

Chapter 3

Simulating Particle Motion

©2010 by Wolfgang Christian
16 August 2009

Adapted from *An Introduction to Computer Simulation Methods* by Harvey Gould, Jan Tobochnik, and Wolfgang Christian

We discuss several numerical methods needed to simulate the motion of particles using Newton's laws and introduce the Ordinary Differential Equation (ODE) editor which it possible to select different numerical algorithms. *EJS* 3D elements are also introduced to model motion in three dimensions.

3.1 Numerical Algorithms

To motivate the need for ordinary differential equation (ODE) solvers, we discuss why the simple Euler algorithm is insufficient. The Euler algorithm assumes that the velocity and acceleration do not change significantly during the time step Δt . Thus, to achieve an acceptable numerical solution, the time step Δt must be chosen to be sufficiently small. If we make Δt too small, we run into several problems. As we do more and more iterations, the round-off error due to the finite precision of any floating point number will accumulate and eventually the numerical results will become inaccurate. Also, the greater the number of iterations, the greater the computer time required for the program to finish. In addition to these problems, the Euler algorithm is unstable for many systems, which means that the errors accumulate exponentially, and thus the numerical solution becomes inaccurate very quickly. For these reasons more accurate and stable numerical algorithms are necessary.

We begin our study of ODE solution algorithms with the simple harmonic oscillator (SHO) model because its analytic solution is well known and because its constantly varying acceleration produces numerical errors without the complexities of other problems. Go to the [Chapter 3](#) source directory and open the SHO Euler model. This model implements a solution to the following two

first-order differential equations for a particle attached to a Hooke's law spring:

$$\frac{dx}{dt} = v \quad (3.1a)$$

$$\frac{dv}{dt} = -\frac{k}{m}x, \quad (3.1b)$$

where x is the displacement from equilibrium and k is the spring constant.

Exercise 3.1. Simple harmonic oscillator

- Examine the SHO Euler model and explain how the implementation of the Euler algorithm in the evolution method solves Newton's second law $F = ma$ for the simple harmonic oscillator.
- The general form of the analytical solution of (3.1) can be expressed as

$$x(t) = A \cos \omega_0 t + B \sin \omega_0 t, \quad (3.2)$$

where $\omega_0^2 = k/m$. What is the form of $v(t)$? Show that (3.2) satisfies (3.1) with $A = x(t=0)$ and $B = v(t=0)/\omega_0$. Add a plot of the analytic solution to the model.

- Test your model with initial displacements of $x = 1$, $x = 2$, and $x = 4$ using a time step sufficiently small so that you do not observe any difference between the numerical and analytical solution graphs. Is the time for the ball to reach $x = 0$ always the same?

To illustrate why we need algorithms other than the simple Euler algorithm

$$v(t + \Delta t) = v(t) + a(t)\Delta t \quad (3.3a)$$

$$y(t + \Delta t) = y(t) + v(t)\Delta t, \quad (3.3b)$$

we make a very simple change and write

$$v(t + \Delta t) = v(t) + a(t)\Delta t \quad (3.4a)$$

$$x(t + \Delta t) = x(t) + v(t + \Delta t)\Delta t, \quad (3.4b)$$

where a is the acceleration. The only difference between this algorithm and the Euler algorithm in (3.3), is that the computed velocity at the end of the interval, $v(t + \Delta t)$, is used to compute the new position, $x(t + \Delta t)$ in (3.4b). As we will see in more detail in Problem 3.2, this modified Euler algorithm is significantly better for oscillating systems. We refer to this algorithm as the Euler-Cromer algorithm.

Problem 3.2. Comparison of Euler algorithms

- Determine the numerical error in the maximum displacement (the amplitude) of the simple harmonic oscillator after the particle has evolved for several cycles with $\Delta t = 0.1$. Is the original Euler algorithm stable for this system? What happens if you run for longer times? Repeat with $\Delta t = 0.01$. For simplicity, choose units such that $k = 1$ and $m = 1$.
- Modify your model and repeat part (a) using the Euler-Cromer algorithm. Is this algorithm any better, and if so, in what way?

- c. Modify your model so that it computes the total energy, $E = mv^2/2 + kx^2/2$. How well is the total energy conserved for the two algorithms?

Perhaps it has occurred to you that it would be better to compute the velocity at the middle of the interval rather than at the beginning or at the end. The *Euler-Richardson* algorithm is based on this idea. This algorithm is particularly useful for velocity-dependent forces such as viscous drag, but does as well as other simple algorithms for forces that do not depend on the velocity. The algorithm consists of using the Euler algorithm to find the intermediate position y_{mid} and velocity v_{mid} at a time $t_{\text{mid}} = t + \Delta t/2$. We then compute the force, $F(y_{\text{mid}}, v_{\text{mid}}, t_{\text{mid}})$ and the acceleration a_{mid} at $t = t_{\text{mid}}$. The new position y_{n+1} and velocity v_{n+1} at time t_{n+1} are found using v_{mid} and a_{mid} and the Euler algorithm. We summarize the Euler-Richardson algorithm as:

$$a_n = F(y_n, v_n, t_n)/m \quad (3.5a)$$

$$v_{\text{mid}} = v_n + \frac{1}{2}a_n\Delta t \quad (3.5b)$$

$$y_{\text{mid}} = y_n + \frac{1}{2}v_n\Delta t \quad (3.5c)$$

$$a_{\text{mid}} = F(y_{\text{mid}}, v_{\text{mid}}, t + \frac{1}{2}\Delta t)/m, \quad (3.5d)$$

and

$$v_{n+1} = v_n + a_{\text{mid}}\Delta t \quad (3.6a)$$

$$y_{n+1} = y_n + v_{\text{mid}}\Delta t. \quad (\text{Euler-Richardson algorithm}) \quad (3.6b)$$

Although twice as many computations per time step are done, the Euler-Richardson algorithm is much faster than the Euler algorithm because we can make the time step larger and still obtain better accuracy than with either the Euler or Euler-Cromer algorithms. A derivation of the Euler-Richardson algorithm is given in Appendix 3B.

Exercise 3.3. The Euler-Richardson algorithm

- Add a second evolution page to the SHO Euler model that implements the Euler-Richardson algorithm. (Right-click on the tab of the Euler evolution page to create a new evolution code page. Implement the Euler-Richardson algorithm on this new page and right-click on the original tab to disable the Euler method code page.) *EJS* ignores disabled code pages so this way is convenient for testing code without creating a new model.
- Compute the SHO energy error after several cycles and compare your results with the Euler algorithm.
- Repeat part (b) for a one-dimensional falling particle model. Determine the error in the computed position when the particle hits the ground using $\Delta t = 0.08, 0.04, 0.02,$ and 0.01 . How do your results compare with the Euler algorithm? How does the error in the velocity depend on Δt for each algorithm?

As we gain more experience simulating various physical systems, we will learn that no single algorithm for solving Newton's equations of motion numerically is superior under all conditions.

3.2 ODE editor

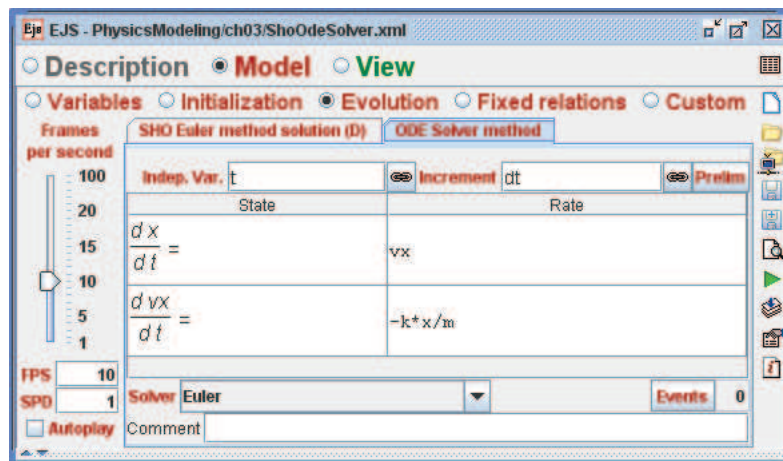


Figure 3.1: The simple harmonic oscillator equations of motion in the ODE editor.

Programming a simple numerical ODE solution algorithm is not difficult, but there are many sophisticated algorithms that require pages of intricate code. These sophisticated numerical techniques can use large step sizes and still produce small errors because they automatically estimate the error and change (adapt) the step size if the error becomes too large. Designing and implementing such advanced algorithms is a job for experts. To make it easy to use such algorithms, *EJS* has a built-in ODE editor that allows the user to enter the differential equations in a natural form and select the numerical methods described in Table 3.1. We now illustrate how to use the editor and investigate the properties of fixed step size ODE solvers. The properties of adaptive step size solvers are studied in the context of gravitational models in Chapter 5.

Many physical models, such as Newton's second law for particle motion in one dimension, give rise to a second-order differential equation that takes the form

$$\frac{d^2x}{dt^2} = a(x, v, t). \quad (3.7)$$

This equation can be converted to two first-order equations by considering both position x and its rate of change (velocity) v to be unknown functions of time:

$$\dot{x} = v \quad (3.8a)$$

$$\dot{v} = a(x, v, t). \quad (3.8b)$$

The initial position x_0 and the initial velocity v_0 are advanced by incrementing the independent variable to obtain x_1 and v_1 , respectively. The process of defining extra variables such as v to reduce the order of a differential equation (but increase the number of equations) can be extended to higher-order derivatives as well. Because higher-order differential equations can be converted into a system of first-order equations, we need only consider general methods of solving systems of

first-order equations. We refer to quantities, such as position and velocity, that describe the state of the system and that are evolved using differential equations as dynamical variables.

Additional particles or additional spatial dimensions add additional equations. In general, a system of particles results in a system of first-order ordinary differential equations with an independent time t variable and with N dependent dynamical variables that can be written as

$$\dot{x}_0 = f_0(x_0, x_1, \dots, x_{N-1}, t) \quad (3.9a)$$

$$\dot{x}_1 = f_1(x_0, x_1, \dots, x_{N-1}, t) \quad (3.9b)$$

$$\vdots$$

$$\dot{x}_{N-1} = f_N(x_0, x_1, \dots, x_{N-1}, t). \quad (3.9c)$$

This system is said to be autonomous if the functions f_i do not explicitly depend on the independent variable. If we think of the variables x_i as components of a state vector \mathbf{x} , this system can be written compactly as

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t). \quad (3.10)$$

The *EJS* ODE editor solves systems of equations of this type. The variable that is used to take the derivative can be any independent parameter and is usually the time t . In this case the solution algorithm produces approximate values of the dynamical variables at discrete time steps Δt .

Consider again the SHO model. Open the modified SHO ODE Solver model in the Chapter 3 source code directory, navigate to the evolution workpanel, and click on the tabs to examine the two evolution pages. The explicit evolution page is disabled and the ODE editor shown in Figure 3.1 is enabled. The independent variable and the step size are shown near the top of the ODE editor page and each dynamical (state) variable and its rate are a row in the editor. A drop-down menu near the bottom lets us choose the particular ODE numerical (solver) algorithm. Right-click on the workpanel tabs to enable and disable pages to test algorithms.

Exercise 3.4. The Euler-Richardson algorithm

Verify that the Euler and Euler-Richardson ODE solvers produce the same results as the explicit implementations used in Exercises 3.2 and 3.3.

We have seen that numerical methods do not produce an exact solution to (3.10) but an approximation to it. It can be shown that if the independent variable step size Δt is small enough, the approximation can be made as accurate as desired. A smaller value of Δt requires more steps to advance the evolution of the system through a unit interval. Hence, accuracy comes at the price of additional computations. Even if we are willing to pay this price, there is a limit to the accuracy because of the finite precision of computer arithmetic.

