Worksheet 1: Integrators

David Beyer, Russell Kajouri, Keerthi Radhakrishnan

October 21, 2024
Institute for Computational Physics, University of Stuttgart

Contents

1 General Remarks

- Deadline for the report is **November 10, 2024, 00:00**
- On this worksheet, you can achieve a maximum of 20 points.
- The report should be written as though it would be read by a fellow student who listens to the lecture, but does not do the tutorials.
- To hand in your report, upload it to ILIAS and make sure to add your team member to your team. If there are any issues with this, please fall back to sending the reports via email
  - David (`dbeyer@icp.uni-stuttgart.de`)
  - Russell (`russell.kajouri@icp.uni-stuttgart.de`)
  - Keerthi (`keerthirk@icp.uni-stuttgart.de`)

- For the report itself, please use the PDF format (we will *not* accept MS Word doc/docx files!). Include graphs and images into the report.
- If the task is to write a program, please attach the source code of the program so that we can test it ourselves. Please name the program source file according to the exercise it belongs to, *e.g.* `ex_2_1.py` for exercise 2.1.
- The report should be 5–10 pages long. We recommend using LaTeX. A good template for a report is available in the ILIAS course
- The worksheets are to be solved in groups of two people. We will not accept hand-in exercises that only have a single name on it.

Testcases

We provide testcases for your programming tasks. That means if you follow the naming conventions and the implementation is correct you will get a passing test for the respective exercise. You can download the archive from the webpage. After downloading, you can unpack it:

```
> tar xzvf worksheet_01.tar.gz
```

Afterwards you should create a directory with the name "solutions":

```
> mkdir solutions
```

Change into the "solutions" directory and start working on your programming tasks, *e.g.* for exercise 2.1:

```
> cd solutions
> gedit ex_2_1.py
```

If you want to test whether you implementation is correct and complete, go into the directory named "test" in the archive you downloaded from the webpage and execute the respective test.

```
> cd ../test
> python3 test_ex_2_1.py
..
----------------------------------------------------------------------

Ran 2 tests in 0.001s

OK
```

If something went wrong if will look similar to the following output:

```
F.
======================================================================

FAIL: test_force (__main__.Tests)
----------------------------------------------------------------------

Traceback (most recent call last):
  File "test_ex_2_1.py", line 23, in test_force
    np.testing.assert_array_equal(ex_2_1.force(self.mass, self.
       gravity), self.f)
  File "/tikhome/kai/.local/lib/python3.6/site-packages/numpy/
     testing/_private/utils.py", line 918, in assert_array_equal
    verbose=verbose, header='Arrays␣are␣not␣equal')
  File "/tikhome/kai/.local/lib/python3.6/site-packages/numpy/
     testing/_private/utils.py", line 841, in assert_array_compare
    raise AssertionError(msg)
AssertionError:
Arrays are not equal
Implementation of the function 'force' seems wrong.
Mismatch: 50%
Max absolute difference: 0.1
Max relative difference: 0.00024271
 x: array([   0.  , -411.92])
 y: array([   0.  , -412.02])

----------------------------------------------------------------------

Ran 2 tests in 0.013s

FAILED (failures=1)
```

You have to look for an error message that gives you more information or you can look at the traceback directly:

- error message: `Implementation of the function 'force'seems wrong.`. In this case you can directly see what function did not work as expected.
- if there is not error message that gives insight you have to check the traceback: `np.testing.assert_array_equal(ex_2_1.force(self.mass, self.gravity), self.f)`. Here you just see that this assertion gave an error and you see that two arrays have been compared for equality.

2 Cannonball

2.1 Simulating a cannonball

In this exercise, you will simulate the trajectory of a cannonball in 2D until it hits the ground. In order to do so you have to solve Newton's equations of motion:

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \mathbf{v}(t) \tag{1}$$

$$\frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t} = \frac{\mathbf{F}(t)}{m}, \tag{2}$$

where $\mathbf{x}(t)$ is the position as a function of time t, m is the mass and $\mathbf{F}(t)$ is the force as a function of time.

At time $t = 0$, the cannonball (mass $m = 2.0\,\mathrm{kg}$) has a position of $\mathbf{x}(0) = \mathbf{0}$ and a velocity of $\mathbf{v}(0) = (60, 60)^\intercal\,\mathrm{m\,s^{-1}}$.

