CONTENTS

1 INTRODUCTION

Few problems in the history of science have drawn as much fascination as the movement of celestial bodies in space. What we now denominate the N-body problem, has been a subject of study since the beginning of history and is still being studied nowadays. One of the main methods to study this problem is through numerical simulations.

Astronomers have historically made their discoveries by looking through the eyepiece of a telescope. Nowadays, however, scientists can take advantage of the known laws of physics to create computer programs that mimic how the Universe works. Those programs, or simulations, can be used together with observations from telescopes as two faces of the same coin, to compare what is happening against what we think is supposed to happen. As the strength of astronomical observations relies on having a good telescope and using it correctly, the same applies to simulations and the numerical tools that are used to create them. The strength of computational astrophysics relies upon the quality of the numerical methods employed.

Our contribution to the field of computational astrophysics is the exploration and creation of new methods that optimize simulations of the gravitational N-body problem. We take advantage of the recent, paramount popularity of Machine Learning methods to find tools that can suit our problem. But let us take things one step at a time.

1.1 Part 1. The gravitational N-body problem

One of the first models of the Universe that is known to history dates from the fourth century BCE. Eudoxus believed that the universe was arranged as a series of nested spheres sharing the same center. The center would be the Earth and there would be one sphere for each of the five known planets, the Sun, and the Moon. Finally, one last sphere contained the stars. Aristotle, being a contemporary of Eudoxus, contributed to improving the model by adding some spheres to counteract the motion of the previous planetary sphere (see a more complete historical description in Americo (2017)). In the second century AD, the Alexandrian astronomer Ptolemaeus formulated a new model in his *Almagest* and Planetary Hypotheses (Toomer (1998)). This model was exceedingly complex. In order to replicate the perceived movement of the celestial bodies in the sky, this model indicated that the movement of each of the five planets is formed by two circles; the epicycle and the deferent (see Figure 1.1). The difficulty of the model lies in the fact that the deferent moves off-center with respect to the Earth but this movement was necessary to account

for the "imperfections" that had been observed by Greek and Babylonian astronomers.

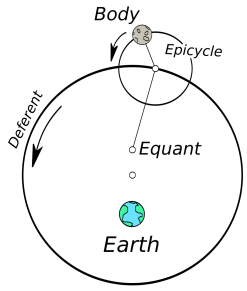
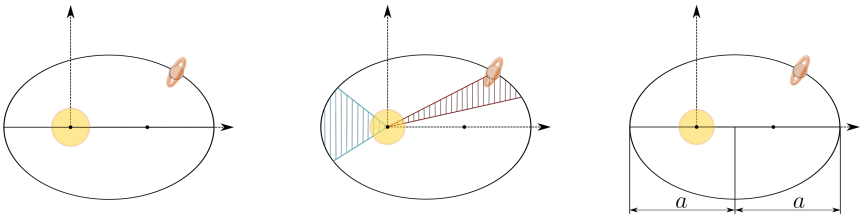


Figure 1.1: Simplified version of Ptolomaeus' model of the Solar System.

Ptolemaeus' *Almagest* became a very relevant text for astronomers in the Islamic world, and by the year 850 al-Fārghanī, a Persian astronomer, had used the current advances to update the astronomical theory explained by Ptolomaeus. In 1543, Copernicus proposed the first heliocentric model of the Solar System in his work: *On the Revolutions of the Celestial Spheres* (Copernicus (1543)). Based on the complex model by Ptolomaeus, he simplified it by positioning the Sun in the center of the Universe instead of the Earth. Between 1609 and 1619, Johannes Kepler published his famous three laws of planetary motion. Challenging the models by Copernicus and Brahe, he stated that the planets followed elliptical orbits with the Sun located in one of the foci (Figure 1.2a). He also established that a planet swept equal areas in equal times, which means that it travels faster when close to the Sun (Figure 1.2b). Finally, his third law states that the orbital period is proportional to the cube of the semi-major axis of the orbit (Figure 1.2c).



(a) First law. **(b)** Second law. **(c)** Third law.

Figure 1.2: The three Kepler laws of planetary motion.

The publication of Isaac Newton's *Philosophiae Naturalis Principia* in 1687 (Newton (1687)) provided for the first time a unified equation of universal gravitation that explained Kepler's empirical results. This law states that the two bodies attract each other with a force ($\vec{F}$) that is proportional to their masses (m) and inversely proportional to the square of the distance between them ($\vec{r}_{12}$);

$$\vec{F} = G\frac{m_1 m_2}{|\vec{r}_{12}|^3}\,\vec{r}_{12}. \tag{1.1}$$

Newton's universal law of gravitation (Equation 1.1) remains valid to our days for many cases. However, with the presentation of Albert Einstein's theory of General Relativity in 1915, it became clear that gravity is a perceived consequence of the motion through space-time. This conclusion led to the final understanding of phenomena such as the abnormal orbit of Mercury and the existence of black holes. However, for the remainder of this work, we will ignore relativistic effects and focus on systems dominated by classical mechanics, i.e. Newtonian mechanics.

1.1.1 The two-body problem

Two celestial bodies attract each other through their gravitational force following Newton's law of gravitation (Equation 1.1)). The force exerted by each body on the others is proportional to its mass and inversely proportional to the distance separating them (see Figure 1.3).

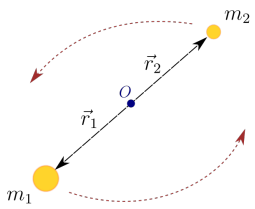


Figure 1.3: Simplified representation of the two-body problem.

Thanks to the equations derived by Kepler for the motion of a body orbiting another, the trajectory of a two-body problem can be calculated analytically. This means that knowing the state of a body around another one, finding its state at any future time is a fast and relatively easy problem to solve. However, it should be taken into consideration that a challenge arises when the eccentricity of the orbit is close to 1 (we will explain the meaning of the eccentricity in Subsection 1.1.2). In this case, it becomes challenging to achieve convergence in the solver for Kepler's equation (Elipe et al. (2017)).

1.1.2 State representation

Every system of N bodies has $6N$ degrees of freedom. The state of the system ($\vec{s}$) can be defined with 6 independent variables for each of the bodies present. The most common method to fully describe a system is with their positions (3 values per particle for a 3-dimensional space) and their velocity (also 3 values for a 3-dimensional space):

$$\vec{s} = [x,\ y,\ z,\ v_x,\ v_y,\ v_z]. \tag{1.2}$$

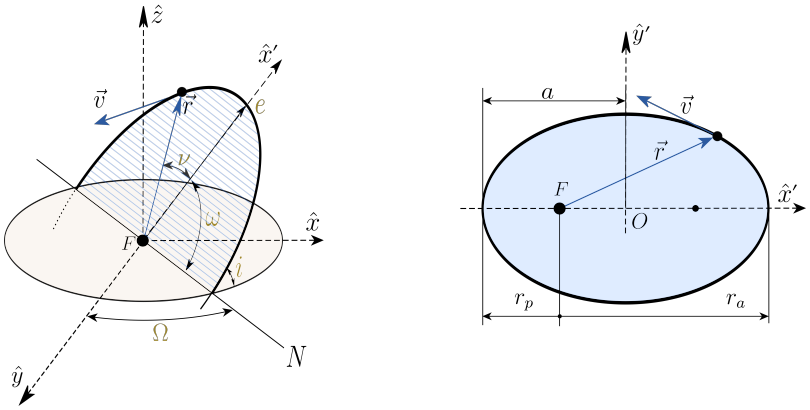
A particular case of the two-body problem is that in which one body is significantly more massive than the other. This is for example the case of our Solar System, as the Sun is more massive than any other body around it. If we imagine a system formed only by

the Sun and one minor body orbiting it (like a small planet or an asteroid), the minor body will orbit the major following an ellipse. In this case, the state of the minor body can be described by its position and velocity, but it becomes more convenient in many cases to use Keplerian elements.

Keplerian elements describe a body focusing on the shape of its orbit (Curtis (2019)). The system is fully described with 6 elements which are shown in Figures 1.4a and 1.4b. In the figures, N represents the line of nodes, i.e., the intersection line between the orbital plane and the ecliptic plane. The ecliptic plane is defined by the equator of the central body.

Semi-major axis (a): the semi-major axis is a measurement of the size of the orbit. It is half the size of the major axis of the ellipse. It is a quantity with units of length.

Eccentricity (e): describes the shape of the orbit. For an ellipse, the eccentricity is a value between 0 and 1. When the eccentricity is 1, the trajectory becomes a parabola, and for eccentricities larger than 1, a hyperbola. This means that the orbiting body is no longer bound to the central one, but has escaped the system. It is an adimensional quantity.



(a) Three-dimensional representation.

(b) Planar orbit representation.

Figure 1.4: Representation of the different elements that define an orbit.

Inclination (i): the inclination is the angle between the orbital plane and the ecliptic plane. It is a positive number between 0° and 180°.

Argument of perigee (ω): angle between the line of nodes N and the eccentricity vector, which is the vector from the center body to the periapsis of the orbit. It therefore defines the position of the periapsis. It is an angle between 0° and 360°.

Right ascension of the ascending node (Ω): angle between the y-axis ($\hat{y}$) and the line of nodes (N). It is an angle between 0° and 360°.

True anomaly (ν): the true anomaly, defines the position of the object in its orbit. It is the angle between the x-axis of the orbit $\hat{x}'$ and the position vector (see Figure 1.4b). It is defined between $0°$ and $360°$.

Therefore, the state of the system $\vec{s}$ can also be defined as:

$$\vec{s} = [a,\ e,\ i,\ \omega,\ \Omega,\ \nu]. \tag{1.3}$$

Additionally, it is important to define the periapsis and apoapsis as the closest and furthest points in the orbit from the central body, respectively.

1.1.3 Hamiltonian Systems and Conservation laws

In systems where the forces are derived from a potential function, the equations of motion can be written as a Hamiltonian system (Easton (1993)). The Hamiltonian formalism is the mathematical structure in which to develop the theory of conservative mechanical systems (Yahalom (2024)). A system of *N* particles interacting via Newtonian gravitational forces is a Hamiltonian system that can be described using Hamilton's formulation. The system is then formed by two first-order ordinary differential equations for each particle n as

$$\frac{d\vec{q}}{dt} = \frac{\partial \mathcal{H}}{\partial \vec{p}_i}, \qquad \frac{d\vec{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \vec{q}_i}, \qquad i = 1, ..., N; \tag{1.4}$$

where $\vec{q}$ and $\vec{p}$ represent the position and momentum vectors, respectively. The momentum can be calculated by multiplying the mass of a particle by its velocity vector. The Hamiltonian $\mathcal{H}$ represents the total energy of the system and is formed by two terms: one for the kinetic energy and one for the potential energy. It is defined as a function of the position and momentum vectors of the system, the masses m of the bodies, and the universal gravitational constant G as

$$\mathcal{H} = \underbrace{\sum_{i=0}^{N-1} \frac{||\vec{p}_i||^2}{2m_i}}_{\text{Kinetic Energy}} \underbrace{- G \sum_{i=0}^{N-2} m_i \sum_{j=i+1}^{N-1} \frac{m_j}{||\vec{q}_j - \vec{q}_i||}}_{\text{Potential Energy}}. \tag{1.5}$$

In Equation 1.5, the Hamiltonian is independent of time. Therefore,

$$\frac{d\mathcal{H}}{dt} = 0, \tag{1.6}$$

which means that the total energy of the system is conserved. A change in energy in a numerical simulation is an indication of the system not following the laws of nature, and can therefore be used as an indication of the quality of numerical simulations (see Subsection 1.1.5 and Chapters 2 to 5).

The second conservation law is the conservation of angular momentum. Angular momentum is defined as the cross product of the position and velocity vectors (see Figure 1.5) in the form:

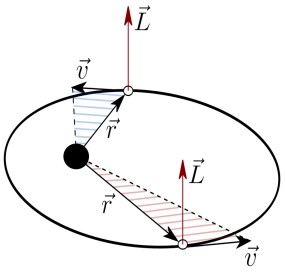


Figure 1.5: Conservation of angular momentum for a system of two bodies.

$$\vec{L} = \sum_{i=0}^{N} m_i \vec{r}_i \times \frac{d\vec{r}_i}{dt}. \tag{1.7}$$

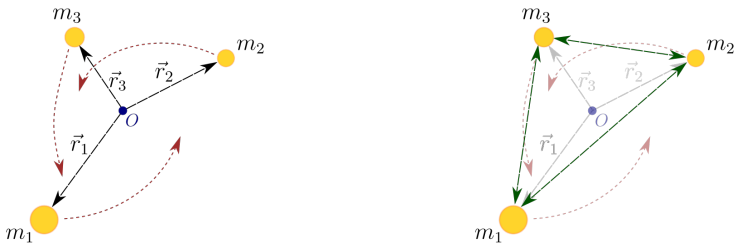
The derivative of the angular momentum equation with respect to time is 0, which means that angular momentum is conserved (Curtis (2019)). This implies that the movement of one body around another remains in a single plane.

1.1.4 The *N*-body problem

Pairs of celestial bodies are usually not found in isolation. Most of them are influenced by other bodies or are a part of a larger system. Thus, planets are found in planetary systems, stars are born in groups denominated star clusters, and all of those are also grouped in galaxies. The Universe is a system of *N*-bodies that interact with one another. In reality, most problems can be simplified by assuming that only the closest, or more massive, bodies will have an influence over the one being studied. For example, the gravitational potential of the center of our galaxy will always influence the movement of every object in the Solar System. However, because this effect will be small compared to other more relevant ones, for example, the influence of the Sun or Jupiter, we can choose to ignore this far-away influence depending on the phenomenon that is to be studied.

The simplest case of the *N*-body problem is a system with three bodies. In a system as the one represented in Figure 1.6a where three equal-mass stars move around their center of mass, each body interacts with the other two (Figure 1.6b) with a force defined by Newton's equation (Equation 1.1). Unlike in the two-body problem where one particle followed an ellipse around the other one[1], for a system of three bodies, the trajectories followed are complex and vary substantially depending on the initial conditions chosen. Three examples of those trajectories are shown in Figure 1.7. We will talk in more depth about chaos in the *N*-body problem and its implications for numerical integration in Subsection 1.1.6. As the number of bodies increases, so does the level of complexity of the trajectories followed by individual bodies.

[1]Fixing the center of the system at the center of mass of one of the bodies.



(a) Representation of the three-body problem. **(b)** Interactions in the three-body problem.

Figure 1.6: Simplified representation of the three-body problem and the interactions between bodies.

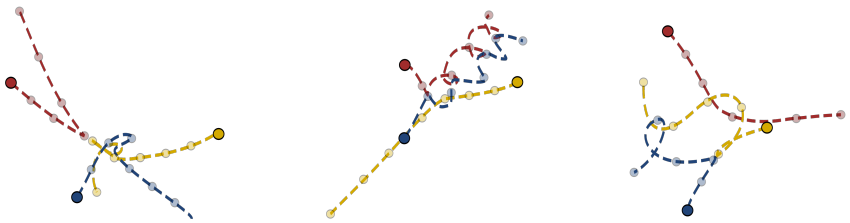


Figure 1.7: Examples of the trajectory of a system of three bodies.

In a two-body problem, the calculation of the gravitational force between the bodies has only one contribution. For a system of *N* bodies, Newton's equation can be rewritten as

$$\vec{F}_i = Gm_i \sum_{j \neq i} \frac{m_j}{|\vec{r}_{ij}|^2} \hat{r}_{ij}, \qquad i = 1, ..., N; \tag{1.8}$$

for each body. To calculate all the forces in a system of three particles, it would take 6 operations (or three times Equation 1.8). This number grows with the number of bodies in the system. For *N* bodies, the complexity -or number of calculations- scales quadratically with N. Knowing that the force exerted on body i by another body j is reciprocal to the one by body j on i, the number of calculations can be reduced.

Unlike in the case of the two-body problem, there is no analytical solution for the equations of motion. Thus, in order to know the future state of the system, we cannot apply a given equation, but we need to take small steps using numerical integrators.

1.1.5 Numerical integration

Celestial bodies move in a continuous phase-space. At each moment in time their positions, velocities, and the forces acting on them vary. However, their trajectories cannot be computed analytically for $N \geq 3$, so we need to use numerical integration methods. Such methods propose approximate solutions for a given integral. For the problem of *N*

bodies moving under the influence of gravity, the differential equation to be solved can be written as a first-order differential equation

$$\begin{pmatrix} \dot{\vec{r}} \\ \dot{\vec{v}} \end{pmatrix} = \begin{pmatrix} \vec{v} \\ -G\sum_{i\neq j} \frac{m_j}{r_{ij}^2}\hat{r}_{ij} \end{pmatrix}. \tag{1.9}$$

In our case, numerical integrators divide the trajectory into discrete parts called steps and solve the equations of motion (Equation 1.9) for each of those consecutively. A representation of the three-body problem can be seen in Figure 1.8a. From the initial conditions or initial state $\vec{s}^{\,t_0}$, the state of the system after one time step Δt can be calculated using Equation 1.9. Once we have the new state at time t_i, the process can be repeated until a final time is reached (see Figure 1.8b).

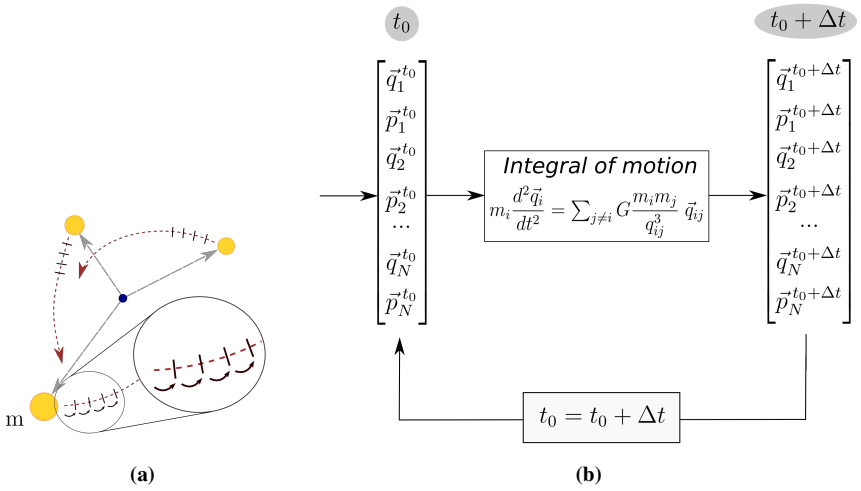


Figure 1.8: Graphic (a) and Schematic (b) representations of a numerical integrator.

Time step size and complexity

The size of the time step is a decision variable. Depending on the problem to be solved, a different value has to be chosen to satisfy certain conditions. A large value of the time step will lead to faster results, as the number of times that the equation of motion needs to be solved is smaller. However, a large time step size is a coarse approximation of the continuous phase space, and therefore the results will be less true to the physical world. In contrast, a small time step size will lead to more accurate results, as it represents a better approximation of the continuity of the trajectory, at the cost of computation time. Finding the right balance between accuracy and computation time is a problem-dependent issue. It will also be the main focus of Chapters 4 and 5 in this thesis. Whereas for simple experiments low accuracy results may suffice, for most cases in astrophysics a good accuracy is required to ensure that the obtained results are true to the actual behavior of celestial bodies.

Despite the time-step size being one of the main parameters that determines computation time, the number of bodies in the system is also a main contributor. In Subsection 1.1.4 we defined complexity as the number of operations required per time step to integrate the system. As N increases, so does the complexity, leading to radically more expensive computations.

Numerical errors

In a numerical simulation, there are many sources of errors. Those will drive our simulation away from the truth. Firstly, we have previously talked about discretization errors. Those appear from the assumption that the trajectory can be made into discrete pieces, whereas in reality, it is continuous. It is directly related to the time-step size. The larger the time-step size, the larger the error incurred.

Secondly, the simulation will suffer from round-off errors. Those refer to the error caused by the limited precision of the computer (Boekholt & Portegies Zwart (2015b)). Most algorithms are by default limited to 15 significant digits to store the solutions, leading to an accumulation of round-off errors at each time step. This error grows with the number of integration steps. Unfortunately, this means that reducing the discretization error (by reducing the time-step size) can lead to an increase in the round-off error.

Additional sources of error will appear depending on the specific configuration of the integrator. We will describe different types of integrators and their implementations. We adopt the denomination of unphysical solutions for those cases in which the errors have grown to a point in which conservation laws are no longer fulfilled and therefore the simulation does not adhere to the laws of physics. In those cases, the simulations are no longer useful for most scientific purposes.

Types of numerical integrators

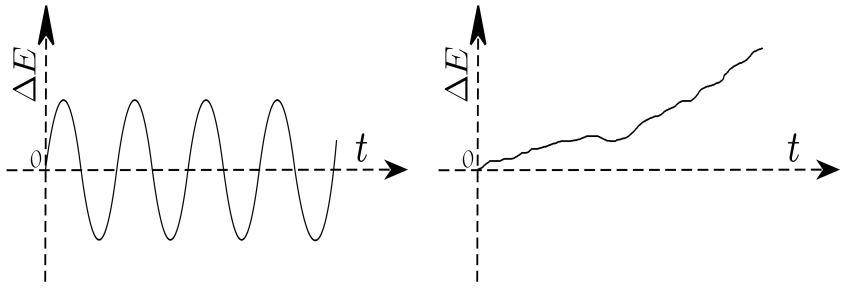
Numerical integrators have been developed and optimized for specific applications. The large range of astronomy problems has also led to a variety of integrators. Here we will review some of the most relevant types of codes used to solve the N-body problem.

The simplest type are pure codes. These do not include any physical parameters that need to be chosen except for the time-step criterion (Zwart & McMillan (2018)). Examples like `Ph4` and `Hermite` will be used in this thesis. Another type are direct N-body codes which include additional parameters to speed up the code or reduce numerical errors. The complexity of these codes scales with N^2. An example of this category is a code specially suitable for the long-term integration of the Solar System; Wisdom-Holmann integrator (Wisdom & Holman (1991)). This code belongs to a special type of integrators denominated *symplectic codes*. Because of their importance for this thesis, they deserve a more in-depth explanation. Finally, in order to reduce the computation time in certain cases, there are approximate methods. For example, tree codes such as the `Barnes-Hut` tree scheme (Barnes & Hut (1986)) assume that only particles close to a given one will contribute to its gravitational potential and the far away particles can be grouped into a single effect. This code saves computation time as its complexity scales with $N\log(N)$.

Symplectic integrators

In Hamiltonian systems, the solutions to the equations lie on a symplectic manifold in phase space. A symplectic integrator is one in which the solution resides on the symplectic manifold. A detailed mathematical description of the definition of symplecticity can be found in Sanz-Serna (1992).

We have talked about the numerical errors that appear in a simulation. Instead of the energy being perfectly conserved, discretization error leads to a linear drift in the energy error over time. In symplectic integrators, the energy error is different than zero, but instead of drifting (see Figure 1.9b), it oscillates around the zero value as in Figure 1.9a.



(a) Symplectic integrator **(b)** Not symplectic integrator

Figure 1.9: Difference in the evolution of the energy error for a symplectic integrator (a) and a regular integrator (b).

Because of this property, they are specially suitable for the long-term integration of planetary systems, as in Rein et al. (2019). In this case, the integrators divide the Hamiltonian into two different parts: one that contains the planetary motions around the central star, and a second one describing the interactions planet-planet.

1.1.6 Chaos

A fundamental characteristic of the N-body problem is its chaotic nature. In a chaotic system, a small perturbation to the initial condition grows exponentially with time. In Figure 1.10, we show how a small change in the position of one particle at the beginning of the simulation can lead to a completely different output. Similarly, numerical errors accumulate during a simulation and can cause the final solution to diverge from the real one. For that reason, for chaotic systems, the necessary precision increases exponentially with time (Srivastava et al. (1990)). In order to prevent the accumulation of numerical errors, Boekholt & Portegies Zwart (2015b) design an integrator with arbitrary precision. This integrator allows for minimizing round-off errors and in converged solutions, discretization errors. Then, it can be compared to other commonly used integrators. They tested their integrator (`Brutus`) against other commonly used ones on the Pythagorean problem (a special case of the democratic three-body problem). Their findings show that only about half of the solutions give accurate results compared to the ones obtained with the accurate integrator.

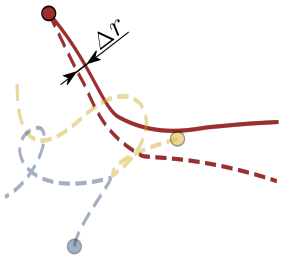


Figure 1.10: Representation of chaos in a three-body problem.

Chaos represents an important challenge for the simulation of N-body systems. Its effect on our methods will be inevitably present in the next chapters.

1.1.7 General assumptions

We have mentioned at the beginning of this chapter that we will not consider General Relativity (GR) in our experiments. Our assumption remains valid as long as we ensure that classical dynamics dominate in our system. For example, when talking about planetary systems, Mercury should not be included as its dynamics are dominated by the general relativity effects caused by its proximity to the Sun. In fact, its anomalous rate of precession was discovered in 1859 but could only be explained by Einstein's general relativity in 1915 (Yahalom (2022)). For other bodies orbiting further away from the Sun, the GR effects become more subtle and can in many cases be considered negligible.

In this work, we consider bodies as point masses. By doing so, we ignore the radius of the body and assume that all the mass is concentrated on its center of mass. Additionally, when looking at planetary systems and star clusters, we ignore stellar evolution and assume that the mass of the star remains constant. Similarly, we ignore effects such as radiation pressure which is mostly relevant for small objects, as shown in examples by Mignard (1982) and Belkin & Kuznetsov (2021).

1.2 Part 2. Machine Learning in Astronomy

The field of computational astrophysics is in constant need of faster and more efficient methods to adapt to the increasing complexity of the simulations performed. Numerical integrators have been developed for decades and are currently optimized for different problems. Despite the work put on perfecting these methods, there is a limit to what they can achieve.

With the fast growth of the field, many new Machine Learning (ML) methods have been created in the past years. Those methods cover a large range of applications from classification tasks to optimal control problems. In Figure 1.11, we show a simplified classification of ML methods. Supervised learning encompasses those methods in which labels are available in the training dataset. This means that for the training data, we know the "real" solutions and the networks will train on learning those. There are two main types of supervised learning algorithms: classification tasks, in which the output is one of the different classes available, and regression, where the output is a rational number or an array of rationals. In contrast, unsupervised learning does not involve the correct solution, but instead focuses on finding patterns in the data. An example is clustering methods, in which the data samples are grouped according to certain characteristics. Another example is that of dimensionality reduction algorithms, in which the goal is to reduce the size of the data while preserving qualities of the data. Finally, the third type of ML is Reinforcement Learning (RL). In this case, an agent is used to learn to make decisions given a certain situation. We will dive deeper into these types of methods in Subsection 1.2.2.

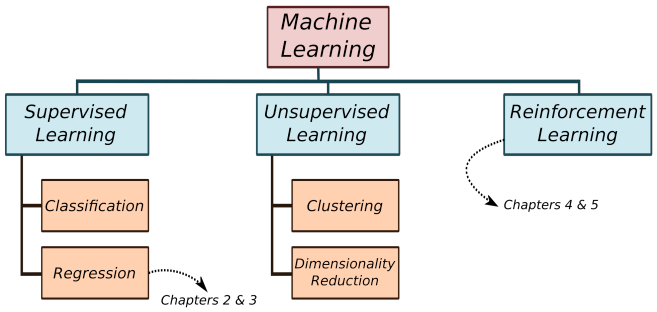


Figure 1.11: General classification of the different types of Machine Learning methods.

Despite the exceptional interest that the field has generated, most of those new methods are usually tested for the solution of simple equations or for overly simplified test cases. There are many possible applications of ML methods to scientific fields, but identifying the transferability of those methods from simple applications to complex cases such as the ones found in computational astrophysics is not trivial. We have explored many such topics and will show a brief description of many interesting applications in Subsection 1.2.3. However, this thesis focuses on two specific topics: physics-aware neural networks (Chapters 2 and 3) and Reinforcement Learning (Chapters 4 and 5). In the next subsections, we will aim to get a fundamental understanding of those two topics.

1.2.1 Physics-aware neural networks

Artificial Neural Networks (ANNs) are a type of supervised Machine Learning algorithms in which a set of inputs is used to obtain a desired output. ANNs are inspired by the function of biological neural networks. Its structure is formed by neurons, generally organized in layers, and connected to one another by nodes. The layers are usually divided into an input layer, an output layer, and hidden layers (as shown in Figure 1.12). The simplest type of ANNs are called Multi-Layer Perceptrons.

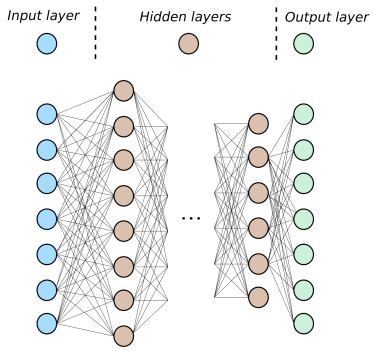


Figure 1.12: Simplified schematic of a basic Multi-Layer Perceptron.

The nodes connecting neurons carry a value denominated weight (W) that multiplies the value from the neuron in the previous layer. Additionally, a bias (b) is added to each layer. It is common to use an activation function f in the neurons to add non-linearities and expand the capabilities of the network. Starting from a set of inputs x, the values at the next layer (z) are calculated as

$$z = f\left(b + \sum_{i=1}^{n} x_i W_i\right) \tag{1.10}$$

for a layer with n neurons.

Both weights and bias are values that are chosen to make the network achieve a certain task. This process is not done manually but through a process denominated training. During training, the outputs achieved by the neural network with a certain set of trainable parameters (weights and biases) are compared to the desired output associated with an input (label). The difference between the desired output and the obtained one is denominated loss function. The number of layers and the number of neurons per layer are part of what is called *hyperparameters* and are values to be chosen in advance.

Neural networks have become extremely popular and many studies are being done to improve the training, activation functions, and hyperparameter optimization. Additionally, there are many studies working on their application to scientific fields. When applying neural networks to physical systems, it is important that the results adhere to the laws of physics. Since neural networks are mere statistical algorithms, physics are

not enforced. Therefore, Raissi et al. (2019a) designed the first type of neural networks that incorporated physical knowledge; Physics-informed Neural Networks. Since then, the field has grown at a rapid pace leading to a wide range of algorithms that in one way or another include some physical knowledge.

We can currently divide these algorithms into two different types depending on whether the physics are added as a soft or hard constraint. An example of the former is physics-informed neural networks (PiNNs) as the physical knowledge is incorporated into the loss function. This means that the physics act as a regularization term during the training, but during inference there is no assurance that the output will adhere to the given constraint. In contrast, some algorithms incorporate physical knowledge into their structure. In this case, the physics represent a hard constraint. Examples of this are Hamiltonian Neural Networks (Greydanus et al. (2019a)), SympNets (Jin et al. (2020a)), but a more detailed study about these networks is performed in Chapter 3.

Each type presents advantages and challenges of its own. In what concerns PiNNs, they showed that they could achieve better performance than regular neural networks with a smaller database. Since generating datasets is generally a computationally expensive task, it resulted in more efficient studies. Additionally, they managed to achieve better extrapolation capabilities in some cases. However, as mentioned before, they did not ensure that the results adhered to physics laws, which in some other cases led to unphysical results. After their initial popularity and many applications (Farea et al. (2024)), many studies have arisen to solve some of the initial challenges and create more robust algorithms. However, there is still progress needed for them to be able to learn intricate physical phenomena such as multi-scale and chaotic behaviors (Antonion et al. (2024)).

Neural networks that incorporate physics into their architecture have not experienced the same popularity as PiNNs, but their results are extremely promising. There are multiple types and each implementation is different. However, a common challenge is the complexity added by creating network structures that represent some physical phenomenon. By creating these structures, the problem becomes more rigid and their use to different problems can be challenging. These challenges will be addressed in Chapter 3. The main advantage of this method is that their results show a better adherence to the physical laws acting upon a system, and therefore better extrapolation to unseen data.

1.2.2 Reinforcement Learning

The second part of this thesis deals with the use of Reinforcement Learning (RL) algorithms. In RL, an agent is trained to learn how to choose between different actions in order to fulfill a desired task in a dynamic environment.

The agent and the environment interact with each other via the action, the state, and the reward function. The action represents the choice made by the agent and that will be implemented in the environment. The state is a representation of the environment that is fed into the agent for it to make a choice. Finally, the reward is the function that quantifies how well the action has worked to achieve the desired goal. A visual representation is presented in Figure 1.13. The agent makes the decision about the action to take based on a policy, which is a decision-making function to determine the control strategy of the agent.

There are multiple RL algorithms available, and they can be generally classified into

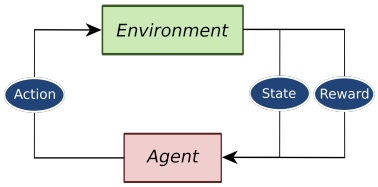


Figure 1.13: Interaction between the agent and the environment in RL algorithms .

model-based or model-free RL. The first use experience to construct an internal model of the transitions and outcomes of the environment (Dayan & Niv (2008)). The latter uses experience to learn directly the state and action values or policies without estimating or using a world model. The latter ones represent simpler algorithms, which becomes useful for preliminary experiments like the ones in this work.

RL algorithms have become widely used for optimal control problems such as self-driving cars, industry automation (Kiran et al. (2022)), finance and trading (Hambly et al. (2023)), neurobiology (Dayan & Niv (2008)), and healthcare (Coronato et al. (2020)), among others. Many RL applications for science problems focus on their use to create engineering tools that can automatically choose some parameters to create better-performing systems. Many examples can be found in the field of fluid mechanics and heat control (Novati et al. (2021); Viquerat et al. (2022)). In astronomy, most applications of RL focus on the optimal control of telescopes to improve the observation strategy (Jia et al. (2023)). However, some techniques have also been applied to the prediction of solar flares. In this thesis, we will focus on the use of RL techniques for the improvement of computational tools.

1.2.3 Applications and challenges

We have identified different applications for Machine Learning in the context of the integration of the gravitational *N*-body problem. As we have only fully developed two main possibilities (physics-aware NNs and Reinforcement Learning), we dedicate this Subsection to elaborate on some of the relevant ideas identified, with our view on their applications and challenges.

Surrogate for integration

The most straight forward application for ML in the integration of a system of N bodies is the replacement of the integrator with a neural network that predicts the state of the system at time t_{i+1} using the state at time t_i as an input. This idea is being applied to multiple fields but with a special focus on fluid mechanics and weather forecasting.

The recent popularity of ideas such as Physics-informed Neural Networks (see Subsection 1.2.1) has led to a variety of neural networks that aim to incorporate physical knowledge into their structure. These neural networks, but also those without physical constraints, can be applied to many problems in astrophysics that deal with time evo-

lution such as stellar evolution, origin, and evolution of planetary systems, star cluster dynamics, etc.

Each of these networks presents advantages and disadvantages of their own, but additional challenges appear when applying them to the gravitational N-body problem instead of the standard test cases. Chapter 2 presents a detailed explanation of the advantages and challenges of a variation of PINNs (Hamiltonian Neural Networks), when applied to a planetary system. Although the introduction of physical constraints into the neural network leads to a better adhesion to the dynamics and better extrapolation capabilities, the training of the networks becomes more challenging. New types of neural networks that include physical knowledge are being developed to overcome those challenges. Chapter 3 shows a new type of neural networks with a structure that resembles that of an integrator. Thanks to their structure, they lead to better extrapolation capabilities and adhesion to the physics.

There are two main challenges that appear when applying neural networks to the N-body problem. The first one is the accumulation of errors made by the neural network. In a chaotic system, this accumulation of errors can lead to large deviations from the ground truth on short time scales. Currently, the accuracy achieved by neural networks cannot compete with that of traditional integrators, making them inadequate for their use in chaotic systems. The second challenge refers to the fixed size of the input. The state of the system is used as an input to the neural network. This size remains fixed for a trained network, which means that it is not possible to change the number of bodies in the system without retraining. This limits the generalization possibilities of a trained network in astronomy.

Predictor for the secular evolution of a system

Similarly to the previous case, neural networks can be used to predict the future state of the system knowing some initial one. In this case, instead of predicting each integration step, the neural network can be used in a planetary system scenario to predict the secular evolution of the keplerian elements of the planets. In a planetary system, most of the evolution of a planet is an elliptical movement around the central body, but close encounters with other planets can lead to changes in the orbit.

By using the neural network only once per orbit, we prevent the accumulation of errors at each integration step. However, setting up the problem in a way that the search space is covered is not straight-forward. In order for the algorithm to be generalizable, the training samples need to be representative of the different scenarios that may appear. This becomes challenging due to the large range of possible configurations of planetary systems. Additionally, since each planet has a different orbital period, it is not clear how to choose a time at which the secular changes are calculated.

Chaos predictor

Understanding how chaotic a system is before running the simulation can be beneficial to choose the right time step size or, in the case of integrators such as `Brutus` (Boekholt & Portegies Zwart (2015a)), the size of the mantissa. The Lyapunov exponent can be used as a measurement of chaos.

A simple neural network can be used to predict the Lyapunov exponent before the simulation. The challenge arises from the training. In chaotic problems, similar initial conditions can lead to radically different outputs. Solving this problem may imply a different choice of inputs other than cartesian coordinates, training a network that is tuned to highly sensitive solutions, and having a large enough dataset to cover a broad range of cases.

Generation of initial conditions

In astronomical simulations, it is often challenging to find a set of initial conditions that is representative of real systems. For example, there are currently some assumptions that can be applied for the initialization of star clusters such as fractal cluster models (Goodwin & Whitworth (2004)) and several density models to choose their masses (see Plummer (1911)). However, some of those models are decades old and make simplifying assumptions. Additionally, in systems in which there is gas present, the choice of initialization becomes even more challenging.

There are multiple ML methods that are growing in popularity for their use as generative systems. Generative Adversarial Networks are a common example. Those could be applied to the problem of finding a set of initial conditions that fit observations but contain all the relevant information to run the simulation.

This problem comes with the challenge that the system must comply with physical laws such as conservation of energy and angular momentum. The use of neural networks for this problem requires the introduction of physical constraints during the training procedure.

System representation

Cartesian coordinates may not be the optimal choice for the representation of the state of a system when using neural networks. Their large variability may make the training more challenging depending on the problem to be solved. On the other hand, keplerian coordinates do not combine well with neural networks. This is due to the combination of linear parameters, such as the semi-major axis and the eccentricity, and periodic ones, such as the angles (see Subsection 1.1.2). The use of sines and cosines in the inputs many times benefits from specific activation functions, which might be incompatible with the linear inputs.

Both Cartesian and Keplerian coordinates have advantages and disadvantages depending on the problem. However, there is currently no state representation designed specifically for its use in combination with neural networks. Finding a state representation that is optimal for a specific problem could make a substantial difference in the final performance of an algorithm.

Autoencoders have been designed to compress information while retaining representative features. This compression can be combined with any of the aforementioned applications to improve the performance of the trained networks.

Choice of integration parameters

The choice of integration parameters can be thought of as a control problem. Although some of these parameters can be chosen a priori, others should be adapted dynamically during the simulation to adapt to fast-changing conditions.

