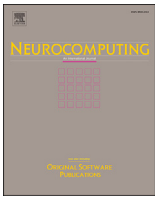
Neurocomputing 654 (2025) 131358

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom



Eigen-informed neural ordinary differential equations: Incorporating knowledge of system properties

Tobias Thummerer* , Lars Mikelsons

University of Augsburg, Am Technologiezentrum 8, Augsburg, 86159, Bavaria, Germany

HIGHLIGHTS

- Incorporate a priori knowledge of system properties from systems and control theory into the training process via differentiable eigenvalues.
- Regularize stability or stiffness during training of neural ODEs.
- Train neural ODEs on undersampled data by incorporating frequency and damping information.
- Improve convergence (up to factor of 6.41 in the median) and generalization (up to factor of 6.92 in the median).
- Learn ODE representations that are faster to solve (up to factor of 1.12 in the median) and require fewer solver steps.

ARTICLE INFO

Communicated by Q. Ni

Keywords:
Neural ordinary differential equation
Scientific machine learning
Eigenvalue
Characteristic value
Eigenmode
Stability
Convergence
Oscillation
Stiffness
Frequency
Damping

ABSTRACT

Using neural ordinary differential equations (ODEs) to model complex systems is still a challenging venture and fails for various reasons, often resulting in convergence to unsatisfactory local minima or unintended termination of the training process due to solver issues. To take a step back, the root of the problem lies in either the complexity of the optimization problem or the lack of data. The negative effects of both aspects can be reduced by incorporating a priori knowledge, a common strategy in *Scientific Machine Learning*. Especially when modeling physical systems, there is almost always more system knowledge available than is used for training. Examples range from knowledge of generic properties such as “the system is stable” to more quantifiable attributes such as “the system oscillates in a known frequency range”. To close the loop, such knowledge can be used to achieve faster convergence and better generalization. In this paper, we focus specifically on *system properties* that can be expressed based on eigenvalues, such as (partial) stability, oscillation capability, frequencies, damping, and stiffness. We explain how such properties can be intuitively integrated into training neural ODEs, provide open-source software for this, and finally show in three academic examples that such *eigen-informed neural ODEs* are able to converge in fewer steps (median up to factor 6.41), generalize better (median up to factor 6.92) and are solvable more efficiently (median up to factor 1.12) compared to pure neural ODEs, and can even reconstruct a given system on undersampled data.

1. Introduction

In this work, we present the method of *eigen-informed* training by extending a common loss function with additional objectives. These objectives are defined based on system properties that depend on eigenvalues. The eigenvalues are calculated for the system matrix (which is obtained by linearization) at multiple points of the solution of the ordinary differential equations (ODEs). By incorporating knowledge based on system properties, we will show that generalization and convergence are enhanced, which reduces the amount of required data. In addition, common pitfalls such as instability and stiffness can be prevented. The corresponding method is investigated in detail in Section 2. In the following, we introduce the general topic and investigate related work in the context of neural ODEs.

1.1. Neural ordinary differential equations

Neural ODEs describe the structural combination of an Artificial Neural Networks (ANNs) and an ODE solver, where the ANNs functions as the right-hand side of the ODEs [1]. With this approach, models of

* Corresponding author.
Email address: tobias.thummerer@uni-a.de (T. Thummerer).

https://doi.org/10.1016/j.neucom.2025.131358
Received 22 April 2025; Received in revised form 23 July 2025; Accepted 20 August 2025
Available online 23 August 2025

dynamic systems can be learned without learning the numerical solving process itself. This powerful combination led to impressive results in different applications in physics [2–5].

A straightforward training process for neural ODEs looks as follows: First, the neural ODE is solved like a common ODE by numerically integrating the right-hand side (here, the ANN). Next, the gathered solution is compared with some reference data (loss function), and finally the ANNs parameters inside the neural ODE are adapted based on the loss function gradient, so a better fit compared to the reference data is achieved.

If large or complex systems are modeled with neural ODEs, the corresponding ANNs become deeper and wider, and two major challenges must be faced: Solvability and convergence. These two challenges are often neglected in simple applications involving neural ODEs, or are handled in a very use case specific way that might not be reusable in other applications.

1.2. Solvability

Before investigating solvability, eigenvalues are briefly introduced in the context of neural ODEs. The eigenvalues of a neural ODE with state $\boldsymbol{x}$ and state derivative $\dot{\boldsymbol{x}}$ are defined as the eigenvalues of the system matrix $\boldsymbol{A} = \frac{\partial \dot{\boldsymbol{x}}}{\partial \boldsymbol{x}}$. Linearization for equilibrium points (where $\dot{\boldsymbol{x}} = \boldsymbol{0}$) is applicable if the *Hartman–Grobman Theorem* is satisfied [6,7], which applies if $\boldsymbol{A}$ has no eigenvalue with a zero real part. It is important to understand that linearization can only ever make a limited statement about the nonlinear system. This is reflected in the discrete evaluation in state space (the linearization is performed for a finite number of determined states), but also by the limited guarantees for how the system behaves in the immediate surroundings of the linearization points (e.g., chaotic or bifurcating systems). Hard statements on the overall trajectory of a nonlinear system cannot be made on the basis of linearization. On the other hand, the presented method is proposed as a regularization strategy — so a *soft* constraint — and small and temporary errors in regularization can still lead to an improved training process, as we show in the experiments, even if this must not be the case for all classes of nonlinear systems. A further discussion on this topic is part of the limitations Section 4.2 and future work 4.3.

A neural ODE is considered *solvable*, if the solver is able to keep all the eigenvalues of the system within its stability region during the construction of the ODE solution. The stability region is the area of eigenvalues in the complex plane, for which the solution of the Dahlquist equation absolutely decreases [8]. There are two options that allow the solver to keep eigenvalues within its stability region:

1. Step size control: The size of the solver's stability region scales inversely with the solver's step size. By taking smaller steps, the numerical solver can scale its stability region to cover all eigenvalues. However, this can still fail if eigenvalues are distributed over a wide range, far from the complex origin (stiff system) or are geometrically unreachable (stable solver and unstable system).
2. Order control: Solvers with dynamic order control (or hybrid solvers that switch between different solvers) are able to switch between different stability region geometries and sizes, depending on eigenvalue positions.

If a neural ODE can not be solved, there is a major issue resulting from that: If the solver terminates the solving process, no solution can be obtained, no loss can be computed, and no parameter updates can be performed. This state cannot be cured by a new solving attempt, because the parameter values and therefore the ODEs (and their solvability) remain unchanged. To prevent this, it is necessary to initialize neural ODEs in a solvable configuration to obtain a solution that can be used for training. But even if the neural ODE is initialized as solvable and the target solution is known to be solvable for the chosen solver, there is no guarantee that changing parameters during the optimization process will not lead to an unsolvable system. So in addition to the solvable initialization, maintaining solvability during training must be considered to obtain a robust training process.

Without further arrangements, for a randomly initialized neural ODE — meaning the ANNs parameters are random values from an initialization routine like, e.g., Glorot [9] — there are no guarantees that the initialized neural ODE is (efficiently) solvable by a given ODE solver. The positions of the eigenvalues of the resulting system are simply not considered during initial parameter selection.

If the neural ODE is not solvable, there are two possible reasons for this: Instability or stiffness.

1.3. Stability

There are countless definitions of the system property *stability*, therefore, this term is introduced in a few lines. In this paper, a system is considered *stable*, if all eigenvalues of the system matrix $\boldsymbol{A}$ have a negative real value. If the system matrix is not constant over time (nonlinear system), a system might be stable for specific locations in state and time, while it may be unstable in others. Furthermore, the process of pushing unstable eigenvalues from the right to the left half of the complex plane is called *stabilization*.

Stabilization of neural ODEs is an active and important topic of research, and several strategies have already been published in this area: Stabilizing linear neural ODEs by constraining weights in the cost function is discussed in [10]. In [11], the stabilization of nonlinear systems is also presented by constraining weights, so that a Lyapunov-stable solution is obtained. [12] provides a novel loss formulation and framework that incorporates the Lyapunov condition into the training process of neural ODEs. Further publications like [13] focus on the stabilization of discrete systems.

However, the method presented in this contribution considers a more generic approach focusing on linear and nonlinear, stable, and unstable systems, as well as additional ODEs properties in addition to stability: Frequencies, damping, stiffness, and oscillation capability. In addition, the eigenvalues of the system are not affected indirectly by constraining weights during optimization, like in [10] and [11], but directly by locating the eigenvalues in a differentiable way and forcing them to specific locations or regions. In this way, also (partly) unstable systems can be trained, like the Van der Pol oscillator (VPO) in the later example.

1.4. Stiffness

Taking into account the actual *solvability*, another important attribute must be investigated, which is *stiffness*. Even if a system is solvable, the solution process can be very inefficient, i.e., the solver takes very small steps to guarantee a stable solution process.

Like stability, the topic of stiffness is discussed in the field of neural ODEs. For example, in [14] it is investigated that scaling (normalization) of the right-hand side is crucial to learn for stiff systems and a specialized and efficient method for sensitivity analysis (quadrature adjoint) is presented. In addition, [15] investigates Simple Temporal Regularization (STEER), which means that the termination time for the solving process is not assumed fixed, but varies between training steps. In this way, smaller loss values could be achieved for stiff systems.

We would like to emphasize that this work is not focusing especially on learning stiff systems but also on preventing unintended stiffness. In addition, not only stiffness, but also multiple properties of the system can be enforced simultaneously with the presented method. To the knowledge of the authors, there is no other method that allows for regularization of a variety of eigenvalue-based system properties to directly compare our approach with.

1.5. Convergence

Even if the neural ODE is (and stays) solvable, a common issue is the convergence to an unsatisfactory local minimum. This can be observed

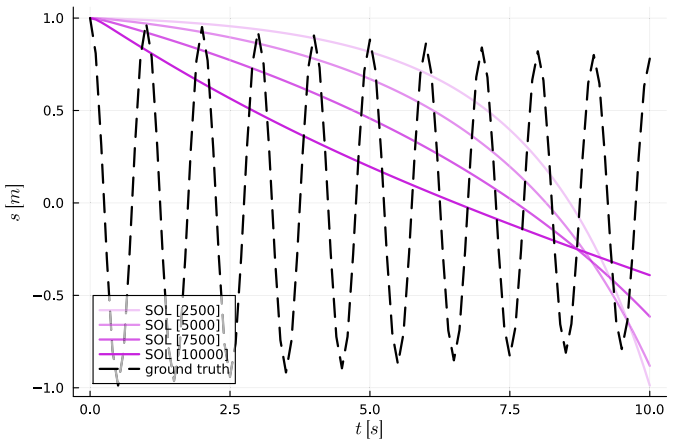


Fig. 1. An example neural ODE of an oscillating system after 2500, 5000, 7500 and 10000 training steps. This neural ODE is suffering from a typical local minimum. The local minimum is relative close to the optimum (black-dashed), leaving it is challenging because the loss function gradient vanishes.

especially for oscillating systems, as exemplified in Fig. 1. Instead of understanding a relatively complicated oscillation, a much simpler average over the oscillation is learned. Without further arrangements, it is often impossible to train a neural ODE to adequately describe an oscillating system over multiple periods or a highly dynamic system solely based on a simple cost function defined on the ODE's solution.

