Differentiable Multiple Shooting Layers

Stefano Massaroli[*]
The University of Tokyo, DiffEqML
massaroli@robot.t.u-tokyo.ac.jp

Michael Poli[*]
KAIST, DiffEqML
poli_m@kaist.ac.kr

Sho Sonoda
RIKEN

Taiji Suzuki
The University of Tokyo, RIKEN

Jinkyoo Park
KAIST

Atsushi Yamashita
The University of Tokyo

Hajime Asama
The University of Tokyo

Abstract

We detail a novel class of implicit neural models. Leveraging time–parallel methods for differential equations, *Multiple Shooting Layers* (MSLs) seek solutions of initial value problems via parallelizable root-finding algorithms. MSLs broadly serve as drop–in replacements for *neural ordinary differential equations* (Neural ODEs) with improved efficiency in number of function evaluations (NFEs) and wall–clock inference time. We develop the algorithmic framework of MSLs, analyzing the different choices of solution methods from a theoretical and computational perspective. MSLs are showcased in long horizon optimal control of ODEs and PDEs and as latent models for sequence generation. Finally, we investigate the speedups obtained through application of MSL inference in neural controlled differential equations (Neural CDEs) for time series classification of medical data.

1 Introduction

For the last twenty years, one has tried to speed up numerical computation mainly by providing ever faster computers. Today, as it appears that one is getting closer to the maximal speed of electronic components, emphasis is put on allowing operations to be performed in parallel. In the near future, much of numerical analysis will have to be recast in a more "parallel" form. Nievergelt, 1964

Discovering and exploiting parallelization opportunities has allowed deep learning methods to succeed across application areas, reducing iteration times for architecture search and allowing scaling to larger data sizes (Krizhevsky et al., 2012; Diamos et al., 2016; Vaswani et al., 2017). Inspired by multiple shooting, time–parallel methods for ODEs (Bock and Plitt, 1984; Diehl et al., 2006; Gander, 2015; Staff and Rønquist, 2005) and recent advances on the intersection of differential equations, implicit problems and deep learning, we present a novel class of neural models designed to maximize parallelization across *time*: differentiable *Multiple Shooting Layers* (MSLs). MSLs seek solutions of *initial value problems* (IVPs) as roots of a function designed to ensure satisfaction of boundary constraints. Figure 1 provides visual intuition of the parallel nature of MSL inference.

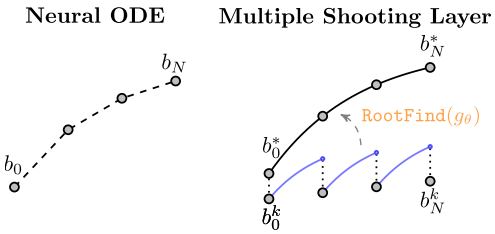

Figure 1: MSLs apply parallelizable root finding methods to obtain differential equation solutions.

[*]Equal contribution. Author order was decided by flipping a coin.

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

An implicit neural differential equation MSL inference is built on the interplay of numerical methods for root finding problems and differential equations. This property reveals the proposed method as a missing link between implicit–depth architectures such as Deep Equilibrium Newtorks (DEQs) (Bai et al., 2019) and continuous–depth models (Weinan, 2017; Chen et al., 2018; Massaroli et al., 2020; Kidger et al., 2020b; Li et al., 2020). Indeed, MSLs can be broadly applied as drop–in replacements for Neural ODEs, with the advantage of often requiring a smaller *number of function evaluations* (NFEs) for neural networks parametrizing the vector field. MSL variants and their computational signature are taxonomized on the basis of the particular solution algorithm employed, such as Newton and *parareal* (Maday and Turinici, 2002) methods.

Faster inference and fixed point tracking Differently from classical multiple shooting methods, MSLs operate in regimes where function evaluations of the vector field can be significantly more expensive than surrounding operations. For this reason, the reduction in NFEs obtained through time–parallelization leads to significant inference speedups. In full-batch training regimes, MSLs provably enable tracking of fixed points across training iterations, leading to drastic acceleration of forward passes (often the cost of a single root finding step). We apply the tracking technique to optimal control of ODEs and PDEs, with speedups in the order of several times over Neural ODEs. MSLs are further evaluated in sequence generation via a latent variant, and as a faster alternative to *neural controlled differential equations* (Neural CDE) (Kidger et al., 2020b) in long–horizon time series classification.

2 Multiple Shooting Layers

Consider the *initial value problem* (IVP)

$$\begin{aligned} \dot{z}(t) &= f_\theta(t, z(t)) \\ z(0) &= z_0 \end{aligned}, \quad t \in [0, T]. \tag{2.1}$$

with state $z \in \mathcal{Z} \subset \mathbb{R}^{n_z}$, parameters $\theta \in \mathcal{W}$ for some space $\mathcal{W}$ of functions $[0, T] \to \mathbb{R}^{n_\theta}$ and a smooth vector field $f_\theta : [0, T] \times \mathcal{Z} \times \mathcal{W} \to \mathcal{Z}$. For all $z \in \mathcal{Z},\ s, t \in [0, T]; s < t$ we denote with $\phi_\theta(z, s, t)$ the solution of (2.1) at time t starting from z at time s , i.e. $\phi_\theta(z, s, t) : (z, s, t) \mapsto z(t)$.

The crux behind *multiple–shooting* methods for differential equations is to turn the initial value problem (2.1) into a *boundary value problem* (BVP). We split the the time interval $[0, T]$ in N sub–intervals $[t_n, t_{n+1}]$ with $0 = t_0 < t_1 < \cdots < t_N = T$ and define N left boundary subproblems

$$z_n(t_n) = b_n \ \text{ and } \ \dot{z}_n(t) = f_\theta(t, z_n(t)),\ t \in [t_n, t_{n+1}] \tag{2.2}$$

where b_n are denoted as *shooting parameters*. At each time $t \in [0, T]$, the solution of (2.2) matches the one of (2.1) iff all the shooting parameters b_n are identical to $z(t_n)$, $b_n = \phi_\theta(z_0, t_0, t_n)$. Using $z(t_n) = \phi_\theta(z(t_{n-1}), t_{n-1}, t_n)$, we obtain the equivalent conditions

$$\begin{aligned} b_0 &= \phi_\theta(z_0, t_0, t_0) = z_0 \\ b_1 &= \phi_\theta(b_0, t_0, t_1) = z_0(t_1) \\ &\vdots \\ b_N &= \phi_\theta(b_{N-1}, t_{N-1}, t_N) = z_{N-1}(t_N) \end{aligned}$$

Let $B := (b_0, b_1, \cdots, b_N)$ and $\gamma_\theta(B, z_0) := (z_0, \phi_\theta(b_0, t_0, t_1), \cdots, \phi_\theta(b_{N-1}, t_{N-1}, t_N))$. We can thus turn the IVP (2.1) into the roots–finding problem the of a function g_θ defined as

$$g_\theta(B, z_0) = B - \gamma_\theta(B, z_0)$$

Definition 1 (Multiple Shooting Layer (MSL))**.** *With $\ell_x : \mathcal{X} \to \mathcal{Z}$ and $\ell_y : \mathcal{Z}^{N+1} \to \mathcal{Y}$ two affine maps, a* `multiple shooting layer` *is defined as the implicit input–output mapping $x \mapsto y$:*

$$\begin{aligned} z_0 &= \ell_x(x) \\ B^* \ &: \ g_\theta(B^*, z_0) = \mathbb{0} \\ y &= \ell_y(B^*) \end{aligned} \tag{2.3}$$

3 Realization of Multiple Shooting Layers

The remarkable property of MSL is the possibility of computing the solutions of all the N IVPs (2.2) in parallel from the shooting parameters in B with any standard ODE solver. This allows for a drastic reduction in the number of vector field evaluations at the cost of a higher memory requirement for the parallelization to take place. Nonetheless, the forward pass of MSLs requires the shooting parameters B to satisfy the nonlinear algebraic matching condition $g_\theta(B, z_0) = \mathbb{0}$, which has also to be solved numerically.

3.1 Forward Model

The *forward* MSL model involves the synergistic combination of two main classes of numerical methods: *ODE solvers* and *root finding* algorithms, to compute $\gamma_\theta(B, z_0)$ and B^*, respectively. There exists a hierarchy between the two classes of methods: the ODE solver will be invoked at each step k of the root finding algorithm to compute $\gamma_\theta(B^k, z_0)$ and evaluate the matching condition $g_\theta(B^k, z_0)$.

Newton methods for root finding Let us denote with B^k the solution of the root finding problem at the k-th step of the Newton method and let $\mathsf{D}g_\theta(B^k, z_0)$ be the Jacobian of g_θ computed in B^k. The solution $B^* \ : \ g_\theta(B^*, z_0) = \mathbb{0}$ can be obtained by iterating the Newton–Raphson fixed point iteration

$$B^{k+1} = B^k - \alpha \left[\mathbb{I}_N \otimes \mathbb{I}_{n_z} - \mathsf{D}\gamma_\theta(B^k, z_0)\right]^{-1} \left[B^k - \gamma_\theta(B^k, z_0)\right] \tag{3.1}$$

which converges quadratically to B^* (Nocedal and Wright, 2006). The exact Newton iteration theoretically (3.1) requires the inverse of the Jacobian $\mathbb{I}_N \otimes \mathbb{I}_{n_z} - \mathsf{D}\gamma_\theta(B^k, z_0)$. Without the special structure of the MSL problem, the Jacobian would have had to be the computed in full, as in the case of DEQs (Bai et al., 2019). Being the Jacobian of dimension $\mathbb{R}^{Nn_z \times Nn_z}$, its computation with *reverse–mode* automatic differentiation (AD) tools scales poorly with state dimensions and number of shooting parameter (cubically in both n_z and N).

Instead, the special structure of the MSL matching function $g_\theta(B, z_0) = B - \gamma_\theta(B, z_0)$ and its Jacobian, opens up application of direct updates where inversion is not required.

Direct multiple shooting Following the treatment of Chartier and Philippe (1993), we can obtain a *direct* formulation of the Newton iteration which does not require the composition of the whole Jacobian nor its inversion. The direct multiple shooting iteration is derived by setting $\alpha = 1$ and multiplying the Jacobian on both sides of (3.1) yielding

$$\left[\mathbb{I}_N \otimes \mathbb{I}_{n_z} - \mathsf{D}\gamma_\theta(B^k, z_0)\right] (B^{k+1} - B^k) = \gamma_\theta(B^k, z_0) - B^k$$

which leads to the following update rule for the individual shooting parameters b_n^k (see Fig. 2):

$$b_{n+1}^{k+1} = \phi_{\theta,n}(b_n^k) + \mathsf{D}\phi_{\theta,n}(b_n^k)\left(b_n^{k+1} - b_n^k\right), \quad b_0^{k+1} = z_0 \tag{3.2}$$

where $\mathsf{D}\phi_{\theta,n}(b_n^k) = \mathrm{d}\phi_{\theta,n}(b_n^k)/\mathrm{d}b_n$ is the sensitivity of each individual flow to its initial condition. Due to the dependence of b_{n+1}^{k+1} on b_n^{k+1}, a complete Newton iteration theoretically requires $N-1$ sequential stages.

Finite-step convergence Iteration (3.2) exhibits convergence to the exact solution of the IVP (2.1) in $N-1$ steps (Gander, 2018, Theorem 2.3). In particular, given perfect integration of the sub–IVPs, b_n^k coincides with the exact solution $\phi_\theta(z_0, t_0, t_n)$ from iteration index $k = n$ onward, i.e. at iteration k only the last $N-k$ shooting parameters are actually updated. Thus, the computational and memory footprint of the method diminishes with the number of iterations. This result can be visualized in the graphical representation of iteration (3.2) in Figure 3 while further details are discussed in Appendix B.1.

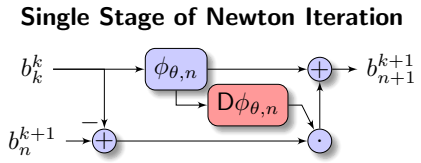

Figure 2: One stage of Newton iteration (3.2).

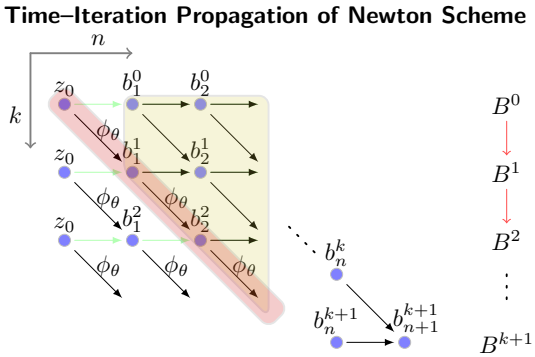

Figure 3: Propagation in k and n of the Newton iteration (3.2). The intertwining between the updates in n and k leads to the finite step convergence result. In fact, by setting $b_0^0 = z_0$, we see how the correcting term $b_n^{k+1} - b_n^k$ multiplying the flow sensitivity $\mathsf{D}\phi_{\theta,n}$ progressively nullifies at the same rate in n and k. As a result, the exact sequential solution of the IVP (2.1) unfolds on the diagonal $k = n$ and the only *active* part of the algorithm is the one above the diagonal (highlighted in yellow).

Numerical implementation Practical implementation of the Newton iteration (3.2) requires an ODE solver to approximate the flows $\phi_{\theta,n}(b_n^k)$ and an algorithm to compute their sensitivities w.r.t. b_n^k. Besides direct application of AD, we show an efficient alternative to obtain all $\mathsf{D}\phi_{\theta,n}$ in parallel alongside the flows, with a single call of the ODE solver.

Efficient exact sensitivities Differentiating through the steps of the forward numerical ODE solver using reverse–mode AD is straightforward, but incurs in high memory cost, additional computation to *unroll* the solver steps and introduces further numerical error on $\mathsf{D}\phi_{\theta,n}$. Even though the memory footprint might be mitigated by applying the *adjoint* method (Pontryagin et al., 1962), this still requires to solve backward the $N - k$ adjoint ODEs and sub–IVPs (2.2), at each iteration k. We leverage *forward* sensitivity analysis to compute $\mathsf{D}\phi_{\theta,n}$ alongside $\phi_{\theta,n}$ in a single call of the ODE solver. This approach, which might be considered as the continuous variant of forward–mode AD, scales quadratically with n_z, has low memory cost, and explicitly controls numerical error.

Proposition 1 (Forward Sensitivity (Khalil, 2002))**.** *Let $\phi_\theta(z, s, t)$ be the solution of (2.1). Then, $v(t) = \mathsf{D}\phi_\theta(z, s, t)$ satisfies the linear matrix–valued differential equations*

$$\dot{v}(t) = \mathsf{D}f_\theta(t, z(t))v(t), \quad v(s) = \mathbb{I}_{n_z} \quad \textit{where } \mathsf{D}f_\theta \textit{ denotes } \partial f_\theta / \partial z.$$

Therefore, at iteration k all $\mathsf{D}\phi_{\theta,n}(b_n^k)$ can be computed in parallel while performing the forward integration of the $N - k$ IVPs (2.2) and their forward sensitivities, i.e.

$$\texttt{ForwardSensitivity}: \ \left\{ b_n^k \mapsto (\phi_{\theta,n}, \mathsf{D}\phi_{\theta,n}) \right\}_{k<n\leq N}$$

which enables full *vectorization* of Jacobian–matrix products between $\partial f_\theta / \partial z$ and v as well as maximizing re–utilization of vector field evaluations. Detailed derivations are provided in Appendix B.2. Appendix C.1 analyzes practical considerations and software implementation of the algorithm.

