15.5.1
Euler's Rule

Euler's rule (Fig. 15.5) is a simple algorithm to integrate the differential equation (15.7) by one step. One simply substitutes the forward difference algorithm for the derivative:

$$\frac{d\mathbf{y}(t)}{dt} \simeq \frac{\mathbf{y}(t_{n+1}) - \mathbf{y}(t_n)}{h} = \mathbf{f}(t, \mathbf{y}) \tag{15.28}$$

$$\Rightarrow \qquad \mathbf{y}_{n+1} \simeq \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n) \tag{15.29}$$

where $y_n \stackrel{\text{def}}{=} y(t_n)$ is the value of y at time t_n. We know from our discussion of differentiation that the error in the forward difference algorithm is $\mathcal{O}(h^2)$, and so this too is the error in Euler's rule.

To indicate the simplicity of this algorithm, we apply it to our oscillator problem for the first time step:

$$y_1^{(0)} = x_0 + v_0 h \qquad y_1^{(1)} = v_0 + h\frac{1}{m}\left[F_{\text{ext}}(t=0) + F_k(t=0)\right] \tag{15.30}$$

Compare these to the projectile equations familiar from the first-year physics:

$$x = x_0 + v_0 h + \tfrac{1}{2}ah^2 \qquad v = v_0 + ah \tag{15.31}$$

We see that in (15.30) the acceleration does not contribute to the distance covered (no h^2 term), yet it does contribute to the velocity here (and so will contribute belatedly to the distance in the next time step). This is clearly a simple algorithm that requires very small h values to obtain precision, yet using small values for h increases the number of steps and the accumulation of the round-off error, which may lead to instability. (Instability is often a problem when you integrate a $y(t)$ which decreases as the integration proceeds, analogous to upward recursion of spherical Bessel functions. In that case, and if you have a linear problem, you are best off integrating *inward* from large times to small times and then scaling the answer to agree with the initial conditions.) Although we do not recommend Euler's algorithm for general use, it is commonly used to start some of the more sophisticated algorithms.

15.5.2
Runge–Kutta Algorithm

Although no one algorithm will be good for solving all ODEs, the fourth-order Runge–Kutta algorithm `rk4`, or its extension with adaptive step size, `rk45`, has proven to be robust and capable of industrial-strength work. Although `rk4` is our recommended standard method, we derive the simpler `rk2` and just give the results for `rk4`.

The Runge–Kutta algorithm for integrating a differential equation is based upon the formal integral of our differential equation:

$$\frac{dy}{dt} = f(t,y) \qquad \Rightarrow \qquad y(t) = \int f(t,y)dt \tag{15.32}$$

$$\Rightarrow \qquad y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t,y)dt \tag{15.33}$$

Listing 15.1: `rk2.java` solves an ODE with RHS given by the method `f()` using a second-order Runge–Kutta algorithm. Note that the method `f()`, which you will need to change for each problem, is kept separate from the algorithm, which is best not to change.

```
// rk2.java:  Runge–Kutta 2nd order  ODE solver

import java.io.*;

public class Rk2  {

  public static void main(String[] argv)
                        throws IOException, FileNotFoundException {

    PrintWriter w =                                        // Output file
            new PrintWriter(new FileOutputStream("rk2.dat"), true);

    double h, t, a = 0., b = 10. ;                          // Endpoints
    double y[] = new double[2], ydumb[] = new double[2];
    double fReturn[] = new double[2];
    double k1[] = new double[2];
    double k2[] = new double[2];
    int i, n=100;

    y[0] = 3. ;   y[1] = -5. ;                              // Initialize
    h = (b-a)/n;
    t = a;
    System.out.println("rk2  t="+t+" , x = " +y[0]+ ", v = " + y[1]);
    w.println(t + " " + y[0] + " " + y[1]);              // Output to file
                                                         // Loop over time
    while (t < b)  {
      if ( (t + h) > b ) h = b - t;                       // The last step
      f(t, y, fReturn);            // Evaluate RHS's and return fReturn
      k1[0] = h*fReturn[0];                   // Compute function values
      k1[1] = h*fReturn[1];
      for ( i=0; i <= 1; i++ ) ydumb[i] = y[i] + k1[i]/2;
      f(t + h/2, ydumb, fReturn);
      k2[0] = h*fReturn[0];
      k2[1] = h*fReturn[1];
      for ( i=0; i <= 1; i++ ) y[i] = y[i] + k2[i];
      t = t + h;
      System.out.println("rk2  t=" +t+ " , x = "+y[0]+", v = "+y[1]);
      w.println(t + " " + y[0] + " " + y[1]);            // Output to file
    }                                                    // End while loop
  }
                                        // RHS FUNCTION of your choice
                                           here
  public static void f(double t, double y[], double fReturn[])
```

```
    { fReturn[0] = y[1];                          // RHS of first equation
      fReturn[1] = -100*y[0]-2*y[1] + 10*Math.sin(3*t); }
}
```

