Lecture 11

Applying ODE solvers: Trajectories

- ◆ 1st order equations: chasing targets
- ◆ 2nd order ODEs of Newtonian dynamics
- ◆ Practical ways to implement leapfrog integration scheme
- ◆ Conservative systems (conservation of energy)
- ◆ Non-conservative systems: friction, aerodynamic drag etc.
- ◆ Chaos sometimes arises in dimensionality > 2 (number of variables > 2).

Applying ODE solvers: Trajectories & time-dependence

- 1st-order systems trajectories of chase – example: dog and duck ← please read about it and see graphics at the end of our course page (required material).
- Motion without friction – the physical pendulum
  constant arm length $r = 1$ → 1-D system, variable = angle φ
  - Equation of motion: $d^2\varphi/dt^2 = - g \sin \varphi$,
  - Initial conditions: $\varphi(0) = \varphi_0$, $d\varphi/dt\,(0) = 0$
  - Has energy integral (energy conservation law):

$$E = U + K = -g \cos \varphi + {}^1/_2\,(r\, d\varphi/dt)^2 = \text{const.}$$

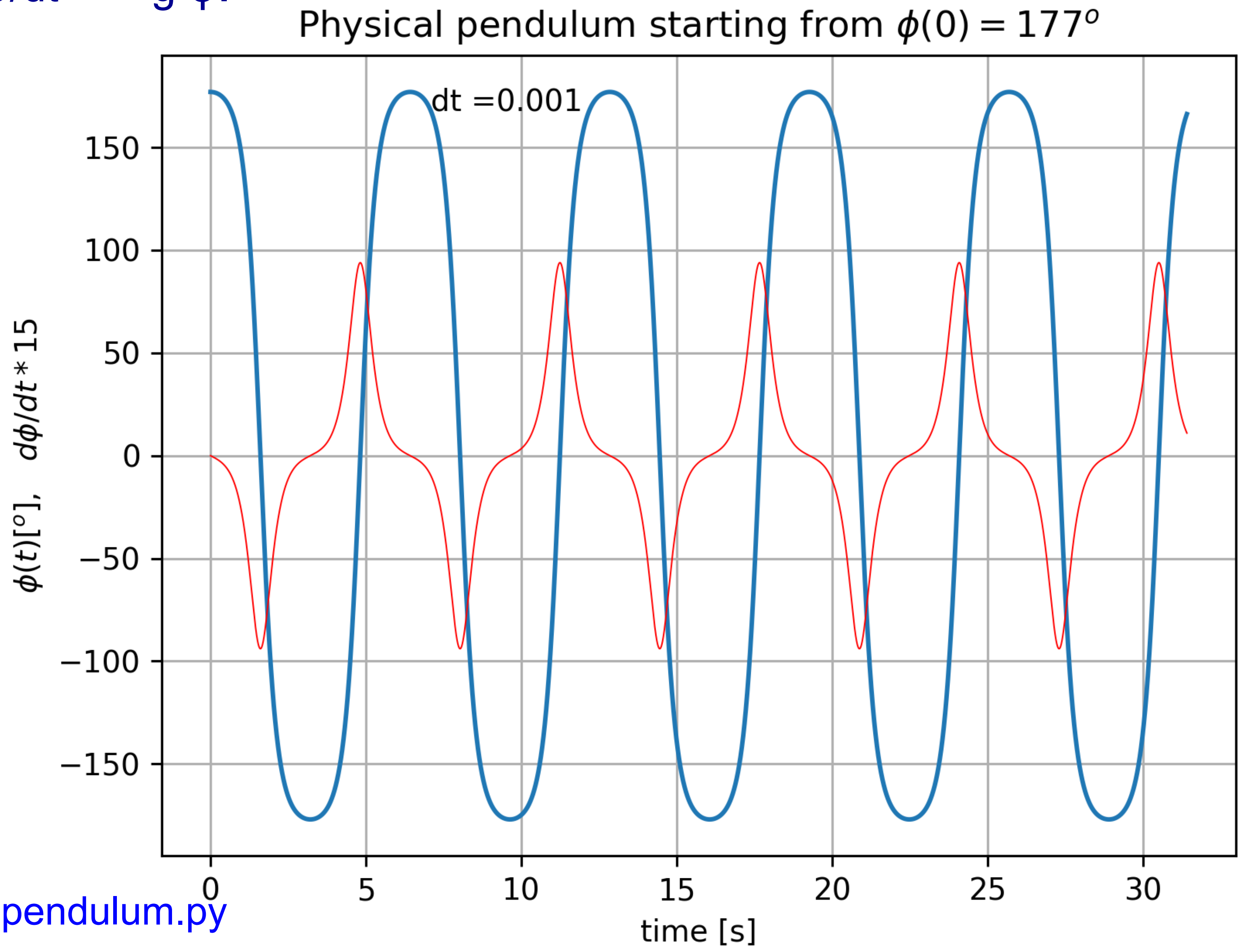
  Integrals of motion can either simplify the solution method (and thus be strictly enforced), or they can be used as a diagnostic tool to characterize the level of computational error in a general-method solution. This latter approach is illustrated in
  - phys-pendulum.py - 2nd order integration with small dt

