Planetary motion simulator

Scientific Computing II, FYS2085

Olavi Kiuru

December 29, 2022

Contents

1 Introduction

The solar system consists of countless numbers of objects that interact with each other gravitationally and whose trajectories we want to calculate. The information of the trajectories can be used to calculate optimal launch parameters of space probes and rockets. With sufficiently accurate calculation, it is possible to calculate the trajectories of asteroids that may collide with the Earth in the future.

In this report, I present a numerical simulation of the n-body problem, implemented with the Fortran programming language. In the n-body problem, one is interested in calculating the trajectories of n celestial bodies that interact with each other through the gravitational force. It is a typical example of a problem in classical mechanics that is not in general solvable analytically. The problem has an analytical solution only for $n = 2$, as well as certain specific configurations for larger n. Therefore, numerical simulations are required to properly predict the time evolution of such systems.

2 Methods

In the calculations, I have used the velocity Verlet algorithm [1]. The algorithm consists of the following steps:

1. Read initial values from file
2. $x_{i+1} = x_i + v_i \Delta t + \frac{1}{2} a_i \left(\Delta t\right)^2$
3. $a_{i+1} = \frac{F(x_{i+1})}{m}$
4. $v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t$
5. If $i \neq n_{\text{iter}}$: go to step 2, $i = i + 1$. Else: continue.
6. End calculation

where x denotes position, v denotes velocity, a denotes acceleration, F denotes force, Δt denotes the time step of one iteration, n_{iter} denotes the maximum number of iterations and i denotes the current iteration. The force between two objects is calculated with Newton's law of universal gravitation

$$F(r) = -\frac{G m_1 m_2}{r^2} \tag{1}$$

where G denotes the gravitational constant, m_i denotes the mass of the ith object, and r denotes the distance between the two objects.

3 Implementation

In this section, I will describe the structure of the program. Furthermore, I will give instructions on how to use the program. The program has a time complexity of $\mathcal{O}(n^2)$ due to the force calculation loop.

3.1 Structure of the program

In the following paragraphs, I will describe the contents of each module. The structure of the program is shown in Fig. 1.

