The Symplectic Adjoint Method: Memory-Efficient Backpropagation of Neural-Network-Based Differential Equations

Takashi Matsubara, *Member, IEEE*, Yuto Miyatake, and Takaharu Yaguchi, *Member, IEEE*

***Abstract*— The combination of neural networks and numerical integration can provide highly accurate models of continuous-time dynamical systems and probabilistic distributions. However, if a neural network is used n times during numerical integration, the whole computation graph can be considered as a network n times deeper than the original. The backpropagation algorithm consumes memory in proportion to the number of uses *times* of the network size, causing practical difficulties. This is true even if a checkpointing scheme divides the computation graph into subgraphs. Alternatively, the adjoint method obtains a gradient by a numerical integration backward in time; although this method consumes memory only for single-network use, the computational cost of suppressing numerical errors is high. The symplectic adjoint method proposed in this study, an adjoint method solved by a symplectic integrator, obtains the exact gradient (up to rounding error) with memory proportional to the number of uses *plus* the network size. The theoretical analysis shows that it consumes much less memory than the naive backpropagation algorithm and checkpointing schemes. The experiments verify the theory, and they also demonstrate that the symplectic adjoint method is faster than the adjoint method and is more robust to rounding errors.**

***Index Terms*— Adjoint method, backpropagation, checkpointing scheme, neural networks, ordinary differential equation (ODE), symplectic integrator.**

I. Introduction

ALTHOUGH neural networks are attracting attention as powerful tools for image recognition and natural-language processing [1], [2], their potential for learning dynamical systems is equally noteworthy. We emphasize that they also show promise for learning dynamical systems [3]. A feedforward neural network that has more than one layer and sufficiently many hidden units is known to approximate an arbitrary nonlinear function [4]. For a target system defined in discrete time, let a state u at the nth time step be denoted u_n, and the state transition be

$$u_{n+1} = \alpha(u_n, u_{n-1}, \ldots). \tag{1}$$

The neural network learns the state transition α. In particular, a set of present and past states $\{u_n, u_{n-1}, \ldots\}$ is given to the neural network as input, and the output of the neural network represents the next state u_{n+1} [5], [6], [7], [8], [9]. This approach is known as the nonlinear autoregressive with exogenous input (NARX) model, or more specifically, as the time-delay neural network (TDNN). When the target system is described by an ordinary differential equation (ODE)

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(t, u(t)) \tag{2}$$

one can sample the state u periodically and consider a discrete-time map, such as [8]

$$u(t + \Delta t) = u(t) + \Psi(t, u(t)) \tag{3}$$

where Δt denotes the step size. This is a special case of the TDNN in (1). However, a neural network often struggles with approximating the discrete-time map Ψ in (3). This is because Ψ is defined by the integral

$$\Psi(t, u(t)) = \int_t^{t+\Delta t} f(\xi, u(\xi))\mathrm{d}\xi \tag{4}$$

which is highly nonlinear even if the time-derivative f is not. To handle the nonlinearity, fractional order methods and fuzzy logic systems (FLSs) have been proposed to model dynamical systems [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. Several studies have used numerical integration techniques for learning an ODE [21], [22]. For example, the Runge–Kutta neural network (RKNN) has an internal feedforward neural network f_{NN} that approximates the time-derivative f and uses it four times per step according to the Runge–Kutta method [23]

$$\begin{aligned}
\Psi_{NN}(t, u(t)) &= \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4 \\
k_1 &= f_{NN}(t, u(t)) \\
k_2 &= f_{NN}\left(t + \frac{\Delta t}{2}, u(t) + \frac{\Delta t}{2}k_1\right) \\
k_3 &= f_{NN}\left(t + \frac{\Delta t}{2}, u(t) + \frac{\Delta t}{2}k_2\right) \\
k_4 &= f_{NN}(t + \Delta t, u(t) + \Delta t k_3).
\end{aligned} \tag{5}$$

Manuscript received 22 December 2021; revised 3 October 2022, 20 December 2022, and 9 January 2023; accepted 31 January 2023. This work was supported in part by the Japan Science and Technology Agency (JST) Core Research for Evolutionary Science and Technology (CREST) under Grant JPMJCR1914, in part by JST Precursory Research for Embryonic Science and Technology (PRESTO) under Grant JPMJPR21C7, and in part by the Japan Society for the Promotion of Science (JSPS) Grants-in-Aid for Scientific Research (KEKNHI) under Grant 19K20344. *(Corresponding author: Takashi Matsubara.)*

Takashi Matsubara is with the Graduate School of Engineering Science, Osaka University, Osaka 560-8531, Japan (e-mail: matsubara@sys.es.osaka-u.ac.jp).

Yuto Miyatake is with the Cybermedia Center, Osaka University, Osaka 560-8531, Japan (e-mail: miyatake@cas.cmc.osaka-u.ac.jp).

Takaharu Yaguchi is with the Graduate School of System Informatics, Kobe University, Hyogo 657-8501, Japan (e-mail: yaguchi@pearl.kobe-u.ac.jp).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2023.3242345.

Digital Object Identifier 10.1109/TNNLS.2023.3242345

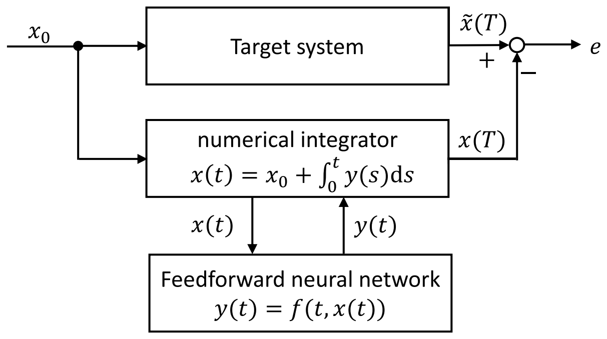


Fig. 1. Block diagram of the system identification.

The whole architecture Ψ_{NN} can be regarded as a neural network four times deeper than the internal neural network f_{NN} and approximates the discrete-time map Ψ. Several other studies also proposed neural networks inspired by numerical integrators [24], [25], [26], [27]. Their architectures are not feedforward, and the backpropagation algorithm should be modified manually [28].

Recent developments in automatic differentiation have made it possible to easily train neural networks that have special architectures [29], [30]. Following that, numerous studies have investigated new architectures for specific subclasses of continuous-time dynamical systems, including Hamiltonian systems on canonical coordinates [31], on arbitrary coordinates [32], and with constraints [33], [34], [35], Euler–Lagrange systems [36], and port-Hamiltonian systems [37]. These systems are associated with special geometric structures (e.g., symplectic structure), and the neural networks will be able to learn these systems with high accuracy if their architectures represent such structures well. ODEs with other conditions [38], [39] and ODEs obtained by spatially discretizing partial differential equations (PDEs) using grids [40] and meshes [41] have also been investigated.

The developments have also enabled to train neural networks that use numerical integrators with adaptive time-stepping, such as the Dormand–Prince method [42]. Using such integrators, neural networks can deal with continuous-time dynamics even when it is sampled irregularly (i.e., with variable time step sizes), whereas other methods, such as recurrent neural networks cannot (see [43] for comparison). In this article, we focus on explicit integrators, the internal states and outputs of which are defined by explicit functions and calculable forward in time. In the context of the system identification, we depict the block diagram in Fig. 1. We assume that the target system is a black box, and only the sampled data [denoted by $\tilde{x}(T)$] is available. Given an initial condition x_0, a numerical integrator solves the initial value problem with the time-derivative defined by a neural network, obtaining the terminal value $x(T)$ and the error e between the sampled data $\tilde{x}(T)$ and the terminal value $x(T)$. The backpropagation algorithm can be applied to obtain the gradients of the error e w.r.t. the parameters. Then, the parameters are updated to minimize the error e. This is a native approach to training neural networks for identifying the target system.

Memory consumption poses a difficulty for such approaches. The automatic differentiation algorithm obtains the gradient of a cost function $\mathcal{L}$ with respect to parameters θ by tracing the computation graph in the backward direction. The computation graph is composed of the operations and arguments that appear during the numerical integration. For example, the RKNN has four internal stages and consumes four times more memory than its internal neural network f_{NN} per step. Moreover, for accurate integration, an adaptive integrator may divide a given period Δt into multiple time steps. Let N, s, and L denote the number of time steps in a given period Δt, the number of internal stages, and the size of the neural network f_{NN}. Then, the memory consumption for naive backpropagation is $O(NsL)$, making a straightforward application troublesome. Assuming that the computational cost of backpropagation is the same as that of forward propagation, the total computational cost is $O(2NsL)$.

To reduce memory consumption, the neural ODE using the adjoint sensitivity method has been proposed [43]. This approach to solving an ODE obtains the gradient $\nabla\mathcal{L}$ of a cost function $\mathcal{L}$ by solving another ODE, the adjoint system [21], [45], [46], [47]. The adjoint method has been widely used to obtain the gradient with respect to the initial condition x_0 for data assimilation. In practice, it uses a numerical integrator and solves the adjoint system backward in time together with the main ODE forward in time. Neural ODE applies the adjoint method to obtaining the gradient with respect to the parameters for training the neural network f_{NN}. The adjoint method does not save the computation graph over time but calculates the time derivative of the gradient while consuming the memory only for the backpropagation of a single use of the neural network. Thus, the memory consumed is proportional only to L, not to N or s. Instead, the computational cost is increased by the additional backward integration of the state x. Moreover, the adjoint method suffers from numerical errors [44]. With a sufficiently small step size, it can partly (but never entirely) suppress these errors, but this leads to a larger number of time steps $\tilde{N}$ and a much higher computational cost $(N+2\tilde{N})sL$, where typically $\tilde{N} \gg N$. Therefore, there must be a tradeoff between memory consumption and computational cost.

