Report for Worksheet 1: Integrators

Markus Baur and David Beyer

November 8, 2019

Contents

1 Cannonball

1.1 Simulating a cannonball

The function for the (constant) gravitational force can be implemented in the following way in Python, it returns the force as a vector:

```
def force(mass, gravity):
    return np.array([0.0, -mass * gravity])
```

The Euler scheme is implemented in this fashion:

```
def step_euler(x, v, dt, mass, gravity, f):
    x += v * dt
    v += f / mass * dt
    return x, v
```

It is crucial to first update the positions $\mathbf{x}$ and then update the velocites $\mathbf{v}$. To simulate the cannonball until it hits the ground, a while-loop is used.

A plot of the simulated trajectory $y(x)$ is shown in Figure 1. As we would expect from the analytical solution

$$y(x) = x - \frac{gx^2}{2v_0^2}, \tag{1}$$

the trajectory looks like a parabola.

1.2 Influence of friction and wind

To account for friction, the function which calculates the force is modified in the following way:

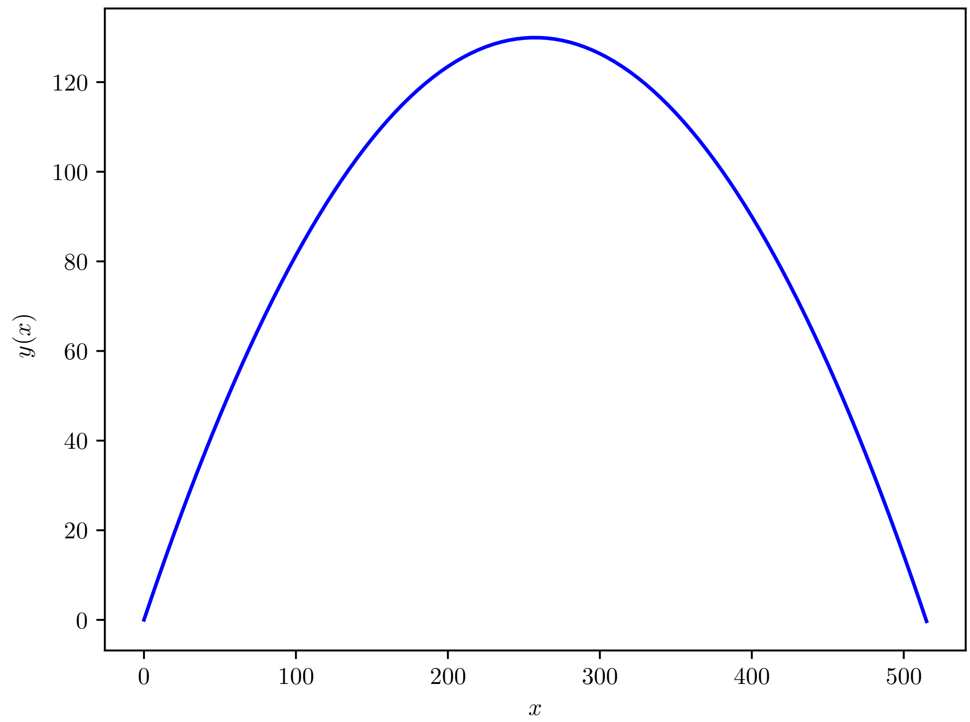

Figure 1: Simulated trajectory $y(x)$ for the system without friction. The used integrator is the Euler scheme.

```
def force(mass, gravity, v, gamma, v_0):
    ret = np.array([0.0, -mass * gravity])
    ret -= gamma * (v - v_0)
    return ret
```

Because the force is now not constant anymore along the trajectory (it varies with the velocity **v**), the function for the Euler step has to be modified as well:

```
def step(x, v, dt, mass, gravity, gamma, v_0):
    f = force(mass, gravity, v, gamma, v_0)
    x += v * dt
    v += f / mass * dt
    return x, v
```

In contrast to the previous task, the force is now evaluated every time the Euler step is called.