To simulate the cannonball, you will use the simple Euler scheme to propagate the position $\mathbf{x}(t)$ and velocity $\mathbf{v}(t)$ at time t to the time $t + \Delta t$ ($\Delta t = 0.1\,\mathrm{s}$):

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t \tag{3}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{\mathbf{F}(t)}{m}\Delta t \tag{4}$$

The Euler scheme can be derived by a first order Taylor expansion of Newton's equations (Eq. 1 and Eq. 2) in time yielding Eq. 3 and Eq. 4.

The force acting on the cannonball is gravity $\mathbf{F}(t) = (0, -mg)^\intercal$, where $g = 9.81\,\mathrm{m\,s^{-2}}$ is the acceleration due to gravity.

Task (3 points)
Write a Python program that simulates the cannonball until it hits the ground ($y \leq 0$) and plot the trajectory. Does the shape of the trajectory depend on the mass of the cannonball?

Hints on preparing the report

- Whenever you are asked to write a program, hand in the program source code together with the report. In the report, you can put excerpts of the central parts of the code.
- When a program should plot something, you should include the plot into the report.
- Explain what you see in the plot!

Hints for this task

- The program you start writing in this task will be successively extended in the course of this worksheet. Therefore it pays off to invest some time to write this program cleanly!
- Throughout the program, you will use NumPy for numerics and Matplotlib for plotting, therefore import them at the beginning:

```
import numpy as np
import matplotlib.pyplot as plt
```

- Model the position and velocity of the cannonball as 2d NumPy arrays:

```
x = np.array([0.0, 0.0])
```

- Implement a function `force(mass, gravity)` that returns the force (as a 2d NumPy array) acting on the cannonball.
- Implement a function `step_euler(x, v, dt, mass, gravity, f)` that performs a single time-step `dt` of the Euler scheme for position `x` and velocity `v` and force `f`. The function returns the new position `x` and velocity `v`.
- Beware that when you implement the Euler step, you should *first* update the position and then the velocity. If you do it the other way round, you have implemented the so-called *symplectic Euler algorithm*, which will be discussed later.
- Remember that NumPy can do element-wise vector operations, so that in many cases there is no need to loop over array elements. Furthermore, these element-wise operations are significantly faster than the loops. For example, assuming that `a` and `b` are NumPy arrays of the same shape, the following expressions are equivalent:

```
for i in range(N):
    a[i] += b[i]
# is the same as
a += b
```

- In the main program, implement a loop that calls the function `step_euler()` and stores the new position in the trajectory until the cannonball hits the ground.
- Store the positions at different times in the trajectory by appending them to a list of values

```
# start with an empty list for the trajectory
trajectory = []
# append a new value of x to the trajectory
trajectory.append(x.copy())
```

Note that when `x` is a NumPy array, it is necessary to use `x.copy()` so that the list stores the values, not a reference to the array. If `x` is a basic type (int, float, string), the call to `copy()` does not work.

- When the loop ends, make the trajectory a NumPy array and then plot the trajectory.

```
# transform the list into a NumPy array, which makes it easier
# to plot
trajectory = np.array(np.trajectory)
# Finally, plot the trajectory
plt.plot(trajectory[:,0], trajectory[:,1], '-')
# and show the graph
plt.xlabel(r"$x$ in m")
plt.ylabel(r"$y$ in m")
plt.show()
```

2.2 Influence of friction and wind

Now we will add the effect of aerodynamic friction and wind on the cannonball. We model friction as a non-conservative force of the form $F_{\text{fric}}(\mathbf{v}) = -\gamma(\mathbf{v} - \mathbf{v}_0)$. In our case, we assume that the friction coefficient is $\gamma = 0.1\,\mathrm{kg\,s^{-1}}$ and that the wind blows parallel to the ground with a wind speed v_w ($\mathbf{v}_0 = (v_w, 0)^\intercal\,\mathrm{m\,s^{-1}}$).

Task (3 points)

- Extend the program from the previous task to include the effects of aerodynamic friction. How is the employed model for friction called and is it realistic for a cannonball?
- Create a plot that compares the following three trajectories:
  - trajectory without friction
  - trajectory with friction but without wind ($v_w = 0$)
  - trajectory with friction and with strong wind ($v_w = -30\,\frac{\mathrm{m}}{\mathrm{s}}$)
- Create a plot with trajectories at various wind speeds v_w. In one of the trajectories, the ball should hit the ground close to the initial position. Roughly what wind speed v_w is needed for this to happen?

