Computing solar system dynamics: Assessment 3

1 Movtiation

The solar system is perhaps the earliest scientifically studied N-body problem. Starting with Brahe's meticulous observations of the motion of celestial bodies, the analysis of which by Kepler's led to his laws of planetary motion, which subsequently laid the foundations for Newton's law of universal gravitation. This physics was understood already in the 17th century and stimulated Boyle to construct a mechanic clockwork model of the solar system known as a Orrery, which in many ways constitutes one of the earliest examples of a physics simulation. To accurately solve the full N-body problem, however, requires a numerical approach only feasible with modern computers. In this assessment our aim is to simulate the motion of the major solar bodies under Newtonian gravity. This will be accomplished by solving through direct numerical integration the N set of equations of motion in which each body is treated as a point mass that interacts with the remaining $N-1$ bodies. The solar system comprises the Sun, planets, moons, dwarf planets[1] and asteroids. A conservative estimate gives over 10^4 bodies with masses in excess of 10^{12}kg. Here we will restrict our considerations to the largest objects, namely the Sun and planets making $N = 9$.

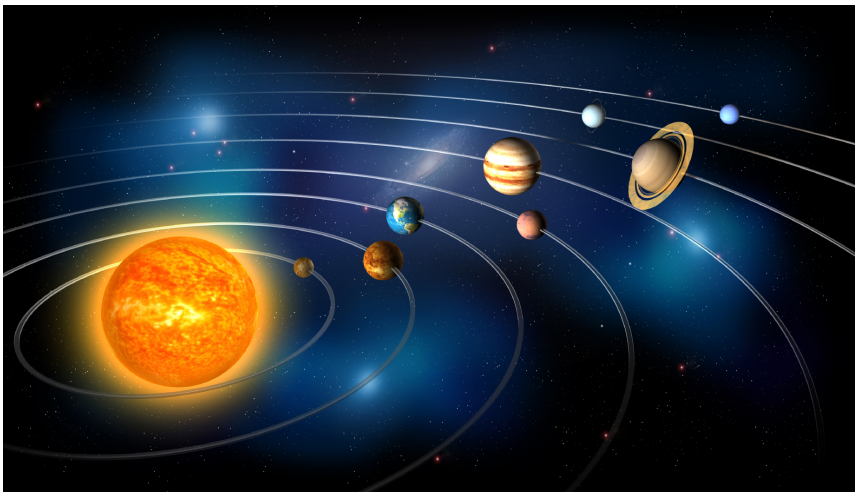

Figure 1: A stock image of a classic artistic visual representation of the solar system.

[1] Recall that Pluto was downgraded to a dwarf planet back in 2006.

2 Assessment objective

This assessment is an opportunity for you to demonstrate that you can:

- implement a well-defined mathematical procedure as working C++ code;
- verify the correctness of the code by looking at a test case;
- simulate a more challenging physical setup (possibly involving submitting jobs to BluePebble);
- post-process output data using Python scripts and comment critically on what you find.

The primary task of this assessment is to **finish the implementation of a C++ program which solves the N-body gravitational problem and compute the orbits of the planets**. The numerical method to be implemented is called *velocity Verlet*, and it is described in detail in the next section. Further sections describe the starter code which comes with this assessment, the format of the input and output data which the code provided handles for you, and a detailed commentary on the main source code file you will edit. At the end are a set of exercises for you to complete which guide you through this task.

3 Velocity Verlet integration

Combining Newton's 2nd law with Newton's universal law of gravitation we arrive at the set of coupled equation of motion for a body p under the gravitational forces from all other bodies in a three-dimensional Cartesian coordinate system

$$\ddot{\vec{r}}_p(t) = \sum_{j=1, j\neq p}^{N} m_j \frac{\vec{r}_j(t) - \vec{r}_p(t)}{|\vec{r}_j(t) - \vec{r}_p(t)|^3}, \tag{1}$$

where the position and acceleration of body $p = 1, 2, \ldots, N$ are

$$\vec{r}_p(t) = \begin{pmatrix} x_p(t) \\ y_p(t) \\ z_p(t) \end{pmatrix}, \quad \ddot{\vec{r}}_p(t) = \frac{\mathrm{d}^2}{\mathrm{d}t^2}\vec{r}_p(t) = \begin{pmatrix} \frac{\mathrm{d}^2}{\mathrm{d}t^2}x_p(t) \\ \frac{\mathrm{d}^2}{\mathrm{d}t^2}y_p(t) \\ \frac{\mathrm{d}^2}{\mathrm{d}t^2}z_p(t) \end{pmatrix}, \tag{2}$$

