Uppsala University
Department of Information Technology


UPPSALA
UNIVERSITET

# The gravitational $N$-body problem

Jannes van Poppelen
Csongor Horváth
Christos Pilichos

# Contents

# 1 Introduction

Every introductory classical mechanics course touches upon the central force problem, which studies the dynamics of two bodies due to their gravitational interaction. This problem is swiftly converted into a one-body problem due to underlying symmetries, which gives rise to an analytical solution [1]. However, like its quantum mechanical analogue, the many-body problem, no general analytical solution exists to the $N$-body problem for arbitrary $N$ [2]. Solutions to the $N$-body problem are therefore approximated using numerical methods, as is done in this work.

Central to the problem is Newton's law of gravitation, which says that for two particles of masses $m_i$ and $m_j$, respectively, particle $i$ experiences a force by particle $j$ given by

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{r_{ij}^2}\hat{\mathbf{r}}_{ij}, \tag{1}$$

where $\mathbf{r}_{ij} = (x_i - x_j)\hat{\mathbf{x}} + (y_i - y_j)\hat{\mathbf{y}}$, $x_i, y_i$ are the $x$ and $y$ coordinate of particle $i$, respectively, and G is the gravitational constant. For $N$ bodies this is taken to be $100/N$. The force on particle $i$ due to the $N-1$ other bodies then generalizes to

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

However, this expression is numerically unstable if $r_{ij} \ll 1$. Instead, so-called Plummer spheres are used, which introduce a small parameter $\varepsilon_0$ to avoid divergences caused by the denominator

$$\mathbf{F}_i = -Gm_i \sum_{j=0,\, j\neq i}^{N-1} \frac{m_j}{(r_{ij} + \varepsilon_0)^3}\mathbf{r}_{ij}. \tag{3}$$

Finally, the dynamics are implemented through the symplectic Euler scheme, which computes accelerations, velocities, and positions according to

$$\mathbf{a}_i^n = \frac{\mathbf{F}_i^n}{m_i}, \qquad \mathbf{u}_i^{n+1} = \mathbf{u}_i^n + \Delta t \mathbf{a}_i^n, \qquad \mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t \mathbf{u}_i^{n+1}. \tag{4}$$

This algorithm is implemented and optimized serially in C. In order to test its performance, several $N$-body simulations, with various $N$ ranging from 10 to 10000 have been done, where the computational domain is the unit square. The masses of the particles need not necessarily be all the same, as this also makes for more interesting dynamics. Even though the code itself is not very complex, various choices regarding the implementation and optimization will be elaborated on.

## 2 Solution

Reading and storing data from input files can be done in various ways. The simplest and most straightforward way is to introduce a `struct` of six variables of length `N` and type `struct`, as this makes reading input using `fread` trivial due to the struct being commensurate with the input data of one particle. Moreover, this proves equally useful in writing away particle data using `fwrite`. An alternative way to store the input data is to create six arrays of length $N$ and type `double` and save one variable per particle ($x$-coordinate, $y$-coordinate, etc.) in each array. While this does not occupy more memory, it does complicate how the reading and writing of data is done.

Time evolution is handled according to the previously described symplectic Euler scheme. In the code, time evolution is handled using a `for` loop. Then, at each time step for each particle, the interactions with the other $N-1$ particles are computed and summed up. This is done using two `for` loops, which yields the expected $\mathcal{O}(N^2)$ complexity. Velocities of particles are updated in the same loop that computes the interactions, which is possible since gravity is a conservative force that is independent of velocities. If there was an attempt to update positions in the same loop that interactions are computed, particles will experience erroneous forces, hence updating positions is done in a separate loop after all velocities are updated. Interactions can be split into two separate loops as well, which prevents computation of the "self-interaction", i.e. gravitational interaction with itself. While this vanishes trivially and need not necessarily be dealt with, explicitly omitting it saves $N \cdot n_{\text{steps}}$ computations, where $n_{\text{steps}}$ is the number of iterations of the simulation. For simulations with small $n_{\text{steps}}$, this saves minimal time, however, when $n_{\text{steps}}$ is sufficiently big it is expected to save time. One must still ensure that simulations are not too long so that numerical instabilities take over. The double loop structure of the code also makes it easy to recognize which parts are parallelizable.

