Worksheet 1: Integrators

Cameron N. Stewart

October 25, 2017

Institute for Computational Physics, University of Stuttgart

Contents

1 Introduction

In this worksheet we use will be using molecular dynamics simulation to look at the trajectory of a cannonball under the influence of gravity, friction, and wind and at a 2D representation of the solar system. We will be studying the behavior of a few different integrators in the latter simulation.

2 Cannonball

2.1 Simulating a cannonball

In this first exercise we simulate a cannonball in 2D under gravity in the absence of friction. The cannonball has a mass $m = 2.0\,\mathrm{kg}$,we take gravitational acceleration to be $g = 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}$, the cannonball has initial position $\boldsymbol{x}(0) = \mathbf{0}$, and initial velocity $\boldsymbol{v}(0) = \begin{pmatrix} 50 \\ 50 \end{pmatrix} \frac{\mathrm{m}}{\mathrm{s}}$. We will use the simple Euler scheme to integrate or system. This is given by:

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \boldsymbol{v}(t)\Delta t \tag{1}$$

$$\boldsymbol{v}(t + \Delta t) = \boldsymbol{v}(t) + \frac{\boldsymbol{F}(t)}{m}\Delta t \tag{2}$$

This is essentially just the Taylor expansion of position and velocity cut off below second order. We implement the simple Euler algorithm in python as follows:

```python
def step_euler(x, v, dt):
    f = compute_forces(x)
    x += v*dt
    v += f*dt/m
    return x, v
```

The forces are computed simply with

```python
def compute_forces(x):
    f = np.array([0.0, -m*g])
    return f
```

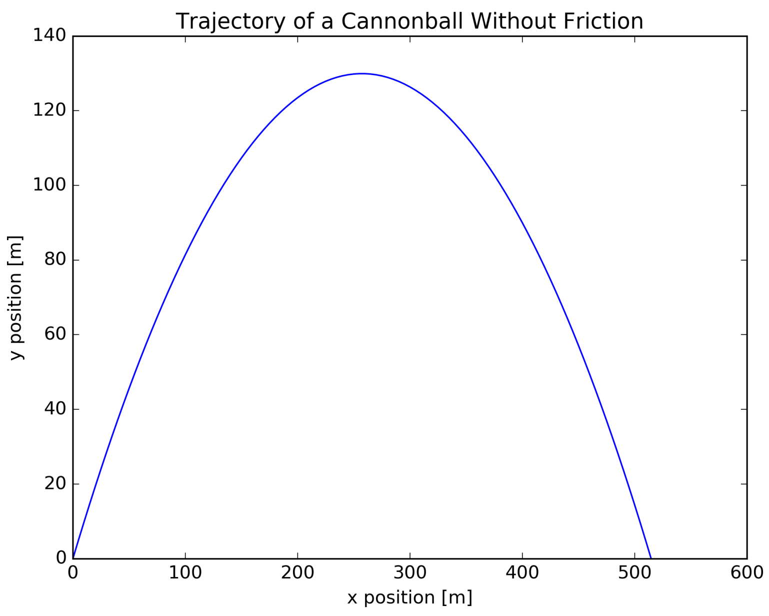

Figure 1: Trajectory of a cannonball with no friction using the simple Euler scheme

We integrate until the cannonball reaches the ground with a timestep $\Delta t = 01\,\mathrm{s}$. The trajectory can be seen in fig. 1 and the source code can be found at `src/cannonball.py`. As expected, the trajectory looks parabolic.

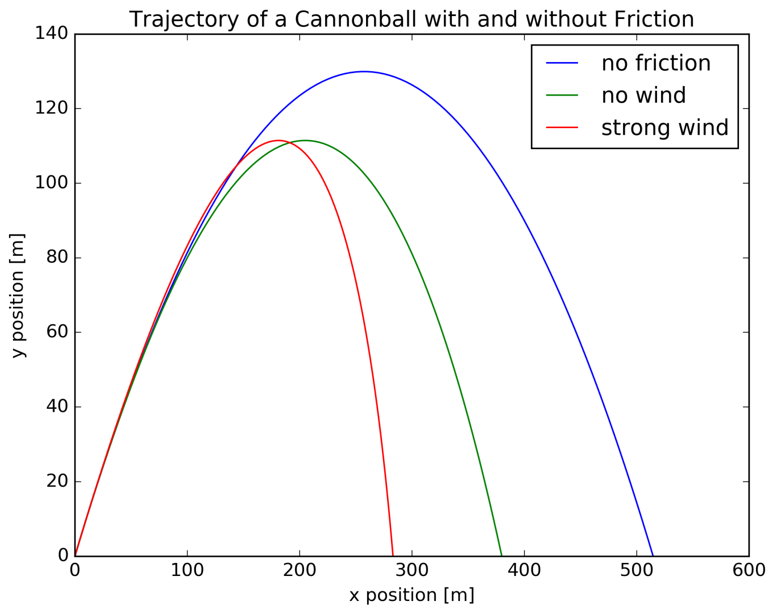

Figure 2: Trajectory of a cannonball with and without friction and for two different wind speeds