Reinforcement Learning can be used during the simulation to choose parameters such as the time-step or time-step parameter (depending on the chosen integrator). Additionally, it could be used to make other choices such as the integrator at each step, or even individual choices for each particle. We study two applications of reinforcement learning for the choice of time step in Chapters 4 and 5.

The case of `Brutus`

A specially interesting case is that of `Brutus` integrator (Boekholt & Portegies Zwart (2015a)). This method allows a free choice of accuracy and mantissa to ensure that the final solution is free of numerical errors to a given accuracy, i.e., the simulation is converged. In order to find a converged solution, it is necessary to run the code multiple times with different choices of accuracy and mantissa until the results do not change significantly. This leads to extremely computationally expensive runs, specially for chaotic problems.

A RL algorithm that can choose those dynamically would be extremely useful to avoid the need for the repetition of the simulation multiple times, therefore leading to a substantial speed-up of each simulation. The main challenge in this case arises from the fact that once the mantissa has been decreased to a certain number of decimal places, it cannot be increased without incurring round-off errors. An alternative option to speed up this convergence process using ML would be to train a neural network to predict a priori what the simulation parameters should be, even thought in this case they would not be changed dynamically during the simulation.

Grouping algorithms

The integration of star clusters can be computationally expensive due to the large number of bodies present. A speed-up of this integration is sometimes achieved by using integrators that group stars depending on the strength of the effect on each other. This is the example of tree codes (see the different types of numerical methods in Subsection 1.1.5). This grouping can be done by taking into account the distance between them, their relative velocity... but this process is expensive and might lead to sub-optimal groupings due to the assumptions made.

Clustering algorithms could be used to find optimal groupings at different moments of the simulation.

Graph Neural Networks

A type of neural networks that has not yet been mentioned is Graph Neural Networks (GNNs). It is specialized for tasks in which the inputs are graphs. In these networks, the graph nodes exchange information with their neighbors. It is specially well-suited for chemical networks where atoms form the edges of the network. Analogously, these networks could be used for the simulation of a system of N-bodies, where each node

represents one of the particles in the system.

1.3 Thesis summary

This thesis compiles the experiments performed on the use of Machine Learning techniques for the simulation of N-body systems. For this work, we have simulated different astrophysical systems and performed the training of machine learning methods on two different categories: physics-aware Neural Networks and Reinforcement Learning.

In Chapter 2, we take the example of a planetary system with a large number of small bodies and apply neural networks to replace parts of the integration with the goal of speeding up the simulations. We use a specific integrator - Wisdom Holmann integrator- that separates the contributions to the acceleration of a body in two terms: the ones by the central body and the perturbing ones by the other bodies in the system. We adapt two types of neural networks to this case: a deep neural network and a Hamiltonian neural network. We identify the challenges of using neural networks in a complex case in astrophysics. While the standard neural networks are easy to set up, they do not lead to results that adhere to the laws of physics. They also do not extrapolate well to unseen data. We observed that small prediction errors accumulated and grew, leading to a drift in the energy error. In contrast, Hamiltonian Neural Networks managed to find solutions that adhere better to the physics and are capable of extrapolating for longer periods of time. However, there were also major challenges. For example, we observe that the large orders of magnitude difference in the masses of the bodies makes the application of Hamiltonian networks extremely challenging. As the calculation of the accelerations is done using automatic differentiation, the large differences in mass created equally large differences in the gradients. As the physics are embedded in the architecture, normalization of the inputs is not possible without breaking the physical relations. These facts contributed to making the training problematic. Additionally, with both networks, we encounter other challenges due to the chaotic nature of the problem. In a chaotic system, prediction errors by the neural networks accumulate and grow exponentially, leading to unphysical solutions. To address this problem, we create a hybrid method that evaluates the network prediction at each step. If the prediction does not satisfy a requirement of accuracy, the calculation is repeated using the analytical equations. Thanks to this method, the integrator can make use of neural networks to speed up the simulation while ensuring robustness in its accuracy.

With the results in Chapter 2, it became clear that the simulation of the N-body problem required more sophisticated machine learning methods to ensure the adherence to the conservation laws. Therefore, in Chapter 3, together with Philipp Horn, we compare different implementations of neural networks with structure-preserving architectures and develop a new type of neural network that represents a generalization of many of those types. SRNNs, SympNets, and HénonNets are three common cases of neural networks that implement hard physical constraints into their network structure. Those networks can be thought of as specific cases of a new type of structure-preserving neural networks: Generalized Hamiltonian Neural Networks (GHNNs). The performance achieved with those neural networks in solving Hamiltonian systems is compared for different experiments, of which the 3-body problem is shown in Chapter 3. The comparison between these

networks is not straight-forward as those rely on different implementations. Therefore, their hyperparameters are chosen for their prediction time to be comparable. The results of the experiments show that the performance of the networks depends on the experiment. For simple cases such as the single and double pendulum, networks that included a larger number of simple updates dominated over those with fewer updates. In contrast, in the 3-body problem, networks with more complex updates (such as Deep HénonNets) achieved a better performance. In any case, MultiLayer Perceptrons, a simple type of physics-unaware neural network, showed the worst extrapolation capabilities compared to the physics-aware ones. It is concluded that the added symplecticity in the structure-preserving neural networks resulted in better performances, both inside and outside the training data.

After studying the advantages and disadvantages of replacing parts of the integration with neural networks, we moved our focus to the use of Machine Learning techniques to choose simulation parameters. When setting up a simulation, lack of expertise makes it common to use the default parameters of the integrator. This can lead to the solutions not being accurate enough or being too computationally expensive. Thus, in Chapter 4 we explore the idea of using Reinforcement Learning techniques that will eliminate the need for expert knowledge by choosing an important simulation parameter for us: the time-step parameter. We apply this method to the chaotic 3-body problem. In this case, there are moments in which the three bodies are close to each other, and the interactions will heavily determine the outcome of the simulation. In these moments, we want the integrator to choose smaller time-step sizes to capture those interactions in detail. In other moments in which the bodies are far from each other, we want the time-step size to increase to save computation time. The time-step parameter directly scales the time-step size chosen at each moment. By allowing this parameter to change dynamically to adapt to the needs of the simulation, we obtain an optimal choice that balances accuracy and computational effort at each time step. We explore the extrapolation of this trained algorithm to other integrators and determine that it can be used without retraining for similar algorithms. Additionally, the method setup can be easily extrapolated without major development changes.

In order to prove the generalization capabilities of the method created in Chapter 4, in Chapter 5 we apply a similar algorithm to a more complex astrophysics problem: the evolution of a star cluster in which some stars have planetary systems orbiting around them. In this case, we need to use different integrators for the star cluster and the planetary system to adapt to the orders of magnitude difference in their scales. Then, those two different subsystems are linked using the `Bridge` method from `AMUSE`. The interaction time between both integrators is a fixed parameter that has to be chosen manually before the simulation. We denominate it the bridge time-step size. Instead of choosing a value and keeping it fixed throughout the simulation, we apply our RL algorithm to allow it to change dynamically to find the optimal choice that balances accuracy and computation time. We find that our algorithm outperforms all of the current options and can adapt to different initializations. Due to the chaotic nature of our method, finding robust baselines proves to be problematic. Therefore, we base our comparisons on the energy error incurred by the simulations. Knowing that our system is highly chaotic, we want to create a method that is robust against suboptimal choices of the RL algorithm. Therefore, we create a hybrid method, similar to the one in Chapter 2, that evaluates the prediction and

reduces the time-step size if considered inadequate. We find that this method leads to improvements of even orders of magnitude in the energy error without a major increase in computational effort.

1.3.1 Future work

We have identified in Subsection 1.2.3 additional ideas of implementations of ML methods to the gravitational N-body problem that we considered potentially interesting. Additionally, despite the promising results achieved with the new methods shown in this thesis, throughout our work, we have found some common challenges that should be addressed in the future. Some of those were common to many of the methods tested.

First of all, we found that it is a common practice to use the state of the system as an input to neural networks. In the case of physics-aware neural networks, this is even a strict requirement. However, this leads to a critical generalization problem: the number of inputs changes with the number of particles present in the system. A network trained for a system of 3 particles cannot be directly applied to one with 4 or more particles present. Finding a solution to this problem is not straightforward, specially as the evolution of physics-aware neural networks relies on hard constraints imposed on their architecture. This problem becomes specially serious for systems with a large number of particles, such as star clusters. In this case, it is impractical to train a neural network for each possible number of bodies. We propose a potential solution in Subsection 1.2.3 that could be further studied. In Chapter 5, we find a different input representation that works for that specific problem and is independent of the number of bodies in the system.

Continuing with the choice of inputs, Cartesian coordinates are the preferred choice in most studies involving the evolution of celestial bodies with neural networks. This option might not always be optimal. In certain cases, Keplerian coordinates are the preferred choice for the interpretability of a problem. However, those result problematic if used as inputs to the neural networks due to their mix between linear variables of different orders of magnitude and periodic ones. Currently, those are the main two choices to represent the problem and provide the ML algorithm with the required information of the system. Finding different system representations that are specially well suited for their combination with neural networks should be a priority for those studying this problem.

Moving to a different challenge that has been present in most of this work, we find that chaos is a major obstacle to obtaining accurate solutions. Prediction errors in the case of neural networks and suboptimal choices of actions in the case of Reinforcement Learning are specially problematic in chaotic systems as those errors accumulate and grow exponentially in time. The current performance of these algorithms does not generally allow for long-term simulations (with the exception of the method shown in Chapter 5). We create a hybrid algorithm to increase the robustness of the methods in Chapters 2 and 5, but in order to achieve truly long-term integrations within a reasonable energy error, the performance capabilities of the ML methods need to drastically improve. With the current popularity of these topics and the potential shown in this, and many other studies, it is reasonable to believe that sooner rather than later these methods will reach the required precisions to be used in chaotic problems.

Regarding problem-specific challenges, structure-preserving neural networks are still not specifically designed for astrophysics problems in which the masses of the bodies

range orders of magnitude. Normalization is a common technique that can minimize this problem, but it is in many cases incompatible with the physics-based structure of those types of networks. By adding hard constraints into the networks, it becomes harder to adapt to certain complex problems. These should be taken into account when designing new types of neural networks for physical problems.

Finally, there is work to be done to continue the projects on this thesis. The methods explored are applied to simple cases found in astrophysics. Extending their capabilities and implementations to a broader range of applications is the most interesting future direction. Including the ML methods as a part of the frameworks used for astrophysics simulations can help to better understand the benefits and limitations of using Machine Learning in computational Astrophysics on a regular basis.

1.3.2 Computational tools

Most of the code has been created for the purpose of this thesis and can be found for each individual chapter. The astrophysics simulations have been developed using `Python`. I have made extensive use of libraries like `AMUSE` (Zwart et al. (2013)) and `ABIE` (Roa et al. (2020)) for the initialization of the planetary systems, triples, and star clusters, and for their integration in time.

For the Machine Learning algorithms, I have made use of `TensorFlow` (Abadi et al. (2015)), `PyTorch` (Paszke et al. (2019)), and additional machine learning packages such as `Scikit Learn` (Pedregosa et al. (2011)), `Gym` (Brockman et al. (2016a)), and `pyDOE`.

All simulations and experiments for Chapters 2, 4, and 5 have been run on an AMD Ryzen 9 5900hs computer, and the computation times incurred can be found for each chapter.

2 A HYBRID APPROACH FOR SOLVING THE GRAVITATIONAL N-BODY PROBLEM WITH ARTIFICIAL NEURAL NETWORKS

Work published in Veronica Saz Ulibarrena, Philipp Horn, Simon Portegies Zwart, Elena Sellentin, Barry Koren, Maxwell X. Cai , 2024, *Journal of Computational Physics*, Volume 496, 112596. Reprinted here in its entirety.

ABSTRACT

Simulating the evolution of the gravitational N-body problem becomes extremely computationally expensive as N increases since the problem complexity scales quadratically with the number of bodies. In order to alleviate this problem, we study the use of Artificial Neural Networks (ANNs) to replace expensive parts of the integration of planetary systems. Neural networks that include physical knowledge have rapidly grown in popularity in the last few years, although few attempts have been made to use them to speed up the simulation of the motion of celestial bodies. For this purpose, we study the advantages and limitations of using Hamiltonian Neural Networks to replace computationally expensive parts of the numerical simulation of planetary systems, focusing on realistic configurations found in astrophysics. We compare the results of the numerical integration of a planetary system with asteroids with those obtained by a Hamiltonian Neural Network and a conventional Deep Neural Network, with special attention to understanding the challenges of this specific problem. Due to the non-linear nature of the gravitational equations of motion, errors in the integration propagate, which may lead to divergence from the reference solution. To increase the robustness of a method that uses neural networks, we propose a hybrid integrator that evaluates the prediction of the network and replaces it with the numerical solution if considered inaccurate. Hamiltonian Neural Networks can make predictions that resemble the behavior of symplectic integrators but are challenging to train and in our case fail when the inputs differ $\sim$7 orders of magnitude. In contrast, Deep Neural Networks are easy to train but fail to conserve energy, leading to fast divergence from the reference solution. The hybrid integrator designed to include the

neural networks increases the reliability of the method and prevents large energy errors without increasing the computing cost significantly. For the problem at hand, the use of neural networks results in faster simulations when the number of asteroids is $\gtrsim$70.

2.1 Introduction

Planetary systems are a special case of the gravitational N-body problem in which a massive central star is orbited by multiple minor bodies, which include planets and asteroids among others. To model the evolution of planetary systems, it is necessary to know the gravitational interaction between the different bodies, which can be calculated using the equations derived by Newton Newton (1999). Unlike the calculation of the gravitational force, the equations of motion can only be solved analytically for two bodies using the relations derived by Kepler in 1609 Kepler (2015). This means that when the system consists of three or more bodies, the equations need to be solved numerically with what we call the integrator. Hermite Makino (1991) and Verlet Verlet (1967) integrators are frequently used for solving the general N-body problem, whereas others such as the Wisdom-Holman integrator Wisdom & Holman (1991) have been developed for the specific case of planetary systems.

Currently, the study of the evolution of N-body systems is limited by the large computational resources required to obtain an accurate[1] solution Greengard (1990); Almojel (2000). Newton's equation of gravitation implies that the computational complexity of the problem scales with N^2. As a consequence, for multiple applications in astrophysics such as the evolution of globular clusters or asteroids around a star Richardson et al. (2009), the large number of bodies in the system is one of the main reasons for the high computational cost.

Machine Learning (ML) is a tool with the potential to ameliorate this problem Chevallier et al. (1998). Although the applications of ML, or more precisely Artificial Neural Networks (ANNs), are scarce for the gravitational N-body problem Tamayo et al. (2016); Lalande & Trani (2022), ANNs have recently demonstrated their potential in other fields Doupe et al. (2019); Basuchoudhary et al. (2017); Mansfield et al. (2020). We study the efficiency of neural networks to replace computationally expensive parts of the integration of N-body systems for astrophysics applications.

Some studies have been carried out to apply ANNs to the two- and three-body gravitational problems to predict the future state of the system. For example, Breen et al. Breen et al. (2020a) in 2020 designed a Deep Neural Network (DNN) to replace the integration of the chaotic three-body problem. Their setup consists of three coplanar bodies of equal mass, with a zero initial velocity, which state is propagated in time using the arbitrary precise Brutus integrator developed by Boekholt and Portegies Zwart, 2015 Boekholt & Portegies Zwart (2015). The ANN receives as inputs the state of the particles at initial time t_0 and the simulation time t. In this simplified approach, the network is able to capture the complex motions of the three bodies, at a fraction of the computational expense.

Since the introduction of Physics-Informed Neural Networks (PINNs) in 2019 Raissi et al. (2019b), the popularity of neural networks with physics knowledge included has grown rapidly Lu et al. (2021); Jin et al. (2020b). The claim is that the introduction of

[1] Accurate refers to solutions with low energy error.

physical properties into the neural network allows for better predictions, better extrapolation capabilities, and less training data. So far, PINNs have not been applied to astrophysics problems. Following the idea of introducing physical knowledge into the neural network, Greydanus et al. Greydanus et al. (2019a) developed in 2019 Hamiltonian Neural Networks (HNNs) to address Hamiltonian mechanics within the network's architecture. To study the performance of their network, they use the gravitational two- and three-body problems as test cases. For the two-body problem, Greydanus et al. found that the HNN can predict the trajectories of the particles better than a baseline DNN. However, for the three-body problem, both networks fail to predict the trajectories. An alternative for HNNs was developed by Chen and Tao in 2021 Chen & Tao (2021), denominated as Generating Function Neural Networks (GFNNs). They tested this approach on the two-body problem with similar inputs as in Greydanus et al. Greydanus et al. (2019a). The comparison with other types of neural networks such as HNNs and SympNets Jin et al. (2020b) shows that GFNNs outperform the other methods for this particular test case. Although the results of Greydanus et al. Greydanus et al. (2019a), and Chen and Tao Chen & Tao (2021) are promising, both references take the two- and three-body problems as test cases to demonstrate the performance of their neural networks. It is not yet certain that the introduction of physics into the neural network represents an advantage for more complicated problem configurations. For that reason, we study the advantages and disadvantages of HNNs when applied to more realistic astrophysics problems, in particular, the orbital evolution of celestial bodies.

We study the use of neural networks for the integration of planetary systems formed by two planets and up to 2,000 asteroids. Due to the popularity of physics-aware neural networks, we compare the results of direct numerical integration with the predictions of a network that includes physical knowledge (HNN) and a conventional Deep Neural Network (DNN). In Subsection 2.2.3, we discuss the setup of a hybrid integrator that uses the neural network but switches to direct numerical integration when the former fails to produce sufficiently reliable answers. This method is faster than the classical integration and more accurate than the neural network. In Section 2.3, we discuss the hyperparameter selection and the training results of the neural networks. In Subsection 2.4.2, we find the improvement in performance by the neural networks in the form of computation time as a function of the number of asteroids in the system, and in Subsection 2.4.3 we show the results of integrating a planetary system. The code is publicly available at `https://github.com/veronicasaz/PlanetarySystem_HNN`.

2.2 Methodology

2.2.1 Numerical integration

We consider a system of N point masses interacting only via their Newtonian gravitational force. The gravitational force exerted on a body i, can be written as a function of mass (m), position vector ($\vec{q}$), and the universal gravitational constant (G) as

$$m_i \frac{d^2\vec{q}_i}{dt^2} = \sum_{j=0,\, j\neq i}^{N-1} G \frac{m_i m_j}{||\vec{q}_{ij}||^3}\vec{q}_{ij}, \qquad \vec{q}_{ij} = \vec{q}_j - \vec{q}_i, \tag{2.1}$$

where the indices i and j denote the celestial bodies.

Knowing the acceleration vector, the state of the system can be evolved in time using an integrator. Wisdom and Holman in 1991 Wisdom & Holman (1991) proposed a symplectic integrator for systems in which one body is much more massive than the others. In our case, we assume that the smaller bodies orbit this massive one and the barycenter of the system is located approximately at the center of the massive body. The other bodies orbit the barycenter in almost Keplerian trajectories.

The Hamiltonian of the system is given by

$$\mathcal{H} = \sum_{i=0}^{N-1} \frac{||\vec{p}_i||^2}{2m_i} - G \sum_{i=0}^{N-2} m_i \sum_{j=i+1}^{N-1} \frac{m_j}{||\vec{q}_j - \vec{q}_i||}, \tag{2.2}$$

where $\vec{p}$ represents the linear momentum vector.

For planetary systems, Equation (2.2) can be split into two parts. Due to the assumption of the Sun being at the barycenter, i = 0 is excluded from the following equations. The first one, the Keplerian part,

$$\mathcal{H}_{\text{Kepler}} = \sum_{i=1}^{N-1} \frac{||\vec{p}_i||^2}{2m_i} - Gm_0 \sum_{j=1}^{N-1} \frac{m_j}{||\vec{q}_j||}, \tag{2.3}$$

contains the terms related to the kinetic energy of the bodies and the potential energy due to the central body (body 0). The second part called interactive part,

$$\mathcal{H}_{\text{inter}} = -G \sum_{i=1}^{N-2} m_i \sum_{j=i+1}^{N-1} \frac{m_j}{||\vec{q}_j - \vec{q}_i||}, \tag{2.4}$$

contains the terms with the potential energy due to the mutual interaction between the orbiting bodies.

The Wisdom-Holman (WH) integrator first propagates the trajectory of the orbiting bodies without taking their mutual interaction into account by performing a Keplerian propagation around the central body. After that, the perturbing acceleration is calculated and converted to a correction of the velocity.

Although Equations (2.3) and (2.4) are expressed in heliocentric coordinates for clarity, WH's integrator uses Jacobian coordinates for parts of its integration, as explained in Wisdom and Holman Wisdom & Holman (1991).

The computing time of the Keplerian propagation scales linearly with the number of bodies (N) as seen in Equation (2.3). In contrast, the interactive part (Equation (2.4)) scales quadratically with the number of bodies. Therefore, it is interesting to find methods to speed up the latter. We use ANNs to replace the interactive part to speed up the calculation of the mutual perturbations.

2.2.2 Neural Network surrogates

In examples such as Breen et al. Breen et al. (2020a) and Greydanus et al. Greydanus et al. (2019a), a neural network is used to replace the integrator. However, this approach

falls short for many astrophysics applications. For example, for the case of a planetary system, the force exerted by the central body is orders of magnitude larger than the mutual forces exerted by the orbiting bodies. If a neural network is used to predict the future state of the system, it will fail to capture the smaller contributions of the planets. We propose a method in which the neural network is integrated into the numerical integration without losing information about the perturbations. We do so by calculating the Keplerian Hamiltonian $\mathcal{H}_{\text{Kepler}}$ analytically and the interactive Hamiltonian using a neural network $\mathcal{H}_{\text{inter}}$.

For systems in which energy is conserved, Hamiltonian Neural Networks (HNNs) constitute an attractive choice since the Hamiltonian of the system can be input as a physical constraint into its architecture. We therefore use HNNs to predict the interactive part of Equation (2.2) similarly to the Neural Interacting Hamiltonian (NIH) designed by Cai et al. Cai et al. (2021a). We study the advantages and disadvantages of HNNs by comparing them to the numerical integration, which we consider the baseline, and to a conventional Deep Neural Network (DNN).

An HNN Greydanus et al. (2019a) receives as inputs the position and linear momentum of all the bodies in the system and outputs the Hamiltonian of the system. In Figure 2.1 we show a comparison of an uninformed neural network (DNN) and a HNN.

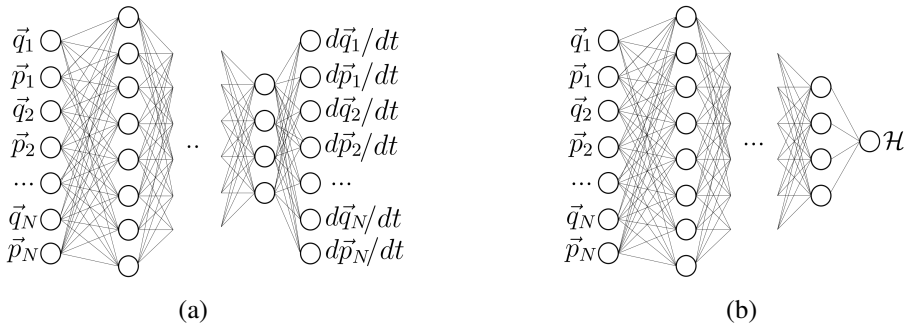


Figure 2.1: Schematic of a a) Deep Neural Network and a b) Hamiltonian Neural Network that predict the derivatives of the inputs with respect to time and the Hamiltonian of the system, respectively. The inputs for both are the position and linear momentum of all objects in the system.

With the output of the HNN and automatic differentiation, the derivatives of the inputs are calculated using Hamilton's equations:

$$-\frac{\partial \mathcal{H}}{\partial \vec{q}} = \frac{d\vec{p}}{dt}, \qquad \frac{\partial \mathcal{H}}{\partial \vec{p}} = \frac{d\vec{q}}{dt}. \tag{2.5}$$

The derivatives are then used to compute the loss function during the training of the network.

Unlike in Greydanus et al. Greydanus et al. (2019a), we use the neural networks for the calculation of the interactive Hamiltonian as expressed in Equation (2.4). This Hamiltonian is only a function of the masses and positions of the different bodies, and the universal gravitational constant. This means that the neural networks from Figure 2.1 can be simplified by eliminating the linear momentum from the inputs. Since the acceleration

requires knowing the masses of the system, the inputs then become:

$$X = [m_1, \vec{q}_1, m_2, \vec{q}_2, ..., m_N, \vec{q}_N]. \tag{2.6}$$

Similarly, the outputs of the DNN are now reduced to the derivatives of the linear momentum with respect to time. By doing this, we achieve a substantial reduction in the number of parameters of the network. We will explain whether the symplectic structure of the integrator is conserved when using neural networks for the calculation of the interactive Hamiltonian in Subsection 2.2.4.

Taking into account this modification of the set of inputs and outputs, the loss function $\mathcal{L}$ for the HNN is the difference between the acceleration calculated using Newton's equation and the one obtained from differentiating the output of the HNN using Equation (2.5):

$$\mathcal{L}_{\text{HNN}}(\theta) = \frac{1}{M}\sum_{i=1}^{M}\left(-\frac{\partial \mathcal{H}_{\text{pred}}}{\partial \vec{q}} - \frac{d\vec{p}}{dt}\right)^2. \tag{2.7}$$

In Equation (2.7), θ represents the trainable parameters of the network, $\mathcal{H}_{pred}$ is the output of the HNN, and the gradients of $\mathcal{H}$ are obtained using automatic differentiation. M is the number of samples for which the loss is being evaluated.

For the DNN, the inputs are the same as for the HNN and the derivatives of the inputs with respect to time are the outputs of the neural network. Therefore, the loss function is written as:

$$\mathcal{L}_{\text{DNN}}(\theta) = \frac{1}{M}\sum_{i=1}^{M}\left(\left[\frac{d\vec{p}}{dt}\right]_{\text{pred}} - \frac{d\vec{p}}{dt}\right)^2. \tag{2.8}$$

2.2.3 Hybrid numerical method

The use of neural networks to replace parts of the integration raises several challenges. Firstly, neural networks cannot be expected to be as accurate as the numerical calculation: the use of ANNs implies a loss in accuracy with the goal of improving computing speed. Secondly, since integration is a repetitive process in which the output of one time step is used as the input for the next one, errors propagate in time. In non-linear systems, this may quickly lead to unphysical solutions. In previous research trying to solve the N-body problem using neural networks, it is common to propagate over short time scales. This implies that the accumulation of errors is not relevant, but does not constitute a realistic case for astrophysics problems. To address this problem, we develop a hybrid method in which the prediction of the neural network is evaluated and replaced by the numerical solution if considered insufficiently accurate.

Evaluating the accuracy of the prediction is not straight-forward since we want to avoid using Newton's equation. Therefore, we use as a measurement of accuracy the fact that accelerations should be fairly smooth in time. We evaluate the prediction of the network by comparing it to the prediction of the previous time step. Since the perturbations are expected to be rather smooth, we assume that a large difference between the acceleration predicted by the network at time $t_0 + \Delta t$ and the acceleration at t_0 is an indication of

either a poor prediction or a region with quick changes in the acceleration. In both cases, it is beneficial to calculate those steps numerically instead of relying on the neural network. Although accelerations are expected to vary smoothly in time, by using a numerical time integrator we need to account for the discretization error when setting the tolerance R for this smoothness criterion. From now on, we use the term "flag" when the prediction of the network is not accepted. We calculate the acceleration $\vec{a}^{\,(t)}$ at time $t = t_0 + \Delta t$ by numerical integration if

$$\frac{||\vec{a}^{\,(t_0)} - \vec{a}^{\,(t)}||}{||\vec{a}^{\,(t_0)}|| + \varepsilon} > R. \tag{2.9}$$

This criterion represents the relative difference between the previous acceleration and the current one. The addition of $\varepsilon = 1 \times 10^{-11}$ prevents the denominator from becoming zero. We adopt $R = 0.3$ to achieve an accurate reproduction of the trajectory whereas higher values result in larger energy errors, as we show in 2.B. The value of R should be chosen according to the specifications of the problem at hand. If computational speed is more important than accuracy, higher values of R could be chosen, whereas if the focus is on accuracy, R should be smaller. A value of 0.3 represents a strict case in which ensuring accuracy is considered more important than achieving a low computational cost.

In Figure 2.2 we show the schematic diagram of the hybrid WH integrator. At time t_0, the state of each body is propagated a time step Δt, assuming that the particle is on a Keplerian trajectory. Afterward, the neural network (f_{NN}) calculates the perturbing accelerations for the given inputs (X). This prediction is evaluated and if considered insufficiently smooth according to the criterion defined in Equation (2.9), the accelerations are recalculated analytically. The perturbing accelerations are then converted into corrections in the velocity and the new state of the system is subsequently used as the starting point for the next time step.

2.2.4 Symplecticity of the integrator

The original Wisdom-Holman integrator is a symplectic integrator Wisdom & Holman (1991). Using a symplectic integrator is essential for long-term stability and energy conservation of Hamiltonian systems Hairer et al. (2006). Some attempts have been made to conserve the symplectic structure of the integrator when using neural networks, such as in Zhu et al. Zhu et al. (2020). When using neural networks as surrogates for the calculation of the interactive part in the new hybrid integrator, it is beneficial to have the same symplectic structure as the original Wisdom-Holman integrator.

The first part in the hybrid Wisdom-Holman integrator is the Keplerian propagation, which is the flow map of a Hamiltonian system and therefore a symplectic map. In the second part, the linear momentum vector is updated with accelerations either calculated by an ANN or using Newton's equation for the interactive part. In this step, the positions are always kept unchanged. If this update forms a symplectic map, the whole hybrid integrator becomes symplectic since concatenations of symplectic maps are again symplectic.

The update of the linear momentum vector only depends on the positions and not on the current momenta. So, the left-hand side of the symplectic condition

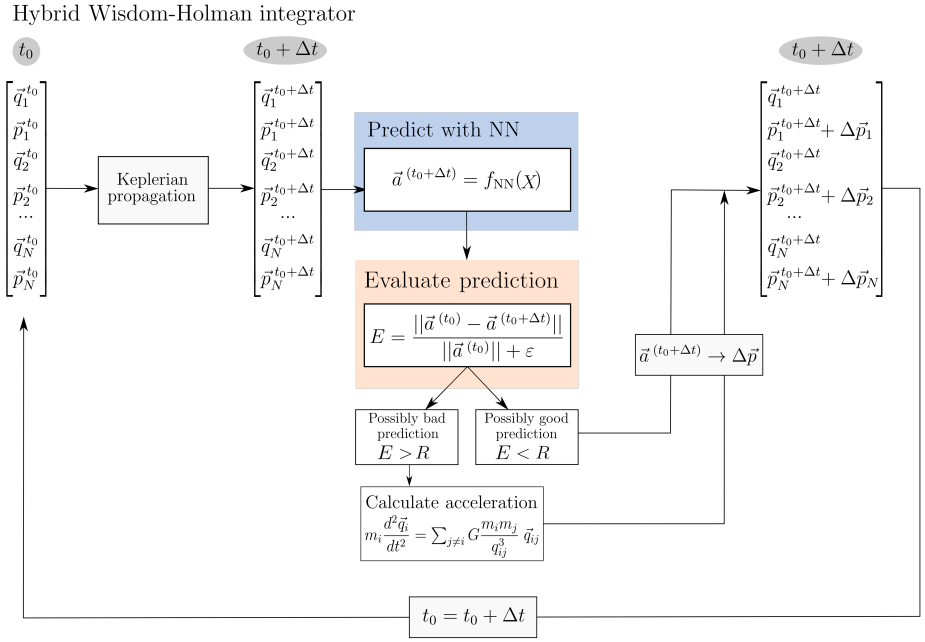


Figure 2.2: Schematic of the hybrid Wisdom-Holman integration. The state of the system defined by $\vec{p}$ and $\vec{q}$ is propagated as a Keplerian trajectory. Then the Neural Network predicts the mutual perturbation between bodies. If the prediction is insufficiently smooth, the accelerations are calculated numerically using Equation (2.1). Finally, these accelerations are added as a change in linear momentum ($\Delta\vec{p}$). This process is repeated in the next time steps.

$$J_{\vec{f}}^T \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} J_{\vec{f}} = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \tag{2.10}$$

of the Jacobian matrix $J_{\vec{f}}$ of a map $\vec{f}(\vec{p},\vec{q})$ simplifies to

$$J_{\vec{f}}^T \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} J_{\vec{f}} = \begin{pmatrix} I & 0 \\ \Delta t \left(\frac{\partial \vec{a}}{\partial \vec{q}}\right)^T & I \end{pmatrix} \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \begin{pmatrix} I & \Delta t \left(\frac{\partial \vec{a}}{\partial \vec{q}}\right) \\ 0 & I \end{pmatrix} \tag{2.11}$$

$$= \begin{pmatrix} I & 0 \\ \Delta t \left(\frac{\partial \vec{a}}{\partial \vec{q}}\right)^T & I \end{pmatrix} \begin{pmatrix} 0 & I \\ -I & -\Delta t \left(\frac{\partial \vec{a}}{\partial \vec{q}}\right) \end{pmatrix} \tag{2.12}$$

$$= \begin{pmatrix} 0 & I \\ -I & \Delta t \left(\left(\frac{\partial \vec{a}}{\partial \vec{q}}\right)^T - \left(\frac{\partial \vec{a}}{\partial \vec{q}}\right) \right) \end{pmatrix}. \tag{2.13}$$

This implies that the Jacobian matrix of the calculated accelerations $\left(\frac{\partial \vec{a}}{\partial \vec{q}}\right)$ has to be symmetric.

If the accelerations are calculated using Newton's equation or using an HNN, they are the gradient of a scalar function, the Hamiltonian. This means that the Jacobian matrix of the accelerations is the Hessian matrix of the Hamiltonian, which is symmetric for continuous second derivatives. However, if a DNN is used in the hybrid integrator, no such statement can be made and the Jacobian matrix of the predicted accelerations can be non-symmetric.

Therefore, we can expect the energy-preserving characteristics of symplectic integrators to be present when using HNNs within the WH integrator but not when including DNNs. This result is investigated numerically in Subsection 2.4.3.

2.2.5 Problem setup

We study two cases: the first one, Jupiter and Saturn orbiting the Sun, and the second one with a large number of asteroids added to the first case, as illustrated in Figure 2.3.

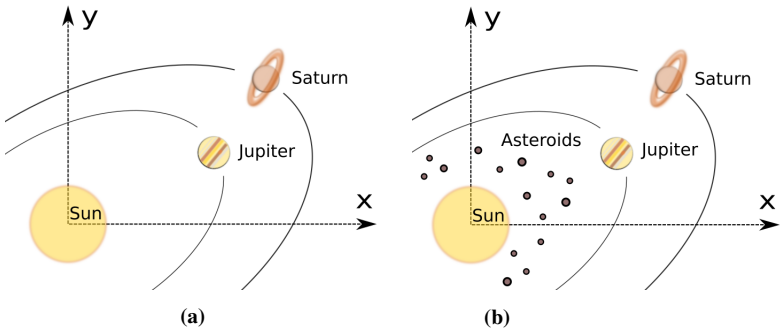


Figure 2.3: Schematic of the problem setup. (a) First study case with the Sun, Jupiter, and Saturn. (b) Second study case with the Sun, Jupiter, Saturn, and asteroids located within Jupiter's orbit.

For the first case in which only the Sun, Jupiter, and Saturn are studied (from now on referred to as SJS), the Hamiltonian of the system is given by Equation (2.2) for $N = 3$ with the Sun as $i = 0$, Jupiter $i = 1$, and Saturn $i = 2$. The interactive part of the Hamiltonian corresponds to the interaction between Jupiter (J) and Saturn (S):

$$\mathcal{H}_{\text{inter}} = -G\frac{m_J m_S}{||\vec{q}_J - \vec{q}_S||}. \tag{2.14}$$

In this case, only one operation suffices to calculate the interactive part, and as a consequence, the use of ANNs will lead to a deceleration of the calculation. However, this setup constitutes an interesting study case. We set up the network for the inputs (X) to be the masses and positions of the two bodies and the output to be the Hamiltonian, as explained in Subsection 2.2.2. Therefore, for the SJS case, the inputs are:

$$X_{\text{SJS}} = [m_J, \vec{q}_J, m_S, \vec{q}_S]. \tag{2.15}$$

In the second case (to which we refer as SJSa), we add N_a asteroids in orbit around the massive central body. The Hamiltonian can again be calculated for the star, the two

planets, and the asteroids with $N = 3 + N_a$. For example, when $N_a = 2$, the interactive Hamiltonian can be expressed as follows:

$$\mathcal{H}_{inter} = -G\frac{m_J m_S}{q_{JS}} - G\frac{m_1 m_J}{q_{1J}} - G\frac{m_1 m_S}{q_{1S}} - G\frac{m_2 m_J}{q_{2J}} - G\frac{m_2 m_S}{q_{2S}} - G\frac{m_1 m_2}{q_{12}}, \quad (2.16)$$

with q representing the magnitude of $\vec{q}$. Because asteroids are orders of magnitude less massive than the planets, it can be safely assumed that the mutual gravitational interaction between asteroids is negligible, and we therefore neglect the last term in Equation (2.16). We also assume that the effect of the asteroids on Jupiter and Saturn is negligible. In contrast, the effect of the planets on the asteroids cannot be neglected. To set up a neural network that predicts the perturbations on the asteroids due to the planets, we separate this interactive Hamiltonian for each of the asteroids. For asteroids 1 and 2, their interactive Hamiltonian is defined as:

$$\mathcal{H}_1 = -G\frac{m_J m_S}{q_{JS}} - G\frac{m_1 m_J}{q_{1J}} - G\frac{m_1 m_S}{q_{1S}}, \quad (2.17)$$

$$\mathcal{H}_2 = -G\frac{m_J m_S}{q_{JS}} - G\frac{m_2 m_J}{q_{2J}} - G\frac{m_2 m_S}{q_{2S}}. \quad (2.18)$$

We now set up the network such that the position and mass of each of the two asteroids correspond to one set of inputs. Therefore:

$$X_{\text{SJSa}} = [m_J, \vec{q}_J, m_S, \vec{q}_S, m_a, \vec{q}_a], \quad (2.19)$$

where the subindex a represents one of the N_a asteroids. This choice of inputs allows the size of the neural network to be independent of the number of asteroids in the system, which implies that the same neural network can be used for any number of asteroids without retraining.

2.3 Neural network results

In this section, we explain the creation of the training dataset, the choice of hyperparameters, and the training results for the Hamiltonian Neural Network and the Deep Neural Network.

2.3.1 Dataset

We generate training and test datasets for each of the two cases: SJS and SJSa. The ranges of values can be found in Table 2.1 of 2.A. From these, the initial conditions are chosen using Latin hypercube sampling Loh (1996) and the simulations are run until the end time is reached. At each time step, the state of the system is saved as a training sample. Then, we verify if the dataset created covers the entire search space, i.e., if there are samples in the full range of true anomaly $[0° - 360°)$, which is displayed in Figure 2.4.