The *EJS* modeling tool is designed to produce real-time simulations that display output after every evolution step. Although present-day computers can execute a simple evolution algorithm in a few milliseconds, *EJS* forces the computer to wait (sleep) until the next frame is scheduled to appear. Increasing the steps per display (SPD) parameter allows the computer to perform multiple evolution steps between frames, thereby speeding data collection. The animation will appear jerky if the computation between frames is lengthy because we require that all computations be complete before the screen is redrawn.

Exercise 3.5. Computation time

Run the SHO model with $\Delta t = 0.1$ and with $\Delta t = 0.01$ and plot the change in energy $E(t) - E_0$ as a function of time. By what factor does the error decrease when the step size is decreased by a factor of ten?

Continue to decrease the time step but increase the SPD parameter by the same factor until you observe that the simulation runs more slowly. The effective frame rate decreases because of the increased number of computations and because of the increasing number of data points stored in the plotting panel curves. You can remove this drawing inefficiency by collecting the fewer data points using the inspector to set the Skip property of the traces to 10 so that only one out of every ten points is stored.

Although the Euler algorithm is not used in practice because it produces a poor approximation unless Δt is much too small, it is simple and is a good introduction to more sophisticated algorithms. A straightforward theoretical analysis shows that the total (global) position error produced by Euler's method decreases linearly with Δt when advancing the model from t_{initial} to t_{final} . Euler's method is therefore termed a method of order one. A numerical method that reduces the total error quadratically with Δt is said to be second-order. In general, the order of a method is the power-law relation between the time step and total error. For example, *EJS* implements an eighth-order method developed by Fehlberg that reduces the error by a factor of $256 = 2^8$ when the time step is reduced by a factor of two.

Problem 3.6. Numerical method order

Run the SHO model using the fourth-order Runge-Kutta ODE solver with various time steps to determine the order of this solver. Should you use the absolute error, the relative error, or the error at the maximum displacement to estimate the error? Does the energy error exhibit the same power dependence on Δt as the position error? Repeat with Fehlberg's eight-order method.

In classical (Newtonian) physics, the position and velocity (momentum) space for the dynamical variables is called *phase space*. The trajectory of a particle or a system of particles through phase space is completely determined if we know the initial state and an expression for its rate of change. The simple harmonic oscillator, for example, corresponds to a two-dimensional phase space.

Problem 3.7. Phase space

- Modify the SHO model to show its phase space trajectory. How does the phase space trajectory evolve using the analytic solution? How does it evolve using Euler's method?
- An easy to understand and implement numerical algorithm described in Appendix 3B is the *Verlet* algorithm. Implement this algorithm for the simple harmonic oscillator and compare the SHO phase space trajectory for this algorithm to the analytic phase space trajectory.

3.3 Effects of Drag Resistance

In Chapter 2 we modeled a projectile near the surface of the Earth without air friction, including a plot of position versus time and an animation of a ball moving through the air. In the following

Table 3.1: Ordinary differential equation solvers are selected using a drop down menu on the editor page. Solvers are permitted to optimize their internal step size and *EJS* uses interpolation to produce solution points at the requested points. The accuracy of the interpolation is guaranteed to meet or exceed the accuracy of the solver algorithm. See Appendix 5.1A for a discussion of adaptive solvers.

<i>EJS</i> ODE solvers.	
Euler	Euler algorithm.
Euler-Richardson	Euler-Richardson second-order algorithm with fixed step size.
Runge-Kutta	Runge-Kutta fourth-order algorithm with fixed step size. A rule of thumb states that RK4 has an accuracy of approximately 10^{-5} .
Bogacki-Shampine	Bogacki-Shampine is a Runge-Kutta-Fehlberg adaptive step size method of order three with four stages. It has the property that the last rate evaluation can be used as the first rate in the next time step so that it uses approximately rate evaluations per step. It uses an embedded second-order method to implement the adaptive step size algorithm.
Cash-Karp	An adaptive step size algorithm based on fourth- and fifth-order Runge-Kutta-Fehlberg methods using coefficients developed by Cash and Karp. This solver is the default for a new page of ODEs because it provides excellent performance and stability.
Fehlberg 8	Fehlberg's eight-order algorithm with fixed step size.
Fehlberg 8(7)	Fehlberg's eight-order algorithm with adaptive step size.
Dormand-Prince 5(4)	A fifth order adaptive step size algorithm based using coefficients developed by Dormand and Prince with built-in interpolation.
Dormand-Prince 8(53)	An 8th order adaptive method from the Dormand-Prince family. It is described in Section II.5, p. 181, of <i>Solving Ordinary Differential Equations I. Nonstiff Problems</i> by Hairer, Nørsett, and Wanner.
Radau 5(4)	An implicit RK 4th order adaptive method described in <i>Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems</i> by Hairer and Wanner. This method is recommended for stiff equations.

we discuss more realistic models that can be simulated by modifying this basic model. Use the Falling Particle model in the Chapter 3 source code directory as a starting point for the exercises and problems in this section.

The analytical solution for free fall near the Earth's surface, (2.4), is well known and thus finding a numerical solution is useful only as an introduction to numerical methods. It is not difficult to think of more realistic models of motion near the Earth's surface for which the equations of motion do not have simple analytical solutions. For example, if we take into account the variation of the Earth's gravitational field with the distance from the center of the Earth, then the force on a particle is not constant. According to Newton's law of gravitation, the force due to the Earth on a particle of mass m is given by

$$F = \frac{GMm}{(R+y)^2} = \frac{GMm}{R^2(1+y/R)^2} = mg\left(1 - 2\frac{y}{R} + \dots\right), \quad (3.11)$$

where y is measured from the Earth's surface, R is the radius of the Earth, M is the mass of the Earth, G is the gravitational constant, and $g = GM/R^2$.

Problem 3.8. Position-dependent force

Modify the Falling Particle model to simulate the fall of a particle with the position-dependent force law (3.11). Assume that a particle is dropped from a height h with zero initial velocity and compute its impact velocity (speed) when it hits the ground at $y = 0$. Determine the value of h for which the impact velocity differs by one percent from its value with a constant acceleration $g = 9.8 \text{ m/s}^2$. Take $R = 6.37 \times 10^6 \text{ m}$. Make sure that the one percent difference is due to the physics of the force law and not the accuracy of your algorithm.

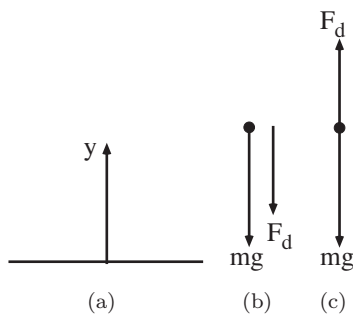


Figure 3.2: (a) Coordinate system with y measured positive upward from the ground. (b) The force diagram for downward motion. (c) The force diagram for upward motion.

For particles near the Earth's surface, a more important modification is to include the drag force due to air resistance. The direction of the drag force $F_d(v)$ is opposite to the velocity of the particle (see Figure 3.2). For a falling body $F_d(v)$ is upward as shown in Figure 3.2(b). Hence, the total force F on the falling body can be expressed as

$$F = -mg + F_d. \quad (3.12)$$

The velocity dependence of $F_d(v)$ is known theoretically in the limit of very low speeds for small objects. In general, it is necessary to determine the velocity dependence of $F_d(v)$ empirically over a limited range of velocities. One way to obtain the form of $F_d(v)$ is to measure y as a function of t and then compute $v(t)$ by calculating the numerical derivative of $y(t)$. Similarly we can use $v(t)$ to compute $a(t)$ numerically. From this information it is possible in principle to find the acceleration as a function of v and to extract $F_d(v)$ from (3.12). However, this procedure introduces errors (see Problem 3.9a) because the accuracy of the derivatives will be less than the accuracy of the measured position. An alternative is to reverse the procedure, that is, assume an explicit form for the v dependence of $F_d(v)$, and use it to solve for $y(t)$. If the calculated values of $y(t)$ are consistent with the experimental values of $y(t)$, then the assumed v dependence of $F_d(v)$ is justified empirically.

The two common assumed forms of the velocity dependence of $F_d(v)$ are

$$F_{1,d}(v) = C_1 v, \quad (3.13a)$$

and

$$F_{2,d}(v) = C_2 v^2, \quad (3.13b)$$

where the parameters C_1 and C_2 depend on the properties of the medium and the shape of the object. In general, (3.13a) and (3.13b) are useful *phenomenological* expressions that yield approximate results for $F_d(v)$ over a limited range of v .

Because $F_d(v)$ increases as v increases, there is a limiting or *terminal velocity* (speed) at which the net force on a falling object is zero. This terminal speed can be found from (3.12) and (3.13) by setting $F_d = mg$ and is given by

$$v_{1,t} = \frac{mg}{C_1}, \quad (\text{linear drag}) \quad (3.14a)$$

$$v_{2,t} = \left(\frac{mg}{C_2}\right)^{1/2}, \quad (\text{quadratic drag}) \quad (3.14b)$$

for the linear and quadratic cases, respectively. It often is convenient to express velocities in terms of the terminal velocity. We can use (3.13) and (3.14) to write F_d in the linear and quadratic cases as

$$F_{1,d} = C_1 v_{1,t} \left(\frac{v}{v_{1,t}}\right) = mg \frac{v}{v_{1,t}}, \quad (3.15a)$$

$$F_{2,d} = C_2 v_{2,t}^2 \left(\frac{v}{v_{2,t}}\right)^2 = mg \left(\frac{v}{v_{2,t}}\right)^2. \quad (3.15b)$$

Hence, we can write the net force (per unit mass) on a falling object in the convenient forms

$$F_1(v)/m = -g \left(1 - \frac{v}{v_{1,t}}\right), \quad (3.16a)$$

$$F_2(v)/m = -g \left(1 - \frac{v^2}{v_{2,t}^2}\right). \quad (3.16b)$$

To determine if the effects of air resistance are important during the fall of ordinary objects, consider the fall of a pebble of mass $m = 10^{-2}$ kg. To a good approximation, the drag force

t (s)	position (m)	t (s)	position (m)	t (s)	position (m)
0.2055	0.4188	0.4280	0.3609	0.6498	0.2497
0.2302	0.4164	0.4526	0.3505	0.6744	0.2337
0.2550	0.4128	0.4773	0.3400	0.6990	0.2175
0.2797	0.4082	0.5020	0.3297	0.7236	0.2008
0.3045	0.4026	0.5266	0.3181	0.7482	0.1846
0.3292	0.3958	0.5513	0.3051	0.7728	0.1696
0.3539	0.3878	0.5759	0.2913	0.7974	0.1566
0.3786	0.3802	0.6005	0.2788	0.8220	0.1393
0.4033	0.3708	0.6252	0.2667	0.8466	0.1263

Table 3.2: Results for the vertical fall of a coffee filter. Note that the initial time is not zero. The time difference is ≈ 0.0247 . This data also is available in the `falling.txt` file in the Chapter 3 source code directory.

is proportional to v^2 . For a spherical pebble of radius 0.01 m, C_2 is found empirically to be approximately 10^{-2} kg/m. From (3.14b) we find the terminal velocity to be about 30 m/s. Because this speed would be achieved by a freely falling body in a vertical fall of approximately 50 m in a time of about 3 s, we expect that the effects of air resistance would be appreciable for comparable times and distances.

Data stored in simple text files can be read into Data Tool for analysis. The Falling Particle model in the Chapter 3 source code directory has a tool button and we use it in Problem 3.9 to read the data in Table 3.2 in order to compare our computational results to empirical data.

