Numerical solution of Newtonian N-body problems by a Fortran-implemented Velocity Verlet Algorithm

Elias Ankerhold

University of Helsinki

December 21, 2021

The Velocity Verlet Algorithm (VVA) is implemented to provide a framework in which N celestial bodies within each others gravity fields can be simulated. Relevant forces are reduced to a purely mechanical Newtonian picture, disregarding general relativity corrections. All computational work is done in Fortran 90, while visualization tools are written in Python 3.9. After a thorough investigation of the general physical phenomena at work and a description of the technical details of the implementation, simulation results of various objects in our solar system are presented. The simulation yields excellent approximations especially in two-body systems. In up to 10-body simulations, the vast majority of orbital parameters are still approximated with less than 1% relative error. Bodies given initial parameters close to the limits of parameter space – whether in spatial or time regime – however exhibit some instabilities and greater inaccuracies. Finally, easily implementable major improvements of the simulation itself and its data analysis options are proposed.

1 Introduction and Methods

1.1 Physical basis

The physical basis of celestial N-body systems is laid by Newtonian mechanics. Even though this does, of course, not cover the whole physical complexity of the processes taking place in the solar system, the numerical errors of the simulation itself are expected to outweigh actual physical inaccuracies of the Newtonian model. The most prominent observation of general relativity correction in our solar system is the procession of the periapsis of Mercury, which moves by 574.1 arcsecs per 36,500 days [1]. As depicted in Table 2, the simulation does not provide valid results for Mercury's orbit when using time spans of similar order of magnitude while including all other planets. Thus, the physical inadequacies of the Newtonian model are not relevant here and can be neglected without further ado.
The simulation is therefore based on Newton's universal law of gravitation, describing the gravitational force $\boldsymbol{F}_{ij}$ between two objects, depending on their mass $m_{i/j}$, their positions $\boldsymbol{r}_{i/j}$ and the gravitational constant $\gamma = 6.67430 \cdot 10^{-11} m^3 kg^{-1} s^{-2}$ [5]:

$$\boldsymbol{F}_{ij}\left(\boldsymbol{r}_i, \boldsymbol{r}_j\right) = -\gamma \frac{m_i m_j}{\left|\boldsymbol{r}_i - \boldsymbol{r}_j\right|^3}\left(\boldsymbol{r}_i - \boldsymbol{r}_j\right) \tag{1}$$

As other forces are negligible, Equation 1 includes all physical phenomena needed here. To compute the total force acting on object i in a system of N bodies, all two-body interactions are added:

$$\boldsymbol{F}_{i,\mathrm{tot}}\left(\boldsymbol{r}_i\right) = \sum_{j \neq i}^{N} \boldsymbol{F}_{ij}\left(\boldsymbol{r}_i, \boldsymbol{r}_j\right) \tag{2}$$

Since it is conserved in every isolated physical system, the total energy is a suitable quantity to provide a way of checking the validity of the simulation. The system deals with purely mechanical processes, therefore the total energy of an N body system simply computes as the sum of the total potential energy E_{pot} and total kinetic energy E_{kin}:

$$H = \underbrace{\frac{1}{2}\sum_{i}^{N}\sum_{j \neq i}^{N} \gamma \frac{m_i m_j}{\boldsymbol{r}_{ij}^2}}_{E_{pot}} + \underbrace{\frac{1}{2}\sum_{i}^{N} m_i \dot{\boldsymbol{r}}_i^2}_{E_{kin}} \tag{3}$$

Finally, a viable coordinate system has to be defined. Although there is a whole zoo of different coordinate systems used in astronomy, most of them being a variation of spherical coordinates, this simulation uses a cartesian coordinate system centered at the initial position of the system's central body – that is, in most cases, the sun.
When simulating real problems, the question of initial values arises. Opposed to information on the masses of celestial bodies, which are readily available in literature, positions and velocities of the planets are not easily accessible and therefore demand further attention. As described, the algorithm uses a cartesian coordinate system. In astronomy however, several coordinate systems are used, none of which are cartesian, equipped with different origins, depending on the scope of the problem they should describe. An attempt was made to transform real-time literature positions and velocities of the planets in the solar system into the cartesian coordinate system used in the simulation. However, due to the complexity of said transformation and the limited availability of trusted sources, this effort was abandoned.
Instead, all planetary bodies are initially placed along the x-axis and given a velocity vector parallel to the y-axis, essentially reducing the problem to a two-dimensional system (with the exception of the moon, whose initial parameters are more complicated). Their distance to the origin r is set

to the respective periapsis of their real orbits. If the length of the orbit's semi-major axis a is known, the absolute velocity can then be computed according to [2]:

$$v = \sqrt{\mu\left(\frac{2}{r} - \frac{1}{a}\right)} \tag{4}$$

Where the standard gravitational parameter $\mu = \gamma m$ is calculated from the gravitational constant γ and the mass of the central body m, yielding the initial values show in Table 1.

