N-Body Newtonian Simulator in 3D Physical Space using Python

Nanometric System Simulation - Nanoscience and Nanotechnology UAB 2022/23

Xabier Oianguren Asua

1 The Model and its Assumptions

According to the postulates of classical mechanics [1], a system of N particles can be completely described by knowing the time evolution of the position of each of the N particles in three dimensional space $\mathbb{R}^3$ (or some subset of it in which the particles are constrained to move). We will denote the position of the k-th particle with a three component vector $\vec{x}_k = (x_k^1, x_k^2, x_k^3)$, while the trajectory of this particle will be described by the curve $\vec{x}_k(t) = (x_k^1(t), x_k^2(t), x_k^3(t))$, with $t \in T \subset \mathbb{R}$.

Let us denote the force or influence exerted by the j-th particle on the k-th particle as $\vec{F}_{kj}(\vec{x}_k(t), \vec{x}_j(t)) = (F_{kj}^1, F_{kj}^2, F_{kj}^3)$, which depends exclusively on the position of both particles at that particular time (in principle only on the relative position for the space to be homogeneous and isotropic), and not explicitly on time (by the conservation of total energy). Then, the postulate of the conservation of total momentum implies that there must exist a force exerted by the k-th particle on the j-th such that: $\vec{F}_{jk}(\vec{x}_j(t), \vec{x}_k(t)) = -\vec{F}_{kj}(\vec{x}_k(t), \vec{x}_j(t))$ (also known as the action-reaction law). Let us denote the total force on the k-th particle as

$$\vec{F}_{Total\ on\ k}(t) := \sum_{j=1;\ j\neq k}^{N} \vec{F}_{kj}(\vec{x}_k(t), \vec{x}_j(t)). \tag{1}$$

Finally, the sum of all the forces on a particle is defined to be the time variation of the momentum of each particle, which is the velocity $\frac{d\vec{x}_k(t)}{dt} \equiv \vec{v}_k(t)$ of the particle times a weighting parameter called its mass $m_k(t)$

$$\frac{d}{dt}\Big(m_k(t)\frac{d\vec{x}_k(t)}{dt}\Big) = \vec{F}_{Total\ on\ k}(t)) \quad \text{for each particle } k \in \{1, 2, ..., N\}. \tag{2}$$

If we assume that the mass of each particle is constant, this leads us to the well known Newton's second law

$$m_k \frac{d^2\vec{x}_k(t)}{dt^2} = \vec{F}_{Total\ on\ k}(t) \quad \text{for each particle k}\in\{1,2,...,\text{N}\}. \tag{3}$$

Thus, as a system of $3N$ second order differential equations, if we know at a certain initial time t_0 the position of the N particles $\vec{x}_1(t_0), ..., \vec{x}_N(t_0)$ and their velocities $\vec{v}_1(t_0), ..., \vec{v}_N(t_0)$, by the existence and uniqueness theorems for the initial value problem [1], there exists a unique solution for the dynamics of the system.

2 The Method

In order to solve the equation system, we will employ the well-known Verlet algorithm [2] that allows a symmetric time evolution under time reversal, which is key for the energetic stability of the numerical simulation. This finite difference scheme consists on first expanding the trajectory in Taylor series for $|\varepsilon| << 1$

$$\vec{x}_k(t+\varepsilon) = \vec{x}_k(t) + \varepsilon\frac{\mathrm{d}}{\mathrm{d}t}\vec{x}_k(t) + \frac{\varepsilon^2}{2}\frac{\mathrm{d}^2}{\mathrm{d}t^2}\vec{x}_k(t) + \frac{\varepsilon^3}{3!}\frac{\mathrm{d}^3}{\mathrm{d}t^3}\vec{x}_k(t) + O(\varepsilon^4), \tag{4}$$

which given Δt is the time increment considered in the discretization of the trajectory, by evaluating $\varepsilon \in \{\Delta t, -\Delta t\}$ leads us to

$$\vec{x}_k(t+\Delta t) + \vec{x}_k(t-\Delta t) = 2\vec{x}_k(t) + \frac{\mathrm{d}^2\vec{x}_k(t)}{\mathrm{d}t^2}\Delta t^2 + O(\Delta t^4). \tag{5}$$

