PY 502, Computational Physics, Fall 2024

Numerical Solutions of Classical Equations of Motion

Anders W. Sandvik, Department of Physics, Boston University

1 Introduction

Classical equations of motion, i.e., Newton's laws, govern the dynamics of systems ranging from very large, such as solar systems and galaxies, to very small, such as molecules in liquids and gases (in cases where quantum mechanical fluctuations can be neglected, which is often the case). In between these extremes, Newton's equations of motion apply, litterally, to "everything that moves".

Exact analytical solutions of the equations of motion exist only for simple systems, of the types that are discussed in elementary classical mechanics courses, and therefore numerical integration methods are very important in practice. Here we will discuss some commonly used differential equation solvers and use them to study the dynamics of mechanical systems, including ones that exhibit chaotic dynamics. We will limit the discussion to a single moving body, although the methods can be easily generalized to many-body systems as well—dynamics of many-body systems will be discussed later in connection with molecular dynamics simulations.

While some of the numerical schemes that we will discuss here are particularly suitable for integrating classical equations of motions, we will also described methods, such as the classic Runge-Kutta algorithm, that are more generally applicable to a large class of ordinary differential equations. The discussion of systems with chaotic dynamics, although here introduced in the context of classical mechanics, is also of relevance more broadly in studies of nonlinear dynamics.

2 Basic algorithms for equations of motion

Consider a single object (here regarded as a point particle) with mass m moving in one dimension. With its time-dependent position denoted $x(t)$, the differential equation governing its dynamics is

$$\ddot{x}(t) = \frac{1}{m} F[x(t), \dot{x}(t), t], \tag{1}$$

where F is the total force acting on the particle, and $\dot{x}$ and $\ddot{x}$ are the first and second time derivatives of x. We have indicated that the force may depend on x, $\dot{x}$ and t. These dependencies typically come from from a position dependent static potential, a velocity dependent damping (friction), and a time-dependent driving force, but there are other natural posibilities as well, e.g., a position dependent friction.

To study the system numerically, it is convenient to rewrite the second-order differential equation (1) as two coupled first-order equations. Giving the velocity its standard symbol $v(t)$, we have

$$\begin{aligned} \dot{x}(t) &= v(t) \\ \dot{v}(t) &= \frac{1}{m} F[x(t), v(t), t]. \end{aligned} \tag{2}$$

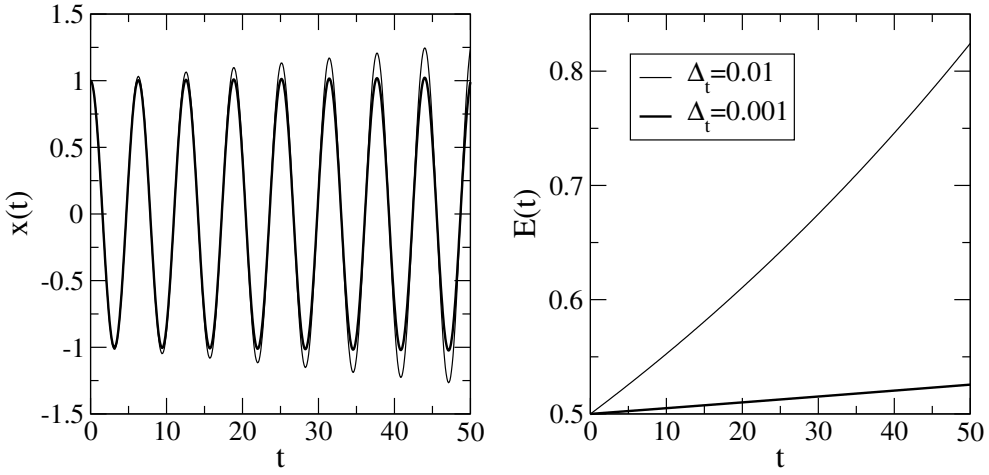

Figure 1: Time dependence of the position x and the energy $E = (1/2)kx^2 + (1/2)mv^2$ of a harmonic oscillator with $k = m = 1$ (which gives an oscillation period 2π) integrated using the Euler method with two different time steps; $\Delta_t = 10^{-2}$ and 10^{-3}.

To integrate this set of equations, we discretize the time-axis as $t = t_0, t_1, \ldots$, with a constant time step $t_{n+1} - t_n = \Delta_t$. The initial values $x_0 = x(t_0)$ and $v_0 = v(t_0)$ are used to start the integration process.

2.1 Euler algorithm

The simplest way to advance the time from t_n to t_{n+1} is to use the first-order approximation;

$$\begin{aligned} x_{n+1} &= x_n + \Delta_t v_n, \\ v_{n+1} &= v_n + \Delta_t a_n, \end{aligned} \tag{3}$$

where $a_n = F(x_n, v_n, t_n)/m$ is the acceleration. Clearly, the error made in each step of this algorithm is $O(\Delta_t^2)$. The method, which is called Euler's forward method, is in general not very useful in practice. For example, in systems with no damping or driving force, the energy should be conserved. However, with the Euler method the energy typically diverges with time, whereas in most higher-order methods the energy errors are bounded. Fig. 1 shows results obtained with the Euler method for the energy and the position of a harmonic oscillator with $k = m = 1$ ($F = -kx$). Even for a time step as as small as 10^{-3}, the energy error is as large as $\approx 5\%$ at $t = 50$ (corresponding to less than 9 oscillations); it grows faster than linearly with t.

There are several ways to proceed to derive more accurate, higher-order integration schemes; we will here discuss manipulations leading to a few practically useful formulas.

