# 15
# Differential Equations Applications

*Part of the fascination of computational physics is that we can solve most any differential equation rather easily. Furthermore, while most traditional (read* analytic*) treatments of oscillations are limited to small displacements about equilibrium, where the restoring forces are* linear*, we will exceed those limits and look at a range of forces and the motion they cause. In Unit I we look at oscillators that may be harmonic for certain parameter values, but then become anharmonic when the oscillations get large, or for other choices of parameters. We let the reader/instructor decide which oscillator (or both) to study. We start off without any external, time-dependent forces, and spend some time examining how well various differential-equation solvers work. We then include time-dependent forces in order to investigate resonances and beats.*

*In Chapter 19 we make a related study of the realistic pendulum, and its chaotic behavior. Some special properties of nonlinear equations are discussed in Chapter 26 on solitons. In Unit II we examine how to solve the* simultaneous *ODEs that arise in projectile and planetary motion.*

## 15.1
## UNIT I. Free Nonlinear Oscillations

**Problems:** In Fig. 15.1 we show a mass $m$ that is attached to a spring that exerts a restoring force toward the origin, as well as a hand that exerts a time-dependent external force on the mass. We are told that the restoring force exerted by the spring is nonharmonic, that is, not simply proportional to displacement from equilibrium, but we are not given details as to how this is nonharmonic. Your **problem** is to solve for the motion of the mass as a function of time. You may assume that the motion is constrained to one dimension.

*Computationyal Physics. Problem Solving with Computers (2nd edn).*
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu
Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim
ISBN: 978-3-527-40626-5

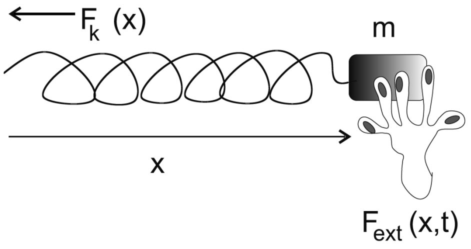

**Fig. 15.1** A mass $m$ attached to a spring with restoring force $F_k(x)$, and with an external agency (hand) subjecting the mass to a time-dependent driving force.

## 15.2 Nonlinear Oscillator (Theory)

This is a problem in classical mechanics for which Newton's second law provides us with the equation of motion

$$F_k(x) + F_{\text{ext}}(x,t) = m\frac{d^2x}{dt^2} \tag{15.1}$$

where $F_k(x)$ is the restoring force exerted by the spring and $F_{\text{ext}}(x,t)$ is the external force. Equation (15.1) is the differential equation we must solve for arbitrary forces. Because we are not told just how the springs depart from being linear, we are free to try out some different models. As our first model, we try a potential that is a harmonic oscillator for small displacements $x$, but also contains a perturbation that introduces a nonlinear term to the force for large $x$ values:

$$V(x) \simeq \tfrac{1}{2}kx^2\left(1 - \tfrac{2}{3}\alpha x\right) \tag{15.2}$$

$$\Rightarrow \quad F_k(x) = -\frac{dV(x)}{dx} = -kx(1-\alpha x) = m\frac{d^2x}{dt^2} \tag{15.3}$$

where we have left off the time-dependent external force. This is the second-order ODE we need to solve. If $\alpha x \ll 1$, we should have essentially harmonic motion.

We can understand the basic physics of this model by looking at Fig. 15.2. As long as $x < 1/\alpha$, there will be a *restoring force* and the motion will be periodic, even though it is only harmonic (linear) for small-amplitude oscillations. Yet, as the amplitude of oscillation gets larger, there will be an asymmetry in the motion to the right and left of the equilibrium position. And if $x > 1/\alpha$, the force becomes repulsive and the mass "rolls" down the potential hill.

As a second model of a nonlinear oscillator, we assume that the spring's potential function is proportional to some arbitrary, *even* power $p$ of the displacement $x$ from equilibrium:

$$V(x) = \frac{1}{p}kx^p \qquad (p \text{ even}) \tag{15.4}$$

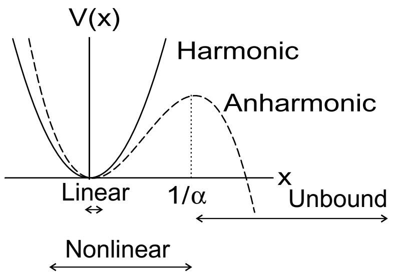

**Fig. 15.2** The potential of an harmonic oscillator (solid curve) and of an oscillator with an anharmonic correction (dashed curve).

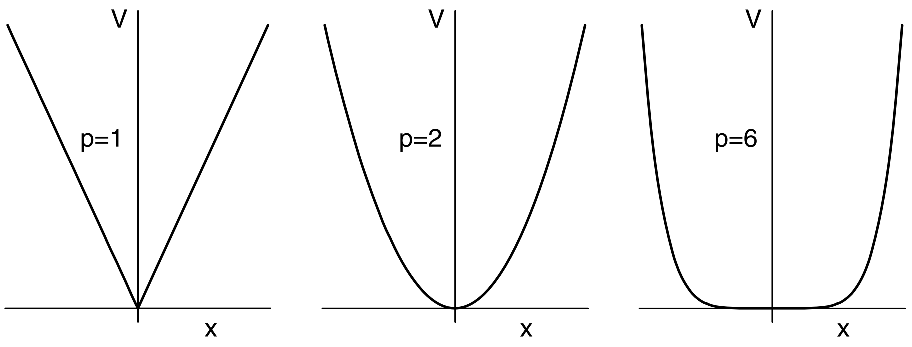

**Fig. 15.3** The shape of the potential energy function $V(x) \propto |x|^p$ for different $p$ values. The linear and nonlinear labels refer to restoring force derived from these potentials.

