Chapter 7:

Displaying Solutions to Differential Equations

Goals:

- To display solutions to dynamical equations using animations and phase space plots.
- To calculate how volumes in phase space change in time.
- To understand Java concepts required to make attractive animations, such as double-buffering and threading.

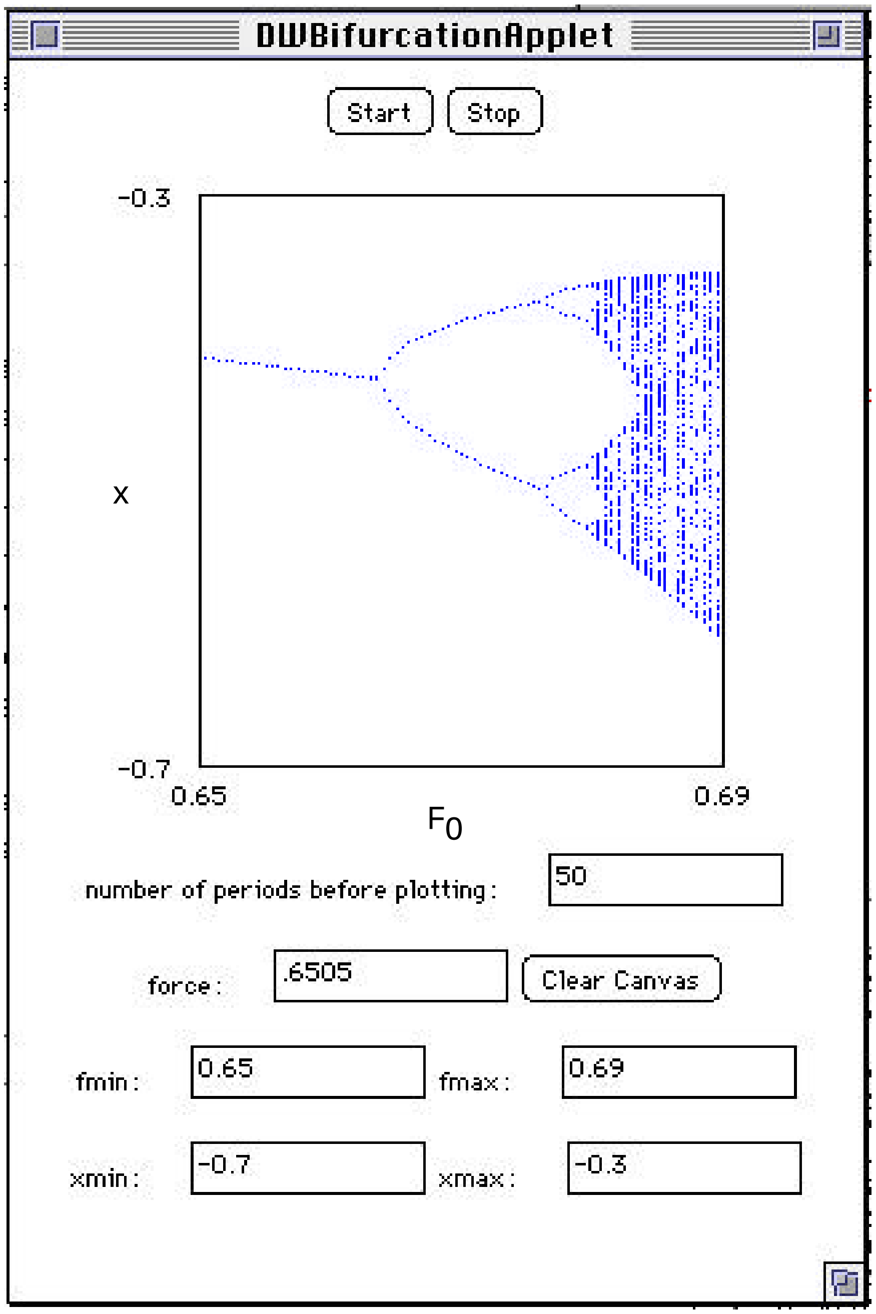
In this chapter we discuss methods of presenting solutions to our equations of motion. Notice that we have divided the problem of understanding our differential equations into two parts: first, we saw how to solve the equations, and now we are seeing how to display that solution. Although this endeavor is small enough that this strategy may not seem necessary, for larger programs it is essential. Always, when faced with a large problem, first divide it into smaller problems, which you can then attack one by one. All programmers do this when tackling a non-trivial problem.

A. Animations. First we will do some animations. To whet your appetite, we'd like you to play around with some applets that use the DoubleWell class to integrate the equations of motion for a particle in a double-well potential (6.28). These applets are in "Programs:Chapter_7:DoubleWell Animations." The first, in the project "DWAnimation.mcp," displays the position x of the particle at a sequence of equally spaced times. The vertical coordinate is $x^4 - x^2 + xF_0\cos(\omega t)$, the potential energy at position x and time t. The parameter values m = γ = 1, k_1 = 2, k_2 = 4, and $\omega = 10\pi$ are fixed, while you can change the amplitude of the drive F_0. Does the motion appear to be periodic when $F_0 = 0.2$? How about $F_0 = 0.6$? $F_0 = 1$?

The animation, along with the x versus t plots that you made in the last chapter, may lead you to suspect that the motion of the particle in the quartic potential is not periodic for all F_0. A second applet, in the project "DWBifurcation.mcp," provides more information about this question. It plots, versus the drive amplitude F_0, the

***

values of x exactly once each period, at times $2\pi n/\omega$, where n is an integer. To change F_0, edit its textField and then press the start button. Figure 7.1 shows the result when for each value of F_0, we chose a starting value of x, evolved the equations of motion for a while so that the transient decayed, and then plotted 50 values of x.



number of periods before plotting: 50

force: .6505

fmin: 0.65 fmax: 0.69

xmin: -0.7 xmax: -0.3

Figure 7.1. Bifurcation diagram for motion of a particle in a double-well potential, described by equations (6.28) and (6.29).

* * *

This type of plot, where we look at x(t) only at periodic points in time, is called a stroboscopic map. If x happens to vary periodically with a period exactly equal to the period between the time points, then we will see the same value of x every time we look (x is constant in the stroboscopic map). For the double-well problem, we suspect that x(t) might vary with the same period as the forcing, and so we use a stroboscopic map with exactly that period. When the forcing magnitude is less than about 0.67, we see in Figure 7.1 that x is constant under the stroboscopic map, indicating that the solution is periodic. For slightly larger values of F_O, x oscillates between two values. That tells us that after 2 applications of the stroboscopic map, x has returned to its original value. The motion is still periodic, but the period has doubled. Figure 7.1 suggests that the motion in the double well might have the same sort of period-doubling bifurcation sequence that we have seen in the logistic map.

