Contents

Check the complete version of the Python code used here in bundled iPython notebook: `final_2p_he.ipynb` for double pendulum and `final_g_he.ipynb` for gravitational orbits.

1. Double Pendulum

In this section, we solve the equation of a double pendulum, plot the angles deviating from equilibrium as functions of time, and give a discussion of the initial conditions, small-angle approximations, and chaotic phenomena.

A double pendulum consists of a mass m_1 suspended by a massless rod of length L_1 from a fixed point pivot and another mass m_2 suspended by a massless rod of length L_2 from m_1. See Figure 1. The angles deviating from the equilibrium are θ_1 of m_1 and θ_2 for m_2. So, the potential energy of the system is

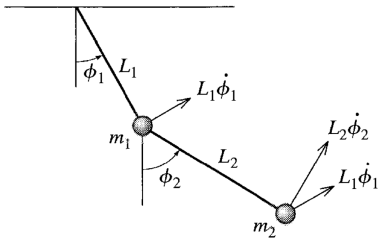


FIG. 1 A double pendulum without damping or driven force. Figure 11.9 of [1].

$$\begin{aligned} U &= m_1 g L_1(1-\cos\phi_1) + m_2 g(L_1(1-\cos\phi_1) + L_2(1-\cos\phi_2)) \\ &= (m_1+m_2)gL_1(1-\cos\phi_1) + m_2 g L_2(1-\cos\phi_2) \end{aligned}$$

the velocity of m_1 is $L_1\dot{\phi}_1$ in the tangential direction, while the velocity of m_2 is the vector sum of its velocity $L_2\dot{\phi}_2$ relatively to m_1 and the velocity of m_1 relative to fixed point pivot $L_1\dot{\phi}_1$. Since by geometry we know the angle between them is $\phi_2-\phi_1$, we have $v_2^2 = L_1^2\dot{\phi}_1^2 + 2L_1L_2\dot{\phi}_1\dot{\phi}_2\cos(\phi_1-\phi_2) + L_2^2\dot{\phi}_2^2$. So, the kinetic energy of the system is

$$\begin{aligned} T &= \frac{1}{2}m_1L_1^2\dot{\phi}_1^2 + \frac{1}{2}m_2\left(L_1^2\dot{\phi}_1^2 + 2L_1L_2\dot{\phi}_1\dot{\phi}_2\cos(\phi_1-\phi_2) + L_2^2\dot{\phi}_2^2\right) \\ &= \frac{1}{2}(m_1+m_2)L_1^2\dot{\phi}_1^2 + m_2L_1L_2\dot{\phi}_1\dot{\phi}_2\cos(\phi_1-\phi_2) + \frac{1}{2}m_2L_2^2\dot{\phi}_2^2 \end{aligned}$$

Now we can find the Lagrangian $\mathcal{L} = T - U$ of the system

$$\mathcal{L} = \frac{1}{2}(m_1+m_2)L_1^2\dot{\phi}_1^2 + m_2L_1L_2\dot{\phi}_1\dot{\phi}_2\cos(\phi_1-\phi_2) + \frac{1}{2}m_2L_2^2\dot{\phi}_2^2 - (m_1+m_2)gL_1(1-\cos\phi_1) - m_2gL_2(1-\cos\phi_2)$$

and the Euler-Lagrangian equations are

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial\mathcal{L}}{\partial\dot{\phi}_1} = \frac{\partial\mathcal{L}}{\partial\phi_1} \quad &\Rightarrow \quad (m_1+m_2)L_1\ddot{\phi}_1 + m_2L_2\ddot{\phi}_2\cos(\phi_1-\phi_2) = -m_2L_2\dot{\phi}_2^2\sin(\phi_1-\phi_2) - (m_1+m_2)g\sin\phi_1 \\ \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial\mathcal{L}}{\partial\dot{\phi}_2} = \frac{\partial\mathcal{L}}{\partial\phi_2} \quad &\Rightarrow \quad L_1\ddot{\phi}_1\cos(\phi_1-\phi_2) + L_2\ddot{\phi}_2 = L_1\dot{\phi}_1^2\sin(\phi_1-\phi_2) - g\sin\phi_2 \end{aligned}$$

the matrix form of them is

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (m_1+m_2)L_1 & m_2L_2\cos(\phi_1-\phi_2) \\ 0 & 0 & L_1\cos(\phi_1-\phi_2) & L_2 \end{pmatrix}}_{M} \begin{pmatrix} \dot{\phi}_1 \\ \phi_2 \\ \dot{\phi}_1 \\ \dot{\phi}_2 \end{pmatrix}^{\bullet} = \underbrace{\begin{pmatrix} \dot{\phi}_1 \\ \dot{\phi}_2 \\ -m_2L_2\dot{\phi}_2^2\sin(\phi_1-\phi_2) - (m_1+m_2)g\sin\phi_1 \\ L_1\dot{\phi}_1^2\sin(\phi_1-\phi_2) - g\sin\phi_2 \end{pmatrix}}_{N}$$

Clearly, the equations of motion depend on five parameters m_1, m_2, L_1, L_2, and g. So for coding, in a Python class, we need

```
    def __init__(self, m1=1.0, m2=1.0, L1=1.0, L2=1.0, g=1.0):
        """
        This function set the internal values of the class
        Parameters
           m1 : float, mass 1
           m2 : float, mass 2
           L1 : float, length of massless rod connecting pivot and m1
           L2 : float, length of massless rod connecting pivot and m2
           g  : float, gravitational acceleration at the earth's surface
        """
        self.m1 = m1
        self.m2 = m2
        self.L1 = L1
        self.L2 = L2
        self.g = g
```

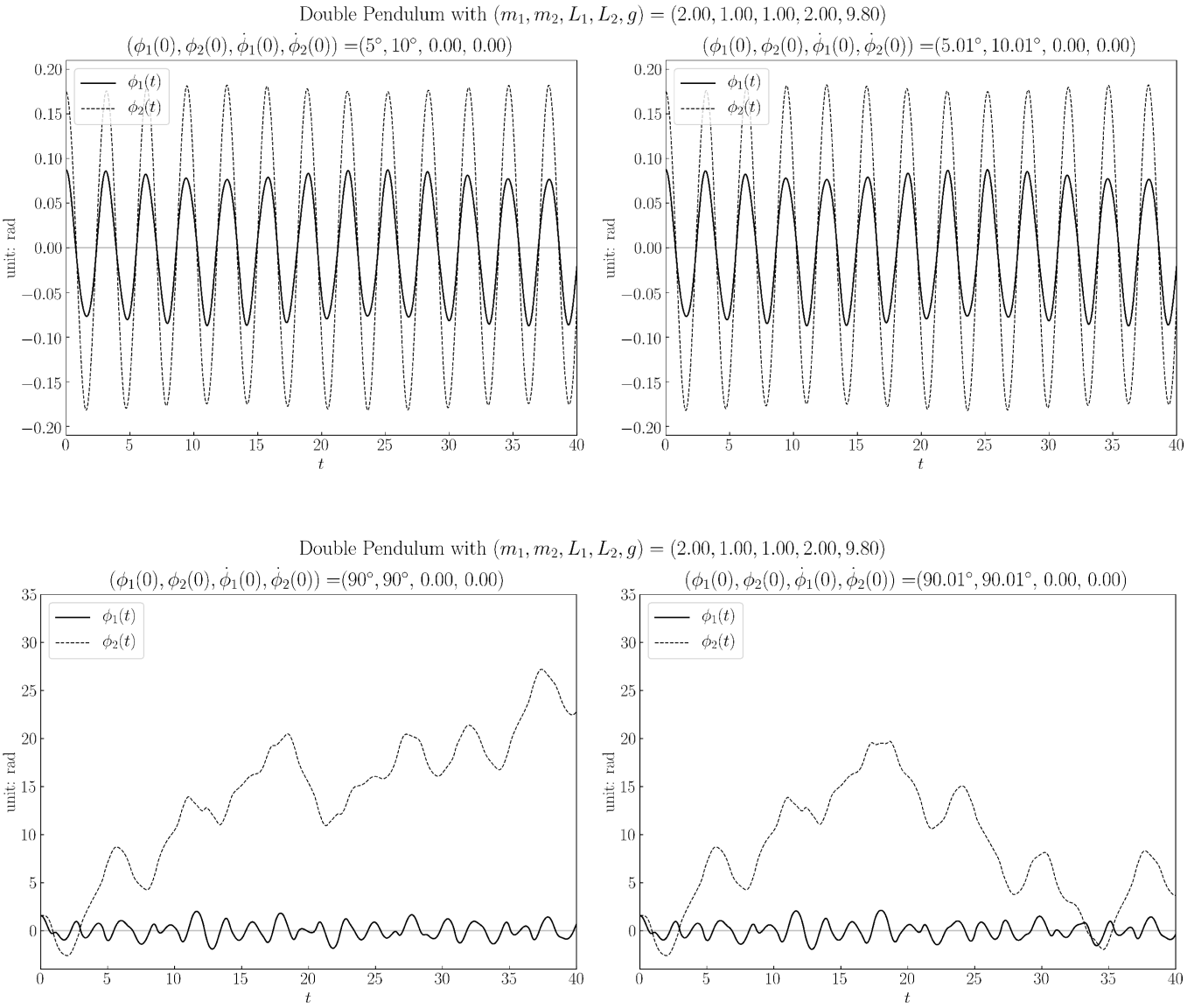
Let $\mathbf{y} = \begin{pmatrix} \phi_1 \\ \phi_2 \\ \dot{\phi}_1 \\ \dot{\phi}_2 \end{pmatrix}$, so $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = M^{-1}N$, and this is exactly the equation we will code. For convenience, in the Python code, the input should be a *row* vector representing $\mathbf{y}$, and the output should also be a *row* vector representing $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}$. Explicitly, this means we should write

```
    def dy_dt(self, t, y):
        """
        This function returns the right-hand side of the diffeq, inv(M).N
        Parameters
           t : float, time
           y : float, a row vector = [phi1, phi2, phi1_dot, phi2_dot]
        Returns
           a row vector representing dy/dt = [phi1_dot, phi2_dot, phi1_ddot, phi2_ddot]
        """
        # M is a 4-row by 4-column matrix
        M = np.array([[1, 0, 0, 0],
                      [0, 1, 0, 0],
                      [0, 0, (self.m1 + self.m2)*self.L1, self.m2*self.L2*np.cos(y[0]-y[1])],
                      [0, 0, self.L1*np.cos(y[0]-y[1]), self.L2]
                     ])
        # N is a 4-row by 1-column matrix
        N = np.array([[y[2]],
                      [y[3]],
                      [-self.m2*self.L2*y[3]*y[3]*np.sin(y[0]-y[1]) - (self.m1+self.m2)*self.g*np.sin(y[0])],
                      [self.L1*y[2]*y[2]*np.sin(y[0]-y[1]) - self.g*np.sin(y[1])]
                     ])
        # v is a 4-row by 1-column matrix
        y_dot = np.dot(np.linalg.inv(M), N)
        # the array returned is a row vector
        return [y_dot[0][0], y_dot[1][0], y_dot[2][0], y_dot[3][0]]
```

