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 $v_K(1\ \mathrm{AU}) = 2.98 \times 10^4\,\mathrm{m\,s^{-1}}$

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.

Leapfrog algoritm:

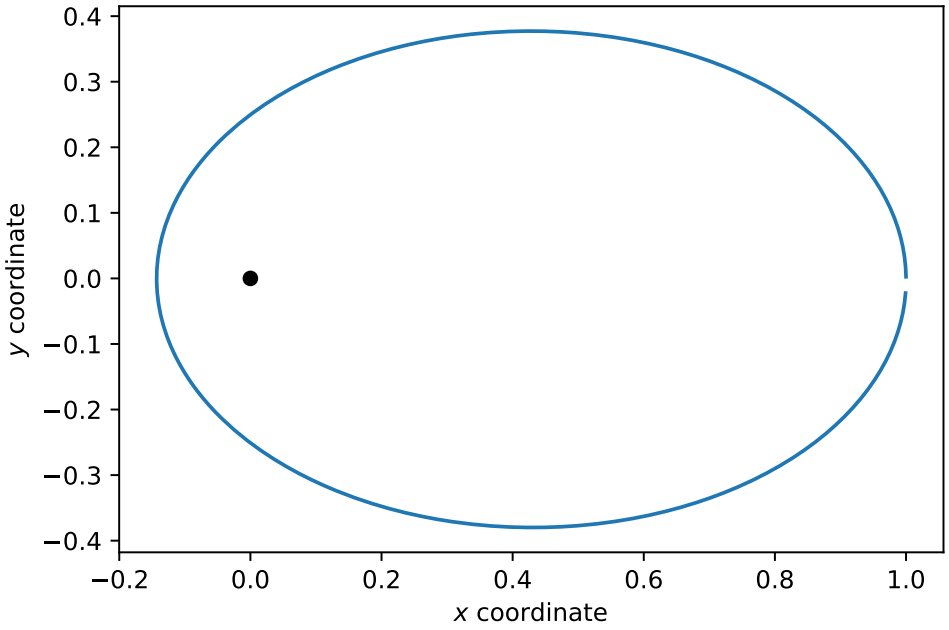
$$v_{i+1/2} = v_i + a(x_i) \cdot \frac{\Delta t}{2} \tag{1}$$

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

This can be implemented using the following iterative python function:

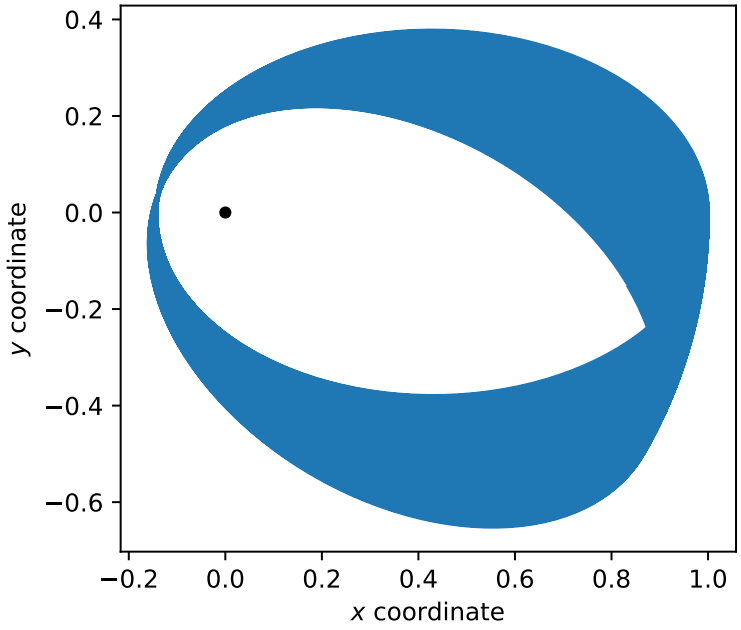
```
        norm = lambda x: np.sqrt(sum([i**2 for i in x]))
        a = lambda x: -G * M_star * x / norm(x)**3

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

            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 = [], []
            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 $\Delta t = 1 \times 10^{-3}$ leads to a good trade-off between accuracy and integration time. With this step-size, one orbit takes approximately 5350 integration steps.



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

As can be seen below, the system is relatively stable even after 100 orbits.



Repeat this with the RK2 and RK4 algorithms and see how the system behaves. Discuss the difference to the leapfrog algorithm. For this, also plot the time evolution of the relative error of the total energy and the time evolution of the total kinetic energy of the system.

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)
```