With the time step and the end time in 2.A, the number of training samples is 3,000,000. We randomly choose a fraction of these for the training. On an AMD Ryzen 9 5900hs,

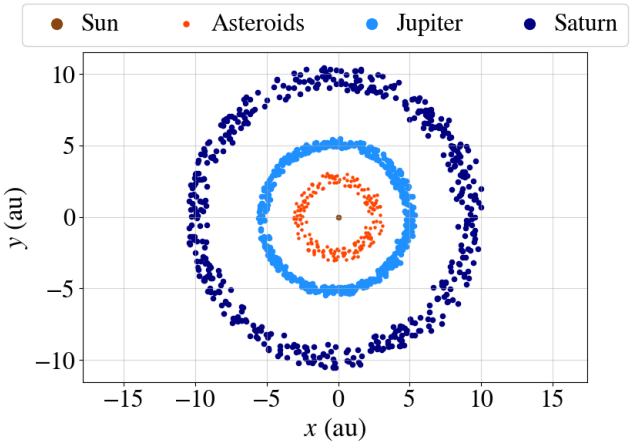


Figure 2.4: Distribution of x and y positions of the Sun, Jupiter, Saturn, and the asteroids in the training dataset.

it takes ∼80 min to generate this dataset. All experiments utilize this same computer architecture.

The accelerations of the planets and asteroids differ by orders of magnitude, which means that normalization of the training data is essential to train the network. However, since HNNs have physics embedded into their architecture, we cannot normalize the inputs or outputs independently without breaking the physical constraints. For example, re-scaling the inputs between 0 and 1 implies that the relation between different inputs does not remain constant. The distributions of inputs and outputs have been included in Figure 2.14 and Figure 2.15, respectively.

2.3.2 Architecture and training parameters

In order to make a fair comparison between the DNN and the HNN, the settings chosen will be common for both of them unless otherwise stated. Each of the two cases studied (SJS and SJSa) requires different neural network hyperparameters. For the SJS case, we adopt a Mean Squared Error (MSE) loss function as indicated in Equation (2.7) for the HNN and Equation (2.8) for the DNN. For SJSa, the accelerations of the different bodies range multiple orders of magnitude, and therefore we implement a weighted MSE for the loss function, i.e, the error in the predicted acceleration of each body is weighted. The weights are applied to the losses defined in Equation(2.7) and Equation (2.8) as:

$$\mathcal{L}_{\text{NN}}(\theta) = W_1\ \mathcal{L}_a(\theta) + W_2\ \mathcal{L}_S(\theta) + W_3\ \mathcal{L}_J(\theta), \tag{2.20}$$

where $\mathcal{L}_a$, $\mathcal{L}_S$, and $\mathcal{L}_J$ represent the MSE loss for the accelerations of the asteroids, Saturn, and Jupiter, respectively. We empirically find that weights of $W_1 = 100$, $W_2 = 10$, and $W_3 = 1$ produce the best results as these weight values relate to differences in orders of magnitude of the accelerations of the bodies. These weights are only necessary due to the impossibility of normalizing the inputs and outputs without breaking the physics

constraints of the HNN. Although normalization is possible with the DNN, we have used the weighted loss function instead to get a fair comparison with the HNN.

For SJS, no hyperparameter optimization is carried out, but the architecture is chosen manually instead. Both the DNN and the HNN have three layers, 200 neurons in the first hidden layer and each hidden layer has 0.7 times the number of neurons of the previous one. The learning rate follows an exponential decay, with an initial learning rate of 0.01, a decay of 0.9, and 2×10^5 steps. We use 150,000 samples with a proportion of 90/10 for training and validation datasets, and 10,000 samples for the test dataset.

For SJSa, the training of the HNN is not straightforward. To find a suitable combination of parameters, we perform a hyperparameter optimization where the variables are the number of training samples, number of layers, number of neurons per layer, and the learning rate parameters. We use a randomized grid search to explore different combinations of those parameters and train $\sim$30 networks for 200 epochs. The results are presented in Figure 2.5, where each simulation is plotted with the training loss along the x-axis and the validation loss along the y-axis. The figure indicates that regardless of the choice of parameters, the training and validation loss cannot be improved simultaneously to achieve the desired accuracy during testing. Among the best solutions, we choose the network architecture with three layers, 300 neurons per layer, and we use 250,000 samples for the training dataset. The test dataset is chosen to consist of 10,000 samples. The learning rate is chosen to follow an exponential decay schedule with an initial learning rate of 10^{-3}, 800,000 steps, and a decay rate of 0.9. The same parameters are used for the DNN.

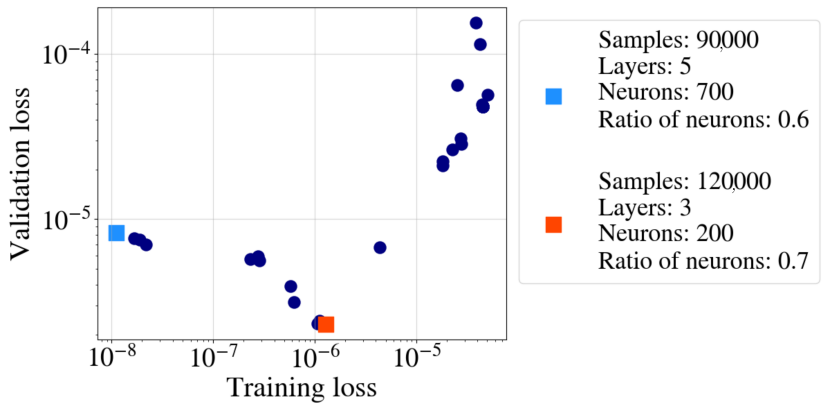


Figure 2.5: Results of the hyperparameter optimization for the Hamiltonian Neural Network. The training loss is plotted against the validation loss. Each point represents one trained network during the hyperparameter optimization. The points with the best training and validation loss are represented by blue and red squares, respectively, and their associated parameters are shown in the legend.

Some of the most commonly used activation functions fail to capture the characteristics of the problem. For example, the activation function has to take into account the large dynamic range of the values of the problem. Therefore, we select the `SymmetricLog` activation function,

$$f(x) = \tanh(x)\log(x\tanh(x) + 1), \tag{2.21}$$

which was specifically designed for this problem by Cai et al. in 2021 Cai et al. (2021a), together with a Glorot weight initialization Glorot & Bengio (2010).

This function behaves similarly to `tanh` close to zero, and like a logarithmic function for larger values. Moreover, it is symmetric for positive and negative values, as seen in Figure 2.6.

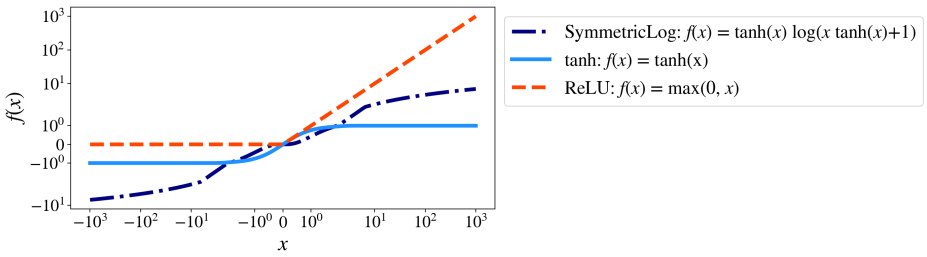


Figure 2.6: Comparison of activation functions. The SymmetricLog activation function was created for this problem by Cai et al. Cai et al. (2021a).

2.3.3 Training results

Both the DNN and the HNN are trained using the Adam optimizer Kingma & Ba (2014) for 2,000 epochs. For SJS, this takes ∼1.3 h and ∼2.3 h for the DNN and the HNN, respectively, on the same computer as we used for the creation of the dataset. For SJSa, the training time is ∼2.5 h and ∼5 h for the DNN and the HNN, respectively.

Once the networks have been trained, we check their accuracy by applying them to the test dataset. For SJS, both the DNN and the HNN converge to a low loss value. Figure 2.7 shows the prediction error for the accelerations obtained with each network. Both networks produce accurate results when the accelerations are large as their output is very close to the $45°$ zero-error line. The relative error grows as the value of the acceleration decreases, since the absolute prediction error is in the order of 10^{-5}. The errors of the DNN are larger than those of the HNN and overestimate the accelerations in the y-direction of Jupiter and in the x-direction of Saturn, and underestimate the y-acceleration of Saturn. This asymmetry leads to a drift in the energy error, as we will explain in Section 2.4. Due to the orbits being almost planar, the accelerations in the z-direction are smaller than for the x- and y-direction, which in Figure 2.7 appears as a larger dispersion of small values.

The hyperparameter optimization in Subsection 2.3.2 shows that we fail to train the HNN for the SJSa case to a satisfactory loss value. Because of the large difference in masses between the asteroids and the planets (∼7 orders of magnitude), when calculating the loss function, some of the gradients of the output with respect to the inputs are required to be extremely large, whereas others have to be small. This leads to the training process focusing on improving the predictions of the accelerations of the asteroids or the planets and, after a certain loss value is achieved, improving one of these implies making the others worse. As a solution, since we successfully trained a network that predicts the accelerations of Jupiter and Saturn, we now train a network that solely predicts the

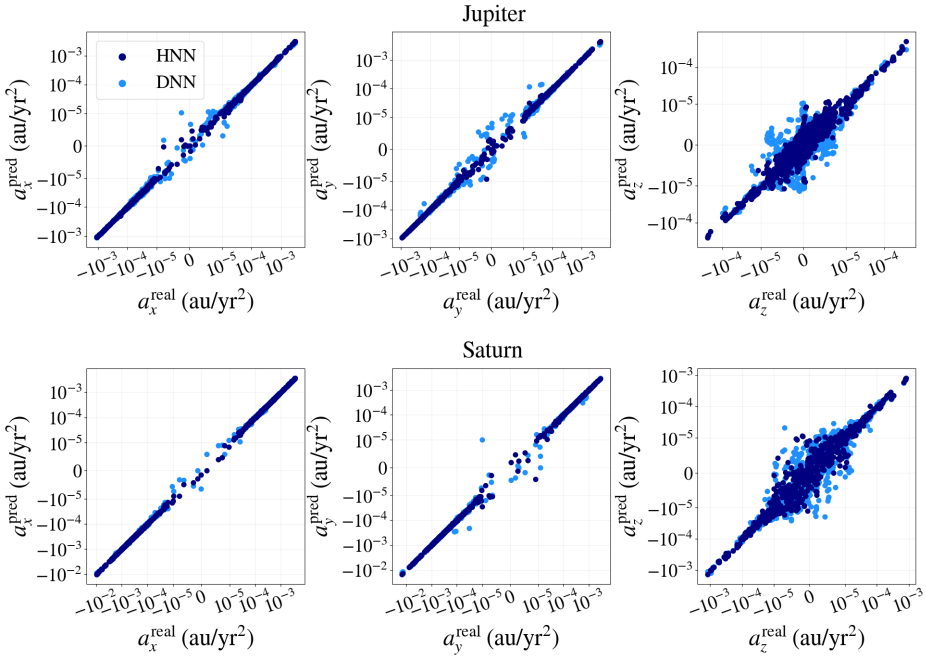


Figure 2.7: Real against predicted values of the acceleration components for the case with the Sun, Jupiter, and Saturn. The real value is compared to the one predicted by the Deep Neural Network and the Hamiltonian Neural Network.

accelerations of the asteroids. We therefore train another HNN where we only include the accelerations of the asteroids in the loss function, ignoring the predictions for Jupiter and Saturn. These results are presented in Figure 2.8.

The DNN is trained with all the bodies in the loss function and can accurately predict the accelerations but, similarly to the predictions for SJS depicted in Figure 2.7, makes errors on the same side of the 45° zero-error line. The HNN trained for the three bodies makes poor predictions for all the outputs, and the HNN trained only for the asteroids predicts the accelerations for the asteroids accurately but (as expected since they are not included in the loss function) fails to predict the accelerations for Jupiter and Saturn (Figure 2.8).

2.3.4 Selection of networks

For SJSa, the HNN fails to predict the accelerations of Jupiter, Saturn, and the asteroids simultaneously. However, if the network is trained with the loss only accounting for the prediction of the accelerations of the asteroids, it can predict these accurately as we discussed in Subsection 2.3.3. For SJSa, we will therefore calculate the accelerations using a combination of two networks: the predictions for Jupiter and Saturn with the network trained for SJS (Figure 2.7), and the prediction for the asteroids with the network that is only trained to predict the accelerations of the asteroids (orange markers in Figure

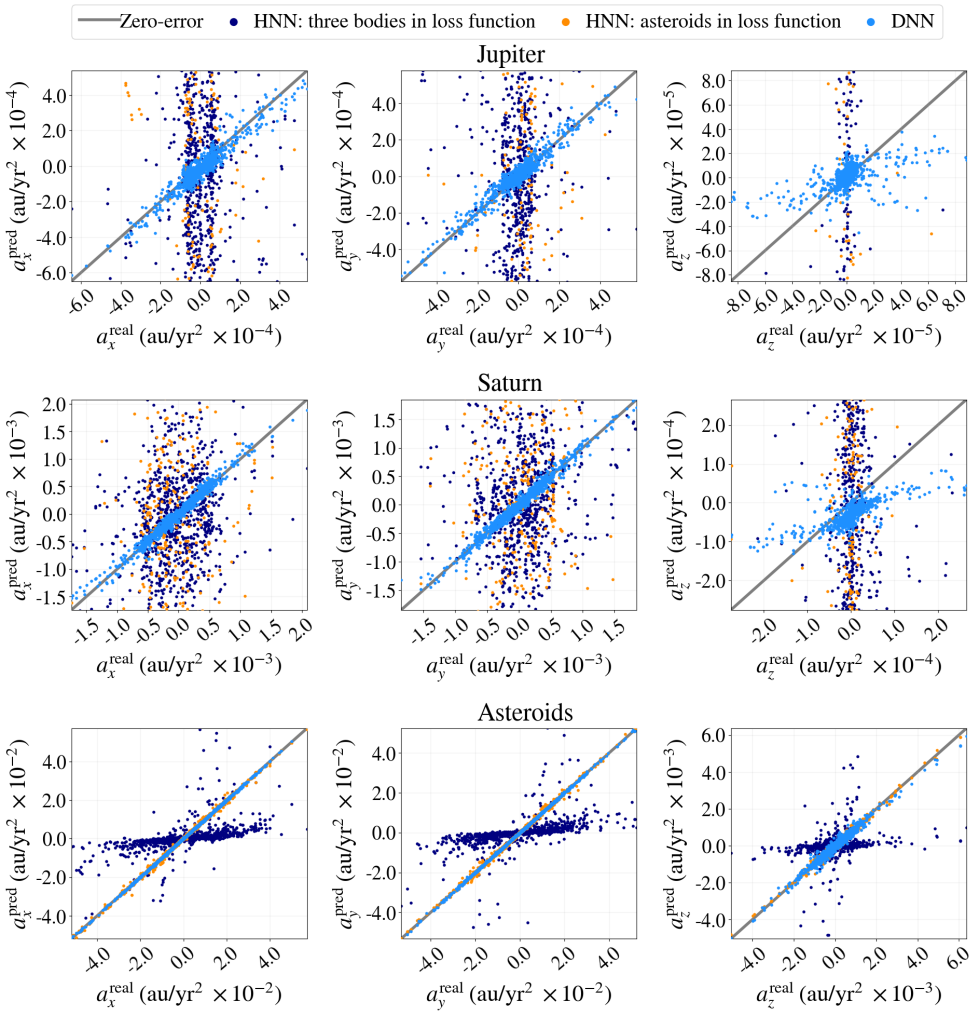


Figure 2.8: Real against predicted values of the acceleration components for the case with the Sun, Jupiter, Saturn, and asteroids. The real value is compared to those predicted by the Deep Neural Network, the Hamiltonian Neural Network for the three bodies, and the Hamiltonian Neural Network with only the asteroids in the loss function.

2.8). This combination of two networks is done with both the HNN and the DNN.

2.3.5 Output of the HNN

It is interesting to understand if the output of the HNN is the same as the actual interactive Hamiltonian of the system (Equation 2.4). To test this hypothesis, we set up an experiment for SJS in which we compare the output of the HNN with the interactive energy of the system.

In Figure 2.9, we show that the predicted values of the interactive Hamiltonian with the HNN, i.e., $f_{\text{HNN}}(X)$ (WH-HNN H in Figure 2.9) are not the same as the interactive energy of the hybrid integrator $\mathcal{H}_{\text{inter}}^{\text{WH-HNN}}$ (WH-HNN Energy in Figure 2.9). The energy of the numerical solution $\mathcal{H}_{\text{inter}}^{\text{WH}}$ is also plotted as WH Energy for reference. The energy evolution of the hybrid integrator exactly coincides with the one of the numerical solution. The output of the HNN does not correspond to the energy value. Therefore, the output of the network does not have physical meaning. This can be explained by realizing that the accelerations obtained with the HNN depend on the relation between the output and the gradients. As a consequence, different combinations of these two variables may lead to similar values of the accelerations.

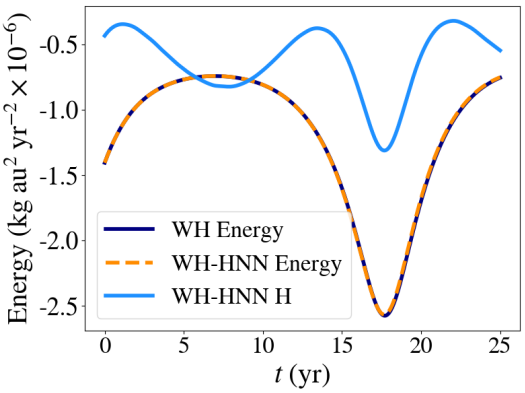


Figure 2.9: Comparison of the output of the Hamiltonian Neural Network with the interactive Hamiltonian of the Wisdom-Holman integrator.

2.4 Results of the hybrid Wisdom-Holman integrator

In this section, we use the networks trained in Section 2.3 in a simulation to further study their performance.

2.4.1 Integration parameters

We initialize the simulation with the state of the Sun, Jupiter, and Saturn from the Horizon System of the Jet Propulsion Laboratory White (2022). We consider a variable number of asteroids initialized with a semi-major axis chosen randomly between 2.2 and 3.2 au, an eccentricity of 0.1, an inclination of $0°$, and a random true anomaly. Then, we use the Wisdom-Holman integrator with a time step (h) of 0.1 yr until a final integration time which depends on the specific case (SJS or SJSa).

2.4.2 Validation of the code

Before discussing the results, we validate the hybrid implementation of the Wisdom-Holman integrator with the neural network. For this purpose, we compare two methods for SJSa: without replacing the HNN result by that of the numerical integrator if the requirement (Equation (2.9)) is not achieved (without flags), and the method with flags as described in Subsection 2.2.3. In Figure 2.10 we show the accelerations of Saturn and two asteroids: asteroid 1 within the limits of the training dataset and asteroid 2 outside to study the extrapolation capabilities of the network. When the prediction of the network is accurate, as it is for Saturn, no flags are needed. However, when the network is not able to reproduce the numerical results, as in the case of asteroid 2, the hybrid integrator detects the poor predictions and replaces these with the results of the numerical calculation. By doing so, the hybrid HNN method becomes significantly more robust against prediction errors. In 2.B, we discuss the number of flags as a function of the parameter R from Equation (2.9).

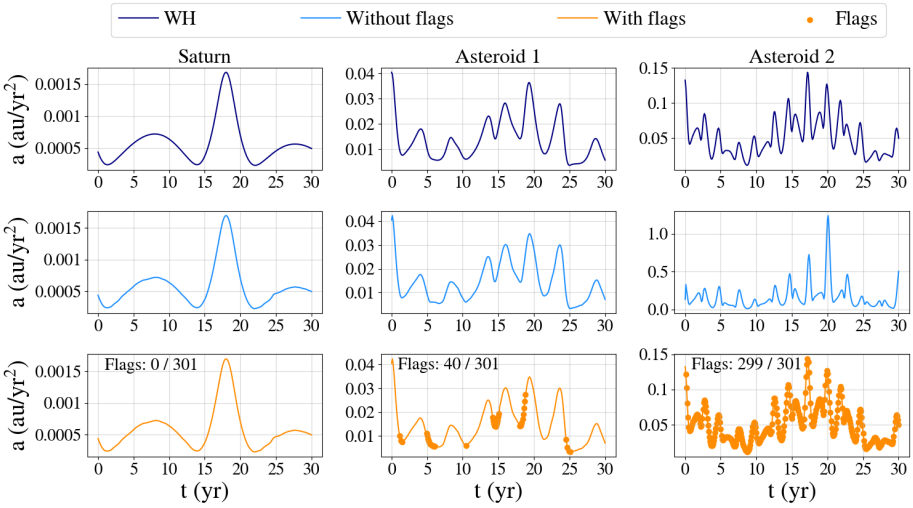


Figure 2.10: Comparison of the accelerations of Saturn, asteroid 1 with an orbit inside the range of the training dataset, and asteroid 2 with an orbit outside the range of training data, using different integration setups. *First row:* Wisdom-Holman integrator, *second row:* Wisdom-Holman integrator with a Hamiltonian Neural Network, and *third row:* hybrid Wisdom-Holman integrator with a Hamiltonian Neural Network and $R = 0.3$. In the third row, the dots represent the points in which the numerical integrator was used because the prediction of the neural network was not considered sufficiently accurate.

We show in Figure 2.10 that the hybrid integrator yields better solutions for the accelerations. However, verifying the predictions of the networks at each time step entails a cost in terms of computing time.

The numerical integration scales with N^2 whereas the neural network result scales with N. For a small number of asteroids, the additional computing time needed to include the neural networks into the integrator makes the method with neural networks more

expensive than the numerical computation. We therefore study what the minimum number of asteroids is to make the use of neural networks computationally less expensive than the numerical computation. In Figure 2.11, three cases are displayed: Wisdom-Holman integrator, WH with HNN without flags, i.e. HNN, and hybrid WH with HNN, i.e, WH-HNN. For a number of asteroids $\leq$70, the use of the HNNs is not preferred above WH as it takes longer to run. However, as the number of asteroids increases, using either HNNs or the hybrid method with HNNs within the integrator results in faster computations, halving the computing time for 2,000 asteroids. Using the hybrid method with the HNN only slightly increases the computing time with respect to the pure HNN case since the prediction for each asteroid is evaluated and replaced individually if necessary. In Figure 2.11b, we see that the hybrid integrator reduces the energy error without significantly increasing the computing time. Since the energy error is dominated by the planets, a small improvement in the energy error implies a significant improvement in the predictions of the accelerations of the asteroids.

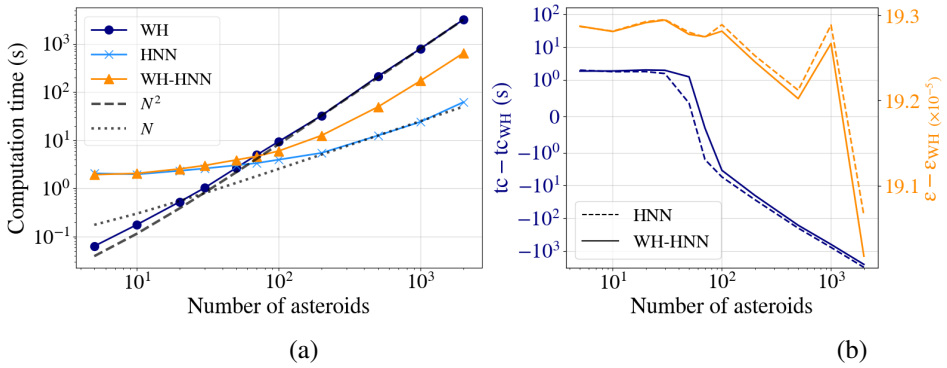


Figure 2.11: Computing time (a), and difference in computing time and energy error with respect to the numerical solution (b) for the integration to 20 years as a function of the number of asteroids. Three cases are shown: numerical integrator (WH), numerical integrator with Hamiltonian Neural Network (HNN), and hybrid numerical integrator with Hamiltonian Neural Network (WH-HNN). The N and N^2 lines are displayed as a reference for linear and quadratic scaling, respectively.

The computing times shown in Figure 2.11 refer to the times for the calculation of the accelerations, i.e., the training times for the neural networks are not included. Once the networks are trained, they can be used in multiple experiments. For example, if the objective is to run 100 experiments, a training time of 2 h is negligible compared to the total computing time.

2.4.3 Trajectory integration

Once the neural networks have been trained, we integrate SJS for 5,000 years (Figure 2.12) and SJSa for 1,000 years (Figure 2.13). To study the extrapolation capabilities of the network, we add two asteroids to SJSa, of which the initial conditions are within the range of training parameters (asteroids 1 and 2) and one asteroid with a semi-major axis outside the range (asteroid 3).

In Figure 2.12, we compare the trajectory, change in eccentricity, and energy error of the hybrid integrator with the HNN and the DNN with respect to the numerical integration. The integrator with the HNN (WH-HNN) reproduces the change in eccentricity of the integrator better than the one with the DNN (WH-DNN). The evolution of the eccentricity is an important indicator of how well the orbit is reproduced using the neural networks. Another indicator is the energy error (third row). Although the WH-HNN leads to a larger energy error than the WH integrator, it shows symplectic behavior. In contrast, the use of the WH-DNN leads to a systematic drift in the energy error. This causes a gradual divergence from the numerical solution. We illustrate this with Figure 2.7, where the DNN produces prediction errors that are asymmetrically distributed around the zero-error line. We conclude that for the SJS case the hybrid integrators with the HNN and the DNN can reproduce the numerical results for short time scales, although the use of the latter results in a systematic deviation from the reference solution.

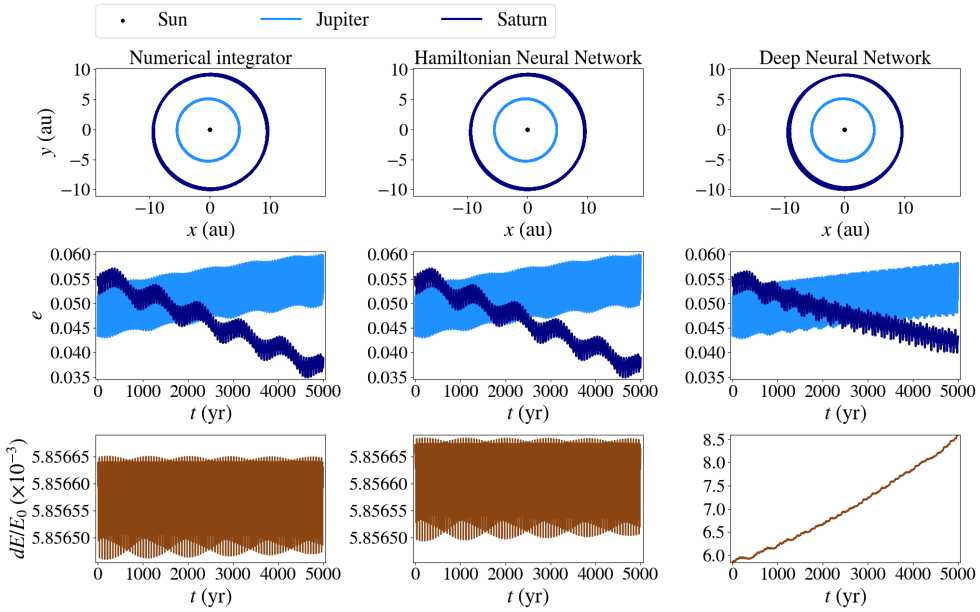


Figure 2.12: Simulation results for the Sun, Jupiter, and Saturn (SJS). The results are generated using the Wisdom-Holman integrator (*left*), hybrid Wisdom-Holman with the Hamiltonian Neural Network (*middle*), and hybrid Wisdom-Holman with the Deep Neural Network (*right*). The trajectories in the x-y plane are shown in the first row, the eccentricities in the second row, and the relative energy error in the third row.

For SJSa, the results in Figure 2.13 show that the trajectories of asteroids 1 and 2 can be predicted with both the WH-HNN and the WH-DNN for short integration times. For longer integration times, the DNN is not able to reproduce the trajectories of the asteroids accurately; there is a systematic drift in the evolution of the eccentricity. Regarding the extrapolation capabilities of the networks, neither the HNN nor the DNN can predict the trajectory of asteroid 3. However, the hybrid integration allows the accelerations to

be adjusted to the numerical values, leading to more accurate trajectories. Regarding the energy error, the behavior observed is the same as in Figure 2.12 since the energy magnitudes of Jupiter and Saturn dominate over the energy magnitudes of the asteroids. We conclude that for short time scales both networks incur in a small error with respect to the numerical integration results, but the HNN achieves a more accurate reproduction of the trajectory of the asteroids over a longer time scale.

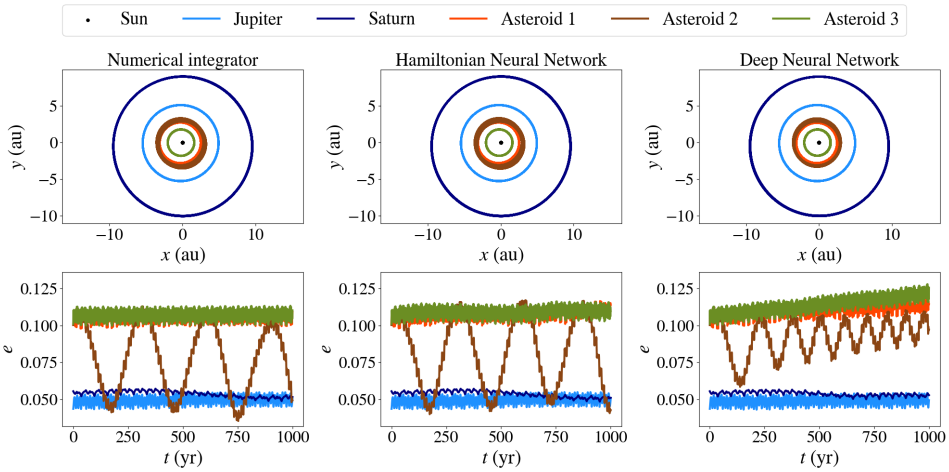


Figure 2.13: Simulation results for the case with the Sun, Jupiter, Saturn, and asteroids. The results are generated with the Wisdom-Holman integrator (*left*), hybrid Wisdom-Holman with the Hamiltonian Neural Network (*middle*), and hybrid Wisdom-Holman with the Deep Neural Network (*right*). The trajectory in the x-y plane is shown in the first row and the eccentricity in the second row. Asteroids 1 and 2 are within the limits of the training dataset and asteroid 3 has a semi-major axis below the lowest limit of the training dataset to study the extrapolation capabilities of the networks.

2.5 Conclusion

In this paper, we studied the use of Artificial Neural Networks for the prediction of accelerations in a planetary system with a star orbited by two planets and a number of asteroids. We compared the results produced by a Deep Neural Network and a Hamiltonian Neural Network. The latter includes physical knowledge about the conservation of energy.

In contrast to previous studies that use neural networks for the gravitational N-body problem, we focused on an actual astrophysics problem. By using a case-specific integrator and modifying the number of bodies and their masses and positions to represent a realistic scenario, we encountered challenges that are not found when using this problem as a test case.

We created a method that circumvents some of the major challenges of using neural networks for the N-body problem. First of all, by using a hybrid integrator that evaluates the prediction and chooses between the numerical or the neural network solution, we

addressed the problem of accumulation of errors over large timescales. Secondly, our setup allows for a variable number of bodies in the system without the need to retrain the network. With the simplest setups found in literature, an increase in the number of bodies in the system implies that the network needs to be retrained. Finally, we use custom activation functions and weights in the loss function to adapt to the characteristics of the problem.

Although based on the optimistic results from the literature Raissi et al. (2019b); Lu et al. (2021); Greydanus et al. (2019a) we expected the HNN to outperform the DNN, in the case with the asteroids, the HNN could not be trained to predict simultaneously the accelerations of the planets and the asteroids. Because of the presence of physics constraints in HNNs, normalization is not possible. This becomes an obstacle for training due to the differences in masses of the bodies. We therefore trained two individual networks for the accelerations of the planets and the asteroids. Although using HNNs has its advantages for the simplified case with Jupiter and Saturn, we demonstrated their limitations for other configurations.

HNNs turn out to be more time-consuming and harder to train, in contrast to the DNN. We had to develop a dedicated activation function specifically for this problem and the hyperparameter optimization performed was time-consuming as well.

With more than 70 asteroids, the integration with the neural networks becomes faster than the direct numerical integration, and for 2,000 asteroids the use of neural networks leads to a halving of the computing time. Since the goal is to create a method that can be used multiple times, the performance comparison does not include the time used for training.

We developed a hybrid integrator to alleviate the problems induced by the introduction of neural networks in the integration process. By verifying the prediction made by the ANN at each time step and replacing this prediction by the numerical integrator if necessary, the integrator becomes more reliable and robust to prediction errors without significantly increasing the computing time. Therefore, for a sufficiently large number of asteroids ($\sim$ 70), we find that the hybrid approach with the HNN proposed here outperforms the direct integration without losing the underlying physics of the system, as opposed to the hybrid integrator with the DNN. Although our study shows that it is beneficial to use physics-aware architectures that conserve the symplectic structure of the integrator, our hybrid method is independent of the network topology chosen. We focused on the simplest cases of neural networks to allow for a better understanding of the underlying challenges of the problem, but further studies should focus on the use of more complex network topologies.

In short, we showed that neural networks can be used to speed up the integration process for problems with a large number of asteroids. However, for long integration times, the prediction errors may accumulate causing the results to diverge with respect to the solution obtained by direct numerical integration. Moreover, if no hybrid integration method that verifies the prediction of the network is used, these prediction errors may lead to unphysical solutions on a short time scale. The use of HNNs is justified for cases in which normalization is not needed to train the network, which in this study means when the masses of the different bodies are of the same order of magnitude. When the HNNs can be trained, they show symplectic behavior, with the energy error oscillating around the initial value. In contrast, DNNs are easy to train and lead to satisfactory solutions,

but are not able to extrapolate to conditions that are not part of the training data and are unsuited for finding solutions that conserve energy.

2.6 Acknowledgments

This publication is funded by the Dutch Research Council (NWO) with project number OCENW.GROOT.2019.044 of the research programme NWO XL. It is part of the project "Unravelling Neural Networks with Structure-Preserving Computing". In addition, part of this publication is funded by the Nederlandse Onderzoekschool Voor Astronomie (NOVA).

Appendix

2.A Dataset parameters

The parameters for the simulation and initial conditions are shown in Table 2.1. The orbital elements which have not been included in the table, i.e., the right ascension of the ascending node and the argument of the periapsis, have been set to zero.

Table 2.1: Summary of dataset parameters. sma is the semi-major axis. J represents Jupiter, S Saturn, and a the asteroids

Parameter	SJS & SJSa
Experiments (train)	500
Experiments (test)	25
Time step	5.0 $\times 10^{-3}$ yr
Final time	30 yr
Mass J (m_J)	9.543e-4 M_{Sun}
Mass S (m_S)	2.857e-4 M_{Sun}
Mass a (m_a)	$[1 \times 10^{19}, 1 \times 10^{20}]$ kg
sma J (a_J)	[4, 8] au
sma S (a_S)	[8.5, 10] au
sma a (a_a)	[2.2, 3.2] au

Parameter	SJS & SJSa
Eccentricity J (e_J)	[0, 0.1]
Eccentricity S(e_S)	[0, 0.1]
Eccentricity a (e_a)	[0, 0.1]
Inclination J (i_J)	[0, 6°]
Inclination S (i_S)	[0, 6°]
Inclination a (i_a)	[0, 6°]
True anomaly J (f_J)	[0, 360°]
True anomaly S (f_S)	[0, 360°]
True anomaly a (f_a)	[0, 360°]

The distribution of inputs and outputs of the training dataset for the case with the Sun, Jupiter, Saturn, and the asteroids is shown in Figure 2.14 and Figure 2.15, respectively.

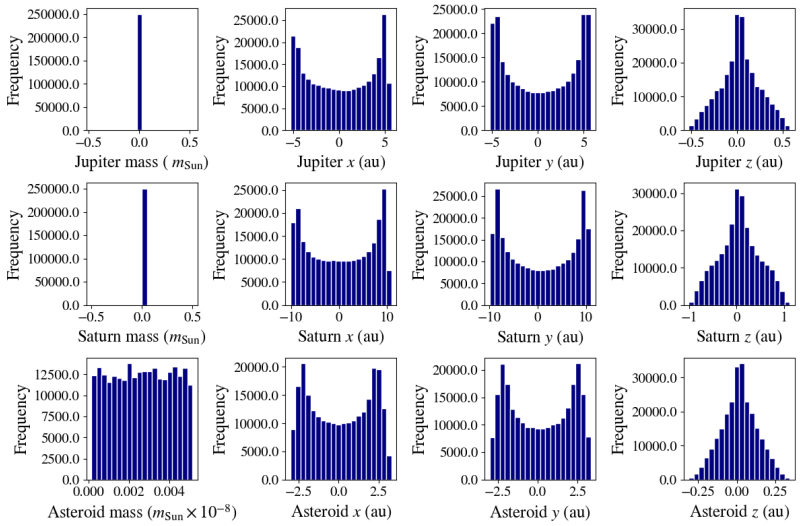


Figure 2.14: Distribution of inputs for the training dataset. The inputs include the position vectors and masses of the two planets and the asteroids

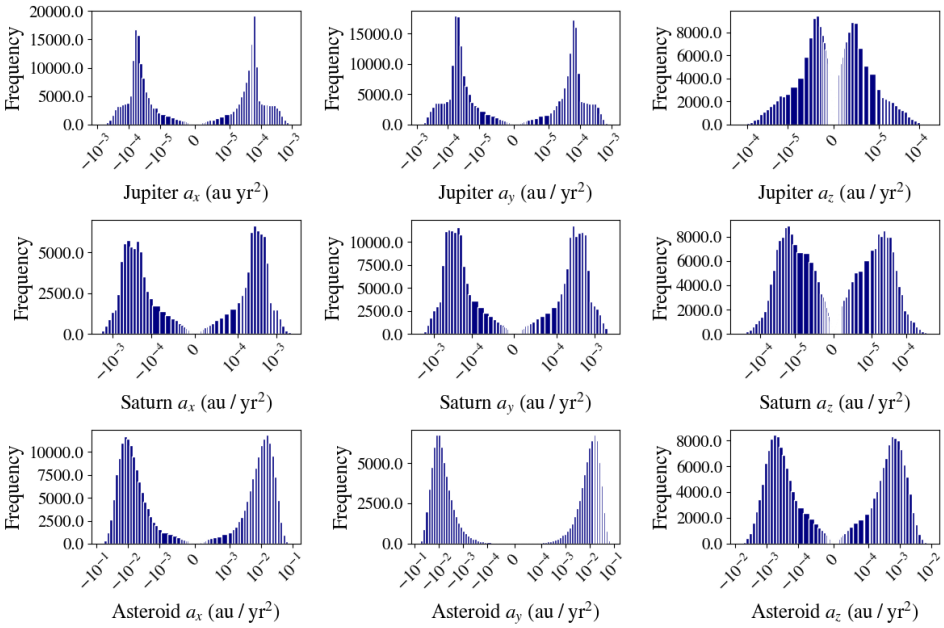


Figure 2.15: Distribution of outputs for the training dataset. The outputs include the acceleration vectors of the two planets and the asteroids.