When we need to solve the problem with specific values of parameters and initial conditions, we need to call the class and use it. For example, if the name of the class is `DoublePendulum`, then we can do

```
# parameters
m1 = 2.0
m2 = 1.0
L1 = 1.0
L2 = 2.0
g = 9.8

# time domain and its mesh
t_start = 0.
t_end = 20.
delta_t = 0.001
t_pts = np.arange(t_start, t_end+delta_t, delta_t)

# initial condition
phi1_0 = np.pi/2.0
phi2_0 = np.pi/2.0
phi1_dot_0 = 0.0
phi2_dot_0 = 0.0

# Instantiate a pendulum
P = DoublePendulum(m1=m1, m2=m2, L1=L1, L2=L2, g=g)

# the solution: phi1(t), phi2(t), phi1_dot(t), phi2_dot(t)
phi1, phi2, phi1_dot, phi2_dot = P.solve_ode(t_pts, phi1_0, phi2_0, phi1_dot_0, phi2_dot_0)
```

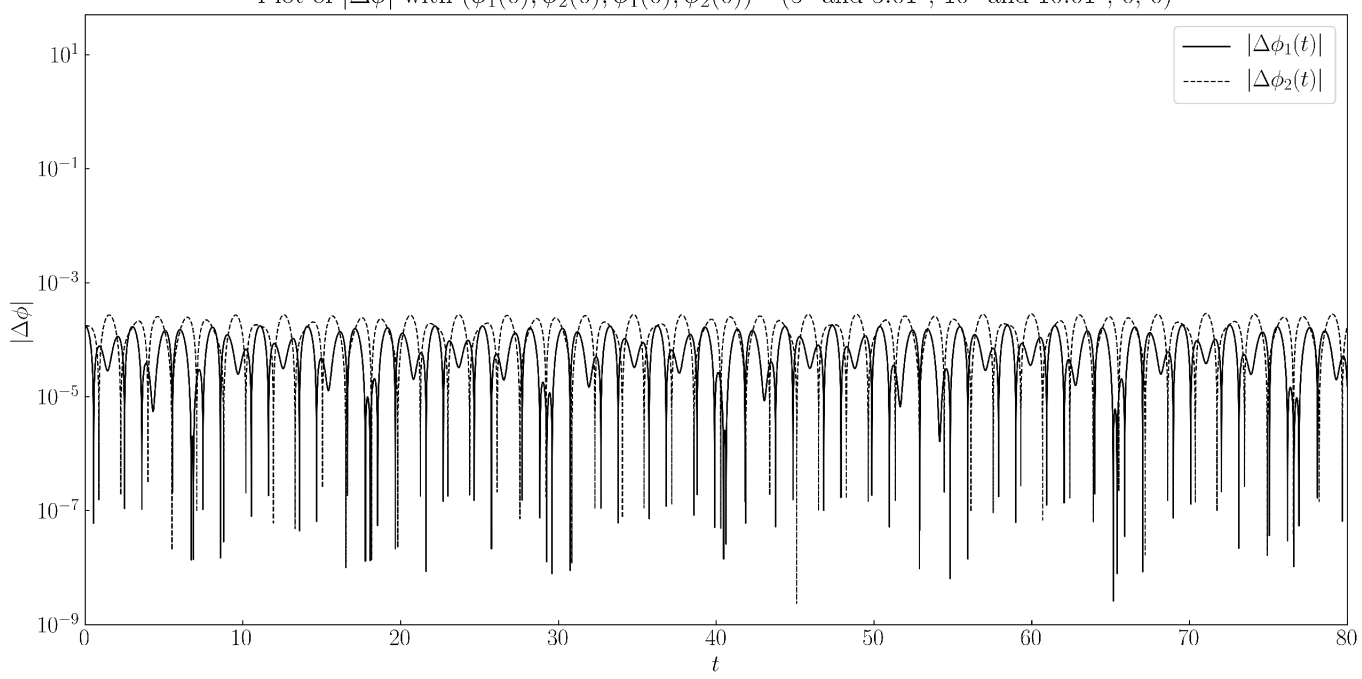
Sample outputs

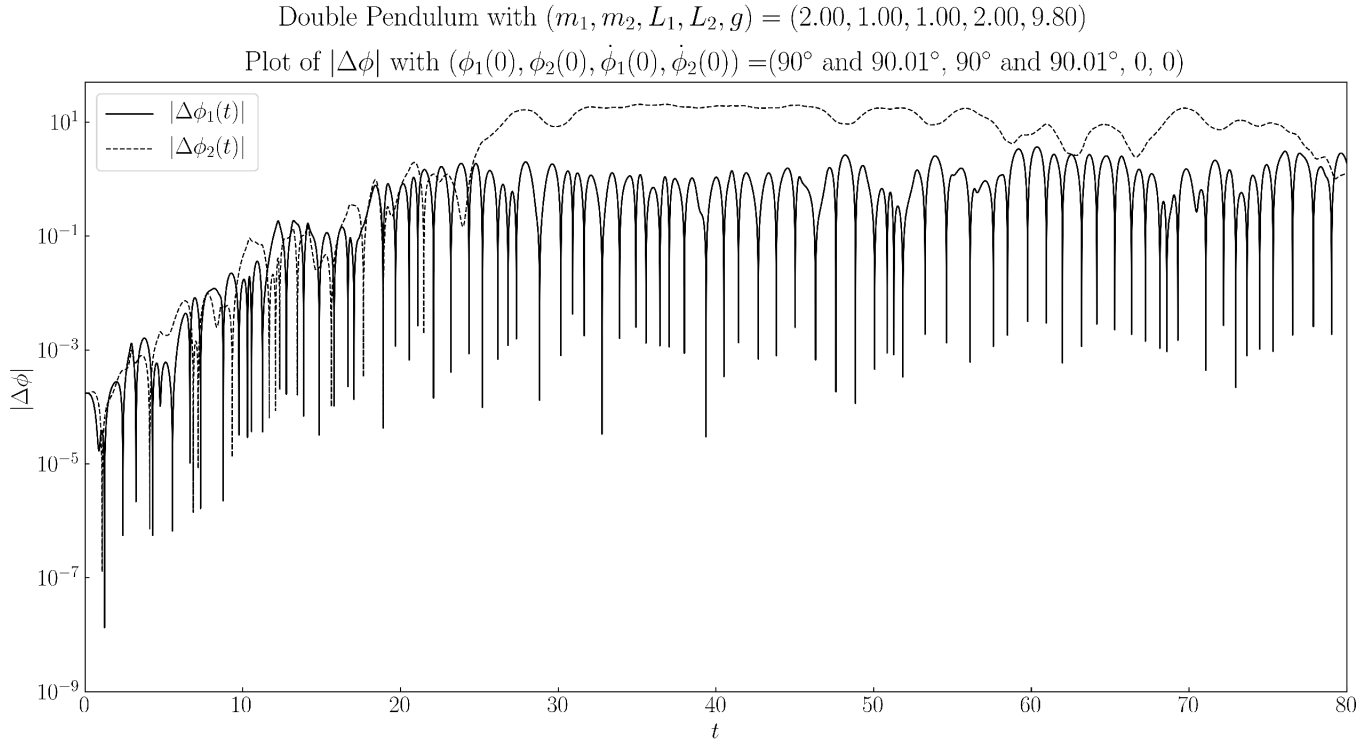


For small angle, we can use $\cos\phi \approx 1 - \dfrac{\phi^2}{2}$ and $\cos(\phi_1 - \phi_2) \approx 0$, then the equations of motion can be approximated as a linear second-order differential equations, and the solution should be periodic. The plots also show that the solutions are not sensitive to initial conditions: slight difference $\Delta\phi_1(0) = \Delta\phi_1(0) = 0.01°$ gives almost the same behavior.

For large initial angles, the preceding approximations fails, and the plots show that the solutions are not periodic. In this case, the solutions are extremely sensitive to initial conditions: slight difference $\Delta\phi_1(0) = \Delta\phi_1(0) = 0.01°$ gives dramatically different behaviors at later time.

We can actually plot $|\Delta\phi_1|, |\Delta\phi_2|$ as a function of time. These plots directly show that for small initial angles, $|\Delta\phi_1|, |\Delta\phi_2| \approx 0.0001$ all time, but for large initial angles, $|\Delta\phi_1|, |\Delta\phi_2|$ increase to about 10, which means they are in chaos. This agrees with what we expected. See plots on the next page.


Double Pendulum with $(m_1, m_2, L_1, L_2, g) = (2.00, 1.00, 1.00, 2.00, 9.80)$
Plot of $|\Delta\phi|$ with $(\phi_1(0), \phi_2(0), \dot{\phi}_1(0), \dot{\phi}_2(0))$ =(5° and 5.01°, 10° and 10.01°, 0, 0)
$|\Delta\phi_1(t)|$
$|\Delta\phi_2(t)|$
$|\Delta\phi|$
t


Double Pendulum with $(m_1, m_2, L_1, L_2, g) = (2.00, 1.00, 1.00, 2.00, 9.80)$
Plot of $|\Delta\phi|$ with $(\phi_1(0), \phi_2(0), \dot{\phi}_1(0), \dot{\phi}_2(0))$ =(90° and 90.01°, 90° and 90.01°, 0, 0)
$|\Delta\phi_1(t)|$
$|\Delta\phi_2(t)|$
$|\Delta\phi|$
t

2. Gravitational Orbits in Cartesian Coordinates

2.1 Two masses

The system of two masses m_1, m_2 under Newtonian gravitational interaction can be described by the Lagrangian

$$\mathcal{L} = \frac{1}{2}m_1\dot{\mathbf{x}}_1^2 + \frac{1}{2}m_2\dot{\mathbf{x}}_2^2 + \frac{Gm_1m_2}{|\mathbf{x}_1 - \mathbf{x}_2|}$$

where $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^2$ are the position vector of m_1 and m_2, G is a constant. The orbit of two masses in $\mathbb{R}^2$ is always our interest. The standard way for human beings to solve it is to convert this two-dimensional problem into a one-dimensional problem in polar coordinate by switching to the center of mass frame. However, for computer, we can perform a straightforward way in Cartesian coordinates; namely, let $\mathbf{x}_1 = (x_1, y_1)$ and $\mathbf{x}_2 = (x_2, y_2)$, then