Menu Project. Period-doubling bifurcation sequence for motion in a double-well potential. Compare Figure 7.1 to Figure 2.2, the possible values of x for different values of r in the logistic map. Determine whether the double-well system has a period-doubling bifurcation sequence, and if so, whether it is similar to that of the logistic map. (You will need to define what you mean by the word "similar" here.) How does your answer depend on the choice of parameters of the double-well system?

Making Animations Look Better. One of the reasons Java has excited a lot of interest is its capabilities for creating animations for the web. In principle, an animation is just a series of images displayed in quick enough succession that the viewer gets an illusion of motion. However, there are techniques for making better-looking animations that you may want to know about.

To see that a certain amount of sophistication is necessary to create an animation that looks even remotely decent, you can look at the very naive animation applet that we have written for the particle in the double well that is in the project SimpleAnimation.mcp, in the folder sequence "Programs:Chapter_7: Animation_Learning_Examples:SimpleAnimation." This applet sets up a loop which at every step increments time by deltat, calculates x(t), and then calls repaint() to display the result.

* * *

```
/*
    Totally naive animation applet
*/

import java.awt.*;
import java.applet.Applet;

public class SimpleAnimationApplet extends Applet
{
    private final static int BALLRADIUS=5;
    DoubleWell dw1;                                  // our DoubleWell dynamical system
    VariableSet vars;                                // variables for DoubleWell
    double time, deltat;
    boolean firstpaint = true;

    int imin, iwidth, jmin, jmax, jheight;                    //  coordinates on applet
    double xmin, xmax, xwidth, ymin, ymax, yheight;           //  limits on x and y values
    int ipts[] = new int[201];                       //  arrays used for putting potential
    int jpts[] = new int[201];                       //  on applet

    Point position;

    public void init() {

        dw1 = new DoubleWell();
        deltat = dw1.getdt();                                 //  get time step
        vars = dw1.getvars();                                 //  get VariableSet

        this.setSize(250,250);                                //  set display size

        setBackground( Color.white );                         //  set white background

        for(int i = 1; i<=100; i++) {                         //  loop; at each step move
            for (int i2 = 1; i2 <=20; i2++) {                 //  particle and then repaint
                repaint();
            }
        System.out.println("i= "+i);
        }
        System.out.println("animation over");
    }                                                //  end of init

    int xtoi(double x){    //  converts x to screen coord i
        return Math.round( (float) (imin + iwidth*(x-xmin)/xwidth) );
    }
```

***

```
int ytoj(double y){    //  converts y to screen coord j
    return Math.round( (float) (jmax - jheight*(y-ymin)/yheight) );
}

Point vartoPoint(VariableSet v) {    //  converts variableset first to x and y,
    double x = v.getx()[0];                //  and then to screen coordinates
    double t = v.gettime();
    return new Point( xtoi(x), ytoj( dw1.potential(x, t) ) );
}

    void draw(Graphics g, Point p)   {
    g.fillOval(p.x-BALLRADIUS ,p.y-BALLRADIUS,
        2*BALLRADIUS,2*BALLRADIUS);                          //  draw ball
}

void putpotential (Graphics g)    {        //  Put potential on canvas
        g.setColor(Color.black);

        double xo;
        ipts[0] = xtoi(xmin);
        jpts[0] = ytoj (dw1.potential(xmin, dw1.gettime() ) );
        for( int i = 1; i<=200; i++){
            xo = xmin + i*xwidth/200;
            ipts[i] = xtoi( xo );
            jpts[i] = ytoj( dw1.potential(xo,dw1.gettime() ) );
            g.drawLine(ipts[i-1], jpts[i-1], ipts[i], jpts[i]);
        }
    }

public void paint( Graphics g ) {

    if(firstpaint == true)     {
        // set xmin, xmax, ymin, ymax:
        xmin = -1.2*dw1.getA();
        xmax =  1.2*dw1.getA();
        xwidth = xmax-xmin;

        ymin = dw1.minpotential(xmin, xmax) + xmin*dw1.getforce();
        ymin = ymin -0.05*Math.abs(ymin);
        ymax = dw1.maxpotential(xmin, xmax) + xmax*dw1.getforce();
        ymax = ymax + 0.05*Math.abs(ymax);
        yheight = ymax-ymin;
```

***

```
            // set imin, iwidth, jmin, jheight:
            imin = this.size().width/20;
            iwidth = this.size().width - 2*imin;
            jmax = size().height - size().height/20;
            jheight = size().height - 2*(size().height/20);

            //  set initial value of position:
            position = vartoPoint(vars);
            firstpaint = false;
        }

        //  Get new position
        time += deltat;
        vars = dw1.nextvars();
        dw1.setvars(vars);
        position = vartoPoint(vars);
        position.move(position.x, position.y-BALLRADIUS-1);

        g.setColor(Color.black);
        putpotential(g);
        g.setColor(Color.blue);
        draw(g,position);
    }                                               //  end of paint

}                                                   //  end of applet
```

Program 7.1 SimpleAnimationApplet.java, an awful-looking animation of the motion of a particle in a double-well potential.

Please **Run** SimpleAnimationApplet and see how terrible it looks.

The two main reasons why the SimpleAnimationApplet looks bad are: (1) the screen updates are irregular, so the motion appears incredibly jerky, and (2) the screen is blank most of the time. These problems have different origins, and they both need to be fixed for the animation to appear acceptable.

The first step towards fixing the timing problem is to pause for a short time after each call to repaint. The sequence

```
    try {   Thread.sleep(5);  }
    catch(InterruptedException e)    {} ;
```

pauses execution for 5 milliseconds.

* * *

Exercise 7.1. Modify SimpleAnimationApplet to pause after each call to repaint, and assess the resulting animation.

Pausing the animation helps the timing, but the animation still flickers badly. This happens because of the default behavior of the method repaint.

Repaint works by scheduling a call to the method update, which first clears the drawing region and then calls paint.[1] We will override update's default behavior and instead have it implement a technique called double-buffering, in which first the new image is painted onto an offscreen buffer and then the buffer contents are displayed on the applet. The screen doesn't go blank between frames, and the animation smoothes out.

Creating the offscreen graphics context is done by first creating a new Image object, and then calling the method getGraphics to create the necessary Graphics object. This Image object contains the re-drawn screen, but holds it in electronic limbo (the buffer) until we are ready for it. One must declare these objects:

```
Image offImage;
Graphics offGraphics;
. . . .
```

Here is the update method:

```
public void update( Graphics g ) {
    // Create offscreen graphics context, if none exists
    if (offGraphics == null) {
        offImage = createImage( size().width, size().height );
    }
    offGraphics = offImage.getGraphics();
    paint(offGraphics);                  // draw into offGraphics buffer
    g.drawImage(offImage, 0, 0, this);   // draw offGraphics buffer onto applet
}
```

The update method no longer clears the display, so this must be done in the paint method before drawing the new stuff via:

```
    g.setColor(getBackground());
    g.fillRect(0, 0, this.size().width, this.size().height);
```

[1]Multiple calls to update can get consolidated into a single screen update. That is why pausing execution smoothed out our animation—it caused each update request to be handled separately.