To derive the second-order algorithm (rk2 Listing 15.1), we expand $f(t,y)$ in a Taylor series about the *midpoint* of the integration interval and retain two terms:

$$f(t,y) \simeq f(t_{n+1/2}, y_{n+1/2}) + (t - t_{n+1/2})\frac{df}{dt}(t_{n+1/2}) + \mathcal{O}(h^2) \qquad (15.34)$$

Because $(t - t_{n+1/2})$ to any odd power is equally positive and negative over the interval $t_n \le t \le t_{n+1}$, the integral of $(t - t_{n+1/2})$ vanishes in (15.33), and we have our algorithm:

$$\int f(t,y)dt \simeq f(t_{n+1/2}, y_{n+1/2})h + \mathcal{O}(h^3) \qquad (15.35)$$

$$\Rightarrow \qquad y_{n+1} \simeq y_n + hf(t_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^3) \quad \text{(rk2)} \qquad (15.36)$$

So, while `rk2` contains the same number of terms as Euler's rule, it obtains a higher level of precision by taking advantage of the cancellation of the $\mathcal{O}(h)$ term (likewise, `rk4` has the integral of the $(t - t_{n+1/2})$ and $(t - t_{n+1/2})^3$ terms vanish). Yet the price for improved precision is having to evaluate the derivative function and y at the intermediate time $t = t_n + h/2$. And there is the rub, for we cannot use this same algorithm to determine $y_{n+1/2}$. The way out of this quandary is to use Euler's algorithm for $y_{n+1/2}$:

$$y_{n+1/2} \simeq y_n + \tfrac{1}{2}h\frac{dy}{dt} = y_n + \tfrac{1}{2}hf(t_n, y_n) \qquad (15.37)$$

Putting the pieces all together gives the complete `rk2` algorithm:

$$\mathbf{y}_{n+1} \simeq \mathbf{y}_n + \mathbf{k}_2 \qquad \text{(rk2)} \qquad (15.38)$$

$$\mathbf{k}_2 = h\mathbf{f}(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2}) \qquad \mathbf{k}_1 = h\,\mathbf{f}(t_n, \mathbf{y}_n) \qquad (15.39)$$

where we use bold font to indicate the vector nature of y and f. We see that the known derivative function $\mathbf{f}$ is evaluated at the ends and midpoint of the interval, but that only the (known) initial value of the dependent variable $\mathbf{y}$ is required. This makes the algorithm self-starting.

As an example of the use of `rk2`, we apply it to solve the ODE for our spring problem:

$$y_1^{(0)} = y_0^{(0)} + hf^{(0)}(\frac{h}{2}, y_0^{(0)} + k_1) \simeq x_0 + h[v_0 + \frac{h}{2}F_k(0)]$$

$$y_1^{(1)} = y_0^{(1)} + hf^{(1)}[\frac{h}{2}, y_0 + \frac{h}{2}f(0, y_0)] \simeq v_0 + \frac{h}{m}\left[F_{\text{ext}}(\frac{h}{2}) + F_k(y_0^{(1)} + \frac{k_1}{2})\right].$$

These equations say that the position $y^{(0)}$ changes due to the initial velocity v_0 and the force at time 0, while the velocity changes due to the external force at $t = h/2$ and the internal force at two intermediate positions. We see that the position now has an h^2 time dependence, which, at last, brings us up to the equation for projectile motion studied in first-year physics.

