The N-body Problem – Part 1 (Two Bodies)

This assignment must be completed individually. Working in groups is not permitted.
Due Friday, January 18th at the start of class.

Background Theory

0.1 Equations of Motion

This project will involve using formulas from physics and calculus which we will justify first. We will develop these formulas here in a conversational style.

Dude: What's goin' on with you?

Maude: Bodies are gravitating.

Dude: Really? What do you mean?

Maude: I'll try to keep things simple, but not too simple. How about two bodies?

Dude: What do you mean by "bodies"?

Maude: Masses. Globs of matter that we idealize as points in space that have mass, which we'll measure in kilograms.

Dude: Measure for yourself–I've heard about these "kilos."

Maude: Dude, you need to learn to code. Let's start by defining a class, getting it to behave, providing I/O, and chaining it together by piping data from one program to another. That's a nontrivial beginning. This is encouraging! How much more do we need before we can let the integrator integrate?

Dude: Let's start with the simplest integrator possible, the Euler method...and for the simplest N-body system: I like your 2-body problem.

Maude: I'm glad you don't have a problem with that problem...but we need a solution. Newton's equations of motion for N bodies form a system of N differential equations, and the challenge to obtain solutions is called the N-body problem.

Dude: Okay, but let's start with the 2-body problem, which is really a 1-body problem of course, since you only have to solve the relative motion between the two particles.

Maude: You say "of course," but if this is going to be a presentation for students, we'd better be more explicit. We should start with Newton's equations of motion for a gravitational many-body system: Suppose the position in the x-y plane of masses m_1 and m_2 at time t are given by

$$\vec{u}_1(t) = (x_1(t), y_1(t))$$
$$\vec{u}_2(t) = (x_2(t), y_2(t))$$

Then the velocity and acceleration vectors are found by finding the first and second order time rates of change (differentiating) so that the velocities are

$$\vec{v}_i(t) = \frac{d}{dt}\vec{u}_i(t) = \left(\frac{d}{dt}x_i(t), \frac{d}{dt}y_i(t)\right)$$

and the accelerations are

$$\vec{a}_i(t) = \frac{d}{dt}\vec{v}_i(t) = \left(\frac{d^2}{dt^2}x_i(t), \frac{d^2}{dt^2}y_i(t)\right)$$

for $i = 1, 2$.

Dude: That's great, Maude. But this body is still at sea. I don't see how we get these meters per second per second for the acceleration. Maybe I'm just a little slow...

Maude: That's ok, acceleration is bound for you. As Hy Zaret and Lou Singer penned in 1959

> There is no disputin', there is no refutin'
> We're all indebted to Sir Isaac Newton
> Because because because...

We can compute the acceleration by combining the expressions for the gravitational force between the masses from Newton's Universal Law of Gravitation: $F = \frac{G \cdot m_1 \cdot m_2}{r^2}$, (here $G \approx 6.67408 \times 10^{-11}$ is the gravitational constant and $r^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$ is the square of the distance between the objects) with the expression for the force from Newton's second law: $\vec{F} = m \cdot \vec{a}$

Equating these expressions for $\vec{F}$ we have

$$m_2 \cdot \vec{a_2} = \frac{G \cdot m_1 \cdot m_2}{r^2} \cdot \frac{\vec{r}}{r}$$

Dude: Maybe ***you*** have. God bless the child that's got his own. What's that $\frac{\vec{r}}{r}$ multiplying the right side?
Maude: How droll of you to inquire so...that, my dear Dude, is the unit vector (vector of length 1) pointing from the mass m_2 back to m_1, which is the direction of the force of attraction from the second mass to the first. Notice that the quantity on the left is a vector, so the quantity on the right should also be a vector with the same magnitude, dude, and direction.