Zero–order approximate iteration In high state dimension regimes, the quadratic memory scaling of the forward sensitivity method might be infeasible. If this is the case, a zero–order approximation of the Newton iteration preserving the finite–step converge property can be employed: the *parareal* method Lions et al. (2001). From the Taylor expansion of $\phi_{\theta,n}(b_n^{k+1})$ around b_n^k

$$\phi_{\theta,n}(b_n^{k+1}) = \phi_{\theta,n}(b_n^k) + \mathsf{D}\phi_{\theta,n}(b_n^k)\left(b_n^{k+1} - b_n^k\right) + o\left(\|b_n^{k+1} - b_n^k\|_2^2\right),$$

we have the following approximant for the correction term of (3.2)

$$\mathsf{D}\phi_{\theta,n}(b_n^k)\left(b_n^{k+1} - b_n^k\right) \approx \phi_{\theta,n}(b_n^{k+1}) - \phi_{\theta,n}(b_n^k). \tag{3.3}$$

Parareal computes the RHS of (3.3) by *coarse*[2] numerical solutions $\psi_{\theta,n}(b_n^k)$, $\psi_{\theta,n}(b_n^{k+1})$ of $\phi_{\theta,n}(b_n^k)$, $\phi_{\theta,n}(b_n^{k+1})$, leading to the forward iteration,

$$b_{n+1}^{b+1} = \phi_{\theta,n}(b_n^k) + \psi_{\theta,n}(b_n^{k+1}) - \psi_{\theta,n}(b_n^k).$$

[2]e.g. few steps of a low–order ODE solver

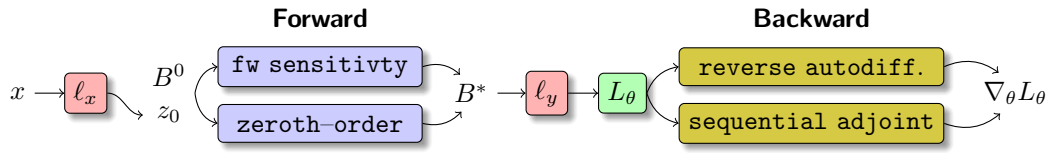

Figure 4: Scheme of the forward–backward pass of MSLs. After applying the input map ℓ_x to the input x and choosing initial shooting parameters B^0, the forward pass is iteratively computed with one of the numerical schemes described in Sec. 3.1 which, in turn, makes use of some ODE solver to compute $\phi_{\theta,n}$ in parallel, at each step. Once the output y and the loss are computed computed by applying ℓ_y and L_θ to B^*, the loss gradients can be computed by standard adjoint methods or reverse–mode automatic differentiation.

3.2 Properties of MSLs

Differentiating through MSL Computing loss gradients through MSLs can be performed by directly back–propagating through the steps of the numerical solver via reverse–mode AD.

A memory efficient alternative is to apply the sequential adjoint method to the underlying Neural ODE. In particular, consider a loss functions computed independently with the values of different shooting parameters, $L(x, B^*, \theta) = \sum_{n=1}^{N} c_\theta(x, b_n^*)$. The adjoint gradient for the MSL is then given by

$$\nabla_\theta L = \int_0^T \lambda^\top(t) \nabla_\theta f_\theta(t, z(t)) \mathrm{d}t$$

where the Lagrange multiplier $\lambda(t)$ satisfies a backward piecewise–continuous linear ODE

$$\begin{aligned} \dot{\lambda}(t) &= -\mathsf{D} f_\theta(t, z(t))\lambda(t) && \text{if } t \in [t_n, t_{n+1}) \\ \lambda^-(t_n) &= \lambda(t_n) + \nabla_b^\top c_\theta(x, b_n) && \lambda(T) = \nabla_b^\top c_\theta(x, b_N) \end{aligned}$$

The adjoint method typically requires the IVP (2.1) to be solved backward alongside λ to retrieve the value of $z(t)$ needed to compute the Jacobians $\mathsf{D} f_\theta$ and $\nabla_\theta f_\theta$. This step introduces additional errors on the final gradients: numerical errors accumulated on $b_N^* \approx z(T)$ during forward pass, propagate to the gradients and sum up with errors on the backward integration of (2.1).

Here we take a different, more robust direction by interpolating the shooting parameters and drop the integration of (2.1) during the backward pass. The values of the shooting parameters retrieved by the forward pass of MSLs are solution points of the IVP (2.1), i.e. $b_n^* = \phi(z_0, t_0, t_n)$ (up to the forward numerical solver tolerances). On this assumption, we construct a `cubic spline` interpolation $\hat{z}(t)$ of the shooting parameters b_n^* and we query it during the integration of λ to compute the Jacobians of f_θ. Further results on back–propagation of MSLs are provided in Appendix B.3. Appendix C.4 practical aspects of the backward model alongside software implementation of the *interpolated* adjoint. The entire scheme of a forward–backward pass of an MSL is shown in Figure 4.

One-step inference: fixed point tracking Consider training a MSL to minimize a twice–differentiable loss function $L(x, B^*, \theta)$ with Lipschitz constant m_L^θ through the gradient descent iteration

$$\theta_{p+1} = \theta_p - \eta_p \nabla_\theta L(x, B_p^*, \theta_p)$$

where η_p is a positive learning rate and B_p^* is the exact root of the matching function g_θ computed with parameters θ_p (i.e. the exact solution of the IVP (2.1) at the boundary points). Due to Lipschitzness of L, we have the following uniform bound on the variation of the parameters across training iterations

$$\|\theta_{p+1} - \theta_p\|_2 \leq \eta_p m_L^\theta.$$

If we also assume γ_θ to be Lipschitz continuous w.r.t z and θ with constants m_γ^θ, m_γ^z and differentiable w.r.t. θ we can obtain the variation of the fixed point B^* to small changes in the model parameters by linearizing solutions around θ_p

$$B_{p+1}^* - B_p^* = [\theta_{p+1} - \theta_p] \frac{\partial \gamma_{\theta_p}(B_p^*, z_0)}{\partial \theta} + o(\|\theta_{p+1} - \theta_p\|_2^2)$$

to obtain the uniform bound

$$\|B_{p+1}^* - B_p^*\|_2 \leq \eta_p m_L^\theta m_\gamma^\theta + o(\|\theta_{p+1} - \theta_p\|_2^2).$$

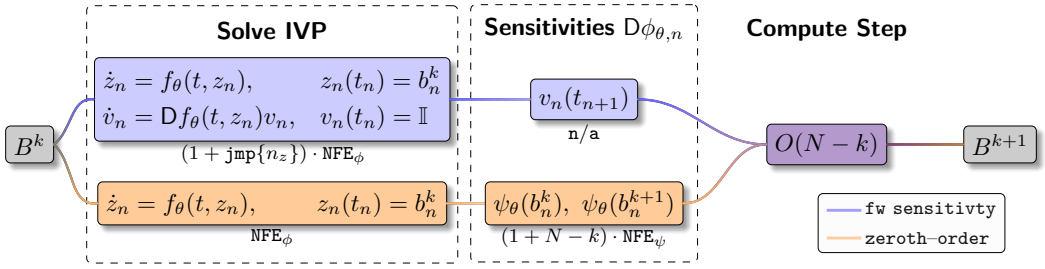

Figure 5: Single iteration computational *span* (McCool et al., 2012) in MSL. We normalize to 1 the cost of evaluating f_θ. The $N-k$ sub–IVPs are solved in parallel in their sub–intervals, thus requiring a minimum span $\mathtt{NFE}_\phi$. *Forward sensitivity* introduces Jacobian–matrix products costs amounting to `jmp`, which can be further parallelized into `jvps`.

Having a bounded variation on the solutions of the MSL for small changes of the model parameters θ, we might think of *recycling* the previous shooting parameters B_p^* as an initial guess for the direct Newton algorithms in the forward pass succeeding the gradient descent update. We show that by choosing a sufficiently small learning rate η_p one Newton iteration can be sufficient to track the true value of B^* during training. In particular, the following bound can be obtained.

Theorem 1 (Quadratic fixed-point tracking)**.** *If f_θ is twice continuously differentiable in z then*

$$\|B_{p+1}^* - \bar{B}_p^*\|_2 \leq M\eta_p^2 \tag{3.4}$$

for some $M > 0$. $\bar{B}_p^*$ is the result of one Newton iteration applied to B_p^*.

The proof, reported in Appendix A.1, relies on the quadratic converge of Newton method. The quadratic dependence of the tracking error bound on η_k allows use of typical learning rates for standard gradient based optimizers to keep the error under control. In this way, we can turn the *implicit* learning problem into an explicit one where the implicit inference pass reduces to one Newton iteration. This approach leads to the following training dynamics:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(x, B^*, \theta)$$
$$B^* \leftarrow \mathtt{apply}\{(3.2), B^*\}$$

We note that the main limitation of this method is the assumption on input x to be constant across training iterations (i.e. the initial condition z_0 is constant as well). If the input changes during the training (e.g. under mini-batch SGD training regime), the solutions of the IVP (2.1) and thus its corresponding shooting parameters may drastically change with x even for small learning rates.

Numerical scaling Each class of MSL outlined in Section 3.1 is equipped with unique computational scaling properties. In the following, we denote with $\mathtt{NFE}_\phi$ the total number of vector field f_θ evaluations done, in parallel across shooting parameters, in a single sub–interval $[t_n, t_{n+1}]$. Similarly, $\mathtt{NFE}_\psi$ indicates the function evaluations required by the *coarse* solver used for *parareal* approximations. Here, we set out to investigate the computational signature of MSLs as parallel algorithms. To this end, we decompose a single MSL iteration into two core steps: solving the IVPs across sub–intervals and computing sensitivities $\mathrm{D}\phi$ (or their approximation). Figure 5 provides a summary of the *algorithmic span*[3] of a single MSL iteration as a function of number of Jacobian vector products (`jvp`), Jacobian matrix products (`jmp`), and vector field evaluations (`NFE`).

`Fw sensitivity` MSL frontloads the cost of computing $\mathrm{D}\phi_{\theta,n}$ by solving the forward sensitivity ODEs of Proposition 1 alongside the evaluation of γ_θ. Forward sensitivity equations involve a `jmp`, which can be optionally further parallelized as n_z `jvps` by paying a memory overhead. Once sensitivities have been obtained along with γ_θ, no additional computation needs to take place until application of the shooting parameter update formula. The forward sensitivity approach thus enjoys the highest degree of time–parallelizability at a larger memory cost.

Zeroth–order MSL computes γ_θ via a total of $\mathtt{NFE}_\phi$ evaluations f_θ, parallelized across sub–intervals. The cheaper IVP solution in both memory and compute is however counterbalanced during calculation

[3]Longest sequential cost, in terms of computational primitives, that is not parallelizable due to problem–specific dependencies. A specific example for MSLs are the sequential $\mathtt{NFE}_\phi$ calls required by the sequential ODE solver for each sub–interval.

of the sensitivities, as this MSL approach approximates the sensitivities $\mathsf{D}\phi_{\theta,n}$ by a zeroth–order update requiring $N-k$ sequential calls to a coarse solver.

The analysis of MSL backpropagation scaling is straightforward, as sequential adjoints for MSLs mirror standard sensitivity techniques for Neural ODEs in both compute and memory footprints. Alternatively, `AD` can be utilized to backpropagate through the operations of the forward pass methods in use. This approach introduces a non–constant memory footprint which scales in the number of forward iterations and thus depth of computational graph.

4 Applications

4.1 Variational MSL

Let $x : \mathbb{R} \to \mathbb{R}^{n_x}$, be an observable of some continuous–time process and let $X = \{x_{-M}, \dots, x_0, \dots, x_N\} \in \mathbb{R}^{(M+N+1)\times n_x}$ be a sequence of observations of $x(t)$ at time instants $t_{-M} < \dots < t_0 < \dots < t_N$. We seek a model able to predict $x_1, \dots, x_N$ given past observations $x_{-M}, \dots, x_0$, equivalent to approximating the conditional distribution $p(x_{1:N}|x_{-M:0})$. To this end we introduce *variational* MSLs (vMSLs) as the following latent variable model:

$$
\begin{aligned}
(\mu, \Sigma) &= \mathcal{E}_\omega(x_{-M:0}) && \text{Encoder } \mathcal{E}_\omega \\
q_\omega(z_0|x_{-M:0}) &= \mathcal{N}(\mu, \Sigma) && \text{Approx. Posterior} \\
z_0 &\sim q_\omega(z_0|x_{-M:0}) && \text{Reparametrization}
\end{aligned}
\quad \Bigg| \quad
\begin{aligned}
B^* : g_\theta(B^*, z_0) &= \mathbb{0} && \text{Decoder } \mathcal{D}_\theta \\
\hat{x}_1, \dots, \hat{x}_N &= \ell(B^*) && \text{Readout } \ell
\end{aligned}
$$

Once trained, such model can be also used to generate new realistic sequences of the observable $x(t)$ by querying the decoder network at a desired z_0. vMSLs are designed to scale data generation to longer sequences, exploiting wherever possible parallel computation in time in both encoder as well as decoder modules. The structure of $\mathcal{E}_\omega$ is designed to leverage modern advances in representation learning for time–series via temporal convolutions (TCNs) or attention operators (Vaswani et al., 2017) to offer a higher degree of parallelizability compared to sequential encoders e.g RNNs, ODE-RNNs (Rubanova et al., 2019) or Neural CDEs (Kidger et al., 2020b). This, in turn, allows the encoder to match the decoder in efficiency, avoiding unnecessary bottlenecks. The decoder $\mathcal{D}_\theta$ is composed of a MSL which is tasked to unroll the generated trajectory in latent space. vMSLs are trained via traditional likelihood methods. The iterative optimization problem can be cast as the maximization of an evidence lower bound (`ELBO`):

$$
\min_{(\theta,\omega)} \mathbb{E}_{z_0 \sim q_\omega(z_0|x_{-M:0})} \sum_{n=1}^{N} \log p_n(\hat{x}_n) - \mathtt{KL}(q_\omega || \mathcal{N}(\mathbb{0}, \mathbb{I}))
$$

with $p_n(\hat{x}_n) = \mathcal{N}(x_n, \sigma_n)$ and the standard deviations σ_n are left as a hyperparameters.

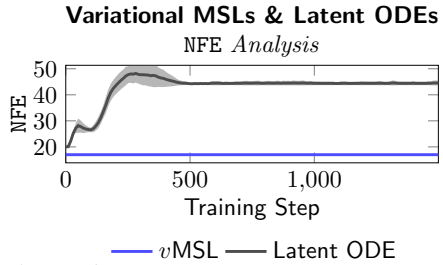

Figure 6: Mean and standard deviation of NFEs during vMSL and Latent Neural ODE across training trials. vMSLs require 60% less NFEs during both training and inference.

Sequence generation under noisy observations We apply vMSLs on a sequence generation task given trajectories corrupted by state–correlated noise. We consider a baseline latent model sharing the same overall architecture as vMSLs, with a Neural ODE as decoder. In particular, the Neural ODE is solved via the `dopri5` solver with absolute and relative tolerances set to 10^{-4}, whereas the vMSL decoder is an instance of `fw sensitivity` MSL. The encoder for both models is comprised of two layers of *temporal convolutions* (TCNs). All decoders unroll the trajectories directly in output space without additional readout networks. The validation on sample quality is then performed by inspection of the learned vector fields and the error against the nominal across the entire state–space. The proposed model obtains equivalent results as the baseline at a significantly cheaper computational cost. As shown in Figure 6, vMSLs require 60% less NFEs for a single training iteration as well as for sample generation, achieving results comparable to standard Latent Neural ODEs (Rubanova et al., 2019). We report further details and results in Appendix E.1.

4.2 Neural Optimal Control

Beyond sequence generation, the proposed framework can be applied to *optimal control*. Here we can fully exploit the drastic computational advantages of MSLs. In fact, we leverage on the natural assumption of finiteness of initial conditions z_0 where the controlled system (or *plant*) is initialized to verify the result of Th. 1. Let us consider a controlled dynamical system

$$\dot{z}(t) = f(t, z(t), \pi_\theta(t, z)), \quad z(0) = z_0 \quad (4.1)$$

with a parametrized policy $\pi_\theta : t, z \mapsto \pi_\theta(t, z)$ and initial conditions z_0 ranging in a finite set $Z_0 = \{z_0^j\}_j$. We consider the problems of stabilizing a low–dimensional *limit cycle* and deriving an optimal boundary policy for a linear PDE discretized as a 200–dimensional ODE.

