1 The 2-body problem: Orbit of a planet around the Sun

Consider the problem of a star of mass $M_* = M_\odot$ (where $M_\odot = 1.99 \times 10^{30}\,\mathrm{kg}$ surrounded by a planet of $M_p = 10^{-3} M_\odot$. At time $t = 0$ the planet is located at coordinates $(1, 0, 0)$ in units of AU$= 1.496 \times 10^{11}\,\mathrm{m}$. The planet's velocity is $(0, 0.5, 0)$ in units of the Kepler velocity at the that location, which is $v_K(1\ \mathrm{AU}) = 2.98 \times 10^4\,\mathrm{m\,s^{-1}}$

1.1 Solve the Kepler orbit of this planet using numerical integration with the leapfrog algorithm. Find an appropriate time step. Plot the result for the first few orbits.

The leapfrog algorithm reads:

$$v_{i+1/2} = v_i + a(x_i) \cdot \frac{\Delta t}{2} \tag{1}$$

$$x_{i+1/2} = x_i + v_{i+1/2} \cdot \frac{\Delta t}{2} \tag{2}$$

It can be implemented using the following iterative python function:

```
        # define helper functions
        norm = lambda x: np.sqrt(sum([i**2 for i in x]))
        a = lambda x: -G * M_star * x / norm(x)**3

        def leapfrog_integrate(x_0, v_0, steps):
            x, v = x_0, v_0

            # single integration step
            def leapfrog_step(x, v):
                new_v = v + a(x) * dt / 2
                new_x = x + new_v * dt / 2
                return new_x, new_v

            xs, vs = [], []
            # iterate, then return position and velocity
            for _ in range(steps):
                x, v = leapfrog_step(x, v)
                xs.append(x)
                vs.append(v)

            return xs, vs
```

After playing around a little bit with the parameters, it seems like a step-size of $\Delta t = 1 \times 10^{-2}$ leads to a good trade-off between accuracy and integration time. With this step-size, one orbit takes approximately 5350 integration steps.

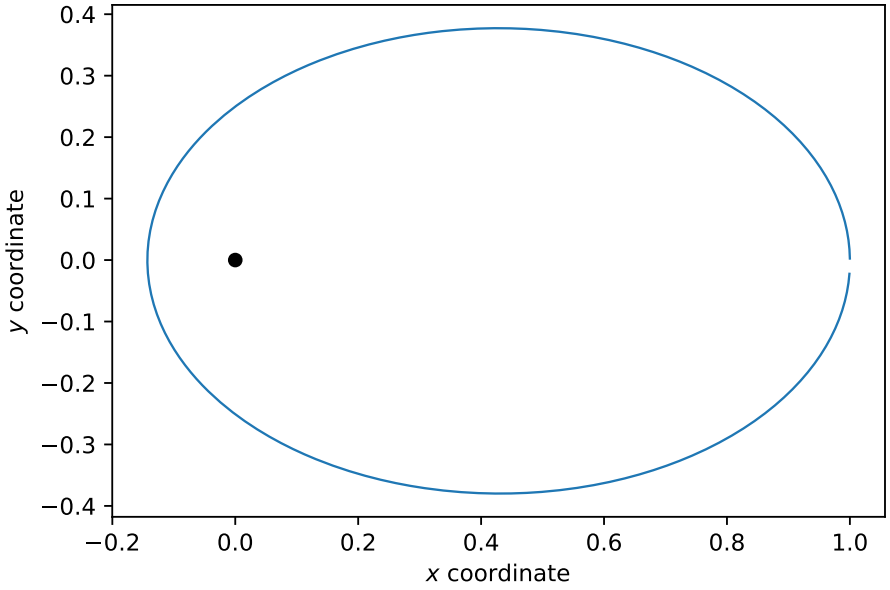


Figure 1: Trajectory of the planet after approximately one orbit ($N = 5350$, $dt = 10^{-2}$).

1.2 Now integrate for 100 orbits and see how the orbit behaves.

As can be seen below, the orbiting planet does not come back to exactly the same spot after each time it circles the star, which is due to the discrete nature of the simulation (e.g. finitely small Δt). The errors add up after each orbit, and the simulated planet starts precessing around the star.

