterations. In Required Project I, you will modify the program “Iterate” to produce a large number of iterations of the logistic map, and experiment a little with different values of r and different initial values of x.

The applet Iterate is cumbersome to use. It must be edited and re-run each time one wants to change the initial value of x, or the parameter r. It is also difficult to visualize the results, because the output is in the form of a list of numbers. Graphing the results would be much better.

Here is a program we wrote to improve this situation. You can get a copy from the folder “Programs/Chapter_2” and run it for yourself. Please look at it carefully to see what it does. (You might recognize several features that it has in common with applet “SecondOrbit”

* * *

from chapter one.) It is a rather primitive program in that it only has a bare-bones minimum of graphics and analysis. To make it more elegant you might, for example, want to fill in some axes and x-values. Then you might be able to see better what is going on. However do be careful not to do so much graphics that you spend all your time making your result pretty and none really understanding it. You should do just enough so that you can see what is happening with the minimum of effort.

This program generates a primitive graph of the iterates using the Graphics method

```
drawRect(i,j,0,0);
```

which draws the point (i,j). It also converts the (n,x_j) coordinates to int screen coordinates (i,j), keeping in mind that the screen coordinate j=0 is at the top of the applet. To accomplish this it uses the methods ifromn and jfromx (defined within the program). The conversion between the double x_j and the int screen coordinate j done by jfromx has a slight additional complication compared to that in ifromn because the Java Math method round (which rounds to the closest integer) converts either float to int or double to long, but not double to int.

```
//    FirstMap.java
import java.awt.*;
import java.awt.event.*;
import java.lang.*;
import java.applet.Applet;

public class FirstMap extends Applet           //  shows results of iterated logistic map
            implements ActionListener {        //  will listen for mouse clicks
    final int APPLETHEIGHT=525;
    final int APPLETWIDTH=650;                 //  applet dimensions
    final int RECTANGLEHEIGHT=400;
    final int RECTANGLEWIDTH=600;              //  dimensions of plotting rectangle
    final int IOFFSET=25, JOFFSET=100;         //  locates plotting rectangle on applet
    int n;                        //  n is a loop index
    double[] da;                  //  declare da as an array to store a list of x-values
    double r, x;                  //  r is the parameter in the mapping
                                  //  x is the variable in the mapping
```

* * *

```
Label promptr;           //  prompt user to input r
Label promptx;           //  prompt user to input x
TextField inputr;        //  user types value of r in this TextField
TextField inputx;        //  user types value of x in this TextField
Button thebutton;        //  button user presses to start iterations

public void init ()    {
    setSize(APPLETWIDTH, APPLETHEIGHT);   //  reset applet size
    da = new double[200];                 //  allocate the array da
    promptr = new Label("value of r:");
    inputr = new TextField ( 10 );
    promptx = new Label("Initial value of x:");
    inputx = new TextField ( 10 );
    thebutton = new Button("Click to start");   //  instantiate Button object
    thebutton.addActionListener (this ); // tell the button to listen for mouse clicks
    add( promptr );                 //  put promptr on applet
    add( inputr );                  //  put TextField on applet
    add( promptx );                 //  put promptx on applet
    add( inputx );                  //  put TextField on applet
    add(thebutton);                 //  put button on applet
}          //  end of init

public void actionPerformed( ActionEvent e )   {  //  handles user inputs
    String s;
    s = inputr.getText();     //  read string in TextField inputr
    r = new Double(s).doubleValue();     //  convert string to double variable r

    if( r <=0 || r > 4)  {          //  check whether r is an allowed value
        showStatus(
            "Please choose a value of r in the range (0,4]");
        return;
    }
        s = inputx.getText();                   //  read string in TextField inputx
        x = new Double(s).doubleValue();     //  convert string to double variable x

        if( x <=0 || x > 1)  {                 //  check whether x is an allowed value
            showStatus(
                    "Please choose a value of x in the range (0,1]");
```