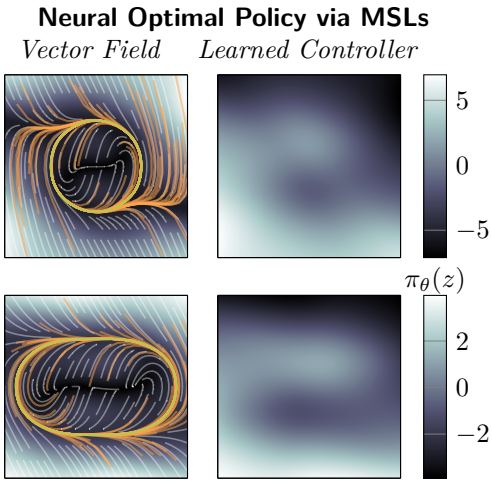

Figure 7: [Left] Closed–loop vector fields and trajectories corresponding to the u_θ-controlled MSL. [Right] Learned controller $u_\theta(z)$ ($z \in \mathbb{R}^2$) for the two different desired limit cycles. Although, the inference of the all trajectory is performed with just two steps of `RK4` (8 `NFE`), the initial accuracy of `dopri5` (> 3000 `NFE`) is preserved throughout training.

Limit cycle stabilization We consider a *stabilization* task where, given a desired closed curve $S_d = \{z \in \mathcal{Z} : s_d(z) = 0\}$, we minimize the 1-norm between the given curve and the MSL solution of (4.1) across the timestamps as well as the *control effort* $|\pi_\theta|$. We verify the approach on a one degree–of–freedom mechanical system, aiming with closed curves S_d of various shapes. Following the assumptions on fixed point tracking and slow–varying–flows of Th. 1, we initialize B_j^0 by `dopri5` adaptive–step solver set with tolerances 10^{-8}. Then, at each training iteration, we perform inference with a single parallel `rk4` step for each sub–interval $[t_n, t_{n+1}]$ followed by a single Newton update. Figure 7 shows the learned vector fields and controller, confirming a successful system stabilization of the system to different types of closed curves. We compare with a range of baseline Neural ODEs, solved via `rk4` and `dopri5`. Training of the controller via MSLs is achieved with orders of magnitude *less* wall–clock time and NFEs. Figure 8 shows the difference in NFEs w.r.t. `dopri5` while Fig. 9 compares wall–clock times per training iteration of MSL, `dopri5` and $rk4$.

We further provide *Symmetric Mean Average Percentage Error* (SMAPE) measurements between trajectories obtained via MSLs and an adaptive–step solver. MSLs initialized with recycled solutions are able to track the nominal trajectories across the entire training process. Additional details on the experimental setup, including wall–clock time comparisons with `rk4` and `dopri5` baseline Neural ODEs is are provided in Appendix E.2.

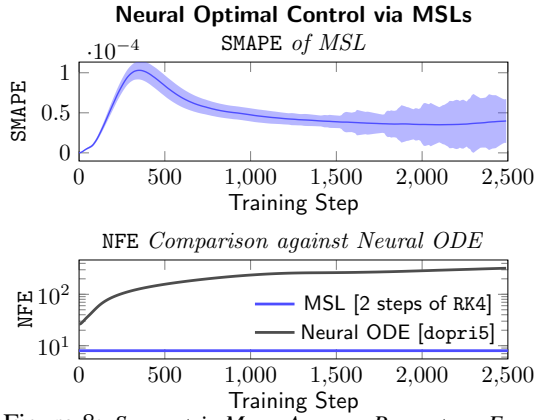

Figure 8: *Symmetric Mean Average Percentage Error* (SMAPE) between solutions of the controlled systems obtained by MSLs and nominal. Compared to Neural ODEs, MSLs solve the optimal control problem with NFE savings of several orders of magnitude by carrying forward their solution across training iterations.

Neural Boundary Control of the Timoshenko Beam We further show how MSLs can be scaled to high–dimensional regimes by tackling the optimal boundary control problem for a linear partial differential equation. In particular, we consider the *Timoshenko beam* (Macchelli and Melchiorri, 2004) model in Hamiltonian form. We derive formalize the boundary control problem and obtain a structure–preserving spectral discretization yielding a 160–dimensional Hamiltonian ODE.

We parameterize the boundary control policy with a multi–layer perceptron taking as input (control feedback) the 160–dimensional discrete state. We train the model in similar setting to the previous example having the MSL equipped with `fw sensitivity` and one step of `rk4` for the parallel

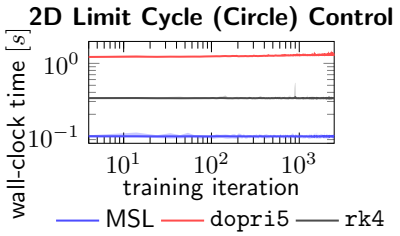

Figure 9: Mean and standard deviation of wall-clock time of complete training iteration (forward/backward passes + GD update) for different solvers on the *circle experiment*.

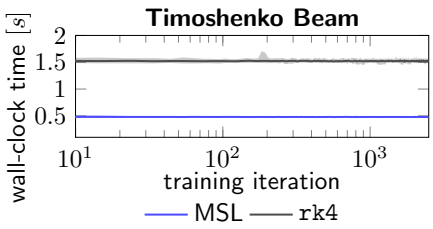

Figure 10: Mean and standard deviation of wall–clock time per training iteration for MSL and Neural ODE across training trials. MSLs require are three times faster than the sequential `rk4` with same accuracy (step size).

integration. We compare the training wall–clock time with a Neural ODE solved by sequential `rk4` in Fig. 10. The resulting speed up of MSL with forward sensitivity is three time faster than the baseline Neural ODE proving that the proposed method is able to scale to high–dimensional regimes. We include a formal treatment of the boundary control problem in Appendix D while further experimental details are provided in Appendix E.3.

4.3 Fast Neural CDEs for Time Series Classification

To further verify the broad range of applicability of MSLs, we apply them to time series classification as faster alternatives to *neural controlled differential equations* (Neural CDEs) (Kidger et al., 2020b). Here, MSLs remain applicable since Neural CDEs are practically solved as ODEs with a special structure, as described in Appendix E.4. We tackle the `PhysioNet` 2019 challenge (Goldberger et al., 2000) on `sepsis` prediction, following the exact experimental procedure described by Kidger et al. (2020b), including hyperparameters and Neural CDE architectures. However, we train all models on the full dataset to enable application of the fixed point tracking technique for MSLs[4]. Figure 11 visualized training convergence of zeroth–order MSL Neural CDEs and the baseline Neural CDE solved with rk4 as in the original paper. Everything else being equal, including architecture and backpropagation via sequential adjoints, MSL Neural CDEs converge with total wall–clock time one order of magnitude smaller than the baseline.

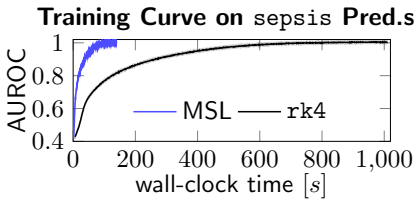

Figure 11: Mean and standard deviation of `AUROC` during training of MSLs and baseline Neural CDEs on sepsis prediction.

5 Related Work

Parallel–in–time integration & multiple shooting MSLs belong to the framework of time–parallel integration algorithms. The study of these methods is relatively recent, with seminal work in the 60s (Nievergelt, 1964). The *multiple shooting* formulation of time–parallel integration, see e.g. (Bellen and Zennaro, 1989) or (Chartier and Philippe, 1993), finally lead to the modern algorithmic form using the Newton iteration reported in (3.2). *Parareal* (Lions et al., 2001) has been successively introduced as a cheaper approximated solution of multiple shooting problem, rapidly spreading across application domains, e.g. optimal control of partial differential equations (Maday and Turinici, 2002). We refer to (Gander, 2015, 2018) as excellent introduction to the topic. We also note recent work (Vialard et al., 2020) introducing *single shooting* terminology for Neural ODEs (Chen et al., 2018), albeit in the unrelated context of learning time–varying parameters.

[4]We note that the training for all models has been performed on a single NVIDIA RTX A6000 with 48`Gb` of GPU memory.

Time–parallelization in neural models In the pursuit for increased efficiency, several works have proposed approaches to parallelization across time or *depth* in neural networks. (Gunther et al., 2020; Kirby et al., 2020; Sun et al., 2020) use multigrid and penalty methods to achieve speedups in ResNets. Meng et al. (2020) proposed a parareal variant of Physics–informed neural networks (PINNs) for PDEs. Zhuang et al. (2021) uses a penalty–variant of multiple shooting with adjoint sensitivity for parameter estimation in the medical domain. Solving the boundary value problems with a regularization term, however, is not guaranteed to converge to a continuous solution. The method of Zhuang et al. (2021) further optimizes its parameters in a full–batch regime, where application of (1) achieves drastic speedups while preserving convergence guarantees. Recent theoretical work (Lorin, 2020) has applied *parareal* methods to Neural ODEs. However, their analysis is limited to the theoretical computational complexity setting and does not involve multiple shooting methods nor derives its implicit differentiation formula.

In contrast our objective is to introduce a novel class of implicit time–parallel models, and to validate their computational efficiency across settings.

6 Conclusion

This work introduces differentiable *Multiple Shooting Layers* (MSLs), a parallel–in–time alternative to neural differential equations. MSLs seek solutions of differential equations via parallel application of root finding methods across solution subintervals. Here, we analyze several model variants, further proving a fixed point tracking property that introduces drastic speedups in full–batch training. The proposed approach is validated on different tasks: as generative models, MSLs are shown to achieve same task performance as Neural ODE baselines with 60% less NFEs, whereas they are shown to offer several orders of magnitude faster in optimal control tasks.

Remarkably few methods have been proposed for parallel integration of ODEs. In part this is because the problems do not have much natural parallelism. (Gear, 1988)

Funding Statement

This work was financially supported by The University of Tokyo, KAIST and RIKEN AIP. All experiments were run on GPUs provided by The University of Tokyo and KAIST.

References

M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems*, pages 690–701, 2019.

A. Bellen and M. Zennaro. Parallel algorithms for initial-value problems for difference and differential equations. *Journal of Computational and applied mathematics*, 25(3):341–350, 1989.

H. G. Bock and K.-J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2):1603–1608, 1984.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of computation*, 19(92):577–593, 1965.

P. Chartier and B. Philippe. A parallel shooting technique for solving dissipative ode's. *Computing*, 51(3-4):209–236, 1993.

R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 6571–6583. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/69386f6bb1dfed68692a24c8686939b9-Paper.pdf.

G. D. Clifford, I. Silva, B. Moody, Q. Li, D. Kella, A. Shahin, T. Kooistra, D. Perry, and R. G. Mark. The physionet/computing in cardiology challenge 2015: reducing false arrhythmia alarms in the icu. In *2015 Computing in Cardiology Conference (CinC)*, pages 273–276. IEEE, 2015.

G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh. Persistent rnns: Stashing weights on chip. 2016.

M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber. Fast direct multiple shooting algorithms for optimal robot control. In *Fast motions in biomechanics and robotics*, pages 65–93. Springer, 2006.

M. J. Gander. 50 years of time parallel time integration. In *Multiple shooting and time domain decomposition methods*, pages 69–113. Springer, 2015.

M. J. Gander. Time parallel time integration. 2018.

C. W. Gear. Parallel methods for ordinary differential equations. *Calcolo*, 25(1-2):1–20, 1988.

A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals. *circulation*, 101(23):e215–e220, 2000.

S. Gunther, L. Ruthotto, J. B. Schroder, E. C. Cyr, and N. R. Gauger. Layer-parallel training of deep residual neural networks. *SIAM Journal on Mathematics of Data Science*, 2(1):1–23, 2020.

J. Jia and A. R. Benson. Neural jump stochastic differential equations. *arXiv preprint arXiv:1905.10403*, 2019.

H. K. Khalil. *Nonlinear systems*, volume 3. Prentice Hall, 2002.

P. Kidger, R. T. Chen, and T. Lyons. " hey, that's not an ode": Faster ode adjoints with 12 lines of code. *arXiv preprint arXiv:2009.09457*, 2020a.

P. Kidger, J. Morrill, J. Foster, and T. Lyons. Neural controlled differential equations for irregular time series. *arXiv preprint arXiv:2005.08926*, 2020b.

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

A. Kirby, S. Samsi, M. Jones, A. Reuther, J. Kepner, and V. Gadepally. Layer-parallel training with gpu concurrency of deep residual neural networks via nonlinear multigrid. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2020.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

X. Li, T.-K. L. Wong, R. T. Chen, and D. Duvenaud. Scalable gradients for stochastic differential equations. In *International Conference on Artificial Intelligence and Statistics*, pages 3870–3882. PMLR, 2020.

J.-L. Lions, Y. Maday, and G. Turinici. Résolution d'edp par un schéma en temps pararéel. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(7):661–668, 2001.

E. Lorin. Derivation and analysis of parallel-in-time neural ordinary differential equations. *Annals of Mathematics and Artificial Intelligence*, 88(10):1035–1059, 2020.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

A. Macchelli and C. Melchiorri. Modeling and control of the timoshenko beam. the distributed port hamiltonian approach. *SIAM Journal on Control and Optimization*, 43(2):743–767, 2004.

A. Macchelli, A. J. Van Der Schaft, and C. Melchiorri. Port hamiltonian formulation of infinite dimensional systems i. modeling. In *2004 43rd IEEE Conference on Decision and Control (CDC)(IEEE Cat. No. 04CH37601)*, volume 4, pages 3762–3767. IEEE, 2004.

Y. Maday and G. Turinici. A parareal in time procedure for the control of partial differential equations. *Comptes Rendus Mathematique*, 335(4):387–392, 2002.

S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama. Dissecting neural odes. *arXiv preprint arXiv:2002.08071*, 2020.

M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

X. Meng, Z. Li, D. Zhang, and G. E. Karniadakis. Ppinn: Parareal physics-informed neural network for time-dependent pdes. *Computer Methods in Applied Mechanics and Engineering*, 370:113250, 2020.

J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Communications of the ACM*, 7(12):731–733, 1964.

J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

A. Pal, Y. Ma, V. Shah, and C. Rackauckas. Opening the blackbox: Accelerating neural differential equations by regularizing internal solver heuristics. *arXiv preprint arXiv:2105.03918*, 2021.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

M. Poli, S. Massaroli, A. Yamashita, H. Asama, and J. Park. Hypersolvers: Toward fast continuous-depth models. *arXiv preprint arXiv:2007.09601*, 2020a.

M. Poli, S. Massaroli, A. Yamashita, H. Asama, and J. Park. Torchdyn: A neural differential equations library. *arXiv preprint arXiv:2009.09346*, 2020b.

L. S. Pontryagin, E. Mishchenko, V. Boltyanskii, and R. Gamkrelidze. The mathematical theory of optimal processes. 1962.

C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit. Diffeqflux. jl-a julia library for neural differential equations. *arXiv preprint arXiv:1902.02376*, 2019.

Y. Rubanova, R. T. Q. Chen, and D. K. Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 5320–5330. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/42a6845a557bef704ad8ac9cb4461d43-Paper.pdf.

L. N. Smith and N. Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

G. A. Staff and E. M. Rønquist. Stability of the parareal algorithm. In *Domain decomposition methods in science and engineering*, pages 449–456. Springer, 2005.

Q. Sun, H. Dong, Z. Chen, W. Dian, J. Sun, Y. Sun, Z. Li, and B. Dong. Penalty and augmented lagrangian methods for layer-parallel training of residual networks. *arXiv preprint arXiv:2009.01462*, 2020.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

F.-X. Vialard, R. Kwitt, S. Wei, and M. Niethammer. A shooting formulation of deep learning. *Advances in Neural Information Processing Systems*, 33, 2020.