2.2 Leapfrog and Verlet algorithms

To simplify the discussion initially, we will here first assume that there is no damping, i.e., the force and the acceleration are velocity independent. We begin by writing the Taylor expansion of

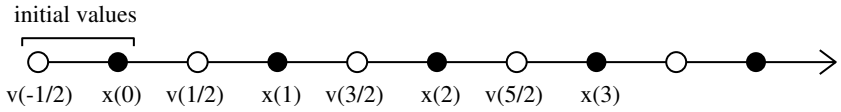

Figure 2: Position and velocity grids used in the leapfrog method. The position is calculated at integer multiples of the time step, $t_n = n\Delta_t$, $n = 0, 1, \ldots$, and the velocity is evaluated at the times $t_{n-1/2} = (n - 1/2)\Delta_t$ between these points. The integration starts with given $n = 0$ values.

$x_{n+1} = x(t_n + \Delta_t)$ to second order in Δ_t;

$$x(t_{n+1}) = x(t_n) + \Delta_t v(t_n) + \frac{1}{2}\Delta_t^2 a(x_n, t_n) + O(\Delta_t^3). \tag{4}$$

Noting that $v(t_n) + (\Delta_t/2)a(x_n, t_n) = v(t_{n+1/2})$ with an error of order Δ_t^2, we can rewrite this as

$$x(t_{n+1}) = x(t_n) + \Delta_t v(t_n + \Delta_t/2) + O(\Delta_t^3). \tag{5}$$

We thus need a formula that propagates the velocity on a time grid with points $t_{n+1/2} = t_n + \Delta_t/2$, i.e., between the integer-labeled time points $t_n = t_0 + n\Delta_t$ used for the position. If we use the first-order expansion of the velocity, $v(t_{n+1/2}) = v(t_{n-1/2}) + \Delta_t a(t_{n-1/2}) + O(\Delta_t^2)$, the error remains $O(\Delta_t^3)$ in Eq. (5), but we have a problem since this requires the acceleration, and hence the position x (on which the force depends), on the grid points $t_{n+1/2}$ used only for the velocity. However, it appears intuitively clear, by symmetry that we should actually use the acceleration at t_{n+1} to propagate the velocity from $t_{n+1/2}$ to $t_{n+3/2}$ (and we will further confirm below that this is the correct way), , i.e.,

$$v(t_{n+3/2}) = v(t_{n+1/2}) + \Delta_t a(x_{n+1}, t_{n+1}). \tag{6}$$

The scheme combining Eqs. (5) and (6) is often called the *leapfrog method*;

$$\begin{aligned} v_{n+1/2} &= v_{n-1/2} + \Delta_t a_n, \\ x_{n+1} &= x_n + \Delta_t v_{n+1/2}. \end{aligned} \tag{7}$$

The leapfrog grid is illustrated in Fig. 2.

We normally have initial conditions in the form of x_0 and v_0. In order to start the leapfrog method we need $v_{-1/2}$, which we can get, up to an error $\sim \Delta_t^2$, using $v_{-1/2} = v_0 - \frac{\Delta_t}{2}a_0 + O(\Delta_t^2)$.

It should be noted here that when we discuss the step error, we normally have in mind the error in $x(t)$. The error in $v(t)$ is normally larger, exemplified by the $O(\Delta_t^2)$ velocity error in the above derivation.

It turns out that the position x in the leapfrog method in fact has a single-step error of order Δ_t^4, i.e., smaller than what might have been expected from the initial expansion (5). This can be seen most easily by deriving the method in another way, starting from two different Taylor expansions:

$$\begin{aligned} x_{n+1} &= x_n + \Delta_t v_n + \frac{1}{2}\Delta_t^2 a_n + \frac{1}{6}\Delta_t^3 \dot{a}_n + O(\Delta_t^4), \\ x_{n-1} &= x_n - \Delta_t v_n + \frac{1}{2}\Delta_t^2 a_n - \frac{1}{6}\Delta_t^3 \dot{a}_n + O(\Delta_t^4). \end{aligned} \tag{8}$$

Adding these two equations gives

$$x_{n+1} = 2x_n - x_{n-1} + \Delta_t^2 a_n + O(\Delta_t^4), \tag{9}$$

which is called the *Verlet algorithm*. It does not contain any velocities explicitly; the next x-value is obtained from two preceding x values.

The Verlet algorithm is in fact completely equivalent to the leapfrog method (7). To see this, consider first the difference $x_n - x_{n-1}$ that is contained in Eq. (9). To cubic order in an expansion about the half-point $x_{n+1/2}$ we have

$$\begin{aligned} x_n - x_{n-1} &= [x_{n-1/2} + (\Delta_t/2)v_{n-1/2} + \tfrac{1}{2}(\Delta_t/2)^2 a_{n-1/2} + \tfrac{1}{6}(\Delta_t/2)^3 \dot{a}_{n-1/2} + \ldots] && (10)\\ &\quad -[x_{n-1/2} - (\Delta_t/2)v_{n-1/2} + \tfrac{1}{2}(\Delta_t/2)^2 a_{n-1/2} - \tfrac{1}{6}(\Delta_t/2)^3 \dot{a}_{n-1/2} + \ldots] && (11)\\ &= \Delta_t v_{n-1/2} + O(\Delta_t^3), && (12) \end{aligned}$$

which we can use in Eq. (9) to obtain the form

$$x_{n+1} = x_n + \Delta_t v_{n-1/2} + \Delta_t^2 a_n + O(\Delta_t^4). \tag{13}$$

