LomaVerse - A Differentiable N-Body Gravitational Simulator in Loma

Samvrit Srinath
UC San Diego, USA
sasrinath@ucsd.edu

Eric Huang
UC San Diego, USA
eth003@ucsd.edu

Abstract

Github Link: This report details the design and implementation of LomaVerse, a differentiable N-body gravitational simulator built in the Loma programming language. By leveraging automatic differentiation, we derive equations of motion directly from a Hamiltonian formulation of the system's energy, enabling a physics-driven approach to simulation. The platform supports multiple numerical integrators, including Symplectic Euler and 4th-order Runge-Kutta, and is validated through the successful modeling of diverse celestial configurations, from stable solar system analogues to the equilibrium dynamics of Lagrange points. We document significant challenges and solutions related to ensuring numerical stability and achieving interactive performance in a web-based visualization frontend. While the original goal of gradient-based trajectory optimization remains as future work, the resulting simulator provides a robust and validated foundation for continued research in differentiable physics.

Github Link

https://github.com/SamvritSrinath/LomaVerse

1 Introduction

Gravitational N-body simulations are foundational tools in astrophysics and aerospace engineering, providing the means to model phenomena like planetary dynamics and galactic evolution for which closed-form solutions are famously intractable. While numerical simulation can predict the evolution of a system from a given set of initial conditions, many critical problems in astrodynamics require optimization—for instance, determining a fuel-optimal trajectory for an interplanetary probe. Such tasks are often addressed with gradient-based methods, yet obtaining the necessary gradients from traditional, non-differentiable physics simulators can be a complex and error-prone undertaking.

This project confronts this challenge through the development of LomaVerse, a gravitational N-body simulator built from the ground up with differentiability as a core principle. By implementing the physics engine in the Loma programming language, we utilize its built-in automatic differentiation (AD) features to create a fully differentiable simulation pipeline. The primary objective was to leverage this capability for complex optimization tasks. While the full scope of trajectory optimization was not achieved, the work resulted in a robust 3D simulator that serves as a powerful proof of concept.

In this report, we present the architecture and implementation of LomaVerse. We begin by detailing the Hamiltonian mechanics that form our physical model and discuss how it naturally interfaces with Loma's AD framework. We then describe the implementation of the system, including the support for multiple numerical integrators (Symplectic Euler and RK4) and the interactive web-based GUI. We present results from several simulation scenarios that validate the physical accuracy of our engine, including a case study on Lagrange point stability. Finally, we discuss the key challenges encountered regarding performance and numerical stability, and outline the path for future work in trajectory optimization.

2 Physical Model

The dynamics of a system comprising n point masses interacting via gravity can be described using several formalisms. While the Newtonian approach provides a direct description of forces, this project utilizes the Hamiltonian formulation, which is more amenable to the energy-based structure of our differentiable programming approach.

2.1 Newtonian Formulation

In the Newtonian framework, the gravitational force exerted on a body i by a body j is given by Newton's law of universal gravitation:

$$\mathbf{F}_{ij} = G\frac{m_i m_j}{||\mathbf{r}_j - \mathbf{r}_i||^3}(\mathbf{r}_j - \mathbf{r}_i) \tag{1}$$

The total force on body i is the vector sum of forces from all other bodies. While this formulation is straightforward to implement for a basic simulation, deriving the gradients necessary for optimization tasks requires manual differentiation of these vector equations, a process that becomes increasingly complex as the number of bodies grows.

2.2 Hamiltonian Formulation

To better leverage Loma's AD capabilities, we adopted a Hamiltonian approach. This formalism describes the system's state using generalized coordinates $\mathbf{q} = (\mathbf{r}_1, \ldots, \mathbf{r}_n)$ and their conjugate momenta $\mathbf{p} = (\mathbf{p}_1, \ldots, \mathbf{p}_n)$, where for a body i in Cartesian coordinates, the momentum is $\mathbf{p}_i = m_i \mathbf{v}_i$.

The system's dynamics are governed by a single scalar function, the Hamiltonian $H(\mathbf{q}, \mathbf{p})$, which represents the total energy of the system ($H = T + U$). The kinetic energy T and potential energy U are given by:

$$T(\mathbf{p}) = \sum_{i=1}^{n} \frac{||\mathbf{p}_i||^2}{2m_i} \tag{2}$$

$$U(\mathbf{q}) = -\sum_{i=1}^{n} \sum_{j>i}^{n} G \frac{m_i m_j}{||\mathbf{r}_j - \mathbf{r}_i||} \tag{3}$$

Thus, the complete Hamiltonian for the N-body system is:

$$H(\mathbf{q}, \mathbf{p}) = \sum_{i=1}^{n} \frac{||\mathbf{p}_i||^2}{2m_i} - \sum_{i=1}^{n} \sum_{j>i}^{n} G \frac{m_i m_j}{||\mathbf{r}_j - \mathbf{r}_i||} \tag{4}$$

For numerical stability, the distance term is "softened" to $\sqrt{||\mathbf{r}_j - \mathbf{r}_i||^2 + \epsilon^2}$ to avoid singularities during close encounters. The time evolution of the system is then elegantly described by Hamilton's equations:

$$\dot{\mathbf{r}}_i = \frac{\partial H}{\partial \mathbf{p}_i} \tag{5}$$

$$\dot{\mathbf{p}}_i = -\frac{\partial H}{\partial \mathbf{r}_i} \tag{6}$$

These equations form a system of first-order ordinary differential equations (ODEs) that provide a direct pathway for applying automatic differentiation.

3 Differentiable Simulation in Loma

A primary goal of LomaVerse was to use Loma's AD capabilities to derive the equations of motion directly from the Hamiltonian, bypassing the need for manual differentiation.

3.1 Deriving Equations of Motion

By implementing the Hamiltonian function (Eq. 4) in Loma, we can generate its partial derivatives automatically. To compute the components of $\dot{\mathbf{r}}_i$ and $\dot{\mathbf{p}}_i$ from Eqs. 5-6, we utilize Loma's forward-mode AD. For example, to find the x-component of the force on body k, $-\partial H / \partial r_{k,x}$, we declare a differentiable version of the system state, set the derivative part ('dval') of only the $r_{k,x}$ component to 1.0, and call the differentiated Hamiltonian function. The derivative of the function's output is the desired partial derivative. This is encapsulated in helper functions within our Loma module.