E. Weinan. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, 2017.

J. Zhuang, N. Dvornek, S. Tatikonda, X. Papademetris, P. Ventola, and J. Duncan. Multiple-shooting adjoint method for whole-brain dynamic causal modeling. *arXiv preprint arXiv:2102.11013*, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] **See Section 3.2 for numerical scaling of the model, and the Appendix for additional information.**
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] **See the Appendix for details on hardware used.**
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Differentiable Multiple Shooting Layers
Supplementary Material

Table of Contents

A Proofs **15**
- A.1 Proof of Theorem 1 . . . 15
- A.2 Proof of Proposition 1 . . . 16

B Additional Theoretical Results **17**
- B.1 Finite–Step Convergence . . . 17
- B.2 Flows Sensitivities $\mathsf{D}\phi_{\theta,n}$ . . . 18
- B.3 Backward Model of Multiple Shooting Layers . . . 18

C Additional Details on the Realization of MSLs **20**
- C.1 Software Implementation of Forward Sensitivity . . . 20
- C.2 Implementation of Direct Newton Method . . . 21
- C.3 Alternative Approaches to MSL Inference . . . 22
- C.4 Implementation of Backward Interpolated Adjoint . . . 23
- C.5 Broader Impact . . . 24

D Neural Network Control of the Timoshenko Beam **25**
- D.1 Port–Based Modeling of the Timoshenko Beam . . . 25
- D.2 Discretization of the Problem . . . 26
- D.3 Control by Neural Approximators and MSL . . . 27

E Experimental Details **28**
- E.1 Variational Multiple Shooting Layers . . . 28
- E.2 Optimal Limit Cycle Control via Multiple Shooting Layers . . . 28
- E.3 Neural Optimal Boudary Control of the Timoshenko Beam . . . 31
- E.4 Fast Neural CDEs for Time Series Classification . . . 33

A Proofs

A.1 Proof of Theorem 1

Theorem 1 (Quadratic fixed-point tracking)**.** *If f_θ is twice continuously differentiable in z then*

$$\|B^*_{p+1} - \bar{B}^*_p\|_2 \leq M\eta_p^2 \tag{3.4}$$

for some $M > 0$. $\bar{B}^*_p$ is the result of one Newton iteration applied to B^*_p.

Proof. For compactness, we neglect the dependence of $\gamma_\theta,\ g_\theta$ on z_0 and we write $\gamma(B,\theta) = \gamma_\theta(B, z_0), g(B,\theta) = g_\theta(B, z_0)$. Recalling that by definition of B^*_{p+1} it holds

$$g(B^*_{p+1}, \theta_{p+1}) = B^*_{p+1} - \gamma(B^*_{p+1}, \theta_{p+1}) = \mathbb{0}, \tag{A.1}$$

we write the 2-jet of the matching equation at B^*_p,

$$\begin{aligned} g(B^*_{p+1}, \theta_{p+1}) = g(B^*_p, \theta_{p+1}) &+ \mathsf{D}g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right] \\ &+ \frac{1}{2}\mathsf{D}^2 g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right]^{\otimes 2} + o(\|B^*_{p+1} - B^*_p\|_2^3) \end{aligned} \tag{A.2}$$

where $\mathsf{D}g,\ \mathsf{D}^2 g$ can be computed thanks to the assumptions on differentiability of f_θ. From the Newton iteration we have that

$$\begin{aligned} \bar{B}^*_p - B^*_p &= -[\mathsf{D}g(B^*_p, \theta_{p+1})]^{-1} g(B^*_p, \theta_{p+1}) \\ \Leftrightarrow \quad \mathbb{0} &= g(B^*_p, \theta_{p+1}) + \mathsf{D}g(B^*_p, \theta_{p+1})\left[\bar{B}^*_p - B^*_p\right] \end{aligned} \tag{A.3}$$

Using $\mathbb{0} = g(B^*_{p+1}, \theta_{p+1})$, we subtract (A.3) from (A.2) yielding

$$\begin{aligned} \mathbb{0} &= \cancel{g(B^*_p, \theta_{p+1})} + \mathsf{D}g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right] - \cancel{g(B^*_p, \theta_{p+1})} - \mathsf{D}g(B^*_p, \theta_{p+1})\left[\bar{B}^*_p - B^*_p\right] \\ &+ \frac{1}{2}\mathsf{D}^2 g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right]^{\otimes 2} + o(\|B^*_{p+1} - B^*_p\|_2^3) \\ &= \mathsf{D}g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right] - \mathsf{D}g(B^*_p, \theta_{p+1})\left[\bar{B}^*_p - B^*_p\right] \\ &+ \frac{1}{2}\mathsf{D}^2 g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right]^{\otimes 2} + o(\|B^*_{p+1} - B^*_p\|_2^3) \\ &= \mathsf{D}g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - \bar{B}^*_p\right] + \frac{1}{2}\mathsf{D}^2 g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right]^{\otimes 2} + o(\|B^*_{p+1} - B^*_p\|_2^3). \end{aligned} \tag{A.4}$$

Being the Jacobian of g_θ

$$\mathsf{D}g(B^*_p, \theta_{p+1}) = \mathbb{I}_N \otimes \mathbb{I}_{n_z} - \mathsf{D}\gamma(B^*_p, \theta_{p+1}) = \begin{bmatrix} \mathbb{I}_{n_z} & \times & \times & \times \\ -\mathsf{D}\phi_{\theta_{p+1},0}(b^*_{0,p}) & \mathbb{I}_{n_z} & \times & \times \\ \times & \ddots & \ddots & \times \\ \times & \times & -\mathsf{D}\phi_{\theta_{p+1},N-1}(b^*_{N-1,p}) & \mathbb{I}_{n_z} \end{bmatrix}$$

always invertible due to the nilpotency of $\mathsf{D}\gamma$, we can solve (A.4) in terms for $B^*_{p+1} - \bar{B}^*_p$, leading to

$$B^*_{p+1} - \bar{B}^*_p = \frac{1}{2}\left[\mathsf{D}g(B^*_p, \theta_{p+1})\right]^{-1}\mathsf{D}^2 g(B^*_p, \theta_{p+1})\left[B^*_{p+1} - B^*_p\right]^{\otimes 2} + o(\|B^*_{p+1} - B^*_p\|_2^3). \tag{A.5}$$

Taking the norm we have

$$\|B^*_{p+1} - \bar{B}^*_p\|_2 \leq \frac{1}{2}\|[\mathsf{D}g(B^*_p, \theta_{p+1})]^{-1}\|_2 \|\mathsf{D}^2 g(B^*_p, \theta_{p+1})\|_2 \|B^*_{p+1} - B^*_p\|_2^2. \tag{A.6}$$

Using

$$\|B^*_{p+1} - B^*_p\|_2 \leq \eta_p m_L^\theta m_\gamma^\theta$$

and $\|\mathsf{D}^2 g(B^*_p, \theta_{p+1})\|_2 = \|\mathsf{D}^2\gamma(B^*_p, \theta_{p+1})\|_2 \leq m^z_{\partial\gamma}$ we obtain

$$\|B^*_{p+1} - \bar{B}^*_p\|_2 \leq \frac{1}{2}\eta_p^2 (m_L^\theta m_\gamma^\theta)^2 m^z_{\partial\gamma}\|[\mathsf{D}g(B^*_p, \theta_{p+1})]^{-1}\|_2. \tag{A.7}$$

Since $R = \mathsf{D}\gamma(B_p^*, \theta_{p+1})$ is a nilpotent matrix then

$$[\mathsf{D}g(B_p^*, \theta_{p+1})]^{-1} = [\mathbb{I}_N \otimes \mathbb{I}_{n_z} - R]^{-1} = \mathbb{I}_N \otimes \mathbb{I}_{n_z} + \sum_{n=1}^{N} R^n$$

and

$$\|[\mathsf{D}g(B_p^*, \theta_{p+1})]^{-1}\|_2 \leq 1 + \sum_{n=1}^{N} \|R\|_2^n.$$

By Lipsichitz continuity of $\gamma(B_p^*, \theta_{p+1})$ we have that

$$\|R\|_2 \leq m_\gamma^z$$

and

$$\|[\mathsf{D}g(B_p^*, \theta_{p+1})]^{-1}\|_2 \leq 1 + \sum_{n=1}^{N} (m_\gamma^z)^n$$

By convergence of finite geometric series we obtain

$$\|[\mathsf{D}g(B_p^*, \theta_{p+1})]^{-1}\|_2 \leq \frac{1 - (m_\gamma^z)^{N+1}}{1 - m_\gamma^z}$$

The final bound on the tracking error norm thus becomes

$$\|B_{p+1}^* - \bar{B}_p^*\|_2 \leq \frac{1}{2}\eta_p^2 (m_L^\theta m_\gamma^\theta)^2 m_{\partial\gamma}^z \frac{1 - (m_\gamma^z)^{N+1}}{1 - m_\gamma^z}. \tag{A.8}$$

The proof of the theorem is completed by setting

$$M > \frac{1}{2}(m_L^\theta m_\gamma^\theta)^2 m_{\partial\gamma}^z \frac{1 - (m_\gamma^z)^{N+1}}{1 - m_\gamma^z}$$

□

A.2 Proof of Proposition 1

Proposition 1 (Forward Sensitivity (Khalil, 2002))**.** *Let $\phi_\theta(z, s, t)$ be the solution of (2.1). Then, $v(t) = \mathsf{D}\phi_\theta(z, s, t)$ satisfies the linear matrix–valued differential equations*

$$\dot{v}(t) = \mathsf{D}f_\theta(t, z(t))v(t), \quad v(s) = \mathbb{I}_{n_z} \quad \textit{where } \mathsf{D}f_\theta \textit{ denotes } \partial f_\theta / \partial z.$$

Proof. The following proof is adapted from (Khalil, 2002, Section 3.3). If $\phi_\theta(z_0, s, t)$ is a solution of (2.1) at time t starting from z_0 at time s, $s < t$; $s, t \in [t_0, t_N]$ then

$$\phi_\theta(z_0, s, t) = z(t) = z_0 + \int_s^t f_\theta(\tau, z(\tau))\mathrm{d}\tau \tag{A.9}$$

Differentiating under the integral sign w.r.t. z yields

$$\begin{aligned} \mathsf{D}\phi_\theta(z_0, s, t) = \frac{\mathrm{d}z(t)}{\mathrm{d}z_0} &= \frac{\mathrm{d}z_0}{\mathrm{d}z_0} + \int_s^t \frac{\partial f_\theta(\tau, z(\tau))}{\partial z(\tau)} \frac{\mathrm{d}z(\tau)}{\mathrm{d}z_0} \mathrm{d}\tau \\ &= \mathbb{I}_{n_z} + \int_s^t \frac{\partial f_\theta(\tau, z(\tau))}{\partial z(\tau)} \frac{\mathrm{d}z(\tau)}{\mathrm{d}z_0} \mathrm{d}\tau \end{aligned} \tag{A.10}$$

We denote $\mathsf{D}f_\theta(t, z(t)) = \partial f_\theta(\tau, z(\tau))/\partial z(\tau)$ and we notice that $\mathrm{d}z(\tau)/\mathrm{d}z_0$ is the flow Jacobian $\mathsf{D}\phi_\theta(z_0, s, \tau)$ at time $\tau \in [s, t]$. Then, the function $v : [s, t] \to \mathbb{R}^{n_z \times n_z}$; $\tau \mapsto \mathsf{D}\phi_\theta(z_0, s, \tau)$ satisfies

$$v(t) = \mathbb{I}_{n_z} + \int_s^t \mathsf{D}f(\tau, z(\tau))v(\tau)\mathrm{d}\tau$$

or, in differential form, v satisfies the IVP

$$\dot{v}(\tau) = \mathsf{D}f(\tau, z(\tau))v(\tau), \quad v(s) = \mathbb{I}_{n_z}.$$

□

B Additional Theoretical Results

B.1 Finite–Step Convergence

We discuss more rigorously the intuitions on the finite–step convergence of direct Newton methods introduced in the main text. The following results are thoroughly detailed in (Gander, 2018). We recall that, by assuming that the first shooting parameter is correctly initialized to z_0 and the numerical integration is *exact* (we can perfectly retrieve the sub-flows $\phi_{\theta,n}$), the shooting parameters b_n^k coincides with the exact solution of (2.1) from Newton iteraion $k = n$ onward. Formally,

Proposition 2 (Finite–step convergence)**.** *If $b_0^0 = z_0$, then solution of the Newton iteration (3.2) are such that*

$$k \geq n \;\Rightarrow\; b_n^k = \phi_\theta(z_0, t_0, t_n). \tag{B.1}$$

Proof. The proof is obtained by induction on the shooting parameter index n (*time* direction) and follows from (Gander, 2018, Theorem 2.3).

i. (`base case:` $n = 0$) For $n = 0$, $b_0^0 = z_0$ by assumption. Moreover, the iteration (3.2) yields $b_0^k = b_0^{k+1} = z_0$ for all naturals k.

ii. (`induction step:` $n \to n+1$) Suppose that

$$k \geq n \;\Rightarrow\; b_n^k = \phi_\theta(z_0, t_0, t_n).$$

We need to show that

$$k+1 \geq n+1 \;\Rightarrow\; b_{n+1}^{k+1} = \phi_\theta(z_0, t_0, t_{n+1}),$$

to conclude the proof by induction. We notice that if we increase k to $k+1$, then $k+1$ is still greater than n yielding $b_n^{k+1} = b_n^k = \phi_\theta(z_0, t_0, t_n)$. Using (3.2), we have

$$\begin{aligned} b_{n+1}^{k+1} &= \phi_{\theta,n}(b_n^k) + \mathsf{D}\phi_{\theta,n}(b_n^k)\left(b_n^{k+1} - b_n^k\right) \\ &= \phi_{\theta,n}(\phi_\theta(z_0, t_0, t_n)) + 0 && \text{by induction hypothesis } b_n^k = b_n^{k+1} = \phi_\theta(z_0, t_0, t_n); \\ &= \phi_\theta(z_0, t_0, t_{n+1}) && \text{by the flow property of ODE solutions;} \end{aligned}$$

where the induction hypothesis has been used thanks to the fact that $k+1 \geq n+1 \Rightarrow k \geq n$.

□

The above result can be also extended to the zeroth–order (*parareal*) method as follows.

Proposition 3 (Finite–step convergence w/ zeroth–order Jacobian approximation)**.** *If $b_0^0 = z_0$, then solution of the approximate Newton iteration*

$$b_{n+1}^{k+1} = \phi_{\theta,n}(b_n^k) + \psi_{\theta,n}(b_n^{k+1}) - \psi_{\theta,n}(b_n^k) \tag{B.2}$$

are such that

$$k \geq n \;\Rightarrow\; b_n^k = \phi_\theta(z_0, t_0, t_n). \tag{B.3}$$

Proof. The proof is identical to the one Proposition 2 where (B.2) is used in the induction step and noticing that the correction term $\psi_{\theta,n}(b_n^{k+1}) - \psi_{\theta,n}(b_n^k)$ nullifies for $k > n$ by induction hypothesis $b_n^k = b_n^{k+1} = \phi_\theta(z_0, t_0, t_n)$. □

Even though Proposition 2 and Proposition 3 show that the direct Newton method (and its zeroth-order approximation) will always converge to the exact solution of (2.1), full convergence after N iterations is completely useless from a practical perspective. If we suppose to use a *fine solver* $\tilde{\phi}_{\theta,n}$ to obtain in parallel accurate numerical approximations of the sub–flows $\phi_{\theta,n}$ and we iterate (3.2) N times, we will also have executed the parallel integration N times. Thus, one could also just have applied the same fine solver sequentially across the N boundary points t_n with one processing thread and obtain the same result. For this reason, we believe that tracking Theorem 1 is a key result to obtain large speedups in the machine learning applications of MSLs.

B.2 Flows Sensitivities $\mathsf{D}\phi_{\theta,n}$

The most computationally demanding stage of the MSL inference is without any doubts the correction term

$$\mathsf{D}\phi_{\theta,n}(b_n^k)\left(b_n^{k+1} - b_n^k\right)$$

of the direct Newton iteration (3.2). In this paper, we propose to either use the forward sensitivity approach of Proposition 1 or to rely on the zeroth–order approximation of parareal. Moreover, we discouraged the use of both reverse–mode AD and backward adjoint sensitivities to compute the full Jacobians $\mathsf{D}\phi_{\theta,n}$ due to their higher memory or computational cost.