Treating the difference $v_{n+1/2} - v_{n-1/2}$ in the same way as done for the similar position difference in Eq. (12), we have

$$v_{n+1/2} - v_{n-1/2} = \Delta_t a_n + O(\Delta_t^3), \tag{14}$$

which confirms the assertion previously used in Eq. (6) but now conforming the size of the error. Using Eq. (14) in Eq. (13) we obtain

$$x_{n+1} = x_n + \Delta_t v_{n+1/2} + O(\Delta_t^4). \tag{15}$$

Comparing Eqs. (14) and (13) with Eqs. (7), we see that we have derived forms identical to the leapfrog algorithm. This, the Verlet and Leapfrog algorithms give identical results for $x(t)$, both with step error $O(\Delta^4)$. The step error in the velocity is $O(\Delta^3)$. In the Verlet form (9) the velocity does not appear explicitly, but it can still be obtained if needed by using Eq. (12) in the form

$$v_{n+1/2} = (x_n - x_{n-1})/\Delta_t + O(\Delta_t^2). \tag{16}$$

Here the step error is larger than in the Leapfrog method, but this is normally not a problem because the error is not propagating or accumulating from step to step. If neede be, three x values can be used to calculate the velocity with $P(\Delta^3)$ error at each step. In Sec. 2.3 we will consider the accumulated errors over an entire long time interval, which can be much larger than the individual step errors.

An important feature of the Leapfrog algorithm is that it is time-reversible, i.e., if we calculate x_{n+1} from x_n and $v_{n+1/2}$ according to Eqs. (7), and then reverse the time direction to go back to x_n, we use the same velocity $v_{n+1/2}$ as in the forward step, up to the change of its sign.. Therefore, in the absence of numerical round-off errors, we arrive back exactly at the original x_n. The same can be seen in the case of the Verlet formulation. This time-reversal symmetry is violated in the Euler metod, Eqs. (3), where we use v_n to go from x_n to x_{n+1}, but $-v_{n+1}$ to go backwards from x_{n+1} to x_n. If we carry out several leapfrog integration steps backwards in time, $t = t_N, \ldots, t_0$, after having completed a calculation from t_0 to t_N, we should arrive back at the original initial

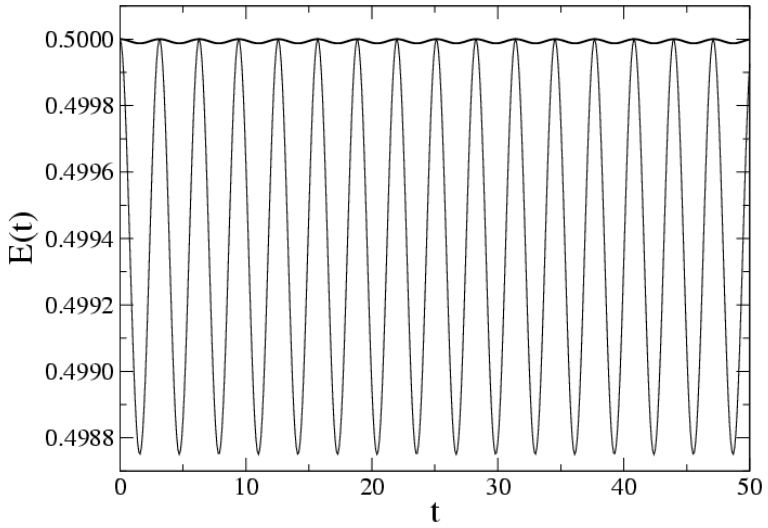

Figure 3: Time dependence of the energy of a harmonic oscillator with $k = m = 1$, integrated using the leapfrog method with $\Delta_t = 10^{-1}$ (thinner curve) and 10^{-2} (thicker curve).

conditions at $t = t_0$. This way the time-reversibility can in fact be used as a check of the sensitivity of a calculation to round-off errors (which will slightly break the symmetry).

An important consequence of the time-reversibility is that the errors are bounded for a system with periodic motion. This follows because integrating backward or forward in time for a full period T involves the acceleration at exactly the same x points (assuming now that the period is exactly a multiple of the time step). Hence, the error δ in some quantity, e.g., the energy, has to be the same at $t = \pm T$; $\delta(T) = \delta(-T)$. Integrating forward from $t = -T$ to 0, starting from the point reached in a previous integration from $t = 0$ to $-T$, we would expect an additional error similar to $\delta(T)$, because the initial conditions of these two integrations only differ by the very small amount $\delta(-T)$, so that the total error after a backward and a forward integration should be $\approx 2\delta(T)$. To be consistent with the time-reversibility, according to which this second integration in fact must bring us back to the original starting point and hence zero error, we must have $\delta(\pm T) = 0$. The error only vanishes completely at times $t - t_0 = T, 2T, \ldots$, but the important point is that there can be no steady error increase, which does affect methods with no time-reversal symmetry (as shown for the Euler method in Fig. 1). The general form of the long-time error scaling is discussed in detail in Sec. 2.3.

The bounded error of the leapfrog method when applied to systems with periodic motion is illustrated in Fig. 3, which shows results for the energy of the same harmonic oscillator as the one studied in Sec. 2.1 using the Euler method. The energy here oscillates with a period half of the periodicity 2π of the system. The vanishing of the error at every half period in this case is a consequence of the symmetric potential, which was not assumed in the discussion above. Note also how small the deviations are from the true energy $E = 1/2$ (compare with Fig. 1), even for Δ_t as large as 0.1.

There is yet a third equivalent formulation of the Verlet method, called the *velocity Verlet* algorithm.