We...well, ***I*** can easily solve this for $\vec{a}_2$:

$$\vec{a}_2 = \frac{G \cdot m_1}{r^3} \cdot \vec{r}$$

Dude: Ok, and the vector $\vec{r}$ is then the vector from the second mass to the first. How do ***you*** get that, because ***I*** don't get it.
Maude: You must have missed that lecture in trigonometry: You subtract the coordinates of m_1 from the coordinates of m_2:

$$\vec{r} = (x_2 - x_1, y_2 - y_1)$$

Dude: Nice, Maude! You were always better at school. Now ***we*** can equate components so that the x

and y components of the acceleration are

$$a_x(t) = \frac{G \cdot m_1}{r^3} \cdot (x_2 - x_1)$$
$$a_y(t) = \frac{G \cdot m_1}{r^3} \cdot (y_2 - y_1)$$

Maude: Good one, Dude! And $r^3 = \left((x_2 - x_1)^2 + (y_2 - y_1)^2\right)^{3/2}$, so all we need to compute these acceleration components are the coordinates of the points where m_1 and m_2 are located.

Dude: But then where do these globs of mass go from there?

Maude: It depends on their velocities. If they were starting from rest, they would just head straight for one another and collide at their center of mass, or if they're going fast enough in some other direction they'd be just, like, "Adios, Dudes!", but if they're like the planets and moons of our solar system, they'd be in some more-or-less stable orbit. Did you get through the integral calculus, Dude?

Dude: Not yet...but I haven't given up hope for the future!

Maude: Ok, well this isn't too tough, I'll just tell you how to do it. Technically the velocity components are found by integrating:

$$v_x(\Delta t) = v_x(0) + \int_0^{\Delta t} a_x(t)dt \approx v_x(0) + a_x(0) \cdot \Delta t$$
$$v_y(\Delta t) = v_y(0) + \int_0^{\Delta t} a_y(t)dt \approx v_y(0) + a_y(0) \cdot \Delta t$$

and then you integrate again to find the new position:

$$x(\Delta t) = x(0) + \int_0^{\Delta t} v_x(t)dt \approx x(0) + v_x(0) \cdot \Delta t$$
$$y(\Delta t) = y(0) + \int_0^{\Delta t} v_y(t)dt \approx y(0) + v_y(0) \cdot \Delta t$$

...focus on the approximation on the right side, which essentially just says that for a nearly linear relation, $y(t) \approx y_0 + m \cdot t$, assuming t is small enough so that the approximation is good.

Repeat this to find subsequent velocities and positions in the same way.

So, Dude: go forth and compute!

1. Start by creating a "Body" object in a header file, `Body.h`, with member variables for position, velocity and mass. Use vectors to represent the position and velocity and write appropriate constructors for creating a Body. Let's start with a `Vector2d` class that will be used for all sorts of 2-dimensional vectors need to represent position, velocity and acceleration as the system evolves under that Newtonian rules, plus a string field for the name:

```
#pragma once
#include <vector>
#include <string>

struct Vector2d {
    double x, y;
    // write a prototype for the constructor here
};

class Body {
private:
    //create member variables for name, position, velocity and  mass;
public:
    // Write a constructor here.
    //---------
    // Write getter and setter functions for velocity, position and mass
    // The mass will be set once by the constructor, so it doesn't need a sette
};

// Define the constructor
```

2. Below is proposed starter code for `main()` showing the creation of Earth and Moon, a `vector`⟨`Body`⟩ to contain them, a small time increment, `deltaT` and an infinite loop where in the `update()` function takes the `bodies` vector (by reference) and uses the theory described above to compute new positions for the vectors. The units used here are in the so-called "mks" system (meters/kilograms/seconds) for distance, mass and time. Earth starts at the origin with velocity vector {`0,0`} and has a mass of 5.972×10^{24} kilograms. The Moon starts on the x-axis at 3.85×10^{7} meters, speeding into the first quadrant with a velocity vector {`0,1023.056`} m/s and a mass of 7.35×10^{22} kilograms.