Sensitivities with $N-k$ `jvp`s A common feature among the aforementioned approaches (but the parareal) is that all $\mathsf{D}\phi_{\theta,n}$ can be computed in parallel at the beginning of each Newton iteration with a single call of the sensitivity routine. An alternative *sequential* approach relies on computing $\mathsf{D}\phi_{\theta,n}(b_n^k)\left(b_n^{k+1} - b_n^k\right)$ directly as a `jvp` during each step of (3.2). This method avoids the computation of the full Jacobians at cost of having to call the `jvp` routine $N-k$ times at each Newton iteration. In such case the only parallel operation performed is the integration of the sub–flows $\phi_{\theta,n}$. Nonetheless, we believe that this direction is worth to be explored in future works.

B.3 Backward Model of Multiple Shooting Layers

We show how MSLs can be trained via standard gradient descent techniques where gradients can be either computed by back–propagating through the operations of the forward pass (parallel/memory intensive) or by using the convergence property of direct Newton method and directly apply a interpolated adjoint routine[5] (sequential/memory efficient). Although we believe that these two approaches to backpropagation are sufficient within the scope of this manuscript as they allow for substantial computational speedups and robustness, we hereby report further theoretical considerations on the backward pass of MSLs. A thorough algorithmic and experimental analysis of the following content is a promising research direction for future work.

Implicit differentiation of MSLs As repeatedly pointed out throughout the paper, MSLs are implicit models and satisfy the implicit relation

$$B^* \;:\; B^* = \gamma_\theta(B^*, z_0). \tag{B.4}$$

It thus make sense to interpret the backward pass of MSLs in an *implicit* sense. In particular, implicit differentiation of the relation (B.4) at B^* leads to the following loss gradients.

Theorem 2 (Implicit Gradients)**.** *Consider a smooth loss function L_θ. It holds*

$$\frac{\mathrm{d}L_\theta}{\mathrm{d}\theta} = \frac{\partial L_\theta}{\partial \theta} + \frac{\partial L_\theta}{\partial \ell_y}\frac{\partial \ell_y}{\partial B^*}\left[\mathbb{I}_{n_z}\otimes \mathbb{I}_N - \mathsf{D}\gamma_\theta(B^*)\right]^{-1}\frac{\partial \gamma_\theta}{\partial \theta} \tag{B.5}$$

where $\mathsf{D}\gamma_\theta(B^*, z_0) \in \mathbb{R}^{Nn_z \times Nn_z}$ is the Jacobian of γ_θ computed at B^*.

Proof. By application of the chain rule to the MSL forward model (2.3) we obtain

$$\frac{\mathrm{d}L_\theta}{\mathrm{d}\theta} = \frac{\partial L_\theta}{\partial \theta} + \frac{\partial L_\theta}{\partial \ell_y}\frac{\partial \ell_y}{\partial B^*}\frac{\mathrm{d}B^*}{\mathrm{d}\theta}.$$

With

$$\begin{aligned} g_\theta(B^*) = \mathbb{0} \;\Rightarrow\; & B^* - \gamma(B^*, z_0) = \mathbb{0} \\ \Leftrightarrow\; & \frac{\partial g_\theta(B^*)}{\partial \theta} + \left[\mathbb{I}_{n_z}\otimes \mathbb{I}_N - \mathsf{D}\gamma_\theta(B^*)\right]\frac{\mathrm{d}B^*}{\mathrm{d}\theta} = 0 \\ \Leftrightarrow\; & \frac{\mathrm{d}B^*}{\mathrm{d}\theta} = \left[\mathbb{I}_{n_z}\otimes \mathbb{I}_N - \mathsf{D}\gamma_\theta(B^*)\right]^{-1}\frac{\partial \gamma_\theta}{\partial \theta}. \end{aligned}$$

Thus,

$$\frac{\mathrm{d}L_\theta}{\mathrm{d}\theta} = \frac{\partial L_\theta}{\partial \theta}\frac{\partial L_\theta}{\partial \ell_y}\frac{\partial \ell_y}{\partial B^*}\left[\mathbb{I}_{n_z}\otimes \mathbb{I}_N - \mathsf{D}\gamma_\theta(B^*)\right]^{-1}\frac{\partial \gamma_\theta}{\partial \theta}$$

[5] Implementation details are provided in Appendix C.4

where the Jacobian $\mathsf{D}\gamma_\theta(B^*)$ is computed as

$$\mathsf{D}\gamma_\theta(B^*) = \begin{bmatrix} \times & \times & \times & \times & \times \\ \mathsf{D}\phi_{\theta,0}(b_0^*) & \times & \times & \times & \times \\ \times & \mathsf{D}\phi_{\theta,1}(b_1^*) & \times & \times & \times \\ & & & & \\ \times & \times & \ddots & \ddots & \times \\ & & & & \\ \times & \times & \times & \mathsf{D}\phi_{\theta,N-1}(b_{N-1}^*) & \times \end{bmatrix}$$

and

$$\frac{\partial \gamma_\theta}{\partial \theta} = \begin{bmatrix} \mathbb{0}_{n_z} \\ \dfrac{\partial \phi_{\theta,0}(b_0)}{\partial \theta} \\ \vdots \\ \dfrac{\partial \phi_{\theta,N-1}(b_{N-1})}{\partial \theta} \end{bmatrix}$$

□

The implicit differentiation routine suggested by Theorem 2 presents two terms which appear to be very demanding both memory and computation–wise:

(i) The inverse Jacobian $[\mathbb{I}_{n_z} \otimes \mathbb{I}_N - \mathsf{D}\gamma_\theta(B^*)]^{-1}$ of the implicit relation;

(ii) The sub–flows sensitivities to the model parameters θ.

In order to retrieve (i) in standard *Deep Equilibrium Models* Bai et al. (2019), one should either compute the full–Jacobian at the fixed point via AD and invert it or "recycle" its low–rank approximation from the Quasi–Newton method of the forward pass. In the case of MSLs we can take advantage of the special structure of the implicit relation to obtain the exact Jacobian inverse in a computationally efficient manner. In fact, if `fw-sensitivity` has been used in the forward pass to compute B^*, then the sensitivities of the sub–flows computed at the last step K of the Newton iteration $\mathsf{D}\phi_{\theta,n}(b_n^K)$ can be stored and re–used to construct the Jacobian $\mathsf{D}\gamma_\theta$. Further, due to the nilpotency of $\mathsf{D}\gamma_\theta$ the inverse of the total Jacobian can be retrieved in closed form by the finite matrix power series

$$[\mathbb{I}_{n_z} \otimes \mathbb{I}_N - \mathsf{D}\gamma_\theta(B^*)]^{-1} = \mathbb{I}_{n_z} \otimes \mathbb{I}_N + \sum_{n=1}^{N} [\mathsf{D}\gamma_\theta(B^*)]^n.$$

Finally, (ii) may be indirectly computed with a single `vjp`

$$v^\top \frac{\partial \gamma_\theta(B^*, z_0)}{\partial \theta}$$

with $v^\top$ being a 1 by Nn_θ row vector defined as

$$v^\top = \frac{\partial L_\theta}{\partial \ell_y} \frac{\partial \ell_y}{\partial B^*} \left[\mathbb{I}_{n_z} \otimes \mathbb{I}_N + \sum_{n=1}^{N} [\mathsf{D}\gamma_\theta(B^*)]^n \right]$$

leading to the implicit cost gradient with a single call of the AD.

C Additional Details on the Realization of MSLs

Effective time–parallelization of MSLs requires implementation of specialized computational primitives. In example, forward sensitivity methods benefit from a breakdown of matrix–Jacobian products into a `vmapped` vector–Jacobian products. Here, we provide code for several key methods and classes which have been incorporated in the `torchdyn` (Poli et al., 2020b) library for neural differential equations and implicit models.

C.1 Software Implementation of Forward Sensitivity

Forward sensitivity analysis is extensively used in MSLs to compute $\mathrm{d}\phi_{\theta,n}/\mathrm{d}b_n$ in parallel for each shooting parameter $b_n,\ n = 0, \dots, N-1$. We showcase how this can be efficiently implemented in `Pytorch` (Paszke et al., 2019). Although the implementation fully accommodates batches n_b of data, i.e. each b_n is a n_b by n_z matrix, we will limit the algorithmic analysis to the unitary batch dimension. The forward sensitivity algorithm aims at computing the solution of the differential equation

$$\begin{pmatrix} \dot{z}_n(t) \\ \dot{v}_n(t) \end{pmatrix} = \begin{pmatrix} f_\theta(t, z_n(t)) \\ \mathsf{D}f_\theta(t, z_n(t))v_n(t) \end{pmatrix}, \quad \begin{pmatrix} \dot{z}_n(0) \\ \dot{v}_n(0) \end{pmatrix} = \begin{pmatrix} b_n \\ \mathbb{I}_{n_z} \end{pmatrix}, \quad t \in [t_n, t_{n+1}] = \mathcal{T}_n$$

for all n, to return $\phi_{\theta,n} = z_n(t_{n+1})$ and $\mathrm{d}\phi_{\theta,n}/\mathrm{d}b_n = v_n(t_{n+1})$. Let Z and V be the tuples containing all z_n and v_n,

$$\begin{aligned} Z &= (z_0, z_1, \dots, z_{N-1}) \in \overbrace{\mathbb{R}^{n_z} \times \cdots \times \mathbb{R}^{n_z}}^{N} \equiv \mathbb{R}^{N \times n_z} \\ V &= (v_0, v_1, \cdots, v_{N-1}) \in \underbrace{\mathbb{R}^{n_z \times n_z} \times \cdots \times \mathbb{R}^{n_z \times n_z}}_{N} \equiv \mathbb{R}^{N \times n_z \times n_z}. \end{aligned}$$

Given a tuple of time instants $T = (\tau^0, \tau^1, \dots, \tau^{N-1}) \in [t_0, t_1] \times [t_1, t_1] \times [t_{N-1}, t_N] \subset \mathbb{R}^{N \times 1}$, f_θ can evaluated in parallel on T and Z as the number N of shooting parameters b_n and sub–intervals $[t_n, t_{n+1}]$ only accounts for a *batch* dimension. From a software perspective, we can obtain

$$F(T, Z) = (f_\theta(\tau^0, z_0), \dots, f_\theta(\tau^{N-1}, z_{N-1}))$$

in a single call of the function f_θ, e.g. an instantiated `PyTorch`'s `nn.Module` object. Conversely, when attempting to compute "$\frac{\partial F}{\partial Z} V$" in parallel, additional software infrastructure is necessary. The main obstacle is that each Jacobian–matrix product (`jmp`)

$$\frac{\partial f_\theta(t, z_n(t))}{\partial z_n} v_n(t)$$

generally requires n_z `autograd` calls. Following the `Jax`'s (Bradbury et al., 2018) approach, we make use of a `PyTorch` implementation[6] of *vectorizing maps* (`vmaps`) to distribute the computation of the individual Jacobian–vectors products (in *batch* for each $n = 0, \dots, N-1$) and compose the `jmp` row–by–row or column–by–column. In particular we define the `vmapped_jmp` function

```
    def vmapped_jmp(y, z, v):
        """ Parallel computation of matrix Jacobian products with vmap
        """
        def get_jvp(v):
            return torch.autograd.grad(y, z, v, retain_graph=True)[0]
        return vmap(get_jvp, in_dims=2, out_dims=2)(v)
```

The forward sensitivity can then be computed as follows

```
class ForwardSensitivity(nn.Module):
    "Forward sensitivity for ODEs. Integrates the ODE returning the state and
    ↪  forward sensitivity"
    def __init__(self, f):
        super().__init__()
```

[6] see https://pytorch.org/docs/master/generated/torch.vmap.html

```python
        self.f = f

    def forward(self, z0, t_span, odeint_func, solver='rk4', atol=1e-5, rtol=1e-5):
        I = eye(z0.shape[-1]).to(z0)
        # handle regular `batch, dim` case as well as `seq_dim, batch, dim`
        v0 = I.repeat(z0.shape[0], 1, 1) if len(z0.shape) < 3 else I.repeat(*z0.shape[:2], 1, 1)

        self.z_shape, self.v_shape = z0.shape, v0.shape

        zv0 = self._ravel_state(z0, v0)
        zvT = odeint_func(self._sensitivity_dynamics, zv0, t_span,
                          solver=solver)[1]
        zT, vT = self._unravel_state(zvT)
        return zT, vT

    def _sensitivity_dynamics(self, t, zv):
        z, v = self._unravel_state(zv)
        # compute vector field
        dz = self.f(t, z.requires_grad_(True))
        # compute fw sensitivity via mjp
        dv = self._mjp(dz, z, v)
        return self._ravel_state(dz, dv)

    def _jmp(self, f, z, v):
        """Parallel computation of matrix jacobian products with vmap
        """
        def get_vjp(v):
            return torch.autograd.grad(f, z, v, retain_graph=True)[0]
        return vmap(get_vjp, in_dims=2, out_dims=2)(v)

    def _ravel_state(self, z, v):
        v = v.reshape(*z.shape[:-1], -1)
        zv = torch.cat([z, v], -1)
        return zv

    def _unravel_state(self, zv):
        z, v = zv[...,:self.z_shape[-1]], zv[...,self.z_shape[-1]:]
        v = v.reshape(*z.shape, self.z_shape[-1])
        return z, v
```

where `odeint` is ODE solver utility of the `torchdyn` (Poli et al., 2020b) library.

C.2 Implementation of Direct Newton Method

Forward sensitivity Newton (`fw sensitivity`) MSL is a variant of the proposed model class which obtains the quantities $\mathrm{D}\phi_{\theta,n}$ directly by augmenting the time–parallelized forward dynamics through the `ForwardSensitivity` class previously detailed. During the evaluation of the *advancement* function $\gamma_\theta(B, z_0) = (z_0, \phi_\theta(b_0, t_0, t_1), \dots, \phi_\theta(b_{N-1}, t_{N-1}, t_N))$, `ForwardSensitivity` maximizes reutilization of vector field f_θ evaluations by leveraging the results to advance both *standard* as well as sensitivity dynamics. This provides an overall reduction in the potentially expensive evaluation of the neural network f_θ, compared to *parareal* (zeroth–order MSL). We hereby report the `PyTorch` implementation for both the `fw sensitivity` MSL and zeroth–order MSL methods

```python
class MSForward(MShootingSolverTemplate):
    """Multiple shooting solver using forward sensitivity analysis on the matching conditions of shooting parameters"""
    def __init__(self, coarse_method='euler', fine_method='rk4'):
        super().__init__(coarse_method, fine_method)
        self.fsens = None

```

```python
    def root_solve(self, odeint_func, f, x, t_span, B, fine_steps, maxiter):
        if self.fsens is None:
            self.fsens = ForwardSensitivity(f)

        dt, n_subinterv = t_span[1] - t_span[0], len(t_span)
        sub_t_span = torch.linspace(0, dt, fine_steps).to(x)
        i = 0
        while i <= maxiter:
            i += 1
            with torch.set_grad_enabled(True):
                B_fine, V_fine = self.fsens(B[i-1:], sub_t_span, odeint_func=odeint_func,
                                            solver=self.fine_method)
            B_fine, V_fine = B_fine[-1], V_fine[-1]
            B_out = torch.zeros_like(B)
            B_out[:i] = B[:i]
            B_in = B[i-1]
            for m in range(i, n_subinterv):
                B_in = B_fine[m-i] + torch.einsum('bij, bj -> bi', V_fine[m-i], B_in - B[m-1])
                B_out[m] = B_in
            B = B_out
        return B
```

```python
class MSZero(MShootingSolverTemplate):
    def __init__(self, coarse_method='euler', fine_method='rk4'):
        """Multiple shooting solver using Parareal updates (zero-order approximation of the Jacobian)
        """
        super().__init__(coarse_method, fine_method)
    def root_solve(self, odeint_func, f, x, t_span, B, fine_steps, maxiter):
        dt, n_subinterv = t_span[1] - t_span[0], len(t_span)
        sub_t_span = torch.linspace(0, dt, fine_steps).to(x)
        i = 0
        while i <= maxiter:
            i += 1
            B_coarse = odeint_func(f, B[i-1:], sub_t_span, solver=self.coarse_method)[1][-1]
            B_fine = odeint_func(f, B[i-1:], sub_t_span, solver=self.fine_method)[1][-1]
            B_out = torch.zeros_like(B)
            B_out[:i] = B[:i]
            B_in = B[i-1]
            for m in range(i, n_subinterv):
                B_in = odeint_func(f, B_in, sub_t_span, solver=self.coarse_method)[1][-1]
                B_in = B_in - B_coarse[m-i] + B_fine[m-i]
                B_out[m] = B_in
            B = B_out
        return B
```

In the above, we employ the finite–step convergence property of Newton MSL iterations to avoid redundant computation. More specifically, at iteration k we do not advance shooting parameters $b_n, \ n < k$ by slicing the tensor B during γ_θ evaluations. Similarly, updates in the form (3.2) are not performed for shooting parameters already at convergence.