* * *

```
            return;
        }
        showStatus("r = "+ r );        // put value of r at bottom of applet
        repaint();
    }                        //  end of actionPerformed method

    //  override Component class update
    //  do not clear background, only call paint
    public void update ( Graphics g ){
        paint( g );
    }

    public void paint ( Graphics g )  {
        g.clearRect(IOFFSET, JOFFSET,
            RECTANGLEWIDTH,RECTANGLEHEIGHT);   //  clears rectangle interior
        g.drawRect(IOFFSET, JOFFSET,
            RECTANGLEWIDTH,RECTANGLEHEIGHT);   //  draws rectangle
        for (n = 0; n<=199; n++)   {       //  main loop
            x = f(x);                      //  find new x
            da[n ] = x;                //  store x in array
            g.drawRect(ifromn(n), jfromx(x), 0, 0);
                //  plots points from run
                //  higher iterations mean larger horizontal coordinate
                //  x-value is vertical coordinate
        }
    }                              //  main loop

    double f (double x )   {
        return r * x * (1 - x);
    }

    double fn (double x, int n )   {       //  iterates the map n-times
                                           //  not used in this program
        int k ;                            //  variable for do loop
        double xp;                         //  temporary storage for x-value

        xp = x;                            //  set initial argument of the function
        for (k  = 1; k<n; k++)   {         //  doing n iterations of the function, f
```

* * *

```
            xp = f(xp);
        }                                   //  end the iterations
        return xp;                          //  set the result
    }                                       //  end fn

    int ifromn(int n) {    //   converts n to screen coordinate i
        //   have 200 points in array and RECTANGLEWIDTH
        //    horizontal pixels in rectangle, so:
        return IOFFSET+ Math.round( (float) (RECTANGLEWIDTH/200.)*n);
    }

    int jfromx(double x) {        //  converts x to screen coordinate j
        float temp;//  temporary variable for intermediate result
        temp = (float) (RECTANGLEHEIGHT*(1- x));
                        // we convert this double result to float to use Math.round
        return  JOFFSET + Math.round(temp);
            //  Math.round is a Java-supplied method that rounds floating-point
            //   numbers.
    }
}
```

Program 2.2. A program for seeing the results of many iterations of a map. Note the way fn is defined in terms of the method f. Notice also how k and xp are declared within fn. All variables defined within a method are local to it in the sense that they are totally invisible outside of the method. Thus k and xp 'disappear' outside of fn.

C. Arrays. One feature of this program that is somewhat new to us is the use of arrays. An array is a structure which consists of many elements stored in a standard pattern. In our program the array da contains 200 double numbers. The first element in an array always has subscript zero, so the array runs from da[0] to da[199] (the 7th element is da[6]). The pattern of an array can be a vector of length n (like da), or a matrix of n rows and m columns, or a matrix with yet more indices. The array is set up in two steps. First, it is declared with the statement

```
double da[];
```

* * *

and then, it is allocated with:

```
da = new double[200];
```

You can also, if you wish, combine these two steps into a single statement:

```
double da[] = new double [ 200 ];
```

Arrays may be declared to contain things besides numbers. For example, an array of type String can store a set of character strings.

In Required Project 1, you will construct a program for seeing what happens for different values of r. The goal is to characterize what happens to x after many iterations, for several different values of r. As you do this project, you may want to copy some of the code that we have written into your program[2], and then modify it appropriately so that it can be used to find out what the large-iteration qualitative behavior is for different values of r. You will find that as r is varied, the behavior of the iterates changes a lot.

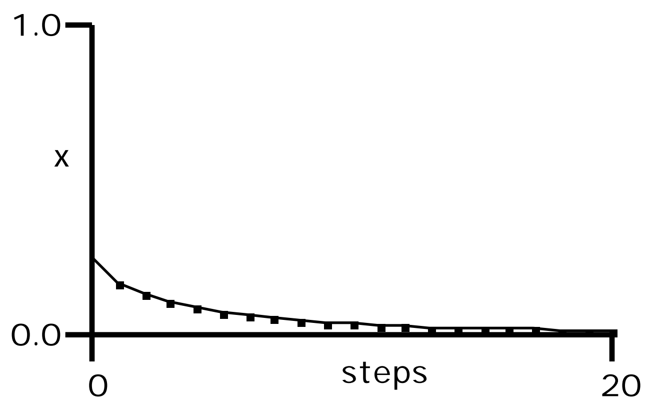
For small values of r the motion is clearly orderly in that, after a while, it settles down to a well-defined pattern. For example, for $0 < r < 1$, the pattern is obvious: Eqs. (2.1) and (2.2) imply that $x_{j+1} < x_j$ so that, as j goes to infinity, x_j goes to zero. In this range of r, the orderly long-term behavior is one in which the population, x_j, simply goes to zero after a long time. However, for larger values of r the motion can seem quite irregular and non-repetitive. This latter kind of motion is termed chaos[3]. We want to take a better look at what happens and then study some portions of the motion in some detail.

