HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2022
ASSIGNMENT 3: THE GRAVITATIONAL N-BODY PROBLEM

It is recommended to do Labs 5-7 before this assignment. It is important that you submit the assignment in time. **See the deadline in Studium.**

1. The N-body problem

1.1. **Governing equations.** Newton's law of gravitation in two dimensions states that the force exerted on particle i by particle j is given by

$$\mathbf{f}_{ij} = -\frac{Gm_i m_j}{r_{ij}^3}\mathbf{r}_{ij} = -\frac{Gm_i m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}\,,$$

where G is the gravitational constant, m_i and m_j are the masses of the particles, r_{ij} is the distance between the particles, and $\mathbf{r}_{ij}$ is the vector that gives the position of particle i relative to particle j. $\hat{\mathbf{r}}_{ij}$ is the normalized distance vector. If $\mathbf{e}_x$ and $\mathbf{e}_y$ are unit vectors in the x and y directions, respectively, then

$$\mathbf{r}_{ij} = (x_i - x_j)\,\mathbf{e}_x + (y_i - y_j)\,\mathbf{e}_y\,,$$

so that

$$r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$$

and

$$\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/r_{ij}\,.$$

Given a distribution of N particles, a straight-forward calculation of the force exerted on particle i by the other $N-1$ particles is given by (using C-style indexing)

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}.$$

Note that by using another formula for the pairwise forces, other types of particle systems governed by Newtonian mechanics can be modeled, e.g in electrodynamics and molecular dynamics.

There is a built-in instability in the given formulation when $r_{ij} \ll 1$. To deal with this, we introduce a slightly modified force that corresponds to so-called Plummer spheres as follows:

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij} + \epsilon_0)^3}\mathbf{r}_{ij}.$$

Date: March 12, 2022.

where ϵ_0 is a small number (we will use 10^{-3}). This effectively caps the maximum force between two particles and acts as a smoother for the simulation.

Using the *symplectic Euler*[1] time integration method, the velocity $\mathbf{u}_i$ and position $\mathbf{x}_i$ of particle i can then be updated with

$$\begin{aligned} \mathbf{a}_i^n &= \frac{\mathbf{F}_i^n}{m_i}\,, \\ \mathbf{u}_i^{n+1} &= \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n\,, \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1}\,, \end{aligned}$$

where Δt is the time step size and $\mathbf{a}_i$ is the acceleration of particle i. Note that this straightforward solution, which we will use in this assignment, becomes computationally expensive for large values of N, since the total number of operations required to compute the forces on all N particles at each time step grows as $\mathcal{O}(N^2)$. There are algorithms that can be used to improve the scaling, but in this assignment we stick with the straightforward $\mathcal{O}(N^2)$ solution.

2. Problem Setting

In this assignment you will implement a code that calculates the evolution of N particles in a gravitational simulation. You will calculate the motion of an initial set of particles that approximates the evolution of a galaxy.

The simulation is done in two spatial dimensions, so the position of each particle will be given by two coordinates (x and y). There will be N particles with positions in a $L \times W$ dimensionless domain (Use $L = W = 1$ in your simulations). This means that the x and y values will be between 0 and 1.

The particle masses and intitial positions and velocities of the particles will be read from a file when the program starts, then simulated for a given number of timesteps, and in the end the final masses, positions and velocities will be written to a results file. (The masses will of course be the same as in the beginning, but they will be written to the results file anyway so that it can be used as input for another simulation later.)

Since fewer bodies have less mass with which to stick together, we want gravity to scale inversely with the number of bodies. Set $G = 100/N$, $\epsilon_0 = 10^{-3}$ and timestep $\Delta t = 10^{-5}$.

3. Files included with the assignment

Download the `Assignment3.tar.gz` file from Studium and unpack it. Inside it you find four directories:

- `compare_gal_files` : a program that can be compiled into a separate executable that can be used to compare different simulation results; you can use this to check the correctness of your results.

[1]The symplectic Euler method is a version of the explicit Euler scheme which is the most basic example of a partitioned Runge-Kutta method (PRK). For the standard Euler scheme, the formula for $\mathbf{x}_i^{n+1}$ would read $\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t \mathbf{u}_i^n$. The symplectic Euler scheme preserves global properties of the gravitational system (e.g. total energy), while the standard Euler does not.

- `input_data` : a set of input files that you can use when testing your program.
- `ref_output_data` : a set of reference output files that you can use to check that your results are correct. The `compare_gal_files` program can be used to compare your result files to the reference output files.
- `graphics` : a small example code showing how to use graphics routines in the "X" window system commonly used in Linux. Being able to plot the galaxy evolution on the screen is a nice way of seeing that the code works, and helpful for debugging. The small test code given here is just an example, if you know of another way of doing graphics that you prefer, use that instead. To make the "X" graphics code work you may need to install some additional library packages related to X development, e.g. `libX11-devel` in Fedora or `libx11-dev` in Ubuntu.