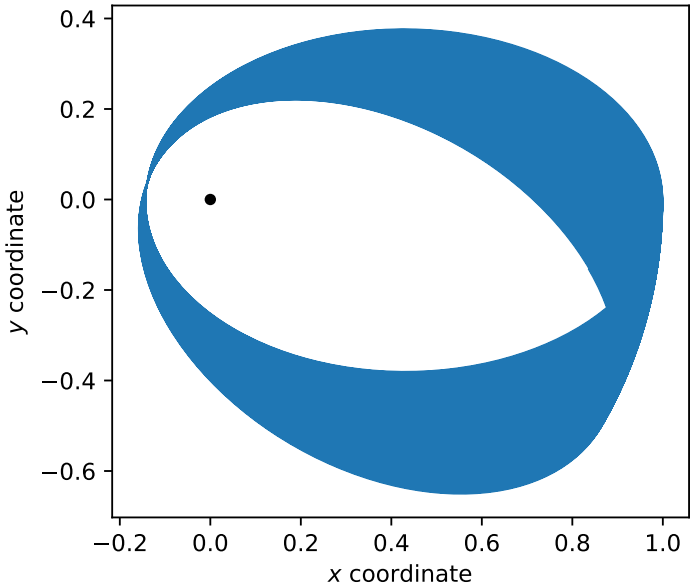


Figure 2: Trajectory of the planet after approximately 100 orbits ($N = 10^5$, $dt = 10^{-2}$).

1.3 Repeat this with the RK2 and RK4 algorithms and see how the system behaves. Discuss the difference to the leapfrog algorithm.

To implement the Runge-Kutta solver, we can use the following function:

```
        def rk_integrate(x_0, v_0, dt, steps, order=2):

            # define derivative function
            def f(t, y):
                x = np.array(y[:3])
                v = np.array(y[3:])

                dv = a(x) * dt
                dx = v * dt
                return list(dx) + list(dv)

            y_0 = list(x_0) + list(v_0)
            # initialize integrator (either Rk2 or Rk4)
            if order == 2:
                rk_integrator = RK23(
                    f, t_0, y0=y_0, t_bound=dt*steps, first_step=dt, max_step=dt
                )
            elif order == 4:
                rk_integrator = RK45(
                    f, t_0, y0=y_0, t_bound=dt*steps, first_step=dt, max_step=dt
                )

            # iterate and return
            ys = []
            for _ in range(steps):
                rk_integrator.step()
                ys.append(rk_integrator.y)

            return np.array(ys)
```

- energy is bound from above and below in Leapfrog, in RK2 energy growth every year
- angular momentum is strictly conserved in Leapfrog, not shown but proovable
- energy error grows linearilly (on average) in RK4, but much slower as RK2 and for a long times it's lower than the LF

As can be seen in the plots below, ...