structure of the time loop in phys. pendulum calculation

```
•    # Step through time, calculating the derivatives
•    # and updating variables. Leapfrog integration scheme,
•    # though it's a bit hard to recognize
•    q = 0.5
•    for i in range(num_steps+1):
•        t = t + dt              # update time
•        # save phi and vphi for a later plot
•        xs[i]  = phi/np.pi*180;   vs[i]=vphi;   ts[i]=t
•        # potential en. divided by mass, U = -g*cos(phi),
•        # so we have -dU/d(phi) = phi" = -g sin(phi)
•        # which is pendulum's equation of motion.
•        accphi = -g * sin(phi)  # acceleration (d^2{phi}/dt^2) of phi
•        vphi = vphi + dt*accphi*q # velocity update before position update
•        phi  =  phi + dt*vphi
•        q = 1
•        # error of energy (phi and vphi must be synchronized correctly)
•        if (i%skip == 0):      # no need to do this often, we can have skip = 20 (every 20. step)
•            phi_syn = phi -dt*vphi/2
•            U = g*(1-cos(phi_syn))  # potential energy/mass, zero at phi=0
•            v = vphi         # since r=1, v (linear speed) = r*dphi/dt = vphi
•            K = v*v/2        # K = kinetic energy/mass
•            E = U + K        # total energy/mass
•            if (i==0):      # save initial energy for comparison
•                E0 = E       # with later values of E
•            errs[i//skip] = (E-E0)/E0  # save value of relative error of energy
•    #
```

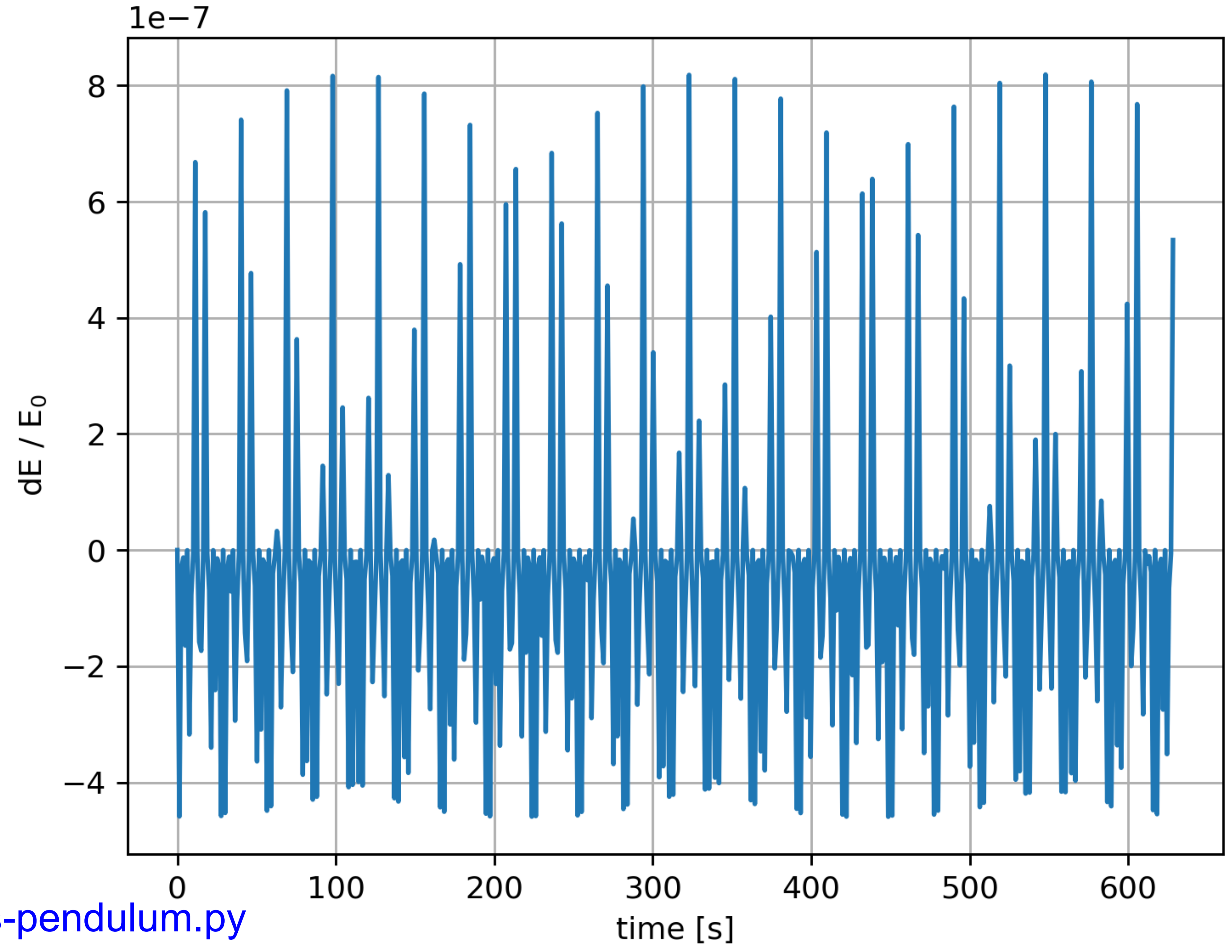
Physical pendulum, length 1 m, Leapfrog integrator. Notice significant deviations of position and speed from the cos(ωt) and -sin(ωt) time dependence valid for small oscillations of mathematical pendulum, governed by a simplified equation of motion

