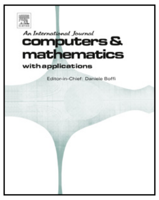
Contents lists available at ScienceDirect

Computers and Mathematics with Applications

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



Comparison of neural closure models for discretised PDEs

Hugo Melchers [a,c,1], Daan Crommelin [a,b], Barry Koren [c], Vlado Menkovski [c], Benjamin Sanderse [a,*]

[a] Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, the Netherlands
[b] Korteweg-de Vries Institute for Mathematics, University of Amsterdam, Science Park 105-107, 1098 XG, Amsterdam, the Netherlands
[c] Eindhoven University of Technology, De Zaale, 5600 MB, Eindhoven, the Netherlands

ARTICLE INFO

Dataset link: https://github.com/HugoMelchers/neural-closure-models

Keywords:
Ordinary differential equations
Neural networks
Neural ODE
Partial differential equations
Multiscale modelling
Closure model

ABSTRACT

Neural closure models have recently been proposed as a method for efficiently approximating small scales in multiscale systems with neural networks. The choice of loss function and associated training procedure has a large effect on the accuracy and stability of the resulting neural closure model. In this work, we systematically compare three distinct procedures: "derivative fitting", "trajectory fitting" with discretise-then-optimise, and "trajectory fitting" with optimise-then-discretise. Derivative fitting is conceptually the simplest and computationally the most efficient approach and is found to perform reasonably well on one of the test problems (Kuramoto-Sivashinsky) but poorly on the other (Burgers). Trajectory fitting is computationally more expensive but is more robust and is therefore the preferred approach. Of the two trajectory fitting procedures, the discretise-then-optimise approach produces more accurate models than the optimise-then-discretise approach. While the optimise-then-discretise approach can still produce accurate models, care must be taken in choosing the length of the trajectories used for training, in order to train the models on long-term behaviour while still producing reasonably accurate gradients during training. Two existing theorems are interpreted in a novel way that gives insight into the long-term accuracy of a neural closure model based on how accurate it is in the short term.

1. Introduction

A number of real-world phenomena, such as fluid flows, can be modelled numerically as a system of partial differential equations (PDEs). Such PDEs are typically solved by discretising them in space, yielding ordinary differential equations (ODEs) over a large number of variables. These full-order models (FOMs) are generally very accurate, but can be computationally expensive to solve. A remedy against this high computational cost is to use 'truncated' models. These do not directly resolve all spatial and/or temporal scales of the true solution of the underlying PDE, thereby lowering the dimensionality of the model. Approaches to create lower dimensional models include reduced-order modelling (ROM [31]), as well as large eddy simulation (LES [30]) and Reynolds-averaged Navier-Stokes (RANS [2]) for fluid flow problems, specifically. In such a truncated model, one or more closure terms appear, which represent the effects that are not directly resolved by the reduced-order model. For a recent overview of closure modelling for reduced-order models, see Ahmed et al. [1].

While closure terms can in some cases be derived from theory (for example, for LES), this is generally not the case. When they cannot be derived from theory, a recent approach is to use a machine learning model to learn the closure term from data. In this approach a specific type of machine learning model is used, called a neural closure model [10]. The overall idea is to approximate a PDE or large ODE system by a smaller ODE system, and to train a neural network to correct for the approximation error in the resulting ODE system. Neural closure models are a special form of neural ODEs [4], which have been the subject of extensive research in the past years, for example by Finlay et al. [7] and Massaroli et al. [21].

A number of different approaches for training neural ODEs and neural closure models are available. An important distinction is between approaches that compare predicted and actual time-derivatives of the ODE ("derivative fitting"), and approaches that compare predicted and actual solutions ("trajectory fitting"). Trajectory fitting itself can be done in two ways, depending on whether the optimisation problem for the neural network is formulated as continuous in time and then discre-

* Corresponding author at: Centrum Wiskunde & Informatica (CWI), Amsterdam, the Netherlands.
E-mail address: b.sanderse@cwi.nl (B. Sanderse).
1 Work performed while at Centrum Wiskunde & Informatica.

https://doi.org/10.1016/j.camwa.2023.04.030

Abbreviations

AD	Automatic differentiation	PDE	Partial differential equation
CNN	Convolutional neural network	POD	Proper orthogonal decomposition
FOM	Full-order model	RANS	Reynolds-averaged Navier-Stokes
LES	Large eddy simulation	RMSE	Root-mean-square error
MOR	Model order reduction	RNN	Recurrent neural network
MSE	Mean-square error	ROM	Reduced-order modelling
NN	Neural network	VPT	Valid prediction time
ODE	Ordinary differential equation		

tised using an ODE solver (optimise-then-discretise), or formulated as discrete in time (discretise-then-optimise).