We require an even $p$ to ensure that the force,

$$F_k(x) = -\frac{dV(x)}{dx} = -kx^{p-1} \tag{15.5}$$

contains an odd power of $p$, which guarantees that it is a *restoring* force even for negative $x$. We display the characteristics of this potential in Fig. 15.3 for $p = 2$, the harmonic oscillator, and for $p = 6$, which is nearly a square well with the mass moving almost freely until it hits the wall at $x \approx \pm 1$. Regardless of the $p$ value, the motion will be periodic, but it will be harmonic only for $p = 2$. Newton's law (15.1) gives the second-order ODE we need to solve

$$F_{\text{ext}}(x,t) - kx^{p-1} = m\frac{d^2x}{dt^2} \tag{15.6}$$

## 15.3 Math: Types of Differential Equations

*The background material in this section is presented to avoid confusion over semantics. The well-versed student may want to skim or skip it.*

- **Order:** A general form for a *first-order* differential equation is

$$\frac{dy}{dt} = f(t,y) \tag{15.7}$$

where the "order" refers to the degree of the derivative on the LHS. The derivative or force function $f(t,y)$ on the RHS, is arbitrary. For instance, even if $f(t,y)$ is a nasty function of $y$ and $t$ such as

$$\frac{dy}{dt} = -3t^2y + t^9 + y^7 \tag{15.8}$$

this is still first order in the derivative. A general form for a *second-order* differential equation is

$$\frac{d^2y}{dt^2} + \lambda\frac{dy}{dt} = f(t, \frac{dy}{dt}, y) \tag{15.9}$$

The derivative function $f$ on the RHS is arbitrary and may involve any power of the first derivative as well. To illustrate,

$$\frac{d^2y}{dt^2} + \lambda\frac{dy}{dt} = -3t^2\left(\frac{dy}{dt}\right)^4 + t^9y(t) \tag{15.10}$$

is a second-order differential equation as is Newton's law (15.1).

In the differential equations (15.7) and (15.9), the time $t$ is the *independent* variable and the position $y$ is the *dependent* variable. This means that we are free to vary the time at which we want a solution, but not the value of the solution $y$ at that time. Note that we usually use the symbol $y$ or $Y$ for the dependent variable, but that this is just a symbol. In some applications we use $y$ to describe a position that is an independent variable instead of $t$.

- **Ordinary and Partial:** Differential equations such as (15.1) and (15.7) are *ordinary* differential equations because they contain only *one* independent variable, in these cases $t$. In contrast, an equation such as the Schrödinger equation,

$$i\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{1}{2m}\left[\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} + \frac{\partial^2\psi}{\partial z^2}\right] + V(\mathbf{x})\psi(\mathbf{x},t) \tag{15.11}$$

(where we have set $\hbar = 1$) contain several independent variables, and this makes it a *partial differential equations (PDE)*. The partial derivative symbol $\partial$ is used to indicate that the dependent variable $\psi$ depends simultaneously on several independent variables. In the early parts of this book we limit ourselves to ordinary differential equations (ODEs). In Chaps. 23–27 we examine a variety of partial differential equations (PDEs).

- **Linear and Nonlinear:** Part of the liberation of computational science is that we are no longer limited to solving *linear equations*. A *linear* equation is one in which only the first power of $y$ or $d^n y/d^n t$ appears; a *nonlinear* equation may contain higher powers. For example

$$\frac{dy}{dt} = g^3(t)y(t) \qquad \text{(linear)} \qquad (15.12)$$

$$\frac{dy}{dt} = \lambda y(t) - \lambda^2 y^2(t) \qquad \text{(nonlinear)} \qquad (15.13)$$

An important property of linear equations is the *law of linear superposition* which lets us add solutions together to form new ones. As a case in point, if $A(t)$ and $B(t)$ are solutions of the linear equation in (15.12), then

$$y(t) = \alpha\, A(t) + \beta\, B(t) \qquad (15.14)$$

is also a solution for arbitrary values of the constants $\alpha$ and $\beta$. In contrast, even if we were clever enough to guess that the solution of the nonlinear equation (15.12) is

$$y(t) = \frac{a}{1 + be^{-\lambda t}} \qquad (15.15)$$

(which you can verify by substitution), we would go amiss if we tried to obtain a more general solution by adding together two such solutions

$$y_1(t) = \frac{a}{1 + be^{-\lambda t}} + \frac{a'}{1 + b'e^{-\lambda t}} \qquad (15.16)$$

(which you can confirm by substitution).

- **Initial and Boundary Conditions:** The general solution of a first-order differential equation always contains one arbitrary constant. A general solution of a second-order differential equation contains two such constants, and so forth. For any specific problem, these constants are fixed by the *initial conditions*. For a first-order equation the sole initial condition is usually the position $y(t)$ at some time. For a second-order equation the two initial conditions are usually position and velocity at some time. Regardless of how powerful a computer you use, the mathematical fact still remains and you must know the initial conditions in order to solve the problem.

In addition to initial conditions, it is possible to further restrict solutions of differential equations. One such way is by *boundary conditions* or *values* that constrain the solution to have fixed values at the boundaries of the solution space. Problems of this sort are called *eigenvalue problems*. They are so demanding that solutions do not always exist, and even when they do exist, they may require a trial-and-error *search* to find them. In Unit II on ODE eigenvalues, we discuss how to extend the techniques of the present unit to boundary-value problems.

