Efficient time stepping for numerical integration using reinforcement learning

Michael Dellnitz[1], Eyke Hüllermeier[2], Marvin Lücke[3], Sina Ober-Blöbaum[1], Christian Offen[1], Sebastian Peitz[4], and Karlson Pfannschmidt[4]

[1] *Department of Mathematics, Paderborn University*
[2] *Department of Computer Science, LMU Munich*
[3] *Modeling and Simulation of Complex Processes, Zuse Institute Berlin*
[4] *Department of Computer Science, Paderborn University*

Abstract

Many problems in science and engineering require an efficient numerical approximation of integrals or solutions to differential equations. For systems with rapidly changing dynamics, an equidistant discretization is often inadvisable as it either results in prohibitively large errors or computational effort. To this end, adaptive schemes, such as solvers based on Runge–Kutta pairs, have been developed which adapt the step size based on local error estimations at each step. While the classical schemes apply very generally and are highly efficient on regular systems, they can behave sub-optimal when an inefficient step rejection mechanism is triggered by structurally complex systems such as chaotic systems. To overcome these issues, we propose a method to tailor numerical schemes to the problem class at hand. This is achieved by combining simple, classical quadrature rules or ODE solvers with data-driven time-stepping controllers. Compared with learning solution operators to ODEs directly, it generalises better to unseen initial data as our approach employs classical numerical schemes as base methods. At the same time it can make use of identified structures of a problem class and, therefore, outperforms state-of-the-art adaptive schemes. Several examples demonstrate superior efficiency. Source code is available at https://github.com/lueckem/quadrature-ML.

1 Introduction

The numerical treatment of a vast number of problems in science and engineering requires the approximation of integrals, such as the computation of volume or mass flow, or the numerical solution of differential equations via *Runge–Kutta methods* [2]. Consequently, schemes for numerical discretization are a key element of scientific computing, and one of the main challenges is to determine a good trade-off between the required accuracy and numerical efficiency.

While the standard approach to developing *quadrature rules* and *numerical schemes to solve ODEs* is based on Taylor series expansions and the associated error bounds determined by higher-order derivatives [2], the advances in data science and machine learning have recently fueled the development of alternative concepts that are based on *training data.* Most of these approaches are developed with the aim to efficiently compute the numerical solution of dynamical systems of high complexity, see, for instance, [18, 25, 15, 19, 20, 12], where the flow map F that takes a state x at time t to a future state $x(t + \Delta t)$ is approximated.

In contrast to that, our work addresses the task of efficiently performing numerical integration for integrands or differential equations of a given problem class to a desired accuracy. To this end, the next sample point at which the integrand or force term is evaluated is determined by finding an *optimal trade-off* between the two conflicting criteria accuracy and numerical efficiency. This task is carried out by a *reinforcement learning* algorithm which, taking past function evaluations and learned knowledge about the problem class into account, determines the next sample point at which to evaluate the function of interest. The efficiency of the proposed approach in comparison to state-of-the-art methods will be demonstrated using examples from the area of computing integrals (quadrature) as well as numerically solving differential equations.

The second key component of quadrature rules or numerical methods — the appropriate weighting of the different function evaluations — will be addressed briefly. While the classical Taylor-series-based construction aims to maximize the asymptotic order of a method on the class of analytic problems, given a finite accuracy and a restricted problem class, tailored weights can yield even more efficient schemes, as will be demonstrated by experiments.

We give a brief overview of numerical integration as well as reinforcement learning in Section 2. In Section 3 we describe our method of learning an optimal time-stepping algorithm for specific problem classes from data, and how to optimize the weights of conventional quadrature rules in order to obtain an even more efficient scheme. We apply these methods to multiple examples in Section 4 and compare the performance to state-of-the-art adaptive time-steppers. Finally, we draw a conclusion in Section 5.

2 Preliminaries

We will briefly review classical numerical quadrature rules and algorithms for differential equations as well as provide an overview of the reinforcement learning framework we will use in our algorithm.

2.1 Numerical Quadrature

The task in numerical quadrature is to efficiently determine the value of the integral

$$I = \int_0^T f(t)\mathrm{d}t \tag{1}$$

from a few function evaluations, while maintaining a high accuracy. The most wide-spread strategy is to divide the interval $[0, T]$ into subintervals $I_i = [t_i, t_{i+1}]$ of length h and to apply a quadrature formula

$$I_i = h \sum_{j=1}^{s} \omega_j f(t_i + c_j h) \tag{2}$$

with fixed nodes $0 \leq c_1 < \ldots < c_s \leq 1$ and weights $\omega_j \in \mathbb{R}$ in each subinterval [3]. The standard numerical quadrature schemes implemented in popular software including MATLAB, Python's SciPy, or the FORTRAN subroutine package QUADPACK combine subdivision strategies with classical quadrature formulas. Their general purpose algorithms are most prominently based on Gauss-Kronrod quadrature [1, 17, 22]. Gauss-Kronrod quadrature approximates the value of the integral using $s = 2r + 1$ nodes and weights such that polynomials up to including order $3r+1$ can be integrated exactly (degree of exactness). The error is of order $3r + 3$ in the length of the biggest subinterval h. A subset of r nodes can be used in a Gauss quadrature formula such that another approximation with degree of exactness $2r - 1$ and of order $2r + 2$ in h can be obtained without any further evaluations of the integrand. Comparing both approximations provides an estimate for the numerical error in the subinterval, based on which further subdivisions are performed. The nodes are in an irrational relation which enhances the robustness of the error estimate, as argued in [22], for instance. However, as a consequence, previous function evaluations cannot be used in subsequent steps.

Many software packages employ the above mentioned adaptive Gauss-Kronrod subdivision algorithm with 21 nodes ($r = 10$) as a default, which we will refer to as `GK21`. Moreover, we denote the associated Kronrod quadrature rule with 21 nodes by `K21` and the Gauss quadrature rule with 21 nodes by `G21`.

2.2 Numerical integration of differential equations

Consider the initial value problem $\dot{x} = f(t, x)$ with initial condition $x(0) = x_0$ on the time interval $[0, T]$. Reformulating the problem in integral form

$$x(T) = x_0 + \int_0^T f(t, x(t))\mathrm{d}t \tag{3}$$

and applying numerical quadrature rules gives rise to integration methods for ordinary differential equations such as *Runge–Kutta* methods [2]. Time-stepping proceeds as follows: for a given step size $h_0 > 0$, first the approximate value x_1 of the solution $x(t_1)$ at time $t_1 = t_0 + h_0$ is computed. Next, the process is repeated with (t_1, h_1, x_1) replacing (t_0, h_0, x_0) to compute x_2 for $h_1 > 0$. Iteration yields an approximation $x_0, x_1, x_2, \dots$ of the solution x at times $t_0, t_1, t_2, \dots$, where $t_i = t_{i-1} + h_{i-1}$ is recursively defined.