Problem 3.9. The fall of a coffee filter

- Determine the terminal velocity from the empirical data given in Table 3.2. Run the Falling Particle model, display the Data Tool, and open the `falling.txt` file in the tool using the [File] \rightarrow [Open] menu item. Select the fit option in the tool and fit the last ten table rows to a straight line. Note that you can select rows for the fit by dragging within the table. What is the falling coffee filter's terminal velocity?
- Use the empirical data in the data file to create a new column showing the coffee filter velocity $v(t)$ using the central difference approximation given by

$$v(t) \approx \frac{y(t + \Delta t) - y(t - \Delta t)}{2\Delta t}. \quad (\text{central difference approximation}) \quad (3.17)$$

Show that if we write the acceleration as $a(t) \approx [v(t + \Delta t) - v(t)]/\Delta t$ and use the backward difference approximation for the velocity,

$$v(t) \approx \frac{y(t) - y(t - \Delta t)}{\Delta t}, \quad (\text{backward difference approximation}) \quad (3.18)$$

we can express the acceleration as

$$a(t) \approx \frac{y(t + \Delta t) - 2y(t) + y(t - \Delta t)}{(\Delta t)^2}. \quad (3.19)$$

Use (3.19) to create a data tool column that shows the acceleration.



Figure 3.3: A falling coffee filter does not fall with constant acceleration due to the effects of air resistance. The motion sensor below the filter is connected to a computer, which records position data and stores it in a text file.

- c. Use your approximate results for $v(t)$ and $a(t)$ to plot a as a function of v and, if possible, determine the nature of the velocity dependence of a . Discuss the accuracy of your results for the acceleration.
- d. Add a quadratic drag resistance to the Falling Ball simulation and set the parameters and the initial conditions to match the empirical data. Choose the terminal velocity as an input parameter, and take as your first guess for the terminal velocity the value you found in part (a). Make sure that your computed results for the height of the particle, do not depend on Δt to the necessary accuracy. Compare your plot of the computed values of $y(t)$ for different choices of the terminal velocity with the empirical values of $y(t)$.
- e. Repeat parts (d) assuming linear drag resistance. What are the qualitative differences between the two computed forms of $y(t)$ for the same terminal velocity?
- f. Visually determine which form of the drag force yields the best overall fit to the data. If the fit is not perfect, what is your criteria for which fit is better? Is it better to match your results to the experimental data at early times or at later times? Or did you adopt another criterion? What can you conclude about the velocity-dependence of the drag resistance on a coffee filter?

Problem 3.10. Effect of air resistance on the ascent and descent of a pebble

- a. Verify the claim made in Section 3.3 that the effects of air resistance on a falling pebble can be appreciable. Compute the speed at which a pebble reaches the ground if it is dropped from rest at a height of 50 m. Compare this speed to that of a freely falling object under the same

conditions. Assume that the drag force is proportional to v^2 and that the terminal velocity is 30 m/s.

- b. Suppose a pebble is thrown vertically upward with an initial velocity v_0 . In the absence of air resistance, we know that the maximum height reached by the pebble is $v_0^2/2g$, its velocity upon return to the Earth equals v_0 , the time of ascent equals the time of descent, and the total time in the air is $2v_0/g$. Before doing a simulation, give a simple qualitative explanation of how you think these quantities will be affected by air resistance. In particular, how will the time of ascent compare with the time of descent?
- c. Do a simulation to determine if your qualitative answers in part (b) are correct. Assume that the drag force is proportional to v^2 . Choose the coordinate system shown in Figure 3.2 with y positive upward. What is the net force for $v > 0$ and $v < 0$? We can characterize the magnitude of the drag force by a terminal velocity even if the motion of the pebble is upward and even if the pebble never attains this velocity. Choose the terminal velocity $v_t = 30$ m/s, corresponding to a drag coefficient of $C_2 \approx 0.01089$. It is a good idea to choose an initial velocity that allows the pebble to remain in the air for a time sufficiently long so that the effect of the drag force is appreciable. A reasonable choice is $v(t=0) = 50$ m/s. You might find it convenient to express the drag force in the form $F_d \propto -\mathbf{v} \cdot \text{Math.abs}(\mathbf{v})$.

There are many ways to determine the maximum height of the pebble in 3.10. We can measure the maximum value of the $y(t)$ curve or the zero crossing of the $v(t)$ curve. We can also determine the maximum height by using an evolution page to save the particle's position before the ODE step, performing the ODE step, and using a third evolution page to print the maximum height when the velocity changes sign

```
if (v*vold < 0) {
    _println("maximum height = " + y);
}
```

where $\mathbf{v} = v_{n+1}$ and $\text{vold} = v_n$. Multiple Evolution pages are evaluated in left to right in the order of the workpanel tabs. This technique is not very accurate unless the step size is small because the maximum height will almost never occur at an evolution step.

EJS allows the user to stop an ODE simulation at times other than at an integer multiple of the step size using an ODE event. An ODE event is triggered when a function $h(t)$ passes through zero as the dynamical system evolves. The solver checks the function at the end of every step. When a zero crossing is detected, the solver discards the result and recomputes the evolution with smaller intermediate steps until it finds the exact intermediate value that produced the event. The event then triggers an action, such as printing the particle height, using dynamical variables evaluated at the intermediate value.

EJS supports three event types. A *zero crossing event* occurs when the function changes sign and a *positive crossing event* occurs when the function goes from positive to negative. A *state event* occurs when the error function goes from positive to negative, but unlike the positive crossing event, the event action must change the dynamical state so that the function $h(t)$ is positive. State event actions are slightly harder to implement and we defer their discussion until Chapter 6. Zero crossing and positive crossing event types are easy to use because they only trigger once when the crossing occurs. We provide a brief introduction to zero-crossing events in Exercise 3.11.

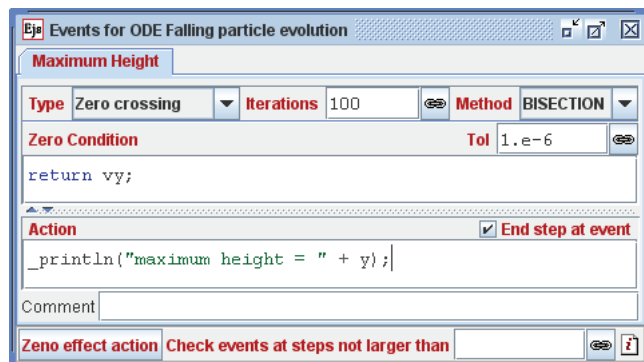


Figure 3.4: An ODE event is triggered when a function changes sign. The top section computes and returns the value of the function. The bottom section contains the action that the event performs.

Exercise 3.11. ODE zero crossing event

- The Falling Particle model in the Chapter 3 source code directory has a disabled zero crossing event that is triggered when the velocity is zero. Navigate to the Evolution workpanel for this model, click on the event button, and right click on the tab near the top of the event editor to enable this event. Run the model and show that the event occurs (to within the specified tolerance) at the particle's maximum height regardless of the time step.
- Add a second event that pauses the simulation when the particle hits the ground.
- Add an event to print the maximum height to the model that you developed in Problem 3.10.

3.4 Two-Dimensional Trajectories

You are probably familiar with two-dimensional trajectory problems in the absence of air resistance. For example, if a ball is thrown in the air with an initial velocity v_0 at an angle θ_0 with respect to the ground, how far will the ball travel in the horizontal direction, and what is its maximum height and time of flight? Suppose that a ball is released at a nonzero height h above the ground. What is the launch angle for the maximum range? Are your answers still applicable if air resistance is taken into account? We consider these and similar questions in the following.

Consider an object of mass m whose initial velocity \mathbf{v}_0 is directed at an angle θ_0 above the horizontal (see Figure 3.53.5(a)). The particle is subjected to gravitational and drag forces of magnitude mg and F_d ; the direction of the drag force is opposite to \mathbf{v} (see Figure 3.53.5(b)). Newton's equations of motion for the x and y components of the motion can be written as

$$m \frac{dv_x}{dt} = -F_d \cos \theta \quad (3.20a)$$

$$m \frac{dv_y}{dt} = -mg - F_d \sin \theta. \quad (3.20b)$$

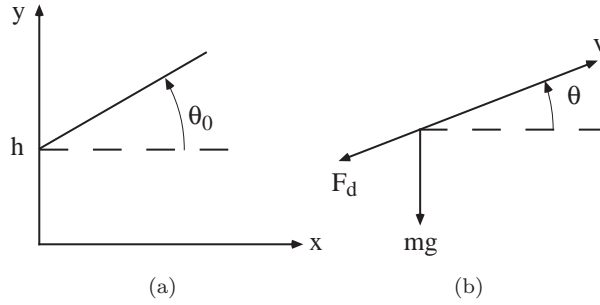


Figure 3.5: (a) A ball is thrown from a height h at an launch angle θ_0 measured with respect to the horizontal. The initial velocity is \mathbf{v}_0 . (b) The gravitational and drag forces on a particle.

For example, let us maximize the range of a round steel ball of radius 4 cm. A reasonable assumption for a steel ball of this size and typical speed is that $F_d = C_2 v^2$. Because $v_x = v \cos \theta$ and $v_y = v \sin \theta$, we can rewrite (3.20) as

$$m \frac{dv_x}{dt} = -C_2 v v_x \quad (3.21a)$$

$$m \frac{dv_y}{dt} = -mg - C_2 v v_y. \quad (3.21b)$$

Note that $-C_2 v v_x$ and $-C_2 v v_y$ are the x and y components of the drag force $-C_2 v^2$. Because (3.21a) and (3.21b) for the change in v_x and v_y involve the square of the velocity, $v^2 = v_x^2 + v_y^2$, we cannot calculate the vertical motion of a falling body without reference to the horizontal component, that is, the motion in the x and y direction is *coupled*.

Problem 3.12. Trajectory of a steel ball

- The Falling 2D Particle model in the Chapter 3 source code directory models the two-dimensional trajectory of a ball moving in air without air friction. Add an EJS zero crossing event to pause the simulation when the projectile hits the ground and compare your computed range with the exact results. For example, assume that a ball is thrown from ground level at an angle θ_0 above the horizontal with an initial velocity of $v_0 = 15$ m/s. Vary θ_0 and show that the maximum range occurs at $\theta_0 = \theta_{\max} = 45^\circ$. What is R_{\max} , the maximum range, at this angle? Compare your numerical result to the analytical result $R_{\max} = v_0^2/g$.
- Suppose that a steel ball is thrown from a height h at an angle θ_0 above the horizontal with the same initial speed as in part (a). If you neglect air resistance, do you expect θ_{\max} to be larger or smaller than 45° ? What is θ_{\max} for $h = 2$ m? By what percent is the range R changed if θ is varied by 2% from θ_{\max} ?
- Consider the effects of air resistance on the range and optimum angle of a steel ball. For a ball of mass 7 kg and cross-sectional area 0.01 m², the parameter $C_2 \approx 0.1$. What are the units of C_2 ? It is convenient to exaggerate the effects of air resistance so that you can more easily determine the qualitative nature of the effects. Hence, compute the optimum angle for $h = 2$ m,

$v_0 = 30$ m/s, and $C_2/m = 0.1$, and compare your answer to the value found in part (b). Is R more or less sensitive to changes in θ_0 from θ_{\max} than in part (b)? Determine the optimum launch angle and the corresponding range for the more realistic value of $C_2 = 0.1$. A detailed discussion of the maximum range of the ball has been given by Lichtenberg and Wills.

Problem 3.13. Comparing the motion of two objects

Consider the motion of two identical objects that both start from a height h . One object is dropped vertically from rest and the other is thrown with a horizontal velocity v_0 . Which object reaches the ground first?

- Give reasons for your answer assuming that air resistance can be neglected.
- Assume that air resistance cannot be neglected and that the drag force is proportional to v^2 . Give reasons for your anticipated answer for this case. Then perform numerical simulations using, for example, $C_2/m = 0.1$, $h = 10$ m, and $v_0 = 30$ m/s. Are your qualitative results consistent with your anticipated answer? If they are not, the source of the discrepancy might be an error in your program. Or the discrepancy might be due to your failure to anticipate the effects of the coupling between the vertical and horizontal motion.
- Suppose that the drag force is proportional to v rather than to v^2 . Is your anticipated answer similar to that in part (b)? Do a numerical simulation to test your intuition.