It is obtained from the original Verlet algorithm as follows: Adding x_{n+1} on both sides of Eq. (9),

$$2x_{n+1} = x_{n+1} + 2x_n - x_{n-1} + \Delta_t^2 a_n + O(\Delta_t^4). \tag{17}$$

we can define the velocity using a two-step difference;

$$v_n = \frac{1}{2\Delta_t}(x_{n+1} - x_{n-1}), \tag{18}$$

which when used in (17) gives the position in terms of x and v at the previous step only;

$$x_{n+1} = x_n + \Delta_t v_n + \frac{1}{2}\Delta_t^2 a_n. \tag{19}$$

In order to obtain an equation for the velocity, we first write the original Verlet equation (9) for x_n instead of x_{n+1};

$$x_n = 2x_{n-1} - x_{n-2} + \Delta_t^2 a_{n-1}, \tag{20}$$

which we add to (9). Rearranging the result in the following way;

$$x_{n+1} - x_{n-1} = x_n - x_{n-2} + \Delta_t^2(a_{n-1} + a_n), \tag{21}$$

and using the velocity definition (18), we obtain an equation for v_n;

$$v_n = v_{n-1} + \frac{1}{2}\Delta_t(a_{n-1} + a_n). \tag{22}$$

Shifting the time step by one and again writing down Eq. (19) for the position, we arrive at the velocity Verlet algorithm:

$$\begin{aligned} x_{n+1} &= x_n + \Delta_t v_n + \frac{1}{2}\Delta_t^2 a_n, \\ v_{n+1} &= v_n + \frac{1}{2}\Delta_t(a_{n+1} + a_n). \end{aligned} \tag{23}$$

This formulation of the Verlet algorithm is completely equivalent to (9) and (7) as far as the propagation of the position is concerned. It may preferrable to the leapfrog method in cases where there is some reason to use the same time grid for x and v, but note that more operations are required at each step of the velocity Verlet algorithm (however, the number of calls to the force/acceleration function is the same, which is typically the part dominating the processor time). It should also be noted that although the algorithms give identical results (up to round-off errors) for x_n, the accuracy in the velocity is four times higher in the leapfrog method becuase it is there defined using x points separated by a single time step, instead of two steps in the velocity Verlet method. However, nothing of course prohibits us from also calculating $v_{n+1/2}$ in terms of x_n and x_{n+1} in the velocity Verlet method or v_n in terms of x_{n-1} and x_{n+1} in the leapfrog method. In view of this, all results are completely equivalent in the two formulations.

2.3 Error build-up in the Verlet/leapfrog method

As we have seen above, the single-step error in the Verlet/leapfrog algorithm is $O(\Delta_t^4)$ for the position and $O(\Delta_t^2)$ for the velocity. We are clearly also very interested in the accumulated errors after a large number of steps have been carried out.

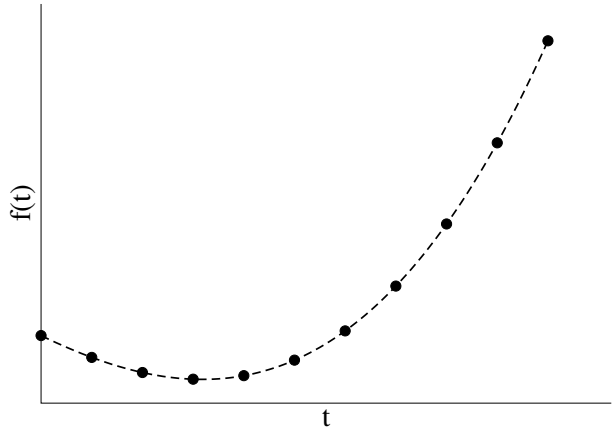

Figure 4: A function that is evaluated on a grid with spacing Δ_t (solid circles) can be approximated by an interpolating polynomial between those points (dashed curve), provided that the function is smooth on the scale Δ_t.

To study the accumulated error in the position x calculated using the Verlet/leapfrog method, we use the original Verlet form of the algorithm, Eq. (9). We introduce a symbol for the deviation δ_n of the calculated value x_n from the exact solution $x_n^{\rm ex}$,

$$x_n = x_n^{\rm ex} + \delta_n, \tag{24}$$

and insert this in Eq. (9). After rearranging, this gives

$$\delta_{n-1} - 2\delta_n + \delta_{n+1} = -(x_{n-1}^{\rm ex} - 2x_n^{\rm ex} + x_{n+1}^{\rm ex}) + \Delta_t^2 a_n + O(\Delta_t^4). \tag{25}$$

Under the assumption that the true solution, the acceleration (i.e., the force), and the deviation are all smooth functions on a time scale set by Δ_t (which is true for Δ_t smaller than some $\Delta_t^{\rm max}$ if the true solution and the force are well behaved functions), we can imagine constructing high-order polynomials that go through all the (t_n, δ_n) and $(t_n, x_n^{\rm ex})$ points, as illustrated in Fig. 4. We can then use these interpolating polynomials to work in continuum time and use the second derivative of a function [$f(t) = \delta(t)$ or $x^{\rm ex}(t)$] in place of the discretized versions appearing in Eq. (25),

$$\frac{d^2 f(t_n)}{dt^2} = \frac{1}{\Delta_t}\frac{d}{dt}[f(t_{n+1/2}) - f(t_{n-1/2})] = \frac{1}{\Delta_t^2}(f_{n-1} - 2f_n + f_{n+1}), \quad \Delta_t \to 0, \tag{26}$$

to obtain from (25)

$$\ddot{\delta}(t_n) = -\ddot{x}^{\rm ex}(t_n) + a(t_n) + O(\Delta_t^2), \tag{27}$$

even though Δ_t is fixed and finite. The error introduced by this "quasi-continuum" approach is given by the order of the imagined interpolating polynomials and will clearly be much smaller than the $O(\Delta_t^4)$ error of the Verlet formula we are working with (under the assumption of smoothness on the scale set by Δ_t). By definition of the exact solution, $\ddot{x}^{\rm ex}(t_n) = a(t_n)$, and hence in Eq. (25) we are left with

$$\ddot{\delta}(t_n) \sim \Delta_t^2. \tag{28}$$

We can write an expression for the error $\delta(T)$ after a large number of integration steps in terms of the second derivative ($T = t_N - t_0$; we will set $t_0 = 0$):

$$\delta(T) = \int_0^T dt \dot{\delta}(t) + \delta(0) = \int_0^T dt \left(\int_0^t dt' \ddot{\delta}(t') - \dot{\delta}(0) \right) + \delta(0). \tag{29}$$