1.2 Computation model

The numerical computation is performed by an implementation of the Velocity Verlet Algoritm (VVA), originally published by Swope et al. [4]. The VVA integrates the general equations of Newtonian motion using a finite time step Δt. In iteration step i, the position, velocity and acceleration of object k in step $i+1$ is computed according to:

$$\begin{aligned}
\ddot{\boldsymbol{r}}_k^{(i)} &= \frac{\boldsymbol{F}_{k,\text{tot}}\left(\boldsymbol{r}_k^{(i)}\right)}{m_k} \\
\boldsymbol{r}_k^{(i+1)} &= \boldsymbol{r}_k^{(i)} + \dot{\boldsymbol{r}}_k^{(i)}\Delta t + \frac{1}{2}\ddot{\boldsymbol{r}}_k^{(i)}\Delta t^2 && (5)\\
\ddot{\boldsymbol{r}}_k^{(i+1)} &= \frac{\boldsymbol{F}_{k,\text{tot}}\left(\boldsymbol{r}_k^{(i+1)}\right)}{m_k} \\
\dot{\boldsymbol{r}}_k^{(i+1)} &= \dot{\boldsymbol{r}}_k^{(i)} + \frac{1}{2}\left(\ddot{\boldsymbol{r}}_k^{(i)} + \ddot{\boldsymbol{r}}_k^{(i+1)}\right)\Delta t && (6)
\end{aligned}$$

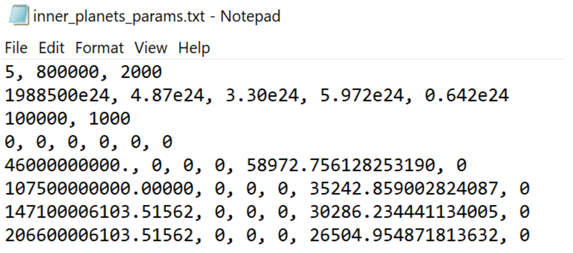
2 Implementation

The code is divided into three modules, successively depending on each other. The basis is formed by the `vectors` module which provides a three-dimensional vector data type along with operator and assignment overloading as well as useful vector-specific operations such as the euclidean norm and the cross product. The `io_manager` depends on the `vectors` module and includes three types of procedures: (1) `prettyprint` subroutines for various input arguments which print data to the terminal in an organized and readable way, (2) file handling procedures reading and creating files and directories and saving data as well as (3) generic user communication subroutines through which most terminal messages are managed, thereby providing an easy possibility to add a logging functionality. Finally, the core of the simulation is written in the `plan_sim` module, which houses the actual VVA implementation as well as various helper procedures initializing the simulation, calculating forces, energies, distances and reordering masses.

2.1 Directory structure and initial file format

In order to run error free, the program relies on the directory structure shown in Figure 2.
Additionally, the input file storing initial values and simulation parameters has to comply strictly with the following formatting specifications:

`number of bodies`, `number of steps`, `duration`
`mass`$_1$, `mass`$_2$, ...
`k`, `m`
$x_1^{(0)}, y_1^{(0)}, z_1^{(0)}, \dot{x}_1^{(0)}, \dot{y}_1^{(0)}, \dot{z}_1^{(0)}$
$x_2^{(0)}, y_2^{(0)}, z_2^{(0)}, \dot{x}_2^{(0)}, \dot{y}_2^{(0)}, \dot{z}_2^{(0)}$
...


inner_planets_params.txt - Notepad
File Edit Format View Help

```
5, 800000, 2000
1988500e24, 4.87e24, 3.30e24, 5.972e24, 0.642e24
100000, 1000
0, 0, 0, 0, 0, 0
46000000000., 0, 0, 0, 58972.756128253190, 0
107500000000.00000, 0, 0, 0, 35242.859002824087, 0
147100006103.51562, 0, 0, 0, 30286.234441134005, 0
206600006103.51562, 0, 0, 0, 26504.954871813632, 0
```

Figure 1: Example input file storing general simulation parameters and initial values of the inner planets. All planets are placed along the x-axis and given an initial velocity parallel to the y-axis.

