4 FINITE-DIFFERENCE METHODS

Consider this problem: Determine the location of a car at 10:30, 11:00, and 12:00 given that the car left Salina, Kansas, at 10:00, traveling west on Interstate-70 at a constant 50 mph. The solution of this problem is governed by an ordinary differential equation

$$v = \frac{dx}{dt} = 50 \text{ mph} \tag{4.1}$$

We are told a rate and an initial condition and, consequently, such problems are called *initial-value problems*. The solution is trivial because the rate is constant; a simple integration of (4.1) yields an algebraic equation that can be solved for the distance from Salina for any time subsequent to 10:00. This problem is analogous to the hard-sphere simulation problem described in the last chapter. In a hard-sphere simulation we need solve only algebraic equations because, between collisions, the spheres follow straight-line trajectories at constant velocities.

But what if either problem were more realistic? In the case of the car, we know, ignoring the use of cruise control, that the car's speed will not be constant: it will vary in response to local traffic, topography of the road, and inattentiveness of the driver. In this more realistic situation not only does the car's speed change with time, but we do not even have an analytic form for *how* the speed changes. Perhaps we have a table of speedometer readings at points during the journey. How do we solve this problem? How do we determine the car's position on I-70 at various points in time? Now we must solve the differential equation and do so numerically—the analytic form for the rate is unknown.

This problem is analogous to a molecular dynamics simulation of molecules whose potential energy varies continuously with distance ("soft bodies"). Because each molecule is simultaneously interacting with many other molecules, soft-body trajectories are not straight lines nor are the velocities constant between collisions. In fact the idea of a collision is not as precise as in the hard-body case. Collisions between soft bodies are not instantaneous; rather, they are strong repulsive interactions that occur over a finite duration. As with the realistic car situation, soft-body simulations require a numerical method for solving differential equations.

The classic tools for attacking initial-value problems are finite-difference methods.[†] These methods replace differentials, such as dx and dt, with finite differences Δx and Δt; they replace differential equations with finite-difference equations; and over a small but finite time Δt, they assume the rate (or some known function of the rate) is constant. Then to solve the car problem, we proceed in the following stepwise fashion: From the known initial position of the car $x(t_0)$, we use the assumed constant rate to approximate the position $x(t_0 + \Delta t)$ after the lapse of the small interval Δt. From this approximate position, with a revised value for the rate (say, from our table of speedometer readings), we step forward another increment to estimate the position at $x(t_0 + 2\,\Delta t)$. After many such steps we have an approximation to the car's path $x(t)$. This strategy we also use in soft-body molecular dynamics.

As indicated in Figure 1.5, molecular dynamics divides into two great tasks: generate the trajectory in phase space and analyze that trajectory for the properties of interest. Each task incurs uncertainties that detract from the reliability of the results. In generating soft-body trajectories, uncertainties arise from the finite-difference method used (truncation error) and from the way it is implemented (round-off error). Therefore, we must be concerned not merely with finite-difference methods per se, but with the broader problem of how those methods can undermine the reliability of simulation results. We begin by introducing a prototypical finite-difference algorithm (Section 4.1) and use that algorithm to define truncation and round-off errors (Section 4.2). The terms *truncation* and *round-off* refer to sources of errors; however, we must consider not only what causes errors but also how errors propagate—whether they grow as the simulation proceeds. This is the issue of algorithmic stability (Section 4.3). For simple finite-difference algorithms applied to linear differential equations, we can readily perform a stability analysis, but for the nonlinear differential equations used in molecular dynamics (Section 4.4) such an analysis is less direct. Moreover, in soft-body simulations we may find that uncertainties in computed trajectories are

[†]W. Thomson, Lecture VI, p. 61: "The problem that I put before you here is given in that work [Lagrange's *Mécanique Analytique*] under the title of vibrations of a linear system of bodies. Lagrange applies what he calls the algorithm of finite differences to the solutions."

compounded by subtle interplay among errors, instabilities, and nonergodicity (Section 4.5).

4.1 A PROTOTYPE: EULER'S METHOD

Several finite-difference methods originate from truncated Taylor expansions and the simplest is Euler's method, which is a Taylor expansion truncated after the first-order term

$$x(t+\Delta t) = x(t) + \dot{x}(t)\,\Delta t \tag{4.2}$$

From the known (or estimated) value of x at t, this method estimates x at $t+\Delta t$ by extrapolating from $x(t)$ the straight line that has slope dx/dt, evaluated at t. As a concrete example, consider Euler's method applied to the ODHO of Figure 2.2. To estimate the oscillator's position x and velocity v, Euler's method uses

