PHAS0100 2021/22: Research Computing with C++

Assignment 2: Gravitational N-body Simulation

Late Submission Policy: Please be aware of the UCL Late Submission Policy, which is here: https://www.ucl.ac.uk/academic-manual/ *(chapter 4, section 3.12)*

Extenuating Circumstances
The course tutors are not allowed to grant Extenuating Circumstances for late submission. You must apply by submitting an Extenuating Circumstances form which can be found on the 2021/22 EC information website. The Department EC Panel (DECP) will then decide whether or not to grant extenuating circumstances and email the module lead confirming that Extenuating Circumstances have been granted and providing a revised submission date.

Assignment Submission
For this coursework assignment, you must submit by uploading a single Zip format archive to Moodle. You must use only the zip tool, not any other archiver, such as .tgz or .rar. Inside your zip archive, you must have a single top-level folder, whose folder name is your student number, so that unzipping this archive reveals this single folder. This top-level folder must contain all the parts of your solution. Inside the top-level folder should be a git repository named PHAS0100Assignment2. All code, images, results and documentation should be inside this git repository. You should git commit your work regularly as the exercise progresses.

You should clone the following repository and use as the starting point for the coursework: https://github.com/UCL/PHAS0100Assignment2

Due to the need to avoid plagiarism, do not use a public GitHub repository for your work - instead, use git on your local disk (with git commit but not git push), and **ensure the hidden .git folder is part of your zipped archive** as otherwise the git history is lost. We will be providing access to a private GitHub classroom repository that you can use as a remote backup option – there will an email/moodle announcement with details on this. It is important you do not push to a public GitHub repository as pieces of identical (or suspiciously similar) work will both be treated as plagiarised. Whilst it is fine to look up definitions and general documentation for C++/libraries online it is not acceptable to use code taken from the internet or other sources or provided by someone else for the problem solutions. The exception to this is the initial PHAS0100Assignment2 project that should be used as a starting point for your project.

We recommend you use Visual Studio Code IDE with the Ubuntu 20.04 docker image for development (the correct Dockerfile is included for you in the above repository). This is not a requirement but if you do use a different setup then you need to ensure that your code compiles and runs on Ubuntu 20.04 and with g++ 9.3.0 (enforcing C++17) and CMake 3.16.3 as this is the environment the markers will use to test the code.

Marks will be deducted for code that does not compile or run. Marks will also be deducted if code is poorly formatted. You can choose to follow your own formatting style but it must be applied consistently. See the Google C++ style guide for a good set of conventions regarding code formatting. In general, if adding to an existing project it is best to follow the existing style.

It is important that you follow the file/folder/executable structure of the PHAS0100Assignment2 of the starting project as well as the form and name of any functions/classes/executables described in any questions below. This is because we will be automating basic checks of the submitted work so if you have not followed the pre-defined structure those checks will fail, and you will lose marks. For a similar reason you should make sure your unit tests are included when `ctest` is run by ensuring each new test file you create has an appropriate `add_test` entry in `PHAS0100Assignment2/Testing/CMakeLists.txt`

Preliminaries
In this assignment you will create a simple gravitational N-body simulation and use it to first model the dynamics of the Solar System and then a larger group of astronomical bodies. You will then apply the

concepts covered in the HPC part of the course to benchmark and improve the performance of your program using shared memory parallelism and OpenMP. When designing, writing and testing your code you should continue to put into practice the material on Modern C++ and Code Quality covered in the first part of the course, including: cmake; unit testing with catch2; error handling with exceptions; OO-design concepts: encapsulation, abstraction, inheritance vs composition, dependency injection; avoid using raw pointers; clear code formatting with effective in-code comments.

Reporting Errors
You can post questions to the Moodle Q&A forum, if you believe that there is a problem with the example CMake code, or the build process in general. To make error reporting useful, you should include in your description of the error: error messages, operating system version, compiler version, and an exact sequence of steps to reproduce the error. Questions regarding the clarity of the instructions are allowed, but obvious “Please tell me the answer” type questions will be ignored.