The duration has to be given in earth days, all other values in SI base units. Simulation info will be printed every k step, every mth value will be saved to file. An example input file is shown in Figure 1.
The program outputs to an automatically created subdirectory of `data` which is named as the epoch integer representation of its creation time. For each body simulated, a separate output file `body_XX.txt` is created. This file stores data in columns separated by spaces. The column headers (not exported) are: x, y, z, $\dot{x}$, $\dot{y}$, $\dot{z}$, $\ddot{x}$, $\ddot{y}$, $\ddot{z}$, F_x, F_y, F_z, time. All exports are done in SI-base units: m, $\frac{m}{s}$, $\frac{m}{s^2}$, $\frac{kgm}{s^2}$, s. The file `energies.txt` stores columns E_{pot}, E_{kin}, time in SI base units $\frac{kgm^2}{s^2}$ and s. An example structure of output files is shown in Figure 2.

2.2 Vector data type

Data structures

In the `vectors` module, three global parameters are defined. Next to the used kind values for real and integer variables, `rk` and `ik` respectively, π is defined by $\pi = 4\arctan(1)$.
The derived type `vec`, representing a three dimensional cartesian vector, is defined by using three real components `x, y, z`. Operators `+, -, *, =` are overloaded with procedures described below.

Procedures

The functions `vadd` and `vminus` implement common vector addition and subtraction. The scalar product of two vectors $\boldsymbol{a}\cdot\boldsymbol{b} = a_i b_i$ and the induced 2-norm $|\boldsymbol{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$ are computed by `scalarprod` and `norm`. Scalar-vector and vector-scalar multiplication are overloaded using `scalar_mult_a` and `scalar_mult_b`, the

```
/
Makefile
├── run/
│   ├── data/
│   │   └── 1640048872/
│   │       ├── body_01.txt
│   │       ├── body_02.txt
│   │       └── energies.txt
│   ├── python_exports/
│   └── my_simulation_params.txt
└── src/
    ├── fortran_sim/
    │   ├── coordinate_conversion.f90
    │   ├── i_o_framework.f90
    │   ├── init_params.out
    │   ├── plan_sim_framework.f90
    │   ├── simulation_starting.f90
    │   ├── start_sim.out
    │   └── vector_framework.f90
    └── python_visualization/
        ├── sim_visualization.py
        └── visu_framework.py
```

Figure 2: File tree of the program. While directories/files marked black are needed in this exact structure, gray ones could be moved and adjusted by the user.

cross product $(\boldsymbol{a} \times \boldsymbol{b})_i = \epsilon_{ijk} a_j b_k$ is implemented in `cross_product`.
Three types of assignments are overloaded as well: `vec_to_vec` copying a vector to another, `scalar_to_vec` assigning the same value to all vector components and `arr_to_vec` copying the entries of a three dimensional real array into a `vec` variable.

2.3 I/O module

Data structures

There are only two global variables in `io_manager`: A single character `path_sep` which can be adjusted to \or /, depending on the system's path separator symbol and a 500 character long string `temp_dump` serving as a temporary write destination to pre-format strings before passing them on.

Procedures

The three basic subroutines `prettyprint_vec_mat`, `prettyprint_vec_arr`, `prettyprint_vec_single` print a matrix of vectors, an array of vectors and a single vector to screen in a readable way. The same functionality for real matrices is provided by `prettyprint_real`.

The file handling procedures are divided into reading and writing subroutines. Reading the initial values from file is done in two steps: First, `get_arr_dims` only reads the number of bodies and steps, which allows the corresponding data structures in `plan_sim` to be allocated. Secondly, `read_sim_params` opens the file again and reads the rest of the simulation parameters and initial values. During this process, the parameters are checked for obvious errors about which the user is warned or the program terminated.

At the beginning of the simulation, a subdirectory of `data` is created by `prepare_subdir`, using the current epoch time as the directory name to prevent naming conflicts. The `data` directory has to be present as show in Figure 2. There are two ways of saving the simulation result to file. Actively used in the current version is, because of the project requirement, subroutine `save_sim_step` which saves the last object positions, velocities, accelerations, forces as well as potential energy, kinetic energy and the time. This subroutine is called every k iteration step and appends the last values to an existing file or creates a new one if none exists already. The second, more efficient way `export_all_sim_results` would not be called during the computation of VVA, but after the simulation is finished. It also saves only every k step to file, but would run orders of magnitudes faster than `save_sim_step` because it only needs to open the file once.