while m_j is the mass of body j. The physicists among you may question why Newton's gravitational constant G does not appear in Eq. (1). The reason is that the quantities appearing in this equation, and others to follow, are all dimensionless. Removing physical dimensions is a crucial technique in scientific computations because it ensures numerical values in our code are reasonably sized, e.g. neither too small nor too big unnecessarily. For example expressing distances for astronomical calculations in terms of SI units is unlikely to achieve this. Instead it is sensible to pick a mass m_s, a length ℓ_s and a time t_s which represent relevant scales for our problem, and then replace basic physical quantities in our equations by ratios with them as

$$r = \frac{\texttt{length}}{\ell_s}, \quad m = \frac{\texttt{mass}}{m_s} \quad \text{and} \quad t = \frac{\texttt{time}}{t_s}. \tag{3}$$

We will fixed m_s and ℓ_s shortly, but for gravitational problems we can in fact derive a characteristic time scale t_s from G itself as

$$t_s = \sqrt{\frac{\ell_s^3}{G m_s}}. \tag{4}$$

This choice ensures G never appears explicitly in our equations.

Notice that Eq. (1) tells us that the acceleration of body p at any time t is a function only of the position of all the bodies at that time. Formally, to solve this set of N coupled ordinary differential equations (ODEs) we integrate from some initial time $t = t_i$ the acceleration $\ddot{\vec{r}}_p(t)$ to get the velocity $\dot{\vec{r}}_p(t)$ and integrate again to get the position $\vec{r}_p(t)$ up to some final time $t = t_f$. Since we have two integrations we need initial conditions for the velocity $\dot{\vec{r}}_p(t_i)$ and position $\vec{r}_p(t_i)$ of each body.

Numerical approaches to integrating coupled ODEs[2] frequently involve discretizing time t into a T equally-sized time steps $\Delta t = (t_f - t_i)/T$. Here we will use the so-called *velocity Verlet* method. It has several useful features. It involves the positions and velocities at integer multiples of the time-step Δt. It has an error of order $(\Delta t)^2$ which is better than a simple Euler method, but is computationally much simpler compared to the more accurate 4th order Runge-Kutta method. It respects time-reversal symmetry and is symplectic (area preserving). These latter properties in particular make it stable for long-time integrations.

We evaluate time on a grid $t_n = t_i + n\Delta t$ indexed by $n = 0, 1, 2, \ldots, T$, so $t_0 = t_i$ and $t_T = t_f$, and we denote the position evaluated at these times as $\vec{r}_p(t_n) = \vec{r}_{p,n}$, similarly for velocity and acceleration. For particle p the velocity Verlet method integrates the position $\vec{r}_{p,n}$ and the velocity $\dot{\vec{r}}_{p,n}$ at time step n one step forward with the following update:

$$\vec{r}_{p,n+1} = \vec{r}_{p,n} + \dot{\vec{r}}_{p,n}\Delta t + \frac{1}{2}\ddot{\vec{r}}_{p,n}\Delta t^2, \tag{5}$$

$$\dot{\vec{r}}_{p,n+1} = \dot{\vec{r}}_{p,n} + \frac{1}{2}\left(\ddot{\vec{r}}_{p,n} + \ddot{\vec{r}}_{p,n+1}\right)\Delta t. \tag{6}$$

Notice that the position update in Eq. (5) needs its current position $\vec{r}_{p,n}$ and velocity $\dot{\vec{r}}_{p,n}$ (computed from an earlier step) and the acceleration $\ddot{\vec{r}}_{p,n}$ which can be computed from all the body positions $\vec{r}_{j,n}$ using Eq. (1). Next, the velocity update in Eq. (6) again uses the current velocity $\dot{\vec{r}}_{p,n}$ and acceleration $\ddot{\vec{r}}_{p,n}$, but also uses the updated acceleration $\ddot{\vec{r}}_{p,n+1}$ which is computed from the updated position $\vec{r}_{p,n+1}$ just calculated in Eq. (5).