3.2 Numerical Integration Methods

With the ability to compute the derivatives $\dot{\mathbf{r}}$ and $\dot{\mathbf{p}}$, we can integrate the system forward in time. The choice of integrator is critical for balancing accuracy, stability, and computational cost.

3.2.1 Symplectic Euler. Our initial implementation utilized the Symplectic Euler method, a first-order integrator particularly well-suited for Hamiltonian systems. Its strength lies in its ability to conserve a "shadow" energy over long integration periods, which prevents the secular energy drift that plagues many standard methods. Its update rules are:

$$\mathbf{p}_k(t + \Delta t) = \mathbf{p}_k(t) + \Delta t \cdot \dot{\mathbf{p}}_k(\mathbf{r}(t), \mathbf{p}(t)) \tag{7}$$

$$\mathbf{r}_k(t + \Delta t) = \mathbf{r}_k(t) + \Delta t \cdot \dot{\mathbf{r}}_k(\mathbf{r}(t), \mathbf{p}(t + \Delta t)) \tag{8}$$

While this method proved effective for stable orbital systems, its first-order nature means that achieving high accuracy for more complex dynamics requires a prohibitively small time step, Δt. During long-term simulations of the Solar System, we observed instabilities such as a drift in the Sun's position. Although the primary cause was traced to non-zero initial momentum in the system, the investigation highlighted the sensitivity of N-body problems to all sources of numerical error.

3.2.2 4th-Order Runge-Kutta (RK4). To achieve higher accuracy and provide a more robust tool for sensitive scenarios, we implemented the classic 4th-order Runge-Kutta method. For a system $\dot{\mathbf{Y}} = f(\mathbf{Y})$, the RK4 step is:

$$k_1 = \Delta t \cdot f(\mathbf{Y}_n)$$
$$k_2 = \Delta t \cdot f(\mathbf{Y}_n + k_1/2)$$
$$k_3 = \Delta t \cdot f(\mathbf{Y}_n + k_2/2)$$
$$k_4 = \Delta t \cdot f(\mathbf{Y}_n + k_3)$$
$$\mathbf{Y}_{n+1} = \mathbf{Y}_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

While not symplectic, RK4's higher-order accuracy provides superior precision over shorter durations for a given step size. This makes it a better choice for modeling chaotic interactions or fast flyby maneuvers where trajectory accuracy is paramount. Our final implementation allows the user to select the preferred integrator, offering a trade-off between long-term energy conservation and short-term precision.

4 System Implementation

The LomaVerse simulator is a full-stack application composed of three primary, decoupled components: a core physics engine written in Loma, a Python-based host server that manages the simulation lifecycle, and a web-based graphical user interface (GUI) for visualization and interaction. This architecture separates the high-performance numerical computation from the user-facing elements, creating a modular

and extensible system. The high level system diagram of the LomaVerse visualizer is below:

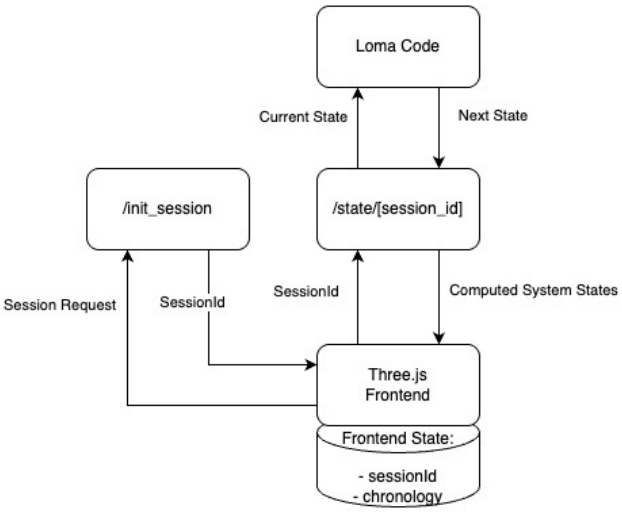

Figure 1: Overview of the LomaVerse architecture.

These components are discussed in further detail later.

4.1 Loma Physics Engine

The heart of the simulator is the physics engine, written entirely in Loma. Implementation details are provided in A. Its responsibility is to accurately model the Hamiltonian dynamics of an N-body system.

4.1.1 Core Data Structures. We define a set of simple structures in Loma to represent the state of the system, which are compiled directly into corresponding C structs. The fundamental types are:

- `Vec3`: A standard 3-component vector for position and momentum.
- `BodyState`: Holds the physical properties of a single celestial body, including its mass, inverse mass, position (`pos`), and momentum (`mom`).
- `SimConfig`: Contains simulation-wide constants such as the gravitational constant G, the time step Δt, and the number of active bodies.
- `BodyDerivative`: A utility struct used exclusively by the RK4 integrator to hold the time derivatives of a body's position and momentum ($\dot{\mathbf{r}}$ and $\dot{\mathbf{p}}$).

4.1.2 Hamiltonian Differentiation and Integration. The engine's core logic resides in its numerical integrators. Rather than manually coding the force calculations, we leverage Loma's automatic differentiation.

We define a function, `n_body_hamiltonian`, that computes the total energy of the system according to Eq. 4. We then apply Loma's `fwd_diff` to this function. This compiler pass automatically generates a new, differentiated function, `d_n_body_hamiltonian`, which can compute not only the Hamiltonian but also its partial derivatives.

Helper functions, such as `get_dH_dp_k_alpha`, use this generated function to calculate the specific derivatives needed for Hamilton's equations ($\dot{\mathbf{r}}_i = \partial H/\partial \mathbf{p}_i$ and $\dot{\mathbf{p}}_i = -\partial H/\partial \mathbf{r}_i$). This AD-driven approach provides the derivatives needed by our numerical integrators, `time_step_system` (Symplectic Euler) and `time_step_system_rk4` (4th-order Runge-Kutta), to advance the simulation state over a single time step, Δt.

4.2 Python Host and Compilation Pipeline

The backend, built with Python and the Flask web framework, orchestrates the entire simulation process, from compilation to serving data.