3.5 Decay processes

The power of mathematics when applied to physics comes in part from the fact that seemingly unrelated problems frequently have the same mathematical formulation. Hence, if we can solve one problem, we can solve other problems that might appear to be unrelated. For example, the growth of bacteria, the cooling of a cup of hot water, the charging of a capacitor in a RC circuit, and nuclear decay all can be formulated in terms of equivalent differential equations.

In 1958 two Cornell University engineering students presented a report titled “The Mechanisms of Cooling Hot Quiescent Liquids” in which they studied the effect of adding cream to a cup of hot coffee. Because typical brewing temperature for coffee is 85C (185F) and drinking temperature is 62C (143F), they studied the effect of adding cream when the coffee is served or adding it just before it is consumed. This is not a trivial problem because a hot body exchanges heat with its surroundings through the simultaneous processes of conduction, convection, evaporation, and radiation. Newton argued that because the thermal energy is proportional to the volume and the energy loss is proportional to the exposed area, the time of cooling is proportional to the diameter. Larger objects therefore cool more slowly. Laplace, Helmholtz, and Kelvin extended Newton’s model by considering gravitational energy and radiation to estimate the age of the Sun to be well over 20 million years. Although the discovery of nuclear fusion greatly increased this age estimate, the balance between cooling and thermonuclear fusion is of fundamental importance in stellar models. Today we use advanced heating, cooling, and energy transport models to debate the impact of global warming. Considering in detail the processes in these climate change models requires advanced supercomputer-based algorithms but we wish to lay the groundwork for such

studies and we start by studying the cooling-coffee problem. We create a simple model of how objects cool and use this simulation with different conditions to predict the final temperature.

Problem 3.14. Cooling of a cup of coffee

If the temperature difference between the water and its surroundings is not too large, the rate of change of the temperature of the water may be assumed to be proportional to the temperature difference. We can formulate this statement more precisely in terms of a differential equation:

$$\frac{dT}{dt} = -r(T - T_s), \quad (3.22)$$

where T is the temperature of the water, T_s is the temperature of its surroundings, and r is the cooling constant. The minus sign in (3.22) implies that if $T > T_s$, the temperature of the water will decrease with time. The value of the cooling constant r depends on the heat transfer mechanism, the contact area with the surroundings, and the thermal properties of the water. The relation (3.22) is sometimes known as Newton's law of cooling, even though the relation is only approximate, and Newton did not express the rate of cooling in this form.

- Create a model that computes the numerical solution of (3.22). Test your program by choosing the initial temperature $T_0 = 100^\circ\text{C}$, $T_s = 0^\circ\text{C}$, $r = 1$, and $\Delta t = 0.1$.
- Model the cooling of a cup of coffee by choosing $r = 0.03$. What are the units of r ? Plot the temperature T as a function of the time using $T_0 = 85^\circ\text{C}$ and $T_s = 17^\circ\text{C}$. Choose an appropriate numerical algorithm making sure that your value of Δt is sufficiently small so that it does not affect your results. What is the appropriate unit of time in this case?
- Suppose that the initial temperature of a cup of coffee is 85°C , but the coffee can be sipped comfortably only when its temperature is $\leq 62^\circ\text{C}$. Assume that the addition of one serving of cream cools the coffee by 5°C and add a button to the model that adds a serving of cream to coffee when pressed. If you are in a hurry and want to wait the shortest possible time, should the cream be added first and the coffee be allowed to cool, or should you wait until the coffee has cooled somewhat before adding the cream? Use your program to "simulate" these two cases. Choose $r = 0.03$ and $T_s = 17^\circ\text{C}$. What is the appropriate unit of time in this case? Assume that the value of r does not change when the cream is added.

Consider a large number of radioactive nuclei. Although the number of nuclei is discrete, we often may treat this number as a continuous variable because the number of nuclei is very large. In this case the law of radioactive decay is that the rate of decay is proportional to the number of nuclei. Thus we can write

$$\frac{dN}{dt} = -\lambda N, \quad (3.23)$$

where N is the number of nuclei and λ is the decay constant. Of course, we do not need to use a computer to solve this decay equation, and the analytical solution is

$$N(t) = N_0 e^{-\lambda t}, \quad (3.24)$$

where N_0 is the initial number of particles. The quantity λ in (3.23) or (3.24) has dimensions of inverse time.

Problem 3.15. Single nuclear species decay

- Create a model that solves and plots the nuclear decay problem. Input the decay constant, λ , from the control window. For $\lambda = 1$ and $\Delta t = 0.01$, compute the difference between the analytical result and the result of the Euler algorithm for $N(t)/N(0)$ at $t = 1$ and $t = 2$. Assume that time is measured in seconds. Repeat with 4th order Runge-Kutta algorithm and use this algorithm for the remainder of this problem.
- A common time unit for radioactive decay is the half-life, $T_{1/2}$, the time it takes for one-half of the original nuclei to decay. Another natural time scale is the time, τ , it takes for $1/e$ of the original nuclei to decay. Use your modified program to verify that $T_{1/2} = \ln 2/\lambda$. How long does it take for $1/e$ of the original nuclei to decay? How is $T_{1/2}$ related to τ ?
- Because it is awkward to treat very large or very small numbers on a computer, it is convenient to choose units so that the computed values of the variables are not too far from unity. Determine the decay constant λ in units of s^{-1} for $^{238}\text{U} \rightarrow ^{234}\text{Th}$ if the half-life is 4.5×10^9 years. What units and time step would be appropriate for the numerical solution of (3.23)? How would these values change if the particle being modeled were a muon with a half-life of 2.2×10^{-6} s?
- Modify your model so that the time t is expressed in terms of the half-life. That is, at $t = 1$ one half of the particles would have decayed and at $t = 2$, one quarter of the particles would have decayed. Use your program to determine the time for 1000 atoms of ^{238}U to decay to 20% of their original number. What would be the corresponding time for muons?

If the number of radioactive nuclei is larger, the rate of change of N can be approximated by a differential equation. Nuclei are, however, discrete and the continuous model fails when N is small. The probability of one nucleus decaying in a time interval Δt is

$$P(\Delta t) = 1 - e^{-\lambda \Delta t}. \quad (3.25)$$

Note that this single particle decay probability is approximately $\lambda \Delta t$ if the exponent is small. An evolution algorithm that decreases the current number of particles n based on the number of decay events is implemented as:

```

int count = 0; // counter for number of decays events
double prob=1-Math.exp(-lambda*dt); // probability of a single decay event
for (int i = 0; i < n; i++) { // loop over every particle
    if (Math.random()<prob) count++; // decay depends on random number
}
n-=count; // reduce n by decay counter

```

Problem 3.16. Random decay

Compare a continuous decay model based on (3.24) to a stochastic decay model based on (3.25). Use a separate Evolution page for each algorithm. Run the model with $N_0 = 1000$ and $\lambda = 0.5$. When do the two algorithms agree and when do you notice a difference between two algorithms?

Multiple nuclear decays produce systems of first-order differential equations. Problem 3.17 asks you to model such a system using the techniques similar to those that we have already used.

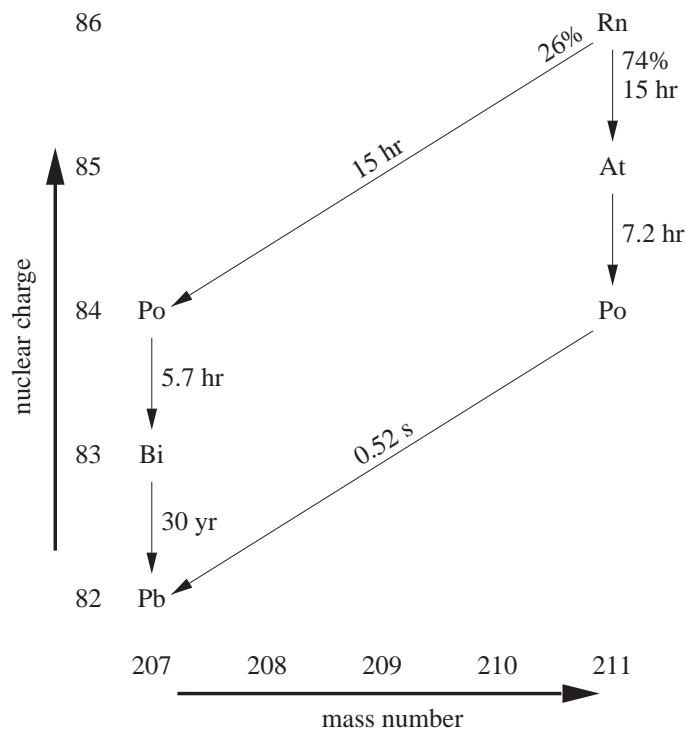


Figure 3.6: The decay scheme of ^{211}Rn . Note that ^{211}Rn decays via two branches, and the final product is the stable isotope ^{207}Pb . All vertical transitions are by electron capture, and all diagonal transitions are by alpha decay. The times represent half-lives.

Problem 3.17. Multiple nuclear decays

- ^{76}Kr decays to ^{76}Br via electron capture with a half-life of 14.8 h, and ^{76}Br decays to ^{76}Se via electron capture and positron emission with a half-life of 16.1 h. In this case there are two half-lives, and it is convenient to measure time in units of the smallest half-life. Write a program to compute the time dependence of the amount of ^{76}Kr and ^{76}Se over an interval of one week. Assume that the sample initially contains 1 gm of pure ^{76}Kr .
- ^{28}Mn decays via beta emission to ^{28}Al with a half-life of 21 h, and ^{28}Al decays by positron emission to ^{28}Si with a half-life of 2.31 min. If we were to use minutes as the unit of time, our program would have to do many iterations before we would see a significant decay of the ^{28}Mn . What simplifying assumption can you make to speed up the computation?
- ^{211}Rn decays via two branches as shown in Figure 3.6. Make any necessary approximations and compute the amount of each isotope as a function of time, assuming that the sample initially consists of $1\ \mu\text{g}$ of ^{211}Rn .

3.6 Visualizing Three-Dimensional Motion

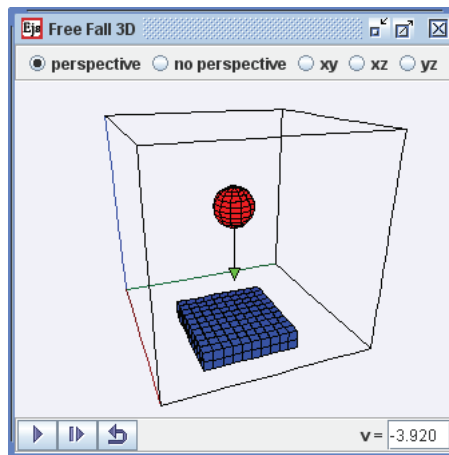



Figure 3.7: The falling 3D Particle model shows a three-dimensional view of a falling ball.

The world in which we live is three-dimensional (3D), and it is fun and sometimes necessary to visualize phenomena in three dimensions. Because we want a three-dimensional visualization framework designed for physics simulations, we have developed a small 3D library that does not require any add-on packages. A new *EJS* library that relies on the Sun Microsystems *Java3D* package is under development and will be released with *EJS* version 4.3. This new library will support hardware accelerated rendering of polygons, lighting and shading, and other high-end 3D visualization techniques.

A 3D view is created by adding a 3D Drawing Panel  to a frame and then adding 3D Elements from the 3D Drawables palette to the drawing panel. The Falling 3D Particle model in the Chapter 3 source code directory simulates the same physics as the 2D model but with a 3D View. The ball falls with constant acceleration $g = -9.8 \text{ m/s}^2$ in the z direction. Click-dragging on the ball changes its height but leaves its velocity unchanged. The reset button stops the animation and sets the initial conditions to $y = 1.8$ and $v_y = 0$.

