COSC3500
2D Orbital Simulation Report

Maxwell Bo (43926871)

August 17, 2018

Description

The task was to create a stock-standard, 2-dimensional gravitational n-body simulator. All bodies were to be assumed to be point masses. The simulation was to be accurate, maintaining a constant total energy, and exhibiting phenomena such as apsidal precession. The simulation was to accept arguments specifying the granularity of the simulation (number of time steps, number of instances of data export), and a file specifiying masses and their initial positions and velocities. The simulation was to produce its output as fast as possible, with minimal slowdown with an increasing n number of bodies.

The simulator did not need to handle collisions between bodies. The MS1 simulator was to be free of OpenMP multiprocessing, which would be implemented in preparation for the MS2 submission.

Implementation

At a high-level, the initial naive simulator:

1. Parsed input parameters and files
2. Constructed `Body` class instances, representing each point mass
3. Packed the `Body`s into a `std::vector<Body>`, to maximize cache locality
4. Calculated forces between all pairwise combinations of n-bodies ($O(n^2)$)
5. Performed Euler's method to derive new velocities and positions
6. Output all necessary data
7. `GOTO 4`

By using a Quadtree, ('a tree datastructure in which each internal node has exactly four children') and the *Barnes-Hut* algorithm[1], the total cost of force calculation could be reduced to $O(n \log n)$, by grouping close-together bodies and approximating forces between the singular grouped pseudo-body, and distant bodies. New Quadtrees were constructed on each seperate simulation step.

Dehen and Read note that the Euler method 'performs very poorly in practice', further noting that 'errors are proportional to Δt^2'. They contrast it with the second-order leapfrog symplectic integrator, which is 'heavily used in collisionless N-body applications'.[2]

Leapfrog can be expressed many in forms[3] including a synchronised form:

$$\begin{aligned} x_i &= x_{i-1} + v_{i-1/2}\,\Delta t \\ a_i &= F(x_i) \\ v_{i+1/2} &= v_{i-1/2} + a_i\,\Delta t \end{aligned}$$

which only requires a single acceleration calculation per every two half timesteps (the timestep Δt must be constant to maintain stability), and a ‘kick-drift-kick’ form

$$\begin{aligned} v_{i+1/2} &= v_i + a_i \frac{\Delta t}{2} \\ x_{i+1} &= x_i + v_{i+1/2}\Delta t \\ v_{i+1} &= v_{i+1/2} + a_{i+1}\frac{\Delta t}{2} \end{aligned}$$

that is stable with variable timstepping, but incurs an additional acceleration calculation per every two half timesteps.

The synchronised form was implemented, but attempts to implement the kick-drift-kick form, and variable timestepping, were left unfinished.

Thus, the final implementation:

1. Parsed input parameters and files
2. Constructed `Body` class instances, representing each point mass
3. Packed the `Body`s into a `std::vector<Body>`, to maximize cache locality
4. Inserted all `Body`s into a fresh `QuadTree` on full timesteps, traversing the `QuadTree` with every `Body` to calculate forces ($O(n \log n)$)
5. Performed the appropriate Leapfrog step to derive new velocities *or* positions
6. Output all necessary data
7. `GOTO 4`

Correctness

I personally believe that the simulation is *mostly* right. By visualising the results with `matplotlib`, we see something that resembles an n-body simulator[1]. Distressingly, the total energy of the system is not constant throughout the system. When bodies are in close proxmity, anomalous energies that are inconsistent with the energy curve are observed. Furthermore, simulations with higher numbers of bodies produce random low energy outliers, but have a smoother total energy curve. There seems to be no signfiicant difference between Euler method and Leapfrog, with respect to energy anomalies.