We can then obtain an estimate of the position of the k-th particle at time $t + \Delta t$ with an error on the order of Δt^4 with

$$\vec{x}_k(t + \Delta t) \simeq 2\vec{x}_k(t) - \vec{x}_k(t - \Delta t) + \frac{\mathrm{d}^2\vec{x}_k(t)}{\mathrm{d}t^2}\Delta t^2. \tag{6}$$

This expression requires the knowledge of the trajectory in the two previous time-steps t and $t - \Delta t$ and the knowledge of the acceleration $\vec{a}_k(t) := \frac{\mathrm{d}^2\vec{x}_k(t)}{\mathrm{d}t^2}$ in the previous time-step t. We might wonder how this equation may be used if we only know the positions $\vec{x}_k(t_0)$ and velocities $\frac{\mathrm{d}\vec{x}_k(t_0)}{\mathrm{d}t}$ at a single time t_0.

On the one hand, the acceleration is directly proportional to the total force (1) by equation (3), which only depends on the instantaneous positions of all the particles in the system at a single time. On the other hand, because we know the velocities $\frac{\mathrm{d}\vec{x}_k(t_0)}{\mathrm{d}t}$ at t_0, we can estimate with them for the first time iteration the positions at $t_0 - \Delta t$ with a Taylor series till first order

$$\vec{x}_k(t_0 - \Delta t) = \vec{x}_k(t_0) - \Delta t\frac{\mathrm{d}\vec{x}_k(t_0)}{\mathrm{d}t} + O(\Delta t^2). \tag{7}$$

The truncated version of this equation implies an error on the order of Δt^2, which will be assumed for the first time iteration. For the rest of iterations however, the equation (6), with an error on the order of Δt^4, can be used.

Finally, even if for the computation of the $\vec{x}_k(t)$ trajectory the explicit trajectory for the velocity $\vec{v}_k(t)$ will be unnecessary for $t > t_0$, it will be interesting to know it for plotting purposes. Thus, we will ease its estimation by the the usage of the truncation of equation (7) as

$$v(t) = \frac{\vec{x}_k(t) - \vec{x}_k(t - \Delta t)}{\Delta t}. \tag{8}$$

3 The Implementation

First of all, we generated a script `N_body_simulator_plot_live.py`, that can be found in our Github repository [3], that allows the simulation and live visualization of the time evolution of a system of N Newtonian particles in 3D. We then generated the script `N_body_simulator_plot_gif.py` that allows the simulation of the same systems but which instead of showing the simulation live (which might be slow), generates a `.gif` animation after the simulation finishes. For this, the Python libraries numpy [4], matplotlib [5] and imageio [6] were employed.

Both scripts were tested using the Coulomb force [7] and Newtonian gravitational force [8] between the particles, as will be seen in the results and can be found in the scripts.

The core function doing all the calculations, called `run_N_Body_simulator`, can be given as input:

- The number N of particles to simulate.
- A matrix (2D numpy array) with the three initial coordinates of each particle in each row (three columns and N rows).
- A matrix (2D numpy array) with the three initial coordinates of each particle's velocity in each row (three columns and N rows).
- A list with the N masses m_k of the particles.
- A list with the different force functions that each particle exerts on the rest.
- A list with additional parameters to be passed to the force functions (the charges for exmaple).
- The initial time `t0`.

- The final time `tf`.
- The number of time iterations we wish in that interval, `timeIts`.
- A parameter `plotEvery` which states every how many iterations we wish to plot the system.
- A dictionary where we can introduce fixed limits of the plot box under the keys `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax`.
- An integer stating how many previous iterations long should the trace or stella left by each particle be visible (in order to have a better idea of the trajectories).
- The version in the `.gif` file generating script has an additional parameter `show_frames`, that can be set to `True` if one wishes to see the generated plots as they are saved. Default is `False`.

See in Listings 1 and 2 as an example, the core function `run_N_Body_simulator` and plotting function `plot_N_particles` for the live script. The core for the script generating the animation is slightly different in that it generates a temporary directory where the plots are saved as `.png` images, which are then merged into a single `.gif` and are erased all together with the temporary directory.