4.2.1 CTypes Foreign Function Interface. The Python host communicates with the high-performance C code in the compiled library via Python's built-in `ctypes` library. We dynamically define Python classes that mirror the C structs and map the compiled C functions (e.g., `time_step_system_rk4`) to Python-callable functions. Data, such as the array of body states, is allocated in Python and passed by reference to the C functions, allowing the physics engine to efficiently modify the state in-place with minimal overhead. For the RK4 integrator, which requires temporary storage, several "scratch space" arrays are also allocated in Python and passed to the C function to be used as temporary buffers.

4.2.2 Flask API Server. The server exposes several REST API endpoints to manage the simulation:

- `/init_session`: Initializes a new simulation. It takes a scenario name, triggers the compilation pipeline, creates a simulation runner (a Python closure containing the state and a reference to the C library function), and returns a unique session ID to the client.
- `/list_scenarios` and `/load_scenario`: Provide support for user-defined systems. The server reads scenario definitions from a local `/scenarios` directory and can initialize a new simulation from any chosen file.
- `/state/[session_id]`: The workhorse endpoint for running the simulation. On each call, it executes the simulation for a pre-configured number of frames (a "chunk") and returns the resulting array of states as a JSON object. This chunking mechanism is vital for maintaining a responsive frontend.

4.3 Frontend Visualization and Interaction

The frontend is a single-page application responsible for all rendering and user interaction, built with standard web technologies.

4.3.1 3D Rendering with Three.js. The visualization is built using the **Three.js** library. We construct a 3D scene containing a ‘PerspectiveCamera‘ with orbit controls, lighting, a background starfield, and a grid helper for spatial reference. Celestial bodies are represented as ‘THREE.Mesh‘ objects with a ‘SphereGeometry‘. To display planet names without them being obscured by other objects, we use the ‘CSS2DRenderer‘, which renders HTML elements that are overlaid on the 3D canvas and positioned according to the 3D world coordinates.

4.3.2 Asynchronous State Management and Rendering Loop. To ensure smooth animation, the frontend maintains a client-side buffer of simulation states (the ‘chronology‘ array). The main animation loop (‘requestAnimationFrame‘) renders one frame from this buffer at a time. When the number of remaining frames in the buffer drops below a certain threshold (e.g., enough for 5 seconds of animation), the client asynchronously fetches the next chunk of frames from the server’s ‘/state‘ endpoint and appends it to the buffer. This predictive fetching ensures that the renderer always has data to display, preventing stuttering and hiding the latency of the server requests.

4.3.3 2D/3D View Toggling. The view toggle is a purely client-side feature. In addition to the 3D trail objects, the application also maintains a simple ‘LinkedList‘ of 2D ‘[x, y]‘ coordinates for each body’s trail. When the user toggles the view, the JavaScript simply hides the 3D ‘WebGL‘ canvas and shows a standard 2D ‘Canvas‘ element. The animation loop then calls a separate ‘render2D‘ function, which uses the 2D Canvas API to draw the system from a top-down perspective using the pre-populated 2D trail data. Because the underlying simulation data is always 3D, switching views is instantaneous and does not require re-running the simulation.

5 Results and Scenario Analysis

Our implementation of LomaVerse successfully models a wide range of N-body configurations, validating the correctness of our Hamiltonian-based physics engine and the automatic differentiation pipeline. The following case studies highlight the simulator’s capabilities across several distinct scenarios, from stable, predictable systems to highly chaotic environments.

5.1 Case Study 1: Stable Binary and Planetary Systems

A fundamental test for any gravitational simulator is its ability to reproduce stable orbital mechanics. We validated LomaVerse against two key scenarios: a simple binary star system and a complex multi-planet system.

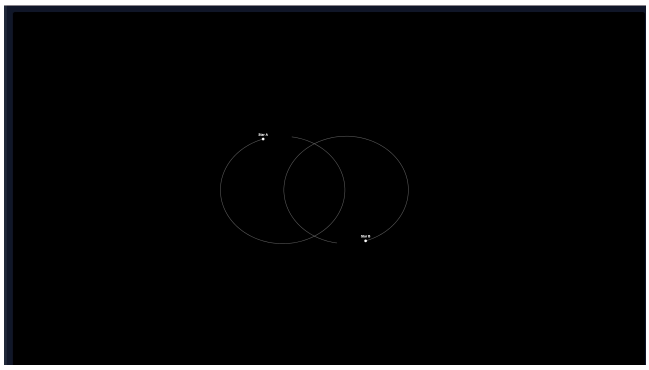

Figure 2: Binary Stars each of 1 Solar Mass having near perfect symmetry with apogee and perigee visible in 2D

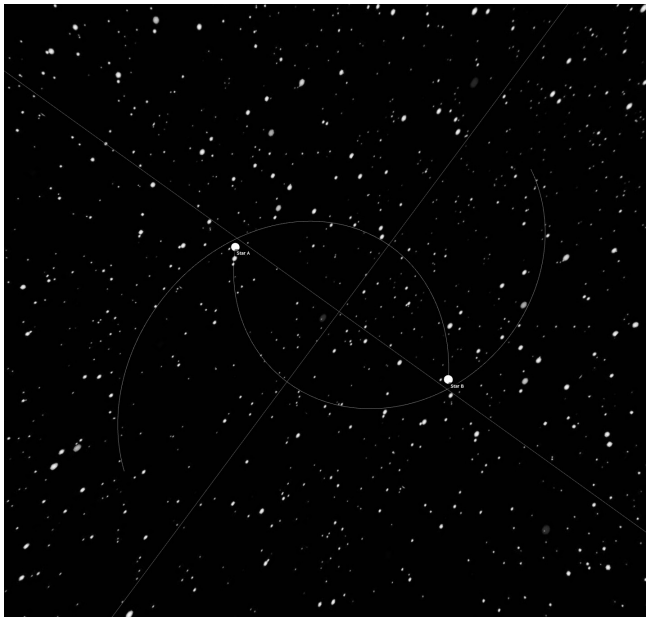

Figure 3: Binary star system with equivalent masses, rendered in 3D to show their orbital planes.

In the binary star simulation, two bodies of equal mass were initialized with velocities to place them in a stable orbit. The system demonstrated perfect, symmetrical orbits as shown in the 2D projection (Figure 2) and the 3D rendering

(Figure 3). Critically, the system conserved momentum and exhibited bounded energy error over long-term integration, confirming the stability of the Symplectic Euler and RK4 integrators.