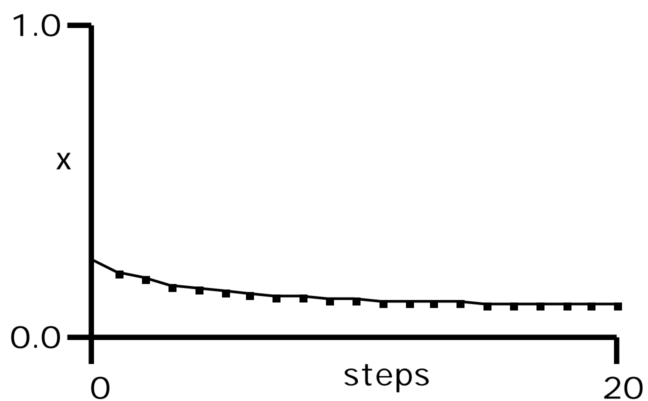
---

[2]Recall that if you highlight something and type in ⌘-C, the computer copies what you have highlighted onto the clipboard. If you put down the cursor and type ⌘-V the material on the clipboard is entered at the point of the cursor.

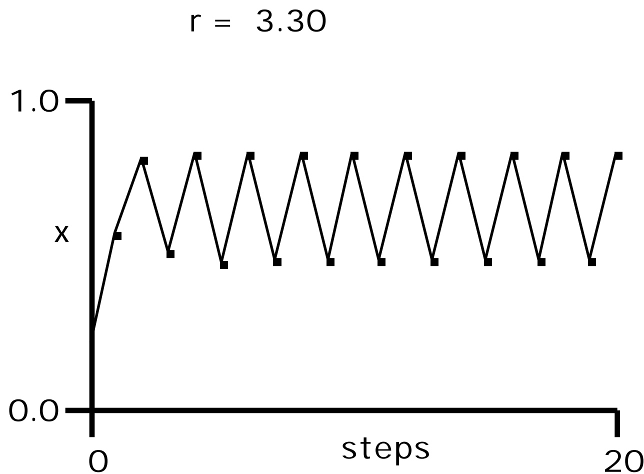
[3]For a non-technical discussion of chaos see the Gleick book mentioned in the bibliography.

* * *


r = 0.90
1.0
x
0.0
0
steps
20


r = 1.10
1.0
x
0.0
0
steps
20

* * *



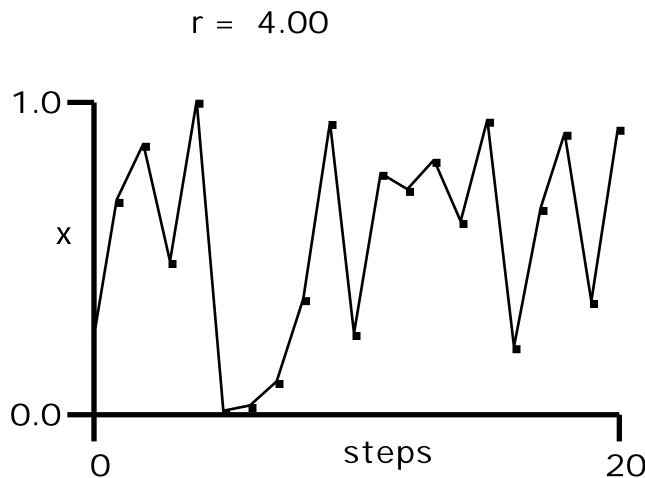


Figure 2.1 x_j versus j for several values of r

D. Graphing the long-time behavior. Figure 2.2 is a graph that shows which values of x will appear, in the long run, for each value of r.[4] We ask you to make a similar plot (hopefully nicer-looking!) in Required Project 1. This plot is done by making a loop that runs through the r-values step by step. For each r-value, one goes through the following steps:

i. one chooses a starting value of x

---

[4]Figure 6.2 of Gould and Tobochnik (2nd edition) is a better version of this same drawing.

ii. then iterates the map a certain number of times to eliminate the unwanted transient values of x

iii. then iterates and plots the next bunch of x-values.

In this way one generates a picture which can show what is going on in the mapping as a function of r.