Several existing methods employ checkpointing schemes [48], [49]. For example, ANODE and ACA retain each step $\{x_n\}_{n=0}^{N-1}$ as a checkpoint with a memory of $O(N)$ during the forward integration [44], [50]. To obtain the gradient, they recalculate the state x from each checkpoint x_n to the next step x_{n+1} and apply the backpropagation algorithm to the step composed of s stages; the memory consumption is $O(sL)$ for each step and $O(N+sL)$ in total. They can, thus, obtain a gradient-free from numerical errors and reduce the computational cost compared with the adjoint method. However, they still consume memory in proportion to both L and the number of internal stages s. The latter is not negligible for a high-order integrator: e.g., $s = 6$ in the Dormand–Prince method [42]. Table I summarizes previous studies.

Given the above-mentioned tradeoff, we propose the *symplectic adjoint method*. This is a special case of the adjoint method. Inspired by a recent study by Sanz-Serna [47], we design a numerical integrator so that the pair of numerical integrators for the adjoint method and main ODE is symplectic. A symplectic integrator conserves the symplectic structure, which defines the relationship between the time-derivative of the system state u and the gradient of the cost function $\mathcal{L}$. Thanks to this property, the symplectic

TABLE I
COMPARISON OF THE PROPOSED METHOD WITH EXISTING METHODS

Methods	Checkpoints	Memory Consumption		Computational Cost
		checkpoint	backprop.	
backpropagation	—	—	NsL	$2NsL$
adjoint method [43]	x_N	1	L	$(N+2\tilde{N})sL$
backpropagation & checkpoint per step [44]	$\{x_n\}_{n=0}^{N-1}$	N	sL	$3NsL$
symplectic adjoint method (proposed)	$\{x_n\}_{n=0}^{N-1}, \{X_{n,i}\}_{i=1}^{s}$	$N+s$	L	$4NsL$

L denotes number of layers in neural network; N and $\tilde{N}$ denote numbers of steps during forward and backward integrations, respectively; s denotes the number of uses of neural network f per step. Typically, $\tilde{N} > N$; otherwise, the adjoint method suffers from severe numerical errors.

adjoint method can obtain the exact gradient (up to rounding errors) of a neural ODE. The symplectic adjoint method uses the same step size for the forward and backward integrations (i.e., $\tilde{N} = N$), and hence, reduces the computational cost compared with the adjoint method. Each stage in the numerical integrator can be obtained explicitly, and the backpropagation can be performed for each stage separately. Combined with a nested checkpointing scheme, the symplectic adjoint method consumes memory in proportion only to L, not to N or s, up to the (often negligible) memory for the checkpoints. Due to the nested checkpointing scheme, the symplectic adjoint method requires some additional computation. Nevertheless, in practice, it works faster than the adjoint method and is competitive with checkpointing schemes.

We summarize the advantages of the proposed symplectic adjoint method as follows: **1) fast computation**—in discrete time, the adjoint method suffers from numerical errors or needs a smaller step size. The proposed method uses a specially designed integrator that obtains the exact gradient in discrete time. It works with the same step size as the forward integration and is, thus, faster than the adjoint method in practice. **2) Small memory consumption**—except for the adjoint method, existing methods apply the backpropagation algorithm to the computation graph of the whole numerical integration or a subset of it [44], [50], [51]. Memory consumption is proportional to the number of steps/stages in the graph *times* the network size. In contrast, the proposed method applies the algorithm only to each use of the neural network, and thus, the memory consumption is only proportional to the number of steps/stages *plus* the network size. **3) Robust to rounding error**—the backpropagation algorithm accumulates the gradient from each use of the neural network and tends to suffer from rounding errors. The proposed method obtains the gradient from each step as a numerical integration and is, thus, more robust to rounding errors. **4) Compatible with any ODE systems**—while this advantage applies to comparison methods, we emphasize that the proposed method is compatible with any systems described by ODEs [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [43].

Preliminary and limited results were presented in a conference paper [52], in which the symplectic adjoint method was evaluated mainly using computer-vision tasks and discussed in the context of modern deep learning. In this article, we have completely rewritten the introduction and evaluated the method in the context of dynamical systems.

II. Background Theory

A. Adjoint Method

Consider a system

$$\frac{\mathrm{d}}{\mathrm{d}t}x = f(t, x, \theta) \tag{6}$$

where x, t, and θ, respectively, denote the system state, an independent variable (e.g., time), and parameters of the function f. We assume that f is Lipschitz continuous. Then, given an initial condition $x(0) = x_0$, there exists a unique solution $x(t)$ given by [53]

$$x(t) = x_0 + \int_0^t f(\tau, x(\tau), \theta)\mathrm{d}\tau. \tag{7}$$

We assume that the solution $x(t)$ is evaluated at the terminal $t = T$ by a function $\mathcal{L}$ as $\mathcal{L}(x(T))$. Our main interest is in obtaining the gradients of $\mathcal{L}(x(T))$ with respect to the parameters θ. However, for simplicity, we first focus on the initial condition x_0, omitting θ.

Consider the solution $\bar{x}(t)$ arising from a perturbed initial condition $\bar{x}_0 = x_0 + \bar{\delta}_0$. Suppose that $\bar{x}(t) = x(t) + \bar{\delta}(t)$. Then

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\bar{\delta} &= \frac{\mathrm{d}}{\mathrm{d}t}(\bar{x} - x)\\ &= f(\bar{x}, t) - f(x, t)\\ &= \frac{\partial f}{\partial x}(x, t)(\bar{x} - x) + o(|\bar{x} - x|)\\ &= \frac{\partial f}{\partial x}(x, t)\bar{\delta} + o(|\bar{\delta}|)\\ \bar{\delta}(0) &= \bar{\delta}_0.\end{aligned} \tag{8}$$

Dividing each element of $\bar{\delta}$ by each element of $\bar{\delta}_0$ and taking the limit as $|\bar{\delta}_0| \to +0$, we define the *variational variable* δ and the *variational system* as

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t) = \frac{\partial f}{\partial x}(x(t), t)\delta(t) \text{ for } \delta(0) = I. \tag{9}$$

Therefore, the solution $\delta(t)$ of the variational system represents the variation of the solution $x(t)$ in the main system. Also, the variational variable $\delta(t)$ at time t represents the Jacobian $(\partial x(t)/\partial x_0)$ of the state $x(t)$ at time t with respect to the initial condition x_0.

We also define the *adjoint variable* λ and the *adjoint system* as

$$\frac{\mathrm{d}}{\mathrm{d}t}\lambda(t) = -\frac{\partial f}{\partial x}(x, t)^\top\lambda(t) \text{ for } \lambda(T) = \lambda_T. \tag{10}$$

Remark 1: The quantity $\lambda^\top \delta$ is time-invariant, i.e., $\lambda(t)^\top \delta(t) = \lambda(0)^\top \delta(0)$.

Proof

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}(\lambda^\top \delta) &= \left(\frac{\mathrm{d}}{\mathrm{d}t}\lambda\right)^\top \delta + \lambda^\top \left(\frac{\mathrm{d}}{\mathrm{d}t}\delta\right) \\ &= \left(-\frac{\partial f}{\partial x}(x,t)^\top \lambda\right)^\top \delta + \lambda^\top \left(\frac{\partial f}{\partial x}(x,t)\delta\right) \\ &= 0. \end{aligned} \tag{11}$$

Remark 2: The adjoint variable $\lambda(t)$ represents the gradient $(\partial \mathcal{L}(x(T))/\partial x(t))^\top$ if the final condition λ_T of λ is set to $(\partial \mathcal{L}(x(T))/\partial x(T))^\top$.

Proof: Using the time invariance of $\lambda^\top \delta$, the facts that $\delta(t) = (\partial x(t)/\partial x_0)$ and $\lambda(T) = (\partial \mathcal{L}(x(T))/\partial x(T))^\top$, and the chain rule, we find

$$\begin{aligned}\lambda(t)^\top \delta(t) &= \lambda(T)^\top \delta(T) \\ &= \frac{\partial \mathcal{L}(x(T))}{\partial x(T)} \frac{\partial x(T)}{\partial x_0} \\ &= \frac{\partial \mathcal{L}(x(T))}{\partial x_0} \\ &= \frac{\partial \mathcal{L}(x(T))}{\partial x(t)} \frac{\partial x(t)}{\partial x_0} \\ &= \frac{\partial \mathcal{L}(x(T))}{\partial x(t)} \delta(t). \end{aligned} \tag{12}$$

■

Thus, backward integration of the adjoint variable λ works as reverse-mode automatic differentiation. The adjoint method obtains the gradient by solving the adjoint system [21], [45], [46], [47]. (Note that, in proving Remark 2, we assumed that the variational variable δ is nonsingular; this assumption always holds if the function f is Lipschitz continuous and t is finite).

B. Adjoint Method for System Identification

The adjoint method has been used for data assimilation, where the initial condition x_0 is optimized by a gradient-based method. For system identification (i.e., parameter adjustment), one can consider the parameters θ as a part of the augmented state $\tilde{x} = [x \quad \theta]^\top$ of the system

$$\frac{\mathrm{d}}{\mathrm{d}t}\tilde{x} = \tilde{f}(\tilde{x}, t), \quad \tilde{f}(\tilde{x}, t) = \begin{bmatrix} f(t, x, \theta) \\ 0 \end{bmatrix}, \; \tilde{x}(0) = \begin{bmatrix} x_0 \\ \theta \end{bmatrix}. \tag{13}$$

For the augmented adjoint variable $\tilde{\lambda} = [\lambda \quad \lambda_\theta]^\top$, the augmented adjoint system is

$$\frac{\mathrm{d}}{\mathrm{d}t}\tilde{\lambda} = -\frac{\partial \tilde{f}}{\partial \tilde{x}}(\tilde{x}, t)^\top \tilde{\lambda} = -\begin{bmatrix} \frac{\partial f}{\partial x}^\top & 0 \\ \frac{\partial f}{\partial \theta}^\top & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ \lambda_\theta \end{bmatrix} = \begin{bmatrix} -\frac{\partial f}{\partial x}^\top \lambda \\ -\frac{\partial f}{\partial \theta}^\top \lambda \end{bmatrix}. \tag{14}$$

Hence, the adjoint variable λ for the system state x is unchanged from (10), the adjoint variable λ_θ for the parameters θ depends on λ as

$$\frac{\mathrm{d}}{\mathrm{d}t}\lambda_\theta = -\frac{\partial f}{\partial \theta}(x, t, \theta)^\top \lambda \tag{15}$$

and $\lambda_\theta(T) = (\partial \mathcal{L}(x(T), \theta)/\partial \theta)^\top$. The variational system is augmented in the same way. Hereafter, we let x denote the state or augmented state without loss of generality.