For each time step n and each body p we compute its gravitational potential energy

$$V_{p,n} = -\sum_{j=1, j\neq p}^{N} \frac{m_j m_p}{|\vec{r}_{j,n} - \vec{r}_{p,n}|}, \tag{7}$$

its kinetic energy

$$K_{p,n} = \frac{1}{2} m_p \, \dot{\vec{r}}_{p,n} \cdot \dot{\vec{r}}_{p,n}, \tag{8}$$

and its angular momentum

$$\vec{L}_{p,n} = m_p \, \vec{r}_{p,n} \times \dot{\vec{r}}_{p,n}. \tag{9}$$

Again, all these quantities are dimensionless based on an energy scale $E_s = Gm_s/\ell_s$ and an angular momentum scale $L_s = Gm_s^3\ell_s$ derived from our basic unit.

There are two crucial result about an N-body problem involving central forces between bodies, like gravitation. First, the forces are conservative meaning that the total energy[3] of the system

$$E_n = \sum_{p=1}^{N} \left(K_{p,n} + \frac{1}{2} V_{p,n} \right), \tag{10}$$

[2] You will explore in more detail methods of solving ODEs in the next part of the course using Python.

[3] The factor of $\frac{1}{2}$ on $V_{p,n}$ in Eq. (10) takes care that we would double counted the potential energies in this sum.

is conserved, so should be identical for every time step n. This explains why orbits of planets are stable over very long times. Second, central forces between the particles produce no net torque on the system, so the total angular momentum

$$\vec{L}_n = \sum_{p=1}^{N} \vec{L}_{p,n}, \tag{11}$$

should not change with time step n. This explains why planets tend to orbit in a plane and Kepler's 2nd law that the orbit sweep out equal areas in equal times. Together these conservation laws give us a powerful tool to quantitatively analyse the quality of our numerical approximate solution obtained from velocity Verlet.

4 Guide to the starter code

The starter code for this assessment contains some files which are identical, or modified versions, of those encountered already in the Week 17 workshop. You will only need to modify two files marked by *. Here is a description of each file:

- `compute_orbits.cpp`*
  This contains the C++ `main()` function. **Most of the work in this assessment is adding functions and code to this file**.
- `vector3d.hpp`
  This is the header file defining the 3d vector class `vec` used in Week 17.
- `vector3d.cpp`
  This is the source code implementing the `vec` class.
- `body.hpp`
  This is a header file defining the `body` class. This is based on our work in Week 17 but the class has been expanded with some additional data and methods.
- `body.cpp`
  This is the source code implementing the `body` class.
- `sun_earth.csv`, `inner_planets.csv`, `solar_system.csv`
  The input data files discussed below.
- `postprocess.py`*
  A Python script that extracts the data from the `.csv` file outputted by the C++ code `compute_orbits.cpp` and generates useful plots of the orbital motion of bodies simulated. **You will only need to slightly modify this file to tell it to load different output data files you will produce**.

Your assessment submission should include the C++ source files `compute_orbits.cpp`, and a PDF document containing the plots generated with some discussion.

5 Data provided and outputted

The input data provided for this assessment is the positions and velocities of Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune in a coordinate system with the Sun at the origin and measured relative to the ecliptic plane at a specific time/date[4] generated by JPL's HORIZONS system. The data is dimensionless based on the choices $\ell_s = 1.50 \times 10^{11}$m, which is the average distance from the

[4] Arbitrarily chosen as 2022-Dec-13 00:00:00.

Sun to the Earth, $m_s = 5.97 \times 10^{24}$kg which is the mass of the Earth, resulting in $t_s = 2.91 \times 10^9$s, which is 92.27 years. This value of t_s indicates to us that the relevant timescale for orbital motion is around 100 years, with Earth's orbital period 0.011 in this unit while Neptune's is 1.79. The input data is a `.csv` file that describes each body in the system with the following format:

`<name>`

`<mass>`

$\langle r_x \rangle$, $\langle r_y \rangle$, $\langle r_z \rangle$

$\langle v_x \rangle$, $\langle v_y \rangle$, $\langle v_z \rangle$

The function `read_init()` provided parses this file and loads the data for each body in a `body` object amended to the end of a vector of `body` objects. You are provided with three input files containing the initial conditions:

- `sun_earth.csv`
  For just the Sun and Earth.
- `inner_planets.csv`
  For the Sun, Mercury, Venus and Earth.
- `solar_system.csv`
  For the Sun and all 8 planets.