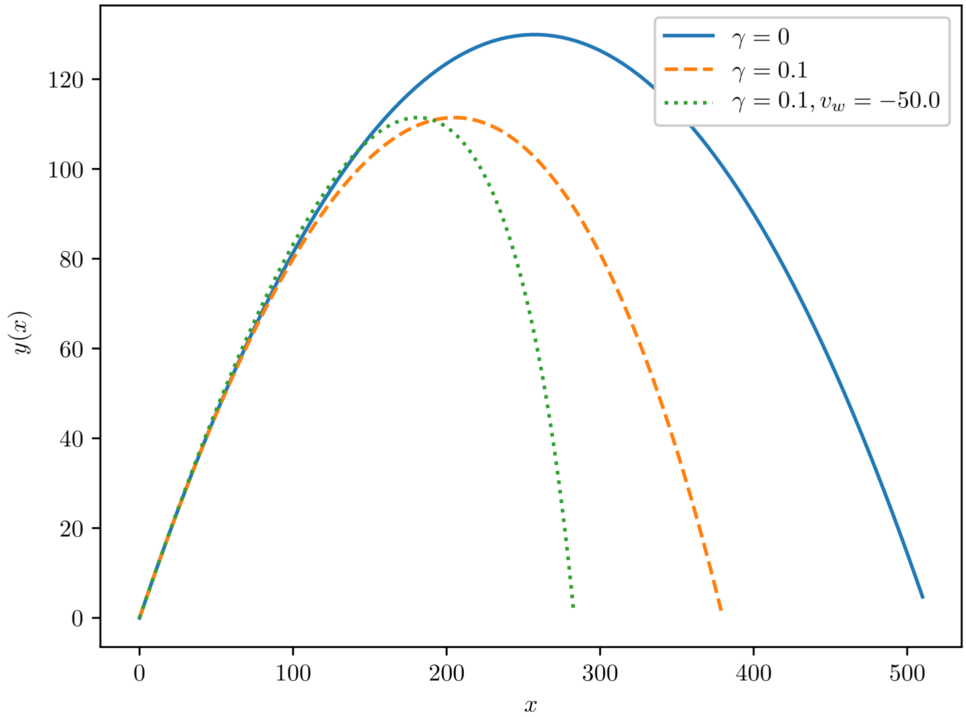

Figure 2: Simulated trajectories $y(x)$ for a friction coefficient $\gamma = 0.1$ and different values of the wind speed v_w. The used integrator is the Euler scheme.

Figure 2 shows simulated trajectories for the system without friction as well as for a system with friction coefficient $\gamma = 0.1$ and wind speeds of $v_w = 0, -50.0$. Comparing the trajectory of the cannonball without friction to the ones with friction, we can easily see that the friction leads to both a decreased maximum height y_{max} and a decreased range x_{max}. This is caused by the disspation of energy through the non-conservative friction force. We can also identify that the negative wind speed $v_w = -50.0$ leads to an even larger decrease of the range x_{max}, this is also expected, because the friction force is proportional to the relative velocity of the air and the cannonball. Because the wind blows in the x-direction only, the maximum height y_{max} is the same for both cases.

In Figure 3 we see multiple trajectories for $\gamma = 0.1$ and different wind speeds. As the wind speed becomes more negative, the range of the cannonball becomes smaller because the friction increases. For $v_w \approx 200$ it hits the ground at its starting point.

2 Solar system

2.1 Simulating the solar system with the Euler scheme

The gravitational force between two particles is calculated using this function:

```
def force(r_ij, m_i, m_j, g):
    return - g * m_i * m_j * r_ij / np.linalg.norm(r_ij) ** 3
```

To calculate all the forces on all the particles, the following function is used.

```
def forces(x, masses, g):
    ret = np.zeros(x.shape)

    for i in range(len(x)):
        for j in range(i + 1, len(x)):
```