When the solution $x(t)$ is evaluated by a functional $\mathcal{C}$ as

$$\mathcal{C}(x(t)) = \int_0^T \mathcal{L}(x(t), t)\mathrm{d}t \tag{16}$$

the adjoint variable λ_C that denotes the gradient $\lambda_C(t) = (\partial \mathcal{C}(x(T))/\partial x(t))^\top$ of the functional $\mathcal{C}$ is given by

$$\frac{\mathrm{d}}{\mathrm{d}t}\lambda_C = -\frac{\partial f}{\partial x}(x,t)^\top \lambda_C + \left(\frac{\partial \mathcal{L}(x(t), t)}{\partial x(t)}\right)^\top, \quad \lambda_C(T) = \mathbf{0}. \tag{17}$$

Chen et al. [43] introduced the *neural ODE*, an ODE in which the time-derivative f is modeled by a neural network, and used the adjoint method for training. In the original implementation, the final value $x(T)$ of the system state x is retained after forward integration, and the pair (x, λ) is integrated backward in time to obtain the gradients. The right-hand sides of the main system in (6) and the adjoint system in (10) are obtained by the forward and backward propagations of the neural network f, respectively. Therefore, the computational cost of the adjoint method is twice that of the ordinary backpropagation algorithm.

After a numerical integrator discretizes the time, Remark 1 does not hold, and thus, the adjoint variable $\lambda(t)$ is not equal to the exact gradient [47], [50]. Moreover, in general, numerical integration backward in time is not consistent with that forward in time. Although a small step size (i.e., a small tolerance) suppresses numerical errors, it also leads to a longer computation time. These facts provide the motivation to obtain the exact gradient with small memory.

III. Symplectic Adjoint Method

A. Runge–Kutta Method

We first discretize the main system in (6). Let t_n, h_n, and x_n denote the nth time step, step size, and state, respectively, where $h_n = t_{n+1} - t_n$. Previous studies employed one of the Runge–Kutta methods, generally expressed as

$$\begin{aligned} x_{n+1} &= x_n + h_n \sum_{i=1}^{s} b_i k_{n,i} \\ k_{n,i} &:= f\left(X_{n,i}, t_n + c_i h_n\right) \\ X_{n,i} &:= x_n + h_n \sum_{j=1}^{s} a_{i,j} k_{n,j}. \end{aligned} \tag{18}$$

The coefficients $a_{i,j}$, b_i, and c_i are summarized as the Butcher tableau [21], [22], [47]. If $a_{i,j} = 0$ for $j \geq i$, the intermediate state $X_{n,i}$ is calculable from $i = 1$ to $i = s$ sequentially; the Runge–Kutta method is then considered explicit. Runge–Kutta methods are not time-reversible in general, i.e., the numerical integration backward in time is not consistent with that forward in time.

Remark 3: When the system in (6) is discretized by the Runge–Kutta method in (18), the variational system in (9) is consistent if it is discretized by the same Runge–Kutta method [22], [54].

Proof: By differentiating each term in the Runge–Kutta method for the main system in (18) with respect to the initial condition x_0

$$\delta_{n+1} = \delta_n + h_n \sum_{i=1}^{s} b_i d_{n,i}$$

$$\begin{aligned}d_{n,i} &= \frac{\partial k_{n,i}}{\partial x_0} = \frac{\partial f(X_{n,i}, t_n + c_i h_n)}{\partial x_0}\\ &= \frac{\partial f(X_{n,i}, t_n + c_i h_n)}{\partial X_{n,i}}\Delta_{n,i}\\ \Delta_{n,i} &:= \frac{\partial X_{n,i}}{\partial x_0} = \delta_n + h_n \sum_{j=1}^{s} a_{i,j} d_{n,j}.\end{aligned} \tag{19}$$

This is the variational system discretized by the same Runge–Kutta method. ■

B. Symplectic Runge–Kutta Method for Adjoint System

We assume $b_i \neq 0$ for $i = 1, \ldots, s$. We suppose the adjoint system to be solved by another Runge–Kutta method with the same step size as that used for the system state x, expressed as

$$\begin{aligned}\lambda_{n+1} &= \lambda_n + h_n \sum_{i=1}^{s} B_i l_{n,i}\\ l_{n,i} &:= -\frac{\partial f}{\partial x}(X_{n,i}, t_n + C_i h_n)^\top \Lambda_{n,i}\\ \Lambda_{n,i} &:= \lambda_n + h_n \sum_{j=1}^{s} A_{i,j} l_{n,j}.\end{aligned} \tag{20}$$

The final condition λ_N is set to $(\partial\mathcal{L}(x_N)\partial x_N)^\top$. Because the time evolutions of the variational variable δ and the adjoint variable λ are expressible by two equations, the combined system is considered a partitioned system. A combination of two Runge–Kutta methods for solving a partitioned system is called a partitioned Runge–Kutta method, where $C_i = c_i$ for $i = 1, \ldots, s$. We introduce the following condition for a partitioned Runge–Kutta method.

Condition 1: $b_i A_{i,j} + B_j a_{j,i} - b_i B_j = 0$ for $i, j = 1, \ldots, s$, and $B_i = b_i \neq 0$ and $C_i = c_i$ for $i = 1, \ldots, s$.

Theorem 1 ([47]): The partitioned Runge–Kutta method in (18) and (20) conserves a bilinear quantity $S(\delta, \lambda)$ if the continuous-time system conserves the quantity $S(\delta, \lambda)$ and Condition 1 holds.

Our proof is as follows.

Proof: Because the quantity S is conserved in continuous time

$$\frac{\mathrm{d}}{\mathrm{d}t} S(\delta, \lambda) = 0. \tag{21}$$

Because the quantity S is bilinear

$$\frac{\mathrm{d}}{\mathrm{d}t} S(\delta, \lambda) = \frac{\partial S}{\partial \delta}\frac{\mathrm{d}\delta}{\mathrm{d}t} + \frac{\partial S}{\partial \lambda}\frac{\mathrm{d}\lambda}{\mathrm{d}t} = S\left(\frac{\mathrm{d}\delta}{\mathrm{d}t}, \lambda\right) + S\left(\delta, \frac{\mathrm{d}\lambda}{\mathrm{d}t}\right) \tag{22}$$

which implies

$$S(d_{n,i}, \Lambda_{n,i}) + S(\Delta_{n,i}, l_{n,i}) = 0. \tag{23}$$

The change in the bilinear quantity $S(\delta, \lambda)$ is

$$\begin{aligned}&S(\delta_{n+1}, \lambda_{n+1}) - S(\delta_n, \lambda_n)\\ &\quad= S\left(\delta_n + h_n \sum_i b_i d_{n,i}, \lambda_n + h_n \sum_i B_i l_{n,i}\right) - S(\delta_n, \lambda_n)\\ &\quad= \sum_i b_i h_n S(d_{n,i}, \lambda_n) + \sum_i B_i h_n S(\delta_n, l_{n,i})\\ &\qquad+ \sum_i \sum_j b_i B_j h_n^2 S(d_{n,i}, l_{n,j})\\ &\quad= \sum_i b_i h_n S\left(d_{n,i}, \Lambda_{n,i} - h_n \sum_j A_{i,j} l_{n,j}\right)\\ &\qquad+ \sum_i B_i h_n S\left(\Delta_{n,i} - h_n \sum_j a_{i,j} d_{n,j}, l_{n,i}\right)\\ &\qquad+ \sum_i \sum_j b_i B_j h_n^2 S(d_{n,i}, l_{n,j})\\ &\quad= \sum_i h_n \left(b_i S(d_{n,i}, \Lambda_{n,i}) + B_i S(\Delta_{n,i}, l_{n,i})\right)\\ &\qquad+ \sum_i \sum_j (-b_i A_{i,j} - B_j a_{j,i} + b_i B_j) h_n^2 S(d_{n,i}, l_{n,j}).\end{aligned} \tag{24}$$

If $B_i = b_i$ and $b_i A_{i,j} + B_j a_{j,i} - b_i B_j = 0$, the change vanishes, i.e., the partitioned Runge–Kutta conserves the bilinear quantity S. Note that b_i must be nonzero, because $A_{i,j} = B_j(1 - a_{j,i}/b_i)$. Therefore, the bilinear quantity $\lambda_n^\top \delta_n$ is conserved as

$$\lambda_N^\top \delta_N = \lambda_n^\top \delta_n \text{ for } n = 0, \ldots, N. \tag{25}$$

■

Because the bilinear quantity S (including $\lambda^\top\delta$) is conserved, Remark 1 holds in discrete time. More precisely, because Remark 3 indicates $\delta_n = (\partial x_n/\partial x_0)$, and λ_N is set to $(\partial\mathcal{L}(x_N)/\partial x_N)^\top$,

$$\begin{aligned}\lambda_n^\top \delta_n &= \lambda_N^\top \delta_N\\ &= \frac{\partial\mathcal{L}(x_N)}{\partial x_N}\frac{\partial x_N}{\partial x_0}\\ &= \frac{\partial\mathcal{L}(x_N)}{\partial x_0}\\ &= \frac{\partial\mathcal{L}(x_N)}{\partial x_n}\frac{\partial x_n}{\partial x_0}\\ &= \frac{\partial\mathcal{L}(x_N)}{\partial x_n}\delta_n.\end{aligned} \tag{26}$$

Therefore, $\lambda_n = (\partial\mathcal{L}(x_N)/\partial x_n)^\top$; the adjoint system solved by the Runge–Kutta method in (20) under Condition 1 provides the exact gradient as the adjoint variable $\lambda_n = (\partial\mathcal{L}(x_N)/\partial x_n)^\top$.