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

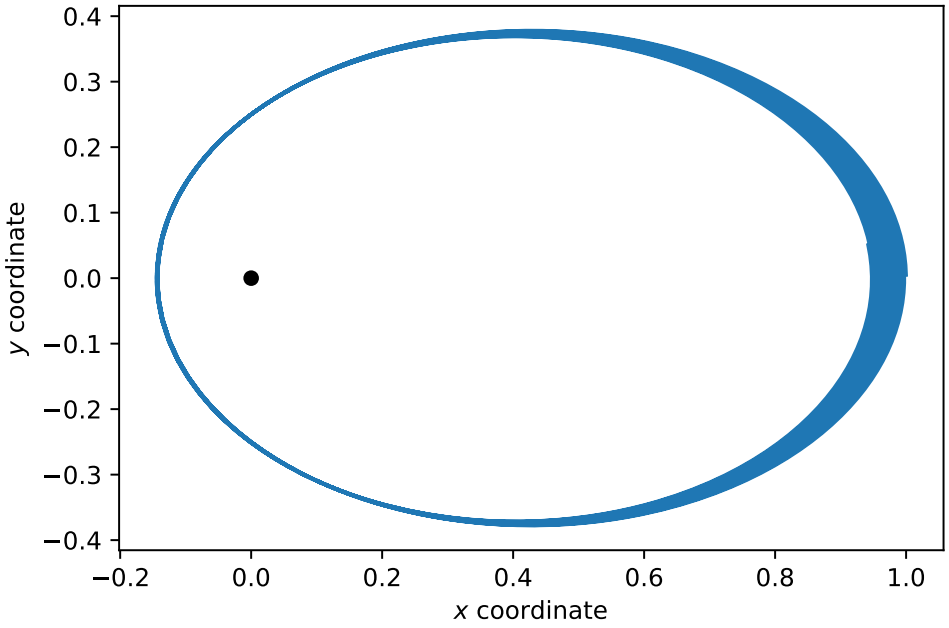


Figure 1: 2nd order Runge-Kutta algorithm

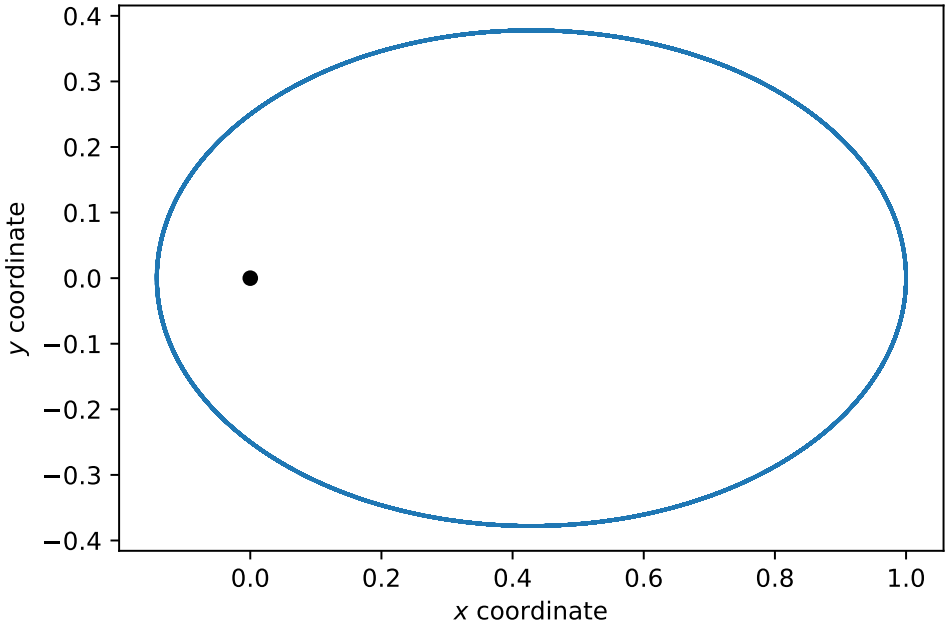


Figure 2: 4th order Runge-Kutta algorithm

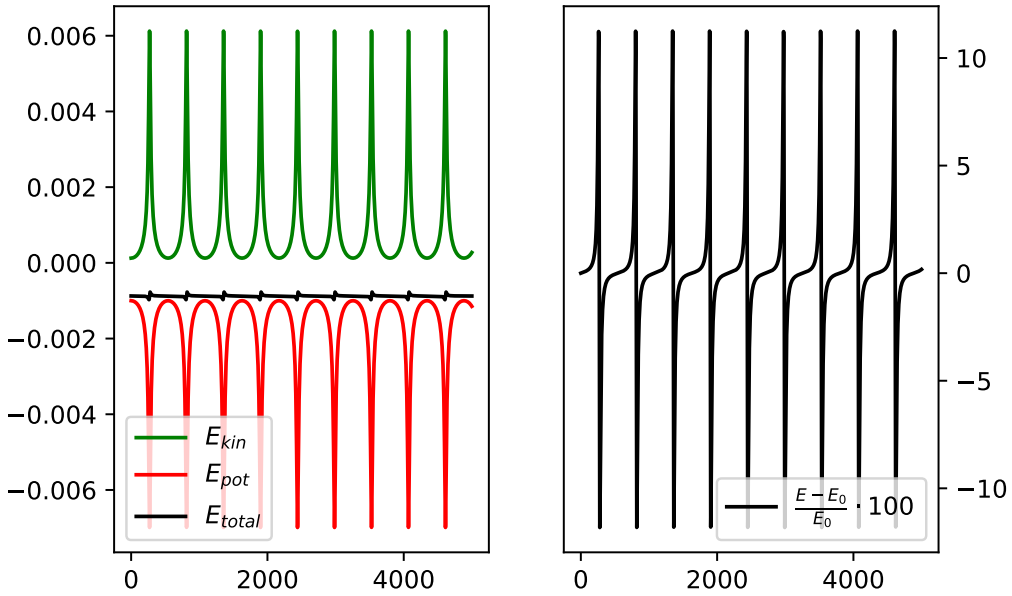


