Scientific computing and software design principles

Steven Schramm
February 10, 2018

Pre-class practice exercise

This course assumes some level of experience in programming in C++ and python. The following is an exercise to test your skills, and I would encourage you to complete the exercise in both programming languages. There are some hard parts in the exercise, so you may want to start with the language that you are stronger in. If you are able to finish the exercise, then you should be ready for the course content. Please try to complete the exercise independently so that you can understand what you need to work on. The solutions are provided as a cross-check, or in case you really get stuck. Your code does not have to exactly match the solution code, but it should produce the same results and use a similar class-based structure.

Note that the solution intentionally does not always follow ideal coding practices. We will discuss some potential improvements during the course. Also note that I tend to use camelCase everywhere, even though the recommended convention in python is to use underscores. You are free to use whichever convention you prefer, although it's best to avoid mixing conventions whenever possible.

0 N-body simulation of the Coulomb force

The goal of this exercise is to create an n-body simulation of the Coulomb force, allowing for us to study simple dynamic electromagnetic systems in a numerical manner. This is very useful as it avoids the need to have analytic solutions to any particular problem, so long as you use sufficiently small time steps. In the case of electromagnetism, analytic solutions do exist for simple particle interactions, and so this is not strictly necessary. However, these quickly become difficult to scale to many-object systems, while numerical solutions continue to function with minimal changes. Furthermore, there are many equations which do not have a known analytic solution, and where such numerical simulations are our best means of studying the behaviour of the universe.

In numerical systems, it is very important to control the initial conditions of the system. Small differences can lead to very different behaviour, depending on what is being studied. Some of the examples in this exercise change drastically with different initial conditions. As such, the "main" program is provided for you (`nBodySim.cpp` and `nBodySim.py`). It is your job to complete the code that these two driving programs depend on. You should never have to modify either `nBodySim` program, although if you prefer different naming conventions (such as underscores in python), you are welcome to do so. Please just ensure that you do not change any of the actual functionality of these driving programs to ensure that you can obtain the same final results.

The two driving programs both run the same six simulations of increasing complexity. While all of the results should be possible to understand from the concepts of the Coulomb force, their exact behaviour may (or may not) be a bit surprising at first. The programs create output files named `partN.dat`, where `N` goes from 1 to 6 inclusive. If you have gnuplot installed (or if you have access to the UniGe DPNC cluster or CERN lxplus, both of which have gnuplot) you will be able to plot the data files using the scripts provided (named `makePlotN.sh`, where `N` corresponds to the data file that it plots). If everything is done correctly, the plots should look like the figures shown at the end of this file.

Please note that you will need to use double-point precision throughout the program (`double` in C++, default in python or float64 in numpy). Some of the simulation steps can otherwise run into floating-point precision limitations.

Steps to follow:

Preparation

0. Download the files from https://cernbox.cern.ch/index.php/s/DH94OaiVpn3IOmC. The password is "UniGe".
   - Again, the results and full code solutions are provided to help you if needed, but please try to complete the exercise without looking at them!
   - For now, you should only look at `Code/cpp/nBodySim.cpp` and `Code/python/nBodySim.py`.
   - Once you have results, you can plot them using the scripts provided in `Results/scripts`.
   - For those without much C++ experience, you can compile the provided code including solutions using "`g++ -Wall --std=c++11 -o nBodySim ./*.cpp ./*.h`". This will compile all of the files ending with ".cpp" and ".h" into an output executable file named nBodySim. You can run it with "`./nBodySim`" after compiling.
   - For those without much python experience, you don't need to compile the program. You can simply run it using "`python nBodySim.py`".