Typos/errors/clarifications:
Highlighted text is used where a typo/error/clarification has been made following the initial release of the assignment. When this happens there will also be a moodle announcement.

Important: Read all these instructions before starting coding. In particular, some marks are awarded for a reasonable git history (as described in Part C), that should show what you were thinking as you developed. The maximum number of marks available is indicated for each section and question with the grade being a percental out of 70 marks total.

Part A: Solar System Simulator (30 marks)

In this first part of the assignment, you will write and test a simple N-body simulator that models the dynamics of the Solar System.

1. a) First you will create a `Particle` class that holds a position and velocity, can have an acceleration applied, and will update its position and velocity when it is stepped through time. The class should provide at least the following three public methods:

   - `Eigen::Vector3d getPosition()` that returns the current position of the object,
   - `Eigen::Vector3d getVelocity()` that returns the current velocity of the object,
   - `void integrateTimestep(Eigen::Vector3d acceleration, double timestep)` which updates the position and velocity as though a duration of length timestep has passed.

   Consider whether any of these functions or arguments should be const, and if so modify them to be so.

   For acceleration $\boldsymbol{a} = (\boldsymbol{a}_x, \boldsymbol{a}_y, \boldsymbol{a}_z)$ and a timestep δt the position and velocity vectors $\boldsymbol{x} = (\boldsymbol{x}_x, \boldsymbol{x}_y, \boldsymbol{x}_z)$ and $\boldsymbol{v} = (\boldsymbol{v}_x, \boldsymbol{v}_y, \boldsymbol{v}_z)$ should be updated according to:

   $$\begin{aligned}\boldsymbol{x}(t+\delta t) &= \boldsymbol{x}(t) + \boldsymbol{v}(t)\,\delta t\\ \boldsymbol{v}(t+\delta t) &= \boldsymbol{v}(t) + \boldsymbol{a}(t)\,\delta t\end{aligned}$$

   b) Write unit tests to confirm that the particle moves as it should when:
      a. There is no acceleration $\boldsymbol{a} = \boldsymbol{0}$
      b. There is a constant acceleration $\boldsymbol{a} = const$
      c. When a fictitious centripetal acceleration $\boldsymbol{a} = -\boldsymbol{x}(t)$ is applied i.e. for a body at position $\boldsymbol{x}(t)$ the acceleration at time t is towards the origin and has magnitude $|\boldsymbol{x}(t)|$

   Hints: When calculating the positions and velocities you should use the `Eigen::Vector3d` type that supports vector addition, subtraction, scalar multiplication etc. For this you will need to `#include <Eigen/Dense>` and examples of how to use can be found here. `Eigen` also provides a useful function to compare vectors, given two vectors `Eigen::Vector3d v1` and `Eigen::Vector3d v2` you can use the `v1.isApprox(v2)` to check if they are equal to each other. There is an optional second argument for `isApprox` that controls the precision of the comparison, so `v1.isApprox(v2, 0.01)` will compare entries with a precision of 0.01.

A simple way to test movement under the fictitious centripetal acceleration is to create a `Particle` with position `(1, 0, 0)` and velocity `(0, 1, 0)` and confirm that after stepping through 2π's worth of time (number of steps * step length) that the particle's final position and velocity are close to the initial values. Note that this is not a typical centripetal acceleration, but it will lead to circular motion for the given initial positions.

[4 marks implementation, 4 marks unit tests, **8 marks total**]

2. a) Now create a new `MassiveParticle` class that acts under the effect of gravity. It needs to move under acceleration like the first class but with the acceleration now being due to its gravitational attraction to other `MassiveParticle`'s. This will require the class to have an extra piece of data, the gravitation parameter $\mu = Gm$, where G is the gravitational constant and m is the mass of the particle. The class should also have a pair of functions to add and remove instances of the same class. These will define the bodies to which it is being gravitationally attracted to. Consider whether copies of the other instances should be stored, or whether pointers should be used, and if so, what type.