The Dormand–Prince method, one of the most popular Runge–Kutta methods, has $b_2 = 0$ [42]. For such methods, the Runge–Kutta method under Condition 1 in (20) is generalized as

$$\begin{aligned}\lambda_{n+1} &= \lambda_n + h_n \sum_{i=1}^{s} \tilde{b}_i l_{n,i}\\ l_{n,i} &:= -\frac{\partial f}{\partial x}(X_{n,i}, t_n + c_i h_n)^\top \Lambda_{n,i}\\ \Lambda_{n,i} &:= \begin{cases}\lambda_n + h_n \sum_{j=1}^{s} \tilde{b}_j \left(1 - \frac{a_{j,i}}{b_i}\right) l_{n,j}, & \text{if } i \notin I_0\\ -\sum_{j=1}^{s} \tilde{b}_j a_{j,i} l_{n,j}, & \text{if } i \in I_0\end{cases}\end{aligned} \tag{27}$$

where

$$\tilde{b}_i = \begin{cases} b_i, & \text{if } i \notin I_0\\ h_n, & \text{if } i \in I_0\end{cases} \quad I_0 = \{i | i = 1, \ldots, s,\ b_i = 0\}. \tag{28}$$

Note that this numerical integrator is no longer a Runge–Kutta method and is an alternative expression for the "fancy" integrator proposed in [47].

Theorem 2 ([47]): The combination of the integrators in (18) and (27) conserves a bilinear quantity $S(\delta,\lambda)$ if the continuous-time system conserves the quantity $S(\delta,\lambda)$.

The original study did not provide detailed proof. We introduce a proof of the theorem as follows.

Proof: By solving the combination of the integrators in (18) and (27), the change in a bilinear quantity $S(\delta,\lambda)$ conserved by the continuous-time dynamics is found to be

$$\begin{aligned}
&S(\delta_{n+1},\lambda_{n+1})-S(\delta_n,\lambda_n)\\
&=S\left(\delta_n+h_n\sum_i b_i d_{n,i},\ \lambda_n+h_n\sum_i \tilde b_i l_{n,i}\right)-S(\delta_n,\lambda_n)\\
&=\sum_i b_i h_n S(d_{n,i},\lambda_n)+\sum_i \tilde b_i h_n S(\delta_n,l_{n,i})\\
&\quad+\sum_i\sum_j b_i\tilde b_j h_n^2 S(d_{n,i},l_{n,j})\\
&=\sum_{i\notin I_0} b_i h_n S\left(d_{n,i},\ \Lambda_{n,i}-h_n\sum_j \tilde b_j(1-a_{j,i}/b_i)l_{n,j}\right)\\
&\quad+\sum_i \tilde b_i h_n S\left(\Delta_{n,i}-h_n\sum_j a_{i,j}d_{n,j},\ l_{n,i}\right)\\
&\quad+\sum_{i\notin I_0}\sum_j b_i\tilde b_j h_n^2 S(d_{n,i},l_{n,j})\\
&=\sum_{i\notin I_0} b_i h_n\big(S(d_{n,i},\Lambda_{n,j})+S(\Delta_{n,i},l_{n,j})\big)\\
&\quad+\sum_{i\notin I_0}\sum_j\big(-b_i\tilde b_j(1-a_{j,i}/b_i)-\tilde b_j a_{j,i}+b_i\tilde b_j\big)\\
&\quad\times h_n^2 S(d_{n,i},l_{n,j})\\
&\quad+\sum_{i\in I_0}\left(\tilde b_i h_n S(\Delta_{n,i},l_{n,j})-\sum_j \tilde b_j a_{j,i}h_n^2 S(d_{n,i},l_{n,j})\right)\\
&=\sum_{i\notin I_0} b_i h_n\big(S(d_{n,i},\Lambda_{n,j})+S(\Delta_{n,i},l_{n,j})\big)\\
&\quad+\sum_{i\in I_0} h_n^2\big(S(d_{n,i},\Lambda_{n,j})+S(\Delta_{n,i},l_{n,j})\big)\\
&=0.
\end{aligned}\tag{29}$$

Hence, the bilinear quantity $S(\delta,\lambda)$ is conserved. ■

Remark 4: The Runge–Kutta method in (20) under condition 1 and the numerical integrator in (27) are explicit backward in time if the Runge–Kutta method in (18) is explicit forward in time.

Proof: Equation (20) can be rewritten as

$$\begin{aligned}
\lambda_n&=\lambda_{n+1}-h_n\sum_{i=1}^{s} b_i l_{n,i}\\
l_{n,i}&=-\frac{\partial f}{\partial x}(X_{n,i},t_n+c_ih_n)^\top\Lambda_{n,i}\\
\Lambda_{n,i}&=\lambda_{n+1}-h_n\sum_{j=1}^{s} b_j\frac{a_{j,i}}{b_i}l_{n,j}.
\end{aligned}\tag{30}$$

Equation (27) can be rewritten as

$$\begin{aligned}
\lambda_n&=\lambda_{n+1}-h_n\sum_{i=1}^{s}\tilde b_i l_{n,i}\\
l_{n,i}&=-\frac{\partial f}{\partial x}(X_{n,i},t_n+c_ih_n)^\top\Lambda_{n,i}\\
\Lambda_{n,i}&=\begin{cases}\lambda_{n+1}-h_n\sum_{j=1}^{s}\tilde b_j\frac{a_{j,i}}{b_i}l_{n,j}, & \text{if } i\notin I_0\\ -\sum_{j=1}^{s}\tilde b_j a_{j,i}l_{n,j}, & \text{if } i\in I_0.\end{cases}
\end{aligned}\tag{31}$$

Because $a_{i,j}=0$ for $j\geq i$ and $a_{j,i}=0$ for $j\leq i$. The intermediate adjoint variable $\Lambda_{n,i}$ is calculable from $i=s$ to $i=1$ sequentially, i.e., the integration backward in time is explicit. ■

We emphasize that Theorems 1 and 2 hold for any ODE systems, even if the systems have discontinuities [55], stochasticity [56], or physics constraints [31]. This is because the theorems do not depend on the properties of a particular system but of Runge–Kutta methods in general.

A partitioned Runge–Kutta method that satisfies condition 1 is symplectic [21], [22]. It is known that a symplectic integrator applied to a Hamiltonian system using a fixed step size conserves a modified Hamiltonian that approximates the system energy. The bilinear quantity S is associated with the symplectic structure but not with a Hamiltonian. Regardless of the step size, a symplectic integrator conserves the symplectic structure, and thereby, conserves the bilinear quantity S. Hence, we call this method the *symplectic adjoint method*. For integrators other than Runge–Kutta methods, one can design the integrator for the adjoint system so that the pair of integrators is symplectic (see [57] for example).

All the above-mentioned considerations about the adjoint method and symplectic adjoint method hold also for implicit Runge–Kutta methods, in which one or more of the internal stages and outputs are defined by implicit functions and calculated by a root-finding algorithm so that the computational cost is not easily calculable. In contrast, the naive backpropagation algorithm and ACA are not applicable to implicit functions because they assume explicit ones. (For the backpropagation algorithm for implicit functions, see [58]).

Algorithm 1 Forward Integration

```
Input: $x_0$
Output: $x_N$,$\{x_n\}_{n=0}^{N-1}$
1: for $n = 0$ to $N-1$ do
2:     Retain $x_n$ as a checkpoint
       According to (18)
3:     for $i = 1$ to $s$ do
4:         Get $X_{n,i}$ using $x_n$ and $k_{n,j}$ for $j < i$
5:         Get $k_{n,i}$ using $X_{n,i}$
6:     end for
7:     Get $x_{n+1}$ using $x_n$ and $k_{n,i}$
8: end for
```

C. Implementation of Symplectic Adjoint Method

The theory given in Section III-B was largely introduced into numerical analysis in [47]. Because the original

Algorithm 2 Backward Integration

```
Input: $x_N, \{x_n\}_{n=0}^{N-1}$
Output: $\lambda_0$
 1: for $n = N-1$ to 0 do
 2:     Load checkpoint $x_n$
        According to (18)
 3:     for $i = 1$ to $s$ do
 4:         Get $X_{n,i}$ using $x_n$ and $k_{n,j}$ for $j < i$
 5:         Get $k_{n,i}$ using $X_{n,i}$.
 6:         Retain $X_{n,i}$ as a checkpoint
 7:     end for
        According to (27)
 8:     for $i = s$ to 1 do
 9:         Get $\Lambda_{n,i}$ using $\lambda_{n+1}$ and $l_{n,j}$ for $j > i$
10:         Load checkpoint $X_{n,i}$
11:         Get $l_{n,i}$ using $\Lambda_{n,i}$ and $X_{n,i}$.
12:         Discard checkpoint $X_{n,i}$
13:     end for
14:     Get $\lambda_n$ using $\lambda_{n+1}$ and $l_{n,i}$
15:     Discard checkpoint $x_n$
16: end for
```

expression included recalculations of intermediate variables, we propose the alternative expression in (27) to reduce the computational cost. The discretized adjoint system in (27) depends on the vector-Jacobian product (VJP) $\Lambda^\top(\partial f/\partial x)$. To obtain it, the computation graph from the input $X_{n,i}$ to the output $f(X_{n,i}, t_n + c_i h_n)$ is required. When the computation graph in the forward integration is entirely retained, the memory consumption and computational cost are of the same order as those for the naive backpropagation algorithm. To reduce memory consumption, we propose the strategy summarized in the following and in Algorithms 1 and 2.

At the forward integration of a neural ODE component, the pairs of system states x_n and time points t_n at time steps $n = 0, \ldots, N-1$ are retained with a memory of $O(N)$ as checkpoints, and all computation graphs are discarded, as shown in Algorithm 1. The backward integration is summarized in Algorithm 2. From the checkpoint x_n, the intermediate states $X_{n,i}$ for s stages are obtained following the Runge–Kutta method in (18) and retained as checkpoints with a memory of $O(s)$, while all computation graphs are discarded. Then, the adjoint system is integrated from $n+1$ to n using (27). This routine is repeated as n goes from $N-1$ to 0.

