

## 663 A JacobianODE technical details

### 664 A.1 Proof of Jacobian-parameterized ordinary differential equations (ODEs)

665 **Proposition 1** (Jacobian-parameterized ODEs). *Let  $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t))$  and let  $\mathbf{J}_{\mathbf{f}}(\mathbf{x}(t)) = \mathbf{J}(\mathbf{x}(t)) =$   
666  $\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}(t))$ . Then given times  $t_0, t$  we can express  $\mathbf{f}$  parameterized by the Jacobian as*

$$\mathbf{f}(\mathbf{x}(t)) = \frac{G(t_0, t; \mathbf{J}, \mathbf{c}) + \mathbf{x}(t) - \mathbf{x}(t_0)}{t - t_0}, \quad (9)$$

667 where

$$G(t_0, t; \mathbf{J}, \mathbf{c}) = \int_{t_0}^t \int_s^t \mathbf{J}(\mathbf{c}_{s,t}(r)) \mathbf{c}'_{s,t}(r) dr ds \quad (10)$$

668 and we have abbreviated  $\mathbf{c}_{s,t}(r) = \mathbf{c}(r; s, t, \mathbf{x}(s), \mathbf{x}(t))$ , a piecewise smooth curve on  $[s, t]$  parame-  
669 terized by  $r$  and beginning and ending at  $\mathbf{x}(s)$  and  $\mathbf{x}(t)$  respectively.

670 *Proof.* For given times  $t, t_0$ , and  $s$ , from the fundamental theorem of calculus, we have that

$$\mathbf{x}(t) - \mathbf{x}(t_0) = \int_{t_0}^t \mathbf{f}(\mathbf{x}(s)) ds$$

671 and

$$\mathbf{f}(\mathbf{x}(t)) - \mathbf{f}(\mathbf{x}(s)) = \int_s^t \mathbf{J}(\mathbf{c}_{s,t}(r)) \mathbf{c}'_{s,t}(r) dr$$

672 Letting

$$H(s, t) = \int_s^t \mathbf{J}(\mathbf{c}_{s,t}(r)) \mathbf{c}'_{s,t}(r) dr = \mathbf{f}(\mathbf{x}(t)) - \mathbf{f}(\mathbf{x}(s))$$

673 We then have that

$$\begin{aligned} \int_{t_0}^t H(s, t) ds &= \int_{t_0}^t \mathbf{f}(\mathbf{x}(t)) - \mathbf{f}(\mathbf{x}(s)) ds \\ &= \int_{t_0}^t \mathbf{f}(\mathbf{x}(t)) ds - \int_{t_0}^t \mathbf{f}(\mathbf{x}(s)) ds \\ &= (t - t_0) \mathbf{f}(\mathbf{x}(t)) - (\mathbf{x}(t) - \mathbf{x}(t_0)) \end{aligned}$$

674 Letting now

$$G(t_0, t; \mathbf{J}, \mathbf{c}) = \int_{t_0}^t H(s, t) ds = \int_{t_0}^t \int_s^t \mathbf{J}(\mathbf{c}_{s,t}(r)) \mathbf{c}'_{s,t}(r) dr ds$$

675 We can see that

$$G(t_0, t; \mathbf{J}, \mathbf{c}) = (t - t_0) \mathbf{f}(\mathbf{x}(t)) - (\mathbf{x}(t) - \mathbf{x}(t_0))$$

676 and thus

$$\mathbf{f}(\mathbf{x}(t)) = \frac{G(t_0, t; \mathbf{J}, \mathbf{c}) + \mathbf{x}(t) - \mathbf{x}(t_0)}{t - t_0}$$

677

□

## 678 A.2 Path integrating to generate predictions

679 As mentioned in section 3, consider an initial observed trajectory  $\mathbf{x}(t_0 + k\Delta t)$ ,  $k = 0, \dots, b$  of length  
 680 at least two ( $b \geq 1$ ). Let  $t_b = t_0 + b\Delta t$ . We compute an estimate of  $\hat{\mathbf{f}}(\mathbf{x}(t_b))$  of  $\mathbf{f}(\mathbf{x}(t_b))$  as

$$\hat{\mathbf{f}}(\mathbf{x}(t_b)) = \frac{G(t_0, t_b; \hat{\mathbf{J}}^\theta, \mathbf{c}) + \mathbf{x}(t_b) - \mathbf{x}(t_0)}{t_b - t_0}, \quad (11)$$

681 In practice, we compute this estimate by constructing a cubic spline on the initial observed trajectory  
 682 using 15 points (i.e.,  $b = 14$ ). Given that the computation of  $G$  involves a double integral - one  
 683 over states and over time - using a spline is computationally advantageous. This is because the path  
 684 integral (and all intermediate steps) can be quickly computed along the full spline, with the result  
 685 of each intermediate step along the path then being summed to approximate the time integral. The  
 686 integral is computed by interpolating 4 points for every gap between observed points, resulting in a  
 687 discretization of 58 points along the spline. Integrals are computed using the trapezoid method from  
 688 `torchquad` [69].

689 Once we have constructed our estimate of  $\hat{\mathbf{f}}(\mathbf{x}(t_b))$  we can estimate  $\mathbf{f}$  at any other point  $\mathbf{x}(t)$  as

$$\hat{\mathbf{f}}(\mathbf{x}(t)) = \begin{cases} H(t_b, t) + \hat{\mathbf{f}}(\mathbf{x}(t_b)), & \text{if } t_b < t \\ \hat{\mathbf{f}}(\mathbf{x}(t_b)) - H(t, t_b), & \text{if } t_b > t \\ \mathbf{f}(\mathbf{x}(t_b)) & \text{if } t_b = t \end{cases}$$

690 where by convention we integrate forwards in time and  $H$  is the path integral defined above, with  $\hat{\mathbf{J}}$   
 691 in place of  $\mathbf{J}$ . In practice, for the integration path  $\mathbf{c}(r; s, t, \mathbf{x}(s), \mathbf{x}(t))$ , we construct a line from  $\mathbf{x}(s)$   
 692 to  $\mathbf{x}(t)$  as

$$\mathbf{c}(r; s, t, \mathbf{x}(s), \mathbf{x}(t)) = \left(1 - \frac{r-s}{t-s}\right) \mathbf{x}(s) + \frac{r-s}{t-s} \mathbf{x}(t)$$

693 to maintain the interpretability of having  $r$  in the range  $[s, t]$  however it can be easily seen that setting  
 694  $r' = \frac{r-s}{t-s}$  we recover the familiar line

$$\mathbf{c}(r') = (1 - r')\mathbf{x}(s) + r'\mathbf{x}(t)$$

695 with  $r'$  taking values on  $[0, 1]$ . Line integrals are computed with 20 discretization steps using the  
 696 trapezoid method from `torchquad` [69].

697 Using  $\hat{\mathbf{f}}(\mathbf{x}(t))$ , we can then generate predictions as

$$\hat{\mathbf{x}}(t + \Delta t) = \mathbf{x}(t) + \int_t^{t+\Delta t} \hat{\mathbf{f}}(\mathbf{x}(\tau)) d\tau \quad (12)$$

698 where the integral can be computed by a standard ODE solver (we used the RK4 method from  
 699 `torchdiffeq` with default values for the relative and absolute tolerance).

## 700 B Control-theoretic analysis details

### 701 B.1 Gramian computation

702 We begin with equation 2 from section 2, in which we separate the locally-linearized dynamics in  
 703 the tangent space into separate pairwise inter-subsystem control interactions. These pairwise control  
 704 interactions are in general of the form

$$\delta \dot{\mathbf{x}}^A(t) = \mathbf{A}(t)\delta \mathbf{x}^A(t) + \mathbf{B}(t)\delta \mathbf{x}^B(t)$$

705 where  $\mathbf{A}$  is the within-subsystem Jacobian,  $\mathbf{B}$  is the across subsystem Jacobian and  $\mathbf{x}^A$ ,  $\mathbf{x}^B$  are the  
 706 states of subsystems A and B, respectively. For a given control system, the time-varying reachability  
 707 Gramian on the interval  $[t_0, t_1]$  is defined as

$$\mathbf{W}_r(t_0, t_1) \triangleq \int_{t_0}^{t_1} \Phi(t_1, \tau) \mathbf{B}(\tau) \mathbf{B}^T(\tau) \Phi^T(t_1, \tau) d\tau$$

where  $\Phi$  denotes the state-transition matrix of the intrinsic dynamics of subsystem A without any input (i.e.,  $\delta \mathbf{x}^A(t) = \Phi(t, t_0) \delta \mathbf{x}^A(t_0)$ ) [56]. Differentiating with respect to  $t$ , we obtain  $\delta \dot{\mathbf{x}}^A(t) = \frac{\partial}{\partial t} \Phi(t, t_0) \delta \mathbf{x}^A(t_0)$ . Noting also that, in the absence of input from subsystem B, we have

$$\delta \dot{\mathbf{x}}^A(t) = \mathbf{A}(t) \delta \mathbf{x}^A(t) = \mathbf{A}(t) \Phi(t, t_0) \delta \mathbf{x}^A(t_0)$$

