N-Body problem

Simulate a (tiny) universe

version 2

Ismail Bennani
<ismail.lahkim.bennani@ens.fr>

1 Project

1.1 Problem description

This problem consists of solving the equations of motions of N arbitrary bodies interacting in a gravitational field of dimension 2.
Let $(c_i)_{i\in[1,N]}$ be our bodies, at each instant $t \in \mathbb{R}$ and for each body c_i, we write $m_i \in \mathbb{R}$ its mass, $\mathbf{p_i}(t) \in \mathbb{R}^n$ its position, $\mathbf{v_i}(t) \in \mathbb{R}^n$ its speed and $\mathbf{a_i}(t) \in \mathbb{R}^n$ its acceleration.
For each other body c_j, we write $r_{i,j}(t) = \|\mathbf{p_j}(t) - \mathbf{p_i}(t)\|$ the euclidian distance between those bodies and $\mathbf{u_{i,j}}(t) = \frac{1}{r_{i,j}(t)}(\mathbf{p_j}(t) - \mathbf{p_i}(t))$ the unit vector from $\mathbf{p_i}(t)$ to $\mathbf{p_j}(t)$.

The gravitational force applied by a body $j \in [1, N]$ on a body $i \in [1, N], i \neq j$ is:

$$\mathbf{F_{i,j}}(t) = G\frac{m_i m_j}{r_{i,j}^2(t)}\mathbf{u_{i,j}}(t)$$

The dynamic of the system is then described by the following equations:

$$\forall i \in [1, N], \begin{cases} \dot{\mathbf{p_i}}(t) = \mathbf{v_i}(t) \\ \dot{\mathbf{v_i}}(t) = \mathbf{a_i}(t) \\ \mathbf{a_i}(t) = \dfrac{1}{m_i} \displaystyle\sum_{j\in[1,N], j\neq i} \mathbf{F_{i,j}}(t) \end{cases}$$

1.2 Expected work

You will write a simulator capable of computing the successive positions in time of N bodies, this simulator will have to use your own Ordinary Differential Equation solver.
The reason this subject is long is to give you several possible improvement paths so that you can choose **what interests you most**. I do not expect you to do

everything presented here.
For example, you could:

- implement optimizations to increase the computation speed of the physics (cf. section 2.1)
- implement different ODE solvers (cf. section 2.2)
- implement additional features in your graphical interface (cf. section 2.3)
- find stable orbits automatically (cf. section 2.4)
- handle collisions (cf. section 2.5)

Every group should implement at the very least a naive version of the physics engine and the ODE solver, that is **without** external libraries. To help you get started, a code skeleton is given to you. It has a class architecture (and interfaces) made to help you with the implementation.
Once you have those naive versions implemented, you can replace some of them with an interface to existing libraries, e.g. `numpy` for vector and matrix operations.

You will have to submit a PDF containing:

- a link to a git repository containing all your source code (please add a fichier requirement.txt if you are using python)
- a concise report explaining your work, in particular:
  - the distribution of tasks in your group
  - the external library used
  - how your code works overall
  - the parts you have improved, the algorithms you have implemented
  - the difficulties you encountered and how you solved them, the technical choices you had to make

You should have a preliminary report ready by the half of the course, I will read it and give you some feedbacks.

I suggest that you use Python as a programming language, but if you prefer using something else please come talk to me.
In section 2.6, I mention some useful libraries for Python, you are free to use any library you want. However, be sure to write in your report what you have implemented yourself and what is from an external library.

The value of your code will not be measured by the number of lines! Write a code that is correct, commented (don't overdo it) and clear, in that order. For example, be sure to choose explicit and informative names for you variables, your functions and your classes.