Graphics to visualize galaxies are also implemented within the code, however, an alternative route has been chosen rather than the graphics routine that was provided with the assignment. Much simpler and less time-consuming is to pipe the $x$- and $y$- coordinates of each particle straight from C to Gnuplot at each timestep [3]. In order to do this, a pipe has to be opened that prints data from the simulation to a separate terminal running Gnuplot. This is done as `FILE * gnuplotPipe = popen("gnuplot -persistent","w")` in the code, where the flag `-persistent` ensures that the window created by Gnuplot does not close after every time step. An example of how such a galaxy looks is provided in figure 1. After every instance that Gnuplot updates the plot, it is important to "flush" the pipe with `fflush(gnuplotPipe)`, which ensures that the positions of all particles in the galaxy are written to Gnuplot before other operations are performed.

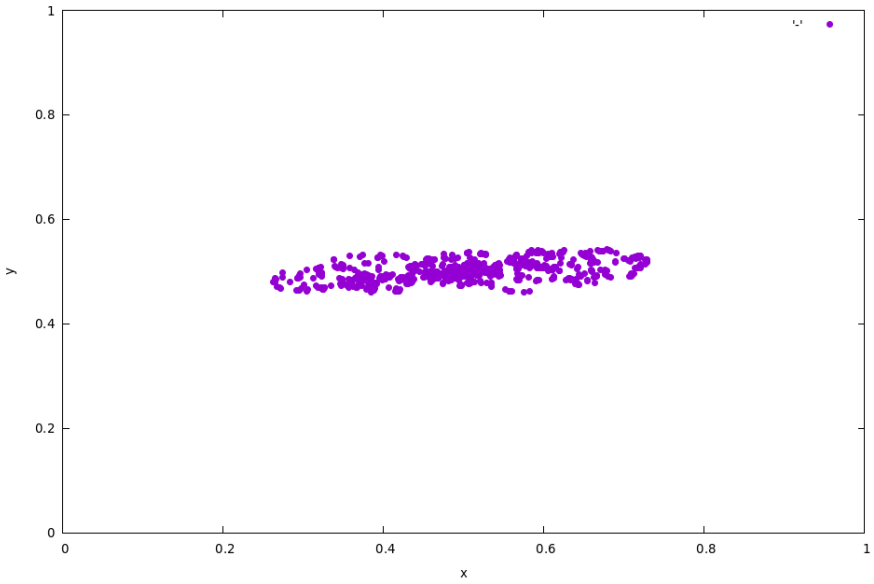



Figure 1: Snapshot of a simulation for $N = 1000$.

## 3 Performance and discussion

After confirming the functionality of the code by comparing various outputs with their corresponding reference outputs, the focus shifted towards enhancing performance by optimizing the serial code. Several methods were explored, with varying degrees of effectiveness, with some yielding significant improvements, while others had negligible impact on the execution time.

Execution times were measured using the `get_wall_seconds` function from any of the HPP labs on a machine with an AMD Ryzen 5 3500 CPU [4]. The source code was compiled with GCC 11 (gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0). Measured times exclude the reading and writing of data so that the focus of individual optimization techniques is solely on the computations done during the time evolution. For reference, simulations were done for 100 time steps using `ellipse_N_03000.gal` as the starting configuration. Time steps were $10^{-5}$ each. To compare the effects of individual optimization techniques, a naive and unoptimized simulation was timed, which took approximately 82.990 wall seconds. For each optimization technique, a short description and its impact on the code are presented.

**Compiler optimization**
Appropriate compiler flags can already significantly optimize the code without having to do any manual alterations. Optimizations technique done by the compiler includes function inlining, strength reduction, loop unrolling, and many more. Compiler flags that were considered are presented below. The effect of each compiler flag is presented in table 1.

- `-O3`: The compiler optimizes aggressively, even if might increase the size of the executable code, or take a longer time to compile.

- `-Ofast`: It follows all the `-O3` optimizations, but also leads to possible unsafe optimizations, such as violation of IEEE floating point rules by getting rid of subnormal numbers.