For a more complex test, we modeled our own Solar System. As seen in Figure 5, LomaVerse accurately captures the distinct orbital inclinations of the planets. The 2D projection in Figure 4 shows stable, near-Keplerian orbits. Over many simulated years, the full N-body interactions cause subtle perturbations, demonstrating a higher level of physical fidelity than a simple two-body calculation.

5.2 Case Study 2: Chaotic Three-Body Dynamics

To verify that LomaVerse can handle non-trivial dynamics, we simulated a classic chaotic three-body problem. Three stellar-mass objects were placed in close proximity with initial conditions designed to produce unstable orbits. As shown in the 2D orbital trails (Figure 6) and the full 3D rendering (Figure 7), the system rapidly deviates from predictability. Orbits become erratic, and bodies experience close encounters that lead to significant scattering. This demonstrates the simulator's ability to capture the sensitive dependence on initial conditions that is the hallmark of chaotic systems.

5.3 Case Study 3: Gravitational Deflection

Beyond stable and chaotic systems, a key application in astrodynamics is modeling gravitational assists. We simulated a scenario where a low-mass probe (a spacecraft) was directed towards a Jupiter-like gas giant. As shown in Figure 8, the probe's trajectory is clearly perturbed by the planet's gravitational field. The simulator correctly models this deflection, which is the fundamental mechanism behind gravity-assist maneuvers used to accelerate real-world spacecraft on interplanetary journeys. This validates the engine's handling of interactions between bodies with vastly different masses.

5.4 Case Study 4: Lagrange Point Stability

A more rigorous demonstration of the simulator's accuracy is its ability to model the subtle equilibrium at a Lagrange point. In any two-body system (e.g., Sun-Earth, Star-Jupiter), there exist five points where the gravitational forces of the two large bodies and the centrifugal force from rotation precisely balance, allowing a third, smaller object to maintain a fixed position relative to them.

We specifically tested the L4 point, which forms an equilateral triangle with the star and a large orbiting planet (our "Jupiter"). In the rotating reference frame of the two large bodies, the L4 and L5 points act as gravitational potential wells, making them linearly stable. To verify this, we created a scenario with a central star, a massive planet, and a small asteroid placed at the L4 point with the correct initial velocity to be co-orbital with the planet.

When running this simulation with the RK4 integrator for higher precision, we observed that the asteroid did not drift away or fall into the star or planet. Instead, it maintained its position, entering a stable "tadpole" orbit around the L4 libration point, perfectly mirroring the behavior of real-world Trojan asteroids in our own Solar System. This result confirms that our simulator correctly models the complex, balanced net forces in a multi-body system, providing a non-trivial validation of the underlying physics engine. The initial conditions for this scenario are available as a saved configuration file in the project's public code repository.

Simulation Locations

We encourage you to see these simulations in action rather than as static images, please head to our Github Repository to see more!

6 Future Work and Conclusion

The primary goal of this project, which was not achieved, was to use LomaVerse for spacecraft trajectory optimization. While we successfully built a robust, differentiable N-body simulator—a significant achievement in itself—the final optimization step remains as the most exciting avenue for future work.

6.1 Planned Trajectory Optimization with a Differentiable Agent

The core unimplemented feature is the introduction of a spacecraft, or "agent," whose trajectory is not merely simulated but actively optimized. The plan was to leverage Loma's reverse-mode AD to find a fuel-optimal path between two celestial bodies. This requires a more sophisticated model than the passive N-body simulation.

6.1.1 Modeling the Agent. The spacecraft would be integrated as an $(N+1)^{th}$ body with a controllable thrust vector $\mathbf{T}(t)$ and, critically, a time-varying mass $m_s(t)$ to account for fuel expenditure. The agent's equations of motion are thus augmented:

$$\dot{\mathbf{r}}_s = \mathbf{v}_s \tag{9}$$

$$\dot{\mathbf{v}}_s = \sum_{j \neq s}^{N} G \frac{m_j}{\|\mathbf{r}_j - \mathbf{r}_s\|^3_{\text{soft}}} (\mathbf{r}_j - \mathbf{r}_s) + \frac{\mathbf{T}(t)}{m_s(t)} \tag{10}$$

The mass decreases according to the Tsiolkovsky rocket equation, where the rate of change is proportional to the thrust magnitude: $\dot{m}_s = -\|\mathbf{T}(t)\|/(I_{sp} g_0)$, where I_{sp} is the specific

Figure 4: A 2D projection of the Solar System simulation. While useful for showing relative positions, it flattens the orbital inclinations.

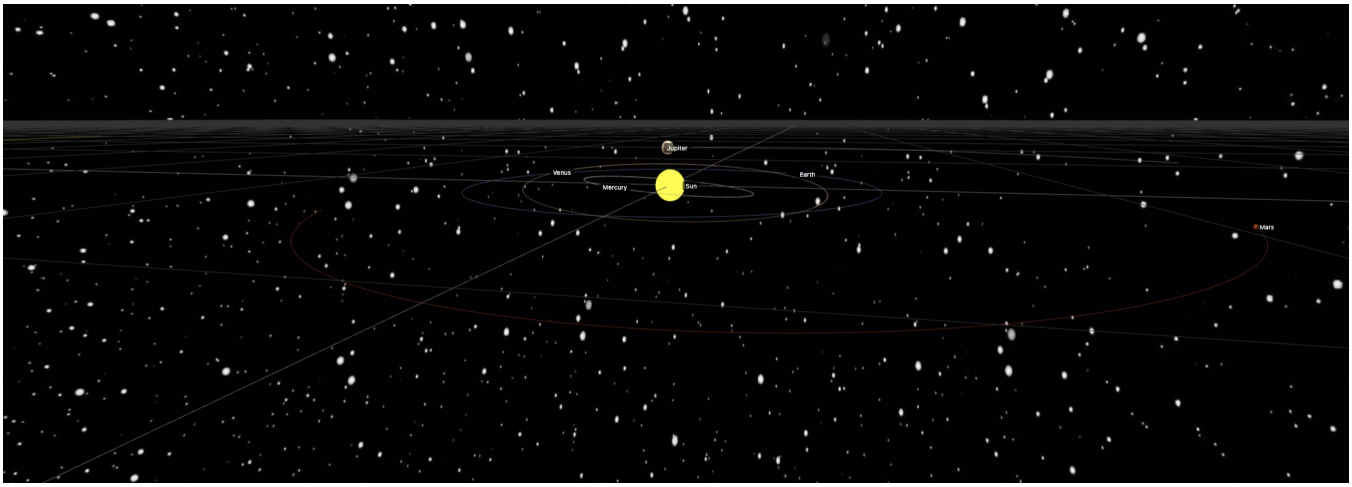

Figure 5: Full 3D visualization of the Solar System simulation, correctly showing the different orbital inclinations of the planets relative to the ecliptic.

impulse of the engine. The control parameters for our optimization problem would be the components of $\mathbf{T}(t)$, discretized over time segments.