However, there are different modifications to the training process to tackle this challenge:

- A common approach is the so-called **growing horizon**, which starts with a small portion of the simulation horizon, i.e., the time span, and successively increases this horizon with consecutive training convergence. Disadvantageously, this introduces multiple new hyperparameters.[1] The determination of these hyperparameters is not trivial. Especially for high-frequency applications, the hyperparameters are very sensitive and problem-specific, and the chances of reusability in another application are low.
- Further, **mini-batching** strategies can be applied, which cut the training trajectory into multiple pieces. Instead of computing the gradient for the entire trajectory at once, leading to averaging the information within the gradient, only one piece of the trajectory (the batch) is used. The length or size of batches is a sensitive hyperparameter. In stochastic mini-batching, the piece to train on is picked randomly, however, more advanced schedulers can be applied. If gradients for multiple pieces are computed in parallel, the method corresponds to **multiple shooting** [16]. Batching, as well as multiple shooting, requires reinitialization of the neural ODEs at various points in time, which requires knowledge of the entire system state $\boldsymbol{x}$ based on data. Whereas this seems trivial for academic examples, this is not feasible for e.g., large-scale systems with hundreds or thousands of states.

To conclude, even if both strategies can be used to enhance convergence for neural ODEs, they heavily rely on appropriate hyperparameters and, in case of mini-batching, knowledge of the system state. In general, both approaches can be seen as *artificial measures*, incorporated by the user. Both strategies might be chosen based on the observation that the system contains high-frequency oscillations, therefore a training starting with a small timespan (growing horizon) or on small timespans in general (mini-batching) is considered promising. However, in this paper, we investigate a more *natural* and direct way of incorporating knowledge of oscillation, frequency, and damping.

To summarize the introduction, solvability, stability, stiffness, and convergence behavior of neural (ordinary) differential equations are important and actively discussed topics in the scientific community, and many proposals have been made to cope with one of these challenges. However, we want to clarify that this work goes beyond individual topics like stability and stiffness alone, and opens up to the incorporation of *system properties*, which depend on eigenvalues, in general (frequencies, damping, etc.). Further, our method does not only focus, for example, on training stable or stiff systems, but also allows for targeting a specific stiffness or stability value. The proposed method could be applied to address multiple or even all of the challenges investigated at once. Besides that, eigen-informed neural ODEs are not only useful to accomplish, e.g., stability, but also allow for improving model quality regarding a variety of other aspects, e.g., fitting under very limited amount of data or even dealing with undersampling.

2. Method

The core idea of the presented method is to force the system eigenvalues into specific positions, ranges, or geometric areas during the neural ODE training. This is achieved by a specially loss function design based on the system eigenvalues. Inside the loss function, the method can be subdivided into three steps:

1. Gather the system matrix (Jacobian) for multiple points in time and compute the corresponding eigenvalues (see Section 2.1),
2. provide the eigenvalue sensitivities needed for the gradient over the loss function (see Section 2.2) and
3. rate the eigenvalue positions (compared to target positions, ranges, or areas) as part of a loss function (see Section 2.3).

Because the evaluation of physical equations in the cost function of ANNs is known as Physics-informed Neural Network (PINN) [17], we want to pursue this naming convention by presenting *eigen-informed neural (ordinary) differential equations*, which evaluate eigenvalues (or eigenvectors) as part of their loss functions. Here, two different types of loss functions can be distinguished:

1. Training based on the solution: Eigenvalues can be computed for each solver step during solving of the ODE.
2. Training based on the derivative: Eigenvalues can be computed for pre-defined points in time and state space by evaluation of the right-hand side, without actually solving the ODE. This is often referred to as *collocation* [18].

In the following, we focus on the more complex case of training on the solution of the eigen-informed neural ODE.

2.1. Eigenvalues of the system

Eigenvalues are computed for the system matrix $\boldsymbol{A} = \frac{\partial \dot{\boldsymbol{x}}}{\partial \boldsymbol{x}}$, i.e., the sensitivities of the state derivatives $\dot{\boldsymbol{x}}$ w.r.t. the states $\boldsymbol{x}$. For linear systems, this matrix is constant over the entire solving process, for nonlinear systems it is not. As soon as a neural ODE uses one or more nonlinear activation functions, it becomes a nonlinear system. Whereas for simple nonlinear systems a symbolic Jacobian can be derived, for more complex systems, the system matrix must be determined numerically for every time step investigated. Depending on the size, the determination of this Jacobian is not computationally trivial, but often the system matrix is already calculated by another algorithm and can be reused. For example, most implicit solvers calculate and store the system Jacobian for solving the next integrator step [19]. In this case, the Jacobian can be reused without a computational effort worth mentioning. The other

[1] What is the horizon growing condition? What is the condition threshold? How much grows the horizon?

way around, a computed Jacobian for the application of this method can be shared with an implicit solver. If not available through another algorithm, the system matrix can be computed using Automatic Differentiation (AD), sampled using finite differences, or could even be symbolically known.

After obtaining the Jacobian, the eigenvalues can be computed. There are multiple algorithms to approximate the eigenvalues of a given matrix $\boldsymbol{A}$. One of the most famous is the approximation of eigenvalues and eigenvectors through iterative QR-decomposition [20]. This algorithm may require many iterations to converge (for a tight convergence criterion), but the iteration count can be reduced by using different shift techniques or the algorithm extension *deflation*. Using the QR-algorithm, the eigenvalues and eigenvectors can be computed.

2.2. Sensitivities of eigenvalues

Computing sensitivities over iterative eigenvalue computations, like the QR-algorithm, using AD is computationally expensive. This is because additionally to the eigenvalue computation itself, all algorithm operations must be executed at derivation level for every algorithm iteration. Furthermore, the numerical precision at derivation level may decrease with larger iteration counts, and an exact sensitivity computation is not guaranteed.

A better approach for sensitivity computation is to provide the sensitivities analytically for the entire iterative process. The advantages are improved performance (the analytical expression needs only to be evaluated one time for an arbitrary number of iterations within the algorithm) and improved numerical accuracy (no risk of loss of numerical precision through iteration).

Let $\boldsymbol{D} = diag(\lambda)$ be the diagonal matrix of eigenvalues λ and $\boldsymbol{U}$ a matrix that contains the corresponding eigenvectors in columns. In the following, we investigate the function $\boldsymbol{D}, \boldsymbol{U} = eigen(\boldsymbol{A})$, which computes eigenvalues and eigenvectors based on a given (system) matrix $\boldsymbol{A}$.

Automatic differentiation (AD). AD is a technique to compute partial derivatives on the fly, so along the actual computations, by performing differentiation based on symbolic expressions, but storing intermediate results numerically. AD comes in two different flavors, depending on whether the sensitivities are determined in (forward) or against (reverse) the direction of the actual computation. Forward AD results in a series of Jacobian-vector-products to propagate the sensitivity with respect to the input, known as *seed*, through the computational graph in order to determine the required sensitivity. Reverse AD on the other hand results in a series of vector-Jacobian-products to propagate the sensitivity with respect to the output (the *adjoint* or *co-tangent*) backwards through the computational graph [21]. Common AD libraries provide an interface for *rules* to define how sensitivities are propagated through a method call forward or in reverse. Such interfaces can be used to define custom rules for eigenvalue and eigenvector computation, e.g., via the QR-algorithm.

The sensitivities (seeds) for forward mode differentiation $\frac{\partial \boldsymbol{D}}{\partial \upsilon}$ and $\frac{\partial \boldsymbol{U}}{\partial \upsilon}$ with respect to an input variable υ are provided in Eq. (1) and (2) based on [22].

$$\frac{\partial \boldsymbol{D}}{\partial \upsilon} = \boldsymbol{I} \circ \left(\boldsymbol{U}^{-1} \cdot \frac{\partial \boldsymbol{A}}{\partial \upsilon} \cdot \boldsymbol{U} \right) \tag{1}$$

$$\frac{\partial \boldsymbol{U}}{\partial \upsilon} = \boldsymbol{U} \cdot \left(\boldsymbol{F} \circ \left(\boldsymbol{U}^{-1} \cdot \frac{\partial \boldsymbol{A}}{\partial \upsilon} \cdot \boldsymbol{U} \right) \right) \tag{2}$$

Here, $\cdot$ is the matrix product and $\circ$ is the Hadamard product (elementwise product). Further let $\boldsymbol{F}$ be defined as:

$$F_{i,j} = \begin{cases} (\lambda_j - \lambda_i)^{-1} & \text{for} i \neq j \\ 0 & \text{elsewhere.} \end{cases} \tag{3}$$

In analogy, the reverse mode sensitivity (adjoint) $\bar{\boldsymbol{A}}$ can be defined as in [22]:

$$\bar{\boldsymbol{A}} = \frac{\partial \gamma}{\partial \boldsymbol{A}} = \boldsymbol{U}^{-T} \cdot \left(\frac{\partial \gamma}{\partial \boldsymbol{D}} + \boldsymbol{F} \circ \left(\boldsymbol{U}^{T} \cdot \frac{\partial \gamma}{\partial \boldsymbol{U}} \right) \right) \cdot \boldsymbol{U}^{T}, \tag{4}$$

where γ is the output variable for the sensitivity analysis.

Both differentiation rules, forward and reverse, are implemented in our Julia package *DifferentiableEigen.jl* (https://github.com/ThummeTo/DifferentiableEigen.jl) for the common AD frameworks. This package is also used in the experiments section.

2.3. Eigenvalue positions

Inducing additional system knowledge in different forms into ANNs has been shown to be an excellent method to improve the speed of training convergence and reduce the amount of training data needed, compared to solving a task by a pure ANNs alone. The concept of neural ODEs itself can be interpreted as the integration of the algorithm *numerical integration* into a machine learning model. Based on that, injecting one or more symbolic ODEs to obtain *physics-enhanced* or *hybrid* neural ODEs further reduces the amount of training data and improves the training convergence speed, because only the remaining, not modeled effects (or equations) need to be understood by the ANNs [23–25].