$d^2\varphi/dt^2 = - g\ \varphi.$



phys-pendulum.py

Physical pendulum, length 1 m, Leapfrog integrator. Notice that
(i) the level of error is < 10^{-6}, which is $O(10^{-6}) = O(dt^2)$; leapfrog is thus 2nd order method
(ii) there is no visible drift of the magnitude of error in t~10^3;
in fact there is no secular truncation error in this simplest of symplectic schemes.



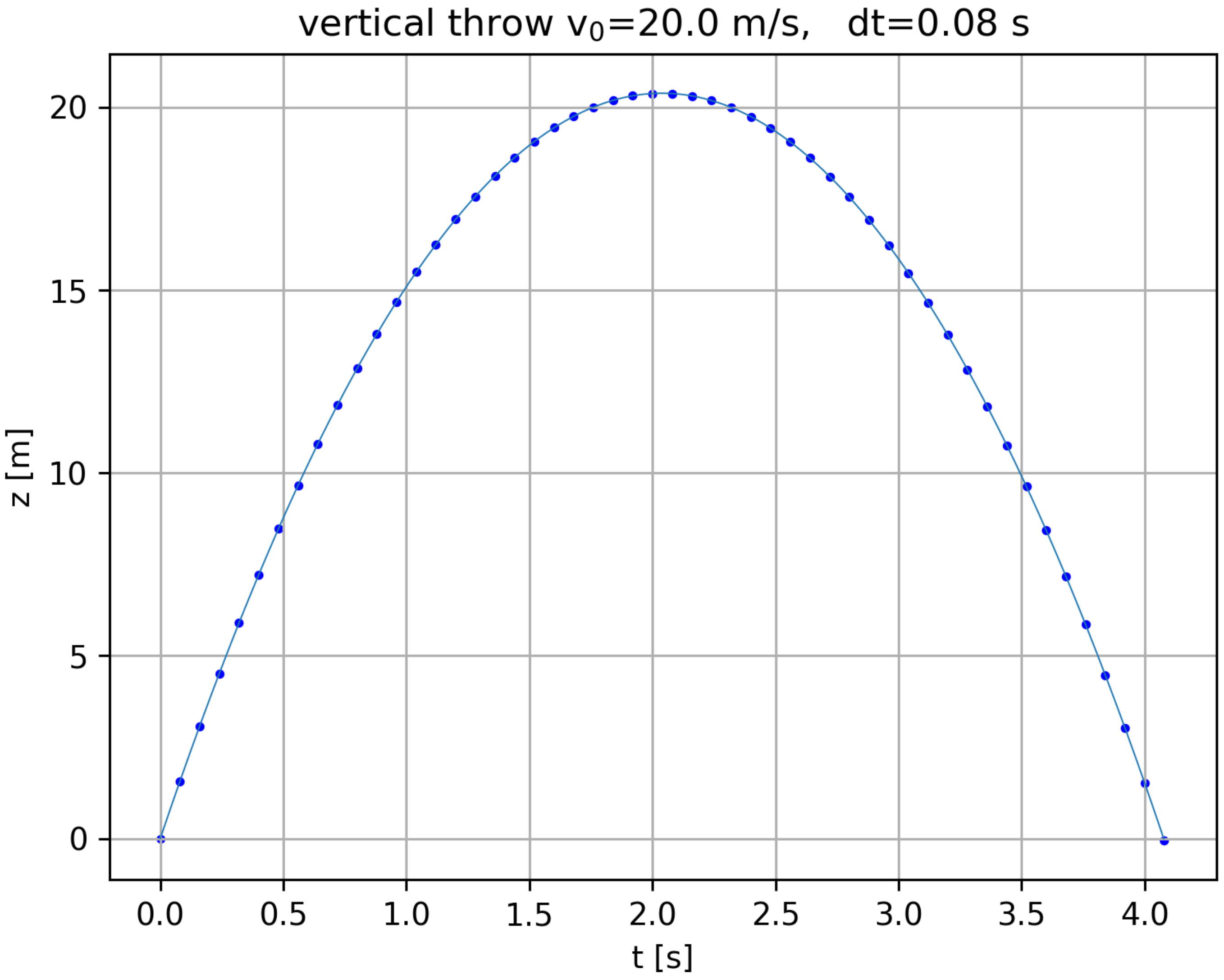
phys-pendulum.py

Applying ODE solvers: Trajectories without and with friction force

- Friction – motion in air with and w/o friction: diff1-throw-6.py
- Drag force formula $\mathbf{F_d} = -\mathbf{v}/|v|\ C_d(\rho\ |v|^2/2)\ A$ (it's a vector), the corresp. acceleration $d^2\mathbf{r}/dt^2 = -\mathbf{v}\ v\ C_d\ \rho/2\ A/M$ (vector),
- where for small balls in air at speeds < ~50 m/s we have the drag coefficient Cd = 0.47, $\rho = 1.25\ \mathrm{kg/m^3}$, and A/M = cross-section area to mass ratio of the object