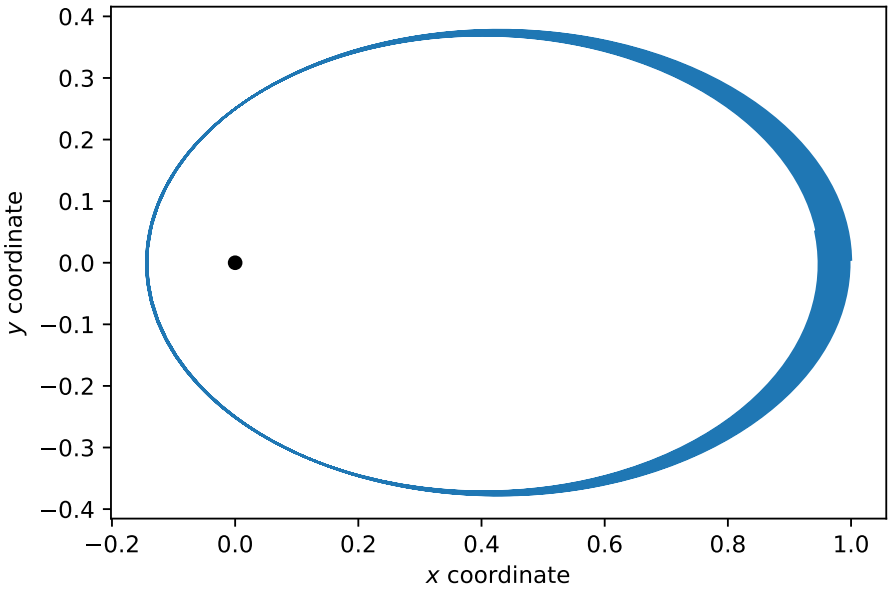


Figure 3: 2nd order Runge-Kutta algorithm

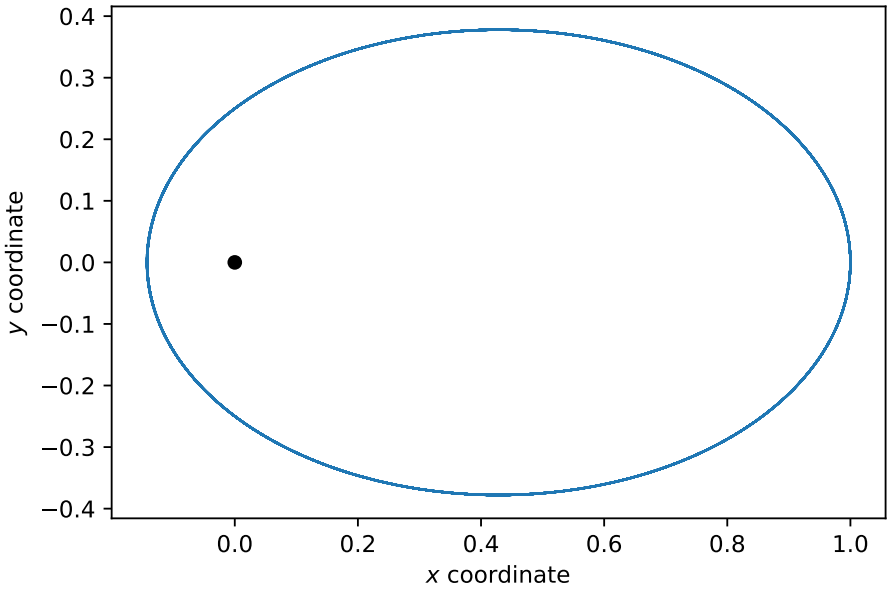


Figure 4: 4th order Runge-Kutta algorithm

1.4 Plot the time evolution of the relative error of the total energy and the time evolution of the total kinetic energy of the system.

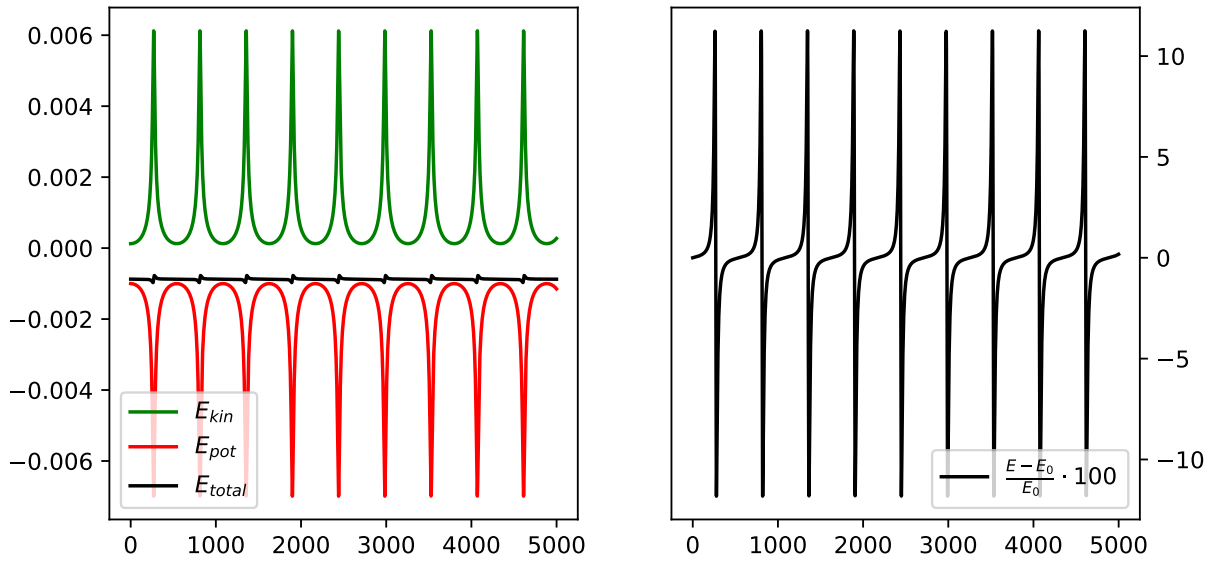


Figure 5: total energy balance as well as relative energy error plotted vs. time for leapfrog solver