Because the computation graph of the neural network f in line 5 was discarded, it is recalculated, and the VJP $\lambda^\top(\partial f/\partial x)$ is obtained using the backpropagation algorithm one-by-one in line 11, where only a single use of the neural network is recalculated at a time. This is why the memory consumption is proportional to the number of checkpoints $N+s$ *plus* the neural network size L, as summarized in Table I. In contrast, existing methods apply the backpropagation algorithm to the computation graph of a single step composed of s stages or multiple steps. The memory consumption is proportional to the number of uses of the neural network between two checkpoints (at least s) *times* the neural network size L, in addition to the memory for checkpoints. Due to the recalculation, the computational cost of the proposed strategy is $O(4NsL)$, whereas those of the adjoint method [43] and ACA [44] are $O((N+2\tilde{N})sL)$ and $O(3NsL)$, respectively. However, the increase in the computation time is much less than that expected theoretically because of other bottlenecks (as demonstrated later).

Other implementation strategies are imaginable, such as ones that retain no checkpoints or checkpoints for all stages. Therefore, we call the above-mentioned implementation strategy the *nested checkpointing scheme*. Unless otherwise stated, we refer to the symplectic adjoint method with the nested checkpointing scheme simply as the symplectic adjoint method hereafter.

IV. Experiments

We evaluated the performance of the proposed symplectic adjoint method and existing methods using PyTorch 1.7.1 [30]. We implemented the symplectic adjoint method by extending the adjoint method implemented in the package torchdiffeq 0.1.1 [43]. For the adjoint method, we used the same numerical integrator as that for solving the main system unless otherwise stated, following previous works [43], [44]. We reimplemented ACA [44] because the interfaces of the official implementation were incompatible with torchdiffeq. The source code is available at https://github.com/tksmatsubara/symplectic-adjoint-method.

A. Continuous-Time Dynamical System

1) Experimental Settings: We evaluated the symplectic adjoint method on learning continuous-time dynamical systems [31], [40], [59]. We followed the experimental settings of HNN++, provided in [40],[1] unless, otherwise, stated.

Many physical phenomena can be modeled using $\mathrm{d}x/\mathrm{d}t = G\nabla H(x)$, where H is system energy and G is a coefficient matrix that determines the behaviors of the energy [60]. Following the release code[1], we chose physical systems described by PDEs, namely, the Korteweg–De Vries (KdV) equation and the Cahn–Hilliard equation. The KdV equation conserves the system energy and has soliton solutions [61], [62]. The energy function H and the time evolution of the KdV equation are expressed as

$$H(u) = \int_x \frac{1}{6}\alpha u^3 + \frac{1}{2}\beta\left(\frac{\partial}{\partial x}u\right)^2$$
$$\frac{\partial}{\partial t}u = \frac{\partial}{\partial x}\nabla H(u). \tag{32}$$

The coefficients α and β determine the spatiotemporal scales. The Cahn–Hilliard equation, which is derived from free-energy minimization, dissipates the system energy and describes, e.g., the phase separation of copolymer melts [61], [62]. The energy function H and the time evolution of the Cahn–Hilliard equation are expressed as

$$H(u) = \int_x \frac{1}{2}(u^2-1)^2 + \gamma\frac{1}{2}\left(\frac{\partial}{\partial x}u\right)^2$$
$$\frac{\partial}{\partial t}u = \frac{\partial^2}{\partial x^2}\nabla H(u) \tag{33}$$

where the coefficient $\gamma > 0$ denotes the mobility of the monomers. The mass u has an unstable equilibrium at $u = 0$ (totally melted) and stable equilibria at $u = -1$ and $u = 1$ (totally separated). We set the spatial mesh size to 50 and the

[1]https://github.com/tksmatsubara/discrete-autograd

TABLE II
RESULTS OBTAINED FOR CONTINUOUS-TIME 1-D PHYSICAL SYSTEMS

	KdV Equation			Cahn–Hilliard Equation		
	MSE ($\times 10^{-3}$)	mem.	time	MSE ($\times 10^{-6}$)	mem.	time
adjoint method [43]	1.61±3.23	**181.4**± 0.0	240±16	5.58±2.12	**181.4**± 0.0	805±25
backpropagation [43]	1.61±3.24	733.9±15.6	**94**± 4	5.45±1.55	3053.5±22.9	**382**±11
ACA [44]	1.61±3.24	734.5±20.3	120± 4	6.00±3.27	780.4±22.9	422±16
proposed	1.61±3.58	182.1± 0.0	141± 7	5.48±1.90	182.1± 0.0	480±19

Mean-squared errors (MSEs) in long-term predictions, peak memory consumption [MiB], and computation time per iteration [ms/itr].

number of periods in a time series to 500 and employed the periodic boundary condition. We generated 90 time–series for training and ten time–series for testing. All experiments were done with double precision.

We implemented the energy function H using a neural network composed of a convolution layer with a kernel size of 3 preceding two fully connected layers with 200 hidden units, and the differential operators $\partial/\partial x$ and $\partial^2/\partial x^2$ as the first- and second-order central difference operators.

The release code uses the option TORCH.BACKENDS.CUDNN.DETERMINISTIC to avoid the nondeterministic behavior and to improve reproducibility. In our experiments, we disabled this option to accelerate the simulations and to compare the results under a more practical situation. We used a batch-size of 100 (instead of the original batch-size of 200) to put a mini-batch into a single NVIDIA TITAN V GPU; when using multiple GPUs, bottlenecks, such as data transfer across GPUs, may affect performance, and a fair comparison becomes difficult. Moreover, we used the eighth-order Dormand–Prince method [21], which is a Runge–Kutta method with adaptive time-stepping, composed of 13 stages. (The number of function evaluations per step was $s = 12$ because the last stage was reused as the first stage of the next step). We set the absolute and relative tolerances to $\mathsf{atol} = 10^{-9}$ and $\mathsf{rtol} = 10^{-7}$, respectively. We evaluated the performance using mean squared errors (MSEs) in the system energy for long-term predictions.

2) Performance: The medians ± standard deviations of 15 runs are summarized in Table II. Due to the accumulated error in the numerical integration, the MSEs had large variances. All methods obtained similar MSEs because all methods obtained the gradients exactly up to a rounding error (except in the case of the adjoint method with a small tolerance, which was nevertheless sufficiently accurate).

We found that, in most cases, the Dormand–Prince method did not divide a given period (i.e., $N = 1$) for the KdV equation; it divided a given period into around five-time steps for the Cahn–Hilliard equation. We obtained the peak memory consumption during additional training iterations (mem. [MiB]), from which we subtracted the memory consumption before training (i.e., memory occupied by the model parameters, loaded data, and so on). The memory consumption still included the optimizer's states and the intermediate results of the multiply–accumulate operation. Because of these bottlenecks, the results agree only approximately with the theoretical orders-of-magnitude in Table I. The symplectic adjoint method consumed much less memory than the naive backpropagation algorithm and ACA.

The computation time per iteration (time [ms/itr]) is also consistent with theoretical values shown in Table I. For obtaining the gradients, the adjoint method integrates the adjoint variable λ, the size of which is equal to the sum of the sizes of the parameters θ and the system state x. With more parameters, the probability that at least one parameter does not satisfy the tolerance value is increased. Accurate backward integration requires a much smaller step size than forward integration (i.e., $\tilde{N}$ is much greater than N), leading to a longer computation time.

In conclusion, the symplectic adjoint method consumes memory to the same extent as the adjoint method but is much faster.

B. Continuous-Time 2-D Dynamical System

1) Experimental Settings: We extended the experimental settings of HNN++, provided in [40], to enable physics simulations of 2-D PDE systems. In this case, the Cahn–Hilliard equation is expressed as

$$H(u) = \int_{x,y} \frac{1}{2}(u^2-1)^2 + \gamma\frac{1}{2}\left(\left(\frac{\partial}{\partial x}u\right)^2 + \left(\frac{\partial}{\partial y}u\right)^2\right)$$
$$\frac{\partial}{\partial t}u = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\nabla H(u). \tag{34}$$

In addition, we used the Allen–Cahn equation, which also expresses the phase separation of copolymer melts but does not conserve the mass

$$\frac{\partial}{\partial t}u = -\nabla H(u) \tag{35}$$

and we used the same energy function H as in the Cahn–Hilliard equation. We employed the fifth-order Dormand–Prince method [42], composed of seven stages ($s = 6$). For adopting the 2-D PDE systems, we used 2-D convolution layers. We used a batch-size of 1 for a single NVIDIA TITAN V GPU. The remaining experimental settings were the same as those for the 1-D case.

2) Performance: The medians ± standard deviations of 15 runs are summarized in Table III. (One out of 15 trials of the adjoint method did not converge in a reasonable time, so we considered only the remaining 14 results). Compared with the 1-D case, we used a lower order numerical integrator, and the elements in the state u increased from 50 to 2500. Hence, the numerical integrator divided a period Δt into more time steps (around 20 for the Cahn–Hilliard equation and around three for the Allen–Cahn equation). Although it is not a common occurrence, the adjoint method suffers from nonnegligible numerical errors in the backward integration, and in this case, its training did not converge; smaller absolute and relative tolerances were needed. Nonetheless, the results were consistent with the theoretical ones in Table I. The

TABLE III
RESULTS OBTAINED FOR CONTINUOUS-TIME 2-D PHYSICAL SYSTEMS

	Cahn–Hilliard Equation			Allen–Cahn Equation		
	MSE ($\times 10^{-3}$)	mem.	time	MSE ($\times 10^{-3}$)	mem.	time
adjoint method [43]*	2.20±1.46	48.7± 0.0	1234±66	4.34±19.37	48.7±0.0	144±19
backpropagation [43]	3.56±6.75	4364.4±299.6	**372**±21	1.81±17.07	457.6±0.0	**48**± 5
ACA [44]	3.64±9.22	169.6± 0.0	407±25	5.46±47.83	168.9±0.0	62± 3
proposed	2.71±9.78	**42.4**± 0.0	476±30	1.95±27.81	**41.8**±0.0	71± 8

Mean-squared errors (MSEs) in long-term predictions, peak memory consumption [MiB], and computation time per iteration [ms/itr]. *One out of 15 trials did not converge in a reasonable time, so we summarized the results of the remaining 14 trials.

symplectic adjoint method obtained slightly better MSEs, but because of the high variances, these improvements were not significant. It consumed less memory than the adjoint method. Following (30) and (31), a naive implementation of the adjoint method retains the adjoint variables $\Lambda_{n,i}$ at all stages $i = 1, \ldots, s$ to obtain their time-derivatives $l_{n,i}$, and then adds them up to obtain the adjoint variable λ_n at the nth time step. However, as (15) shows, the adjoint variable λ_θ for the parameters θ is not used in obtaining its time-derivative $(d/dt)\lambda_\theta$. One can add up the adjoint variable $\Lambda_{\theta n,i}$ for the parameters θ at stage i one-by-one without retaining it, thereby reducing the memory consumption in proportion to the number of parameters *times* the number of stages. A similar optimization is applicable to the adjoint method.