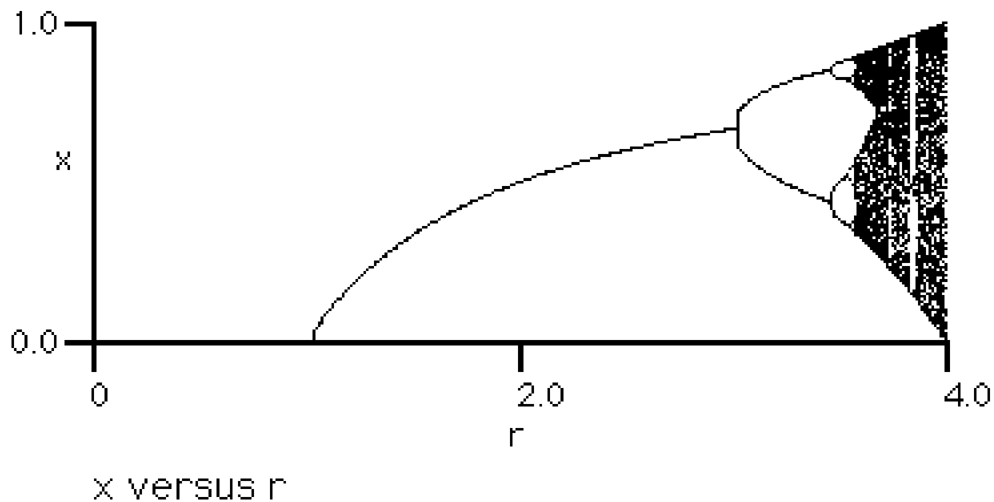


Figure 2.2 . A plot showing how the long-run-possible values of x can depend upon r for the map x   r*x*(1-x).

E. Making Graphs . By now it may be clear that it would be extremely handy if you had available some way just to make a little plot of some function of f(x) versus x. Four ways to do this are to:

1. Do the plots using a standard package such as CricketGraph, KaleidaGraph, or MatLab.

2. Use the little programs for doing rough plots which we have written.

3. Use fancier programs for doing nicer plots which you can get from the World Wide Web (if you decide to try this route, you might want to look at http://www.sci.usq.edu.au/staff/leighb/graph).

4. Write your own programs.

You might think that the simplest thing is to use the commercial package. However, this approach is far from trivial. We discuss the

* * *

necessary steps and why they are complicated in Appendix A. Options 2 and 3 are very similar in spirit. The advantage of option 3 is that you can do fancier plots, and the advantage of option 2 is that we might be better able to help you out if you have a problem. As for option 4, we expect that as the course progresses you will be modifying the programs that we have written to that they do what you want.

First we show how to plot a graph using a Java class that we have written. The applet "LogisticPlot," shown below, calculates and plots the logistic map f(x) = r*x*(1-x), as a function of x. This applet is in the project Graph.mcp, in the folder Graph.

```
// LogisticPlot.java
import java.awt.*;
import java.applet.Applet;

public class LogisticPlot extends Applet {
// demonstrates using the GraphMaker class to make a graph

    final int APPLET_HEIGHT=400;     // constants for size of applet and of graph
    final int APPLET_WIDTH=600;      // final variables can't be changed
    final int graph_height=300;
    final int graph_width=500;

    double[] x;             // x is the set of x-values
    double[] y;             // y is the set of values f(x)
    int k ;                 // a loop variable
    double r;               // the map's control parameter
    GraphMaker gm;          // the GraphMaker object that makes the plot
    Dataset thedata;        // Dataset object to hold the (x,y) pairs

    public void init() {
        setSize(APPLET_WIDTH, APPLET_HEIGHT);        // resize applet so that there
                                                     // is room for the graph
        gm = new GraphMaker(graph_width, graph_height);
                                                     // set up our GraphMaker
        x = new double[201];
        y = new double[201];
```

* * *

```
        r = 3.5;
        for (k=0; k<=200; k++){                    //  loop to calculate (x,y) pairs
            x[k] = k/200.;                         //  x goes from 0 to one.
            y[k] = r*x[k]*(1-x[k]);
        }

        thedata = new Dataset(x, y, x.length);     //  puts (x,y) pairs into a Dataset
        gm.setData(thedata);                       //  gives Dataset to GraphMaker
        add(gm);                          //  puts GraphMaker on applet--plots graph
    }
}
```

Program 2.3. This applet plots f(x) = rx(1-x) versus x for x between 0.0 and 1.0.