2.2 Influence of friction and wind

Next we will include a velocity dependant friction term in the force given by

$$F_{\text{fric}}(\boldsymbol{v}) = -\gamma(\boldsymbol{v} - \boldsymbol{v}_0) \tag{3}$$

where $\boldsymbol{v}_0 = \begin{pmatrix} v_{\text{w}} \\ 0 \end{pmatrix} \frac{\text{m}}{\text{s}}$ is the wind speed. The compute forces function was modified into the following form:

```
def compute_forces(x, v, y, vw):
    f_fric = -y*(v - np.array([vw, 0.0]))
    f = np.array([0.0, -m*g])+f_fric
    return f
```

We used a value of $\gamma = 0.1$ for the friction coefficient and once again used a time step of $\Delta t = 0.1$.

Figure 2 shows this simulation in three different cases. The original parabola is shown with no friction along with the cases $v_{\text{w}} = 0$ and $v_{\text{w}} = -50$. Adding friction lowers the maximum height and distance where as adding wind only changes the distance. The code can be found in `src/cannonball_fric.py`.

Finally, we ran the simulation for various wind speeds until the cannonball landed near its original launching point as seen in fig. 3. This occurred at wind speed near $v_{\text{w}} = -200$. The code can be seen at `src/cannonball_fric2.py`

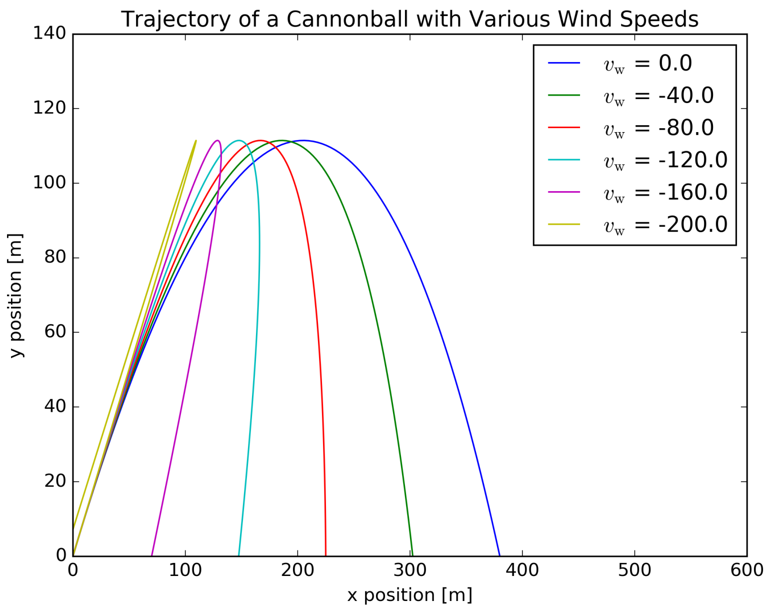

Figure 3: Trajectory of a cannonball with friction at various wind speeds

3 Solar system

In this exercise we perform and MD simulation of a 2d model of the solar system (with fewer planets than our own). The force on any given planet is given by a superposition of the force from all of other planets i.e.

$$\boldsymbol{F}_i = \sum_{\substack{j=0 \\ i \neq j}}^{N} \boldsymbol{F}_{ij} \tag{4}$$

where the force on i from j is given by

$$\boldsymbol{F}_{ij} = -G m_i m_j \frac{\boldsymbol{r}_{ij}}{|\boldsymbol{r}_{ij}|^3} \tag{5}$$

where m is the respective massses, G is the gravitational constant, and $/bmr_{ij}$ is the vector from j to i. This was implemented specifically as follows:

```
def compute_forces(x):
    f = np.zeros(x_init.shape)
    for i in range(M):
        for j in range(M):
            if i != j:
                r_ij = x[:,i] - x[:,j]
                f[:,i] += -g*m[i]*m[j]*r_ij/(la.norm(r_ij)**3)
    return f
```