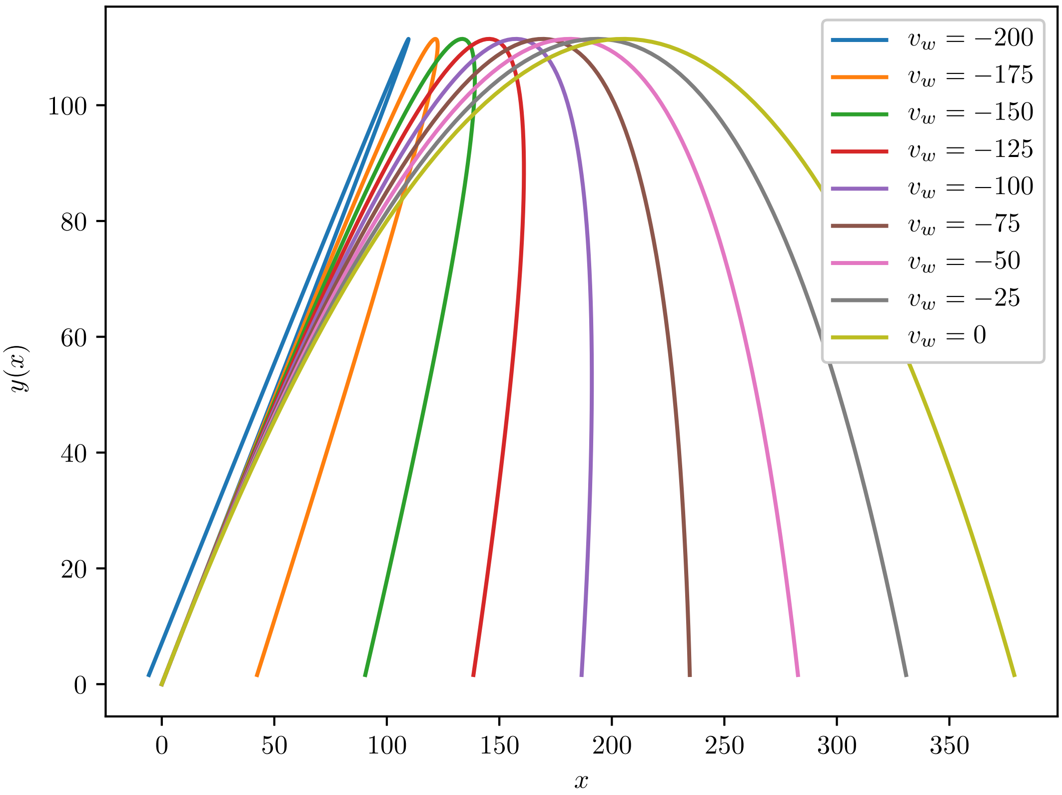

Figure 3: Simulated trajectories $y(x)$ for different values of the friction coefficient γ and the wind speed v_w. For $v_w = -200$, the cannonball hits the ground closely to the starting point. The used integrator is the Euler scheme.

```
            f = force(x[j] - x[i], masses[i], masses[j], g)
            ret[i] -= f
            ret[j] += f

    return ret
```

The function returns an array which contains all forces.

A small modification of the Euler step is necessary because the particles have different masses:

```
def step_euler(x, v, dt, mass, g):
    x += v * dt
    f = forces(x, mass, g)
    # calculate acceleration per coordinate dimension
    f[:, 0] /= mass
    f[:, 1] /= mass
    v += f * dt
    return x, v
```

In simulations with many particles, the computationally most expensive step is the evaluation of the forces $\mathbf{F}_{ij}$. For a system of n particles, the complexity of this task scales like $\mathcal{O}\left(n^2\right)$ because the pairwise forces $\mathbf{F}_{ij}$ have to be evaluated for every of the $n(n-1)$ possible combination of i, j (or at least half of them using Newton's third law).

