N-Body Simulation

Hien Ha

Step 1: A full N-body simulator

1 1.1: Multiple free objects

The function *force_calculation* computes the force between two bodies, involving simple arithmetic operations and a square root calculation, which are of constant time $O(1)$, while *updateBody* loops over N in its outer loop, and N times on its inner loop, each step involves N^2 force calculations (due to the nested loops) resulting in a time complexity of $O(N^2)$. This implementation has the expected time complexity by complying with the aforementioned architecture.

For optimisations, a more efficient data structure could be used, for example, via Barnes-Hut Algorithm. By dividing the simulation volume into cubic cells via an octree, and reduce the number of pairwise force calculations by approximating cells further away. An improvement to reduce floating point error is to use Kahan summation by keeping track of the error with a compensation, increasing the precision of the arithmetic operations.

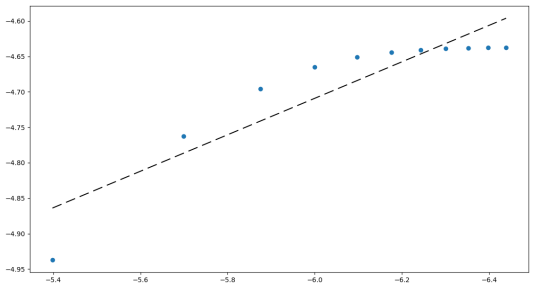
With the initial conditions stated in test1.sh(included in the submission), the largest stable timestep size for this scenario is 0.0008027, found through experimentation. The formula to estimate the convergence order of Euler's method is given by:

$$q = \frac{log(\frac{e_{n+1}}{e_n})}{log(\frac{h_{n+1}}{h_n})} \text{ where } e_{n+1} = |value_{true} - value_{numerical} \text{ with } h_{n+1}| \tag{1}$$

However due to no collisions, once bodies approach within a small enough distance, they accelerate away due to $\hat{d3}$ term in denominator of the force calculation, therefore, with observations of the simulation, convergence is not possible with the time limit used.

2 1.2: Collisions

To modify *NBodySimulation* for collision, a conditional branch is introduced when calculating Euclidean distance between two bodies. A derived class inheriting from *NBodySimulation* called *NBodySimulationCollision* is implemented in *step1.cpp*. With collision, to demonstrate the convergence order of this equation of N-Body problem, a log graph can be plotted with the largest timestep collision against iteratively halving Δt, using experimental data taken with initial conditions in test1.sh:



Step 2: Advanced Methods

The chosen optimisation is to implement Runge-Kutta Fourth Order method(RK4). On a shorter timescale and more complex force models, in addition to being higher order, RK4 can be more accurate and efficient than other methods like Implicit Euler, Verlet, and lower order methods. However, RK4 is explicit, so only conditionally stable. It has a time complexity of $O(N^2)$. The method is given as:

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + k_1/2, t_n + h/2) \\ k_3 &= hf(y_n + k_2/2, t_n + h/2) \\ k_4 &= hf(y_n + k_3, t_n + h) \\ y_{n+1} &= y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6 \end{aligned} \tag{2}$$

With Euler integration, for collision to happen, the timestep size has to be very small and occasionally require a small tolerance, therefore very computationally expensive. On the other hand, with RK4, the bodies can collide given a small enough time step, still larger than Euler method's. RK4 is faster than Euler's, with up to 40% lower runtime, there are less numerical error therefore converges faster. It's also more stable at higher Δt, where numerical solutions remain bounded and do not diverge significantly, and more flexible in terms of optimisation, where adaptive Δt and error estimation is possible.

Step 3: Vectorisation

Vectorisation on RK4 improves performance over serial execution at higher N. At lower number of bodies and simpler starting conditions, serial code is slightly faster, where run time of serial code with initial conditions in test1.sh is 32ms while vectorised code takes 36ms. However, with initial conditions in step-3_big.sh at 125 bodies, runtime for vectorised code is 92s and stable at $dt = 0.001$ while serial code takes 97s. *omp simd* pragmas are used to vectorise for loops and clauses such as private, shared, and reduction is experimented to prevent data races.No vectorisation on *updateBody* due to conditionals, so more fine-grained vectorisation within RK4 is implemented. Furthermore Vtune Profiler indicated a 8% increase in effective CPU utilisation, indicating succesful vectorisation.

Step 4: Instruction-level parallelism

Due to collisions, course-grained parallelism of outer loop of *updateBody* is not possible due N bodies decrementing. If enabled, collisions cause the last body to be missed for a timestep. The parallelisation is approached in 2 angles. Number one, fine-grained parallelism is employed inside RK4 function. The parallelisation uses 3 threads with *omp_set_num_threads*(3) across 3 CPUs with a redundant CPU. The 3 threads is used to process the 3 dimensions in parallel inside a *omp parallel* pragma, each computing positions, velocity, acceleration of each component using *omp_get_thread_num*() for indexing dimensions. *omp barrier* was used to prevent data races and while *omp single* is used to calculate distance at each step of RK4 without exiting parallel, which is also used to synchronised threads due to implicit *omp barrier* at the end of *omp single* construct. This allows parallelism in RK4, but the drawback is that it doesn't scale with number of cores. The second approach is to parallelise the inner loop of *updateBody* using *parallel for reduction(max:maxV)*etc. This immediately improves performance and convergence happens much faster. Test case with N=125 and $\Delta t = 0.001$ converges in less than a second. This approach scales much better. On the other hand, the intuition is to combine both approach, is malpractice due to parallism is handled by processes. Furthermore, a peculiar problem occurred when convergence is achieved. The master thread inside *parallel for* pragma is still looping and evaluating in a very long cycle. Therefore a mechanism is introduced. By checking N bodies < 2 after every iteration and terminate upon true, ensures safe thread termination.