Project A:
Solar System

Computer Modelling

Due: 16:00 Thursday, Week 8, Semester 2

Aims

In this project, you will write code to describe an N-body system interacting through Newtonian gravity. The code shall, eventually, simulate the motion of the main bodies of our solar system, starting from realistic initial conditions. Your code will enable you to compare simulation results to astrophysical data.

1. Background

1.1. Gravity

Simulations of gravitational systems are a key tool in modern astrophysics. Within our solar system, Newtonian Gravity is sufficient for almost every observation we can make.

The gravitational force on a body of mass m_1 at position r_1 from a body with mass m_2 at position r_2 is given by

$$\boldsymbol{F} = -Gm_1m_2\frac{\boldsymbol{r_1} - \boldsymbol{r_2}}{|\boldsymbol{r_1} - \boldsymbol{r_2}|^3} \tag{1}$$

and the gravitational potential of the two by:

$$U_{12} = -\frac{Gm_1m_2}{|\boldsymbol{r_1} - \boldsymbol{r_2}|} \tag{2}$$

1.2. N-Body simulations

When we upgrade simulations from two bodies (like the ones we simulated last semester) to arbitrary numbers, there are several important changes we have to understand.

First the force on any one particle is now the sum of the forces on it from all the other particles:

$$\boldsymbol{F}_i = \sum_j \boldsymbol{F}_{ij} = -Gm_i \sum_j m_j \frac{\boldsymbol{r_i} - \boldsymbol{r_j}}{|\boldsymbol{r_i} - \boldsymbol{r_j}|^3} \tag{3}$$

We have to be more careful with the potential energy, and avoid double counting, as the potential on A due to B should not be added to the potential of B due to A - this is the same energy. There are a few ways of implementing this. You can either do the complete sum and then divide by two, or by avoiding the double count in the first place:

$$U = \frac{1}{2} \sum_i \sum_{j \neq i} U_{ij} \tag{4}$$

$$= \sum_i \sum_{j > i} U_{ij} \tag{5}$$

Note that in each case we must avoid the potential of a particle with itself.

1.3. Units

The choice of units to use in a simulation is important - the simulation can be more accurate and easier to understand if the units give values which within a few orders of magnitude of one. In solar system simulations, using metres or seconds would lead to very awkward units.

In this simulation you shall adopt the units:

- astronomical units (AU) for length,
- Earth days for time
- Earth mass for mass

To do this we must make sure the constant G is in these units. G is usually expressed as $6.6743 \times 10^{11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$, and this is equal to $8.8877 \times 10^{-10}\,\mathrm{AU^3\,M_{earth}{}^{-1}\,day^{-2}}$

1.4. Centre-of-mass corrections

The initial conditions of an N-body system are unlikely to have non-vanishing linear momentum. This leads to an undesirable drift during long simulations. To remove this, we subtract the centre-of-mass velocity from each particle's initial velocity. If the bodies have initial velocities $\boldsymbol{v}_i$, then the simulation has a total momentum of:

$$\boldsymbol{P} = \sum_i m_i \boldsymbol{v}_i \tag{6}$$

If we subtract the centre-of-mass velocity $\boldsymbol{v}_{\text{com}}$, which is

$$\boldsymbol{v}_{\text{com}} = \frac{1}{\sum_i m_i} \boldsymbol{P} \tag{7}$$

from each body's motion at the start of the simulation, then there will be no drift to the overall system.

1.5. Astrophysical Observables

There are various measured orbital properties that we can compare to our simulation:

- Perihelion and aphelion, the smallest and largest distance of a planet from the sun
- Orbital periods, the time for a complete orbit
- For the moon, perigee and apogee, the min and max distances from the Earth

The word "periapsis" means either "perigee" or "perihelion" depending whic body is under discussion.

1.6. Trajectory Files

Trajectory files are often used within scientific modelling to represent a set of points (in three-dimensional Cartesian space) at a number of different steps within an ordered time series. Once the data has been stored in a trajectory file we can then visualise it in a number of ways, for example by animating it or by plotting all the points simultaneously. The trajectory file can be used to perform further analysis, or to compare different runs, as well as for artistic purposes.