While in the simplest case all step sizes h_i are identical (constant time-stepping), more elaborate schemes choose step sizes dynamically based on error estimates that are available from previous steps. In this way, numerical integration can benefit from large time steps in regions where the solution does not change rapidly while satisfying accuracy requirements in regions which require small step sizes [9, 8]. Efficient time-stepping methods that are implemented in MATLAB or Python's SciPy package [28] include most prominently the Dormand–Prince pair `RK45`, consisting of a fifth order Runge–Kutta scheme together with a fourth order method that is used to estimate the local error of each step. Based on the error estimate, a step size controller then computes the step size for the next iteration [4, 23]. If the error estimate of a step is larger than some desired tolerance, the proposed step is rejected and the time-stepper tries again with a smaller step size, hence, wasting function evaluations. Other methods include the 8th order scheme `DOP853` [9, 21], or multi-step strategies for stiff equations [9] as well as automatic solver selection strategies [30].

The above-mentioned black box solvers are designed for an application to a broad range of problems. However, for problems with additional structure (for instance, Hamiltonian systems), specialized algorithms achieve significant advantages over off-the-shelf methods by making use of symmetries, conserved quantities, or symplecticity [7]. On the other hand, a lack of regularity can cause order reductions or instabilities of classical integration schemes that are hard to overcome, though approaches are available in some cases [13].

2.3 Data-driven approaches to time-stepping

While specialized numerical integration schemes have traditionally been developed by manually identifying structures first and then designing appropriate numerical schemes, in the following we will explore a data-driven approach to develop time-stepping strategies that adapt automatically to given problem classes and error tolerances. In our approach, these strategies are learned by a neural network for specific problem classes, such as irregular integrands or dynamical systems that exhibit chaotic motions. Here, classical adaptive schemes can suffer from inefficiencies, for instance because proposed steps get rejected frequently when an error estimate becomes too large. In comparison, our data-driven scheme can learn features specific to the problem class and use that knowledge to optimize its time-stepping strategy.

2.3.1 Reinforcement learning

In reinforcement learning, an agent acting in an environment E, typically modelled in the form of a Markov decision process $(\mathcal{S}, \mathcal{A}, p, r)$, is considered with $\mathcal{S}$ the set of possible states, $\mathcal{A}$ the set of possible actions, p the so-called transition function, and r the reward function. At each discrete time step i, the agent finds itself in a state $s_i \in \mathcal{S}$ of the environment. It decides on an action $a_i \in \mathcal{A}$ and receives an immediate reward $r_i = r(s_i, a_i) \in \mathbb{R}$ that depends on the current state and the action taken (and perhaps also on the successor state). The behavior of the agent is captured by its *policy* $\pi \colon \mathcal{S} \to P(\mathcal{A})$ that prescribes actions given states. More specifically, the policy is not necessarily deterministic and defines a probability distribution over the actions available in a given state. Likewise, the environment E can be stochastic: the successor state s_{i+1} resulting from action a_i in the current state s_i, is determined by the transition function $p : \mathcal{S} \times \mathcal{A} \to P(\mathcal{S})$. In the following, we will use the notation $r_i, s_{i+1} \sim E$ as a shorthand whenever s_i and a_i are clear from the context.

Given a sequence of states and actions $(s_i, a_i, s_{i+1}, a_{i+1}, \dots, s_T)$ starting at state s_i, R_i denotes the sum of discounted future rewards $R_i = \sum_{j=i}^{T-1} \gamma^{(j-i)} r(s_j, a_j)$, where $\gamma \in [0, 1]$ is a discount factor. The actions are the result of sampling from the (stochastic) policy π, which is not known in advance. The goal is to learn a policy which, given an initial state $s_0 \in \mathcal{S}$, maximizes the expected sum of discounted future rewards.

We consider the expected future rewards with respect to a specific action. The action-value function is defined as follows:

$$Q^\pi(s_i, a_i) = \mathbb{E}_\pi[R_i \mid s_i, a_i] \tag{4}$$

It assumes that the agent picks action a_i in state s_i, while subsequently picking actions according to policy π. This definition can be further unrolled into a recursive definition:

$$Q^\pi(s_i, a_i) = \mathbb{E}_{r_i, s_{i+1} \sim E}\big[r(s_i, a_i) + \gamma \mathbb{E}_{a_{i+1} \sim \pi}[Q^\pi(s_{i+1}, a_{i+1})]\big] \tag{5}$$

This equation is known as the Bellman equation [14, 26]. Notice that the outer or inner expectation is not needed if the environment or policy function is deterministic. The function

$$Q(s_i, a_i) = \mathbb{E}_{r_i, s_{i+1} \sim E}\Big[r(s_i, a_i) + \gamma \max_{a_{i+1} \in \mathcal{A}} Q(s_{i+1}, a_{i+1})\Big], \tag{6}$$

evaluates the action a_t in state s_t based on the premise that optimal actions are chosen in subsequent steps, and is called *Q-function*. The optimal policy $\pi^* \colon \mathcal{S} \to \mathcal{A}$ can be characterized as

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q(s, a). \tag{7}$$

In this article, we use a reinforcement learning approach based on Q-learning [29], where the Q-function is approximated by a neural network parametrized by θ, which we will denote by Q_θ. The parameter θ is initialized using a suitable initialization scheme [6]. The training then proceeds over the course of several episodes. In each episode the learner starts in a state $s_0 \in \mathcal{S}$ of the environment and explores the state space using the function Q_θ in conjunction with a probabilistic policy. This results in a trajectory $((s_i, a_i, r_i))_{i=0}^{H}$, where s_i is the state in step i, a_i the chosen action, and r_i is the resulting reward. The horizon H is the final time step of the trajectory and marks the end of one episode. The training targets Q' are defined as follows:

$$Q'(s_i, a_i) = r_i + \gamma \max_{a \in \mathcal{A}} Q_\theta(s_{i+1}, a), \tag{8}$$

where $\gamma \in [0, 1]$ is a discount factor for future rewards. Given a loss function $L \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, the parameter vector $\hat{\theta}$, to be used in the next episode, is calculated using empirical risk minimization:

$$\hat{\theta} \leftarrow \arg\min_{\theta \in \Theta} \frac{1}{H+1} \sum_{i=0}^{H} L\big(Q'(s_i, a_i), Q_\theta(s_i, a_i)\big) + R(\theta), \tag{9}$$

where $R(\theta)$ is an additional regularization term. In the following, we use the typical L_2 loss, i. e., $L(y, \hat{y}) = \|y - \hat{y}\|_2$. We continue to run episodes until the Q-function converges. This approach and similar variants are often referred to as *deep Q-learning* [16]. While convergence to the exact Q-function is only rigorously proved for classical Q-learning in a finite state and action space setting [10], there are numerous publications showing the success of deep Q-learning in many applications, see [5] for an overview.

3 An RL algorithm for efficient adaptive integration

Our algorithm addresses two main challenges of numerical integration schemes:

(i) the selection of step sizes in situations of rapidly changing behavior (e. g., bursts),

(ii) qualitative changes in behavior, for instance in hybrid or switched systems, or systems which chaotic and regular regions.

The approach is to train a model that can recommend optimal step sizes to use during integration and is superior to classical adaptive schemes. We define an optimal step size as being as large as possible while a certain desired error tolerance is not exceeded in the integration step. This allows us, in particular, to address issues such as the use of too large step sizes ("overshooting") when the integrand changes rapidly, while being too conservative in other regions. The proposed algorithm employs reinforcement learning to adapt to a particular class of problems. The learning only has to be conducted once, and the obtained model can then be used to integrate similar functions much more efficiently. We describe the training of the efficient time-stepper in Section 3.1.