Continuing this strategy, *system properties* can also be used as additional knowledge and can improve different aspects of neural ODE training. This paper especially focuses on eigenmodes, meaning eigenvalues and eigenvectors, which can describe the following system attributes:

- Stability (all eigenvalues on the left half of the complex plane, see Section 2.3.1)
- Oscillation capability (existence of conjugate complex eigenvalue pairs, see Section 2.3.2)
- Frequency (imaginary positions of conjugate complex eigenvalue pairs, see Section 2.3.3)
- Damping (positions of eigenvalues, see Section 2.3.4)
- Stiffness (ratio between largest and smallest, negative real positions of eigenvalues, see Section 2.3.5)
- …

The way to include knowledge of these system properties in a cost function of a neural ODE, to obtain an *eigen-informed* neural ODE, is shown in the following subsections.

In the following, let $\lambda(t)$ denote the eigenvalues of the corresponding system for the time instant t and $\lambda_i(t)$ the i-th eigenvalue. The order of the eigenvalues is not important as long as it does not change[2] over t. The real part of a complex eigenvalue λ_i is denoted by $\Re(\lambda_i)$, the imaginary part by $\Im(\lambda_i)$.

For $a, b \in \mathbb{R}$, let $\epsilon(a, b)$ be an arbitrary function that evaluates the deviation between a and b, common choices are the well known mean squared error (MSE) and mean absolute error (MAE). Please note that ϵ can not only be designed to enforce a *specific* frequency, damping or stiffness, but also to allow for a given *range* or *geometry* of target values. This is especially useful, if an upper and lower bound for the corresponding system attribute is known, such as, e.g., frequencies in mechanical or electrical systems. Furthermore, let $\max(a, b)$ and $\min(a, b)$ be the maximum/minimum functions of two elements a and b and $\max(\boldsymbol{c})$ and $\min(\boldsymbol{c})$ the maximum/minimum elements of a vector $\boldsymbol{c} \in \mathbb{R}^n$.

2.3.1. Stability (STB)

Probably the most intuitively known system property, but often neglected during training of neural ODEs, is *stability*. Many physical systems are stable, and even unstable mechanical or electrical systems relevant in practice often only contain an unstable subsystem. As already stated, neural ODEs are *not* stable by design. Stability can be achieved (and preserved) by forcing all real parts of the system eigenvalues to be negative. For unstable systems, only the stable subset[3] of eigenvalues

[2] If the order of eigenvalues changes, e.g., because of function output with lexicographic sorting, eigenvalues must be tracked between time steps, so they can be uniquely identified.

[3] This subset may vary over time, stable eigenvalues may become unstable and the other way round.

can be forced to the negative real half-plane. Further, nonlinear systems can be unstable for specific parts in state space while being stable in others. A rotational pendulum, for example, is stable close to the stable equilibrium position but shows unstable eigenvalues as soon as it is closer to the unstable equilibrium position. In such cases, stability should be enforced only for stable regions of the state space, of course. A simple stability loss function l_{STB} that forces a completely stable system is described in the following:

$$l_{STB}(\lambda) = \sum_{i=1}^{|\lambda|} \epsilon_{STB}(\max(\Re(\lambda_i(t)), 0), 0). \tag{5}$$

The max function with second argument 0 ensures, that only eigenvalues with positive real value (unstable) are taken into account. The error function ϵ_{STB} rates the deviation of the real part of the eigenvalue compared to 0, where marginally stable eigenvalues are located. Of course, other thresholds or ranges can be deployed if needed, for example, to promote a stability margin instead of marginal stability or even allow instability to a certain extent.

2.3.2. Oscillation capability (OSC)

Oscillation of a system can be determined by identifying conjugate complex eigenvalue pairs, meaning eigenvalue pairs with identical real parts, but negated imaginary parts. Eigenvalues cannot appear with a non-zero imaginary part without a conjugate complex partner. As a consequence, to force two eigenvalues into an eigenvalue pair, it is necessary to start by synchronizing the real parts of the eigenvalues. For a given set $\mathbf{\Omega}$ of eigenvalue pairs $(\lambda_a, \lambda_b) \in \mathbf{\Omega}$ with $\lambda_a, \lambda_b \in \lambda$, a cost function that forces oscillation can be defined as:

$$l_{OSC}(\mathbf{\Omega}) = \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{OSC}(\Re(\lambda_a), \Re(\lambda_b)). \tag{6}$$

A point worth mentioning is, that it is not necessary to know exactly which eigenvalues should be paired. The knowledge *that* eigenvalue pairs exist and the number of pairs is sufficient, because any two eigenvalues have the potential to establish a conjugate complex eigenvalue pair.[4] However, as already stated, also in this case, the eigenvalues need to be tracked so that the pairs can be maintained during training.

After laying the foundation for oscillation capability, it might be interesting to enforce a specific oscillation frequency (s. Section 2.3.3) or damping (s. Section 2.3.4).

2.3.3. Frequency (FRQ)

Oscillation frequencies in dynamical systems are measured (and therefore known) in $Hz = \frac{1}{s}$. This information must be converted to eigenvalue positions in order to specify a loss function. As already stated, an oscillation is described by a pair of eigenvalues that share the same real part and a negated imaginary part. The frequency $f(\lambda)$ of an eigenvalue λ can be calculated using the well-known equation:

$$f(\lambda) = \frac{|\Im(\lambda)|}{2 \cdot \pi}. \tag{7}$$

Based on a ground truth frequency $\tilde{f}_{ab}$ for the eigenvalue pair (λ_a, λ_b), a frequency based loss function may look as follows:

$$l_{FRQ}(\mathbf{\Omega}) = \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_a)) + \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_b)). \tag{8}$$

If λ_a and λ_b are already paired, it applies that $|\Im(\lambda_a)| = |\Im(\lambda_b)|$, which results in $f(\lambda_a) = f(\lambda_b)$. Applying this, Eq. (8) can also be written only depending on one of λ_a or λ_b as:

$$l_{FRQ}(\mathbf{\Omega}) = 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_a)) = 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_b)). \tag{9}$$

Note that combined with eigenvalue tracking, this step reduces the computational effort, because the loss function only depends on one instead of two eigenvalues per pair. As a result, it would suffice to compute the sensitivity of only one eigenvalue per pair. However, this requires modification of the QR-algorithm sensitivity rules, and a detailed examination is part of future work.

2.3.4. Damping (DMP)

Similar to the frequency, also the damping δ can be defined for an eigenvalue λ:

$$\delta(\lambda) = \frac{-\Re(\lambda)}{|\lambda|} = \frac{-\Re(\lambda)}{\sqrt{\Re(\lambda)^2 + \Im(\lambda)^2}}. \tag{10}$$

In analogy to Eq. (8), the damping loss function l_{DMP} is defined straight forward for the ground truth damping $\tilde{\delta}_{ab}$ of an eigenvalue pair:

$$l_{DMP}(\mathbf{\Omega}) = \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_a)) + \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_b)). \tag{11}$$

Because $\Re(\lambda_a) = \Re(\lambda_b)$ and $|\lambda_a| = |\lambda_b|$ for eigenvalue pairs, it applies that $\delta(\lambda_a) = \delta(\lambda_b)$ and Eq. (11) simplifies to:

$$l_{DMP}(\mathbf{\Omega}) = 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_a)) = 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_b)). \tag{12}$$

This simplification to only depend on a single eigenvalue per eigenvalue pair results in the same benefits as for the frequency equations.

2.3.5. Stiffness (STF)

Finally, also the stiffness $\tilde{\sigma}$ of a system may be known or at least an upper or lower boundary. The stiffness loss function for stable systems can be defined as:

$$l_{STF}(\lambda) = \epsilon_{STF}\left(\frac{max(\Re(\lambda))}{min(\Re(\lambda))}, \tilde{\sigma}(t)\right). \tag{13}$$

This stiffness definition basically only applies to stable systems, but for gradient determination it is important to provide a loss function that is defined for unstable systems, as well. Please note that the error function ϵ_{STF} can be formulated not only to aim at a specific stiffness $\tilde{\sigma}$. Another approach is to allow for a predefined stiffness range. In addition to training the neural ODE to a known stiffness or range, this feature can also be used to enhance the simulation performance of the resulting neural ODE. Common ODE solvers use the ODE stiffness as criterion for the step size control; the stiffer the ODE the more integration steps need to be performed. If the training goal is to solve the resulting neural ODE in as few steps as possible, the secondary objective of the training could be $\tilde{\sigma}(t) = 1$ to promote a fast simulating model.

2.3.6. Further properties

Further properties from systems and control theory can be used within the proposed method, for example, the rise and settling time, time constants, natural frequency, and more [26,27]. However, for reasons of space, we do not explain or validate all of them in detail and limit ourselves to the introduced properties above.

2.4. Training & computational cost

Common optimizers for gradient-based parameter optimization are designed to perform parameter changes based on a *single* gradient. As soon as additional gradients are obtained, it is necessary to deploy a strategy on how to further progress with multiple gradients. Basically, there are three obvious ways to include one or more property loss functions along with the original loss function in the training process:

[4] This is because a neural ODE of sufficient dimension (states and parameters) can approximate any dynamic system, so any system eigenvector configuration.

1. Extending the original loss function, for example, by mathematically *adding* further losses. This results in a single loss function gradient that can be passed to the optimizer. This corresponds to *explicit regularization* by adding *regularization terms*.
2. If an optimizer robust to gradient changes is used (e.g., optimizers using momentum), all loss function gradients can be passed one after another to the optimizer to perform multiple optimization steps. For optimizers with momentum, depending on the parameterization, this results in a similar optimization compared to summing up multiple losses and determining a single gradient.
3. One can switch between gradients based on a defined gradient criterion. This way, only a single gradient is selected and applied.

We apply the most common approach, which is to sum up the individual losses. However, we found that it is valuable to examine the individual optimization directions and would like to investigate potentials of that approach in the future. A great feature of generating multiple gradients for neural ODEs is that multiple optimization directions are generated at a very low computational cost. This is due to the fact that essential parts of the gradient computation can be reused, see the loss function l gradient defined in Eq. (14) with ODEs solution $\boldsymbol{X}$ and ANNs parameters $\boldsymbol{\theta}$. Note that this loss function does not explicitly depend on parameters, like it would be the case for e.g., L1 or L2 regularization, but could be extended to do so if required.

$$\frac{\partial l(\boldsymbol{X}(\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}} = \frac{\partial l(\boldsymbol{X}(\boldsymbol{\theta}))}{\partial \boldsymbol{X}(\boldsymbol{\theta})} \cdot \frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{14}$$

The first part $\frac{\partial l(\boldsymbol{X}(\boldsymbol{\theta}))}{\partial \boldsymbol{X}(\boldsymbol{\theta})}$ of the loss function gradient is computationally cheap in general, it basically depends on the complexity of the used error function inside the loss function. The second factor, the Jacobian $\frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ on the other hand, is computationally expensive, because it requires differentiation through the entire ODE solution $\boldsymbol{X}$, which requires differentiation through the ODE solver. For every gradient that depends on the ODE solution, $\frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ can be reused after being created once.