Thus setting the equations equal to each other and canceling  $\delta \mathbf{x}^A(t_0)$  from both sides we obtain

$$\frac{\partial}{\partial t} \Phi(t, t_0) = \mathbf{A}(t) \Phi(t, t_0)$$

Note that  $\Phi(t_0, t_0) = \mathbf{I}$ . Now, letting  $\Gamma(t, \tau) = \Phi(t, \tau) \mathbf{B}(\tau) \mathbf{B}^T(\tau) \Phi^T(t, \tau)$  and differentiating the Gramian expression with respect to the second argument (and using the Leibniz integral rule) yields

$$\begin{aligned} \frac{\partial}{\partial t} \mathbf{W}_r(t_0, t) &= \frac{\partial}{\partial t} \int_{t_0}^t \Gamma(t, \tau) d\tau \\ &= \Gamma(t, t) \frac{\partial}{\partial t}(t) - \Gamma(t, t_0) \frac{\partial}{\partial t}(t_0) + \int_{t_0}^t \frac{\partial}{\partial t} \Gamma(t, \tau) d\tau \\ &= \mathbf{I} \mathbf{B}(t) \mathbf{B}^T(t) \mathbf{I}(1) - 0 + \int_{t_0}^t \frac{\partial}{\partial t} \Gamma(t, \tau) d\tau \end{aligned}$$

and observing

$$\begin{aligned} \frac{\partial}{\partial t} \Gamma(t, \tau) &= \left( \frac{\partial}{\partial t} \Phi(t, t_0) \right) \mathbf{B}(\tau) \mathbf{B}^T(\tau) \Phi^T(t, \tau) + \Phi(t, t_0) \mathbf{B}(\tau) \mathbf{B}^T(\tau) \left( \frac{\partial}{\partial t} \Phi^T(t, \tau) \right)^T \\ &= \mathbf{A}(t) \Phi(t, t_0) \mathbf{B}(\tau) \mathbf{B}^T(\tau) \Phi^T(t, \tau) + \Phi(t, t_0) \mathbf{B}(\tau) \mathbf{B}^T(\tau) \Phi^T(t, \tau) \mathbf{A}^T(t) \\ &= \mathbf{A}(t) \Gamma(t, \tau) + \Gamma(t, \tau) \mathbf{A}^T(t) \end{aligned}$$

we can continue

$$\begin{aligned} \frac{\partial}{\partial t} \mathbf{W}_r(t_0, t) &= \mathbf{B}(t) \mathbf{B}^T(t) + \mathbf{A}(t) \int_{t_0}^t \Gamma(t, \tau) d\tau + \left( \int_{t_0}^t \Gamma(t, \tau) d\tau \right) \mathbf{A}^T(t) \\ &= \mathbf{B}(t) \mathbf{B}^T(t) + \mathbf{A}(t) \mathbf{W}_r(t_0, t) + \mathbf{W}_r(t_0, t) \mathbf{A}^T(t) \end{aligned}$$

which illustrates that the reachability Gramian can be solved for using an ODE integrator (with initial condition  $\mathbf{W}_r(t_0, t_0) = \mathbf{0}$ ) [55]. In practice, to compute the reachability Gramians using the trained JacobianODE models, we fit a cubic spline  $\mathbf{c}(t)$  to the reference trajectory  $\mathbf{x}(t)$  and compute  $\mathbf{J}(t) = \mathbf{J}(\mathbf{c}(t))$ . We can then parse the Jacobian matrix into its component submatrices and compute the Gramians accordingly.

The reachability Gramian is a symmetric positive semidefinite matrix [56]. The optimal cost of driving the system from state  $\delta \mathbf{x}_0$  to state  $\delta \mathbf{x}_1$  on time interval  $[t_0, t_1]$  can be computed as

$$(\delta \mathbf{x}_1 - \Phi(t_1, t_0) \delta \mathbf{x}_0)^T \mathbf{W}_r^{-1}(t_0, t_1) (\delta \mathbf{x}_1 - \Phi(t_1, t_0) \delta \mathbf{x}_0)$$

Thus each eigenvalue of  $\mathbf{W}_r$  reflects the ease of control along the corresponding eigenvector [56, 58]. Eigenvectors with larger corresponding eigenvalues will have smaller inverse eigenvalues, and thus scale the cost down along those directions when computing the cost as above.

## B.2 Extension to non-autonomous dynamics

While we deal with autonomous systems in this work, we note that this construction can easily be extended to include non-autonomous dynamical systems, of the form  $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, t)$  by constructing an augmented state  $\tilde{\mathbf{x}} \in \mathbb{R}^{n+1}$  which is simply the concatenation of  $\mathbf{x}$  with  $t$ . This yields the autonomous dynamics  $\dot{\tilde{\mathbf{x}}} = \tilde{\mathbf{f}}(\tilde{\mathbf{x}})$ , where  $\tilde{\mathbf{f}} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}$  is the concatenation of  $\mathbf{f}$  with a function that maps all states to 1.

### B.3 Extension to more than two subsystems

Consider a nonlinear dynamical system in  $\mathbb{R}^n$ , defined by  $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t))$ . Suppose now that the system is composed of  $K$  subsystems, where the time evolution of subsystem  $k$  is given by  $\mathbf{x}^{(k)}(t) \in \mathbb{R}^{n_k}$ .  $\mathbf{x}(t)$  is then comprised of a concatenation of the  $\mathbf{x}^{(k)}(t)$ , with  $\sum_{k=1}^K n_k = n$ . Given a particular reference trajectory,  $\mathbf{x}(t)$ , with  $\mathbf{J}$  as the Jacobian of  $\mathbf{f}$ , then the tangent space dynamics around the reference trajectory for subsystem  $\alpha \in [1, \dots, K]$  are given by

$$\delta \dot{\mathbf{x}}^{(\alpha)}(t) = \sum_{k=1}^K \mathbf{J}^{k \rightarrow \alpha}(\mathbf{x}(t)) \delta \mathbf{x}^{(k)}(t)$$

where  $\mathbf{J}^{k \rightarrow \alpha}(\mathbf{x}(t)) \in \mathbb{R}^{n_\alpha \times n_k}$  is the submatrix of  $\mathbf{J}(\mathbf{x}(t))$  in which the columns correspond to subsystem  $k$  and the rows correspond to subsystem  $\alpha$ . Now, for a given subsystem  $\beta \in [1, \dots, K]$  with  $\beta \neq \alpha$ , we wish to analyze the ease with which  $\beta$  can control  $\alpha$  locally around the reference trajectory, without intervention from other subsystems. Discounting the interventions from other subsystems equates to setting  $\delta \mathbf{x}^{(k)}(t) = 0$  for  $k \neq \alpha, \beta$ , leaving the expression

$$\delta \dot{\mathbf{x}}^{(\alpha)}(t) = \mathbf{J}^{\alpha \rightarrow \alpha}(\mathbf{x}(t)) \delta \mathbf{x}^{(\alpha)}(t) + \mathbf{J}^{\beta \rightarrow \alpha}(\mathbf{x}(t)) \delta \mathbf{x}^{(\beta)}(t)$$

which quantifies the influence  $\beta$  can exert over  $\alpha$  in the absence of perturbations from any other subsystem. Using this representation, the reachability Gramian can be computed as described above. Performing this procedure for all pairs of subsystems  $\alpha, \beta \in [1, \dots, K]$  thus characterizes all pairwise control relationships between subsystems along the reference trajectory.

## C Supplementary results

### C.1 Control and communication capture different phenomena

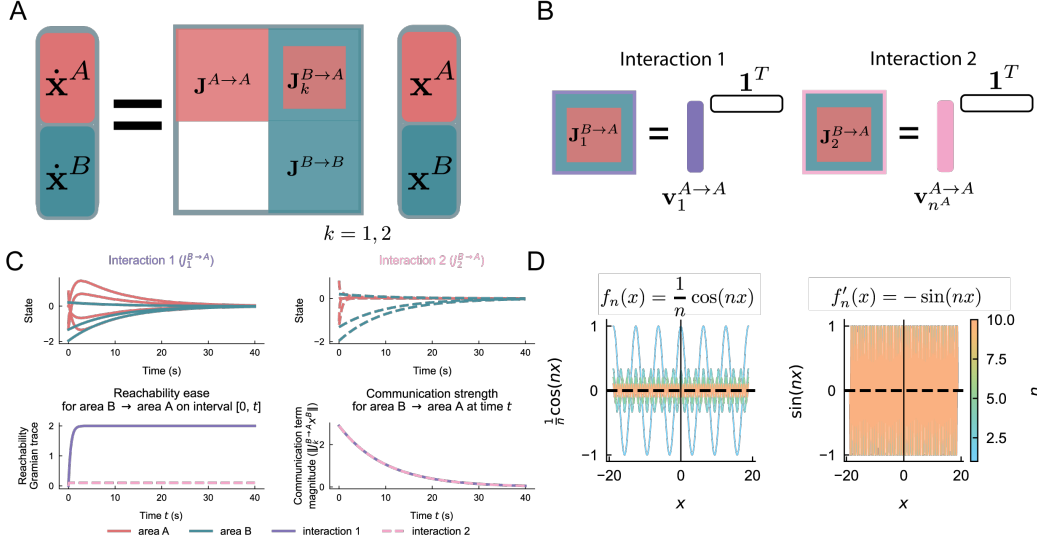
In the Introduction (section I) we note that measuring communication between two systems is different than measuring control. To illustrate this, consider two brain areas, A and B, whose dynamics are internally linear. The areas are connected only through a linear feedforward interaction term from area B to area A (Figure S1A). Concretely, we consider the dynamics:

$$\begin{aligned} \dot{\mathbf{x}}^A(t) &= \mathbf{J}^{A \rightarrow A} \mathbf{x}^A(t) + \mathbf{J}^{B \rightarrow A} \mathbf{x}^B(t) \\ \dot{\mathbf{x}}^B(t) &= \mathbf{J}^{B \rightarrow B} \mathbf{x}^B(t) \end{aligned}$$