C.3 Alternative Approaches to MSL Inference

On Newton and Quasi-Newton methods for MSL The root–finding problem arising in MSLs can also be approached by standard application of Newton or Quasi–Newton algorithms. Although Quasi–Newton algorithms can provide improved computational efficiency by maintaining a low–rank approximation of the Jacobian $Dg_\theta(B, z_0)$ rather than computing it from scratch every iteration, this

advantage does not translate well to the MSL case. Popular examples include, e.g., the Broyden family Broyden (1965) employed in *Deep Equilibrium Models* (DEQs) Bai et al. (2019). As discussed in the main text, thanks to the special structure of the Jacobian of the MSL problem, the direct Newton algorithm (3.2) can be applied without computation and inversion of the full Jacobian.

Root finding via gradient descent A completely different approach to solve the implicit forward MSL pass (2.3) is to tackle the root–finding via some gradient–descent (GD) method minimizing $\|g_\theta(B)\|_2^2$, i.e.

$$B^* = \operatorname{argmin} \frac{1}{2}\|g_\theta(B)\|_2^2.$$

In the case of MSL, all GD solutions (i.e. minima of $\|g_\theta(B)\|_2^2$) are the same of the the root finding ones. This can be intuitively checked by inspecting the zeros of the gradient, i.e.

$$\nabla_B \frac{1}{2}\|g_\theta(B)\|_2^2 = Dg_\theta|_B g_\theta(B)$$

and, since $Dg_\theta|_B$ is nonsingular for all B,

$$\forall \tilde{B}^* \ : \ \nabla_B \frac{1}{2}\|g_\theta(\tilde{B}^*)\|_2^2 = \mathbb{0} \Rightarrow g_\theta(\tilde{B}^*) = \mathbb{0}.$$

C.4 Implementation of Backward Interpolated Adjoint

We provide pseudo–code for our implementation of MSLs with backward gradients obtained via interpolated adjoints. The implementation relies on cubic interpolation utilities provided by `torchcde` Kidger et al. (2020b). Interpolation is used to obtain values of $z(t)$ without a full backsolve from $z(T)$.

```
def _gather_odefunc_interp_adjoint(vf, vf_params, solver, atol, rtol, interpolator, solver_adjoint, atol_adjoint, rtol_adjoint, integral_loss, problem_type, maxiter=4, fine_steps=4):
    "Prepares definition of autograd.Function for interpolated adjoint sensitivity analysis of the above `ODEProblem`"
    class _ODEProblemFunc(Function):
        @staticmethod
        def forward(ctx, vf_params, x, t_span, B=None):
            t_sol, sol = generic_odeint(problem_type, vf, x, t_span, solver, atol, rtol, interpolator, B,
                                        True, maxiter, fine_steps)
            ctx.save_for_backward(sol, t_span, t_sol)
            return t_sol, sol

        @staticmethod
        def backward(ctx, *grad_output):
            sol, t_span, t_sol = ctx.saved_tensors
            vf_params = torch.cat([p.contiguous().flatten() for p in vf.parameters()])

            # initialize adjoint state
            xT, λT, μT = sol[-1], grad_output[-1][-1], torch.zeros_like(vf_params)
            λT_nel, μT_nel = λT.numel(), μT.numel()
            xT_shape, λT_shape, μT_shape = xT.shape, λT.shape, μT.shape
            A = torch.cat([λT.flatten(), μT.flatten()])

            spline_coeffs = natural_cubic_coeffs(x=sol.permute(1, 0, 2).detach(), t=t_sol)
            x_spline = CubicSpline(coeffs=spline_coeffs, t=t_sol)

            # define adjoint dynamics
            def adjoint_dynamics(t, A):
                if len(t.shape) > 0: t = t[0]
                x = x_spline.evaluate(t).requires_grad_(True)
```

```
                t = t.requires_grad_(True)
                λ, μ = A[:λT_nel], A[-μT_nel:]
                λ, μ = λ.reshape(λT.shape), μ.reshape(μT.shape)
                with torch.set_grad_enabled(True):
                    dx = vf(t, x)
                    dλ, dt, *dμ = tuple(grad(dx, (x, t) + tuple(vf.parameters()), -λ,
                                        allow_unused=True, retain_graph=False))

                    if integral_loss:
                        dg = torch.autograd.grad(integral_loss(t, x).sum(), x, allow_unused=True, retain_graph=True)[0]
                        dλ = dλ - dg

                    dμ = torch.cat([el.flatten() if el is not None else torch.zeros(1)
                                    for el in dμ], dim=-1)
                return torch.cat([dλ.flatten(), dμ.flatten()])

            # solve the adjoint equation
            n_elements = (λT_nel, μT_nel)
            for i in range(len(t_span) - 1, 0, -1):
                t_adj_sol, A = odeint(adjoint_dynamics, A, t_span[i - 1:i + 1].flip(0), solver, atol=atol, rtol=rtol)
                # prepare adjoint state for next interval
                A = torch.cat([A[-1, :λT_nel], A[-1, -μT_nel:]])
                A[:λT_nel] += grad_output[-1][i - 1].flatten()

            λ, μ = A[:λT_nel], A[-μT_nel:]
            λ, μ = λ.reshape(λT.shape), μ.reshape(μT.shape)
            return (μ, λ, None, None, None)

    return _ODEProblemFunc
```

C.5 Broader Impact

Differential equations are the language of science and engineering. As methods (Jia and Benson, 2019) and software frameworks (Rackauckas et al., 2019; Li et al., 2020; Poli et al., 2020b) are improved, yielding performance gains or speedups (Poli et al., 2020a; Kidger et al., 2020a; Pal et al., 2021), the range of applicability of neural differential equations is extended to more complex and larger scale problems. As with other techniques designed to reduce overall training time, we expect a net positive environment impact from the adoption of MSLs in the framework.

Application domains for MSLs include environments with real–time constraints, for example control and high frequency time series prediction. Shorter inference wall–clock and training iteration times should yield more robust models that can, in example, be retrained online at higher frequencies as more data is collected.

D Neural Network Control of the Timoshenko Beam

In this section we derive the dynamic model of the Timoshenko beam, the boundary control and the structure–preserving discretization of the problem.

D.1 Port–Based Modeling of the Timoshenko Beam

Linear distributed port-Hamiltonian systems (Macchelli et al., 2004) in one-dimensional domains take the form

$$\frac{\partial z}{\partial t}(x,t) = P_1 \frac{\partial}{\partial x}(\mathcal{L}(x)z(x,t)) + (P_0 - G_0)\mathcal{L}(x)z(x,t) \tag{D.1}$$

with distributed state $z \in \mathbb{R}^{n_z}$ and spatial variable $x \in [a,b]$. Moreover, $P_1 = P_1^\top$ and invertible, $P_0 = -P_0^\top,\ G_0 = G_0^\top \geq 0$, and $\mathcal{L}(\cdot)$ is a bounded and Lipschitz continuous matrix-valued function such that $\mathcal{L}(x) = \mathcal{L}^\top(x)$ and $\mathcal{L}(x) \geq \kappa I$, with $\kappa > 0$, $\forall x \in [a,b]$. Given the Hamiltonian (total energy) of the system

$$H = \|z\|^2_{\mathcal{L}} = \langle z,\, \mathcal{L}z\rangle_{L^2}\,,$$

its variational derivative corresponds to the term $\mathcal{L}(x)z(x,t)$:

$$\frac{\delta H}{\delta z}(z(x,t),x) = \mathcal{L}(x)z(x,t)$$

A particular example from continuum mechanics that falls within the systems class (D.1) is the Timoshenko beam with no dissipation (Macchelli and Melchiorri, 2004). This system takes the following form:

$$\frac{\partial}{\partial t}\begin{pmatrix} p_t \\ p_r \\ \varepsilon_r \\ \varepsilon_t \end{pmatrix} = \begin{bmatrix} 0 & 0 & 0 & \partial_z \\ 0 & 0 & \partial_z & 1 \\ 0 & \partial_z & 0 & 0 \\ \partial_z & -1 & 0 & 0 \end{bmatrix} \left(\begin{bmatrix} (\rho A)^{-1} & 0 & 0 & 0 \\ 0 & (I_\rho)^{-1} & 0 & 0 \\ 0 & 0 & EI & 0 \\ 0 & 0 & 0 & K_{\text{sh}}GA \end{bmatrix} \begin{pmatrix} p_t \\ p_r \\ \varepsilon_r \\ \varepsilon_t \end{pmatrix} \right), \tag{D.2}$$

where ρ is the mass density, A is the cross section area, I_ρ is the rotational inertia, E is the Young modulus, I the cross section moment of area, $K_{\text{sh}} = 5/6$ is the shear correction factor and G the shear modulus.

For this examples the matrices $P_0,\ G_0,\ P_1,\ \mathcal{L}$ and are given by

$$\begin{gathered} P_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}, \quad P_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad G_0 = \mathbb{0}_{4\times 4}, \\ \mathcal{L}(z) = \begin{bmatrix} (\rho A)^{-1} & 0 & 0 & 0 \\ 0 & (I_\rho)^{-1} & 0 & 0 \\ 0 & 0 & EI & 0 \\ 0 & 0 & 0 & K_{\text{sh}}GA \end{bmatrix}. \end{gathered} \tag{D.3}$$

We investigate the boundary control of the Timoshenko beam model. As control input, the following selection is made (cantilever-free beam)

$$\pi_\partial = \begin{pmatrix} EI\varepsilon_r(b,t) \\ K_{\text{sh}}GA\varepsilon_t(b,t) \\ (\rho A)^{-1}p_t(a,t) \\ (I_\rho)^{-1}p_r(a,t) \end{pmatrix} \tag{D.4}$$

Notice that the control expression can be rewritten compactly as follows

$$\pi_\partial = \mathcal{B}_\partial \begin{pmatrix} \mathcal{L}x(b,t) \\ \mathcal{L}x(a,t) \end{pmatrix}, \quad \text{where} \quad \mathcal{B}_\partial = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \tag{D.5}$$

To put system (D.2) in impedance form, the outputs are selected as follows

$$y_\partial = \begin{pmatrix} (\rho A)^{-1} p_t(b,t) \\ (I_\rho)^{-1} p_r(b,t) \\ -EI\varepsilon_r(a,t) \\ -K_{\text{sh}} GA \varepsilon_t(a,t) \end{pmatrix} \tag{D.6}$$

This is compactly written as

$$y_\partial = \mathcal{C}_\partial \begin{pmatrix} \mathcal{L}x(b,t) \\ \mathcal{L}x(a,t) \end{pmatrix}, \quad \text{where} \quad \mathcal{C}_\partial = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}. \tag{D.7}$$

With this selection of inputs and outputs, the rate of the Hamiltonian is readily computed

$$\dot{H} = \pi_\partial^\top y_\partial. \tag{D.8}$$

Within the purpose of this paper we restrict to the case of *a cantilever beam undergoing a control action at the free end*

$$\pi_\partial = \begin{pmatrix} \pi_{\partial,1} \\ \pi_{\partial,2} \\ 0 \\ 0 \end{pmatrix}, \tag{D.9}$$

where $\pi_{\partial,1}$ is the control torque and $u_{\partial,2}$ is the control force.

D.2 Discretization of the Problem

To discretize system (D.2), since the problem is linear, one can either rely on a energy formulation or a co-energy one. Given the coenergy variables

$$\begin{pmatrix} v_t \\ v_r \\ \sigma_r \\ \sigma_t \end{pmatrix} = \begin{bmatrix} (\rho A)^{-1} & 0 & 0 & 0 \\ 0 & I_\rho^{-1} & 0 & 0 \\ 0 & 0 & EI & 0 \\ 0 & 0 & 0 & K_{\text{sh}} GA \end{bmatrix} \begin{pmatrix} p_t \\ p_r \\ \varepsilon_r \\ \varepsilon_t \end{pmatrix} \tag{D.10}$$

and introducing the bending and shear compliance

$$C_b = (EI)^{-1}, \qquad C_s = (K_{\text{sh}} GA)^{-1}, \tag{D.11}$$

system (D.2) is rewritten as

$$\begin{bmatrix} \rho A & 0 & 0 & 0 \\ 0 & I_\rho & 0 & 0 \\ 0 & 0 & C_b & 0 \\ 0 & 0 & 0 & C_s \end{bmatrix} \frac{\partial}{\partial t} \begin{pmatrix} v_t \\ v_r \\ \sigma_r \\ \sigma_t \end{pmatrix} = \begin{bmatrix} 0 & 0 & 0 & \partial_z \\ 0 & 0 & \partial_z & 1 \\ 0 & \partial_z & 0 & 0 \\ \partial_z & -1 & 0 & 0 \end{bmatrix} \begin{pmatrix} v_t \\ v_r \\ \sigma_r \\ \sigma_t \end{pmatrix}, \tag{D.12}$$