- `-march`: The compiler utilizes the features of the CPU architecture that is used for compiling the code. However, enabling this flag restricts the portability of the executable code.

| Method | Execution time (s) | Benefits (%) |
|---|---|---|
| `-O3` | 22.566 | 72.80 |
| `-Ofast` | 3.346 | 95.96 |
| `-O3 -march=native` | 21.509 | 74.08 |
| `-Ofast -march=native` | 3.075 | 96.29 |

Table 1: Effect of compiler flags on the execution time.

By using the `-Ofast` flag , the execution time was reduced by 96.29%, while the `-O3` flag benefitted only 72.80%. Moreover, using `-march=native` reduced the execution time even further. It is worth mentioning that the accuracy after each simulation is checked to ensure that accuracy is not traded off for computational gain. For the remaining optimization techniques, the flags `-Ofast -march=native` are used as they resulted in the greatest performance increase. Note that this already avoids the computation of the self-interaction, as will be elaborated further on.

**Strength reduction**

One way to reduce the computational load of the code is to choose operations that are computationally the most efficient. To this end, we proceeded with a variety of changes such as the `constant` declaration for the number of timesteps `nsteps`, the step size `dt`, the number of particles `N`, and the gravitational constant `G`. Furthermore, math functions like `pow` were eliminated, and floating-point multiplication was preferred over floating-point division. Differences in the code can be seen in the two code blocks below. The first block is unoptimized, while the second block contains all changes mentioned above.

```
rij= sqrt(pow((particles[i].x-particles[j].x),2) \
+pow((particles[i].y-particles[j].y),2));
C= (-G*(particles[j].m))/(pow((rij+eps0),3));
ax+= C*(particles[i].x-particles[j].x);
ay+= C*(particles[i].y-particles[j].y);
```

```
rij= sqrt((particles[i].x-particles[j].x) \
        *(particles[i].x-particles[j].x) \
        +(particles[i].y-particles[j].y ) \
        *(particles[i].y-particles[j].y));
denom= 1/((rij+0.001)*(rij+0.001)*(rij+0.001));
C= (-G*(particles[j].m)*denom);
```

```
ax+= C*(particles[i].x-particles[j].x);
ay+= C*(particles[i].y-particles[j].y);
```

The denominator of the force was also computed explicitly in order to avoid multiple arithmetic operations on the line `C` was computed. It turned out that the optimization techniques that were implemented had marginal, if any, improvements in the execution time. Since the compiler takes care of strength reduction, this is not entirely unexpected. Math functions were avoided when possible. A possible sacrifice of accuracy for a smaller execution time could be achieved by using `sqrtf` instead of `sqrt`. While this significantly improved the performance, it also reduced the accuracy by a tiny amount. Since more accurate results are preferred, usage of `sqrtf` was avoided. Since it is good practice, the `pow` function from the `math` library has been replaced by equivalent ones.

**Function Inlining**.
Inlining functions removes function calls by pasting the body of a function whenever it is read by the compiler. Together with the `static` keyword to restrict the function only to the same file it is called from, this can speed up code if the function gets called many times. Squaring and cubing of numbers is done many times per time step in the code, which means it might be beneficial to `inline` the `pow2` and `pow3` functions as presented below.

```
static inline double pow2(double x){
        return x*x;
        }
static inline double pow3(double x){
        return x*x*x;
        }
```

However, once again minimal improvement in the execution time was found, with the fastest time of 3.084 seconds. Due to the rather high variance in the measured times, one should not conclude that inlining functions has a negative effect on the execution time.