4. Assignment

You must write a program that solves the given equations of motion for a galaxy with the given initial conditions. Your final program should be written as efficiently as possible.

Start by writing a code that gives correct results, then think about how it can be optimized. Remember to use both compiler optimization flags and your own code changes. Try to apply the optimization techniques you have learned about in the course so far.

Input to your program: The program that you create should be called `galsim` and it should accept five input arguments as follows:

```
./galsim N filename nsteps delta_t graphics
```

where the input arguments have the following meaning:
`N` is the number of stars/particles to simulate
`filename` is the filename of the file to read the initial configuration from
`nsteps` is the number of timesteps
`delta_t` is the timestep Δt
`graphics` is 1 or 0 meaning graphics on/off.

If the number of input arguments is not five, the program should print a message about the expected input arguments and then stop.

Output from your program: The program `galsim` should simulate the stars/-particles for the given number of timesteps. If run with `graphics` set to 1, the program should show the stars moving on the screen during the simulation. In the end, the final positions and velocities of the stars should be saved to a file called `result.gal` using the same binary file format as for the input file.

File format used for initial configuration (and storage of result): The binary file format used will simply consist of a sequence of `double` numbers giving the mass, position, velocity, and "brightness" of each particle. The "brightness" is a number that is not directly used in our simulations, but that should be kept and

stored together with each particle in the result file, so that the result file has the same format as the input file. Both mass and "brightness" for each particle should be the same in the result file as in the input file. The order of the numbers in the binary file format is as follows:

```
particle 0 position x
particle 0 position y
particle 0 mass
particle 0 velocity x
particle 0 velocity y
particle 0 brightness
particle 1 position x
particle 1 position y
particle 1 mass
particle 1 velocity x
particle 1 velocity y
particle 1 brightness
...
```

So, the total file size will be `N*6*sizeof(double)`. Note that the same format should be used for the output file with the final result from the simulation. This means that a result of a previous simulation can later be used as starting point for a new simulation.

Implementation: Implement the straightforward $O(N^2)$ algorithm and verify for some small cases that your code works properly (about small test cases, see below under "Input data"). When you are confident that your code works correctly, move on to measure performance and consider possible optimizations.

Note that we are so far in this course only considering optimization of *serial* code — in case you already know how to parallelize using threads, don't do that here; this assignment is about writing an efficient serial code. You will get the chance to work with threaded programs later in the course.

Graphics is nice to have and recommended, but not mandatory. If you have difficulties making the graphics routines work on the computer you are using, it is OK to skip the graphics. The important thing for us is performance, and not how to use graphics routines. But it is strongly recommended use graphics if you can, it is a nice way to illustrate what you are doing and makes it easy to see if your code works properly for the small test cases. The graphics routines do work on the Linux computers[2] in the lab rooms, so you can at least plot things there even if you are unable to make it work on your own computer.

Portability: You can choose freely which computer you want to use for your performance testing in this assignment — you can use one of the university's Linux computers, or your own computer, or some other computer you have access to. However, even if you have used a different computer for your performance testing,

[2]You can also use SSH tunneling of X11 (X-windows). This is useful if you want to open graphical displays from the remote machine on your local computer. To achieve this, an X11 server must be running on your local machine. The X11 connections are then tunneled and automatically encrypted by your SSH client. Log in with `ssh -X user@host` or `ssh -Y user@host`.

you must still ensure that the code is portable so that it can be compiled and run on the university's Linux computers. This is necessary to make sure that your teachers can test your code. The code that you submit must compile and run on `fredholm.it.uu.se` (or `arrhenius.it.uu.se`).

Report: You must write a report that motivates the efficiency of your implementation using time measurements and a description of the optimisation techniques you have used or attempted to use. In this report, also analyse how the computational time depends on `N`. Remember that one of the course goals involves written communication — the assignment reports is part of how we examine this goal and we expect the highest level of quality. Write clearly and concisely, using figures and tables as appropriate.

When writing your report, remember that *reproducibility* is important: whenever you write a report that includes some timing measurements, or other computational experiments of some kind, you should make sure that the report includes all information necessary so that the reported results become reproducible. The reader of your report should be able to reproduce your results. Your report should make it clear exactly what it is you are reporting, if you have some timings it should be explained precisely what you were measuring: how was the measurement made, was it for the whole program run or for some specific part of the code, what parameters were used, how many timesteps, etc. Details about the used computer, CPU model, compiler version, and compiler optimization options should also be included.