- The boldfaced quantities are vectors in 3D, i.e. are composed of 3 components along x,y,z. In the code, you can try operating on length 3 vectors from numpy, or just define enough symbols like vx,vy,vz, in order to write all equations in component form. However, REMEMBER that integrations
- do NOT perform full time step to evolve vx and x first, then vy and y, and finally vz and z. Operate on all 3 components of vectors in consecutive lines of code (if written separately), or on vectors as such.

Vertical throw in vacuum – 2nd order 1D ODE

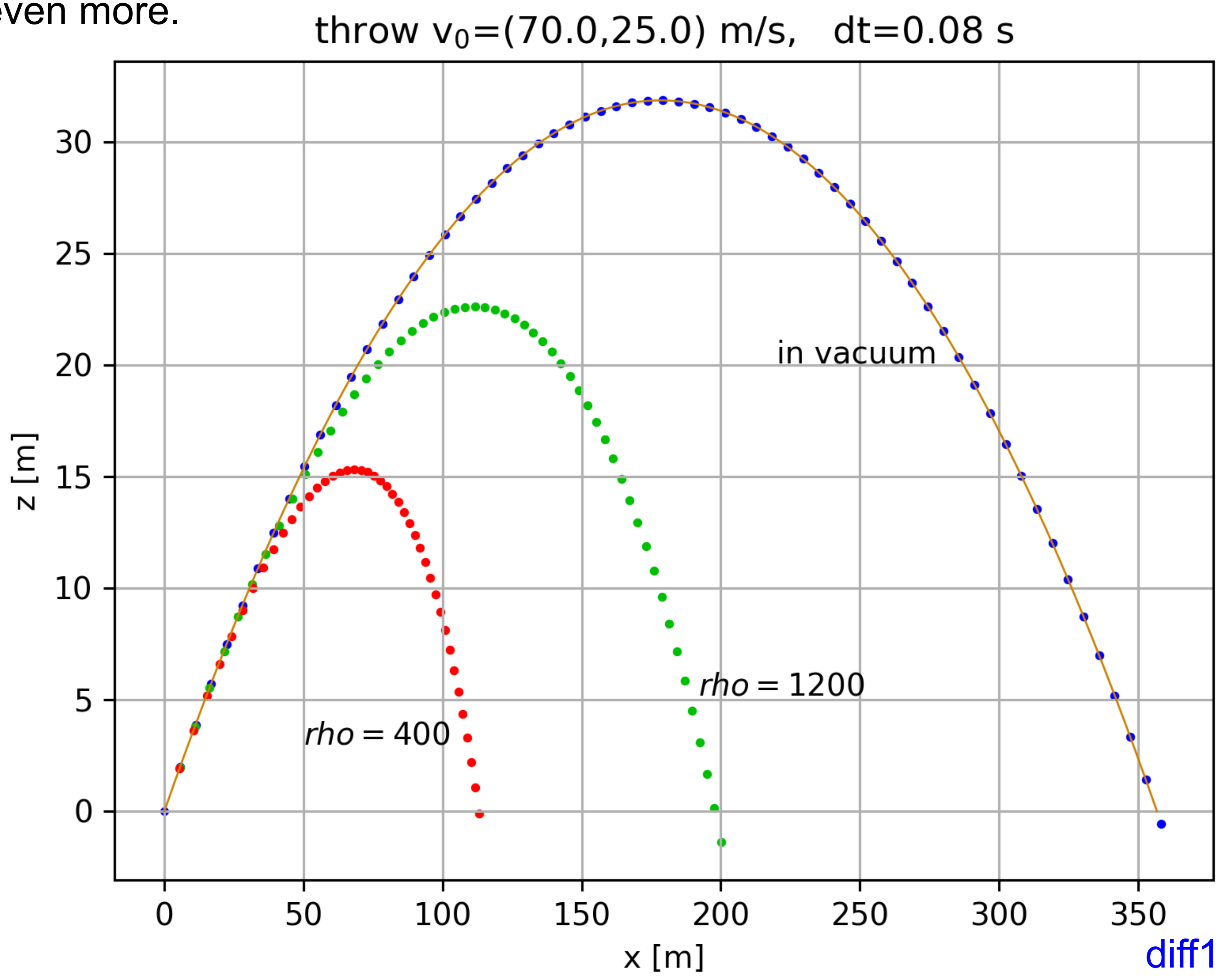
Trapezoidal rule. Equations so simple (no higher order derivatives that this scheme returns exactly *zero* errors even with large steps (all the steps taken by the program are shown as dots, line = theory)