Here, the  $\mathbf{J}$  matrices are time-invariant. When considering communication between brain areas, one might aim to find the subspaces in which communication between B and A occurs, as well as the messages passed [33, 93]. In this construction, the subspace in which area B communicates with area A is explicitly given by  $\mathbf{J}^{B \rightarrow A}$ , and thus the message, or input, from area B to area A at any time  $t$  is simply given by  $\mathbf{J}^{B \rightarrow A} \mathbf{x}^B(t)$ . We pick the dimensions of each region to be  $n_A = n_B = 4$  and let the eigenvectors of the (negative definite matrix)  $\mathbf{J}^{A \rightarrow A}$  be given by  $\mathbf{v}_i, i = 1, 2, 3, 4$ . The eigenvectors are numbered in order of decreasing real part of their corresponding eigenvalue. Now, suppose we construct the interaction matrix  $\mathbf{J}^{B \rightarrow A}$  in two different ways: If we let (1)  $\mathbf{J}_1^{B \rightarrow A} = \mathbf{v}_1 \mathbf{1}^T$  (Figure S1B, left), then the signal from B is projected onto the most stable mode of region A, whereas if we let (2)  $\mathbf{J}_2^{B \rightarrow A} = \mathbf{v}_4 \mathbf{1}^T$  (Figure S1B, right), the signal is projected onto the least stable mode. While interaction 1 and interaction 2 communicated messages with identical magnitudes, interaction 2 led to much lower cost reachability control (Figure S1C). This illustrates that control depends not only on the directions along which the areas can communicate, but also on how *aligned* the communication is with the target area's dynamics.

### C.2 Derivative estimation is not implied by function estimation

An alternative approach to directly estimating the Jacobians would be to learn an approximation  $\hat{\mathbf{f}}$  of the function  $\mathbf{f}$ , and then approximate the Jacobian via automatic differentiation (i.e., an estimate  $\hat{\mathbf{J}} = \frac{\partial}{\partial \mathbf{x}} \hat{\mathbf{f}}$ , as with the NeuralODEs). While this can be effective in certain scenarios, it is not generally the case that approximating a function well will yield a good approximation of its derivative. To illustrate this, we recall an example from Latrémoière et al. [59], in which functions



**Figure S1: Communication versus control in linear systems and the challenge of Jacobian estimation.** (A) Setup of two linearly connected brain areas, A and B. (B) Interaction matrices projecting signals from area B onto either the most stable (Interaction 1) or least stable (Interaction 2) eigenvectors of area A. (C) Although both interactions communicate identical signal magnitudes, Interaction 2 provides significantly enhanced reachability due to alignment with unstable modes of area A dynamics. (D) An illustrative example demonstrating that accurate approximation of a function (left panel) does not guarantee accurate approximation of its derivative (right panel), emphasizing the necessity of directly estimating the Jacobian.

773  $f_n = \frac{1}{n} \cos(nx)$  are used to approximate the function  $f(x) = 0$  (Figure S1D). While these approx-  
 774 imations improve with increasing  $n$  (i.e.,  $\lim_{n \rightarrow \infty} f_n = f$ ), this is not the case for the derivative  
 775 ( $\lim_{n \rightarrow \infty} f'_n = -\sin(nx) \neq f'$ ). This demonstrates the necessity of learning  $\mathbf{J}$  directly (rather than  
 776 first approximating  $\mathbf{f}$ ), which we also demonstrate empirically. In the context of machine learning,  
 777 this setting could be interpreted as overfitting [59]. As long as function estimates match at the specific  
 778 points in the training set, how the function fluctuates between these points is not constrained to match  
 779 the true function. For this reason, we included the Frobenius norm Jacobian regularization in our  
 780 implementation of the NeuralODEs (appendix D.3).

### 781 C.3 Full benchmark dynamical systems results

782 We here present the full results for all considered example dynamical systems. 10 time-step trajectory  
 783 predictions along with Jacobian estimation and estimated Lyapunov spectra are displayed in Figure  
 784 S2. Jacobian estimation errors using the 2-norm are presented in Table 2.

### 785 C.4 Ablation studies

786 To determine the value of the different components of the JacobianODE learning framework, we  
 787 performed several ablation studies. We chose to evaluate on the Lorenz system and the task-trained  
 788 RNNs, as these together provide two common settings of chaos and stability, as well as low- and  
 789 high-dimensional dynamics.

790 **Ablating the Jacobian-parameterized ODEs** The JacobianODE framework constructs an estimate  
 791 of the initial time derivative  $\mathbf{f}$  via a double integral of the Jacobian as described in section 3.1. To  
 792 briefly recall, Jacobian path integration is described as

$$\mathbf{f}(\mathbf{x}(t_f)) - \mathbf{f}(\mathbf{x}(t_i)) = \int_{\mathcal{C}} \mathbf{J} ds = \int_{t_i}^{t_f} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr, \quad (13)$$

793 When we do not have access to the initial derivative and base point  $\mathbf{f}(\mathbf{x}(t_i))$ ,  $\mathbf{x}(t_i)$ , we use the  
 794 following formulation

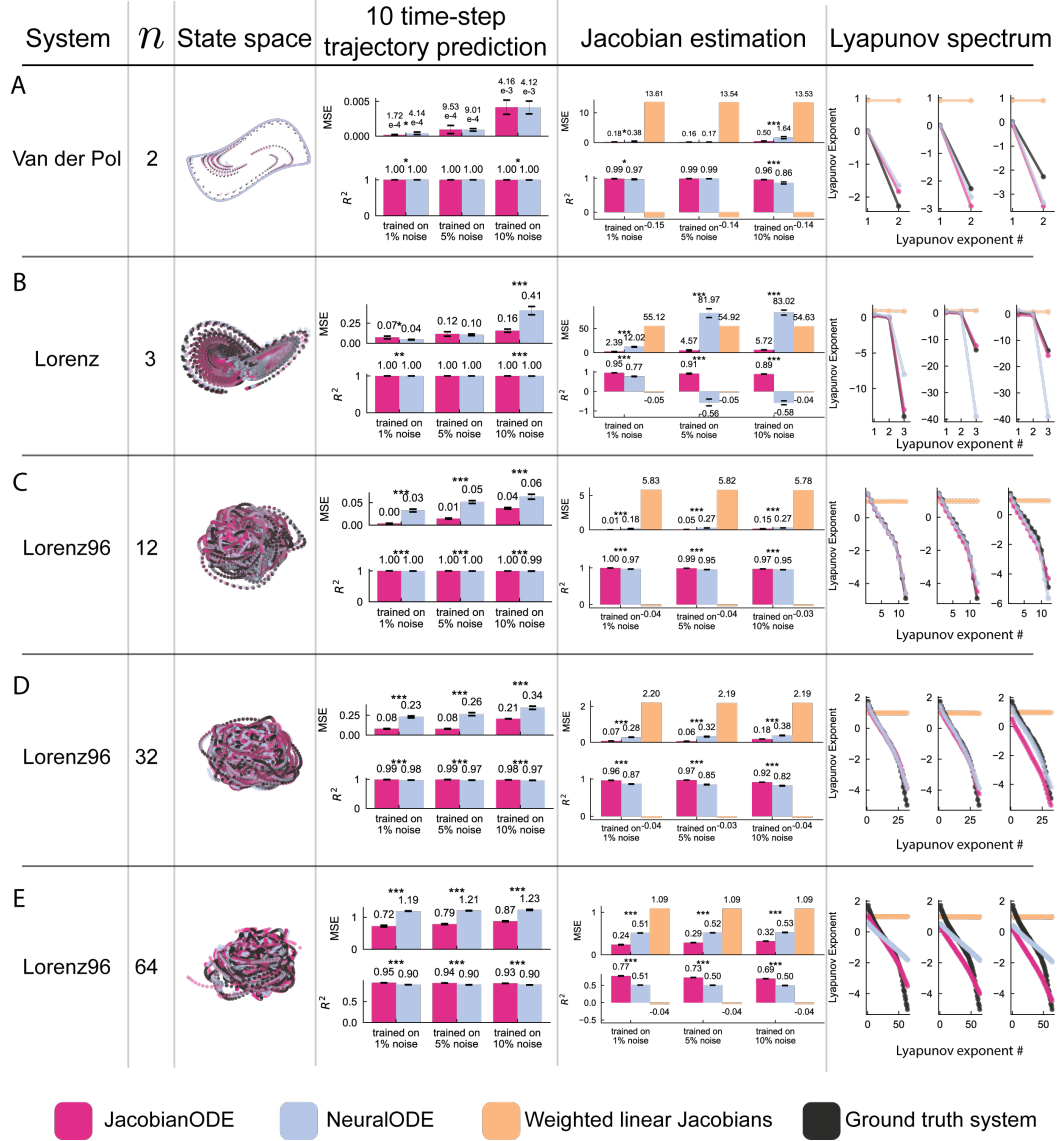


Figure S2: **Full dynamical systems prediction results.** State space representations, 10 time-step trajectory predictions, Jacobian estimation, and Lyapunov spectrum estimates for each of (A) the Van der Pol oscillator, (B) the Lorenz system, and the Lorenz96 system with (C) 12, (D) 32, and (E) 64 dimensions.