In `compute_orbits.cpp` there is a function `save_data()` provided which saves information about each `body` object to a `.csv` file. The format of this data starts with a header line `<dt>, <T>, <Tsave>, <N>` recording the time step, number of time steps, save time steps and number of bodies, and a second line containing `<name>` for each of the `<N>` bodies in the system simulation. Then for each save time step:

`<time>`, `<E>`, `<Lmag>`

$\langle r_x \rangle$, $\langle r_y \rangle$, $\langle r_z \rangle$, $\langle v_x \rangle$, $\langle v_y \rangle$, $\langle v_z \rangle$,

$\vdots$

For all $<$ `N` $>$ bodies

The Python script `postprocess.py` provided reads this data in and produces plots from it.

6 Getting started

Let's have a closer look at the main file `compute_orbits.cpp` which you will be working on. The file begins with the usual standard header includes, along with those for the `vec` and `body` classes implemented in `vector3d.cpp` and `body.cpp`, as:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>

#include "vector3d.hpp"
#include "body.hpp"

using std::cout, std::endl;

```

```
// *** Function declarations ***

// ------------------------------------------------------------
// ADD YOUR FUNCTION DECLARATIONS HERE



// ------------------------------------------------------------
// Read input data from file
void read_init(std::string input_file, std::vector<body> &system);
// Read the components of a 3d vector from a line
void read_vector3d(std::stringstream& data_line, double& x, double& y, double& z);
// Save the data to file
void save_data(std::ofstream& savefile, const std::vector<body> &system, double t);
```

There are then 3 function declarations, one for `read_init()` which reads the input `.csv` files, `read_vector3d()` which is used by `read_init()` to read in vectors components comma separated on a line, and `save_data()` which writes the data for all the bodies of the system for a specific time to the save file. These functions are implemented at the bottom of the file. You will be creating some additional functions to implement velocity Verlet and compute the energy and angular momentum. Their declarations should go here, as the comments suggest.

The file continues by defining the `main()` function. Our program will accept command line arguments which are processed here:

```
int main(int argc, char* argv[])
{
  // Checking if number of arguments is equal to 4:
  if (argc != 6) {
    cout << "ERROR: need 4 arguments - compute_orbits <input_file> <output_file> <dt> <T
    > <Tsave>" << endl;
    return EXIT_FAILURE;
  }
  // Process command line inputs:
  std::string input_file = argv[1];
  std::string output_file = argv[2];
  double dt = atof(argv[3]); // Time step
  int T = atoi(argv[4]); // Total number of time steps
  int Tsave = atoi(argv[5]); // Number of steps between each save

  std::vector<body> system; // Create an empty vector container for bodies
  read_init(input_file, system); // Read bodies from input file into system
  int N = system.size(); // Number of bodies in system
```

As the error message states the program expects an input filename, output filename, time step, number of time steps and save time steps to be specified by the user. The next three lines are important:

```
  std::vector<body> system; // Create an empty vector container for bodies
  read_init(input_file, system); // Read bodies from input file into system
  int N = system.size(); // Number of bodies in system
```

A standard `vector` container `system` is created to store `body` objects. It is initialised empty. Then the function `read_init()` reads the specified input file and populates `system` with the bodies stored there. For convenience the number of bodies it reads in is stored in `N` from the size of `system`. Next, some useful output is provided to the user on the console confirming the calculation parameters:

```
  cout << "--- Orbital motion simulation ---" << endl;
  cout << " number of bodies N: " << N << endl;
  for(int p=0; p < N; p++)
  {
    cout << "- " << system[p].get_name() << endl; // Display names
  }
  cout << "       time step dt: " << dt << endl;
  cout << "  number of steps T: " << T << endl;
  cout << "   save steps Tsave: " << Tsave << endl;
```

The rest of the `main()` function opens the output file, sets the file stream to double precision and writes the expected header information with the calculation parameters:

```
  std::ofstream savefile (output_file); // Open save file
  if (!savefile.is_open()) {
    cout << "Unable to open file: " << output_file << endl; // Exit if save file has not
     opened
    return EXIT_FAILURE;
  }
  savefile << std::setprecision(16); // Set the precision of the output to that of a
    double
  // Write a header for the save file
  savefile << dt << "," << T << "," << Tsave << "," << N << endl;
  for(int p=0; p < (N-1); p++)
  {
    savefile << system[p].get_name() << ",";
  }
  savefile << system[N-1].get_name() << endl;