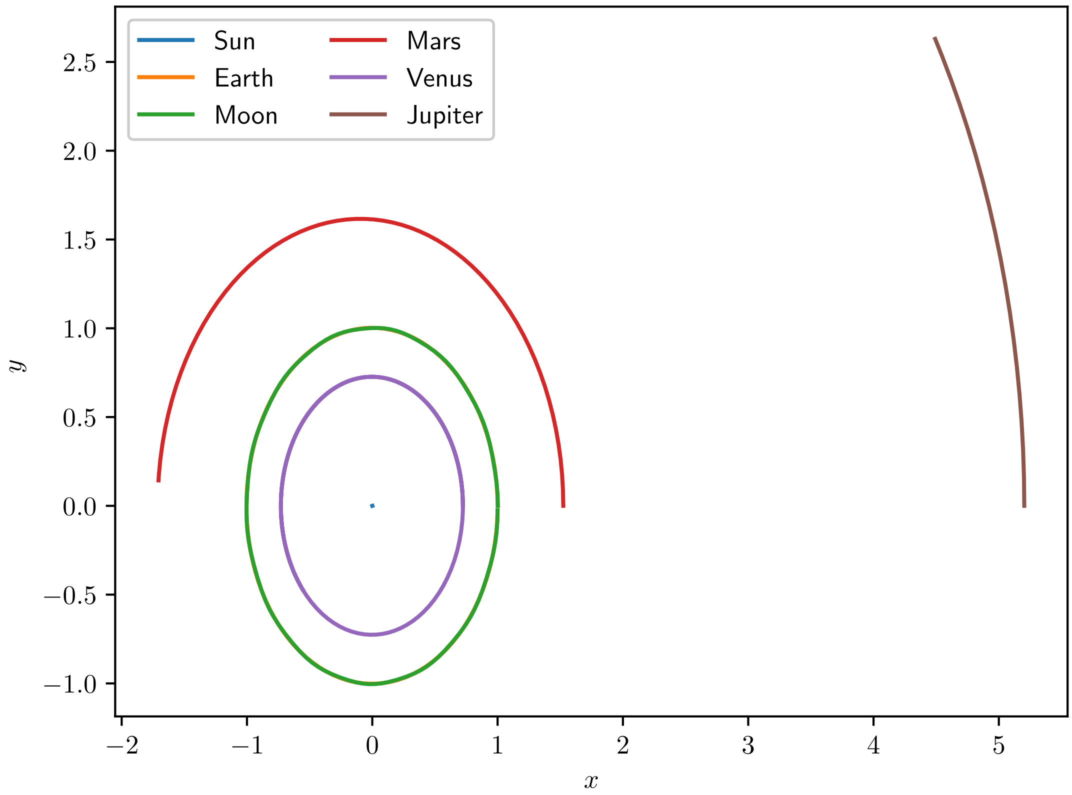

Figure 4:

2.2 Integrators

2.2.1 Velocity Verlet algorithm

The Velocity Verlet algorithm can be derived in the following way: For the positions $\mathbf{x}$, we perform a Taylor expansion up to the second order in Δt:

$$\begin{aligned} \mathbf{x}(t+\Delta t) &= \mathbf{x}(t) + \dot{\mathbf{x}}(t)\cdot\Delta t + \frac{\ddot{\mathbf{x}}(t)}{2}\cdot\Delta t^2 + \mathcal{O}(\Delta t^3) \\ &= \mathbf{x}(t) + \mathbf{v}(t)\cdot\Delta t + \frac{\mathbf{a}(t)}{2}\cdot\Delta t^2 + \mathcal{O}(\Delta t^3) \end{aligned} \tag{2}$$

For the velocites $\mathbf{v}$, we also perform a Taylor expansion up to the second order:

$$\begin{aligned} \mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \dot{\mathbf{v}}(t)\cdot\Delta t + \frac{\ddot{\mathbf{v}}(t)}{2}\cdot\Delta t^2 + \mathcal{O}(\Delta t^3) \\ &= \mathbf{v}(t) + \mathbf{a}(t)\cdot\Delta t + \frac{\ddot{\mathbf{v}}(t)}{2}\cdot\Delta t^2 + \mathcal{O}(\Delta t^3) \end{aligned} \tag{3}$$

To get an expression for $\ddot{\mathbf{v}}(t)$, we perform a Taylor expansion of $\dot{\mathbf{v}}(t+\Delta t)$:

$$\dot{\mathbf{v}}(t+\Delta t) = \dot{\mathbf{v}}(t) + \ddot{\mathbf{v}}(t)\cdot\Delta t + \mathcal{O}(\Delta t^2) \tag{4}$$

We can solve this expression for $\ddot{\mathbf{v}}(t)$:

$$\ddot{\mathbf{v}}(t) = \frac{\dot{\mathbf{v}}(t+\Delta t) - \dot{\mathbf{v}}(t)}{\Delta t} + \mathcal{O}(\Delta t) = \frac{\mathbf{a}(t+\Delta t) - \mathbf{a}(t)}{\Delta t} + \mathcal{O}(\Delta t) \tag{5}$$

Plugging $\ddot{\mathbf{v}}(t)$ into Equation 3, we get

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\cdot\Delta t + \frac{\mathbf{a}(t+\Delta t) + \mathbf{a}(t)}{2}\cdot\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{6}$$

Equation 2 and Equation 6 define the Velocity Verlet algorithm.

2.2.2 Verlet algorithm

The Verlet algorithm which was presented in the lecture can also be derived from the Velocity Verlet algorithm. Using Equation 2, we can write $\mathbf{x}(t)$ as

$$... \tag{7}$$

As we can see in ..., the Verlet algorithm needs the positions $\mathbf{x}(t)$ and $\mathbf{x}(t - \Delta t)$ to calculate the position $\mathbf{x}(t + \Delta t)$. However, the inital conditions only include the position at exactly one point in time t_0, this means that the initial conditions are not sufficient to solve the problem with the Verlet algorithm. To get the value of $\mathbf{x}(t_0 - \Delta t)$ we have to use another integrator like the Euler scheme, this results in a bigger error.

2.2.3 Implementation of the symplectic Euler algorithm

2.2.4 Implementation of the Velocity Verlet algorithm

2.3 Long-term stability