C. Continuous Normalizing Flow

1) Experimental Settings: We evaluated the proposed symplectic adjoint method on training continuous normalizing flows, which learn probabilistic distributions [63]. A normalizing flow is a neural network that approximates a bijective map g and obtains the exact likelihood of a sample u by the change of variables $\log p(u) = \log p(z) + \log|\det(\partial g(u)/\partial u)|$, where $z = g(u)$ and $p(z)$ denote the corresponding latent variable and its prior, respectively [64], [65], [66]. In the case of continuous normalizing flow, the map g is modeled by stacked neural ODE components, in particular, $u = x(0)$ and $z = x(T)$. The log-determinant of the Jacobian was obtained by a numerical integration together with the system state x as $\log|\det(\partial g(u)/\partial u)| = -\int_0^T \mathrm{Tr}(\partial f/\partial x(x(t), t)\mathrm{d}t$. The trace operation Tr was approximated by the Hutchinson estimator [67]. We adopted the experimental settings of the continuous normalizing flow FFJORD[2] [63], unless stated, otherwise.

We examined five real-tabular datasets: MINIBOONE, GAS, POWER, HEPMASS, and BSDS300 [68]. All experiments were done with single precision. The network architectures were the same as those that achieved the best results in the original experiments. All architectures except that for the MINIBOONE dataset were composed of multiple neural ODE components of different sizes, and each result was obtained as the total usage of all components. We employed the fifth-order Dormand–Prince integrator [42]. We set the absolute and relative tolerances to $\mathsf{atol} = 10^{-8}$ and $\mathsf{rtol} = 10^{-6}$, respectively. The neural networks were trained using the Adam optimizer [69] with a learning rate of 10^{-3}. We used a batch-size of 1000 for all datasets to put a mini-batch into a single NVIDIA GeForce RTX 2080Ti GPU with 11 GB of memory, whereas the original experiments employed a batch-size of 10 000 for the latter three datasets on multiple GPUs. Nonetheless, the naive backpropagation algorithm consumed the entire memory in the case of the BSDS300 dataset.

We also examined the MNIST dataset [70] using a single NVIDIA RTX A6000 GPU with 48 GB of memory. Following the original study, we employed a multiscale architecture and set the tolerance to $\mathsf{atol} = \mathsf{rtol} = 10^{-5}$. We set the learning rate to 10^{-3} and then reduced it to 10^{-4} at the 250th epoch. While the original experiments used a batch-size of 900, we set the batch-size to 200 following the official code[2]. The naive backpropagation algorithm consumed the entire memory.

2) Performance: The medians ± standard deviations of three runs are summarized in Table IV. As expected, there were no significant differences among the negative log-likelihoods (NLLs) for most methods. The naive backpropagation algorithm obtained slightly worse results on the GAS, POWER, and HEPMASS datasets. Because of the repeated use of the neural network, each method accumulated the gradient of the parameters θ for each use. Let $\theta_{n,i}$ denote the parameters used in the ith stage of the nth step even though their values are unchanged. The backpropagation algorithm obtains the gradient $(\partial\mathcal{L}/\partial\theta)$ with respect to the parameters θ by accumulating the gradient over all stages and steps one-by-one

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{\substack{n=0,\ldots,N-1,\\ i=1,\ldots,s}} \frac{\partial \mathcal{L}}{\partial \theta_{n,i}}. \tag{36}$$

When the step size h_n at the nth step is sufficiently small, the gradient $(\partial\mathcal{L}/\partial\theta_{n,i})$ at the ith stage may be insignificant compared with the accumulated gradient and be rounded off during the accumulation. In contrast, ACA accumulates the gradient within a step and then over time steps; this process can be expressed informally as

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{n=0}^{N-1}\left(\sum_{i=1}^{s} \frac{\partial \mathcal{L}}{\partial \theta_{n,i}}\right). \tag{37}$$

Furthermore, according to (20) and (15), the (symplectic) adjoint method accumulates the adjoint variable λ (i.e., the transpose of the gradient) within a step and then over time steps

$$\lambda_{\theta,n} = \lambda_{\theta,n+1} - h_n \times \sum_{i=1}^{s} B_i\left(-\frac{\partial f}{\partial \theta_{n,i}}(X_{n,i}, t+C_i h_n, \theta_{n,i})^\top \Lambda_{n,i}\right). \tag{38}$$

[2]https://github.com/rtqichen/ffjord

TABLE IV
RESULTS OBTAINED FOR CONTINUOUS NORMALIZING FLOWS

	MINIBOONE			GAS			POWER		
	NLL	mem.	time	NLL	mem.	time	NLL	mem.	time
adjoint method [43]	10.59±0.17	170± 0	0.74±0.04	-10.53±0.25	24± 0	4.82±0.29	-0.31±0.01	**8.1**± 0.0	6.33±0.18
backpropagation [43]	10.54±0.18	4,436±115	0.91±0.05	-9.53±0.42	4,479±250	12.00±0.93	-0.24±0.05	1710.9±193.1	10.64±2.73
ACA [44]	10.57±0.30	306± 0	0.77±0.02	-10.65±0.45	73± 0	3.98±0.14	-0.31±0.02	29.5± 0.5	5.08±0.88
proposed	10.49±0.11	**95**± 0	0.84±0.03	-10.89±0.11	**20**± 0	4.39±0.23	-0.31±0.02	9.2± 0.0	5.73±0.43

	HEPMASS			BSDS300			MNIST		
	NLL	mem.	time	NLL	mem.	time	NLL	mem.	time
adjoint method [43]	16.49±0.25	40± 0	4.19±0.15	-152.04±0.09	577±0	11.70±0.44	0.918±0.011	1,086±4	10.12±0.88
backpropagation [43]	17.03±0.22	5,254±137	11.82±1.33	—	—	—	—	—	—
ACA [44]	16.41±0.39	88± 0	3.67±0.12	-151.27±0.47	757±1	6.97±0.25	0.919±0.003	4,332±1	7.94±0.63
proposed	16.48±0.20	**35**± 0	4.15±0.13	-151.17±0.15	**283**±2	8.07±0.72	0.917±0.002	**1,079**±1	9.42±0.32

Medians ± standard deviations of three runs for negative log-likelihood (NLL), peak memory consumption [MiB], and computation time per iteration [s/itr].

In these cases, even when the step size h_n at the nth step is small, the gradient summed within a step (over s stages) may still be significant and robust to rounding errors. This is the reason the adjoint method, ACA, and the symplectic adjoint method performed better than the naive backpropagation algorithm for some datasets. Note that this approach requires additional memory consumption to store the gradient summed within a step, and it is applicable to the backpropagation algorithm with a slight modification.

We obtained the peak memory consumption (mem. [MiB]), from which we subtracted the memory consumption before training. The results roughly agreed with the theoretical values shown in Table I. The symplectic adjoint method consumed much less memory than the naive backpropagation algorithm and ACA; it also consumed less memory than the adjoint method in some cases, such as the case for the 2-D dynamical systems.

On the other hand, the computation time per iteration (time [s/itr]) did not agree with the theory; the naive backpropagation algorithm was slower than that expected in many cases. A method with high memory consumption may have to wait for a retained computation graph to be loaded or for memory to be freed, leading to an additional bottleneck. As with the dynamical systems, the adjoint method was slower in many cases, especially for the BSDS300 and MNIST datasets. The symplectic adjoint method was free from the above bottlenecks and had faster performance in practice; it was faster than the adjoint method for all datasets but MINIBOONE.

Thus, the symplectic adjoint method is superior to (or at least competitive with) the adjoint method and naive backpropagation in terms of both memory consumption and computation time. The proposed symplectic adjoint method and ACA have a tradeoff between memory consumption and computation time.

D. Detailed Investigations

We further evaluated the proposed symplectic adjoint method on training continuous normalizing flows with different settings.

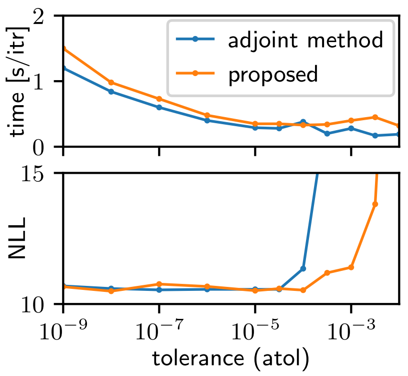


Fig. 2. Computation time per iteration and NLL for MINIBOONE dataset with different tolerances.

1) Robustness to Tolerance: The adjoint method provides gradients with numerical errors. To evaluate the robustness against tolerance, we employed the MINIBOONE dataset and varied the absolute tolerance atol while maintaining the relative tolerance at $\mathsf{rtol} = 10^2 \times \mathsf{atol}$. The computation time per iteration during training is summarized in the upper panel of Fig. 2: it fell as the tolerance increased. After training, we obtained the NLLs with $\mathsf{atol} = 10^{-8}$, as summarized in the bottom panel of Fig. 2. The adjoint method performed well only with $\mathsf{atol} < 10^{-4}$. With $\mathsf{atol} = 10^{-4}$, the numerical error in the backward integration was nonnegligible, and the performance degraded. With $\mathsf{atol} > 10^{-4}$, the adjoint method destabilized. The symplectic adjoint method performed well even with $\mathsf{atol} = 10^{-4}$. With $10^{-4} < \mathsf{atol} < 10^{-2}$, it performed to a certain level, while the numerical error in the forward integration was nonnegligible. Because of the exact gradient, the symplectic adjoint method is more robust to a large tolerance than the adjoint method, and thus, is potentially faster with an appropriate tolerance.

