Project A
Solar System

Computer Modelling 2017-18

Due: 5pm Thursday, Week 8, Semester 2

1 Aims

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

2 Tasks

2.1 Simulation code

Your code builds on the `Particle3D` class you wrote in the previous exercise. It should

- Read in an arbitrary number of particles and their initial positions and velocities from an input file,
- Read simulation parameters from a second input file,
- Set up a list of `Particle3D` objects to hold the particles,
- Correct for initial centre-of-mass motion,
- Simulate the evolution of the system using the velocity Verlet time integration algorithm,
- Write a trajectory file for the simulation that can be visualised using VMD.

Use this code to simulate the Solar System, including the Sun, all planets, Pluto, Earth's Moon, and Halley's Comet. Detailed suggestions on how to implement these objectives are given in Section 3 below. As a rough indication, this part of the code should be finished after the teaching break in semester 2. *If your code is far behind schedule at that stage, be sure to get in touch with the teaching staff and TA's to help you!*

2.2 Astrophysical observables

Your code should include functionality to compare simulation outcomes to astrophysical data, as well as measure the general accuracy of the simulation. The following data should be written to Terminal or an output file:

- Apo- and periapses for all bodies except the Sun. That means apo- and perigee for Earth's Moon, and apo- and perihelia for all other bodies.
- Orbital period lengths, in Earth days or years, for all bodies except the Sun. For Earth's Moon, determine its period to orbit Earth.
- Fluctuations of the total energy of the system.

There are various ways to implement the functionalities above, and you should choose your own. Orbital geometry data and period lengths should be determined independently of each other.

3 Detailed Instructions

In this project, you will generalise your simulation program from Exercise 3 so that you can simulate an arbitrary number of particles. The reference test case should be the two-particle simulation you performed in Exercise 3 – you should check that your modified program can replicate the results for this simple system before moving on to the more complex simulations. Next, you should set up the simulation with Mercury and Venus orbiting the Sun before completing the Solar System.

The largest cost in a simulation of this type is in computing all the pairwise interactions between particles. Typing all the $N(N-1)$ ordered particle pairings to compute the force and energy is clearly going to get tedious even for just three particles (where there would be 6 interactions). Ponder the cost of this operation for an entire protein or for a galaxy of objects where the number of particles required could be on the order of millions. We will use lists and loops to repeat the force calculation for all pairs of particles rather than try to write each interaction individually.

The instructions below are a suggestion of how to gradually build your project code by incrementally adding complexity and constantly testing it. You do not have to follow this suggested route, but beware that TA's and lecturers can best help you with issues and debugging if you do not stray too far from it. *Do not attempt to write the entire code in one go. Good on you if you can, but very likely you will introduce a lot of bugs that will be very hard to find.*

3.1 Gravitational Force and Energy between two Particles

In Exercise 3 we considered two particles interacting via the Morse potential. Now you should adapt the force vector and energy calculation routines so they return the gravitational interaction between two `Particle3D` objects instead.

Create two methods that take two `Particle3D` objects as their arguments and return the force vector and the potential energy, respectively. Assume for simplicity that the gravitational constant is 1.

3.2 Particle List Methods

You should now add a set of methods that can operate on *lists* of particles. Using these you can easily extend simulations to an arbitrary number of particles. You should add functionality that:

- Updates the velocity for all particles in a list. Remember the total force vector for a particle is the sum of the forces due to all the other particles. You should formulate an algorithm to compute this sum for a single particle and then consider how you might modify this so it operates on each particle in turn.
- Updates the position for all the particles in a list.
- Calculates the total energy due to particle pair interactions and their kinetic energies. Take care here to avoid double-counting interactions.

3.3 Visualization

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.

VMD is a standard tool that is used for visualising trajectories in a variety of scientific fields ranging from biology to astrophysics.

The format of the trajectory file for VMD (for a system containing two points) 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 title line (in the example above it specifies the point number). Following the header lines are the lines specifying the Cartesian coordinates for the particles (one for each point), which consist of: the particle label (some text) and the x, y, and z coordinates for that particle at this trajectory entry.

You should already have a `__str__()` method in your `Particle3D` class that produces a String in the correct format for a VMD trajectory file. Write a method to write out a complete entry for a single timestep to a VMD trajectory file. While you can write out trajectory information at every single timestep, often this leads to very large trajectory files. 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 using something like:

```
[user@cplab001 ~]$ vmd myTrajFile.xyz
```

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.

3.4 N-body Simulation Code

Write a program to simulate many-body systems. This can be based on the simulation code you wrote for Exercise 3. It should contain all the functionality requested in section 2.1.