The scalar loss function l rating the deviation of the solution can be replaced by a vector loss function $\boldsymbol{l}$, which also rates, e.g., multiple eigenvalue deviations. Based on the loss function vector output, a loss function *Jacobian* can be obtained, with only little impact on the computational performance.[5] This Jacobian consists of multiple optimization directions. The loss function Jacobian is shown in Eq. (15) and uses the same solution Jacobian $\frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$.

$$\frac{\partial \boldsymbol{l}(\boldsymbol{X}(\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}} = \frac{\partial \boldsymbol{l}(\boldsymbol{X}(\boldsymbol{\theta}))}{\partial \boldsymbol{X}(\boldsymbol{\theta})} \cdot \frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{15}$$

To conclude, for *cheap* error functions like MSE and similar, the computational cost for a single as well as for multiple gradients is driven by the cost of the Jacobian over the solution $\frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$.

To continue, the gradient over a cost function $\boldsymbol{l}_\lambda$ containing the eigenvalue operation can be defined as:

$$\frac{\partial \boldsymbol{l}_\lambda(\lambda(\boldsymbol{X}(\boldsymbol{\theta})))}{\partial \boldsymbol{\theta}} = \frac{\partial \boldsymbol{l}_\lambda(\lambda(\boldsymbol{X}(\boldsymbol{\theta})))}{\partial \lambda(\boldsymbol{X}(\boldsymbol{\theta}))} \cdot \frac{\partial \lambda(\boldsymbol{X}(\boldsymbol{\theta}))}{\partial \boldsymbol{X}(\boldsymbol{\theta})} \cdot \frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}. \tag{16}$$

Similar to the loss function defined on the solution, the Jacobian $\frac{\partial \boldsymbol{l}_\lambda(\lambda)}{\partial \lambda}$ basically depends on the used error function and is computationally cheap in general, whereas the eigenvalue Jacobian $\frac{\partial \lambda(\boldsymbol{X})}{\partial \boldsymbol{X}}$ needs derivation through the eigenvalue operations and is in general expensive.[6] Computationally advantageous is that as for the solution Jacobian $\frac{\partial \boldsymbol{X}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$, the eigenvalue Jacobian $\frac{\partial \lambda(\boldsymbol{X})}{\partial \boldsymbol{X}}$ can be shared between all loss functions that consider eigenvalues and needs only to be determined once.

In conclusion, besides some computational savings, the number of time instances in which the system matrix and eigenvalues are computed should be handled deliberately because they may have a significant impact on the overall training performance, depending on the dimensionality and complexity of the system. On the other hand, calculating *additional* gradients based on already determined state- and eigenvalue-Jacobians is computationally cheap because the computationally expensive Jacobians can be cached and reused. Quite simply speaking, if losses are determined for the ODEs solution and one system property, further system properties can be investigated with negligible computational overhead. As stated above, if implicit solvers are implemented, the system Jacobian can be shared between the solver and the loss function, resulting in major computational benefits and making eigen-informed training particularly interesting (and computationally cheap) for applications where implicit solvers are used anyway.

3. Experiments

The aim of the following examples and this work is to showcase an additional tool for dealing with data poverty or poor convergence in neural ODE applications, and not a universal strategy that supersedes existing approaches under all conditions. Furthermore, we know no other method that allows for regularization of all ODEs' properties investigated, e.g., frequencies, damping coefficients, stability and stiffness, that would allow for a meaningful comparison.

In the following, three examples are given to show the applicability and benefits of the presented method. During the experimental section, we baseline our approach against a common neural ODE under the very same training conditions. To compare various *eigen-informed* approaches to the training process without regularization, we repeat the experiments with different loss function configurations, i.e., different combinations of regularized system properties. Here, the gradient SOL refers to a pure neural ODE, so one that is trained solely on the ODE solution. Adding further gradients corresponds to eigen-informed neural ODEs in different configurations.

All experiments are repeated 50 times, plots and tables show the median, as well as 25th and 75th percentiles. Because we are more interested in the general applicability than in the best solutions (possibly only by chance), we primarily investigate median values in the discussions. The *Tsit5* solver [28] (default parameterization) is used for solving the neural ODEs and the *Adam* optimizer [29] for training the neural ODEs. An exponential decay learning rate scheduler is applied with a decay rate of $2 \cdot 10^{-3}$ and an initial learning rate of 10^{-2} that falls to 10^{-3}. Note that *Tsit5* is an explicit Runge-Kutta method with limited stability (only a part of the left complex plane is stable), so it is (intentionally) not a suitable solver for stiff equations. Because the systems to be learned are not stiff, this is a reasonable choice. However, if neural ODEs become stiff during training, this solver will need to take very small steps.

The ground truth data for the system properties (eigenvalue positions) is determined based on analysis of the simulation results of the ground truth system. All examples use a common, simple loss function to rate the ODEs Solution (SOL):

$$l_{SOL}(\boldsymbol{X}) = \sum_{x \in \boldsymbol{X}} \epsilon_{SOL}(x_1, \tilde{x}_1(t)), \tag{17}$$

with $\tilde{x}_1(t)$ being the ground truth ODE solution for the first entry of the state vector x_1 (the position) and $\epsilon_{SOL}(a, b)$ the MAE. Eigen-informed neural ODEs are obtained by adding additional objectives (error functions) as introduced. For all eigenvalue errors, the MAE is also applied. To obtain a single-objective optimization problem, all objectives are scaled and summed up. Formally, the resulting optimization problem states:

$$\min_{\boldsymbol{p} \in \mathbb{R}^{|\boldsymbol{p}|}} \quad \sum_{e \in \boldsymbol{E}} \alpha_e \cdot l_e(\boldsymbol{X}(\boldsymbol{p}), \lambda(\boldsymbol{X}(\boldsymbol{p})), \tag{18}$$

[5] In this paper's examples, computation time for the loss Jacobian compared to the loss gradient increased by ≈ 1.5 % using forward mode AD.

[6] Besides the iterative nature of the QR-algorithm, for sensitivity estimation through the eigenvalue determination also the inverse of $\boldsymbol{U}$ (s. Eqs. (1), (2) and (4)) is needed.

Table 1
ANN layout for the examples.

Index	Type	Activation	Dimension	Parameters
1	Pre-Process	identity	0	0
2	Dense	tanh	$2 \to 32$	96
3	Dense	identity	$32 \to 1$	33
4	Post-Process	identity	0	0
				Sum: 129

where $\boldsymbol{E} \subseteq \{SOL, FRQ, DMP, STB, OSC, STF\}$ — depending on the experiment. The gradients of the eigen-value operations are scaled to match the order of magnitude of the solution gradient, so the corresponding values for $\boldsymbol{\alpha}$ are 10^0 (SOL), 10^1 (FRQ), 10^1 (DMP), 10^{-1} (STB), 10^1 (OSC) and 10^{-4} (STF). The corresponding software is written in the Julia programming language (v1.11) [30] using the neural ODEs framework *DiffEqFlux.jl* (v4.2) [31] and is part of our open source repository *DifferentiableEigen.jl*[7] (v0.2). The scripts to reproduce the results are open source and part of the repository's examples section.

The potential of the presented methodology is shown for three different use cases:

- A weakly damped translational oscillator (linear system) to show the influence of different gradient setups (see Section 3.1),
- the same oscillator trained on sparse and undersampled data, which does not fulfill the Nyquist-Shannon sampling theorem (see Section 3.2) and
- the nonlinear VPO, to show the applicability for nonlinear systems (see Section 3.3).

All ANNs for the neural ODEs are set up as described in Table 1. Simple pre- and post-processing layers are applied, to shift and scale data (min/max normalization) in order to prevent saturation of the ANN [3]. Please note that the output dimension is 1 for systems consisting of 2 states, because only the highest derivative (acceleration) is determined for the considered second order ODEs.

During the following experiments, we investigate the following qualitative (plot) and quantitative (table) properties:

Generalization investigates the performance on unknown data, giving a measure to rate whether the model learned the intended behavior or just over-fitted. As a quantitative measure, we compare the loss values of the eigen-informed and pure neural ODEs on unknown testing data by continuing to simulate the trajectory.

Convergence speed indicates the rate at which the loss function decreases and is investigated by counting the number of optimization steps, until the loss falls below a limit. As a quantitative measure, we compare the number of optimization steps that the eigen-informed neural ODE requires to outperform the final loss value of the pure neural ODE (in the median).

Solving time and number of solver steps in order to reach the simulation termination time. This corresponds to the stiffness of the system and defines the cost of solving the ODE. Therefore, fewer steps are desirable. As a quantitative measure, we compare the median solving time of eigen-informed and pure neural ODEs.

Computational cost of the training in terms of the median time required to determine the loss function gradient. Of course, eigen-informed regularization requires more time compared to the pure neural ODE, caused by the additional arithmetic operations required. However, quantification makes sense in order to weigh up the benefits in applications with limited computing resources.

[7] https://github.com/ThummeTo/DifferentiableEigen.jl

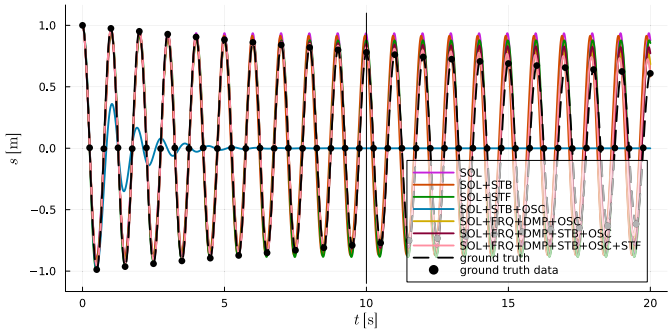


Fig. 2. Comparison of the neural ODE solutions (oscillator position), trained with different gradients. The trajectory for the experiment closest to the median final loss is plotted. The first 10 s are training data, the last 10 s are for testing.

3.1. Linear system: weakly-damped oscillator

As already stated in the introduction, especially oscillating systems are a challenging training task for neural ODEs. By also applying a weak damping to the system, a longer simulation horizon must be considered because training on a single oscillation period will hardly capture the damping effect. The state space equation of the translational spring-damper-oscillator with position s and velocity v is given in Eq. (19):

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} \dot{s} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ \frac{-c \cdot s - d \cdot v}{m} \end{bmatrix}, \tag{19}$$

with $c = 4\pi^2 \frac{\mathrm{N}}{\mathrm{m}}$, $d = 0.05 \frac{\mathrm{N \cdot s}}{\mathrm{m}}$ and $m = 1\,\mathrm{kg}$. Please note that this oscillator is a linear system.[8] Training data is sampled equidistantly with a frequency of 10 Hz, and the simulation and training horizon are both 10 s. Intentionally, no mini-batching is used, training is performed on the entire ODE solution.