[1]and that’s good enough for me

Performance & Scaling

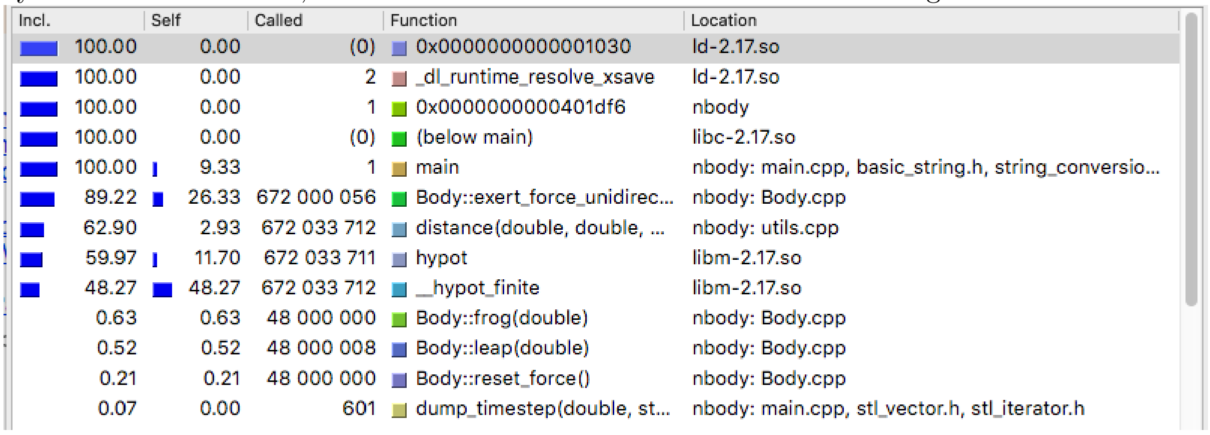
Henceforth, the use of the ‘recognizable’ refers to eyeballing the output data, and making no significant effort to investigate the data more rigorously.

The addition of the `-march=native` compiler flag, which enables the use of all CPU specific instructions, provided no recognizable improvement in running time, but was left enabled in the instance that it improved performance on `goliath`.

The use of both th GCC and Clang Profile-Guided Optimisation features provided no recognizable improvement in running time.

Distressingly, `-O0`, `-O1`, `-O2`, `-O3` showed no recognizable improvement in running time. `-Ofast` led to an -4%-ish performance regression.

By profiling with `callgrind`, we saw that execution was dominated by only one incredibly costly user-defined method, with an exclusive cost of 26.33% of total running time.



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000001030	ld-2.17.so
100.00	0.00	2	_dl_runtime_resolve_xsave	ld-2.17.so
100.00	0.00	1	0x0000000000401df6	nbody
100.00	0.00	(0)	(below main)	libc-2.17.so
100.00	9.33	1	main	nbody: main.cpp, basic_string.h, string_conversio...
89.22	26.33	672 000 056	Body::exert_force_unidirec...	nbody: Body.cpp
62.90	2.93	672 033 712	distance(double, double, ...	nbody: utils.cpp
59.97	11.70	672 033 711	hypot	libm-2.17.so
48.27	48.27	672 033 712	__hypot_finite	libm-2.17.so
0.63	0.63	48 000 000	Body::frog(double)	nbody: Body.cpp
0.52	0.52	48 000 008	Body::leap(double)	nbody: Body.cpp
0.21	0.21	48 000 000	Body::reset_force()	nbody: Body.cpp
0.07	0.00	601	dump_timestep(double, st...	nbody: main.cpp, stl_vector.h, stl_iterator.h

Performance fixes were divised.

```
 void Body::exert_force_unidirectionally(const Body& there) {
     const double m1 = m;
     const double m2 = there.m;

     const double delta_x = there.x - x;
     const double delta_y = there.y - y;

-    const double r = distance(x, y, there.x, there.y);
+    const double r = hypot(delta_x, delta_y);
-    const double r2 = r * r;
+    const double r2 = pow(r, 2);

     const double F = (G * m1 * m2) / r2;

     // turn the displacement vector between our two points into a force vector
     // of the desired magnitude
     const double scale_factor = F / r;

     Fx += delta_x * scale_factor;
     Fy += dumpsdelta_y * scale_factor;
 }
```

Instead of recalculating Δx and Δy twice (the second time in `distance`), we calculate them only once and make a direct call to `hypot`, rather than making a call to `distance`. We also used the specialized `pow` provided by `cmath`. This yielded a recognizable 15%ish performance improvement.

```
diff --git a/Assignment_1/src/Body.cpp b/Assignment_1/src/Body.cpp

+void Body::exert_force_bidirectionally(Body& there) {
+    const double m1 = m;
+    const double m2 = there.m;
+
+    const double delta_x = there.x - x;
+    const double delta_y = there.y - y;
+
+    const double r = hypot(delta_x, delta_y);
+    const double r2 = pow(r, 2);
+
+    const double F = (G * m1 * m2) / r2;
+
+    // turn the displacement vector between our two points into a force vector
+    // of the desired magnitude
+    const double scale_factor = F / r;
+
+    Fx += delta_x * scale_factor;
+    Fy += delta_y * scale_factor;
+
+    there.Fx -= delta_x * scale_factor;
+    there.Fy -= delta_y * scale_factor;
+}

diff --git a/Assignment_1/src/main.cpp b/Assignment_1/src/main.cpp
@@ -258,8 +259,7 @@ int main(int argc, char **argv) {
                  for (size_t j = i + 1; j < bodies.size(); j++) {
                      auto& y = bodies[j];
-                     x.exert_force_unidirectionally(y);
-                     y.exert_force_unidirectionally(x);
+                     x.exert_force_bidirectionally(y);
                  }
             }
         }
```