Hints

- Please use a new file for this exercise. If you want to reuse some parts of a previous exercise you can import it into the new file via, *e.g.* `import ex_2_1`

- Extend the function `force(mass, gravity)` such that it also takes the velocity `v`, the friction constant and the wind velocity as an argument. This can be achieved by recycling the force function of the previous task within a new function with the same name.
  E.g.:

```
import ex_2_1

def force(mass, gravity, v, gamma, v_0):
    return ex_2_1.force(mass, gravity) + ...
```

- Wrap the main loop into a function so that you can create several trajectories at different values of γ and v_w in a single program.
- You can add legends to the plots like this:

```
# make a plot with label "f(x)"
plt.plot(x, y, label="f(x)")
# make the labels visible
plt.legend()
# show the graph
plt.show()
```

3 Solar system

The goal of this exercise is to simulate a part of the solar system (Sun, Venus, Earth, the Moon, Mars, and Jupiter) in two dimensions, and to test the behavior of different integrators.

As in the previous tasks we want to solve Newton's equations (Eq. (1) and Eq. (2) of motion by numerical integration.

In contrast to the previous task, you will now have to simulate several "particles" (in this case planets and the sun, in the previous case a cannonball) that interact while there is no constant or frictional force. In the following, $\mathbf{x}_i$ denotes the position of the ith "particle" (likewise, the velocity $\mathbf{v}_i$ and acceleration $\mathbf{a}_i$).

The behavior of the solar system is governed by the gravitational force between any two "particles":

$$\mathbf{F}_{ij} = -Gm_i m_j \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3} \tag{5}$$

where $\mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$ is the distance between particle i and j, G is the gravitational constant, and m_i is the mass of particle i. The total force on any single particle is:

$$\mathbf{F}_i = \sum_{\substack{j=0 \\ i \neq j}}^{N} \mathbf{F}_{ij} \tag{6}$$

3.1 Simulating the solar system with the Euler scheme

The file `solar_system.pkl.gz`, which can be downloaded from the lecture home page, contains the names of the "particles", the initial positions, velocities, masses of a part of the solar system and the gravitational constant. The lengths are given in astronomical units au (*i.e.* the distance between earth and sun), the time in years, and the mass in units of the earth's mass.

Task (4 points)

- *Make a copy* of the program from the cannonball exercise and modify it to yield a program that simulates the solar system.
- Simulate the solar system for one year with a time-step of $\Delta t = 0.0001$.
- Create a plot that shows the trajectories of the different "particles".
- Perform the simulation for different time-steps $\Delta t \in \{0.0001, 0.001\}$ and plot the trajectory of the moon (particle number 2) in the rest frame of the earth (particle number 1). Are the trajectories satisfactory?
- Modern simulations handle up to a few billion particles. Assume that you would have to do a simulation with a large number of particles. What part of the code would use the most computing time?

Hints

- The file `solar_system.npz` can be read as follows:

```
import numpy as np
# load initial positions and masses from file
data = np.load('solar_system.npz')
names = data['names']
x_init = data['x_init']
v_init = data['v_init']
m = data['m']
g = data['g']
```

  Afterwards, `names` is a list of names of the planets that can be used in a plot to generate labels, `x_init` and `v_init` are the initial positions and velocities, `m` are the masses and `g` is the gravitational constant.

- As there are 6 "particles" now, the position vector `x` and the velocity vector `v` are now (2×6)-arrays, and the mass `m` is an array with 6 elements.

- The function you need to modify the most is the function `force(x_12, m_1, m_2, g)`, which is now required to compute the gravitational forces according to equation (6).

- In order to compute the forces on all particles implement a function `forces(x, m, g)` that loops over the positions `x` and returns the forces on all particles.

- When computing the forces, keep in mind Newton's third law, *i.e.* when particle j acts on particle i with the force $\mathbf{F}_{ij}$, particle i acts on particle j with the force $-\mathbf{F}_{ij}$.

3.2 Integrators

In the previous exercises, you have used the Euler scheme (*i.e.* a simple mathematical method to solve a initial value problem) to solve Newton's equations of motion. It is the simplest integrator one could think of. However, the errors of the scheme are pretty large, and also the algorithm is not *symplectic*.

Symplectic Euler algorithm

The simplest symplectic integrator is the *symplectic Euler algorithm*:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t \tag{7}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \Delta t)\Delta t \tag{8}$$

where $\mathbf{x}(t)$ are the positions and $\mathbf{a}(t) = \left(\frac{\mathbf{F}(t)}{m}\right)$ is the acceleration at time t. Compare the algorithm to the simple Euler scheme of equations (3) and (4).