By definition of the initial conditions, the error $\delta(0) = 0$. Since the Verlet formula is completely symmetric with respect to reversal of the time direction ($n \to -n$), the Taylor expansion of the error around t_0 can have no contribution of first order (or any odd order), and hence also $\dot{\delta}(0) = 0$. With the second derivative bounded by the quadratic form (28), the integral (29) clearly can scale no worse than as $T^2\Delta_t^2$. Hence, the error in x after a finite number of steps scales in the worst case scenario as Δ_t^2 in the time discretization and as T^2 in the total integration time. Since there is an unknown prefactor in the Δ_t dependence of the error in Eq. (28)—the sign can even be mixed—the T-scaling can be much better in practice, as we have seen explicitly in Fig. 3 in the case of the harmonic oscillator. Note again that the above arguments hold only when Δ_t is sufficiently small for the solution to be smooth on this scale. The way the velocity is defined as a discrete derivative of the coordinate, its error will clearly also scale as Δ_t^2 for any t.

2.4 Verlet/leapfrog methods for damped systems

In the derivation of the Verlet algorithm, and its leapfrog formulation, we assumed that the force-function contains no explicit velocity dependence. With a velocity dependent force, $F(x, v, t)$, the problem is that the acceleration $a_n = a(x_n, v_n, t_n)$ in (7) should be evaluated at the time-points corresponding to the position x_n, on which we do not have the velocity. To circumvent this problem, we first separate the damping term from the rest of the force terms and write

$$a(x, v, t) = \frac{1}{m}[F(x, t) - G(v)]. \tag{30}$$

If we simply approximate $a(x_n, v_n, t_n)$ by $[F(x_n, t_n) - G(v_{n-1/2})]/m$, we make an error of order Δ_t. Since a is multiplied by Δ_t^2 to obtain x_{n+1}, the resulting algorithm acquires a contribution to the x error scaling as Δ_t^3; an order lower than the Verlet algorithm without damping term. To remedy this, we can use a scheme based on an intermediate estimation $\hat{x}_{n+1}$ of the position, obtained using the above approximation $G(v_{n-1/2})$ in place of $G(v_n)$. We can subsequently correct for the $O(\Delta_t^3)$ error in $\hat{x}_{n+1}$ by doing a second step, where we use a better approximation of the velocity; $v_n = (\hat{x}_{n+1} - x_{n-1})/(2\Delta_t)$. The error in this velocity, and hence in $G(v_n)$, is $O(\Delta_t^2)$, i.e., one order smaller than in the first approximation and sufficient to render x_{n+1} calculated with it accurate to the same order as in the original leapfrog algorithm. To summarice this modified algorithm, these are the steps to be performed in the leapfrog version of the Verlet algorithm with a damping term:

$$\begin{aligned}
\hat{v}_{n+1/2} &= v_{n-1/2} + \Delta_t[F(x_n, t_n) - G(v_{n-1/2})]/m, \\
\hat{x}_{n+1} &= x_n + \Delta_t \hat{v}_{n+1/2}, \\
v_n &= (\hat{x}_{n+1} - x_{n-1})/(2\Delta_t), \\
v_{n+1/2} &= v_{n-1/2} + \Delta_t[F(x_n, t_n) - G(v_n)]/m, \\
x_{n+1} &= x_n + \Delta_t v_{n+1/2}.
\end{aligned} \tag{31}$$

This algorithm requires more work than the simple leapfrog method without damping. However, in cases where the processor time is dominated by evaluating the function F, the differences are in practice only minor.

In the important special case of linear damping, i.e.,

$$G(v) = \gamma v = \frac{x_{n+1} - x_{n-1}}{2\Delta_t} + O(\Delta_t^2), \tag{32}$$

the algorithm can be simplified. Starting from the Verlet form (9) we can write

$$x_{n+1} = 2x_n - x_{n-1} + \frac{\Delta_t^2}{m}[F_n - \gamma(x_{n+1} - x_{n-1})/(2\Delta_t)] + O(\Delta_t^4), \tag{33}$$

which we can rearrange as

$$x_{n+1}(1 + \gamma\Delta_t/2m) = 2x_n - x_{n-1}(1 - \gamma\Delta_t/2m) + \frac{\Delta_t^2}{m}F_n + O(\Delta_t^4), \tag{34}$$

or

$$\begin{aligned} x_{n+1} &= x_n + \frac{(1 - \gamma\Delta_t/2m)(x_n - x_{n-1}) + \Delta_t^2 F_n/m}{1 + \gamma\Delta_t/2m}, \\ &= x_n + \frac{\Delta_t(1 - \gamma\Delta_t/2)v_{n-1/2} + \Delta_t^2 F_n}{1 + \gamma\Delta_t/2}. \end{aligned} \tag{35}$$

In this expression we can identify the velocity $v_{n+1/2}$ and obtain the leapfrog algorithm in the presence of linear damping;

$$\begin{aligned} v_{n+1/2} &= \frac{(1 - \gamma\Delta_t/2m)v_{n-1/2} + \Delta_t F_n/m}{1 + \gamma\Delta_t/2m}, \\ x_{n+1} &= x_n + \Delta_t v_{n+1/2}. \end{aligned} \tag{36}$$