* * *

which fills in a rectangle the size of the drawing region with the background color.

Exercise 7.2. Add double-buffering to SimpleAnimationApplet and see how much better the resulting animation looks.

The animation should now look OK, so you might think that all problems are solved. However, this applet has the undesirable feature that it doesn't stop when the applet window is closed (or, in a browser, when the viewer leaves the page that the applet is on). To see that this is so, close the Applet window while the animation is running and watch the Java console window. The iterations continue (indicated by the scrolling data on the screen) even though the Applet window is gone.

The DWAnimationApplet stops automatically whenever it is not visible. Also, it is controllable by the viewer--as you saw, it can be stopped, the parameters changed, and then restarted. All this is possible because the animation has been put into a separate thread of execution. Each time the animation thread pauses, the applet checks to see if any buttons have been pressed or if the display is not visible, and adjusts itself accordingly.

A thread is a separate flow of control within a program. Here we will tell you a bit about them (hopefully enough so that you understand how to create an animation thread), and refer you for more detailed discussions in Beginning Java, chapter 10, Exploring Java, chapter 8, and to the discussion on Sun web site, at http://java.sun.com/docs/books/tutorial/essential/threads/index.html. We will illustrate the process using the applet DBThreadedApplet, which is in the project “DBThreadedAnimation.mcp” in the folder "Animation Learning Examples."

Running a thread involves creating an object of the class Thread, which targets another object that has a method called run. The Thread object starts, stops, and resumes the execution thread; while it is going, it calls the run method of the target object. In our case, the target object is DBThreadedApplet.

Now Java requires any object that wants to serve as a target of a Thread to declare that it has an appropriate run method. One way to do this is to make the target object a subclass of Thread, via:

```
class Animation extends Thread
```

***

However, DBThreadedApplet is already a subclass of Applet, and Java does not allow objects to inherit from more than one class. Instead we have DBThreadedApplet implement an interface called Runnable. An interface is a list of methods that define some set of behavior for an object; implementing the Runnable interface means that the object is guaranteed to have a run method. So, we declare our target object DBThreadedApplet via:

```
public class DBThreadedApplet extends Applet implements Runnable
```

We must declare a Thread object in DBThreadedApplet via:

```
Thread animatorthread;
```

The creation of the thread is done inside the applet's start method, which is called when the applet is first displayed and also each time the applet is revisited in a Web page. The target object for the animatorThread is DBThreadedApplet itself, referred to as this. This start method creates and starts the animatorThread.

```
public void start()    {
    if (frozen)    {
        //Do nothing.  The user has stopped the motion.
    }
    else    {
        if (animatorThread == null ){
            animatorThread = new Thread (this);
        }
        animatorThread.start ();
    }
}
```

The animation is stopped and the offscreen buffer is destroyed when the applet's stop method is called, which occurs whenever the applet stops being visible:

```
public void stop(){        //  called when user moves off the page
    //  stop animating thread
    animatorThread = null;

    // Get rid of objects necessary for double buffering
    offGraphics = null;
    offImage = null;
}
```

Finally we write a run method (as promised when we declared "DBThreadedApplet implements Runnable"):

```
public void run() {
```

***

```
        while (Thread.currentThread() == animatorThread)        {
            //advance the  position and display ball.
            repaint();

            try  { Thread.sleep(20); }                    //  pause briefly
            catch (InterruptedException e)   {}
        }       //  end of animation loop
    }           //  end of run
```

DWAnimationApplet has a fancier user interface than DBThreadedApplet (it has buttons for starting and stopping it, and textFields for changing parameters), but the threading is basically identical. Please go through the listing of DWAnimationApplet.java in appendix A and make sure that you understand it.

Incidentally, all along we have been using an interface for our applets which listen for events. Therefore, when DWAnimationApplet is declared using the statement

```
public class DWAnimationApplet extends Applet implements Runnable, ActionListener
```

we are stating that DWAnimationApplet will implement both the Runnable interface and the ActionListener interface. As just described, to implement the Runnable interface, the applet must have a run method. To implement the ActionListener interface, the applet must have a method called `actionPerformed`:

```
public void actionPerformed(ActionEvent evt).
```

You can go back and verify that all the applets which implement the ActionListener interface do indeed have an actionPerformed method.

B. Phase portraits. There are two fundamentally different ways of presenting the solution to a classical mechanics problem. One is to plot coordinates (and momenta) like x and p as functions of time, which we did at the end of Chapter 6 as well as in the animations we just did in this Chapter. Now we discuss the other presentation method, the "phase plane portrait," in which orbits are drawn as trajectories in the (x, p) plane.

Interactive computation permits a particularly appealing way of displaying the phase plane portrait: One draws the phase space (i.e. the (x, p) plane) and the user chooses initial conditions by clicking with the mouse. To add this functionality to a Java program, we tell our Canvas where the phase portrait is plotted to listen for mouse events. This is done by implementing the MouseListener interface.

***

To do this, we must make sure that the methods mouseClicked, mousePressed, mouseReleased, mouseEntered, and mouseExited are all present. In the DWPhasePortraitApplet below, all of the methods called for by the interface except for mousePressed do nothing, but they all need to be defined.

mousePressed

public abstract void mousePressed(MouseEvent evt)

Invoked when the mouse button has been pressed on a component.

mouseClicked

public abstract void mouseClicked(MouseEvent evt)

Invoked when the mouse button has been clicked on a component.

mouseReleased

public abstract void mouseReleased(MouseEvent evt)

Invoked when the mouse button has been released on a component.

mouseEntered

public abstract void mouseEntered(MouseEvent evt)

Invoked when the mouse enters a component.

mouseExited

public abstract void mouseExited(MouseEvent evt)

Invoked when the mouse exits a component.

The methods getX and getY in the MouseEvent class return the x and y position of mouse events.