The guts of the program are in the lines

```
gm.setData(thedata);
add(gm);
```

These lines cause the GraphMaker gm to put a graph of the Dataset thedata on the applet. Here is the information about the classes Dataset and GraphMaker that is used by "LogisticPlot.java:"

```
Class Dataset
public class Dataset
     extends java.lang.Object

This class holds the (x,y) pairs of data.

// Constructor (called when object is instantiated)
public Dataset(double[] dx, double [] dy, int n)
Parameters:
     dx -- array of x values
     dy -- array of y values
     n -- number of (x,y) points
```

```
Class GraphMaker
public class GraphMaker
     extends java.awt.Canvas
```

* * *

```
This class makes a graph of either one or two
Datasets.  The paint method of this class handles
all the drawing operations of the graph.

//  Constructor
public GraphMaker(int w, int h)
Parameters:
     w -- graph width (including margin) in pixels
     h -- graph height (including margin) in pixels

//  Methods
public void setData(Dataset d1)
     parameter:
          d1 -- Dataset to be acquired
     reads in single Dataset d1
public void setData(Dataset d1, Dataset d2)
     parameter:
          d1 -- first Dataset to be acquired
          d2 -- second Dataset to be acquired
     reads in two Datasets d1, d2
public void paint(Graphics g)
     paints the graph.
```

(The method paint is called automatically when the Sun-supplied method add(gm) puts the GraphMaker on the applet.) As you can see, GraphMaker also has a method not used in "LogisticPlot.java." If you call the method setData with two Dataset arguments instead of one, then GraphMaker will plot out both datasets on the same scale.

Notice that neither Dataset nor GraphMaker have been defined anywhere in "LogisticPlot.java." These apparently undefined classes are, of course, not really undefined. They simply appear in other files, called "Dataset.java" and "GraphMaker.java." We have put these files in the same folder as "LogisticPlot.java." They can also be in another folder; in any case they need to be **Add**ed to the CodeWarrior project if Java is to find them.

Before we tell you more about the classes Dataset and GraphMaker, we would like you to run the CodeWarrior project "graph.mcp," which is in the "graph" subfolder of the "Chapter_2" folder. Notice

that the project contains six files. The html file “LogisticPlot.html” and classes.zip are both in familar roles, as is our applet “LogisticPlot.java.” As we saw, LogisticPlot uses the classes Dataset and GraphMaker, which are defined in “Dataset.java” and “GraphMaker.java.” Both Dataset and GraphMaker in turn call upon methods defined in the class Util, which is defined in “Util.java.”

To use the classes GraphMaker, Dataset, and Util, we do not have to know very much. First, we should know that they contain a set of variables and methods which we can use. Second, we should know the significance and types of the variables, and the syntax for using the methods. The API (Application Programming Interface) information like that given above contains exactly what we need to know to use these classes.

The Appendix of this chapter has printouts of GraphMaker, Dataset, and Util. GraphMaker in particular is long, but they are all rather straightforward combinations of simple pieces.

Appendix: Files.

Here we discuss how one would go about using MatLab or some other package to draw a graph using numbers calculated in a Java program.

These packages can either read from data typed in by hand or from a file on the Mac's disk. Certainly, we do not wish to recopy the data by hand. To avoid the awful job of recopying, one must put data from the Java program onto a disk in a form suitable for reading by another program. This means learning how to put the data into files, which are covered in Chapter 8 of Beginning Java and in Chapter 10 of Exploring Java, second edition.

First, some preliminaries. So far we have done all our output by drawing on applets using the paint method. But it is also possible to print text output on the window that MetroWerks Java calls "Java Output" (Netscape calls it the "Java Console," when using Netscape you can view it by choosing "Show Java Console" in the Netscape **Options** menu). The commands used to print out text on the Java Console are nearly identical to those used to write data into files. In any case, it is worth knowing them because the Java Console is a

scrolling window. This means that you can print out lots and lots of numbers on it without worrying about running off the end, which is sometimes very convenient.

The two main commands you need to know are System.out.print and System.out.println. The following applet, which calculates the squares of the integers 1 through 10, demonstrates how they work.

```
//  Squares.java
//  prints out the squares of the integers 1 through 10 onto the Java Console
import java.applet.Applet;
public class Squares extends Applet  {
    int i;
    public void init() {
        for(i=1; i<= 10;  i++)   {
            System.out.print(i);                   //  print the value of i
            System.out.print("    ");              //  print space between columns
            System.out.println(i*i);               //  print i*i, then new line
        }   //  end of for loop
    }   // end of init
}   //  end of applet
```

Note that print and println know how to print out numbers as well as strings (in contrast to the Graphics method drawString, which only puts strings on an applet).

