PHYSICS-INFORMED DEEP Q-LEARNING FOR N-BODY SIMULATIONS [*]

LUCA WEISHAUPT[§ †] AND ANDREW ZHANG[§ †]

Abstract. The n-body problem in astronomy involves predicting the motion of celestial bodies under gravitational attraction. Simulating systems with more than two bodies quickly becomes computationally expensive. Thus, various optimization schemes have been developed, including symplectic integrators, tree-based methods, fast multipole solvers, and adaptive timestepping. We propose a reinforcement learning algorithm using deep Q-learning for optimal step size selection in adaptive timestepping, where physical constraints are explicitly encoded in the training paradigm. Although our results serve only as a proof of concept, they demonstrate the feasibility of physics-informed reinforcement learning for n-body systems.

Key words. n-body simulations, reinforcement learning, numerical methods

MSC codes. 68Q25, 68T05, 68U20, 68W25, 70F10

1. Introduction.

The n-body problem is a fundamental and challenging problem in physics with applications in astronomy, molecular dynamic, and beyond. It involves predicting the motion of n celestial bodies under the influence of mutual forces from potential fields. It's most widespread applications are in understanding the dynamics of planetary systems, galaxies, and other astronomical bodies, as well as molecular dynamics [3, 9, 4, 13]. Due to the inherent complexity of the n-body problem, for $n > 2$, it is generally impossible to obtain a closed-form solution. Instead, researchers rely on computationally demanding numerical simulations [2, 8].

1.1. N-body ODEs.

It is helpful to first consider the simple case of $n = 2$, for which a closed-form solution can be obtained. Given two celestial bodies with masses m_1 and m_2, separated by a distance r, the gravitational force acting between them is described by Newton's law of universal gravitation:

$$F = \frac{Gm_1m_2}{r^2}$$

where G is the gravitational constant. To find the motion of the two celestrial bodies, we can apply Newton's second law of motion, $F = ma$, to each body:

$$F_1 = m_1\frac{\mathrm{d}^2\vec{r_1}}{\mathrm{d}t^2} = -Gm_1m_2\frac{\vec{r}_{1,2}}{r_{1,2}^3}$$

$$F_2 = m_2\frac{\mathrm{d}^2\vec{r_2}}{\mathrm{d}t^2} = Gm_1m_2\frac{\vec{r}_{1,2}}{r_{1,2}^3}$$

where $\vec{r_1}$ and $\vec{r_2}$ are the position vectors of the two bodies, $\vec{r}_{A,B} = \vec{r}_B - \vec{r}_A$, and F_i is the total force acting on body i. By solving these coupled second-order ordinary differential equations, we can obtain the closed-form solution for the two-body problem in terms of elliptical orbits described by the relative position $\vec{r} = \vec{r}_{1,2}$ and reduced mass $\mu = \frac{m_1m_2}{m_1+m_2}$. The gravitational two-body problem is also known as the Kepler problem. An example of the solution to such a system is shown in Figure 1. However, for systems

[§] Authors contributed equally.

[*] Submitted to the course staff of 18.337 on May 16, 2023.

Funding: This work was funded by the MIT Banana Lounge.

[†] Massachusetts Institute of Technology, Cambridge, MA (lucaweis@mit.edu, andrewzh@mit.edu).

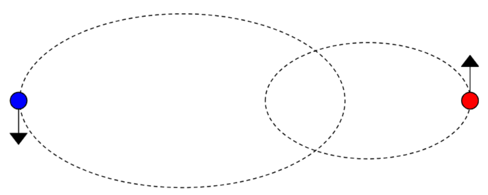

Fig. 1: An example of a stable solution to a 2 body problem. The colored dots represent the planets while the dotted lines represent their trajectories. A closed form solution can be calculated for this problem.

with three or more bodies, no general closed-form solution exists due to the increased complexity and nonlinearity of the problem. For an n-body system, the motion of each celestial body is governed by a set of second-order ordinary differential equations derived from Newton's law of gravitation and Newton's second law of motion:

$$F_i = m_i \frac{\mathrm{d}^2 \vec{r_i}}{\mathrm{d}t^2} = \sum_{j=1, j \neq i}^{n} G m_i m_j \frac{\vec{r}_{i,j}}{r_{i,j}^3}$$

for $i = 1, 2, \ldots, n$. These equations describe the gravitational interactions between all pairs of celestial bodies in the 3-dimensional system, resulting in a highly coupled and nonlinear set of $6n$ ordinary differential equations. Due to the intricate nature of these equations, finding a closed-form solution becomes infeasible as n increases, and researchers resort to numerical simulations to approximate the motion of celestial bodies in such systems.

1.2. N-body simulations. Existing methods for numerically solving the n-body problem include direct summation, tree code methods, fast multipole methods, and symplectic integrators. Direct n-body summation is a conceptually simple but computationally expensive approach involving the calculation of gravitational forces between all pairs of particles in a system at each timestep [3]. Tree code methods, such as the Barnes-Hut algorithm, group particles based on proximity and employ a hierarchical tree structure to reduce the time complexity to $\mathcal{O}(n \log n)$ [2]. The fast multipole method leverages multipole expansions and the Taylor series to approximate gravitational forces between distant particles, making it suitable for large systems with high symmetry [8]. On the other hand, symplectic integrators maintain energy conservation and momentum by preserving the symplectic structure of Hamiltonian systems, making them ideal for long-term planetary system simulations [11].

Adaptive timestepping can significantly improve the efficiency of computational methods involving numerical integration by balancing the tradeoff between step size and estimated error [6]. The Dormand-Prince (RKDP) method, a classic integrator which calculates 4th-order and 5th-order Runge-Kutta approximations, can be used to estimate the error at the next timestep by taking the difference between these two solutions. An adaptive timestepping algorithm can be produced as follows: If this error is lower than a given tolerance, the step size is increased for the next timestep. If the tolerance is exceeded, the step is rejected and the algorithm estimates the error with a smaller step [7]. This is implemented as `DP5` in Julia[a], `RK5` in SciPy[b], and `ode45` in MATLAB[c]. More sophisticated methods employ variable step and variable order integration to further enhance efficiency [12]. Recently, reinforcement learning (RL) paradigms have been proposed for adaptive timestepping in numerical integration [6], although they have not yet been applied to n-body simulations.