2 Implementation

Sections 2.1, 2.2 and 2.3 describe **one** way to implement the simulator from the given skeleton. You **are not** required to follow this implementation.
Sections 2.4 et 2.5 are here to give you improvement ideas for your simulator, and some pointers to ressources that will help you achieve them.
In this project, we will not use the universal system units. The SI units are such that the values we work with have very different orders of magnitude (speed of the earth relative to the sun $\approx 10^4$ m.s^{-1}, distance earth-sun $\approx 10^{11}$, weight of the sun $\approx 10^{30}$ kg, ...). Thereby, the floating point approximation errors are all the more important. If you are using real world values in your simulations, I advise you to use the following units:

- distance: parsec (pc)
- mass: solar mass ($M_\odot$)
- speed: kilometer per second (km.s^{-1})
- gravitational constant: pc.${M_\odot}^{-1}$.km^2.s^{-2}

Unit Tests: the given skeleton already contains some unit tests that your code should pass. Add your own tests on top of those as your progress in your development.

2.1 Physics Engine

The purpose of the physics engine is to compute an approximation of the dynamics of the problem. On the one hand it computes the values that the ODE solver will integrate, and on the other hand it provides the graphical interface with the necessary data to show the bodies on screen (positions, speeds).
In regards to collisions between bodies, this engine can either try to handle them in a more or less realistic fashion (make the bodies bounce on collision, make them merge, ...) or ignore them altogether (you will assimilate a body to its center of mass). The methods explained here ignore collisions, refer to section 2.5 if you wish to handle them.
Note: Even if you decide to assimilate the bodies to their center of mass, there is a small probability that two points end at the exact same position, don't forget to handle this case.

Note that the speed of this module is crucial if you want to simulate a big number of bodies. Indeed, the dynamic of the system has to be recomputed at least once per step of the ODE solver.

- **Naive method** $\mathcal{O}(N^2)$
  The most naive method is the one where the accelerations of a body are computed by iterating throught the other $N-1$ bodies and applying the

formula above.
However, this can be sped by noticing that

$$\mathbf{F_{i,j}} = -\mathbf{F_{j,i}}$$

Thanks to this remark, we can get the same result as before with only half the computations.

- **Barnes-Hut Simulation** $\mathcal{O}(Nlog(N))$
  We can reduce the number of computations by sacrificing a bit of accuracy. This method consists of constructing a tree (octree) in which the leaves are the bodies, and the nodes represent portions of space (*cell*) containing their children. In each cell we define a virtual body located at the center of the cell, and whose mass is equal to the sum of the masses of all the bodies in its cell.
  Then, in order to compute the gravitational force applied to a specific body, we can replace the contribution of the groups that are too far away by that of the virtual body of their containing cell.
  https://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation
  http://arborjs.org/docs/barnes-hut

- **Fast multiple methods** $\mathcal{O}(N)$
  https://en.wikipedia.org/wiki/Fast_multipole_method
  A Hierarchical $\mathcal{O}(N)$ Force Calculation Algorithm

Note
The provided skeleton implements a Vector class that overloads the usual arithmetic operators $+$, $-$, $*$, etc.. Operators overload: explanation, list of operators

2.2 Ordinary Differential Equations solver

The purpose of the ODE solver is to integate the dynamics of the system. It receives an initial state and derivatives of the dynamics from the physics engine then computes the new state at a given horizon.
Solving an ODE is computing the value of a function $\mathbf{y} : \mathbb{R} \to \mathbb{R}^k$ defined by

$$\forall t \geq 0, \dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)) \qquad \mathbf{y}(0) \text{ connue}$$

at specific times. We write $(t_n) \in \mathbb{R}^{\mathbb{N}}$ the times at which we want to compute the value of $\mathbf{y}$, and $\mathbf{y_n} = y(t_n)$.