diff1-throw-6.py

effects of air drag on motion

Red line is for a 10-cm diameter sphere of density 0.4*water density, the green one for 1.2*water density. The density of a ping-pong ball is even lower & it would be slowed down even more.

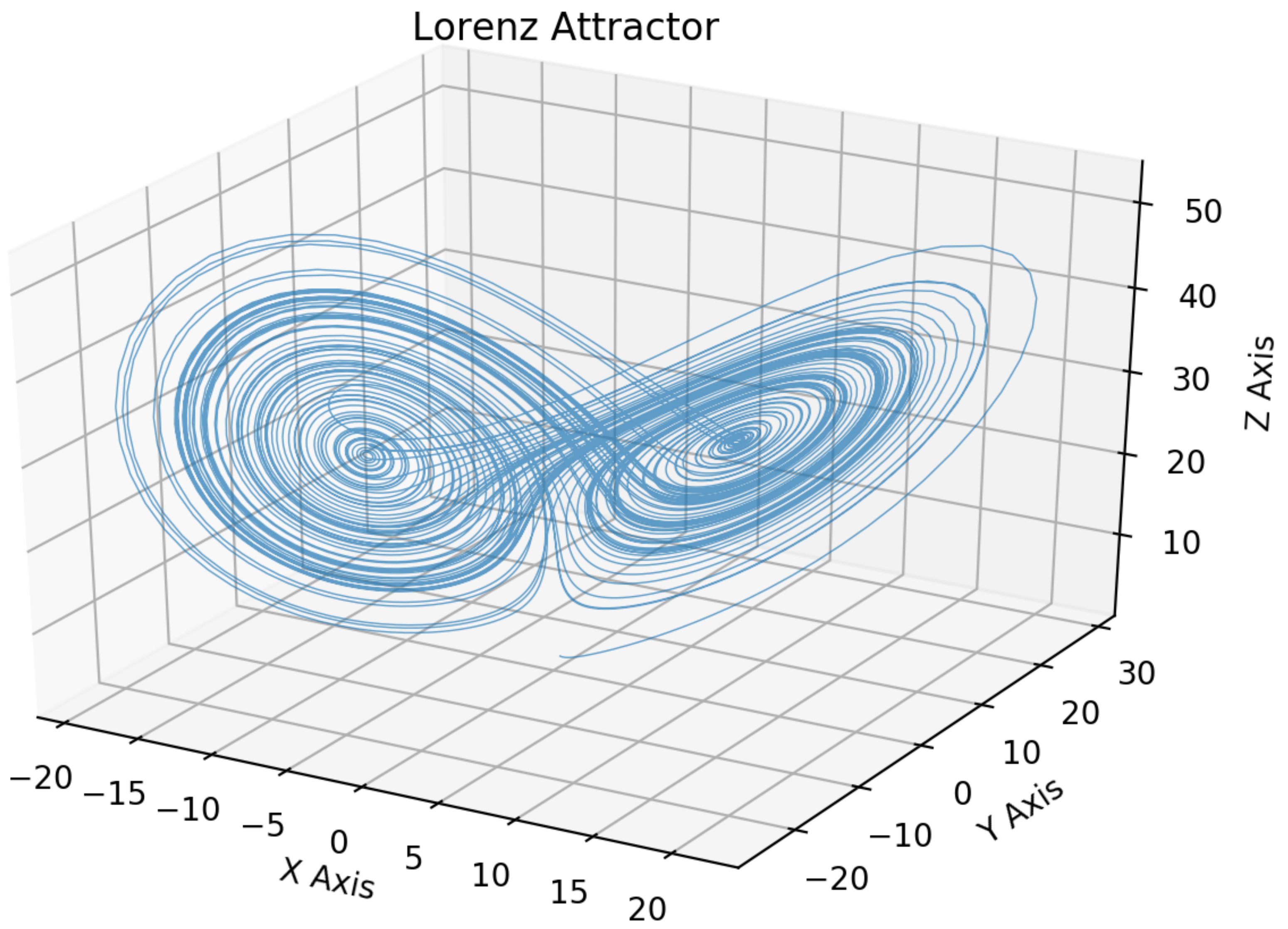


diff1-throw-6.py

Applying ODE solvers: Trajectories

- Nonlinear dynamical systems, chaos in n>2
  - Lorenz attractor lorenz-attractor.py
  - Chaos sometimes arises in dimensionality > 2 (more than 2 dependent variables, for instance 3 or more positions (coordinates) evolving under 1st-order ODEs, or 2 position component and two corresp. velocities, governed by the Newton's dynamics.
  - Chaos only happens in nonlinear systems
  - Chaotic solutions are by definition: non-periodic and extremely sensitive to the initial values of variables.

dX/dt, dY/dt, dZ/dt are known non-linear, coupled functions of X,Y, and Z.
There are a few parameters in the equations, the phase trajectory of the system is only looking like a butterfly with appropriate choice of these constants.



Chaotic solutions of simple, regular ODEs

- Lorenz attractor is derived from a simplified meteorological model. It has very few variables, only 3.
- The butterfly effect is present
- The solution is chaotic
- It is attracted to two points, and switches between their vicinity

- Orbits of Pluto and *all* other planets are chaotic too
- Starting with planets displaced by mere meters, after about 20 million years, there
- position on the Earth sky is unpredictable (which must be annoying to astrologers)
- conclusions on are the distant future of planetary systems are of statistical nature, just like the weather in Toronto exactly 1 month from now.

Applying ODE solvers: Trajectories

- Nonlinear dynamical systems, chaos in n>2
  - Chaos only happens in nonlinear systems.
  - There is another system with even simpler chaos:

  the motion of a ball on a massless Hooke's law spring, also subject to vertical gravity: chaotic_ball-2t.py

In this system, we use an easily identifiable, canonical variant of the leapfrog integrator method. It consists of kick and push parts, where kick is the update of velocity and push is the update of position, in this order. (Please learn this nice method well! It may be found on the final exam.)

Time step =

- half push
- evaluation of acceleration (at midpoint)
- full kick
- half push

```
dt = 0.000025
dt_2 = dt/2

# Step through time, calculating the radial acceleration of test mass
# given by formula:    f = -4(r-1), where 4 is the spring constant,
# and updating variables. Leapfrog integration scheme

for i in range(num_steps+1):
    xs[i], ys[i], zs[i] = (x,y,z)          # storage for plotting

    # half-push
    x = x + dt_2*vx
    y = y + dt_2*vy
    z = z + dt_2*vz

    r = np.sqrt(x*x + y*y + z*z)
    f_r = -4.*(r-1.)/r          # radial acceleration / r

    fx = x * f_r                # x/r etc. are the components of radial versor, f_r = f/r
    fy = y * f_r
    fz = z * f_r -1.            # -1 is downward gravity of unit strength
```

(cont'd on next page..)

```
    # full kick
    vx = vx + dt*fx
    vy = vy + dt*fy
    vz = vz + dt*fz

    # half-push
    x = x + dt_2*vx
    y = y + dt_2*vy
    z = z + dt_2*vz

# error of energy
    if (i%skip == 0):                       # E calculation only every skip-th time
        r = (x*x+y*y+z*z)**0.5              # distance from point of attraction
        Phi = 2.*(r-1.)**2 + z              # potential energy/mass
        E = Phi +(vx*vx+vy*vy+vz*vz)/2      # total energy/mass
        if (i==0):  E0 = E              # I actually like not to indent single instructions
        errs[i//skip] = E/E0 -1   # relative energy error

# end of time loop
```