Convergence. As can be seen in Fig. 2, most neural ODEs are capable of representing the oscillation frequency of the linear system after a training for $5,000$ steps; only the configuration $SOL + STB + OSC$ cannot converge satisfactorily within the limited training setup. However, on testing data (right half of the plot), we find that damping is not sufficiently present and the amplitude is too high in some places, motivating a more detailed investigation of the quantitative results.

Before investigating the quantitative deviations, in Fig. 3 the convergence behavior (loss over training) is presented. Basically, all gradient configurations run into (insufficient) local minima, except the configurations containing FRQ and DMP which can leave the original local minima early.

Whereas the local optima the neural ODEs converge to look similar, we find that the trajectories that include $FRQ + DMP + OSC$ converge to a better optimum (median $\approx 0.0145 - 0.0179$), compared to the remaining loss functions (median > 0.0263). This can be investigated in Table 2. We find that these configurations lead to significant improvements w.r.t. the found optimum. Further investigating the column *Hit*, giving the first training step for that the final loss value of the pure neural ODE is reached by the eigen-informed neural ODE, we find values around $1,000$ for the best performing gradients — compared to the $5,000$ training steps of the neural ODE — resulting in approximately five times faster convergence of the considered eigen-informed approaches.

On testing data (Table 3), a very similar pattern can be observed: The last three loss functions feature significantly better loss values (median

[8] Training a linear system with a nonlinear neural ODE is not the most reasonable approach. This was done to always maintain the same neural ODE topology for all examples.

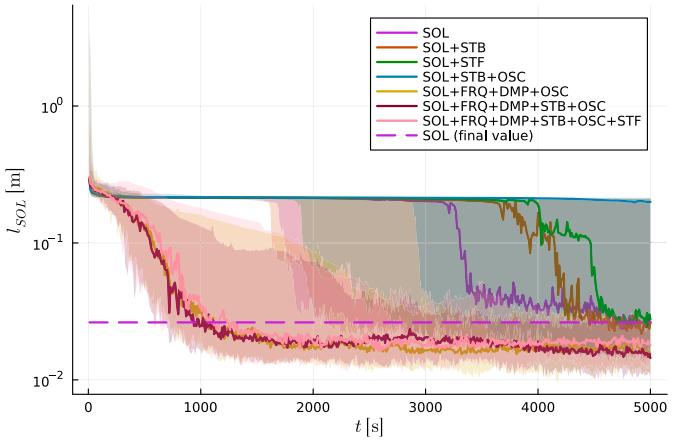


Fig. 3. Comparison of the neural ODE convergence behavior, trained with different gradients. Figure shows the median loss defined on the ODE solution l_{SOL}. Semi-transparent ribbons show the 25th and 75th percentiles. Pink-dashed shows the final loss of the pure neural ODE (baseline), the intersection with other curves allows to check at which training step the baseline is outperformed.

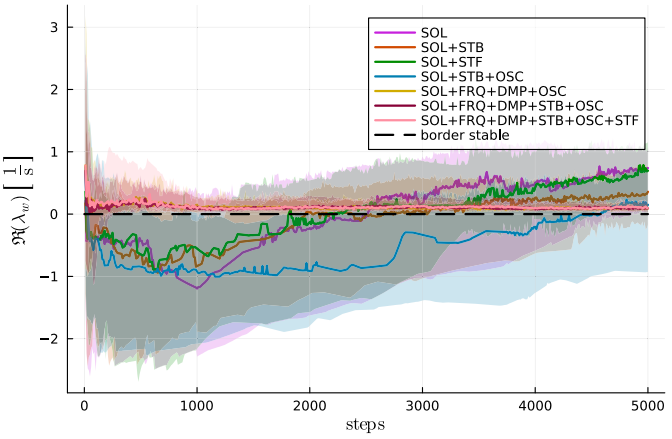


Fig. 4. Comparison of the median neural ODE stability (rated by the maximum real value of the most unstable eigenvalue λ_w), trained with different gradients. Stable systems are located beneath the border stable line (black-dashed). Semi-transparent ribbons show the 25th and 75th percentiles.

Table 2
Median, 25th and 75th percentiles for the final loss value of different loss functions for training data. *Hit* identifies the first step, in which the pure neural ODE final loss value is outperformed.

Loss	P_{25}	Median	P_{75}	Hit
SOL	0.0154	0.0263	0.2085	n.a.
SOL + STB	0.0148	0.0266	0.2116	4460
SOL + STF	0.0159	0.0277	0.2121	4760
SOL + STB + OSC	0.0145	0.1993	0.2137	n.a.
SOL + FRQ + DMP + OSC	0.0109	0.0155	0.0259	1050
SOL + FRQ + DMP + STB + OSC	0.0106	0.0145	0.0233	940
SOL + FRQ + DMP + STB + OSC + STF	0.0138	0.0179	0.0245	1070

Table 3
Median, 25th and 75th percentiles for the final loss values of different loss functions for testing data.

Loss	P_{25}	Median	P_{75}
SOL	0.0676	0.1068	0.165
SOL + STB	0.0571	0.106	0.1664
SOL + STF	0.0488	0.0973	0.1668
SOL + STB + OSC	0.0758	0.1597	0.1671
SOL + FRQ + DMP + OSC	0.0377	0.0511	0.0739
SOL + FRQ + DMP + STB + OSC	0.0333	0.0524	0.0717
SOL + FRQ + DMP + STB + OSC + STF	0.0367	0.0574	0.0838

$\approx 0.0511 - 0.0574$) compared to the remaining ones (> 0.0973). We observe that for this example, the more valuable information is included in the gradients FRQ and DMP. The gradient configurations including only STB, STF or OSC, are not able to convince in this case, because they show only limited (or no) quantitative benefit.

To summarize the first observations, we find that incorporation of frequency and damping information significantly improves the training of the spring-damper-pendulum in terms of convergence speed, as well as the found minimum. Regularization of stability and stiffness only seems to be of limited value in the considered, very easy use case. However, this observation should not be generalized too far beyond the application considered.

Stability. In addition, we compare the stability of the solution (see Fig. 4), which shows the maximum real value over time of the most unstable eigenvalue λ_w. As expected, all gradient configurations containing the STB gradient stabilize the system quickly during the first training steps and prevent the eigenvalues from leaving the stable half of the complex plane too far. Slightly unstable eigenvalues are only weakly punished, caused by a small regularization multiplier (10^{-4}), to prevent an oscillating optimization. However, a larger regularization multiplier can be applied to enforce real stability but was intentionally omitted here.

Table 4
Median times for gradient computation, number of accepted and rejected solver steps for different loss functions on the training part of the trajectory.

Loss	Sim. Time [ms]	Grad. Time [s]	Acc. Steps	Rej. Steps
SOL	2.47	0.93	89	6
SOL + STB	2.15	4.07	90	0
SOL + STF	2.19	4.08	84	1
SOL + STB + OSC	2.41	4.02	95	0
SOL + FRQ + DMP + OSC	2.2	4.11	84	0
SOL + FRQ + DMP + STB + OSC	2.24	4.13	86	1
SOL + FRQ + DMP + STB + OSC + STF	2.22	4.11	88	0

Computational performance. A brief investigation of solver statistics (s. Table 4) shows that the regularized training does not negatively affect the number of required solver steps that stay similar across all investigated gradients. However, the number of rejected steps decreased (0-1 vs. 6), showing that eigen-informed regularization can lead to solutions with better numerical properties. As expected, the computational cost (in terms of time) for the determination of the loss function gradient including eigenvalues is higher, approximately four times compared to the cost for a pure neural ODE. However, the simulation times (inference) are similar across all approaches.

3.2. Linear system: insufficient data sampling frequency

Another interesting use case is the training of a neural ODE with data that does not satisfy the Nyquist-Shannon sampling theorem. To reproduce a signal that contains a maximum frequency of b ($\frac{1}{s}$), data samples with a sampling distance $< \frac{1}{2\,b}$ (s) are needed. As a consequence, it is not possible to train a neural ODE (or any other model) to reproduce high-frequency effects that are not captured by a sufficiently high data sampling rate. In practical applications, the data recording frequency is often limited by different factors, but the eigenmodes of the system are often known. This information can be used in eigen-informed neural ODEs.

For this example, the same model as in Section 3.1 is used, but with a damping of $d = 0.5 \frac{\text{N·s}}{\text{m}}$. The maximum (and only) frequency of the oscillating system shall be $f_{max} = 1\,\text{Hz}$. The maximum allowed distance between data samples Δt_{max} can now be calculated using the Nyquist–Shannon sampling theorem:

$$\Delta t_{max} = \frac{1}{2 \cdot f_{max}} = \frac{1}{2 \cdot 1\,\frac{1}{\text{s}}} = 0.5\,\text{s}. \tag{20}$$

Therefore, data sampling with $\Delta t > \Delta t_{max}$ will lead to an unrecoverable signal. For this example, we intentionally violate the theorem by selecting $\Delta t = 0.75\ s > 0.5\ s = \Delta t_{max}$.

Convergence. As can be seen in Fig. 5, the resulting data sampling points are very sparse. As expected, the neural ODE based on the SOL gradient is unable to replicate the signal and converges to a local minimum. In contrast, gradient configurations containing frequency and damping information lead to a very good fit after $5,000$ training steps and are capable of recovering the original ground truth signal.

This success can be quantitatively validated by investigation of the loss function values for training (Table 5) and testing (Table 6).

As expected, the convergence behavior (s. Fig. 6) of the gradient configurations including FRQ + DMP looks excellent compared to the pure neural ODE (SOL), which is trapped in a local minimum. Even if some runs for the neural ODE (compare the 25th percentile) are able to converge to a better local minimum compared to the median, the gradients including FRQ + DMP are again able to outperform even the best runs

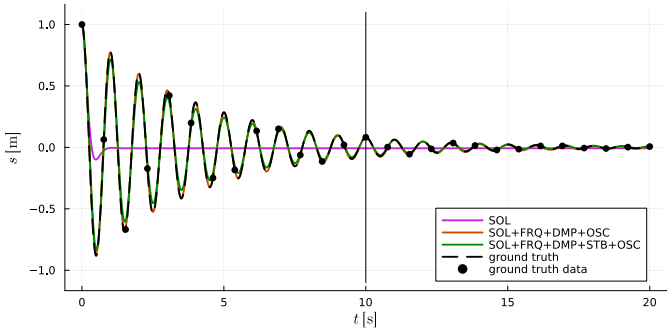


Fig. 5. Comparison of the neural ODE solutions (oscillator position), trained with different gradients. The trajectory for the experiment closest to the median final loss is plotted. The first 10 s are training data, the last 10 s are for testing. The gradient configurations including FRQ + DMP (orange, green) lay close to the ground truth (black-dashed). For training, undersampled data points (black dots) are used.

Table 5
Median, 25th and 75th percentiles for the final loss value of different loss functions for training data. *Hit* identifies the first step, in which the pure neural ODE final loss value is outperformed.