1.3. Related work. Several n-body simulators have been implemented in Julia, such as `NBodySimulator.jl`, `AstroNBodySim.jl`, and `Molly.jl`, with varying degrees of optimization and specialization. `NBodySimulator.jl`[d] provides a basic infrastructure for simulating n-particle systems with customizable potential fields, while `AstroNBodySim.jl`[e] focuses on hybrid parallelism, utilizing multi-threading, distributed parallelism, and GPU acceleration for performance enhancement. `Molly.jl`[f] is a specialized n-body simulator designed for modeling molecular dynamics. These packages have implemented a variety of optimizations including Runge-Kutta adaptive timestepping, but do not currently provide a mechanism for RL-based adaptive timestepping. Though adaptive timestepping using Runge-Kutta pairs is straightforward, it can be inefficient for chaotic systems [6]. To that end, RL is a promising approach as it has been successful in many situations involving complex sequential decision making [1].

Adaptive timestepping has not received much attention from the field of RL, perhaps due to the difficulty in translating a continuous range for the timestep interval $\mathrm{d}t$ into a discrete action space $\mathcal{A}$ required for the RL training process. To the best of our knowledge, there has only been one effort to apply RL to adaptive timestepping for numerical integration and ODE solvers [6]. Their approach, implemented in Python, is briefly described as follows: A discrete set of n possible timestep intervals $\{h_1, h_2, \ldots, h_n\}$ are chosen *a priori*, forming an action space $\mathcal{A}$. An agent parameterized by a neural network $\mathcal{F}$ indirectly observes the state $s_i \in \mathcal{S}$ of the dynamical system E at time t_i via an *evaluation trace* e_i composed of the previous step size and the intermediate function evaluations given by a standard integrator such as RKDP[g]. The agent selects an action $a_i \in \mathcal{A}$ using a policy $\pi : \mathcal{S} \to P(\mathcal{A})$. The system is stepped with step size a_i to arrive at state s_{i+1} (with trace e_{i+1}) at time t_{i+1}, while simultaneously computing a *ground truth* s'_{i+1} using a strict (low-tolerance) adaptive integrator from t_i to t_{i+1}. A reward r_i is computed based on the distance between s_{i+1} and s'_{i+1} with a function $f : (s, s') \to r$.

Though the proposed RL scheme was shown to work well for certain numerical quadrature tasks and numerical ODE solver tasks, including the double-pendulum and Lorenz chaotic systems, it has not been adapted specifically for solving n-body problems. We make the following improvements upon the scheme in [6]: (1) We explicitly encode the masses of the bodies into the evaluation trace observed by the agent. (2) We introduce a hybrid reward composed of both position-velocity error and

[a]https://docs.sciml.ai/DiffEqDocs/stable/solvers/ode_solve/

[b]https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.RK45.html

[c]https://www.mathworks.com/videos/solving-odes-in-matlab-6-ode45-117537.html

[d]https://docs.sciml.ai/NBodySimulator/stable/

[e]https://juliaastrosim.github.io/AstroNbodySim.jl/dev/

[f]https://juliamolsim.github.io/Molly.jl/stable/docs/

[g]For a system with k variables using an integrator with p function evaluations per step, the evaluation trace is a vector of size $k \times p + 1$ containing the results of each function evaluation for each variable, and the previous step size a_{i-1}. Although it is more intuitive for the agent to simply observe the $(k+1)$-dimensional vector composed of s_i and a_i, in practice the agent benefits from the additional context provided by the full evaluation trace. Note that for integrators involving a single function evaluation per step, such as Euler's method, the two schemes are identical.

energy error, which aims to preserve the Hamiltonian of the system. (3) We enable the agent to look back across multiple consecutive previous timesteps in order to obtain more context. (4) We implement our algorithm in Julia, building the environment from scratch and using `Flux.jl`[h] for training the agent's neural network.

2. Methods.

Our training paradigm is described in Algorithm 2.1 and Figure 2. Below we first explain reinforcement learning in detail, and then describe each aspect of our algorithm including the environment, reward function, and agent.

Algorithm 2.1 Adaptive timestepping RL training scheme

```
1: Initialize environment $E$ with action space $\mathcal{A}$, maximum time $t_{max}$, position-
   velocity reward function $f_s$, and energy reward function $f_e$
2: Initialize agent $\mathcal{F}$ with policy $\pi$
3: for each episode in NUM_EPISODES do
4:   Initialize new ODE system $\mathcal{K}$ in $E$ with state $s_0$ at time $t_0$
5:   while $t_i < t_{max}$ do
6:     Generate Q-values from observation $e_i$ using agent $\mathcal{F}(e_i) = Q_{\mathcal{A},i}$
7:     Select action $a_i \in \mathcal{A}$ with probability distribution $P(\mathcal{A}) = \pi(Q_{\mathcal{A},i})$
8:     Step $\mathcal{K}$ from $t_i$ to $t_{i+1} = t_i + a_i$ with 5th-order Runge-Kutta integrator to
       get predicted state $s_{i+1}$ and evaluation trace $e_{i+1}$
9:     Integrate $\mathcal{K}$ from $t_i$ to $t_{i+1} = t_i + a_i$ with low-tolerance adaptive integrator
       to get ground truth state $s'_{i+1}$
10:    Compute Hamiltonians $H(s_i)$ and $H(s_{i+1})$
11:    Compute position-velocity reward $r_s = f_s(s_{i+1}, s'_{i+1}, a_i)$
12:    Compute energy reward $r_e = f_e(H(s_i), H(s_{i+1}), a_i)$
13:    Compute hybrid reward $r_i = \alpha r_s + (1 - \alpha) r_e$ with weighting factor $\alpha$
14:    Store $(e_i, a_i, r_i, e_{i+1})$ in agent's memory
15:    if global step is multiple of UPDATE_FREQ then
16:      Compute target Q-values using Bellman equation with discount factor $\gamma$:
         $Q'_{\mathcal{A},i} = r_i + \gamma \mathcal{F}(s_{i+1})$
17:      Update $\mathcal{F}$ using loss $\mathcal{L}(Q_{\mathcal{A},i}, Q'_{\mathcal{A},i})$
18:    end if
19:    Update observation and global step
20:  end while
21: end for
```