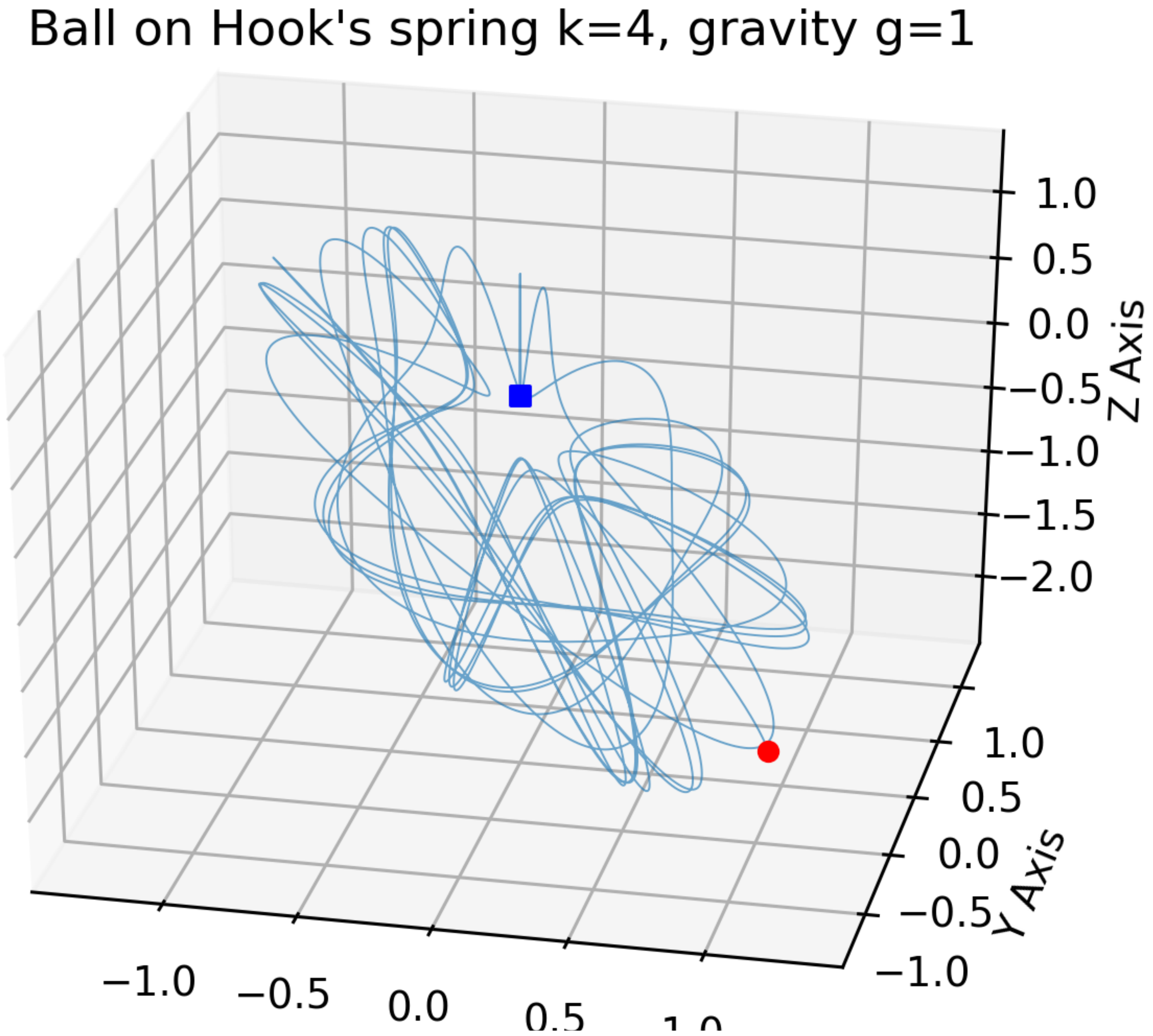
Applying ODE solvers: Trajectories

◆ Ball on a spring.
◆ The force law is linear in r, but the system is nonlinear. Why? Because variables x,y,z influence each other’s rate of change via $r = (x*x + y*y + z*z)^{1/2}$, i.e. nonlinearly.