Loss	P_{25}	Median	P_{75}	Hit
SOL	0.0086	0.073	0.0737	n.a.
SOL + FRQ + DMP + OSC	0.0046	0.0182	0.1018	920
SOL + FRQ + DMP + STB + OSC	0.0041	0.0124	0.0475	780

Table 6
Median, 25th and 75th percentiles for the final loss values of different loss functions for testing data.

Loss	P_{25}	Median	P_{75}
SOL	0.0036	0.009	0.0093
SOL + FRQ + DMP + OSC	0.0006	0.0016	0.0665
SOL + FRQ + DMP + STB + OSC	0.0004	0.0013	0.0119

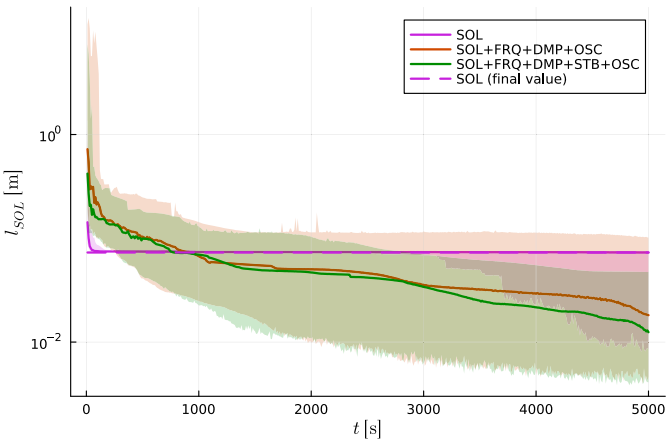


Fig. 6. Comparison of the neural ODE convergence behavior, trained with different gradients. Figure shows the median loss defined on the ODE solution l_{SOL}. Semi-transparent ribbons show the 25th and 75th percentiles. Pink-dashed shows the final loss of the pure neural ODE (baseline), the intersection with other curves allows to check at which training step the baseline is outperformed.

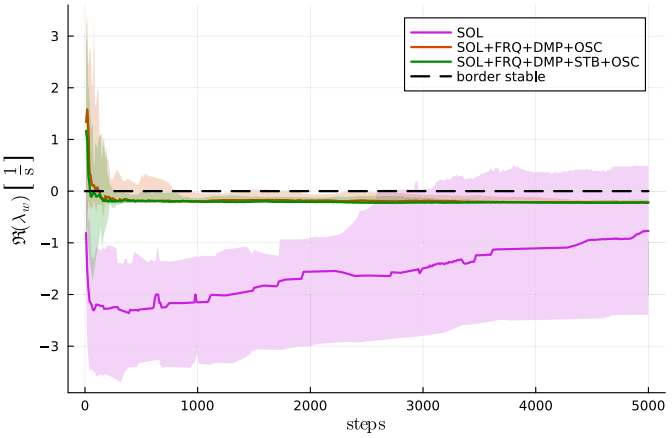


Fig. 7. Comparison of the median neural ODE stability (rated by the maximum real value of the most unstable eigenvalue λ_w), trained with different gradients. Stable systems are located beneath the border stable line (black-dashed). Semi-transparent ribbons show the 25th and 75th percentiles.

of SOL. This is to be expected; otherwise, the Nyquist-Shannon theorem would be violated.

Stability. Regarding stability (s. Fig. 7), especially the configuration that includes the STB gradient is stabilized very fast (compare the *very* beginning of the plot) and is maintained stable throughout the training process. In addition, the configuration SOL + FRQ + DMP + OSC (orange) is also able to achieve stability without regularization. This is to be expected, because damping (DMP) is highly related to stability (STB) and therefore stability is implicitly enforced by incorporation of damping, in this case. The pure neural ODE (SOL) is and stays stable, however, it is also *too* stable, featuring a too strong decay of the oscillation, as can also be observed in the previous solution plot (Fig. 5).

Computational performance. Again investigating the properties of the numerical solution, we find that the pure neural ODE is solved faster and with fewer steps. However, it should be noted that this is what we expect when investigating the shape of the solution (see Fig. 5), because the pure neural ODE is unable to produce meaningful results. As a result, a comparison is not meaningful and does not create any added value

Table 7
Median times for gradient computation, number of accepted and rejected solver steps for different loss functions on the training part of the trajectory. Brackets indicate invalid runs, that did not converge to an acceptable solution.

Loss	Sim. Time [ms]	Grad. Time [s]	Acc. Steps	Rej. Steps
SOL	(1.41)	(0.55)	(52)	(1)
SOL + FRQ + DMP + OSC	2.01	1.67	85	0
SOL + FRQ + DMP + STB + OSC	2.19	1.67	85	0

here. The computational cost for gradient determination is slightly more than three times as much for eigen-informed training, compare Table 7.

3.3. Nonlinear system: Van der Pol oscillator (VPO)

Finally, a marginally stable and nonlinear system is also observed: The VPO. The nonlinear system is given by the well-known state space equation in Eq. (21):

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{v} \\ \dot{\nu} \end{bmatrix} = \begin{bmatrix} \dot{v} \\ \mu \cdot (1 - v^2) \cdot \dot{v} - v \end{bmatrix}, \tag{21}$$

with $\mu = -0.025$. Training data is sampled equidistantly with $\Delta t = 1.57\,\mathrm{s}$ to obtain 41 samples over a simulation and training horizon of 62.8 s each. No batching is used, training is applied to the entire ODE solution for 5,000 training steps.

Convergence. Training on the solution gradient performs quite well on training data, however for testing, the damping suspends (see Fig. 8).

Adding the STB gradient slightly improves the fit. However, further adding STB + OSC leads to convergence in a bad local minimum. The gradient configurations that include FRQ + DMP with and without STB can predict the behavior of the VPO, further training and larger ANNs topologies will improve the fit. These observations can be quantitatively verified by investigation of Table 8. In analogy to the previous examples, convergence based on the loss function l_{SOL} can be observed in Fig. 9. In this example, adding the STB gradient (yellow) leads to significantly faster convergence compared to the remaining configurations. However, we also find that in this specific case, eigen-informed training (FRQ + DMP) converges faster *during* progress of training, but does not converge to a significantly better optimum *at the end* of the training.

However, this makes investigation of testing data even more interesting (compare Table 9): The loss functions containing FRQ + DMP achieve significantly better fits compared to the pure neural ODE and the remaining gradients. This nicely shows that even if a similar quantitative local minimum was found by pure neural ODE and eigen-informed approaches using FRQ + DMP, the extrapolation (generalization) has improved significantly for the eigen-informed neural ODE.

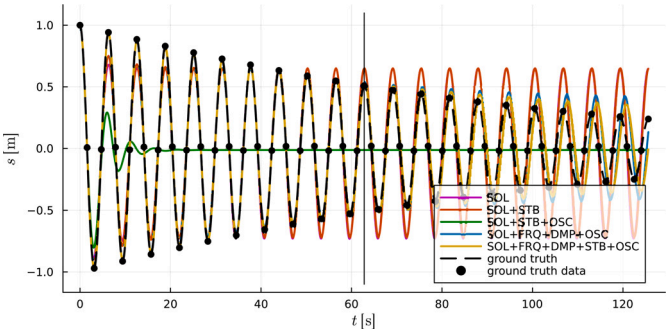


Fig. 8. Comparison of the neural ODE solutions (v), trained with different gradients. The trajectory for the experiment closest to the median final loss is plotted. The first 62.8 s are training data, the last 62.8 s are for testing. The gradient configuration SOL + FRQ + DMP + OSC (blue) and + STB (yellow) lay close to the ground truth (black-dashed). Further training will improve the fit.

Table 8
Median, 25th and 75th percentiles for the final loss value of different loss functions for training data. *Hit* identifies the first step, in which the pure neural ODE final loss value is outperformed.

Loss	P_{25}	Median	P_{75}	Hit
SOL	0.0115	0.0184	0.1692	n.a.
SOL + STB	0.0098	0.0159	0.1757	4370
SOL + STB + OSC	0.0125	0.026	0.1731	n.a.
SOL + FRQ + DMP + OSC	0.0131	0.0231	0.1991	n.a.
SOL + FRQ + DMP + STB + OSC	0.0116	0.0187	0.1992	4400

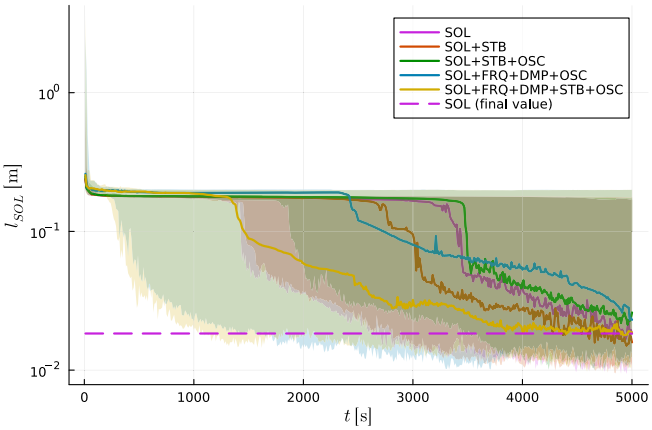


Fig. 9. Comparison of the neural ODE convergence behavior, trained with different gradients. Figure shows the median loss defined on the ODE solution l_{SOL}. Semi-transparent ribbons show the 25th and 75th percentiles. Pink-dashed shows the final loss of the pure neural ODE (baseline), the intersection with other curves allows to check at which training step the baseline is outperformed.

Table 9
Median, 25th and 75th percentiles for the final loss values of different loss functions for testing data.

Loss	P_{25}	Median	P_{75}
SOL	0.0633	0.0846	0.0904
SOL + STB	0.0429	0.0833	0.0904
SOL + STB + OSC	0.0487	0.0888	0.0907
SOL + FRQ + DMP + OSC	0.0346	0.0569	0.1091
SOL + FRQ + DMP + STB + OSC	0.0325	0.0639	0.1072

Stability. As for the previous examples, the gradients containing STB stabilize the systems at the beginning of the training and are able to keep the systems close to stable, see Fig. 10. Note that stronger regularization will lead to a fully stable system, however, it slows down convergence. In addition, in this special case, training the *pure* neural ODE already leads to a stable system. However, as can be seen from the previous examples, this is not the general case.

Computational performance. For the computation times (compare Table 10), we find again an increase by factor of three to four for eigen-informed gradients. However, the resulting neural ODE is solvable in fewer steps and does not require rejection of steps for this example with the best-performing loss functions.