2.B Hybrid method

In Equation (2.9), we showed the criterion for rejecting the prediction of the neural network. Depending on the value of R chosen, the balance between energy error and computing time changes. We show in Figure 2.16 the number of flags for three values of R for the integration of asteroid 1 with the HNN. As R increases, the method becomes less strict and the accelerations predicted are further from those calculated numerically. If R is 0.1, all HNN values are rejected, and the whole simulation is done ignoring the predictions of the HNN.

The computing time decreases as the number of flags needed is reduced, i.e, for larger values of R.

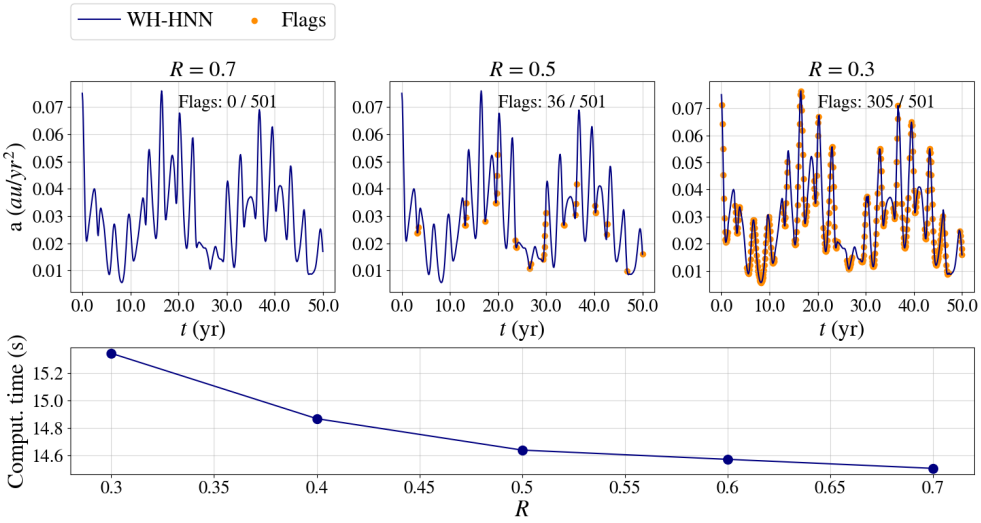


Figure 2.16: Comparison of the number of flags in the integration of Asteroid 1 with different values of R (Equation (2.9)). *First row:* accelerations with flags for three different values of R. *Second row:* Computing time as a function of R.

3 A Generalized Framework of Neural Networks for Hamiltonian Systems

Based on the work by Philipp Horn, Veronica Saz Ulibarrena, Barry Koren, Simon Portegies Zwart, 2025, *Journal of Computational Physics*, Volume 521, Part 1, 113536. Adapted for this thesis.

ABSTRACT

When solving Hamiltonian systems using numerical integrators, preserving the symplectic structure may be crucial for many problems. At the same time, solving chaotic or stiff problems requires integrators to approximate the trajectories with extreme precision. Integrating Hamilton's equations to a level of scientific reliability such that the answer can be used for scientific interpretation, may lead to computationally expensive simulations. In some cases, a neural network can be a viable alternative to numerical integrators, offering high-fidelity solutions orders of magnitudes faster.

To understand the role of preservation of symplecticity in problems where neural networks are used, we analyze three well-known neural network architectures that include the symplectic structure inside the neural network's topology. Between these neural network architectures, many similarities can be found. This allows us to formulate a new, generalized framework for these architectures. In the generalized framework Symplectic Recurrent Neural Networks, SympNets, and HénonNets are special cases. Additionally, this new framework enables us to find novel neural network topologies by transitioning between the established ones.

We compare new Generalized Hamiltonian Neural Networks (GHNNs) against the already established SympNets, HénonNets, and physics-unaware multilayer perceptrons. This comparison is performed for the gravitational three-body problem. In order to perform a fair comparison, the hyperparameters of the different neural networks are chosen such that the prediction speeds of all four architectures are the same during inference. A special focus lies on the capability of the neural networks to generalize outside the training data. The GHNNs outperform all other neural network architectures for the problem considered.

3.1 Introduction

Specialized neural network architectures for scientific machine learning are still scarce. In most cases, neural network architectures from different areas of machine learning are used for scientific applications without any adaptations. If prior knowledge is included, it is usually done through additional terms to the loss function. This is the case of Physics-Informed Neural Networks (PINNs) (Raissi et al. (2019b)) and all of its variants.

Imposing physics constraints through the loss function does have advantages. For example, it does not require extensive expertise in neural network architectures and can be done quickly. It proves to be very useful in cases with limited data to train the neural network. However, including prior knowledge only through the loss function also has its drawbacks. Adding an additional loss term introduces a new hyperparameter, a parameter to determine the weighting between the error on the data and how far the neural network is allowed to deviate from the physics constraint. Furthermore, this constraint is only a soft one, which means that the neural network is not strictly constrained, it may deviate from it. Also, the neural network's loss only increases if the constraint is not obeyed at certain predetermined inputs called collocation points, with which the network is trained. The constraint is not enforced globally but point-wise only, and increasing the number of points slows down the training.

Including prior knowledge into the topology of the neural network requires the development of a new topology for every structure one wants to preserve. This is more labor-intensive and may break the approximation properties of neural network architectures for which we understand their behavior. However, if such a specialized topology can be found, the physics constraints are hard ones and are actually enforced everywhere; inside and outside the training data. Furthermore, it does not require the introduction of new hyperparameters.

3.1.1 Hamiltonian Systems

We focus on structure-preserving neural networks for Hamiltonian systems. While the neural networks analyzed in this work are applicable to all Hamiltonian systems, our numerical experiment is focused on a case of Hamiltonian systems from mechanics: the gravitational three-body problem. The state $\vec{s}$ of a Hamiltonian system is described by its generalized momentum $\vec{p}$ and generalized position $\vec{q}$. The system's dynamics are described by Hamilton's equations:

$$\begin{pmatrix} \dot{\vec{p}} \\ \dot{\vec{q}} \end{pmatrix} = -J\nabla\mathcal{H}(\vec{p},\vec{q}), \quad J = \begin{pmatrix} 0 & I_d \\ -I_d & 0 \end{pmatrix}, \quad \vec{p},\vec{q} \in \mathbb{R}^d, \tag{3.1}$$

with $\mathcal{H} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ being the Hamiltonian of the system and I_d the d-dimensional identity matrix.

The data the neural networks are trained on consists of multiple trajectories with different initial states. The trajectory data consists of multiple states $\vec{s}_i$ along the individual trajectories that are a fixed timestep h apart from each other. The feature and label pairs of the training data are the states $\vec{s}_i$ and $\vec{s}_{i+1}$, respectively. Hence, the neural networks learn an approximation of the flow map of the system $\varphi_h : \vec{s}(t) \mapsto \vec{s}(t+h)$. This flow map

can also be approximated numerically by integrators if the Hamiltonian $\mathcal{H}$ of the system is known.

The motivation to use neural networks for Hamiltonian systems instead of numerical integrators can be diverse. One reason could be that the Hamiltonian of the system is unknown, only data of trajectories is observed. For other Hamiltonian systems, the use of numerical integrators can be computationally expensive. This can be the case because calculating the Hamiltonian is costly or because extremely small timestep sizes are required. For example in Breen et al. (2020b) the equations of motion are stiff and chaotic and therefore an accurate approximation requires small timesteps. In all these cases, the use of neural networks can be a viable alternative. However, for the case where the Hamiltonian of the system is known and one is interested in a faster alternative to numerical integrators, the generation of the training data and the training time have to be taken into account for the total computational cost. Therefore, a neural network surrogate can only provide a reasonable speedup if the number of trajectories that have to be calculated is much larger than the training set. For application to the three-body problem shown in Section 3.5, the Hamiltonian is known and the data is created by a numerical integrator.

It is well established that symplectic integrators can be superior to non-structure-preserving numerical integrators when applied to certain Hamiltonian systems. Especially long-term energy preservation and increased stability often are the two major benefits of symplectic integrators (Hairer et al. (2006)). In symplectic integrators, the numerical flow map $\Phi_h : \vec{s}_i \mapsto \vec{s}_{i+1}$ defined by the integrator has the same geometric structure as the real flow map. This structure is given by:

$$\left(\frac{\partial \varphi_h}{\partial \vec{s}}\right)^T J \frac{\partial \varphi_h}{\partial \vec{s}} = J, \quad \vec{s} = \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} \in \mathbb{R}^{2d}. \tag{3.2}$$

This symplectic property is the structure that is preserved in most structure-preserving neural networks for Hamiltonian systems. Some benefits of this structure have been studied thoroughly for numerical integrators. However, symplecticity in numerical integrators is not always fully understood and the benefits cannot be carried over directly to neural networks.

In this work, we give a short overview of already published neural networks for Hamiltonian systems in Section 3.2. In Section 3.3, we unite three different neural network architectures in one generalized framework called Generalized Hamiltonian Neural Networks (GHNNs). We are able to do this by presenting a new point of view on SympNets and HénonNets. This generalized framework not only includes Symplectic Recurrent Neural Networks (SRNNs), SympNets, and HénonNets, but also new neural network topologies that lie beyond the capabilities of these three architectures. At the same time, the GHNNs preserve the good theoretical properties of the SympNets, and HénonNets. In Section 3.4, we highlight important aspects related to the implementation of GHNNs. By comparing GHNNs against SympNets, HénonNets, and non-structure-preserving neural networks in Section 3.5, we show the superior performance of a GHNN topology that cannot be categorized as SRNN, SympNet, or HénonNet. For completeness, in Section 3.5, we also compare our GHNNs with PINN-type neural networks. All neural networks are trained with topologies that offer similar prediction speeds to achieve a fair comparison.

3.2 Previously introduced NNs for Hamiltonian Systems

Many specialized neural network architectures for data from Hamiltonian systems have already been proposed (Greydanus et al. (2019b); Chen et al. (2020); Jin et al. (2020a); Burby et al. (2021); Xiong et al. (2021)), each of them promising to surpass the previously introduced ones in numerical experiments. Moreover, each architecture tries to remove restrictions imposed on the earlier architectures. SRNNs improve on HNNs by removing the need for time derivatives in the training data and by enforcing symplecticity. SympNets remove the restriction to separable Hamiltonian systems of SRNNs and prove a universal approximation theorem. HénonNets claim to achieve a higher accuracy than SympNets by learning Hénon maps instead of unit triangular updates.

Instead of only adding to this collection of neural networks, we would like to present a generalizing framework that combines three of these types of neural networks, namely: SRNNs, SympNets, and HénonNets, in a single architecture while allowing the creation of new neural network topologies that lie beyond the capabilities of these other architectures. To introduce this framework, we first have a look at the already published architectures.

3.2.1 Hamiltonian Neural Networks

The first approach to neural networks specifically designed for Hamiltonian systems was presented in Greydanus et al. (2019b). These Hamiltonian Neural Networks (HNN) try to learn the Hamiltonian of a system using a multilayer perceptron (MLP). To learn the Hamiltonian, not only data on positions and momenta is needed but also data on its derivatives. Then a mean-square loss can be formulated by iterating over all N datapoints:

$$\mathcal{L}_{\text{HNN}} = \frac{1}{N} \sum_{i=1}^{N} \left|\left| \begin{pmatrix} \dot{\vec{p}}_i \\ \dot{\vec{q}}_i \end{pmatrix} + J \nabla \mathcal{H}_\theta(\vec{p}_i, \vec{q}_i) \right|\right|_2^2 . \tag{3.3}$$

For inference, any stable and consistent numerical integrator can be chosen to calculate $\vec{s}_{i+1}$ from $\vec{s}_i$. Also, the step size can be arbitrary since it is not part of the training. See Figure 3.1, for a schematic of a Hamiltonian Neural Network.

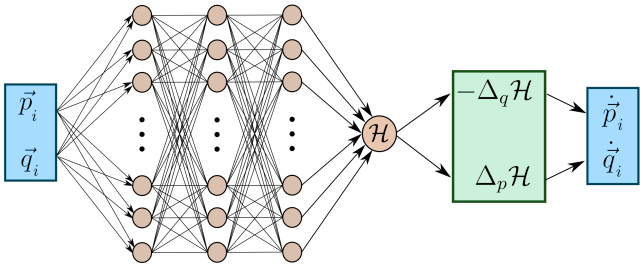


Figure 3.1: Schematic of a Hamiltonian Neural Network. The input and output layers are colored blue. All trainable parameters are in the multilayer perceptron (orange). In the green layer the gradients of the multilayer perceptron with respect to its inputs are calculated using automatic differentiation.

Hamiltonian Neural Networks require additional data and the symplectic structure is only preserved if a symplectic integrator is used during inference. Furthermore, not only an approximation error in the learned Hamiltonian is accumulated, but also an additional numerical error. For these reasons, we are not going to analyze HNNs in more detail. Instead, we take a closer look at the Symplectic Recurrent Neural Networks that are heavily inspired by HNNs but without the above-mentioned shortcomings.

3.2.2 Symplectic Recurrent Neural Networks

Symplectic Recurrent Neural Networks (SRNNs) were introduced in Chen et al. (2020). An SRNN learns a separable Hamiltonian

$$\mathcal{H}_\theta(\vec{p}, \vec{q}) = T_{\theta_1}(\vec{p}) + U_{\theta_2}(\vec{q}), \tag{3.4}$$

parameterized by two independent neural networks, one for the kinetic energy T_{θ_1} and one for the potential energy U_{θ_2}. The main reason behind the use of recurrent neural networks is to compensate for noisy data by using multiple combined timesteps to calculate the error. However, the important improvement over Hamiltonian Neural Networks is not reflected in the name of this architecture and is also present if SRNNs are used with simple multilayer perceptrons instead of recurrent neural networks. The important improvement is that in SRNNs the integrator is always symplectic and is embedded in the neural network's topology itself. This means that the SRNN predicts the state of the Hamiltonian system after a timestep of size h, instead of only the derivatives of the state at the current time. Therefore, a loss function can be defined with the data as introduced in Section 3.1.1 and no additional information on the derivatives is needed. The error is then backpropagated through the integrator into the two neural networks composing the learned Hamiltonian (see Figure 3.2).

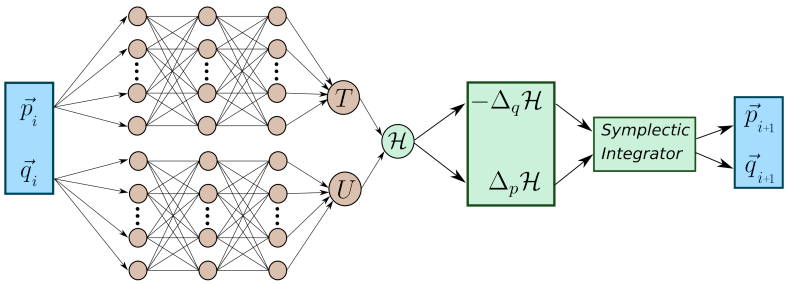


Figure 3.2: Schematic of a Symplectic Recurrent Neural Network with two independent multilayer percetrons (orange) for the learned Hamiltonian. Also, the symplectic integrator is included in the neural network itself. There are no trainable parameters in the green parts of the neural network. Instead, only predefined mathematical operations are performed.

A Hamiltonian Neural Network learns a function that approximates the real Hamiltonian of the system. The approximation error together with the numerical error of the integrator, which is used for inference, add up. The SRNN directly learns the flow map of the system. As noted in Chen et al. (2020), the learned Hamiltonian inside the SRNN

adjusts to the symplectic integrator used during training and compensates for the numerical error of the integrator. Therefore, a convergence to a low value of the loss function does not imply that the learned Hamiltonian inside the neural network converges to the Hamiltonian underlying the data. Because of this behavior, the learned Hamiltonian only provides limited interpretability and it should not be used in combination with any integrator or timestep other than the one used during training.

3.2.3 SympNets

As a third approach we consider SympNets. SympNets have been introduced independently of the idea of learning a Hamiltonian and instead focus directly on learning symplectic maps (Jin et al. (2020a)). This is done by concatenating so-called upper and lower unit triangular updates (f_{up} and f_{low}, respectively):

$$f_{\text{up}}\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix} \coloneqq \begin{pmatrix}\vec{p}+\nabla V(\vec{q})\\ \vec{q}\end{pmatrix} = \begin{bmatrix} I & \nabla V(\cdot)\\ 0 & I\end{bmatrix}\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix}, \tag{3.5}$$

$$f_{\text{low}}\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix} \coloneqq \begin{pmatrix}\vec{p}\\ \vec{q}+\nabla V(\vec{p})\end{pmatrix} = \begin{bmatrix} I & 0\\ \nabla V(\cdot) & I\end{bmatrix}\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix}. \tag{3.6}$$

In the right-hand side of equations (3.5) and (3.6) we adopted the short-hand notation of Jin et al. (2020a). This is a slight abuse of matrix vector multiplication, which is helpful to obtain a compact notation of concatenated unit triangular updates. In every update, also called module, a different parameterized gradient ∇V is learned (Figure 3.3).

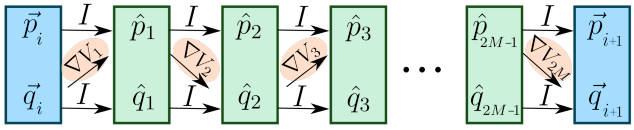


Figure 3.3: Schematic of a SympNet with $2M$ modules alternating between upper and lower modules. In orange it is highlighted where the trainable parameters are located.

Different possible parameterizations of ∇V can be introduced, the most general update being the gradient module (here as upper gradient module):

$$\nabla V(\vec{q}) \coloneqq K^T \text{diag}(\vec{a})\sigma(K\vec{q}+\vec{b}). \tag{3.7}$$

In a gradient module, the trainable parameters are: a weight matrix $K \in \mathbb{R}^{n\times d}$, a bias $\vec{b} \in \mathbb{R}^n$ and a scale factor $\vec{a} \in \mathbb{R}^n$. The width n of this module can be freely chosen. Additionally, a nonlinearity is present in the form of an activation function σ. SympNets composed of only gradient modules are called G-SympNets.

For these SympNets, a universal approximation theorem can be proven (Jin et al. 2020a, Theorem 5). In this theorem, the width and depth of the G-SympNet are not constrained. This means that, in contrast to a multilayer perceptron where one hidden layer is enough to approximate a large class of functions, a SympNet might require a large number of also wide modules to approximate a given symplectic map (Hornik (1991)). The proof of the universal approximation capabilities of SympNets is based on the fact

that four unit triangular updates can form a Hénon-like map (defined below) and that any symplectic map can be approximated by a concatenation of these Hénon-like maps (Turaev (2002)).

3.2.4 HénonNets

As an improvement to the SympNets, the HénonNets were published in Burby et al. (2021). Instead of learning the unit triangular updates, they directly focus on learning Hénon-like maps:

$$h[V,\vec{\eta}]\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix} \coloneqq \begin{pmatrix}-\vec{q}+\nabla V(\vec{p})\\ \vec{p}+\vec{\eta}\end{pmatrix} = \begin{bmatrix}\nabla V(\cdot) & -I\\ I & 0\end{bmatrix}\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix} + \begin{pmatrix}0\\ \vec{\eta}\end{pmatrix}. \tag{3.8}$$

In order to create a Hénon layer, the same Hénon-like map is iterated four times. The reasoning behind this choice is the universal approximation theorem for Hénon-like maps (Turaev (2002)):

$$l[V,\vec{\eta}]\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix} \coloneqq h[V,\vec{\eta}]\circ h[V,\vec{\eta}]\circ h[V,\vec{\eta}]\circ h[V,\vec{\eta}]\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix}. \tag{3.9}$$

In one Hénon layer, the function V and the vector $\vec{\eta}\in\mathbb{R}^d$ are learned. As a parameterization for V any multilayer perceptron can be used. See Figure 3.4, for a schematic of a single Hénon layer.

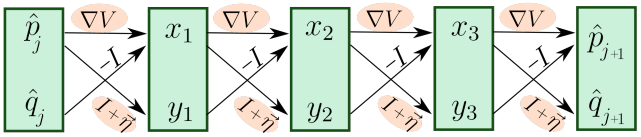


Figure 3.4: Schematic of a single Hénon layer.

Since the HénonNet architecture is based on the same theory as SympNets, they are universal approximators of symplectic maps as well (Burby et al. (2021)). Because HénonNets directly focus on learning Hénon-like maps and because of the greater flexibility in the choice of the function V, HénonNets are able to achieve a lower loss than SympNets with fewer training epochs. In Burby et al. (2021), different numerical experiments are performed to corroborate this statement.

3.3 Generalized Framework

When analyzing the three approaches in more detail, one can see that the SRNNs, SympNets, and HénonNets are actually more similar than they seem to be. This similarity allows us to introduce a generalized framework unifying all three neural network architectures. All possible SRNNs, SympNets, and HénonNets are included in this generalized framework. But also, new symplectic neural network topologies can be derived from it.

A major step to show the similarities between Symplectic Recurrent Neural Networks and SympNets was made in Horn et al. (2022). It shows that SympNets can be seen

as a concatenation of specific SRNNs. Every two gradient modules learn a separable Hamiltonian, which is parameterized by two multilayer perceptrons with one hidden layer, one multilayer perceptron for the kinetic and one for the potential energy. With this learned Hamiltonian, one step of a Symplectic Euler (SE) method is performed. The next two layers learn a different Hamiltonian, perform another Symplectic Euler step and so on. This new point of view on SympNets is illustrated in Figure 3.5.

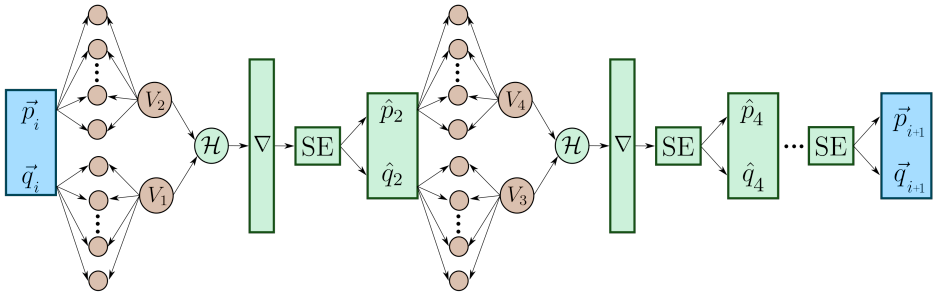


Figure 3.5: A SympNet as a concatenation of Symplectic Recurrent Neural Networks.

A similar equivalence exists between HénonNets and SympNets and therefore also between HénonNets and SRNNs. We can show that one Hénon layer is equivalent to a concatenation of four specific unit triangular updates:

$$l[V,\vec{\eta}]\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix} = \begin{bmatrix} I & \nabla V_4(\cdot) \\ 0 & I \end{bmatrix}\begin{bmatrix} I & 0 \\ \nabla V_3(\cdot) & I \end{bmatrix}\begin{bmatrix} I & \nabla V_2(\cdot) \\ 0 & I \end{bmatrix}\begin{bmatrix} I & 0 \\ \nabla V_1(\cdot) & I \end{bmatrix}\begin{pmatrix}\vec{p}\\ \vec{q}\end{pmatrix}, \tag{3.10}$$

with:

$$V_1 = f \circ V\,, \qquad \text{where } f(\vec{s}) = -\vec{s}, \tag{3.11a}$$
$$V_2 = V \circ f\,, \tag{3.11b}$$
$$V_3 = f \circ V \circ g\,, \qquad \text{where } g(\vec{s}) = -\vec{s} - \vec{\eta}, \tag{3.11c}$$
$$V_4 = V \circ h\,, \qquad \text{where } h(\vec{s}) = \vec{s} - \vec{\eta}. \tag{3.11d}$$

The derivation of this expression can be found in the original paper.

In a HénonNet, V is parameterized by any multilayer perceptron. Hence, V_1, V_2, V_3, and V_4 are given by the same neural network, only with slight modifications in their weights and biases:

- Multiplying the weights and biases in the output layer of V by -1 yields V_1.
- Multiplying the weights in the first hidden layer of V by -1 yields V_2.
- First subtracting $W_1\vec{\eta}$ from the biases in the first hidden layer of V, then multiplying the weights in the first hidden layer by -1 and finally multiplying the weights and biases in the output layer by -1 yields V_3.
- Subtracting $W_1\vec{\eta}$ from the biases in the first hidden layer of V yields V_4.

Here, W_1 is the weight matrix of the first hidden layer in V.

Therefore, HénonNets consist of the same unit triangular updates as SympNets. The main difference is that per Hénon layer one arbitrary multilayer perceptron is trained and four unit triangular updates are performed with slight modifications of this multilayer perceptron. Note that since modifications of the same multilayer perceptron are reused, the four unit triangular updates are not independent and share weights and biases. Since SympNets are, due to the unit triangular updates, equivalent to a concatenation of SRNNs and because HénonNets are built out of the same unit triangular updates, HénonNets are equivalent to a concatenation of SRNNs as well.

An overview of the three different neural network architectures is given in Table 3.1. All three approaches are phrased as possible multiple integration steps with learned Hamiltonians. Although SRNNs were originally introduced this way, we just saw that SympNets and HénonNets can be formulated in a similar fashion. Moreover, this highlights the fact that not only SRNNs learn a Hamiltonian, but SympNets and HénonNets learn multiple hidden Hamiltonians as well. Whether those Hamiltonians approximate the underlying Hamiltonian of the system is not clear. The purpose of presenting all approaches in this formulation is to showcase the similarities and to discuss the differences.

Table 3.1: Overview of three of the different neural networks for Hamiltonian systems that were introduced in the literature so far.

	SRNNs	**SympNets** ($2n$ gradient modules)	**HénonNets** (n Hénon layers)
Definition of the input to output mapping	$\text{SRNN}(\vec{p},\vec{q}) = \text{SI}(H_{\text{NN}},\vec{p},\vec{q})$	$\text{SympNet}(\vec{p},\vec{q}) = \text{SE}(H_{\text{NN}_n},\cdot,\cdot) \circ \cdots \circ \text{SE}(H_{\text{NN}_1},\vec{p},\vec{q})$	$\text{HénonNet}(\vec{p},\vec{q}) = \text{SE}(H_{\text{NN}_{2n}},\cdot,\cdot) \circ \cdots \circ \text{SE}(H_{\text{NN}_1},\vec{p},\vec{q})$
Details of the learned Hamiltonians	$H_{\text{NN}}(\vec{p},\vec{q}) = T_{\text{NN}}(\vec{p}) + U_{\text{NN}}(\vec{q})$	$H_{\text{NN}_i}(\vec{p},\vec{q}) = T_{\text{NN}_i}(\vec{p}) + U_{\text{NN}_i}(\vec{q})$	$H_{\text{NN}_{2i-1}}(\vec{p},\vec{q}) = V_{\text{NN}_i}(-\vec{p}) + V_{\text{NN}_i}(\vec{q})$ and $H_{\text{NN}_{2i}}(\vec{p},\vec{q}) = V_{\text{NN}_i}(-\vec{p}-\vec{\eta}) + V_{\text{NN}_i}(\vec{q}-\vec{\eta})$
Parameterization of the learned Hamiltonians	T_{NN} and U_{NN} can be any multilayer perceptrons.	T_{NN_i} and U_{NN_i} are multilayer perceptrons with one hidden layer.	V_{NN_i} can be any multilayer perceptron.
The integrators that can be used	SI can be any symplectic integrator that is explicit for separable Hamiltonians.	SE is the Symplectic Euler integrator.	SE is the Symplectic Euler integrator.

The point of view in which layers correspond to integration steps for learned ordinary differential equations is not new. It is similar to the one introduced for ResNets in E (2017). Moreover, it is used to analyze the stability of neural networks in Haber & Ruthotto (2017) and to later introduce Neural Ordinary Differential Equations in Chen

et al. (2018). Also, it is interesting to note that approximating symplectic maps by concatenating integration steps for Hamiltonian systems is close to the key idea in the proof that any symplectic map can be approximated by a concatenation of Hénon-like maps (Turaev (2002)). This is what both the universal approximation theorems of the SympNets and HénonNets are based on.

Alternatively, one could write all neural network architectures as concatenations of unit triangular updates. This is more in line with how SympNets and HénonNets are introduced but it is also possible for SRNNs. Since symplectic integrators that are explicit for separable Hamiltonian systems alternate between updating positions and momenta, they also perform unit triangular updates (Yoshida (1990b)). The formulation as unit triangular updates shows the similarities as well. However, it is less elegant for the SRNNs because which updates are performed and how the learned maps V are connected depends on the symplectic integrator that is used. In both formulations it becomes clear how one can create a generalized framework covering all these neural networks for Hamiltonian systems and even enabling to introduce new ones.

3.3.1 Generalized Hamiltonian Neural Networks

To get an overview of how the different methods intersect, we present Figure 3.6, where it is shown that there is already overlap between the different methods. Any SympNet with only two gradient modules is also an SRNN and any SRNN using the symplectic Euler method and only one hidden layer in both multilayer perceptrons is also a SympNet. Furthermore, any HénonNet using only one hidden layer in all multilayer perceptrons for the learned maps V is also a SympNet (but with additional constraints).

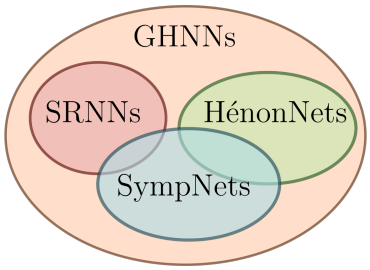


Figure 3.6: Venn diagram showing the overlap between the different neural network architectures for Hamiltonian systems.

We propose a novel type of neural networks for Hamiltonian systems. We denominate this new neural network architecture Generalized Hamiltonian Neural Networks (GHNNs). All types of neural networks for Hamiltonian systems discussed so far can be found as special cases of this framework. This is achieved by removing as many restrictions as possible from the individual layers, while still keeping the symplectic structure in the topology of the neural network.

In the same formulation as used in Table 3.1, in which layers are considered to be

integration steps with learned Hamiltonians, the new architecture can be written as:

$$\begin{aligned} \mathrm{GHNN}(\vec{p},\vec{q}) &= \mathrm{SI}_n(\mathcal{H}_{\mathrm{NN}_n},\cdot,\cdot)\circ\cdots\circ\mathrm{SI}_1(\mathcal{H}_{\mathrm{NN}_1},\vec{p},\vec{q}), \\ \mathcal{H}_{\mathrm{NN}_i}(\vec{p},\vec{q}) &= T_{\mathrm{NN}_i}(\vec{p}) + U_{\mathrm{NN}_i}(\vec{q}), \end{aligned} \tag{3.12}$$

with T_{NN_i} and U_{NN_i} being any multilayer perceptrons and SI_i any symplectic integrator that is explicit for separable Hamiltonian systems.

To recognize the SRNNs as special cases of the GHNNs one has to restrict the GHNNs to one symplectic integrator ($n = 1$). The SympNets can be found in the GHNNs by using the symplectic Euler method for all symplectic integrators SI_i and only using one hidden layer in all multilayer perceptrons of all the learned Hamiltonians H_{NN_i}. Building a HénonNet from this general framework requires more restrictions. First, again only the symplectic Euler method can be used. Second, an even number of integrators has to be used. Finally, while general multilayer perceptrons can be used for the Hamiltonians, many weights and biases are shared between the kinetic and potential energies and also between two different Hamiltonians. The details of this weight sharing are provided in Table 3.1 and the list after Equations 3.11a to 3.11d.

Consequently, one way to create neural networks that are not included in one of the three existing architectures would be to use an integrator different from the symplectic Euler method. Or, instead of using a different integrator, another approach could be to use more than the one hidden layer in the multilayer perceptrons that compose the learned Hamiltonians in SympNets. One such GHNN with two hidden layers is shown in Figure 3.7.

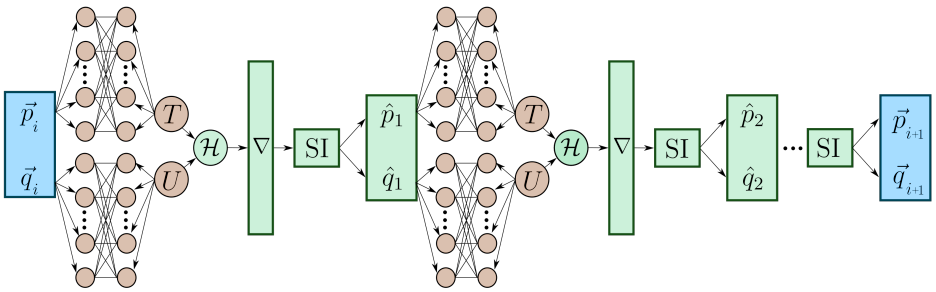


Figure 3.7: Schematic of a Generalized Hamiltonian Neural Network.

In SympNets, only one hidden layer is used since this suffices to prove the universal approximation theorem. However, from machine learning literature we know that using deep NNs instead of wide ones can lead to better approximation capabilities with fewer trainable parameters (Telgarsky (2015)).

3.4 Implementation

Contrary to how we introduced GHNNs, we do not implement them using the concept of learned Hamiltonians and performing integration steps. While SRNNs were implemented that way, it requires calculating gradients of the neural networks with respect

to their input during training and also during inference. On the one hand, this can be done with automatic differentiation, but on the other hand, it considerably slows down the prediction speed of the neural network. For that reason, we instead precalculate the gradients of multilayer perceptrons with one and two hidden layers. The same is done in the original implementation of the gradient modules in SympNets. This does not imply a restriction of the GHNNs. Mathematically, they perform the same calculations as if they were implemented as concatenated integrators. It simply enables a better comparison with SympNets. For a fair comparison we also use this implementation for HénonNets.

Calculating the gradient of a multilayer perceptron with one hidden layer is straightforward. The result can be found in Jin et al. (2020a) for example:

$$\nabla_{\vec{q}}\text{MLP}(\vec{q}) = \nabla_{\vec{q}}\vec{w}^T\Sigma(W\vec{q}+\vec{b}) = W^T\text{diag}(\vec{w})\sigma(W\vec{q}+\vec{b}). \tag{3.13}$$

Here $W \in \mathbb{R}^{n\times d}$ refers to the weight matrix in the hidden layer, $\vec{b} \in \mathbb{R}^n$ to the bias vector in the hidden layer, $\vec{w} \in \mathbb{R}^n$ to the weight matrix/vector in the output layer and Σ to the activation function, with σ its derivative. Both are applied element-wise. In this calculation, $\vec{q}$ can be exchanged with $\vec{p}$ for the gradient needed for a lower triangular update.

The gradient of a multilayer perceptron with two hidden layers is far more convoluted. The derivation can be found in the original paper. The gradients of multilayer perceptrons with three or more hidden layers can be calculated as well. However, because the expression then becomes even more complicated, we compute these gradients using automatic differentiation. For our numerical experiments in the next section, only multilayer perceptrons with up to two layers are used for a fair computation speed comparison with the other neural network architectures.

All code to train the neural networks for this project is written in `Python` using `PyTorch` (Paszke et al. (2019)). The code including methods to generate and process the data for the numerical experiments can be found on `GitHub` (`https://github.com/AELITTEN/GHNN`). Our data can also be found on `Zenodo` (`https://zenodo.org/records/11032352`) and the 850 neural networks trained for this work are available in a separate GitHub repository
(`https://github.com/AELITTEN/NeuralNets_GHNN`).

3.5 Numerical Experiments

In order to compare the new Generalized Hamiltonian Neural Networks with existing neural networks for Hamiltonian systems and with physics-unaware neural networks as well, we perform multiple experiments. For a fair and objective comparison it is not obvious how to choose the topology hyperparameters of the different neural networks (such as the number of neurons and the number of layers). Instead of comparing neural networks with a similar number of trainable parameters, we compare neural networks with similar prediction speeds during inference and training. Due to the computationally expensive need to calculate the gradients of the learned Hamiltonians inside SRNNs, comparing SRNNs to the other neural networks is unfair. As a consequence, we do not include SRNNs in the experiments.

The hyperparameters of the neural networks that lead to similar prediction speeds, and are therefore used for the numerical experiments in this section, can be found in Table 3.2.

Table 3.2: Topology hyperparameters of the different neural networks used for the numerical experiments. The number of layers (except for MLP) is the number of layers of the MLPs that compose the learned Hamiltonians.

NN type	Learned Hamiltonains	Layers	Neurons per Layer	Trainable parameters 3-body problem
MLP	-	5	128	69260
SympNet	10	1	50	8000
HénonNet	10	1	50	2030
GHNN	5	2	25	8500
Deep HénonNet	6	2	25	2568

For all different types of neural networks, including GHNNs, it is not obvious how to choose the hyperparameters. As a starting point for the SympNet hyperparameters, we take a look at the SympNets from Jin et al. (2020a). To guarantee that we do not restrict the capabilities of the SympNets, we use the largest SympNets of this paper, which are the ones used for the 3-body problem considered in Jin et al. (2020a). Since GPUs are used for the training of the neural networks and for inference and since large matrix-vector products in each layer can be efficiently parallelized, the main hyperparameter determining the speed is the depth of the neural network and not the number of neurons in each layer. Hence, the total number of independent layers in the SympNets and GHNNs is kept the same. The SympNets have one hidden layer per multilayer perceptron, two multilayer perceptrons per learned Hamiltonian and ten learned Hamiltonians. The GHNNs have two hidden layers per multilayer perceptron, two multilayer perceptrons per learned Hamiltonian and five learned Hamiltonians. A HénonNet performs four sequential operations with one trained multilayer perceptron. It has one independent multilayer perceptron per two learned Hamiltonians. Therefore, only one quarter of the total number of independent layers of a SympNet can be used in a HénonNet to end up at the same prediction speed. The deep HénonNets are HénonNets with two hidden layers per multilayer perceptron. These were not tested in the original paper that introduced the HénonNets (Burby et al. (2021)). Since a physics-unaware multilayer perceptron does not need a large depth to be a universal approximator, only five layers are used.

While the general framework of our GHNNs allows the use of any symplectic integrator that is explicit for separable Hamiltonians, only the Symplectic Euler method is used for the numerical experiments. Usually a higher-order accurate integrator like the Störmer-Verlet integrator leads to higher accuracy in the prediction of the next state. But, since the Hamiltonians are only learned and do not necessarily correlate with the real Hamiltonian of the system, a better performance of a GHNN with a higher-order accurate integrator cannot be expected. Using the Störmer-Verlet integrator would definitely decrease the prediction speed and might also lead to problems during the training since the unit triangular updates are no longer independent, which complicates the optimization. Nevertheless, investigating the use of other integrators could be an interesting topic for future research.