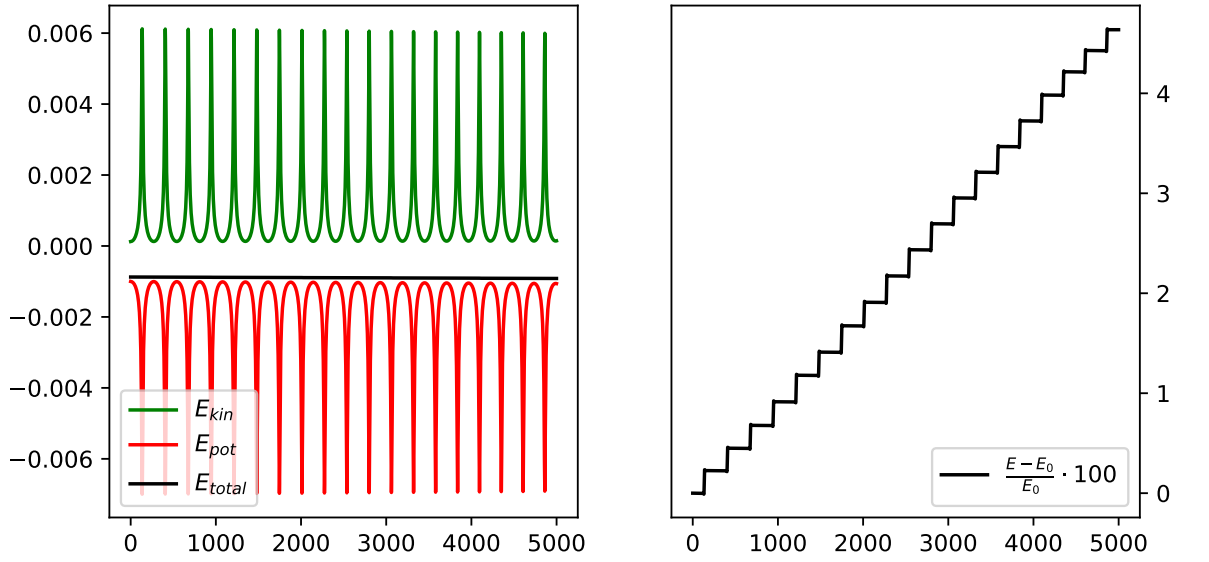


Figure 6: total energy balance as well as relative energy error plotted vs. time for second-order Runge-Kutta