Listing 15.2: `rk4.java` solves an ODE with RHS given by the method `f()` using a fourth-order Runge–Kutta algorithm. Note that the method `f()`, which you will need to change for each problem, is kept separate from the algorithm, which is best not to change.

```
// rk4.java: 4th order Runge-Kutta ODE Solver for arbitrary y(t)

import java.io.*;

public class RK4 {

  public static void main(String[] argv)
                        throws IOException, FileNotFoundException {

    PrintWriter w =                                    // Output to file
            new PrintWriter(new FileOutputStream("rk4.dat"), true);

    double h, t;
    double ydumb[] = new double[2];
    double y[] = new double[2];
    double fReturn[] = new double[2];
    double k1[] = new double[2];
    double k2[] = new double[2];
    double k3[] = new double[2];
    double k4[] = new double[2];
    double a = 0. ;    double b = 10. ;                // Endpoints
    int i, n = 100;

    y[0] = 3. ;    y[1] = -5. ;                        // Initialize
    h = (b-a)/n;
    t = a;
    System.out.println("rk4_rhl t = "    // Printout for initial step
                    + t + " , x = " + y[0] + ", v = " + y[1]);
    w.println(t + " " + y[0] + " " + y[1]);  // Output answer to file
                                                  // Loop over time
    while (t < b) {
      if ( (t + h) > b ) h = b - t;                    // Last step
      f(t, y, fReturn);          // Evaluate RHS's, return in fReturn
      k1[0] = h*fReturn[0];                  // Compute function values
      k1[1] = h*fReturn[1];
      for ( i=0; i <= 1; i++ )  ydumb[i] = y[i] + k1[i]/2;
      f(t + h/2, ydumb, fReturn);
      k2[0] = h*fReturn[0];
      k2[1] = h*fReturn[1];
      for ( i=0; i <= 1; i++ )  ydumb[i] = y[i] + k2[i]/2;
      f(t + h/2, ydumb, fReturn);
      k3[0] = h*fReturn[0];
      k3[1] = h*fReturn[1];
      for ( i=0; i <= 1; i++ )  ydumb[i] = y[i] + k3[i];
      f(t + h, ydumb, fReturn);
      k4[0] = h*fReturn[0];
```

```
      k4[1] = h*fReturn[1];
      for (i=0;i <= 1; i++)y[i]=y[i]+(k1[i]+2*(k2[i]+k3[i])+k4[i])/6;
      t = t + h;
      System.out.println("in Rk4, t = "                    // Printout
                        + t + " , x = " + y[0] + ", v = " + y[1]);
      w.println(t + " " + y[0] + " " + y[1]);              // File output
    }                                                     // End while loop
  }

                                                  // FUNCTION of your choice here
  public static void f( double t, double y[], double fReturn[] )
    { fReturn[0] = y[1];                                  // RHS 1st eq
      fReturn[1] = -100*y[0]-2*y[1] + 10*Math.sin(3*t); }  // RHS 2nd
}
```

rk4 The fourth-order Runge–Kutta method `rk4` (Listing 15.2) obtains $\mathcal{O}(h^4)$ precision by approximating y as a Taylor series up to h^2 (a parabola) at the midpoint of the interval. This approximation provides an excellent balance of power, precision, and programming simplicity. There are now four gradient (k) terms to evaluate with four subroutine calls needed to provide a better approximation to $f(t, y)$ near the midpoint. This is computationally more expensive than the Euler method, but its precision is much better, and the steps size h can be made larger. Explicitly, `rk4` requires the evaluation of four intermediate slopes, and these are approximated with the Euler algorithm:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \tfrac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{15.40}$$

$$\mathbf{k}_1 = h\mathbf{f}(t_n, \mathbf{y}_n) \qquad \mathbf{k}_2 = h\mathbf{f}(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2})$$

$$\mathbf{k}_3 = h\mathbf{f}(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_2}{2}) \qquad \mathbf{k}_4 = h\mathbf{f}(t_n + h, \mathbf{y}_n + \mathbf{k}_3)$$