The applet DWPhasePortraitApplet plots phase portraits for the particle in the double-well potential, using the classes in “DoubleWell.java.” This applet has a separate plotting thread so that the user can interrupt it, clear the screen, and change the initial conditions. There is also a button that allows the user to change colors. The plotting is done on a separate canvas, which makes sure that the buttons aren't in the way. The pixel locations of

***

the mouse clicks are converted to (x, p) coordinates using the methods `itox` and `jtop`. This applet is in the project "DoubleWellPhasePortrait.mcp," which can be found in the folder sequence "Programs:Chapter_7: Phase Portraits."

```
// DWPhasePortraitApplet.java
/*
    applet to construct phase portrait for motion of particle in
    double-well potential
*/

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class DWPhasePortraitApplet extends Applet
                                        implements Runnable, ActionListener
{
    DoubleWell dw1;
    VariableSet tvars;
    DWPhasePortraitCanvas b;    //  canvas that draws phase portrait

    Button startButton, stopButton, resetButton;     // buttons to control flow
    Button colorButton;             //  button to allow color change
    TextField forceTF;              //  text fields for user input of force,
    TextField transienttimeTF;      //  time discarded before plotting starts

    Thread animatorThread;          /*  put plotting in separate thread so that user
                                    can stop it, change parameters, and restart it  */
    int delay;                      /*  delay between plotting calls so that program
                                    registers button clicks while it's going  */
    boolean frozen = false;         //  true if user has stopped the motion

    public void init() {
        int width = 400;            //  width of drawing canvas
        int height = 400;           //  height of drawing canvas
        delay = 10;                 //  delay in milliseconds between frames


        dw1 = new DoubleWell();     //  instantiate dynamical system object

        this.setSize(450,550);      //  set applet size

        b = new DWPhasePortraitCanvas(dw1);
```

* * *

```
b.setSize(width,height);                        // set canvas size
b.setBackground( Color.white );                 // set white background
b.repaint();                                    // draw axes on canvas

dw1.setdt(0.02);                                // set time step
dw1.setomega(2.*Math.PI/5.);                    // set drive frequency

// put the user interface components on the applet
Panel p = new Panel();
startButton = new Button("Start");
startButton.addActionListener( this );
stopButton = new Button("Stop");
stopButton.addActionListener( this );
p.add( startButton );
p.add( stopButton );

Panel p2 = new Panel();
Label transienttimeprompt = new Label("time delay before plotting:");
transienttimeTF = new TextField (Integer.toString(b.transienttime), 10);
p2.add( transienttimeprompt );
p2.add( transienttimeTF );

Panel p3 = new Panel();
p3.setLayout(new FlowLayout());
Label fprompt = new Label ("force:");
forceTF = new TextField (10);
forceTF.setText(Double.toString(dw1.getforce()));
Button resetButton = new Button("Clear Canvas");
resetButton.addActionListener( this )
p3.add( fprompt );
p3.add( forceTF );
p3.add( resetButton );

Panel p4 = new Panel();
Button colorButton = new Button("New Color");
colorButton.addActionListener( this );
p4.add( colorButton );
Panel p5 = new Panel();
Label textlabel = new Label("Click on canvas to set starting point");
p5.add( textlabel );

add(p);
add(b);
add(p2);
```

* * *

```
        add(p3);
        add(p4);
        add(p5);

   }                        //  end of init

     public void start()    {
        if (frozen)     {
            // Do nothing.  The user has stopped the motion.
        }
        else    {
            if (animatorThread == null ){
                animatorThread = new Thread (this);         //  make thread
            }
            animatorThread.start ();                   //  start thread
        }
     }

     public void stop(){         //  called when user moves off the page
        //  stop animating thread
        animatorThread = null;
     }

     public void run() {
        //  just to be nice, lower plotting thread's priority
        //  so it can't interfere with other processing going on.
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);

        //  This is the animation loop.
        while (Thread.currentThread() == animatorThread)       {
            // advance the position and plot.
            b.repaint();

            //  Delay briefly.
            try{
                Thread.sleep(delay);
            }
            catch (InterruptedException e)   {
                break;
            }
        }       //  end of animation loop
     }          //  end of run
```

* * *

```
  public void actionPerformed(ActionEvent evt)              //  handle user instructions
  { if (arg.getSource() == startButton) && (frozen == true))
    {    frozen = false;
            start();                                        //  start plotting
         }
         else if (evt.getSource() == stopButton)     {
            frozen = true;
            //  stop animating thread
            animatorThread = null;                          //  stop plotting
         }
         else if (evt.getSource() == resetButton)    {
            //  stop, then clear plotting canvas
            frozen = true;
            animatorThread = null;
            b.clear = true;
            b.repaint();                                    //  clear canvas
         }
         else if (evt.getSource() == colorButton)    {
            //  change color
            b.theColor = b.changeColor(b.theColor);
         }

         //  for any event, set parameters to the values in the TextFields
         String s = transienttimeTF.getText();       //  read string for transienttime
         b.transienttime = new Integer(s).intValue();       //  convert to double
         s = forceTF.getText();                             //  read string for force
         dw1.setforce(new Double(s).doubleValue() );        //  convert to double

    }                    //  end of actionPerformed method
}                        //  end of applet

class DWPhasePortraitCanvas extends Canvas                  //  canvas for plotting
                                  implements MouseListener  //  phase portrait
{
    boolean firstpaint = true;
    boolean clear = false;

    DoubleWell dw1;
    VariableSet vars, nextvars;
    double deltat;
    int n_var = 2;
    private double xvec[] = new double[n_var];
                                              //  xvec[0]=position, xvec[1]=momentum
    private double time;
```

* * *

```
Point position;                           //  position of point on graph:
                                          //  position.x = force, position.y = x

double pmin, pmax, xmin, xmax;            //  minimum and maximum values of
                                          //  x and p to be plotted on graph
double pheight, xwidth;                   //  ranges of x and p on graph
int imin, iwidth, jmax, jheight;          //  pixel values

int transienttime;                        //  number of periods discarded
                                          //  before plotting
Color theColor;
float redpart, greenpart, bluepart;

//  constructor for DWPhasePortraitCanvas:
DWPhasePortraitCanvas(DoubleWell mydw)    {
    super();                  //  first call Canvas constructor

    dw1 = mydw;                       //  get DoubleWell object
    vars = dw1.getvars();             //  get DoubleWell variables
    deltat = dw1.getdt();

    nextvars = vars;

    //  set initial values of transienttime and force:
    transienttime = 0;
    dw1.setforce(0.9);

    // set initial values of fmin, fmax, xmin, xmax:
    pmin = -2.;
    pmax = 2.;
    pheight = pmax-pmin;

    xmin = -1.5 ;
    xmax =  1.5 ;
    xwidth = xmax-xmin;

    firstpaint = true;

    theColor = Color.blue;
    addMouseListener( this );    // tells canvas to handle mouse events
}                                // end of DWPhasePortraitCanvas constructor

public void mousePressed( MouseEvent evt )    {                //  mouse pressed
    double[] temp = {itox( evt.getX() ), jtop(evt.getX())};    //  location of click
```