Table 2: Mean 2-norm error  $\langle \|\mathbf{J} - \hat{\mathbf{J}}\|_2 \rangle$  for each system and noise level. Errors are reported as mean  $\pm$  standard deviation, with statistics computed over five random seeds.

Project	Training noise	JacobianODE	NeuralODE	Weighted Linear
VanDerPol (2 dim)	1%	<b>0.7 <math>\pm</math> 0.1</b>	1.0 $\pm$ 0.3	6.05
	5%	<b>0.71 <math>\pm</math> 0.05</b>	<b>0.71 <math>\pm</math> 0.08</b>	6.03
	10%	<b>1.31 <math>\pm</math> 0.05</b>	2.2 $\pm$ 0.2	6.02
Lorenz (3 dim)	1%	<b>3.2 <math>\pm</math> 0.2</b>	8.6 $\pm$ 0.3	17.06
	5%	<b>4.9 <math>\pm</math> 0.9</b>	25.9 $\pm$ 1.5	17.02
	10%	<b>6.0 <math>\pm</math> 0.1</b>	26.4 $\pm$ 0.9	16.95
Lorenz96 (12 dim)	1%	<b>0.8 <math>\pm</math> 0.1</b>	3.5 $\pm$ 0.2	14.94
	5%	<b>1.75 <math>\pm</math> 0.09</b>	4.1 $\pm$ 0.1	14.93
	10%	<b>2.91 <math>\pm</math> 0.09</b>	4.2 $\pm$ 0.1	14.92
Lorenz96 (32 dim)	1%	<b>3.75 <math>\pm</math> 0.08</b>	7.8 $\pm$ 0.1	16.29
	5%	<b>3.10 <math>\pm</math> 0.09</b>	8.1 $\pm$ 0.2	16.26
	10%	<b>4.89 <math>\pm</math> 0.05</b>	8.6 $\pm$ 0.1	16.28
Lorenz96 (64 dim)	1%	<b>8.4 <math>\pm</math> 0.1</b>	12.63 $\pm$ 0.02	16.84
	5%	<b>9.04 <math>\pm</math> 0.07</b>	12.66 $\pm$ 0.03	16.81
	10%	<b>9.42 <math>\pm</math> 0.06</b>	12.72 $\pm$ 0.05	16.82
Task trained RNN	1%	<b>30.4 <math>\pm</math> 0.4</b>	38.565 $\pm$ 0.006	39.29
	5%	<b>29.4 <math>\pm</math> 0.9</b>	38.5611 $\pm$ 0.0004	38.91
	10%	<b>36.0 <math>\pm</math> 0.1</b>	38.5600 $\pm$ 0.0004	38.70

$$\mathbf{f}(\mathbf{x}(t)) = \frac{G(t_0, t; \mathbf{J}, \mathbf{c}) + \mathbf{x}(t) - \mathbf{x}(t_0)}{t - t_0}, \quad (14)$$

795 where

$$G(t_0, t; \mathbf{J}, \mathbf{c}) = \int_{t_0}^t \int_s^t \mathbf{J}(\mathbf{c}_{s,t}(r)) \mathbf{c}'_{s,t}(r) dr ds \quad (15)$$

796 Alternatively, one could use Equation 13 and learn  $\mathbf{f}(\mathbf{x}(t_i))$  and  $\mathbf{x}(t_i)$  as learnable parameters ( $\mathbf{x}(t_i)$  is  
797 needed as the first point of the path, i.e.  $\mathbf{c}(t_i) = \mathbf{x}(t_i)$  inside the integral) [61]. Here, the path integral  
798 between  $\mathbf{x}(t_i)$  and  $\mathbf{x}(t_f)$  is a linear interpolation, as before. These ablated models were trained on  
799 10 time-step prediction, as with the original JacobianODEs. For these models, we completed a full  
800 sweep over the loop closure loss weight  $\lambda_{\text{loop}}$  in order to determine the best hyperparameter.

801 **Ablating teacher forcing** We trained models without any teacher-forcing. That is, models were  
802 able to generate only one-step predictions, without any recursive predictions. Again we did a full  
803 hyperparameter sweep to pick  $\lambda_{\text{loop}}$ .

804 **Ablating loop closure loss** We ablated the loop closure loss in two ways. The first was to set  
805  $\lambda_{\text{loop}} = 0$  to illustrate what would happen if there were no constraints placed on the learned Jacobians.  
806 The second was to instead use the Jacobian Frobenius norm regularization that was used for the  
807 NeuralODEs (details are in appendix D.3). We did a full sweep to pick  $\lambda_{\text{jac}}$ , the Frobenius norm  
808 regularization weight.

809 **Ablation results** The performance of the ablated models on Jacobian estimation in the Lorenz  
810 system are presented in Table 3. The original JacobianODE outperforms all ablated models, indicating  
811 that all components of the JacobianODE training framework improve the model’s performance in  
812 this setting. Ablating the Jacobian-parameterized initial derivative estimate resulted in a slight  
813 decrease in the estimation loss. This is potentially because the network could offload some of the  
814 responsibility for generating correct trajectory predictions onto the estimated base point  $\mathbf{x}(t_i)$  and  
815 derivative estimate  $\hat{\mathbf{f}}(\mathbf{x}(t_i))$ , slightly reducing the necessity of estimating correct Jacobians. Ablating  
816 the teacher forcing annealing predictably led to worse Jacobian estimation, as the network no longer  
817 has to consider how errors will propagate along the trajectory. The most dramatic increase in error

Table 3: Mean Frobenius norm error  $\langle \|\mathbf{J} - \hat{\mathbf{J}}\|_F \rangle$  for different model ablations on the Lorenz system with 10% observation noise. Errors are reported as mean  $\pm$  standard deviation, with statistics computed over 8 test trajectories, each consisting of 1200 points.

Model variant	Frobenius norm error
JacobianODE (original)	<b>6.46 <math>\pm</math> 1.50</b>
With learned base derivative point	7.87 $\pm$ 1.14
No teacher forcing	11.59 $\pm$ 1.30
No loop closure	58.22 $\pm$ 2.00
With Jacobian penalty instead of loop closure	9.59 $\pm$ 2.45

was with the ablation of the loop closure loss. Without this important regularization, the learned Jacobians reproduced the dynamics but were not constrained to be conservative, resulting in poor Jacobian estimation. The inclusion of the Frobenius penalty on the Jacobians mitigated this, although it did not encourage accurate Jacobian estimation to the same degree as the loop closure loss.

Table 4: Mean Frobenius norm error  $\langle \|\mathbf{J} - \hat{\mathbf{J}}\|_F \rangle$  for different model ablations on the task-trained RNN with 10% observation noise. Errors are reported as mean  $\pm$  standard deviation, with statistics computed over 409 test trajectories, each consisting of the 49 points from the second delay and response epochs.

Model variant	Frobenius norm error
JacobianODE (original)	180.13 $\pm$ 1.55
With learned base derivative point	186.28 $\pm$ 1.17
No teacher forcing	187.72 $\pm$ 1.63
No loop closure	313.31 $\pm$ 29.88
With Jacobian penalty instead of loop closure	<b>163.69 <math>\pm</math> 4.22</b>

We then tested the ablated the models on Jacobian estimation in the task-trained RNN, with results presented in Table 4. Again, ablating the Jacobian-parameterized derivative estimates, teacher forcing, and loop closure resulted in worse Jacobian estimation. Interestingly, in this setting, the inclusion of a penalty on the Frobenius norm of the Jacobians outperformed the use of the loop closure loss. This could potentially be because the loop closure loss is more difficult to drive to zero in high dimensional systems, or because the loop closure loss is more important in chaotic systems like the Lorenz system considered above. Future work should consider in what contexts each kind of regularization is most beneficial to JacobianODE models.

### C.5 NeuralODEs achieve improved performance at the cost of increased inference time

In the main paper, we implemented both the JacobianODEs and the NeuralODEs as four-layer MLPs, with the four layers having sizes of 256, 1024, 2048, and 2048 respectively. This was done for the fairest architectural comparison between the models, to ensure that both models had the same representational capacity when generating their respective outputs. However, there are many architectural changes that we could make to this setup that impact performance. We hypothesized based on the discussion in appendix C.2 that increasing the hidden layer size of the NeuralODEs would improve Jacobian estimation, as larger models have been known to learn smoother representations. Furthermore, we wondered whether including residual blocks in place of the standard MLP implementation would improve Jacobian estimation.

To test this, we implemented the NeuralODEs as four-layer residual networks and tested three different sizes of hidden layer: 1024, 2048, and 4096. Results are in Table 5. For nearly all models, these changes yielded only marginal improvements over the original NeuralODE model. Only the model with 4096-dimensional hidden layers under 10% training noise achieves a performance near the *original* JacobianODEs. However, the performance limit is still below that of the JacobianODEs, even with a large increase in the representational capacity of the model. It is furthermore of note that the NeuralODEs were able to significantly improve performance only in the high noise setting. This suggests that high noise is necessary for the model to be forced to learn the response of the system to perturbation. In contrast, JacobianODEs perform similarly across all noise levels, indicating a stronger inductive bias to learn the response of the system to perturbation.



Table 5: Mean Frobenius norm error  $\langle \|\mathbf{J} - \hat{\mathbf{J}}\|_F \rangle$  for different model types on the task-trained RNN data across observation noise levels. Errors are reported as mean  $\pm$  standard deviation, with statistics computed over 409 test trajectories.