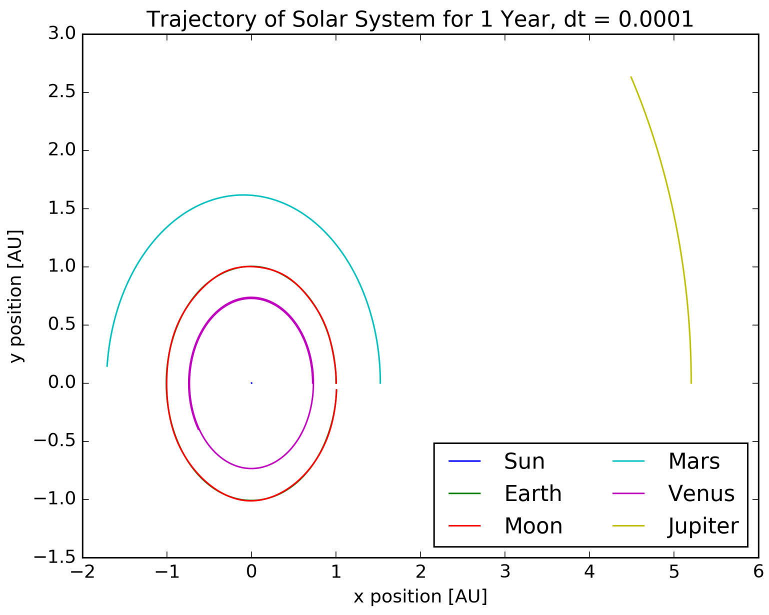

Figure 4: Trajectory of a 2d model of the solar system using a simple Euler integrator and a time step of $\Delta t = 0.0001$.

3.1 Simulating the solar system with the Euler scheme

The names, masses, initial positions, and initial velocities of the planets as well as the gravitational constant are read in from `src/solar_system.pkl.gz` using the Python module Pickle:

```
datafile = gzip.open('Solar_system.pkl.gz')
name, x_init, v_init, m, g = pickle.load(datafile)
datafile.close()
```

The first simulation was done using the simple Euler algorithm from the cannonball exercise. The simulation was run for 1 year ($t = 1.0$) with time steps of $\Delta t = 0.0001$. The trajectory can be seen in fig. 4 and the code is at `src/solar1.py`. In one year the Earth orbits the Sun once as expected and the moon doesn't leave the Earth.

Now we would like to run this simulation using a few different time steps and look at the trajectory of the moon in the rest frame of the Earth. This is seen in fig. 5 and the code is at `src/solar2.py`. Any time step larger than $\Delta t = 0.0001$ launches the moon into deep space. An unexpected behavior for the moon.

3.2 Integrators

The unexpected ejection of the moon illustrates the flawed nature of the simple Euler algorithm we've been using. In particular the algorithm is not symplectic meaning that the Hamiltonian is not conserved. One simple symplectic algorithm can be obtained by

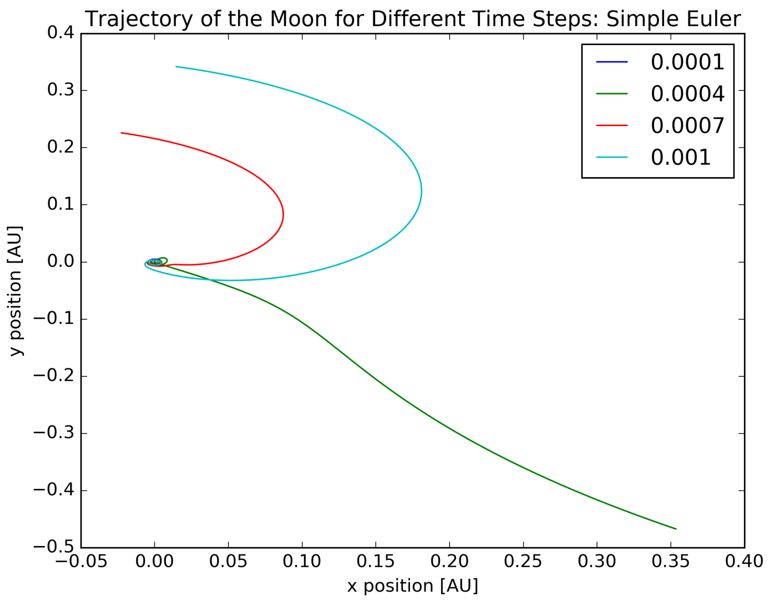

Figure 5: The trajectory of the moon in the rest frame of the Earth for various time steps

switching the order of the velocity and position updates like this:

$$\boxed{\begin{aligned} \boldsymbol{v}(t+\Delta t) &= \boldsymbol{v}(t) + \boldsymbol{a}(t)\Delta t \qquad (6)\\ \boldsymbol{x}(t+\Delta t) &= \boldsymbol{x}(t) + \boldsymbol{v}(t+\Delta t)\Delta t. \qquad (7) \end{aligned}}$$