   The `MassiveParticle` class should have the following member functions:

   - `double getMu()` which returns the gravitational parameter,
   - `void addAttractor(`*`a massive particle`*`)` to add an attractor,
   - `void removeAttractor(`*`a massive particle`*`)` to remove an attractor,
   - `void calculateAcceleration()` which calculates the gravitational acceleration on this particle based on the list of attractors and stores the result as a `Vector3d` data member,
   - `void integrateTimestep(`*`timestep`*`)` which updates the position and velocity as though a duration of length timestep has passed.

   Note that you need to decide on the specific form and type for the arguments. Again, consider the `const` ness of these functions and arguments, and apply the `const` modifier to any that would benefit.

   The class should implement the same equations of motion as the `Particle` class, with the acceleration now being calculated in the `MassiveParticle`'s `calculateAcceleration` by summing the gravitational attraction from all the bodies included with the `addAttractor` function. Given the position $\boldsymbol{x}$ for the current body and of $\boldsymbol{x_i}$ for attractor i, the total acceleration should be:

   $$\boldsymbol{a} = \sum_i \frac{-\mu_i}{r_i^2}\hat{\boldsymbol{r}}_i, \text{ where } \hat{\boldsymbol{r}}_i \text{ is the normalised } \boldsymbol{r}_i = \boldsymbol{x} - \boldsymbol{x_i}$$

   where the sum over i is over all attractors added to the body.

   The result of `calculateAcceleration` should be stored as a data member in `MassiveParticle` and should be used as the acceleration term in the `MassiveParticle ::integrateTimestep` definition. N.B. it is important for the Part B OpenMP parallelisation that a user of the class needs to first call `calculateAcceleration` and then `integrateTimestep` as two separate steps i.e. don't include `calculateAcceleration` inside the definition of `integrateTimestep.`

   b) Test that the new class:

   a. Still models linear motion correctly i.e. that a single `MassiveParticle` with no attractors will travel with a constant velocity
   b. Will allow two bodies to gravitationally attract each other. To test this, create two bodies with $\mu = 1$, and add each to the other as an attractor. Use the following initial conditions $\boldsymbol{x_1} = (1,0,0)$, $\boldsymbol{v_1} = (0,0.5,0)$, $\boldsymbol{x_2} = (-1,0,0)$, $\boldsymbol{v_2} = (0,-0.5,0)$. Then test that the two bodies remain at a distance of approximately 2 over a full orbit.

Hints: note that the new `MassiveParticle` class needs to model acceleration like the `Particle` class but its `void MassiveParticle::integrateTimestep` function does not require an acceleration argument as the acceleration is now calculated based on the gravitational attraction to the list of attractors. Consider how you might reuse the behavior of the `Particle` class, is it more appropriate to use inheritance or composition.

[5 marks implementation, 3 marks unit tests, **8 marks total**]

3. Using the `MassiveParticle` class you developed you will now build an application to model the motion of the Sun and planets in the Solar System:
   a. Create a command line app that can be run as `./bin/solarSystemSimulator` from the build directory. It should have arguments to control the step-size and the length of time (or number of timesteps) to simulate.
   [2 marks]
   b. The command line app should print a useful help message to tell the user what arguments to use if the -h or –help switch is used. With zero command line arguments, the program should not crash, and should respond with the help message.
   [1 marks]
   c. First the application should create a set of `MassiveParticle`'s corresponding to the major bodies (the Sun plus 8 planets) within the Solar System and with appropriate initial conditions. The initial positions, velocities and values for μ of the Solar System bodies are based on the epoch of 2021-01-0T00:00:00Z and are provided in `Code/Lib/nbsimSolarSystemData.ipp`
   [3 marks]
   d. For each `MassiveParticle` representing a solar system body add all other `MassiveParticle` bodies to their list of attractors i.e. make each body feel the gravitational attraction of all other bodies.
   [2 marks]
   e. Next you need to implement the evolution of the solar system with time. To do this you should create an outer loop to loop over timesteps. Then within this loop there should be two separate for loops, each looping over all of the Solar System bodies:
      i. In the first for loop the code should call the `MassiveParticle::calculateAcceleration` method to update the gravitational accelerations for all bodies given their current positions.
      ii. Then in the second for loop where the position and velocity of each body is updated using their `MassiveParticle::integrateTimestep` method.
   [3 marks]
   f. Add appropriate messages to screen summarising the position of the solar system bodies at the start and end of the simulation. Use a timestep of 0.000274 years (this is ~0.1 days) and then run the program for 1 year of simulated time.
   [3 marks]

   To demonstrate f. you should take a screengrab of the output showing the final positions of each body and commit before submitting.