6.1.2 Loss Function and Optimization. With the agent integrated, we can define a scalar loss function L that quantifies the mission's objectives. For a rendezvous mission, a suitable loss function would be a weighted sum penalizing the final

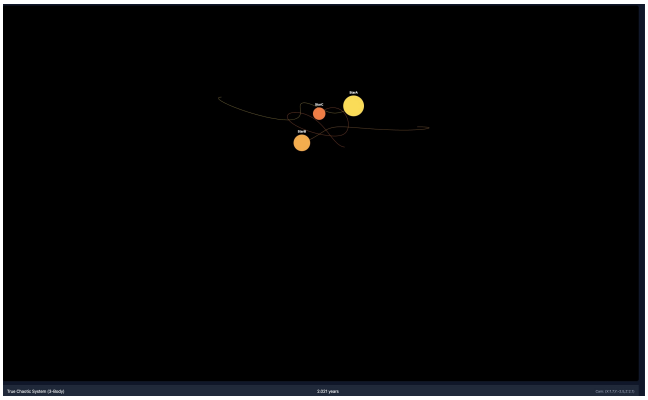

Figure 6: Chaotic orbital trails of a three-body system projected onto a 2D plane, demonstrating unpredictability.

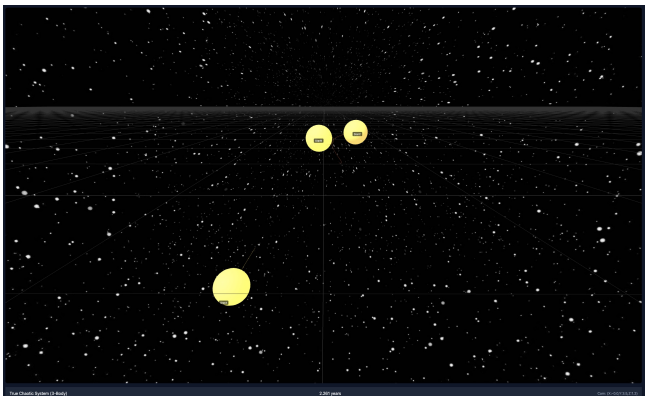

Figure 7: Three-dimensional rendering of the complex and unpredictable orbits in a chaotic three-body system.

position and velocity errors relative to a target, plus the total fuel consumption (proportional to the total impulse or ΔV):

$$L = w_1 \|\mathbf{r}_s(T_f) - \mathbf{r}_{\text{target}}(T_f)\|^2 + w_2 \|\mathbf{v}_s(T_f) - \mathbf{v}_{\text{target}}(T_f)\|^2 + w_3 \int_0^{T_f} \|\mathbf{T}(t)\| dt \tag{11}$$

The entire simulation, which propagates the full N-body system including the thrusting spacecraft and computes L, becomes a differentiable function of the thrust control parameters. By applying 'reverse_diff' to this function, we can efficiently compute the gradient $\nabla_{\mathbf{T}(t)} L$. This gradient would then be used in an optimization loop (e.g., gradient descent with momentum) to iteratively update the thrust profile, converging on a fuel-optimal trajectory. Successfully implementing this would be a powerful demonstration of differentiable programming for a complex, real-world problem in astrodynamics.

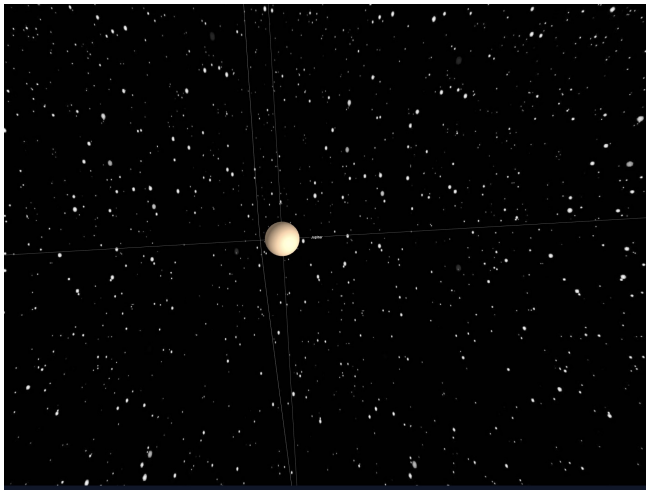

Figure 8: A spacecraft's trajectory being altered by Jupiter's gravity, demonstrating a gravitational deflection and possible assist. The trail line is faint but to the left side of Jupiter (akin to the New Horizons Spacecraft)

7 Challenges and Learnings

Developing LomaVerse presented several significant challenges that offered valuable insights into the iterative process of building differentiable physics simulators, from initial conception to a functional prototype.

7.1 The Iterative Path: From Concept to 3D Visualization

A primary learning from this project was the importance of an incremental and validated development process. The journey from a simple idea to a functional tool underscored that a rigorous understanding of the underlying physics and a clear problem definition are prerequisites for success. Our development proceeded in deliberate stages:

(1) **Core Compilation:** The first step was simply ensuring a basic Hamiltonian model could be correctly written in Loma and compiled correctly.
(2) **Basic Frontend Integration:** We then built a minimal Python host to call the compiled Loma code and visualize the output in 2D, validating that the data pipeline between the host and the physics engine was working.
(3) **Expansion to 3D:** With a working 2D model, we expanded the physics to three dimensions, updating the vector math in Loma and the visualization logic on the frontend.
(4) **User-Specified Inputs:** Finally, we built the scenario creator and the associated backend logic, allowing

users to define their own systems. This layered approach, where each step was validated before proceeding, was crucial for managing complexity and debugging effectively.