Listing 1: Core function "run_N_Body_simulator" that takes care for the computation of the time iterations and generation of the plot.

```
import numpy as np

def run_N_Body_simulator(N, positions_now, velocities_now, masses, additional_parameters,
                    force_list, t0, tf, timeIts, plotEvery, limits, J_trace):
    assert J_trace>=3, "For the Verlet algorithm, saving at least 3 time iterations is
    necessary, so choose J_trace>=3"
    # Create figure
    fig = plt.figure(figsize=(10,10))

    # Compute the times in which the simulator will compute a step:
    times = np.linspace(start=t0, stop=tf, num=timeIts)
    # Get time increment delta t
    dt = times[1]-times[0]
    # initialize positions
    positions = np.zeros((J_trace, positions_now.shape[0], positions_now.shape[1]))
    # for the first time iteration we will estimate the previous position of the
    # particle using a simple Euler rule given the initial velocities
    positions[:] = positions_now-dt*velocities_now
    positions[0,:,:] = positions_now # copy the same position in all J
    # constant to avoid repeated calculation
    dt2_masses = dt**2/(np.array(masses)[:,np.newaxis])

    for it, t in enumerate(times):
        # a Nx3 array (matrix) where we will save the forces in each time
        forces = np.zeros(positions_now.shape)

        # Step 1, compute the total force on each particle
        for k, xk in enumerate(positions[0]):
            for j, xj in enumerate(positions[0]): # each of the other particles
                if j!=k: # does not self-interact!
                    for force in force_list:
                        forces[k,:] += force(xk, xj, masses[k], masses[j],
    additional_parameters[k], additional_parameters[j])

        # Step 2, compute the position of the particles in the next time
        # first move all the positions one step onward
        positions[1:, :, :] = positions[:-1,:,:] # copy in the slots from 1 to J the ones that
     were in 0 to J-1

        positions[0,:,:] = 2*positions[1,:,:] - positions[2,:,:] + forces*dt2_masses

        # Step 3, compute the velcity in the next time for plotting purposes
        velocities_next = (positions[0,:,:]-positions[1,:,:])/dt

        # Plot the particles
        if it%plotEvery==0:
            plot_N_particles(fig, positions, velocities_next, t,
```

```
                                limits['xmin'], limits['xmax'], limits['ymin'],
                                limits['ymax'], limits['zmin'], limits['zmax'])

        # prepare for the next time iteration
        velocities_now = velocities_next
        # for the position it is already prepared
```

Listing 2: Plotting function "plot_N_particles".

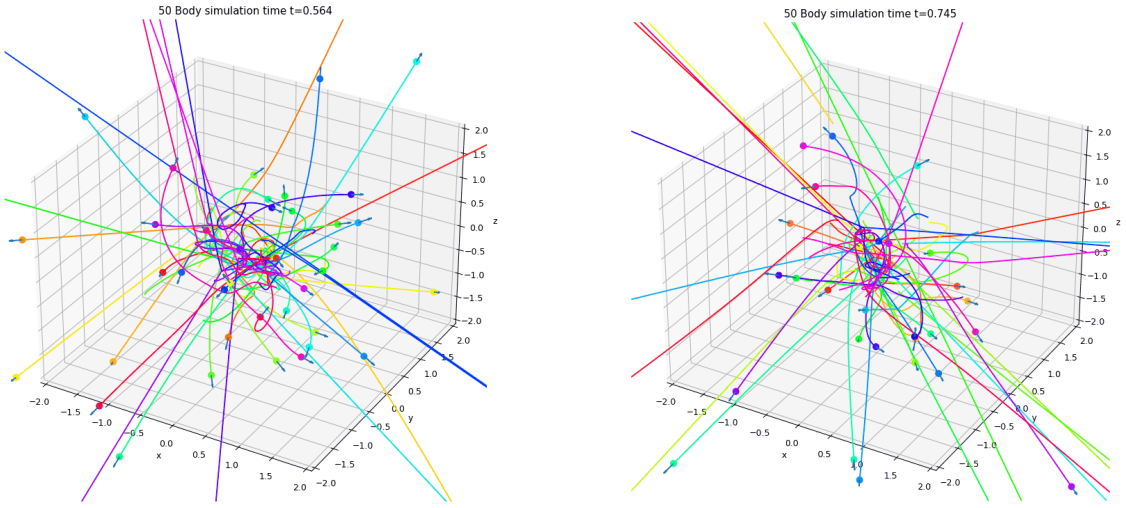
```
import numpy as np
import matplotlib.pyplot as plt

def plot_N_particles(fig, positions, velocities, t, xmin=-1, xmax=1, ymin=-1, ymax=1, zmin=-1,
     zmax=1):
    fig.clf()
    # Clear previous plots
    ax = fig.add_subplot(111, projection='3d')
    cmap = matplotlib.cm.get_cmap('hsv') # to have the same colors of th trace and the body

    # Plot particles
    ax.scatter3D(positions[0,:,0], positions[0,:,1], positions[0,:,2],
                 c=cmap(np.arange(positions.shape[1])/positions.shape[1]), s=50)

    # plot velocity vectors
    ax.quiver(positions[0,:,0], positions[0,:,1], positions[0,:,2],
              velocities[:,0], velocities[:,1], velocities[:,2], length=0.2, normalize=True)


    # plot the traces
    for j in range(positions.shape[1]): # for each particle its trace separately
        ax.plot3D(positions[:,j,0], positions[:,j,1], positions[:,j,2], '-', c=cmap(j/
    positions.shape[1]) )


    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    ax.set_xlim((xmin, xmax))
    ax.set_ylim((ymin, ymax))
    ax.set_zlim((zmin, zmax))
    ax.set_title(f"{positions.shape[1]} Body simulation time t={t:4.3}")

    fig.canvas.draw_idle()
    fig.canvas.flush_events()
```