Using the velocity Verlet algorithm (23) with damping, we have a problem analogous to that in the leapfrog method; to evaluate v_{n+1} we need a_{n+1}, which itself depends explicitly on v_{n+1}. We can proceed in a way similar to what we did above, first obtaining an estimate $\hat{v}_{n+1}$ based on approximating $a_{n+1} = a(x_{n+1}, v_{n+1}, t_{n+1})$ by $a(x_{n+1}, v_n + a_n\Delta_t, t_{n+1})$ and then refining;

$$\begin{aligned} x_{n+1} &= x_n + \Delta_t v_n + \frac{1}{2}\Delta_t^2 a_n, \\ \hat{v}_{n+1} &= v_n + \frac{1}{2}\Delta_t[a_n + a(x_{n+1}, v_n + \Delta_t a_n, t_{n+1})], \\ v_{n+1} &= v_n + \frac{1}{2}\Delta_t[a_n + a(x_{n+1}, \hat{v}_{n+1}, t_{n+1})]. \end{aligned} \tag{37}$$

Also in this case the algorithm can be further simplified in the case of linear damping. The expression corresponding to Eq. (21) in the presence of linear damping is

$$x_{n+1} - x_{n-1} = x_n - x_{n-2} + \frac{\Delta_t^2}{m}(F_{n-1} + F_n) - \frac{\gamma}{2}\Delta_t(x_{n+1} - x_{n-1} + x_n - x_{n-2}), \tag{38}$$

which leads to the following algorithm [the formula for the position in the original velocity Verlet algorithm with no damping, Eq. (23), remains unchanged]:

$$\begin{aligned} x_{n+1} &= x_n + \Delta_t v_n + \frac{1}{2}\Delta_t^2 a_n, \\ v_{n+1} &= \frac{1}{1 + \Delta_t\gamma/2m}[v_n(1 - \Delta_t\gamma/2m) + \frac{\Delta_t}{2m}(F_{n-1} + F_n)]. \end{aligned} \tag{39}$$

2.5 The Runge-Kutta method

The Runge-Kutta (RK) method can be considered the most classic of all high-order schemes. There is a whole range of methods of this type of different orders, but the RK name is most commonly associated with the fourth-order variant [discretization error $O(\Delta_t^5)$]. We will here outline one way of constructing this algorithm; however, without proving the error scaling. We will first give a complete proof of the much simpler second-order RK method.

For simplicity, before considering the equations of motion, we will consider the slightly easier case of a single first-order differential equation,

$$\dot{x}(t) = f[x(t), t]. \tag{40}$$

The second-order RK method for this equation corresponds to the mid-point rule of function integration: If we know the value of the function $f[x(t), t]$ at the mid-point of an interval $[t_n, t_{n+1}]$, its integral is known up to an error $O(\Delta_t^3)$:

$$\int_{t_n}^{t_{n+1}} f[x(t), t]dt = \Delta_t f[x(t_{n+1/2}), t_{n+1/2}] + O(\Delta_t^3). \tag{41}$$

In the case of a function $f(t)$, we could simply evaluate it at $t_{n+1/2}$, but in the present case we do not know the value of $x_{n+1/2} = x(t_{n+1/2})$ and hence we need to approximate it. We only need an approximant linear in Δ_t to keep the error of the integral cubic, so we can simply use the Euler formula for (40):

$$\hat{x}_{n+1/2} = x_n + \frac{\Delta_t}{2} f(x_n, t_n) + O(\Delta_t^2), \tag{42}$$

where $\hat{}$ is again used to indicate an intermediate step of the calculation. We then arrive at the second-order RK formula,

$$x_{n+1} = x_n + \Delta_t f(\hat{x}_{n+1/2}, t_{n+1/2}) + O(\Delta_t^3), \tag{43}$$

which is typically written in the algorithm form

$$\begin{aligned} k_1 &= \Delta_t f(x_n, t_n), \\ k_2 &= \Delta_t f(x_n + k_1/2, t_{n+1/2}), \\ x_{n+1} &= x_n + k_2. \end{aligned} \tag{44}$$

In the case of the equations of motion for x and v, we use the Euler formula for both quantities in the first step to obtain approximants for $x_{n+1/2}$ and $v_{n+1/2}$ and then use these to obtain better estimates in the same way as we did above, in Sec. 2.4, giving

$$\begin{aligned} k_1 &= \Delta_t a(x_n, v_n, t_n), \\ l_1 &= \Delta_t v_n, \\ k_2 &= \Delta_t a(x_n + l_1/2, v_n + k_1/2, t_{n+1/2}), \\ l_2 &= \Delta_t (v_n + k_1/2), \\ v_{n+1} &= v_n + k_2, \\ x_{n+1} &= x_n + l_2. \end{aligned} \tag{45}$$

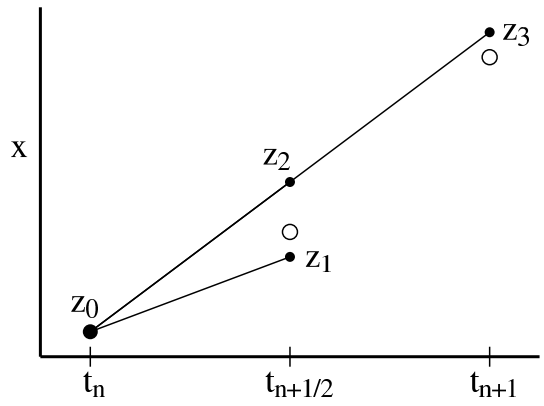

Figure 5: Illustration of the scheme used to estimate the values of the function f at the true (t, x) points $(t_{n+1/2}, x_{n+1/2})$ and (t_{n+1}, x_{n+1}) (here shown as open circles) based on the known point $z_0 = (t_n, x_n)$ (large solid circle). A point z_1 is first generated using $f(z_0)$ as the derivative. The points z_2 and z_3 are obtained using $f(z_1)$ as the derivative.

