The n-Body Problem

Project 1 – Due Friday Mar 15

You will make a program to efficiently simulate n "bodies" in a 2D world. Each body has a mass, position, and velocity. Each body is gravitationally drawn towards every other body (and thus all draws every other body towards it). See wikipedia.org/wiki/N-body_problem for more details.

The Physics

The simulation will use Newtown's Law of Universal Gravitation, the Superposition Principle, and Newton's Second Law of Motion to determine how the bodies move within the 2D space. In the equations, for the i-th body m_i is its mass in kg and x_i, y_i, and z_i are its position in m.

- ***Newton's Law of Universal Gravitation***: the strength of the gravitational force F_{ij} between two bodies i and j is

  $$F_{ij} = G\frac{m_i m_j}{r_{ij}^2}$$

  where $G = 6.6743015\mathrm{e}{-}11\ \frac{\mathrm{Nm}^2}{\mathrm{kg}^2}$ is the gravitational constant, $r_{ij} = \sqrt{\Delta x_{ij}^2 + \Delta y_{ij}^2 + \Delta z_{ij}^2} + \varepsilon$ is the Euclidian distance, $\Delta x_{ij} = x_j - x_i$, $\Delta y_{ij} = y_j - y_i$, and $\Delta z_{ij} = z_j - z_i$ are the differences in the positions of the two bodies, and $\varepsilon = 10^{-9}$ is a "softening" parameter to avoid division-by-zero.

- ***Superposition Principle:*** the net forces on a body in the x or y directions are the sum of all forces acting on the body in the direction. Thus, the overall forces on body i in the x and y directions are

  $$F_x = \sum_{j=1}^{n} F_{ij}\frac{\Delta x_{ij}}{r_{ij}} \qquad F_y = \sum_{j=1}^{n} F_{ij}\frac{\Delta y_{ij}}{r_{ij}} \qquad F_z = \sum_{j=1}^{n} F_{ij}\frac{\Delta z_{ij}}{r_{ij}}$$

  (note: the order of subtraction in Δx_{ij}, Δy_{ij}, and Δz_{ij} is important in the sums).

- ***Newton's Second Law of Motion:*** the acceleration (in $\mathrm{m/s}^2$) of a body in a direction is proportional to the force applied to it in that direction and its mass:

  $$a_x = \frac{F_x}{m_i} \qquad a_y = \frac{F_y}{m_i} \qquad a_z = \frac{F_z}{m_i}$$

The naïve approach to computing all of the F_{ij} would be to compute them for every single i and j (except when $i = j$ when it would be undefined). ***Newton's Third Law of Motion*** states that for every force there is an equal and opposite force, i.e. $F_{ij} = F_{ji}$ (the gravitational force that body i applies to j is the same as the force applied from j to i). This means that you only have to compute F_{ij} for $j < i$ which is half of the number of computations.

Numerical Integration

Once you compute the acceleration for a body, you can then compute its new velocity and position for the next time. Every time step you will have to compute a new acceleration for every single body since acceleration is dependent on the current position and causes the current position to change. The repetitive evaluation of the values is called numerical integration. We cannot solve the integrals mathematically, but we can evaluate them over-and-over again to get an estimate of their true values. For this particular problem, the best numerical integration technique is called leapfrog integration.

In leapfrog integration, we compute the acceleration (as above) for the current time t (s), use that to compute the velocity halfway to the next time step ($t + {\Delta t}/{2}$), and finally use that to compute the position for the next time step ($t + \Delta t$). Thus, all of the velocities are a half-step off from the positions – they are "leapfrogging" with the positions.

Overall, for each body:

- Compute its current accelerations a_x, a_y, and a_z as above, using the current positions
- Compute its velocities v_x, v_y, and v_z for time $t + {\Delta t}/{2}$ by adding $a_x \Delta t$, $a_y \Delta t$, and $a_z \Delta t$ to its velocities from time $t - {\Delta t}/{2}$ (respectively)
- Compute its positions x, y, and z for time $t + \Delta t$ by adding $v_x \Delta t$, $v_y \Delta t$, and $v_z \Delta t$ to its positions from time t (respectively)

And then repeat this for all time steps. As Δt is decreased the quality of the results increase since you are applying the approximate values over a smaller range before computing new values. However, decreasing Δt greatly increases the amount of computation required.

Reminder: every iteration you are using current values to compute future values, and must make sure that you don't update values for the current time until everyone has used them.

Programs

For this project, you must write several variations of this program. All of the programs take 5 or 6 command line arguments:

- time-step: a floating-point number > 0 in seconds that corresponds to Δt above
- total-time: a floating-point number $\geq \Delta t$ in seconds for total time to be simulated (T)
- outputs-per-body: number of positions to output per body
- input.npy: a file to load a matrix that contains the initial data
- output.npy: a file to write a matrix to that contains the output data
    - ***NOTE:*** first row is initial positions and last row is final positions computed regardless of the spacing of the remainder of the output positions
- num-threads *optional*: the number of threads to parallelize over
    - serial programs: don't have this argument at all
    - parallel programs: defaults to a reasonable value for the hardware/affinity

The input matrix npy file is an n-by-7 matrix with one row per body and columns:

- mass m_i (in kg)
- initial x, y, z position x_i, y_i, and z_i (in m)
- initial x, y, z velocity v_x, v_y, and v_z (in $\mathrm{m/s}$)

The code provided to you already reads these arguments and validates them. But that is all that is done. Look at the code for a list of variables established from the command-line arguments.