2.1. Reinforcement learning.

Reinforcement learning is a machine learning technique for teaching an agent (usually parameterized by a neural network) to make optimal decisions in a given environment. The agent learns by observing the environment, performing an action, and receiving an appropriate reward. Over time, the agent adjusts its *behavior policy* to maximize its total rewards over time. Q-learning is one of many RL algorithms, in which the agent learns to predict the Q-value (quality value, corresponding to the cumulative reward under an optimal policy) of every possible action under every possible observation in a Markov Decision Process (MDP) environment. Though the $\mathcal{Q}$ function can be simply given by a $e \times |\mathcal{A}|$ table of Q-scores for e possible observations and $|\mathcal{A}|$ possible actions, in more complex environments it becomes infeasible to store the Q-table explicitly.

[h]https://fluxml.ai/Flux.jl/stable/

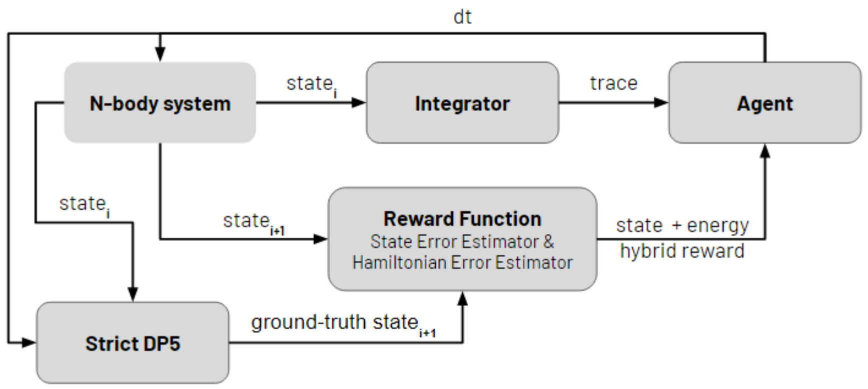

Fig. 2: Schematic showing the RL training paradigm used for the n-body system. An agent chooses timesteps (dt) and receives a hybrid reward composed of a state (i.e. position-velocity) term and a Hamiltonian (i.e. energy) term. The position-velocity reward is determined based on the difference between the next state predicted by stepping the system using the agent's chosen timestep and the ground truth calculated using a strict (low-tolerance) `DP5()` integrator. The energy reward is determined based on the difference in total system energy (the Hamiltonian) between the last timestep and the current timestep.

Deep Q-learning combines the Q-learning algorithm with a neural network as the agent, where the input layer of the neural network receives the observation from the environment and the output layer contains $|\mathcal{A}|$ neurons representing the Q-scores of $|\mathcal{A}|$ possible actions. The agent can then choose an action based on an *epsilon-greedy policy*, in which the agent selects a non-optimal action with probability ϵ and otherwise selects the optimal (highest $\mathcal{Q}$) action. This policy aims to balance exploration of the environment with exploitation of current beliefs about the quality of each action. The training process involves iteratively updating the network's parameters to minimize the difference between the predicted Q-values and the target Q-values, which are approximated using the Bellman equation:

$$\mathcal{Q}(e_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} [\mathcal{Q}(e_{t+1}, a_{t+1})]$$

In the above equation, $\mathcal{Q}(s_t, a_t)$ is the Q-value for action a_t at observation e_t, r is the immediate reward received after taking action a_t at observation e_t, γ is the discount factor that balances the importance of immediate and future rewards, and $\max_{a_{t+1}} [\mathcal{Q}(e_{t+1}, a_{t+1})]$ is the predicted Q-value at the next step, assuming an optimal future policy. The Bellman equation essentially states that the optimal Q-value for a state-action pair is equal to the immediate reward obtained plus the discounted maximum cumulative reward achievable from the next step onwards. By iteratively applying the Bellman equation and updating the Q-values based on observed rewards, the agent's predicted Q-values eventually converge to the true Q-values [6].

Q-learning agents are often trained using *experience replay*, in which the agent's experiences (i.e. their observation, action, and reward at each time step) are stored in a *replay buffer*. In each training step, a batch of experiences is randomly sampled from the buffer, which provides a moving window over past experiences (usually spanning many timesteps across multiple episodes). Experience replay allows the agent to learn from previous iterations of the environment instead of only the most recent iteration, which could lead to overfitting [10].

2.2. Environment. In order to learn the patterns for a particular class of problems, a separate agent must be trained for each family of ODEs. For each training episode, the agent steps through an ODE system which uses the same set of equations but different initial conditions. The environment encapsulates the ODE systems used to train the agent, and also defines the action space, maximum duration of each simulation, and the reward function. On every step, the environment computes the next state of the system twice: first using the agent's chosen step size and a standard 4th or 5th order Runge-Kutta integrator, and again using a low-tolerance adaptive integrator. The position-velocity error is given by the Euclidean norm of the difference between these two states. The energy error is given by the difference between the Hamiltonian of the current state and the Hamiltonian of the agent-predicted next state. The environment returns the position-velocity error, the energy-error, the aggregated reward, and the evaluation trace to the agent.

2.3. ODE system. We first reproduce the results of [6] for Lorenz systems before implementing our custom algorithm on n-body systems. Accordingly, we define two `System` types: `LorenzSystem` and `NBodySystem`. Each type has a `system_step!()` method which calculates the next state using either a 4th-order (RK4) or 5th-order (RKDP) integration scheme, as well as a `system_gt!()` method which computes the ground truth estimate of the next state using the `DP5()` solver[i] in `DifferentialEquations.jl` with `reltol=1e-8`, the tolerance used by the `DP5()` solver. Furthermore, the `NBodySystem` type also has a `get_energy()` method for estimating the Hamiltonian (total energy) of the system. For `LorenzSystem`, the state is a 3-dimensional vector giving the (x, y, z) position of the system. For `NBodySystem`, the state is a $6n$-dimensional vector giving the position and velocity of each of n bodies. For both systems, the evaluation trace is a vector of size $4s + n + 1$ or $5s + n + 1$ depending on whether RK4 or RKDP is used, respectively, where $s = 6n$ is the size of the state vector. The $(+n)$ comes from appending the masses of each body and the $(+1)$ comes from appending the previous timestep.