* * *

```
        dw1.setvars( new VariableSet(dw1.gettime(), temp));
        repaint();
    }
    // other mouse event methods required for MouseListener interface
    public void mouseReleased(MouseEvent evt) {}
    public void mouseEntered(MouseEvent evt) {}
    public void mouseExited(MouseEvent evt) {}
    public void mouseClicked(MouseEvent evt) {}

    public Color changeColor(Color c)    {
        if(c == Color.blue) {return Color.magenta;}
        else if(c == Color.magenta) {return Color.darkGray;}
        else if(c == Color.darkGray) {return Color.cyan;}
        else if(c == Color.cyan) {return Color.red;}
        else if(c == Color.red) {return Color.green;}
        else {return Color.blue;}
    }

    int ptoj(double p){                 //  converts p to screen coord j
        return Math.round( (float) (jmax - jheight*(p-pmin)/pheight) );
    }

    double jtop(int j){                 //  converts screen coord j to p
        return pmin + (jmax - j)*pheight/jheight;
    }

    int xtoi(double x){                 //  converts x to screen coord i
        return Math.round( (float) (imin + iwidth*(x-xmin)/xwidth) );
    }

    double itox( int i){                //  converts screen coordinate i to x
        return xmin + (i - imin)*xwidth/iwidth;
    }

    Point vartoPoint(VariableSet v) {    //  converts  VariableSet to plotting point
        return new Point( xtoi( v.getx()[0] ), ptoj( v.getx()[1] ) );
    }

    public void drawbox(Graphics g){    //  draw box, numbers, and labels for graph
            g.setColor(Color.black);
            g.drawRect(imin, jmax-jheight, iwidth, jheight);

            g.drawString(Double.toString(xmin), imin-10, jmax+15);
            g.drawString(Double.toString(xmax), imin+iwidth-10, jmax+15);
```

* * *

```
            g.drawString("x", imin+iwidth/2-10, jmax+15);
            g.drawString(Double.toString(pmin), imin-30, jmax+5);
            g.drawString(Double.toString(pmax), imin-30, jmax-jheight+5);
            g.drawString("p", imin-30, jmax-jheight/2+10);
    }

    public void update(Graphics g)   {
        paint( g );
    }

    public void paint ( Graphics g )  {

        if(firstpaint == true)     {
            // set imin, iwidth, jmin, jheight based on size of the canvas:
            imin   =  3*(getSize().width/20);
            iwidth =  getSize().width - (3*imin/2);
            jmax   =  getSize().height - 2*(size().height/20);
            jheight = getSize().height - 3*(size().height/20);

            drawbox(g);        //  draw the box, numbers, and labels

            position = vartoPoint(vars);
            firstpaint = false;
        }

        //  Get new position
        nextvars = dw1.nextvars();
        time = nextvars.gettime();
        dw1.setvars(nextvars);

        vars = nextvars;

//  discard points if time < transienttime :
        if((time > transienttime-.001) )    {
            position.move(vartoPoint(nextvars).x, vartoPoint(nextvars).y);

            g.setColor(theColor);
            g.drawRect(position.x, position.y, 0, 0);         // put point (x, p) on canvas
        }

        if(clear == true) {    //  clear canvas if button was pressed
            g.clearRect(0, 0, size().width,size().height);
            clear = false;
        }
```

* * *

```
        drawbox(g);
    }                                   //  end of update
}
```

Exercise 7.3. Use the DWPhasePortraitApplet to plot phase space portraits of the evolution of the double-well system. How does changing the initial conditions affect the motion at long times? Characterize how these portraits depend on the drive amplitude F_0. Make sure that you understand how the behavior of the animation (position versus time) is related to the structure of the phase plot.

Problem 7.1. In 1963 E. Lorenz, in work that essentially started the chaos business, considered the behavior of the system of equations:

$$\frac{d}{dt}x = p(y-x) \tag{7.1a}$$

$$\frac{d}{dt}y = -xz + rx - y \tag{7.1b}$$

$$\frac{d}{dt}z = xy - bz\,. \tag{7.1c}$$

Write a program that uses fourth order Runge-Kutta to integrate these equations for the parameter values p = 10, b = 8/3, and, r = 28 (we suggest Δ = 0.01 as a good step size). In this system of three first order equations, each point in phase space is described by a three-dimensional vector (x,y,z). Since we only have two-dimensional graphics, have your applet plot a phase space portrait of the orbit projected onto the (z, y) plane. Then try to get a feel for the three-dimensional structure of the "attractor" (i.e. the orbit in phase space that the system converges to) by projecting it onto different planes. Although the motion appears smooth and continuous, it is nevertheless said to be “chaotic.” Can you explain why? (Hint: try graphing x or y vs. time).

C. Volumes in phase space. Now instead of just plotting a single orbit in the (x,p) plane, we plot a whole group of trajectories that start out near each other together as a set of curves in the (x, p) plane. Thus we are considering how regions in phase space evolve.

***

Problem 7.2. Write a program for the damped particle in the double well potential that plots the phase space evolution of many points that all start out in a small area. Choose a value of F_0 where the long-time motion is relatively simple. How does the evolution of the areas change as γ, the damping parameter of the system, is varied?

We can calculate how the area of a phase space region changes as time evolves. Consider a phase space region that at time $t = t_0$ is a small rectangle, with vertices $\mathbf{z}_{00} = \begin{pmatrix} x_0 \\ p_0 \end{pmatrix}$, $\mathbf{z}_{10} = \begin{pmatrix} x_0 + \delta x \\ p_0 \end{pmatrix}$, $\mathbf{z}_{10} = \begin{pmatrix} x_0 \\ p_0 + \delta p \end{pmatrix}$, and $\mathbf{z}_{11} = \begin{pmatrix} x_0 + \delta x \\ p_0 + \delta p \end{pmatrix}$.

The equation prescribing the configuration at time $t = t_0 + \Delta$ can be written in the form:

$$\mathbf{z}(t_0 + \Delta) = \mathbf{f}(\mathbf{z}(t_0), t_0). \tag{7.2}$$

Here, $\mathbf{z}(t) = \begin{pmatrix} x(t) \\ p(t) \end{pmatrix}$ and $\mathbf{f}(\mathbf{z}(t), t) = \begin{pmatrix} g(x, p, t) \\ h(x, p, t) \end{pmatrix}$. At time $t_0 + \Delta$ the four vertices have moved:

$$\begin{aligned}
\mathbf{z}_{00}(t_0 + \Delta) &= \mathbf{f}(\mathbf{z}_{00}(t_0), t_0); \\
\mathbf{z}_{10}(t_0 + \Delta) &= \mathbf{z}_{00}(t_0 + \Delta) + \delta x \begin{pmatrix} \partial g/\partial x \\ \partial h/\partial x \end{pmatrix}; \\
\mathbf{z}_{01}(t_0 + \Delta) &= \mathbf{z}_{00}(t_0 + \Delta) + \delta p \begin{pmatrix} \partial g/\partial p \\ \partial h/\partial p \end{pmatrix}; \\
\mathbf{z}_{11}(t_0 + \Delta) &= \mathbf{z}_{00}(t_0 + \Delta) + \delta x \begin{pmatrix} \partial g/\partial x \\ \partial h/\partial x \end{pmatrix} + \delta p \begin{pmatrix} \partial g/\partial p \\ \partial h/\partial p \end{pmatrix};
\end{aligned} \tag{7.3}$$

plus corrections that are higher order in δx and δp.