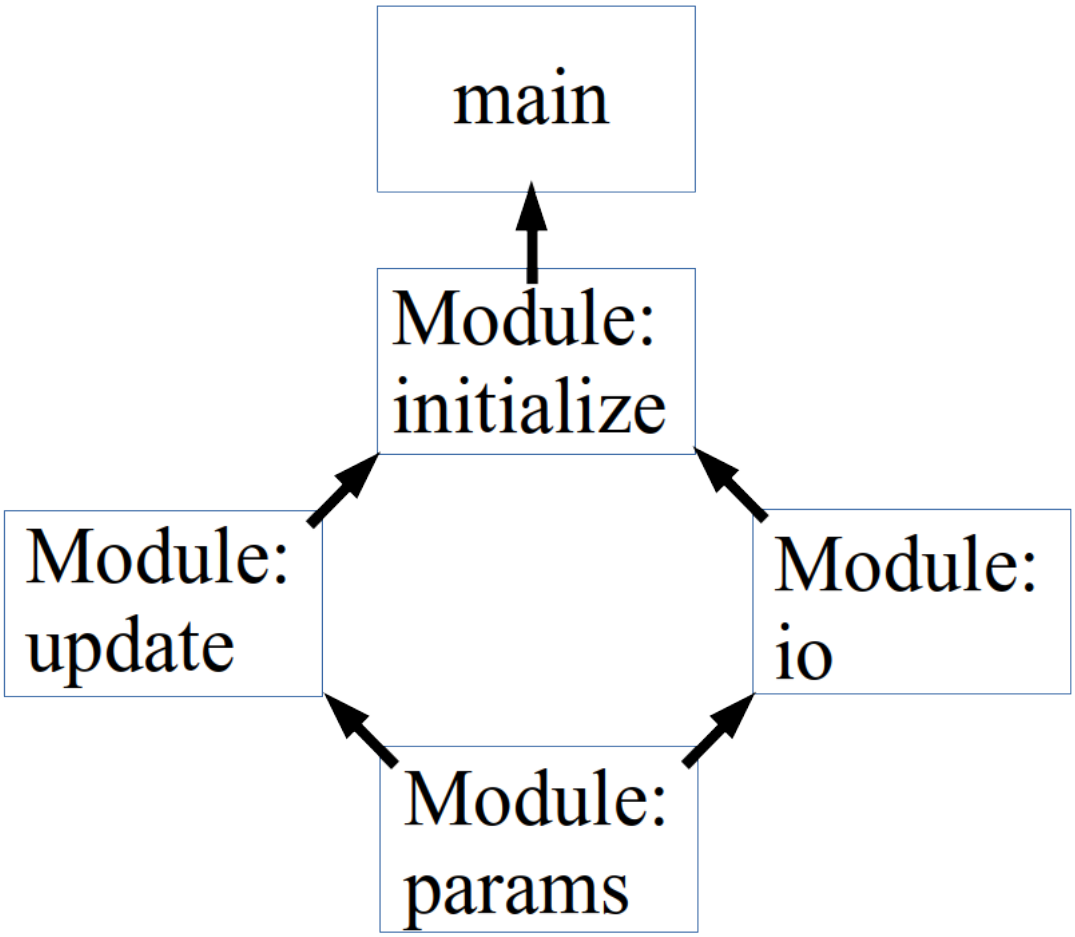

Figure 1: Structure of the program. The arrows denote the dependencies of each part of the program.

In module `params`, defined in the file `params_mod.f90`, I have defined all the numerical constants used in the program. The units that I have used are au for

distance, year for time and solar mass $M_\odot$ for mass. In these units, the gravitational constant $G = 4\pi^2$ au^3 year^{-2} $M_\odot^{-1}$. I have selected to use double precision for the real numbers in this program.

In module `update`, defined in the file `update_mod.f90`, I have defined functions and subroutines related to updating the values of the variables. The functions `update_position`, `update_velocity`, `update_acceleration`, and `update_force` update the position, velocity, acceleration and force arrays, respectively, using the velocity Verlet algorithm described in Sec. 2. The subroutine `update_all` combines all the aforementioned update functions into one subroutine that the main function can call.

In module `io`, defined in the file `io_mod.f90`, I have defined four subroutines related to input and output streams. The subroutine `print_status` prints the current position of all objects, as well as the number of objects in the simulation, the number of iteration steps completed, and the number of data points written to the output file, to the standard output stream. The subroutine `input_error` prints out the error message given when there is an error in reading the input file. It gives instructions on how the input file should be formatted. The subroutine `init_ovito` initializes the `output.xyz` file that the outputs are also written to. The subroutine `print_ovito` writes the current position to the `output.xyz` file in the `.xyz` format.

In module `initialize`, defined in the file `initialize_mod.f90`, I have defined two subroutines that initialize the values of the variables in the program. The subroutine `initial_values` reads from the input file given by the user the initial values of position and velocity, as well as the number of objects, their masses, the size of the time step used in the simulation, the number of iterations, and how often data should be printed to the output files and to the standard output stream. The proper format of the input file is described in Sec. 3.2. The subroutine `initialize_all` calls the previously described `initial_values` subroutine and uses the information read from the file to calculate the force and acceleration at the start using the functions defined in the `update` module.

The program `main`, defined in the file `main.f90`, uses all the modules described above to calculate the positions of all the objects at every time step. The positions, velocities, accelerations, forces and masses are implemented as allocatable arrays of real numbers. For the vector quantities, three different arrays are implemented: one for the x, y, and z-components, respectively. The size of the arrays is given in the input file.

3.2 Usage instructions

Here, I will give instructions on how to compile and run the program. I will also describe the required format of the input files.

To compile the program, you can use the command `make` in the `src` folder. This creates the executable `simulation`. More detailed instructions for compiling the program, as well as other options if the `make` command does not work, can be found in the `README.md` file in the `src` folder.

Once you have created the executable, you can move to the `run` folder. Here, you run the program with the command `../src/simulation input-file`, where `input-file` is the input file that you want to use. These same instructions can also be found in the `README.md` file in the `run` folder.