Program-user communication mainly runs through the subroutine `print_sim_info` which is called every m step and prints the current step, total steps, the steps saved to file and the number of bodies simulated as well as their current positions to the terminal. There are three subroutines `error`, `warning` and `info`, which are called in according events and print the passed message to the terminal. Although there is no direct advantage of passing user messages through one of these subroutines yet, they offer easy implementation of a log-to-file functionality, if needed in the future.

2.4 Simulation core

Without exceptions, all values are internally saved and used in SI base units. Even if at some point, user information is displayed in different units, internally, only SI base units are used. Consequentially, all data import and export is done in SI units, except for the simulation running time, which is read in earth days for convenience. The exported time stamps are in seconds however.

Data structures

Two constants are defined in `plan_sim`: The gravitational constant `g = 6.67430e-11` as well as `sec_per_day = 24 * 3600` simply stores how many seconds are in an earth day.
General simulation parameters are loaded into `integer n_step`, `n_body`, `f_step` and `s_step` store the number of simulation steps, number of bodies, k that prints info to terminal every kth step and saves every mth step to file, respectively. The `real step` and `d_dur` are used to calculate the step size in seconds from the total running time provided in days.
Vectorial simulation results of positions, velocities, accelerations and forces are stored globally in allocatable matrices of vectors `pos`, `vel`, `acc` and `forces`. In an N-body simulation with n steps, their dimensions are (N, n). Scalar values are

also saved to global, allocatable `real` arrays `masses`, `e_pot` and `e_kin`.

Procedures

To allocate data structures, compute the step size and prepare simulation parameters, subroutine `initialize_sim` has to be called before starting any actual computation. Together with the module's main subroutine `vel_verlet`, it is the only module procedure that accesses global data structures. All other procedures are in fact functions and only work with local copies of the arguments passed to them.

`vel_verlet` manages the simulation and calls helper functions when needed. First, a mass matrix of the form $m_{ij} = m_i m_j$ and $m_{ii} = 0$ is pre-calculated by `get_mass_matrix`. This saves computation time as less operations have to be performed during each iteration step. Based on the initial positions, `dist_vecs` computes an $N \times N$ vector array of all distance vectors between all bodies $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$. Passing this array to (a) `scal_dists_inv` and (b) `force_vecs` returns (a) a scalar $N \times N$ array with entries $r_{ij} = \frac{1}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}$ and $r_{ii} = 0$ as well as (b) an N vector array of resulting forces $\boldsymbol{F}_{i,\text{tot}}(\boldsymbol{r}_i)$. Scalar energies are calculated by `kin_en` and `pot_en`. Finally, the initial positions and velocities read from file and the resulting values computed from them are written into the first entry of the respective global variables.

For the remaining iteration steps, the subsubroutine `verlet_step` of subroutine `vel_verlet`, where the actual VVA is implemented, is called in a `do` loop. `verlet_step` performs a single iteration of the VVA by using Equation 5 and the functions mentioned above. The values are written directly into the global data structures. Using `if-modulo` constructs, `vel_verlet` checks when information should be printed to the terminal and values are to be saved to file. This is done by calling the according subroutines from `io_manager`. Even though required by the project's description, this is highly inefficient, since the two `if` statements slow down computation a lot while only being needed rarely.

2.5 Compilation

The program is equipped with a makefile which compiles the main simulation program. Simply using the `"make"` command in the main directory of the program package will automatically create `start_sim.out` in `src/fortran_sim/`. Alternatively, the program can be compiled using `"gfortran vector_framework.f90 i_o_framework.f90 plan_sim_framework.f90 simulation_starting.f90 -o start_sim.out"` in `src/fortran_sim/`.
To compile the initial parameter helper `coordinate_conversion.f90`, the command `"gfortran vector_framework.f90 coordinate_conversion.f90 -o init_params.out"` has to be used in `src/fortran_sim/`.