2.4. Physics-informed reward function. The reward $r_i = \alpha r_s + (1-\alpha) r_e$ is a weighted average of the position-velocity reward r_s and the energy reward r_e. They are given by the following set of equations:

$$\begin{aligned} r_s &= f_s(s_{i+1}, s'_{i+1}, a_i) \\ r_e &= f_e(H(s_i), H(s_{i+1})) \end{aligned}$$

where f_s and f_e are the position-velocity and energy reward functions, respectively, s_i is the state and a_i is the action at time t_i, and H is the Hamiltonian estimator[j]. f_s and f_e are parameterized by predefined tolerances `STATE_TOL` and `ENERGY_TOL`, respectively. r_s incentivizes the agent to minimize the difference d_s between the predicted state vector and ground truth state vector, while r_e incentivizes the agent to minimize the difference d_e in total energy between the predicted state vector and the previous state vector. We adopt the reward scaling scheme in [6] for both error

[i]https://docs.sciml.ai/DiffEqDocs/stable/solvers/ode_solve/

[j]Note that the calculation of the reward uses the state vectors rather than evaluation traces.

terms:

$$r_s = \begin{cases} \log_{10}\left(\frac{\texttt{STATE\_TOL}}{d_s}\right), & d_s > \texttt{STATE\_TOL} \\ c_1 \log\left(c_2 \cdot a_i\right), & d_s < \texttt{STATE\_TOL} \end{cases}$$

$$r_e = \begin{cases} \log_{10}\left(\frac{\texttt{ENERGY\_TOL}}{d_e}\right), & d_e > \texttt{ENERGY\_TOL} \\ c_3 \log\left(c_4 \cdot a_i\right), & d_e < \texttt{ENERGY\_TOL} \end{cases}$$

where c_1, c_2, c_3, c_4 are scaling factors. The behavior of these reward functions is such that (1) the reward is negative when the error exceeds the tolerance and decreases monotonically with the magnitude of the error and (2) the reward is positive when the error is below the tolerance and increases monotonically with the step size. The scaling factors are empirically chosen (described in subsection 3.1 and subsection 3.2).

2.5. Agent. The agent is a standard fully-connected neural network comprised of an input layer with dimension $e \times l$ (where e is the size of the evaluation trace and l is the number of consecutive steps observed by the agent), 5 hidden layers with dimension 64, and an output layer with dimension equal to the size of $\mathcal{A}$. Each layer uses a rectified linear unit (ReLU) activation. The output of the last layer is a vector of predicted Q-values corresponding to the possible actions. The agent then chooses the next timestep with an epsilon-greedy policy as described in subsection 2.1, where we restrict the non-optimal choices to the two step sizes[k] with index $i-1$ or $i+1$ (where i is the index of the optimal action). The agent is trained using `Flux.jl` with the `Flux.Adam()` optimizer[l] and a learning rate of `1e-4`. To speed up training, one backwards pass is taken every `UPDATE_FREQ` steps. In each backwards pass, a batch of b examples is randomly chosen from a replay buffer storing the past M consecutive timesteps (which may span multiple episodes). A lookback mechanism is implemented in which each example in the batch is extended backwards to create a sequence of l consecutive timesteps. If a chosen experience is fewer than l timesteps from the beginning of an episode, the sequence is start-padded with duplicates of the beginning timestep until a sequence of size l is obtained. The evaluation trace is extracted for each step in this experience sequence and concatenated, and all samples in the batch are stacked to form an array of b rows by $e \times l$ columns. In the results presented below, we use the following hyperparameters: $M = 10000$, $b = 32$, $l = 5$.

3. Results.

3.1. Lorenz system. We first reproduce the results from [6] for the Lorenz system, despite using different hyperparameters and a different architecture for the agent. The Lorenz system is described with the following ODEs:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x)$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z$$

We use the standard parameters $\sigma = 10, \beta = \frac{8}{3}, \rho = 28$ and simulate the system from $t_0 = 0$ to $t_{max} = 10.0$ for 20 training episodes. Each training episode is initialized

[k]Each with conditional probability $\frac{1}{2}$.
[l]https://fluxml.ai/Flux.jl/v0.10/training/optimisers/

with random starting conditions $x \in [-10, 10]$, $y \in [-10, 10]$, $z \in [15, 35]$, sampled in increments of 0.1. This sample space contains points both inside and outside the attractor [6]. We give the agent the same log-scale action space as [6]:

$$\mathcal{A} = \{0.02, 0.022, 0.025, 0.029, 0.033, 0.039, 0.045, 0.052, 0.060, 0.070\}$$

Because the energy reward does not apply in this case, we have $r_i = r_s$. The scaling factors c_1, c_2 are calculated such that

$$r_s = \begin{cases} 0.1, & d_s < \texttt{STATE\_TOL} \text{ and } a_i = \min(\mathcal{A}) \\ 2.0, & d_s < \texttt{STATE\_TOL} \text{ and } a_i = \max(\mathcal{A}) \end{cases}$$

where `STATE_TOL=1e-4`. We set `UPDATE_FREQ = 1` so one backwards pass is taken for every step. Our agent successfully learns over 20 episodes, receiving rewards of around 0.8 by the end of training. The loss of the network with respect to the predicted Q-values also decreases over time, and the error per timestep decreases from over 0.08 to under the tolerance (10^{-4}) (Figure 3). We also show some example predicted trajectories in Figure 4.

