The rectangular and trapezoidal algorithms converge relatively slowly and are therefore not recommended in general. In practice, Simpson's rule is adequate for functions that are reasonably well behaved, that is, functions that can be adequately represented by a polynomial. If $f(x)$ is such a function, we can adopt the strategy of evaluating the area for a given number of intervals n and then doubling the number of intervals and evaluating the area again. If the two evaluations are sufficiently close to one another, we stop. Otherwise, we again double n until we achieve the desired accuracy. Of course, this strategy will fail if $f(x)$ is not well behaved. An example of a poorly behaved function is $f(x) = x^{-1/3}$ at $x = 0$, where $f(x)$ diverges. Another example where this strategy might fail is when a limit of integration is equal to $\pm\infty$. In many cases we can eliminate the problem by a change of variables.

Problem 11.3 Simpson's rule

(a) Write a class that implements Simpson's rule. Either adapt your program so that it uses Simpson's rule directly or convince yourself that the result of Simpson's rule can be expressed as

$$S_n = (4T_{2n} - T_n)/3, \tag{11.8}$$

where T_n is the result from the trapezoidal approximation for n intervals. It is not necessary to provide a visualization of the area. Use your program to evaluate the integral of $f(x) = (2\pi)^{-1/2} e^{-x^2}$ for $|x| \leq 1$. Do you obtain the same result by choosing the interval [0, 1] and then multiplying by two?

(b) Determine the same integrals as in Problem 11.2 and discuss the relative merits of the various approximations.

(c) Evaluate the integral of the function $f(x) = 4\sqrt{1 - x^2}$ for $|x| \leq 1$. What value of n is needed for four decimal accuracy? The reason for the slow convergence can be understood by reading Appendix 11A.

*(d) Our strategy for approximating the value of the definite integrals we have considered has been to compute F_n and F_{2n} for reasonable values of n. If the difference $|F_{2n} - F_n|$ is too large, then we double n until the desired accuracy is reached. The success of this strategy is based on the implicit assumption that the sequence $F_n, F_{2n}, \ldots$ converges to the true integral F. Is there a way of extrapolating this sequence to the limit? Explore this idea by using the trapezoidal approximation. Because the error for this approximation decreases approximately as n^{-2}, we can write $F = F_n + Cn^{-2}$ and plot F_n as a function of n^{-2} to obtain the extrapolated result F. Apply this procedure to the integrals considered in some of the previous problems and compare your results to those found from the trapezoidal approximation and Simpson's rule alone. A more sophisticated application of this idea is known as *Romberg* integration (see Press et al.). ■

Because integration is usually a routine task, we have implemented the integration methods discussed in this section in the `Integral` class in the numerics package. Listing 11.3 illustrates how the `Integral` class is used. Method `Integral.ode` computes the integrals using an `ODE` solver as discussed in the following. The solver uses an adaptive step size to achieve the desired tolerance.

Listing 11.3 The `IntegralCalcApp` program tests the `Integral` class.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.*;

public class IntegralCalcApp extends AbstractCalculation {
   public void reset() {
      control.setValue("a", 0);
      control.setValue("b", 1);
      control.setValue("tolerance", 1.0e-2);
      control.setValue("f(x)", "sin(2*pi*x)^2");
   }

   public void calculate() {
      Function f;
      String fx = control.getString("f(x)");
      try { // read in function to integrate
         f = new ParsedFunction(fx);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         return;
      }
      double a = control.getDouble("a");
      double b = control.getDouble("b");
      double tolerance = control.getDouble("tolerance");
      double area = Integral.ode(f, a, b, tolerance);
      control.println("ODE area = "+area); // uses RK45MultiStep solver
      area = Integral.trapezoidal(f, a, b, 2, tolerance);
      control.println("Trapezoidal area = "+area);
      area = Integral.simpson(f, a, b, 2, tolerance);
      control.println("Simpson area = "+area);
      area = Integral.romberg(f, a, b, 2, tolerance);
      control.println("Romberg area ="+area);
   }

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

The algorithms in the `Integral` class are implemented using static methods so that they are easy to invoke. These methods accept a *tolerance* parameter that allows the user to specify the acceptable relative precision. Because computers store floating point numbers using a fixed number of decimal places, we use relative precision to determine the accuracy of the integration method. However, relative precision is meaningful only if the result is different from zero. If the result is zero, the only possible check is for absolute precision. Because numerical integration algorithms should check the precision of their output, the `Util` class in the numerics package defines the following general purpose method for computing the relative precision from an absolute precision and a numerical result.

```
public static double relativePrecision(final double epsilon,
    final double result) {
  return (result > defaultNumericalPrecision)
```