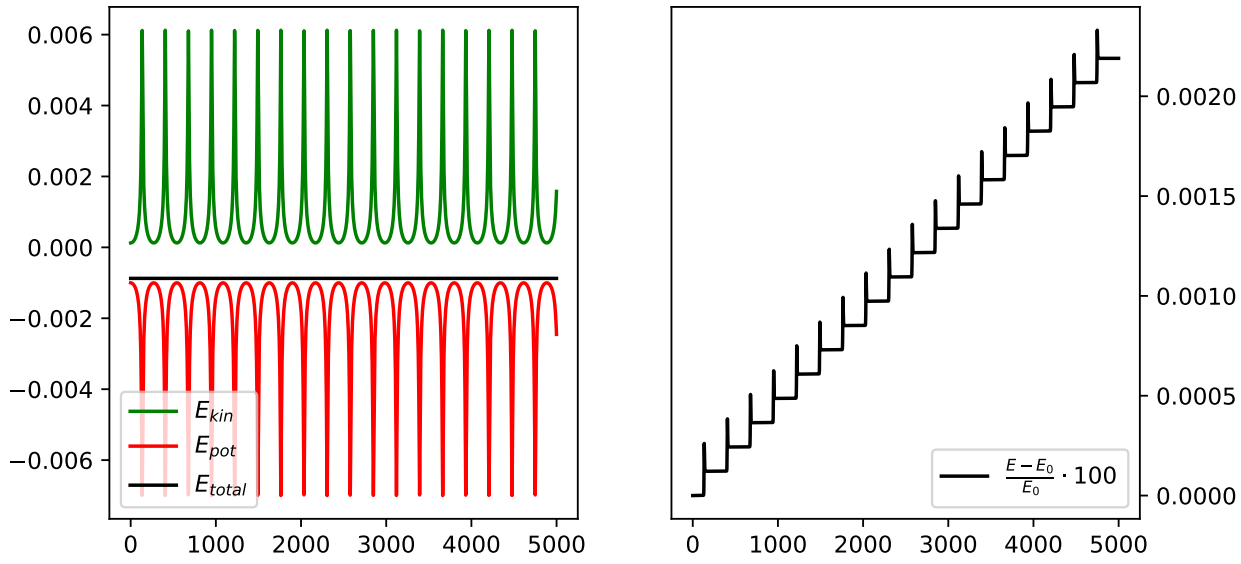


Figure 7: total energy balance as well as relative energy error plotted vs. time for fourth-order Runge-Kutta

2 N-body problem

Let us now consider a N-body problem, with $N > 2$ gravitationally interacting objects. We will solve this system with the *direct summation method*. First, however, let us cast the problem in dimensionless units. The N-body equation reads

$$\frac{d\vec{x}_i}{dt} = \vec{v}_i \tag{3}$$

$$m_i \frac{d\vec{v}_i}{dt} = G m_i \sum_{k \neq i} m_k \frac{\vec{x}_k - \vec{x}_i}{|\vec{x}_k - \vec{x}_i|^3} \tag{4}$$

2.1 Show that with an appropriate scaling $\vec{x} = \xi \vec{x}'$, $\vec{v} = \Phi \vec{v}'$, $t = \tau t'$ and $m = \mu m'$ (with ξ, Φ, τ and μ constants) the equations for N-body dynamics can be brought into dimensionless form, where the gravitational constant G is absorbed into the variables. Which condistions must ξ, Φ, τ and μ obey and how many of them can be freely chosen?

Dimensionless quantities must fulfill:

$$\frac{\mathrm{d}\vec{x_i}'}{\mathrm{d}t'} = \vec{v_i'} \tag{5}$$

$$m_i' \frac{\mathrm{d}\vec{v_i'}}{\mathrm{d}t'} = m_i' \sum_{k \neq i} m_k' \frac{\vec{x_k}' - \vec{x_i}'}{|\vec{x_k}' - \vec{x_i}'|^3} \tag{6}$$

From Equation 5 we can calculate

$$\frac{\mathrm{d}\vec{x_i}'}{\mathrm{d}t'} \overset{CR}{=} \frac{\tau}{\xi} \frac{\mathrm{d}\vec{x}}{\mathrm{d}t} \overset{!}{=} \vec{v_i}' = \frac{\vec{v_i}}{\phi} \tag{7}$$

$$\Rightarrow \phi = \frac{\xi}{\tau} \tag{8}$$

Since $[G] = \frac{\mathrm{m}^3}{\mathrm{kg\ s}^2}$, we can couple μ, τ and ξ. Together we have 4 unknowns and two constraints, therefore we can choose two paratemer freely. We choose: $\mu = M_*$ und $\xi = 1$ AU. It follows that $\tau = \sqrt{\frac{\xi^3}{G\mu}} \approx 5.02 \cdot 10^6 \mathrm{s} \approx 58.1 \mathrm{d}$ and $\phi = \sqrt{\frac{G\mu}{\xi}} \approx 3 \cdot 10^4 \frac{\mathrm{m}}{\mathrm{s}}$.

From now on we will do everything in these dimensionless units.

