IN104: N-Body Problem

Ismail Bennani
ismail.lahkim.bennani@ens.fr

April 27, 2021

N-Body Problem

Given N bodies at positions $p_i = (x^i, y^i)$ with velocities $v_i = (v_x^i, v_y^i)$, accelerations $a_i = (a_x^i, a_y^i)$ and masses m_i, simulate their dynamics and show them on screen.

DEMO

Dynamics of the system:

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

where

$$\mathrm{F}_{\mathrm{i,j}}(t) = G \frac{m_i m_j}{r_{i,j}^2(t)} \mathrm{u}_{\mathrm{i,j}}(t)$$

A first implementation

DEMO

Problem 1: Data structures

Problem 1: Data structures

```
bodiesX = [400, 450]
bodiesY = [300, 350]
bodiesVX = [0, 0]
bodiesVY = [0, -0.02]
bodiesMass = [10, 1]
```

Need a structure to pack the positions, velocities and mass of the body

Body

```
diffX = bodiesX[j] - bodiesX[i]
diffY = bodiesY[j] - bodiesY[i]
...
unitX = diffX / norm
unitY = diffY / norm
...
bodiesAX[i] += unitX * G * bodiesMass[i] \
    * bodiesMass[j] / (norm * norm)
bodiesAY[i] += unitY * G * bodiesMass[i] \
    * bodiesMass[j] / (norm * norm)
```

Need a structure to represent a vector of dim 2 and overriden operators

Vector2

Data structures

Body

```python
class Body:
    def __init__(self, position, velocity=Vector2(0, 0), mass=1, color=(255, 255, 255), draw_radius=50):
        self.position = position
        self.velocity = velocity
        self.mass = mass
        self.color = color
        self.draw_radius = draw_radius

    def __str__(self):
        return "<pos:%s, vel:%s, mass:%.2f>" % (self.position, self.velocity, self.mass)
```

This class is used to pack the data of a body.

`color` and `draw_radius` are used by `Screen` to draw the planets.

Data structures

World

```python
class World:
    def __init__(self):
        self._bodies = []

    def add(self, body):
        """ Add `body` to the world.
            Return a unique ID for `body`.
        """
        new_id = len(self._bodies)
        self._bodies.append(body)
        return new_id

    def get(self, id_):
        """ Return the body with ID `id`.
            If no such body exists, return None.
        """
        if (id_ >= 0 and id_ < len(self._bodies)):
            return self._bodies[id_]
        return None

    def bodies(self):
        """ Return a generator of all the bodies. """
        for body in self._bodies:
            yield body

    def __len__(self):
        """ Return the number of bodies """
        return len(self._bodies)

    def __str__(self):
        return "Bodies: %d\n\t%s" % \
            (len(self),
             '\n\t'.join([str(i) + ": " + str(self._bodies[i])
                          for i in range(len(self))]))
```

This class is used to represent the state of the world, it stores all the bodies in a list and exposes methods to add and get them.

The method `bodies` is a **generator**, it makes it possible to iterate through the list of bodies without actually generating a list in memory.

Generators are a useful tool, read the doc: https://wiki.python.org/moin/Generators

Data structures

Vector, Vector2

vector.py

Vector represents a vector of dimension n. It overrides the arithmetic operators $+$, *, - and $/$, and it defined the *array access* operator `[]`.

It is used by the solver to represent its state (4 dimension per body: x,y,v_x and v_y).

Vector2 inherits from Vector and is a version that represents vectors of dim. 2. In particular it can be initialized using its constructor: `Vector2(x, y)`, it also provides getters and setters methods for x and y: `get_x`, `get_y`, `set_x` and `set_y`.

It is used by `Body` to represent its position and velocity, by `Screen` to represent its size and by `Camera` to represent its position.

Problem 2: Modularity

Problem 2: Modularity

A well thought program is composed of different modules with well-defined interfaces.

Modules should not care about the implementation of other modules.

This way, any functionality can be reimplemented without affecting other ones.

Example: Solver I

In the simple example, I used explicit euler to approximate the dynamics.

$$\forall n \geq 0, y_{n+1} = y_n + dt * f(t, y_n) \qquad y_0 \in \mathbb{R}$$

Implementation:

```
for i in range(len(bodiesX)):
    bodiesX[i] += dt * bodiesVX[i]
    bodiesY[i] += dt * bodiesVY[i]
    bodiesVX[i] += dt * bodiesAX[i]
    bodiesVY[i] += dt * bodiesAY[i]
```

Example: Solver II

Leapfrog integration:
with x_i the positions, v_i the velocities and a_i the accelerations:

$$\forall n \geq 0, \begin{array}{l} v_{i+1} = v_i + \frac{h}{2}(a_i + a_{i+1}) \\ x_{i+1} = x_i + h * v_i + \frac{h^2}{2} * a_i \end{array}$$

This method needs a_i and a_{i+1} to compute the state of the system at step $i+1$. We would need to update the code to keep track of the last accelerations that we computed.

But how could we implement both methods, and switch between them easily ? This is what modularity is about.
We should use classes !

Modularity: Interfaces

Historically called facade pattern, it is a very powerful **pattern design**, (cf Beck, Kent; Cunningham, Ward (September 1987). *Using Pattern Languages for Object-Oriented Program*. OOPSLA '87 workshop).

An interfaces is a description of the methods that a class **must** implement. A modular code should have well-defined interfaces. Additionally, interfaces allows to decouple the code: a module A that needs the features of a module B should depend on the *interface* of B, not an implementation.