3.1 Cool Celestial Mechanics: the 3 Body Problem

Both under gravity and the Coulomb interaction, most initial conditions lead to chaotic motion of the particles. However, there are some very specific conditions in which the particles describe visually satisfying stable orbits. In fact their search is still today an active research topic and the discovery of a new stable orbit is celebrated. In particular for $N = 3$ and gravity (the well known three body problem), Ref. [9] has gathered in a `json` file a dataset of interesting intial conditions with their names and authors, even with links to their original papers. In the third script of the repository `3_body_problem.py`, we implemented a menu that shows the user the avilable author, condition and mode list and lets them choose. The chosen option is then simulated and a `.gif` animation is generated.

4 Some Results

We gathered some of the resulting animations under the directory `/Example_Results` of the github repository [3], but we plot here a shot of some for completeness.



(a) **(b)**

Figure 1: Examples of $N = 50$ particles with random initial positions, velocities, masses and charges, all normally distributed with mean 0 and standard deviation 1 (the mass is taken to be the absolute value).

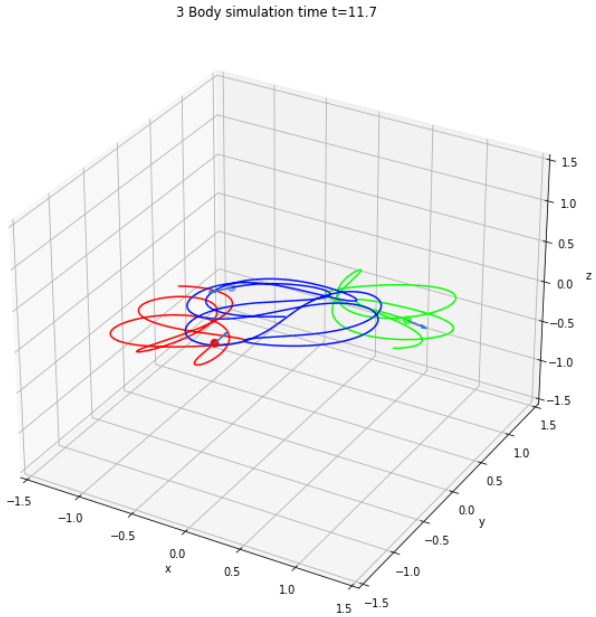


Figure 2: Example of a 3 body stable orbit provided by Ref. [9]. Authors: M. Suvakov, Orbit name: Yin Yang, III.3.A.