2.2 Write an N-body code for arbitrary N with the leapfrog integration and a constant time step.

One way to prevent numerical divergences in case of very close encounters is to use the *softening length parameter* ε and modify the gravitational force among massive particles:

$$m_i \frac{d\vec{v}_i}{dt} = G m_i \sum_{k \neq i} m_k \frac{\vec{x}_k - \vec{x}_i}{|\vec{x}_k - \vec{x}_i|^3 + \varepsilon^3} \tag{9}$$

However, ε should be small enough to keep the simulation physically consistent. We set:

$$\varepsilon = L N^{-1/3} \cdot \frac{1}{10000}, \tag{10}$$

where L is the typical size of the system in dimensionless units and N is the number of particles.

We can implement this directly when defining a class holding all of the information about a planet:

```
        class Planet():
            def __init__(self, r_0, v_0, m):
                self.r = r_0
                self.v = v_0
                self.m = m
                self.trajectory = []
                self.velocities = []

            def update_velocity(self, planets, dt):
                for p in planets:
                    if p == self:
                        continue
                    epsilon = 2 * len(planets)**(-1/3) / 10000
                    delta_r = p.r - self.r
                    a = G * p.m * delta_r / (norm(delta_r)**3 + epsilon**3)
                    self.v = np.add(self.v, a * dt)

                self.velocities.append(self.v)

            def update_position(self, dt):
                self.r = np.add(self.r, self.v * dt)
                self.trajectory.append(self.r)
```

As a simple test problem, solve the following sample binary star problem: two stars of mass 1 at initial locations $\vec{x}_1 = (-0.5, 0, 0)$, $\vec{x}_2 = (+0.5, 0, 0)$ and initial velocities $\vec{v}_1 = (0, -0.5, 0)$, $\vec{v}_2 = (0, +0.5, 0)$. Choose an appropriate time step! Plot the resulting trajectories of both starts in the (x, y)-plane (projection). Plot also the time evolution of the relative error of the total energy of the system.

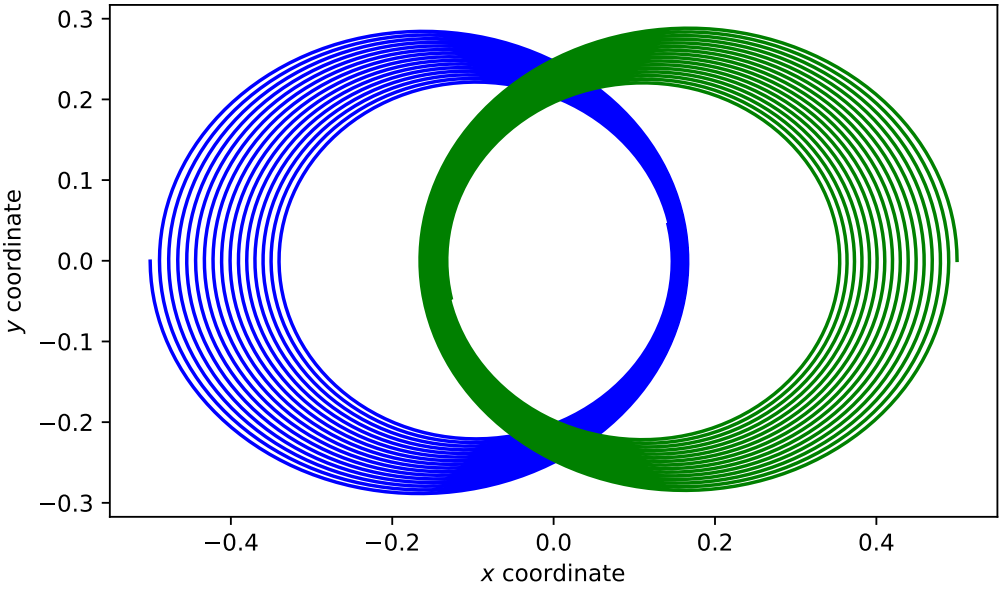


Figure 8

Now add a third star with mass 0.1 and initial position $\vec{x}_3 = (1, 6, 2)$ and initial velocity $\vec{v}_3 = (0, 0, 0)$. Show how this third star "falls into" the binary, interacts with it, and gets eventually ejected. Plot the trajectories of all three stars in the (x, y)-plane (projection). Again, plot the time evolution of the relative error of the total energy of the system.