Moreover, given a specific class of problems, the weights of classical quadrature and integration methods may be suboptimal. We show how to calculate weights that are specifically tailored to the problem class in order to obtain an even more efficient scheme in Section 3.2.

3.1 Training the time-stepper via reinforcement learning

We first discuss training an efficient time-stepper for quadrature tasks in Section 3.1.1. Then we focus on the numerical integration of ordinary differential equations (ODEs) in Section 3.1.2.

3.1.1 Quadrature tasks

We assume that the functions we want to integrate are sampled from a set X with probability measure P, i.e., a class of problems, and aim to train a neural network (NN) to perform the subdivision of an initial integration interval into subintervals. The quadrature rule that is used to integrate the subintervals is fixed. We restrict the algorithm to integrate "from left to right", that is, given the current subinterval $[t_i, t_{i+1}]$, the NN chooses the next step size (subinterval length) h^+ so that the next subinterval is given by $[t_{i+1}, t_{i+1} + h^+] = [t_{i+1}, t_{i+2}]$. Provided with a sufficient amount of training data, the NN may be able to predict the future course of the function to some extent and, thus, choose a suitable step size.

Assuming the quadrature rule uses s function evaluations $(f(t_i + c_j h))_{j=1,\ldots,s}$ in the subinterval $[t_i, t_{i+1}]$, where the c_j are the nodes of the quadrature rule (cf. (2)), we provide the subinterval length h as well as the function evaluations as an input to the time-stepper. It can then select the subsequent subinterval length h^+ from a finite set of options $\{h_1, \ldots, h_n\}$.

The time-stepping NN is trained via Q-learning [29]. More specifically, the NN approximates the Q-function, where Q receives the *state* s_i at step i and an *action* a_i as inputs which, in our case, are

$$\begin{aligned} s_i &= (h, f(t_i + c_1 h), \ldots, f(t_i + c_s h)) \,, \\ a_i &= h^+ \,. \end{aligned}$$

As described in equation (8), the value of the Q-function can be defined implicitly as

$$Q(s_i, a_i) = r_i + \gamma \max_{a_{i+1} \in \mathcal{A}} Q(s_{i+1}, a_{i+1}),$$

with reward r_i and discount factor $\gamma \in [0, 1]$. In preliminary experiments, a discount factor of $\gamma = 0$ resulting in a myopic Q-function yielded the same performance as non-zero values. Therefore, we simply set $\gamma = 0$ in all experiments. The neural network receives s_i as an input and returns all possible Q-values, i. e., the output is defined as

$$(Q(s_i, h_1), \ldots, Q(s_i, h_n)) \,.$$

We then simply select the action with the highest Q-value, i. e., the highest expected reward, as our next step size. The training of the neural network is conducted as described in Section 2.3.1, that is, for every episode a function f is sampled from the problem class, integrated by the time-stepper, and the resulting rewards used to optimize the weights of the NN. To ensure sufficient exploration during the training phase, we do not always select the step size with the highest Q-value, but employ a probabilistic policy that sometimes selects one of the other step sizes randomly.

Reward function

To learn a strategy that selects a step size that is as large as possible while an error tolerance tol is not exceeded, we have to choose an appropriate reward function. In our case, the reward r_i should depend on the integration error ε in the integration step t and on the selected step size h^+. Here, the integration error ε is defined as

$$\varepsilon = |I - \hat{I}|,$$

where $I = \int_{t_i}^{t_{i+1}} f(t)\, \mathrm{d}t$ is the exact integral of the current step and $\hat{I}$ denotes the result of the quadrature formula. If the exact integral I is not known, we approximate it numerically with high accuracy (e. g., using a cumulative quadrature with step size multiple orders smaller than h^+). A straightforward definition of the reward function is

$$r_i = \begin{cases} 0, & \varepsilon > \mathsf{tol} \\ h^+, & \varepsilon < \mathsf{tol} \end{cases} .$$

However, in practice, it is advantageous to use a reward function with negative rewards for larger errors. For problems exhibiting chaotic behavior such as the ones we consider in this paper, we expect integration errors to be scattered across several orders of magnitude. This is why we chose to use the following reward function

$$r_i = \begin{cases} \log_{10}(\frac{\mathsf{tol}}{\varepsilon}), & \varepsilon > \mathsf{tol} \\ a \log(b \cdot h^+), & \varepsilon < \mathsf{tol} \end{cases} , \tag{10}$$

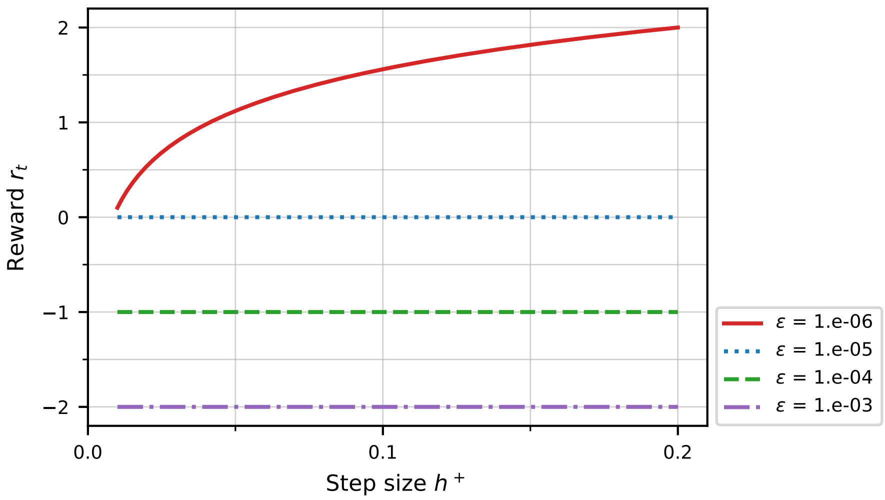


Figure 1: Behavior of the reward function (10) for varying ϵ. The error tolerance is fixed at 1×10^{-5}.

which combines both properties. It generates a reward of $r_i = -m$ if $\varepsilon = 10^m \cdot \mathsf{tol}$. The parameters a and b are chosen such that the positive and negative rewards are roughly on the same scale. The behavior of reward function (10) with respect to varying integration error ϵ is shown in Figure 1. This is the reward function we employ for all experiments throughout the paper. Generally speaking, problem-depending reward functions can be necessary for efficient training.

Neural network architecture

Throughout this paper, we use fully connected neural networks with four hidden layers and five times the number of input nodes per hidden layer. It should be noted that for the examples studied in the following, a significantly smaller network topology might be sufficient. All NNs use rectified linear unit (ReLU) activation functions and employ the Adam stochastic gradient descent optimizer [11] for training.

Limitations

It is apparent that the above constructed method can only be successful if the information contained in the state s_i, i.e., the function evaluations in one subinterval $[t_i, t_{i+1}]$, is sufficient to form some belief about the structure of $f(t)$ for $t > t_{i+1}$. Thus, we do not expect the method to perform particularly well on very broad classes of problems, where past function evaluations are not informative enough about possible continuations of the function. We do expect our method to excel on function classes of which the functions share some common and predictable behavior.

When the information contained in the state s_i is not sufficient to form an adequate prediction of the structure of f, one can supplement past states to the input of the time-stepper, thereby increasing the available information. However, this considerably increases the complexity of the learning problem and, in the experiments that we conducted, did not lead to a significant increase in performance that would justify the additional complexity.