$$\mathcal{L} = \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2) + \frac{Gm_1m_2}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}$$

and Euler-Lagrange equations are

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{x}_1} = \frac{\partial \mathcal{L}}{\partial x_1} \quad &\Rightarrow \quad \ddot{x}_1 = -Gm_2\frac{x_1 - x_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \\
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{y}_1} = \frac{\partial \mathcal{L}}{\partial y_1} \quad &\Rightarrow \quad \ddot{y}_1 = -Gm_2\frac{y_1 - y_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \\
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{x}_2} = \frac{\partial \mathcal{L}}{\partial x_2} \quad &\Rightarrow \quad \ddot{x}_2 = Gm_1\frac{x_1 - x_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \\
\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{y}_2} = \frac{\partial \mathcal{L}}{\partial y_2} \quad &\Rightarrow \quad \ddot{y}_2 = Gm_1\frac{y_1 - y_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}}
\end{aligned}$$

the matrix form of them is

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \dot{x}_1 \\ \dot{y}_1 \\ \dot{x}_2 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{x}_2 \\ \dot{y}_2 \\ -Gm_2\dfrac{x_1 - x_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \\ -Gm_2\dfrac{y_1 - y_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \\ Gm_1\dfrac{x_1 - x_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \\ Gm_1\dfrac{y_1 - y_2}{((x_1 - x_2)^2 + (y_1 - y_2)^2)^{\frac{3}{2}}} \end{pmatrix} \tag{$*$}$$

This is exactly the equation we will code.

Clearly, the equations of motion depend on three parameters m_1, m_2, and G. So, in a Python class, we need

```
    def __init__(self, m1=1.0, m2=1.0, G=1.0):
        """
        This function set the internal values of the class.
        Parameters
           m1 : float, mass 1
           m2 : float, mass 2
           G  : float, gravitational constant
        """
        self.m1 = m1
        self.m2 = m2
        self.G = G
```

Here $\mathbf{v}$ is a eight-dimensional vector, and $\dfrac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}$ has already been expressed explicitly. So, in a Python class, we have

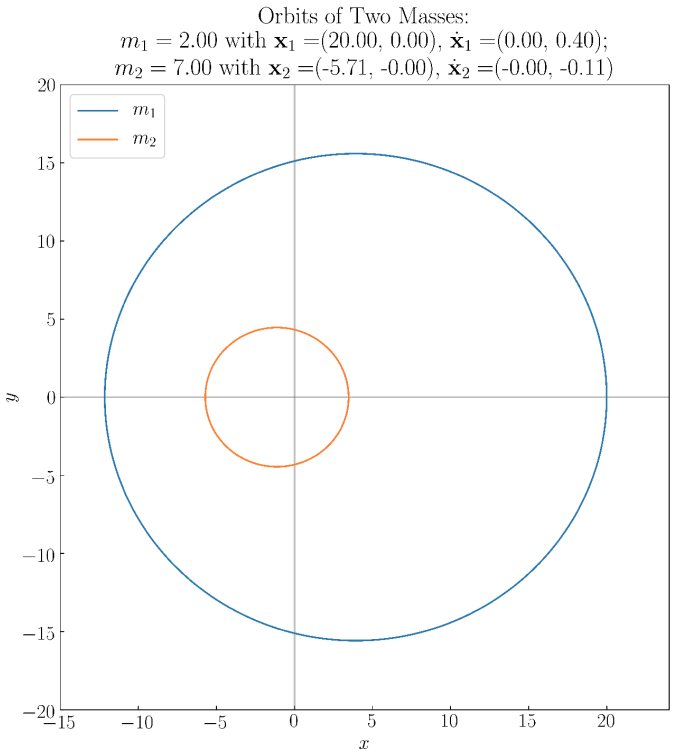
```
    def dv_dt(self, t, v):
        """
        This function returns the right-hand side of the diffeq
        Parameters
           t : float, time
           v : float, a row vector = [x1, y1, x2, y2, x1_dot, y1_dot, x2_dot, y2_dot]
        Returns
           a row vector representing dv/dt = [x1_dot, y1_dot, x2_dot, y2_dot, x1_ddot, y1_ddot, x2_ddot, y2_ddot]
        """
        return [v[4], v[5], v[6], v[7],
                ( -self.G*self.m2*(v[0]-v[2]) ) / ( ((v[0]-v[2])**2 + (v[1]-v[3])**2)**(3/2) ),
                ( -self.G*self.m2*(v[1]-v[3]) ) / ( ((v[0]-v[2])**2 + (v[1]-v[3])**2)**(3/2) ),
                (  self.G*self.m1*(v[0]-v[2]) ) / ( ((v[0]-v[2])**2 + (v[1]-v[3])**2)**(3/2) ),
                (  self.G*self.m1*(v[1]-v[3]) ) / ( ((v[0]-v[2])**2 + (v[1]-v[3])**2)**(3/2) )
               ]
```