2) Different Runge–Kutta Methods: The Runge–Kutta family includes various integrators characterized by their Butcher tableaux [21], [22], [47], such as the Heun–Euler (adaptive Heun), Bogack Shampine (bosh3), fifth-order Dormand–Prince (dopri5), and eighth-order Dormand–Prince

TABLE V
RESULTS OBTAINED FOR GAS DATASET WITH DIFFERENT RUNGE–KUTTA METHODS

	$p = 2,\ s = 2$		$p = 3,\ s = 3$		$p = 5,\ s = 6$		$p = 8,\ s = 12$	
	mem.	time	mem.	time	mem.	time	mem.	time
adjoint method [43]	**21**± 0	247.47± 7.52	**22**±0	18.32±0.88	24± 0	5.34±0.31	28± 0	9.77±0.81
backpropagation [43]	—	—	—	—	4433±255	11.85±1.10	—	—
ACA [44]	607±30	232.90±13.81	69±2	17.72±1.38	73± 0	4.15±0.21	138± 0	9.36±0.55
proposed	589±14	262.99± 5.19	43±2	18.59±0.75	**20**± 0	4.78±0.32	**21**± 0	11.41±0.23

Peak memory consumption [MiB], and computation time per iteration [s/itr].

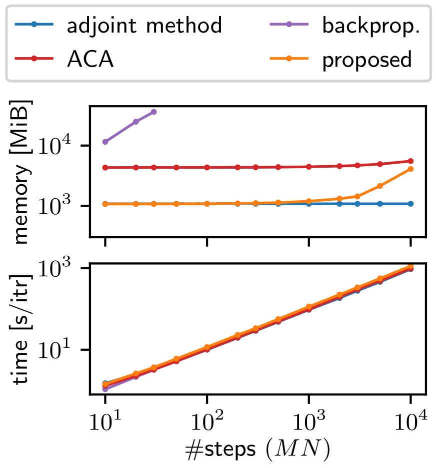


Fig. 3. Memory consumption and computation time with different numbers of steps for $\tilde{N} = N$.

(dopri8) methods. These methods have orders $p = 2, 3, 5$, and 8 using $s = 2, 3, 6$, and 12 function evaluations, respectively. We examined these methods using the GAS dataset; the results are summarized in Table V. The naive backpropagation algorithm consumed the entire memory in some cases; this is indicated in Table V by dashes. We omit the NLLs because all methods used the same tolerance and achieved the same NLLs.

Compared with ACA, the symplectic adjoint method suppressed the memory consumption more significantly as the number of function evaluations s increased; this agrees with the theoretical predictions in Table I. In the Heun–Euler case, all methods were extremely slow, and all but the adjoint method consumed more memory. A lower order method has to use an extremely small step size to satisfy the tolerance, thereby increasing N, computation time, and the memory for checkpoints. This indicates the limitations of methods that depend on lower order integrators, such as MALI [51]. With the eighth-order Dormand–Prince method, the adjoint method performs more quickly because the backward integration easily satisfies the tolerance with a higher order method (i.e., $\tilde{N} \simeq N$). Nonetheless, in terms of computation time, the fifth-order Dormand–Prince method is the best choice; for this case, the symplectic adjoint method greatly reduces the memory consumption and is faster than all but ACA.

3) Memory for Checkpoints: To evaluate the memory consumption with varying numbers of checkpoints, we used the fifth-order Dormand–Prince method and varied the number of steps N for MNIST by manually varying the step size. We show the results on a log–log scale in Fig. 3. Note that, with the adaptive stepping, FFJORD needed approximately $N = 200$ steps for MNIST and fewer steps for other datasets. Because we set $\tilde{N} = N$, but $\tilde{N} > N$ in practice, the adjoint method is expected to require a longer computation time.

The memory consumption roughly follows the theoretical predictions summarized in Table I. The adjoint method needs a memory of $O(L)$ for the backpropagation, and the symplectic adjoint method needs an additional memory of $O(N + s)$ for checkpoints. Until the number of steps N exceeds a thousand, the memory for checkpoints is negligible compared with that for backpropagation. Compared with the symplectic adjoint method, ACA needs a memory of $O(sL)$ for backpropagation over s stages. The increase in memory is significant until the number of steps N reaches tens of thousands. For some stiff (nonsmooth) ODEs, a numerical integrator may need thousands of steps. This number can be reduced by employing a higher order integrator, such as the eighth-order Dormand–Prince method. For even stiffer ODEs, implicit integrators may be used, but they are out of the scope of this study and of those listed in Table I. We conclude that, in practical ranges, the symplectic adjoint method needs the same level of memory as the adjoint method and much less than the other methods.

A possible implementation strategy alternative to the nested checkpointing scheme in Algorithms 1 and 2 would be retaining all intermediate states $X_{n,i}$ during the forward integration. The computational cost and memory consumption of this are $O(3NsL)$ and $O(Ns + L)$, respectively. In this case, the memory needed for checkpoints could be nonnegligible with a practical number of steps.

V. CONCLUSION

The symplectic adjoint method proposed here solves the adjoint system by a symplectic integrator with appropriate checkpoints and thereby provides the exact gradient. It only applies the backpropagation algorithm to each use of the neural network, and thus, consumes much less memory than the naive backpropagation algorithm and ACA. Its memory consumption is competitive with that of the adjoint method because the memory consumed by checkpoints is negligible in most cases. It provides the exact gradient with the same step size as that used for the forward integration and is, therefore, faster in practice than the adjoint method, which requires a small step size to suppress numerical errors.

Contrary to the theoretical prediction, the naive backpropagation algorithm is sometimes slower than the symplectic adjoint method because it may have to wait for a retained computation graph to be loaded or for memory to be freed.

Hence, if a memory budget is limited, the symplectic adjoint method is superior in terms of both memory consumption and computation time. The symplectic adjoint method and ACA have a tradeoff between memory consumption and computation time; ACA always works faster, and the symplectic adjoint method always consumes much less memory.

The symplectic adjoint method is based on the symplecticity of numerical integrators, and a similar methodology potentially designs integrators to obtain gradients for integrators other than the Runge–Kutta family [57]; this is included in future work. As shown in the experiments, the best integrator and checkpointing scheme may depend on the target system and computational resources. For example, Kim et al. [71] have demonstrated that quadrature methods can reduce the computational cost of the adjoint system for a stiff equation in exchange for additional memory consumption. Practical packages provide many integrators, from which the best ones can be chosen [72], [73]. In the future, we will offer the proposed symplectic adjoint method for inclusion in such packages.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[3] O. Nelles, *Nonlinear System Identification*. Berlin, Germany: Springer, 2001.

[4] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control, Signals Syst.*, vol. 2, no. 4, pp. 303–314, 1989.

[5] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Trans. Neural Netw.*, vol. 1, no. 1, pp. 4–27, Mar. 1990.

[6] S. Chen, S. A. Billings, and P. M. Grant, "Non-linear system identification using neural networks," *Int. J. Control*, vol. 51, no. 6, pp. 1191–1214, 1990.

[7] J. Sjöberg, H. Hjalmarsson, and L. Ljung, "Neural networks in system identification," *IFAC Proc. Volumes*, vol. 27, no. 8, pp. 359–382, 1994.

[8] A. U. Levin and K. S. Narendra, "Recursive identification using feedforward neural networks," *Int. J. Control*, vol. 61, no. 3, pp. 533–547, Mar. 1995.

[9] D. S. Clouse, C. L. Giles, B. G. Horne, and G. W. Cottrell, "Time-delay neural networks: Representation and induction of finite-state machines," *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 1065–1070, Sep. 1997.

[10] A. Ibeas, A. Esmaeili, J. Herrera, and F. Zouari, "Discrete-time observer-based state feedback control of heart rate during treadmill exercise," in *Proc. 20th Int. Conf. Syst. Theory, Control Comput. (ICSTCC)*, Oct. 2016, pp. 537–542.

[11] J. Na, S. Wang, Y.-J. Liu, Y. Huang, and X. Ren, "Finite-time convergence adaptive neural network control for nonlinear servo systems," *IEEE Trans. Cybern.*, vol. 50, no. 6, pp. 2568–2579, Jun. 2020.

[12] J. Na, Y. Huang, X. Wu, S. Su, and G. Li, "Adaptive finite-time fuzzy control of nonlinear active suspension systems with input delay," *IEEE Trans. Cybern.*, vol. 50, no. 6, pp. 2639–2650, Jan. 2019.

[13] S. Wang, X. Ren, J. Na, and T. Zeng, "Extended-state-observer-based funnel control for nonlinear servomechanisms with prescribed tracking performance," *IEEE Trans. Autom. Sci. Eng.*, vol. 14, no. 1, pp. 98–108, Jan. 2016.

[14] S. Wang, J. Na, and X. Ren, "RISE-based asymptotic prescribed performance tracking control of nonlinear servo mechanisms," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 48, no. 12, pp. 2359–2370, Dec. 2017.

[15] S. Wang, H. Yu, J. Yu, J. Na, and X. Ren, "Neural-network-based adaptive funnel control for servo mechanisms with unknown dead-zone," *IEEE Trans. Cybern.*, vol. 50, no. 4, pp. 1383–1394, Apr. 2020.

[16] F. Zouari, "Adaptive internal model control of a DC motor drive system using dynamic neural network," *J. Softw. Eng. Appl.*, vol. 5, no. 3, pp. 168–189, 2012.

[17] F. Zouari, K. B. Saad, and M. Benrejeb, "Adaptive backstepping control for a single-link flexible robot manipulator driven DC motor," in *Proc. Int. Conf. Control, Decis. Inf. Technol. (CoDIT)*, May 2013, pp. 864–871.

[18] F. Zouari, K. B. Saad, and M. Benrejeb, "Adaptive backstepping control for a class of uncertain single input single output nonlinear systems," in *Proc. 10th Int. Multi-Conf. Syst., Signals Devices*, Mar. 2013, pp. 1–6.