Body	Mass $[10^{24}\ kg]$	Periapsis $[10^9\ m]$	Orbital Velocity $[10^3\ \frac{m}{s}]$
Sun	1988500	0	0
Mercury	0.330	46.0	58.97
Venus	4.87	107.5	35.24
Earth	5.97	147.1	30.29
Mars	0.64	206.6	26.50
Jupiter	1898	740.5	13.71
Saturn	568	1352.6	10.16
Uranus	86.8	2741.3	7.11
Neptune	102	4444.5	5.50
Moon	0.073	0.363*	1.08*

Table 1: Initial parameters for bodies of the solar system. Masses and periapsis data according to [6], velocity calculated by Equation 4 using [7] and [3]. **Values of the lunar orbit around the earth.*

2.6 Execution

To execute the main program and start the simulation, the input file has to be provided as a command line argument. Thus, in a file structure as in Figure 2, the execution command is: `"./start_sim.out ../../my_simulation_params.txt"` executed in `src/fortran_sim/`.
The helper program is executed by running `"./init_params.out"`.

3 Results

3.1 Two-body problems

3.1.1 Sun and Jupiter

Using the initial parameters of Table 1 and vectorizing them as described in section 2 only leaves the step size and total duration of the simulation to be determined. In a first run, where the orbital period of Jupiter should be investigated, the latter is set to 4500 days, close to the expected period of 4331 days [6]. Varying the number of steps between $5 \cdot 10^7$ (maximum that could run on the available hardware) and 10 allows to test the robustness of the simulation with respect to the orbital period, which is extracted from the simulation data. This is achieved by computing Jupiter's absolute distance to the sun at every saved simulation step and finding the periodicity of the resulting data as show in Figure 3. The simulation proves to be very stable and exhibits only small variation between 4325.4 and 4410.0 days over the whole range of number of steps down to 50. It finally collapses upon further reduction, where no stable solutions are produced.
Another approach is equally promising: Averaging over a larger number of revolutions, of course tolerating the vast increase in step size, yields similar results as simulating a single orbital period with higher resolution. In fact, the differences are negligible, as shown in Figure 3.
In both cases, the best approximation was achieved using $5 \cdot 10^4$ steps, thus a step size of 2.16 and 43.2 hours for simulating 4500 days and $90,000$ days, respectively. The

resulting orbital period of Jupiter is 4229 days, only varying by 0.07% from the literature value of 4331 days.
Reducing the simulation running time to the expected value of 4331 days interestingly causes the simulated orbital period to deviate stronger, as it is found to be 4323 days.

Upon further analysis of the motion of the sun, an obvious flaw in the assumption of initial values emerges. Even though Jupiter's orbit is stable with respect to its relative distance to the sun, the sun itself drifts linearly. Since the sun accounts for 99.9 % of the total mass in the system on hand and therefore essentially is its center of mass, the whole system drifts linearly through space, as clearly visible in Figure 4. This drift can however easily be explained by flawed initial parameters, where the sun is assumed to have a velocity of 0, which is, in reality, not true. Thus, the system has a non-zero total momentum at time zero, causing the observed drift. Before analyzing the motion and orbital period of the sun, the linear movement has to be corrected for, as done in Figure 5. This yields an orbital radius of $\approx 0.65 \cdot 10^9 m$ of the sun and an orbital period of 4338 days. Comparing the sun's orbital radius with its radius of $\approx 0.69 \cdot 10^9 m$ shows that the sun stays inside its own volume during the orbital movement. Its orbital period of 4338 days is very close to Jupiter's orbital period. This can be easily understood by realizing that the only two bodies existing in the described system sync their orbital periods according to Newton's *actio et reactio*.

3.1.2 Sun and Earth

Analog to the simulation of the two-body system consisting of the sun and Jupiter, the earth's movement around the sun can be computed. Using initial parameters from Table 1 and a step size of 6.9 seconds ($5 \cdot 10^6$ steps over 400 days) results in the best approximation for the earth's orbital period – a year. In fact, the simulated value of 365.20 days extracted from Figure 6 is in almost perfect agreement with the literature value [6] and illustrates nicely the need for gap years in our calendar.

3.1.3 Earth and Moon

Another interesting time period on earth is, of course, a month. Even though the length of a month varies between 28 and 31 days in the Gregorian calendar, astronomically, the orbital period of the moon is 27.3 days [6]. Running the simulation with $5 \cdot 10^6$ steps over 30 days, resulting in a step size of 0.5 seconds, gives a good approximation of moon's orbital period – 26.8 days (Figure 7).