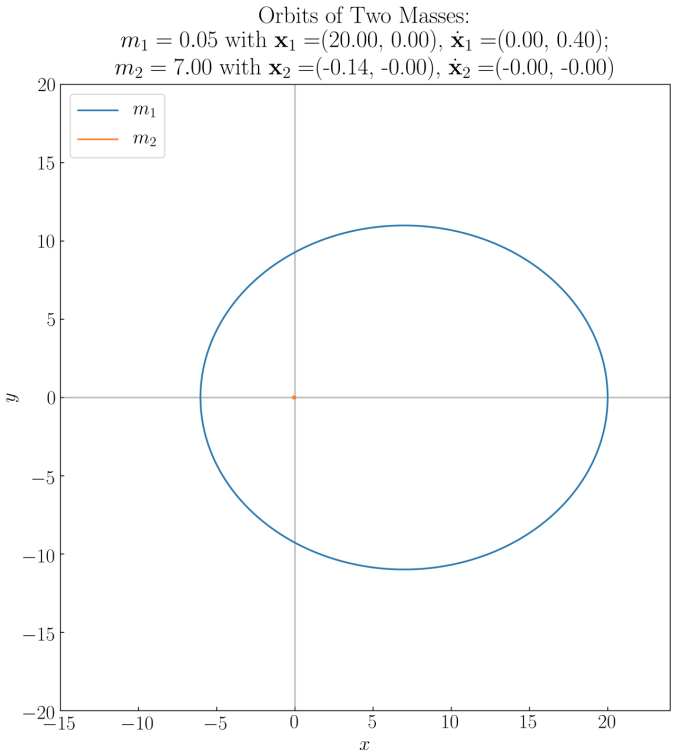
When we need to solve the problem with specific values of parameters and initial conditions, we need to call the class and use it. For example, if the name of the class is `TwoMassesGravity`, then we can do

```
# parameters
m1 = 2
m2 = 7
G = 1

# time domain and its mesh
t_start = 0.0
t_end = 500.
delta_t = 0.001
t_pts = np.arange(t_start, t_end+delta_t, delta_t)

# initial conditions
x1_0, y1_0 = 20, 0
x2_0, y2_0 = -(m1/m2)*x1_0, -(m1/m2)*y1_0
x1_dot_0, y1_dot_0 = 0, 0.4
x2_dot_0, y2_dot_0 = -(m1/m2)*x1_dot_0, -(m1/m2)*y1_dot_0

# Instantiate a TwoMassGravity class
P = TwoMassesGravity(m1=m1, m2=m2, G=G)
x1, y1, x2, y2, x1_dot, y1_dot, x2_dot, y2_dot = P.solve_ode(t_pts, x1_0, y1_0, x2_0, y2_0, x1_dot_0, y1_dot_0, x2_dot_0, y2_dot_0)
```

where for convenience, we always set $G = 1$.

Sample outputs





In preceding outputs, we set $\mathbf{x}_2(0) = -\dfrac{m_1}{m_2}\mathbf{x}_1(0)$ and $\dot{\mathbf{x}}_2(0) = -\dfrac{m_1}{m_2}\dot{\mathbf{x}}_1(0)$ to ensure that their center of mass is the

origin $(0,0)$. The two plots above confirm that if $\dfrac{m_2}{m_1}$ becomes larger and the other initial conditions are held fixed, m_2 will have a extremely small orbit around the center of mass, or say it is almost stationary. An real example of the second plot is the Sun-Earth system: $m_\odot \gg m_\oplus$, and so it is reasonable to treat the sun stationary when deal with related problems.

Using leapfrog method

Leapfrog method is a numerical method to find the solution of $\ddot{x} = A(x)$ for $t \in I$, where $A(x)$ is the given function. Let $v = \dot{x}$, then $\dot{v} = A(x)$. Impose a partition of I: $t_0 < t_1 < \cdots < t_i < t_{i+1} < \cdots < t_f$ with equal step $t_{i+1} - t_i = \Delta t$. The leapfrog method uses the middle step $t_i < t_{i+\frac{1}{2}} < t_{i+1}$ to calculate v and x. By definition of derivative, we have

$$\dot{v}(t_i) = A(x(t_i)) = \frac{v(t_{i+\frac{1}{2}}) - v(t_i)}{\frac{\Delta t}{2}} \qquad \text{(forward derivative with half step)}$$

$$\dot{x}(t_{i+\frac{1}{2}}) = \frac{x(t_{i+1}) - x(t_i)}{\Delta t} \qquad \text{(central derivative)}$$

$$\dot{v}(t_{i+1}) = A(x(t_{i+1})) = \frac{v(t_{i+1}) - v(t_{i+\frac{1}{2}})}{\frac{\Delta t}{2}} \qquad \text{(forward derivative with half step)}$$

rewrite them as

$$v(t_{i+\frac{1}{2}}) = v(t_i) + A(x(t_i))\frac{\Delta t}{2}$$
$$x(t_{i+1}) = x(t_i) + v(t_{i+\frac{1}{2}})\Delta t$$
$$v(t_{i+1}) = v(t_{i+\frac{1}{2}}) + A(x(t_{i+1}))\frac{\Delta t}{2}$$

So, starting from $i = 0$, we can find $v(t_{\frac{1}{2}})$ from the first equation with providing the initial conditions $x(t_0)$, $v(t_0)$, and then calculate the second equation, the third equation, and so on. Finally, we will have the solutions as the list of $\{x(t_0), x(t_1), x(t_2), ..., x(t_f)\}$ and of $\{v(t_0), v(t_{\frac{1}{2}}), v(t_1), v(t_{\frac{3}{2}}), ..., v(t_f)\}$.

In our problem, we have four second-order ODEs which means we need to write four sets of Leapfrog iteration algorithm. For example, for x_1 and $\dot{x}_1$, we have

$$\dot{x}_1(t_{i+\frac{1}{2}}) = \dot{x}_1(t_i) + \underbrace{\left(-Gm_2\frac{x_1(t_i) - x_2(t_i)}{((x_1(t_i) - x_2(t_i))^2 + (y_1(t_i) - y_2(t_i))^2)^{\frac{3}{2}}}\right)}_{\texttt{self.dv\_dt(t,v)[4]}}\frac{\Delta t}{2}$$

$$x_1(t_{i+1}) = x_1(t_i) + \dot{x}_1(t_{i+\frac{1}{2}})\Delta t$$

refresh

$$\dot{x}_1(t_{i+1}) = \dot{x}_1(t_{i+\frac{1}{2}}) + \underbrace{\left(-Gm_2\frac{x_1(t_{i+1}) - x_2(t_{i+1})}{((x_1(t_{i+1}) - x_2(t_{i+1}))^2 + (y_1(t_{i+1}) - y_2(t_{i+1}))^2)^{\frac{3}{2}}}\right)}_{\texttt{self.dv\_dt(t,v)[4]}\text{, but for new } \texttt{v}}\frac{\Delta t}{2}$$

