Exercises

3.12 **Solving differential equations: the pendulum.**[1] [2] (Computation) ④

Physical systems usually evolve continuously in time; their laws of motion are differential equations. Computer simulations must approximate these differential equations using discrete time steps. In this exercise, we will introduce the most common and important method used for molecular dynamics simulations, together with fancier techniques used for solving more general differential equations.

We will use these methods to solve for the dynamics of a pendulum:

$$\frac{d^2\theta}{dt^2} = \ddot{\theta} = -\frac{g}{L}\sin(\theta). \tag{1}$$

This equation gives the motion of a pendulum with a point mass at the tip of a massless rod[3] of length L. You may wish to rederive it using a free-body diagram.

Go to our web site [129] and download the pendulum files for the language you will be using. The animation should show a pendulum oscillating from an initial condition $\theta_0 = 2\pi/3$, $\dot{\theta} = 0$; the equations being solved have $g = 9.8\,\mathrm{m/s^2}$ and $L = 1\,\mathrm{m}$.

There are three independent criteria for picking a good algorithm for solving differential equations: *fidelity*, *accuracy*, and *stability*.

Fidelity. In our time step algorithm, we do not make the straightforward choice—using the current $(\theta(t), \dot{\theta}(t))$ to produce $(\theta(t+\delta), \dot{\theta}(t+\delta))$. Rather, we use a staggered algorithm: $\theta(t)$ determines the acceleration and the update $\dot{\theta}(t) \to \dot{\theta}(t+\delta)$, and then $\dot{\theta}(t+\delta)$ determines the update $\theta(t) \to \theta(t+\delta)$:

$$\dot{\theta}(t+\delta) = \dot{\theta}(t) + \ddot{\theta}(t)\,\delta, \tag{3.72}$$
$$\theta(t+\delta) = \theta(t) + \dot{\theta}(t+\delta)\,\delta. \tag{3.73}$$

Would it not be simpler and make more sense to update θ and $\dot{\theta}$ simultaneously from their current values, so that eqn 3.73 would read $\theta(t+\delta) = \theta(t) + \dot{\theta}(t)\,\delta$? This simplest of all time-stepping schemes is called the *Euler method*, and should not be used for ordinary differential equations (although it is sometimes used for solving partial differential equations).

(a) *Try the Euler method. First, see why reversing the order of the updates to θ and $\dot{\theta}$,*

$$\begin{aligned}\theta(t+\delta) &= \theta(t) + \dot{\theta}(t)\,\delta,\\ \dot{\theta}(t+\delta) &= \dot{\theta}(t) + \ddot{\theta}(t)\,\delta,\end{aligned} \tag{2}$$

in the code you have downloaded would produce a simultaneous update. Swap these two lines in the code, and watch the pendulum swing for several turns, until it starts looping the loop. Is the new algorithm as good as the old one? (Make sure you switch the two lines back afterwards.)

The simultaneous update scheme is just as accurate as the one we chose, but it is not as faithful to the physics of the problem; its fidelity is not as good. For subtle reasons that we will not explain here, updating first $\dot{\theta}$ and then θ allows our algorithm to exactly simulate an approximate Hamiltonian system;[4] it is called a *symplectic* algorithm.[5] Improved versions of this algorithm—like the Verlet algorithms below—are often used to simulate systems that conserve energy (like molecular

[1] From *Statistical Mechanics: Entropy, Order Parameters, and Complexity* by James P. Sethna, copyright Oxford University Press, 2007, page 58. A pdf of the text is available at pages.physics.cornell.edu/sethna/StatMech/ (select the picture of the text). Hyperlinks from this exercise into the text will work if the latter PDF is downloaded into the same directory/folder as this PDF.

[2] This exercise and the associated software were developed in collaboration with Christopher Myers. See also *Numerical Recipes* [106, chapter 16].

[3] We will depict our pendulum emphasizing the rod rather than the mass; the equation for a physical rod without an end mass is similar.

[4] Equation 3.73 is Hamiltonian dynamics for non-interacting particles, and m times eqn 3.72 is the momentum evolution law $\dot{p} = F$ for a system of particles with infinite mass.

[5] It conserves a *symplectic form*. In non-mathematician's language, this means our time step perfectly simulates a Hamiltonian system satisfying Liouville's theorem and energy conservation, but with an approximation to the true energy.

dynamics) because they exactly[6] simulate the dynamics for an approximation to the Hamiltonian—preserving important physical features not kept by just approximately solving the dynamics.

Accuracy. Most computational methods for solving differential equations (and many other continuum problems like integrating functions) involve a step size δ, and become more accurate as δ gets smaller. What is most important is not the error in each time step, but the accuracy of the answer after a fixed time T, which is the accumulated error after T/δ time steps. If this accumulated error varies as δ^n, we say that the algorithm has nth order cumulative accuracy. Our algorithm is not very high order!