$$x(t+\Delta t) = x(t) + \dot{x}(t)\,\Delta t \tag{4.3}$$

$$v(t+\Delta t) = v(t) + \dot{v}(t)\,\Delta t \tag{4.4}$$

Now, according to (2.26), the ODHO has

$$\dot{x}(t) = v(t) \tag{4.5}$$

$$\dot{v}(t) = \ddot{x}(t) = -\frac{\gamma}{m}x(t) = -\omega^2 x(t) \tag{4.6}$$

where ω is the frequency of the oscillation. Therefore, by using these expressions in (4.3) and (4.4), Euler's method for the ODHO becomes

$$x(t+\Delta t) = x(t) + v(t)\,\Delta t \tag{4.7}$$

$$v(t+\Delta t) = v(t) - \omega^2 x(t)\,\Delta t \tag{4.8}$$

For specified values of the mass m, spring constant γ, and initial conditions $x(t_0)$ and $v(t_0)$, iterative application of (4.7) and (4.8) generates an approximation to the trajectory $\{x(t), v(t)\}$. This calculated trajectory will have errors associated with it; moreover, those errors may accumulate as the iterative calculation proceeds. These issues are discussed in the next two sections.

4.2 ERRORS

A finite-difference method incurs two types of errors: truncation error and round-off error. *Truncation error* refers to the accuracy with which a finite difference method approximates the true solution to a differential equation. When a finite-difference equation is written in a Taylor series form, truncation error is measured by the first nonzero term that has been omitted from the series. The Taylor series is

$$x(t+\Delta t) = x(t) + \frac{dx(t)}{dt}\,\Delta t + \frac{1}{2}\,\frac{d^2x(t)}{dt^2}\,\Delta t^2 + \frac{1}{3!}\,\frac{d^3x(t)}{dt^3}\,\Delta t^3 + \cdots \tag{4.9}$$

and comparing this with (4.2) identifies the truncation error (te) in Euler's method as

$$\text{te} = \frac{1}{2}\,\frac{d^2x(t)}{dt^2}\,\Delta t^2 \tag{4.10}$$

A method whose truncation error varies as $(\Delta t)^{n+1}$ is said to be an nth-order method; hence, Euler's is first order. We typically use a system of units such that $\Delta t < 1$; therefore, for a given time-step size, high-order methods have smaller truncation errors than low-order methods. Truncation error is inherent in the algorithm; its value is the same regardless of how the actual calculations are performed, whether on a computer, on a calculator, or by pencil and paper.

In contrast, *round-off error* encompasses all errors that result from the implementation of the finite-difference algorithm. For example, round-off error is affected by the number of significant figures kept at each stage of the calculation, by the order in which the calculations are actually performed, and by any approximations used in evaluating square roots, exponentials, and so on.

Both round-off and truncation errors can be subdivided into global and local errors. *Local error* is that incurred during one step (Δt) of the algorithm, while *global error* is local error accumulated over the entire calculation. It is local truncation error, such as (4.10), that is used to define the order of a method. But instead of local errors, we should be more concerned with global errors, and in general, global error varies by one factor of Δt *less* than local error. To see this, consider an nth-order algorithm that has local truncation error

$$\text{lte} = k x^{(n+1)}\,\Delta t^{n+1} \tag{4.11}$$

where k is a constant and $x^{(n+1)}$ means the $(n+1)$th derivative of x. Over M integration steps, each of size Δt, the global truncation error is

$$\text{gte} = k \sum_{i=1}^{M} x_i^{(n+1)} \Delta t^{n+1} = k \, \Delta t^{n+1} \sum_{i=1}^{M} x_i^{(n+1)} \tag{4.12}$$

In general the derivatives under the sum will vary in value from one step to the next, but we can apply a mean-value theorem: there will be an average value of the M-derivatives such that the M-term sum can be replaced by M times the average,

$$\text{gte} = k \, \Delta t^{n+1} M \overline{x^{(n+1)}} \tag{4.13}$$

Here the overbar indicates the average. Now the number of steps M is related to the total duration t of the calculation by $M = t/\Delta t$; hence,

$$\text{gte} = k \, \Delta t^{n+1} \frac{t}{\Delta t} \overline{x^{(n+1)}} = k \, \Delta t^{n} \, t \overline{x^{(n+1)}} \tag{4.14}$$

thus, gte varies as $(\Delta t)^n$, although lte varies as $(\Delta t)^{(n+1)}$. QED. If, instead of gte, we evaluate the average gte per step, $\langle \text{gte} \rangle$, then we have a global error that behaves with Δt in the same way as the local truncation error,

$$\langle \text{gte} \rangle = \frac{\text{gte}}{M} = k \, \Delta t^{n+1} \overline{x^{(n+1)}} \tag{4.15}$$

The results (4.14) and (4.15) apply to one-step methods, which are those that use information only from the current step to estimate x at the next step. In contrast, multistep methods use estimates for x at previous steps, as well as the current step, to estimate x at the next step; in those cases, gte is a more complicated function of the step size.