* * *

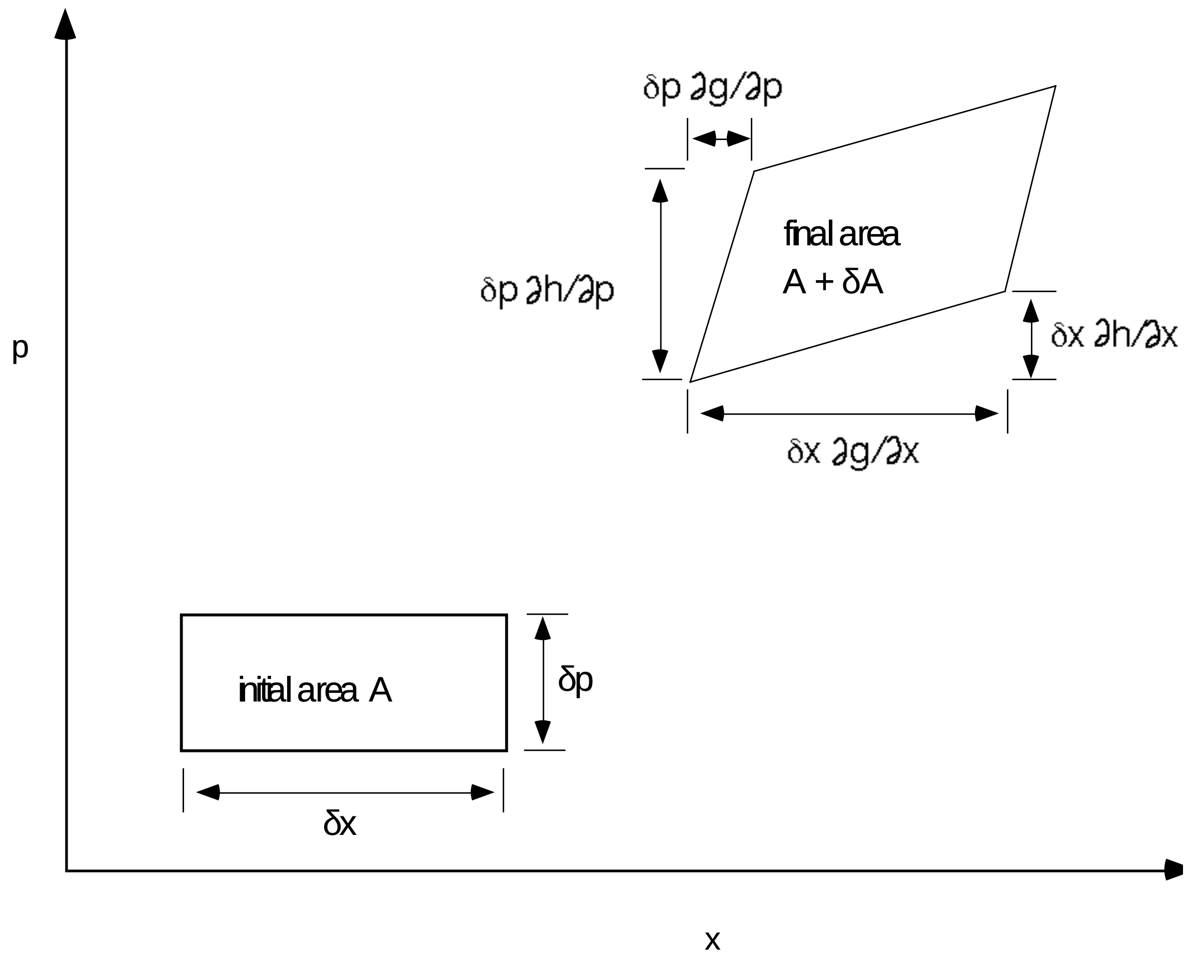


Figure 7.2. Diagram of how a phase space region evolves. The region on the left maps into the one on the right after a time Δ has elapsed.

Problem 7.3. Show analytically that the change δA in the area A obeys:

$$\frac{\delta A}{A} = \left(\frac{\partial g}{\partial x}\frac{\partial h}{\partial p} - \frac{\partial g}{\partial p}\frac{\partial h}{\partial x} - 1 \right). \tag{7.4}$$

In the limit when the time step $\Delta \to 0$ (in other words, when the dynamical system is described by a set of differential equations), then this result for the area change simplifies even more.

* * *

Problem 7.4. In the limit $\Delta \to 0$, $g \approx x + \Delta \frac{dx}{dt}$ and $h \approx p + \Delta \frac{dp}{dt}$. Show analytically that in this limit the time evolution of the area A satisfies the differential equation

$$\frac{1}{A}\frac{dA}{dt} = \frac{\partial}{\partial x}\left(\frac{dx}{dt}\right) + \frac{\partial}{\partial p}\left(\frac{dp}{dt}\right). \qquad (7.5)$$

Now let's specialize to systems described by equations of the form:

$$m\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} = -\frac{dV(x)}{dx} + F_0\cos(\omega t) \ . \qquad (7.6)$$

(As you recall, the equation of motion for the particle in the double well has this form, with $V(x) = -k_1x^2/2 + k_2x^4/4$.)

Exercise 7.4 Show that the phase space evolution for equation (7.6) is described by equation (7.5) with $\frac{dx}{dt} = \frac{p}{m}$ and $\frac{dp}{dt} = F_0\cos(\omega t) - \frac{\gamma p}{m} - \frac{dV(x)}{dx}$, and hence:

$$\frac{1}{A}\frac{dA}{dt} = -\frac{\gamma}{m}. \qquad (7.7)$$

The solution to equation (7.7) is:

$$A(t) = A(t = t_0)\exp\left(-\frac{\gamma}{m}(t - t_0)\right) \ . \qquad (7.8)$$

Thus, whenever $\gamma \neq 0$, phase space areas shrink exponentially.

Menu Project. Mapping Regions for a Double Well System.

