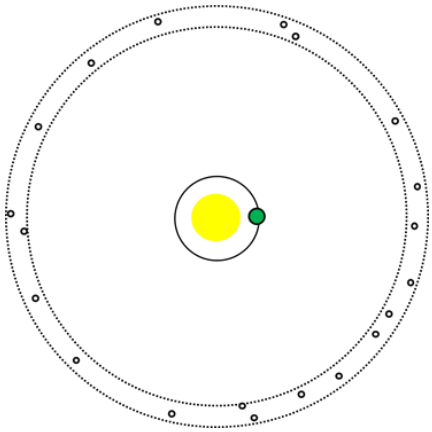
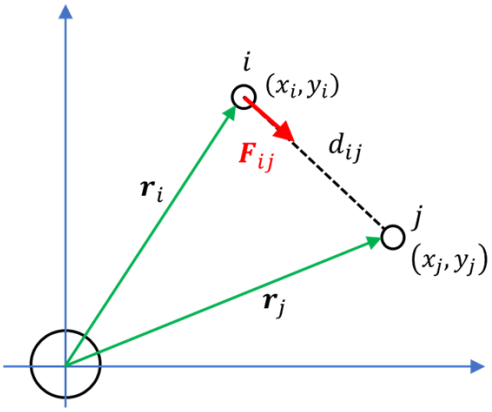
N-Body Simulation

CBP -02-04-22

This is all about N-bodies moving under gravitational attraction. I have in mind the following scenario, simulating the Earth, Sun and a bunch of asteroids. The code will be written in C (template provided) which will generate an Octave file to plot out the orbits (provided).The task is to code the simulation (it involves some loops) then to parallelize it and measure the speed-up. No need to understand the physics, which comes next, only the algorithm is important.

The Physical Model





The scenario is shown on the left, the green Earth orbiting the Sun, with a few asteroids shown in the 'asteroid belt', the diagram is almost to scale. Now the physics. The force between two masses $m_1$and $m_2$is attractive (negative sign) and is given by

$$F = -G\frac{m_1 m_2}{d_{12}^2}$$

Note how the distance d_{12}^2 appears in the denominator (bottom). This means that the larger the separation between the masses, the smaller the force. The above expression gives us the *magnitude* of the attractive force. \the distance between the two masses is just

$$d_{ij} = \sqrt{\left[\left(x_i - x_j\right)^2 + (y_i - y)^2\right]}$$

In 2D (or 3D) we can write the expression for force as a vector equation (vectors are bold)

$$\boldsymbol{F}_{ij} = -G\frac{m_1 m_2}{d_{ij}^2}\frac{(\boldsymbol{r}_i - \boldsymbol{r}_j)}{d_{ij}}$$

This is the force *on mass i due to mass j* as shown in the above diagram. There are two components on the right side of this expression. First, we have the *magnitude* of the force,

$$-G\frac{m_1 m_2}{d_{ij}^2}$$

and second, we have a term that gives us the direction of the force

$$\frac{(\boldsymbol{r}_i - \boldsymbol{r}_j)}{d_{ij}}$$

This is just the vector between the two masses divided by the length of the vector, in other words it has length = 1.0 but points from from the *j*'th mass to the *i*'th mass. The negative sign reverses this, so that the force vector on *i* from *j* points towards *j* i.e. *j* attracts *i.*