Global truncation error and global round-off error (gre) both depend on the size of the integration step Δt, so having chosen an algorithm, we must determine the magnitude of Δt that produces acceptably small global errors. Unfortunately, gte and gre are affected differently by changes in the step size. Global truncation error decreases with decreasing Δt, as indicated above. In contrast, gre depends on the number of calculations: increasing the number of calculations increases the opportunities for round-off and produces higher gre. Thus at some point, decreasing Δt for the same duration $\tau = t_{\text{final}} - t_{\text{initial}}$ does not produce more accurate results, as shown in Figure 4.1. Usually the value of Δt having the smallest total error is too small to be useful—too much computer time would be required for a simulation. Some value of Δt larger than that for minimum total error is used; its value is determined empirically in test calculations.

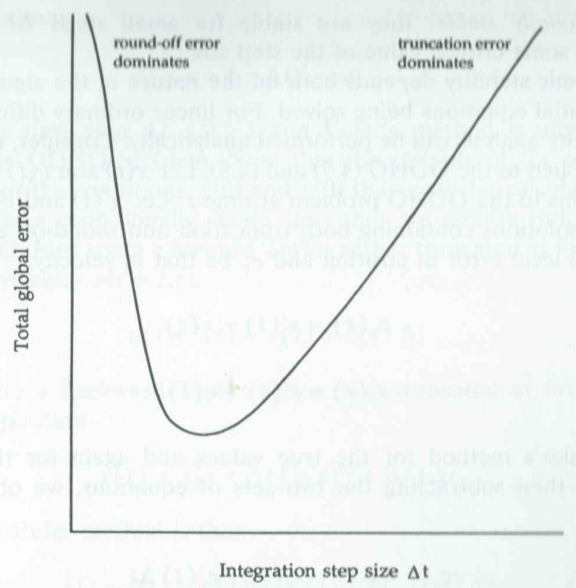

FIGURE 4.1 Schematic of how truncation and round-off errors contribute to the total global error generated by a finite-difference algorithm. At large step sizes the global error is dominated by truncation errors in the algorithm; however, for small step sizes global error is dominated by round-off. For each step size, the behavior shown applies for global error accumulated over a fixed duration of calculation, not for a fixed number of integration steps.

The principal defenses against round-off errors are to write efficient, nonredundant code and to use high-precision arithmetic: double precision rather than single. The principal defense against truncation error is to reduce the step size. However, if Δt must be made unacceptably small, the only recourse is to change finite-difference algorithms.

4.3 ALGORITHMIC STABILITY

In addition to the magnitudes of errors involved in using finite-difference methods, we must also be concerned with how the algorithm propagates those errors. This is the issue of algorithmic stability—a concept that is distinct from the idea of trajectory stability discussed in Sections 2.4 and 2.5. If an algorithm amplifies errors from one step to the next, then the algorithm is *unstable*, and ultimately the calculation will abort in a numerical overflow. Conversely, if the algorithm does not amplify errors from one step to the next, then the method is *stable*. Most algorithms used in molecular dynamics are *conditionally stable*: they are stable for small steps Δt but become unstable at some critical value of the step size.

Algorithmic stability depends both on the nature of the algorithm and on the differential equations being solved. For linear ordinary differential equations, stability analysis can be performed analytically. Consider, again, Euler's method applied to the ODHO (4.7) and (4.8). Let $x(t)$ and $v(t)$ represent the true solutions to the ODHO problem at time t. Let $x'(t)$ and $v'(t)$ represent erroneous solutions containing both truncation and round-off errors. Let e_x be the total local error in position and e_v be that in velocity,

$$e_x(t) = x'(t) - x(t) \tag{4.16}$$

$$e_v(t) = v'(t) - v(t) \tag{4.17}$$

Writing Euler's method for the true values and again for the erroneous values and then subtracting the two sets of equations, we obtain, for the ODHO,

$$e_x(t+\Delta t) = e_x(t) + e_v(t)\,\Delta t \tag{4.18}$$

$$e_v(t+\Delta t) = e_v(t) - \omega^2 e_x(t)\,\Delta t \tag{4.19}$$

which relate the errors at time $t+\Delta t$ to those at time t. These two equations can be written in vector form as

$$\mathbf{e}(t+\Delta t) = \mathbf{A}\mathbf{e}(t) \tag{4.20}$$

where $\mathbf{e} = (e_x e_v)^T$ is the column vector of errors and $\mathbf{A}$ is the stability matrix