Next we consider how to get numbers actually into a file. Probably the simplest way is first to print the numbers out onto the Java Console. Then you can choose **Select All** from the **Edit** menu of the Java Console window (or type ⌘-A), **Copy** (⌘-C) and **Paste** (⌘-V) all the output into a text processor like SimpleText or Microsoft Word, and finally use the text processing program to save the file. If this solution is acceptable to you, you can skip the rest of this appendix.

You can avoid cutting-and-pasting by having your Java program open a file and write directly into it, but there are complications. The problems arise because applets are designed to run using a browser over the World Wide Web. Applets run on the machine of the person who is browsing. Therefore, your applets will be run by total strangers who have no idea whether you are a vicious computer

hacker and who don't want to give you any opportunity to damage their machine. To minimize the chance of a problem, the browsers give Java applets very limited privileges. Java applets run using Netscape are not allowed to write into files on the host machine. Applets can send the data back to the machine where the applet came from over the Internet, but this is slow. If you're interested in learning how to do this, you can read chapter 16 of Beginning Java or chapter 11 of Exploring Java.

However, we will often write Java programs that run on our own machine (for instance, when we use CodeWarrior). Now CodeWarrior imposes the same security restrictions on applets that Netscape does, so that does not solve the problem. However, one can write a Java application (as opposed to an applet) that is able to read from and write into files. Java applications have the same privileges as computer programs written in other languages such as C, Fortran, or Pascal; in particular, they can read from and write into files. Java applications have a slightly different structure than applets.

In this course we concentrate on using the graphical capabilities of applets as opposed to the input-output capabilities of applications. But just in case you really want to use files, here are two applications, one which writes into and the other that reads from files.

The first application, “WriteSquares,” writes output into a file. First, a File object is created (this amounts to telling the computer the name of the file to be used) and then it is assigned to a FileOutputStream. Then, the file is designated a PrintStream, which tells Java that print and println will be used. Then print and println are called to put the numbers into the file. Finally, we call close so that Java knows that we don't need access to the file any longer.

```
// Writesquares.java
// prints out the squares of the integers 1 through 10 into the file "squares.dat"

import java.io.*;
public class WriteSquares {

    public static void main(String args[]) throws IOException {
        int i;
```

* * *

```
        File out = new File("squares.dat");
        FileOutputStream fout =  new FileOutputStream( out );
        PrintStream pout = new PrintStream(fout);
        for(i=1; i<= 10;  i++)   {
            pout.print(i);                     //  print the value of i in orbit.dat
            pout.print("    ");                //  print space between columns
            pout.println(i*i);                 //  print i*i, then new line
        }                                      //  end of for loop
        fout.close();
    }   //  end of main

}       //  end of class WriteSquares
}
```

Program 2.4 . This application writes the integers from 1 to 10 and their squares into the file squares.dat.

The second application “ReadPairs” reads up to five columns of integers from the file "squares.dat" and prints the first two columns out onto the Java Console.

```
//  ReadPairs.java
//  reads up to five columns of integers from file "squares.dat"
//   and prints first two columns out on Java console
import java.io.*;
import java.util.*;

public class ReadPairs    {
    public static void main(String[] args) throws IOException     {
        int count;
        int i[];
        String line;
        File inputFile = new File("squares.dat");
        FileInputStream fis = new FileInputStream(inputFile);
        DataInputStream input = new DataInputStream( fis );
        count = 0;
        i = new int[5];
        try{
            while ((line = input.readLine()) != null)       {
```

* * *

```
                count=0;
                StringTokenizer linetoken = new StringTokenizer( line );
                while (linetoken.hasMoreTokens() ){
                    i[count] = Integer.parseInt(linetoken.nextToken());
                    count++;
                }
                System.out.print(i[0]);
                System.out.print("    ");
                System.out.println(i[1]);
            }
        }
        catch (EOFException eof ) { }
        input.close();
    }                   //  end of main
}                       //  end of application ReadPairs
```

Program 2.5. This application reads up to five columns of integers from the file "squares.dat" and prints the first two columns out onto the Java Console.

So now you have two ways to get your data into a file. Once you have the file, you will need to take it and read it into the package that actually draws the graph. The procedure for this depends on which package you decide to use, so we refer you to the documentation or the help for the package.

* * *