Note that timing results should not include calls to graphics routines. When measuring timings, run your code with graphics turned off to be sure that graphics routines are not disturbing your timings.

5. Input data

Unpacking the `Assignment3.tar.gz` file gives a directory called `input_data` with various input files that you can use. The smallest ones with just a few particles are good to use to verify that your forces and timestepping are working properly.
`circles_N_2.gal` : two stars with equal mass moving in circles.
`circles_N_4.gal` : four stars with equal mass moving in circles.
`sun_and_planet_N_2.gal` : one heavy particle and one lighter particle orbiting the heavy one, like a planet around a sun.
`sun_and_planets_N_3.gal` : sun and two planets.
`sun_and_planets_N_4.gal` : sun and three planets.

For the larger cases that are more interesting for performance evaluation, the input files have the following form:

`ellipse_N_01500.gal`

where the number, 1500 in this case, is the number of stars N.

Those initial distributions have an elliptic shape similar to Figure 5.1, and their evolution may lead to (for example) something like Figure 5.2.

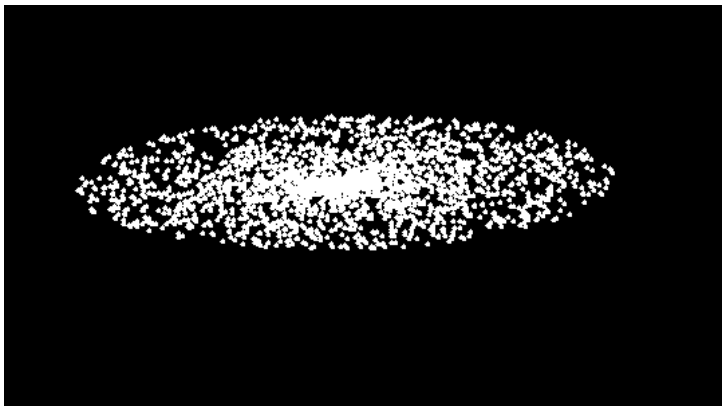

FIGURE 5.1. Example of an initial distribution of a galaxy with 2500 stars

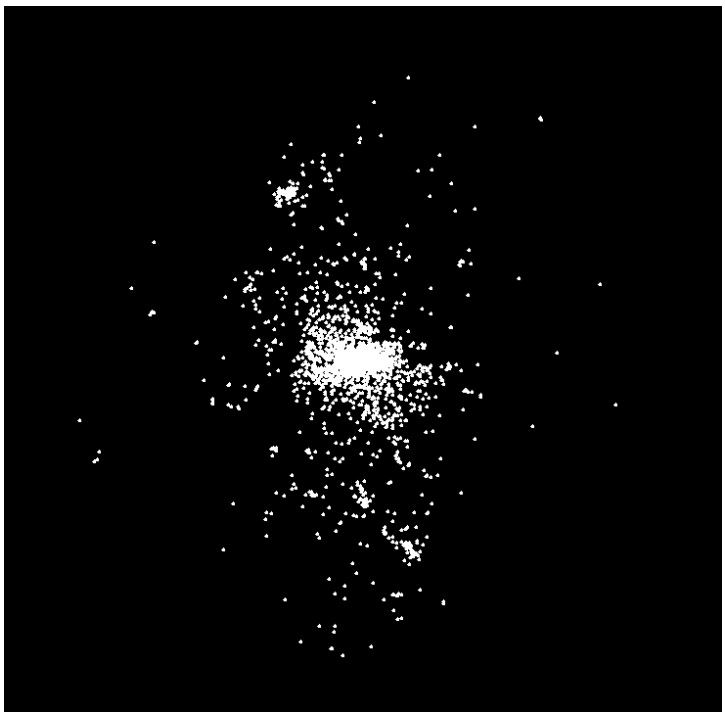

FIGURE 5.2. Example of a possible distribution of a galaxy with 2500 stars after some time (this picture is for a different initial distribution, you should not expect to get precisely this result)

5.1. **Note: simulations will always be unstable for long times.** The small test cases with for example particles moving in circles, or like a sun with a few planets, are good for verifying that the force computation and time stepping is working properly. However, even when the computations are done correctly this kind of simulations will become unstable when running for a long time since small

errors (there is always some effect of rounding errors) will accumulate over time and sooner or later this will always be seen.

So, when looking at the small simulations of a few planets etc. you should mostly focus on the initial behavior, for example check that the particles move in circles or orbit the sun in the expected way a few times, but if looking at it for longer times it is not strange that the simulations "go crazy", that is to be expected.

(Are there other more accurate and stable methods than Euler, what methods? You don't have to implement other methods, that is another course.)