3.2 N-body problem: full solar system

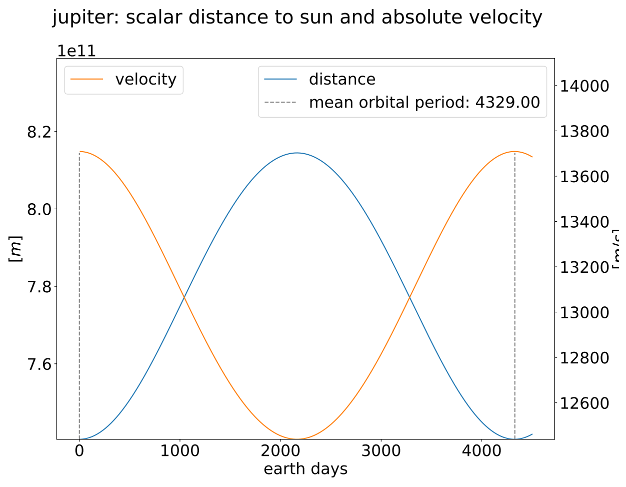
The ultimate goal of the simulation is, of course, to be able to approximate the movements of celestial bodies within the solar system – including the earth's moon. Using the same step size and simulation parameters as in subsubsection 3.1.1 yields very good approximations of the orbital period of the inner planets. Compared to literature values [6], the differences are negligible. For Mercury, the relative error is 0.18%, while for Venus, Earth and Mars, it is smaller than 0.1%. Thus, even a comparably large step size of 2.16 hours suffices to extract very good approximations of orbital periods of planets. However, this simulation lacks the resolution to resolve the moon's movement properly. Interestingly, after about 1800 days, the moon escapes the sphere of influence of the earth and moves on a solar orbit afterwards. This is clearly visible in Figure 8.

Body	Orbital Period [days]		Error [%]
	Simulation	Literature	
Mercury	199.2	88.0	55.82
Venus	224.3	224.7	0.18
Earth	365.3	365.2	0.03
Mars	687.1	687.0	0.02
Jupiter	4321.6	4331	0.22
Saturn	10,675	10,747	0.67
Uranus	30,378	30,589	0.69
Neptune	58,390	59,800	2.41

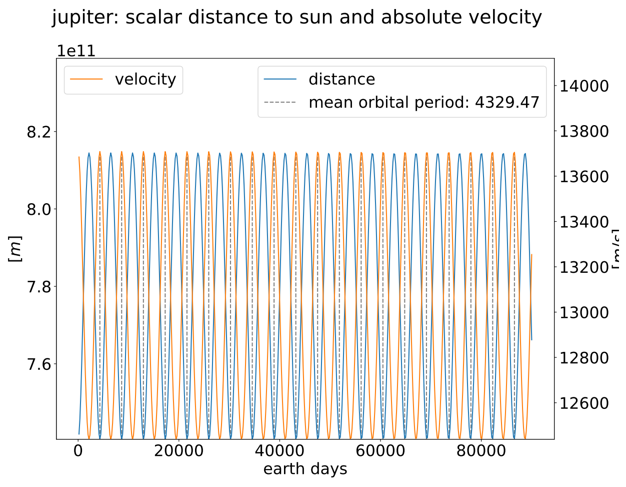
Table 2: Simulated orbital periods and literature values [6]. $1 \cdot 10^7$ steps were used to simulate 61,000 days, resulting in a step size of 8.8 minutes. With the exception of Mercury and Neptune, the relative error is smaller than 1%, proving the simulation to be rather precise even for rough step sizes and as much as 9 bodies.

4 Conclusion

The simulation proves to be able to provide – given reasonable parameters – very accurate approximations of the trajectories in our solar system. If focused and adjusted to specific scenarios, such as sun-earth simulations, the orbital period of two-body systems can be computed with high precision. N-body simulations are also able to produce stable and accurate solutions. When expanding the simulated time span to capture a full orbital period of Neptune however, the simulation start to collapse, resulting in overestimation of the orbital period of Mercury by more than factor 2 and an escape of the moon which moves on a solar orbit after some simulation time.
Even though the VVA itself offers little to no opportunities to be improved, there are a number of possibilities for the framework in which it is embedded. The main limitation of running high precision simulations is not CPU speed, but memory availability. This is not very surprising, since all simulation results are saved in RAM during runtime. In an N-body system with n simulation steps, there are $N \cdot n \cdot 4$ `vec` variables to be saved, each consisting of 3 `real` variables. Using a real kind of 16, $N \cdot n \cdot 4 \cdot 3 \cdot 16$ bytes are needed. Simulating 9 bodies with $5 \cdot 10^6$ steps therefore demands about 8.5 GB of RAM. Additionally, scalar values are saved as well. However, the VVA step $i + 1$ merely relies on step i and is completely independent of any earlier steps. An obvious improvement would thus be to automatically overwrite the `vec` arrays after some number of steps as their results are already saved to file. This would allow



