Because the differential cross section is usually independent of ϕ, we need to consider beam particles at only $\phi = 0$. We have to take into account the fact that in a real beam, there are more particles at some values of b than at others. That is, the number of particles in a real beam is proportional to $2\pi b \Delta b$, the area of the ring between b and $b + \Delta b$, where we have integrated over the values of ϕ to obtain the factor of 2π. Here Δb is the interval between the values of b used in the program. Because there is only one target in the beam, the target density is $n = 1/(\pi R^2)$.

The scattering program requires the `Scatter`, `ScatterAnalysis`, and `ScatterApp` classes. The `ScatterApp` class in Listing 5.6 organizes the startup process and creates the visualizations. As usual, it extends `AbstractSimulation` by overriding the `doStep` method. However, in this case a single step is not a time step. A step calculates a trajectory and scattering angle for the given impact parameter. After a trajectory is calculated, the impact parameter is incremented and the panel is repainted. If necessary, you can eliminate this visualization to increase the computational speed. If the new impact parameter exceeds the beam radius `bmax`, the animation is stopped and the accumulated data is analyzed. Note that the `calculateTrajectory` method returns `true` if the calculation succeeded and that an error message is printed if the calculation fails. Including a failsafe mechanism to stop a computation is good programming practice.

Listing 5.6 A program that calculates the scattering trajectories and computes the differential cross section.

```
package org.opensourcephysics.sip.ch05;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class ScatterApp extends AbstractSimulation {
   PlotFrame frame = new PlotFrame("x", "y", "Trajectories");
   ScatterAnalysis analysis = new ScatterAnalysis();
   Scatter trajectory = new Scatter();
   double vx;    // speed of the incident particle
   double b, db; // impact parameter and increment
   double bmax;  // maximum impact parameter

   public ScatterApp() {
      frame.setPreferredMinMax(-5, 5, -5, 5);
      frame.setSquareAspect(true);
   }

   public void doStep() {
      if(trajectory.calculateTrajectory(frame, b, vx)) {
         analysis.detectParticle(b, trajectory.getAngle());
      }
      else {
         control.println("Trajectory did not converge at b = "+b);
      }
      frame.setMessage("b = "+decimalFormat.format(b));
      b += db; // increases the impact parameter
      frame.repaint();
      if(b>bmax) {
         control.calculationDone("Maximum impact parameter reached");
         analysis.plotCrossSection(b);
      }
   }
```



```
   public void initialize() {
      vx = control.getDouble("vx");
      bmax = control.getDouble("bmax");
      db = control.getDouble("db");
      b = db/2; // starts b at average value of first interval 0->db
      // b will increment to 3*db/2, 5*db/2, 7*db/2 ...
      frame.setMessage("b = 0");
      frame.clearDrawables(); // removes old trajectories
      analysis.clear();
   }

   public void reset() {
      control.setValue("vx", 3);
      control.setValue("bmax", 0.25);
      control.setValue("db", 0.01);
      initialize();
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new ScatterApp());
   }
}
```

The `Scatter` class shown in Listing 5.7 calculates the trajectories by expressing the equation of motion as a rate equation. The most important method is `calculateTrajectory`, which calculates a trajectory by stepping the differential equation solver and adding the resulting data to a trail to display the path. Because the beam source is far away, we stop the calculation when the distance of the scattered particle from the target exceeds the initial distance. Note the use of the ternary `?:` operator. This very efficient and compact operator uses three expressions. The first expression evaluates to a boolean. If this expression is true, then the statement after the `?` is executed. If this expression is false, then the statement after the `:` is executed. However, because some potentials may trap particles for long periods of time, we also stop the calculation after a predetermined number of time steps.

Listing 5.7 A class that models particle scattering using a central force law.

```
package org.opensourcephysics.sip.ch05;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class Scatter implements ODE {
   double[] state = new double[5];
   RK4 odeSolver = new RK4(this);

   public Scatter() {
      odeSolver.setStepSize(0.05);
   }

   boolean calculateTrajectory(PlotFrame frame, double b, double vx) {
      state[0] = -5.0; // x
      state[1] = vx;   // vx
      state[2] = b;    // y
      state[3] = 0;    // vy
```