Model type	Training noise 5%	Training noise 10%	Learnable parameters
JacobianODE (original)	<b>174.1 <math>\pm</math> 2.4</b>	<b>180.1 <math>\pm</math> 1.5</b>	4.02e+07
NeuralODE (original)	294.0 $\pm$ 2.4	293.9 $\pm$ 2.4	6.85e+06
NeuralODE (1024 dim. hidden layers)	291.6 $\pm$ 2.3	289.8 $\pm$ 2.3	3.41e+06
NeuralODE (2048 dim. hidden layers)	288.5 $\pm$ 2.3	275.0 $\pm$ 2.7	1.31e+07
NeuralODE (4096 dim. hidden layers)	275.8 $\pm$ 2.4	184.2 $\pm$ 5.6	5.14e+07

While the increased hidden layer size seems to induce smoother hidden representations (as expected), it comes with increasing computational cost. Recall that the NeuralODE models directly parameterize the dynamics as  $\dot{\mathbf{x}} = \hat{\mathbf{f}}^\theta(\mathbf{x})$ . The Jacobian at state  $\mathbf{x}$  is computed by directly backpropagating through the Neural ODE  $\hat{\mathbf{f}}^\theta$ , thereby significantly increasing compute. On the other hand, the JacobianODEs only require a forward pass. We evaluated the Jacobian inference time of JacobianODE and NeuralODE models with four hidden layers of the same size on an H100 GPU. Each model was timed on inferring the Jacobians of 100 batches of the 128-dimensional task-trained RNN, with each batch containing 16 sequences of length 25. Timings were repeated ten times for each model. Results are plotted in Figure S3.

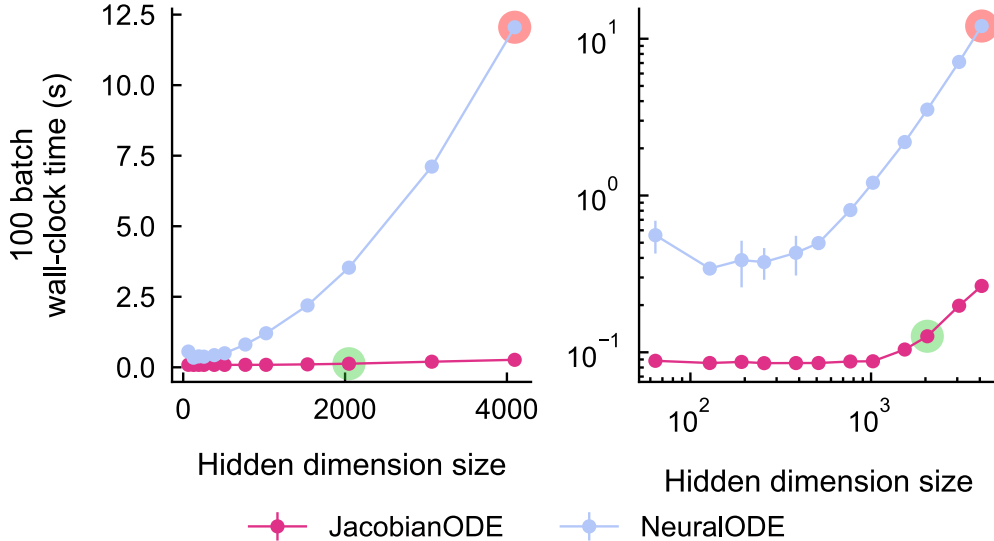


Figure S3: **JacobianODEs achieve highly efficient Jacobian inference.** Jacobian inference times computed over ten repetitions of 100 batches (error bars indicate mean  $\pm$  standard deviation). Red circles indicate the inference time corresponding to the largest hidden layer dimension of the highest performing NeuralODE model in Table 5. Green circles indicate the inference time corresponding to the largest hidden layer dimension of the highest performing JacobianODE model in Table 5. Each plot illustrates the same data but with different x and y scaling.

The JacobianODE achieves much faster inference times than the NeuralODE – approximately two orders of magnitude faster at large hidden dimension sizes. Furthermore, as shown in Table 5, the JacobianODE with a maximum hidden layer size of 2048 outperforms the NeuralODE with a maximum hidden layer size of 4096 on Jacobian estimation, and does so with orders of magnitude faster inference (Figure S3, green and red circles, Table 5). This suggests that while both architectures appear to have inference times that scale approximately exponentially, the JacobianODE achieves more favorable scaling across every hidden layer size we tested. Our analysis therefore illustrates that it is possible to improve the NeuralODEs' Jacobian estimation with larger models, but the inference time scaling renders these models ill-equipped for important settings such as real-time control and the analysis of high-volume neural data.

## D Experimental details

### D.1 Dynamical systems data

We used the following dynamical systems for the testing and validation of Jacobian estimation.

**Van der Pol oscillator** We implement the classic Van der Pol oscillator [64]. The system is governed by the following equations

$$\begin{aligned}\dot{x} &= y \\ \dot{y} &= \mu(1 - x^2)y - x\end{aligned}$$

We pick  $\mu = 2$  in our implementation.

**Lorenz** We implement the Lorenz system introduced by Lorenz [65] as

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z\end{aligned}$$

with the typical choices for parameters ( $\sigma = 10, \rho = 28, \beta = 8/3$ ).

**Lorenz96** We implement the Lorenz96 system introduced by Lorenz [66] and defined by

$$\dot{x}_i = (x_{(i+1) \pmod N} - x_{(i-2) \pmod N})x_{(i-1) \pmod N} - x_i \pmod N + F$$

with  $F = 8$  and  $N \in \{12, 32, 64\}$ .

**Simulation** We use the `dysts` package to simulate all dynamical systems [67, 68]. The characteristic timescale of their Fourier spectrum  $\tau$  is selected and the systems are sampled with respect to  $\tau$ . For all systems, the training data consisted of 26 trajectories of 12 periods, sampled at 100 time steps per  $\tau$ . The validation data consisted of 6 trajectories of 12 periods sampled at 100 time steps per  $\tau$ . The test data consisted of 8 trajectories of 12 periods sampled at 100 time steps per  $\tau$ . Trajectories were initialized using a random normal distribution with standard deviation 0.2. The simulation algorithm used was Radau. Batches were constructed by moving a sliding window along the signal. The sequence length was selected such that the generated predictions would generate 10 novel time points (i.e., 11 time steps for the NeuralODE, and 25 time steps for the JacobianODE, due to the 15 time steps used to estimate the initial time derivative  $\mathbf{f}$ ).

**Noise** We define  $P\%$  observation noise with  $P = 100p$  in the following way. Let  $A_{\text{signal}} = \mathbb{E}[\|x(t)\|_2^2]$  be the expected squared norm of the signal with  $\mathbf{x}(t) \in \mathbb{R}^n$ . Then consider a noise signal  $\eta(t) \in \mathbb{R}^n$  where each component  $\eta_i(t) \sim \mathcal{N}(0, \frac{1}{\sqrt{n}}p\sqrt{A_{\text{signal}}})$ . Then

$$\mathbb{E}[\|\eta(t)\|_2^2] = \mathbb{E}\left[\sum_{i=1}^n \eta_i^2(t)\right] = \sum_{i=1}^n \mathbb{E}[\eta_i^2(t)] = \sum_{i=1}^n \frac{1}{n} p^2 A_{\text{signal}} = p^2 A_{\text{signal}}$$

and thus the noise percent is

$$\sqrt{\frac{\mathbb{E}[\|\eta(t)\|_2^2]}{\mathbb{E}[\|x(t)\|_2^2]}} = \sqrt{\frac{p^2 A_{\text{signal}}}{A_{\text{signal}}}} = p$$

### D.2 Task-trained RNN

The task used to train the RNN was exactly as defined in section 5. The hidden dimensionality of the RNN was 128, and the input dimensionality was 10, where the first four dimensions represented the one-hot encoded "upper" color, the second four dimensions represented the one-hot encoded "lower" color, and the last two dimensions represented the one-hot encoded cue. The RNN used for the task had hidden dynamics defined by

$$\begin{aligned}\tau \dot{\mathbf{h}}(t) &= -\mathbf{h} + \mathbf{W}_{hh}\sigma(\mathbf{h}(t)) + \mathbf{W}_{hi}\mathbf{u}(t) + \mathbf{b} \\ \mathbf{o}(t) &= \mathbf{W}_{oh}\mathbf{h}(t)\end{aligned}$$

with  $\tau = 50$  ms, which for the purposes of training was discretized with Euler integration with a time step of  $\Delta t = 20$  ms.  $\mathbf{W}_{hh}$  is the 128x128 dimensional matrix that defines the internal dynamics,  $\mathbf{W}_{hi}$  is the 128 x 10 dimensional matrix that maps the input into the hidden state,  $\mathbf{W}_{oh}$  is the 4x128 dimensional output matrix that maps the hidden state a four dimensional output  $\mathbf{o}(t)$ , and  $\mathbf{b}$  is a static bias term.  $\sigma$  was taken to be an exponential linear unit activation with  $\alpha = 1$ . The RNN hidden state was split into two "areas" each with 64 dimensions. The input matrix  $\mathbf{W}_{hi}$  was masked during training so that inputs could only flow into the first 64 dimensions – the "visual" area. The same procedure was performed for the output matrix  $\mathbf{W}_{oh}$ , except the mask was such that outputs could stem only from the second group of 64 dimensions – the "cognitive" area. The within-area subblocks of the matrix  $\mathbf{W}_{hh}$  were first initialized such that the real part of the eigenvalues were randomly distributed on the interval  $[-0.1, 0]$  and the imaginary part of the eigenvalues were randomly distributed on the interval  $[0, 2\pi]$ . The eigenvectors were random orthonormal matrices. We then computed the matrix exponential of this matrix. The across-area weights were first initialized to be random normal, then divided by the 2-norm of the resulting matrix and multiplied by 0.05. The input and output matrices were initialized as random normal and then scaled by the 2-norm of the resulting matrix. The static bias  $\mathbf{b}$  was initialized at  $\mathbf{0}$ . After initialization, all weights could be altered unimpeded (except for the masks). Notably, the inputs  $\mathbf{u}(t)$  were present only during the stimulus and cue presentation epochs – otherwise the network evolved autonomously. The loss was computed via cross entropy loss on the RNN outputs during the response period (the final 250 ms of the trial).