(a)



(b)

Figure 3: Absolute distance to the sun and scalar velocity of Jupiter over the course of (a) a single revolution with high resolution (step size of 2.16 hours) and (b) 20 averaged full orbital periods with lower step size of 43.2 hours. The difference in the two approaches are negligible, proving the robustness of the simulation. The reported value of 4329 days only varies by 0.07% from the literature value of 4331 days [6]

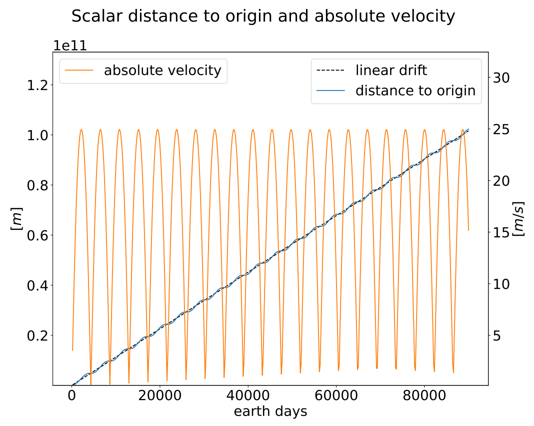


Figure 4: Absolute distance to origin and scalar velocity of the sun. While the velocity is very stable, the distance to origin oscillates around a linear movement, steadily drifting away from the origin.

for higher precision simulation without having to improve any hardware or altering the actual VVA. The resulting longer real-time running time could be complemented by introducing a log-to-file functionality, whose foundations are already laid in the program architecture.

If, for example through the optimization described above, more accurate results are produced, it might also be worth to improve the data analysis method. Using the absolute distance to the sun r and plotting against the time would allow to fit a parameterized ellipse to the data. As the corresponding fitting function, the polar parameterization of an ellipse relative to its focus would be used:

$$r(\theta) = \frac{a(1-e^2)}{1 \pm e\cos\theta} \tag{7}$$

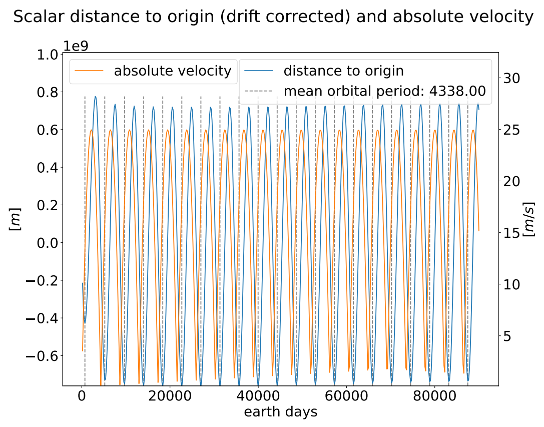


Figure 5: Drift-corrected absolute distance to origin and scalar velocity of the sun. After subtracting a linear drift from the distance to origin, the actual oscillations of the sun's orbital movement become visible. The radius of its orbit is $\approx 0.65 \cdot 10^9 m$ and therefore only slightly smaller than the sun's radius of $\approx 0.69 \cdot 10^9 m$.

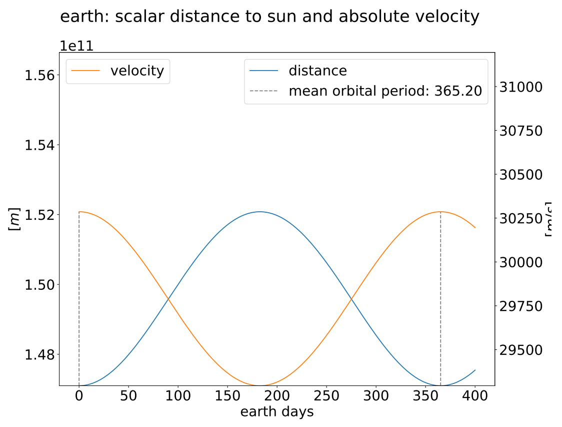


Figure 6: Absolute distance to sun and scalar velocity of the earth. Running the simulation over 400 days using $5 \cdot 10^6$ steps, thus a time step of 6.9 seconds, yields the best approximation for the orbital period of the earth. The reported value of 365.20 days is in exact agreement with the literature value [6].