   To check that the system is evolving in a reasonably stable way, I wrote the `squareDistance()` function to see if the Earth/Moon distance wasn't changing to radically:

```
double squareDistance(Body a, Body b) {
return ((a.getPosition().x - b.getPosition().x)
      * (a.getPosition().x - b.getPosition().x)
      + (b.getPosition().y - b.getPosition().y)
      * (b.getPosition().y - b.getPosition().y));
}
```

   Write the `update()` function that will follow the Newtonian laws. Be careful not to actually change a body's position until all the updates have been computed. That is, we want to model the forces to be acting simultaneously, so if you move the Moon before computing its force on Earth, then you're out of sync.

```
int main() {
    //create bodies
    Body Earth("Earth", {0, 0}, {0,0}, 5.972e24);
    Body Moon("Moon", { 3.85e7, 0}, {0,1023.056 }, 7.35e22);
    std::vector<Body> bodies{ Earth,Moon };
    //set time increment
    double deltaT = 1e-5;
    //update positions
    int i{ 0 };
    while (1) {
        update(bodies, deltaT);
        if (++i % 200000 == 0) {
            cout << "dist = " << sqrt(squareDistance(bodies[0], bodies[1]))
                 << endl;
            for (int i = 0; i < bodies.size(); ++i)
                printBods(bodies[i]);
        }
        if(i%800000==0)
            cin.get();
    }
}
```

Here's a shell of starter code for the `update` function

```
void update(std::vector<Body>& bodies, double dT) {
    // Define vectors to hold acceleration and velocity/position updates for ea
    // .......
    // Write a loop for computing total acceleration of each body.
    // This involves the distance r between the bodies, which you can compute
    // separately as sqrt(squareDistance(Body,Body);
    //..............
    // Maybe an output here to make sure you're getting what you want:
    /*for (int i = 0; i < bodies.size(); ++i) {
        cout << "acc[" << i << "]=(" << accelerations[i].x << ", "
             << accelerations[i].y << ")\n";
    }
    cin.get();*/
// Compute changes in velocity as acceleration * dT
    // .............
    // Use the changes is velocity to update the velocities
    // ...........
    // Compute changes in position Vector2ds
    // Use the changes in position to update the positions
}
```

Here's a typical run of the program:

```
dist = 3.85e+07
time:        2   Earth      x-coor 0.00661896 y-coor 1.17256e-07
time:        2    Moon      x-coor 3.85e+07 y-coor 2046.11
dist = 3.85e+07
time:        4   Earth      x-coor 0.0264758 y-coor 9.38047e-07
time:        4    Moon      x-coor 3.85e+07 y-coor 4092.22
dist = 3.85e+07
time:        6   Earth      x-coor 0.0595704 y-coor 3.16591e-06
time:        6    Moon      x-coor 3.85e+07 y-coor 6138.34
dist = 3.85e+07
time:        8   Earth      x-coor 0.105903 y-coor 7.50438e-06
time:        8    Moon      x-coor 3.85e+07 y-coor 8184.45

dist = 3.85e+07
time:       10   Earth      x-coor 0.165473 y-coor 1.4657e-05
time:       10    Moon      x-coor 3.85e+07 y-coor 10230.6
dist = 3.85e+07
time:       12   Earth      x-coor 0.238281 y-coor 2.53273e-05
time:       12    Moon      x-coor 3.85e+07 y-coor 12276.7
dist = 3.85e+07
time:       14   Earth      x-coor 0.324327 y-coor 4.02188e-05
time:       14    Moon      x-coor 3.85e+07 y-coor 14322.8
dist = 3.85e+07
time:       16   Earth      x-coor 0.423611 y-coor 6.00351e-05
time:       16    Moon      x-coor 3.85e+07 y-coor 16368.9

dist = 3.85e+07
time:       18   Earth      x-coor 0.536133 y-coor 8.54796e-05
time:       18    Moon      x-coor 3.85e+07 y-coor 18415
dist = 3.84999e+07
time:       20   Earth      x-coor 0.661893 y-coor 0.000117256
time:       20    Moon      x-coor 3.84999e+07 y-coor 20461.1
dist = 3.84999e+07
time:       22   Earth      x-coor 0.80089 y-coor 0.000156068
time:       22    Moon      x-coor 3.84999e+07 y-coor 22507.2
dist = 3.84999e+07
time:       24   Earth      x-coor 0.953126 y-coor 0.000202618
time:       24    Moon      x-coor 3.84999e+07 y-coor 24553.3
```

...and so on...