Particle class

1. Create a class named `Particle` which describes the particle: the charge, the mass, three spatial positions (x, y, and z), and three velocity components (v_x, v_y, v_z). We will only accept integer charges, while floating point values are accepted for the other components. In the driver program, the charge will be specified as integer multiples of the elementary charge, while the mass will be specified in kg.
2. Define the constructor, which allows for the user to specify the charge, mass, and spatial positions in the order mentioned above. In this case, the initial velocity components should be zero. Verify that the mass is positive.
3. Define another constructor or extend the previous constructor (depending on the programming language) which allows the user to specify the charge mass, spatial positions, and velocity components in the order mentioned above. Verify that the mass is positive and the speed is below that of light.
4. Define class member interface functions that return the values of the eight parameters.
5. Add a class function named `getDistance` which returns the distance of the particle with respect to the origin (0,0,0).
6. Add a class function named `getSpeed` which returns the speed of the particle (magnitude of the velocity three-vector).
7. Add a function named `updateKinematics` which allows the user to change all six of the spatial and velocity components of the particle at once.
8. Add an output operator that translates a particle into a string form as appropriate to C++ and python.
    - For C++, the following should work: "`std::cout << particle << std::endl`", where `particle` was previously declared as a non-pointer instance of the Particle class.
    - For python, the following should work: "`print str(particle)`", where `particle` was previously declared as an instance of the Particle class.
    - **Important**: the particle should be written out as follows: " x y z v_x v_y v_z mass charge", where each quantity is printed out separated by a space from the previous quantity. Note that the output should only print out the values, not a string saying "x=5, y=5, ..." or similar. This is critical to allow the plotting scripts to interpret the output format. For an example of the expected format, look at the file `Results/dat/part1.dat` in your favourite text editor.

ParticleSystem class

9. Create a class named `ParticleSystem`, which contains a vector or list of (pointers to) particles as appropriate. This should be the only variable that belongs to the class.
10. Create the constructor, which initializes the empty vector/list and does nothing else.
11. Create the destructor in C++, which properly frees the Particles. That is, the ParticleSystem is responsible for managing the memory associated with Particles.
12. Add a class function named `addParticle`, which takes a (pointer to a) Particle and puts it in the vector/list.
13. Add a function named `writeToFile` that prints the entire contents of the vector/list of Particles to an output file.
    - This should make use of the method that you added in step 8 to print each particle
    - This should take an argument of an already opened file. It should not open/write/close a file on each call.
    - **Important**: the full set of particles should be written out on a single line of the file, not on separate lines. This is needed for the plotting scripts to work. However, after printing out all of the particles, there should be a line break (start of a new line).
14. Add a function named `evolve`. This should take a single floating point input representing the time step under which to evolve the system, and the input is typically sub-millisecond time steps.
    - This is the most complex part of the program, and where the whole simulation occurs. If you get stuck, you are welcome to look at and copy the solution for this part, as I understand that the simulation code is hard to put together for the first time. However, please only do this for one of C++ or python, and then use your own code as a guide for writing the program in the other language rather than looking at the second solution.
    - Remember that we are using units of elementary charge = 1. Under this choice of units, the Coulomb constant is $k_e = 2307077.5 \times 10^{-34}$.

- Recall that every particle acts on every other particle, either repulsively or attractively. Technically there could be neutral particles and the code would still work, but those aren't very interesting when we only have the Coulomb force in the simulation.
- As some advice, I followed these steps (indentation is on purpose):
  1. Calculate the updates
     ```
         Calculate the force vector between all pairs of particles
         Calculate the projection of that force vector on the $x$, $y$, and $z$ axes for all pairs of particles
     Calculate the new velocity for each particle using the force vector of all other particles acting upon it
     Calculate the new position for each particle using the force velocity that was just calculated
     ```
  2. Apply the updates. Note that this must happen **after** calculating all of the updates. We need to use the position at time t_i to calculate the evolution to t_{i+1}. If we apply updates as we calculate them, then we are mixing the two time points in the calculation of the evolution.