How the total number of trainable parameters depends on depth, width and the dimension of the Hamiltonian system widely varies for the different neural network architectures. This can be seen in the last column in Table 3.2. The number of trainable parameters heavily depends on the dimension of the Hamiltonian system for SympNets and HénonNets. On the other hand, for the MLP and GHNN the dimension of the Hamiltonian system only has a small effect. This can be explained by the fact that the number of trainable parameters in both an MLP and a GHNN scales with n^2:

$$\begin{aligned}
\text{Trainable parameters in an MLP} &= (l-1)n^2 + (4d+l)n + 2d, && \text{(3.14a)}\\
\text{Trainable parameters in a SympNet} &= 2m(d+2)n, && \text{(3.14b)}\\
\text{Trainable parameters in a HénonNet} &= m/2((l-1)n^2 + (d+l+1)n + d)\\
&\overset{l=1}{=} m/2((d+2)n+d), && \text{(3.14c)}\\
\text{Trainable parameters in a GHNN} &= 2m((l-1)n^2 + (d+l+1)n), && \text{(3.14d)}
\end{aligned}$$

where l refers to the number of hidden layers, n to the number of neurons per layer, d to the dimension of the Hamiltonian system (state $\vec{s}$ is in $\mathbb{R}^{2d}$) and m is the number of learned Hamiltonians, i.e., half the number of gradient modules or twice the number of Hénon maps, respectively.

For the experiment, 50 neural networks per architecture are trained using the Adam algorithm (Kingma & Ba (2015)). Loss plots for all neural networks and all numerical experiments are provided in Appendix 3.A. All 50 of these neural networks sharing the same architecture are identical except for the seeds in the random initialization of their weights. These weights are drawn from a normal distribution with a mean of 0 and a variance of 0.1. All biases are initialized as 0. The `PyTorch` seed is set to 1 for the first neural network, to 2 for the second, and so on. Resulting in different but repeatable initial weights for all neural networks with the same architecture. This allows a quantification of the influence of the initialization of the neural networks and enables us to be certain about whether one architecture is better than the other (given our chosen hyperparameters).

Error plots are provided to compare the different architectures. For these figures, the mean absolute error over the trajectories

$$\left\{ \begin{pmatrix} \vec{p}_i^{\,T}(t) & \vec{q}_i^{\,T}(t) \end{pmatrix}^T \mid i \in S_{\text{test}}, t \leq t_{\text{end}} \right\}$$

in the test data S_{test} is calculated for all neural networks. For a fixed time t the mean $\mu(t)$ over all 50 neural networks is calculated as:

$$\mu(t) = \frac{1}{50} \sum_{j=1}^{50} \mu_j(t), \tag{3.15}$$

with

$$\mu_j(t) = \frac{1}{2d|S_{\text{test}}|} \sum_{i \in S_{\text{test}}} \left|\left| \begin{pmatrix} \vec{p}_{i,\text{data}}(t) \\ \vec{q}_{i,\text{data}}(t) \end{pmatrix} - \begin{pmatrix} \vec{p}_{i,\text{NN}_j}(t) \\ \vec{q}_{i,\text{NN}_j}(t) \end{pmatrix} \right|\right|_1. \tag{3.16}$$

Additionally, as a measure of the variance due to different random initializations of the weights and biases, an area with the upper and lower bounds given by the 90% and 10% quantiles of $\mu_j(t)$ at each point in time is plotted in the figures.

3.5.1 3-Body Problem

We compare the neural networks on data of a gravitational 3-body problem. In this Hamiltonian system, three bodies of different masses orbit each other according to Newton's law of gravitation. The gravitational N-body problem is important for astrophysics. The dynamics of planets, stars and galaxies are described by this Hamiltonian system as long as no relativistic effects are considered.

The system, in all three spatial dimensions, is of the total dimension $d = 3N$, with $\vec{q} = \left(\vec{q}_1^T \cdots \vec{q}_N^T\right)^T \in \mathbb{R}^d$ and the canonical coordinates of each body $\vec{q}_i$ being the coordinates in Euclidean space. The Hamiltonian of the gravitational N-body problem can be written as

$$H(\vec{p}, \vec{q}) = \frac{1}{2}\vec{p}^T M^{-1}\vec{p} - \sum_{i=1}^{N}\sum_{j=1}^{i} G\frac{m_i m_j}{||\vec{q}_j - \vec{q}_i||_2}, \tag{3.17}$$

where G is the gravitational constant and M the mass matrix, which consists of the masses m_i of the N bodies and is given in three-dimensional space by

$$M \coloneqq \text{diag}(m_1, m_1, m_1, m_2, \cdots, m_N, m_N, m_N). \tag{3.18}$$

For our numerical experiments, we simplify the problem. First of all, we restrict ourselves to the 3-body problem in two spatial dimensions; the overall system is six-dimensional. Moreover, we choose all masses to be dimensionless and all equal to one; $m_i = 1$. Further, we scale the system to $G = 1$ (Heggie & Mathieu (1986)). This results in

$$H(\vec{p}, \vec{q}) = \frac{1}{2}\vec{p}^T\vec{p} - \sum_{i=1}^{3}\sum_{j=1}^{i}\frac{1}{||\vec{q}_j - \vec{q}_i||_2}, \quad \text{with: } \vec{q}_i \in \mathbb{R}^2. \tag{3.19}$$

The data of the 3-body system is generated using the arbitrarily high-precision code `Brutus` (Boekholt & Portegies Zwart (2015a)). This enables us to make sure that the data the neural networks are trained on is actually the ground truth. Since the 3-body problem is a chaotic problem, the initial conditions are chosen such that the trajectories start with rather simple and easy to predict dynamics that become increasingly more chaotic over time. The initial conditions are the same as in the 3-body data of the papers introducing HNNs and SympNets (Greydanus et al. (2019b); Jin et al. (2020a)).

The random initial positions of the bodies are on a circle around the origin with the radius r drawn uniformly from $[0.9, 1.2]$. Furthermore, all three bodies have the same initial distance from each other. With these initial positions, one can calculate the momenta of the three bodies such that they would stay on the circle with radius r, orbiting the origin. These momenta are multiplied by different random uniform factors from $[0.8, 1.2]$ and chosen as initial momenta. With the initial conditions arranged in such a way, only small accelerations and slow exchange in kinetic and potential energy occur until $t \approx 5$. Afterwards, close encounters of the bodies may occur, leading to high accelerations and quick exchange in potential and kinetic energy. As in Jin et al. (2020a), 5000 trajectories are generated, a final time $t_{\text{end,train}} = 5$, and a step size $h = 0.5$ are used for training. In contrast to Jin et al. (2020a), a final time $t_{\text{end}} = 7$ is used for testing to check whether the neural networks can generalize from the simple behavior to the case with close encounters. The data is split into 80% training, 10% validation and, 10% test data.

The benefit of the high number of trainable parameters inside a multilayer perceptron becomes apparent in Figure 3.8. Inside the training data the multilayer perceptrons have an error about five times lower than the SympNets. However, the middle graph then directly reveals the most severe drawback of the physics-unaware multilayer perceptrons. After only two timesteps outside of training data, the error of the predictions by the multilayer perceptrons is higher than the error of the four physics-aware neural networks. Moreover, the error of the GHNNs inside the training data is five times lower than the error of the multilayer perceptrons. Hence, the GHNNs are highly accurate and able to generalize at the same time. Again, the HénonNets with one hidden layer per MLP have the highest error of all neural networks inside the training data and are able to generalize slightly better than the multilayer perceptrons outside the training data.

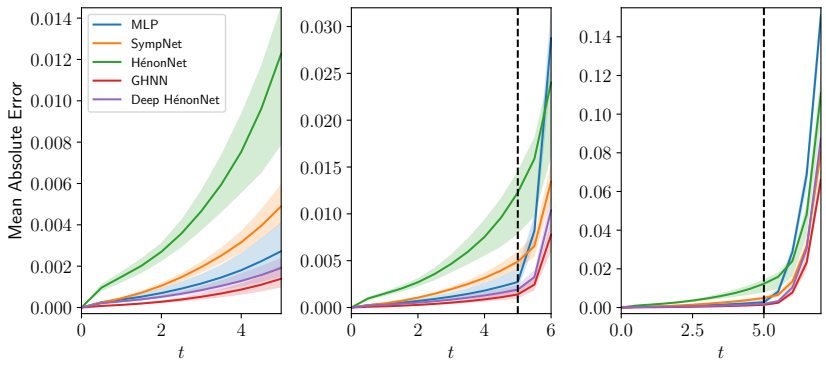


Figure 3.8: Comparison of the mean error over all the trajectories of the three bodies, in the test data for 50 MLPs, SympNets, HénonNets, and GHNNs. Additionally to the HénonNets with one hidden layer per learned V, the error of 50 HénonNets with two hidden layers is shown (Deep HénonNet). The lighter-colored areas indicate the variations between the differently initialized neural networks. The left graph shows the time range included in the training data. The middle graph shows the error inside the time range included in the training data and two steps outside, with the black dashed line indicating where the time range included in the training data ends. The right graph shows the errors along the full trajectories.

Deep HénonNets showed worse performance in similar experiments for the single and double pendulum (see original paper). In that case, the results is due to only three Hénon layers being used to achieve the same prediction speed as all the other neural networks. Since a large depth might be required by the structure-preserving neural networks in order to approximate certain symplectic maps, only three Hénon layers might be insufficient. For the 3-body problem, however, the roles are reversed. The deep HénonNets have a smaller error than all other methods except for the GHNN, both inside and outside the training data. This shows that to approximate the flow map of the 3-body problem, more complex updates of the state are of greater importance than many updates. For the same reason, the GHNNs outperform the SympNets by an order of magnitude.

3.6 Conclusion

We introduced an overarching framework that unifies three widely-used neural network architectures specifically designed to learn the symplectic flow maps of Hamiltonian systems: SRNNs, SympNets, and HénonNets. Additionally, this framework allows for the creation of novel neural network topologies that cannot be built from the individual architectures. These new Generalized Hamiltonian Neural Networks (GHNNs) combine the benefits and flexibility of all three aforementioned neural network architectures.

In the numerical experiment performed (single and double pendulum in the original paper and gravitational 3-body problem shown here), 50 neural networks of four different architectures (MLP, SympNet, HénonNet, and GHNN) were trained with the hyperparameters chosen such that a prediction is realizable at the same speed for all networks. In this experiment, our neural networks, the GHNNs, outperform all other neural networks for the single pendulum data. The extrapolation capabilities of the GHNNs are astonishingly good. The errors of the GHNNs and the SympNets are similar when trained on data for the double pendulum system. Of both, the errors are remarkably smaller than those of HénonNets and multilayer perceptrons. For the 3-body problem, the GHNNs have better accuracy than all other neural networks, with the errors being over one order of magnitude smaller than those of the SympNets. This drastic variation in the difference between the errors of the GHNNs and SympNets (also HénonNets and deep HénonNets) suggests that there are symplectic maps, where in order to approximate these, complex unit triangular updates are essential and others where many simple ones are the better choice. Due to the second layer in the MLPs of the GHNNs and deep HénonNets, far more complex unit triangular updates can be learned by these methods.

These numerical experiments show that a universal approximation theorem and a high number of trainable parameters are insufficient to guarantee good approximation capabilities. A universal approximation theorem can be proven for all three symplectic neural network architectures and the number of trainable parameters in the SympNets and GHNNs for the 3-body problem are almost the same. We conclude that it is important how the trainable parameters are arranged in the neural network. The choice of how to arrange the trainable parameters can decrease the prediction error more than one order of magnitude while preserving the same calculation speed.

At the same time, the HénonNets demonstrate that having too few trainable parameters also limits the accuracy of the predictions. In a comparison with a similar number of trainable parameters for all neural network architectures, as done in Burby et al. (2021), HénonNets might outperform other approaches. In addition, all neural networks in our numerical experiments were trained for many epochs to ensure that a good optimum was found in the loss landscape. This allowed for a fair comparison of the approximation capabilities of the different architectures. In Burby et al. (2021), the number of epochs is quite limited and the HénonNets have, opposite to our findings, a smaller error than the SympNets. This indicates that the additional structure in the HénonNets might be helpful in reaching a good approximation quickly but restricts the approximation when the neural network is trained for a long time.

Overall, the added symplecticity in all structure-preserving neural networks yields a better generalization and stability outside the training data. Moreover, the structure-preserving architectures achieve comparable or better accuracy than the multilayer per-

ceptrons with far fewer trainable parameters. This can be useful for the application to large Hamiltonian systems where large neural networks are needed and memory is limited.

Acknowledgment

This work was funded by the Dutch Research Council (NWO), in the framework of the program 'Unraveling Neural Networks with Structure-Preserving Computing' (file number OCENW.GROOT.2019.044).

Appendix

3.A Loss during training of the different architectures

The training behavior of the different neural network architectures is given in Figure 3.9, which shows that the SympNets and the GHNNs achieve a similar loss for the single

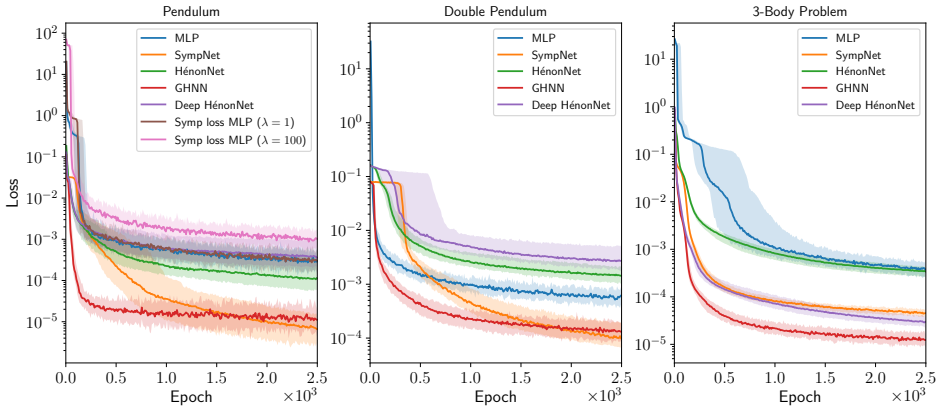


Figure 3.9: Comparison of the validation loss during the first 2500 epochs of training for all three Hamiltonian systems. The solid lines show the behavior of the median of the 50 neural networks trained per architecture. The lighter-colored areas indicate the variations between the differently initialized neural networks.

and double pendulum. However, the GHNNs need less than 500 epochs to reach a good minimum in the loss landscape, whereas the SympNets take around 1500 epochs to find a similar minimum.

In Figure 3.10, the loss is plotted over all 25000 epochs. It is clear that after 5000 epochs, the loss of almost all neural networks for all three Hamiltonian systems has nearly converged. After 5000 epochs, only the loss of the GHNNs on the pendulum data and the loss of the MLPs on the 3-body problem data keep improving significantly.

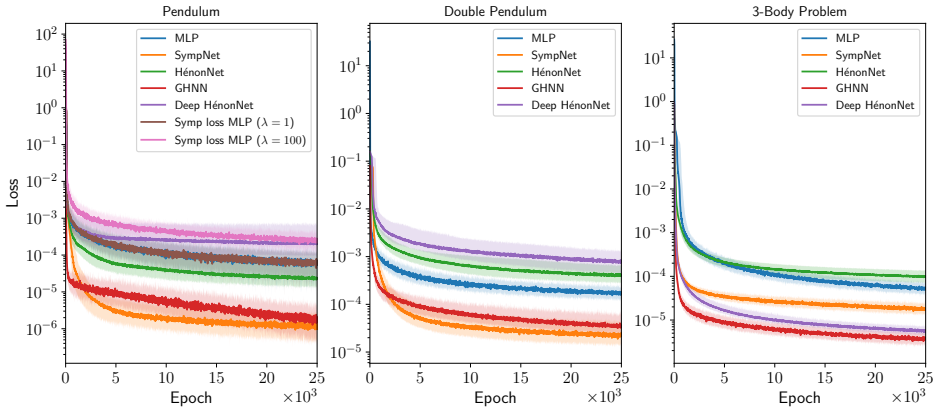


Figure 3.10: Comparison of the validation loss during the entire 25000 epochs of training for all three Hamiltonian systems. The solid lines show the behavior of the median of the 50 neural networks trained per architecture. The lighter-colored areas indicate the variations between the differently initialized neural networks.

4 Reinforcement Learning for Adaptive Time-Stepping in the Chaotic Gravitational Three-Body Problem

Work publicly available in Veronica Saz Ulibarrena, Simon Portegies Zwart, *[insert link arxiv]* currently under review for publication on *Communications in Nonlinear Science and Numerical Simulation*. Reprinted here in its entirety.

ABSTRACT

Many problems in astrophysics cover multiple orders of magnitude in spatial and temporal scales. While simulating systems that experience rapid changes in these conditions, it is essential to adapt the (time-) step size to capture the behavior of the system during those rapid changes and use a less accurate time step at other, less demanding, moments. We encounter three problems with traditional methods. Firstly, making such changes requires expert knowledge of the astrophysics as well as of the details of the numerical implementation. Secondly, some parameters that determine the time-step size are fixed throughout the simulation, which means that they do not adapt to the rapidly changing conditions of the problem. Lastly, we would like the choice of time-step size to balance accuracy and computation effort. We address these challenges with Reinforcement Learning by training it to select the time-step size dynamically. We use the integration of a system of three equal-mass bodies that move due to their mutual gravity as an example of its application. With our method, the selected integration parameter adapts to the specific requirements of the problem, both in terms of computation time and accuracy while eliminating the expert knowledge needed to set up these simulations. Our method produces results competitive to existing methods and improve the results found with the most commonly-used values of time-step parameter. This method can be applied to other integrators without further retraining. We show that this extrapolation works for variable time-step integrators but does not perform to the desired accuracy for fixed time-step integrators.

4.1 Introduction

Reinforcement Learning (RL) has recently been growing in popularity for multiple applications. Based on the positive results of RL as a control mechanism, we explore its use for the simulation of the motion of celestial bodies due to their mutual gravity. In these simulations, one of the most important choices to be made in advance is the time-step size. Integrators can often be classified by their time-stepping implementation. Although fixed-size integrators such as the leapfrog scheme Yoshida (1990a) are easier to implement, they lack the versatility of variable time-stepping methods Heggie & Hut (2003); Aarseth (2003). For example, Hut et al. (1995) presents a method to guarantee time symmetry in any integration scheme when applying variable time-step sizes and emphasizes the relevance of adapting the time-step size in dynamical computer simulations of particles. Examples of integrators that implement variable step-size include `Hermite` Makino & Aarseth (1992) integrator, which is frequently used for solving the general *N*-body problem. A more optimized time-stepping scheme based on the true dynamical time of the individual particles was designed by Zemp et al. (2007).

These integrators use characteristic values of the system to estimate the optimal time step size at each integration step. This estimation is currently not a physical quantity but an empirical approximation. For example, `Hermite` uses the free fall time of pairs of particles as an indication of how closely the particles in the system are interacting with each other and calculates an appropriate time-step, as explained in Aarseth & Lecar (1975); Pelupessy et al. (2012), and Capuzzo-Dolcetta et al. (2013). A more detailed description of the time-stepping implementation of `Hermite` is presented in Subsection 4.2.1. Even in these integrators where the time-step is determined at runtime, there is a control parameter that needs to be set manually before the start of the simulation. This so-called *time-step parameter* generally has a value between 10^{-4} and 1 and scales the internally determined integration time step. The wide range of this free parameter illustrates the complexity of making an expert-knowledge-based decision. Modern integration algorithms are geared to high efficiency while conserving energy. The automated method in which the time step is decided internally can perform excellently for the right choice of the time-step parameter. A poor choice will result in energy errors that are above the acceptable values, or unnecessarily computationally expensive calculations. A value of 10^{-2} is commonly used without a convergence study that determines its optimal value.

In traditional N-body calculations, the time-step parameter is fixed. While running over a relaxation timescale (or longer), the system's topology changes, rendering the predetermined time step parameter gradually less optimal. This results in degradation of the simulation results and loss of efficiency over time. Ideally, the time-step parameter should depend on the characteristics of the system, allowing it to be reduced during periods of high density, and increased in episodes of low density.

Despite the problem of deciding the time-step parameter being general, it is most notorious during 3-body interactions. We therefore use as an example application of our methodology a system of three equal-mass bodies that move under the influence of their mutual gravity. In the 3-body problem, long-lasting wide encounters (hierarchical triples) are intermixed with short-duration close encounters (democratic triples or scrambles) Stone & Leigh (2019).

The rapidly changing conditions imply that different step sizes are needed at different

points of the evolution. We overcome these drawbacks by using RL techniques for the automation of the choice of the time-step parameter during the simulation. We train the algorithm to find the optimum balance between accuracy and computation time.

Although this application of RL has not yet been explored in astrophysics, Dellnitz et al. (2023) used reinforcement learning to choose the optimum time step for the integration of systems with rapidly changing dynamics. They overcome the inefficiencies that may arise in complex systems such as chaotic systems by combining ODE solvers with data-driven time-stepping controllers. Their scheme outperforms recently developed numerical procedures in problems in which traditional schemes show inefficient behavior.

Most examples of RL in astronomy focus on the optimal control of telescopes Nousiainen et al. (2022); Jia et al. (2023); Yatawatta & Avruch (2021) but RL has also been used for other applications. Yi et al. (2023) explore the use of DeepQLearning for the prediction of major solar flares and find that the performance achieved is noticeably better than that of convolutional neural networks with a similar structure. Moster et al. (2021) uses an adaptation of reinforcement learning to make predictions without unlabeled data using a reward function.

A field that presents certain similarities to astrophysics is computational fluid dynamics (CFD). In this case, the use of reinforcement learning is still at an early stage but experiencing a significant growth Viquerat et al. (2022). Here, reinforcement learning can be used for applications such as drag reduction, shape optimization, and conjugate heat transfer. More related to our case are applications in which reinforcement learning is used to choose simulation parameters. For example, Novati et al. (2021) uses RL techniques to retrieve unknown coefficients in turbulence models. This field experiences challenges that can also be extrapolated to the case of astrophysical simulations. The computational cost of fluid mechanics simulations is generally high, which becomes a critical factor for the use of RL algorithms since classical methods suffer from low sample efficiency, i.e, they require large sample numbers to generate accurate results. Additionally, turbulence in most CFD simulations can possibly lead to a degree of stochasticity that hampers efficiency the training process. A similar application is found in robotics Krothapalli et al. (2011), in which reinforcement learning is used to decide the grid size (resolution) for robot navigation. The method aims to obtain finer grids next to obstacles for better resolution.

Using reinforcement learning for dynamically choosing simulation parameters in Astrophysics presents unique challenges. Fields such as fluid mechanics or weather prediction suffer from chaotic dynamics. Similarly in astrophysics, this can be perceived by the algorithm as stochasticity, which can make the training process less efficient. Additionally, computation time becomes a crucial consideration for simulations in which the number of bodies is large or a high accuracy is required. Other challenges, such as the large variability in the accelerations Ulibarrena et al. (2024) and the hierarchical structure of some systems can be mitigated with simplified examples like the equal-mass 3-body problem.

We study the use of DeepQLearning, a reinforcement learning technique, to learn the optimal choice of time-step parameter at each step of the integration of the gravitational 3-body problem. In Section 5.2 we present our study case and our method. In Section 5.3 we present the results of our experiment and show a comparison with modern methods. In Section 4.4.2, we show an example of other possible uses of our method by applying

it to a fixed-size integrator. The code is publicly available at `https://github.com/veronicasaz/ThreeBodyProblem_astronomy`.

4.2 Methodology

We present the method for integrating the motion of celestial bodies, the adopted initial conditions, our adopted DeepQLearning strategy, and the reward function.

4.2.1 Integration of *N*-body systems

The gravitational force exerted on a body *j* in a system of *N* point masses interacting via their Newtonian gravitational forces is given by Newton's equation of motion

$$m_j \frac{d^2 \vec{q}_j}{dt^2} = \sum_{k=0,\ k\neq j}^{N-1} G \frac{m_j m_k}{||\vec{q}_{jk}||^3} \vec{q}_{jk}, \qquad \vec{q}_{jk} = \vec{q}_k - \vec{q}_j. \tag{4.1}$$

Here (m_j) is the mass of particle j, $\vec{q}$ is the position vector, and G is the universal gravitational constant. The indices j and k represent the celestial bodies.

To calculate the evolution of a system of *N*-bodies, we use a numerical integrator. For our experiment, we adopt the fourth-order accurate predictor-corrector Hermite scheme Makino & Aarseth (1992). This integrator has a variable and individual block time-steps Nitadori & Makino (2008). Variable refers to the possibility of having different values of the time-step at different steps of the simulation and individual block refers to the clustering of time steps separated by factors of two and shared among the particles in the group. These time steps are not time-symmetric, but this can be approximately-solved with a small adaptation, Makino et al. (2006).

We adopt the implementation for Hermite integrator from the Astrophysical Multipurpose Software Environment (AMUSE) Portegies Zwart et al. (2009); Portegies Zwart & McMillan (2018). The code is called `Hermite`, and its time step is calculated from the free-fall time of the individual particles Hut et al. (1995). This variable represents the time it would take two particles to collapse due to their own gravitational attraction. The time-step is calculated according to Aarseth (1985); Nitadori & Makino (2008) in *N*-body units as a function of the accelerations $\mathbf{a}$ and its k derivatives ($\mathbf{a}^{(k)}$) as

$$\Delta t_i = \mu \left(\frac{|\mathbf{a}||\mathbf{a}^{(2)}| + |\dot{\mathbf{a}}|^2}{|\dot{\mathbf{a}}||\mathbf{a}^{(3)}| + |\mathbf{a}^{(2)}|^2} \right)^{1/2}. \tag{4.2}$$

Here $\Delta t_i \propto \mu$, where μ is the time-step parameter, and we adopt Einstein's convention for derivatives. μ is a design parameter that can be used to increase or decrease the accuracy of the simulation. A small value of μ leads to a small time-step, and therefore to higher accuracy but longer integration time. One criterion for our approach is to optimize the total integration time.

The other criterion is the validity of the calculation. This abstract measure, in some cases of a chaotic system, can be estimated using the energy error of the system. Any integrator for a problem that lacks an analytic solution introduces an energy error. Using

Table 4.1: Initial values of mass, position, and velocity for the three particles.

PARTICLE	$q_x\ (au)$	$q_y\ (au)$	$v_x\ (km/s)$	$v_y\ (km/s)$
1	0	0	0	10
2	[5, 20]	[0, 10]	-10	0
3	[-10, 0]	[5, 20]	0	0

this metric as an indication of accuracy is limited to cases in which all bodies have similar masses. In Section 4.5 we provide further discussion on the limitations and possible future improvements for this metric. We calculate the energy error by taking the difference between the total energies of the system initially and at time step i. The total energy E can be calculated as the sum of the kinetic E_k and the potential energy E_p of the system, and the energy error becomes

$$\Delta E_i = \frac{(E_{k,i} + E_{p,i}) - (E_{k,0} + E_{p,0})}{E_{k,0} + E_{p,0}} = \frac{E_i - E_0}{E_0}. \tag{4.3}$$

A large energy error is an indication of unphysical solutions, and the aim is to keep this error as small as possible.

4.2.2 Initial conditions: the 3-body problem

The gravitational 3-body problem is the lowest-*N* chaotic Newtonian dynamical system for celestial mechanics. In addition, any system of three bodies will eventually lead to the ejection of one body, leaving the other two bound in a pair Heggie (1975). Due to the combination of finite lifetime, chaotic behavior, and rapidly changing topologies, the gravitational 3-body problem represents an ideal study case to test and validate our RL strategy.

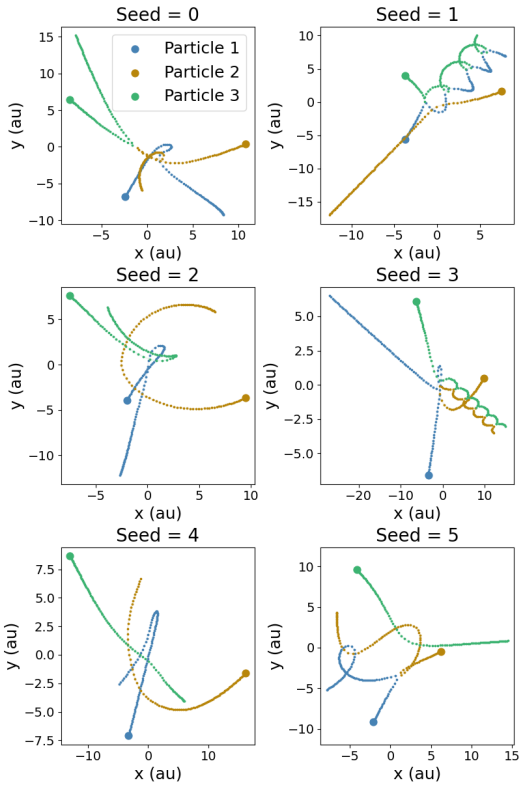
Since systems of three or more bodies are dynamically unstable and notoriously chaotic, any energy error inevitably leads to a different outcome. This was demonstrated in Boekholt & Portegies Zwart (2015), where they adopted the arbitrary precise *N*-body code Brutus, and argued that a relative energy conservation of 10^{-4} suffices to preserve the physics. We adopt the same criterion here.

A proper choice of the time step is paramount for a reliable integration, small energy error, and acceptable runtime. Close encounters, which are inevitable, ensure that the integration time step has to vary over several orders of magnitude in order to comply with our main objective: find the largest possible time step for an acceptable energy error.

The 3-body problem is characterized by the masses, positions, and velocities of the three particles at some moment in time.

For clarity, we choose the masses to be identical and equal to one solar mass ($1\ M_\odot$) and we limit ourselves to two dimensions (x and y). One particle is initialized in the center of the reference frame of the three bodies with zero velocity. The two other particles are positioned randomly following a uniform distribution with a fixed velocity according to the limits dictated in Table 4.1. We keep astronomical units (au) as a baseline for position, $M_\odot$ for mass, and km/s for velocity.

In Figure 4.1, we show the evolution of six different realizations of this system for 100 steps. Each system has an initial seed, as indicated in the various panels, and the runs are performed with $\mu = 10^{-4}$. The center of mass of the system has been initially positioned at the center of the reference frame. Each of these six systems behaves fundamentally differently on the short time scales of the integration, but more importantly, close encounters are alternated with wide excursions. After a finite time, the systems dissolve into two bound particles and a single escaping particle. This is most noticeable in the examples with seeds 1 and 3.



Particle	q_x (au)	q_y (au)
Seed 0		
2	13.232	7.152
3	-6.028	13.173
Seed 1		
2	11.255	7.203
3	-1.14$\times 10^{-3}$	9.535
Seed 2		
2	11.540	0.259
3	-5.497	11.530
Seed 3		
2	13.261	7.081
3	-2.909	12.662
Seed 4		
2	19.505	5.472
3	-9.727	15.722
Seed 5		
2	8.330	8.707
3	-2.067	18.779

Figure 4.1: Different initializations run for 100 steps with a time-step parameter of 1×10^{-4}. The initial state is marked with a circle. On the right, we show the initial values associated with each random seed.

4.2.3 Deep QLearning

The adopted Reinforcement Learning method interacts with the environment, in our case the integration of the 3-body system, in order to select the optimal time-step parameter at each step. To understand the challenges associated to the use of RL in astrophysics simulations, we use a method that is interpretable and does not require the choice of a large

number of training parameters. We adopt Q-learning for its relative simplicity and demonstrated efficiency compared to other RL methods Sutton & Barto (2018). Additionally, it has been shown to remain effective in stochastic environments (see for example Hung & Givigi (2016)).

Q-learning is a method that optimizes itself to choose an action A which maximizes a reward value R. It is a model-free method that learns through trial and error Sutton & Barto (2018) an action-value function Q that approximates the optimal action-value function Yi et al. (2023). In contrast to other popular methods, Q-learning focuses on exploration of the best possible actions and generally shows faster convergence Zaghbani et al. (2024).

In our problem, the action represents the choice of time-step parameter. The action space is discrete of size 10, where action 0 corresponds to a time-step parameter of $\mu = 10^{-4}$ (slowest and most precise calculation) and action 9 to $\mu = 10^{-1}$ (fastest and least accurate). For the other actions, μ has a value logarithmically spaced between these two extremes. Using a discrete action space means that the RL algorithm is only aware of the action number, and not of the value of the time-step parameter associated with it. This allows us to be able to expand to other integrators for which the time-step calculation is implemented differently (see Subsection 4.4.1). The choice of the number of actions in this work is made to allow for a simple interpretation of the results. This number can be increased to allow for a more refined algorithm.

The observation space (S) corresponds to the state of the system: positions, velocities, and masses of the three particles in the system. We add the current energy error in the observation space to allow the system to also account for the current accuracy of the simulation. For the state vector, we convert the positions and velocities to dimensionless units with base 1 au for the distance and the sum of the masses of the bodies for the mass Hénon (1971). Although a different choice of the observation space could lead to a better performance of the algorithm, we choose Cartesian coordinates to avoid expert bias in the algorithm and leave the finding of a better state vector for future work.

We adopt an extension of Q-learning called Deep Q-networks since our problem requires a continuous observation space to account for the state of the particles being continuous. A deep Q-network (DQN) combines reinforcement learning with deep neural networks Mnih et al. (2015) which are used to approximate the optimal value of the Q-function

$$Q(S, A) = \max_{\pi} \mathbb{E}\left[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...\right], \tag{4.4}$$

which is the maximum sum of the rewards multiplied by the discount value γ at each time step t. This maximum is chosen following a behavior policy $\pi = P(A|S)$ after making an observation S with an action taken A. To balance the trade-off between exploration and exploitation, the algorithm includes a stochastic exploration strategy called ϵ-greedy, by which a random action is chosen with probability p instead of the one selected by the algorithm Viquerat et al. (2022). This value ϵ is reduced during the training to favor exploitation over exploration. To avoid the inherent instabilities of RL, the method uses experience replay, which stores the data in a training database and randomizes the chosen training sample to eliminate correlations in the observation sequence Mnih et al. (2015).

Deep Q-networks make use of a neural network instead of a Q table. In order to avoid instability and variability during training Yu et al. (2018), it is common to use two

different networks: the QNet and the Target network. The first is updated at every step of the training algorithm, whereas the weights of the Target net are only updated with the ones of the QNet after multiple steps. The predictions to further select the best Q-value are made with the target net, allowing for further stability of the results than with only one network. Both networks have the same architecture and their only difference is the frequency at which their weights are updated.

We implement our deep Q-learning method using `PyTorch` Imambi et al. (2021) with an environment created using `Gym` Brockman et al. (2016b).

4.2.4 Reward function

The reinforcement learning algorithm will learn to choose the optimum strategy for each specific realization of the system. We define this optimum based on the balance between accuracy and computing time.

Our reward function consists of two terms: the first accounts for the total energy error of the system at a given step, and the second one for the computing time. We discuss the comparison between different expressions and weights in 4.A. The chosen form for the reward function is

$$R = -W_0 \frac{1}{\text{step}} \frac{\log_{10}\left(|\Delta E_i| \,/10^{-10}\right)}{|\log_{10}(\Delta E_i)|^3} + W_1 \frac{1}{|\log_{10}(\mu)|}, \tag{4.5}$$

where $W_{[0,1]}$ represents the weights of the reward function, which are design choices. The first term in Equation 4.5 represents a decreasing slope with the value becoming zero when the energy error approaches $\Delta E = 10^{-10}$, as can be seen in Figure 4.2. We design this term so that the reward obtained by achieving energy errors below 10^{-10} has a smaller slope than for errors above 10^{-10}. We achieve this by dividing the first term by the cube of the logarithm of the energy error. Additionally, we divide this term by the current integration step. By doing this, a large energy error is penalized more at the beginning of the integration than at further steps. If the energy error grows to a larger value early in the simulation, this error will continue to accumulate. By adding this term, we aim to prevent early-on large energy errors. The second term accounts for the computational effort by penalizing the use of a small value of μ.

In Figure 4.2, we present the value of the reward function for different steps and initializations as a function of the total energy error at that step (top panel), and as a function of the computation time of the step (bottom panel). We observe from the first plot a decreasing trend of R for larger values of the energy error. For equal values of the energy errors, a larger μ yields a larger value of R to account for the computation time. In the bottom panel we observe that there is no clear trend for the computation time as the energy error is the dominant term.

The weights have been chosen to balance the values of those two terms and can be found in Table 5.2. A comparison of different choices is also found in 4.A. W_0 is chosen to be 3,000 to balance the energy error and computation time terms. This is the only weight that has to be actively chosen during the training. With this value of W_0, as can be seen in Figure 4.2, a maximum can be achieved with either a low energy error or a large time step parameter. The value of W_1 is fixed to 4 for the second term to range between 1

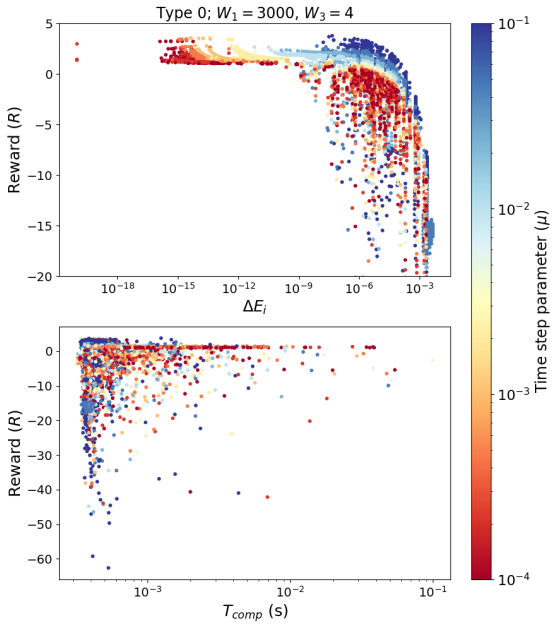


Figure 4.2: Reward value as defined in Equation 4.5. The top plot shows the reward as a function of ΔE_i, and the bottom plot as a function of the computation time. The color is the size of the time step parameter.

and 4 for simplicity of the interpretation of the results. However, this value could be fixed to 1 instead, and W_0 would remain the only variable to be tuned.

4.2.5 Method setup

In Figure 4.3 we show the DQN algorithm used. Starting with an initialization of the system at time i (as shown in Subsection 4.2.2), we evolve the state (X) for a time (Δt), which results in the state of the system X_{i+1} at time $i + 1$. This means that we find the positions and velocities of the particles in the system at a later time. To create a training database, we take one sample from each evolution step. The sample is formed by the State (S) and the reward (R) associated with the given action (A) at step t. The reward is calculated using the energy error for that step (as in Equation 4.3) and the time-step size as explained in Subsection 4.2.4. The state (S) that is fed to the training agent is formed by the positions, velocities, and masses of the bodies (X) as well as the energy error (as explained in Subsection 4.2.3)

To train the algorithm, we generate a random batch from the training database generated and use it to update the weights of the QNet. After a given number of training steps (t), the Target network is updated with the weights of the QNet. The current state S is used as input to the Target net to predict the Q-values associated with every possible

action in the action space. Then, the action with the largest Q-value is selected to evolve the system for the next time step, and this process is repeated recursively. In some cases, a random action is taken instead of the predicted one to allow escaping local minima.