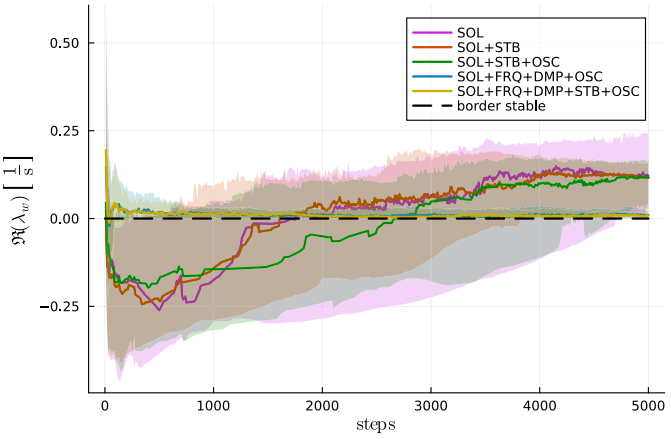


Fig. 10. Comparison of the median neural ODE stability (rated by the maximum real value of the most unstable eigenvalue λ_w), trained with different gradients. Stable systems are located beneath the border stable line (black-dashed). Semi-transparent ribbons show the 25th and 75th percentiles.

Table 10
Median times for gradient computation, number of accepted and rejected solver steps for different loss functions on the training part of the trajectory.

Loss	Sim. Time [ms]	Grad. Time [s]	Acc. Steps	Rej. Steps
SOL	2.55	1.56	93	2
SOL + STB	2.41	5.24	89	7
SOL + STB + OSC	1.75	5.25	68	0
SOL + FRQ + DMP + OSC	2.36	5.61	87	0
SOL + FRQ + DMP + STB + OSC	2.18	5.51	84	0

4. Conclusion

Finally, we summarize our work and highlight current and future research.

4.1. Summary

We highlighted a suitable strategy to induce additional system knowledge in the form of ODEs' properties into a neural ODE training process and obtained an *eigen-informed neural ODE*. From a technical view, ODEs' properties are translated to eigenvalue positions. These positions of eigenvalues can be intuitively considered in the loss function design. To maintain a gradient-based training for eigen-informed neural ODEs, the sensitivities over the eigenvalue operations are needed. For this purpose, we provide a suitable implementation in the Julia programming language called *DifferentiableEigen.jl* (https://github.com/ThummeTo/DifferentiableEigen.jl), which also includes the example scripts to reproduce the experiments presented.

Common neural ODEs tend to converge to local minima and are not protected from becoming unsolvable. We showed that eigen-informed neural ODEs are capable of avoiding or reducing these problems and outperform pure neural ODEs in the presented disciplines. In three academic examples, we investigated that eigen-informed neural ODEs are able to converge in fewer steps, generalize better, and even lead to a numerical system that can be solved faster, compared to pure neural ODEs (see Table 11). We further exemplified that eigen-informed neural ODEs are capable of handling modeling applications that are unsolvable for common neural ODEs: Eigen-informed neural ODEs allow for training on the basis of insufficient data — meaning data that does not fulfill the Nyquist-Shannon sampling theorem — if the system eigenmodes are known.

Last but not least, knowledge of ODE properties like stability, frequencies, damping, and/or stiffness can improve training convergence

Table 11
Quantitative summary (factors of improvement) of results. The values compare the median for selected eigenvalue-regularization experiments and the median of the corresponding neural ODE without eigen-informed regularization. Values larger than 1 correspond to improvements.

Example	Loss SOL + FRQ + DMP	Accuracy (testing)	Reduced solv. time	Conv. speed
Pendulum	+ OSC	≈ 2.09	≈ 1.12	≈ 4.76
Pendulum (undersamp.)	+ STB + OSC	≈ 6.92	n.a.	≈ 6.41
VPO	+ STB + OSC	≈ 1.33	≈ 1.08	≈ 1.17

in many different use cases, regardless of issues from local minima or instability. The presented technique, the eigen-informed neural ODE, can often be implemented for a moderate additional computational cost compared to the deployment of pure neural ODEs, especially if implicit solvers are used. Whether the trade-off (additional computational cost, but faster convergence) is economical must be examined on a case-by-case basis. For problems that are not solvable at all by pure neural ODEs, deploying the presented method will often be beneficial.

4.2. Limitations

The presented method of eigen-informed training features limitations in its current state, which are briefly highlighted in this section.

Need for a priori knowledge. This work is largely based on the availability of additional system knowledge (such as stability, frequencies, etc.) next to pure data. A legitimate question is therefore whether and to what extent such system knowledge is available in real applications. The answer to this question depends largely on the area of application, and there are certainly areas in which little or none of this additional knowledge is available. Nevertheless, we would like to highlight three examples of areas of application where additional system knowledge is often available and which are therefore predestined candidates for the application of this method:

Mechanical engineering The eigenvalue positions of classical mechanical systems are mainly defined by the stiffness, damping, and mass/inertia of the components. Engineers can intuitively derive system properties and thus eigenvalue positions from the component parameters.

Electrical engineering Similarly, the eigenvalues in electrical systems are determined by the known parameters of the installed components (e.g. capacitors and inductors). Especially oscillating systems (*oscillating circuits*) are widely studied; for example, especially frequencies of circuits can be approximated using various rules of thumb.

Fluid Dynamics Further, in computational fluid dynamics, some systems can be linearized for examination, especially if in the steady state. Different fluid properties, such as the well-known Reynolds number, correlate with system properties such as stiffness and stability. Experts in the field are able to define limit values for stiffness and stability for a given simulation, which, in turn, can be used for regularization.

Of course, there are also counterexamples, e.g., pure black-box applications are unsuitable for the application of the method, i.e., if no prior knowledge of the system to be modeled is available (e.g., brain dynamics) or should be assumed (e.g., climate models). As a final note, the properties of the system could also be determined based on data analysis (operational modal analysis). Nevertheless, the usual limitations for data quality (e.g., limited noise) also apply here.

Validity of linearization. Although the method is widely applicable for linear systems, nonlinear systems must first be linearized. Linearization is not suitable for all types of nonlinear systems. For example, the informative value of linearization is limited near chaotic, branching, and

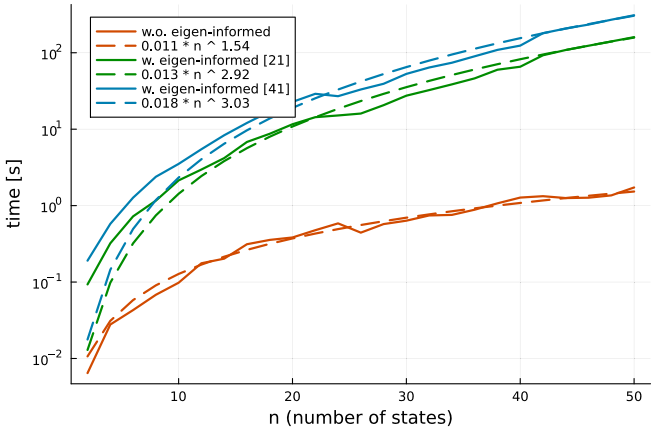


Fig. 11. Comparision of the computation time for a neural ODE gradient with and without eigen-informed regularization. The number of linearization points for eigen-informed training is given in square brackets. The experimental curves are fitted by least-squares approximations of exponential functions (coefficients given in the legend). Note, that the y-axis features a logarithmic scale.

discontinuous points in state space. For systems including such features, it must be investigated separately whether the proposed method can be applied, and making general statements on these classes of systems remains open area of research.

Computational overhead. Although we showed that eigen-informed training is beneficial in various aspects, we also noted the additional complexity of the approach, caused by *nested differentiation* that is required for linearization during the computation of the parameter gradient. Whereas we highlighted that the obtained Jacobian could be reused (e.g., implicit solvers), we did no performance evaluation for this strategy. For explicit solvers, we note a significantly longer computation time for gradient determination (by factors of two to four) compared to common neural ODE training. The size of the matrix for the determination of the eigenvalues equals the number of states within the system. Investigation of the computational cost for the determination of the eigenvalues and the sensitivity analysis of eigenvalue algorithms in the presence of larger (non-academic) systems requires detailed examinations and further optimizations to make eigen-informed training efficient. Special algorithms for sparse systems need to be considered. Note that, for example, the proposed sensitivity equations involve matrix inversion, which is computationally expensive, especially for large matrices. This can be quantitatively shown by investigating the computation times for gradients for an example ODE in Fig. 11.

As can be seen, the cost for both approaches is approximately increasing exponentially with the size of the system n. Whereas the cost for training a pure neural ODE training is $\approx n^{1.5}$, the cost of eigen-informed training is $\approx n^3$. This can be justified by matrix inversion within the AD rules, which has a computational complexity of $\mathcal{O}(n^3)$ in general. However, methods for sparse matrices can be considered for such cases but are part of current and future work. Furthermore, for training in large applications with many states and eigenvalues, a reliable eigenvalue tracking method as in [32,33] and its application are an active research topic.

Applicability to real-world models. Although we investigated the benefits of eigen-informed training for academic example ODEs, the applicability to real-world applications was not investigated in this article. This covers, for example, challenges such as high-dimensionality, multi-timescale or hybrid systems, and noisy, incomplete, or uncertain measurements (data).

All mentioned limitations are major parts of future work planned on the topic.

4.3. Future research & research questions

Further future and open research questions are given in the following section.

Detailed analysis of the (multi-objective) optimization problem. The structure of the multi-objective optimization problem could be examined in more detail, similar to, e.g., in [34]. For example, the individual error terms that make up the cost function (and thus the gradient) can be examined individually and discussed. This includes in particular constraints that lead to contradictory optimization directions. Exploring the Pareto front and investigation of, e.g., adaptive balancing between objectives (in this work, we use statically determined factors) seem to be intriguing research directions. Finally, investigation of training on multiple (possibly switching) gradients, e.g., based on a gradient criterion, is an interesting direction for future research.

Other types of differential equations. Basically, the presented method is not limited to the use in neural ODEs but opens up to the entire family of (neural) differential equations. The presented *eigen-informed* optimization is applicable to many other (engineering) applications and to parameter optimization of differential equations in general. This, for example, includes neural partial differential equations that are often used for fluid dynamics simulations which have a natural tendency toward stiffness and instability caused by the problem domain.

Error functions. In addition to the very simple error functions presented for eigenvalues, there are many other interesting options to explore. In addition to specific locations for eigenvalues, geometric areas can be defined that correspond to intuitive system properties.

Collocation. Eigen-informed training is not limited to training on the neural ODE solution. It is also applicable to training on the neural ODE derivative, for example, in *collocation training* [18]. Even though this should be possible from a theoretical point of view (collocation can be seen as a simplification of a training on the solution), this was not further investigated in this work, and the final validation is therefore future research.