## 15.4 Dynamical Form for ODEs (Theory)

A standard form for ODEs, which has found use both in numerical analysis [9] and in classical dynamics [22–24], is to express ODEs of *any order* as $N$ simultaneous, first-order ODEs:

$$\frac{dy^{(0)}}{dt} = f^{(0)}(t, y^{(i)})$$

$$\frac{dy^{(1)}}{dt} = f^{(1)}(t, y^{(i)}) \tag{15.17}$$

$$\ddots \tag{15.18}$$

$$\frac{dy^{(N-1)}}{dt} = f^{(N-1)}(t, y^{(i)}) \tag{15.19}$$

where the $y^{(i)}$ dependence of $f$ indicates that it may depend on all the components of $y$, but not on the derivatives $dy^{(i)}/dt$. These equations can be expressed more compactly by use of the $N$-dimensional vectors $\mathbf{y}$ and $\mathbf{f}$:

$$d\mathbf{y}(t)/dt = \mathbf{f}(t, \mathbf{y}) \tag{15.20}$$

$$\mathbf{y} = \begin{pmatrix} y^{(0)}(t) \\ y^{(1)}(t) \\ \ddots \\ y^{(N-1)}(t) \end{pmatrix} \qquad \mathbf{f} = \begin{pmatrix} f^{(0)}(t, \mathbf{y}) \\ f^{(1)}(t, \mathbf{y}) \\ \ddots \\ f^{(N-1)}(t, \mathbf{y}) \end{pmatrix}$$

The utility of such compact notation is that we can study the properties of the ODEs, as well as develop algorithms to solve them, by dealing with the single equation (15.20) without having to worry about the individual equations. To see how this works, let us express a second-order differential equation, namely, Newton's law,

$$\frac{d^2x}{dt^2} = \frac{1}{m} F\left(t, \frac{dx}{dt}, x\right) \tag{15.21}$$

in the standard dynamical form (15.20). The rule is that the RHS may not contain any explicit derivatives, although individual components of $y^{(i)}$ may represent derivatives. To pull this off, we define the position $x$ as the dependent variable $y^{(0)}$ and the velocity $dx/dt$ as the dependent variable $y^{(1)}$:

$$y^{(0)}(t) \overset{\text{def}}{=} x(t) \qquad y^{(1)}(t) \overset{\text{def}}{=} \frac{dx}{dt} = \frac{dy^{(0)}}{dt} \tag{15.22}$$

The second-order ODE (15.21) is now two simultaneous, first-order ODEs,

$$\frac{dy^{(0)}}{dt} = y^{(1)}(t) \qquad \frac{dy^{(1)}}{dt} = \frac{1}{m} F(t, y^{(0)}, y^{(1)}) \tag{15.23}$$

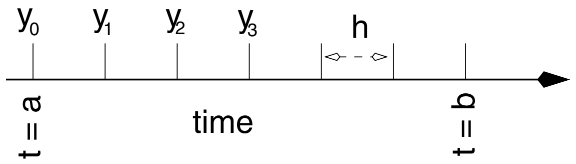

**Fig. 15.4** The steps of length $h$ taken in solving a differential equation. The solution starts at time $t = a$ and is integrated to $t = b$.

This expresses the acceleration (the second derivative in (15.21)) as the first derivative of the velocity [$y^{(2)}$]. These equations are now in the standard form (15.20) with the derivative or force function $\mathbf{f}$ having the two components

$$f^{(0)} = y^{(1)}(t) \qquad f^{(1)} = \frac{1}{m}F(t, y^{(0)}, y^{(1)}) \tag{15.24}$$

where $F$ may be an explicit function of time, as well as of position and velocity.

To be even more specific, we apply these definitions to our spring problem (15.6) to obtain the coupled first-order equations:

$$\frac{dy^{(0)}}{dt} = y^{(1)}(t) \qquad \frac{dy^{(1)}}{dt} = \frac{1}{m}\left[F_{\text{ext}}(x,t) - ky^{(0)}(t)^{p-1}\right] \tag{15.25}$$

where $y^{(0)}(t)$ is the position of the mass at time $t$, and $y^{(1)}(t)$ is its velocity. In the standard form, the components of the force/derivative function, and the initial conditions are

$$\begin{aligned} f^{(0)}(t,\mathbf{y}) &= y^{(1)}(t) \qquad & f^{(1)}(t,\mathbf{y}) &= \frac{1}{m}\left[F_{\text{ext}}(x,t) - k(y^{(0)})^{p-1}\right] \\ y^{(0)}(0) &= x_0 \qquad & y^{(1)}(0) &= v_0 \end{aligned} \tag{15.26}$$

Breaking a second-order differential equation into two first-order ones is not just an arcane mathematical maneuver. In classical dynamics it occurs when transforming the single Newtonian equation of motion involving position and acceleration, (15.1), into two *Hamiltonian* equations involving position and momentum:

$$\frac{dp_i}{dt} = F_i \qquad m\frac{dy_i}{dt} = p_i \tag{15.27}$$

## 15.5 ODE Algorithms

The classic way to solve a differential equation is to start with the known initial value of the dependent variable, $y_0 \equiv y(t = 0)$, and then use the derivative