When the energy error of the simulation reaches a tolerance value of 10^{-1} or the maximum number of steps per episode is reached, the simulation is terminated and a new set of initial conditions is chosen to restart the process for the next episode.

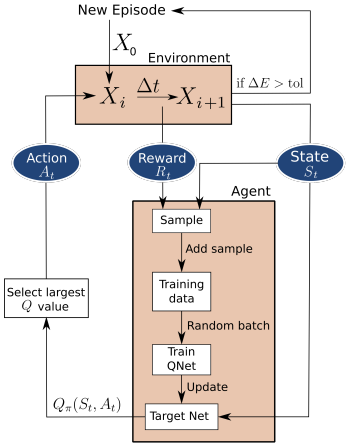


Figure 4.3: Schematic of the DeepQL setup. For each episode, the state of the system is evolved using the selected action, and together with the energy error and state, the QNet and the Target Networks are trained and updated, respectively, to select the optimal action.

4.2.6 Training parameters

The training parameters are selected based on manual experimentation due to the large computational resources needed for an automatized hyperparameter optimization procedure. The main hyperparameters and simulation parameters are shown in Table 5.2. The first rows correspond to the maximum number of episodes, steps per episode, and the energy error tolerance to stop each episode, as explained in Subsection 4.2.5. The latter is chosen based on the assumption that a larger energy error indicates an unphysical solution. The maximum number of steps is chosen to be 100 as it is in that interval of time when most close encounters occur, but this number could be made arbitrarily large. After a time equal to Δt, we evaluate the physical properties of the N-body system and choose a new action accordingly. The value of Δt differs from the internal time-step size (see Equation 4.2). The last rows in Table 5.2 correspond to the number of actions chosen to train the algorithm, the minimum and maximum values of the time-step parameter μ (Subsection 4.2.3), and the weights of the reward function (Equation4.5).

In Figure 4.4, we show the evolution of the training process at each episode using the cumulative reward, the average reward, the slope of the energy error, and the final energy

Table 4.2: Training and simulation parameters.

GLOBAL SEARCH	
MAX EPISODES	2,000
MAX STEPS PER EPISODE	100
ΔE TOLERANCE	1×10^{-1}
Δt	1×10^{-1} (YR)
HIDDEN LAYERS	9
NEURONS PER LAYER	200
LEARNING RATE	1×10^{-3}
BATCH SIZE	128
TEST DATA SIZE	5
NUMBER OF ACTIONS	10
$[\mu_{\text{MIN}},\ \mu_{\text{MAX}}]$	$[1 \times 10^{-4},\ 1 \times 10^{-1}]$
$W_{[0,1]}$	$[3,000\ ,\ 4.0]$

error. We can observe large oscillations due to the presence of random functions in the training procedure and the large differences in behavior between the various initializations. We show in black a fit of these oscillations. However, due to the chaotic nature of the problem, it is not possible to use these metrics as a measurement of the quality of the training procedure.

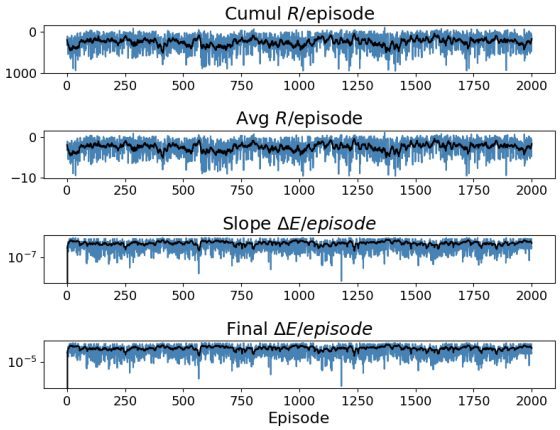
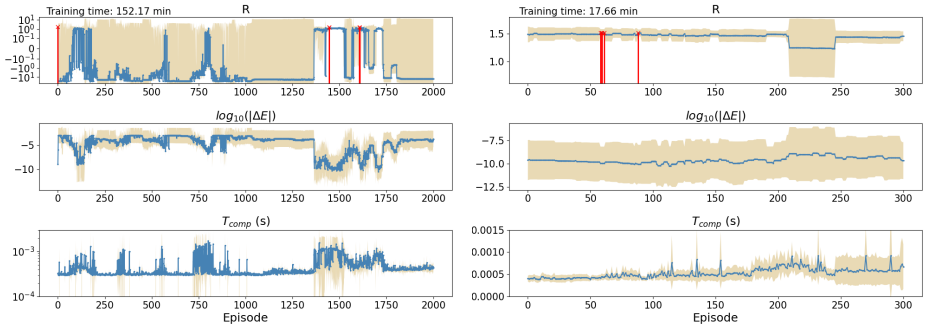


Figure 4.4: Evolution of the cumulative reward per episode (first row), the average reward per episode (second row), the rate of growth of the energy error per episode (third row), and the final energy error per episode (fourth row).

To overcome this problem, we create a test dataset that contains a set of initial conditions. At the end of each training episode the performance of the RL algorithm is tested on those initializations and the results are saved as a measurement of the evolution of the training procedure. In this way, we prevent randomness in our evaluation of the training

by using the same initial realizations at each episode and therewith create a robust testing method. It is important to mention that this method incurs in additional computation time. Depending on the needs of the experiment, the testing can be done every X episodes to save computational effort. The size of the test data used is specified in Table 5.2. In Figure 4.5a, we show the same training as in Figure 4.4 but with our testing method. We plot the average and standard deviation of each of the metrics for the testing samples. It is now possible to distinguish between the models with high reward and those which perform poorly. We then mark with a red line those episodes in which the reward achieved is the largest. We explore those models and others with similar reward values and choose the model at episode 1444 as our best-performing candidate.



(a) **(b)**

Figure 4.5: Evolution of the average (blue) and standard deviation (orange) of different metrics of the test dataset per episode of: the reward value (first row), the energy error (second row), and the computation time (third row) for the global training **(a)** and for the local training performed after the global one **(b)**.

The training time changes depending on the actions taken at each episode and the test data size. With our settings, training for 300 epochs takes 17 min (on an AMD Ryzen 9 5900hs). The training times are included in the figure corresponding to the training evolution (Figure 4.5a).

The reward function landscape is made of many local maximums. Since this function is composed of two terms, a local maximum could be one of the solutions in which the algorithm always chooses the most restrictive actions. In this case, the energy error term would be maximized at the cost of computation time. On the other hand, another local maximum could be the choice of the least restrictive action to maximize the computation time term. We aim to obtain an algorithm that can balance those two, and it is therefore essential to choose the correct weights for the reward function. Even when the weights are optimal, the random nature of the training may lead the algorithm towards one of these local maximums. We therefore repeat the training multiple times with different seeds until the maximum reward represents our objectives. The model chosen comes from a training with an initial weight distribution corresponding to seed 1.

The model chosen is capable of identifying certain features of the integration and adapting to it. However, we did not find the result fully satisfactory. Therefore, we perform a local search starting from the weights of the selected model. Using the training

Table 4.3: Training and simulation parameters of the local search

LOCAL SEARCH	
MAX EPISODES	300
LEARNING RATE	1×10^{-8}
TEST DATA SIZE	5

conditions shown in Table 4.3, we further train the RL algorithm for 300 episodes with a lower learning rate to find solutions that are close in the optimization landscape to the one obtained with the global search. This training is shown in Figure 4.5b. We evaluate the models at the episodes with the largest reward value and take the best-performing one. We choose the model at episode 88 for our final results. This procedure could be further repeated to obtain consecutively better results, but on account of the computation time limitation, we restrict ourselves to one local search.

4.3 Results

In this section, we discuss the results of the integration of a system of three particles using the trained RL algorithm and compare it with the canonical approach without retraining.

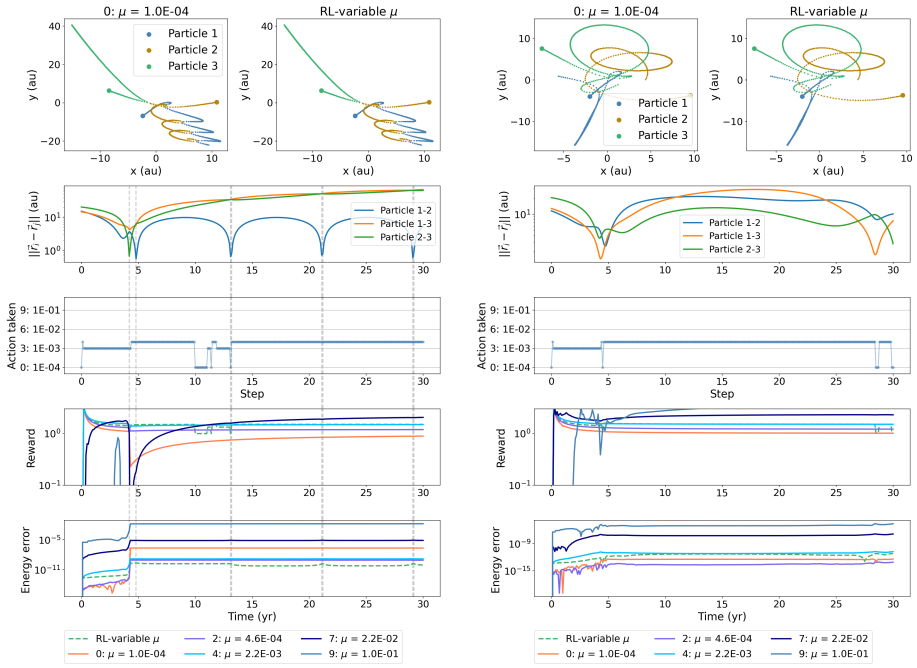
4.3.1 Integration results

Once the algorithm has been trained using the parameters shown in Subsection 4.2.6 the model can be used repeatedly without the need for retraining or without any expert knowledge involved. We aim to obtain an algorithm that chooses small values of the time-step parameter when close encounters occur, and large values when the particles are farther away from each other. One remaining challenge is the gradually growing energy error, and the fact that once increased, is it not likely to decrease by large amounts. We mitigate this problem by including the energy error in the state S of the DQN algorithm. The algorithm then takes the relative local error and the global energy error into account for making a decision.

To illustrate the working of our trained RL algorithm, we evolve the initial conditions (see Figure 4.1) for 300 time steps (30 years). Since our algorithm was trained for the integration up to 100 steps, training to 300 steps allows us to also understand the extrapolation capabilities of the trained model. The results are presented in Figure 4.6a, Figure 5.13, and Figure 4.7.

In the first row, we show the trajectory of the three particles using a fixed value for $\mu = 10^{-4}$ (left), and the results of the RL algorithm (right). They look indistinguishable. In the second row, we present the pairwise mutual distance between the three particles. This is a good method to identify close encounters. The third row gives the action chosen by the RL algorithm for each time step. The last two rows represent the reward and the energy error for each study case, respectively.

We can observe how the trained RL model correctly identifies close encounters and adopts a more restrictive action to prevent a large jump in energy error when necessary.



(a) **(b)**

Figure 4.6: Comparison of fixed-size time-step parameters with our RL model for 300 time steps (30 years). We present the trajectory in Cartesian coordinates (top row panels), the pairwise distance between particles (second row), the actions taken by the RL algorithm (third row), the reward for each case (fourth row), and the energy error at each time step for each study case (fifth row), for initializations with Seed 0 **(a)** and Seed 2 **(b)**.

When the distance between the particles does not correspond to close encounters, our trained model increases the time-step parameter, leading to faster calculations with acceptable accuracy. The selected action is smaller during close encounters and larger when the system is hierarchical.

Our initial intuition led us to think that the optimal RL algorithm would select a less restrictive action unless there is a close encounter, in which case a more restrictive action is needed. Therefore, we expected lower peaks in the action plot (third row of Figure 4.6a) when the bodies have a close encounter. However, studying our trained RL model, we observed that the high-reward solutions many times skip some of those peaks without an increase in the energy error. If we take into consideration that the energy error will not decrease substantially as the simulation progresses, we realize that contrary to our initial intuition, choosing a smaller time-step parameter might not be the optimal choice once our simulation has advanced. As an example, if at step 50 our energy error is on the order of 10^{-7}, we need to choose a less restrictive action during the close encounter to keep the energy error lower than that. However, if the energy error after 50 steps is 10^{-4}, and the cost of choosing a less restrictive action is of the same order, choosing a restrictive action

to achieve a lower energy error will not have a noticeable effect in the global simulation.

We show the results of our trained RL algorithm for three representative initializations of the 3-body problem. For the result with Seed 0 in Figure 4.6a, we observe how the algorithm correctly identifies close encounters and adapts by choosing a lower action. In some close encounters, this behavior is not seen. As explained before, choosing a lower action is only necessary depending on our current energy error. In some cases, choosing for a lower action will not lead to an improvement of the accuracy. This is the case, as we can observe that even by skipping a close encounter, the energy error still remains constant throughout the simulation. Additionally, this simulation is run to 300 steps (30 years), whereas the training is done with simulations up to 100 steps, which speaks of the excellent extrapolation capabilities of our method. In the case of Seed 2 shown in Figure 5.13, the close encounters are less pronounced, but the algorithm still identifies them correctly and chooses a lower time-step parameter accordingly. Finally, in the results for Seed 3 (Figure 4.7) we can observe a close encounter between the three bodies followed by consecutive close encounters between two of them. In this case, the algorithm adapts at each encounter to prevent the energy error from increasing. This leads to a fast oscillation of the actions taken at each step.

To better appreciate the performance of the trained RL algorithm, we present in Figure 4.8 the energy error and computation time at the end time of 100 initializations run for 300 steps. We compare the RL algorithm (orange) with a rather inaccurate fixed value $\mu = 10^{-1}$ (blue), and with a rather accurate fixed value $\mu = 10^{-4}$ (green). As expected, the calculations for $\mu = 10^{-1}$ produce larger energy errors but require lower computation effort, whereas the calculations for $\mu = 10^{-4}$ are generally more precise at a larger computational cost. The majority of the calculations with $\mu = 10^{-4}$ are unnecessarily accurate as they conserve energy better than 10^{-10}, but at a cost of more than twice the computation time of our RL solution. It should be noted that the computation time refers to the integration time, and does not include the prediction time of the network. In most problems, this time is negligible compared to the time it takes to integrate the system. We additionally show the results for the case with μ closest to 10^{-2} since that is the most commonly used value in astronomy simulations. Our RL algorithm achieves better energy errors than this commonly-used value while avoiding the large computation times corresponding to a low value of μ. Additionally, it does so without any expert knowledge needed to set up the simulation. This result is promising, further fine-tuning of the training procedure could lead to even better results.

4.4 Generalization capabilities

In this section, we discuss the generalization potential of our method when choosing a different integrator.

4.4.1 Generalization capabilities: different integrators

The RL algorithm is set up so that it can be applied to any integrator without the need to modify its source code. This means that applying it to different integrators is effortless. Therefore, we study the generalization capabilities of our method when applied to other

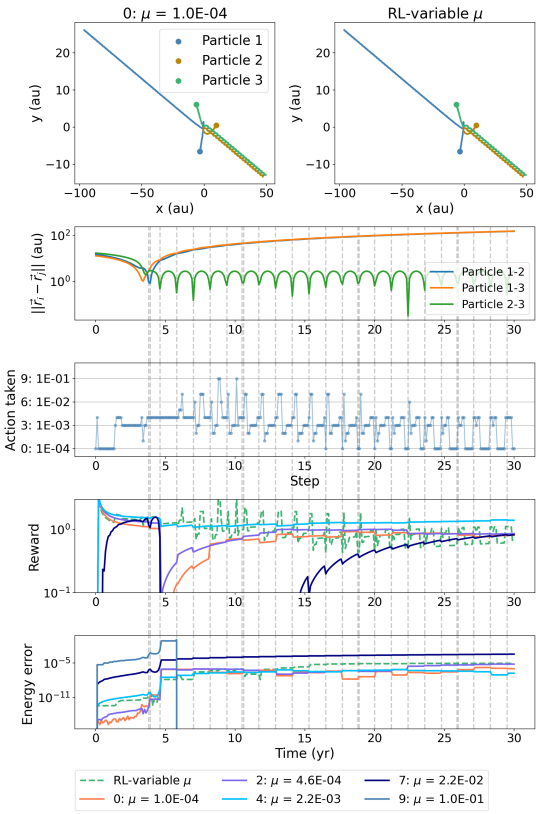


Figure 4.7: Comparison of fixed-size time-step parameters with our RL model for the initialization with Seed 3 run for 300 time steps (30 years). We present the trajectory in Cartesian coordinates (top row panels), the pairwise distance between particles (second row), the actions taken by the RL algorithm (third row), the reward for each case (fourth row), and the energy error at each time step for each study case (fifth row).

integrators.

The trained algorithm chooses an action ranging from 0 to 10, where the first is the most accurate time-step parameter and the second is the least. Some integrators like `Huayno` Pelupessy et al. (2012) also use the time-step parameter, whereas others can use other parameters to determine the size of the time step. For example, the `Symple` integrator allows choosing the order and the time step size. We use the trained algorithm for these two other integrators with the ranges of actions in Table 4.4 and compare their performance. We also adopt the implementation of these algorithms from AMUSE Portegies Zwart et al. (2009); Portegies Zwart & McMillan (2018).

The ranges for the actions are chosen for the integrators to have similar values of

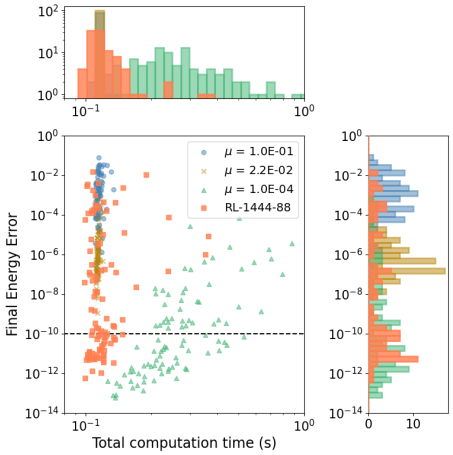


Figure 4.8: Distribution of computation time and energy error at the last step for 100 initializations run for 300 steps (30 years). Each color represents one study case, from the most inaccurate fixed value in blue ($\mu = 10^{-1}$), to the most accurate ones ($\mu = 10^{-4}$) in green. The results for the RL algorithm using variable μ are presented in orange.

Table 4.4: Ranges of actions for different integrators

	VALUE	MINIMUM VALUE	MAXIMUM VALUE
HERMITE	μ	10^{-4}	10^{-1}
HUAYNO	μ	10^{-5}	10^{-1}
SYMPLE	Δt	10^{-7}	10^{-2}

computing time and energy error for their least and most accurate actions. We see in Figure 4.9 the final energy error and computing times for 100 initializations integrated for 100 steps. We observe how `Symple`, even with the most accurate time-step size, has a large range of final energy errors due to its fixed time-step size. It therefore requires a small time-step size to achieve similar energy errors as with `Hermite`, which leads to a large computation time. Similarly, we choose a smaller lower value for the time-step parameter, which leads to the simulations being slightly more computationally expensive.

We evolve the system with seed 0 for 300 steps (30 years) with the aforementioned integrators. In Figure 4.10a, we show the actions taken for each integrator and the energy error incurred. Similarly to `Hermite`, the RL algorithm with `Huayno` can identify close encounters and adopt a more restrictive action to keep the energy error reasonably constant during close encounters, although the final energy error is approximately two orders of magnitude larger. From the results obtained with `Symple`, we can conclude that the trained algorithm does not extrapolate well to fixed time-step algorithms as it reaches the maximum allowed value of energy error early in the simulation.

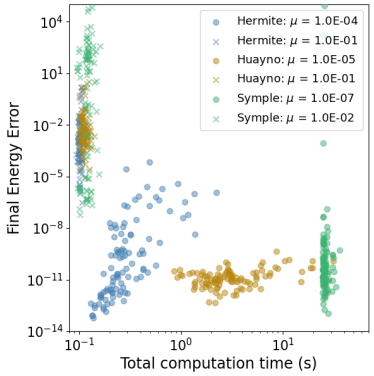


Figure 4.9: Comparison of the final energy error and computation time for the least and most accurate actions with `Hermite`, `Huayno`, and `Symple` integrators run for 100 steps.

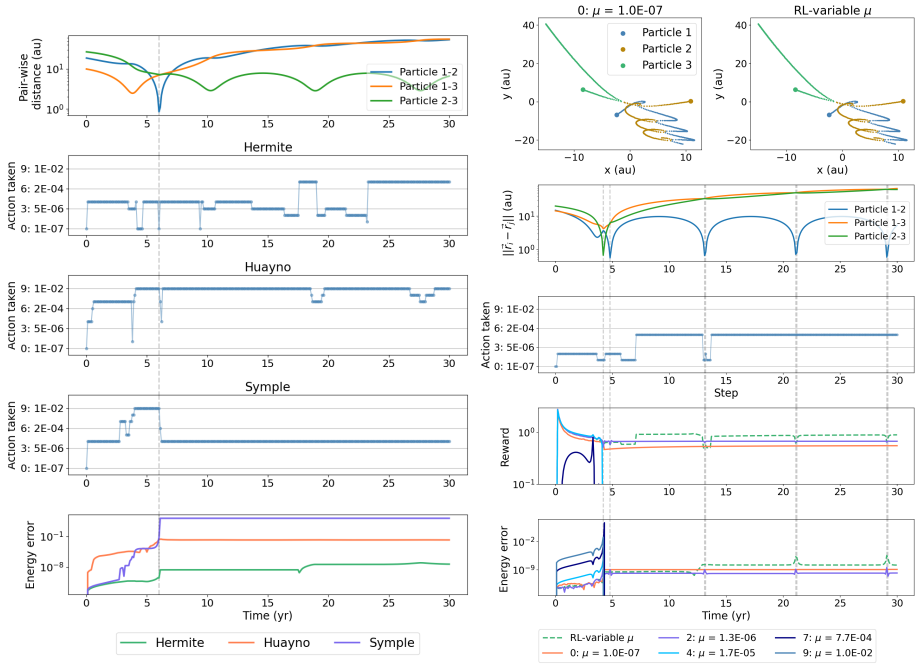
Table 4.5: Training and simulation parameters for `Symple`

GLOBAL SEARCH	
MAX EPISODES	3000
LEARNING RATE	1×10^{-3}
TEST DATA SIZE	5
NUMBER OF ACTIONS	10
$[\mu_{\text{MIN}},\ \mu_{\text{MAX}}]$	$[1 \times 10^{-7},\ 1 \times 10^{-2}]$
$W_{[0,1,2]}$	$[3,000\ ,\ 4]$

4.4.2 Training and integration with Symple

In Subsection 4.4.1, we learn that the trained model does not extrapolate well for the `Symple` integrator. This was to be expected since `Symple` does not have an internal integration calculation and the RL method is directly choosing the time step instead of the time-step parameter. This level of extrapolation cannot be expected from current RL techniques. Although the trained model does not extrapolate, we show that our method can be extrapolated without the need for changes in the reward function or other RL parameters. To do so, we use the same method to train a new model suited for `Symple`. Similarly, this method could be applied to other integrators and problem setups. We train a new model with the parameters shown in Table 4.5. The training metrics and the results of applying the model to the integration with `Symple` can be found in 4.B.

We apply the trained model to the evolution of the initialization with seed 0 for 30 years. The results in Figure 4.10b can be compared with those in Figure 4.10a. In contrast with the model trained for `Hermite`, this new model can correctly identify close encounters and adapt the actions taken accordingly. Whereas the least restrictive cases with fixed time step reach the maximum allowed energy error early in the integration, our trained model is able to adapt the actions to finish the integration with a low energy error.



(a) **(b)**

Figure 4.10: **(a)** Comparison of actions taken and energy error for three different integrators. The top panel shows the pairwise distance between bodies. **(b)** Comparison of fixed-size time step with our RL model for the initialization with Seed 0 for 300 time steps (30 years). We present the trajectory in Cartesian coordinates (top row panels), the pairwise distance between particles (second row), the actions taken by the RL algorithm (third row), the reward for each case (fourth row), and the energy error at each time step for each study case (fifth row).

Finally, we show the distribution of computation time and energy error for 100 initializations run for 300 steps for `Symple` in Figure 4.11. The main problem with using `Symple` integrator is the large spread in final energy error for different initializations. By not having an adaptable time-step the accuracy of the simulation will depend on how chaotic each scenario is. We show that our RL model can achieve results that are mostly concentrated around low energy values, but the computation times achieved are not good enough to consider the trained model to be competitive. We believe that further training with optimized training parameters and a larger network would lead to a significant improvement of these results. We will leave that experiment for future work.

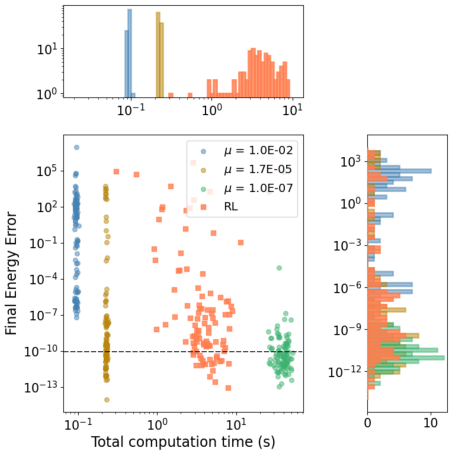


Figure 4.11: Distribution of computation time and energy error at the last step for 100 initializations run for 300 steps. Each color represents one study case, from the most inaccurate fixed value in blue, to the most accurate one in green. The results for the RL algorithm using variable μ are presented in orange.

4.5 Discussion and conclusions

We designed a method in which an agent is trained to choose the best time-step parameter for the simulation of the chaotic 3-body problem. We have overcome some main disadvantages of modern integrators, such as our baseline `Hermite` integrator. Firstly, we eliminate the need for expert knowledge when choosing this parameter. Secondly, we allow it to change during the simulation to adapt to the rapidly changing conditions of the problem. Finally, the algorithm is trained to balance computation time and accuracy.

The reward function determines the correct functioning of the method. Currently, we use the energy error as a metric of the accuracy of the simulation. Although this applies to the case shown here, when the ratios of the masses of the bodies are different, the energy error becomes an unreliable metric for the correctness of the dynamics. In this case,

the energy contribution of the bodies with low mass is orders of magnitude smaller than that of larger bodies. Our method could be made applicable to the hierarchical N-body problem by changing the energy term of the reward function with another metric Rauch & Holman (1999); Pham (2024).

We have shown our implementation of the reward function containing two different terms to account for the energy error and computation time. We observe that our method is capable of achieving better energy conservation than most of the fixed time-step parameter cases for equivalent values of energy error and computation time. Our trained network can be used without the need for retraining in the case of other integrators. However, the performance achieved for fixed time-step size integrators is worse than that for variable time-step size methods. Although the final energy error for integrators other than `Hermite` is larger, by retraining our model we can adapt its use to other integrators, such as `Symple`, to achieve better performance with few training episodes. We therefore show how our method can be easily generalized to other integrators without the need to adapt their implementation. Similarly, this framework can be used in many other similar problems with the necessary adaptations in reward function. Additionally, the state vector depends on the number of bodies in the system. In order to apply this method to higher-N problems, the model would have to be retrained. Overcoming this issue is left for future work.

Currently, our method performs the observation and chooses the action after a check time Δt. In future work, we aim to eliminate this requirement and perform the check after every internal time step. By doing this, the integration becomes more flexible and the time-step parameter can be adapted after smaller steps when close encounters occur. Furthermore, the hyperparameter optimization was done via manual experimentation. In our opinion, a systematic approach would yield better training results and help improve the final performance of the algorithm. Additionally, more complex RL methods could also contribute to an improvement in the method performance.

Although the results shown are preliminary, they open a new scientific opportunity for the inclusion of Machine Learning into astronomical simulations. The method presented is general enough to be applied to a variety of integrators and cases without major changes in the method itself and shows to be promising in eliminating the need for expert knowledge for setting up a simulation.

4.6 Acknowledgments

This publication is funded by the Dutch Research Council (NWO) with project number OCENW.GROOT.2019.044 of the research programme NWO XL. It is part of the project “Unraveling Neural Networks with Structure-Preserving Computing". In addition, part of this publication is funded by the Nederlandse Onderzoekschool Voor Astronomie (NOVA).

Appendix

4.A Reward function comparison

Finding the right reward function for the reinforcement problem is vital to obtain optimal results. We try different combinations of the two terms: the first being the energy error at a certain step, and the second the time step parameter to account for the computational time. Those are shown in Equation 4.6 and Equation 4.7:

$$\text{Type 1:} \qquad R = -W_0 \frac{(A+10)}{|A|^3} + W_1 C, \tag{4.6}$$

$$\text{Type 2:} \qquad R = -W_0 A + W_1 C, \tag{4.7}$$

where

$$A = \log_{10}\left(|\Delta E_i|\right), \qquad B = \frac{1}{|\log_{10}(\mu)|}. \tag{4.8}$$

We show the behavior of each of those as a function of the energy error, and the computation time in Figure 4.12. Type 1 yields larger values of the reward for low energy errors and for high values of the time step parameter (shown in blue). The reward value rapidly decreases as the energy error rises to unphysical values. Type 2 presents a structure in which the maximum rewards are achieved by low energy errors, but a low computation time cannot lead to maximum values of the reward during the training procedure.

For our training, we choose type 1 (Equation 4.6 and Equation 4.5) since it shows the best adherence to our requirements.

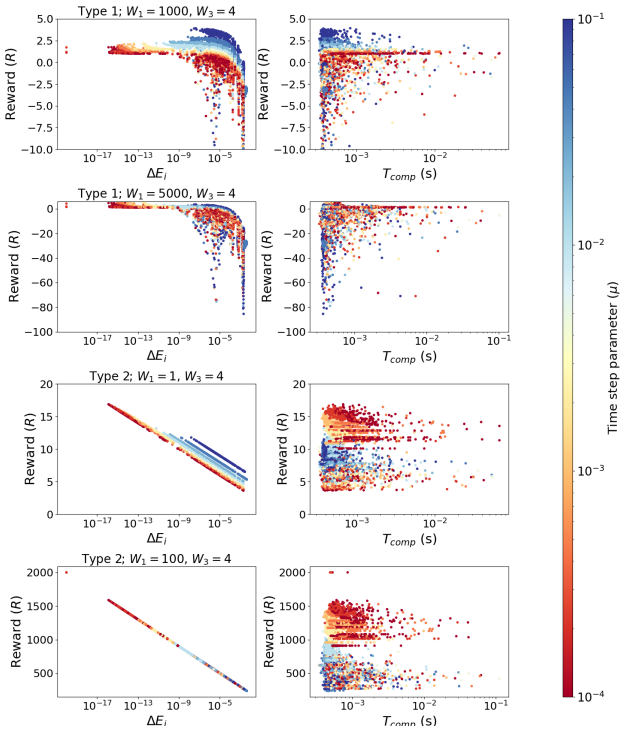


Figure 4.12: Reward value for different reward choices as in Equation 4.6 to Equation 4.7. The left plot shows the reward as a function of ΔE_i, and the right plot as a function of the computation time. The color represents the size of the time step parameter.

4.B Training and results for other initializations of Symple

We show the training evolution of the model for the `Symple` integrator. We show the evolution of the reward, energy error, and computation time in Figure 4.13. The episodes with the largest reward values are indicated with red lines. We select the model at episode 1949.

We show other examples of the performance of our model trained for `Symple`. In Figure 4.14a, we see the evolution of the integration for the initialization with seed 1. As before, the model adapts the actions to keep the energy error constant while increasing the action number as the simulation progresses to improve the computation time. In Figure 4.14b we see similar results, with our model being able to keep the energy error constant.

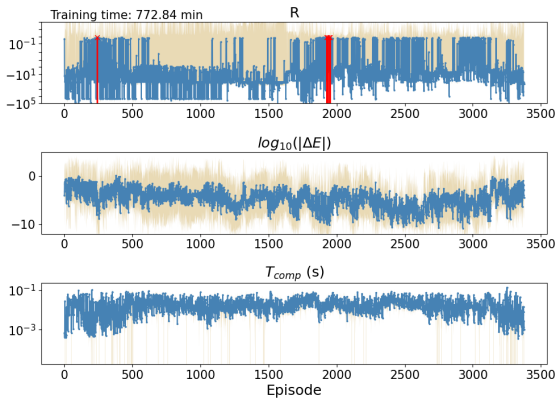
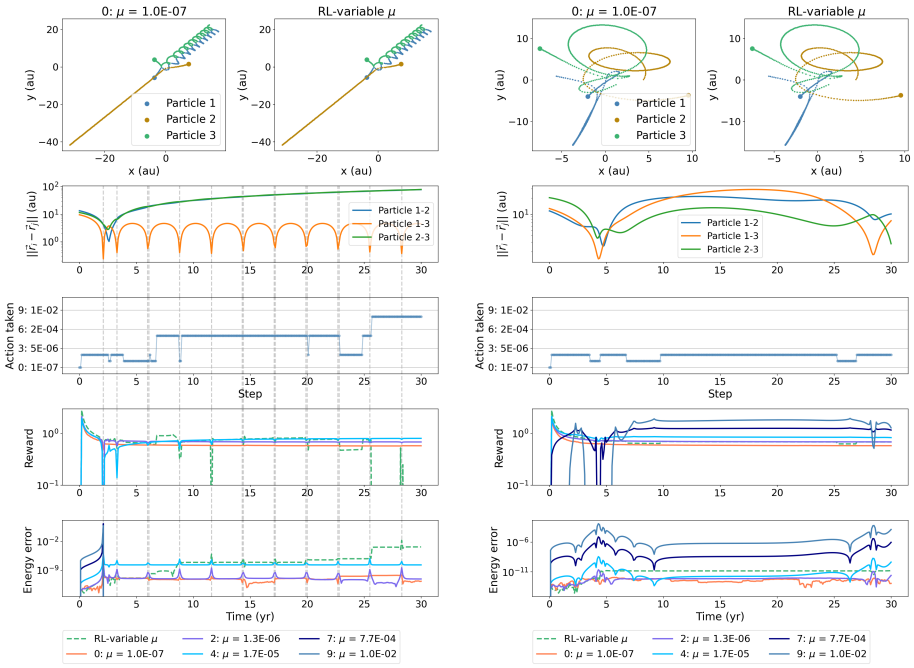


Figure 4.13: Evolution of the average and standard deviation of different metrics of the test dataset per episode of: the reward value (first row), the energy error (second row), and the computation time (third row).



(a) **(b)**

Figure 4.14: Comparison of fixed-size time-step parameters with Reinforcement learning. We show the trajectory in Cartesian coordinates (top panel), the pairwise distance between particles (second panel), the action taken at each step (third panel), the reward for each case (fourth panel), and the energy error at each time step for each study case (fifth panel), for initializations with Seed 1 **(a)** and Seed 2 **(b)**.

5 REINFORCEMENT LEARNING FOR ADAPTIVE TIME-STEPPING OF BRIDGED CLUSTER DYNAMICS SIMULATIONS

Work in preparation by Veronica Saz Ulibarrena, Simon Portegies Zwart. Reprinted here in its entirety.

ABSTRACT

Astrophysical simulations often involve multi-scale and multi-physics scenarios, which entails the need for methods that can handle those. A common approach involves separating the system into multiple subsystems according to their characteristics, with each subsystem being integrated using problem-specific methods. Then, the systems are coupled on a given time scale. The coupling time scale, or coupling integration time, is determined manually and remains fixed throughout the simulation. In this work, we introduce a novel approach that leverages reinforcement learning techniques to automatically select the coupling time step during simulations. Our method effectively balances computation time and accuracy by adapting the time-step size to the characteristics of the simulation at each step. We test our method on a multi-scale problem: a star cluster in which one star contains a planetary system. We perform multiple tests on clusters with different (low) numbers of bodies and find that the method remains robust for multiple cases. Additionally, we test the effect of changing the integrators for the subsystems and find that our method is independent of this choice. Similarly, we implement a parameter that scales the time steps to tune the accuracy requirements of the problem and find that our method remains valid in this case. For long-term integration, where energy errors tend to accumulate due to the chaotic nature of the problem, we find that our reinforcement learning method can achieve better results than the methods with fixed-time steps, but all cases lead to large energy errors in time. We develop a hybrid strategy that can detect sudden jumps in energy error and recursively reduce the time-step size at a given step to prevent them. The hybrid method achieves performances orders of magnitude better compared to standard methods, without significantly increasing the computation time. This method en-

sures the robustness of simulations that use reinforcement learning techniques. With our method, we eliminate the need for expert knowledge and achieve simulations that balance computation time and accuracy while adapting to the needs of the simulation at each step. This method can be directly extrapolated to a large range of astrophysical simulations.

5.1 Introduction

Astrophysics simulations are becoming more complex with time. By increasing the number of bodies involved, or the physical phenomena taken into consideration, the results obtained are more true to the physical reality and therefore easier to compare with observational results. This increase in complexity leads to the need for more efficient methods that can take into consideration the computational limitations of the problem to be studied.

Whereas 4th order direct integration methods through such as `Hermite` Makino & Aarseth (1992), `Ph4`, and `Brutus` Boekholt & Portegies Zwart (2015b) can result in very accurate simulations (with the right choice of simulation parameters), they are not suitable for simulating large systems. The computational effort of direct integration methods scales with N^2, with N being the number of bodies in the system. For example, the computational effort to integrate a star cluster, which can contain from dozens to millions of stars, can lead to simulations that last several months depending on the time scale over which it is integrated. To circumvent this problem, some methods have been designed to speed up this computation at the cost of accuracy. For example, the hierarchical tree algorithm from `Barnes-Hut Tree Code` Barnes & Hut (1986) groups bodies that are far away, whereas interactions from bodies close to each other are calculated directly.

Another problem arises when a system is composed of multiple hierarchical systems. For example a planetary system with moons, and a star cluster with planetary systems, among others. In these cases, the difference in scales leads to a decision problem: to integrate the system on the scale of the largest subsystem to speed up the calculation while missing the subtleties of the smallest subsystem, or to integrate the system using a time-step size consistent with the scale of the smallest subsystem, leading to impractically-large computation times. This problem can be effectively addressed by using methods that can separate these subsystems to allow the integration of each of those using the right integration settings. Examples of these methods are `Lonely planets` [cite], `Nemesis` Zwart et al. (2020), and `Bridge`, a method designed by Fujii et al. (2007), and which we will further look into in Subsection 5.2.1. Similarly, many of these methods can be used when there are many physical phenomena involved in the problem, such as gravitational interactions and hydrodynamical processes, radiation processes, stellar evolution, etc.

These methods provide a solution that allows to numerically integrate systems “separately". However, those still need to communicate with each other and ensure that each subsystem includes the effect of its environment. This coupling is dealt with differently in various methods. A common challenge is finding the time scale at which those systems need to interact (or communicate with each other) to keep our simulation accurate enough for our purpose, depending on the scientific question, while keeping the computational cost low. The choice of this coupling time step is normally made manually based on expert knowledge and kept fixed throughout the simulation. As explained in Saz Ulibarrena

& Portegies Zwart (2024), most chaotic systems benefit from a variable time-step size Heggie & Hut (2003); Aarseth (2003). Indeed, most integration codes nowadays include some internal calculation of the time-step size to ensure that it adapts to the conditions of the problem. Examples are found in Aarseth & Lecar (1975); Pelupessy et al. (2012), where the free-fall time of pairs of particles is used as a measure of how closely particles are interacting and used to estimate an adequate step size. However, this type of solution is not available for coupling codes such as `Bridge` yet.