Note that the system defined in 1.1 is an first order ODE:

$$\forall i \in [1, N], \begin{cases} \dot{\mathbf{p}}_\mathbf{i}(t) = \mathbf{v_i}(t) \\ \dot{\mathbf{v}}_\mathbf{i}(t) = \mathbf{a_i}(t) \\ \mathbf{a_i}(t) = \dfrac{1}{m_i} \displaystyle\sum_{j \in [1,N], j \neq i} \mathbf{F_{i,j}}(t) \end{cases}$$

can be written as

$$\forall i \in [1, N], \begin{bmatrix} \dot{\mathbf{p_i}}(t) \\ \mathbf{v_i}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{v_i}(t) \\ \frac{1}{m_i} \sum_{j \in [1,N], j \neq i} \mathbf{F_{i,j}}(t) \end{bmatrix}$$

$\mathbf{y}(t)$ is therefore a $2*i$ dimensional vector containing the positions and velocities of the bodies.

- **Naive method: explicit Euler**
  This is a first order approximation of the dynamics, it is cheap to compute but unstable and unprecise:

  $$\forall n \geq 0, \mathbf{y_{n+1}} = \mathbf{y_n} + (t_{n+1} - t_n) * \mathbf{f}(t_n, \mathbf{y_n}) \qquad \mathbf{y_0} = \mathbf{y}(0)$$

  http://www.maths.lth.se/na/courses/FMN050/media/material/part14.pdf

- **Leapfrog algorithm**
  This is a second order approximation, it is more expansive to compute than explicit euler but also more stable for oscillatory movements. It is specific to situations where you want to compute a position and a velocity from known accelerations, which is exactly our case.
  Let $i \in [1, N]$, we write $p_n^i = p_i(t_n)$, $v_n^i = v_i(t_n)$, $a_n^i = a_i(t_n)$ and $\Delta t_n = t_{n+1} - t_n$, and we have the following approximation:

  $$\forall i \in [1, N], \forall k \in \mathbb{N}, \begin{cases} p_{k+1}^i = p_k^i + \Delta t_n v_k^i + \frac{1}{2} {\Delta t_n}^2 a_k^i \\ v_{k+1}^i = v_k^i + \frac{1}{2} \Delta t_n (a_k^i + a_{k+1}^i) \end{cases}$$

  https://en.wikipedia.org/wiki/Leapfrog_integration

- **Runge Kutta family**
  This describes one algorithm that can be used at several orders, these methods are among the most used. Here is a small selection of useful links that should help you understand these methods and implement them.
  - http://www.scholarpedia.org/article/Runge-Kutta_methods
  - A family of embedded Runge-Kutta formulae, Dormand and Prince
  - Coefficients for the study of Runge-Kutta integration processes, Butcher

- **Implicit Euler**
  This is a first order approximation. Implicit means that the approximation at step $n+1$ is not defined as an explicit function of the values at step n, but as the solution of a fixed point equation. The implicit methods are usually very expensive to compute because they require significantly more

evaluation of the derivative function $\mathbf{f}$, but they are also more stable and generally more precise:

$$\forall n \geq 0, \mathbf{y_{n+1}} = \mathbf{y_n} + (t_{n+1} - t_n) * \mathbf{f}(t_{n+1}, \mathbf{y_{n+1}}) \qquad \mathbf{y_0} = \mathbf{y}(0)$$

We are looking for y_{n+1} the fixed point of the function

$$\mathbf{G}(\mathbf{x}) = \mathbf{y_n} + (t_{n+1} - t_n) * \mathbf{f}(t_{n+1}, \mathbf{x})$$

Several methods exist to solve such a problem, for example the Newton-Raphson method.
https://www.f-legrand.fr/scidoc/docimg/numerique/euler/implicite/implicite.html

2.3 Graphical interface

The purpose of the interface is first of all to show the state of the system on screen. Secondly, it can be used to interact with the system: for example to add a body when the left mouse button is clicked, or delete a body on right click.
I suggest that you use `pygame` as a drawing library but you can use whatever library you like.

- **Pygame** https://www.pygame.org
  They have an excellent tutorial on their website.

I advise that you decorrelate the units of your physics engine with the ones used by the graphics library. Write methods to change between the base of the world (physics) and the base of your camera (screen), that is methods that will convert a world coordinate to a screen coordinate and vice versa. That will enable you to control the scale at which you want to render your world, and also to move your viewpoint during simulation.