This algorithm is rarely used in practice, since the Verlet/leapfrog method is both simpler and more accurate. The algorith does, however, serve as a good warm-up for studying the significantly more complicated fourth-order RK algorithm.

We again first consider the single equation (40). Again using results from simple function integration, we know that if the values of the function $f[x(t), t]$ are evaluated at the time points $t_n, t_{n+1/2}$, and t_n, we can apply Simpson's formula to obtain the integral over the range $[t_n, t_{n+1}]$ up to an error $O(\Delta_t^5)$, i.e.,

$$x_{n+1} = x_n + \frac{\Delta_t}{6}(f_n + 4f_{n+1/2} + f_{n+1}) + O(\Delta_t^5). \tag{46}$$

We hence need to find approximations for $f_{n+1/2}$ and f_{n+1} up to errors $O(\Delta_t^4)$. The somewhat obscure scheme used for this purpose is illustrated in Fig. 5. As in the second-order derivation above, we begin by using an Euler formula to obtain an approximation for $x_{n+1/2}$. This time there will be two such estimates, and we use a superscript to distinguish between them;

$$\begin{aligned} \hat{x}^1_{n+1/2} &= x_n + \frac{\Delta_t}{2} f(x_n, t_n) \\ \hat{x}^2_{n+1/2} &= x_n + \frac{\Delta_t}{2} f(\hat{x}^1_{n+1/2}, t_{n+1/2}). \end{aligned} \tag{47}$$

Since f is the derivative of x, what we have done is to first extrapolate $x_{n+1/2}$ from x_n using derivative information at the initial point (x_n, t_n), denoted z_0 in Fig. 5. The second extrapolation is based on derivative information at the estimated point $(\hat{x}^1_{n+1/2}, t_{n+2})$, denoted z_1 in the figure, but still using (x_n, t_n) as the point of departure. This second point at $t_{n+1/2}$ is denoted z_2 in the figure. The idea of this procedure is, roughly, that if the function f is smooth on the scale Δ_t, the actual point $x_{n+1/2}$ should be between z_1 and z_2. In the RK method the desired function value $f_{1+n/2}$ is approximated by the average of the function at these two estimated points;

$$f_{n+1/2} \approx \frac{1}{2}[f(x^1_{n+1/2}, t_{n+1/2}) + f(x^2_{n+1/2}, t_{n+1/2})]. \tag{48}$$

We will not prove here the remarkable fact that this approximation has the required accuracy, i.e., an $O(\Delta_t^4)$ error. In order to estimate x_{n+1}, the function value at $(\hat{x}^2_{n+1/2}, t_{n+1/2})$ (i.e., z_2) is used to obtain an approximation of the derivative,

$$\hat{x}_{n+1} = x_n + \Delta_t f(x^2_{n+1/2}, t_{n+1/2}). \tag{49}$$

This is in the spirit that the derivative should be evaluated at the mid-point between (x_n, t_n) and (x_{n+1}, t_{n+1}). Not knowing the actual point, we use the estimate z_2 for this. Again, it can be proven that this leads to an error $O(\Delta_t^4)$ in x_{n+1}. The fourt-order RK algorithm resulting from the above procedures is traditionally written as

$$\begin{aligned} k_1 &= \Delta_t f(x_n, t_n), \\ k_2 &= \Delta_t f(x_n + k_1/2, t_{n+1/2}), \\ k_3 &= \Delta_t f(x_n + k_2/2, t_{n+1/2}), \\ k_4 &= \Delta_t f(x_n + k_3, t_{n+1}), \\ x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned} \tag{50}$$

The correctness of this scheme up to an error of order $O(\Delta_t^5)$ can be proven using a Taylor expansion.

Before adapting the RK algorithm to equations of motion, we write down a more general form of the RK scheme for two coupled equations;

$$\begin{aligned} \dot{x}(t) &= f(x, y, t), \\ \dot{y}(t) &= g(x, y, t), \end{aligned} \tag{51}$$

for which the RK algorithm generalizes to

$$\begin{aligned} k_1 &= \Delta_t f(x_n, y_n, t_n), \\ l_1 &= \Delta_t g(x_n, y_n, t_n), \\ k_2 &= \Delta_t f(x_n + k_1/2, y_n + l_1/2, t_{n+1/2}), \\ j_2 &= \Delta_t g(x_n + k_1/2, y_n + l_1/2, t_{n+1/2}), \\ k_3 &= \Delta_t f(x_n + k_2/2, y_n + l_2/2, t_{n+1/2}), \\ l_3 &= \Delta_t g(x_n + k_2/2, y_n + l_2/2, t_{n+1/2}), \\ k_4 &= \Delta_t f(x_n + k_3, y_n + l_3, t_{n+1}), \\ l_4 &= \Delta_t g(x_n + k_3, y_n + l_3, t_{n+1}), \\ x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ y_{n+1} &= y_n + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4), \end{aligned} \tag{52}$$

which in turn easily generalizes to any number of coupled equations.