Further system attributes. In addition to the system properties investigated, there are a variety of additional attributes from system and control theory [26,27] that might be interesting to incorporate as part of eigen-informed neural ODEs. Furthermore, the investigations in this article were restricted to the system matrix $\boldsymbol{A} = \frac{\partial \dot{\boldsymbol{x}}}{\partial \boldsymbol{x}}$. For systems with inputs $\boldsymbol{u}$ and outputs, additional system properties, such as, e.g., controllability, depend on the linearization $\boldsymbol{B} = \frac{\partial \dot{\boldsymbol{x}}}{\partial \boldsymbol{u}}$. Investigation of further system properties depending on other linearizations in addition to $\boldsymbol{A}$ to tackle controllability, observability, and more, within neural ODEs seems a promising direction for future research.

Alternatives to QR-algorithm. In general, other algorithms for eigenvalue approximation, like computation via ANNs as in [35] could be interesting, due to the seamless compatibility with AD and (possibly) improved computational performance. In addition, there are different methods to estimate the largest eigenvalue for a moderate computational cost compared to the calculation of *all* eigenvalues, which could be an efficient way to estimate the stability gradient / regularization.

Detailed investigation of extrapolation. Because eigen-informed neural ODEs offer the potential to apply soft constraints to the model that are valid beyond training data, we assume improved extrapolation behavior. This includes tests w.r.t. small unknown deviations (like perturbations around the actual training data) and large unknown deviations, like generalization beyond the subpart of state space the model was trained on. However, this will be validated in detail as part of future work.

Controller design. Future work covers the extension of this concept to the use in physics-enhanced neural ODEs [3], the combination of a conventional ODE, ANNs and an ODE solver and universal differential equations [24] in general. Besides dealing with instability and local minima, this will also open up to nonlinear control design. We will show that eigen-informed physics-enhanced neural ODEs can be used to train powerful nonlinear controllers, simply by defining a loss function based on a target trajectory, together with an additional gradient/regularization for pushing the unstable eigenvalues to the left half of the complex plane — satisfying the universal controller design objective.

CRediT authorship contribution statement

Tobias Thummerer: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Conceptualization. **Lars Mikelsons:** Writing – review & editing, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Funding

The contributions to this research were partially funded by the project: Unleash Potentials in Simulation (UPSIM), ITEA3-Project, Nr. 19006, https://www.upsim-project.eu/. The authors thank the organizations for their funding.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Tobias Thummerer reports that financial support was provided by Federal Ministry of Education and Research Berlin Office. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data and code are open source on GitHub: https://github.com/ThummeTo/DifferentiableEigen.jl

References

[1] T.Q. Chen, Y. Rubanova, J. Bettencourt, D. Duvenaud, Neural ordinary differential equations, CoRR abs/1806.07366, arXiv preprint arXiv:1806.07366, (2018) http://arxiv.org/abs/1806.07366.

[2] A. Ramadhan, J. Marshall, A. Souza, G.L. Wagner, M. Ponnapati, C. Rackauckas, Capturing missing Physics in climate model parameterizations using neural differential equations, (2020) https://doi.org/10.48550/ARXIV.2010.12559, https://arxiv.org/abs/2010.12559.

[3] T. Thummerer, J. Stoljar, L. Mikelsons, NeuralFMU: presenting a workflow for integrating hybrid neuralodes into real-world applications, Electronics 11 (19) (2022) 3202, https://doi.org/10.3390/electronics11193202.

[4] V. Tac, F.S. Costabal, A.B. Tepole, Data-driven tissue mechanics with polyconvex neural ordinary differential equations, Comput. Methods Appl. Mech. Eng. 398 (2022) 115248, https://doi.org/10.1016/j.cma.2022.115248.

[5] X. Xie, A.K. Parlikad, R.S. Puri, A neural ordinary differential equations based approach for demand forecasting within power grid digital twins, in: 2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), 2019, pp. 1–6, https://doi.org/10.1109/SmartGridComm.2019.8909789.

[6] P. Hartman, A lemma in the theory of structural stability of differential equations, Proc. Amer. Math. Soc. 11 (4) (1960) 610–620.

[7] D.M. Grobman, Homeomorphism of systems of differential equations, Dokl. Akad. Nauk SSSR 128 (5) (1959) 880–881.

[8] G. Dahlquist, Convergence and stability in the numerical integration of ordinary differential equations, Math. Scand. 4 (1956) 33–53.

[9] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Y.W. Teh, M. Titterington (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics of Proceedings of Machine Learning Research, vol. 9, PMLR, Chia Laguna Resort, Sardinia, Italy, 2010, pp. 249–256. https://proceedings.mlr.press/v9/glorot10a.html.

[10] A. Tuor, J. Drgona, D. Vrabie, Constrained neural ordinary differential equations with stability guarantees, (2020) https://doi.org/10.48550/ARXIV.2004.10883, https://arxiv.org/abs/2004.10883.

[11] Q. Kang, Y. Song, Q. Ding, W.P. Tay, Stable neural ODE with Lyapunov-stable equilibrium points for defending against adversarial attacks, 2021, https://doi.org/10.48550/ARXIV.2110.12976. https://arxiv.org/abs/2110.12976.

[12] I.D.J. Rodriguez, A. Ames, Y. Yue, Lyanet: a Lyapunov framework for training neural ODES, in: International Conference on Machine Learning, PMLR, 2022, pp. 18687–18703.

[13] D. Floryan, On instabilities in neural network-based Physics simulators, arXiv preprint arXiv:2406.13101, (2024). https://arxiv.org/abs/2406.13101.

[14] S. Kim, W. Ji, S. Deng, Y. Ma, C. Rackauckas, Stiff neural ordinary differential equations, chaos, An Interdiscip. J. Nonlinear Sci. 31 (9) (2021).

[15] A. Ghosh, H.S. Behl, E. Dupont, P.H.S. Torr, V. Namboodiri, STEER, Simple temporal regularization for neural ODEs, arXiv preprint arXiv:2006.10711, (2020) https://arxiv.org/abs/2006.10711.

[16] E.M. Turan, J. Jäschke, Multiple shooting for training neural differential equations on time series, IEEE Control Syst. Lett. 6 (2021) 1897–1902.

[17] M. Raissi, P. Perdikaris, G. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707, https://doi.org/10.1016/j.jcp.2018.10.045.

[18] E. Roesch, C. Rackauckas, M.P.H. Stumpf, Collocation based training of neural ordinary differential equations, Stat. Appl. Genet. Mol. Biol. 20 (2) (2021) 37–49, https://doi.org/10.1515/sagmb-2020-0025, (cited 2023-Dec-29).

[19] U.M. Ascher, L.R. Petzold, Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998. arXiv:https://doi.org/10.1137/1.9781611971392. https://epubs.siam.org/doi/pdf/10.1137/1.9781611971392.

[20] J.G.F. Francis, The QR transformation—part 2, Comp. J. 4 (4) (1962) 332–345. arXiv:https://doi.org/10.1093/comjnl/4.4.332, https://academic.oup.com/comjnl/article-pdf/4/4/332/8201663/040332.pdf.

[21] F. Sapienza, J. Bolibar, F. Schäfer, B. Groenke, A. Pal, V. Boussange, P. Heimbach, G. Hooker, F. Pérez, P.-O. Persson, C. Rackauckas, Differentiable programming for differential equations: a review, arXiv preprint arXiv:2406.09699, (2024). https://arxiv.org/abs/2406.09699.

[22] M.B. Giles, An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation, (2008).

[23] T. Thummerer, L. Mikelsons, J. Kircher, NeuralFMU: towards structural integration of FMUs into Neural networks, in: M. Sjölund; L. Buffoni; A. Pop, L. Ochel (Eds.), Proceedings of 14th Modelica Conference 2021, September 20-24, 2021, Linköping, Sweden, 2021, https://doi.org/10.3384/ecp21181297.

[24] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, A. Edelman, Universal differential equations for scientific machine learning, arXiv preprint arXiv:2001.04385, (2021).

[25] F. Sorourifar, Y. Peng, I. Castillo, L. Bui, J. Venegas, J.A. Paulson, Physics-enhanced neural ordinary differential equations: application to industrial chemical reaction systems, Ind. Eng. Chem. Res. 62 (38) (2023) 15563–15577.

[26] N.S. Nise, Control Systems Engineering, John Wiley & Sons, 2020.

[27] P.J. Antsaklis, A.N. Michel, A Linear Systems Primer, Springer Science & Business Media, 2007.

[28] C. Tsitouras, Runge–kutta pairs of order 5(4) satisfying only the first column simplifying assumption, Comput. Math. Appl. 62 (2) (2011) 770–775, https://doi.org/10.1016/j.camwa.2011.06.002.

[29] D.P. Kingma, J. Ba, Adam: a method for stochastic optimization, arXiv preprint arXiv:1412.6980, (2017).

[30] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah, Julia: a fresh approach to numerical computing, SIAM Rev. 59 (1) (2017) 65–98, https://doi.org/10.1137/141000671, 10.1137/141000671.

[31] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, V. Dixit, DiffEqFlux.jl - a julia library for neural differential equations, CoRR abs/1902.02376, arXiv preprint arXiv:1902.02376, (2019) http://arxiv.org/abs/1902.02376.

[32] E. Safin, D. Manteuffel, Advanced eigenvalue tracking of characteristic modes, IEEE Trans. Antennas Propag. 64 (7) (2016) 2628–2636.

[33] R. Alden, F. Qureshy, Eigenvalue tracking due to parameter variation, IEEE Trans. Autom. Control 30 (9) (1985) 923–925.

[34] N. Khodadadi, L. Abualigah, E.-S.M. El-Kenawy, V. Snasel, S. Mirjalili, An archive-based multi-objective arithmetic optimization algorithm for solving industrial engineering problems, IEEE Access 10 (2022) 106673–106698, https://doi.org/10.1109/ACCESS.2022.3212081.

[35] Z. Yi, Y. Fu, H.J. Tang, Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix, Comput. Math. Appl. 47 (8) (2004) 1155–1164, https://doi.org/10.1016/S0898-1221(04)90110-1.

Author biography

Tobias Thummerer, M.Sc. is an advanced doctoral student at University of Augsburg. He completed both his Bachelor's and Master's degree in "Engineering and Computer Sciences" at this same university. His research focuses on Scientific Machine Learning, and he is specifically engaged with the area of Hybrid Modeling, the combination of simulation models – often from engineering – and novel machine learning approaches. His research includes the development of new methods in this field, but explicitly also the improvement of new and established approaches so that they can be applied in a demanding real-world environment.

Prof. Dr.-Ing. Lars Mikelsons has served since 2018 as Chair of Mechatronics at the University of Augsburg, leading interdisciplinary research at the nexus of mechanical, electronic, thermal, and systems engineering integrated with scientific machine learning. Previously, he worked in industry as a research engineer and project manager at Bosch Rexroth and Bosch Corporate Research (2011–2018). His work focuses on model-based development and validation of mechatronic systems, virtual prototyping, deep-learning-based engineering methods, and in particular hybrid modeling methods.