2.4 Stable orbits

If you wish to make nice drawings, you can simulate stable orbits (also called *choregraphies*), that is choose initial positions of the bodies such that, after some time, their movement becomes periodic. Finding these stable orbits is not an easy problem, there have been numerous research works on this subject. Here is a small list of pointers that might be usefull if you wish to deepen this part of the project.

- An accessible explanation http://www.scholarpedia.org/article/N-body_choreographies
- Research paper: Classification of symmetry groups for planar n-body choreographies, it describes a family of solution for arbitrary N, illustrated here
- Thesis: Periodic Solutions to then n-Body Problem, some solutions in chapter 8

2.5 Collisions

If you wish to handle collisions, your bodies c_i need to have a size. You can for example assimilate a body to a sphere of radius r_i.
The problem of handling collisions can be solved in two steps: find all the collisions that happened at a given step, then resolve them, that is compute the new positions and velocities of the bodies involved.

Detection First of all, we can approximate the instant of a collision by assuming that they can only happen at the specific instant at which you sample your dynamics, that is at one of the t_n. In that case, at each step, you need to look for couple of bodies that are overlapping (the case of 3 bodies or more overlapping at the same time is more complex and can be ignored as a first approximation).
Another solution is to monitor the instants t_i and t_{i+1} such that two bodies are overlapping at t_{i+1} but not at t_i, then look for a more precise instant in $[t_i, t_{i+1}]$ at which the collision happened. For that you need to:

- interpolate the positions between those instants: the simplest interpolation is the linear one, but some ODE solvers like the ones in the Runge-Kutta family compute polynomial interpolations of the solution as part of their computations, this can be used instead.
- approximate the instant at which the collision happened in the interpolation: dichotomy, Newton-Raphson method, ...

Resolution Once that a collision has been detected, you need to handle it: that is make it so the bodies don't overlap anymore, then update their respective velocities to take into account the collision.
We could also, in the case of stars for example, make the bodies merge. This would also require to compute a new position for the resulting body, a new mass and a new velocity.

Note that the collision detection algorithm are usually very expensive: the most naive one tests all of the $\frac{N(N-1)}{2}$ couple of bodies at each step of the simulation. Some methods exist to reduce the number of computations, they try to discriminate, for each body, the ones that are too far away (no computation needed) from the ones that are close enough that they should be tested more precisely. The Barnes-Hut simulation method uses such a partitioning of the space (cf. section 2.1).
Else here is a paper dealing specifically with this topic: Fast Collision Detection among Multiple Moving Spheres.

2.6 Tools

Here are some tools that might help you. I remind you that you **must** implement at least the naive versions of each module of the project, it is only afterwards that you can replace some of your modules with interfaces to one of these libraries. I expect that you improve some parts of your projects, but you will not have time to implement everything that is discussed in this subject. Feel free to use existing libraries.

- numpy: scientific computation library. You will probably not be able to write faster vectorial and matricial operations on a CPU.
- SciPy: numerical algorithms library. You will find ODE solvers, interpolation algorithms, etc..

Voici quelques outils qui peuvent vous aider. Je rappelle que vous **devez** implémenter au moins les versions naïves de chaque module du projet, ce n'est que par la suite que vous pouvez remplacer un de vos modules naïf par une interface vers une de ces librairies. J'attends de vous que vous approfondissiez certaines parties du projet, mais vous n'aurez pas le temps de tout faire vous-même, alors n'hésitez pas à utiliser des librairies déjà existantes.

- numpy: librairie de calcul scientifique. Vous ne pourrez pas faire plus efficace que ça pour vos opérations vectorielles et matricielles sur CPU.
- SciPy: librairie d'algorithmes numériques. Vous y trouverez des algorithmes d'intégration déjà fait, des algorithmes d'interpolation, d'optimisation.