The XYZ format (for e.g. two particles) looks like this:

```
2
Point = 1
s1 x11 y11 z11
s2 x21 y21 z21
2
Point = 2
s1 x12 y12 z12
s2 x22 y22 z22
⋮
2
Point = m
s1 x1m y1m z1m
s2 x2m y2m z2m
```

You can see that the file consists of `m` repeating units (where `m` is the number of steps in the trajectory) and that each unit consists of two header lines: the first specifies the number of points to plot (this should be the same for each unit) and a comment line (in the example above it specifies the point number). Following that, the file specifies the label and the Cartesian coordinates of each particle at this trajectory entry.

2. Coding Tasks

In this project, you will generalise your simulation program from Exercise 3 so that you can simulate an arbitrary number of classical point particles under Newtonian gravitation.

It is better to **begin early, before pressure from other courses builds up**.

Your code should build on the `Particle3D` class and the Velocity Verlet integrator your wrote in semester 1. Check the feedback you got on that exercise on LEARN, and fix any issues identified.

Document your code as you go along using both docstrings for functions and inline # comments explaining what you are doing.

2.1. Basic simulation code

Modify your code to use an arbitrary number particles in a list instead of just two. Represent the full system as a list of particles, where e.g. `particles[i]` is a Particle3D instance.

You will need code to:

- compute all the separations between each pair of particles,
- compute all the forces at once from this array,
- update all the particle velocities at once,
- update all the particle positions at once,
- compute the total potential and kinetic energies.

Computing the pair separations ($\boldsymbol{r}_i - \boldsymbol{r}_j$) every step is the most time-consuming element of an N-body simulation. That method should exploit `numpy`, as well as symmetries if available. Typing all the $N(N-1)$ ordered particle pairings to compute the force and energy is tedious and inflexible. You should therefore automatically go through the particles using loops.

Finally, add a step before the simulation to subtract the centre-of-mass velocity from the system.

2.2. Basic tests

First, test your new system with a simulation of just the Earth and the sun. In the units we have chosen (days, AU, Earth masses), this is a very simple simulation - you just need to know that the sun's mass is close to 333,000 times the Earth's mass.

Determine the initial conditions for this scenario, putting the sun at the centre and stationary, and run your simulation with them and a time step of 1 day. If everything is working then the Earth should orbit it in one year. Check also that the total energy is conserved.

2.3. Full simulation

Use the file `solar_system.dat` which contains initial conditions for all the planets, the sun, the moon, and Halley's comet, to create a simulation of the solar system.

Modify your simulation so it makes numpy arrays of the planet positions and velocities, the time, and the energy across the simulation.

Perform some tests and make some plots to ensure the planets are orbiting correctly.

Make a plot of the orbit of the Mercury around the sun, and check that it is close to the

true value of 88 days. Make another plot of the orbit of the moon around the Earth, and make a similar check.

2.4. Visualisation

Modify your code so that it saves the complete trajectories of all the particles in a simulation to an XYZ-format file, as described above.

Your P3D `__str__()` method should already produce a string in the correct format following Exercise 2 feedback. Using it, now code a method that writes out a complete entry for a single time-step to a trajectory file.

Run the system for 20 years with a time step of 1 day, and save the XYZ file this generates. Use either the program VMD (see Appendix), or if it doesn't work on your system, the script `plot_xyz.py` on LEARN to view an animation of the two-particle system: run `python plot_xyz.py my_file.xyz` for a basic animation, or `python plot_xyz.py --help` for more options.

2.5. Measuring observables

Add functions to compute the observables (apsides, orbital periods, and energy deviations) as described above for the entire system. These functions should operate on the arrays you generated in section 2.3, rather than trying to do the calculation during the integration.

There are several ways you can calculate both the apsides and periods. Ensure that if the method you use fails, e.g. because the planet has only partially completed its orbit, then the method indicates this to the user in some way, for example by returning zero or the special value called `NaN` (short for "Not-a-Number").

Carefully test the methods on your simulation.

2.6. User input

Add a user input system for selecting:

- the simulation parameters δt and N_{step}
- the initial conditions file
- the name of output energy and xyz files

It's up to you how you do this, but the more convenient it is for the user the better - having to manually type out the values every time you want to run is sub-optimal.

2.7. Performance

Early optimization is the root of all evil. Make sure your code runs before trying this section.