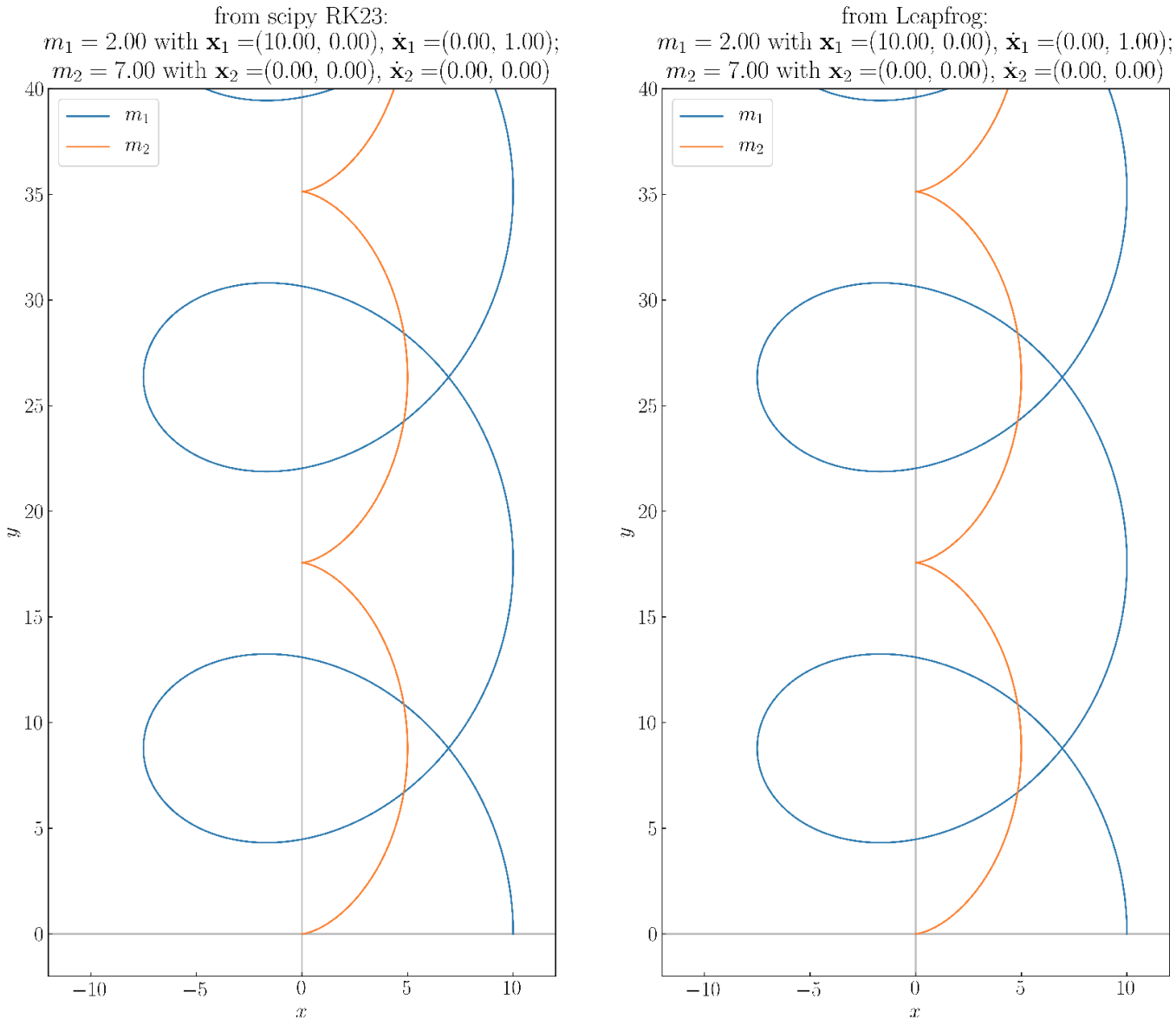
where in code the formula in parentheses is just `self.dv_dt(t,v)[4]` which we already coded before.

Attention! After implement of the first two equations for x_1, y_1, x_2, y_2, we need to refresh our vector. The reason is that the value of x_1 is now $x_1(t_{i+1})$, same for others, and we need to use it in `self.dv_dt(t,v)[4]` in the third equation, but the value of $\dot{x}$ is not $\dot{x}_1(t_{i+1})$ right now and it will reach it by implement of the third equation.

$$\text{refresh} : (x_1, y_1, x_2, y_2, \dot{x}_1, \dot{y}_2, \dot{x}_1, \dot{y}_2) \leftarrow (x_1(t_{i+1}), y_1(t_{i+1}), x_2(t_{i+1}), y_2(t_{i+1}), \dot{x}_1(t_i), \dot{y}_2(t_i), \dot{x}_1(t_i), \dot{y}_2(t_i))$$

See the actual code and the outputs obtained from Leapfrog method comparing to the one obtained from Runge-Kutta method of order 3(2), `scipy.integrate.RK23` below.

```python
    def solve_ode_Leapfrog(self, t_pts, x1_0, y1_0, x2_0, y2_0, x1_dot_0, y1_dot_0, x2_dot_0, y2_dot_0):
        """
        Solve the ODE given initial conditions with the Leapfrog method.
        Parameters
           same as self.solve_ode
        Returns
           same as self.solve_ode
        """
        delta_t = t_pts[1] - t_pts[0]

        # initialize the arrays for {x1, x1_dot, x1_dot_half}, {y1, y1_dot, y1_dot_half}, {x2, x2_dot, x2_dot_half}, {y2, y2_dot, y2_dot_half}
        num_t_pts = len(t_pts)
        x1, y1, x2, y2 = np.zeros(num_t_pts), np.zeros(num_t_pts), np.zeros(num_t_pts), np.zeros(num_t_pts)
        x1_dot, y1_dot, x2_dot, y2_dot = np.zeros(num_t_pts), np.zeros(num_t_pts), np.zeros(num_t_pts), np.zeros(num_t_pts)
        x1_dot_half, y1_dot_half, x2_dot_half, y2_dot_half = np.zeros(num_t_pts), np.zeros(num_t_pts), np.zeros(num_t_pts), np.zeros(num_t_pts)

        # initial conditions
        x1[0], y1[0], x2[0], y2[0] = x1_0, y1_0, x2_0, y2_0
        x1_dot[0], y1_dot[0], x2_dot[0], y2_dot[0] = x1_dot_0, y1_dot_0, x2_dot_0, y2_dot_0

        # the algorithm
        for i in np.arange(num_t_pts - 1):
            t = t_pts[i]

            v = [x1[i], y1[i], x2[i], y2[i], x1_dot[i], y1_dot[i], x2_dot[i], y2_dot[i]]

            x1_dot_half[i] = x1_dot[i] + self.dv_dt(t,v)[4]*delta_t/2.0
            x1[i+1] = x1[i] + x1_dot_half[i]*delta_t
            y1_dot_half[i] = y1_dot[i] + self.dv_dt(t,v)[5]*delta_t/2.0
            y1[i+1] = y1[i] + y1_dot_half[i]*delta_t
            x2_dot_half[i] = x2_dot[i] + self.dv_dt(t,v)[6]*delta_t/2.0
            x2[i+1] = x2[i] + x2_dot_half[i]*delta_t
            y2_dot_half[i] = y2_dot[i] + self.dv_dt(t,v)[7]*delta_t/2.0
            y2[i+1] = y2[i] + y2_dot_half[i]*delta_t

            v = [x1[i+1], y1[i+1], x2[i+1], y2[i+1], x1 dot[i], y1 dot[i], x2 dot[i], y2 dot[i]]

            x1_dot[i+1] = x1_dot_half[i] + self.dv_dt(t,v)[4]*delta_t/2.0
            y1_dot[i+1] = y1_dot_half[i] + self.dv_dt(t,v)[5]*delta_t/2.0
            x2_dot[i+1] = x2_dot_half[i] + self.dv_dt(t,v)[6]*delta_t/2.0
            y2_dot[i+1] = y2_dot_half[i] + self.dv_dt(t,v)[7]*delta_t/2.0

        return x1, y1, x2, y2, x1_dot, y1_dot, x2_dot, y2_dot
```