In this project, you will investigate the evolution of phase space regions for the damped driven motion of a particle in the double well. The idea is to take a whole bunch of points (at least a thousand) bunched into a fairly small area, propagate them all forward in time using the equations of motion, and see what the resulting region looks like. Can you think of a way of estimating the area of the resulting region? In any case, try to determine if the area seems to behave as expected from equation (7.8)

* * *

According to equation (7.8), how the area of the region changes in time should be independent of the value of F_0. However, the shape of the region does depend on F_0. Investigate the evolution of the shapes of the regions in the different regimes. This project is quite open-ended, and we expect you to do interesting and imaginative work.

D. Area-Preserving Systems. We have just seen that the rate of change of areas of regions in phase space is proportional to the damping constant in equation (7.6). If the damping is zero, then the areas in phase space remain constant for all time—a much different result than an exponential decay!

This result is closely related to a proposition known in classical mechanics as Liouville's theorem, which states that a Hamiltonian system is volume preserving in phase space. “Hamiltonian” means that the system can be described in terms of pairs of generalized coordinates and momenta q_i, p_i for i=1,2,...N, and that these coordinates obey Hamilton's equations, which are:

$$\frac{dq_i}{dt} = \frac{\partial H}{\partial p_i}$$
$$\frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i}.$$

The motion is thus controlled by the Hamiltonian function, H(q,p). Examples of Hamiltonian systems include the undamped pendulum and motion in a central potential. Hamiltonian dynamics arise not only in mechanical systems with no friction but also in a variety of other problems, such as the paths followed by magnetic field lines in a plasma, the mixing of fluids, and motion in particle accelerators.

Exercise 7.5 Determine H for an undamped pendulum with equation of motion:

$$\frac{d^2\theta}{dt^2} = -\sin\theta \qquad (7.9)$$

in terms of the angular coordinate, θ, and the momentum, $p = \frac{d\theta}{dt}$.

* * *

Problem 7.5 Show analytically that the pendulum equation (7.9) preserves areas in phase space.

Exercise 7.6. Consider the case of undamped one dimensional motion in a uniform gravitational field. Do an analytical calculation to determine what happens to a rectangle representing various initial conditions (such as the rectangle shown in figure 7.2) after the system has evolved for a time δt. Is the area in phase space conserved in this case?

Hamiltonian systems preserve phase space areas exactly as they evolve in time, but when we solve differential equations on the computer, we use a finite time step and make a finite error, and unless we are very careful we will not capture this feature of the real system. In the next chapter we will see how important this type of error can be.

Appendix A: The applet DWAnimationApplet

```
// DWAnimationApplet.java
/*
    applet (with double-buffering but no clipping) of motion of particle
    in double-well potential
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class DWAnimationApplet extends Applet implements Runnable, ActionListener
{
    DoubleWell dw1;
    VariableSet tvars;
    DoubleWellCanvas b;
    Thread animatorThread;
    int delay;
    boolean frozen = false;
    Button startButton, stopButton, resetButton;
    TextField forceTF;
    TextField dtTF;
    TextField delayTF;

    public void init ()
    {
        int width = 250;
```

* * *

```
int height = 250;
delay = 200;        //  delay in milliseconds between frames


dw1 = new DoubleWell();

this.setSize(320,400);

b = new DoubleWellCanvas(dw1);
b.setSize(width,height);                        //  set canvas size
b.setBackground( Color.white );                 //  set white background

Panel p = new Panel();
startButton = new Button("Start");
startButton.addActionListener( this );
stopButton = new Button("Stop");
stopButton.addActionListener( this );
p.add( startButton );
p.add( stopButton );

Panel p2 = new Panel();
Label delayprompt = new Label("animation delay (in millisec):");
delayTF = new TextField (Integer.toString(delay), 10);
p2.add( delayprompt );
p2.add( delayTF );

Panel p3 = new Panel();
p3.setLayout(new FlowLayout());
Label dtprompt = new Label("deltat:");
Label fprompt = new Label ("force:");
dtTF = new TextField (10);
dtTF.setText(Double.toString(dw1.getdt()));
forceTF = new TextField (10);
forceTF.setText(Double.toString(dw1.getforce()));
resetButton = new Button("Reset");
resetButton.addActionListener( this );
p3.add( dtprompt );
p3.add( dtTF );
p3.add( fprompt );
p3.add( forceTF );
p3.add( resetButton );

add(p);
add(b);
```

***

```
        add(p2);
        add(p3);

    }                           //  end of init

    public void start()
    {
        if (frozen)     {
            //Do nothing.  The user has stopped the motion.
        }
        else     {
            if (animatorThread == null ){
                animatorThread = new Thread (this);
            }
            animatorThread.start ();
        }
    }

    public void stop()          //  called when user moves off the page
    {
        //  stop animating thread
        animatorThread = null;

        //Get rid of objects necessary for double buffering
        b.offGraphics = null;
        b.offImage = null;
    }

    public void run() {
        //  just to be nice, lower this thread's priority
        //  so it can't interfere with other processing going on.
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);


        // Remember the starting time.
        long startTime = System.currentTimeMillis();

        //  This is the animation loop.
        while (Thread.currentThread() == animatorThread)        {
            //advance the  position and display ball.
            b.repaint();

            //Delay depending on how long we have taken so far.
            try{
```
* * *

```
                startTime += delay;
                Thread.sleep(Math.max(0,
                                     startTime-System.currentTimeMillis()));
            }
            catch (InterruptedException e)   {
                break;
            }
        }          //  end of animation loop
    }              //  end of run


  public void actionPerformed(ActionEvent evt)
  { if (evt.getSource() == startButton && (frozen == true))
    {   frozen = false;
            start();
        }
        else if (evt.getSource() == stopButton)      {
            frozen = true;
            //  stop animating thread
            animatorThread = null;
        }
        else if (evt.getSource() == resetButton)     {
            dw1.setvars( dw1.resetvars );
        }

        //  for any event, reset parameters if they have changed
        //  read both deltat and force
        String s = dtTF.getText();                          //  read string for deltat
        dw1.setdt( new Double(s).doubleValue() );           //  convert to double
        s = forceTF.getText();                              //  read string for force
        dw1.setforce(new Double(s).doubleValue() );         //  convert to double
        s = delayTF.getText();                              //  read string for delay
        delay = new Integer(s).intValue() ;
        b.firstpaint = true;
    }                    //  end of actionPerformed method

}                        //  end of applet

class DoubleWellCanvas extends Canvas  {

    private static final int BALLRADIUS = 5;

    Dimension offDimension;           //  Dimension of offscreen buffer
    Image offImage;
```