Exercise 3.18. Three-dimensional models

- a. Load and inspect the Falling 3D Particle model and describe the differences between this model and the 2D model.
- b. Run the model and test the navigation controls.
 - Left-button click-dragging on the particle changes its position.
 - Left-button click-dragging on an empty regions rotates the view.
 - Left-button shift-click-dragging zooms in and out.
 - Left-button control-click-dragging translates (pans) the view.

- Left-button alt-click-dragging displays a 3D cursor on the scene (if the panel is Enabled).
- c. Add an initial x-velocity component to the model and add a 3D trail to show the trajectory in the view. Model the trajectory as the ball bounces past the end of the table top.

Although the 3D drawing panel is designed for three-dimensional visualizations, it also can show two-dimensional projections as shown in Figure 3.7. The model uses an integer variable that is set by radio buttons and is bound to the Projection property in the 3D panel to select the projection. The 3D panel's Projection property custom editor shows all five projections ordered according to their integer selector.

We will require only a small subset of the 3D elements to create the three-dimensional visualizations in this book and will introduce the necessary objects as needed. Readers may wish to run the 3D demonstration programs on the *EJS* information Wiki for an overview of additional 3D drawing capabilities.

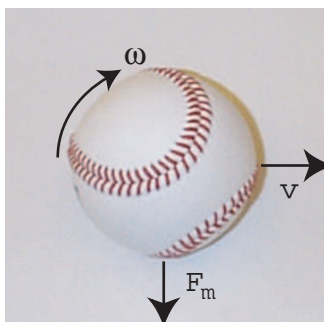


Figure 3.8: The Magnus force on a spinning ball pushes a ball with topspin down.

Of particular interest to baseball fans is the curve of balls in flight due to their rotation. This force was first investigated in 1850 by G. Magnus and the curvature of the trajectories of spinning objects is now known as the *Magnus effect*. It can be explained qualitatively by observing that the speed of the ball's surface relative to the air is different on opposite edges of the ball. If the drag force has the form $F_{\text{drag}} \sim v^2$, then the unbalanced force due to the difference in the velocity on opposite sides of the ball due to its rotation is given by

$$F_{\text{magnus}} \sim v\Delta v. \quad (3.26)$$

We can express the velocity difference in terms of the ball's angular velocity and radius and write

$$F_{\text{magnus}} \sim vr\omega. \quad (3.27)$$

The direction of the Magnus force is perpendicular to both the velocity and the rotation axis. For example, if we observe a ball moving to the right and rotating clockwise (that is, with topspin), then the velocity of the ball's surface relative to the air at the top, $v + \omega r$, is higher than the velocity at the bottom, $v - \omega r$. Because the larger velocity will produce a larger force, the Magnus effect will contribute a force in the downward direction. These considerations suggest that the Magnus force can be expressed as a vector product:

$$F_{\text{magnus}}/m = C_M(\boldsymbol{\omega} \times \mathbf{v}), \quad (3.28)$$

where m is the mass of the ball. The constant, C_M , depends on the radius of the ball, the viscosity of air, and other factors such as the orientation of the stitching. We will assume that the ball is rotating fast enough so that it can be modeled using an average value. (If the ball does not rotate, the pitcher has thrown a knuckleball.) The total force on the baseball is given by

$$\mathbf{F}/m = \mathbf{g} - C_D|\mathbf{v}|\mathbf{v} + C_M(\boldsymbol{\omega} \times \mathbf{v}). \quad (3.29)$$

Equation (3.29) leads to the following rates for the velocity components:

$$\frac{dv_x}{dt} = -C_Dvv_x + C_M(\omega_yv_z - \omega_zv_y) \quad (3.30a)$$

$$\frac{dv_y}{dt} = -C_Dvv_y + C_M(\omega_zv_x - \omega_xv_z) \quad (3.30b)$$

$$\frac{dv_z}{dt} = -C_Dvv_z + C_M(\omega_xv_y - \omega_yv_x) - g, \quad (3.30c)$$

where we will assume that $\boldsymbol{\omega}$ is a constant. The rate for each of the three position variables is the corresponding velocity. Typical parameter values for a 149 gram baseball are $C_D = 6 \times 10^{-3}$ and $C_M = 4 \times 10^{-4}$. See the book by Adair for a more complete discussion.

Problem 3.19. Curveballs

Model a spinning baseball using (3.30). Assume that the initial ball is released at $z = 1.8$ m above and $x = 18$ m from home plate. Set the initial angle above the horizontal, the initial speed, and the spin using fields in the user interface. Plot the vertical and horizontal deflection of the baseball as it travels toward home plate.

- First set the drag and Magnus forces to zero and test your program using analytical results for a 40 m/s fast ball. What initial angle is required for the pitch to pass over home plate at a height of 1.5 m?
- Add the drag force with $C_D = 6 \times 10^{-3}$. What initial angle is required for this pitch to be a strike assuming that the other initial conditions are unchanged? Plot the vertical deflection with and without drag for comparison.
- Add topspin to the pitch using a typical spin of $\omega_y = 200$ rad/s and $C_M = 4 \times 10^{-4}$. How much does topspin change the height of the ball as it passes over the plate? What about backspin?
- How much does a 35 m/s curve ball deflect if it is pitched with an initial spin of 200 rad/s?
- Add a 3D visualization of the baseball's trajectory to using a 3D trail to display the path of the ball.

Coupled three-dimensional equations of motion occur in electrodynamics when a charged particle travels through electric and magnetic fields. The equation of motion can be written in vector form as:

$$m\dot{\mathbf{v}} = q\mathbf{E} + q(\mathbf{v} \times \mathbf{B}), \quad (3.31)$$

where m is the mass of the particle, q is the charge, and \mathbf{E} and \mathbf{B} represent the electric and magnetic fields, respectively. For the special case of a constant magnetic field, the trajectory of a

charged particle is a spiral along the field lines with a cyclotron orbit whose period of revolution is $2\pi m/qB$. The addition of an electric field changes this motion dramatically.

The rates for the velocity components of a charged particle using units such that $m = q = 1$ are

$$\frac{dv_x}{dt} = E_x + v_y B_z - v_z B_y \quad (3.32a)$$

$$\frac{dv_y}{dt} = E_y + v_z B_x - v_x B_z \quad (3.32b)$$

$$\frac{dv_z}{dt} = E_z + v_x B_y - v_y B_x. \quad (3.32c)$$

The rate for each of the three position variables is again the corresponding velocity.

Problem 3.20. Motion in electric and magnetic fields

- Model the two-dimensional motion of a charged particle in a constant electric and magnetic field with the magnetic field in the \hat{z} direction and the electric field in the \hat{y} direction. Assume that the initial velocity is in the x - y plane.
- Why does the trajectory in part (a) remain in the x - y plane?
- In what direction does the charge particle drift if there is an electric field in the x direction and a magnetic field in the z direction if it starts at rest from the origin? What type of curve does the charged particle follow?
- Create a three-dimensional simulation of the trajectory of a particle in constant electric and magnetic fields. Verify that a charged particle undergoes spiral motion in a constant magnetic field and zero electric field. Predict the trajectory if an electric field is added and compare the results of the simulation to your prediction. Consider electric fields that are parallel to and perpendicular to the magnetic field.

Although the trajectory of a charged particle in constant electric and magnetic fields can be solved analytically, the trajectories in the presence of dipole fields cannot. A magnetic dipole with dipole moment $\mathbf{p} = |p|\hat{p}$ produces the following magnetic field:

$$\mathbf{B} = \frac{\mu_0 m}{4\pi\epsilon_0 r^3} [3\hat{p} \cdot \hat{r}]\hat{r} - \hat{p}. \quad (3.33)$$

(The distinction between the symbol p for the dipole moment and p for momentum should be clear from the context.)

Problem 3.21. Motion in a magnetic dipole field

Model the Earth's Van Allen radiation belt using the following formula for the dipole field:

$$\mathbf{B} = B_0 \left(\frac{R_E}{R}\right)^3 [3\hat{p} \cdot \hat{r}]\hat{r} - \hat{p}, \quad (3.34)$$

where R_E is the radius of the Earth and the magnetic field at the equator is $B_0 = 3.5 \times 10^{-5}$ tesla. Note that a 1 MeV electron at 2 Earth radii travels in very tight spirals with a cyclotron period that is much smaller than the travel time between the north and south poles. Better visual results can be obtained by raising the electron energies by a factor of ~ 1000 . Use classical dynamics, but include the relativistic dependence of the mass on the particle speed.

3.7 Levels of Simulation

So far we have considered models in which the microscopic complexity of the system of interest has been simplified considerably. Consider for example, the motion of a pebble falling through the air. First we reduced the complexity by representing the pebble as a particle with no internal structure. Then we reduced the number of degrees of freedom even more by representing the collisions of the pebble with the many molecules in the air by a velocity-dependent friction term. The resultant phenomenological model is a fairly accurate representation of realistic physical systems. However, what we gain in simplicity, we lose in range of applicability.

In a more detailed model, the individual physical processes would be represented microscopically. For example, we could imagine doing a simulation in which the effects of the air are represented by a fluid of particles that collide with one another and with the falling body. How accurately do we need to represent the potential energy of interaction between the fluid particles? Clearly the level of detail that is needed depends on the accuracy of the corresponding experimental data and the type of information in which we are interested. For example, we do not need to take into account the influence of the moon on a pebble falling near the Earth's surface. And the level of detail that we can simulate depends in part on the available computer resources.

The terms *simulation* and *modeling* are frequently used interchangeably. In this text, a model is a conceptual representation of a physical system and its properties and modeling is the process whereby we construct this representation. Computer modeling requires (1) a description and an analysis of the problem, (2) the identification of the variables and the algorithms, (3) the implementation on a specific hardware-software platform, (4) the execution of the implementation and analysis of the results, (5) refinement and generalization, and (6) the presentation of results. A simulation is an implementation of a model that allows us to test the model under different conditions with the objective of learning about the model's behavior. The applicability of the results of the simulation to those of the real (physical) system depends on how well the model describes reality.

3.8 Simulations

The following models are implemented in *EJS* and are downloadable from the OSP Collection in the compADRE digital library.

SHO Euler

The SHO Euler model solves the first-order differential equations for a particle attached to a Hooke's law spring using Euler's method. Euler's method is implemented explicitly on the Evolution workpanel in order to teach Java syntax. We use Euler's method to study the simple harmonic oscillator (SHO) because its analytic solution is well known and because its constantly varying acceleration produces easy to detect numerical errors without multi-variable force expressions. See Section 3.1.

SHO ODE Solver

The SHO ODE Solver model demonstrates how to use the *EJS* ODE editor to solve a differential equation. Examine this model in *EJS* and right-click on the Evolution workpanel tabs to enable and disable pages to test algorithms. The explicit evolution page is disabled and the ODE editor page is enabled in the default configuration. See Section 3.2.

Falling Particle

The Falling Particle model shows a falling ball and plots its position as a function of time. The ball falls with constant acceleration and click-dragging on the ball changes its height but leaves its velocity unchanged. The reset button stops the animation and sets the initial conditions. Users are encouraged to analyze the position data using the Data Tool. This model is a starting point for the exercises and problems in Section 3.3. Users can, for example, enable the zero crossing event in the Evolution workpanel to compute the maximum height of the particle.

Falling 3D Particle

The Falling 3D Particle model simulates the physics free fall with a 3D view. The ball falls with constant acceleration in the z direction. Click-dragging on the ball changes its height but leaves its velocity unchanged. The reset button stops the simulation and sets the initial conditions. An ODE event is used to reverse the velocity when the ball reaches the floor and the coefficient of restitution reduces the speed of the ball after the bounce. The time between bounces decreases but this "Zeno effect" is resolved by removing the acceleration if the ball is resting on the floor. See Section 3.6.