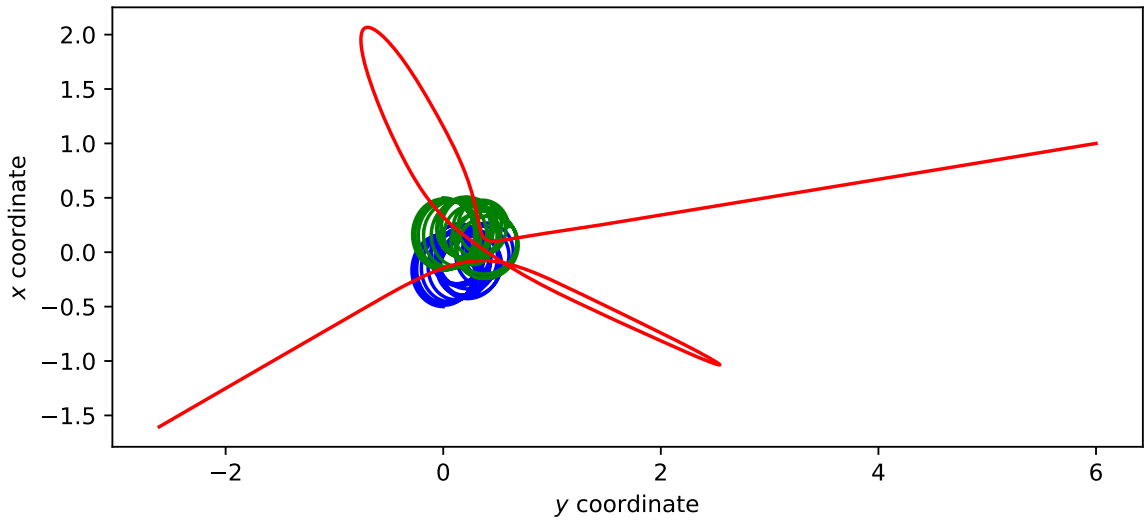


Figure 9

Play with the time step by varying it at least a factor of 10 (but keep the final time of the integration fixed) and describe whether the results change or not. The effect of changing the time step can also be stronger than in this case, to see this also try $\vec{x}_3 = (1, 6, 3)$

- overall behavior is similar: star falls in, gets "thrown around", then ejected
- details are different, e.g. direction of ejection

2.3 Real N-body problem ($N >> 3$)

Set up a spherical cloud of 30 randomly positioned stars of mass 1. The cloud radius is 1. Use a uniform random number generator for this. An easy way is to choose randomly (x, y, z) between -1 and $+1$, and reject (and redo) stars that have $1 < \sqrt{x^2 + y^2 + z^2}$. Give each particle a random velocity (uniform) with a maximum of 0.1. Use the same rejection trick as for the positions. Evolve the system over an appropriate time-scale (is there a characteristic time-scale for gravitationally interacting cluster?).

A system of 30 randomly-distributed planets (in a sphere) can be created using the following function:

```
def create_30_body_system():
    planets = []
    for _ in range(30):
        valid_r, valid_v = False, False
        while not valid_r:
            x = uniform(-1, 1)
            y = uniform(-1, 1)
            z = uniform(-1, 1)
            if np.sqrt(x**2 + y**2 + z**2) <= 1:
                valid_r = True
        while not valid_v:
            vx = uniform(-.1, .1)
            vy = uniform(-.1, .1)
            vz = uniform(-.1, .1)
            if np.sqrt(vx**2 + vy**2 + vz**2) <= .1:
                valid_v = True

        r, v, m = np.array([x, y, z]), np.array([vx, vy, vz]), 1
        p = Planet(r, v, m)

        planets.append(p)

    return planets
```