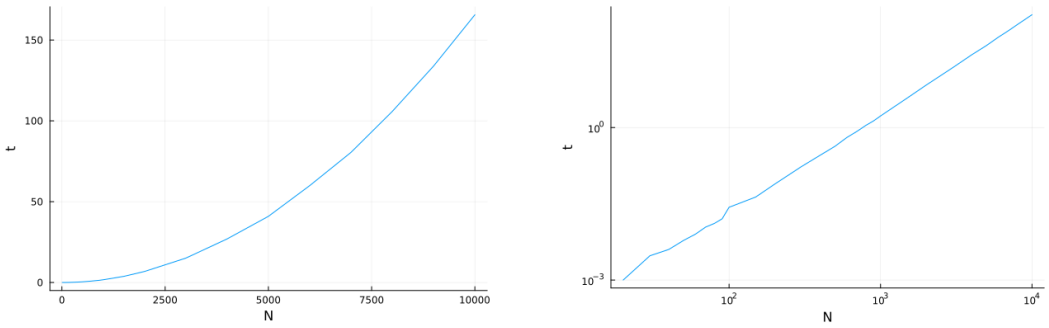
**Elimination of branches**
Several branches can be created by including `if` statements in a code. If a branch is present in a loop, this can significantly increase the execution time. As previously mentioned, a lot of computations can be avoided if self-interactions are not computed. In the code, two different approaches were tried. A first attempt made use of two loops in computing the interactions. The first loop `for(j=0;j<i;i++){...}` computes interactions from `j=0` to `j=i-1` , whereas the second loop `for(j=i+1;j<N;j++){...}` computes interactions from `j=i+1` to `j=N`, so that effectively computations for `i=j` are avoided. For 100 time steps, the time saved is not that much, however, if the number of time steps is increased, the effect is more visible. Alternatively, both loops were merged into one loop `for(j=0;j<N;j++)`, which when followed by `if(j!=i){...}`, also entirely skips computations for `i=j`. Even though this creates a branch for which the condition has to be checked at every iteration, it proved considerably faster than the previous approach.

**Loop unrolling**
There was also an attempt to manually unroll loops, however, with the `-Ofast` flag, this is already handled by the compiler. Any attempts to do this manually indeed showed no real performance increases.

**Register Keyword**
The `register` keyword passes a message to the compiler that a variable is being used constantly so that it can be put in the register for fast access. Unfortunately, using this for some variables did not allow for noticeable performance increases.

**Newton's Third Law**
To attempt to further improve on the complexity of the algorithm, we attempted to cut the number of computations by using Newton's third law, which states that $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$. Naively using both for-loops would result in $N(N-1)$ computations, whereas implementing Newton's third law gives $\sum_{i=1}^{N}(N-i) = \frac{N}{2}(N-1)$ computations, which should cut the number of computations in half. However, this requires an additional array of length $\frac{N}{2}(N-1)$ to be defined to store the interactions for later use, which becomes increasingly expensive for large $N$. Moreover, this array will have to be updated for each time step, which significantly decreases the efficiency of the cod. While being a creative effort to reduce the number of computations, Newton's third law does not end up yielding improvements.

**Quadaritc complexity**
In order to verify the $\mathcal{O}(N^2)$ complexity of the algorithm, simulations have been done from $N = 10$ up to $N = 10000$. Results are plotted in figure 2. While figure 2a looks like it satisfies the quadratic behaviour, the plot has been reproduced with logarithmic axes in order to verify the behaviour. Performing a linear fit gives a slope of roughly 1.934, which confirms the quadratic complexity of the algorithm. Note that a different machine was used to compute execution times than before, which should be fine since this part only concerns itself with the quadratic complexity.




(a) (b)

Figure 2: (a), Simulation times, in seconds, as a function of the number of particles. (b), Simulation times, in seconds, as a function of the number of particles, plotted with logarithmic axes.

**Conclusion**
Many different manual implementations of optimization techniques have been tried, however, most of these were in vain, since the optimization flags of the compiler already handle these. The most noticeable improvement besides compiler flags, was avoiding computing the self-interactions at every time step. Ultimately, the fast execution time turned out to be 3.075 wall seconds on a machine with an AMD Ryzen 5 3500 CPU.

## References


[1] H. Goldstein, C. Poole, and J. Safko, "Classical mechanics," 2002.

[2] P. Coleman, *Introduction to many-body physics*. Cambridge University Press, 2015.

[3] T. Williams, C. Kelley, and many others, "Gnuplot 5.4.5: an interactive plotting program." http://gnuplot.sourceforge.net/, March 2010.

[4] J. Rantakokko et al., "Lab assignments HPP," *Available through high performance programming course page on Studium*, 2023.