The only printed output (besides errors) is the time taken to compute everything in seconds (print with the `%g` format specifier). Your timing must include all computations, any operations you do to convert the data to a format you like better, and allocating memory. You should not include parsing the command-line arguments, loading the input file (just loading it, anything with its data must be timed), saving the output file (just saving it, anything with its data must be timed), computing/reporting the time taken, and cleanup (freeing memory/destroying objects).

Note: when `matrix_from_npy_path()` or `matrix_from_npy()` is used, the `Matrix` object is connected to the file, any changes to the `Matrix` you make will also change the file itself. You will either need to copy the `Matrix` (which must be timed) or never write to it.

You will make four programs:

- `nbody-s` – computes the results in serial using the naïve approach
- `nbody-s3` – computes the results in serial using Newton's 3rd Law of Motion
- `nbody-p` – computes the results in parallel using the naïve approach
- `nbody-p3` – computes the results in parallel using Newton's 3rd Law of Motion

Example Data and Scripts

On Canvas I have provided several example data sets along with their settings to run them. I also provide the expected output data so you can check if you code is operating properly. **All of these examples have 1000 output values per body.**

- sun-earth: 2-body setup with just the Sun and the Earth
  - expected results use $\Delta t = 1 \text{ min} = 60 \text{ s}$ and $T = 1 \text{ year} = 31557600 \text{ s}$
- solar-system-inner: 5-body setup with the Sun and Mercury, Venus, Earth, and Mars
  - expected results use $\Delta t = 1 \text{ min} = 60$, and $T = 1 \text{ year} = 31557600 \text{ s}$
- solar-system: 9-body setup with the Sun and all (official) planets
  - expected results use $\Delta t = 1 \text{ min} = 60 \text{ s}$ and $T = 1 \text{ year} = 31557600 \text{ s}$ or $T = 100 \text{ years} = 3155760000 \text{ s}$
- pluto-charon: 2-body setup that has is stable with the barycenter outside of both bodies
  - expected results use $\Delta t = 1 \text{ s}$ and $T = 3.5 \text{ days} = 302400 \text{ s}$[1]
- figure8: 3-body setup that has a stable figure-8 pattern
  - expected results use $\Delta t = 0.01 \text{ s}$, and $T = 2.11 \text{ s}$
- figure8-rotate: figure8 example with an initial angular momentum
  - expected results use $\Delta t = 1\text{e-}4 \text{ s}$, and $T = 75 \text{ s}$
- random25: 25-body setup with random values
  - expected results use $\Delta t = 0.1 \text{ s}$ and $T = 500000 \text{ s}$
- random100, random1000, and random10000: large setups with random values
  - expected results use $\Delta t = 0.01 \text{ s}$ and $T = 1000 \text{ s}$

I am providing you with several Python scripts for checking the results, visualizing the results, and generating new inputs. You run them like `python3 file_name.py args`…. For example, to check if 2 npy files are (almost) equal:

```
python3 compare_npy.py my_earth_out.npy earth_exp_out.npy
```

Running any of the scripts with the `-h` argument will show helpful information. Feel free to modify these files for new features and share them (including with me).

Benchmarks

You must reach these on Expanse using the above parameters except that $T = 10s$. Your minimum speed over a few runs must be smaller than these values (i.e. you must be faster).

To be published later…

[1] About half of Pluto's day, not sure why I didn't do 1 Pluto day (551856 seconds)

Hints and Guidelines

- Don't use the `pow()` function to compute powers
- When using a shared node on Expanse, `get_num_cores_affinity()` in util.h will give you the number of cores actually allocated for your task, otherwise it returns the number of ***logical*** cores
- There is a lot of math that is redundant and can be simplified/moved around to reduce the computational effort
- Start with testing your programs with the smaller examples and checking the results
- Start with the serial program in both cases and make sure it works
- When doing the 3rd-law programs make sure the threads are actually doing equal work (since $j < i$ is not the same for all values of i)
- Don't commit any of the npy files (I don't need or want them)
- Use two structs for the thread information: one that contains all the information that is the same between all threads and the other that has the info unique to a particular thread (this one also includes a pointer to the one that is shared between all threads)
- Explore AoS, SoA, and AoSoA versions of structs.
- Organize your code to reduce redundancies, for example, make a function in a separate file that just performs a single F_{ij} and use that function in all of your programs

Analysis

You will also submit an analysis of how your programs function, answering the following questions. Remember that when recording time to take the minimum of about 3 runs. Also, we are caring about high-performance for large examples here. While I did give you lots of small examples, that was just for testing. You should be doing your analysis on huge samples (***hundreds of thousands if not millions of bodies***). I am expecting a few plots or similar to convey all of the information in your analysis.

- What is the speedup and efficiency of the naïve parallel program compared to the serial program? Evaluate at several different sizes and talk about the trend. Is it scalable?
- Repeat the above but for the 3rd-law program.
- In what situations are the serial programs faster than the parallel programs? What is the "crossover point" (i.e. when they are the same speed)?
- In what situations is the naïve program faster than the 3rd-law program (if ever)? In what situations is the 3rd-law program faster than the naïve program (if ever)? If they are faster in different circumstances, find the "crossover point".

Rubric

10 pts for `nbody-s`
10 pts for `nbody-s3`
10 pts for `nbody-p`
10 pts for `nbody-p3`
25 pts for sufficient speed in all cases (limits will be published)
20 pts analysis
15 pts code quality