7.2 Performance, Load Balancing, and Compiler Speed

A persistent challenge was ensuring the web-based GUI remained responsive while the backend performed computationally intensive simulations. The differentiation and compilation of Loma code introduces a noticeable startup delay, a factor that became more pronounced on older hardware (specifically, an Intel i5 MacBook), leading to significant delays. This highlights that for LomaVerse to be "production-ready," optimizations to both our Loma code and potentially the compiler itself would be a necessary effort.

Furthermore, generating future states for the simulation is a synchronous, blocking operation on the server. Our initial approach of calculating large batches of frames led to significant UI lag. We mitigated this by implementing a chunking strategy: the server calculates a smaller number of frames per request, and the frontend asynchronously requests the next chunk before the current one is exhausted. This client-pull architecture creates a much smoother user experience but requires careful tuning of the chunk size to balance network overhead and server computation time.

7.3 Numerical Stability and Physical Conservation

In N-body simulations, small errors can compound over time. We encountered this directly with our Solar System model, where an initially imperceptible net momentum caused the entire system, including the Sun, to drift. This is physically inaccurate, as a closed system's center of mass should not accelerate. To solve this, we implemented a center-of-mass correction: before starting a simulation, the host code calculates the total momentum of all bodies and imparts an equal and opposite momentum to the most massive body (the central star). This ensures the net momentum of the system is zero, stabilizing the simulation and preventing drift. This experience highlighted the importance of conserving physical quantities in long-duration numerical simulations.

8 Conclusion

This project succeeded in its primary goal of creating a functional, 3D, N-body gravitational simulator, LomaVerse, using the Loma differentiable programming language. We successfully implemented and validated multiple integration schemes (Symplectic Euler, RK4), created a flexible system for defining and running diverse celestial scenarios, and built an interactive web-based frontend for visualization. Through this process, we gained practical experience with the challenges of numerical stability, performance optimization, and the nuances of working within a novel compiler framework.

Although the ultimate goal of trajectory optimization using reverse-mode AD was not reached, the resulting LomaVerse platform provides a robust and validated foundation for this future work. The challenges encountered have clarified the path forward, highlighting the need for compiler performance improvements and a carefully designed agent-based optimization model. Ultimately, LomaVerse stands as a successful proof-of-concept, demonstrating the significant potential of differentiable programming for solving complex problems in computational physics and astrodynamics.