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{bmatrix} \tag{4.21}$$

Now, if $\mathbf{A}$ has any eigenvalue λ that lies outside the unit circle ($\lambda^2 > 1$) on the complex plane, then $\mathbf{A}$ amplifies errors and the algorithm is unstable. The eigenvalues satisfy the characteristic equation

$$|\mathbf{A} - \lambda \mathbf{I}| = 0 \tag{4.22}$$

where $\mathbf{I}$ is the identity matrix. For $\mathbf{A}$ given by (4.21) this equation is

$$(1-\lambda)^2 + \omega^2(\Delta t)^2 = 0 \tag{4.23}$$

which has solutions

$$\lambda = 1 \pm \Delta t\sqrt{-\omega^2} \tag{4.24}$$

Thus, for any time step Δt, $|\lambda| > 1$ and Euler's method is unstable when applied to the ODHO problem. Note that the stability of the algorithm is independent of the conditions $x(0)$ and $v(0)$ that start the calculations.

To illustrate a conditionally stable algorithm, we modify Euler's method for the ODHO. First write a forward Taylor series, truncated at first order, to estimate the velocity $v(t+\Delta t)$,

$$v(t+\Delta t) = v(t) + \dot{v}(t)\,\Delta t \tag{4.25}$$

and then write a backward Taylor series, also truncated at first order, to estimate the position

$$x(t) = x(t+\Delta t) - \dot{x}(t+\Delta t)\,\Delta t \tag{4.26}$$

The modified Euler method is then

$$x(t+\Delta t) = x(t) + [v(t) + \dot{v}(t)\,\Delta t]\,\Delta t \tag{4.27}$$

$$v(t+\Delta t) = v(t) + \dot{v}(t)\,\Delta t \tag{4.28}$$

For the ODHO the time derivatives are given by (4.5) and (4.6), so the algorithm becomes

$$x(t+\Delta t) = \left[1 - \omega^2(\Delta t)^2\right] x(t) + v(t)\,\Delta t \tag{4.29}$$

$$v(t+\Delta t) = v(t) - \omega^2 x(t)\,\Delta t \tag{4.30}$$

The stability matrix for this method is

$$\mathbf{A} = \begin{bmatrix} \left(1-\omega^2(\Delta t)^2\right) & \Delta t \\ -\omega^2 \Delta t & 1 \end{bmatrix} \tag{4.31}$$

Solving the characteristic equation shows that the step size Δt must satisfy

$$\left|2 - \omega^2(\Delta t)^2 \pm \omega\,\Delta t\sqrt{\omega^2(\Delta t)^2 - 4}\right| < 2 \tag{4.32}$$

That is, we must have $-2 < \omega\,\Delta t < 2$ for this modified Euler method to provide stable solutions to the ODHO problem.

It is important to realize that stable solutions are not necessarily accurate solutions. This point is illustrated in Figure 4.2, which shows ODHO phase-