For the following simulations, we will have to use an adaptive time-step: at the beginning of each iteration, we estimate the value of the time-step as

$$\Delta t = C \cdot \frac{d_{min}}{|\vec{v}_{max}|}, \tag{11}$$

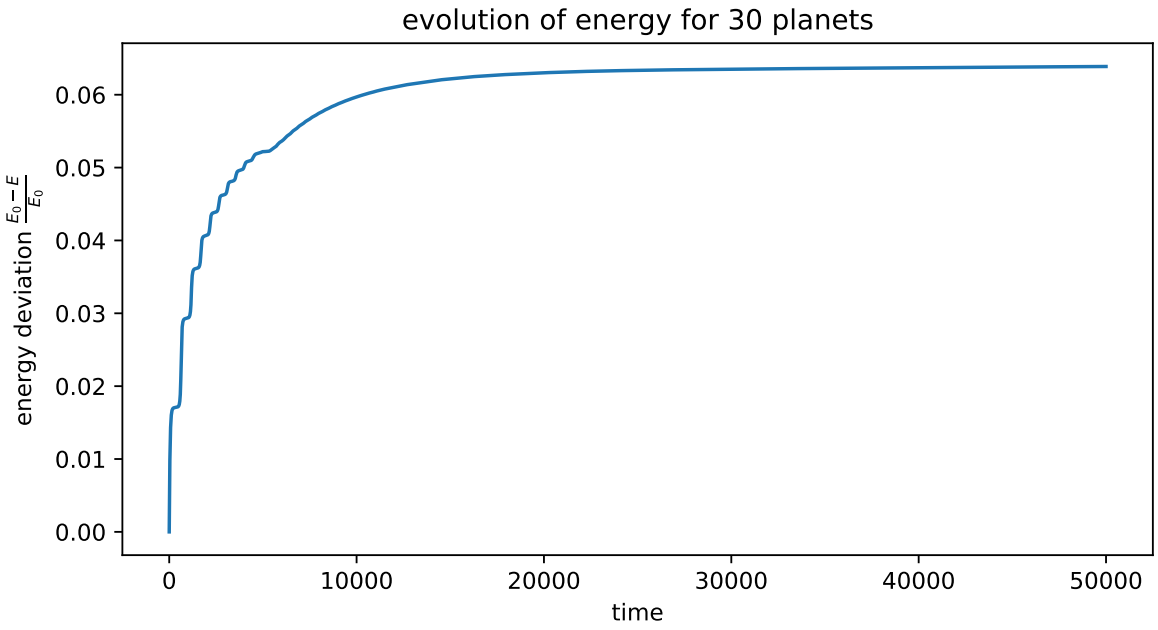
where $C = 0.01$. d_{min} is the minimum distance between 2 particles and $|\vec{v}_{max}|$ is the velocity of the fastest particle in the system. If the evolution is unstable or if the relative error of the total energy gets higher than 0.05, decrease the value of C (note that in this case you will need more iteration steps to evolve the system to the same time scale).

To implement this, at each time step we run the following function:

```
def calculate_adaptive_dt(planets):
    ds, vs = [], []
    for p in planets:
        for q in planets:
            if p is q:
                continue
            ds.append(norm(p.r - q.r))
        vs.append(norm(p.v))
    d_min = min(ds)
    v_max = max(vs)
    return 0.01 * d_min / v_max
```

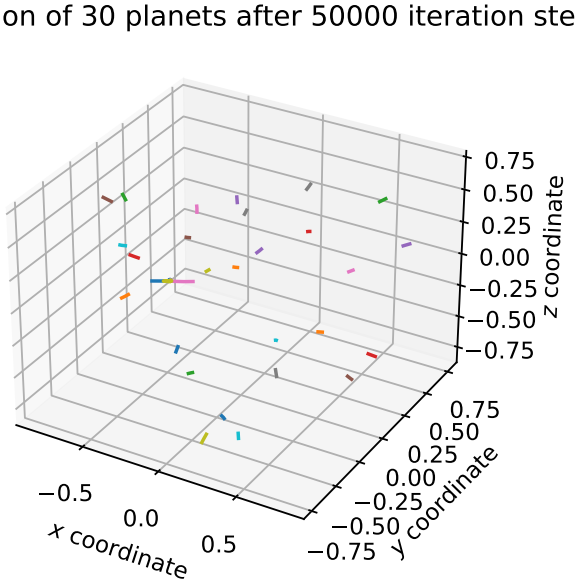
Plot the time evolution of the relative error of the total energy.

...



Plot the evolution in 3D and note any interesting patterns that emerge from the evolution. What are these? Why do they appear?

...



Now redo the simulation with $N = 60$ and masses $m = 0.5$. Run this and the previous simulation for 100 iterations each. Compare both the results and run times between the two.

...