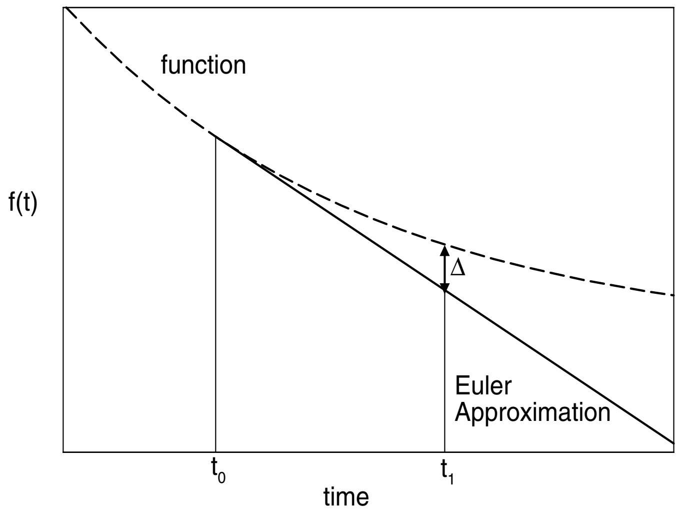

**Fig. 15.5** Euler's algorithm for the forward integration of a differential equation by one time step. The linear extrapolation is seen to cause the error Δ.

function $f(t, y)$ to find an approximate value for $y$ at a small step $\Delta t = h$ forward in time, that is, $y(t = h) \equiv y_1$. Once you can do that, you solve the ODE for all $t$ values by just continuing stepping to large times, one $h$ at a time (Fig. 15.4).[1]

It is simplest if the time steps used throughout the integration remain constant in size, and that is mostly what we shall do. Industrial-strength algorithms, such as the one we discuss in Section 15.5.2, adapt the step size by making $h$ larger in those regions where $y$ varies slowly (this speeds up the integration and cuts down on the roundoff error), and make $h$ smaller in those regions where $y$ varies rapidly (this provides better precision).

Error is always a concern when integrating differential equations because derivatives require small differences, and small differences are prone to subtractive cancellations. In addition, because our stepping procedure for solving the differential equation is a continuous extrapolation of the initial conditions with each step building on a previous extrapolation, this is somewhat like a castle built on sand; in contrast to interpolation, there are no tabulated values on which to anchor your solution.

[1] To avoid confusion, notice that $y^{(n)}$ is the $n$th component of the $y$ vector, while $y_n$ is the value of $y$ after $n$ time steps. (Yes, there is a price to pay for elegance in notation.)

### 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 $y_n \overset{\text{def}}{=} y(t_n)$ 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.

```java
// 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.

```java
// 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 methods work, but without all the pain. We do give sample `rk2`, `rk4` and `rk45` codes on the CD, and list the latter two in Listings 15.2 and 15.3.

1. Write your own `rk2` method. Design your method for a general ODE; this means making the derivative function $f(t, x)$ a separate method.

2. Use your `rk2` method as the basis for a program that solves the equation of motion (15.6) or (15.25). Be sure to use double precision to help control subtractive cancellation. Have your program output and plot both the position $x(t)$ and velocity $dx/dt$ as functions of time.

3. Once your ODE solver compiles and executes, do a number of things to check that it is working well and to help you pick a reasonable value for the step size $h$.

   (a) Adjust the parameters in your potential so that it corresponds to a pure harmonic oscillator (set $p = 2$ or $\alpha = 0$). For this case we have an analytic result to compare with:

   $$x(t) = A\sin(\omega_0 t + \phi) \qquad v(t) = \omega_0 A\cos(\omega_0 t + \phi) \qquad \omega_0 = \sqrt{k/m}.$$

   (b) Pick values of $k$ and $m$ such that the period $T = 2\pi/\omega$ is a nice number with which to work (something like $T$ =1).

   (c) Start with a step size $h \simeq T/5$, and make $h$ smaller and smaller until the solution looks smooth, has a period that remains constant for a large number of periods, and which agrees with the analytic result. (As a general rule of thumb, we suggest that you start with $h \simeq T/100$, where $T$ is a characteristic time for the problem at hand. Here we want you to start with a large $h$ so that you can see how the solution improves.)

   (d) Make sure that you have exactly the same initial conditions for the analytic and numeric solutions (zero displacement, nonzero velocity), and then plot the two together. It is good if you cannot tell them apart, yet that only ensures that there is $\sim$ 2 places of agreement.

   (e) Try different initial velocities, and verify that a *harmonic* oscillator is *isochronous*, that is, its period does *not* change as the amplitude varies.

4. Now that you know you can get a good solution of an ODE with `rk2`, **compare** the solutions obtained with the **rk2, rk4**, and **rk45** solvers. You will find methods for all three on the CD, or you can try writing your own (but recall our caveat).

**Tab. 15.1** Comparison of ODE solvers for different equations.

| Equation | Method | Initial $h$ | No. of flops | Time (ms) | Relative error |
|---|---|---|---|---|---|
| (15.41) | rk4 | 0.01 | 1000 | 5.2 | $2.2 \times 10^{-8}$ |
| | rk45 | 1.00 | 72 | 1.5 | $1.8 \times 10^{-8}$ |
| (15.42) | rk4 | 0.01 | 227 | 8.9 | $1.8 \times 10^{-8}$ |
| | rk45 | 0.1 | 3143 | 36.7 | $5.7 \times 10^{-11}$ |

5. Make a table of your comparisons similar to Tab. 15.1. There we compare `rk4` and `rk45` for the two equations,

$$2y\,y'' + y^2 - y'^2 = 0 \tag{15.41}$$

$$y'' + 6y^5 = 0 \tag{15.42}$$

   with initial conditions $(y(0), y'(0)) = (1, 1)$. Equation (15.41) yields oscillations with variable frequency, and has an analytic solution with which to compare. Equation (15.42) corresponds to our standard potential (15.4), with $p = 6$. Although we have not tuned `rk45`, the table shows that by setting its tolerance parameter to a small enough number, `rk45` will obtain better precision than `rk4`, but that it requires $\sim 10$ times more floating-point operations, and takes $\sim 5$ times longer.