15. Add a function named `evolveAndWriteToFile`. This should take several arguments as described below. The purpose of this function is to make use of the previously defined `evolve` and `writeToFile` functions.
    - Argument 1: The number of times to `evolve` the system. This value is typically in the thousands.
    - Argument 2: The number of time-steps to ignore before calling `writeToFile`. That is, if the first argument is 400 and the second argument is 100, then we should write to the file 5 times (before evolving the system, after 100 steps, after 200 steps, after 300 steps, and after 400 steps). The "modulo operation" may be helpful here. In the driver program, this value is always ten.
    - Argument 3: The size of the time step. In the driver program, this value is always 0.1 ms. Combined with the previous option, that means that we are writing the state of the system to the file every millisecond.
    - Argument 4: The name of the file to write to. That means that you have to use this name to open an output file once, and then that open file should be passed to `writeToFile` each time it is called. At the end of the function, don't forget to close the file.
16. Compile, debug, and run the program!
17. If your code is working, you should be getting large output files named "partX.dat", and you should be able to use the plotting scripts to reproduce the figures shown below.
18. Repeat all of the above for the second programming language (C++ and python).

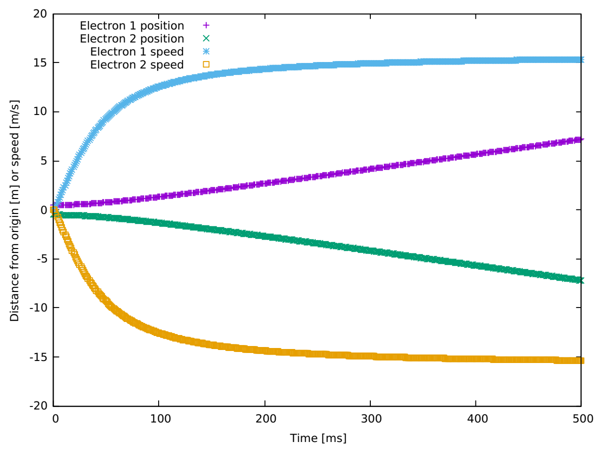

Figure 1: The behaviour of two electrons which start out close to each other at rest, and rapidly move away. Note how the velocity grows quickly and then plateaus as the two electrons are far enough away that the Coulomb repulsion force becomes negligible.

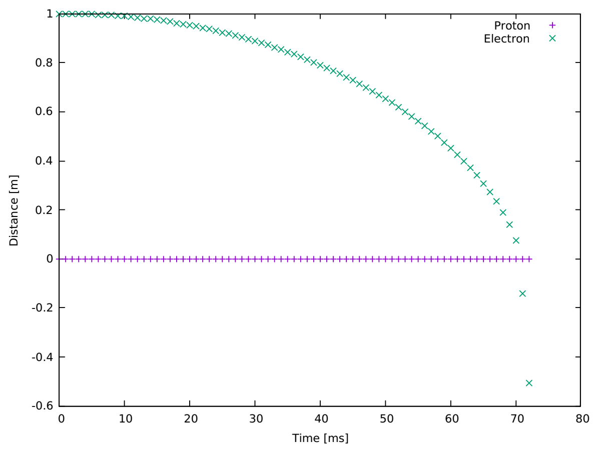

Figure 2: The behaviour of an electron (green) and proton (purple) which start at rest. The electron approaches the proton, while the proton barely moves due to its large mass. The electron passing through the proton is an example of simulation limitations, as the Coulomb force alone cannot explain what happens.

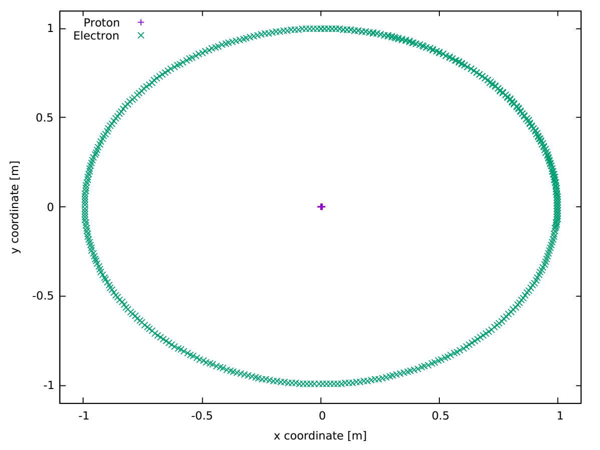

Figure 3: An electron (green) orbits a proton (purple), thanks to its initial velocity.