Appendix 3A: Accuracy and Stability

Now that we have learned how to use numerical methods to find numerical solutions to simple first-order differential equations, we need to develop some practical guidelines to help us estimate the accuracy of the various methods. Because we have replaced a differential equation by a difference equation, our numerical solution is not identically equal to the true solution of the original differential equation, except for special cases. The discrepancy between the two solutions has two causes. One cause is that computers do not store numbers with infinite precision, but rather to a maximum number of digits that is hardware and software dependent. As we have seen, Java allows the programmer to distinguish between *floating point* numbers, that is, numbers with decimal points, and *integer* numbers. Arithmetic with numbers represented by integers is exact, but we cannot solve a differential equation using integer arithmetic. Arithmetic operations involving floating point numbers, such as addition and multiplication, introduce *roundoff error*. For example, if a computer only stored floating point numbers to two significant figures, the product 2.1×3.2 would be stored as 6.7 rather than 6.72. The significance of roundoff errors is that they accumulate as the number of mathematical operations increases. Ideally, we should choose algorithms that do not significantly magnify the roundoff error, for example, we should avoid subtracting numbers that are nearly the same in magnitude.

The other source of the discrepancy between the true answer and the computed answer is the error associated with the choice of algorithm. This error is called the *truncation error*. A truncation error would exist even on an idealized computer that stored floating point numbers with infinite precision and hence had no roundoff error. Because the truncation error depends on the choice of algorithm and can be controlled by the programmer, you should be motivated to learn more about numerical analysis and the estimation of truncation errors. However, there is no general prescription for the best algorithm for obtaining numerical solutions of differential equations. We will find in later chapters that the various algorithms have advantages and disadvantages, and the appropriate selection depends on the nature of the solution, which you might not know in advance, and on your objectives. How accurate must the answer be? Over how large an interval do you need the solution? What kind of computer(s) are you using? How much computer time and personal time do you have?

In practice, we usually can determine the accuracy of a numerical solution by reducing the value of Δt until the numerical solution is unchanged at the desired level of accuracy. Of course, we have to be careful not to make Δt too small, because too many steps would be required and the computation time and roundoff error would increase.

In addition to accuracy, another important consideration is the *stability* of an algorithm. It might happen that the numerical results are very good for short times, but diverge from the true solution for longer times. This divergence might occur if small errors in the algorithm are multiplied many times, causing the error to grow geometrically. Such an algorithm is said to be *unstable* for the particular problem. We consider the accuracy and the stability of the Euler algorithm in Problems 3.22 and 3.23.

Problem 3.22. Accuracy of the Euler algorithm

- a. Use the Euler algorithm to compute the numerical solution of $dy/dx = 2x$ with $y = 0$ at $x = 0$ and $\Delta x = 0.1, 0.05, 0.025, 0.01,$ and 0.005 . Make a table showing the difference between the exact solution and the numerical solution. Is the difference between these solutions a decreasing function of Δx ? That is, if Δx is decreased by a factor of two, how does the difference change? Plot the difference as a function of Δx . If your points fall approximately on a straight line, then the difference is proportional to Δx (for $\Delta x \ll 1$). The numerical method is called *n*th order if the difference between the analytical solution and the numerical solution is proportional to $(\Delta x)^n$ for a fixed value of x . What is the order of the Euler algorithm?
- b. One way to determine the accuracy of a numerical solution is to repeat the calculation with a smaller step size and compare the results. If the two calculations agree to p decimal places, we can reasonably assume that the results are correct to p decimal places. What value of Δx is necessary for 0.1% accuracy at $x = 2$? What value of Δx is necessary for 0.1% accuracy at $x = 4$?

Problem 3.23. Stability of the Euler algorithm

- a. Consider the differential equation

$$R \frac{dQ}{dt} = -\frac{Q}{C}. \quad (3.35)$$

with $Q = 10$ at $t = 0$. This equation represents the discharge of a capacitor C through a resistor R . Choose $R = 2000 \Omega$ and $C = 10^{-6}$ farads. Do you expect $Q(t)$ to increase or decrease with

- t ? Does $Q(t)$ change indefinitely or does it reach a steady-state value? Create a model to solve (3.35) numerically using the Euler algorithm. What value of Δt is necessary to obtain three decimal accuracy at $t = 0.005$?
- b. What is the nature of your numerical solution to (3.35) at $t = 0.05$ for $\Delta t = 0.005, 0.0025,$ and 0.001 ? Does a small change in Δt lead to a large change in the computed value of Q ? Is the Euler algorithm stable for reasonable values of Δt ?

Appendix 3B: Numerical Integration of Newton's Equation of Motion

We summarize several of the common finite difference methods for the solution of Newton's equations of motion with continuous force functions. The number and variety of algorithms currently in use is evidence that no single method is superior under all conditions.

To simplify the notation, we consider the motion of a particle in one dimension and write Newton's equations of motion in the form

$$\frac{dv}{dt} = a(t), \quad (3.36a)$$

$$\frac{dx}{dt} = v(t), \quad (3.36b)$$

where $a(t) \equiv a(x(t), v(t), t)$. The goal of finite difference methods is to determine the values of x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + \Delta t$. We already have seen that Δt must be chosen so that the integration method generates a stable solution. If the system is conservative, Δt must be sufficiently small so that the total energy is conserved to the desired accuracy.

The nature of many of the integration algorithms can be understood by expanding $v_{n+1} = v(t_n + \Delta t)$ and $x_{n+1} = x(t_n + \Delta t)$ in a Taylor series. We write

$$v_{n+1} = v_n + a_n \Delta t + O((\Delta t)^2), \quad (3.37a)$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O((\Delta t)^3). \quad (3.37b)$$

The familiar Euler algorithm is equivalent to retaining the $O(\Delta t)$ terms in (3.37):

$$v_{n+1} = v_n + a_n \Delta t \quad (3.38a)$$

$$x_{n+1} = x_n + v_n \Delta t. \quad (\text{Euler algorithm}) \quad (3.38b)$$

Because order Δt terms are retained in (3.38), the local truncation error, the error in one time step, is order $(\Delta t)^2$. The global error, the total error over the time of interest, due to the accumulation of errors from step to step is order Δt . This estimate of the global error follows from the fact that the number of steps into which the total time is divided is proportional to $1/\Delta t$. Hence, the order of the global error is reduced by a factor of $1/\Delta t$ relative to the local error. We say that an algorithm is n th order if its global error is order $(\Delta t)^n$. The Euler algorithm is an example of a *first-order* algorithm.

The Euler algorithm is asymmetrical because it advances the solution by a time step Δt , but uses information about the derivative only at the beginning of the interval. We already have found that the accuracy of the Euler algorithm is limited and that frequently its solutions are not stable. We also found that a simple modification of (3.38) yields solutions that are stable for oscillatory systems. For completeness, we repeat the Euler-Cromer algorithm here:

$$v_{n+1} = v_n + a_n \Delta t, \quad (3.39a)$$

$$x_{n+1} = x_n + v_{n+1} \Delta t. \quad (\text{Euler-Cromer algorithm}) \quad (3.39b)$$

An obvious way to improve the Euler algorithm is to use the mean velocity during the interval to obtain the new position. The corresponding *midpoint* algorithm can be written as

$$v_{n+1} = v_n + a_n \Delta t, \quad (3.40a)$$

and

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n) \Delta t. \quad (\text{midpoint algorithm}) \quad (3.40b)$$

Note that if we substitute (3.40a) for v_{n+1} into (3.40b), we obtain

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2. \quad (3.41)$$

Hence, the midpoint algorithm yields second-order accuracy for the position and first-order accuracy for the velocity. Although the midpoint approximation yields exact results for constant acceleration, it usually does not yield much better results than the Euler algorithm. In fact, both algorithms are equally poor, because the errors increase with each time step.

A higher order algorithm whose error is bounded is the *half-step* algorithm. In this algorithm the average velocity during an interval is taken to be the velocity in the middle of the interval. The half-step algorithm can be written as

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a_n \Delta t, \quad (3.42a)$$

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} \Delta t. \quad (\text{half-step algorithm}) \quad (3.42b)$$

Note that the half-step algorithm is not self-starting, that is, (3.42a) does not allow us to calculate $v_{\frac{1}{2}}$. This problem can be overcome by adopting the Euler algorithm for the first half step:

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2} a_0 \Delta t. \quad (3.42c)$$

Because the half-step algorithm is stable, it is a common textbook algorithm. The Euler-Richardson algorithm, a widely used half-step algorithm, can be motivated as follows. We first write $x(t + \Delta t)$ as

$$x_1 \approx x(t + \Delta t) = x(t) + v(t) \Delta t + \frac{1}{2} a(t) (\Delta t)^2. \quad (3.43)$$

The notation x_1 implies that $x(t + \Delta t)$ is related to $x(t)$ by one time step. We also may divide the step Δt into half steps and write the first half step, $x(t + \frac{1}{2} \Delta t)$, as

$$x(t + \frac{1}{2} \Delta t) \approx x(t) + v(t) \frac{\Delta t}{2} + \frac{1}{2} a(t) \left(\frac{\Delta t}{2}\right)^2. \quad (3.44)$$

The second half step, $x_2(t + \Delta t)$, may be written as

$$x_2(t + \Delta t) \approx x(t + \frac{1}{2}\Delta t) + v(t + \frac{1}{2}\Delta t)\frac{\Delta t}{2} + \frac{1}{2}a(t + \frac{1}{2}\Delta t)(\frac{\Delta t}{2})^2. \quad (3.45)$$

We substitute (3.44) into (3.45) and obtain

$$x_2(t + \Delta t) \approx x(t) + \frac{1}{2}[v(t) + v(t + \frac{1}{2}\Delta t)]\Delta t + \frac{1}{2}[a(t) + a(t + \frac{1}{2}\Delta t)](\frac{1}{2}\Delta t)^2. \quad (3.46)$$

Now $a(t + \frac{1}{2}\Delta t) \approx a(t) + \frac{1}{2}a'(t)\Delta t$. Hence to order $(\Delta t)^2$, (3.46) becomes

$$x_2(t + \Delta t) = x(t) + \frac{1}{2}[v(t) + v(t + \frac{1}{2}\Delta t)]\Delta t + \frac{1}{2}[2a(t)](\frac{1}{2}\Delta t)^2. \quad (3.47)$$

We can find an approximation that is accurate to order $(\Delta t)^3$ by combining (3.43) and (3.47) so that the terms to order $(\Delta t)^2$ cancel. The combination that works is $2x_2 - x_1$, which gives the Euler-Richardson result:

$$x_{\text{er}}(t + \Delta t) \equiv 2x_2(t + \Delta t) - x_1(t + \Delta t) = x(t) + v(t + \frac{1}{2}\Delta t)\Delta t + O(\Delta t)^3. \quad (3.48)$$

The same reasoning leads to an approximation for the velocity accurate to $(\Delta t)^3$ giving

$$v_{\text{er}} \equiv 2v_2(t + \Delta t) - v_1(t + \Delta t) = v(t) + a(t + \frac{1}{2}\Delta t)\Delta t + O(\Delta t)^3. \quad (3.49)$$

A bonus of the Euler-Richardson algorithm is that the quantities $|x_2 - x_1|$ and $|v_2 - v_1|$ give an estimate for the error. We can use these estimates to change the time step so that the error is always within some desired level of precision. We will see that the Euler-Richardson algorithm is equivalent to the second-order Runge-Kutta algorithm (see (3.59)).