Now we can write this as force components to prepare for coding

$$F_{ij}^{(x)} = -G\frac{m_1 m_2}{d_{ij}^2}\frac{(x_i - x_j)}{d_{ij}}$$

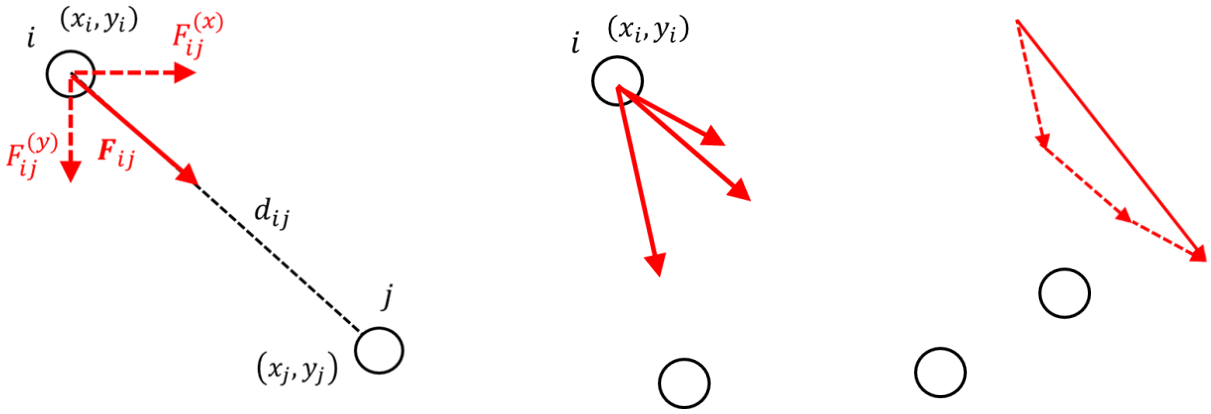
$$F_{ij}^{(y)} = -G\frac{m_1 m_2}{d_{ij}^2}\frac{(y_i - y_j)}{d_{ij}}$$

Now each object *j* will apply a force to object *i* so the total force on object *i* will be

$$F_i^{(x)} = \sum_{j\neq i}^{N} F_{ij}^{(x)}$$

$$F_i^{(y)} = \sum_{j\neq i}^{N} F_{ij}^{(y)}$$

Here's some graphics to help. Below left you can see the x and y-components of a single force. On the right you can see how the forces from three masses (*j*) add to produce a summed force on *i*. Note that the closer the masses are together the larger the forces. The detail (rightmost) shows how the three forces add together.



When we know the total force on the object *i* then we calculate its acceleration, then speed, then update its location, all using x-y components.

Finally, we need to use the forces to get the masses to move. Using Newton's second law, we can update the components of the masses' velocity after a time interval Δt, e.g.,

$$\Delta v_i^{(x)} = \frac{F_i^{(x)}}{m_i}\Delta t$$

and use the new velocity to update the position of the mass

$$\Delta x_i = v_i^{(x)}\Delta t$$

Algorithm

This is written out for the x-component only. Replicate lines for the y-component. The following loops will be contained in an outer loop which runs for **`nrItns`** which is a parameter.

for each body *i* {
  set $F_i^{(x)}$ = 0

  for each body *j* which is not *i* {
    calc the force on *i* due to *j* $F_{ij}^{(x)}$
    then sum the force into i $F_i^{(x)}$ += $F_{ij}^{(x)}$
  }

  use $F_i^{(x)}$ to find accel, speed and
  hence calculate new position
}

Coding

The template code does everything except the main computational loop which you will parallelize. The bodies are instances of the following **struct**

```
struct Body_struct {
    double mass;
    double pos[2];
    double vel[2];
};
```

scalar mass
vector x,y position
vector vX, vY speeds

You will see in the code that any array **`bodies[]`** is available whose elements are the above struct.

You will need X and Y components of vectors. These are arrays of dimension 2. Here's an example of how to use these vectors, for the example of the total force on the i'th object. X and Y are indices 0 and 1 defined in nbody.h.

```
tot_force_i[X] = 0.0;
tot_force_i[Y] = 0.0;
```

The following variables are declared at the top of **main()**; you will need these to code your loops. Their relationship to the maths symbols is shown. You are likely to code the expressions in the order shown (refer to detailed pseudo-code below)