Figure 3

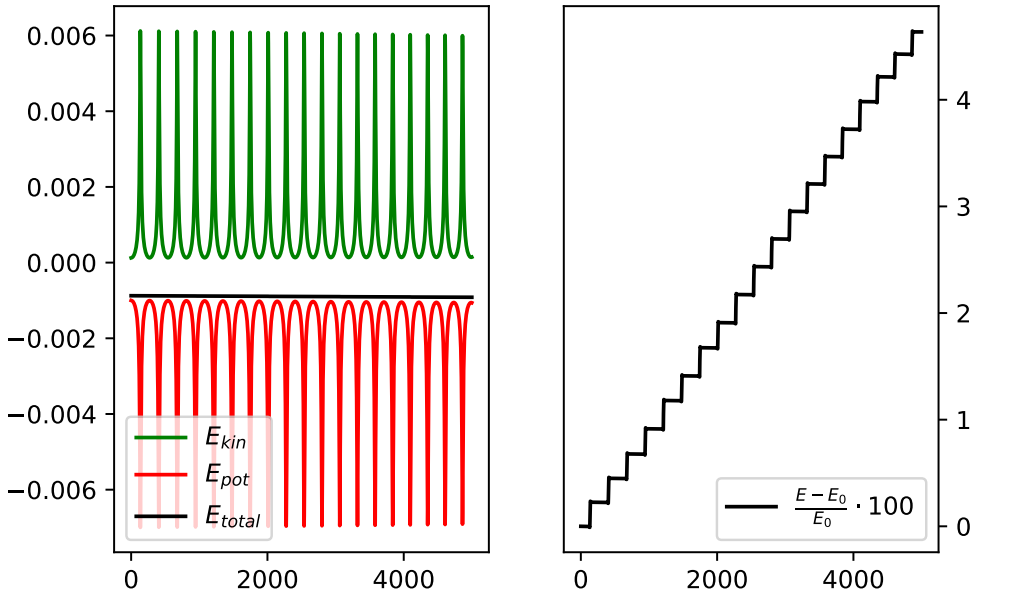


Figure 4

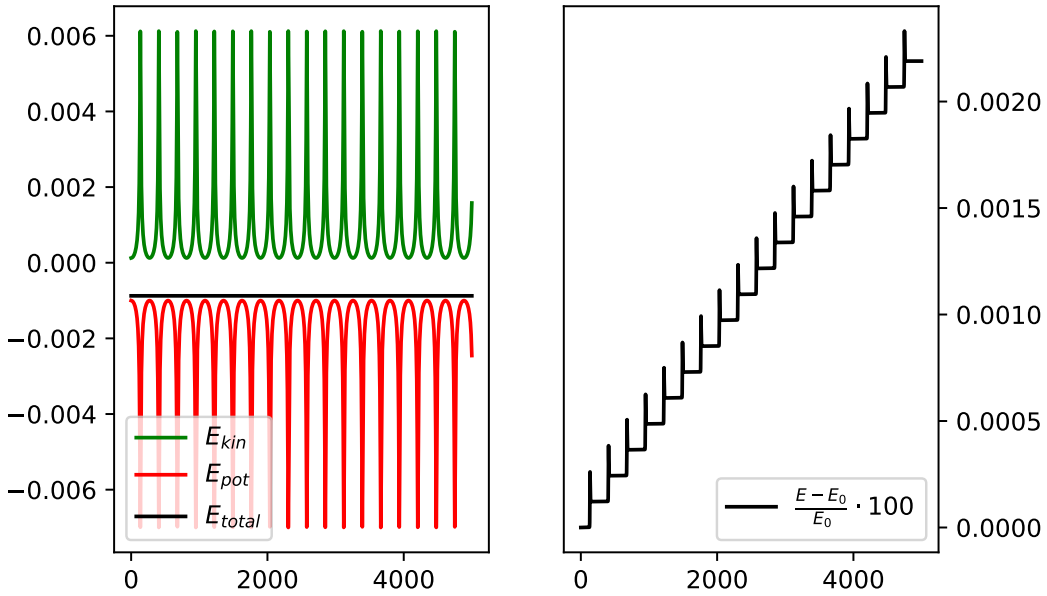


Figure 5

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

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

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