In several recent studies, neural closure models have been applied to fluid flow problems. The considered approaches include derivative fitting [9,26,20,3], discretise-then-optimise [18], and optimise-then-discretise [33,20]. Derivative fitting is also used on a comparable but distinct problem by San and Maulik [32]. There, Burgers' equation is solved using model order reduction (MOR) by means of proper orthogonal decomposition (POD), resulting in an approximate ODE for which the closure term is approximated by a neural network.

Training neural ODEs efficiently and accurately has been the subject of some previous research. However, in the context of neural closure models, most of this earlier work either does not consider certain relevant aspects or is not directly applicable. For example, Onken and Ruthotto [24] compare discretise-then-optimise and optimise-then-discretise for pure neural ODEs (i.e. ODEs in which the right-hand side only consists of a neural network term). They omit a derivative fitting approach since such an approach is not applicable in the contexts considered there. Ma et al. [19] compare a wide variety of training approaches for neural ODEs, however with an emphasis on the computational efficiency of different training approaches rather than on the accuracy of the resulting model. Roesch et al. [29] compare trajectory fitting and derivative fitting approaches, considering pure neural ODEs on two very small ODE systems. As a result, the papers mentioned above are not fully conclusive to make general recommendations regarding how to train neural closure models.

The purpose of this paper is to perform a systematic comparison of different approaches for constructing neural closure models. Compared to other works, the experiments performed here are not aimed at showing the efficacy of neural closure models for a particular problem type, but rather at making general recommendations regarding different approaches for neural closure models. To this end, neural closure models are trained on data from two different discretised PDEs, in a variety of ways. One of these PDEs, the Kuramoto-Sivashinsky equation, is chaotic and discretised into a stiff ODE system. This gives rise to additional challenges when training neural closure models. The results of this paper confirm that discretise-then-optimise approaches are generally preferable to optimise-then-discretise approaches. Furthermore, derivative fitting is found to be unpredictable, producing excellent models on one test set, but very poor models on the other. We give theoretical support to our results by reinterpreting two fundamental theorems from the fields of dynamical systems and time integration in terms of neural closure models.

This paper is organised as follows: Section 2 describes a number of different approaches that are available for training neural closure models. Section 3 gives a number of theoretical results that can be used to predict the short-term and long-term accuracy of models based on how they are trained and what error they achieve during training. Section 4 performs a number of numerical experiments in which the same neural closure model is trained in multiple ways on the same two test equations, and the accuracy of the resulting models is compared. Finally, Section 5 provides conclusions and recommendations. The code used to perform the numerical experiments from Section 4 is available online at https://github.com/HugoMelchers/neural-closure-models.

2. Preliminaries: approaches for neural ODEs

In this paper, neural networks will be used as closure models for discretised PDEs. Here, a time-evolution of the form $\frac{\partial u}{\partial t} = F(u)$ is discretised into an ODE system $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}), \mathbf{u} \in \mathbb{R}^{N_x}$, such that taking progressively finer discretisations (resulting in larger values of N_x) produces more accurate solutions. However, instead of taking a very fine discretisation, a relatively coarse discretisation will be used and a neural network (NN) closure term will be added to correct for the spatial discretisation error. This neural network depends not only on the vector $\mathbf{u}$, but also on a vector ϑ of trainable parameters:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \mathrm{NN}(\mathbf{u}; \vartheta). \tag{1}$$

Some of the theory regarding neural closure models also applies to neural ODEs, in which the neural network is the only term in the right-hand side:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathrm{NN}(\mathbf{u}; \vartheta). \tag{2}$$

In both cases, the result is a system of ODEs over a vector $\mathbf{u}(t)$, in which the right-hand side depends not only on $\mathbf{u}(t)$ but also on some trainable parameters ϑ:

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = g(\mathbf{u}; \vartheta). \tag{3}$$

Note that in these models, the ODE is assumed to be autonomous, i.e., the right-hand side is assumed to be independent of t. However, the work presented in this paper can be extended to non-autonomous ODEs, by extending the neural network to depend on t or on some time-dependent control signal as well as on $\mathbf{u}(t)$, and by including this additional data in the training data set. The general form (3) covers more model types than just the neural ODEs and neural closure models of equations (1) and (2). Specifically, the output of the neural network does not have to be one of the terms in the right-hand side function, but can also be included in other ways. For example, Beck et al. [3] train neural networks to predict the eddy viscosity in an LES closure term, rather than to predict the entire closure term, in order to ensure stability of the resulting model. In this work, the specific form (1) is used, with the exception that the output of the neural network is passed through a simple linear function Δ_{fwd}, listed as a non-trainable layer in Tables B.4 and B.5, which ensures that the solutions of the neural ODE satisfy conservation of momentum.