  // ------------------------------------------------------------
  // ADD YOUR CODE HERE



  save_data(savefile, system, 0); // Example of saving data (for initial state here)



  // ------------------------------------------------------------

  savefile.close();
  return EXIT_SUCCESS;
}
```

The section commented for you to add your code contains one example line showing the use of the `save_data()` function. This will simply write to the output file the initial data about the bodies read in from the input file. Finally, the output file is closed and the program ends. Beneath this you will find the section for writing your function implementations, and the code for the 3 functions already defined.

As a warm-up you should compile the code using the following command

```
$ g++ -std=c++17 vector3d.cpp body.cpp compute_orbits.cpp -o compute_orbits
```

This should work and produce an executable `compute_orbits`. Running this without any command line argument will generate

```
$ ./compute_orbits
ERROR: need 4 arguments - compute_orbits <input_file> <output_file> <dt> <T> <Tsave>
```

as expected. Instead run it with the following

```
$  ./compute_orbits sun_earth.csv test_case.csv 1e-3 1 1
--- Orbital motion simulation ---
 number of bodies N: 2
- Sun
- Earth
       time step dt: 0.001
  number of steps T: 1
   save steps Tsave: 1
```

The code has successfully read in the 2 bodies, the Sun and the Earth, from the input file `sun_earth.csv`, and displays them along with the calculation parameters. You should fine a new file `test_case.csv` has appeared. We can examine it contents to find:

```
$  more test_case.csv
0.001,1,1,2
Sun,Earth
0,0,0
0,0,0,0,0,0
0.16,0.9688,-0.0001,-579.3728,92.11879999999999,-0.0012
```

As stated, the initial data about the Sun and Earth have been stored in the specified output file.

7 Exercises

Open the source file `compute_orbits.cpp`. As discussed above the code compiles and runs already, but doesn't calculate anything. However, everything you need to implement the orbital motion calculation is defined and loaded in by the code provided. The majority of this assessment is implementing this code. Hints and suggestions are given below. The following exercises break this task up into manageable pieces

---

Exercise 1. [15 marks] Write a function `compute_energy_L()` which takes `system` and computes the gravitational potential energy from Eq. (7), kinetic energy from Eq. (8) and angular momentum from Eq. (9) for each body in it. You should exploit the helpful methods of `vec` class for handling 3d vectors in these calculations. Examine `body.hpp` and you will see that the `body` class contains private members `gpe_`, `ke_` and `L_` for storing these quantities and methods for setting/getting them that you should use. Add to the `main()` function, just above the line `save_data(savefile, system, 0);`, a call `compute_energy_L()` so the energy and angular momentum are computed for the initial system. Compile and run the code with the following line

```
$ ./compute_orbits sun_earth.csv test_case.csv 1e-3 1 1
```

Examine the output file `test_case.csv`. A correct computation should yield:

```
$ more test_case.csv
0.001,1,1,2
Sun,Earth
0,-167101.389602767,576.0353796292302
0,0,0,0,0,0
0.16,0.9688,-0.0001,-579.3728,92.11879999999999,-0.0012
```

The values -167101 and 576 on the 3rd line down are the total energy and magnitude of the total angular momentum in our dimensionless units.

Exercise 2. [15 marks] Next implement an essential sub-function for the velocity Verlet method `update_acc()` which takes `system` and computes the acceleration on each body due to the gravity of all other bodies. This code will be quite similar to that used to compute the gravitational potential energy. Store the result in the private member `acc_` of `body`.

Exercise 3. [30 marks] We now move to the central task of the assessment implementing the velocity Verlet method. Implement a function `vel_verlet()` which takes the `system` and a time step `dt` and compute the update of the position and velocity of each body. You will need to use `update_acc()` at the start to compute

each bodies current acceleration. Then compute the position update according to Eq. (5). You are advised to create a new copy `new_system` of `system` inside your function to store the results of the updated position. Next, call `update_acc()` on `new_system` to get the accelerations at the new time step. Finally we then apply the velocity update in Eq. (6) which involves both current and updated quantities. Finally replace `system` with `new_system`.

Exercise 4. [10 marks] Let's now use this solver to integrate the equation of motion. In `main()` add a loop over `T` time steps which repeatedly applies `vel_verlet()` to `system`. Every `Tsave` steps calls `compute_energy_L()` and then save the data to the output file. This completes the program!

Exercise 5. [10 marks] Check your code compiles successfully. Check it produces sensible results for a simple test case by running the following command line:

```
$ ./compute_orbits sun_earth.csv test_case.csv 1e-5 1080 1
```

This will evolve the Sun and the Earth alone for 0.0108 units of time, equivalent to just slightly less than 364 days. Open the Python script `postprocess.py` and make sure that it reads the file `test_case.csv`. Run this script and when it finishes open one of the plots `orbits_closeup.png` it saves. If your code works correctly you should see Fig. 2 which shows the Earth almost completing one orbit, as expected. Include your figure in your solutions document. Examining the plot `energy.png` shows the % change in the energy over time. You should find this never exceeds `5e-5` indicating energy conservation is very good. The plot `angular_momentum.png` similarly shows the % change in its magnitude over time. This should be extremely small at around `1e-13` close to the numerical precision for all our calculations because velocity Verlet is symplectic. Confirm this behaviour with plots in your document.

Exercise 6. [10 marks] Now do further tests to get a feel for how accurate this method is by evolving for longer and with different time steps `dt`. Run the following command lines:

```
$ ./compute_orbits sun_earth.csv check_1e-3.csv 1e-3 1000 1
$ ./compute_orbits sun_earth.csv check_1e-4.csv 1e-4 10000 10
$ ./compute_orbits sun_earth.csv check_1e-5.csv 1e-5 100000 100
$ ./compute_orbits sun_earth.csv check_1e-6.csv 1e-6 1000000 1000
```

Apply `postprocess.py` to each and examine the plot `energy.png`. In your document write a short analysis of this data including some plots of `energy.png` and `orbits_closeup.png`. Why is `dt = 1e-3` in our dimensionless time a bad choice? How is the error behaving as we reduce the time-step?

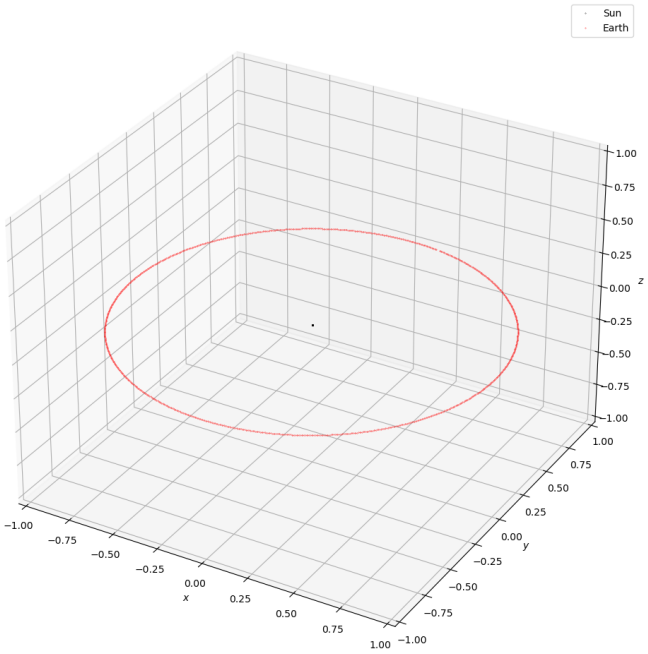

Figure 2: The motion of Earth computed from the velocity Verlet solver of Newton's equations of motion for about 364 days. We recover a near circular orbit with a period of 1 year.

Exercise 7. [5 marks] We add more bodies to the calculation by using the input file `inner_planets.csv`. Compute this 4 body problem with reasonable accuracy for close to 1 year of time with:

```
$ ./compute_orbits inner_planets.csv inner_orbits.csv 1e-6 10800 10
```

You should find that the plot `orbits_closeup.png` displays the orbits of Mercury, Venus and Earth. Add this image to your document.

Exercise 8. [5 marks] Finally use the input file `solar_system.csv` and simulate its evolution of all the planets for 2 units of dimensionless time, equivalent to 184.5 years, using an appropriate `dt`. The plots `orbits_inner.png`, `orbits_solar_system.png` should produce nice representations of the planetary orbits over this period. Examine the plot `motion_sun.png` also. Is this behaving as expected? Add these plots to your document along with some brief comments and discussion of the parameters you used.

It is worth reflecting on what have you accomplished. You have taken initial conditions for the planets and demonstrated that Newton's theory in Eq. (1) correctly describes their motion as orbiting around the Sun!