3.1.2 Numerical Integration of Differential Equations

The approach discussed above can readily be extended to the numerical integration of ordinary differential equations (ODEs). Here Runge-Kutta methods take the role of quadrature rules. A Runge-Kutta method with s stages can be written as

$$x(t_{i+1}) - x(t_i) = \int_{t_i}^{t_{i+1}} f(t, x(t)) \mathrm{d}t \approx h \sum_{j=1}^{s} b_j k_j, \quad h = t_{i+1} - t_i \,.$$

The stages k_j correspond to approximations of the function values $f(\tilde{t}_j, x(\tilde{t}_j))$ for s distinct time points $\tilde{t}_j \in [t_i, t_{i+1}]$, and thus, correspond to the function evaluations at the nodes in the quadrature setting. The weights b_j correspond to the weights of a quadrature rule.

Hence, we can apply the same method as presented in Section 3.1.1 in order to train an efficient time-stepper. The input of the NN is now given by the state

$$s_i = (h, k_1, \dots, k_s).$$

Note that the stages k_j are vectors, such that the total input dimension of the NN is $1 + sd$, where d is the dimension of $x(t)$. We define the integration error for the step i, which is needed to calculate the reward, as the deviation (in 2-norm) of the Runge Kutta estimate for $x(t_{i+1})$ from the correct solution when starting at the same initial value as the Runge Kutta estimate (local, step-wise error). If the exact solution $x(t_{i+1})$ is not known, we approximate it numerically with sufficiently high accuracy.

3.2 Optimization of weights

Now that we have constructed a reinforcement learning approach, which is able to train accurate models for efficient time-stepping, it is natural to ask what happens, if we adapt the weights of the quadrature rule to the given problem class as well. Recall that a quadrature rule is of the following general form (cf. (2))

$$\hat{I}_i = h \sum_{j=1}^{s} \omega_j f(t_i + c_j h)$$

for suitable choices of the nodes c_j and the weights ω_j. In case of ODEs, Runge-Kutta methods use the recurrence relation

$$x_{i+1} = x_i + h \sum_{j=1}^{s} b_j k_j \ ,$$

where the b_j are the weights, k_j the stages, and x_i the states of the trajectory.

We propose to optimize these weights (i. e., ω_j or b_j) using linear regression, such that the expected integration error for the specific problem class is minimized. For the case of numerical quadrature, we consider input instances of the form $\mathbf{f}_i = (f(t_i + c_1 h), \dots, f(t_i + c_s h))$, while the ground-truth integral I_i is the output to be predicted. (Again, if I_i is not analytically known, we approximate it numerically with high accuracy.) The optimization problem

$$\mathbf{w}^* = \underset{\mathbf{w} \in \mathbb{R}^s}{\arg\min} \sum_{i=1}^{n} (h_i \mathbf{w}^T \mathbf{f}_i - I_i)^2,$$

with n being the size of the dataset, can then be solved using off-the-shelf libraries.

The application to Runge-Kutta methods is straightforward as well. During each evaluation of our reinforcement learning model on a random trajectory, we record the vectors $\mathbf{k} = (k_1, \dots, k_\ell)$ to be used as input, while we let $\Delta = x(t_{i+1}) - x_i$ be the corresponding output to be predicted, where $x(t_{i+1})$ is the correct solution when starting at $x(t_i) = x_i$. (Again, if $x(t_{i+1})$ is not analytically known, we approximate it numerically with high accuracy.) In case the state-space of the ODE is multi-dimensional (with d denoting the number of dimensions), we split each instance into d separate ones, since we are only interested in one global set of weights. As before, we solve

$$\mathbf{b}^* = \underset{\mathbf{b} \in \mathbb{R}^s}{\arg\min} \sum_{i=1}^{n} (h_i \mathbf{b}^T \mathbf{k}_i - \Delta_i)^2,$$

to obtain an optimized weight vector $\mathbf{b}^*$.

As the optimal time-stepping technique may depend on the integration formula used, and vice-versa, it is recommended to train the time-stepping and to optimize the weights synchronously. To avoid the weights to be overly adapted to one particular trajectory, we use an exponentially weighted moving average

$$\mathbf{b}_{\text{ep}+1} = \alpha \mathbf{b}^* + (1 - \alpha) \mathbf{b}_{\text{ep}}$$

to update the weights, where ep was the last reinforcement learning epoch performed and $\mathbf{b}^*$ are the optimized weights for that epoch. In our experiments $\alpha = 0.05$ produced stable results. In the following numerical experiments, we evaluate the performance of our reinforcement learning approach both with and without weight optimization.

4 Numerical experiments

In this section, we will apply the methods presented above to several example problems and compare their performance to state-of-the-art adaptive techniques. First, we will discuss learning an optimal subdivison of an interval into subintervals for challenging quadrature tasks in Section 4.1. Then we address learning an optimal time-stepper for the numerical solution of differential equations in Section 4.2.

4.1 Numerical quadrature

The most challenging tasks in numerical quadrature are, besides functions with singularities or jumps, functions that exhibit a quickly changing and erratic behavior on different scales, so that the optimal subinterval sizes differ significantly. We show that the efficient time-stepping method we propose can successfully deal with such functions in the following example.

4.1.1 Example: Double Pendulum

Due to their frequent and sharp changes of direction, it is difficult to perform quadrature on the angular velocities of a double pendulum, e. g., in order to recover the angular positions of the two pendulums. The double pendulum system is explained in more detail in Section 4.2.3, where we learn a time-stepper to efficiently solve the double pendulum ODE. Here, we will assume a trajectory $f(t)$ of the angular velocity of the first pendulum is given for $0 \leq t \leq 100$ and the task is to calculate the integral $\int_0^{100} f(t)\mathrm{d}t$.

We calculate the trajectory $f(t)$ using the `RK45` adaptive solver on the double pendulum ODE with very high accuracy. The initial condition for the double pendulum is randomly sampled from a fixed energy set. Hence, we obtain a class of functions f equipped with some probability measure, and the goal is to find an optimal efficient time-stepping technique to divide the integration interval into subintervals. Here, “optimal” refers to using the least expected function evaluations while maintaining an expected integration error lower than some bound (expectation with respect to the above mentioned probability measure).