The format of the input file is the following:

```
n
x1 x2 ... xn
y1 y2 ... yn
z1 z2 ... zn
vx1 vx2 ... vxn
vy1 vy2 ... vyn
vz1 vz2 ... vzn
m1 m2 ... mn
dt
n_iter
n_print
n_output
```

where `n` is the number of objects, `xn`, `yn`, and `zn` are the initial coordinates of the nth object, `vxn`, `vyn`, and `vzn` are the initial velocities of the nth object, `mn` is the mass of the nth object, `dt` is the size of the time step in the simulation, `n_iter` is the amount of iterations in the simulation, the positions of the objects are printed to the standard output stream every `n_print` iterations, and the positions of the objects are printed to the output files every `n_output` iterations. For the units used, see the discussion in Sec. 3.1 about the `params` module. All input variables starting with an n should be given as integers, while the rest are real numbers. The positions are printed to the standard output stream in the following format:

```
Number of objects: n
Number of steps completed: m
Number of values written to file: k
Position of objects:
x1 y1 z1
x2 y2 z2
...
xn yn zn
```

The format of the output files is discussed below.

The program outputs two files, `output.dat` and `output.xyz`. The `output.dat` contains the positions of the objects in the following format:

```
x1 x2 ... xn
y1 y2 ... yn
z1 z2 ... zn
```

```
x1 x2 ... xn
y1 y2 ... yn
z1 z2 ... zn
...
x1 x2 ... xn
y1 y2 ... yn
z1 z2 ... zn
```

where the number after x, y, or z denotes the number of the object and every three rows the simulation proceeds a certain time step declared in the input file. This output file can be used for plotting 2D figures of the simulation with the python script `plot.py` found in the `run` folder.

The `output.xyz` file consists of blocks with the following format:

```
n
Time: elapsed time
label1 x1 y1 z1
label2 x2 y2 z2
...
labeln xn yn zn
```

where n is the number of objects, elapsed time is replaced with the time in years since the simulation started, and the labels are capital letters. Note that the labels are unique for a maximum of 27 objects, after which they start repeating. The `output.xyz` file can be used in an appropriate program, e.g. Ovito [2], to make 3D animations of the simulations. The labels are used by Ovito to differentiate the objects from each other.

4 Results

In this section, the results of my simulation are presented.

4.1 Jupiter orbiting the Sun

The input file `jupiter-sun-input.dat` includes the input data used for this part. The initial velocity of Jupiter is chosen as $v = 2\pi r_{\text{Jup}}/T_{\text{Jup}}$, where r_{Jup} is the orbital radius of Jupiter and T_{Jup} is the orbital period of Jupiter. The initial velocity of the Sun is chosen such that the total momentum of the system is zero. I started off with the time step $dt = 10^{-3}$ years. With this time step I got the period of Jupiter to be $T_{\text{Jup}} \approx 11.95$ years. This is visualized in Fig. 2a. With a time step of $dt = 10^{-2}$ years, Jupiter still moves almost a full period, but with a time step of $dt = 10^{-1}$ years, Jupiter only completes approximately three quarters of the orbit. At this point the orbit is also no longer constant as Jupiter starts moving away from the Sun. Thus, to get a reasonable approximation of the period one should

use a time step of at least $dt = 10^{-2}$ years, but preferably 10^{-3} years or smaller. The orbits of Jupiter with time steps of $dt = 10^{-3}$ and 10^{-2} years are shown in Figs. 2a and 2b, respectively.