Verlet algorithm

Another symplectic integrator is the *Verlet algorithm*, which has been derived in the lecture:

$$\mathbf{x}(t+\Delta t) = 2\,\mathbf{x}(t) - \mathbf{x}(t-\Delta t) + \mathbf{a}(t)\Delta t^2 + \mathcal{O}\left(\Delta t^4\right) \tag{9}$$

Velocity Verlet algorithm

An alternative to the Verlet algorithm is the *Velocity Verlet algorithm*:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{\mathbf{a}(t)}{2}\Delta t^2 + \mathcal{O}\left(\Delta t^4\right) \tag{10}$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{\mathbf{a}(t)+\mathbf{a}(t+\Delta t)}{2}\Delta t + \mathcal{O}\left(\Delta t^4\right). \tag{11}$$

Task (3 points)

- Derive the Velocity Verlet algorithm. To derive the position update, use a Taylor expansion of $\mathbf{x}(t+\Delta t)$ truncated after second order. To derive the velocity update, Taylor-expand $\mathbf{v}(t+\Delta t)$ up to the second order. To obtain an expression for $\mathrm{d}^2\mathbf{v}(t)/\mathrm{d}t^2$, use a Taylor expansion for $\mathrm{d}\mathbf{v}(t+\Delta t)/\mathrm{d}t$ truncated after the first order.
- Rearranging the equations of the Velocity Verlet algorithm, show that it is equivalent to the standard Verlet algorithm. First express $\mathbf{x}(t+\Delta t)$ using $\mathbf{x}$, $\mathbf{v}$ and $\mathbf{a}$ at $(t+\Delta t)$. in Equation (10). Then rearrange Equation (10) to express $\mathbf{x}(t)$. Add the two equations and then group velocity terms together. Put all velocity terms on one side of equation (11) and use them to plug them into your previous equation.

Implementation

Even if you know the equations of the algorithms, this does not mean that it is immediately obvious how to implement them correctly and efficiently and how to use them in practice.

For example, in the case of the Euler scheme (equations (3) and (4)), it is very simple to accidentally implement the symplectic Euler scheme instead. The following is pseudocode for a step of the Euler scheme:

1. $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v}\Delta t$

2. $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a}\Delta t$

If you simply exchange the order of operations, this becomes the symplectic Euler algorithm.

Another example for an algorithm that is tricky to use in a simulation is the Verlet algorithm.

Task (1 point)
Study equation (9). Why is it difficult to implement a simulation based on this equation in practice? What is missing?

Therefore, the Velocity Verlet algorithm is more commonly used in simulations. Unfortunately, implementing equations (10) and (11) has its pitfalls, too. Note that the algorithm requires both $\mathbf{a}(t)$ and $\mathbf{a}(t+\Delta t)$ in equation (11). As computing $\mathbf{a}(t)$ requires to compute the forces $\mathbf{F}(t)$, this would make it necessary to compute the forces twice. To avoid this, one can store not only the positions $\mathbf{x}$ and velocities $\mathbf{v}$ in a variable, but also the accelerations $\mathbf{a}$ and implement a time-step of the algorithm as follows:

1. Update positions as per equation (10), using the value of $\mathbf{a}$ stored in the previous time-step.
2. Perform the first half of the velocity update of equation (11): $\mathbf{v} \leftarrow \mathbf{v} + \frac{\mathbf{a}}{2}\Delta t$
3. Compute the new forces and update the acceleration: $\mathbf{a} \leftarrow \frac{\mathbf{F}}{m}$.
4. Perform the second half of the velocity update with the new acceleration $\mathbf{a}$.

Task (3 points)

- Implement the symplectic Euler algorithm and the Velocity Verlet algorithm in your simulation of the solar system.
- Run the simulation with a time-step of $\Delta t = 0.01$ for 1 year for the different integrators and plot the trajectory of the moon in the rest frame of the earth.

Hint If you have written the rest of the program cleanly, it should be enough to implement new functions `step_eulersym(x,v,dt)` and `step_vv(x,v,a,dt)` and to modify the main loop accordingly to call these functions to use a different integrator.

3.3 Long-term stability

An important property for Molecular Dynamics simulations is the *long-term stability*.

Task (3 points)

- During the simulation, measure the distance between earth and moon in every time-step.
- Run the simulation with a time-step of $\Delta t = 0.01$ for 20 years for the different integrators and plot the distance between earth and moon over time. Compare the results obtained with the different integrators!