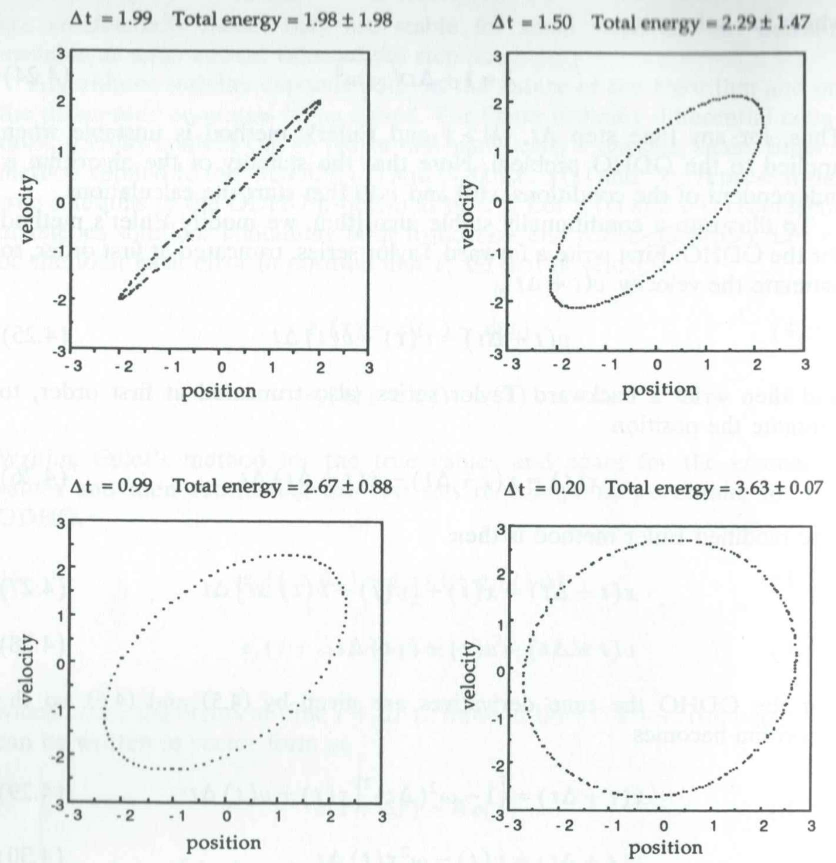

FIGURE 4.2 Stable solutions are not necessarily accurate. Shown here are phase-plane trajectories for the ODHO calculated from the modified Euler algorithm (4.29) and (4.30) using four values of the time-step Δt. Each trajectory shows 200 steps starting from $x(0) = v(0) = 2$, with $m = \gamma = 1$ (arbitrary units). Thus in each case the true solution to the ODHO is a circle with total energy equal to 4. Each of the four values of Δt is below the stability limit $\Delta t = 2$, but the accuracy becomes acceptable only when $\Delta t < 0.02$.

plane trajectories calculated from the modified Euler method (4.29) and (4.30). The figure shows how the shape of the trajectory is affected by the size of the integration time step Δt. Calculations attempted with Δt only marginally greater than the stability limit ($\Delta t = 2$) quickly led to numerical overflows. The values of the mass, spring constant, and initial conditions were set such that the true solution to the ODHO describes a circle in the phase

plane, corresponding to a total energy $E = 4$ arbitrary units. The calculations show that although the algorithm is stable for $\Delta t < 2$, the total energy is not properly conserved at its initial value of $E = 4$ until the time step is reduced to $\Delta t < 0.02$, which is a value 100 times smaller than the stability limit.

The lesson from these simple examples is that a finite-difference algorithm applied to a particular set of differential equations yields a stability matrix that may produce inherently stable, unstable, or conditionally stable behavior. Our first response to an unstable algorithm is to decrease the step size; however, if stability can be achieved only by using an intolerably small Δt, then we must either change algorithms or find another problem (i.e., change the differential equations).

Molecular dynamics involves nonlinear ordinary differential equations, so the analytic stability analysis described above cannot be used. Instead, we may do an approximate analysis by linearizing the differential equations or we may attack the stability problem using the methods of Lyapunov [1]. Both strategies are cumbersome when applied to the equations of motion used in molecular dynamics, and what we do instead is a numerical analysis using trial values for Δt. That is, through a series of short test runs, we identify the critical step Δt at which the algorithm becomes unstable and then choose a smaller operational value that establishes conservation of energy.

4.4 ALGORITHMS FOR MOLECULAR DYNAMICS

Of the large number of finite-difference methods that can be devised, the most commonly used are the Runge–Kutta (RK) methods [2]. These methods have the structure of Euler's method (in fact, Euler's is the first-order RK algorithm)

$$x(t+\Delta t) = x(t) + \dot{x}\,\Delta t \tag{4.33}$$

The various RK methods differ from one another in how the slope $\dot{x}$ is estimated. Each RK algorithm estimates the slope at points along the interval Δt and then computes a weighted average to get a single value to be used in (4.33). For low-order RK algorithms, the number of slope evaluations during one step equals the order of the method. Thus the popular fourth-order RK algorithm (see Exercise 4.11) makes four estimates of the slope for each step forward in the solution procedure.

Runge–Kutta methods generally have good stability characteristics; nevertheless, they have been little used in molecular dynamics because, for large numbers of molecules, the RK algorithms are too slow. In molecular dynamics the evaluation of intermolecular forces is by far the most time-consuming calculation. Therefore, since a fourth-order RK algorithm would require four force evaluations per atom per step, such an algorithm would execute almost four times slower than a method that needs only one force evaluation per atom per step. Runge–Kutta methods have been used in simulations of systems involving a very few degrees of freedom [3].

The RK methods suggest that the order of a method can often be increased by using positions and velocities from several points in time rather than from just the current time. We would like to use such information, but we want to avoid the expense of evaluating intermolecular forces more than once per atom per step. These conflicting goals can be met in two general ways: (a) use positions and velocities from previously calculated steps, the principal example of this approach being Verlet's algorithm, or (b) estimate positions and velocities for future steps. The latter are the predictor–corrector algorithms.