## 15.6 Solution for Nonlinear Oscillations (Assessment)

Use your `rk4` program to study anharmonic oscillations by trying powers in the range $p = 2$–$12$, or anharmonic strengths in the range $0 \leq \alpha x \leq 2$. Do *not* include any time-dependent forces yet. Note that for large values of $p$ you may need to decrease the step size $h$ from the value used for the harmonic oscillator because the forces and accelerations get large near the turning points.

1. Check that, regardless of how nonlinear you make the force, the solution remains periodic with constant amplitude and period, for a given initial condition and value of $p$ or $\alpha$. In particular, check that the maximum speed occurs at $x = 0$ and that the minimum speed occurs at maximum $x$. This is all just a consequence of energy conservation.

2. Verify that different initial conditions do indeed lead to different periods (a *nonisochronous* oscillator).

3. Why do the shapes of the oscillations change for different $p$'s or $\alpha$'s?

4. Devise an algorithm to determine the period of the oscillation by recording times at which the mass passes through the origin. Note that, be-

cause the motion may be asymmetric, you must record at least four times.

5. Construct a graph of the deduced period as a function of initial energy.
6. Verify that as the initial energy approaches $k/6\alpha^2$, or as $p$ gets large, the motion is oscillatory but not harmonic.
7. Verify that at $E = k/6\alpha^2$, the motion of the oscillator changes from oscillatory to translational. See how close you can get to this *separatrix*, and verify that at this energy a single oscillation takes an infinite time. (There is no separatrix for the power-law potential.)

### 15.6.1
### Precision Assessment: Energy Conservation

We have not explicitly built energy conservation into our ODE solvers. Nonetheless, unless you have explicitly included a frictional force, it follows from Newton's laws that energy must be a constant for all values of $p$ or $\alpha$. That being so, the constancy of energy in your numerical solution is a demanding test.

1. Plot the potential energy $\mathrm{PE}(t) = V[x(t)]$, the kinetic energy $\mathrm{KE}(t) = mv^2(t)/2$, and the total energy $E(t) = \mathrm{KE}(t) + \mathrm{PE}(t)$, for hundreds of periods. Comment on the correlation between $\mathrm{PE}(t)$ and $\mathrm{KE}(t)$, and how it depends on the potential's parameters.
2. Check the long-term *stability* of your solution by plotting

$$-\log_{10}\left|\frac{E(t) - E(t=0)}{E(t=0)}\right| \simeq \text{number places precision}$$

   for a large number of periods. Because $E$ should be independent of time, the numerator is the absolute error in your solution, and when divided by $E(0)$, becomes the relative error (say $10^{-11}$). That being the case, $-\log_{10}$ of the relative error gives you the approximate number of decimal places of precision in your solution. If you cannot achieve 11 or more places, then you need to decrease the value of $h$ or look for bugs in your program.
3. Because a particle bound by a large $p$ oscillator is essentially "free" most of the time, you should observe that the average of its kinetic energy over time exceeds its average potential energy. This is actually a physical explanation of the Virial theorem for a power-law potential:

$$\langle \mathrm{KE} \rangle = \frac{p}{2}\langle \mathrm{PE} \rangle \tag{15.43}$$

Verify that your solution satisfies the Viral theorem. (Those readers who have worked on the perturbed oscillator problem can use this relation to deduce an effective $p$ value, which should be between 2 and 3.)

## 15.7 Extensions: Nonlinear Resonances, Beats and Friction

**Problem:** So far, our oscillations have been rather simple. We have ignored friction and assumed that there are no external forces (hands) to influence the system's natural oscillations. Determine

1. how the oscillations change when friction is included,
2. how the resonances and beats of nonlinear oscillators differ from those of linear oscillators, and
3. how introducing friction affects resonances.

### 15.7.1 Friction: Model and Implementation

The real world is full of friction, and it is not all bad. For while it may make it harder to pedal a bike through the wind, it also tends to stabilize oscillations. The simplest models for friction are *static*, *kinetic or sliding*, and *viscous* friction:

$$F_f^{(\text{static})} \leq -\mu_s N \qquad F_f^{(\text{kinetic})} = -\mu_k N \frac{v}{|v|} \qquad F_f^{(\text{viscous})} = -bv \tag{15.44}$$

Here $N$ is the *normal force*, $\mu$ and $b$ are parameters, and $v$ is the velocity. This model for static friction is clearly appropriate for objects at rest, while the model for kinetic friction is most appropriate for an object sliding on a dry surface. If the surface is lubricated, or if the object is moving through a viscous medium (like air), then a frictional force proportional to velocity is a better model.

1. Modify your code for harmonic oscillations to include the three types of friction modeled in (15.44), and observe how the motion changes from the frictionless case. Note that this means there should be *two* separate simulations, one including static plus kinetic friction, and another for viscous friction.
2. *Hint:* For the simulation with static and kinetic friction, each time the oscillator has $v = 0$ you need to check that the restoring force exceeds the static force of friction. If not, the oscillation must end at that instant. Check that your simulation terminates at nonzero $x$ values.