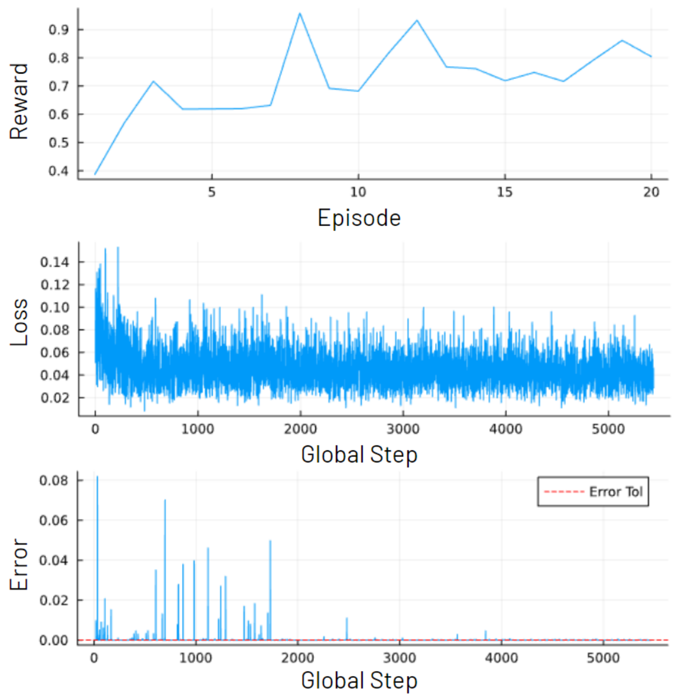

Fig. 3: Training history for the `LorenzSystem` agent. Top to bottom: Average reward per episode, loss per step, position error per step. This figure is analogous to Figure 3a in [6].

We observe the timestepping behavior of the agent during the last training episode (Figure 5). As expected, the agent makes steps of varying size, ranging from the smallest timestep (0.020) to the second-largest (0.060). The step sizes oscillate and are somewhat inversely correlated with the value of the z axis. Although the stepwise

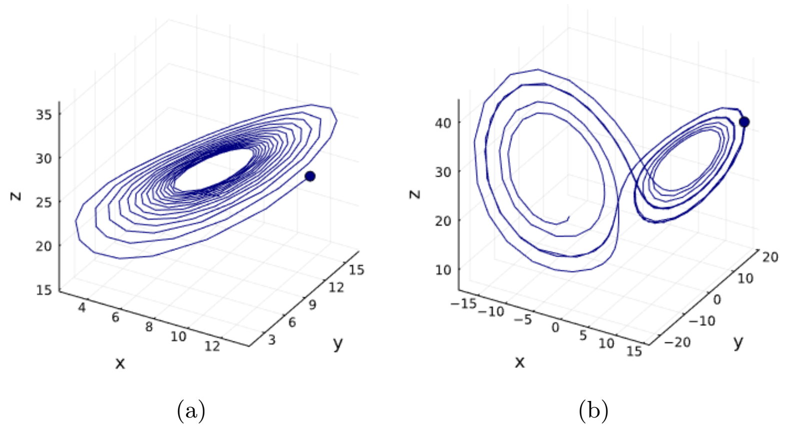

Fig. 4: Example trajectories predicted using RL adaptive timestepping for the Lorenz system.

errors vary substantially, on average the error remains below the stated tolerance of 10^{-4}.

Finally, we benchmark the timestepping efficiency of the agent against the standard `DP5()` solver in `DifferentialEquations.jl` (Figure 6) for a new system simulated from $t_0 = 0$ to $t_{max} = 20$ with initial conditions $(x, y, z) = (10, 10, 10)$. During this evaluation process, the agent selects actions with a fully-greedy policy ($\epsilon = 0$). We plot the number of function evaluations used to achieve a given error. The number of function evaluations scales linearly with the number of steps taken, with the scaling factor depending on the order of the integrator used. Although the agent is trained to select steps based on a fixed error tolerance, in practice we found that the error-speed tradeoff can be adjusted dynamically after training by applying an integer adjustment term ω to the index i of the optimal action[m]. Thus, the agent can achieve higher or lower errors than its training error threshold, while using fewer or more function evaluations, respectively. In all cases, the agent requires fewer function evaluations than `DP5()` to achieve a given error threshold. These results are comparable to those in [6].

3.2. Solar system.

After demonstrating the basic RL adaptive timestepping paradigm for the Lorenz system, we implement fully the modified technique described in section 2 for an n-body system. The motion of the bodies in the n-body system is determined by solving the ODEs defined in subsection 1.1. As a proof of concept, we investigate a configuration of 5 bodies, modeling the Sun and its four nearest planets in the solar system, i.e. Mercury through Mars. Despite appearing to be stable, the solar system is chaotic, and determining the exact location of any planet in the future is impossible. Here we simulate the motion of the solar system for 5×10^6 seconds, or a little under 58 days.

[m]The adjusted index $i + \omega$ is constrained by the limits of the action space.