For the training data, we generated 4096 random trials, and used 80% for training and the remainder for validation. The batch size used was 32. Training was performed for 40 epochs. The learning rate was 0.0005. For use with the Jacobian estimation models, data was batched and used for training exactly as was done with the other dynamical systems data (see appendix D.1). Observation noise was also computed in the same way.

### D.3 NeuralODE details

NeuralODE models directly estimate the time derivative  $\mathbf{f}$  with a neural-network parameterized function  $\hat{\mathbf{f}}^\theta$ . Then the Jacobians can be computed as  $\hat{\mathbf{J}} = \frac{\partial}{\partial \mathbf{x}} \hat{\mathbf{f}}^\theta$ .

The NeuralODEs were implemented as described in section 4 and ODE integration was done exactly as for the JacobianODE using the `torchdiffeq` package with the RK4 method [70]. To regularize the NeuralODE we implemented a Frobenius norm penalty on the estimated Jacobians, i.e.

$$\mathcal{L}_{\text{jac}} = \lambda_{\text{jac}} \langle \left\| \hat{\mathbf{J}}(\mathbf{x}(t)) \right\|_F^2 \rangle$$

where  $\hat{\mathbf{J}}$  is the estimated Jacobian computed via automatic differentiation and  $\lambda_{\text{jac}}$  is a hyperparameter that controls the relative weighting of the Jacobian penalty. As mentioned in the main text, this penalty prevents the model from learning unnecessarily large eigenvalues and encourages better Jacobian estimation.

### D.4 Weighted linear Jacobian details

We implemented a baseline Jacobian estimation method using weighted linear regression models as described in Deyle et al. [72]. Given a reference point  $\mathbf{x}(t^*)$  at which the (discrete) Jacobian will be computed, all other points are weighted according to

$$w_k = \exp \frac{-\theta \|\mathbf{x}(t_k) - \mathbf{x}(t^*)\|}{\bar{d}}$$

where

$$\bar{d} = \sum_{i=1}^T \|\mathbf{x}(t_i) - \mathbf{x}(t^*)\|$$

is the average distance from  $\mathbf{x}(t^*)$  to all other points. We then perform a linear regression using the weighted points (and a bias term), the result of which is an estimate of the discrete Jacobian at  $\mathbf{x}(t^*)$ ,

which can be converted to continuous time by subtracting the identity matrix and dividing by the sampling time step (i.e.,  $\hat{\mathbf{J}} = \frac{\hat{\mathbf{J}}_{\text{discrete}} - \mathbf{I}}{\Delta t}$ , where  $\hat{\mathbf{J}}_{\text{discrete}}$  is the discrete Jacobian). The parameter  $\theta$  tunes how strongly the regression is weighted towards local points. To pick  $\theta$ , we sweep over values range from 0 to 10, and pick the value that yields the best one-step prediction according to

$$\mathbf{x}(t^* + 2\Delta t) = \mathbf{x}(t^* + \Delta t) + e^{\hat{\mathbf{J}}\Delta t}(\mathbf{x}(t^* + \Delta t) - \mathbf{x}(t^*))$$

where  $\hat{\mathbf{J}}$  is the estimated Jacobian. This form of prediction has been previously been used to learn Jacobians in machine learning settings [59]. To test the method, we pick  $\theta$  based on data with observation noise at a particular noise level, then add in the denoised data to the data pool in order to compute regressions and estimate Jacobians at the true points.

## D.5 Model details

All models were implemented as four-layer MLPs, with the four layers having sizes of 256, 1024, 2048, and 2048 respectively. All models used a sigmoid linear unit activation. JacobianODE models output to the dimension  $n^2$  which was then reshaped into the matrix of the appropriate dimension. NeuralODEs output to the dimension  $n$ .

## D.6 Lyapunov spectrum computation

To compute the Lyapunov spectrum, we employ a QR based algorithm [116, 117]. We discretize the Jacobians using the matrix exponential (i.e.,  $\hat{\mathbf{J}}_{\text{discrete}} = e^{\hat{\mathbf{J}}\Delta t}$ ) and then propagate a bundle of small vectors through the Jacobians, using QR to ensure the perturbations remain bounded.

## D.7 ILQR

We implement the standard algorithm for ILQR, the details of which can be found in Li and Todorov [75] and Tassa et al. [76]. In brief, the ILQR algorithm linearizes the system dynamics around a nominal trajectory using the Jacobian, and then iteratively optimizes the control sequence using forward and backward passes to minimize the total control cost. The state cost matrix  $\mathbf{Q}$  was a diagonal matrix with 1.0 along the diagonal. The final state cost matrix  $\mathbf{Q}_f$  was a diagonal matrix with 1.0 along the diagonal. The control cost matrix  $\mathbf{R}$  was a diagonal matrix with 0.01 along the diagonal. The control matrix was a  $128 \times 128$  matrix in which the  $64 \times 64$  block corresponding to the first 64 neurons (the "visual" area) was the 64-dimensional identity matrix. The control algorithm was seeded with only the initial state of the test trajectory with 5% noise. The control sequence was initialized random normal with standard deviation 0.001 and mean 0. The ILQR algorithm was run for a max of 100 iterations. The regularization was initialized at 1.0, with a minimum of  $1 \times 10^{-6}$  and a maximum of  $1 \times 10^{10}$ .  $\Delta_0$  was set to 2, as in Tassa et al. [76]. If the backward pass failed 20 times in a row, the optimization was stopped. The list of values for the line search parameter  $\alpha$  was  $1.1^{-k^2}$  for  $k \in 0, \dots, 9$  (see Tassa et al. [76]). The linear model used for the linear baseline was computed via linear regression.

## D.8 Training details

All models were implemented in PyTorch. The batch size used was 16. Training epochs were limited to 500 shuffled batches. Validation epochs were limited to 100 randomly chosen batches. Testing used all testing data. Training was run for a maximum of 1000 epochs, 3 hours, or until the early stopping was activated (see appendix D.8.6), whichever came first.

### D.8.1 Generalized Teacher Forcing

The Jacobian can be best learned when training predictions are generated recursively (i.e., replacing  $\mathbf{x}(t)$  by  $\hat{\mathbf{x}}(t)$ ). However, in chaotic systems, and/or systems with measurement noise (as considered here), this could lead to catastrophic divergence of the predicted trajectory from the true trajectory during training. We therefore employ Generalized Teacher Forcing when training all models [62]. Generalized Teacher Forcing prevents catastrophic divergence by forcing the generated predictions along the line from the prediction to the true state. Specifically, for a given predicted state  $\hat{\mathbf{x}}(t)$  and

985 true state  $\mathbf{x}(t)$ , the teacher forced state is

$$\tilde{\mathbf{x}}(t) = (1 - \alpha)\hat{\mathbf{x}}(t) + \alpha\mathbf{x}(t)$$

986 with  $\alpha \in [0, 1]$ . This effectively forces the predictions along a line from the prediction to the true  
 987 state, by an amount with proportion  $\alpha$ .  $\alpha = 1$  corresponds to fully-forced prediction (i.e., one-step  
 988 prediction) and  $\alpha = 0$  corresponds to completely unforced prediction (i.e., autonomous prediction).  
 989 Hess et al. [62] suggested that a good estimate of  $\alpha$  is

$$\alpha = \max \left( \max_p \left[ 1 - \frac{1}{\|\mathcal{G}(\mathbf{J}_{T:2}^{(p)})\|} \right], 0 \right) \quad (16)$$

990 where  $\mathbf{J}_{T:2}^{(p)}$  are the Jacobians of the modeled dynamics computed at data-constrained states,  $p$   
 991 indicates the batch or sequence index, and

$$\|\mathcal{G}(\mathbf{J}_{T:2}^{(p)})\| = \left\| \left( \prod_{k=0}^{T-2} \mathbf{J}_{T-k} \right)^{\frac{1}{T-1}} \right\|$$

992 effectively computes the discrete maximum Lyapunov exponent. In our implementation, we compute  
 993  $\|\mathcal{G}(\mathbf{J}_{T:2}^{(p)})\|$  using a QR-decomposition-based Lyapunov computation algorithm [116]. As the Jacobian  
 994 of the dynamics is necessary to compute this quantity, the JacobianODEs enjoy an advantage over  
 995 other models in that the Jacobians are directly output by the model, and do not have to be computed  
 996 via differentiating the model itself.

997 We furthermore employ a slightly modified version of the suggested annealing process in Hess et al.  
 998 [62], which sets  $\alpha_0 = 1$  and updates  $\alpha_n$  as

$$\alpha_n = \gamma\alpha_{n-1} + (1 - \gamma)\alpha$$

999 where  $\alpha$  is computed according to equation [16]. Following the suggested hyperparameters, we set  
 1000  $\gamma = 0.999$  and update  $\alpha_n$  every 5 batches. Once the teacher forced state  $\tilde{\mathbf{x}}(t)$  is computed, it can  
 1001 simply replace  $\mathbf{x}(t)$  in equation [12] to generate predictions.