5.2. **Check your own results by comparing to reference output data.** To make it easier for you to check your own results, the `Assignment3.tar.gz` file contains in the `ref_output_data` directory a few reference galaxy distributions in the following files:
`ellipse_N_00010_after200steps.gal`
`ellipse_N_00100_after200steps.gal`
`ellipse_N_00500_after200steps.gal`
`ellipse_N_01000_after200steps.gal`
`ellipse_N_02000_after200steps.gal`
and a small program `compare_gal_files.c` for comparison of galaxy files.

As the filenames suggest, those files contain galaxy distributions after 200 timesteps. So, for example, if you run your own simulation for 200 timesteps starting from `ellipse_N_01000.gal` your result should be close to the reference result in `ellipse_N_01000_after200steps.gal`.

The `compare_gal_files.c` program outputs the maximum difference between particle positions as `pos_maxdiff`. Since the coordinates for particle positions are supposed to be between 0 and 1, for one timestep the errors in positions should be small, on the level of rounding errors for the floating-point precision used. Errors will then of course accumulate during the simulation, but since the reference output data are for relatively short simulations, only 200 timesteps, error accumulation should not be too bad. Errors should still be within a few orders of magnitude from the size of rounding errors.

6. Deliverables

It is important that you submit the assignment in time. **See the deadline in Studium.**

You should package your code and your report into a single `A3.tar.gz` file that you submit in Studium. There should be a makefile so that issuing "make" produces the executable `galsim`. The source file should be named `galsim.c` and include all functions. **Note**, all code should be included in this source file. This is important when we do the plagiarism check using *Moss - Measure of software similarity*. If we can't do the plagiarism check the assignment will not be approved.

Unpacking your submitted file `A3.tar.gz` should give a directory `A3` and inside that there should be a makefile so that simply doing "make" should produce your executable file "`galsim`". The `A3` directory should also contain your report, as a file called `report.pdf`.

In addition to the report, your `A3` directory should also include a text file called `best_timing.txt` that should contain just two lines: on the first line, just one number giving the fastest time (in wall seconds) that you were able to achieve for the input case `ellipse_N_03000.gal` with $\Delta t = 10^{-5}$ and 100 timesteps. On the second line, the name of the CPU model used.

Apart from `report.pdf` and `best_timing.txt` files, your submission should only contain the C source code file and makefile. No object files or executable files should be included, and no input/output (.gal) or other binary files.

Part of our checking of your submissions will be done using a script that automatically unpacks your file, builds your code, and runs it for some test cases, checking both result accuracy and performance. For this to work, it is necessary that your submission has precisely the requested form.

To be sure that your submission has the correct form, you should check it yourself before submitting, using the `check-A3.sh` script that is included with the assignment. To use the `check-A3.sh` script, first set execute permission for it, then copy your `A3.tar.gz` file into the Assignment3 directory, and then run `./check-A3.sh` when standing in the Assignment3 directory. Make sure that you do the checking on `fredholm.it.uu.se` or `arrhenius.it.uu.se` as we will do the checking on these machines as well. The `check-A3.sh` script will then try to unpack your `A3.tar.gz` file, check that it has the expected contents, and compile and run your program trying to check that it seems correct. If the checking is successful, the output from the script should say "Congratulations, your A3.tar.gz file seems OK!". If you do not get that result, look carefully at the script output to figure out what went wrong, then fix the problem and try again.

If you have used your own computer for this assignment and then move it to the university's Linux computers you may find that some change is needed to make the code portable there, e.g. adding `-lm` to link to the math library. Check also for additional warnings by using the `-Wall` flag. Moreover, run you code through the `valgrid` memory checker so you don't have any undetected memory errors.

The report shall be in pdf format (called `report.pdf`) and shall contain the following sections:

- The Problem (very brief)
- The Solution (Describe the data structures, the structure of your codes, and how the algorithms are implemented. Are there other options, and why did you not use them?)
- Performance and discussion. Present experiments where you investigate the performance of the algorithm and your code. Describe optimizations you have used, or tried to use, and the measured effect of those optimization attempts. Include a figure with a plot of measured execution time as a function of N to confirm the expected $\mathcal{O}(N^2)$ complexity.
- References, all the references you have used for code and theory. If we find that you have copied code or code sections without a proper reference, it will be considered as plagiarism.

Submission

When you are done, upload your final `A3.tar.gz` file in Studium. Note that the uploaded file should have precisely that name, and that you should have checked it using the `check-A3.sh` script before uploading it. Did it pass the check with

```
Congratulations, your A3.tar.gz file seems OK!
```

If not, you are not done yet and should NOT submit until it passes the check.