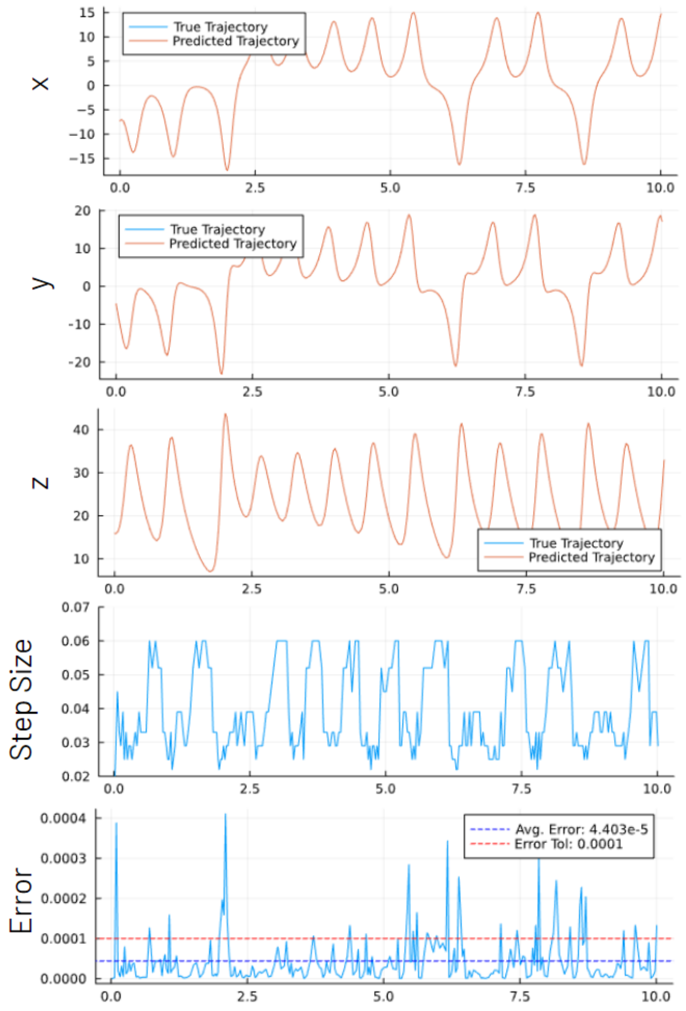

Fig. 5: Stepping behavior of the `LorenzSystem` agent for the last training episode (Episode 20, trajectory shown in Figure 4b). Top to bottom: Trajectory for x, trajectory for y, trajectory for z, step size, position error per step.

Table 1: Planets used to train the `NBodySystem` agent. We use SI units.

Planet	Weight (kg)	Radius from Sun (m)	Velocity (m/s)
Sun	1.989×10^{30}	0.0	0.0
Mercury	0.33011×10^{24}	57.909×10^{9}	47.36×10^{3}
Venus	4.8675×10^{24}	108.208×10^{9}	35.02×10^{3}
Earth	5.972×10^{24}	149.6×10^{9}	29.78×10^{3}
Mars	0.64171×10^{24}	227.9×10^{9}	24.13×10^{3}

We use SI units in our simulation, which means our gravitational constant is $G = 6.67430 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$. The exact parameters used in our simulation are shown in Table 1. The initial state of each simulation is generated by rotating each

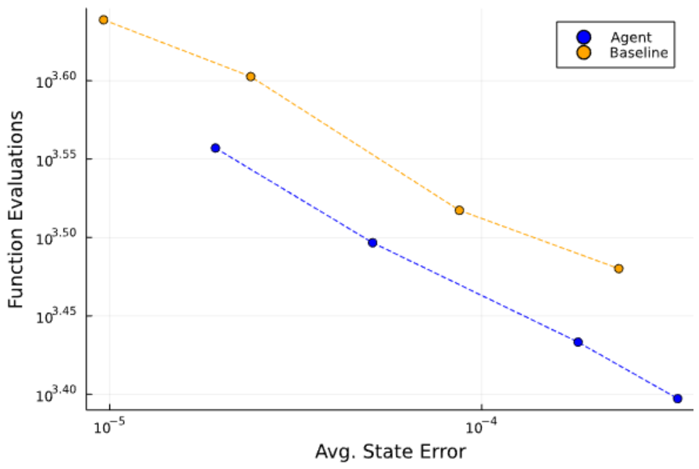

Fig. 6: Benchmarking the `LorenzSystem` agent versus a baseline `DP5()` adaptive solver with varying tolerance. The system is simulated from $t_0 = 0$ to $t_{max} = 20$ with initial conditions $(x, y, z) = (10, 10, 10)$. The y-axis shows the total number of function evaluations and the x-axis shows the average position (state) error for that episode. Different target error values can be achieved by the agent by boosting the index of the chosen step size (to increase error) or reducing the index of the chosen step size (to reduce error). Note that it takes the agent fewer function evaluations than `DP5()` to achieve a given error.

planet at a random angle along its orbit, while maintaining its distance from the Sun. Note that here we model the solar system as a series of flat, concentric circular orbits, though in reality they are elliptical and do not occupy the same plane.

For this system the agent has an action space comprising the following time steps, in seconds:

$$\mathcal{A} = \{3000, 6000, 9000, 12000, 15000, 18000, 21000, 24000, 27000, 30000\}$$

The scaling factors c_1, c_2, c_3, c_4 are calculated such that

$$r_s = \begin{cases} 0.1, & d_s < \texttt{STATE\_TOL} \text{ and } a_i = \min(\mathcal{A}) \\ 2.0, & d_s < \texttt{STATE\_TOL} \text{ and } a_i = \max(\mathcal{A}) \end{cases}$$

$$r_e = \begin{cases} 0.1, & d_e < \texttt{ENERGY\_TOL} \text{ and } a_i = \min(\mathcal{A}) \\ 2.0, & d_e < \texttt{ENERGY\_TOL} \text{ and } a_i = \max(\mathcal{A}) \end{cases}$$

where `STATE_TOL=1e-4` and `ENERGY_TOL=3e-6`. The total reward is $r = \alpha r_s + (1 - \alpha) r_e$, where $alpha$ is set to 0.5. We plot the rewards, losses, and errors over 10 episodes of training (Figure 7). We show an example predicted trajectory in Figure 8. Getting the `NBodySystem` agent to converge is substantially more difficult than getting the `LorenzSystem` agent to converge, as shown by the loss function which begins increasing after an initial decrease, despite the apparent increase in total reward. We suspect that this may be improved with (1) additional hyperparameter tuning, (2) reducing the absolute scale of values, and (3) implementing an exponential moving average-based teacher-student distillation paradigm in the style of [5] in order to mitigate instability in the target Q-values.