[19] F. Zouari, K. B. Saad, and M. Benrejeb, "Robust adaptive control for a class of nonlinear systems using the backstepping method," *Int. J. Adv. Robot. Syst.*, vol. 10, no. 3, p. 166, Mar. 2013.

[20] F. Zouari, A. Ibeas, A. Boulkroune, J. Cao, and M. M. Arefi, "Neural network controller design for fractional-order systems with input nonlinearities and asymmetric time-varying pseudo-state constraints," *Chaos, Solitons Fractals*, vol. 144, Mar. 2021, Art. no. 110742.

[21] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, vol. 8. Berlin, Germany: Springer, 1993.

[22] E. Hairer, C. Lubich, and G. Wanner, *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. vol. 31. Berlin, Germany: Springer, 2006.

[23] Y.-J. Wang and C.-T. Lin, "Runge–Kutta neural network for identification of dynamical systems in high accuracy," *IEEE Trans. Neural Netw.*, vol. 9, no. 2, pp. 294–307, Mar. 1998.

[24] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE Trans. Neural Netw.*, vol. 9, no. 5, pp. 987–1000, Sep. 1998.

[25] P. Ramuhalli, L. Udpa, and S. S. Udpa, "Finite-element neural networks for solving differential equations," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1381–1392, Nov. 2005.

[26] K. S. McFall and J. R. Mahan, "Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions," *IEEE Trans. Neural Netw.*, vol. 20, no. 8, pp. 1221–1233, Aug. 2009.

[27] K. Rudd, G. Di Muro, and S. Ferrari, "A Constrained backpropagation approach for the adaptive solution of partial differential equations," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 25, no. 3, pp. 571–584, Mar. 2014.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[29] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia, PA, USA: SIAM, 2008.

[30] A. Paszke et al., "Automatic differentiation in PyTorch," in *Proc. Autodiff Workshop Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2017.

[31] S. Greydanus, M. Dzamba, and J. Yosinski, "Hamiltonian neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2019.

[32] Y. Chen, T. Matsubara, and T. Yaguchi, "Neural symplectic form: Learning Hamiltonian equations on general coordinate systems," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2021.

[33] M. Finzi, K. A. Wang, and A. G. Wilson, "Simplifying Hamiltonian and Lagrangian neural networks via explicit constraints," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020.

[34] P. Jin, Z. Zhang, I. G. Kevrekidis, and G. E. Karniadakis, "Learning Poisson systems and trajectories of autonomous systems via Poisson neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Feb. 18, 2022, doi: 10.1109/TNNLS.2022.3148734.

[35] T. Matsubara and T. Yaguchi, "FINDE: Neural differential equations for finding and preserving invariant quantities," 2022, *arXiv:2210.00272*.

[36] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho, "Lagrangian neural networks," in *Proc. ICLR Workshop Integr. Deep Neural Models Differ. Equ.*, 2020, pp. 1–9.

[37] Y. D. Zhong, B. Dey, and A. Chakraborty, "Dissipative SymODEN: Encoding Hamiltonian dynamics with dissipation and control into deep learning," in *Proc. ICLR Workshop Integr. Deep Neural Models Differ. Equ.*, 2020, pp. 1–6.

[38] D. Pang, X. Le, X. Guan, and J. Wang, "LFT: Neural ordinary differential equations with learnable final-time," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Oct. 25, 2022, doi: 10.1109/TNNLS.2022.3213308.

[39] M. Lehtimaki, L. Paunonen, and M.-L. Linne, "Accelerating neural ODEs using model order reduction," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, May 26, 2022, doi: 10.1109/TNNLS.2022.3175757.

[40] T. Matsubara, A. Ishikawa, and T. Yaguchi, "Deep energy-based modeling of discrete-time physics," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020.

[41] M. Horie, N. Morita, T. Hishinuma, Y. Ihara, and N. Mitsume, “Isometric transformation invariant and equivariant graph convolutional networks,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2022.

[42] J. R. Dormand and P. J. Prince, “A reconsideration of some embedded Runge–Kutta formulae,” *J. Comput. Appl. Math.*, vol. 15, no. 2, pp. 203–211, Jun. 1986.

[43] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, “Neural ordinary differential equations,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2018.

[44] J. Zhuang, N. Dvornek, X. Li, S. Tatikonda, X. Papademetris, and J. Duncan, “Adaptive checkpoint adjoint method for gradient estimation in neural ODE,” in *Proc. Int. Conf. Mach. Learn. (ICLR)*, 2020.

[45] R. M. Errico, “What is an adjoint model?” *Bull. Amer. Meteorolog. Soc.*, vol. 78, no. 11, pp. 2577–2591, Nov. 1997.

[46] Q. Wang, “Forward and adjoint sensitivity computation of chaotic dynamical systems,” *J. Comput. Phys.*, vol. 235, pp. 1–13, Feb. 2013.

[47] J. M. Sanz-Serna, “Symplectic Runge–Kutta schemes for adjoint equations, automatic differentiation, optimal control, and more,” *SIAM Rev.*, vol. 58, no. 1, pp. 3–33, Jan. 2016.

[48] A. Griewank and A. Walther, “Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation,” *ACM Trans. Math. Softw.*, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[49] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, “Memory-efficient backpropagation through time,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4132–4140.

[50] A. Gholaminejad, K. Keutzer, and G. Biros, “ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs,” in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 730–736.

[51] J. Zhuang, N. C. Dvornek, S. Tatikonda, and J. S. Duncan, “MALI: A memory efficient and reverse accurate integrator for neural ODEs,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021.

[52] T. Matsubara, Y. Miyatake, and T. Yaguchi, “Symplectic adjoint method for exact gradient of neural ODE with minimal memory,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2021.

[53] H. K. Khalil, *Nonlinear Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.

[54] P. B. Bochev and C. Scovel, “On quadratic invariants and symplectic structure,” *BIT*, vol. 34, no. 3, pp. 337–345, Sep. 1994.

[55] C. Herrera, F. Krach, and J. Teichmann, “Neural jump ordinary differential equations: Consistent continuous-time prediction and filtering,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2020.

[56] Z. Li, J. V. Murkute, P. K. Gyawali, and L. Wang, “Rogressive learning and disentanglement of hierarchical representations,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2020.

[57] T. Matsuda and Y. Miyatake, “Generalization of partitioned Runge–Kutta methods for adjoint systems,” *J. Comput. Appl. Math.*, vol. 388, May 2021, Art. no. 113308.

[58] S. Bai, J. Z. Kolter, and V. Koltun, “Deep equilibrium models,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2019.

[59] S. Saemundsson et al., “Variational integrator networks for physically meaningful embeddings,” in *Proc. Int. Conf. Artif. Intell. Statist. (NeurIPS)*, vol. 108, 2020.

[60] D. Furihata and T. Matsuo, *Discrete Variational Derivative Method: A Structure-Preserving Numerical Method for Partial Differential Equations*. Boca Raton, FL, USA: CRC Press, 2010.

[61] D. Furihata, “Finite difference schemes for ∂u/∂t=(∂/∂x)αδG/Δu that inherit energy conservation or dissipation property,” *J. Comput. Phys.*, vol. 156, no. 1, pp. 181–205, Nov. 1999.

[62] D. Furihata, “A stable and conservative finite difference scheme for the Cahn-Hilliard equation,” *Numerische Math.*, vol. 87, no. 4, pp. 675–699, Feb. 2001.

[63] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, “FFJORD: Free-form continuous dynamics for scalable reversible generative models,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2018.

[64] L. Dinh, D. Krueger, and Y. Bengio, “NICE: Non-linear independent components estimation,” in *Proc. Workshop Int. Conf. Learn. Represent. (ICLR)*, 2014.

[65] L. Dinh, J. Sohl-Dickstein, and S. Bengio, “Density estimation using real NVP,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017.

[66] D. J. Rezende and S. Mohamed, “Variational inference with normalizing flows,” in *Proc. Int. Conf. Mach. Learn.*, vol. 37, 2015, pp. 3–6.

[67] M. F. Hutchinson, “A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines,” *Commun. Statist.-Simul. Comput.*, vol. 19, no. 2, pp. 433–450, Jan. 1990.

[68] G. Papamakarios, T. Pavlakou, and I. Murray, “Masked autoregressive flow for density estimation,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2017.

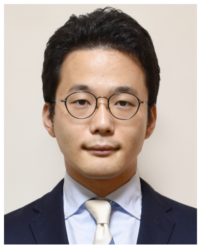
[69] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015.

[70] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2323, Nov. 1998.

[71] S. Kim, W. Ji, S. Deng, Y. Ma, and C. Rackauckas, “Stiff neural ordinary differential equations,” 2021, *arXiv:2103.15341*.

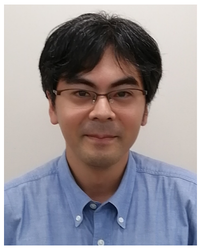
[72] A. C. Hindmarsh et al., “SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers,” *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 363–396, 2005.

[73] C. Rackauckas et al., “Universal differential equations for scientific machine learning,” 2020, *arXiv:2001.04385*.



Takashi Matsubara (Member, IEEE) was an Assistant Professor with the Graduate School of System Informatics, Kobe University, Hyogo, Japan, from 2015 to 2020. He is currently an Associate Professor with the Graduate School of Engineering Science, Osaka University, Osaka, Japan. His research interests include Bayesian and geometric inductive bias for data-driven modeling methods.

Yuto Miyatake was an Assistant Professor with the Graduate School of Engineering, Nagoya University, Nagoya, Japan. He has been an Associate Professor with Cybermedia Center, Osaka University, Osaka, Japan, since 2018. His research interests include numerical analysis of differential equations, numerical linear algebra, and computational uncertainty quantification.



Takaharu Yaguchi (Member, IEEE) was an Assistant Professor with the Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, Japan. He has been an Associate Professor with the Graduate School of System Informatics, Kobe University, Hyogo, Japan, since 2015. He has been working on mathematical engineering, with an emphasis on mathematical modeling, numerical analysis, geometric mechanics, and deep learning and its applications to physical simulation and psychology.