## Note 1

- Consider a spring of constant $k$ with one end attached to the origin and the other end to a particle of mass $m$ immersed in a fluid.

- Here the friction constant between the particle and the fluid is $\zeta$, but we assume that the spring is friction free.

- Let $R(t)$ be the temporal position of the particle at time $t$, and $V(t)$ its velocity.

- There are two forces acting on the particle; the friction force exerted by the fluid and the restoring force of the spring.

- The friction force is assumed to be proportional to the velocity of the particle and points in the opposite direction.

- The spring force is assumed to be proportional to the distance from the origin and points towards the origin.

## Note 2

- Then, the following two differential equations for the time evolution of the position and the velocity of the particle, must be simultaneously solved.

- Eq. (1) is simply the definition of the velocity, and Eq. (2) represents the force balance, which is known as Newton’s 2nd law of motion.

## Note 3

- We here use the Euler method to numerically solve the ordinary differential equations (1) and (2).

- For a small time increment $\Delta t$, the time integrals from $t\_i$ to $t\_i+\Delta t$ can be approximated to first order as shown in Eqs. (3) and (4).

## Note 4

- Before running our simulation, let us import the necessary libraries

## Note 5

- First, we define the dimensionality of our system, and the number of steps we want to take.

- Then, we define R and V as two-dimensional vectors, which will contain the instantaneous position and velocity of our particle.

- To visualize the trajectory, we also create arrays Rs and Vs, which will contain the positions and velocities at all times.

- Finally, we also define arrays to store the energy and the time values for the whole trajectory.

## Note 6

- To create animations using “matplotlib”, we will use the "FuncAnimation" procedure of the animation module.

- This module requires that we define two additional procedures, "init" and "animate".

- "init" is just an initialization procedure that defines all the graphical elements that will later be used.

- "animate" is the main procedure, it will be called everytime we want to update the graph, thus, it will be responsible for updating our solution by performing the time integration.

## Note 7

- Here, we visualize the time evolution of the particle trajectory.

- The black point represents the position of the particle at the given time.

- In addition, the entire trajectory of the particle is given by updating the red line at each step

- Notice that the particle traces what appears to be a closed orbit.

### Note 8

- Let us look in more detail at what is happening to the particle.

- For this, we have plotted the x and y positions, as well as the total energy, as a function of time.

- As expected, the x and y coordinates show oscillatory motion.

- What is probably more surprising, are the oscillations in the total energy.

- In this case, we have turned off friction, so ideally the energy of our system should be conserved.

- The oscillations in the energy are an artifact of the approximate solution scheme we have used.

- To reduce these fluctuations, we can use a smaller time step, or even adopt a higher order integrator, for example the Runge-Kutta 4th order method.

- However, even though the energy is not strictly conserved, notice that there is no notable systematic drift.

- The energy oscillates around a fixed value.

- For physical simulations, this is a crucial aspect of a good integrator.

### Note 9

- Ideally, the solutions to this harmonic problem in the absence of friction are closed orbits.

- To check if this is the case, we make a parametric plot of the x, y positions of the particle.

- Indeed, our orbit appears to be a closed one, even though a more careful examination will show a slightly spiraling trajectory.