A weak form suitable for mixed finite elements is readily obtained by considering its weak form using test functions $(\mu_t, \mu_r, \nu_r, \nu_t)$ and the integration by parts applied to the first two lines. In this formulation, the Dirichlet boundary condition have to be incorporated as essential boundary conditions

$$\begin{aligned} \langle \mu_t,\, \rho A \partial_t v_t \rangle_\Omega &= -\langle \partial_z \mu_t,\, \sigma_t \rangle_\Omega + \mu_t(b)\pi_{\partial,2}, \\ \langle \mu_r,\, I_\rho \partial_t v_r \rangle_\Omega &= -\langle \partial_z \mu_r,\, \sigma_r \rangle_\Omega + \langle \mu_r,\, \sigma_t \rangle_\Omega + \mu_r(b)\pi_{\partial,1}, \\ \langle \nu_r,\, C_b \partial_t \sigma_r \rangle_\Omega &= \langle \nu_r,\, \partial_z v_r \rangle_\Omega, \\ \langle \nu_t,\, C_s \partial_t \sigma_t \rangle_\Omega &= \langle \nu_t,\, \partial_z v_t \rangle_\Omega - \langle \nu_t,\, v_r \rangle_\Omega, \end{aligned} \tag{D.13}$$

where $\Omega = [a,b]$ and $\langle f,\, g \rangle_\Omega = \int_a^b fg \,\mathrm{d}x$. Introducing the following Galerkin basis functions

$$\begin{aligned} \mu_t &= \sum_{i=1}^{N_{v_t}} \varphi_{v_t}^i \mu_t^i, & \mu_r &= \sum_{i=1}^{N_{v_r}} \varphi_{v_r}^i \mu_r^i, & \nu_r &= \sum_{i=1}^{N_{\sigma_r}} \varphi_{\sigma_r}^i \nu_r^i, & \nu_t &= \sum_{i=1}^{N_{\sigma_t}} \varphi_{\sigma_t}^i \nu_r^i, \\ v_t &= \sum_{i=1}^{N_{v_t}} \varphi_{v_t}^i v_t^i, & v_r &= \sum_{i=1}^{N_{v_r}} \varphi_{v_r}^i v_r^i, & \sigma_r &= \sum_{i=1}^{N_{\sigma_r}} \varphi_{\sigma_r}^i \sigma_r^i, & \sigma_t &= \sum_{i=1}^{N_{\sigma_t}} \varphi_{\sigma_t}^i \sigma_r^i, \end{aligned} \tag{D.14}$$

a finite-dimensional system is obtained

$$\begin{bmatrix} M_{\rho A} & \times & \times & \times \\ \times & M_{I_\rho} & \times & \times \\ \times & \times & M_{C_b} & \times \\ \times & \times & \times & M_{C_s} \end{bmatrix} \begin{bmatrix} \dot{\underline{v}}_t \\ \dot{\underline{v}}_r \\ \dot{\underline{\sigma}}_r \\ \dot{\underline{\sigma}}_t \end{bmatrix} = \begin{bmatrix} \times & \times & \times & -D_1^\top \\ \times & \times & -D_2^\top & -D_0^\top \\ \times & D_2 & \times & \times \\ D_1 & D_0 & \times & \times \end{bmatrix} \begin{bmatrix} \underline{v}_t \\ \underline{v}_r \\ \underline{\sigma}_r \\ \underline{\sigma}_t \end{bmatrix} + \begin{bmatrix} \times & B_F \\ B_T & \times \\ \times & \times \\ \times & \times \end{bmatrix} \begin{bmatrix} \pi_{\partial,1} \\ \pi_{\partial,2} \end{bmatrix},$$

$$\begin{bmatrix} y_{\partial,1} \\ y_{\partial,2} \end{bmatrix} = \begin{bmatrix} \times & B_T & \times & \times \\ B_F & \times & \times & \times \end{bmatrix} \begin{bmatrix} \underline{v}_t \\ \underline{v}_r \\ \underline{\sigma}_r \\ \underline{\sigma}_t \end{bmatrix}. \tag{D.15}$$

The mass matrices $M_{\rho h},\ M_{I_\theta},\ M_{\mathcal{C}_b},\ M_{C_s}$ are computed as

$$\begin{aligned} M_{\rho A}^{ij} &= \left\langle \varphi_{v_t}^i,\, \rho A \varphi_{v_t}^j \right\rangle_\Omega, \qquad & M_{C_b}^{pq} &= \left\langle \varphi_{\sigma_r}^p,\, C_b \varphi_{\sigma_r}^q \right\rangle_\Omega, \\ M_{I_\rho}^{mn} &= \left\langle \varphi_{v_r}^m,\, I_\rho \varphi_{v_r}^n \right\rangle_\Omega, \qquad & M_{C_s}^{rs} &= \left\langle \varphi_{\sigma_t}^l,\, C_s \varphi_{\sigma_t}^s \right\rangle_\Omega, \end{aligned} \tag{D.16}$$

where $i, j \in \{1, N_{v_t}\},\ m, n \in \{1, N_{v_r}\},\ p, q \in \{1, N_{\sigma_r}\},\ l, s \in \{1, N_{\sigma_t}\}$. Matrices $D_1,\ D_2,\ D_0$ assume the form

$$\begin{aligned} D_1^{lj} &= \left\langle \varphi_{\sigma_t}^l,\, \partial_z \varphi_{v_T}^j \right\rangle_\Omega, \\ D_2^{pn} &= \left\langle \varphi_{\sigma_r}^p,\, \partial_z \varphi_{v_r}^n \right\rangle_\Omega, \end{aligned} \qquad D_0^{rn} = -\left\langle \varphi_{\sigma_t}^r,\, \varphi_{v_r}^n \right\rangle_\Omega. \tag{D.17}$$

Vectors $B_F,\ B_T$ are computed as ($i \in 1, N_{v_t}$ and ($m \in 1, N_{v_r}$)

$$B_F^i = \varphi_{v_t}^i(b), \qquad B_T^j = \varphi_{v_r}^j(b). \tag{D.18}$$

D.3 Control by Neural Approximators and MSL

Due to invertibility of the mass matrix we can reduce the above equation to a controlled linear system representing the discretized dynamics of the boudary-controlled Tymoshenko beam

$$\begin{aligned} \dot{\underline{z}}(t) &= A\underline{z}(t) + B\pi_\partial(t) \\ y_\partial(t) &= C\underline{z}(t) \end{aligned} \tag{D.19}$$

with

$$z = \begin{bmatrix} \underline{v}_t \\ \underline{v}_r \\ \underline{\sigma}_r \\ \underline{\sigma}_t \end{bmatrix}, \quad \pi_\partial = \begin{bmatrix} \pi_{\partial,1} \\ \pi_{\partial,2} \end{bmatrix}, \quad y_\partial = \begin{bmatrix} y_{\partial,1} \\ y_{\partial,2} \end{bmatrix} \tag{D.20}$$

and

$$\begin{aligned} A &= \begin{bmatrix} \times & \times & \times & -M_{\rho A}^{-1} D_1^\top \\ \times & \times & -M_{I_\rho}^{-1} D_2^\top & -M_{I_\rho}^{-1} D_0^\top \\ \times & M_{C_b}^{-1} D_2 & \times & \times \\ M_{C_s}^{-1} D_1 & M_{C_s}^{-1} D_0 & \times & \times \end{bmatrix}, \\ B &= \begin{bmatrix} \times & M_{\rho A}^{-1} B_F \\ M_{I_\rho}^{-1} B_T & \times \\ \times & \times \\ \times & \times \end{bmatrix}, \quad C = \begin{bmatrix} \times & B_T & \times & \times \\ B_F & \times & \times & \times \end{bmatrix} \end{aligned} \tag{D.21}$$

We consider a parametrization $u_{\partial,\theta}$ with parameters θ of the boundary controller π_∂ via a multi–layer perceptron. The neural network controller $\pi_{\partial,\theta}$ takes as input the discretized state of the PDE $\pi_\partial(t) = \pi_{\partial,\theta}(\underline{z}(t)),\ \ t \mapsto z \mapsto u_{\partial,\theta}$. We apply the MSL to the controlled system

$$\dot{\underline{z}}(t) = A\underline{z}(t) + B\pi_{\partial,\theta}(\underline{z}(t))$$

Further details on the experimental setup and numerical results are given in Appendix E.3.

E Experimental Details

Experimental setup Experiments have been performed on a workstation equipped with a 48 threads AMD RYZEN THREADRIPPER 3960X a NVIDIA GEFORCE RTX 3090 GPUs and two NVIDIA RTX A6000. The main software implementation has been done within the `PyTorch` framework. Some functionalities rely on `torchdyn` (Poli et al., 2020b) ODE solvers and `torchcde` (Kidger et al., 2020b) cubic splines interpolation utilities for the *interpolated* version of the adjoint gradients.

Common experimental settings In all experiments to setup the multiple shooting problem, we choose an evenly spaced discretization of the time domain $[t_0, t_N]$, i.e.

$$\forall n = 1, \dots, N \quad t_n = t_{n-1} + \frac{1}{N}(t_N - t_0)$$

E.1 Variational Multiple Shooting Layers

Dataset We apply *variational multiple shooting layers* (vMSL) to trajectory generations of various dynamical systems. In particular, we consider the Van Der Pol oscillator

$$\begin{aligned} \dot{p} &= q \\ \dot{q} &= \alpha(1 - p^2)q - p \end{aligned}$$

as well as the Rayleigh Duffing system

$$\begin{aligned} \dot{p} &= q \\ \dot{q} &= \alpha p - 2p^3 + (1 - q^2)q \end{aligned}$$

We generate a dataset of 10000 trajectories by solving the above systems until $T = 1$. Each trajectory consists of 20 regularly sampled observations subject to additive noise ϵ where $\epsilon \sim \mathcal{N}(0, \Sigma)$, with Σ not diagonal i.e state–correlated noise.

Models and training Both vMSLs as well as Latent Neural ODE baselines are trained for Latent Neural GDEs are trained for 300 epochs with ADAM (Kingma and Ba, 2014). We schedule the learning rate using one cycle policies (Smith and Topin, 2019) where the cycle peak for the learning rate is 10^{-2}, set to be reached at epoch 100. The encoder architecture is shared across all models as is defined as two layers of *temporal convolutions* (TCNs), followed by a linear layer operating on flattened features. Between each TCN layer we introduce a maxpool operator to reduce sequence length. We solve Neural ODEs with `dopri5` solver with tolerances 10^{-4}.

We experiment with both `fw sensitivity` MSL as well as zeroth–order MSL as vMSL decoders. In all cases, we perform a single iteration of the chosen forward method. The parallelized ODE solves apply a single step of *Runge–Kutta 4*. We note that vMSL *number of function evaluation* (NFE) measurements also include the initialization calls to the vector field performed by the coarse solver to obtain shooting parameters B_0. Fig. 12 provides visualizations for decoder samples (extrapolation) of all models compared to ground–truth trajectories while Fig. 13 displays the learned vector fields of both vMSL and Latent ODE model.

To train all models we set the output–space prior $p(\hat{x}) := \mathcal{N}(x, \sigma)$ with $\sigma = 0.1$.

E.2 Optimal Limit Cycle Control via Multiple Shooting Layers

In the optimal control tasks we considered a simple mechanical system of the form

$$\begin{aligned} \dot{q}(t) &= p(t) \\ \dot{p}(t) &= \pi_\theta(q(t), p(t)) \end{aligned}, \quad z = [q, p].$$

evolving in a time span $[t_0, t_N] = [0, 10]$ and we fixed $N = 99$. The task was the one of stabilizing the state of different *loci* $S_d = \{z \in \mathcal{Z} : s_d(z) = 0\}$ by minimizing $|s_d(z(t))|$, $s_d : [q(t), p(t)] \mapsto s_d(q(t), p(t))$. Specifically, we chose the following *loci* of points

1. $s_d(q(t), p(t)) = q^2(t) + p^2(t) - 1$ [unit circle]
2. $s_d(q(t), p(t)) = \sqrt{(q(t) - \alpha)^2 + p^2(t)}\sqrt{(q(t) + \alpha)^2 + p^2(t)} - k$

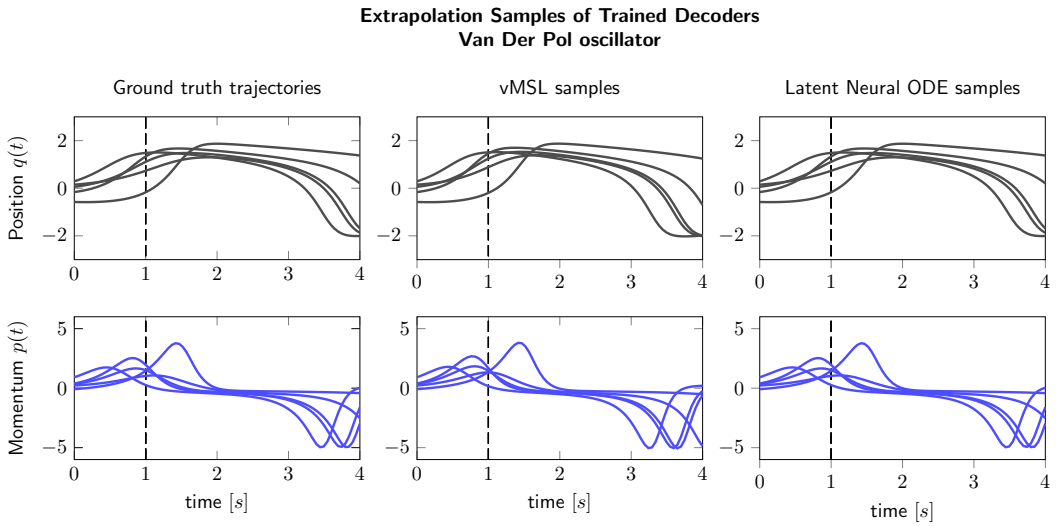

Figure 12: Samples of vMSLs and Latent Neural ODE baselines in the trajectory generation task on Van Der Pol oscillators. The samples are obtained by querying the decoders at desired initial conditions. The models extrapolate beyond $T = 1$ used in training.

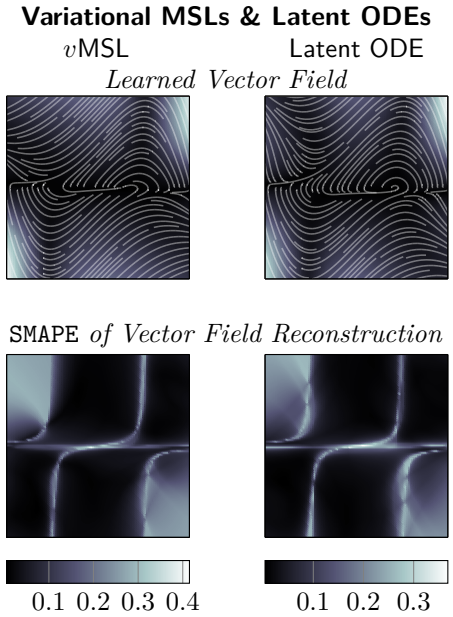

Figure 13: Learned vector fields by vMSL and Latent ODE decoders trained on noisy trajectories of the Van der Pol oscillator. vMSL models obtain the same result at a significantly cheaper NFE cost.

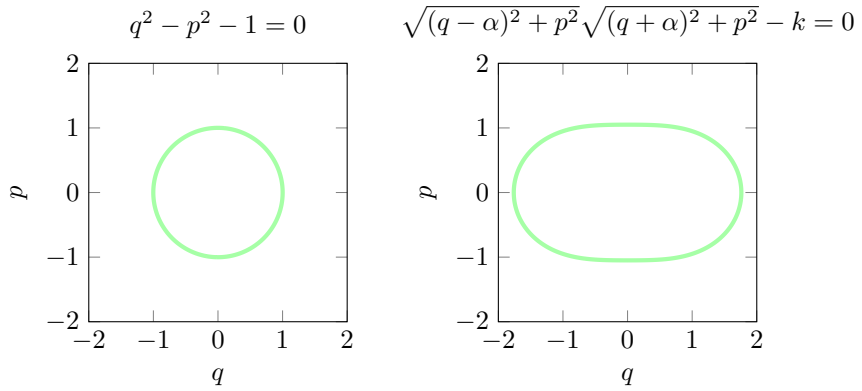

Figure 14: Desired *loci* in the state space, i.e. *limit cycles* to be created in the vector field through the control action $u_\theta(q(t), p(t))$.

across timestamps. The desired curves s_d are displayed in Fig. 14.

We compared the performance of MSL with the one of a standard (sequential) Neural ODE trained with `dopri5` and `rk4` solver. The objective was to show that MSL can achieve the same control performance while drastically reducing the computational cost of the training.

Models and training The loss function used to train the controlled was chosen as

$$\begin{aligned} \min_{\theta} \quad & \frac{1}{2N|Z_0|} \sum_{j=0}^{|Z_0|} \sum_{n=0}^{N} \left| s_d\left(b^*_{n,j}\right) \right| + \alpha \left\| \pi_\theta(b^*_{n,j}) \right\|_1 \\ \text{subject to} \quad & B^*_j : g_\theta(B^*_j, z^j_0) = \mathbb{0} \\ & z^j_0 \in Z_0, \quad \alpha \geq 0 \end{aligned}$$

where $B^*_j = (b^*_{0,j} \ \cdots \ b^*_{N,j})$. It penalizes the distance of trajectories from the desired curve as well as the control effort. In both the MSL and the Neural ODE baseline the controller $\pi_\theta(q, p)$ has been chosen as a neural network composed with two fully–connected layers of 32 neurons each and hyperbolic tangent activation. In the forward pass of MSL we performed a single iteration of the `fw sensitivity`–type algorithm. The parallelized ODE solver applies a single step of *Runge–Kutta 4* to each shooting parameter b_n. The backward pass has been instead performed with reverse–mode AD. At the beginning of the training phase, the shooting parameters B^0_0 have been initialized with with the sequential `dopri5` solver with tolerances set to 10^{-8}, i.e. $B^0_0 = \{\tilde{\phi}_\theta(z_0, t_0, t_n)\}_n$. As described in the main text, B^0 has then been updated at each optimization step with the B^* of the previous iteration to track the changes in the parameters θ and preserving the ability to track the "true" solution $\{\tilde{\phi}_\theta(z_0, t_0, t_n)\}_n$ with a single iteration of the Newton method (following the results of Theorem 1). The time horizon has been set to $[0, 10s]$ and we fixed $N = 100$ shooting parameters. The baseline Neural ODE has been instead trained with standard `dopri5` solver with tolerances set to 10^{-5} and the sequential `rk4` solver with N steps over the time horizon.