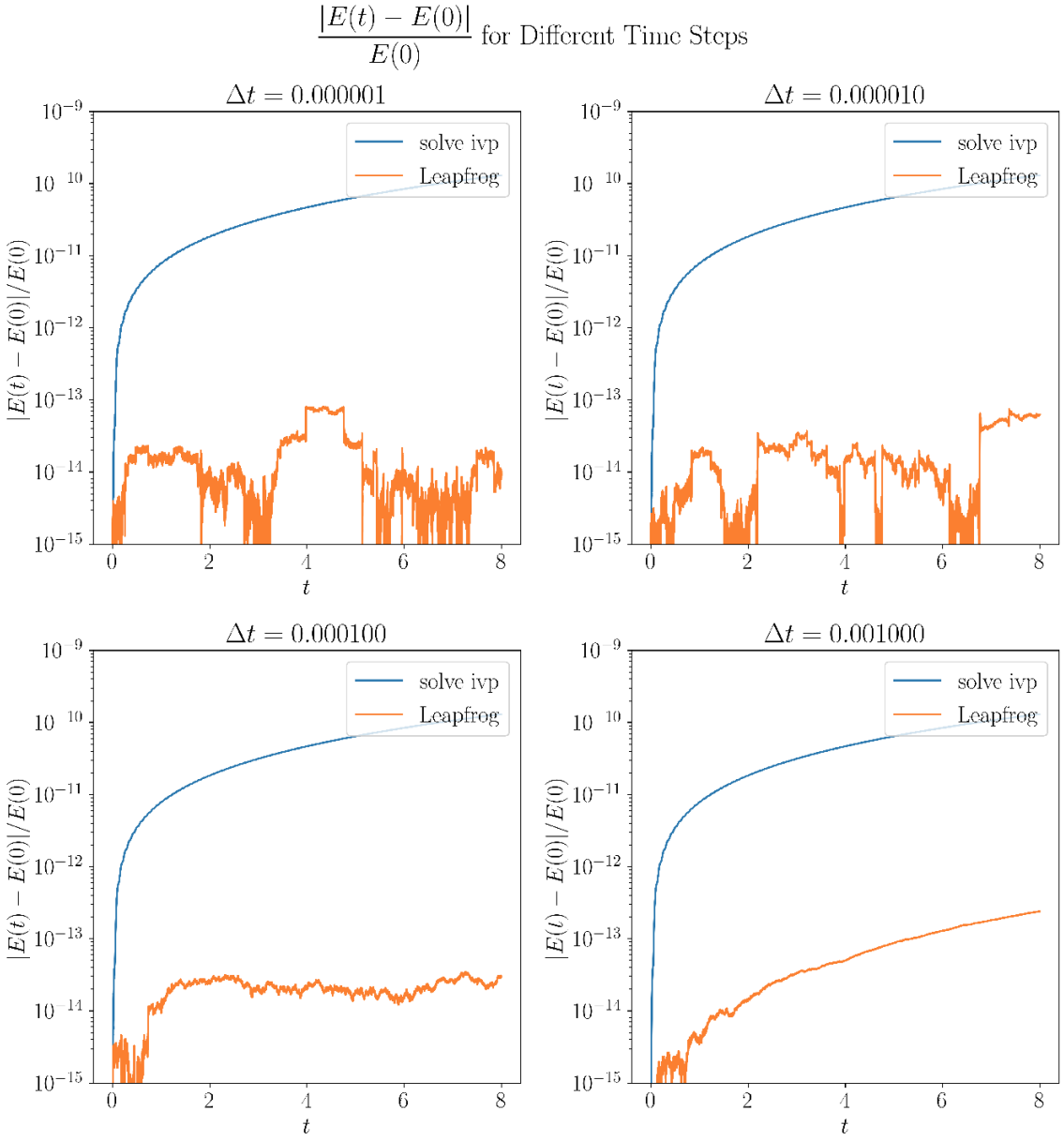


Note that the initial conditions of this one does not give the center of mass at origin.

One of the advantages of leapfrog method is it conserved energy every period. *Note that it does not conserved energy at every point.* See [2] for details. In order to verify it, we need another utility function in our class

```
95      def energy(self, t_pts, x1, y1, x2, y2, x1_dot, y1_dot, x2_dot, y2_dot):
96          """
97          Evaluate the energy as a function of time
98          """
99          K = (1/2)*self.m1*(x1_dot**2 + y1_dot**2) + (1/2)*self.m2*(x2_dot**2 + y2_dot**2)
100         U = - self.G * self.m1 * self.m2 / np.sqrt((x1-x2)**2 + (y1-y2)**2)
101         return K + U
```

and we need to analyze the plots of $\dfrac{|E(t) - E(0)|}{E(0)}$ as a function of time t with different time steps Δt. See plots on the next page where the initial conditions are what we used in the first plots.



We see that for small time step, the leapfrog method conserved energy roughly in a periodic way, but not at every point; however, for slightly large time step, it does not conserved energy as t increases. The Runge-Kutta method of order 3(2) does not conserved energy under any cases. Since our system does conserve energy, it is a good reason to choose leapfrog method using small time step to solve the equations of motion.

2.2 Three masses

For the case of three masses m_1, m_2, and m_3 under Newtonian interactions, the Lagrangian of the system is going to be

$$\mathcal{L} = \frac{1}{2}m_1\dot{\mathbf{x}}_1^2 + \frac{1}{2}m_2\dot{\mathbf{x}}_2^2 + \frac{1}{2}m_2\dot{\mathbf{x}}_3^2 + \frac{Gm_1m_2}{|\mathbf{x}_1-\mathbf{x}_2|} + \frac{Gm_1m_3}{|\mathbf{x}_1-\mathbf{x}_3|} + \frac{Gm_2m_3}{|\mathbf{x}_2-\mathbf{x}_3|}$$

Again, we are interested in the orbit in $\mathbb{R}^2$. So, let $\mathbf{x}_i = (x_i, y_i)$ for $i = 1, 2, 3$, then

$$\begin{aligned}\mathcal{L} = \quad & \frac{1}{2}m_1(\dot{x}_1^2+\dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2+\dot{y}_2^2) + \frac{1}{2}m_3(\dot{x}_3^2+\dot{y}_3^2) \\ & + \frac{Gm_1m_2}{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}} + \frac{Gm_1m_3}{\sqrt{(x_1-x_3)^2+(y_1-y_3)^2}} + \frac{Gm_2m_3}{\sqrt{(x_2-x_3)^2+(y_2-y_3)^2}}\end{aligned}$$

and Euler-Lagrange equations are

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{x}_1} = \frac{\partial \mathcal{L}}{\partial x_1} \quad \Rightarrow \quad \ddot{x}_1 = -Gm_2\frac{x_1 - x_2}{((x_1-x_2)^2+(y_1-y_2)^2)^{\frac{3}{2}}} - Gm_3\frac{x_1 - x_3}{((x_1-x_3)^2+(y_1-y_3)^2)^{\frac{3}{2}}}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{y}_1} = \frac{\partial \mathcal{L}}{\partial y_1} \quad \Rightarrow \quad \ddot{y}_1 = -Gm_2\frac{y_1 - y_2}{((x_1-x_2)^2+(y_1-y_2)^2)^{\frac{3}{2}}} - Gm_3\frac{y_1 - y_3}{((x_1-x_3)^2+(y_1-y_3)^2)^{\frac{3}{2}}}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{x}_2} = \frac{\partial \mathcal{L}}{\partial x_2} \quad \Rightarrow \quad \ddot{x}_2 = Gm_1\frac{x_1 - x_2}{((x_1-x_2)^2+(y_1-y_2)^2)^{\frac{3}{2}}} - Gm_3\frac{x_2 - x_3}{((x_2-x_3)^2+(y_2-y_3)^2)^{\frac{3}{2}}}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{y}_2} = \frac{\partial \mathcal{L}}{\partial y_2} \quad \Rightarrow \quad \ddot{y}_2 = Gm_1\frac{y_1 - y_2}{((x_1-x_2)^2+(y_1-y_2)^2)^{\frac{3}{2}}} - Gm_3\frac{y_2 - y_3}{((x_2-x_3)^2+(y_2-y_3)^2)^{\frac{3}{2}}}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{x}_3} = \frac{\partial \mathcal{L}}{\partial x_3} \quad \Rightarrow \quad \ddot{x}_3 = Gm_1\frac{x_1 - x_3}{((x_1-x_3)^2+(y_1-y_3)^2)^{\frac{3}{2}}} + Gm_2\frac{x_2 - x_3}{((x_2-x_3)^2+(y_2-y_3)^2)^{\frac{3}{2}}}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{y}_3} = \frac{\partial \mathcal{L}}{\partial y_3} \quad \Rightarrow \quad \ddot{y}_3 = Gm_1\frac{y_1 - y_3}{((x_1-x_3)^2+(y_1-y_3)^2)^{\frac{3}{2}}} + Gm_2\frac{y_2 - y_3}{((x_2-x_3)^2+(y_2-y_3)^2)^{\frac{3}{2}}}$$