This fix halved execution time, for obvious reasons.[2].

```
diff --git a/Assignment_1/src/Body.cpp b/Assignment_1/src/Body.cpp
@@ -83,11 +82,11 @@ double Body::kinetic_energy() const {
 double Body::gravitational_potential_energy(const Body& there) const {
     const double R = distance(x, y, there.x, there.y); // final distance, aka, to edge

-    return (-G * m * there.m) / R;
+    return (-Gm * there.m) / R;
 }

@@ -101,7 +100,7 @@ void Body::exert_force_unidirectionally(const Body& there) {

-    const double F = (G * m1 * m2) / r2;
```

[2] I had to throw out all my old profile data

```
+    const double F = (Gm * m2) / r2;

@@ -113,7 +112,6 @@ void Body::exert_force_unidirectionally(const Body& there) {

-    const double F = (G * m1 * m2) / r2;
+    const double F = (Gm * m2) / r2;

@@ -189,6 +189,7 @@ std::vector<Body> parse_input_file(std::ifstream& input_fh) {

     for (size_t i = 0; i < bodies.size(); i++) {
         bodies[i].m = masses[i];
+        bodies[i].Gm = G * masses[i];
     }
```

Precomputing Gm yielded a 3%ish performance improvement.

References

[1] J. E. Barnes and P. Hut, "A hierarchical O(n-log-n) force calculation algorithm," *Nature*, vol. 324, p. 446, 1986.

[2] W. Dehnen and J. I. Read, "N-body simulations of gravitational dynamics," *European Physical Journal Plus*, vol. 126, p. 55, May 2011.

[3] R. D. Skeel, "Variable step size destabilizes the störmer/leapfrog/verlet method," *BIT Numerical Mathematics*, vol. 33, pp. 172–175, Mar 1993.