chaotic_ball-2t.py

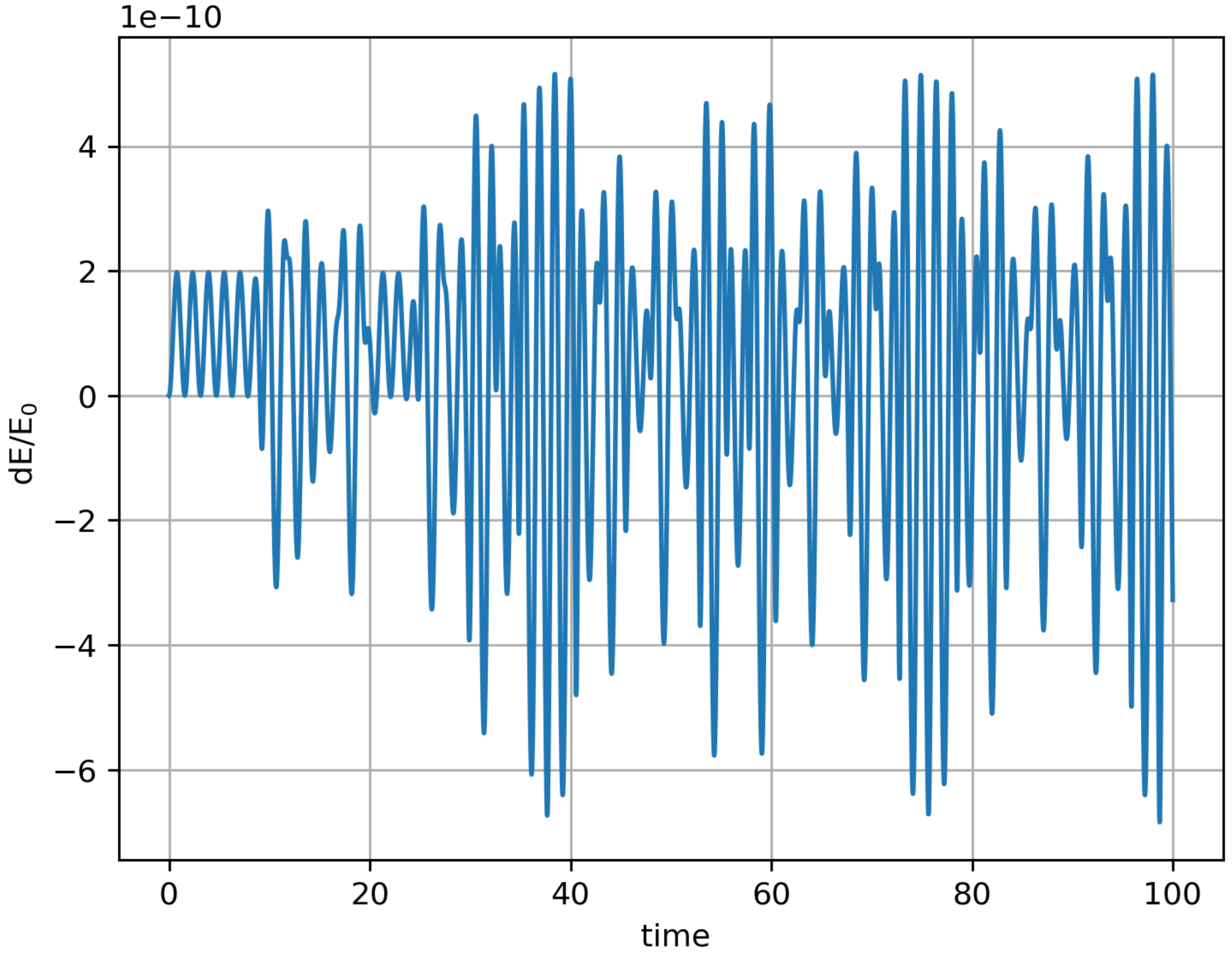
You can experiment yourself to see the sensitivity to initial conditions, by changing, say, the 7th digit of the initial position of a body in this program.

The final position at t=100 will be completely different!



Ball on Hook's spring k=4, gravity g=1
Z Axis
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Y Axis
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

Error of energy (with respect to initial energy) of a ball attached to a linear spring and subject to downward gravity. Leapfrog scheme with dt = 2.5e-5 keeps the error below dE/E < 5e-10 ~ dt^2 forever, although the positional error inexorably grows with time. But this is like error in timing, not in the generally more important amplitude of oscillations.



:

(next) Lecture 12 preview:

- **ODEs. Ordinary diff. eqs.:**
- N-body problems:
  2-Body problem, 3-Body problem, R3B (restricted 3B), circular R3B.
- Symplectic integrators are suitable for celestial mechanics

- **PDEs. Partial differential equations:**
- Wave equation in 2 dimensions
- Pond or swimming pool surface

- What is machine learning?