This is the **Symplectic Euler Algorithm**. We implement this algorithm in python as follows:

```python
def step_eulersym(x, v, dt):
    f = compute_forces(x)
    v += f*dt/m
    x += v*dt
    return x, v
```

We now derive another symplectic integrator, the **Velocity Verlet Algorithm**. We begin by expanding the position and velocity to second order.

$$\boldsymbol{x}(t+\Delta t) = \boldsymbol{x}(t) + \frac{\partial}{\partial t}\boldsymbol{x}(t)\Delta t + \frac{1}{2}\frac{\partial^2}{\partial t^2}\boldsymbol{x}(t)\Delta t^2 \qquad (8)$$

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \frac{\partial}{\partial t}\boldsymbol{v}(t)\Delta t + \frac{1}{2}\frac{\partial^2}{\partial t^2}\boldsymbol{v}(t)\Delta t^2 \qquad (9)$$

Now we look for an expression for the second order term of velocity. We expand the time derivative of velocity to first order as

$$\frac{\partial}{\partial t}\boldsymbol{v}(t+\Delta t) = \frac{\partial}{\partial t}\boldsymbol{v}(t) + \frac{\partial^2}{\partial t^2}\boldsymbol{v}(t)\Delta t. \qquad (10)$$

Solving for the term of interest leaves us with

$$\frac{\partial^2}{\partial t^2}\boldsymbol{v}(t)\Delta t = \frac{\partial}{\partial t}\boldsymbol{v}(t+\Delta t) - \frac{\partial}{\partial t}\boldsymbol{v}(t) \tag{11}$$

Inserting into eq. 9 leaves us with

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \frac{1}{2}\left(\frac{\partial}{\partial t}\boldsymbol{v}(t) + \frac{\partial}{\partial t}\boldsymbol{v}(t+\Delta t)\right)\Delta t. \tag{12}$$

Inserting velocity and acceleration for the time derivatives in eqs. 8 and 12 gives us the final **Velocity Verlet Algorithm**

$$\boxed{\begin{aligned} \boldsymbol{x}(t+\Delta t) &= \boldsymbol{x}(t) + \boldsymbol{v}(t)\Delta t + \frac{\boldsymbol{a}(t)}{2}\Delta t^2 \qquad (13) \\ \boldsymbol{v}(t+\Delta t) &= \boldsymbol{v}(t) + \frac{\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t)}{2}\Delta t \qquad (14) \end{aligned}}$$

We can implement this algorithm in python in this way:

```python
def step_vv(x, v, a, dt):
    x += v*dt+a*dt**2/2
    v += a*dt/2
    a = compute_forces(x)/m
    v += a*dt/2
    return x, v, a
```

We do this in two steps so that we don't have to save current acceleration time step.

From this we now derive the **Verlet Algorithm**. We begin by writing eq. 13 for the current time step in terms of the previous time step. It looks like

$$\boldsymbol{x}(t) = \boldsymbol{x}(t-\Delta t) + \boldsymbol{v}(t-\Delta t)\Delta t + \frac{\boldsymbol{a}(t-\Delta t)}{2}\Delta t^2. \tag{15}$$

We then rearrange eq. 13 to solve for the position at the current time step giving

$$\boldsymbol{x}(t) = \boldsymbol{x}(t+\Delta t) - \boldsymbol{v}(t)\Delta t - \frac{\boldsymbol{a}(t)}{2}\Delta t^2. \tag{16}$$

Adding the previous two equations gives

$$2\boldsymbol{x}(t) = \boldsymbol{x}(t+\Delta t) + \boldsymbol{x}(t-\Delta t) + \left[\boldsymbol{v}(t-\Delta t) - \boldsymbol{v}(t)\right]\Delta t + \frac{1}{2}\left[\boldsymbol{a}(t-\Delta t) - \boldsymbol{a}(t)\right]\Delta t^2 \tag{17}$$

Now we write eq. 14 for the current time step in terms of the previous time step and rearrange the velocity terms to the left side giving

$$\boldsymbol{v}(t-\Delta t) - \boldsymbol{v}(t) = -\frac{1}{2}\left[\boldsymbol{a}(t-\Delta t) + \boldsymbol{a}(t)\right]\Delta t \tag{18}$$

