ward Euler method are often used for such cases. We will encounter one such case in Section 8.3.1.

2.4 The leapfrog method

So far we have discussed solutions of ODEs purely from the mathematical point of view. However, there are physical systems possessing special properties that should be respected in the numerical solution. Hamiltonian systems fall into this category. Take the simple harmonic oscillator as an example. The motion is oscillatory and time reversible, i.e., if the velocity at a certain point in its path is reversed, the trajectory will move backward over the original path, as if time is reversed, $t \rightarrow -t$. The motion is also area preserving in phase space which is a multidimensional space consisting of all coordinates and momenta [31]. Two phase space trajectories for the harmonic oscillator are shown in Figure 2.5.

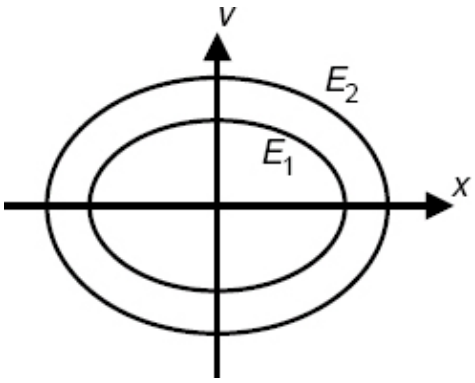

Figure 2.5: Phase space plot of the harmonic oscillator for two energies, $E_2 > E_1$.

Their shape is an ellipse, controlled by the energy

$$E = \frac{1}{2}mv^2 + \frac{1}{2}m\omega^2 x^2, \quad m = \text{mass}, \ \omega = \text{frequency}. \tag{2.38}$$

For a given energy E, the area within each ellipse is proportional to E and is constant because energy is conserved. The Euler and the Runge-Kutta methods do not preserve these properties. Below we discuss the leapfrog method that does (also known as the Verlet method [92]).

Let us start with the equations of motion of a one-dimensional Hamiltonian system,

$$\frac{dx}{dt} = v, \tag{2.39}$$

$$\frac{dv}{dt} = a(x), \tag{2.40}$$

where $a(x) = F(x)/m$ is the acceleration. The force $F(x)$ depends on the coordinate x only. For the harmonic oscillator, $a(x) = -\omega^2 x$. The important thing is that the RHS of the first equation (2.39) depends on the velocity v only, and the second equation (2.40) on the coordinate x only. This separation of coordinates and velocity is required for the leapfrog method (Appendix 2.A).

Given the values x_0 and v_0 at t, the leapfrog method calculates the values x_1 and v_1 at $t + h$ in three steps,

$$x_{1/2} = x_0 + v_0 \frac{h}{2}, \tag{2.41}$$

$$v_1 = v_0 + a(x_{1/2})h, \tag{2.42}$$

$$x_1 = x_{1/2} + v_1 \frac{h}{2}. \tag{2.43}$$

First, half a step is taken to obtain the position $x_{1/2}$ at the midpoint. Then a full step is taken to calculate the new velocity v_1 using acceleration evaluated at $x_{1/2}$. Finally, the new position x_1 is found by taking another half a step, using the new velocity. Each step uses the newest values as they become available.

If the leapfrog method looks a lot like the midpoint (RK2) method, it partially is, at least in the order of accuracy. The leapfrog method is second order, as is the midpoint method. But there is a big difference: the leapfrog method is area-preserving. To understand this property, let us think of the algorithm as a transformation in phase space that maps the points (x, v) at t to the new points (q, p) at $t+h$. Figure 2.6 illustrates such a transformation.

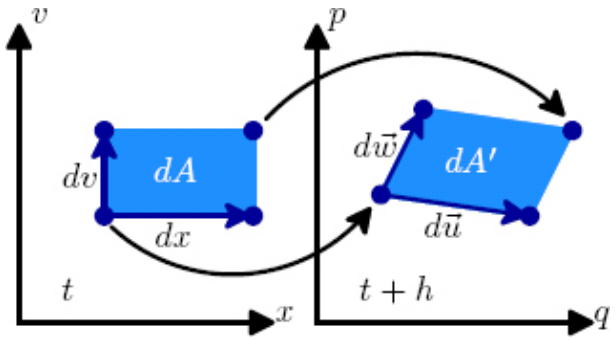

Figure 2.6: Schematic of area transformation. The transformation maps the points in phase space from $[x, v]$ at t to $[q, p]$ at $t + h$, e.g., $[x, v] \to [q(x, v), p(x, v)]$ for the lower-left corner, similarly for other points. In the leapfrog method, the area is preserved, $dA' = dA$.