4.4.1 Verlet's Algorithm

The simplest finite-difference method that has been widely used in molecular dynamics is a third-order Störmer algorithm, first used by Verlet [4] and known to simulators as Verlet's method. The algorithm is a combination of two Taylor expansions, combined as follows. First write the Taylor series for position from time t forward to $t + \Delta t$:

$$x(t+\Delta t) = x(t) + \frac{dx(t)}{dt}\Delta t + \frac{1}{2}\frac{d^2x(t)}{dt^2}\Delta t^2 + \frac{1}{3!}\frac{d^3x(t)}{dt^3}\Delta t^3 + O(\Delta t^4) \tag{4.34}$$

Then write the Taylor series from t backward to $t - \Delta t$:

$$x(t-\Delta t) = x(t) - \frac{dx(t)}{dt}\Delta t + \frac{1}{2}\frac{d^2x(t)}{dt^2}\Delta t^2 - \frac{1}{3!}\frac{d^3x(t)}{dt^3}\Delta t^3 + O(\Delta t^4) \tag{4.35}$$

Adding these two expansions eliminates all odd-order terms, leaving

$$x(t+\Delta t) = 2x(t) - x(t-\Delta t) + \frac{d^2x(t)}{dt^2}\Delta t^2 + O(\Delta t^4) \tag{4.36}$$

This is Verlet's algorithm for positions. It has a local truncation error that varies as $(\Delta t)^4$ and hence is third order, even though it contains no third-order derivatives. Nor does (4.36) for positions involve any function of the velocities; the acceleration in (4.36) is, of course, obtained from the intermolecular forces and Newton's second law. To estimate velocities, practitioners have contrived various schemes, one being an estimate for the velocity at the half-step:

$$v(t+\tfrac{1}{2}\Delta t) \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{4.37}$$

Verlet himself used the first-order central difference estimator

$$v(t) \approx \frac{x(t+\Delta t) - x(t-\Delta t)}{2\,\Delta t} \tag{4.38}$$

Verlet's algorithm is a two-step method because it estimates $x(t+\Delta t)$ from the current position $x(t)$ and the previous position $x(t-\Delta t)$. Therefore it is not self-starting: initial positions $x(0)$ and velocities $v(0)$ are not sufficient to begin a calculation, and something special must be done at $t = 0$ (say, a backward Euler method) to get $x(-\Delta t)$.

The Verlet algorithm offers the virtues of simplicity and good stability for moderately large time steps. In its original form it treated molecular velocities as less important than positions—a view in conflict with the attitude that the phase-space trajectory depends equally on positions and velocities. Modern formulations [5–7] of the method often overcome this asymmetric view.

4.4.2 General Predictor–Corrector Algorithms

Predictor–corrector methods are composed of three steps: prediction, evaluation, and correction. In particular, from the current position $x(t)$ and velocity $v(t)$ the steps are as follows:

1. Predict the position $x(t+\Delta t)$ and velocity $v(t+\Delta t)$ at the end of the next step.
2. Evaluate the forces at $t+\Delta t$ using the predicted position.
3. Correct the predictions using some combination of the predicted and previous values of position and velocity.

As a simple example, consider the following predictor–corrector based on Euler's method and applied to the ODHO:

1. *Predict* $x(t+\Delta t)$ and $v(t+\Delta t)$ using Euler's method (4.7) and (4.8):

$$x(t+\Delta t) = x(t) + v(t)\,\Delta t$$

$$v(t+\Delta t) = v(t) - \omega^2 x(t)\,\Delta t$$

2. *Evaluate* the force at $t + \Delta t$:

$$\frac{f(t+\Delta t)}{m} = \frac{dv}{dt} = -\omega^2 x(t+\Delta t) \tag{4.39}$$

3. *Correct* the predictions. Here we choose the same form as Euler's method but compute the slopes at the end of the step rather than at the beginning:

$$x(t+\Delta t) = x(t) + v(t+\Delta t)\,\Delta t \tag{4.40}$$

$$v(t+\Delta t) = v(t) - \omega^2 x(t+\Delta t)\,\Delta t \tag{4.41}$$

The force evaluation implicit in (4.8) would actually be done in the previous step $(t - \Delta t)$ and stored for use in the current step, so this algorithm meets the goals of requiring only one force evaluation per step while providing an algorithm of higher order than Euler's method.

Predictor–corrector methods offer great flexibility in that many choices are possible for both the prediction and correction steps. They may be either one-step, in which case they are self-starting, or multistep methods, in which case something special must be done to start the calculation. With judicious combinations of predictor and corrector they offer good stability because the corrector step amounts to a feedback mechanism that can dampen instabilities that might be introduced by the predictor.

Any given predictor–corrector algorithm can be made more elaborate and of higher order by repeating the evaluation and correction steps. Let P be prediction, E be evaluation, and C be correction. Then the procedure described above can be represented as PEC. If the corrected positions and velocities are used as second predictions, we obtain the algorithm $PEC(EC) = P(EC)^2$. Obviously, the E and C steps can be repeated as many times as desired, $P(EC)^n$. This strategy, while simple to implement, is rarely done in molecular dynamics because each E-step requires force calculations; a $P(EC)^n$ simulation will execute nearly n times slower than the PEC simulation.

4.4.3 Gear's Predictor–Corrector Algorithms

Predictor–corrector algorithms were first introduced into molecular dynamics by Rahman [8]. Those commonly used in molecular dynamics are often taken from the collection of methods devised by Gear [9]. The one used in the programs in Appendices I and L consists of the following steps.