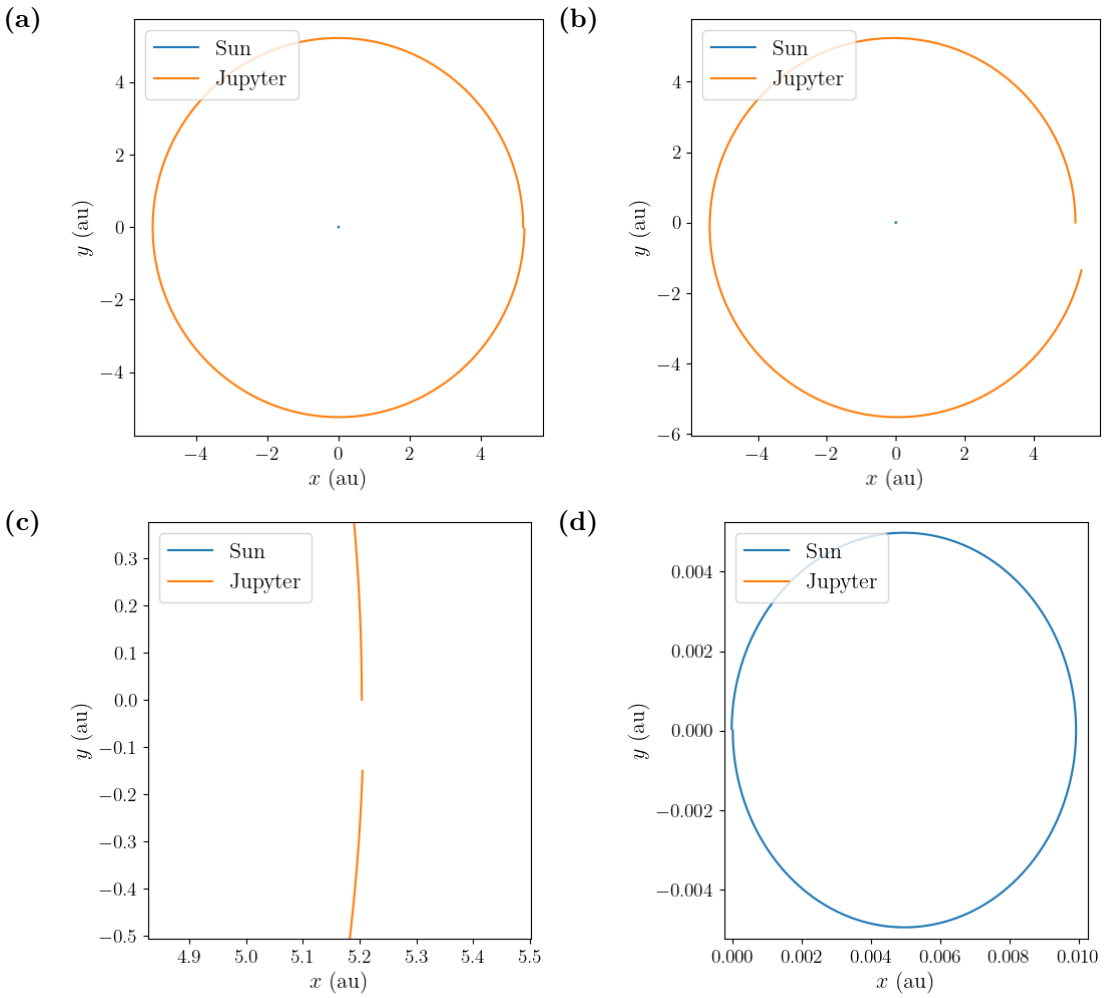

Figure 2: a. Orbit of Jupiter around the sun with the time step $dt = 10^{-3}$ years. Duration of simulation: 11.95 years b. Orbit of Jupiter around the sun with the time step $dt = 10^{-2}$ years. Duration of simulation: 12 years. c. Orbit of Jupiter simulated for the duration of the actual period of Jupiter, 11.86 years, and a time step of $dt = 10^{-4}$ years. The simulation does not complete a full orbit. d. Orbit of the Sun from the simulation described in part c.

The actual period of Jupiter is approximately 11.86 years. Simulating the system for that long with a time step of $dt = 10^{-4}$ years, Jupiter is approximately 0.15 au from completing one period in the simulation. The Jupiter in the simulation

thus has a longer period than the real Jupiter. This is shown in Fig. 2c. In the same simulation we see that the Sun has also moved approximately one period so the period of the Sun equals the period of Jupiter. The orbit of the Sun has a radius of approximately 0.005 au, as shown in Fig. 2d.

4.2 Length of a month and a year

To calculate the length of a month, I have used the input file `moon-earth-input.dat`. As above, I have made the total momentum of the system zero. With a time step of $dt = 10^{-6}$ years, I get the orbital period of the Moon to be 22.83 days. This is smaller than the correct orbital period of 27.32 days and gives an error of over 15 %. This is probably due to some error in the initial conditions. The small masses of the objects also may cause some errors in the calculations. The orbit of the Moon is shown in Fig. 3a.

Similarly to the above calculation, I calculated the length of a year by the orbital period of the Earth, orbiting around the Sun. The result was 367.08 days with a time step of $dt = 10^{-4}$ years. This estimation is within 0.5 % of the actual value of 365.25 days. The orbit of the Earth is shown in Fig. 3b.