One of the most common energy drift-free higher order algorithms is commonly attributed to Verlet. We write the Taylor series expansion for x_{n-1} in a form similar to (3.37b):

$$x_{n-1} = x_n - v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2. \quad (3.50)$$

If we add the forward and reverse forms, (3.37b) and (3.50) respectively, we obtain

$$x_{n+1} + x_{n-1} = 2x_n + a_n(\Delta t)^2 + O((\Delta t)^4) \quad (3.51)$$

or

$$x_{n+1} = 2x_n - x_{n-1} + a_n(\Delta t)^2. \quad (\text{leapfrog algorithm}) \quad (3.52a)$$

Similarly, the subtraction of the Taylor series for x_{n+1} and x_{n-1} yields

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t}. \quad (\text{leapfrog algorithm}) \quad (3.52b)$$

Note that the global error associated with the leapfrog algorithm (3.52) is third-order for the position and second-order for the velocity. However, the velocity plays no part in the integration

of the equations of motion. The leapfrog algorithm is also known as the explicit central difference algorithm. Because this form of the leapfrog algorithm is not self-starting, another algorithm must be used to obtain the first few terms. An additional problem is that the new velocity (3.52b) is found by computing the difference between two quantities of the same order of magnitude. Such an operation results in a loss of numerical precision and may give rise to roundoff errors.

A mathematically equivalent version of the leapfrog algorithm is given by

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 \quad (3.53a)$$

$$v_{n+1} = v_n + \frac{1}{2} (a_{n+1} + a_n) \Delta t. \quad (\text{velocity Verlet algorithm}) \quad (3.53b)$$

We see that (3.53), known as the *velocity* form of the Verlet algorithm, is self-starting and minimizes roundoff errors. Because we will not use (3.52) in the text, we will refer to (3.53) as the Verlet algorithm.

We can derive (3.53) from (3.52) by the following considerations. We first solve (3.52b) for x_{n-1} and write $x_{n-1} = x_{n+1} - 2v_n \Delta t$. If we substitute this expression for x_{n-1} into (3.52a) and solve for x_{n+1} , we find the form (3.53a). Then we use (3.52b) to write v_{n+1} as:

$$v_{n+1} = \frac{x_{n+2} - x_n}{2\Delta t}, \quad (3.54)$$

and use (3.52a) to obtain $x_{n+2} = 2x_{n+1} - x_n + a_{n+1}(\Delta t)^2$. If we substitute this form for x_{n+2} into (3.54), we obtain

$$v_{n+1} = \frac{x_{n+1} - x_n}{\Delta t} + \frac{1}{2} a_{n+1} \Delta t. \quad (3.55)$$

Finally, we use (3.53a) for x_{n+1} to eliminate $x_{n+1} - x_n$ from (3.55); after some algebra we obtain the desired result (3.53b).

Another useful algorithm that avoids the roundoff errors of the leapfrog algorithm is due to Beeman and Schofield. We write the *Beeman* algorithm in the form:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{6} (4a_n - a_{n-1}) (\Delta t)^2 \quad (3.56a)$$

$$v_{n+1} = v_n + \frac{1}{6} (2a_{n+1} + 5a_n - a_{n-1}) \Delta t. \quad (\text{Beeman algorithm}) \quad (3.56b)$$

Note that (3.56) does not calculate particle trajectories more accurately than the Verlet algorithm. Its advantage is that it generally does a better job of maintaining energy conservation. However, the Beeman algorithm is not self-starting.

The most common finite difference method for solving ordinary differential equations is the *Runge-Kutta* method. To explain the many algorithms based on this method, we consider the solution of the first-order differential equation

$$\frac{dx}{dt} = f(x, t). \quad (3.57)$$

Runge-Kutta algorithms evaluate the rate, $f(x, t)$, multiple times in the interval $[t, t + \Delta t]$. For example, the classic fourth-order Runge-Kutta algorithm, which we will discuss in the following,

evaluates $f(x, t)$ at four times t_n , $t_n + a_1\Delta t$, $t_n + a_2\Delta t$, and $t_n + a_3\Delta t$. Each evaluation of $f(x, t)$ produces a slightly different rate r_1 , r_2 , r_3 , and r_4 . The idea is to advance the solution using a weighted average of the intermediate rates:

$$y_{n+1} = y_n + (c_1r_1 + c_2r_2 + c_3r_3 + c_4r_4)\Delta t. \quad (3.58)$$

The various Runge-Kutta algorithms correspond to different choices for the constants a_i and c_i . These algorithms are classified by the number of intermediate rates $\{r_i, i = 1, \dots, N\}$. The determination of the Runge-Kutta coefficients is difficult for all but the lowest order methods, because the coefficients must be chosen to cancel as many terms in the Taylor series expansion of $f(x, t)$ as possible. The first non-zero expansion coefficient determines the order of the Runge-Kutta algorithm. Fortunately, these coefficients are tabulated in most numerical analysis books.

To illustrate how the various sets of Runge-Kutta constants arise, consider the case $N = 2$. The second-order Runge-Kutta solution of (3.57) can be written using standard notation as:

$$x_{n+1} = x_n + k_2 + O((\Delta t)^3), \quad (3.59a)$$

where

$$k_2 = f\left(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t. \quad (3.59b)$$

$$k_1 = f(x_n, t_n)\Delta t \quad (3.59c)$$

Note that the weighted average uses $c_1 = 0$ and $c_2 = 1$. The interpretation of (3.59) is as follows. The Euler algorithm assumes that the slope $f(x_n, t_n)$ at (x_n, t_n) can be used to extrapolate to the next step, that is, $x_{n+1} = x_n + f(x_n, t_n)\Delta t$. A plausible way of making a more accurate determination of the slope is to use the Euler algorithm to extrapolate to the midpoint of the interval and then to use the midpoint slope across the full width of the interval. Hence, the Runge-Kutta estimate for the rate is $f(x^*, t_n + \frac{1}{2}\Delta t)$, where $x^* = x_n + \frac{1}{2}f(x_n, t_n)\Delta t$ (see (3.59c)).

The application of the second-order Runge-Kutta algorithm to Newton's equation of motion (3.36) yields

$$k_{1v} = a_n(x_n, v_n, t_n)\Delta t \quad (3.60a)$$

$$k_{1x} = v_n\Delta t \quad (3.60b)$$

$$k_{2v} = a\left(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t + \frac{\Delta t}{2}\right)\Delta t \quad (3.60c)$$

$$k_{2x} = \left(v_n + \frac{k_{1v}}{2}\right)\Delta t, \quad (3.60d)$$

and

$$v_{n+1} = v_n + k_{2v} \quad (3.61a)$$

$$x_{n+1} = x_n + k_{2x}. \quad (\text{second-order Runge Kutta}) \quad (3.61b)$$

Note that the second-order Runge-Kutta algorithm in (3.60) and (3.61) is identical to the Euler-Richardson algorithm.

Other second-order Runge-Kutta type algorithms exist. For example, if we set $c_1 = c_2 = \frac{1}{2}$ we obtain the endpoint method:

$$y_{n+1} = y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2 \quad (3.62a)$$

where

$$k_1 = f(x_n, t_n)\Delta t \quad (3.62b)$$

$$k_2 = f(x_n + k_1, t_n + \Delta t)\Delta t. \quad (3.62c)$$

And if we set $c_1 = \frac{1}{3}$ and $c_2 = \frac{2}{3}$, we obtain Ralston's method:

$$y_{n+1} = y_n + \frac{1}{3}k_1 + \frac{2}{3}k_2 \quad (3.63a)$$

where

$$k_1 = f(x_n, t_n)\Delta t \quad (3.63b)$$

$$k_2 = f(x_n + \frac{3}{4}k_1, t_n + \frac{3}{4}\Delta t)\Delta t. \quad (3.63c)$$

Note that Ralston's method does not calculate the rate at uniformly spaced subintervals of Δt . In general, a Runge-Kutta method adjusts the partition of Δt as well as the constants a_i and c_i so as to minimize the error.

In the *fourth-order* Runge-Kutta algorithm, the derivative is computed at the beginning of the time interval, in two different ways at the middle of the interval, and again at the end of the interval. The two estimates of the derivative at the middle of the interval are given twice the weight of the other two estimates. The algorithm for the solution of (3.57) can be written in standard notation as

$$k_1 = f(x_n, t_n)\Delta t \quad (3.64a)$$

$$k_2 = f(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t \quad (3.64b)$$

$$k_3 = f(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2})\Delta t \quad (3.64c)$$

$$k_4 = f(x_n + k_3, t_n + \Delta t)\Delta t, \quad (3.64d)$$

and

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \quad (3.65)$$

The application of the fourth-order Runge-Kutta algorithm to Newton's equation of motion

(3.36) yields

$$k_{1v} = a(x_n, v_n, t_n)\Delta t \quad (3.66a)$$

$$k_{1x} = v_n\Delta t \quad (3.66b)$$

$$k_{2v} = a\left(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (3.66c)$$

$$k_{2x} = \left(v_n + \frac{k_{1v}}{2}\right)\Delta t \quad (3.66d)$$

$$k_{3v} = a\left(x_n + \frac{k_{2x}}{2}, v_n + \frac{k_{2v}}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (3.66e)$$

$$k_{3x} = \left(v_n + \frac{k_{2v}}{2}\right)\Delta t \quad (3.66f)$$

$$k_{4v} = a(x_n + k_{3x}, v_n + k_{3v}, t + \Delta t) \quad (3.66g)$$

$$k_{4x} = (v_n + k_{3x})\Delta t, \quad (3.66h)$$

and

$$v_{n+1} = v_n + \frac{1}{6}(k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}) \quad (3.67a)$$

$$x_{n+1} = x_n + \frac{1}{6}(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}). \quad (\text{fourth-order Runge-Kutta}) \quad (3.67b)$$

Because Runge-Kutta algorithms are self-starting, they are frequently used to obtain the first few iterations for an algorithm that is not self-starting.

As we have discussed, one way to determine the accuracy of a solution is to calculate it twice with two different values of the time step. One way to make this comparison is to choose time steps Δt and $\Delta t/2$ and compare the solution at the desired time. If the difference is small, the error is assumed to be small. This estimate of the error can be used to adjust the value of the time step. If the error is too large, then the time step can be halved. And if the error is much less than the desired value, the time step can be increased so that the program runs faster.

A better way of controlling the step size was developed by Fehlberg who showed that it is possible to evaluate the rate in such a way as to simultaneously obtain two Runge-Kutta approximations with different orders. For example, it is possible to run a fourth-order and fifth-order algorithm in tandem by evaluating five rates. We thus obtain different estimates of the true solution using different weighed averages of these rates:

$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 \quad (3.68a)$$

$$y_{n+1}^* = y_n + c_1^*k_1 + c_2^*k_2 + c_3^*k_3 + c_4^*k_4. \quad (3.68b)$$

Because we can assume that the fifth-order solution is closer to the true solution than the fourth order algorithm, the difference $|y - y^*|$ provides a good estimate of the error of the fourth-order method. If this estimated error is larger than the desired tolerance, then the step size is decreased. If the error is smaller than the desired tolerance, the step size is increased. The RK45 ODE solver as well as the one on the *EJS* ODE page implements this technique for choosing the optimal step size.

In applications where the accuracy of the numerical solution is important, adaptive time step algorithms should always be used. As stated in *Numerical Recipes*: “Many small steps should

tiptoe through treacherous terrain, while a few great strides should speed through uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more.”

Adaptive step size algorithms are not well suited for tabulating functions or for simulation because the intervals between data points are not constant. An easy way to circumvent this problem is to use a method that takes multiple adaptive steps while checking to insure that the last step does not overshoot the desired fixed step size. The `ODEMultistepSolver` implements this technique. The solver acts like a fixed step size solver, even though the solver monitors its internal step size so as to achieve the desired accuracy.