Listing 15.3: `rk45.java` solves an ODE with RHS given by the method `f()` using a Runge–Kutta algorithm with adaptive step size that may yield fifth-order precision. Note that the method `f()`, which you will need to change for each problem, is kept separate from the algorithm, which is best not to change.

```
// rk45: Runge-Kutta-Fehlberg adaptive step size ODE solver

import java.io.*;

public class Rk45  {

  public static void main(String[] argv)
                        throws IOException, FileNotFoundException {
    PrintWriter w =
        new PrintWriter(new FileOutputStream("rk45_rhl.dat"), true);
    PrintWriter q =                                   // File output
       new PrintWriter(new FileOutputStream("rk45_exact.dat"), true);

    double h, t, s, s1, hmin, hmax, E, Eexact, error;
    double y[] = new double[2];
```

```
double fReturn[] = new double[2];
double ydumb[] = new double[2];
double err[] = new double[2];
double k1[] = new double[2];
double k2[] = new double[2];
double k3[] = new double[2];
double k4[] = new double[2];
double k5[] = new double[2];
double k6[] = new double[2];
double Tol = 1.0E-8;                                  // Error tolerance
double a = 0., b = 10. ;                                   // Endpoints
int i, j, flops, n = 20;

y[0] = 1. ;   y[1] = 0. ;                                  // Initialize
h = (b-a)/n;                                     // Temp number of steps
hmin = h/64;   hmax = h*64;                     // Min and max step sizes
t = a;
j = 0;
long timeStart = System.nanoTime();
System.out.println("" + timeStart + "");
flops = 0;
Eexact = 0. ;
error = 0. ;
double sum = 0. ;
                                                        // Loop over time
while (t < b)  {
  if ( (t + h) > b ) h = b - t;                            // Last step
  f(t, y, fReturn);     // evaluate both RHS's, return in fReturn
  k1[0] = h*fReturn[0];
  k1[1] = h*fReturn[1];
  for ( i=0; i <= 1; i++ ) ydumb[i] = y[i] + k1[i]/4;
  f(t + h/4, ydumb, fReturn);
  k2[0] = h*fReturn[0];
  k2[1] = h*fReturn[1];
  for (i=0; i <= 1; i++) ydumb[i] = y[i]+3*k1[i]/32 + 9*k2[i]/32;
  f(t + 3*h/8, ydumb, fReturn);
  k3[0] = h*fReturn[0];
  k3[1] = h*fReturn[1];
  for ( i=0; i <= 1; i++ ) ydumb[i] = y[i] + 1932*k1[i]/2197
    -7200*k2[i]/2197. + 7296*k3[i]/2197;
  f(t + 12*h/13, ydumb, fReturn);
  k4[0] = h*fReturn[0];
  k4[1] = h*fReturn[1];
  for ( i=0; i <= 1; i++ ) ydumb[i] = y[i]+439*k1[i]/216 -8*k2[i]
    + 3680*k3[i]/513 -845*k4[i]/4104;
  f(t + h, ydumb, fReturn);
  k5[0] = h*fReturn[0];
  k5[1] = h*fReturn[1];
  for ( i=0; i <= 1; i++ ) ydumb[i] = y[i] -8*k1[i]/27 + 2*k2[i]
    -3544*k3[i]/2565 + 1859*k4[i]/4104 -11*k5[i]/40;
  f(t + h/2, ydumb, fReturn);
  k6[0] = h*fReturn[0];
  k6[1] = h*fReturn[1];
  for ( i=0; i <= 1; i++ ) err[i] = Math.abs( k1[i]/360
    - 128*k3[i]/4275 - 2197*k4[i]/75240 + k5[i]/50. +2*k6[i]/55);
                                                        // Accept step size
  if ( err[0] < Tol || err[1] < Tol || h <= 2*hmin ) {
```

```
        for ( i=0; i <= 1; i++ ) y[i] = y[i] + 25*k1[i]/216.
          + 1408*k3[i]/2565. + 2197*k4[i]/4104. - k5[i]/5.;
        t = t + h;
        j++ ;
      }
      if ( err[0]==0 || err[1]==0 ) s = 0;        // Trap division by 0
      else s = 0.84*Math.pow(Tol*h/err[0], 0.25);
                                                    // Reduce step
      if ( s  <  0.75 && h > 2*hmin )  h /= 2.;
                                                   // Increase step
      else if ( s > 1.5 && 2* h  <  hmax ) h *= 2.;
      flops++ ;
      E = Math.pow(y[0], 6.) + 0.5*y[1]*y[1];
      Eexact = 1. ;
      error = Math.abs((E-Eexact)/Eexact);
      sum += error;
    }
    System.out.println(" < error> =  " + sum/flops);
    System.out.println("flops = " + flops + "");
    long timeFinal = System.nanoTime();
    System.out.println("" + timeFinal + "");
    System.out.println("Nanoseconds = " + (timeFinal-timeStart));
  }

                                        // Enter your own RHS here!
  public static void f(double t, double y[], double fReturn[])  {
    fReturn[0] = y[1];
    fReturn[1] = -6.*Math.pow(y[0], 5.);
    return;
  }
}
```

rk45 (Listing 15.3) A variation of `rk4`, known as the Runge–Kutta–Fehlberg method, or `rk45` [25], automatically doubles the step size and tests to see how an estimate of the error changes. If the error is still within acceptable bounds, the algorithm will continue to use the larger step size and thus speed up the computation; if the error is too large, the algorithm will decrease the step size until an acceptable error is found. As a consequence of the extra information obtained in the testing, the algorithm obtains $\mathcal{O}(h^5)$ precision, but often at the expense of extra computing time. Whether that extra time is recovered by being able to use a larger step size depends upon the application.

15.5.3 Assessment: rk2 v. rk4 v. rk45

While you are free to do as you please, we do *not* recommend that you write your own `rk4` method, unless you are very careful. We will be using `rk4` for some high precision work, and unless you get every fraction and method call just right, your `rk4` may appear to work well, but still not give all the precision that you should have. We do, regardless, recommend that you write your own `rk2`, as doing so will make it clearer as to how the Runge–Kutta