code them in `dv_dt(self, t, v)`

```
    def __init__(self, m1=1.0, m2=1.0, m3=1.0, G=1.0):
        """
        This function set the internal values of the class.
        Parameters
           m1 : float, mass 1
           m2 : float, mass 2
           m3 : float, mass 3
           G  : float, gravitational constant
        """
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
        self.G = G

    def dv_dt(self, t, v):
        """
        This function returns the right-hand side of the diffeq
        Parameters
           t : float, time
           v : float, a row vector = [x1, y1, x2, y2, x3, y3, x1_dot, y1_dot, x2_dot, y2_dot, x3_dot, y3_dot]
        Returns
           a row vector representing dv/dt = [x1_dot, y1_dot, x2_dot, y2_dot, x3_dot, y3_dot, x1_ddot, y1_ddot, x2_ddot, y2_ddot, x3_ddot, y3_ddot]
        """
        return [v[6], v[7], v[8], v[9], v[10], v[11],
                (-self.G*self.m2*(v[0]-v[2])) / (((v[0]-v[2])**2+(v[1]-v[3])**2)**(3/2)) + (-self.G*self.m3*(v[0]-v[4])) / (((v[0]-v[4])**2+(v[1]-v[5])**2)**(3/2)),
                (-self.G*self.m2*(v[1]-v[3])) / (((v[0]-v[2])**2+(v[1]-v[3])**2)**(3/2)) + (-self.G*self.m3*(v[1]-v[5])) / (((v[0]-v[4])**2+(v[1]-v[5])**2)**(3/2)),
                ( self.G*self.m1*(v[0]-v[2])) / (((v[0]-v[2])**2+(v[1]-v[3])**2)**(3/2)) + (-self.G*self.m3*(v[2]-v[4])) / (((v[2]-v[4])**2+(v[3]-v[5])**2)**(3/2)),
                ( self.G*self.m1*(v[1]-v[3])) / (((v[0]-v[2])**2+(v[1]-v[3])**2)**(3/2)) + (-self.G*self.m3*(v[3]-v[5])) / (((v[2]-v[4])**2+(v[3]-v[5])**2)**(3/2)),
                ( self.G*self.m1*(v[0]-v[4])) / (((v[0]-v[4])**2+(v[1]-v[5])**2)**(3/2)) + ( self.G*self.m2*(v[2]-v[4])) / (((v[2]-v[4])**2+(v[3]-v[5])**2)**(3/2)),
                ( self.G*self.m1*(v[1]-v[5])) / (((v[0]-v[4])**2+(v[1]-v[5])**2)**(3/2)) + ( self.G*self.m2*(v[3]-v[5])) / (((v[2]-v[4])**2+(v[3]-v[5])**2)**(3/2))
               ]

    def solve_ode(self, t_pts, x1_0, y1_0, x2_0, y2_0, x3_0, y3_0, x1_dot_0, y1_dot_0, x2_dot_0, y2_dot_0, x3_dot_0, y3_dot_0, abserr=1.0e-9, relerr=1.0e-9):
        """
        Solve the ODE given initial conditions and specified abserr and relerr with solve_ivp from SciPy.
        Parameters
           x1_0, y1_0         : initial position of m1
           x2_0, y2_0         : initial position of m2
           x3_0, y3_0         : initial position of m3
           x1_dot_0, y1_dot_0 : initial velocity of m1
           x2_dot_0, y2_dot_0 : initial velocity of m2
           x3_dot_0, y3_dot_0 : initial velocity of m3
        Returns
           the list of x1(t), y1(t), x2(t), y2(t), x3(t), y3(t), x1_dot(t), y1_dot(t), x2_dot(t), y2_dot(t), x3_dot(t), y3_dot(3)
        """
        v_0 = [x1_0, y1_0, x2_0, y2_0, x3_0, y3_0, x1_dot_0, y1_dot_0, x2_dot_0, y2_dot_0, x3_dot_0, y3_dot_0]
        solution = solve_ivp(self.dv_dt, (t_pts[0], t_pts[-1]), v_0, t_eval=t_pts, method='RK23', atol=abserr, rtol=relerr)
        x1, y1, x2, y2, x3, y3, x1_dot, y1_dot, x2_dot, y2_dot, x3_dot, y3_dot = solution.y

        return x1, y1, x2, y2, x3, y3, x1_dot, y1_dot, x2_dot, y2_dot, x3_dot, y3_dot
```

Sample outputs

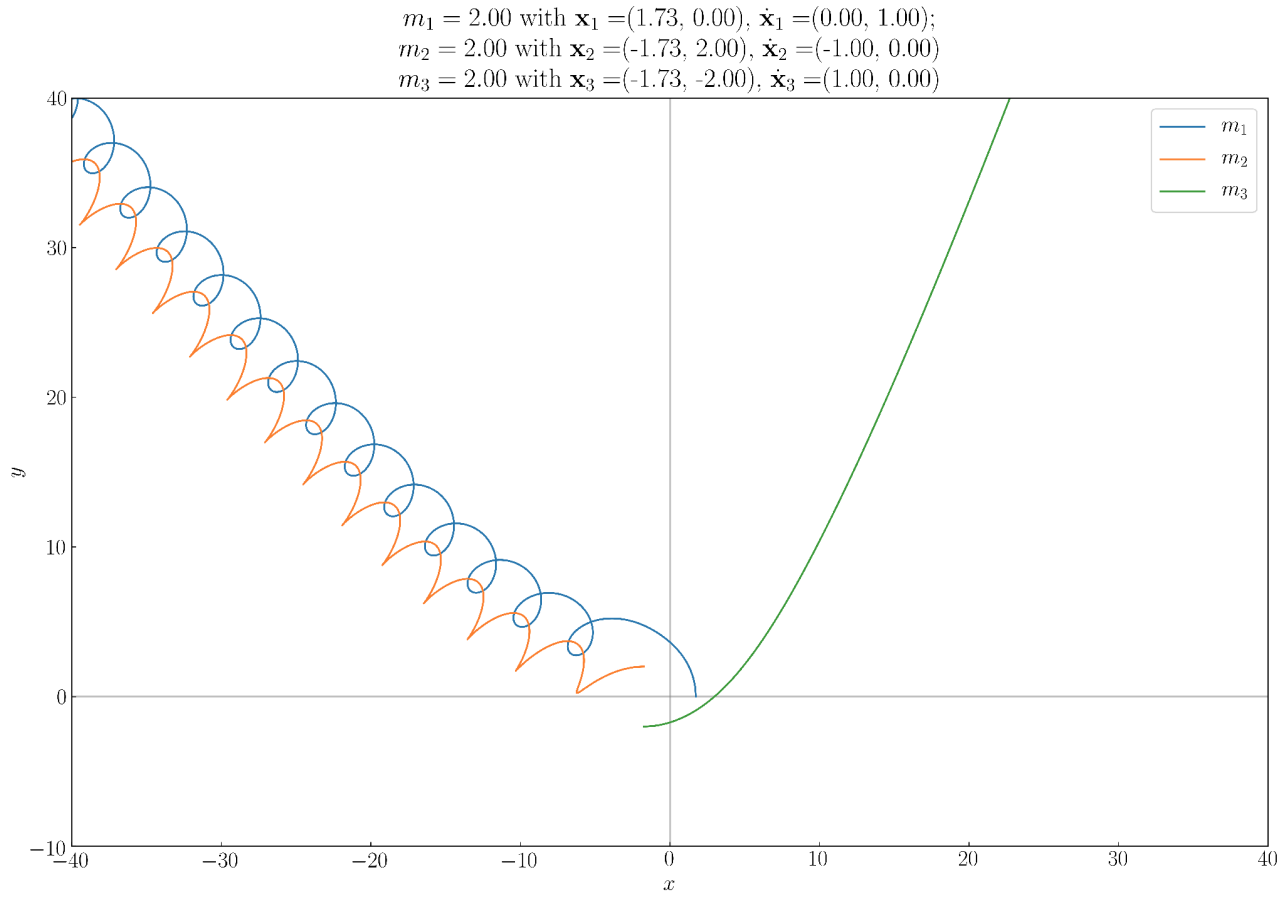
We can going to consider three initial cases.

Initial case 1: Equal mass on the vertices of a right triangle, namely $\mathbf{x}_1 = (\sqrt{3}, 0)$, $\mathbf{x}_2 = (-\sqrt{3}, 2)$, and $\mathbf{x}_3 = (-\sqrt{3}, -2)$, and each with tangential velocity.

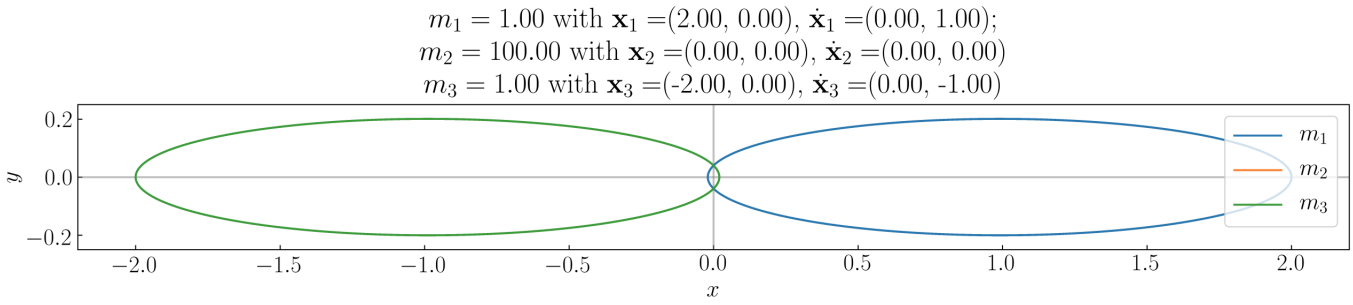
Initial case 2: Large mass is stationary at the middle of the line segment connecting m_1 and m_2. Velocities of m_1 and m_2 are in opposite y-direction.