In the case of the equations of motion, we make the identification $x \to v$, $y \to x$, $f \to a$, $g \to v$

(note that v corresponds to a function $g(x, y, t) = x$ in the notation used above). We then obtain

$$\begin{aligned}
k_1 &= \Delta_t a(x_n, v_n, t_n), \\
l_1 &= \Delta_t v_n, \\
k_2 &= \Delta_t a(x_n + l_1/2, v_n + k_1/2, t_{n+1/2}), \\
l_2 &= \Delta_t (v_n + k_1/2), \\
k_3 &= \Delta_t a(x_n + l_2/2, v_n + k_2/2, t_{n+1/2}), \\
l_3 &= \Delta_t (v_n + k_2/2), \\
k_4 &= \Delta_t a(x_n + l_3, v_n + k_3, t_{n+1}), \\
l_4 &= \Delta_t (v_n + k_3), \\
v_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\
x_{n+1} &= x_n + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4).
\end{aligned} \tag{53}$$

The RK method is clearly not time-reversal symmetric, and hence the errors are unbounded even for periodic motion. Fig. 6 shows the time dependence of the energy obtained using the RK method for the same harmonic oscillator that was previously studied with the Euler (Fig. 1) and leapfrog (Fig. 3) methods. Comparing the RK and leapfrog methods at $\Delta = 0.1$, it can be seen that the errors are considerably smaller in the case of the RK method within the time range shown. However, the error grows linearly with time, and hence for long times the leapfrog method will in fact perform better in this case although its single-step error scaling is worse. However, for aperiodic motion, or motion with a very long period, there may be no practical advantage of the bounded error of the leapfrog method—the error can still grow large within a period—and then the RK method is often preferrable.

An advantage of the RK method is that it can be used with a variable time step. This is not possible in the Verlet/leapfrog method, because data at t_n and $t_{n+1/2}$ are needed to advance the time to t_{n+1}. Using different time-steps $t_{n+1} - t_n$ and $t_n - t_{n-1}$ would clearly ruin the symmetry of the scheme and lead to a worse error scaling. In the RK method, only data at t_n are needed to advance to t_{n+1}, and hence Δ_t can be chosen arbitrarily in each step. There are adaptive methods that use this property of the Runge-Kutta scheme to automatically adjust the time step to keep the error within specified limits.

2.6 Motion in more than one dimension

The methids we have discussed above generalize very easily to motion in more than one dimension. The equations of motion become vector equations;

$$\begin{aligned}
\dot{x}(t) &= \vec{v}(t) \\
\dot{\vec{v}}(t) &= \frac{1}{m}\vec{F}[\vec{x}(t), \vec{v}(t), t],
\end{aligned} \tag{54}$$

which separate into equations for the components x_α, v_α ($\alpha = 1, 2, 3$ in three dimensions). These equations are coupled only through the force function, the components F_α of which typically depend on all components of $\vec{x}$ and, in the case of damped motion, $\vec{v}$.

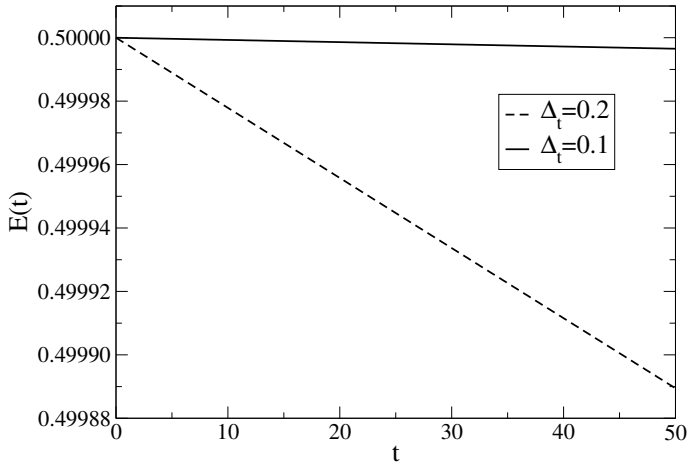

Figure 6: Time dependence of the energy of a harmonic oscillator with $k = m = 1$ integrated using the 4th-order Runge-Kutta method with time steps $\Delta_t = 0.2$ and 0.1.

As a simple example, we consider planetary motion, which takes place in a single plane (in the case of a single planet considered here), i.e., we study the equations of motion in two dimensions. For a planet of mass m moving in the gravitational field of a star with mass M, the force experienced by m is given by

$$\vec{F}(r) = -\frac{GMm}{r^3}\vec{r}, \tag{55}$$

where $\vec{r}$ is the position vector of the planet with respect to the star and $r = |\vec{r}|$. In principle this is a two-body problem, but a result of elementary mechanics is that it can be reduced to a one-body problem for a reduced mass $\mu = Mm/(m + M)$. We will here for simplicity assume that $M \gg m$ and treat the star as a stationary body, using the original m for the mass of the planet. We then have the equations of motion for the x and y coordinates of the planet:

$$\begin{aligned} \dot{x} &= v_x, \\ \dot{v}_x &= -GMx/r^3, \\ \dot{y} &= v_y, \\ \dot{v}_y &= -GMy/r^3, \end{aligned} \tag{56}$$

where $r = \sqrt{x^2 + y^2}$. The leapfrog algorithm (7) for these equations is

$$\begin{aligned} v_x(n+1/2) &= v_x(n-1/2) - \Delta_t GMx(n)[x^2(n) + y^2(n)]^{-3/2}, \\ x(n+1) &= x(n) + \Delta_t v_x(n+1/2), \\ v_y(n+1/2) &= v_y(n-1/2) - \Delta_t GMy(n)[x^2(n) + y^2(n)]^{-3/2}, \\ y(n+1) &= y(n) + \Delta_t v_y(n+1/2). \end{aligned} \tag{57}$$

This scheme of course generalizes to other types of forces. The Runge-Kutta method can also easily be used in more than one dimension.