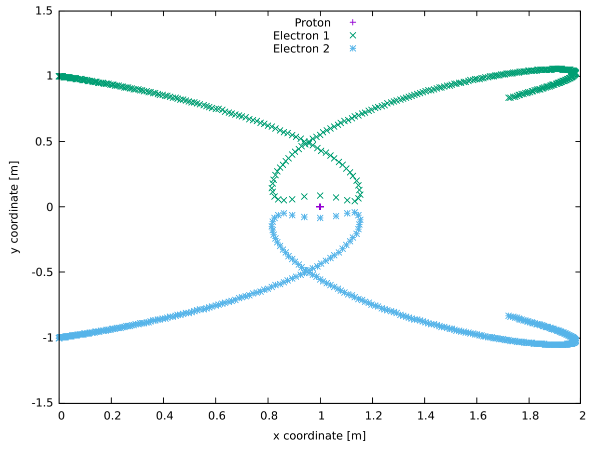

Figure 4: A pair of electrons (green and blue) start at rest and identically far from a proton. They are attracted to the proton, but repelled by each other. This pushes them initially past the proton, then they loop back, before again being in the dominantly repulsive regime. They then repel strongly and fly apart.

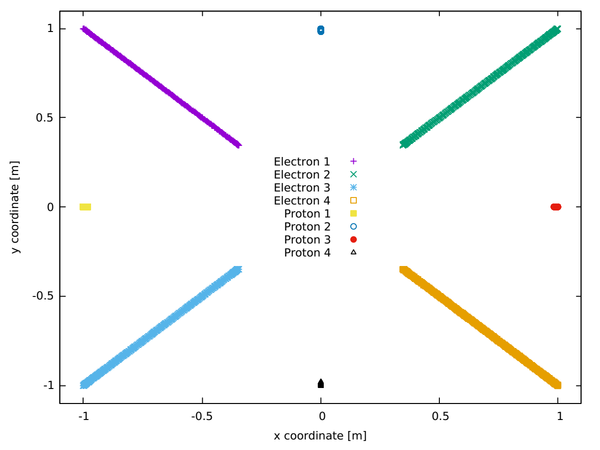

Figure 5: A grid of four electrons and four protons, spaced equally apart. The electrons start with an inward force from the set of protons. However, they eventually encounter the repulsive force of the other electrons, and thus back away. This behaviour continues, with the electrons oscillating back and forth.

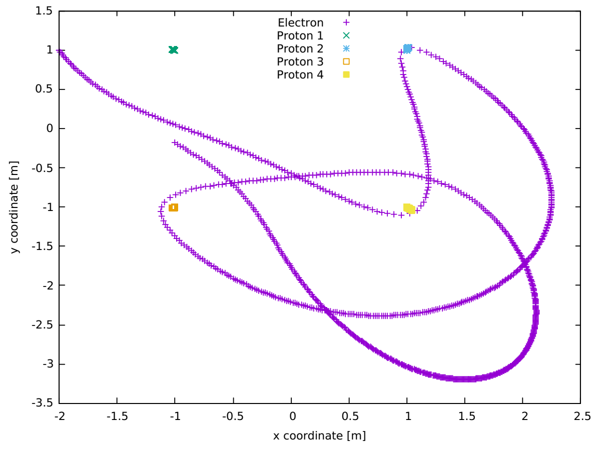

Figure 6: A single electron (purple) is fired into a grid of four equally spaced protons. With the right initial conditions, the electron has a very erratic behaviour as it feels the pull of each individual proton, thus altering its original trajectory in a very non-trivial manner.