The corners of the rectangular area element at t are transformed to new points at $t + h$. When h is small enough, the transformed points form an approximate parallelogram, whose area dA' is given by the cross product of the vectors $d\vec{u}$ and $d\vec{w}$. We can associate a transformation with each step in the leapfrog method represented by Eqs. (2.41) to (2.43). Using these transformations, it can be shown that the area is preserved, $dA' = dA$. Details are given in Section 2.A. The leapfrog method is also time reversible. We leave its proof to Exercise E2.4.

We wish to write a standalone leapfrog subroutine for a system of ODEs more general than Eqs. (2.39) and (2.40). Similar to

Eq. (2.27), let us introduce the coordinate and velocity vectors as $\{r\} = r_1, r_2, ..., r_n$ and $\{v\} = v_1, v_2, ..., v_n$, respectively. The system of ODEs can be written as two sets, with $i = 1, 2, ..., n$,

$$\frac{dr_i}{dt} = f_i(\{v\}, t), \tag{2.44}$$

$$\frac{dv_i}{dt} = g_i(\{r\}, t). \tag{2.45}$$

The function $f_i(\{v\}, t)$ depends on the velocity $\{v\}$ only, and $g_i(\{r\}, t)$ on the coordinate $\{r\}$ only. Again we need the separation of $\{r\}$ and $\{v\}$ for the leapfrog method to be applicable. We can think of $f_i(\{v\}, t)$ as the generalized velocity, and $g_i(\{r\}, t)$ the generalized acceleration. For Hamiltonian systems, they become the actual velocity and acceleration. For others, such as time-dependent quantum systems discussed in Chapter 8, they are pseudo velocity and acceleration.

By vectorizing the leapfrog method (2.41) to (2.43) in a way analogous to Eq. (2.16), we can write a leapfrog solver for Eqs. (2.44) and (2.45) as follows.

Program listing 2.6: The leapfrog method (`leapfrog.py`)

```
def leapfrog(lfdiffeq, r0, v0, t, h):      # vectorized leapfrog
    """ vector leapfrog method using numpy arrays.
        It solves general (r,v) ODEs as:
        dr[i]/dt = f[i](v), and dv[i]/dt = g[i](r).
        User supplied lfdiffeq (id, r, v, t) returns
        f[i](r) if id=0, or g[i](v) if id=1.
        It must return a numpy array if i>1 """
    hh = h/2.0
     r1 = r0 + hh*lfdiffeq(0, r0, v0, t)        # 1st: r at h/2
```

```
    using v0
        v1 = v0 + h* lfdiffeq (1, r1, v0, t+hh)  # 2nd: v1 using a(r) at h/2
        r1 = r1 + hh*lfdiffeq(0, r0, v1, t+h)     # 3rd: r1 at h using v1
        return r1, v1
```

Like `RK2/RK4` requiring `diffeq`, `leapfrog` needs a user supplied function that returns the RHS of Eqs. (2.44) and (2.45). This function is defined as `lfdiffeq(id, r, v, t)`. Note that by our convention, the leapfrog method assumes a different interface in the derivative function (`lfdiffeq`) than the Runge-Kutta methods. See Program 2.7 for an example.

Here, `lfdiffeq` is basically the same as `diffeq` except for the `id` flag: when `id=0`, `lfdiffeq` returns $f_i(\{v\}, t)$, and when `id=1`, it returns $g_i(\{r\}, t)$. We could have split it into two functions, but it is generally more convenient to have just one function. As before, we put `leapfrog` in the file `ode.py`, which should be imported before any ODE solver is used.

THE HARMONIC OSCILLATOR WITH THE LEAPFROG METHOD

As an example, let us apply the leapfrog method to a Hamiltonian system, the simple harmonic oscillator. The equations of motion analogous to Eqs. (2.39) and (2.40) are

$$\frac{dx}{dt} = v, \quad \frac{dv}{dt} = -\omega^2 x. \tag{2.46}$$

The solution is a perfect harmonic,

$$x = A\cos(\omega t + \varphi), \quad v = -\omega A \sin(\omega t + \varphi). \tag{2.47}$$

The constants A and φ are the amplitude and initial phase, respectively. We see that $x^2/A^2 + v^2/(\omega A)^2 = 1$, indeed an ellipse in phase space.

The program to solve Eq. (2.46) numerically is as follows (ω = 1 rad/s).