## 1002 D.8.2 Loop closure loss

1003 We implemented a loop closure loss as defined in section [3.2]. For each loop, we used 20 randomly  
 1004 chosen points from the batch. For each batch, we constructed the same number of loops as there were  
 1005 batches. Path integrals were discretized in 20 steps and computed using the trapezoid method from  
 1006 torchquad [69].

## 1007 D.8.3 Validation loss

1008 All models were validated on 10 time-step prediction task with teacher forcing parameter  $\alpha = 0$  (i.e.,  
 1009 autonomous prediction).

## 1010 D.8.4 Learning rate scheduling

1011 For all models, the learning rate was annealed in accordance with teacher forcing annealing. Given  
 1012 an initial and final learning rates  $\eta_i$  and  $\eta_f$  we compute the effective learning rate as

$$\eta = \eta_f + \sigma(\alpha_n)(\eta_i - \eta_f)$$

1013 where  $\alpha_n$  is the current value of the teacher forcing parameter and

$$\sigma(\alpha_n) = \frac{\alpha_n}{\alpha_n + (1 - \alpha_n)e^{-k\alpha_n}}$$

1014  $\sigma(\alpha_n)$  is a scaling function with  $\sigma(1) = 1$  and  $\sigma(0) = 0$  and for which the shape of the scaling is  
 1015 controlled by the parameter  $k$ . For positive values of  $k$ , the scaling is super-linear, and for negative  
 1016 values of  $k$  it is sub-linear. We use  $k = 1$ , ensuring that the learning rate does not decrease too  
 1017 quickly at the start of learning. We set  $\eta_i = 10^{-4}$  and  $\eta_f = 10^{-6}$  for all models.

### 1018 D.8.5 Optimizer and weight decay

1019 All models were trained with PyTorch’s AdamW optimizer with the learning rate as described above,  
1020 and weight decay parameter  $10^{-4}$ . All other parameters were default ( $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ,  $\epsilon =$   
1021  $10^{-8}$ ).

### 1022 D.8.6 Early stopping

1023 For all models, we implemented an early stopping scheme that halted the training if the validation  
1024 loss improved by less than 1% for two epochs in a row.

### 1025 D.8.7 Added noise during learning

1026 For the models trained on the task-trained RNN dynamics, we added 5% Gaussian i.i.d. noise (defined  
1027 relative to the norm of the training data with observation noise already added). Noise was sampled  
1028 for each batch and added prior to the trajectory generation step of the learning process. Additional  
1029 noise was not added for the loop closure computation.

### 1030 D.8.8 Hyperparameter selection

1031 For the JacobianODEs, the primary hyperparameter to select is the loop closure loss  
1032  $\lambda_{\text{loop}}$ . To select this hyperparameter, we trained JacobianODE models with  $\lambda_{\text{loop}} \in$   
1033  $[0, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10]$ . For each run, the epoch with the lowest trajectory  
1034 validation loss ( $\mathcal{L}_{\text{traj}}$ ) is kept. Then, for this model, we compute the one-step prediction error on  
1035 validation data, the validation loop closure loss ( $\mathcal{L}_{\text{loop}}$ ), and the percentage of Jacobian eigenvalues  
1036 on all validation data that have a decay rate faster than the sampling rate  $\frac{1}{\Delta t}$ . We exclude any models  
1037 that meet any of the following criteria:

- 1038 1. **One-step prediction error greater than the persistence baseline.** The persistence baseline  
1039 is computed as the mean error between each time step  $k\Delta t$  and the subsequent time step  
1040  $(k+1)\Delta t$  across the dataset, and constitutes a sanity check for whether a model is capturing  
1041 meaningful information about the dynamics.
- 1042 2. **Loop closure loss greater than  $\sqrt{n}$ ,** where  $n$  is the system dimension (see appendix [D.11](#)  
1043 for the derivation of this bound). As discussed in the main text, we are interested in Jacobians  
1044 that not only solve the trajectory prediction problem, but that also are constructed so that the  
1045 rows of the matrix are approximately conservative vector fields.
- 1046 3. **More than 0.1% of the Jacobian eigenvalues have a decay rate faster than the sampling**  
1047 **rate  $\frac{1}{\Delta t}$ .** Since large negative eigenvalues do not impact trajectory prediction, the models  
1048 may erroneously learn Jacobians with large negative eigenvalues. If the decay rate of these  
1049 eigenvalues is faster than the sampling rate, we can infer that the eigenvalues are not aligned  
1050 with the observed data.

1051 If none of the models that meet criterion (2) meet criterion (1), we discount criterion (2), as this  
1052 suggests that a loop closure loss below  $\sqrt{n}$  bound is too strict to obtain good prediction on this  
1053 system. Additionally, if none of the models that meet criterion (3) meet criterion (1), we discount  
1054 criterion (1), as this suggests that noise is very high in the data, which leads to both high one-step  
1055 prediction error, and large negative eigenvalues to compensate for the perturbations introduced by the  
1056 noise. Of the remaining models, we select the one with the lowest trajectory validation loss  $\mathcal{L}_{\text{traj}}$ .

1057 For the NeuralODEs, we needed to select the hyperparameter  $\lambda_{\text{jac}}$ , which regularized the mean frobe-  
1058 nius norm of the Jacobians computed through automatic differentiation. Again, to select this hyperpa-  
1059 rameter, we trained JacobianODE models with  $\lambda_{\text{jac}} \in [0, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10]$ .  
1060 We followed exactly the above procedure with the exception of criterion (2), which was deemed  
1061 unnecessary, as computing the Jacobians implicitly via a gradient of the model (using automatic  
1062 differentiation) ensures that the rows of the matrix are conservative.

1063 All other hyperparameters (model size, learning rate, length of initial trajectory, number of discretiza-  
1064 tion steps, etc.) were fixed for all systems. Given the wide range of systems and behaviors and  
1065 dimensionalities that the JacobianODEs are capable of capturing, this indicates that the method is  
1066 robust given a reasonable choice of these hyperparameters.

## D.8.9 Model hyperparameters and training details

Below are presented the details of all Jacobian estimation models considered in the main paper.

Table 6: Hyperparameters used and model details for each system and noise level. Training time is reported in seconds.

System	Noise	Model	Loop closure weight	Jacobian penalty	Training time (s)	Final epoch	Learning rate	Min learning rate	Weight decay	Learnable parameters
VanDerPol (2 dim)	1%	JacobianODE	0.0010	0	882.139	15	0.0001	1.00e-06	0.0001	6.568e+06
	1%	NeuralODE	0	1.00e-06	949.217	21	0.0001	1.00e-06	0.0001	6.564e+06
	5%	JacobianODE	0.0001	0	692.980	11	0.0001	1.00e-06	0.0001	6.568e+06
	5%	NeuralODE	0	0	249.451	6	0.0001	1.00e-06	0.0001	6.564e+06
	10%	JacobianODE	0.010	0	678.794	10	0.0001	1.00e-06	0.0001	6.568e+06
	10%	NeuralODE	0	0.0010	713.221	16	0.0001	1.00e-06	0.0001	6.564e+06
Lorenz (3 dim)	1%	JacobianODE	0.0010	0	972.268	16	0.0001	1.00e-06	0.0001	6.578e+06
	1%	NeuralODE	0	0.0010	889.819	18	0.0001	1.00e-06	0.0001	6.566e+06
	5%	JacobianODE	0.010	0	1.147e+03	18	0.0001	1.00e-06	0.0001	6.578e+06
	5%	NeuralODE	0	0.0010	680.651	15	0.0001	1.00e-06	0.0001	6.566e+06
	10%	JacobianODE	0.010	0	388.346	6	0.0001	1.00e-06	0.0001	6.578e+06
	10%	NeuralODE	0	0.010	421.488	8	0.0001	1.00e-06	0.0001	6.566e+06
Lorenz96 (12 dim)	1%	JacobianODE	0.0010	0	1.817e+03	31	0.0001	1.00e-06	0.0001	6.857e+06
	1%	NeuralODE	0	1.00e-06	1.892e+03	32	0.0001	1.00e-06	0.0001	6.587e+06
	5%	JacobianODE	0.0010	0	716.408	12	0.0001	1.00e-06	0.0001	6.857e+06
	5%	NeuralODE	0	0.0010	1.147e+03	19	0.0001	1.00e-06	0.0001	6.587e+06
	10%	JacobianODE	0.010	0	1.213e+03	18	0.0001	1.00e-06	0.0001	6.857e+06
	10%	NeuralODE	0	0.0001	851.803	14	0.0001	1.00e-06	0.0001	6.587e+06
Lorenz96 (32 dim)	1%	JacobianODE	0.010	0	5.160e+03	85	0.0001	1.00e-06	0.0001	8.665e+06
	1%	NeuralODE	0	0.0010	4.204e+03	43	0.0001	1.00e-06	0.0001	6.633e+06
	5%	JacobianODE	0.0010	0	1.341e+03	22	0.0001	1.00e-06	0.0001	8.665e+06
	5%	NeuralODE	0	0.0010	3.855e+03	39	0.0001	1.00e-06	0.0001	6.633e+06
	10%	JacobianODE	0.0001	0	868.262	14	0.0001	1.00e-06	0.0001	8.665e+06
	10%	NeuralODE	0	0.0010	2.695e+03	28	0.0001	1.00e-06	0.0001	6.633e+06
Lorenz96 (64 dim)	1%	JacobianODE	0.0010	0	2.749e+03	40	0.0001	1.00e-06	0.0001	1.497e+07
	1%	NeuralODE	0	0.010	4.801e+03	30	0.0001	1.00e-06	0.0001	6.706e+06
	5%	JacobianODE	0.0001	0	1.748e+03	27	0.0001	1.00e-06	0.0001	1.497e+07
	5%	NeuralODE	0	0.010	4.879e+03	30	0.0001	1.00e-06	0.0001	6.706e+06
	10%	JacobianODE	0.0010	0	1.701e+03	25	0.0001	1.00e-06	0.0001	1.497e+07
	10%	NeuralODE	0	0.010	4.237e+03	26	0.0001	1.00e-06	0.0001	6.706e+06
Task-trained RNN	1%	JacobianODE	0.0001	0	2.496e+03	32	0.0001	1.00e-06	0.0001	4.016e+07
	1%	NeuralODE	0	0	9.777e+03	38	0.0001	1.00e-06	0.0001	6.854e+06
	5%	JacobianODE	0.0001	0	2.352e+03	29	0.0001	1.00e-06	0.0001	4.016e+07
	5%	NeuralODE	0	1.00e-05	7.770e+03	27	0.0001	1.00e-06	0.0001	6.854e+06
	10%	JacobianODE	0.010	0	2.172e+03	27	0.0001	1.00e-06	0.0001	4.016e+07
	10%	NeuralODE	0	1.00e-05	5.260e+03	18	0.0001	1.00e-06	0.0001	6.854e+06