3. For your simulations with viscous friction, investigate the qualitative changes that occur for increasing $b$ values:

| | | |
|---|---|---|
| **Underdamped** | $b < 2m\omega_0$ | Oscillation within an decaying envelope |
| **Critically damped** | $b = 2m\omega_0$ | Nonoscillatory, finite-time decay |
| **Over damped** | $b > 2m\omega_0$ | Nonoscillatory, infinite-time decay. |

### 15.7.2
### Resonances and Beats: Model and Implementation

All stable physical systems will oscillate if displaced slightly from their rest positions. The frequency $\omega_0$ with which such a system executes small oscillations about its rest positions is called its *natural frequency*. If an external, sinusoidal force is applied to this system, and if the frequency of the external force equals $\omega_0$, then a *resonance* may occur in which the oscillator absorbs energy from the external force, and the amplitude of oscillation increases with time. If the oscillator and the driving force remain in phase over time, then the amplitude will continue to grow, unless there is some mechanism, such as friction or nonlinearities, to limit the growth.

If the frequency of the driving force is close to the natural frequency of the oscillator, then a related phenomenon, known as *beating*, may occur. In this situation, the oscillator acquires an additional amplitude due to the external force which is slightly out of phase with the natural vibration of the oscillator. There then results either constructive or destructive interference between the two oscillations. If the frequency of the driver is very close to the natural frequency, then the resulting motion,

$$x \simeq x_0 \sin \omega t + x_0 \sin \omega_0 t = \left(2x_0 \cos \tfrac{\omega-\omega_0}{2} t\right) \sin \tfrac{\omega+\omega_0}{2} t \tag{15.45}$$

looks like the natural vibration of the oscillator at the average frequency $\frac{\omega+\omega_0}{2}$, yet with an amplitude $2x_0 \cos \frac{\omega-\omega_0}{2} t$ that varies with the slow *beat frequency* $\frac{\omega-\omega_0}{2}$.

## 15.8
## Implementation: Inclusion of Time-Dependent Force

To extend our simulation to include an external force,

$$F_{\text{ext}}(t) = F_0 \sin \omega t,$$

we need a force function $\mathbf{f}(t, \mathbf{y})$ with explicit time dependence.

1. Add the time-dependent external force to the space-dependent restoring force in your program (do not include friction yet).

2. Start by using a very large value for the magnitude of the driving force $F_0$. This should lead to *mode locking* (the 500-pound gorilla effect), where the system is overwhelmed by the driving force, and, after the transients die out, the system oscillates in phase with the driver regardless of its frequency.

3. Now lower $F_0$ until it is close to the magnitude of the natural restoring force of the system. You need to have this near equality for beating to occur.

4. For a harmonic system, verify that the "beat frequency," that is, the number of variations in intensity per unit time, equals the frequency difference $(\omega - \omega_0)/2\pi$ in cycles per second. You will be able to do this only if $\omega$ and $\omega_0$ are close.

5. Once you have a value for $F_0$ matched well with your system, make a series of runs in which you progressively increase the frequency of the driving force for the range $\omega_0/10 \leq \omega \leq 10\omega_0$.

6. Make of plot of the maximum amplitude of oscillation that occurs as a function of the frequency of the driving force.

7. Explore what happens when you make both linear and nonlinear systems resonate. If the nonlinear system is close to being harmonic, you should get beating instead of the blowup that occurs for the linear system. Beating occurs because the natural frequency changes as the amplitude increases, and thus the natural and forced oscillations fall out of phase. Yet once out of phase, the external force stops feeding energy into the system, and amplitude decreases. Yet with the decrease in amplitude, the frequency of the oscillator returns to its natural frequency, the driver and oscillator get back into phase, and the entire cycle repeats.

8. Investigate now, how the inclusion of viscous friction modifies the curve of amplitude versus driving-force frequency. You should find that friction broadens the curve.

9. Notice how the character of the resonance changes as the exponent $p$ in the potential $V(x) = k|x|^p/p$ is made larger and larger. At large $p$, the mass effectively "hits" the wall and falls out of phase with the driver.

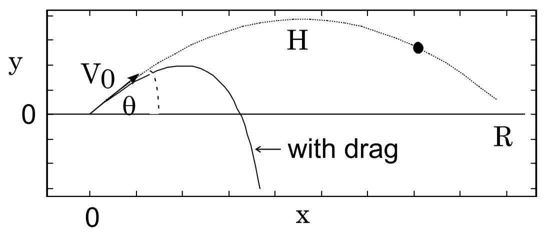

**Fig. 15.6** The trajectory of a projectile fired with initial velocity $V_0$ in the $\theta$ direction. The solid curve includes air resistance.

## 15.9 UNIT II. Balls, not Planets, Fall Out of the Sky

**Problem:** Golf players and baseball outfielders claim that hit balls appear to fall straight down out of the sky at the end of their trajectories (the solid curve in Fig. 15.6). Your **problem** is to determine whether there is a simple physics explanation for this effect, or whether it is "all in the mind's eye". And while you are wondering why things fall out of the sky, see if you can use your new-found numerical tools to explain why planets do not fall out of the sky.