Predict molecular positions $\mathbf{r}_i$ at time $t + \Delta t$ using a fifth-order Taylor series based on positions and their derivatives at time t. Thus, the derivatives $\dot{\mathbf{r}}_i$, $\ddot{\mathbf{r}}_i$, $\mathbf{r}_i^{(\mathrm{iii})}$, $\mathbf{r}_i^{(\mathrm{iv})}$, and $\mathbf{r}_i^{(\mathrm{v})}$ are needed at each step; these are also predicted at

time $t + \Delta t$ by applying Taylor expansions at t:

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \dot{\mathbf{r}}_i(t)\,\Delta t + \ddot{\mathbf{r}}_i(t)\frac{(\Delta t)^2}{2!} + \mathbf{r}_i^{(\mathrm{iii})}(t)\frac{(\Delta t)^3}{3!} + \mathbf{r}_i^{(\mathrm{iv})}(t)\frac{(\Delta t)^4}{4!} + \mathbf{r}_i^{(\mathrm{v})}(t)\frac{(\Delta t)^5}{5!} \tag{4.42}$$

$$\dot{\mathbf{r}}_i(t+\Delta t) = \dot{\mathbf{r}}_i(t) + \ddot{\mathbf{r}}_i(t)\,\Delta t + \mathbf{r}_i^{(\mathrm{iii})}(t)\frac{(\Delta t)^2}{2!} + \mathbf{r}_i^{(\mathrm{iv})}(t)\frac{(\Delta t)^3}{3!} + \mathbf{r}_i^{(\mathrm{v})}(t)\frac{(\Delta t)^4}{4!} \tag{4.43}$$

$$\ddot{\mathbf{r}}_i(t+\Delta t) = \ddot{\mathbf{r}}_i(t) + \mathbf{r}_i^{(\mathrm{iii})}(t)\,\Delta t + \mathbf{r}_i^{(\mathrm{iv})}(t)\frac{(\Delta t)^2}{2!} + \mathbf{r}_i^{(\mathrm{v})}(t)\frac{(\Delta t)^3}{3!} \tag{4.44}$$

$$\mathbf{r}_i^{(\mathrm{iii})}(t+\Delta t) = \mathbf{r}_i^{(\mathrm{iii})}(t) + \mathbf{r}_i^{(\mathrm{iv})}(t)\,\Delta t + \mathbf{r}_i^{(\mathrm{v})}(t)\frac{(\Delta t)^2}{2!} \tag{4.45}$$

$$\mathbf{r}_i^{(\mathrm{iv})}(t+\Delta t) = \mathbf{r}_i^{(\mathrm{iv})}(t) + \mathbf{r}_i^{(\mathrm{v})}(t)\,\Delta t \tag{4.46}$$

$$\mathbf{r}_i^{(\mathrm{v})}(t+\Delta t) = \mathbf{r}_i^{(\mathrm{v})}(t) \tag{4.47}$$

Evaluate the intermolecular force $\mathbf{F}_i$ on each molecule at time $t + \Delta t$ using the predicted positions. For continuous potential energy functions $u(r_{ij})$ that act between atoms i and j, the force on each molecule is given by

$$\mathbf{F}_i = -\sum_{j \neq i} \frac{\partial u(r_{ij})}{\partial r_{ij}}\,\hat{\mathbf{r}}_{ij} \tag{4.48}$$

where $\hat{\mathbf{r}}_{ij}$ is the unit vector in the $\mathbf{r}_{ij}$-direction. Evaluation of forces is time consuming because the sum in (4.48) must be performed for *each* molecule i in the system. However, there are several ways to save time; one is that Newton's third law (2.6) can be applied,

$$\mathbf{F}(\mathbf{r}_{ij}) = -\mathbf{F}(\mathbf{r}_{ji}) \tag{4.49}$$

to decrease the amount of computation by a factor of 2. Other time-saving devices are discussed in Section 5.1.

Correct the predicted positions and their derivatives using the discrepancy $\Delta\ddot{\mathbf{r}}_i$ between the predicted acceleration and that given by the evaluated force $\mathbf{F}_i$. With the forces at $t + \Delta t$ obtained from (4.48), Newton's second law (2.1) can be used to determine the accelerations $\ddot{\mathbf{r}}_i(t+\Delta t)$. The difference between the predicted accelerations and evaluated accelerations is then formed,

$$\Delta\ddot{\mathbf{r}}_i = \left[\ddot{\mathbf{r}}_i(t+\Delta t) - \ddot{\mathbf{r}}_i^P(t+\Delta t)\right] \tag{4.50}$$