It is worth to be noticed that both the *parallelized* `rk4` integration step of MSL and the *sequential* `rk4` integration in the Neural ODE baseline operates with the same step size of $0.1s$.

All models have been trained for 2500 epochs with a single batch of 2048 initial conditions (q_0, p_0) uniformly distributed in $[-2, 2] \times [-2, 2]$ with ADAM (Kingma and Ba, 2014) optimizer and learning rate 10^{-4}.

For the *circle* desired limit cycle, the training procedure has been repeated with different initial conditions and neural network initializations in a Monte Carlo Simulation of 50 runs. Further, at each training step of MSL we solved the forward system using `dopri5` with absolute and relative tolerances set to 10^{-5} to compute `SMAPE` with the current MSL solution across training iterations shown in Fig. 8. Similarly, throughout the training of the baseline Neural ODE we recorded the `NFE`s of the forward pass across iterations. We also repeated the training of each model recording the wall–clock time of every training iteration.

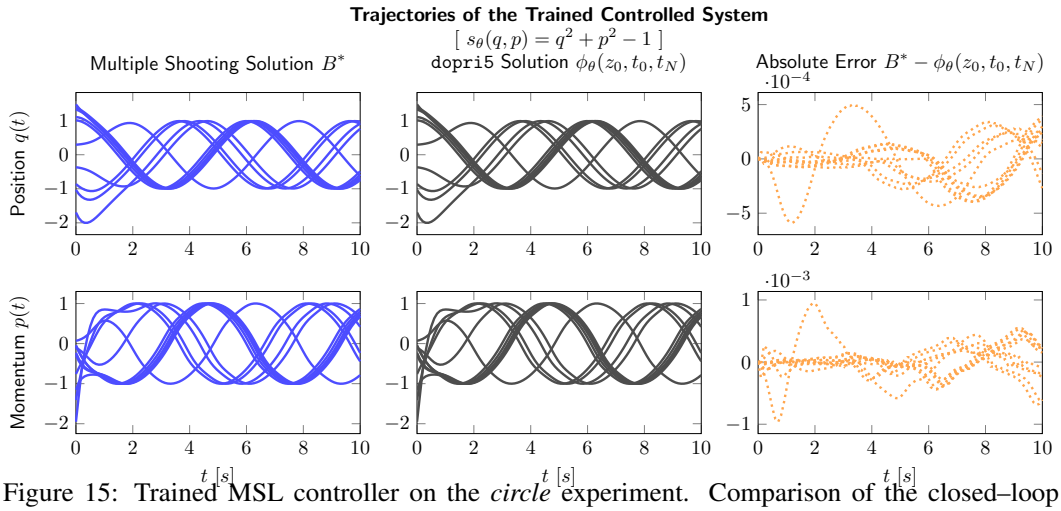

Figure 15: Trained MSL controller on the *circle* experiment. Comparison of the closed–loop trajectories obtained with MSL B^* and the `dopri5` counterpart.

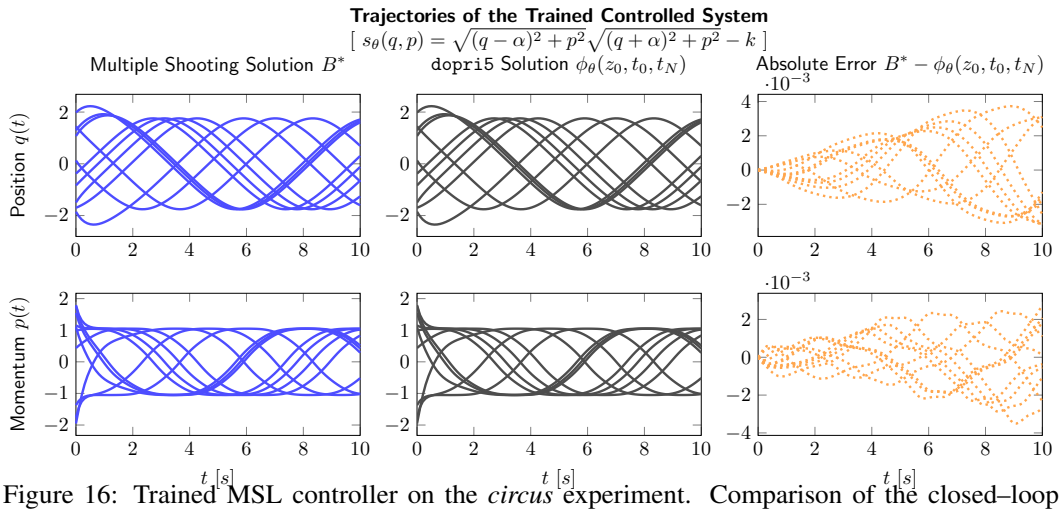

Figure 16: Trained MSL controller on the *circus* experiment. Comparison of the closed–loop trajectories obtained with MSL B^* and the `dopri5` counterpart.

Analysis of results Figures 15 and 16 display the resulting trajectories of the trained MSL in the *circle* and *circus* control tasks. In particular, we compared the last MSL forward solution B^* with the trajectories obtained with the accurate sequential solver with the trained π_θ.

We also notice that the MSL and Neural ODE baseline converge to very similar controllers and closed–loop vector fields, as it is shown in Fig. 17.

In Fig. 18, we report the wall-clock times of each forward–backward passes across training iterations. It can be noticed how MSLs encompass sequential approaches with a 10x speedup compared to `dopri5` (even though maintaining a similar accuracy in the solutions) and a 3x speedup w.r.t. the sequential `rk4` solver with the same number of steps per sub–interval.

E.3 Neural Optimal Boudary Control of the Timoshenko Beam

With this experiment we aimed at showing the scaling of `fw sensitivity` MSL to higher–dimensional regimes in a neural–network optimal control tasks. In particular, we wished to investigate if the acceleration property of *one–step* MSLs established by Theorem 1 holds when the system state has hundreds of dimensions.

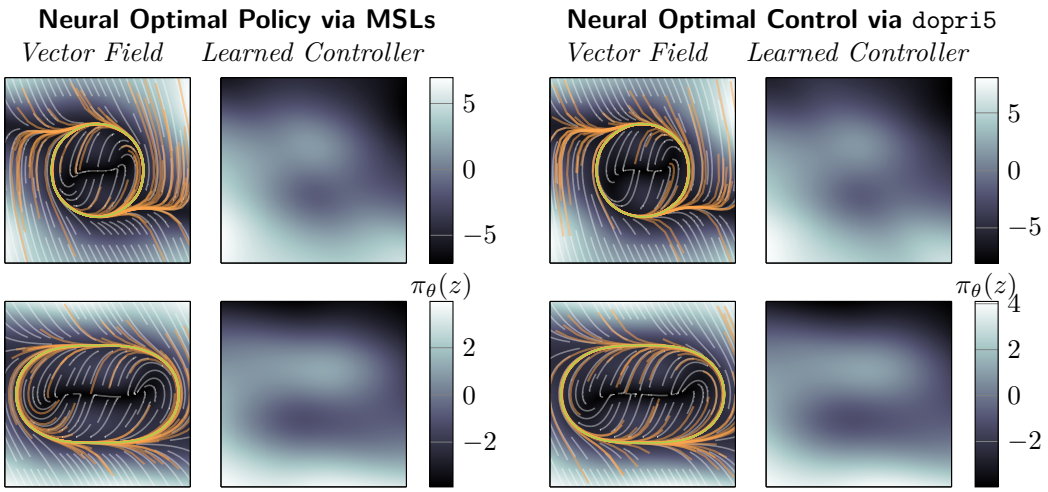

Figure 17: Comparison between the learned controllers and closed loop vector fields for the MSL and Neural ODE baseline, in different tasks.

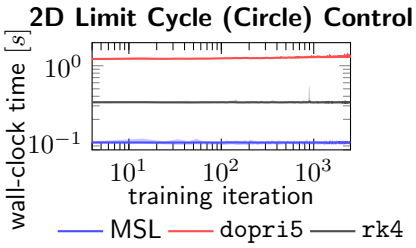

Figure 18: Wall-clock time of complete training iteration (forward/backward passes + GD update) for different solvers on the *circle experiment*

Model and training We kept an identical training setup to the limit cycle control task of E.2. However, we chose a time horizon of $5s$ and we fixed $N = 500$ shooting parameters. We only compared the proposed MSL model to the sequential `rk4` as we empirically noticed how `dopri5` was extremely slow to perform a single integration of the discretized PDE (possibly due to the stiffness of the problem) and was also highly numerically unstable (high rate of *underflows*).

We implemented a software routine based on the `fenics` Alnæs et al. (2015) computational platform to obtain the finite–elements discretization (namely, matrices A and B in (D.19)) of the PDE given the physical parameters of the model, the number of elements, and the initial condition of the beam. We chose a 50 elements discretization of the Timoshenko PDE for a total of 200 dimensions of the discretized state $\underline{z}(t)$ and we initialized the distributed state as $z(x, 0) = [\sin(\pi x), \sin(3\pi x), 0, 0]$.

Since the experiment focus was the numerical performance of MSL training compared to Neural ODE baselines, we considered a simple stabilization task where the cantilever beam had to be straight. For this reason we selected the following loss criterium

$$L_\theta = \frac{1}{N} \sum_{n=0}^{N} \left(\|b^*_{n,\sigma_r}\|_2 + \|b^*_{n,\sigma_t}\|_2 \right) + \alpha \left\| \pi_{\partial,\theta}(b^*_n) \right\|_1$$

being b^*_{n,σ_r}, b^*_{n,σ_t} the portions of the shooting parameters corresponding to $\underline{\sigma}_r$ and $\underline{\sigma}_t$, respectively. The boundary controller was designed as a four-layers neural network with 16 neurons per layer, `softplus` activation on the first two hidden layers and hyperbolic tangent activation on the third.

Analysis of results We report additional experimental results. Figure 19 displays the trajectories of the system with the learned boundary control policy. It can be seen how the displacements variables for each of the finite elements swiftly goes to zero (straight beam configuration) with zero

velocity proving the effectiveness of the proposed model. Finally, Fig. 20 shows the initial and final configurations of the finite elements over the spatial domain $x \in [0, 1]$.

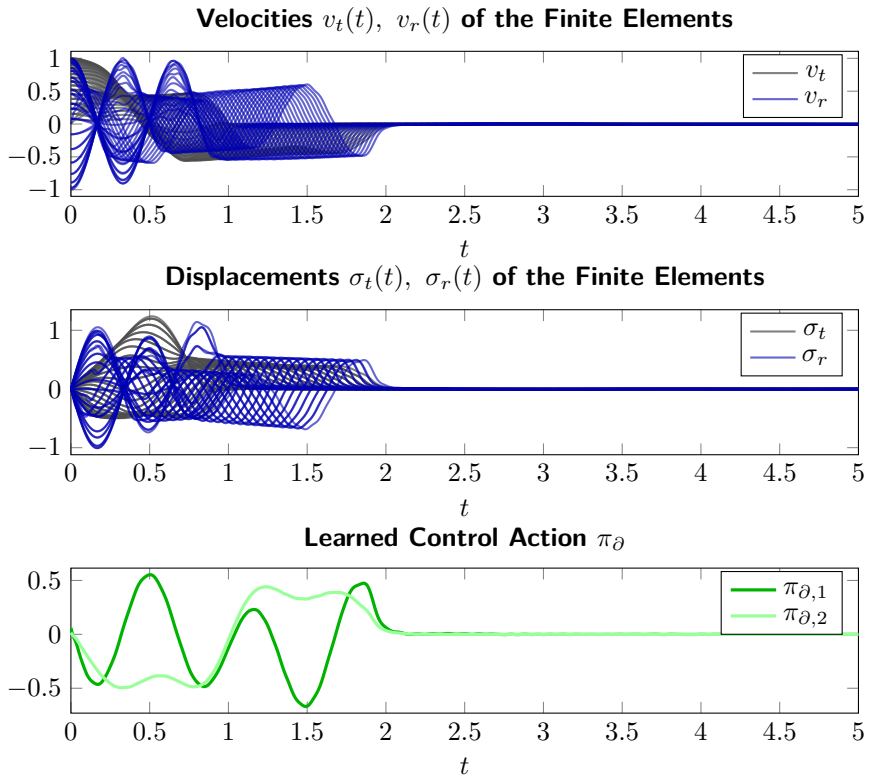

Figure 19: Trajectories of the finite elements states and learned control policy along the trajectory.

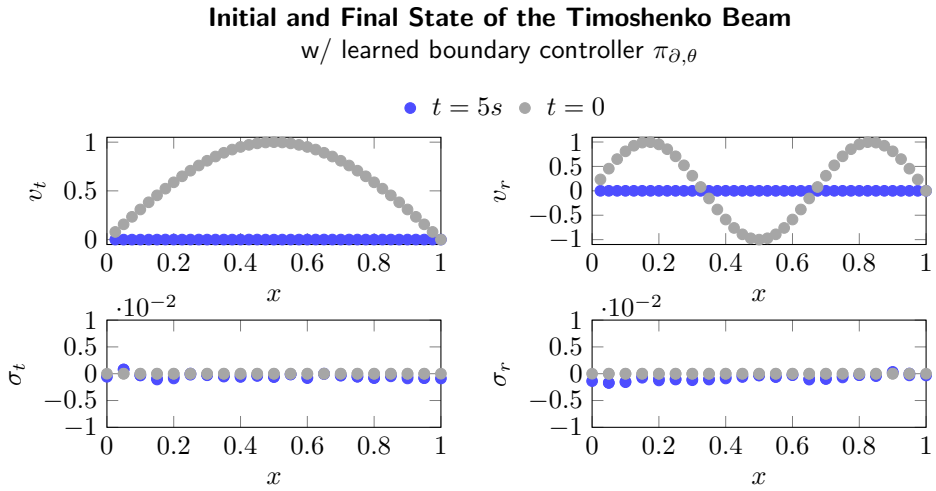

Figure 20: Initial and final (discretized) state of the controlled Timoshenko beam after training with MSL.

E.4 Fast Neural CDEs for Time Series Classification

Dataset We consider sepsis prediction with data from the `PhysioNet` 2019 challenge. In particular, the chosen dataset features 40335 variable length time series of patient features. The task involves

predicting whether patients develop sepsis over the course of their *intensive care unit* (ICU) stay, using the first 72 hours of observations. Since positive and negative classes are highly imbalanced, we report *area under the receiver operating characteristic* (`AUROC`) as task performance metric. For more details see (Kidger et al., 2020b), which contains the experimental setup followed in this work, and (Clifford et al., 2015) for more details on the dataset and task. The data split is performed according to (Kidger et al., 2020b) with 70% train, 15% validation and 15% test. The 70% split corresponds to 28233 time series, which in this experiment is taken as batch size to enable application of tracking MSLs relying on Theorem 1.

Models and training All model hyperparameters are collected from (Kidger et al., 2020b) for a fair comparison. We train a standard *neural controlled differential equation* (Neural CDE) and an equivalent Neural CDE solved with a zeroth–order MSL. Both baseline and MSL Neural CDEs use standard *reverse mode autodiff* to compute gradients. We train for 1000 epochs (here equivalent to iterations due to full–batch training) with a learning rate of 10^{-4} for `AdamW` (Loshchilov and Hutter, 2017) and with weight decay regularization of 0.03.