```
      field[2] = 0; // magnetic field
   }

   double dsSquared(int i, double t, double x, double y) {
      double dt = t-path[0][i];
      double dx = x-path[1][i];
      double dy = y-path[2][i];
      return dx*dx+dy*dy-dt*dt;
   }

   void calculateRetardedField(double x, double y, double[] field) {
      int first = 0;
      int last = numPts-1;
      double ds_first = dsSquared(first, t, x, y);
      if(ds_first>=0) { // field has not yet propagated to the location
         electrostaticField(x, y, field);
         return;
      }
      while((ds_first<0)&&(last-first)>1) {
         int i = first+(last-first)/2; // bisect the interval
         double ds = dsSquared(i, t, x, y);
         if(ds<=0) {
            ds_first = ds;
            first = i;
         } else {
            last = i;
         }
      }
      double t_ret = path[0][first]; // time where ds changes sign
      r[0] = x-evaluate(t_ret); // evaluate x at retarded time
      r[1] = y;                 // evaluate y at retarded time
      // derivative of x at retarded time
      v[0] = Derivative.centered(this, t_ret, dt);
      v[1] = 0;                 // derivative of y at retarded time
      // acceleration of x at retarded time
      a[0] = Derivative.second(this, t_ret, dt);
      a[1] = 0;                 // acceleration of y at retarded time
      double rMag = Vector2DMath.mag2D(r); // magnitdue of r
      u[0] = r[0]/rMag-v[0];
      u[1] = r[1]/rMag-v[1];
      double r_dot_u = Vector2DMath.dot2D(r, u);
      double k = rMag/r_dot_u/r_dot_u/r_dot_u;
      // u cross a is perpendicular to plane of motion
      double u_cross_a = Vector2DMath.cross2D(u, a);
      double[] temp = {r[0], r[1]};
      temp = Vector2DMath.crossZ(temp, u_cross_a); // r cross u
      // (c*c - v*v) where c = 1
      double c2v2 = 1-Vector2DMath.dot2D(v, v);
      double ex = k*(u[0]*c2v2+temp[0]);
      double ey = k*(u[1]*c2v2+temp[1]);
      field[0] = ex;
      field[1] = ey;
      field[2] = k*Vector2DMath.cross2D(temp, r)/rMag;
   }

   public void draw(DrawingPanel panel, Graphics g) {
```



```
      circle.setX(evaluate(t));
      circle.draw(panel, g); // draw the charged particle on the screen
   }

   public double evaluate(double t) {
      return 5*Math.cos(t*vmax/5.0);
   }
}
```

The `RadiatingCharge` class computes the electric field due to an oscillating charge using the Liénard–Wiechert potentials. We choose units such that the speed of light $c = 1$. As the charge moves, it stores its ith data point in a two-dimensional array `path[3][i]` containing the time, its x-position, and its y-position. To find the retarded time at the position (x, y), we use the `dsSquared` method to compute the square of the space-time interval between the given location and points along the path. The square of the space-time separation is defined as

$$\Delta s^2 = \Delta x^2 + \Delta y^2 - c^2 \Delta t^2, \tag{10.45}$$

where $\Delta x = x - x_{\text{path}}$, $\Delta y = y - y_{\text{path}}$, and $\Delta t = t - t_{\text{path}}$. The last point on the path contains the current position of the charge so Δs^2 must be positive because Δt is zero (unless the charge is at the observation point (x, y) in which case Δs^2 is zero and the field is infinite due to the $1/r^2$ dependence). The `calcRetardedField` method evaluates Δs^2 at the first point in the trajectory to determine if it is negative. We assume the charge was stationary for $t < 0$ and compute the electrostatic field if Δs^2 is positive at the trajectory's first point where $t = 0$. If Δs^2 is negative at the trajectory's first point, we repeatedly bisect the path into smaller and smaller segments while checking to see if Δs^2 remains negative at the beginning of the segment and positive at the end. In this way we can find the retarded time when we have a path segment bounded by two data points. Note that the `RadiatingCharge` class uses the `Vector2DMath` class to perform the necessary vector arithmetic. This helper class is not listed but is available in ch10 code package.

The `RadiatingEFieldApp` program is shown in Listing 10.7. It displays the electric field in the xy-plane using a `Vector2DFrame`. The `calculateFields` method computes the retarded field at every grid point. The simulation's `doStep` method invokes this method after it moves the charge.

Listing 10.7 The `RadiatingEFieldApp` program computes the radiating electric and magnetic fields using Liénard–Wiechert potentials.

```
package org.opensourcephysics.sip.ch10;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.Vector2DFrame;

public class RadiatingEFieldApp extends AbstractSimulation {
   Vector2DFrame frame = new Vector2DFrame("x", "y", "Electric field");
   RadiatingCharge charge = new RadiatingCharge();
   int gridSize;      // linear dimension of grid used to compute fields
   double[][][] Exy; // x and y components of electric field
   double xmin = -20, xmax = 20, ymin = -20, ymax = 20;

   public RadiatingEFieldApp() {
      frame.setPreferredMinMax(xmin, xmax, ymin, ymax);
```