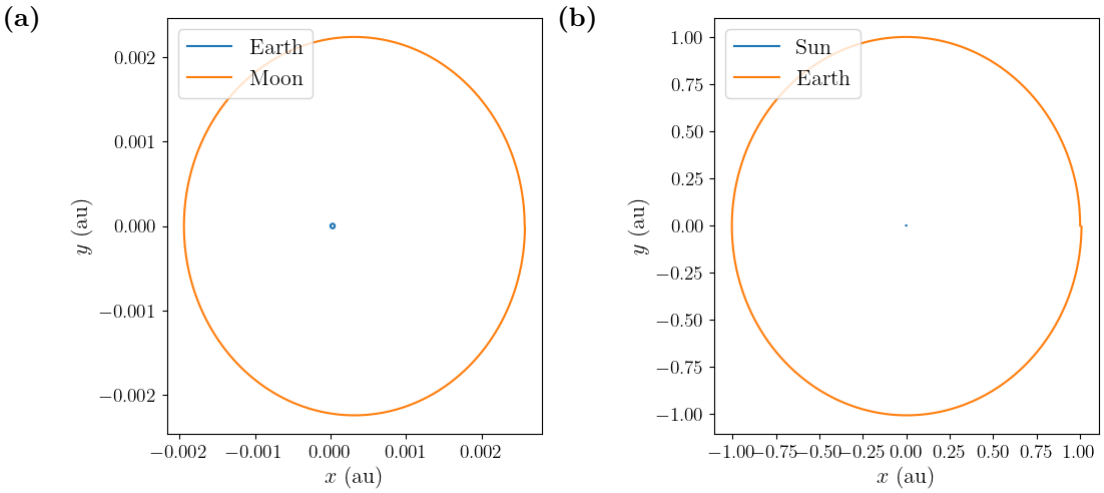

Figure 3: a. Orbit of the Moon around Earth with a time step of $dt = 10^{-6}$ years and simulation time 22.83 days (0.0625 years). b. Orbit of the Earth around the Sun with a time step of $dt = 10^{-4}$ years and a simulation time of 367.08 days (1.005 years).

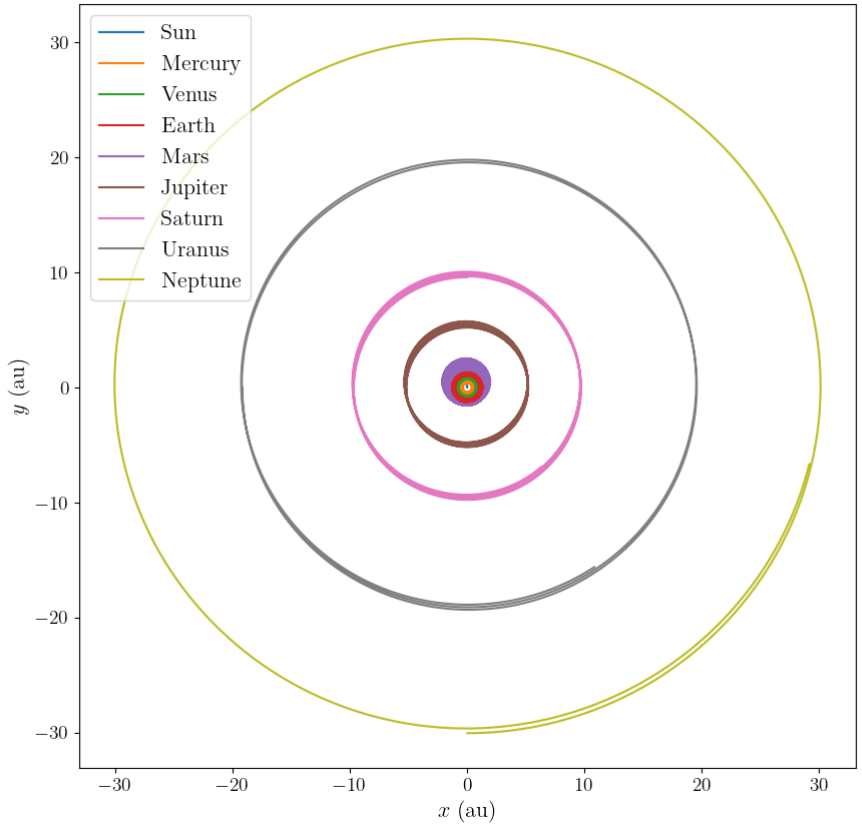

Figure 4: Full solar system simulated with a time step of $dt = 10^{-4}$ years for a period of 200 years. The whole system is moving slowly up and to the left because the total momentum of the system is not zero.