Having said that, the typical errors slowing student codes down are computing expensive quantities over and over again, and improper usage of `numpy`. The worst offender is by far computing $\boldsymbol{r}_i - \boldsymbol{r}_j$ multiple times per step.

If needed, modify your code to only compute pair separations once per step. That alone should result in acceptable performance: a recent Intel Core i5 should run the full simulation for 100 years and dt of 1 day in well under 2 minutes.

3. Submission

3.1. Anonymising

Your submitted code will be anonymised and then uploaded for a plagiarism check using `moss`.

To enable this, ensure none of your files include your exam number (e.g. B0000). Also ensure your names and matriculation numbers appear only on the headers of the python files, and not elsewhere.

3.2. What to submit

Collect in a directory:

- all required simulation input files
- all your simulation python file(s)
- a readme telling us how to run your code and what the format and units of the input and output files are
- the XYZ trajectory file for a representative simulation rum

Submissions larger than 50 MB will not be accepted. Hence, make informed choices on how often to print planetary positions to file (is every single time step necessary?), how many digits are relevant (use formatted outputs), and restrict the overall simulation time to reasonable values (what is the longest period in the system, and how many cycles do you need?).

3.3. Compressing your submission directory

Windows Right-click on the folder, and under the "Send-to" menu, select "Compressed (zipped) folder". Rename the file that is created to include your university ID.

Mac Ctrl-click on the folder and click "Compress name_of_folder". Rename the file that is created to include your university ID.

Linux Run this in a terminal (without the $):

```
$ tar -czvf project-YOUR-UNI-ID.tar.gz
   project_directory_name
```

and verify the command ran without errors. Check that all the files you want are included by running

```
$ tar -tf project-YOUR-UNI-ID.tar.gz
```

3.4. Submitting

Submit the compressed package through the course LEARN page, by **16:00 on Thursday, week 8 of semester 2**. Successful submissions are emailed a receipt.

4. Marking Scheme

This assignment counts for 25% of your total course mark.

1. Code compiles and works with inputs provided [**5**]
2. Input and output formats sensible and appropriate [**5**]
3. The simulation is physically correct [**10**]

4. Code has all required functionality to measure observables: list methods & `numpy`, correct file I/O, centre-of-mass correction, total energy tracking, apsides and orbital periods for both planets and the Moon in both cases. **[20]**

5. Code layout, naming conventions, and comments are clear and logical **[10]**

Total: 50 marks.

A. Plotting with VMD

Operating systems: VMD works well under Windows and Linux. It used to work well on Macs until Mojave (10.14), but Catalina (10.15) broke it. Running VMD on Catalina requires a special test build and heavy circumventing binary notarization, but can be done. Running VMD under Big Sur may not be doable at all.

Plotting trajectories with VMD in the CPLab is straightforward, provided one is aware of a few potential pitfalls:

1. **Order of magnitude:** VMD is designed with *molecules* in mind, and expects figures near unity (10^6 is fine). If the scale is very large, though, VMD may crash. The units we chose in this project should work.

2. **Visualization:** Particles are assigned to an element based on their labels, and then given a radius proportional to the covalent radius of that element. Do not expect to see anything if your trajectory coordinates are in km.

3. **File size:** It is possible to write a trajectory file of 108 particles for a million time steps, but such trajectory file would be unwieldy. Think about how you can produce file output every n-th timestep, and whether n can be passed to your code together with other simulation parameters.

You should use VMD to visualise the trajectories for the simulations you run. You can view a trajectory in VMD by opening the XYZ file you saved within it.

You should experiment with visual representations in VMD (Graphics → Representations menu item) to find different ways of representing the time series. In particular:

- Use the "Points" drawing method to visualise the trajectories as particles moving in time. Can you speed up and slow down the animation?

- Use the "Points" drawing method to create a static representation of the entire trajectory. You will need to use the "Trajectory" tab of the representations window to set the trajectory range (lower and upper limit of steps to display) and stride

(increment between displayed steps) to generate a meaningful representation.

You could also experiment with the `PBCTools` plugin for VMD (http://www.ks.uiuc.edu/Research/vmd/plugins/pbctools/), which allows you to relay information about periodic boundary conditions to VMD (which is not included in `.xyz` files). For instance, you can set box size and display its edges by typing

```
vmd > pbc set {5.0 5.0 5.0} -all
vmd > pbc box
```