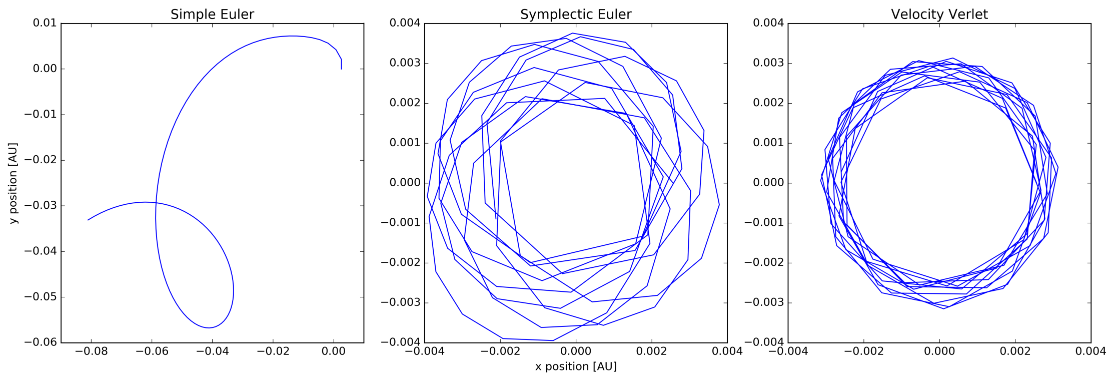

Figure 6: Trajectory of the moon using three different integrators and a time step of $\Delta t = 0.01$.

Inserting into eq. 17 leaves us with

$$2\boldsymbol{x}(t) = \boldsymbol{x}(t+\Delta t) + \boldsymbol{x}(t-\Delta t) - \boldsymbol{a}(t)\Delta t^2 \tag{19}$$

and we can finally rearrange to find the **Verlet Algorithm**:

$$\boxed{\boldsymbol{x}(t+\Delta t) = 2\boldsymbol{x}(t) - \boldsymbol{x}(t-\Delta t) + \boldsymbol{a}(t)\Delta t^2} \tag{20}$$

So we see that the **Verlet** and **Velocity Verlet** are equivalent. This is difficult to implement since we would need to keep a history of the position in order to use the $\boldsymbol{x}(t+\Delta t)$ term.

We now simulate our solar system for 1 year using a time step $\Delta t = 0.01$ for each of the three integrators. We can see the trajectory of the moon for all three in fig. 6. The two symplectic algorithms manage to keep the moon bound to the earth unlike the simple Euler algorithm. The code for this plot can be found at `src/solar3.py`.

3.3 Long term stability

Finally we run the simulation for $t = 10$ years and plot the distance between the moon and earth. This is plotted in fig. 7. We see that eventually the symplectic Euler algorithm rockets the moon from the earth but the velocity Verlet algorithm manages to keep them bonded.

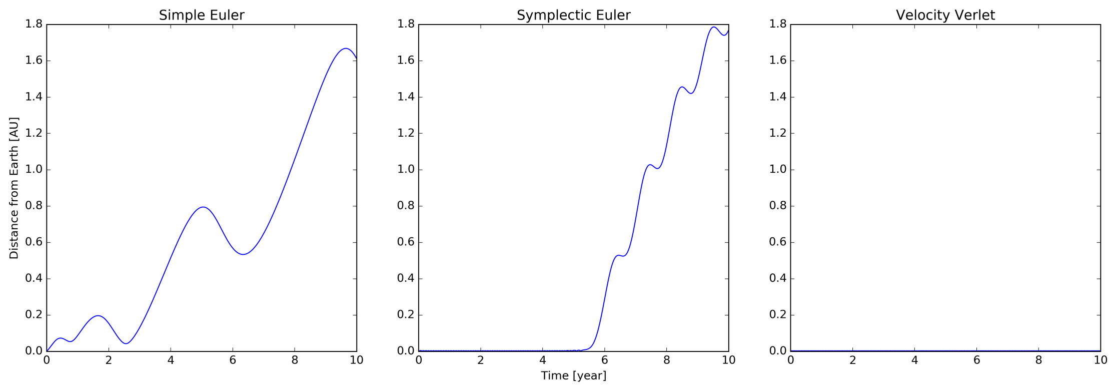

Figure 7: Distance between the Earth and moon using three different integrators and a time step of $\Delta t = 0.01$.