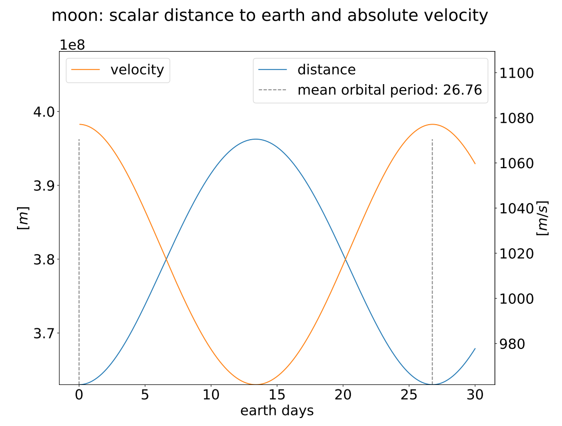


Figure 7: Absolute distance to earth and scalar velocity of the moon. The orbital period of 26.76 days is in good agreement with the literature value of 27.3 days [6].

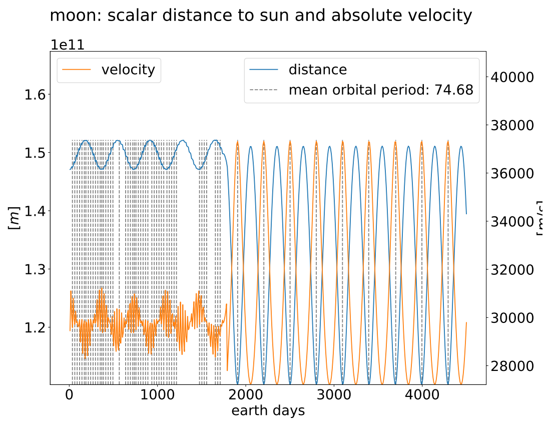


Figure 8: Absolute distance to sun and scalar velocity of the moon. Until about day 1800 of the simulation, the moon revolves around the earth. The two resulting oscillations are clearly visible: A small oscillation with a period of ≈ 28 days happens on top of a larger one, exhibiting a period of ≈ 365 days. Around day 1800, the moon escapes the earth's sphere of influence and continues its trajectory on a solar orbit. This is caused by the step size being too large, resulting in numerical inaccuracies which lead the orbit to collapse.

Since the central body is in the focus point of the elliptical orbit of its satellites, this would yield the simulated semi-major axis a and the eccentricity e. By using Kepler's Third Law, the orbital period can then be calculated with the help of the standard gravitational parameter μ:

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{8}$$

This method would potentially allow for more accurate estimations of the orbital period – if combined with high precision simulations.

References

[1] G. M. Clemence. "The Relativity Effect in Planetary Motions". In: *Rev. Mod. Phys.* 19 (4 Oct. 1947), pp. 361–364. DOI: 10.1103/RevModPhys.19.361. URL: https://link.aps.org/doi/10.1103/RevModPhys.19.361.

[2] Jack J. Lissauer and Imke de Pater. "Fundamental Planetary Science: physics, chemistry, and habitability." In: New York, NY, USA: Cambridge University Press, 2019, pp. 29–31.

[3] NASA Jet Propulsion Laboratory: Solar System Dynamics. *Astrodynamic Parameters*. URL: https://ssd.jpl.nasa.gov/astro_par.html. (accessed: December 21, 2021).

[4] William C. 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: *The Journal of Chemical Physics* 76.1 (1982), pp. 637–649. DOI: 10.1063/1.442716. eprint: https://doi.org/10.1063/1.442716. URL: https://doi.org/10.1063/1.442716.

[5] The NIST Reference on Constants, Units and Uncertainty. *Newtonian constant of gravitation.* URL: `https://physics.nist.gov/cgi-bin/cuu/Value?bg`. (accessed: December 21, 2021).

[6] Williams, Dr. David R. *Planetary Fact Sheet - Metric.* 21 October 2019. URL: `https://nssdc.gsfc.nasa.gov/planetary/factsheet/`. (accessed: December 21, 2021).

[7] Willman Jr, Alexander J. *Sol Planetary System Data.* 12 July 2021. URL: `https://www.princeton.edu/~willman/planetary_systems/Sol/`. (accessed: December 21, 2021).