Initial case 3: Equal mass on a line, the middle mass moves in y-direction and the other two moves towards each other.

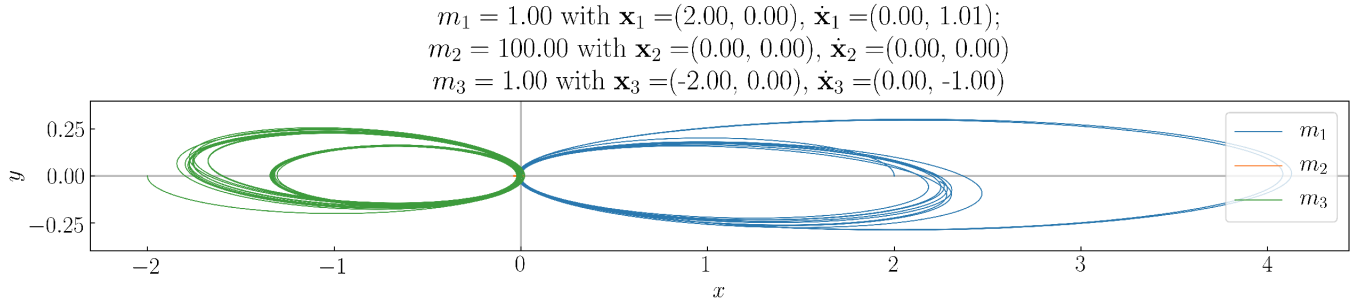
Initial case 1 We see that two of them tend to orbit around each other.



Initial case 2 The middle one has mass $m_2 \gg m_1 = m_3$ and it is stationary if $\dot{\mathbf{x}}_1 = -\dot{\mathbf{x}}_3 = (0, 1)$

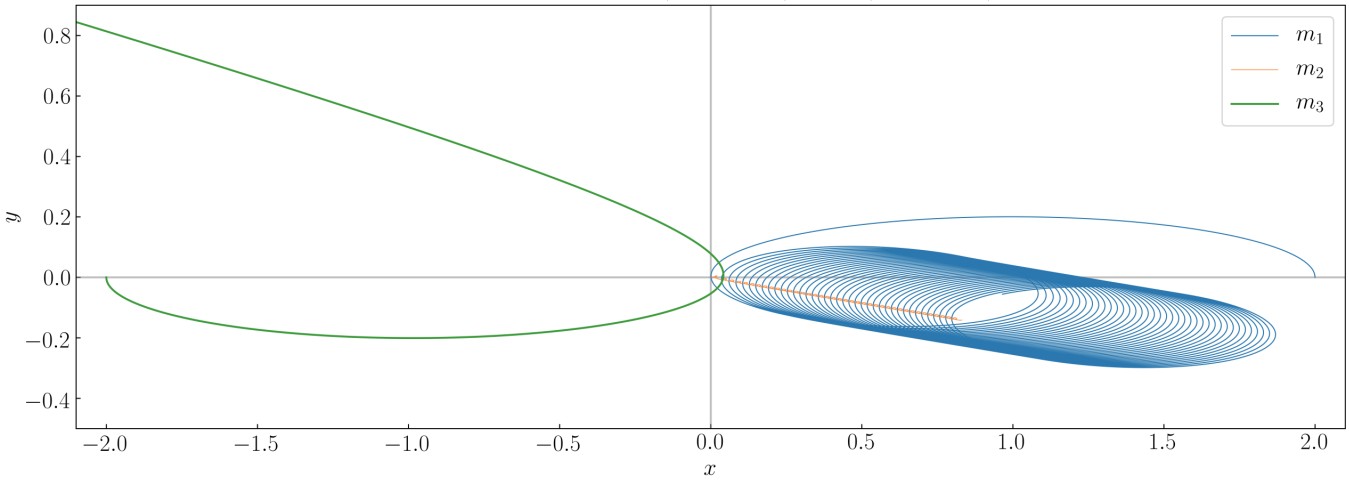


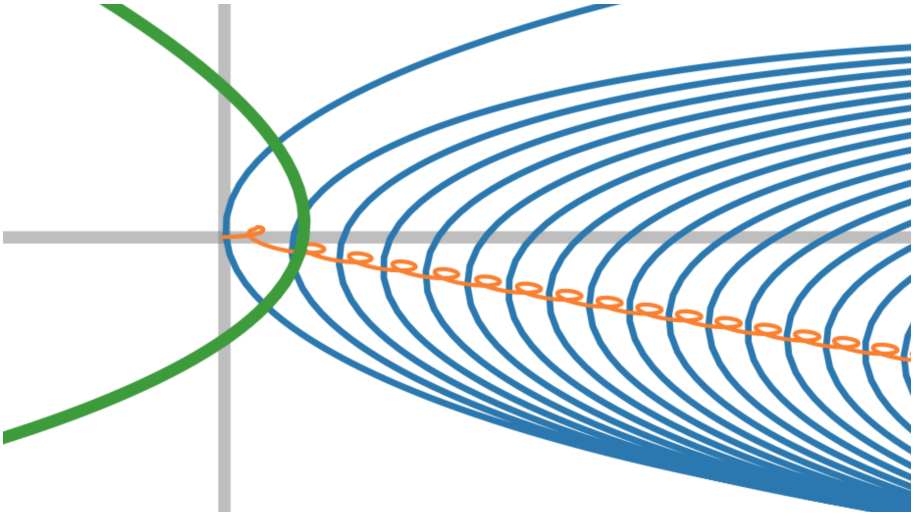
If we increase the velocity of one of the smaller mass a little bit, the behavior will be dramatically different. So, the three-body system under Newtonian gravitational interactions is actually chaotic.



Also, one can change the mass of one the smaller mass, the behavior again will be totally different. Again, two of them tend to orbit around each other. Notice that in this case, the middle, large mass is not moving in a straight line.

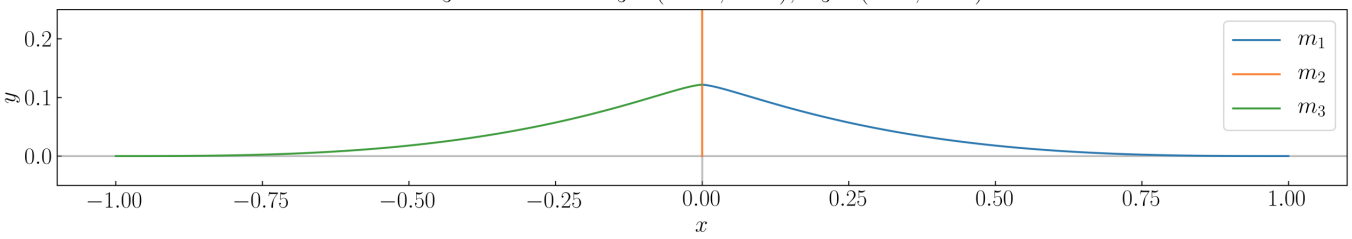
$m_1 = 2.00$ with $\mathbf{x}_1$ =(2.00, 0.00), $\dot{\mathbf{x}}_1$ =(0.00, 1.00);
$m_2 = 100.00$ with $\mathbf{x}_2$ =(0.00, 0.00), $\dot{\mathbf{x}}_2$ =(0.00, 0.00)
$m_3 = 1.00$ with $\mathbf{x}_3$ =(-2.00, 0.00), $\dot{\mathbf{x}}_3$ =(0.00, -1.00)





Initial case 3 The two masses on two sides will finally collide and their orbits stop after that.

$m_1 = 1.00$ with $\mathbf{x}_1$ =(1.00, 0.00), $\dot{\mathbf{x}}_1$ =(-1.00, 0.00);
$m_2 = 1.00$ with $\mathbf{x}_2$ =(0.00, 0.00), $\dot{\mathbf{x}}_2$ =(0.00, 1.00)
$m_3 = 1.00$ with $\mathbf{x}_3$ =(-1.00, 0.00), $\dot{\mathbf{x}}_3$ =(1.00, 0.00)



[1] J. Taylor, *Classical Mechanics* (University Science Books, 2005).

[2] R. Skeel, *Integration Schemes for Molecular Dynamics and Related Applications*, in *The Graduate Student's Guide to Numerical Analysis '98*, edited by M. Ainsworth, J. Levesley, and M. MarIetta (Springer-Verlag, Berlin, 1999), p. 191.