A Loma Engine

```
 --- Struct Definitions ---
class Vec3:
    x: float
    y: float
    z: float

class BodyState:
    pos: Vec3
    mom: Vec3
    mass: float
    inv_mass: float

class SimConfig:
    G: float
    dt: float
    epsilon_sq: float
    num_bodies: int

# New struct to hold derivatives for RK4
class BodyDerivative:
    d_pos: Vec3 # Represents velocity (dr/dt)
    d_mom: Vec3 # Represents force (dp/dt)

# --- Hamiltonian Function (3D) ---
def n_body_hamiltonian(states: In[Array[BodyState, 20]],
                       config: In[SimConfig]) -> float:
    total_kinetic_energy: float = 0.0; total_potential_energy: float = 0.0; i: int = 0; j: int = 0
    dx: float; dy: float; dz: float; dist_sq: float; inv_dist_soft: float
    s_i_pos_x: float; s_i_pos_y: float; s_i_pos_z: float; s_j_pos_x: float; s_j_pos_y: float; s_j_pos_z:
        float
    s_i_mom_x: float; s_i_mom_y: float; s_i_mom_z: float; s_i_inv_mass: float; s_i_mass: float; s_j_mass:
        float
    i = 0
    while (i < config.num_bodies, max_iter := 20):
        s_i_mom_x = states[i].mom.x; s_i_mom_y = states[i].mom.y; s_i_mom_z = states[i].mom.z
        s_i_inv_mass = states[i].inv_mass
        total_kinetic_energy = total_kinetic_energy + (s_i_mom_x*s_i_mom_x + s_i_mom_y*s_i_mom_y +
            s_i_mom_z*s_i_mom_z) * s_i_inv_mass * 0.5
        i = i + 1
    i = 0
    while (i < config.num_bodies, max_iter := 20):
        s_i_pos_x = states[i].pos.x; s_i_pos_y = states[i].pos.y; s_i_pos_z = states[i].pos.z
        s_i_mass = states[i].mass
        j = i + 1
        while (j < config.num_bodies, max_iter := 20):
            s_j_pos_x = states[j].pos.x; s_j_pos_y = states[j].pos.y; s_j_pos_z = states[j].pos.z
            s_j_mass = states[j].mass
            dx = s_j_pos_x - s_i_pos_x; dy = s_j_pos_y - s_i_pos_y; dz = s_j_pos_z - s_i_pos_z
```

```
            dist_sq = dx*dx + dy*dy + dz*dz
            inv_dist_soft = 1.0 / sqrt(dist_sq + config.epsilon_sq)
            total_potential_energy = total_potential_energy - config.G * s_i_mass * s_j_mass *
                inv_dist_soft
            j = j + 1
        i = i + 1
    return total_kinetic_energy + total_potential_energy

d_n_body_hamiltonian = fwd_diff(n_body_hamiltonian)

def get_dH_dr_k_alpha(states_val: In[Array[BodyState, 20]], config_val: In[SimConfig], k: In[int],
     alpha: In[int]) -> float:
    d_states: Array[Diff[BodyState], 20]; d_config: Diff[SimConfig]; idx: int = 0
    while(idx < 20, max_iter := 20):
        if idx < config_val.num_bodies:
            d_states[idx].pos.x.val = states_val[idx].pos.x; d_states[idx].pos.y.val =
                states_val[idx].pos.y; d_states[idx].pos.z.val = states_val[idx].pos.z
            d_states[idx].mom.x.val = states_val[idx].mom.x; d_states[idx].mom.y.val =
                states_val[idx].mom.y; d_states[idx].mom.z.val = states_val[idx].mom.z
            d_states[idx].mass.val = states_val[idx].mass; d_states[idx].inv_mass.val =
                states_val[idx].inv_mass
        else:
            d_states[idx].pos.x.val = 0.0; d_states[idx].pos.y.val = 0.0; d_states[idx].pos.z.val = 0.0
            d_states[idx].mom.x.val = 0.0; d_states[idx].mom.y.val = 0.0; d_states[idx].mom.z.val = 0.0
            d_states[idx].mass.val = 1.0; d_states[idx].inv_mass.val = 1.0
        d_states[idx].pos.x.dval = 0.0; d_states[idx].pos.y.dval = 0.0; d_states[idx].pos.z.dval = 0.0
        d_states[idx].mom.x.dval = 0.0; d_states[idx].mom.y.dval = 0.0; d_states[idx].mom.z.dval = 0.0
        d_states[idx].mass.dval = 0.0; d_states[idx].inv_mass.dval = 0.0
        idx = idx + 1
    d_config.G.val = config_val.G; d_config.G.dval = 0.0; d_config.dt.val = config_val.dt;
         d_config.dt.dval = 0.0; d_config.epsilon_sq.val = config_val.epsilon_sq;
         d_config.epsilon_sq.dval = 0.0; d_config.num_bodies = config_val.num_bodies
    if k < config_val.num_bodies:
        if alpha == 0: d_states[k].pos.x.dval = 1.0
        elif alpha == 1: d_states[k].pos.y.dval = 1.0
        else: d_states[k].pos.z.dval = 1.0
    return d_n_body_hamiltonian(d_states, d_config).dval

def get_dH_dp_k_alpha(states_val: In[Array[BodyState, 20]], config_val: In[SimConfig], k: In[int],
     alpha: In[int]) -> float:
    d_states: Array[Diff[BodyState], 20]; d_config: Diff[SimConfig]; idx: int = 0
    while(idx < 20, max_iter := 20):
        if idx < config_val.num_bodies:
            d_states[idx].pos.x.val = states_val[idx].pos.x; d_states[idx].pos.y.val =
                states_val[idx].pos.y; d_states[idx].pos.z.val = states_val[idx].pos.z
            d_states[idx].mom.x.val = states_val[idx].mom.x; d_states[idx].mom.y.val =
                states_val[idx].mom.y; d_states[idx].mom.z.val = states_val[idx].mom.z
            d_states[idx].mass.val = states_val[idx].mass; d_states[idx].inv_mass.val =
                states_val[idx].inv_mass
        else:
```

```
            d_states[idx].pos.x.val = 0.0; d_states[idx].pos.y.val = 0.0; d_states[idx].pos.z.val = 0.0
            d_states[idx].mom.x.val = 0.0; d_states[idx].mom.y.val = 0.0; d_states[idx].mom.z.val = 0.0
            d_states[idx].mass.val = 1.0; d_states[idx].inv_mass.val = 1.0
        d_states[idx].pos.x.dval = 0.0; d_states[idx].pos.y.dval = 0.0; d_states[idx].pos.z.dval = 0.0
        d_states[idx].mom.x.dval = 0.0; d_states[idx].mom.y.dval = 0.0; d_states[idx].mom.z.dval = 0.0
        d_states[idx].mass.dval = 0.0; d_states[idx].inv_mass.dval = 0.0
        idx = idx + 1
    d_config.G.val = config_val.G; d_config.G.dval = 0.0; d_config.dt.val = config_val.dt;
         d_config.dt.dval = 0.0; d_config.epsilon_sq.val = config_val.epsilon_sq;
         d_config.epsilon_sq.dval = 0.0; d_config.num_bodies = config_val.num_bodies
    if k < config_val.num_bodies:
        if alpha == 0: d_states[k].mom.x.dval = 1.0
        elif alpha == 1: d_states[k].mom.y.dval = 1.0
        else: d_states[k].mom.z.dval = 1.0
    return d_n_body_hamiltonian(d_states, d_config).dval
# Symplectic Euler Integrator
def time_step_system(current_states: In[Array[BodyState, 20]], config: In[SimConfig], next_states:
     Out[Array[BodyState, 20]]):
    k: int = 0; dH_dr_kx: float; dH_dr_ky: float; dH_dr_kz: float; dH_dp_kx: float; dH_dp_ky: float;
         dH_dp_kz: float
    k = 0
    while(k < config.num_bodies, max_iter := 20):
        dH_dr_kx = get_dH_dr_k_alpha(current_states, config, k, 0); dH_dr_ky =
            get_dH_dr_k_alpha(current_states, config, k, 1); dH_dr_kz = get_dH_dr_k_alpha(current_states,
            config, k, 2)
        next_states[k].mom.x = current_states[k].mom.x - config.dt * dH_dr_kx; next_states[k].mom.y =
            current_states[k].mom.y - config.dt * dH_dr_ky; next_states[k].mom.z =
            current_states[k].mom.z - config.dt * dH_dr_kz
        next_states[k].pos.x = current_states[k].pos.x; next_states[k].pos.y = current_states[k].pos.y;