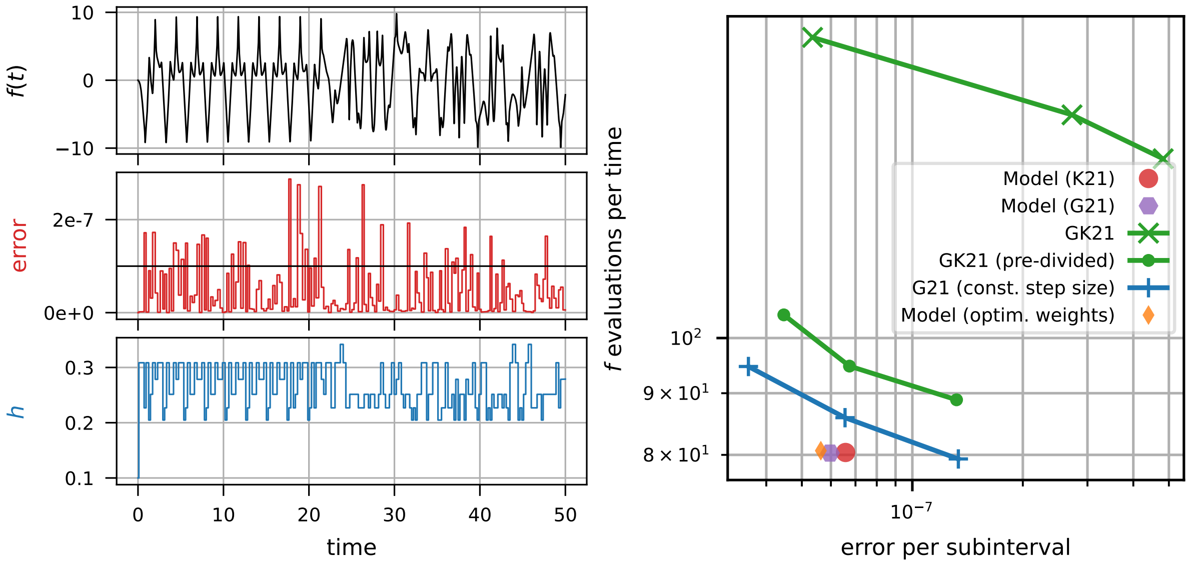
We set the error tolerance to $\mathsf{tol} = 10^{-7}$ and train a time-stepping model (as described in Section 3.1) with 20 step sizes evenly spaced on a log scale $h \in [0.1, 0.7]$. For the quadrature itself we employ the Kronrod rule with 21 nodes (`K21`) on each subinterval for a fair comparison to the Gauss-Kronrod adaptive subdivision algorithm (`GK21`), which is the default adaptive quadrature rule in many software packages like Python's *SciPy*.

The results are shown in Figure 2. As the instance of a typical time-stepping in 2a indicates, the model predominantly selects time steps (subinterval lenghts) between 0.2 and 0.3 and achieves subinterval integration errors close to the desired tolerance. The average error, which is shown in 2b, is slightly below the desired tolerance at about 7×10^{-8} and the model uses approximately 80 function evaluations per time unit. In comparison, the standard adaptive subdivision algorithm `GK21` needs substantially more evaluations to achieve a similar error. The reason is that it needs to execute many subdivisions before subintervals are of an appropriate length (~ 0.3) for the required tolerance, and already evaluated function values are not reusable in the next subdivision step[1]. Hence, this comparison is not fair, as we only gave interval lengths of appropriate size ($0.1 \leq h \leq 0.7$) to our model as options. Therefore, we also computed the performance of `GK21` after having pre-divided the whole integration interval $[0, 100]$ into subintervals of length 0.35, and then applying `GK21` to each subinterval. As can be seen in Figure 2b, this approach still performs rather poorly compared to our model.

Note that there exist adaptive subdivision schemes, where function evaluations can be reused in subsequent subdivision steps, e.g., Romberg quadrature. Such quadrature rules necessarily evaluate the integrand on nodes that are in a rational relation. However, high order quadrature rules, such as Gauss quadrature, require the evaluation of the integrand on an irrational grid. Moreover, it has been argued [22] that using irrational grids improves the robustness of error estimations as correlations between function values are avoided. [2] Independently on whether function evaluations can be reused or not, subdivision techniques are generally not efficient for the task at hand. The reason is that small tweaks to the step

[1] For example, dividing the initial interval $[0, 100]$ into $[0, 50]$ and $[50, 100]$ already “costs” 21 function evaluations, and so on.

[2] In interpreted languages such as Python or MATLAB the bottleneck to efficiency is in many cases the interpreter rather than evaluations of complex integrands. The default choices of quadrature algorithms have been justified in [22] by the fact that after each subdivision the positions, where the integrand needs to be evaluated, can be passed to the function representing the integrand all at once in a vectorized form. In this context, effectively only the number of times a subdivision needs to be performed counts.



(a) time-stepping

(b) Performance comparison

Figure 2: Quadrature of double pendulum. (a) time-stepping (division into subintervals) obtained by the model. (b) Performance comparison between the trained model and GK21. We show the model with K21 and G21 quadrature rules as well as optimized weights for the G21 nodes. "GK21" refers to the adaptive subdivision algorithm applied to the whole integral from $t = 0$ to $t = 100$ (plotted are three different tolerances for subdivision); for "GK21 (pre-divided)" the whole interval was first pre-divided into subintervals of length 0.35 and then GK21 was applied to each subinterval. Furthermore, we show G21 applied to subintervals of constant sizes. All points are the average over an ensamble of 20 integrals, where the integrand $f(t)$ is sampled as described above.

size are required, e.g., between 0.2 and 0.3, whereas subdivision techniques at least double the number of evaluations in a subdivision step. Hence they often produce an unnecessarily precise approximation when they choose to further subdivide a subinterval, or a too large error when they choose to not subdivide.

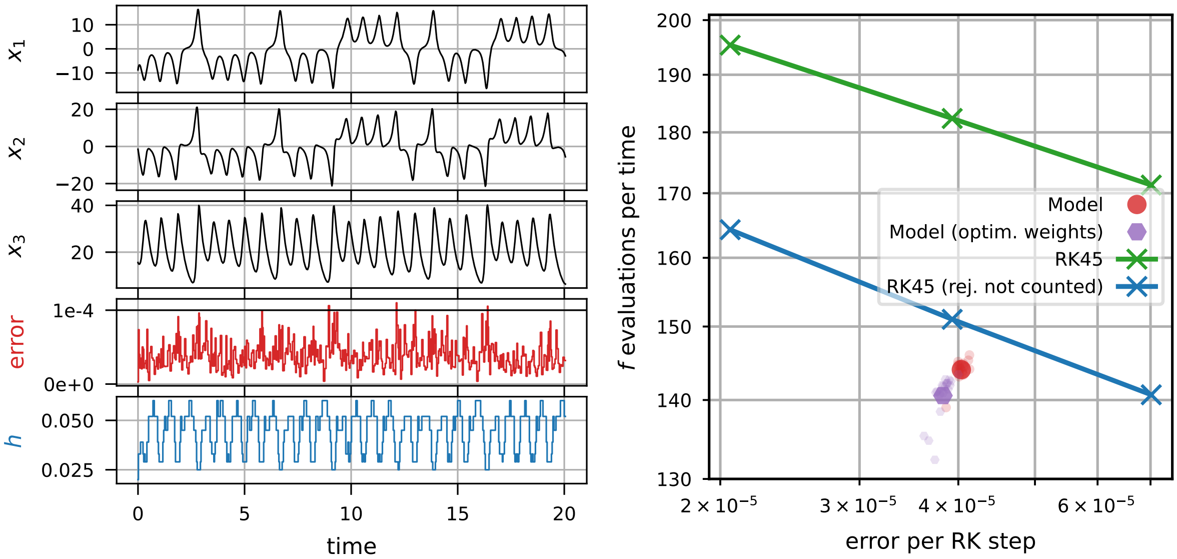
Due to the inefficiency of subdivision techniques, we also compared our trained model against a G21 quadrature rule with constant subinterval length in Figure 2b. Both the time-stepping model with K21 quadrature as well as the model with G21 quadrature (which has a higher order than K21) are significantly more efficient, hence showing that the learned adaptive time-stepping is actually advantageous. The model with G21 quadrature achieved an average error of about 6×10^{-8} using 80.27 function evaluations per time while a constant time-stepper needs 87.20 function evaluations per time to achieve the same average error. Thus, the adaptive time-stepping of our model leads to a reduction of required function evaluations of about 8% compared to subintervals of constant size.