In some languages, there is a primitive way to implement interfaces, **but not in python**.

Modularity: Interfaces

Example

In the skeleton code for this project, interfaces are classes with a name that starts with I, their methods are not implemented. Example:

```
class IEngine:
    def __init__(self, world):
        self.world = world

    def derivatives(self, t0, y0):
        raise NotImplementedError

    def make_solver_state(self):
        raise NotImplementedError
```

DO NOT modify the code of this interface, if you want to implement an engine, you should create a new class and inherit from `IEngine` and reimplement the methods `derivatives` and `make_solver_interface`

A modular approach to the problem

A modular approach to the problem

Solver

```python
class ISolver:

    # NOTE: our systems do not depend on time,
    # so the input t0 will never be used by the
    # the derivatives function f
    # However, removing it will not simplify
    # our functions so we might as well keep it
    # and build a more general library that
    # we will be able to reuse some day

    def __init__(self, f, t0, y0, max_step_size=0.01):
        self.f = f
        self.t0 = t0
        self.y0 = y0
        self.max_step_size = max_step_size

    def integrate(self, t):
        """ Compute the solution of the system at t
            The input `t` given to this method should be increasing
            throughout the execution of the program.
            Return the new state at time t.
        """
        raise NotImplementedError
```

This is the ODE solver, the `integrate` method computes the dynamic of the system until time `t`.

A modular approach to the problem I

Engine

```python
class IEngine:
    def __init__(self, world):
        self.world = world

    def derivatives(self, t0, y0):
        """ This is the method that will be fed to the solver
            it does not use it's first argument t0,
            its second argument y0 is a vector containing the positions
            and velocities of the bodies, it is laid out as follow
                [x1, y1, x2, y2, ..., xn, yn, vx1, vy1, vx2, vy2, ..., vxn, vyn]
            where xi, yi are the positions and vxi, vyi are the velocities.

            Return the derivative of the state, it is laid out as follow
                [vx1, vy1, vx2, vy2, ..., vxn, vyn, ax1, ay1, ax2, ay2, ..., axn, ayn]
            where vxi, vyi are the velocities and axi, ayi are the accelerations.
        """
        raise NotImplementedError

    def make_solver_state(self):
        """ Returns the state given to the solver, it is the vector y in
                y' = f(t, y)
            In our case, it is the vector containing the
            positions and speeds of all our bodies:
                [x1, y1, x2, y2, ..., xn, yn, vx1, vy1, vx2, vy2, ..., vxn, vyn]
            where xi, yi are the positions and vxi, vyi are the velocities.
        """
        raise NotImplementedError
```

A modular approach to the problem II
Engine

This is the physics engine, given the positions and velocities of the planets (`y0`) and their characteristics (stored in `world`), it computes their accelerations.

`make_solver_state` is used to convert from the representation used by the engine (`world` of type `World`) to the representation used by the solver (a `Vector` with 4 dimension per body).

A modular approach to the problem

Simulator

```python
class Simulator:
    def __init__(self, world, Engine, Solver):
        self.t = 0
        self.world = world

        self.engine = Engine(self.world)

        # Engine uses World to represent the state
        # of the world while Solver uses a
        # vector to represent the current state of
        # the ODE system.
        # The method Engine.make_solver_state computes
        # the vector of state variables (the positions
        # and velocities of the bodies) as a Vector

        y0 = self.engine.make_solver_state()

        self.solver = Solver(self.engine.derivatives, self.t, y0)

    def step(self, h):
        y = self.solver.integrate(self.t + h)

        for i in range(len(self.world)):
            b_i = self.world.get(i)

            b_i.position.set_x(y[2 * i])
            b_i.position.set_y(y[2 * i + 1])

            b_i.velocity.set_x(y[len(self.world) + 2 * i])
            b_i.velocity.set_y(y[len(self.world) + 2 * i + 1])

        self.t += h
```

This is the main class of the program. It instantiates an engine and a solver and exposes a `step` function that simulates `h` seconds. This concludes the logic part, we still need to handle the graphics.

A modular approach to the problem

Camera

```python
class Camera:
    def __init__(self, screen_size):
        self.screen_size = screen_size
        self.position = Vector2(0, 0)
        self.scale = 1

    def to_screen_coords(self, position):
        """ Converts the world-coordinate position to a screen-coordinate. """
        raise NotImplementedError

    def from_screen_coords(self, position):
        """ Converts the screen-coordinate position to a world-coordinate. """
        raise NotImplementedError
```

This very simple class handles makes it possible to move and scale the view easily, it implements a base change from the coordinate system of the world to the one defined by `self.position` and `self.scale`.

A modular approach to the problem

Screen

I implemented a basic screen class to help you get started. You must refer to pygame documentation to implement new features.

I **do not** expect you to test this class.

Tests

Tests

Some tests are already provided, once you implement all the missing methods, you should pass all the tests.
These tests are also a good example of how the features are used, take a look.

Additionally, testing is an important part of software development, write your own tests as you write new features.
Remember: tests should be about interfaces, not implementations.
You should always check that a feature does **what** it is supposed to do, not **how** it does it.
In particular, you should not be testing internal methods of a class as they are implementation details that should be irrelevant.

Second step

Implement a better solver, a faster physics engine, a better screen, with more features, find stable orbits, etc..

cf sujet.pdf