Training a neural network corresponds to minimising a certain loss function, which must be chosen ahead of time. Some loss functions are such that their gradients, which are used by the optimiser, can be computed in different ways. In this section, an overview of different available approaches is given.

2.1. Derivative fitting

With derivative fitting, also referred to as non-intrusive training [28], the loss function compares the predicted and actual time-derivatives (i.e. right-hand sides) of the neural ODE (Fig. 1). In this

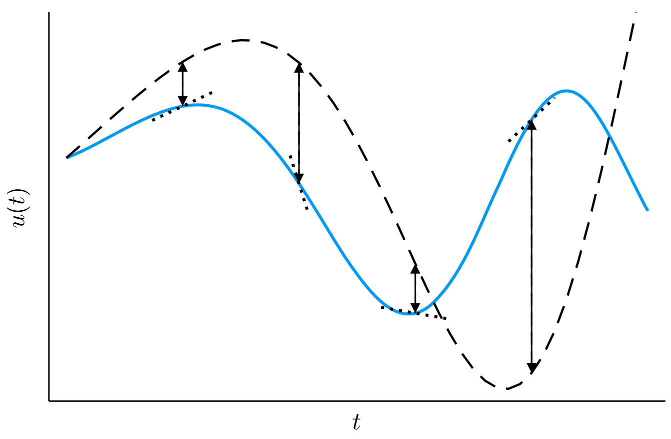


Fig. 1. A visual comparison of the two types of neural ODE training: given a reference trajectory (solid), one can train the neural ODE to match the time-derivative of the trajectory (dotted lines), or to result in accurate ODE solutions (dashed line and arrows).

paper, the loss function used for derivative fitting will be a mean-square error (MSE):

$$\text{Loss}\left(\vartheta, \mathbf{u}_{\text{ref}}, \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u}_{\text{ref}}\right) = \frac{1}{N_x N_s N_p} \sum_{i=1}^{N_s} \sum_{j=1}^{N_p} \left\| \frac{\mathrm{d}\mathbf{u}_{\text{ref}}^{(j)}}{\mathrm{d}t}(t_i) - g\left(\mathbf{u}_{\text{ref}}^{(j)}(t_i); \vartheta\right) \right\|_2^2 . \quad (4)$$

Here, N_x is the size of the vector $\mathbf{u}_{\text{ref}}$, N_s is the number of snapshots in each trajectory of the training data, and N_p is the number of trajectories (i.e. ODE solutions). The value and time-derivative of the j^{th} trajectory at time t_i are given by $\mathbf{u}_{\text{ref}}^{(j)}(t_i)$ and $\frac{\mathrm{d}\mathbf{u}_{\text{ref}}^{(j)}}{\mathrm{d}t}(t_i)$, respectively.

The main advantage of derivative fitting is that in order to compute the gradient of the loss function with respect to ϑ, one only has to differentiate through the neural network itself. This makes derivative fitting a relatively simple approach to use. A disadvantage of derivative fitting is that the training data must consist of not just the values $\mathbf{u}$, but also their time derivatives $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}$. This data is not always available, for example in cases where the trajectories $\mathbf{u}(t)$ are obtained as measurements from a physical experiment. In this work, the training data is obtained through a high-resolution numerical algorithm. Hence, the derivatives to be used for training are available. In cases where exact derivatives are not available, they can be estimated from the available data for $\mathbf{u}(t)$ itself, as described by Roesch et al. [29]. While they obtain good results with approximated derivatives, in general it is to be expected that substituting real time-derivatives by approximations also decreases the accuracy of the neural network.

2.2. Trajectory fitting: discretise-then-optimise

An alternative to derivative fitting is trajectory fitting, also referred to as intrusive training [28], embedded training [20], or a solver-in-the-loop setup [37]. Here, the loss function compares the predicted and actual trajectories of the neural ODE (Fig. 1). Unless otherwise specified, trajectory fitting will also be done with the MSE loss function:

$$\text{Loss}\left(\vartheta, \mathbf{u}_{\text{ref}}\right) = \frac{1}{N_x N_t N_p} \sum_{i=1}^{N_t} \sum_{j=1}^{N_p} \left\| \mathbf{u}^{(j)}(t_i) - \mathbf{u}_{\text{ref}}^{(j)}(t_i) \right\|_2^2 , \quad (5)$$

$$\text{where } \frac{\mathrm{d}\mathbf{u}^{(j)}}{\mathrm{d}t} = g\left(\mathbf{u}^{(j)}; \vartheta\right) \text{ and } \mathbf{u}^{(j)}(0) = \mathbf{u}_{\text{ref}}^{(j)}(0). \quad (6)$$

Trajectory fitting involves applying an ODE solver to the neural closure model to perform N_t time steps, where N_t is a hyper-parameter that must be chosen ahead of time. Computing the gradient of the loss function involves differentiating through the ODE solving process and can be done in two separate ways. One way to do this is by directly differentiating through the computations of the ODE solver. This approach is called **discretise-then-optimise**.