By optimizing the weights of the G21 quadrature rule (as described in Section 3.2) the performance can be enhanced further, leading to a reduction of required function evaluations of about 8.5%.

4.2 Numerical solution of differential equations

In the following, we apply our framework to several dynamical systems which exhibit chaotic motions. In this context, classical adaptive schemes often suffer from step rejections when the estimated local errors become too big. This drives up the number of function evaluations.

In all following experiments the integration scheme used by our time-stepping model is the same 5-th order Runge-Kutta scheme that is employed by the conventional RK45 adaptive integrator, in order to enable a fair performance comparison. For RK45, we use the implementation in the Python package SciPy. We will show that our learned method obtains the required accuracy without the need of a step rejection mechanism and its performance exceeds RK45, even if function evaluations caused by rejected steps are not counted. This suggests that our approach learns to avoid the mistakes of traditional time-steppers.



(a) time-stepping

(b) Performance comparison

Figure 3: Lorenz System. (a) time-stepping obtained by the model. (b) Performance comparison between the model and `RK45`. For the time-stepping model (red) and the model with optimized weights (magenta) we plot the performance for an ensemble of 20 uniformly drawn initial conditions, integrated until $T = 100$ (small dots), and the ensemble average (big dot). The data for `RK45` was obtained by choosing different desired tolerances for the local error estimates and averaging the performance over the ensemble. The blue data points ("rej. not counted") show the performance if the function evaluations of step rejections of `RK45` are not counted.

4.2.1 Example: Lorenz system

A frequently used benchmark for ODE integrators is the chaotic Lorenz system, i. e.,

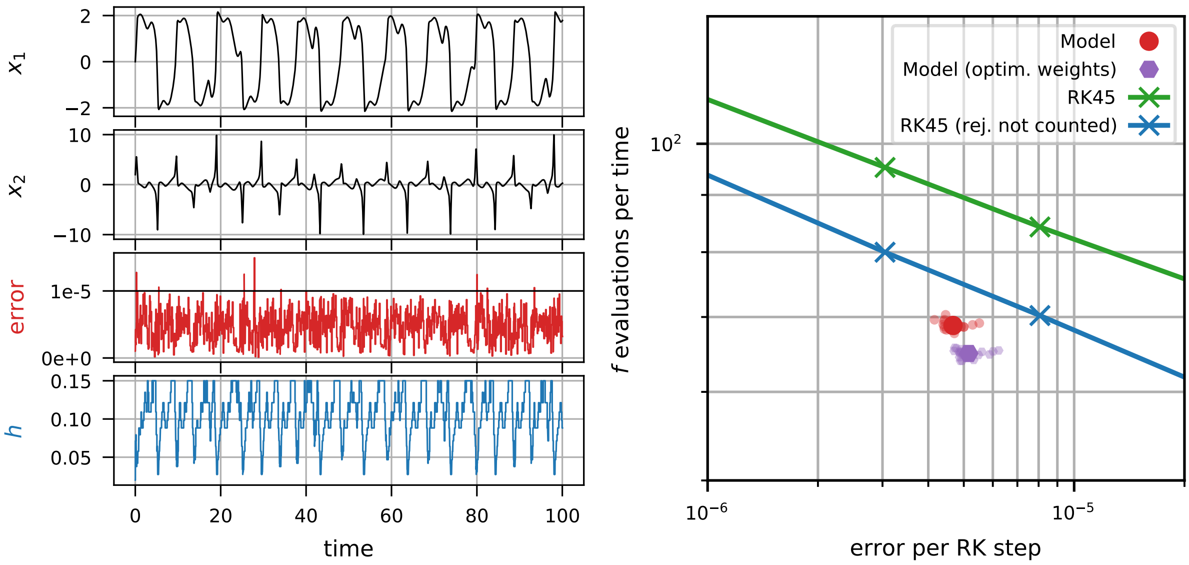
$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \sigma(x_2 - x_1) \\ x_1(\rho - x_3) - x_2 \\ x_1 x_2 - \beta x_3 \end{pmatrix},$$

with the standard parameter values $\sigma = 10$, $\beta = \frac{8}{3}$ and $\rho = 28$. In this regime, the system exhibits chaotic behavior. We train a model with step sizes

$$h \in \{0.02, 0.022, 0.025, 0.029, 0.033, 0.039, 0.045, 0.052, 0.060, 0.070\},$$

a tolerance of $\mathsf{tol} = 10^{-4}$, and the time horizon $[0, T] = [0, 100]$. For every training episode we sample a new initial condition $x(0)$ uniformly from the set $[-10, 10] \times [-10, 10] \times [15, 35]$, which contains a large portion of the attractor but also points away from the attractor.

The performance of the trained model is shown in Figure 3. We see in 3b that it significantly outperforms `RK45`. Using on average of 143.9 function evaluations per time unit, the model achieves an average error of about 4×10^{-5}, while `RK45` requires approximately 181.8 function evaluations to obtain the same average error. This corresponds to a reduction of the required function evaluations by approximately 21 % at the same level of accuracy. As can be seen in Figure 3b, the main reason for this substantial improvement is that `RK45` rejects poorly chosen time steps if the internal error estimation exceeds a certain bound. However, even without counting the rejections, the performance of our approach is slightly superior. By employing the technique of optimizing the weights of the used Runge-Kutta scheme (as described in 3.2), the performance can be enhanced further, so that the model can achieve the same accuracy as `RK45` with 23 % fewer function evaluations.



(a) time-stepping

(b) Performance comparison

Figure 4: Chaotic forced Van der Pol oscillator. (a) time-stepping obtained by the model. (b) Performance comparison between the model and `RK45`. For the time-stepping model (red) and the model with optimized weights (magenta) we plot the performance for an ensemble of 20 uniformly drawn initial conditions, integrated until $T = 100$ (small dots), and the ensemble average (big dot). The data for `RK45` was obtained by choosing different desired tolerances for the local error estimates and averaging the performance over the ensemble. The blue data points ("rej. not counted") show the performance if the function evaluations of step rejections of `RK45` are not counted.