Hints:

You can use basic C++ command line parsing:
http://www.cplusplus.com/articles/DEN36Up4/
Or you could try integrating a command line parser such as:
https://github.com/CLIUtils/CLI11(which is header only).

To access the Solar System data you can `#include "nbsimSolarSystemData.ipp"` in the main app or the code where you need it. An example of how you can access the contents of the array is:

```
int ibody = 0;
std::cout << nbsim::solarSystemData.at(ibody).name << std::endl;
std::cout << nbsim::solarSystemData.at(ibody).mu << std::endl;
std::cout << nbsim::solarSystemData.at(ibody).position << std::endl;
std::cout << nbsim::solarSystemData.at(ibody).velocity << std::endl;
```

Range based for loops will also work.

When implementing the functionality to generate the positions and velocities for the solar system bodies you will also be marked based on OO concepts, so you should consider whether it would make sense to implement using classes/functions etc. In Part C you of the assignment you will need to implement some different functionality to generate a different set of starting conditions and when designing your code you should consider the ease with which new types of generators can be added.

Here is some example pseudo-code to demonstrate the required loop structure to evolve the Solar System bodies:

```
// outer loop over time
for(loop over timesteps)
{
  // one for loop to calculate accelerations
  for(loop over all bodies)
  {
    // call each body's calculateAcceleration
  }

  // then a second for loop to update positions and velocities
  for(loop over all bodies)
  {
    // call each body's integrateTimestep
  }
}
```

[a-f, **14 marks total**]

Part B: Improving the Simulation (30 marks)

In this part of the assignment you will quantify the accuracy of the simulation and benchmark its performance using timers. You will then improve the performance of the simulation using OpenMP to parallelise the relevant parts.

4. As there is no energy being added to or removed from the system of bodies the total energy $E_{total} = E_{kinetic} + E_{potential}$ should remain constant as the simulation evolves. However given the simplistic numerical integration (see 1.a) the conservation of total energy will depend on the step size. To check this you should calculate:
    a. The kinetic energy $E_{kinetic}$ following:

    $$E_{kinetic} = \frac{1}{2}\sum_{\mathrm{i}} \mu_i |\boldsymbol{v}_i|^2$$

    where $\boldsymbol{v}_i$ and μ_i are the velocity and gravitation parameter of the i'th body respectively, and the loop over i is over all solar system bodies,

    b. and the gravitational potential energy of the system energy $E_{potential}$ following:

    $$\boldsymbol{E}_{potential} = -\frac{1}{2}\sum_{i}\sum_{j\neq i}\frac{\mu_i\mu_j}{|\boldsymbol{x}_i - \boldsymbol{x}_j|}$$

    where $\boldsymbol{x}_i$ is the position of the i'th body, the first loop is over all solar system bodies and the second loop is over all except for $j = i$ to avoid calculating a self-interaction term.

You should write code to calculate $E_{kinetic}$, $E_{potential}$ and E_{total} and output it to screen at the beginning and end of the simulation. Simulate 100 years of time with various step sizes (5-10 different values is sufficient) and summarise the results in the README.md the simulation evolves.

Notes: In the above we use μ instead of mass when calculating the kinetic and potential energies. The equations implicitly include the gravitational constant G as a constant factor in μ. This means the units for energy are not SI units but a function of years, AU and kg, but they are internally consistent and will result in sensible numerical energy values in the range -1 to 1.