In Gear's algorithms for second-order differential equations, this difference term is used to correct all predicted positions and their derivatives; thus,

$$\mathbf{r}_i = \mathbf{r}_i^P + \alpha_0\,\Delta\mathbf{R2} \tag{4.51}$$

$$\dot{\mathbf{r}}_i\,\Delta t = \dot{\mathbf{r}}_i^P\,\Delta t + \alpha_1\,\Delta\mathbf{R2} \tag{4.52}$$

$$\frac{\ddot{\mathbf{r}}_i(\Delta t)^2}{2!} = \frac{\ddot{\mathbf{r}}_i^P(\Delta t)^2}{2!} + \alpha_2\,\Delta\mathbf{R2} \tag{4.53}$$

$$\frac{\mathbf{r}_i^{(\text{iii})}(\Delta t)^3}{3!} = \frac{\mathbf{r}_i^{(\text{iii})P}(\Delta t)^3}{3!} + \alpha_3\,\Delta\mathbf{R2} \tag{4.54}$$

$$\frac{\mathbf{r}_i^{(\text{iv})}(\Delta t)^4}{4!} = \frac{\mathbf{r}_i^{(\text{iv})P}(\Delta t)^4}{4!} + \alpha_4\,\Delta\mathbf{R2} \tag{4.55}$$

$$\frac{\mathbf{r}_i^{(\text{v})}(\Delta t)^5}{5!} = \frac{\mathbf{r}_i^{(\text{v})P}(\Delta t)^5}{5!} + \alpha_5\,\Delta\mathbf{R2} \tag{4.56}$$

where

$$\Delta\mathbf{R2} \equiv \frac{\Delta\ddot{\mathbf{r}}_i(\Delta t)^2}{2!} \tag{4.57}$$

TABLE 4.1 Values of α_i Parameters in Gear's Predictor–Corrector Algorithm[a] for Second-Order Differential Equations Using Predictors of Order q

α_i	$q=3$	$q=4$	$q=5$
α_0	$\frac{1}{6}$	$\frac{19}{120}$	$\frac{3}{16}$
α_1	$\frac{5}{6}$	$\frac{3}{4}$	$\frac{251}{360}$
α_2	1	1	1
α_3	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{11}{18}$
α_4	—	$\frac{1}{12}$	$\frac{1}{6}$
α_5	—	—	$\frac{1}{60}$

[a]From ref. 9, except that for $q=5$, $\alpha_0 = 3/16$ seems to be somewhat better than Gear's original value.

The parameters α_i promote numerical stability of the algorithm. The α_i depend on the order of the differential equations to be solved and on the order of the Taylor series predictor. Gear determined their values by applying each algorithm to linear differential equations and analyzing the resulting stability matrices. For a q-order predictor, the values of the α_i were chosen to make the local truncation error of $O(\Delta t^{q+1})$; then the method will be stable for second-order differential equations with global truncation error of $O(\Delta t^{q+1-2}) = O(\Delta t^{q-1})$. Values of the α_i for third-, fourth-, and fifth-order predictors are given in Table 4.1.

4.4.4 Energy Conservation

Finite-difference algorithms used in molecular dynamics are often judged by their ability to conserve energy, and in this section we illustrate such tests. To compare the Verlet and Gear algorithms, we computed the dynamics of a single collision between two otherwise isolated Lennard-Jones atoms with the motion restricted to one dimension. Initially the atoms were separated by a distance of 2.6σ and each atom was given an initial approach velocity $v^* = v(m/\varepsilon)^{1/2} = 1$. The equations of motion were then integrated through the collision until the atoms were again separated by 2.6σ. To accumulate a global error, we compared the total energy at each step with its value at the first step,

$$ge = \sqrt{\sum_{k=1}^{M}\left[E^*(0) - E^*(k\,\Delta t)\right]^2} \tag{4.58}$$

where $E^* = E/\varepsilon$. This sum was accumulated as a function of step size Δt, and since in each case the initial and final positions of the atoms were the same, the total duration $(M\,\Delta t)$ of each run was the same.

We show in Figure 4.3 how global error in the Gear algorithm changes with step size in both single- and double-precision arithmetic. Here we use the time step in dimensionless form: $\Delta t^* = \Delta t/\sigma(m/\varepsilon)^{1/2}$, where ε and σ are potential parameters and m is the atomic mass. For values of ε, σ, and m characteristic of argon, $\Delta t^* = 0.005$ corresponds to about 0.011 psec. In Figure 4.3, for time steps larger than $\Delta t^* = 0.005$, the single- and double-precision results coincide. However, for time steps smaller than 0.001, single-precision round-off error dominates truncation error; so as the time step is decreased below 0.001 in the single-precision calculation, the global error in energy increases. This computed behavior confirms the general trend shown schematically in Figure 4.1. In contrast, the double-precision error continues to decrease as Δt^* is decreased below 0.001.