4.2.2 Example: Forced Van der Pol oscillator

Next we study the Van der Pol oscillator with periodic forcing:

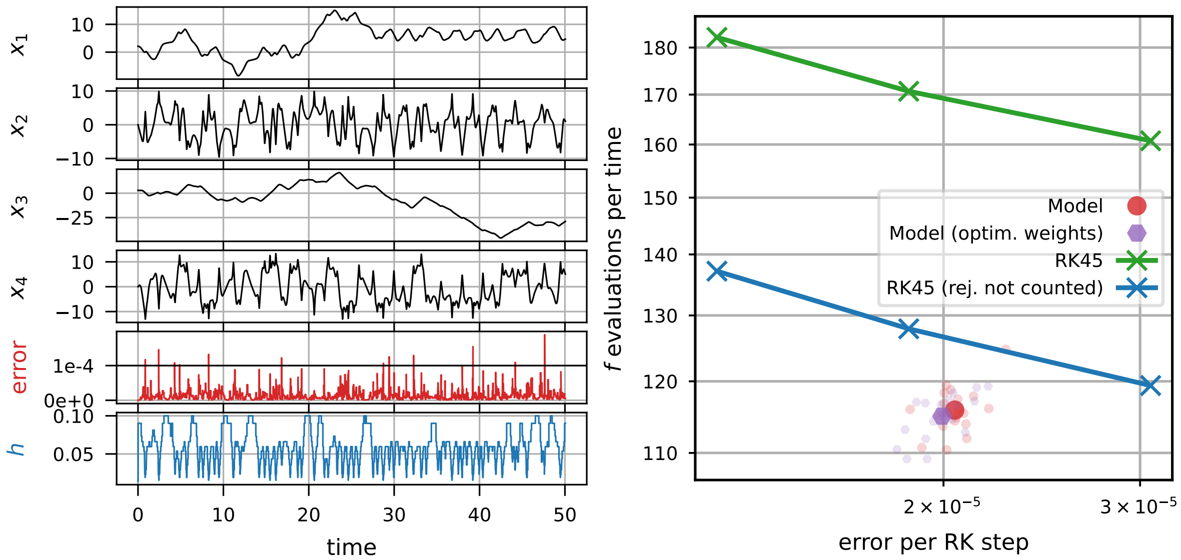
$$\frac{d^2x}{dt^2} - \mu(1-x^2)\frac{dx}{dt} + x = A\sin(\omega t)$$

For the parameters $\mu = 5$, $A = 5$ and $\omega = 2.465$, proposed by Tsatsos [27], the system exhibits chaotic behavior. We train a time-stepper with 20 step sizes evenly spaced on a log scale $h \in [0.02, 0.15]$ and $\mathsf{tol} = 10^{-5}$.

Figure 4a displays the time-stepping used by our model. Figure 4b compares the efficiency of our model with SciPy's `RK45`. While `RK45` suffers from step size rejections, which drives up the number of function evaluations, our model satisfies a given local error tolerance with fewer function evaluations. For an average error of about 5×10^{-6}, it is using on average 68.8 function evaluations per time unit, while `RK45` needs approximately 90.7 function evaluations for the same average error. Our model needs approximately 24 % less function evaluations than `RK45` with the same average error tolerance (28 % in case of the model with weight optimization enabled). Plotting the number of function evaluations of `RK45` without counting function evaluations of rejected steps, we see a comparable behaviour to our method. The experiment suggests that our model learns to satisfy a local error tolerance without the need for step size rejections. This leads to lower computational costs compared to classical time-stepping control mechanisms.

4.2.3 Example: Double Pendulum

Another well-known chaotic system is the double pendulum, i. e., a pendulum with a second pendulum attached to its end. The system is parametrized by the length and mass of each pendulum. (The limbs are modeled as massless and the masses of each pendulum as pointmasses at the end of the limbs.) We set the two lengths and two masses to 1, gravitational acceleration to 10, and consider motion in the two-dimensional vertical plane only (pendulums can move left and right). The dynamics of the double



(a) time-stepping

(b) Performance comparison

Figure 5: Double Pendulum. (a) time-stepping obtained by the model, $x = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]$ (b) Performance comparison between the model and `RK45`. For the time-stepping model (red) and the model with optimized weights (magenta) we plot the performance for an ensemble of 20 uniformly drawn initial conditions, integrated until $t_1 = 100$ (small dots), and the ensemble average (big dot). The data for `RK45` was obtained by choosing different desired tolerances for the local error estimates and averaging the performance over the ensemble. The blue data points ("rej. not counted") show the performance if the function evaluations of step rejections of `RK45` are not counted.

pendulum can then be characterized by an ODE of the state

$$x = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2],$$

where the θ_i refer to the angles of the pendulums and the $\dot{\theta}_i$ to the angular velocities [24].

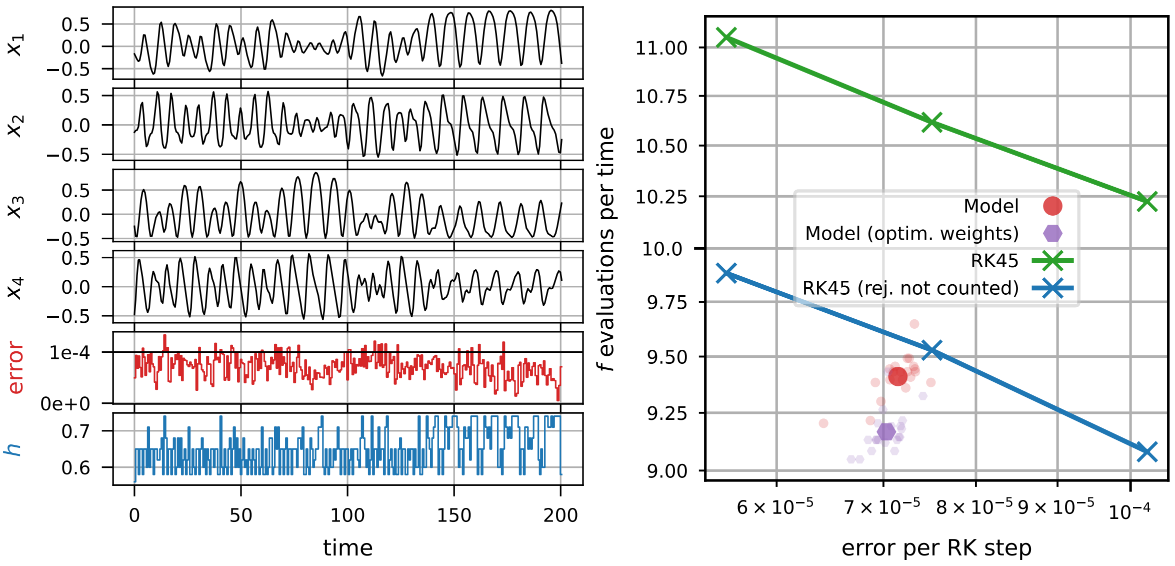
We train a time-stepping model with 20 step sizes evenly spaced on a log scale $h \in [0.014, 0.1]$ and $\mathsf{tol} = 10^{-4}$. The initial condition $x(0)$ is randomly drawn such that the resulting energy is always identical. We picked an energy level high enough for both pendulums to regularly flip.

The trained model is shown in Figure 5, and we see in 5b that it significantly outperforms `RK45`. Using on average 115.9 function evaluations per time unit, the model achieves an average error of about 2×10^{-5}, while `RK45` requires approximately 168.7 function evaluations to obtain the same average error. This corresponds to a reduction of the required function evaluations by approximately 31 % at the same level of accuracy. As can be seen in Figure 5b, the main reason for this substantial improvement is that `RK45` rejects poorly chosen time steps if the internal error estimation exceeds a certain bound. However, even without counting the rejections, the performance of our approach is slightly superior. By employing the technique of optimizing the weights of the used Runge-Kutta scheme (as described in 3.2), the performance can be enhanced slightly further, so that the model can achieve the same accuracy as `RK45` while using 32 % less function evaluations.