Machine Learning (ML) is being studied as a method to speed up simulations in many fields. It is common to use ML methods as a proxy to replace the integrator to improve efficiency by skipping expensive calculations Cai et al. (2021b); Greydanus et al. (2019a); Ulibarrena et al. (2024). Additionally, an interesting application consists of using ML methods to replace choices that are otherwise made manually. When setting up a simulation, there are many choices to be made; from the initial conditions, to the integrator, to the time-step size. Many of these choices are made based on expert knowledge, whereas other parameters are often left as their default value. This can lead to poor results or inefficient simulations. Additionally, expert knowledge is not always available for every experiment.

Reinforcement Learning (RL) methods are used to make choices automatically based on some values to be optimized. One of the most important parameters to be chosen in a simulation is the time-step size. As mentioned before, many integrators include the automatic calculation of the optimum time-step size at each step, but there is no current solution to obtain an estimation for coupling systems. We develop a Reinforcement Learning algorithm that automatically chooses this time-step size. In addition to adapting to the conditions of the problem at each step and providing a solution that optimizes accuracy and computation time, our method reduces the expert knowledge needed to run these simulations.

We focus on the case of a star cluster in which some stars have planetary systems orbiting them. Using the `Bridge` method, we train a reinforcement learning algorithm to choose the optimum time-step size at each simulation step.

In Section 5.2, we explain in detail our implementation of the `Bridge` method, the initial conditions chosen for the problem, the integration settings, and the main parameters involved in the training of a RL algorithm. In Section 5.3, we show the training of the RL algorithm and the results of applying the trained method to our simulations. We compare the performance of the trained model for a variety of initializations, integration scales, and the number of stars in the cluster. We also study the use of our trained RL algorithm in a multi-physics scenario: a star surrounded by a proto-planetary disk. Finally in Section 5.5, we discuss the possible uses and limitations of the method created and summarize the goals and results of our work.

The code is publicly available at `https://github.com/veronicasaz/RL_bridgedCluster`github.com/veronicasaz/RL_bridgedCluster.

5.2 Methodology

In our method, we couple different subsystems and integrate a reinforcement learning algorithm into the simulation.

5.2.1 System setup

We look into systems in which different scales are involved. Specifically, a cluster of stars where some have planets orbiting them. Integrating such a system becomes quadratically more expensive as the number of bodies increases. Therefore, optimizing the computation time becomes an essential part of astronomical simulations. Additionally, the choice of time-step size should be based on the fundamental scales of the system. This scale can be radically different for the star cluster and the individual planetary systems. Hence, it is relevant to look into methods that allow the separation into multiple subsystems.

We take as our integration method the `Bridge` as defined in `AMUSE` Portegies Zwart & McMillan (2018); Portegies Zwart et al. (2009); Zwart et al. (2013). It is a method that separates a system into different subsystems. Each of them can then be integrated using a different code and integration settings. This allows the use of the most adequate methods in systems where there are fundamental differences in the behavior of the subsystems that form it. For example, it can be used when there are different physical processes involved. An example is a cluster in which some stars have a disk of material around them, where gravity and hydrodynamics have to be coupled. Another example in which a `Bridge` results useful is when the system consists of different scales. Integrating a large system separately from a smaller one allows for a better choice of integration parameters. A more detailed description of the `Bridge` can be found in Fujii et al. (2007); Zwart et al. (2013).

The fundamental idea of `Bridge` is that a system is divided into different subsystems, which are then integrated separately. After a certain time (Δt_B), the acceleration caused by one subsystem onto the other is calculated and used to modify the velocity of each particle. Then the subsystems are evolved again individually and the process is repeated until a final time is reached.

Although the bridge can be used in many cases: such as when coupling multi-physics phenomena, in many cases, we find that one particle should be included in the integration of both subsystems. This application is not implemented in `Bridge`. To solve that, we develop a "`inclusive Bridge`" in which one particle is common to both subsystems. In this case, in addition to exchanging information about the potential between subsystems, we use the `Bridge` to update the state of the common particle, as seen in Figure 5.1. For the case of a cluster of stars in which one star has a planetary system, we divide the system into two subsystems: the star cluster and the planetary system, respectively. Then, we calculate the potential of the cluster on the planetary system and use it to update the velocities of the planets and central star. We then evolve the planetary system using the adopted integrator for a time Δt_B and update the state of the common particle in the cluster subsystem. For the next step, we ignore the effect of the planets on the star cluster, and proceed to evolve (drift) the start cluster for a time Δt_B. Finally, we use the latest state of the star cluster to update the particles in the planetary system by drifting its center of mass. This process is repeated until the final time or maximum number of steps is reached. From this point onward, every mention of the `Bridge` will refer to our `inclusive Bridge`.

In Figure 5.2, we show a comparison of our `inclusive Bridge` against the `Bridge` method as implemented in `AMUSE` and with direct integration, i.e., integrating the whole system together. We can see that our method achieves orders of magnitude better accuracy than the regular `Bridge`. Nevertheless, both lead to energy errors orders of magnitude

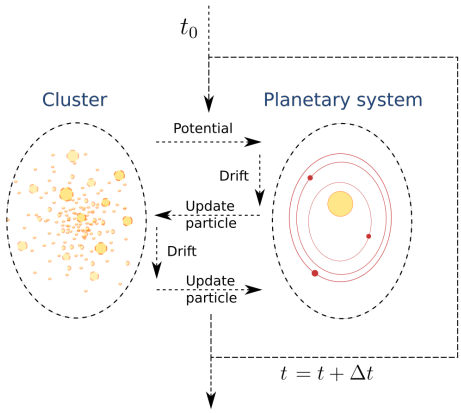


Figure 5.1: Schematic of the `(inclusive) Bridge` method as developed for this application. The system is divided into two subsystems: a star cluster and a planetary system, with one star being common for both. The potential of the cluster on the planetary system is calculated and used to update the velocities of the particles in the planetary system. Then the planets and central star are evolved and the state of the central star in the cluster subsystem is updated with that of the planetary system. Afterwards, the cluster subsystem is evolved and the state of the center of mass of the planetary system is updated using the latest cluster information. This process is repeated for a number of steps.

larger than direct integration. The error in both `Bridge` methods compared to direct integration results from them being very simple coupling strategies. We will discuss the advantages and shortcomings of using this method compared to some more complex ones in Section 5.5. This error is also derived from the fact that the `Bridge` time step is not adapting to the needs of the problem, whereas the direct integration case implements an adaptable time-stepping strategy. We will aim to improve on this using RL. The advantage of the `Bridge` with respect to direct integration is the possibility of using several integrators. Specially, in the case when different physical processes are present it is not possible to use direct integration. Secondly, the `Bridge` method is an approximation made to reduce the computation time in systems with large N. This advantage cannot be seen in Figure 5.2 as the number of bodies is not large enough. Additionally, it is also not a fair comparison as the direct integrator is developed in a high-performing language, whereas the `Bridge` methods are `Python` implementations. This results in better computation times with direct integration than with both `Bridge` implementations. In Section 5.5, Figure 5.19 we show a comparison of the performance (accuracy and computation time) for our `inclusive Bridge` method compared to direct integration and the RL implementation for different numbers of bodies. Although the advantage of using approximate methods appears for a larger N, for the purpose of this work, we limit ourselves to N between 5 and 15 to simplify the analysis.

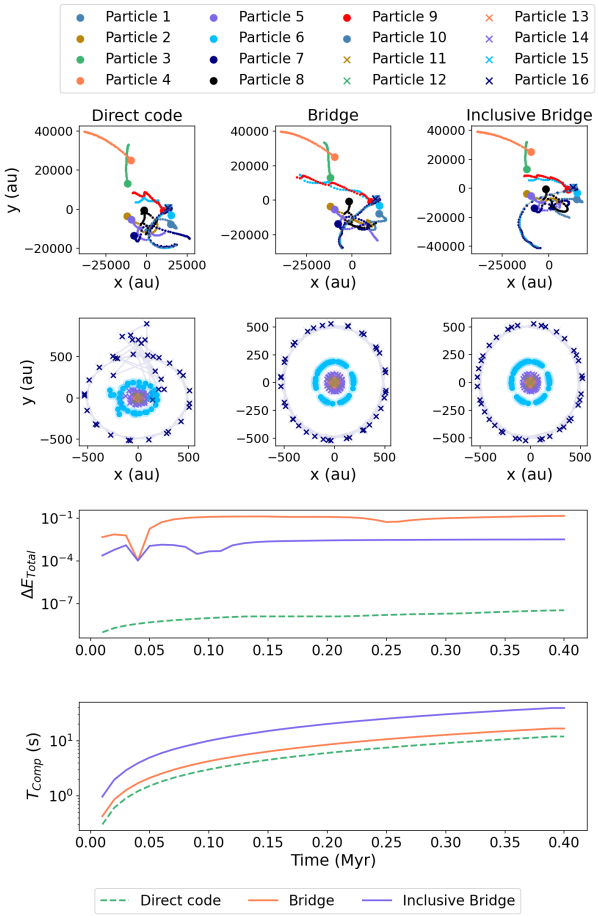


Figure 5.2: Comparison of our `(inclusive) Bridge` against the `Bridge` method as implemented in `AMUSE` and with direct integration.

Table 5.1: Initial conditions and integration settings of a cluster with one planetary system orbiting one of the stars.

STAR CLUSTER	
NUMBER OF STARS	[5 - 20]
MASS RANGE OF THE STARS	[1, 100] M_{Sun}
RADIUS OF THE CLUSTER	0.1 PARSEC
VIRIAL RATIO	0.5

PLANETARY SYSTEM	
INNER DISK RADIUS	10 AU
OUTER DISK RADIUS	100 AU
DISK MASS	0.02 M_{Sun}

INTEGRATION	
CLUSTER CODE	PH4
CLUSTER CODE ε	10^{-2}
PLANETARY SYSTEM CODE	HUAYNO
PLANETARY SYSTEM CODE ε	10^{-2}
CHECK STEP SIZE	$10^{-2}\ Myr$

5.2.2 Scientific setup

We take a simplified example of a star cluster with a small number of stars (between 5 and 20) and place a planetary system around one of those stars. Note that only the effect of Newtonian gravity is taken into consideration.

The masses of the stars are chosen following a power-law (Salpeter (1955)) distribution and the cluster is initialized using a fractal cluster model Goodwin & Whitworth (2004) with the values indicated in Table 5.1. Then, the last star is chosen as the common one between subsystems, and a planetary system is created around it assuming an oligarchical planetary growth Tremaine (2015); Kokubo & Ida (2002).

Each subsystem is evolved with a different integrator. For the star cluster we use `Ph4` integrator as implemented in `AMUSE` Portegies Zwart & McMillan (2018). This code is a direct integration code with internal calculation of the time step for each particle. For the planetary system, we use the `AMUSE` implementation of `Huayno` Jänes et al. (2014), which is well suited for the integration of planetary systems. Each code is initialized with a time-step parameter (ε) as indicated in Table 5.1, which scales the internally-calculated time-step sizes. The check step size refers to the frequency at which the state of the system is saved and the time-step size of the `Bridge` re-evaluated. This step has no relation with the actual integration steps of either subsystem or the `Bridge`.

There are many time steps involved in this setup:

- **Time-step size of the cluster subsystem**: this is the time-step used for the integration of the star cluster. When using `Ph4`, this time step is calculated internally and multiplied by a scaling parameter (ε).

- **Time-step size of the planetary subsystem**: this is the time-step used for the integration of the planetary system. When using `Huayno`, this time step is calculated internally and multiplied by a scaling parameter (ε) which might be different from the one for the cluster.

- **Bridge time step** Δt_B: this is the time scale on which the two subsystems exchange information. This means, how often the method calculates the potential of one system on the other and updates the common particle. This time-step size has to be chosen at the start of the simulation and there is no current solution to choose

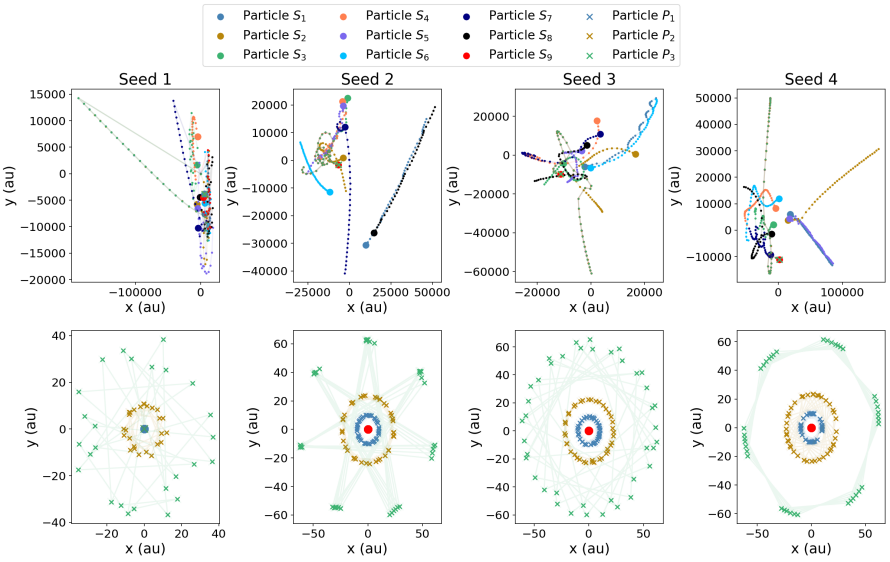


Figure 5.3: Initializations for Seeds 1 to 4 run for 40 steps (0.4 Myr) with a `Bridge` time-step of 5×10^{-5} Myr. The setup is formed by 9 stars and 3 planets.

an optimal value. This is the time-step size that we will determine using the RL algorithm.

- **Check step size**: we apply this denomination to the time scale we use to save the state of the system and to apply the reinforcement learning method. This means, after how much time we want to re-evaluate the choice of `Bridge` time step.

Using the aforementioned initial conditions, we plot in Figure 5.3 four examples of the integration of this system with 9 stars for seeds 1 to 4. The top row represents the evolution of the positions of the particles in the cluster. The second row shows the evolution of the planetary system. The stars are represented by a circle and the planets by an "x". This system is fundamentally chaotic, which will lead to large differences in its evolution depending on the initial conditions and time-step sizes. As a result of the system being chaotic, the optimal `Bridge` time-step size is different depending on the initialization. Additionally, since the conditions vary quickly in time, the optimal time-step size might also vary along the simulation.

5.2.3 Energy error

Since the dynamics is chaotic, the only method to evaluate the accuracy of a simulation is based on conservation laws. We use the energy error as an indication of accuracy. A discussion on the limitations of this metric will follow in Section 5.5.

The use of numerical integrators leads to energy errors. A large value of the energy error is an indication of unphysical solutions. We can therefore understand the energy error as a measurement of accuracy, or the validity of our simulation. The total energy error

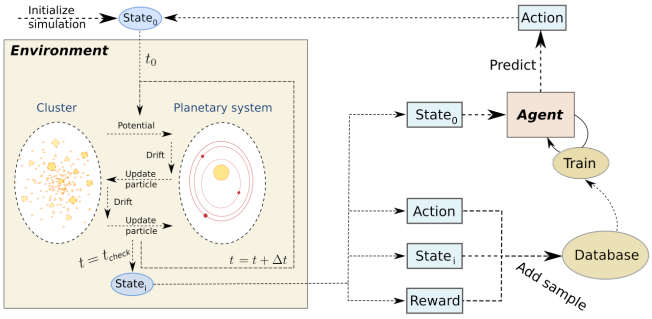


Figure 5.4: Schematic of the interaction between the Environment and the Agent.

is the sum of the kinetic and potential energy of the system. In this work, we denominate energy error to the relative difference of the energy at time step i and the initial time-step as

$$\Delta E_i = \frac{(E_{k,i} + E_{p,i}) - (E_{k,0} + E_{p,0})}{E_{k,0} + E_{p,0}} = \frac{E_i - E_0}{E_0}. \tag{5.1}$$

5.2.4 Reinforcement Learning

We train a reinforcement learning (RL) method to estimate the optimal `Bridge` time-step size. We choose Deep Q-learning as our RL algorithm to maximize a reward value ***R*** Sutton & Barto (2018). By supplying the algorithm with information from different astronomical simulations, it learns to take actions that maximize the reward. We adopt the specific implementation of Q-learning called Deep Q-networks (DQN) to allow for a continuous observation space Mnih et al. (2015). More information about Deep Q-learning and the reasoning behind this choice can be found in Saz Ulibarrena & Portegies Zwart (2024).

There are different elements interacting in the DQN method, as seen in Figure 5.4:

- **Environment**: the environment is composed of the astronomy simulations. The data obtained from them is used to create a dataset of the states, rewards, and actions. The composition of the environment is as explained in Subsection 5.2.2 and the astronomical simulations are initialized randomly during the training.

- **Agent**: the agent is the reinforcement learning algorithm that is trained to choose the actions. It is composed by two neural networks; namely the Q-net and the Target net. The weights of the Q-net are updated at each training step, whereas the Target net is only updated after an arbitrary number of steps. Both networks receive as input the State (S) of the system generated by the environment and produce the denominated Q-values associated with each possible action. Then, the action with the largest Q-value is selected and used for the next step of the simulation. The reward function obtained from the environment is used as the loss value to train the networks.

To create the agent we have to choose the hyperparameters of the neural networks as well as other training parameters. The specific values chosen will be mentioned in Subsection 5.3.2.

- **State**: the state is the representation of the environment that is used as an input to the neural networks in the agent. It must be formed by values that are representative of the physical state of the problem at a given time.

  In Saz Ulibarrena & Portegies Zwart (2024), as in many studies dealing with neural networks in the gravitational *N*-body problem, the state is chosen to be the cartesian coordinates representing the positions and velocities of each particle of the system. However, this leads to a fundamental problem: the input size is dependent on *N*, leading to limited extrapolation capabilities. To circumvent this problem, we define the state (***S***) as

$$\mathbf{S} = \left[\sum_{i}^{N-1} V_{n_i \to n_c} \quad , \quad -\log_{10}(\Delta E) \right] \tag{5.2}$$

  where $\sum_i^{N-1} V_{n_i \to n_c}$ is the gravitational potential of every star in the cluster (n_i) at the position of the common star (n_c). The second term is the current energy error of the simulation. This term is included to take into account that once the error is at a certain value, it is not likely to be reduced by large amounts. It is therefore important for the RL algorithm to consider this to avoid incurring unnecessary computational cost. A more detailed discussion on this remark can be found in Saz Ulibarrena & Portegies Zwart (2024).

- **Actions**: the actions (***A***) are the possible values of a decision variable. The reinforcement learning algorithm is trained to choose between these values to optimize a reward function. The actions are taken from a finite-size array which contains the value of the control variable associated with each action. Our decision variable is the `Bridge` time-step size. At each step, an action is chosen to determine the value of Δt_B to be taken for the next steps of the system simulation. The number of actions allowed as well as the minimum and maximum values are shown for each case in Subsection 5.3.2.

- **Reward function**: the reward is the function to be optimized by reinforcement learning. For the study in hand, we want to balance accuracy and computation time. Therefore, we take a function that balances both the energy error and the computation time. We write this function as in Saz Ulibarrena & Portegies Zwart (2024)

$$\mathbf{R} = -W_1 \frac{\log_{10}\left(|\Delta E|/10^{-10}\right)}{|\log_{10}(|\Delta E|)|^3} + W_2 \frac{1}{\log_{10}(A)}. \tag{5.3}$$

  The first term corresponds to the energy error at a given step normalized by 10^{-10} and divided by the cube of the energy error. The logarithm is used to linearize the range of values in this term. The second term corresponds to the computation time represented by the inverse of the time-step size taken. $W_{1,2}$ are the weights used to balance these two terms. They are a design choice and the used values can be found in Subsection 5.3.2. This reward function is specifically designed for the problem

of the simulation of a number of bodies interacting via their gravitational forces Saz Ulibarrena & Portegies Zwart (2024).

5.3 Results

We show the results obtained from training the RL algorithm and its application to different cases of the start cluster simulation.

5.3.1 Validation of the results

There is no analytical solution to the problem of a group of bodies ($N > 2$) interacting via Newtonian gravity. Additionally, it is chaotic. This means that finding a baseline with which to compare the results becomes challenging. Once the RL algorithm is trained, its results can be compared with those obtained without the use of RL. However, different values of the time step(s) will lead to radically different outcomes. To simplify this problem, we fix the values of the time-step parameters of the integrators involved (see Table 5.1) and study the effect of the `Bridge` time-step (Δt_B) on the outcome of the simulations.

We perform a convergence study to further understand the effect of Δt_B on the accuracy and computation time. Additionally, we want to understand which range of values should be used for the actions ($\boldsymbol{A}$). To do so, we simulate the systems with initializations with seeds 1 to 4 (see Figure 5.3) for 40 steps using different values of Δt_B. We expect to find a value of Δt_B for which the energy error does not improve further if being reduced, but instead just results in larger computation times. Then we can say that we have found a converged solution. This definition is not rigorous, as in order to find a truly converged solution in a chaotic system we would need to use an algorithm with arbitrary precision such as `Brutus` Boekholt & Portegies Zwart (2015b).

We show the results in Figure 5.5. The value of Δt_B for which the simulation converges depends on the initialization. Some cases such as the one with seed 4 are more chaotic, which results in the minimum value of Δt_B considered not being small enough to find convergence in the energy error. However, for a case such as the one with seed 1, convergence is reached for relatively large values of Δt_B. This study supports the idea that the optimal choice of time step largely depends on the initial conditions.

Although it is difficult to get a clear conclusion about the range of values that should be used for the given simulations, we can observe that in most cases convergence is reached for time step sizes between 10^{-4} and 10^{-5} Myr. We therefore choose an intermediate value of 5×10^{-5} as the lowest limit for the RL actions. Regarding the higher limit, we can see that depending on the simulation a time step size of 10^{-2} Myr may yield large energy errors. We choose this value as the upper limit for the RL actions.

5.3.2 Training results

We defined the environment in Subsection 5.2.2 and the RL method in Subsection 5.2.4. With that information, and the settings in Table 5.2, we obtain the trained models.

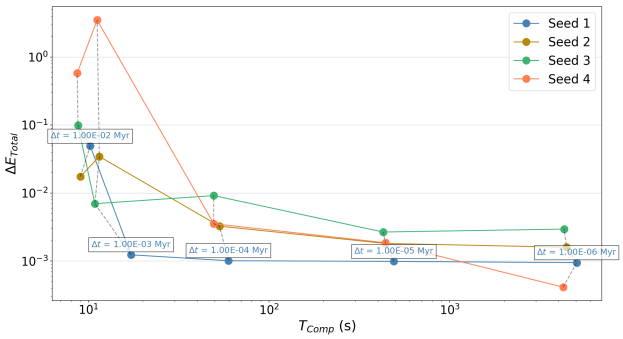


Figure 5.5: Energy error and computation time for the simulation of initializations with seeds 1 to 4 run for 40 steps (0.4 Myr) with different fixed `Bridge` time-step sizes. The time-step sizes are indicated in the figure.

Table 5.2: Training and simulation parameters.

COMMON	
NUMBER OF PLANETS	VARIABLE
MAX STEPS PER EPISODE	40
ΔE TOLERANCE	1×10^0
HIDDEN LAYERS	5
NEURONS PER LAYER	200
BATCH SIZE	125
TEST DATA SIZE	3
NUMBER OF ACTIONS	10
RANGE OF ACTIONS	$[5 \times 10^{-5}, 10^{-2}]$ MYR
$W_{1,2}$	[50, 1]
BRIDGE ε	1.0

First of all, the number of planets depends on the mass of the central star, and will therefore vary depending on the other settings. We set the maximum number of integration steps to 40, which with a check step size of 10^{-2} corresponds to 0.4 Myr.

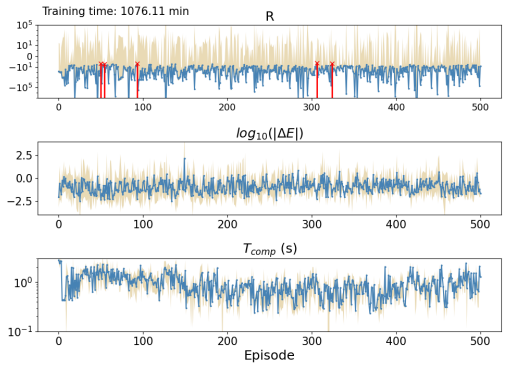
In Table 5.2 we show the network architecture. Due to the chaotic nature of the problem, we had to find a baseline on which to calculate the reward during the training. Therefore, we test the model at each episode on a fixed set of test cases. We use 3 test cases. Ideally, this number should be increased for a better indication of the performance of the model at each episode, but due to computational limitations, we choose to keep this number low. Note that the model will be evaluated on a larger number of test cases after the training is finished.

The actions is a discrete array of length 10 with values of Δt_B that range from 5×10^{-5} to 10^{-2} Myr. Similarly to the implementation in `Hermite` and `Huayno`, we define a time-step parameter ε for the `Bridge` that multiplies the value in the actions. This value is set by default to 1. Finally, the weights for the reward function are shown in Table 5.2 and chosen so that the first term in Equation 5.3 is 50 times larger than the second term.

We start the training with a global search. In this case, we limit the number of stars

to 5 and keep a large value of the learning rate (as seen in the table in Figure 5.6) to keep the computation cost low. We train for 500 episodes and evaluate the performance of the models using the average reward at each episode. In Figure 5.6, we present the results of this global training. The rows in the Figure represent the reward value, energy error, and computation time for each episode. In blue we show the average value obtained from the test cases, and in yellow the standard deviation. We also show the total training time at the top left corner. We mark in red the five episodes with the largest reward and choose the best-performing model from them.

After the global training, the results are still not satisfactory and the reward oscillates. We therefore perform a local search (Figure 5.7) starting from the best performing model (model at episode 50) for 50 episodes with a lower learning rate. Then, we choose the best-performing model; i.e., the one at episode 27.

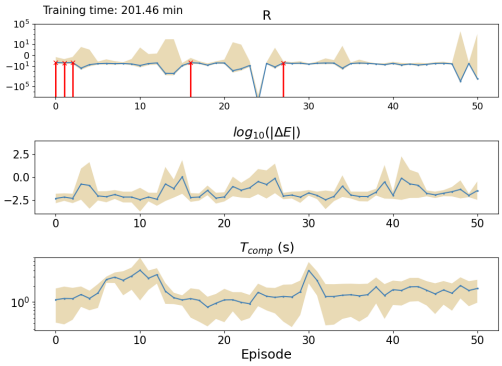


Global search	
Number of stars	5
Max episodes	500
Learning rate	1×10^{-3}
Model chosen	50

Figure 5.6: Evolution of the average (blue) and standard deviation (orange) of different metrics of the test dataset per episode of: the reward value (first row), the energy error (second row), and the computation time (third row) for the global training. With the corresponding training and simulation parameters.

We want to evaluate the performance of this best model. To do that, we run multiple simulations using the trained model and compare the results with those without the RL model. Figure 5.8 shows a schematic representation of the plot we will be using for the comparison of the performance. The runs with different fixed Δt_B ideally form a Pareto front that ranges from the cases with large computation time requirements and low energy errors (right of the plot) to the ones with small computation times and large errors (left of the plot). This Pareto front represents the best accuracy that can be achieved with fixed time-step sizes. Below this curve is the optimum region. Anything below the curve represents a better-performing case. We aim to obtain a method that is located on the Pareto front (to eliminate the expert knowledge) or below (to obtain better-performing methods).

In Figure 5.9, we show the average and standard deviation in computation time and energy error for 10 initializations run for 0.4 Myr. We compare the results of RL model 27 (RL-27) with those with fixed Δt_B. We do that for 5, 9, and 15 stars. The actual energy errors obtained for each of the runs are shown as dots but for simplicity, the computation



Local search 1	
Number of stars	5
Max episodes	50
Learning rate	1×10^{-4}
Model chosen	27

Figure 5.7: Evolution of the average (blue) and standard deviation (orange) of different metrics of the test dataset per episode of: the reward value (first row), the energy error (second row), and the computation time (third row) for the local training performed after the global one. With the corresponding training and simulation parameters.

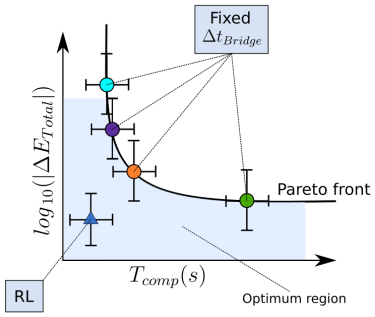


Figure 5.8: Schematic representation of the comparison of fixed Δt_B with the RL method.

time associated is ignored in the plot. An optimum value balances energy error (y-axis) and computation time (x-axis). We observe that the trained model does not present an advantage over the fixed time-step cases.

To solve this performance issue, we further train the network including simulations with variable N and a lower learning rate. The results of the training are shown in Figure 5.10. We choose the model at episode 173 and compare the results in Figure 5.11. We see the improvement in performance achieved by the RL-173 model. For $N = 5$, the model performs similarly to the one with $\mu = 1.6 \times 10^{-4}$, but with a shorter average computation time and smaller spread in energy error. For $N = 9$ our model results in both an improvement in energy error and computation time with respect to the fixed times-step cases. For $N = 15$, the results become harder to interpret as the fixed-step methods do not behave as expected (see Figure 5.8). Nevertheless, the algorithm achieves a better performance than the cases with fixed Δt_B.

We have shown that our model performs as well, or better, than the best-performing

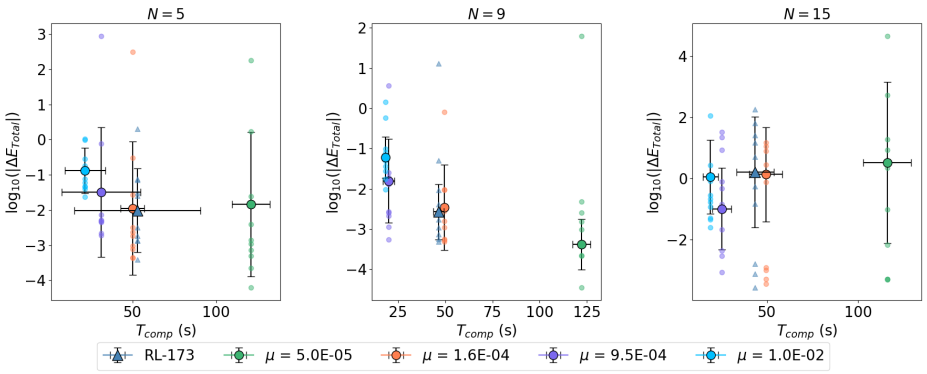
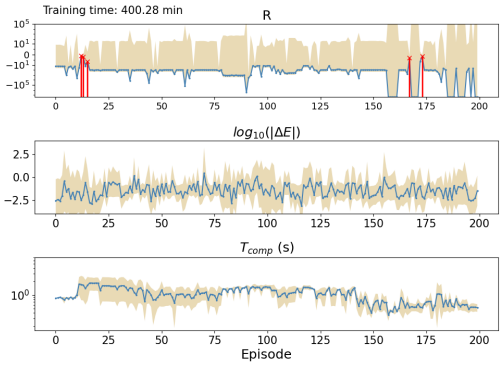


Figure 5.9: Average and standard deviation of the energy error and computation time for 10 different initializations run for 0.4 Myr. The results of the RL-27 model are compared to those of fixed Δt_B. The results of the RL model are unsatisfactory.



Local search 2	
Number of stars	[5-20]
Max episodes	200
Learning rate	1×10^{-4}
Model chosen	173

Figure 5.10: Evolution of the average (blue) and standard deviation (orange) of different metrics of the test dataset per episode of: the reward value (first row), the energy error (second row), and the computation time (third row) for the local training for different bodies. With the corresponding training and simulation parameters.

fixed-size case. When initializing a simulation, it is common to keep the default values used for Δt_B which generally leads to suboptimal results. Our method allows the achievement of optimal performance in terms of computational time and accuracy without the need for any expert knowledge or convergence study.

5.3.3 Integration results

We perform multiple experiments with model RL-173 to better understand its performance and extrapolation capabilities.

We show the individual behavior of the model for initializations with seed 4 (Figure 5.12 (a)) and seed 2 (Figure 5.12 (b)) with different numbers of stars. The top row rep-

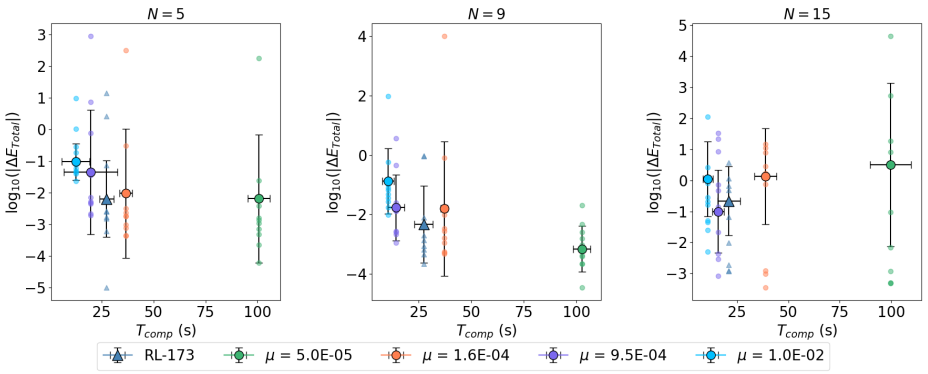


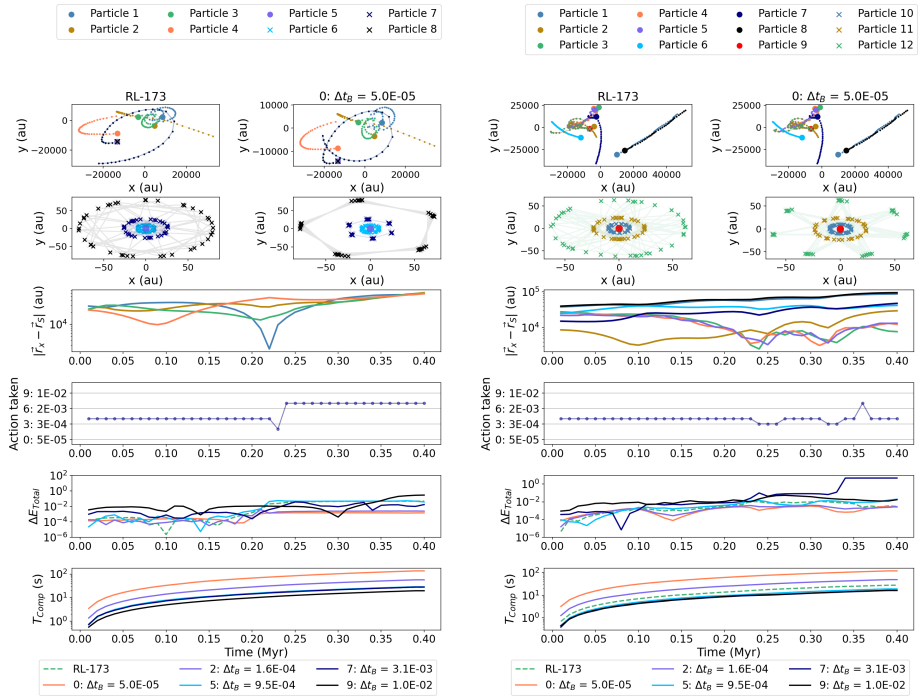
Figure 5.11: Average and standard deviation of the energy error and computation time for 10 different initializations run for 0.4 Myr. The results of the RL-173 model are compared to those of fixed Δt_B.

resents the position of the bodies in the star cluster. The left column shows the evolution using the RL model and the right one the results with the best-performing model with fixed Δt_B. The second row shows the evolution of the planetary system around the central star. The third row is the distance from each star to the one with the planetary system, which contains information about close encounters. The fourth row is the actions taken by the RL model at each step. Finally, the last two rows represent the energy error and computation time of each integration case (RL model and various fixed Δt_B).

We see in Figure 5.12 (a) that there is only one close encounter, and that the RL model recognizes it and chooses a more restrictive action (smaller time-step size). After the close encounter, the stars get further from each other, and the model chooses a less restrictive action, saving computation time. In Figure 5.12 (b), we see a case for 9 stars. We recognize two or three close encounters and see that the RL model adapts accordingly. Finally, it achieves an energy error on the lowest range without incurring large computation times.

In Figure 5.13, we see two other examples. Figure 5.13 (a), shows an example with 15 bodies. In this case, we do not see any close encounters and the algorithm learns to keep a constant action which represents a balance between energy error and computation time. In contrast, in Figure 5.13 (b), the model recognizes close encounters and adapts the otherwise large actions, obtaining an energy error that is lower than most of the other fixed-size cases with a very low computation time.

In Figures 5.12 and 5.13, we observe sudden jumps in the energy error for certain cases. Additionally, the planets may escape their central star or move to more eccentric orbits. To understand these jumps in energy error, we plot in Figure 5.14 the evolution of the distance between each planet and the central star, to better understand these cases. We do this for the same scenarios as in Figures 5.12 (b) and 5.13 (b). Additionally, we present the semi-major axis and eccentricity values for a better understanding of the dynamical evolution of the planetary system. We observe how the jumps in energy error correlate with the distance of a planet to the central star increasing radically. Similarly, it can be observed in these cases that the eccentricity also increases. The most common scenario is the outermost planet (Planet 3) suddenly changing its orbit when a nearby star perturbs it.



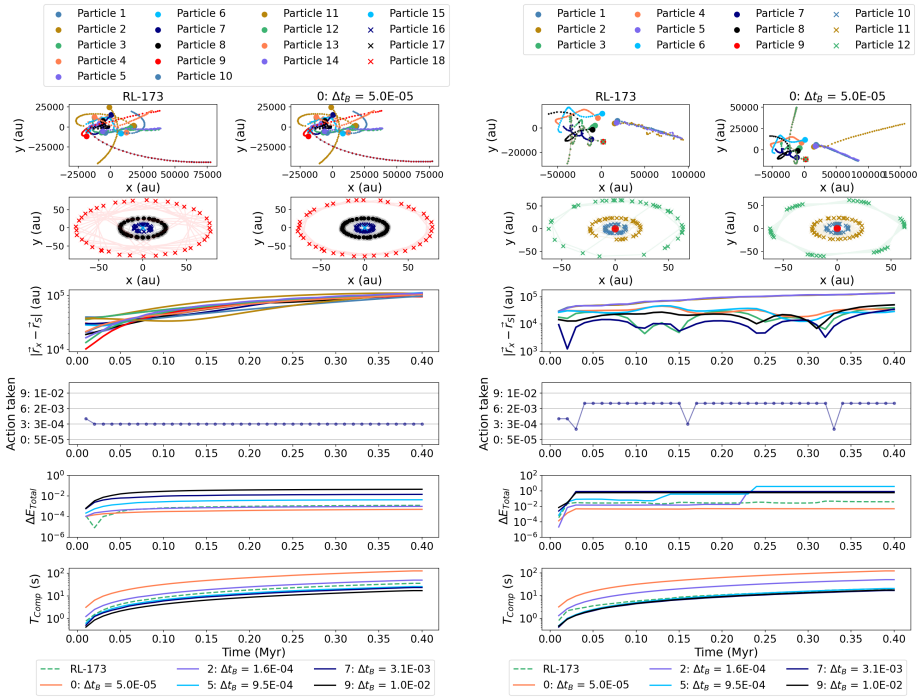
(a) **(b)**

Figure 5.12: Comparison of fixed-size time-step parameters with our RL model for 40 time steps (0.4 Myr). We present the trajectory in Cartesian coordinates of the star cluster (top-row panels) and the planetary system (second-row panels), the distance between each star to the one containing the planetary system (third row), the actions taken by the RL algorithm (fourth row), the energy error at each time step for each study case (fifth row), and the computation time for each study case (last row), for initializations with seed 4 (a) and seed 2 (b).