## D.9 Information about computing resources and efficiency

All models were able to be trained on a single H100 GPU, with 80 GB of memory.

**Jacobian inference times** Jacobian inference times for the JacobianODE and NeuralODE models are discussed in appendix C.5. As discussed in that section models were implemented with four hidden layers of the same size, and tested on 100 batches of the 128-dimensional task-trained RNN data, with each batch consisting of 16 sequences of length 25. Timings were repeated ten times for each model. See section and Figure S3 for details.

**Training time** Total training times for each of the chosen models are presented in Table 6. Furthermore, we include the training time (including backward pass) for 100 batches (with 16 sequences per batch), using 10 time-step prediction, in Table 7.

Table 7: Trajectory training time (seconds) for each system and noise level.

Model	Lorenz (3 dim)	VanDerPol (2 dim)	Lorenz96 (12 dim)	Lorenz96 (32 dim)	Lorenz96 (64 dim)	Task trained RNN
1% noise						
JacobianODE	15.772	13.737	11.604	16.469	16.214	23.855
NeuralODE	8.215	8.403	11.852	12.075	18.192	14.955
5% noise						
JacobianODE	12.422	16.772	10.655	13.121	12.949	23.197
NeuralODE	6.191	5.841	13.501	8.021	21.307	30.209
10% noise						
JacobianODE	12.590	15.719	18.447	10.506	18.900	22.482
NeuralODE	11.083	9.875	15.269	8.008	17.826	33.074

## D.10 Statistical details

All statistics were computed using `scipy`. For the comparison between JacobianODE and NeuralODE trajectory and Jacobian predictions, as well as the comparison of Gramian traces and minimum



1082 eigenvalues, we used a two-sample t-test. For the comparison of ILQR control accuracies and errors,  
 1083 we used a Wilcoxon signed-rank test.

#### 1084 **D.11 Derivation of loop closure loss bound**

1085 We consider the loop closure loss as defined in [3.2](#). We are interested in estimating a bound on the  
 1086 error

$$\mathbb{E} \left[ \frac{1}{n} \left\| \int_{\mathcal{C}_{\text{loop}}^{(l)}} \mathbf{J} ds \right\|_2^2 \right]$$

1087 where  $n$  is the system dimension. While in theory this quantity should be 0, in practice due to  
 1088 numerical estimation error, it will not be. First recall, that

$$\int_{\mathcal{C}_{\text{loop}}^{(l)}} \mathbf{J} ds = \sum_{i=1}^L \int_{\mathbf{x}(t_i \pmod{L})}^{\mathbf{x}(t_{i+1} \pmod{L})} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr$$

1089 where  $L$  is the number of loop points and  $\mathbf{c}$  is a line from  $\mathbf{x}(t_i \pmod{L})$  to  $\mathbf{x}(t_{i+1} \pmod{L})$ . We  
 1090 assume that

$$\mathbb{E} \left[ \left\| \int_{\mathbf{x}(t_i \pmod{L})}^{\mathbf{x}(t_{i+1} \pmod{L})} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] = \mathbb{E} \left[ \left\| \int_{\mathbf{x}(t_j \pmod{L})}^{\mathbf{x}(t_{j+1} \pmod{L})} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right]$$

1091  $\forall i, j \in [1, \dots, L]$ , which is justified as the numerical error accrued will likely be similar along different  
 1092 lines for the same system. Thus

$$\begin{aligned} \mathbb{E} \left[ \frac{1}{n} \left\| \int_{\mathcal{C}_{\text{loop}}^{(l)}} \mathbf{J} ds \right\|_2^2 \right] &= \mathbb{E} \left[ \frac{1}{n} \left\| \sum_{i=1}^L \int_{\mathbf{x}(t_i \pmod{L})}^{\mathbf{x}(t_{i+1} \pmod{L})} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] \\ &\leq \frac{1}{n} \mathbb{E} \left[ \sum_{i=1}^L \left\| \int_{\mathbf{x}(t_i \pmod{L})}^{\mathbf{x}(t_{i+1} \pmod{L})} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] \\ &= \frac{1}{n} \sum_{i=1}^L \mathbb{E} \left[ \left\| \int_{\mathbf{x}(t_i \pmod{L})}^{\mathbf{x}(t_{i+1} \pmod{L})} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] \\ &= \frac{1}{n} \sum_{i=1}^L \mathbb{E} \left[ \left\| \int_{\mathbf{x}(t_0)}^{\mathbf{x}(t_1)} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] \\ &= \frac{L}{n} \mathbb{E} \left[ \left\| \int_{\mathbf{x}(t_0)}^{\mathbf{x}(t_1)} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] \end{aligned}$$

1093 where we have used the fact that the errors are assumed to be equivalent for each of the line segments  
 1094 comprising the overall loop path. Recall now that for the trapezoid integration rule, the error  $E$  in  
 1095 integrating  $\int_a^b f(x) dx$  can be computed as  $E = -\frac{(b-a)^3}{12M^2} f''(u)$  for some  $u \in [a, b]$ . Thus the squared  
 1096 error is bounded as

$$E^2 \leq \left\| \max_{u \in [a, b]} \frac{(b-a)^3}{12M^2} f''(u) \right\|_2^2$$

1097 In our case, when path integrating lines, we are effectively integrating from  $a = 0$  to  $b = 1$ .  
 1098 Furthermore, let  $\mathbf{g}(r) = \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r)$ , the integrand of our line integrations. We assume that  
 1099  $\mathbb{E} \left[ \|\max_r \mathbf{g}''(r)\|_2^2 \right] \propto n \cdot \sqrt{n}$  where the first  $n$  comes from the fact that the norm involves summing  
 1100 over the  $n$  components of the vector  $\mathbf{g}''(r)$  and the second  $\sqrt{n}$  involves an assumption that loop



1101 closure loss will be more difficult to compute accurately in higher dimensions, though this will be  
 1102 more pronounced as dimensionality initially starts increasing. Thus  $\mathbb{E} \left[ \|\max_r \mathbf{g}''(r)\|_2^2 \right] = kn \cdot \sqrt{n}$   
 1103 for some  $k \in \mathbb{R}^+$ . Now, continuing on,

$$\begin{aligned}
 \mathbb{E} \left[ \frac{1}{n} \left\| \int_{\mathcal{C}_{\text{loop}}^{(l)}} \mathbf{J} ds \right\|_2^2 \right] &\leq \frac{L}{n} \mathbb{E} \left[ \left\| \int_{\mathbf{x}(t_0)}^{\mathbf{x}(t_1)} \mathbf{J}(\mathbf{c}(r)) \mathbf{c}'(r) dr \right\|_2^2 \right] \\
 &\leq \frac{L}{n} \mathbb{E} \left[ \left\| \frac{1^3}{12M^2} \max_r \mathbf{g}''(r) \right\|_2^2 \right] \\
 &= \frac{L}{12^2 n M^4} \mathbb{E} \left[ \left\| \max_r \mathbf{g}''(r) \right\|_2^2 \right] \\
 &= \frac{Lkn \cdot \sqrt{n}}{12^2 n M^4} \\
 &= \frac{Lk\sqrt{n}}{12^2 M^4}
 \end{aligned}$$

1104 Finally, assuming that the number of discretization steps  $M$  was chosen to be large enough such that  
 1105  $M^4 \geq \frac{1}{12^2} Lk$  we finally obtain

$$\mathbb{E} \left[ \frac{1}{n} \left\| \int_{\mathcal{C}_{\text{loop}}^{(l)}} \mathbf{J} ds \right\|_2^2 \right] \leq \sqrt{n}$$

## 1106 D.12 Data and code availability

1107 The data for all analyzed systems can be downloaded [here](#). Code to implement all models, training,  
 1108 and control theoretic analyses are included with the Supplementary Material submission.