As we may now also have many particles in a simulation it makes sense to read in the simulation parameters (number of steps, time step, etc.) from one file and the details of the particles (number of particles, labels, masses, starting positions, starting velocities) from another file. This means you program should take three command line arguments, i.e.:

```
[user@cplab001 ~]$ python ParticleManyBody.py particle.input
    param.input traj.xyz
```

where `particle.input` is the file containing the particle details; `param.input` is the file containing the simulation parameters; and `traj.xyz` is the name of the output file containing the trajectory (this must have a `.xyz` extension to work correctly with VMD, see previous section).

This completed program now looks similar to many of the large software packages that exist for performing this type of simulation.

3.5 Initial Conditions and Test Case

Now you have a working code we will start to simulate more complex systems and use a simple test case before adding more functionality to your program.

Set up a three-body system that includes the Sun, Mercury and Venus. You should use realistic masses and initial conditions. You may want to use NASA/JPLs HORIZONS system to obtain those values at a given moment in time: http://ssd.jpl.nasa.gov/horizons.cgi. If you use this source, change 'Ephemeris Type' to 'Vector Table' and, if you want, use 'Table Settings' to adjust the output units. Then, choose a common start date and obtain data for all 'Target Bodies'.

Remember that the Sun also has non-trivial initial conditions and that you need to adjust the numerical value of the gravitational constant in the force and energy calculations to reflect your choice of units.

Use VMD to visualise the simulations and make sure they are working correctly. I.e., do you get closed orbits for Mercury and Venus around the Sun? How many timesteps does one Mercury orbit take? In your units, how many Earth days does this correspond to, and is it close to the known orbital period of Mercury? A time step of $dt = 1\text{day}$ should give reasonable results; if it does not, your time integrator is wrong.

3.6 Centre-of-mass Motion Correction

Often, the initial conditions of a N-body system have non-vanishing linear momentum, which will lead to drift of the centre-of-mass during the simulation. To remove this, you need to subtract the centre-of-mass velocity from each particle's initial velocity. If your bodies have initial velocities $\boldsymbol{v}_i$, then the simulation has a total momentum of

$$\boldsymbol{P} = \sum_i m_i \boldsymbol{v}_i \tag{1}$$

Adjust the initial velocity of each body in the system (including the Sun) by the negative of the centre-of-mass velocity $\boldsymbol{v}_{\text{com}}$, which is given as

$$\boldsymbol{v}_{\text{com}} = \frac{1}{\sum_i m_i} \boldsymbol{P} \tag{2}$$

You can test your centre-of-mass correction by adding a constant, e.g. to the v_x-component of the initial velocities of *all* particles and re-run the simulation. The trajectory should look identical, with no perceivable drift of the centre-of-mass at long run times.

3.7 Full Solar System

Now add appropriate initial parameters (masses, positions, velocities) for the remaining planets in the Solar System, Earth's Moon, Pluto and Halley's Comet. Run the simulation and confirm that it works correctly. You will need to increase the total run time to confirm that the outer planets move on closed orbits. Does the Moon orbit the Earth? This will be hard to ascertain from the trajectory visualization, and adding functionality to the code to produce orbital data will help with the analysis of the code performance and accuracy.

3.8 Adding Observables

You should now add functionality to your code to obtain orbital data of the bodies in the Solar System. In particular, your code should produce output on

- Apo- and periapses of all bodies, except the Sun. Consider the Moon's orbit around Earth, not the Sun.
- Orbital period lengths for all bodies, except the Sun. Again, study Moon's orbit around Earth.

You should also track fluctuations in the total energy of the system during the simulation, to be able to assess its accuracy.

It is up to you to implement the above-mentioned functionality in a way of your choosing. However, you should determine orbital geometries and periods independently, and *not* use known relations like Kepler's laws to deduce one from the other. It is preferred to be able to track partial orbit completions instead of only complete orbits.

4 Submission

Package the subdirectory that contains `Particle3D.py`, your simulation program, input files, and the resulting trajectory file for a representative simulation run. For instance, if your subdirectory is called `project`, you can use the `tar` and `gzip` commands by running:

```
[user@cplab001 ~]$ tar cvf project.tar project
[user@cplab001 ~]$ gzip project.tar
```

In the documentation of your simulation program, describe how to run it and what the format and units of the input and output files are.

We will not accept submissions that are larger than 50MB total. 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?). One of your group members should submit the compressed package (e.g., `project.tar.gz`) through the course LEARN page, by **5pm on Thursday, week 8 of semester 2**.

5 Marking Scheme

This assignment counts for 20% of your total course mark.

1. Code compiles and works with inputs provided [4]
2. Input and output formats sensible and appropriate [4]
3. Code has all required functionality: list methods, file I/O, centre-of-mass correction, velocity Verlet integrator, total energy tracking, orbit geometries, orbit periods, special consideration for Moon. [20]
4. Code layout, naming conventions, and comments are clear and logical [12]

Total: 40 marks.