Program listing 2.7: Oscillator with leapfrog method (`sho_lf.py`)

```
import ode                          # get ODE solvers
import math as ma                   # get math functions
import matplotlib.pyplot as plt     # get matplotlib plot functions

def oscillator (id, x, v, t):       # return dx/dt or dv/dt
    if (id==0):                     # calc velocity
        return v
    else:                           # calc acceleration
        return −x                   # −ω²x, ω = 1,

def go():                           # main program
    x, v = 1.0, 0.0                 # initial values
    t, h = 0.0, 0.1                 # init time, step size
     xa, va = [],[]                 # declare arrays for plotting
    while t<4*ma.pi:                # loop for two periods
          xa.append(x)              # record position and velocity
          va.append(v)
          x, v = ode.leapfrog(oscillator, x, v, t, h) # solve it
          t = t + h

    plt.figure ()                   # start a figure
    plt.plot(xa,va)                 # plot vel−pos
    plt.xlabel('x (m)')             # add labels
    plt.ylabel('v (m/s)')
    plt.show()                      # show figure
```

```
27 go()                    # run the program
```

The program first imports several libraries: our ODE solvers in `ode.py`, standard mathematical functions, as well as Matplotlib. According to the leapfrog scheme (2.44) and (2.45), we have $n = 1$ in Eq. (2.46) (with r replaced by x), and $f_1(v, t) = v$, $g_1(x, t) = -x$. Therefore, the function `oscillator` returns v if `id=0`, and $-x$ otherwise. The main program `go()` is very similar to the one in Program 2.5. It sets up the initial values, and enters the `while` loop which runs for two periods, storing the position and velocity at each time step before advancing to the next step with `leapfrog`. The velocity-position curve in phase space is plotted after the loop.

Figure 2.7 (top) shows a sample plot from the program above. For comparison, results using the Euler and RK2 methods are also shown (generated separately, see Project P2.4). Starting from the initial point (1, 0), the path by Euler's method spirals outward noticeably. The RK2 and leapfrog methods produce ellipses similar to what is expected from Figure 2.5. Although hard to see on the scale shown, the path of the RK2 method is not closed and the area is not preserved. If we wait long enough, it would spiral outward eventually, but at a slower pace than Euler's method. In contrast, the leapfrog method has a closed path, thus preserving the area.

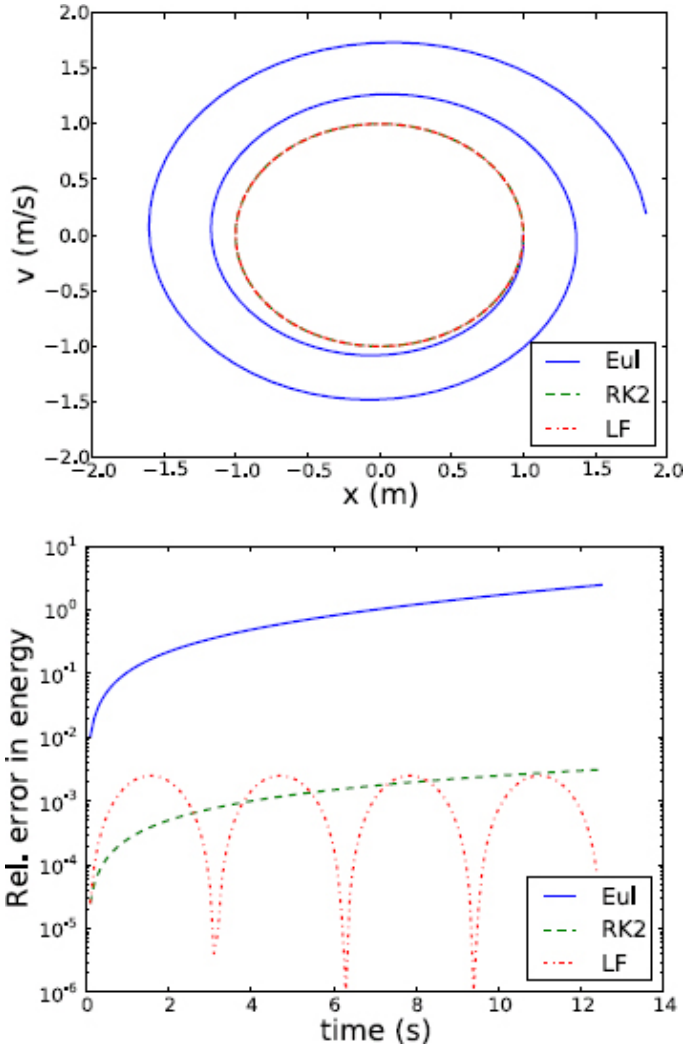

Figure 2.7: Phase space plot of the harmonic oscillator (top), and the relative error in energy (bottom) for $\omega = 1$ rad/s. The results are calculated with the Euler (Eul), RK2, and the leapfrog (LF) methods.

We can get a clearer picture by considering the relative error in energy defined as $|E_{numeric} - E_{exact}|/E_{exact}$.[3] The numerical energy $E_{numeric}$ can be obtained by slightly modifying Program 2.7. The results are shown in Figure 2.7 (bottom) as a semilog plot. When plotting results that vary over orders of magnitude, the logarithmic scale (semilog or log-log) should be used to show clearly the difference. The energy by Euler's method increases with time. The RK2 method shows a similar trend, albeit with an offset and