*There are two points to this problem. First, the physics of the motion of projectiles with drag and of planets are interesting and full of surprises. Second, these topics give us the opportunity to extend our ODE solver to two dimensions, where there may be more work for the computer, and there is little more work for us.*

## 15.10 Theory: Projectile Motion with Drag

Figure 15.6 shows the initial velocity $V_0$ and inclination $\theta$ for a projectile. If we ignore air resistance, the projectile has only the force of gravity acting on it and therefore has a constant acceleration $g = 9.8\,\mathrm{m/s^2}$ in the negative $y$ direction. The analytic solutions to the equations of motion are

$$x(t) = V_{0x}t \qquad y(t) = V_{0y}t - \tfrac{1}{2}gt^2 \tag{15.46}$$

$$v_x(t) = V_{0x} \qquad v_y(t) = V_{0y} - gt \tag{15.47}$$

where $(V_{0x}, V_{0y}) = V_0(\cos\theta, \sin\theta)$. Solving for $t$ as a function of $x$, and substituting it into the $y(t)$ equation, shows that the trajectory is a parabola:

$$y = \frac{V_{0y}}{V_{0x}}x - \frac{g}{2V_{0x}^2}x^2 \tag{15.48}$$

Likewise, it is easy to show (dotted curve in Fig. 15.6) that without friction the range $R = 2V_0^2 \sin\theta\cos\theta/g$, and the maximum height $H = V_0^2\sin^2\theta/(2g)$.

The parabola of frictionless motion is symmetric about its midpoint, and so does not describe a ball dropping out of the sky. We want to determine if the inclusion of air resistance leads to trajectories that are much steeper at their ends than at their beginnings (solid curve in Fig. 15.6). The basic physics is Newton's second law in two dimensions for a frictional force $\mathbf{f}^{(f)}$ opposing motion, and a vertical gravitational force $-mg\hat{\mathbf{e}}_y$:

$$\mathbf{f}^{(f)} - mg\hat{\mathbf{e}}_y = m\frac{d^2\mathbf{x}(t)}{dt^2} \tag{15.49}$$

$$\Rightarrow \qquad f_x^{(f)} = m\frac{d^2x}{dt^2} \qquad\qquad f_y^{(f)} - mg = m\frac{d^2y}{dt^2}. \tag{15.50}$$

where the bold symbols indicate vector quantities.

The force of friction $\mathbf{f}^{(f)}$ is not a basic force of nature with a universal form, but rather, it is an approximate model of the physics of viscous flow, with no one expression being accurate for all velocities. We know it always opposes motion, which means it is in a direction opposite to that of the velocity. A simple model for air resistance assumes that the frictional force is proportional to some power $n$ of the projectile's speed [26,27]:

$$\mathbf{f}^{(f)} = -k\,m\,|v|^n\,\frac{\mathbf{v}}{|v|} \tag{15.51}$$

where the $-\mathbf{v}/|v|$ factor ensures that the frictional force is always in a direction opposite to that of the velocity. (If $n$ is odd, then we may leave off the last factor; however, even $n$ requires it to ensure the correct sign.) Though a frictional force proportional to a power of the velocity is more accurate of nature than a constant force, it is still a simplification. Indeed, physical measurements indicate that the power $n$ varies with velocity, and so the most accurate model would be a numerical one that uses the empirical velocity dependence $n(v)$.

With a power law for friction, the equations of motion are

$$\frac{d^2x}{dt^2} = -k\,v_x^n\,\frac{v_x}{|v|} \qquad \frac{d^2y}{dt^2} = -g - k\,v_y^n\,\frac{v_y}{|v|} \qquad |v| = \sqrt{v_x^2 + v_y^2} \tag{15.52}$$

We shall consider three values for $n$, each of which represents a different model for the air resistance: (1) $n = 1$ for low velocities; (2) $n = 3/2$, for medium velocities; and (3) $n = 2$ for high velocities.

## 15.10.1 Simultaneous Second Order ODEs

Even though (15.52) are simultaneous, second-order ODEs, we can still use our regular ODE solver on them after expressing them in standard form

$$\frac{d\mathbf{Y}}{dt} = \mathbf{F}(t, \mathbf{Y}) \qquad \text{(standard form)} \tag{15.53}$$

where we use uppercase $\mathbf{Y}$ and $\mathbf{F}$ to avoid confusion with the coordinates and frictional force. We pick $\mathbf{Y}$ to be the 4D vector of dependent variables:

$$Y^{(0)} = x(t) \qquad Y^{(1)} = \frac{dx}{dt} \qquad Y^{(2)} = y(t) \qquad Y^{(3)} = \frac{dy}{dt} \tag{15.54}$$

We express the equations of motion in terms of $\mathbf{Y}$ to obtain the standard form:

$$\frac{dY^{(0)}}{dt}\left(\equiv \frac{dx}{dt}\right) = Y^{(1)} \qquad \frac{dY^{(1)}}{dt}\left(\equiv \frac{d^2x}{dt^2}\right) = \frac{1}{m}f_x^{(f)}(\mathbf{y})$$

$$\frac{dY^{(2)}}{dt}\left(\equiv \frac{dy}{dt}\right) = Y^{(3)} \qquad \frac{dY^{(3)}}{dt}\left(\equiv \frac{d^2y}{dt^2}\right) = \frac{1}{m}f_y^{(f)}(\mathbf{y}) - g$$

And now we just read off the components of the force function $\mathbf{F}(t, \mathbf{Y})$:

$$F^{(0)} = Y^{(1)} \qquad F^{(1)} = \frac{1}{m}f_x^{(f)} \qquad F^{(2)} = Y^{(3)} \qquad F^{(3)} = \frac{1}{m}f_y^{(f)} - g.$$

## 15.10.2 Assessment

1. Modify your `rk4` program to solve the simultaneous ODEs for projectile motion, (15.52), for a frictional force with $n = 1$, and to plot the results.
2. Check that you get a plot similar to the dotted one shown in Fig. 15.6.
3. In general, it is not possible to compare analytic and numerical results to realistic problems because analytic expressions do not exist. However, we do know the analytic expressions for the frictionless case, and we may turn friction off in our numerical algorithm and then compare the two. This is not a guarantee that we have handled friction correctly, but it is a guarantee that we have something wrong if the comparison fails. Modify your program to also have it compute trajectories with the friction coefficient $k = 0$.
4. The model of friction (15.51) with $n = 1$ is appropriate for low velocities. Modify your program to handle $n = 3/2$ (medium-velocity friction) and $n = 2$ (high-velocity friction). To make a realistic comparison, adjust the value of $k$ for the latter two cases such that the initial force of friction, $kV_0^n$, is the same for all.
5. What is your conclusion about balls falling out of the sky?