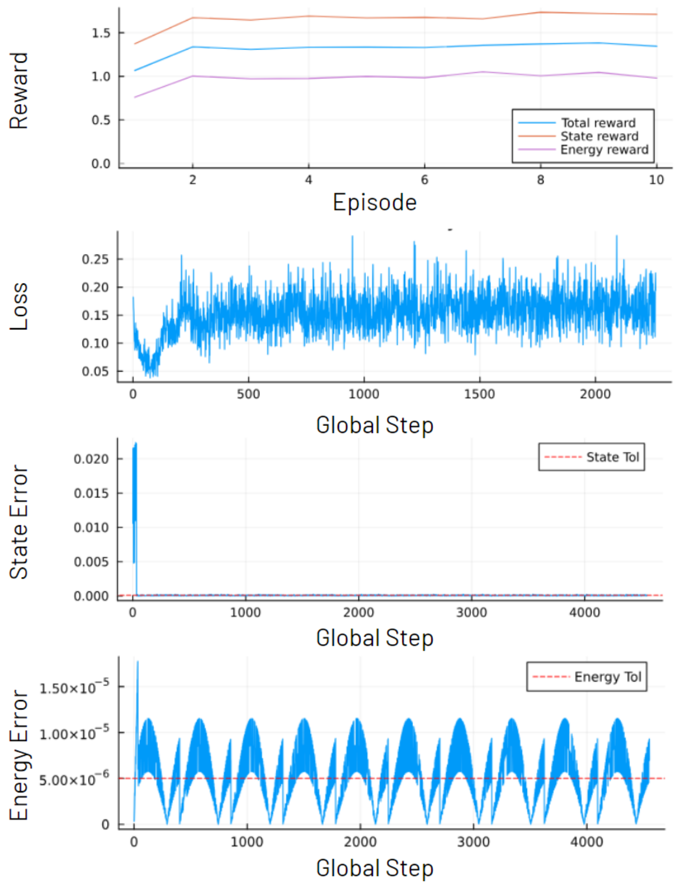

Fig. 7: Training history for the `NBodySystem` agent. Top to bottom: Average rewards per episode, loss per step, position-velocity error per step, energy error per step.

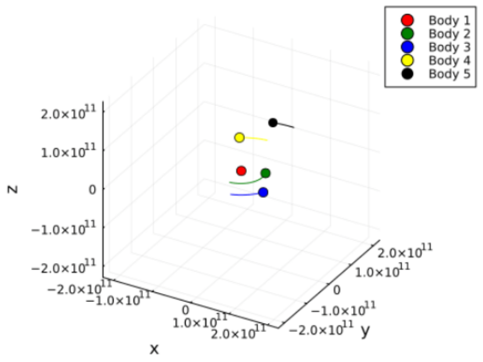

Fig. 8: Example trajectory predicted using RL adaptive timestepping for the solar system. One frame of an animation shown.

As before, we observe the timestepping behavior of the agent during the last training episode (Figure 9). There is still some variation in the agent's chosen step sizes, although the variation is much less than that of the `LorenzSystem` agent. On average the position-velocity and energy errors remain below their respective tolerances.

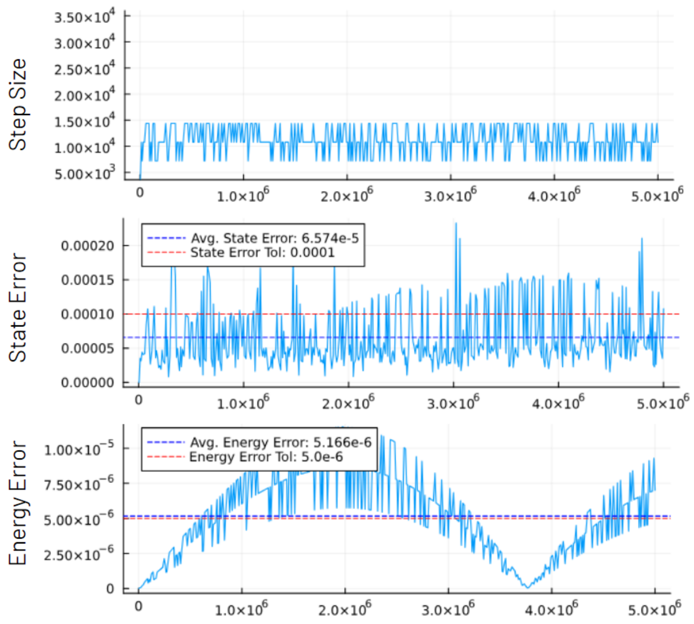

Fig. 9: Stepping behavior of the `NBodySystem` agent for the last training episode (Episode 10, trajectory shown in Figure 8). Top to bottom: step size, position-velocity error per step, energy error per step.

Finally, we again benchmark the timestepping efficiency of the agent against the standard `DP5()` solver in `DifferentialEquations.jl` (Figure 10) for a new system simulated from $t_0 = 0$ to $t_{max} = 20$ with all the planets initialized along the positive x axis. During this evaluation process, the agent selects actions with a fully-greedy policy ($\epsilon = 0$). As with the Lorenz system, the agent requires achieves a much lower state error than `DP5()` with the same number of function evaluations (Figure 10a). However, the agent is not more efficient than `DP5()` for the energy error (Figure 10b).

3.3. Absolute runtimes. Our results above demonstrate that our agent is more efficent than the `DP5()` baseline solver in terms of function evaluations, but unfortunately this does not translate to faster runtimes for the systems we tested. For the solar system implementation, our RL approach is substantially slower (Figure 11). We hypothesize that the computational cost of running the forward pass through our agent neural network outweighs the time saved from the fewer function evaluations, for low n. Moreover, though we implemented our adaptive timestepping scheme using custom types and methods in a way that is easily understandable and extensible, there are obvious tradeoffs in terms of time and space complexity. It is likely that [6] did not benchmark absolute runtimes of their approach for these reasons. Further investigation should establish whether the theoretical efficiency advantage of our model is manifested with additional optimization and a large n (e.g. on the order of 10^4).