            next_states[k].pos.z = current_states[k].pos.z
        next_states[k].mass = current_states[k].mass; next_states[k].inv_mass = current_states[k].inv_mass
        k = k + 1
    k = 0
    while(k < config.num_bodies, max_iter := 20):
        dH_dp_kx = get_dH_dp_k_alpha(next_states, config, k, 0); dH_dp_ky =
            get_dH_dp_k_alpha(next_states, config, k, 1); dH_dp_kz = get_dH_dp_k_alpha(next_states,
            config, k, 2)
        next_states[k].pos.x = next_states[k].pos.x + config.dt * dH_dp_kx; next_states[k].pos.y =
            next_states[k].pos.y + config.dt * dH_dp_ky; next_states[k].pos.z = next_states[k].pos.z +
            config.dt * dH_dp_kz
        k = k + 1

# --- RK4 Integrator ---
def time_step_system_rk4(current_states: In[Array[BodyState, 20]],
                     config: In[SimConfig],
                     next_states: Out[Array[BodyState, 20]],
                     # Scratch space arrays passed in to avoid compiler bug
                     k1: Out[Array[BodyDerivative, 20]],
                     k2: Out[Array[BodyDerivative, 20]],
                     k3: Out[Array[BodyDerivative, 20]],
```

```
                    k4: Out[Array[BodyDerivative, 20]],
                    intermediate_states: Out[Array[BodyState, 20]]):
    k: int = 0; dt: float = config.dt; dt_half: float = dt * 0.5; dt_sixth: float = dt / 6.0

    # k1 = f(y_n)
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        k1[k].d_pos.x = get_dH_dp_k_alpha(current_states, config, k, 0); k1[k].d_pos.y =
            get_dH_dp_k_alpha(current_states, config, k, 1); k1[k].d_pos.z =
            get_dH_dp_k_alpha(current_states, config, k, 2)
        k1[k].d_mom.x = -get_dH_dr_k_alpha(current_states, config, k, 0); k1[k].d_mom.y =
            -get_dH_dr_k_alpha(current_states, config, k, 1); k1[k].d_mom.z =
            -get_dH_dr_k_alpha(current_states, config, k, 2)
        k = k + 1

    # k2 = f(y_n + dt*k1/2)
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        intermediate_states[k].pos.x = current_states[k].pos.x + k1[k].d_pos.x * dt_half;
            intermediate_states[k].pos.y = current_states[k].pos.y + k1[k].d_pos.y * dt_half;
            intermediate_states[k].pos.z = current_states[k].pos.z + k1[k].d_pos.z * dt_half
        intermediate_states[k].mom.x = current_states[k].mom.x + k1[k].d_mom.x * dt_half;
            intermediate_states[k].mom.y = current_states[k].mom.y + k1[k].d_mom.y * dt_half;
            intermediate_states[k].mom.z = current_states[k].mom.z + k1[k].d_mom.z * dt_half
        intermediate_states[k].mass = current_states[k].mass; intermediate_states[k].inv_mass =
            current_states[k].inv_mass
        k = k + 1
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        k2[k].d_pos.x = get_dH_dp_k_alpha(intermediate_states, config, k, 0); k2[k].d_pos.y =
            get_dH_dp_k_alpha(intermediate_states, config, k, 1); k2[k].d_pos.z =
            get_dH_dp_k_alpha(intermediate_states, config, k, 2)
        k2[k].d_mom.x = -get_dH_dr_k_alpha(intermediate_states, config, k, 0); k2[k].d_mom.y =
            -get_dH_dr_k_alpha(intermediate_states, config, k, 1); k2[k].d_mom.z =
            -get_dH_dr_k_alpha(intermediate_states, config, k, 2)
        k = k + 1

    # k3 = f(y_n + dt*k2/2)
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        intermediate_states[k].pos.x = current_states[k].pos.x + k2[k].d_pos.x * dt_half;
            intermediate_states[k].pos.y = current_states[k].pos.y + k2[k].d_pos.y * dt_half;
            intermediate_states[k].pos.z = current_states[k].pos.z + k2[k].d_pos.z * dt_half
        intermediate_states[k].mom.x = current_states[k].mom.x + k2[k].d_mom.x * dt_half;
            intermediate_states[k].mom.y = current_states[k].mom.y + k2[k].d_mom.y * dt_half;
            intermediate_states[k].mom.z = current_states[k].mom.z + k2[k].d_mom.z * dt_half
        k = k + 1
    k = 0
    while (k < config.num_bodies, max_iter := 20):
```

```
        k3[k].d_pos.x = get_dH_dp_k_alpha(intermediate_states, config, k, 0); k3[k].d_pos.y =
            get_dH_dp_k_alpha(intermediate_states, config, k, 1); k3[k].d_pos.z =
            get_dH_dp_k_alpha(intermediate_states, config, k, 2)
        k3[k].d_mom.x = -get_dH_dr_k_alpha(intermediate_states, config, k, 0); k3[k].d_mom.y =
            -get_dH_dr_k_alpha(intermediate_states, config, k, 1); k3[k].d_mom.z =
            -get_dH_dr_k_alpha(intermediate_states, config, k, 2)
        k = k + 1

    # k4 = f(y_n + dt*k3)
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        intermediate_states[k].pos.x = current_states[k].pos.x + k3[k].d_pos.x * dt;
            intermediate_states[k].pos.y = current_states[k].pos.y + k3[k].d_pos.y * dt;
            intermediate_states[k].pos.z = current_states[k].pos.z + k3[k].d_pos.z * dt
        intermediate_states[k].mom.x = current_states[k].mom.x + k3[k].d_mom.x * dt;
            intermediate_states[k].mom.y = current_states[k].mom.y + k3[k].d_mom.y * dt;
            intermediate_states[k].mom.z = current_states[k].mom.z + k3[k].d_mom.z * dt
        k = k + 1
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        k4[k].d_pos.x = get_dH_dp_k_alpha(intermediate_states, config, k, 0); k4[k].d_pos.y =
            get_dH_dp_k_alpha(intermediate_states, config, k, 1); k4[k].d_pos.z =
            get_dH_dp_k_alpha(intermediate_states, config, k, 2)
        k4[k].d_mom.x = -get_dH_dr_k_alpha(intermediate_states, config, k, 0); k4[k].d_mom.y =
            -get_dH_dr_k_alpha(intermediate_states, config, k, 1); k4[k].d_mom.z =
            -get_dH_dr_k_alpha(intermediate_states, config, k, 2)
        k = k + 1

    # y_{n+1} = y_n + dt/6 * (k1 + 2*k2 + 2*k3 + k4)
    k = 0
    while (k < config.num_bodies, max_iter := 20):
        next_states[k].pos.x = current_states[k].pos.x + (k1[k].d_pos.x + 2.0*k2[k].d_pos.x +
            2.0*k3[k].d_pos.x + k4[k].d_pos.x) * dt_sixth
        next_states[k].pos.y = current_states[k].pos.y + (k1[k].d_pos.y + 2.0*k2[k].d_pos.y +
            2.0*k3[k].d_pos.y + k4[k].d_pos.y) * dt_sixth
        next_states[k].pos.z = current_states[k].pos.z + (k1[k].d_pos.z + 2.0*k2[k].d_pos.z +
            2.0*k3[k].d_pos.z + k4[k].d_pos.z) * dt_sixth
        next_states[k].mom.x = current_states[k].mom.x + (k1[k].d_mom.x + 2.0*k2[k].d_mom.x +
            2.0*k3[k].d_mom.x + k4[k].d_mom.x) * dt_sixth
        next_states[k].mom.y = current_states[k].mom.y + (k1[k].d_mom.y + 2.0*k2[k].d_mom.y +
            2.0*k3[k].d_mom.y + k4[k].d_mom.y) * dt_sixth
        next_states[k].mom.z = current_states[k].mom.z + (k1[k].d_mom.z + 2.0*k2[k].d_mom.z +
            2.0*k3[k].d_mom.z + k4[k].d_mom.z) * dt_sixth
        next_states[k].mass = current_states[k].mass; next_states[k].inv_mass = current_states[k].inv_mass
        k = k + 1
```