## 15.11 Exploration: Planetary Motion

Newton's explanation of the motion of the planets in terms of a universal law of gravitation is one of the great achievements of science. He was able to prove that the planets traveled along elliptical paths with the sun at one vertex, and predicted periods of motion that agreed with observation. All Newton needed to postulate was that the force between a planet of mass $m$ and the sun of mass $M$ is given by

$$f = -\frac{GmM}{r^2} \tag{15.55}$$

Here $r$ is the distance between the planet of mass $m$ and sun of mass $M$, $G$ is a universal constant, and the minus sign indicates that the force is attractive and lies along the line connecting the planet and sun (Fig. 15.7). The hard part for Newton was solving the resulting differential equations, since he had to invent calculus to do it. Whereas the analytic solution is complicated, the numerical solution is not. Even for planets, the basic equation of motion is still

$$\mathbf{f} = m\mathbf{a} = m\frac{d^2\mathbf{x}}{dt^2} \tag{15.56}$$

with the force (15.55) having components (Fig. 15.7):

$$f_x = f\cos\theta = f\frac{x}{r} \qquad f_y = f\sin\theta = f\frac{y}{r} \qquad (r = \sqrt{x^2+y^2}) \tag{15.57}$$

The equation of motion yields two simultaneous, second-order ODEs:

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^3} \qquad \frac{d^2y}{dt^2} = -GM\frac{y}{r^3} \tag{15.58}$$

In Section 15.10.1 we described how to write simultaneous, second-order ODEs in the standard `rk4` form. The same method applies here, but without friction but with the gravitational force now having two, nonconstant components.

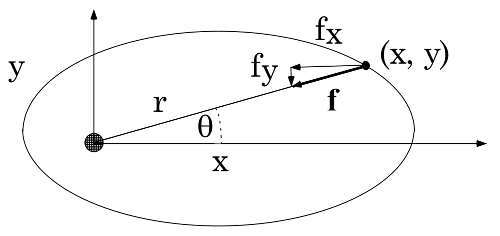

**Fig. 15.7** The gravitational force on a planet a distance $r$ from the sun. The $x$ and $y$ components of the force are indicated.

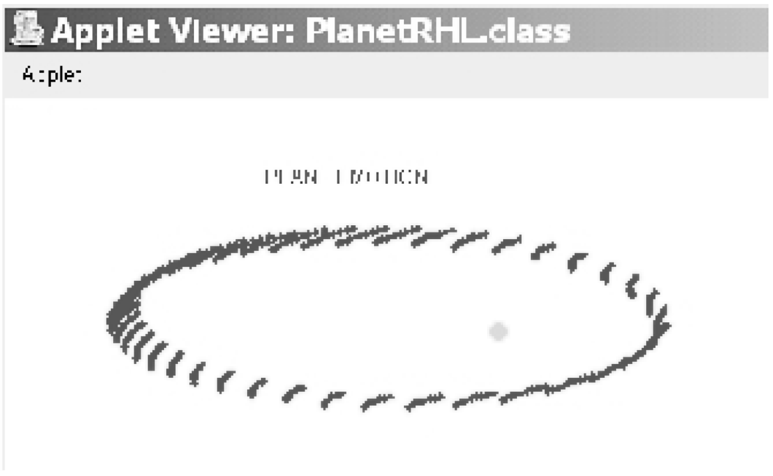

**Fig. 15.8** The precession of a planet's orbit when the gravitational force $\propto 1/r^4$.

## 15.11.1 Implementation: Planetary Motion

1. To keep the calculation simple, assume units such that $GM = 1$, and that the initial conditions are

$$x(0) = 0.5 \qquad y(0) = 0 \qquad v_x(0) = 0.0 \qquad v_y(0) = 1.63.$$

2. Modify your ODE solver program to solve (15.58).
3. Make the number of time steps large enough so that you can see the planet's orbit repeat on top of itself.
4. You may need to make the time step small enough so that the orbit closes upon itself, as it should, and just repeats. This should be a nice ellipse.
5. Experiment with initial conditions until you obtain the ones that produce a circular orbit (a circle is a special case of an ellipse).
6. Once you have good precision, see the effect of progressively increasing the initial velocity until the orbit opens up and becomes a hyperbola.
7. Use the same initial conditions as produced the ellipse, but new investigate the effect of the power in (15.55) being $1/r^4$ rather than $1/r^2$. You should find that the orbital ellipse now rotates or precesses (Fig. 15.8). In fact, as you should verify, even a slight variation from exactly an inverse square power law will cause the orbit to precess.

#### 15.11.1.1 **Exploration: Restricted Three-Body Problem**

Extend the previous solution for planetary motion to one in which a satellite of tiny mass moves under the influence of two planets of equal mass $M = 1$. Consider the planets as rotating about their center of mass in circular orbits, and of such large mass that they are uninfluenced by the satellite. Assume that all motions remain in the $x$–$y$ plane, and that units are such that $G = 1$.