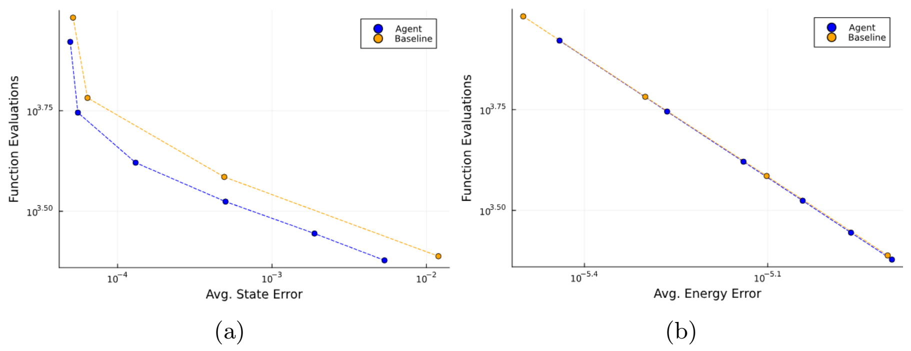

Fig. 10: Benchmarking the `NBodySystem` agent versus a baseline `DP5()` adaptive solver with varying tolerance. The system is simulated from $t_0 = 0$ to $t_{max} = 20$ with all the planets initialized along the positive x axis. The y-axis shows the total number of function evaluations and the x-axis shows the average position-velocity error (a) or energy error (b) for that episode. The agent is more efficient than `DP5()` with regards to the position-velocity error, but not more efficient with regards to the energy error.

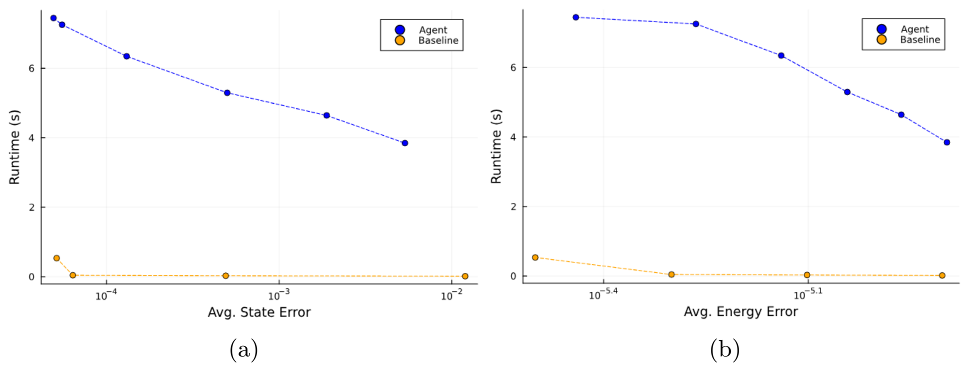

Fig. 11: Comparing absolute runtimes of the `NBodySystem` agent versus a baseline `DP5()` adaptive solver with varying tolerance. The system is simulated from $t_0 = 0$ to $t_{max} = 20$ with all the planets initialized along the positive x axis. The y-axis shows the episode runtime and the x-axis shows the average position-velocity error (a) or energy error (b) for that episode. The agent is substantially slower than the baseline solver.

4. Conclusion.

4. Conclusion. Traditional research in dynamical systems has recently been enriched by developments in artificial intelligence, including reinforcement learning. Here, we extend RL to simulating the n-body problem, a centuries-old physics problem which has motivated the development of numerous high-performance algorithms but as of yet been relatively untouched by machine learning. Our Q-learning agent is inspired by previous work in adaptive timestepping for ODEs, but we adopt a physics-informed approach by using a hybrid reward function which explicitly incentivizes the agent to preserve the Hamiltonian of an n-body system. Moreover, the agent is provided additional context including consecutive prior timesteps as well as the mass of each body. Our results, though only a proof of concept, demonstrate that a physics-informed reinforcement learning approach is possible for n-body systems. Nevertheless, challenges remain in improving the stability of training and generalizability to diverse initial conditions. Though we implement our RL agent in Julia with a focus on readability and extensibility, additional work remains to be done on optimizing the time and space complexity of our system to make it competitive with state-of-the-art adaptive solvers.

Source code. Our source code is available on GitHub: https://github.com/aspartate/nbody-rl-public.

REFERENCES

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, *A brief survey of deep reinforcement learning*, arXiv preprint arXiv:1708.05866, (2017).

[2] J. Barnes and P. Hut, *A hierarchical o (n log n) force-calculation algorithm*, nature, 324 (1986), pp. 446–449.

[3] P. G. Breen, C. N. Foley, T. Boekholt, and S. P. Zwart, *Newton versus the machine: solving the chaotic three-body problem using deep neural networks*, Monthly Notices of the Royal Astronomical Society, 494 (2020), pp. 2465–2470.

[4] M. X. Cai, S. P. Zwart, and D. Podareanu, *Neural symplectic integrator with hamiltonian inductive bias for the gravitational n-body problem*, arXiv preprint arXiv:2111.15631, (2021).

[5] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, *Emerging properties in self-supervised vision transformers*, in Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 9650–9660.

[6] M. Dellnitz, E. Hüllermeier, M. Lücke, S. Ober-Blöbaum, C. Offen, S. Peitz, and K. Pfannschmidt, *Efficient time-stepping for numerical integration using reinforcement learning*, SIAM Journal on Scientific Computing, 45 (2023), pp. A579–A595.

[7] J. R. Dormand and P. J. Prince, *A family of embedded runge-kutta formulae*, Journal of computational and applied mathematics, 6 (1980), pp. 19–26.

[8] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of computational physics, 73 (1987), pp. 325–348.

[9] P. Kumar, A. Das, and D. Gupta, *Differential euler: Designing a neural network approximator to solve the chaotic three body problem*, arXiv preprint arXiv:2101.08486, (2021).

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, arXiv preprint arXiv:1312.5602, (2013).

[11] P. Saha and S. Tremaine, *Symplectic integrators for solar system dynamics*, Astronomical Journal (ISSN 0004-6256), vol. 104, no. 4, p. 1633-1640., 104 (1992), pp. 1633–1640.

[12] L. Shampine and W. Zhang, *Rate of convergence of multistep codes started by variation of order and stepsize*, SIAM journal on numerical analysis, 27 (1990), pp. 1506–1518.

[13] V. S. Ulibarrena, P. Horn, S. P. Zwart, E. Sellentin, B. Koren, and M. X. Cai, *Hybrid integration of the gravitational n-body problem with artificial neural networks*, NeurIPS Machine Learning and the Physical Sciences Workshop, (2022).