* * *

```
    Graphics offGraphics;          //  Graphics object for offscreen buffer
    boolean firstpaint = true;

    DoubleWell dw1;
    VariableSet vars;
    double deltat;
    double force;
    int n_var = 2;
    private double xvec[] = new double[n_var];     // xvec[0]=position,
                                                   // xvec[1]=velocity
    private double time;
    private double y;                              // vertical position, set by potential
    Point position;

    double xmin, xmax, ymin, ymax;                 //  minimum and maximum values of
                                                   //  x and y to be plotted on graph
    double xwidth, yheight;
    int imin, iwidth, jmax, jheight;               //  pixel values

    int ipts[] = new int[201];             //  arrays for graph of potential versus x
    int jpts[] = new int[201];


    //  constructor for DoubleWellCanvas:
    DoubleWellCanvas(DoubleWell mydw)  {
        super();                   //  first call Canvas constructor

        dw1 = mydw;
        vars = dw1.getvars();

    }

    int xtoi(double x){    //  converts x to screen coord i
        return Math.round( (float) (imin + iwidth*(x-xmin)/xwidth) );
    }

    int ytoj(double y){    //  converts y to screen coord j
        return Math.round( (float) (jmax - jheight*(y-ymin)/yheight) );
    }

    Point vartoPoint(VariableSet v) {
        double x = v.getx()[0];
        double t = v.gettime();
        return new Point( xtoi(x), ytoj( dw1.potential(x, t) ) );
```

* * *

```
    }

    void clipToAffectedArea( Graphics g, Point oldp, Point newp,
                                                            int radius)     {
        int x = Math.min( oldp.x, newp.x) - radius;
        int y = Math.min( oldp.y, newp.y) - radius;
        int w = ( Math.max( oldp.x, newp.x) + radius) - x;
        int h = ( Math.max( oldp.y, newp.y) + radius) - y;
        g.setClip( x, y, w, h );
    }

    void draw(Graphics g, Point p)   {
        g.fillOval(p.x-BALLRADIUS ,p.y-BALLRADIUS,
            2*BALLRADIUS,2*BALLRADIUS);                          //  draw ball
    }

    void putpotential(Graphics g)     {
            //  Put potential on canvas
            g.setColor(Color.black);


            double xo;
            ipts[0]     = xtoi(xmin);
            jpts[0] = ytoj (dw1.potential(xmin, dw1.gettime() ) );
            for( int i = 1; i<=200; i++){
                xo = xmin + i*xwidth/200;
                ipts[i] = xtoi( xo );
                jpts[i] = ytoj( dw1.potential(xo,dw1.gettime() ) );
                g.drawLine(ipts[i-1], jpts[i-1], ipts[i], jpts[i]);

            }
        }


    public void paint(Graphics g)     {

        //  erase old image
        g.setColor(getBackground());
        g.fillRect(0,0,this.getSize().width, this.getSize().height);

        //  Draw potential
        putpotential(g);

        //  Draw new position
```
***

```
        g.setColor(Color.blue);
        draw(g, position);
    }

    public void update ( Graphics g ){
    //  Create the offscreen graphics context, if no good one exists.
        if (offGraphics == null)  {
            offImage = createImage( getSize().width, getSize().height);
        }
            offGraphics = offImage.getGraphics();

        if(firstpaint == true)     {
            // set xmin, xmax, ymin, ymax:
            xmin = -1.2*dw1.getA();
            xmax =  1.2*dw1.getA();
            xwidth = xmax-xmin;

            ymin = dw1.minpotential(xmin, xmax) + xmin*dw1.getforce();
            ymin = ymin -0.05*Math.abs(ymin);
            ymax = dw1.maxpotential(xmin, xmax) + xmax*dw1.getforce();
            ymax = ymax + 0.05*Math.abs(ymax);
            yheight = ymax-ymin;

            // set imin, iwidth, jmin, jheight:
            imin = this.getSize().width/20;
            iwidth = this.getSize().width - 2*imin;
            jmax = getSize().height - getSize().height/20;
            jheight = getSize().height - 2*(getSize().height/20);

            position = vartoPoint(vars);
            firstpaint = false;
        }

        //  Get new position
        time += deltat;
        vars = dw1.nextvars();
        dw1.setvars(vars);
        position.move(vartoPoint(vars).x, vartoPoint(vars).y-BALLRADIUS-1);

        //  Set clipping rectangle
//      clipToAffectedArea(offGraphics, lastposition, position, BALLRADIUS);

        //  Set clipping rectangle on screen
//      clipToAffectedArea(g, lastposition, position, BALLRADIUS);
```

* * *

```
        paint(offGraphics);                     //  draw into offGraphics buffer

        g.drawImage(offImage, 0, 0, this);      //  draw buffer onto Canvas

//      offGraphics.dispose();
    }

}
```

Appendix B: Making Animations Run Faster

One reason that using Java to animate web pages is an attractive strategy is that if a user has enough patience to wait for the applet to download, then the animation speed depends only on the processing speed of the user's machine and not at all on the speed of the network connection. We have checked that our DWAnimationApplet looks acceptable on the slowest machines in the MacLab (PowerMac 7100's). If your machine is faster, then you can decrease the delay between frames, which makes the animation go faster and look better, without destroying the responsiveness of the buttons. (The easiest way to check this out is to reduce the delay and see if the ball speeds up. If you have a really fast machine, you might want to reduce the time step so that the particle doesn't move quite so far between frames.)

However, at some point you could be desperate to get an animation to run faster. For it to run faster on any machine, then you need to reduce the amount of computation that occurs between frames. If you are using an efficient algorithm for the computations, then the strategy most likely to speed things up is to repaint only those parts of the image that change between updates by clipping the drawing region using the `setClip()` method of the Graphics class.

***

```
setClip
public abstract void
     setClip(int x, int y, int width, int height)
Sets a clipping rectangle for this graphics context to
the rectangle specified by the given coordinates.
Graphics operations performed with this graphics context
have no effect outside the clipping area.
Parameters:
     x - the x coordinate of the new clip rectangle
     y - the y coordinate of the new clip rectangle
     width - the width of the new clip rectangle
     height - the height of the new clip rectangle
```

We should warn you that when we experimented with clipping by displaying a static quartic potential in our DWAnimationApplet and repainting only the ball, the improvement in performance was marginal and (particularly when run using Netscape) the applet looked much worse. So we cannot guarantee that clipping will be a useful strategy for your animation. But it's worth trying.

If you are not so worried about anonymous web-surfers and want an applet to run faster on your own machine, there are a two effective strategies you can use. They are: (1) use a faster computer (effective, but can be expensive), and (2) use a just-in-time compiler. Apple's just-in-time compiler, which they claim speeds up execution on PowerMacs by a factor of 10 (and certainly ran DWAnimationApplet for us much faster than Metrowerks Java did), can be downloaded from the Apple web site, at http://www.apple.com/macos/java. Internet Explorer 3.0 and above and Netscape 3.0 and above on both Windows and on Mac machines have built-in just-in-time compilers. Therefore, many anonymous web-surfers will see your animations run fast without doing anything special.

***