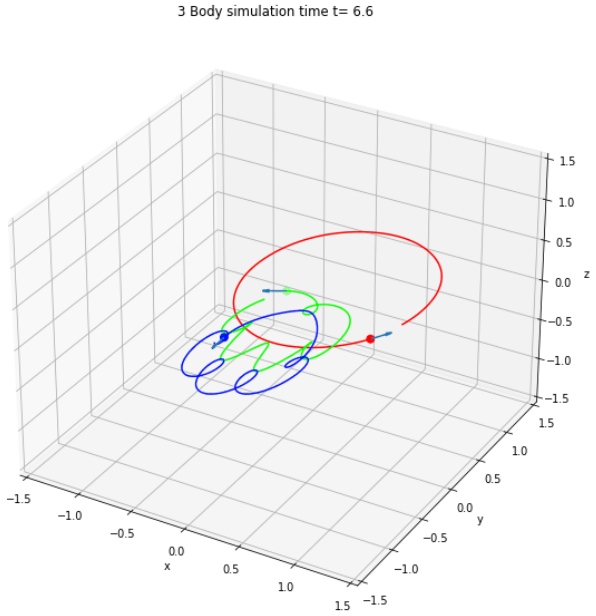


Figure 3: Example of a 3 body stable orbit provided by Ref. [9]. Authors: Matthew Sheen, Orbit Name: Hand-in-hand-oval.

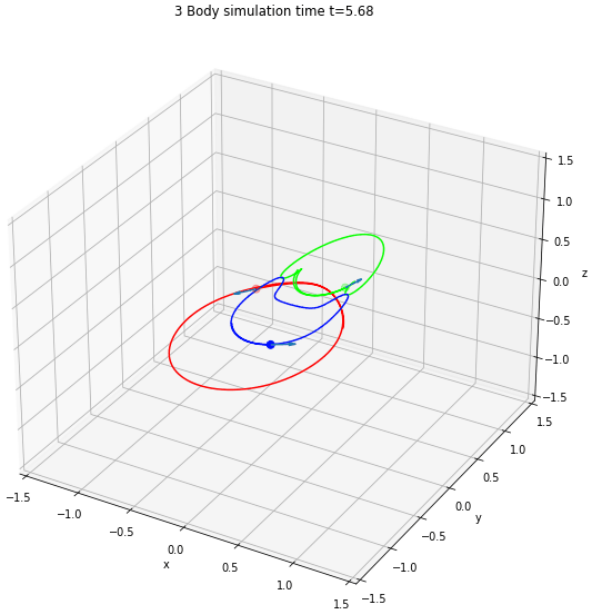


Figure 4: Example of a 3 body stable orbit provided by Ref. [9]. Authors: Matthew Sheen, Orbit Name: Oval, Catface and Starfish. Note that the name is quite literal.

References

[1] V. I. Arnol'd, *Mathematical methods of classical mechanics*, vol. 60. Springer Science & Business Media, 2013.

[2] D. Frenkel, B. Smit, and M. A. Ratner, *Understanding molecular simulation: from algorithms to applications*, vol. 2. Academic press San Diego, 1996.

[3] "Github repository with the python scripts and notebook generated for the report." https://github.com/Oiangu9/_Miscellaneous/tree/main/SSN.

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[5] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[6] A. Klein, S. Wallkötter, S. Silvester, A. Tanbakuchi, actions user, P. Müller, J. Nunez-Iglesias, M. Harfouche, Dennis, A. Lee, M. McCormick, OrganicIrradiation, A. Rai, A. Ladegaard, T. D. Smith, H. van Kemenade, G. Vaillant, jackwalker64, J. Nises, M. Komarčević, rreilink, lschr, M. Schambach, Andrew, C. Dusold, C. Gohlke, DavidKorczynski, F. Kohlgrüber, G. Yang, and G. Inggs, "imageio/imageio: v2.26.0," Feb. 2023.

[7] "Wikipedia entry of the vectorial coulomb force." https://en.wikipedia.org/wiki/Coulomb%27s_law#Vector_form_of_the_law.

[8] "Wikipedia entry of the vectorial coulomb force." https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation#Vector_form.

[9] "Dataset with interesting initial conditions for the 3 body problem." https://observablehq.com/@rreusser/periodic-three-body-initial-conditions.