4.3 The solar system

To simulate the entire solar system I placed the planets at their average distances from the Sun with a 90° rotation between each subsequent planet. The velocities were calculated as before, but with some minor adjustments to make the system stable. The initialization can be found in the file `solar-system-input.dat`. A plot of the solar system can be seen in Fig. 4. The file `solar-system.gif` in the included `run` folder, shows an animation of the whole solar system. Note that the size of the planets and the Sun are not to scale.

We use this simulation of the solar system to calculate the period of Mercury.

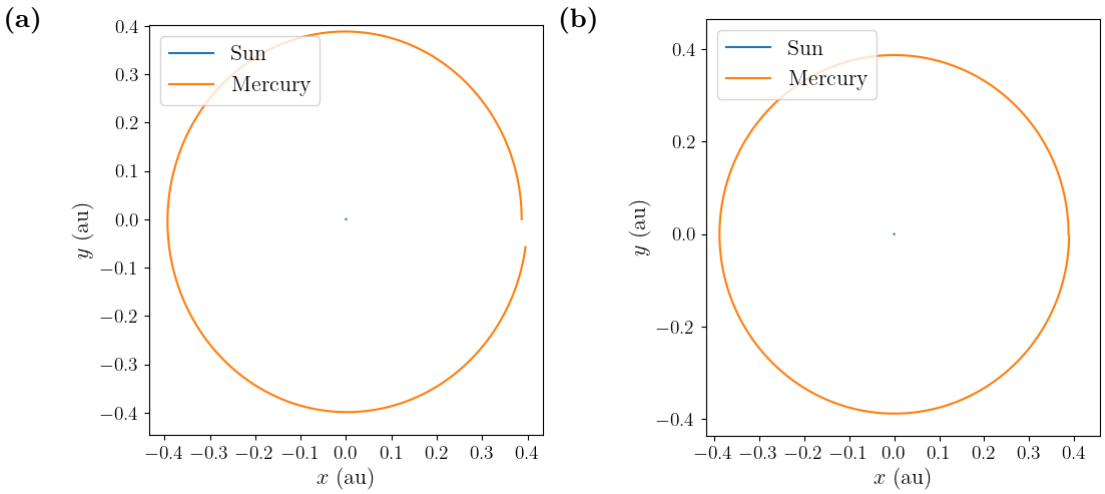

Figure 5: a. Orbit of Mercury with a time step of $dt = 10^{-4}$ years. b. Orbit of Mercury with a time step of $dt = 10^{-5}$ years. Both simulations were ran for 0.241 years.

The actual period of Mercury is $T_{\text{Mer}} \approx 0.241$ years. With a time step of $dt = 10^{-3}$ the simulation gives a period of 0.3 years. With a time step of 10^{-4} years the simulated period is much closer to the real period and with a time step of 10^{-5} we get the actual period in the simulation. This is shown in Fig. 5.

5 Conclusions

In this report, I presented a numerical simulation that calculates the orbits of objects bound together by the gravitational force. The results I got were in good agreement with the correct values. For example, the program predicted the length of a year within 0.5 % of the actual value. However, some of the calculation were not so accurate and especially the orbital period of the Moon had an error of over 15 %.

The time complexity of the program cannot be improved from $\mathcal{O}(n^2)$ unless a completely different approach is chosen, due to the nature of Newton's universal law of gravitation. However, the time of the force calculation could be cut in half by only calculating all the pairs of objects once instead of calculating them twice, once for each member of the pair.

Currently, not all of the program's interactions with input and output streams are handled by the `io` module. This is something that could be fixed to make the code more consistent and thus, easier to maintain. In the current version of the program, the length of the time step is a constant read from the input file. Thus,

it is not needed to give it as an argument to the update functions, as is currently done, and it could instead be defined in the `params` module. However, this is not done in case future optimisation of the code would require updating the length of the time step during the simulation.

References

[1] WC Swope et al. "A computer-simulation method for the calculation of equilibrium-constants for the formation of physical clusters of molecules - application to small water clusters". In: *Journal of Chemical Physics* 76.1 (1982), pp. 637–649. ISSN: 0021-9606. DOI: 10.1063/1.442716.

[2] Alexander Stukowski. "Visualization and analysis of atomistic simulation data with OVITO-the Open Visualization Tool". In: *Modelling and Simulation in Materials Science and Engineering* 18.1 (Jan. 2010). ISSN: 0965-0393. DOI: {10.1088/0965-0393/18/1/015012}.