It also is possible to combine the results from a calculation using two different values of the time step to yield a more accurate expression. Consider the Taylor series expansion of $f(t + \Delta t)$ about t :

$$f(t + \Delta t) = f(t) + f'(t)\Delta t + \frac{1}{2!}f''(t)(\Delta t)^2 + \dots \quad (3.69)$$

Similarly, we have

$$f(t - \Delta t) = f(t) - f'(t)\Delta t + \frac{1}{2!}f''(t)(\Delta t)^2 + \dots \quad (3.70)$$

We subtract (3.70) from (3.69) to find the usual central difference approximation for the derivative

$$f'(t) \approx D_1(\Delta t) = \frac{f(t + \Delta t) - f(t - \Delta t)}{2\Delta t} - \frac{(\Delta t)^2}{6}f'''(t). \quad (3.71)$$

The truncation error is order $(\Delta t)^2$. Next consider the same relation, but for a time step that is twice as large.

$$f'(t) \approx D_1(2\Delta t) = \frac{f(t + 2\Delta t) - f(t - 2\Delta t)}{4\Delta t} - \frac{4(\Delta t)^2}{6}f'''(t). \quad (3.72)$$

Note that the truncation error is again order $(\Delta t)^2$, but is four times bigger. We can eliminate this error to leading order by dividing (3.72) by 4 and subtracting it from (3.71):

$$f'(t) - \frac{1}{4}f'(t) = \frac{3}{4}f'(t) \approx D_1(\Delta t) - \frac{1}{4}D_1(2\Delta t),$$

or

$$f'(t) \approx \frac{4D_1(\Delta t) - D_1(2\Delta t)}{3}. \quad (3.73)$$

It is easy to show that the error for $f'(t)$ is order $(\Delta t)^4$. Recursive difference formulas for derivatives can be obtained by canceling the truncation error at each order. This method is called *Richardson extrapolation*.

Another class of algorithms are *predictor-corrector* algorithms. The idea is to first *predict* the value of the new position:

$$x_p = x_{n-1} + 2v_n\Delta t. \quad (\text{predictor}) \quad (3.74)$$

The predicted value of the position allows us to predict the acceleration a_p . Then using a_p , we obtain the *corrected* values of v_{n+1} and x_{n+1} :

$$v_{n+1} = v_n + \frac{1}{2}(a_p + a_n)\Delta t \quad (3.75a)$$

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n)\Delta t. \quad (\text{corrected}) \quad (3.75b)$$

The corrected values of x_{n+1} and v_{n+1} are used to obtain a new predicted value of a_{n+1} , and hence a new predicted value of v_{n+1} and x_{n+1} . This process is repeated until the predicted and corrected values of x_{n+1} differ by less than the desired value.

Note that the predictor-corrector algorithm is not self-starting. The predictor-corrector algorithm gives more accurate positions and velocities than the leapfrog algorithm and is suitable for very accurate calculations. However, it is computationally expensive, needs significant storage (the forces at the last two stages, and the coordinates and velocities at the last step), and becomes unstable for large time steps.

As we have emphasized, there is no single algorithm for solving Newton's equations of motion that is superior under all conditions. It is usually a good idea to start with a simple algorithm, and then to try a higher order algorithm to see if any real improvement is obtained.

We now discuss an important class of algorithms, known as *symplectic* algorithms, which are particularly suitable for doing long time simulations of Newton's equations of motion when the force is only a function of position. The basic idea of these algorithms derives from the Hamiltonian theory of classical mechanics. We first give some basic results needed from this theory to understand the importance of symplectic algorithms.

In Hamiltonian theory the generalized coordinates, q_i , and momenta, p_i , take the place of the usual positions and velocities familiar from Newtonian theory. The index i labels both a particle and a component of the motion. For example, in a two particle system in two dimensions, i would run from 1 to 4. The Hamiltonian (which for our purposes can be thought of as the total energy) is written as

$$H(q_i, p_i) = T + V, \quad (3.76)$$

where T is the kinetic energy and V is the potential energy. Hamilton's theory is most relevant for non-dissipative systems, which we consider here. For example, for a two particle system in two dimensions connected by a spring, H would take the form:

$$H = \frac{p_1^2}{2m} + \frac{p_2^2}{2m} + \frac{p_3^2}{2m} + \frac{p_4^2}{2m} + \frac{1}{2}k(q_1 - q_3)^2 + \frac{1}{2}k(q_2 - q_4)^2, \quad (3.77)$$

where if the particles are labeled as A and B , we have $p_1 = p_{x,A}$, $p_2 = p_{y,A}$, $p_3 = p_{x,B}$, $p_4 = p_{y,B}$, and similarly for the q_i . The equations of motion are written as first-order differential equations known as Hamilton's equations:

$$\dot{p}_i = -\frac{\partial H}{\partial q_i} \quad (3.78a)$$

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad (3.78b)$$

which are equivalent to Newton's second law and an equation relating the velocity to the momentum. The beauty of Hamiltonian theory is that these equations are correct for other coordinate systems such as polar coordinates, and they also describe rotating systems where the momenta become angular momenta, and the position coordinates become angles.

Because the coordinates and momenta are treated on an equal footing, we can consider the properties of flow in phase space where the dimension of phase space includes both the coordinates and momenta. Thus, one particle moving in one dimension corresponds to a two-dimensional phase space. If we imagine a collection of initial conditions in phase space forming a volume in phase space, then one of the results of Hamiltonian theory is that this volume does not change as the system evolves. A slightly different result, called the *symplectic* property, is that the sum of the areas formed by the projection of the phase space volume onto the planes, q_i, p_i , for each pair of coordinates and momenta also does not change with time. Numerical algorithms that have this property are called symplectic algorithms. These algorithms are built from the following two statements which are repeated M times for each time step.

$$p_i^{(k+1)} = p_i^{(k)} + a_k F_i^{(k)} \delta t \quad (3.79a)$$

$$q_i^{(k+1)} = q_i^{(k)} + b_k p_i^{(k+1)} \delta t, \quad (3.79b)$$

where $F_i^{(k)} \equiv -\partial V(q_i^{(k)})/\partial q_i^{(k)}$. The label k runs from 0 to $M - 1$ and one time step is given by $\Delta t = M\delta t$. (We will see that δt is the time step of an intermediate calculation that is made during the time step Δt .) Note that in the update for q_i , the already updated p_i is used. For simplicity, we assume that the mass is unity.

Different algorithms correspond to different values of M , a_k , and b_k . For example, $a_0 = b_0 = M = 1$ corresponds to the Euler-Cromer algorithm, and $M = 2$, $a_0 = a_1 = 1$, $b_0 = 2$, and $b_1 = 0$ is equivalent to the Verlet algorithm as we will now show. If we substitute in the appropriate values for a_k and b_k into (3.79), we have

$$p_i^{(1)} = p_i^{(0)} + F_i^{(0)} \delta t \quad (3.80a)$$

$$q_i^{(1)} = q_i^{(0)} + 2p_i^{(1)} \delta t \quad (3.80b)$$

$$p_i^{(2)} = p_i^{(1)} + F_i^{(1)} \delta t \quad (3.80c)$$

$$q_i^{(2)} = q_i^{(1)} \quad (3.80d)$$

We next combine (3.80a) and (3.80c) for the momentum coordinate and (3.80b) and (3.80d) for the position, and obtain

$$p_i^{(2)} = p_i^{(0)} + (F_i^{(0)} + F_i^{(1)}) \delta t \quad (3.81a)$$

$$q_i^{(2)} = q_i^{(0)} + 2p_i^{(1)} \delta t. \quad (3.81b)$$

We take $\delta t = \Delta t/2$ and combine (3.81b) with (3.80a) and find

$$p_i^{(2)} = p_i^{(0)} + \frac{1}{2}(F_i^{(0)} + F_i^{(1)})\Delta t \quad (3.82a)$$

$$q_i^{(2)} = q_i^{(0)} + p_i^{(0)}\Delta t + \frac{1}{2}F_i^{(0)}(\Delta t)^2, \quad (3.82b)$$

which is identical to the Verlet algorithm, (3.53), because for unit mass the force and acceleration are equal.

Reversing the order of the updates for the coordinates and the momenta also leads to symplectic algorithms:

$$q_i^{(k+1)} = q_i^{(k)} + b_k \delta t p_i^{(k)}, \quad (3.83a)$$

$$p_i^{(k+1)} = p_i^{(k)} + a_k \delta t F_i^{(k+1)} \quad (3.83b)$$

A third variation uses (3.79) and (3.83) for different values of k in one algorithm. Thus, if $M = 2$, which corresponds to two intermediate calculations per time step, we could use (3.79) for the first intermediate calculation and (3.83) for the second.

Why are these algorithms important? Because of the symplectic property, these algorithms will simulate an exact Hamiltonian, although not the one we started with in general (see Problem 3.2, for example). However, this Hamiltonian will be close to the one we wish to simulate if the a_k and b_k are properly chosen.

References and Suggestions for Further Reading

- F. S. Acton, *Numerical Methods That Work*, The Mathematical Association of America (1990), Chapter 5.
- Robert. K. Adair, *The Physics of Baseball*, third edition, Harper Collins (2002).
- Byron L. Coulter and Carl G. Adler, “Can a body pass a body falling through the air?,” *Am. J. Phys.* **47**, 841–846 (1979). The authors discuss the limiting conditions for which the drag force is linear or quadratic in the velocity.
- Alan Cromer, “Stable solutions using the Euler approximation,” *Am. J. Phys.* **49**, 455–459 (1981). The author shows that a minor modification of the usual Euler approximation yields stable solutions for oscillatory systems including planetary motion and the harmonic oscillator (see Chapter 4).
- Paul L. DeVries, *A First Course in Computational Physics*, John Wiley & Sons (1994).
- Denis Donnelly and Edwin Rogers, “Symplectic integrators: An introduction,” *Am. J. Phys.*, to be published.
- A. P. French, *Newtonian Mechanics*, W. W. Norton & Company (1971). Chapter 7 has an excellent discussion of air resistance and a detailed analysis of motion in the presence of drag resistance.
- Ian R. Gatland, “Numerical integration of Newton’s equations including velocity-dependent forces,” *Am J. Phys.* **62**, 259–265 (1994). The author discusses the Euler-Richardson algorithm.
- Stephen K. Gray, Donald W. Noid, and Bobby G. Sumpter, “Symplectic integrators for large scale molecular dynamics simulations: A comparison of several explicit methods,” *J. Chem. Phys.* **101** (5), 4062–4072 (1994).

- Margaret Greenwood, Charles Hanna, and John Milton, “Air resistance acting on a sphere: Numerical analysis, strobe photographs, and videotapes,” *Phys. Teacher* **24**, 153–159 (1986). More experimental data and theoretical analysis are given for the fall of ping-pong and styrofoam balls. Also see Mark Peastrel, Rosemary Lynch, and Angelo Armenti, “Terminal velocity of a shuttlecock in vertical fall,” *Am. J. Phys.* **48**, 511–513 (1980).
- Michael J. Kallaher, editor *Revolutions in Differential Equations: Exploring ODEs with Modern Technology*, The Mathematical Association of America (1999).
- K. S. Krane, “The falling raindrop: variations on a theme of Newton,” *Am. J. Phys.* **49**, 113–117 (1981). The author discusses the problem of mass accretion by a drop falling through a cloud of droplets.
- George C. McGuire, “Using computer algebra to investigate the motion of an electric charge in magnetic and electric dipole fields,” *Am. J. Phys.* **71** (8), 809–812 (2003).
- Rabindra Mehta, “Aerodynamics of sports balls,” in *Ann. Rev. Fluid Mech.* **17**, 151–189 (1985).
- Neville de Mestre, *The Mathematics of Projectiles in Sport*, Cambridge University Press (1990). The emphasis of this text is on solving many problems in projectile motion, for example, baseball, basketball, and golf, in the context of mathematical modeling. Many references to the relevant literature are given.
- Tao Pang, *Computational Physics*, Cambridge University Press (1997).
- William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992). Chapter 16 discusses the integration of ordinary differential equations.
- Emilio Segré, *Nuclei and Particles*, second edition, W. A. Benjamin (1977). Chapter 5 discusses decay cascades. The decay schemes described briefly in Problem 3.17 are taken from C. M. Lederer, J. M. Hollander, and I. Perlman, *Table of Isotopes*, sixth edition, John Wiley & Sons (1967).
- Lawrence F. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman and Hall (1994).