(b) *Solve eqns 3.72 and 3.73 to give $\theta(t+\delta)$ in terms of $\theta(t)$, $\dot\theta(t)$ and $\ddot\theta(t)$ for our staggered algorithm. Comparing to the Taylor series $x(t+\tau) = x(t)+v\tau+\frac{1}{2}a\tau^2+O(\tau^3)$ applied to $\theta(t)$, what order in δ is the error for θ in a single time-step? Looking at eqn 3.73, what is the error in one time step for $\dot\theta$? Given that the worst of the two accuracies should determine the overall accuracy, and that the time step error accumulates over more steps as the step size decreases, what order should the cumulative accuracy be for our staggered algorithm?*

(c) *Plot the pendulum trajectory $\theta(t)$ for time steps $\delta = 0.1$, 0.01, and 0.001. Zoom in on the curve at one of the coarse points (say, $t = 1$) and visually compare the values from the three time steps. Does it appear that the trajectory is converging[7] as $\delta \to 0$? What order cumulative accuracy do you find: is each curve better by a factor of 10, 100, $1000\ldots$?*

A rearrangement of our staggered time-step (eqns 3.72 and 3.73) gives the *velocity Verlet* algorithm:

$$\begin{aligned} \dot\theta(t+\delta/2) &= \dot\theta(t) + \tfrac{1}{2}\ddot\theta(t)\,\delta, \\ \theta(t+\delta) &= \theta(t) + \dot\theta(t+\delta/2)\,\delta, \\ \dot\theta(t+\delta) &= \dot\theta(t+\delta/2) + \tfrac{1}{2}\ddot\theta(t+\delta)\delta. \end{aligned} \tag{3}$$

The trick that makes this algorithm so good is to cleverly split the velocity increment into two pieces, half for the acceleration at the old position and half for the new position. (Initialize $\ddot\theta$ once before starting the loop.)

(d) *Show that N steps of our staggered time-step would give the velocity Verlet algorithm, if we shifted the velocities before and afterward by $\mp\frac{1}{2}\delta\ddot\theta$.*

(e) *As in part (b), write $\theta(t+\delta)$ for velocity Verlet in terms of quantities at t. What order cumulative accuracy does this suggest?*

(f) *Implement velocity Verlet, and plot the trajectory for time steps $\delta = 0.1$, 0.01, and 0.001. What is the order of cumulative accuracy?*

Stability. In many cases high accuracy is not crucial. What prevents us from taking enormous time steps? In a given problem, there is usually a typical fastest time scale: a vibration or oscillation period (as in our exercise) or a growth or decay rate. When our time step becomes a substantial fraction of this fastest time scale, algorithms like ours usually become *unstable*; the first few time steps may be fairly accurate, but small errors build up exponentially until the errors become unacceptable (indeed, often one's first warning of problems are machine overflows).

(g) *Plot the pendulum trajectory $\theta(t)$ for time steps $\delta = 0.1, 0.2, \ldots, 0.8$, using a small-amplitude oscillation $\theta_0 = 0.01$, $\dot\theta_0 = 0.0$, up to $t_{\max} = 10$. At about what δ_c does it go unstable? How does δ_c compare with the characteristic time period of the pendulum? At $\delta_c/2$, how accurate is the amplitude of the oscillation?* (You will need to observe several periods in order to estimate the maximum amplitude of the solution.)

In solving the properties of large, nonlinear systems (e.g., partial differential equations (PDEs) and molecular dynamics) stability tends to be the key difficulty. The maximum step-size depends on the local configuration, so highly nonlinear regions can send the system unstable before one might expect. The maximum safe stable step-size often has accuracy far higher than needed.

The Verlet algorithms are not hard to code. There are higher-order symplectic algorithms for Hamiltonian systems, but they are mostly used in unusual applications (planetary motion) where high accuracy is demanded, because they are typically significantly less stable. In systems of differential equations where there is no conserved energy or Hamiltonian, or even in Hamiltonian systems (like high-

[6]Up to rounding errors.
[7]You may note that the approximate answers seem to extrapolate nicely to the correct answer. One can use this to converge more quickly to the correct answer. This is called Richardson extrapolation and is the basis for the Bulirsch–Stoer methods.

energy collisions) where accuracy at short times is more crucial than fidelity at long times, we use general-purpose methods.

ODE (Ordinary Differential Equation) packages. The general-purpose solvers come in a variety of basic algorithms (Runge-Kutta, predictor-corrector, ...), and methods for maintaining and enhancing accuracy (variable step size, Richardson extrapolation). There are also *implicit* methods for *stiff* systems. A system is stiff if there is a large separation between the slowest and fastest relevant time scales; implicit methods often allow one to take time steps much larger than the fastest time scale (unlike the explicit Verlet methods you studied in part (d), which go unstable). Large, sophisticated packages have been developed over many years for solving differential equations—switching between algorithms and varying the time steps to most efficiently maintain a given level of accuracy. They solve $\mathrm{d}\mathbf{y}/\mathrm{d}t = \mathbf{dydt}(\mathbf{y}, t)$, where for us $\mathbf{y} = [\theta, \dot{\theta}]$ and $\mathbf{dydt} = [\dot{\theta}, \ddot{\theta}]$. They typically come in the form of subroutines or functions, which need the following as arguments:

- initial conditions $\mathbf{y}_0$,
- the right-hand side **dydt**, a function of the vector $\mathbf{y}$ and time t, which returns a vector giving the current rate of change of $\mathbf{y}$, and
- the initial and final times, and perhaps intermediate times, at which the trajectory $\mathbf{y}(t)$ is desired.

They often have options that

- ask for desired accuracy goals, typically a relative (fractional) accuracy and an absolute accuracy, sometimes set separately for each component of $\mathbf{y}$,
- ask for and return derivative and time step information from the end of the last step (to allow efficient restarts after intermediate points),
- ask for a routine that computes the derivatives of **dydt** with respect to the current components of $\mathbf{y}$ (for use by the stiff integrator), and
- return information about the methods, time steps, and performance of the algorithm.

You will be supplied with one of these general-purpose packages, and instructions on how to use it.

(h) *Write the function* **dydt**, *and use the general-purpose solver to solve for the motion of the pendulum as in parts (a)–(c), and informally check that the trajectory is accurate.*