In our method, we have not considered cases in which one planet becomes unbound or increases the distance to the central star. Once this happens, to keep the energy error bound, the planet should be integrated in the same N-body code as the star cluster, instead of remaining within the planetary system code. More complex bridging algorithms such as `Nemesis` Zwart et al. (2020) include this option in their implementation. We leave this addition to future works.

5.4 Experiments

We have seen that the trained RL model manages to achieve results that are better than those with fixed Δt_B. Those results were obtained with conditions similar to those used to train the model. Therefore, we want to understand its performance when applied to different scenarios. To do that, we perform experiments in which we modify the final integration time, the integrators used, and the time-step parameter.



(a) **(b)**

Figure 5.13: Comparison of fixed-size time-step parameters with our RL model for 40 time steps (0.4 Myr). We present the trajectory in Cartesian coordinates of the star cluster (top-row panels) and the planetary system (second-row panels), the distance between each star to the one containing the planetary system (third row), the actions taken by the RL algorithm (fourth row), the energy error at each time step for each study case (fifth row), and the computation time for each study case (last row), for two initializations with seeds 3 (a) and 4 (b).

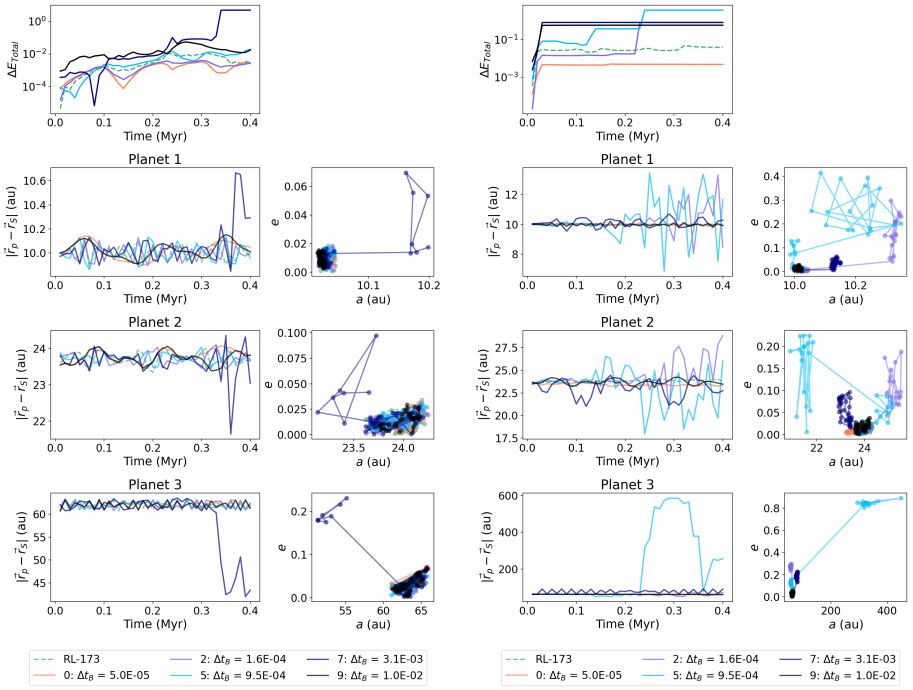
5.4.1 Number of bodies

In Figure 5.9, we show a comparison of the performance of the RL model for 10 different initializations for three cases of N. We use this plot as the baseline with which to compare the other experiments. These systems are chaotic, and the results depend on the initializations. We discuss this further in Section 5.5.

5.4.2 Long term integration

The model is trained on simulations that were run until a final time of 0.4 Myr. We study the performance of the trained model on longer integration times to understand the possible use of our method for long-term simulations.

In Figure 5.15, we show two examples of the simulation with seeds 1 (a) and 2 (b) with different numbers of bodies. We observe here that the model is also able to identify close encounters and chooses more restrictive actions to keep the energy constant and low.

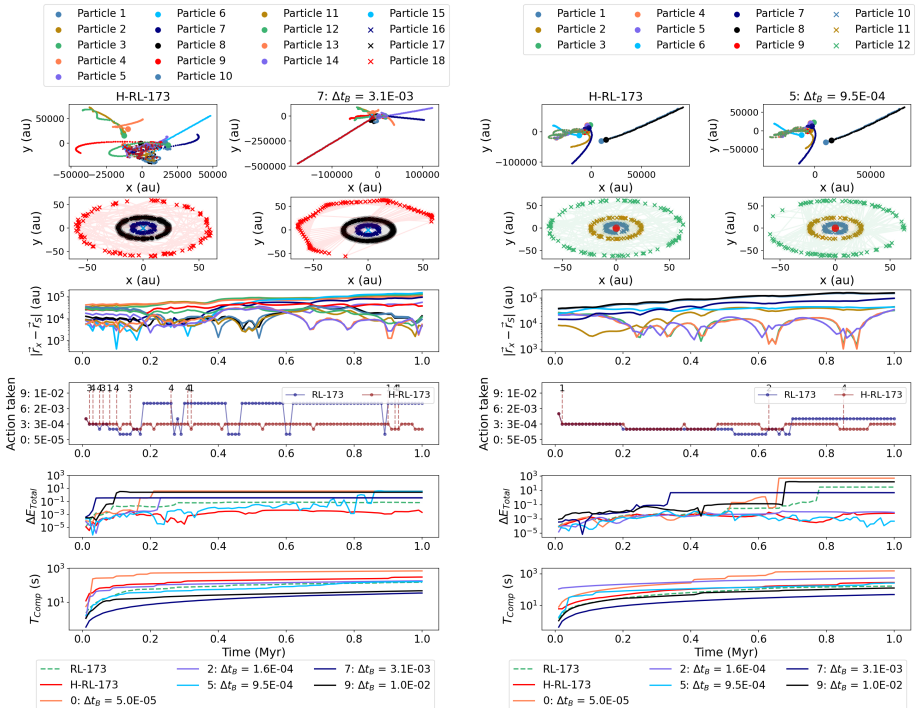


(a) **(b)**

Figure 5.14: Comparison of fixed-size time-step parameters with our RL model for 40 time steps (0.4 Myr). We present the energy error (top row), the time evolution of the distance of each planet to their central star (left panels), and the evolution of the semi-major axis (a) against the eccentricity (e) (right panels) for each planet with seeds 2 (a) and 4 (b).

We can see in Figure 5.15 (a) that the energy error achieved is smaller than with the most accurate fixed-step size while the computation cost remains small. In Figure 5.15 (b), we see a case without pronounced close encounters, but some bodies escape the system at $t \approx 0.7$ Myr due to a sudden increase in the energy error. We observe this behavior in many cases. Making a mistake in the choice of time-step size can lead to sudden jumps in energy error. The RL algorithm can keep the energy error low for longer than other cases shown, but at $t \approx 0.8$ it still experiences a sudden increase.

For long-term integration, it is essential to avoid mistakes in the choice of time-step size. A wrong choice in the action by the RL model can result in a long simulation being inaccurate, and therefore unusable. To prevent this, we propose the implementation of a method that can identify sudden jumps in energy error and adapt the action accordingly to correct for a wrong choice of time step. A similar idea was shown in Ulibarrena et al. (2024). We evaluate the energy error with respect to the previous step. If this relative error is larger than a certain choice value, we repeat the integration step with a time-step size that corresponds to an action lower or, in the case that we were already at the lowest action, reduces the time-step to half its size. This process can be recursively repeated



(a) **(b)**

Figure 5.15: Comparison of fixed-size time-step parameters with our RL model for 100 time steps (1 Myr). We present the trajectory in Cartesian coordinates of the star cluster (top-row panels) and the planetary system (second-row panels), the distance between each star to the one containing the planetary system (third row), the actions taken by the RL algorithm (fourth row), the energy error at each time step for each study case (fifth row), and the computation time for each study case (last row), for two initializations with Seeds 1 and 2.

until the energy error falls within the desired limits. We choose a maximum number of 4 iterations to avoid incurring too large computation time. The error tolerance is defined as

$$\Delta t_B = \frac{\Delta t_B}{2} \qquad \text{for} \qquad \log_{10}(\Delta E_i) - \log_{10}(\Delta E_{i-1}) > 0.3. \tag{5.4}$$

The results obtained with this method, denominated as H-RL (for Hybrid-RL) are included in Figure 5.15. The hybrid implementation manages to prevent sudden jumps in energy error. On the fourth row, we show the actions taken by the H-RL method, and also at which steps the hybrid method was activated (according to Equation 5.4) and the number of iterations it performed to lower the energy error to below the threshold.

The H-RL method results in energy errors orders of magnitude lower than the best result without incurring a much larger additional computation time (Figure 5.15 (a)). For the cases where the RL method experienced a sudden jump in energy error (5.15 (b)), the hybrid method prevents this jump, leading to a final energy error that is on the order of the most accurate results of the fixed time-step cases.

In Figure 5.16, we present a similar analysis for the integration up to 1 Myr, including the RL and the H-RL results. We find that for $N = 9$ and $N = 15$, the RL model results in a similar performance to that of the fixed time-step cases. However, the H-RL further reduces the final energy error at the cost of some computation time. This leads to better-performing results for $N = 9, 15$. For $N = 5$ we find that the RL method is comparable in performance to those with fixed-size time-step. The H-RL model results in a large standard deviation in the energy error for different runs. This is the result of the chaotic behavior in the system, which we discuss further in Section 5.5.

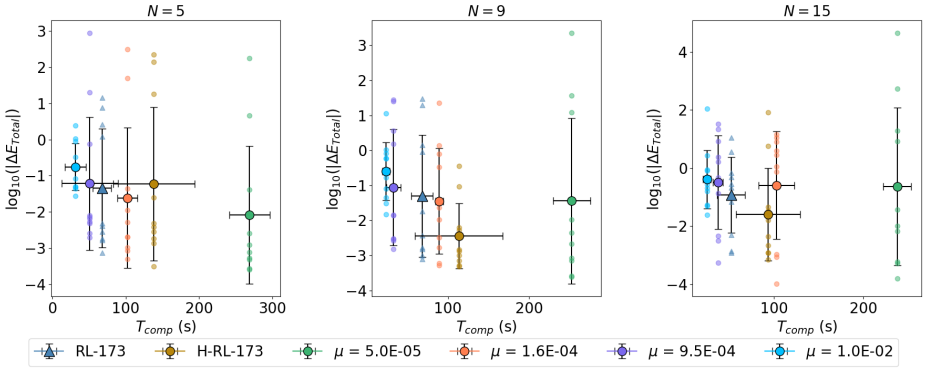


Figure 5.16: Average and standard deviation of the energy error and computation time for 10 different initializations run for 1 Myr. The results of the RL-173 and the H-RL-173 models are compared to those of fixed Δt_B.

This hybrid method is especially relevant for long-term integration as an increase in the energy error is rarely reversible. For the purpose of simplicity, we will not include the hybrid integrator results in the following experiments.

5.4.3 Numerical integrators

In this case, we want to understand whether our trained model is independent of the choice of integrator for the subsystems involved. We therefore replace the Cluster integrator with `Hermite` and the planetary system integrator with `Ph4`.

We can see in this case (Figure 5.17) that the performance of the RL model remains comparable to the baseline case. It achieves better results in terms of a smaller standard deviation than the fixed cases for $N = 5$. For $N = 9$, the average for the RL method is located on the Pareto front, which means that the results perform similarly than the fixed time-step cases. For $N = 15$ we again encounter results that are surprising, such as the most accurate case of fixed time-step resulting in a larger average energy error than some of the larger step-size cases.

5.4.4 Application of a time-step parameter

A time-step parameter (ε) is used in integrators such as `Hermite` and `Huayno` to scale the size of the time steps to allow to make the simulations faster (large ε) or more accurate

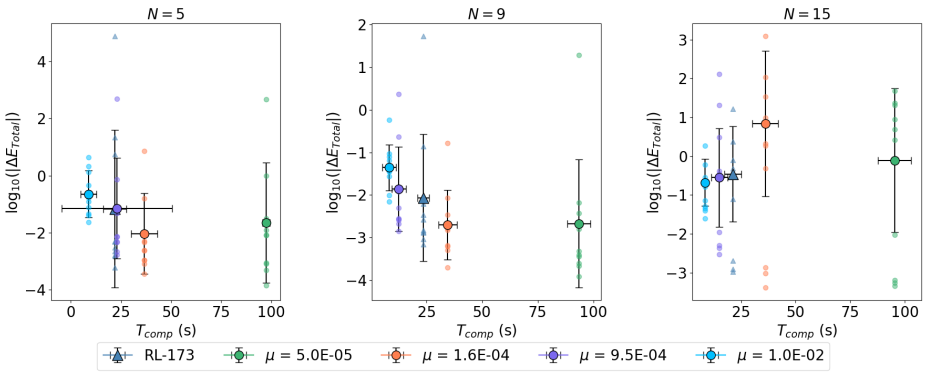


Figure 5.17: Average and standard deviation of the energy error and computation time for 10 different initializations run for 0.4 Myr. The results of the RL-173 model are compared to those of fixed Δt_B. The numerical integrators used in this case are different from those used for training.

(small ε). We implement a similar feature to scale the values of Δt_B. In Figure 5.18, we show that the performance does not decrease by scaling the actions by 10^{-2}. For all cases, the RL method performs better, or similarly to the fixed-size cases. Additionally, it can be seen that for $N = 15$ all cases result in similar average energy errors. This is an indication that the values chosen are unnecessarily small (see Subsection 5.3.1). For $N = 5$ and $N = 9$, the results obtained with the RL model are unequivocally better than those with fixed-step size.

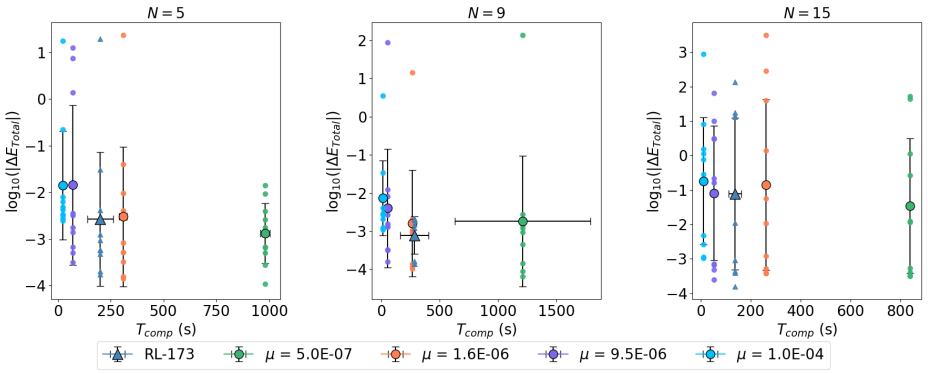


Figure 5.18: Average and standard deviation of the energy error and computation time for 10 different initializations run for 0.4 Myr. The results of the RL-173 model are compared to those of fixed Δt_B. The time-step parameter is changed to 10^{-2}.

5.5 Discussion and conclusions

We have trained a reinforcement learning algorithm to automatically find an optimum value for the coupling integration time-step size of a star cluster. By doing so, we have

eliminated the need for expert knowledge and experimentation needed to set up a simulation with an adequate value of Δt_B, therefore saving time and computational resources. Our method balances energy error (i.e., accuracy) and computation time, and finds results that are better, or in the worst cases similar, to the best-performing cases of fixed time step. Additionally, our method allows to vary the value of Δt_B dynamically to adapt to the needs of the simulation, which is essential in chaotic problems due to their fast-changing conditions.

The performance of our RL method was better than that of all the other options. However, for long integration times, none of the fixed time-step sizes achieved good accuracies. At some point during the integration, due to a close encounter, the time-step size would need to be smaller than what is allowed, which led to steep jumps in the energy error. Although our method managed to keep the energy error constant for longer periods of time than the fixed step-size cases, we want to create a method that is robust for longer periods of time. Therefore, we implemented a hybrid method that identifies sudden changes in energy error and recursively reduces the time-step size at a given step to prevent the increase in energy error. The results are very similar to those of the RL when there was no sudden increase in energy error, but the hybrid implementation prevents those in the necessary cases without a major increase in computation time. In many cases, this hybrid method led to an accuracy that was orders of magnitude better than the best-performing option of fixed time-step size. This method can be used for astronomical simulations to improve their performance and accuracy without adding expert knowledge.

In this work, we have made some assumptions. First of all, we are using energy error as the main measurement of accuracy. A low value of the energy error is an indication of the method adhering to the physical laws but does not ensure that the dynamics of individual bodies are correct. We show a convergence study based on the energy error because convergence based on the dynamics is not possible in a chaotic problem. Any small change in time-step size would lead to a different realization Irani et al. (2024).

We have shown how the performance of our method compares to that of the fixed-step size using the mean and standard deviation of the accuracy and computation time for 10 initializations. Many variations in performance are due to the fact that the problem is largely chaotic. Different initializations result in large ranges of performances even with the same choice of integration settings. It becomes challenging to find a straightforward method to measure the performance of any method. When interpreting these plots, the chaotic nature of the problem needs to be taken into consideration, as many show non-intuitive results. For example, the performance might be worse for low time-step sizes than for larger ones in some cases.

Also due to the chaotic nature of the problem, it is not trivial to find a baseline with which to compare the results. Every change in time-step size (and initial conditions) will lead to dramatic changes in the dynamical evolution of the system. There is no current method to choose or dynamically adapt the time-step size of the `Bridge` method, and therefore we do not have a baseline to compare our new results with. Additionally, we many times compare our results with the most accurate case of fixed-time step size, but this does not always result in a good performance itself. Actually, our method is currently the best-performing solution.

For a low number of bodies, we can compare the `Bridge` method with a direct integrator such as `Ph4`. In Figure 5.19, we plot the accuracy (final energy error) and com-

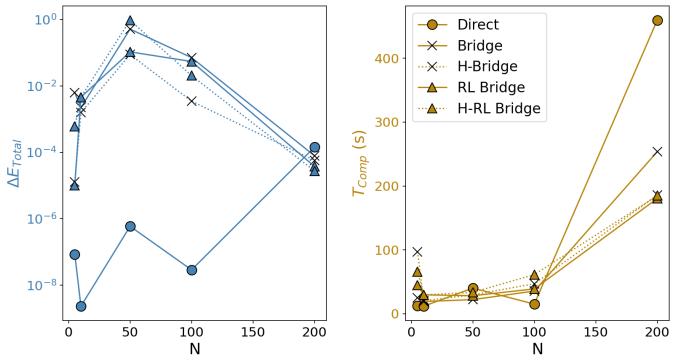


Figure 5.19: Comparison of the total energy error and computation time for an initialization with seed 3 run 40 steps with 9 stars. We compare the results with direct integration, with our `Bridge`, a hybrid implementation of the `Bridge`, and the cases with RL and H-RL.

putation time as a function of the number of stars in the cluster. We can see how direct integration is not suitable for the integration of a large number of bodies as the computation cost increases quadratically with N. The energy error with the `Bridge` method is orders of magnitude larger for a small number of bodies but becomes comparable as N increases. Additionally, we show the results with our RL method and with our hybrid implementation. This comparison is however not completely fair as the direct code has been optimized for speed, whereas the `Bridge` codes in all their variations are simple `Python` implementations. The gap in computation time could be reduced by optimizing the implementation. Similarly, as mentioned in Subsection 5.2.1 Figure 5.2, the `Bridge` methods are simple coupling methods which accuracy could be improved with a more complex implementation.

We have currently performed experiments for $N = 5, 9, 15$ and trained the RL algorithm with clusters with up to 20 stars. Future work should focus on increasing this number in order to allow the use of this method for a larger variety of star clusters. Additionally, we have not performed a hyperparameter optimization, which we believe could also slightly improve the performances shown here.

The general nature of the method allows us to extrapolate to a large variety of systems. The planets could be replaced with a central star surrounded by a protoplanetary disk or a cloud of gas and the method would still be applicable without any changes in the method. Also, the number of bodies can be scaled largely without the need for retraining, although the performance might decrease as the setup diverges from the one used to train the RL method.

We have tested the method for different integrators, for the inclusion of a time-step parameter to adapt the level of accuracy desired, and for long-term integration including our hybrid method. More tests can be performed. However, we show here that the performance remains robust with these changes and only varies slightly.

In short, we have created a method that eliminates the need for the manual choice of the coupling time-step size and changes it dynamically to adapt to chaotic problems. We have built a hybrid method that makes the RL solution robust to mistakes and results

in more accurate simulations for an optimized computation time. This method can be applied to a large variety of astrophysics simulations without major changes to improve their performance.

5.6 Acknowledgments

This publication is funded by the Dutch Research Council (NWO) with project number OCENW.GROOT.2019.044 of the research programme NWO XL. It is part of the project “Unraveling Neural Networks with Structure-Preserving Computing". In addition, part of this publication is funded by the Nederlandse Onderzoekschool Voor Astronomie (NOVA).

Appendix

5.A Summary of experiments

We show a summary of the experiments performed with our RL and H-RL methods. We show the averages and standard deviations of multiple runs for each of the `Bridge` time-step cases.

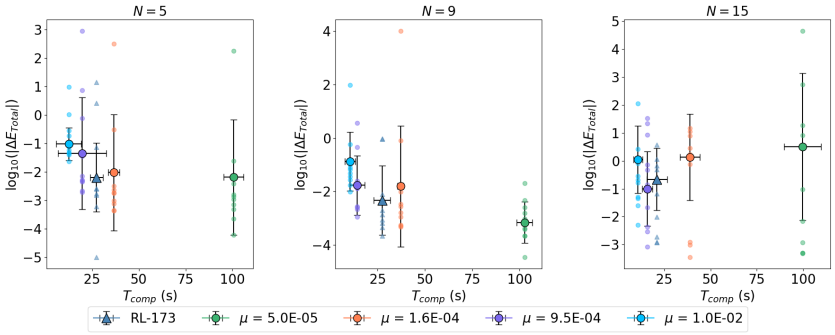


Figure 5.20: Results of the simulations run for 40 steps with the baseline settings.

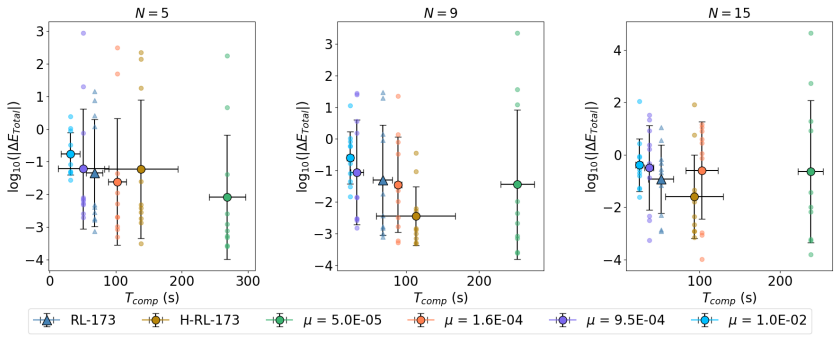


Figure 5.21: Results of the simulations run for 100 steps with the baseline settings. The solutions with the hybrid method are also displayed here.

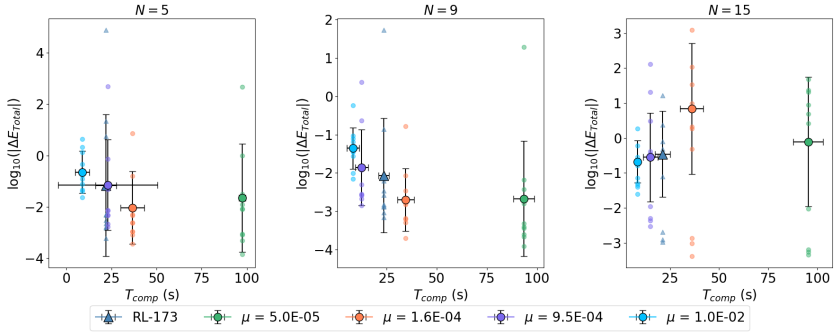


Figure 5.22: Results of the simulations run for 40 steps with a different choice of integrators.

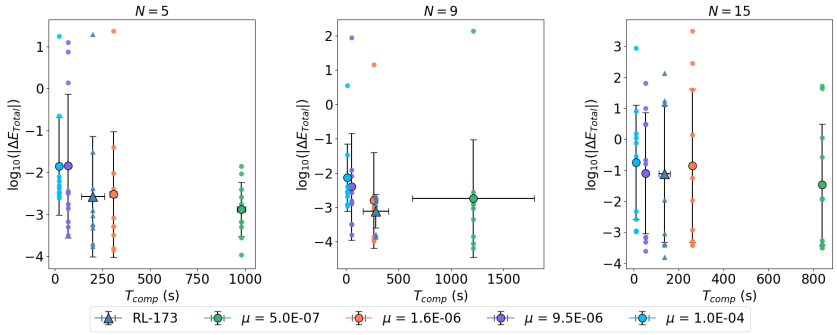


Figure 5.23: Results of the simulations run for 40 steps with a time-step parameter of 10^{-2}.

BIBLIOGRAPHY

Aarseth, S. A. 2003, Gravitational N-body Simulations (Cambridge University press, 2003)

Aarseth, S. J. 1985, in Multiple time scales (Elsevier), 377–418

Aarseth, S. J. & Lecar, M. 1975, Annual Review of Astronomy and Astrophysics, 13, 1

Abadi, M., Agarwal, A., Barham, P., et al. 2015, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, software available from tensorflow.org

Almojel, A. I. 2000, Computers & Electrical Engineering, 26, 297

Americo, M. 2017, The Classical Outlook, 92, 94

Antonion, K., Wang, X., Raissi, M., & Joshie, L. 2024, Academic Journal of Science and Technology, 9, 46

Barnes, J. & Hut, P. 1986, nature, 324, 446

Basuchoudhary, A., Bang, J. T., & Sen, T. 2017, Machine-learning Techniques in Economics: New Tools for Predicting Economic Growth (Springer)

Belkin, S. & Kuznetsov, E. 2021, Acta Astronautica, 178, 360

Boekholt, T. & Portegies Zwart, S. 2015a, Computational Astrophysics and Cosmology, 2

Boekholt, T. & Portegies Zwart, S. 2015b, Computational Astrophysics and Cosmology, 2, 1

Boekholt, T. & Portegies Zwart, S. 2015, Computational Astrophysics and Cosmology, 2, 1

Boekholt, T. & Portegies Zwart, S. 2015, Computational Astrophysics and Cosmology, 2, 2

Breen, P. G., Foley, C. N., Boekholt, T., & Portegies Zwart, S. 2020a, Monthly Notices of the Royal Astronomical Society, 494, 2465

Breen, P. G., Foley, C. N., Boekholt, T., & Portegies Zwart, S. 2020b, Monthly Notices of the Royal Astronomical Society, 494, 2465

Brockman, G., Cheung, V., Pettersson, L., et al. 2016a, arXiv preprint arXiv:1606.01540

Brockman, G., Cheung, V., Pettersson, L., et al. 2016b, OpenAI Gym

Burby, J., Tang, Q., & Maulik, R. 2021, Plasma Physics and Controlled Fusion, 63, 024001

Cai, M. X., Portegies Zwart, S., & Podareanu, D. 2021a, arXiv preprint arXiv:2111.15631

Cai, S., Wang, Z., Wang, S., Perdikaris, P., & Karniadakis, G. E. 2021b, Journal of Heat Transfer, 143, 060801

Capuzzo-Dolcetta, R., Spera, M., & Punzo, D. 2013, Journal of Computational Physics, 236, 580

Chen, R. & Tao, M. 2021, arXiv preprint arXiv:2103.05632

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. 2018, in Advances in Neural Information Processing Systems, Vol. 31 (Curran Associates, Inc.)

Chen, Z., Zhang, J., Arjovsky, M., & Bottou, L. 2020, in 8th International Conference on Learning Representations, ICLR 2020

Chevallier, F., Chéruy, F., Scott, N., & Chédin, A. 1998, Journal of Applied Meteorology, 37, 1385

Copernicus, N. 1543, On the Revolutions of the Heavenly Spheres (Princeton University Press)

Coronato, A., Naeem, M., De Pietro, G., & Paragliola, G. 2020, Artificial Intelligence in Medicine, 109, 101964

Curtis, H. D. 2019, Orbital mechanics for engineering students (Butterworth-Heinemann)

Dayan, P. & Niv, Y. 2008, Current opinion in neurobiology, 18, 185

Dellnitz, M., Hüllermeier, E., Lücke, M., et al. 2023, SIAM Journal on Scientific Computing, 45, A579

Doupe, P., Faghmous, J., & Basu, S. 2019, Value in Health, 22, 808

E, W. 2017, Communications in Mathematics and Statistics, 5, 1

Easton, R. W. 1993, SIAM Review, 35, 659

Elipe, A., Montijano, J., Rández, L., & Calvo, M. 2017, Celestial Mechanics and Dynamical Astronomy, 129, 415

Farea, A., Yli-Harja, O., & Emmert-Streib, F. 2024, AI, 5, 1534

Fujii, M., Iwasawa, M., Funato, Y., & Makino, J. 2007, Publications of the Astronomical Society of Japan, 59, 1095

Glorot, X. & Bengio, Y. 2010, in Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, 249–256

Goodwin, S. P. & Whitworth, A. P. 2004, Astronomy & Astrophysics, 413, 929

Greengard, L. 1990, Computers in Physics, 4, 142

Greydanus, S., Dzamba, M., & Yosinski, J. 2019a, CoRR, abs/1906.01563 [1906.01563]

Greydanus, S., Dzamba, M., & Yosinski, J. 2019b, in Advances in Neural Information Processing Systems, Vol. 32 (Curran Associates, Inc.)

Haber, E. & Ruthotto, L. 2017, Inverse Problems, 34, 014004

Hairer, E., Lubich, C., & Wanner, G. 2006, Geometric Numerical Integration (Springer Berlin), 644

Hambly, B., Xu, R., & Yang, H. 2023, Mathematical Finance, 33, 437

Heggie, D. & Hut, P. 2003, The Gravitational Million-Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics (Cambridge University Press, 2003)

Heggie, D. C. 1975, Monthly Notices of the Royal Astronomical Society, 173, 729

Heggie, D. C. & Mathieu, R. D. 1986, in The Use of Supercomputers in Stellar Dynamics, Vol. 267 (Springer), 233–235

Hénon, M. H. 1971, apss, 14, 151

Horn, P., Saz Ulibarrena, V., Koren, B., & Portegies Zwart, S. 2022, in ECCOMAS2022

Hornik, K. 1991, Neural Networks, 4, 251

Hung, S.-M. & Givigi, S. N. 2016, IEEE transactions on cybernetics, 47, 186

Hut, P., Makino, J., & McMillan, S. 1995, The Astrophysical Journal Letters, 443, L93

Imambi, S., Prakash, K. B., & Kanagachidambaresan, G. 2021, Programming with TensorFlow: solution for edge computing applications, 87

Irani, A. A., Leigh, N. W., Boekholt, T. C., & Zwart, S. P. 2024, Astronomy & Astrophysics, 689, A24

Jänes, J., Pelupessy, I., & Zwart, S. P. 2014, Astronomy & Astrophysics, 570, A20

Jia, P., Jia, Q., Jiang, T., & Liu, J. 2023, The Astronomical Journal, 165, 233

Jin, P., Zhang, Z., Zhu, A., Tang, Y., & Karniadakis, G. 2020a, Neural Networks, 132, 166

Jin, P., Zhang, Z., Zhu, A., Tang, Y., & Karniadakis, G. E. 2020b, Neural Networks, 132, 166

Kepler, J. 2015, Pragae 1609

Kingma, D. & Ba, J. 2014, International Conference on Learning Representations

Kingma, D. & Ba, J. 2015, in 3rd International Conference on Learning Representations, ICLR 2015

Kiran, B. R., Sobh, I., Talpaert, V., et al. 2022, IEEE Transactions on Intelligent Transportation Systems, 23, 4909

Kokubo, E. & Ida, S. 2002, The Astrophysical Journal, 581, 666

Krothapalli, U., Wagner, T., & Kumar, M. 2011, in Infotech@ Aerospace 2011, 1533

Lalande, F. & Trani, A. 2022, The Astrophysical Journal, 938, 18

Loh, W.-L. 1996, The Annals of Statistics, 24, 2058

Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. 2021, SIAM Review, 63, 208

Makino, J. 1991, The Astrophysical Journal, 369, 200

Makino, J. & Aarseth, S. J. 1992, Publications of the Astronomical Society of Japan, 44, 141

Makino, J., Hut, P., Kaplan, M., & Saygın, H. 2006, New Astronomy, 12, 124

Mansfield, L. A., Nowack, P. J., Kasoar, M., et al. 2020, npj Climate and Atmospheric Science, 3, 1

Mignard, F. 1982, Icarus, 49, 347

Mnih, V., Kavukcuoglu, K., Silver, D., et al. 2015, nature, 518, 529

Moster, B. P., Naab, T., Lindström, M., & O'Leary, J. A. 2021, Monthly Notices of the Royal Astronomical Society, 507, 2115

Newton, I. 1687, Newton: Principia Mathematica (Routledge), 97–105

Newton, I. 1999, The Principia: mathematical principles of natural philosophy (University of California Press)

Nitadori, K. & Makino, J. 2008, New Astronomy, 13, 498

Nousiainen, J., Rajani, C., Kasper, M., et al. 2022, Astronomy & Astrophysics, 664, A71

Novati, G., de Laroussilhe, H. L., & Koumoutsakos, P. 2021, Nature Machine Intelligence, 3, 87

Paszke, A., Gross, S., Massa, F., et al. 2019, in Advances in Neural Information Processing Systems, Vol. 32 (Curran Associates, Inc.), 8024–8035

Pedregosa, F., Varoquaux, G., Gramfort, A., et al. 2011, Journal of Machine Learning Research, 12, 2825

Pelupessy, F. I., Jänes, J., & Portegies Zwart, S. 2012, New Astronomy, 17, 711

Pham, D. N. 2024, arXiv preprint arXiv:2407.10037

Plummer, H. C. 1911, Monthly Notices of the Royal Astronomical Society, Vol. 71, p. 460-470, 71, 460

Portegies Zwart, S. & McMillan, S. 2018, Astrophysical Recipes; The art of AMUSE

Portegies Zwart, S., McMillan, S., Harfst, S., et al. 2009, New Astronomy, 14, 369

Raissi, M., Perdikaris, P., & Karniadakis, G. E. 2019a, Journal of Computational physics, 378, 686

Raissi, M., Perdikaris, P., & Karniadakis, G. E. 2019b, Journal of Computational Physics, 378, 686

Rauch, K. P. & Holman, M. 1999, The Astronomical Journal, 117, 1087

Rein, H., Tamayo, D., & Brown, G. 2019, Monthly Notices of the Royal Astronomical Society, 489, 4632

Richardson, D., Michel, P., Walsh, K., & Flynn, K. 2009, Planetary and Space Science, 57, 183

Roa, J., Hamers, A. S., Cai, M. X., & Leigh, N. W. 2020, Moving Planets Around (The MIT Press)

Salpeter, E. E. 1955, Astrophysical Journal, vol. 121, p. 161, 121, 161

Sanz-Serna, J. M. 1992, Acta Numerica, 1, 243–286

Saz Ulibarrena, V. & Portegies Zwart, S. 2024, Submitted to Communications in Nonlinear Science and Numerical Simulation

Srivastava, N., Kaufman, C., & Müller, G. 1990

Stone, N. C. & Leigh, N. W. C. 2019, Nature, 576, 406

Sutton, R. S. & Barto, A. G. 2018, Reinforcement Learning: An Introduction (MIT press)

Tamayo, D., Silburt, A., Valencia, D., et al. 2016, The Astrophysical Journal Letters, 832, L22

Telgarsky, M. 2015, `https://arxiv.org/abs/1509.08101`Representation Benefits of Deep Feedforward Networks

Toomer, G. 1998, Ptolemy's Almagest (Princeton University Press)

Tremaine, S. 2015, The Astrophysical Journal, 807, 157

Turaev, D. 2002, Nonlinearity, 16, 123

Ulibarrena, V. S., Horn, P., Zwart, S. P., et al. 2024, Journal of Computational Physics, 496, 112596

Verlet, L. 1967, Physical Review, 159, 98

Viquerat, J., Meliga, P., Larcher, A., & Hachem, E. 2022, Physics of Fluids, 34

White, D. B. 2022, in ASCEND 2022, 4342

Wisdom, J. & Holman, M. 1991, Astronomical Journal (ISSN 0004-6256), vol. 102, Oct. 1991, p. 1528-1538., 102, 1528

Xiong, S., Tong, Y., He, X., et al. 2021, in 9th International Conference on Learning Representations, ICLR 2021

Yahalom, A. 2022, Symmetry, 15, 39

Yahalom, A. 2024, Entropy, 26, 986

Yatawatta, S. & Avruch, I. M. 2021, Monthly Notices of the Royal Astronomical Society, 505, 2141

Yi, K., Moon, Y.-J., & Jeong, H.-J. 2023, The Astrophysical Journal Supplement Series, 265, 34

Yoshida, H. 1990a, Physics Letters A, 150, 262

Yoshida, H. 1990b, Physics Letters A, 150, 262

Yu, W., Wang, R., Li, R., Gao, J., & Hu, X. 2018, in 2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI), IEEE, 6–11

Zaghbani, I., Jarray, R., & Bouallègue, S. 2024, in 2024 IEEE 28th International Conference on Intelligent Engineering Systems (INES), IEEE, 000245–000250

Zemp, M., Stadel, J., Moore, B., & Carollo, C. M. 2007, Monthly Notices of the Royal Astronomical Society, 376, 273

Zhu, A., Jin, P., & Tang, Y. 2020, arXiv preprint arXiv:2004.13830

Zwart, S. F. P., McMillan, S. L., van Elteren, A., Pelupessy, F. I., & de Vries, N. 2013, Computer Physics Communications, 184, 456

Zwart, S. P. & McMillan, S. 2018, Astrophysical Recipes: the art of AMUSE (IoP Publishing)

Zwart, S. P., Pelupessy, I., Martínez-Barbosa, C., van Elteren, A., & McMillan, S. 2020, Communications in Nonlinear Science and Numerical Simulation, 85, 105240