Figure 1: Observed energy anomaly while bodies in close proxmity - Leapfrog

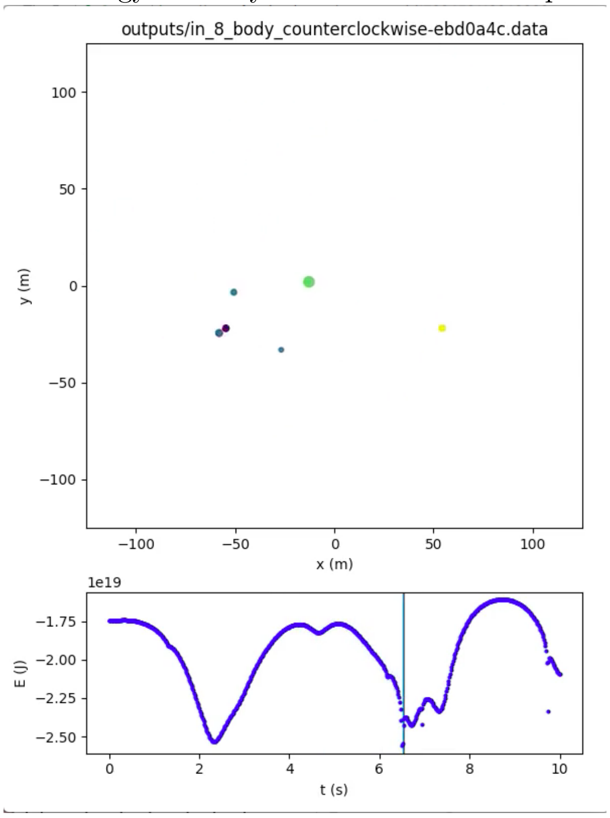


Figure 2: Observed energy anomaly while bodies in close proxmity - Euler method

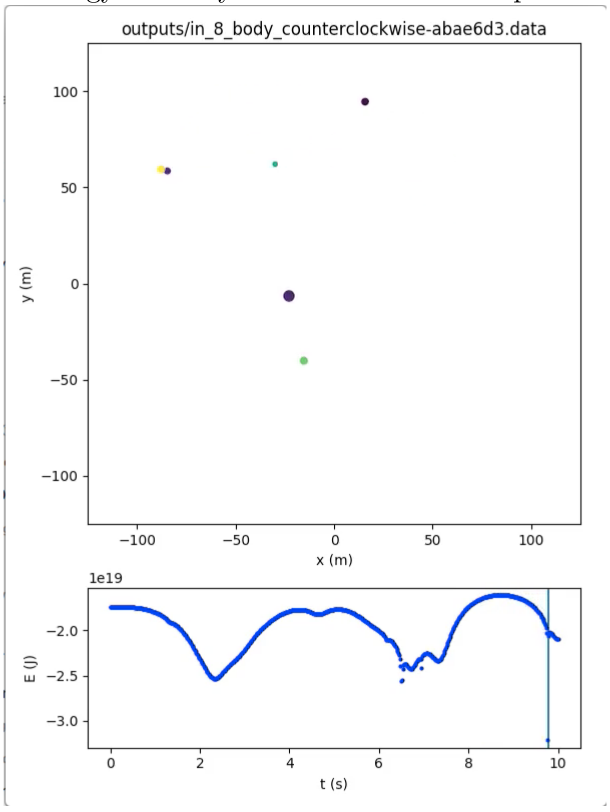


Figure 3: Observed energy anomaly while bodies in close proxmity - Euler method

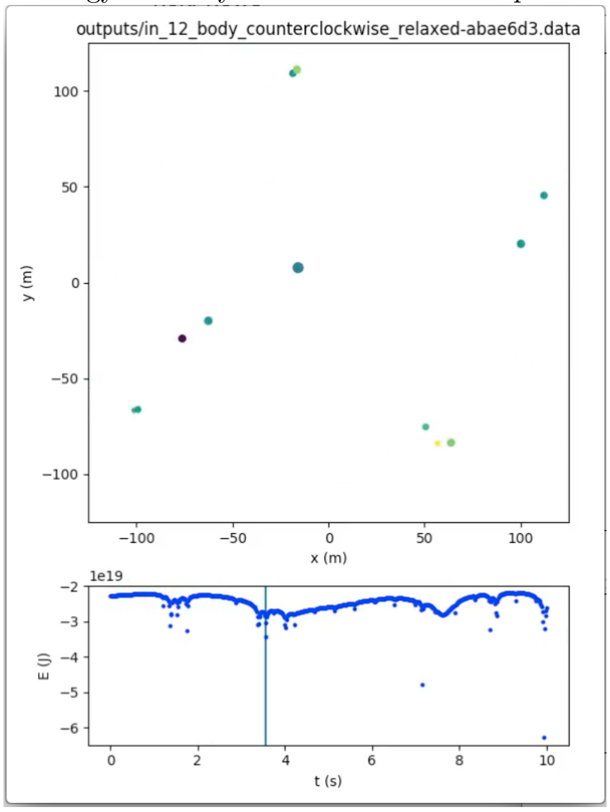
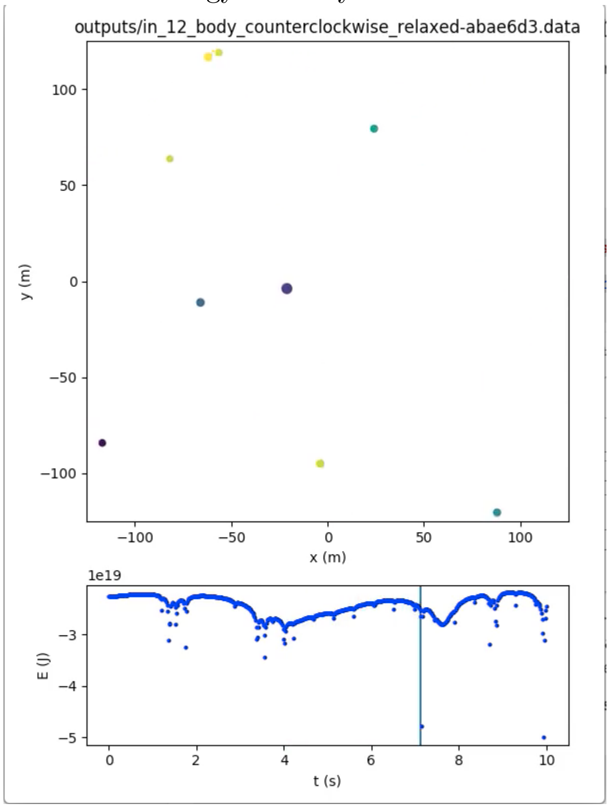


Figure 4: Observed energy anomaly - random outlier - Leapfrog


outputs/in_12_body_counterclockwise_relaxed-abae6d3.data
y (m)
x (m)
1e19
E (J)
t (s)