`r[X]` `r[Y]`	$x_i - x_j$ $y_i - y$	Components of vector from *j* to *i.*
`dist`	$d_{ij} = \sqrt{\left[\left(x_i - x_j\right)^2 + (y_i - y)^2\right]}$	Length of above vector
`force_mag`	$F = -\frac{Gm_i m_j}{d_{ij}^2}$	Magnitude of attractive gravity force
`force_ij[X]` `force_ij[Y]`	$F_{ij}^{(x)} = F\left(\frac{x_i - x_j}{d_{ij}}\right)$ $F_{ij}^{(y)} = F\left(\frac{y_i - y_j}{d_{ij}}\right)$	x and y-components of the force **from *j* to *i*.**
`tot_force_i[X]` `tot_force_i[Y]`	$F_i^{(x)} = \sum_{j \neq i}^{N} F_{ij}^{(x)}$ $F_i^{(y)} = \sum_{j \neq i}^{N} F_{ij}^{(y)}$	x and y-components of total force ***on i.***
`dv[X]` `dv[Y]`	$\Delta v_i^{(x)} = \frac{F_i^{(x)}}{m_i}\Delta t$ $\Delta v_i^{(y)} = \frac{F_i^{(y)}}{m_i}\Delta t$	Updating x and y-components of the velocity of ***body i***.

Pseudo-code

for loop iterating over **`nrBodies`** using index i
  // First deal with forces on *i* from *j*
  set **`tot_force_i[X]`** to zero
  and same for Y
  for loop iterating over **`nrBodies`** using index j
    if j is the same as i bypass all the following
    get the difference between the **`body[i] pos[X]`** and **`body[] pos[X]`**
    do the same for **`pos[X]`**
    calc **`dist`** which is the length of the vector with the above components
    calc **`force_mag`** the magnitude of the gravity force on *i* from *j*
    calc the X component and *add* it to **`force_ij[X]`**
    same for the Y
  end for

  // Now update velocities and speeds
  calc **`dv[X]`** the velocity change using **`tot_force_i[X]`** and **`bodies[i].mass`** and dT
  same for dv[Y]
  update the bodies velocity **`bodies[i].vel[X] +=`** …
  same for Y
  update bodies position **`bodies[i].pos[X] +=`** … using the updated velocity and dT
end for

Test your Code

Make sure the line of code `writeDataToOctave(logfile, time, bodies, nrBodies);` at the head of the main computational loop is active. Run a simulation with **one thread** and a small number of asteroids. In Octave, run the script **log.m** and you should see the Earth orbit and those of your asteroids. If these are not circles, you must revisit your code. When all OK comment out the above line.

Parallelization

You should choose a suitable **for**-loop. I suggest you use the **#pragma omp parallel for private(*<list>*)** directive where <list> is a comma-separated list of variables you want to be private to each thread, in other words NOT SHARED. That may take some trial and error.

Investigation

This is a huge computational problem with nested loops. I ran some successful simulations with the following parameter set

nrItns = 5000; **1000 asteroids (user input)** **1 to 16 threads**

A Touch more Physics

Calculating Object orbital speeds

When a mass moves in a circle then the force acting to keep it on the circle is

$$-\frac{mv^2}{r}$$

so, when the force is supplied by gravity, from a central mass *M* (think Sun) then we have

$$-\frac{mv^2}{r} = -G\frac{Mm}{r^2}$$

After a bit of simplification, solving for *v* we have

$$v = \sqrt{\frac{GM}{r}}$$

This is used in the code to fix the initial velocity of the bodies around the Sun given the distance *r* we place them from the Sun. Then they will move on circular orbits. Any other speed will result in an elliptical path or a hyperbolic path (the object flies off to infinity).

Also look at the expression for *v*. The mass *m* of the object has disappeared! So, the speed does not depend on the mass of the object! So, all objects at the same distance from the Sun will orbit with the same speed!

Scaling

Some of the numbers we could use are frighteningly large. The mass of the Sun is over 10^{30} kg and the Earth is over 10^{24}kg. The distance between Earth and Sun is around 150 million km. Putting such large numbers into variable might be troublesome, even with doubles. So, instead we choose to *scale* the original equations. We scale as follows

(i) Divide all distances by the Sun-Earth distance. So, the Earth's orbit is now 1 unit.

(ii) Divide all masses by the mass of the Sun. So, all masses are less than 1

(iii) Set G = 1 (its physical value was around 10^{-11} metres-cubed per kg second squared).

Taking these scaling factors into account, the effect is to make simulated time run faster by around 5,000,000 times. So the physical Earth takes 365 days to orbit the sun, but the simulated earth takes 6.3171 secs (unless I made a boo-boo). Which is why it makes sense to simulate.