4.2.4 Example: Hénon-Heiles System

The Hénon-Heiles system is related to galactic dynamics and exhibits chaotic behavior. It is given by the ODE

$$\frac{d}{dt}\begin{pmatrix} x \\ p_x \\ y \\ p_y \end{pmatrix} = \begin{pmatrix} p_x \\ -x - 2xy \\ p_y \\ -y - (x^2 - y^2) \end{pmatrix},$$



(a) time-stepping

(b) Performance comparison

Figure 6: Henon-Heiles system. (a) time-stepping obtained by the model (b) Performance comparison between the model and RK45. For the time-stepping model (red) and the model with optimized weights (magenta) we plot the performance for an ensemble of 20 uniformly drawn initial conditions, integrated until $T = 500$ (small dots), and the ensemble average (big dot). The data for RK45 was obtained by choosing different desired tolerances for the local error estimates and averaging the performance over the ensemble. The blue data points ("rej. not counted") show the performance if the function evaluations of step rejections of RK45 are not counted.

where (x, y) refer to coordinates in two-dimensional space and (p_x, p_y) to the respective momenta. We train a time-stepping model with step sizes

$$h \in \{0.56, 0.58, 0.6, 0.62, 0.65, 0.68, 0.71, 0.74, 0.77, 0.8\}$$

and $\mathsf{tol} = 10^{-4}$ by integrating from $t = 0$ to $t = 500$. The initial positions $(x(0), y(0))$ are drawn uniformly from the triangle given by the contour-line of the Hénon-Heiles potential with potential energy $\frac{1}{6}$ and the momenta such that the resulting total energy is equal to $\frac{1}{6}$. As a consequence, the position $(x(t), y(t))$ is bounded within the above-mentioned triangle.

The trained model is shown in Figure 6, and we see in 6b that it significantly outperforms RK45. Using on average 9.41 function evaluations per time unit, the model achieves an average error of about 7×10^{-5}, while RK45 requires approximately 10.69 function evaluations to obtain the same average error. This corresponds to a reduction of the required function evaluations by approximately 7 % at the same level of accuracy. As can be seen in Figure 6b, the main reason for this substantial improvement is that RK45 rejects poorly chosen time steps if the internal error estimation exceeds a certain bound. However, even without counting the rejections, the performance of our approach is slightly superior. By employing the technique of optimizing the weights of the used Runge-Kutta scheme (as described in 3.2), the performance can be enhanced further, so that the model can achieve the same accuracy as RK45 but with 14 % fewer function evaluations.

5 Conclusion

Combining classical numerical integrators with learned adaptive time-stepping strategies yields new numerical schemes which efficiently perform quadrature tasks and integrate differential equations. These schemes outperform state-of-the-art numerical procedures on problem classes for which traditional adaptive schemes show inefficient behaviour, such as chaotic systems. This is achieved by tailoring the step size controller to the problem classes at hand and to the desired tolerances. Thus, our integration method combines the benefits of traditional numerical integration with data-driven optimization. Our strategy is especially useful in situations, where a time-stepper can be trained offline and all computational tasks that need to be performed online belong to a restricted class of problems.

Experiments on the optimal choice of not only the time-stepping but also the associated quadrature or Runge-Kutta weights suggest a great potential for further improvement. Next to improvements in local accuracy, tailored weights can lead to structure preserving properties of the numerical scheme, which benefit long-term simulations. For future work, it is of interest to develop a framework for a simultaneous selection of both time steps and quadrature weights and to design integrators that balance structure preservation, efficiency, and accuracy automatically. Finally, the approach of a data-driven time-stepping strategy could be extended to a data-driven creation of space-time grids for the numerical integration of partial differential equations.

Source code

Our source code is freely accessible on GitHub: https://github.com/lueckem/quadrature-ML.

Author Contribution Statement

Conceptualization and methodology: all authors.
Software: M. Lücke, K. Pfannenschmidt.
Writing: M. Lücke, C. Offen, S. Peitz, K. Pfannschmidt.

References

[1] Scipy v1.5.4 reference guide: Integration and ODEs (scipy.integrate.quad). https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html. Online; accessed 10 December 2020.

[2] P. Deuflhard and F. Bornemann. *Scientific computing with ordinary differential equations*, volume 42. Springer Science & Business Media, 2012.

[3] P. Deuflhard and A. Hohmann. *Numerical Analysis in Modern Scientific Computing.* Springer New York, 2003.

[4] J. Dormand and P. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980.

[5] J. Fan, Z. Wang, Y. Xie, and Z. Yang. A theoretical analysis of deep q-learning. In A. M. Bayen, A. Jadbabaie, G. Pappas, P. A. Parrilo, B. Recht, C. Tomlin, and M. Zeilinger, editors, *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, volume 120 of *Proceedings of Machine Learning Research*, pages 486–489, The Cloud, 10–11 Jun 2020. PMLR.

[6] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[7] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations.* Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 2013.

[8] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II.* Springer Berlin Heidelberg, 1996.

[9] E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I.* Springer Berlin Heidelberg, 1993.

[10] M. J. Kearns and S. P. Singh. Finite-sample convergence rates for q-learning and indirect algorithms. In *Advances in neural information processing systems*, pages 996–1002, 1999.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[12] S. Klus, F. Nüske, S. Peitz, J.-H. Niemann, C. Clementi, and C. Schütte. Data-driven approximation of the Koopman generator: Model reduction, system identification, and control. *Physica D: Nonlinear Phenomena*, 406:132416, 2020.

[13] M. Knöller, A. Ostermann, and K. Schratz. A fourier integrator for the cubic nonlinear schrödinger equation with rough initial data. *SIAM Journal on Numerical Analysis*, 57(4):1967–1986, 2019.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In *ICLR (Poster)*, 2016.

[15] Y. Liu, J. Kutz, and S. Brunton. Hierarchical deep learning of multiscale differential equation time-steppers. *ArXiv*, abs/2008.09768, 2020.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, feb 2015.

[17] R. Piessens, E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner. *Quadpack*. Springer Berlin Heidelberg, 1983.

[18] M. L. Piscopo, M. Spannowsky, and P. Waite. Solving differential equations with neural networks: Applications to the calculation of cosmological phase transitions. *Phys. Rev. D*, 100:016002, Jul 2019.

[19] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv:1711.10561*, 2017.

[20] F. Regazzoni, L. Dedè, and A. Quarteroni. Machine learning for fast and reliable solution of time-dependent differential equations. *Journal of Computational Physics*, 397:108852, 2019.

[21] SciPy. v1.5.4 reference guide: Integration and ODEs (scipy.integrate.solve_ivp). https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html. Online; accessed 4 December 2020.

[22] L. Shampine. Vectorized adaptive quadrature in MATLAB. *Journal of Computational and Applied Mathematics*, 211(2):131 – 140, 2008.

[23] L. F. Shampine and M. W. Reichelt. The MATLAB ODE suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, 1 1997.

[24] T. Shinbrot, C. Grebogi, J. Wisdom, and J. A. Yorke. Chaos in a double pendulum. *American Journal of Physics*, 60(6):491–499, 1992.

[25] J. Sirignano and K. Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339 – 1364, 2018.

[26] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[27] M. Tsatsos. *The Van Der Pol Equation*. PhD thesis.

[28] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[29] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[30] Wolfram Research Inc. Wolfram Language & System Documentation Center: The design of the NDSolve framework. https://reference.wolfram.com/language/tutorial/NDSolveDesign.html. Online; accessed 4 December 2020.