[**5 marks total**]

5. Next you should benchmark the runtime of your code using `std::chrono::high_resolution_clock` and `std::clock`, see the example here: https://en.cppreference.com/w/cpp/chrono/c/clock

   Benchmark the time it takes your simulation to run for 100 year's worth of simulated time and for a number of different timestep sizes (again, 5-10 different values is fine). Add a summary of the results to the README.md file, this can be combined with the results from question 4. Based on these choose a value of timestep that is a good balance between simulation run time (a few minutes is reasonable) and accuracy (see question 4) and document in the README.md.

   N.B. when doing this you do not want to the benchmarking time to be dominated my many `std::cout`'s so if you have any debugging/informational couts anywhere inside the loop over time you should have a way to toggle them on/off when benchmarking. If you need this consider using using pre-processor directives to toggle on/off the `std::cout`'s, you could even link these to the `-DCMAKE_CXX_FLAGS_DEBUG=Debug` cmake build type.

[**4 marks total**]

6. In order to show your program parallelises appropriately, we must first increase the size of the system (i.e. the number of bodies) by many orders of magnitude. This requires the generation of many random particles. For a particle with random radius r and initial angle θ a stable orbit around a much more massive particle at the origin can be generated by the equations

$$\begin{aligned} r_x &= r\sin(\theta) \\ r_y &= r\cos(\theta) \\ v_x &= \frac{-1}{\sqrt{r}}\cos(\theta) \\ v_y &= \frac{1}{\sqrt{r}}\sin(\theta) \end{aligned}$$

   You should create a function or class method which can return a vector of particles, the first of which represents a central body with large mass, zero velocity and positioned at the origin. The remaining particles should have random positions in stable orbits (as given by the velocity relations above). You should then use your generator to initialise a simulation of 2000 particles.

   Hints:

   If you want to check your generator is working sensibly, a good test that would result in understandable results would be to generate a random solar system of ten bodies with radii selected from a uniform distribution between 0.4 AU, the approximate radius of Mercury and 40 AU, the approximate radius of Neptune.

   Implementing different initial conditions (i.e. choosing between the different generators) can be done by creating a single initial condition generator class defining the interface and subclassing that to implement specific initial conditions. Consider how your design might be extended by other developers wishing to add new initial condition generators? Also consider how you might use polymorphism to ensure your code does not depend on a single initial condition generator.

[**6 marks total**]

7. Next you should use OpenMP to parallelise your simulation to improve its performance. Use the 2000 particle initial conditions as the use-case and choose an appropriate step size and simulation time that results in a run time of between 1 and 5 minutes before parallelisation.

   a. Parallelise your simulation using OpenMP. Think carefully about which parts of the simulation can be parallelised. For example, the outer loop over time cannot be parallelised as each subsequent timestep depends on the output of the previous timestep. Also, it is important that within each timestep the loop over bodies to calculate forces needs to be completed for all bodies before the subsequent loop over bodies to update their position and velocities starts.

   [7 marks]

   b. Benchmark again, this time with respect to the number of threads using the `OMP_NUM_THREADS` environment variable to set the number of threads. Provide a summary in the README.md of benchmarking results before/after parallelisation

   [3 marks]

   c. For large numbers of particles, the calculations of the kinetic and potential energies implemented in parts 4.a and 4.b may become a bottleneck. To parallelise these, you should use an OpenMP reduction. See the Week 8 lecture notes for examples of OpenMP reductions.

   [4 marks]

   N.B. marks will be based on the implementation of parallelisation and the ability to benchmark and not on actual performance achieved so you will not be penalised for having a machine with a low number of cores. You should still increase `OMP_NUM_THREADS` above the number of cores on your machine and summarise the results. If using docker you can increase the number of cores docker has access to through the docker desktop application under: settings -> resources -> CPUs.

[**14 marks total**]

Part C: Additional Considerations (10 marks)

1. Nice git commit log.

   [3 marks]

2. Clear code formatting and effective in-code commenting.

   [4 marks]

3. Update the front page README.md to give clear build instructions, and instructions for use.

   [3 marks]