In the discretise-then-optimise approach, the neural ODE is embedded in an ODE solver, for example an explicit Runge-Kutta method. In such an ODE solver, the next solution snapshot $\mathbf{u}(t + \Delta t)$ is computed from $\mathbf{u}(t)$ by performing one step of the ODE solver, which generally involves applying the internal neural network multiple times (depending on the number of stages of the ODE solver). This is repeated to obtain a predicted trajectory over a time interval of length $T = N_t \Delta t$. Since all the computations done by an ODE solver are differentiable, one can simply compute the required gradient by differentiating through all the time steps performed by the ODE solver. The discretise-then-optimise approach effectively transforms a neural ODE into a discrete model, in which the time series is predicted by advancing the solution by a fixed time step Δt at a time. As such, any training approach that can be applied to discrete models of the form $\mathbf{u}(t + \Delta t) = \text{model}(\mathbf{u}(t))$ can also be applied to neural ODEs trained using this approach.

2.3. Trajectory fitting: optimise-then-discretise

Differentiating through the computations of the ODE solver is not always a possibility, for example if the ODE solver is implemented as black-box software. In such cases, trajectory fitting with the loss function (5) can still be used by computing gradients with the **optimise-then-discretise** approach. In this approach, the required gradients are computed either by extending the ODE with more variables that store derivative information, or by solving a second "adjoint" ODE backwards in time after the original "forward" ODE solution is computed. These two methods can be considered continuous-time analogues to forward-mode and reverse-mode automatic differentiation (AD), respectively.

The adjoint ODE approach was popularised by Chen et al. [4], who demonstrate that on some problems the adjoint ODE approach can be used to train a neural ODE with much lower memory usage than other approaches. Ma et al. [19] find that the adjoint ODE approach is computationally more efficient than the forward mode approach for ODEs with more than 100 variables and parameters. As such, a description of the forward mode approach is omitted here. The adjoint ODE approach is the optimise-then-discretise approach that will be tested here. This approach can be implemented in three different ways. The implementation used in this work is the *interpolating adjoint method*, in which the gradient of the loss function is computed by first solving the forward ODE (3) to obtain the trajectory $\mathbf{u}(t)$, and then solving the adjoint ODE system

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{y}^\top = -\mathbf{y}^\top \frac{\partial}{\partial \mathbf{u}} g(\mathbf{u}; \vartheta), \quad \mathbf{y}(T) = \mathbf{0}, \quad (7a)$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{z}^\top = -\mathbf{y}^\top \frac{\partial}{\partial \vartheta} g(\mathbf{u}; \vartheta), \quad \mathbf{z}(T) = \mathbf{0}, \quad (7b)$$

from $t = T$ backwards in time until $t = 0$, performing discrete updates to $\mathbf{y}(t)$ at times $t_i, i = N_t, N_t - 1, \ldots, 2, 1$. After the adjoint ODE system is solved, the gradient $\frac{\mathrm{dLoss}}{\mathrm{d}\vartheta}$ is given by $\mathbf{z}(0)$. For implementation details and an overview of other optimise-then-discretise methods, see Chapter 3 of Melchers [22].

Note that the two trajectory fitting approaches, i.e. discretise-then-optimise and optimise-then-discretise, both require choosing N_t, the number of time steps that the solution prediction is computed for. As will be described in Section 3, choosing N_t either too small or too large may have negative consequences for the accuracy of the trained model. For the optimise-then-discretise approach, the gradients used by the optimiser are computed as the solution of an ODE over a time span of $T = N_t \Delta t$. Since the numerically computed ODE solution is inexact, choosing a larger value of N_t will generally result in less accurate gradients, which may also decrease the accuracy of the trained model.

2.4. Algorithm comparison

An overview of the advantages and disadvantages of different approaches is given in Table 1. Here, the term 'long-term' refers to the

Table 1
An overview of the differences between the three training approaches outlined in Section 2.

	Derivative fitting	Trajectory fitting	
		Discretise-then-optimise	Optimise-then-discretise
Differentiability required	NN	NN, f, ODE solver	NN, f
Accuracy of loss function gradients	Exact	Exact	Approximate
Learns long-term accuracy	No	Yes	Yes
Requires time-derivatives of training data	Yes	No	No
Computational cost	Low	High	High

accuracy of predictions when solving the ODE over multiple time steps as opposed to only considering the instantaneous error in the time-derivative of the solution. Note that the computational cost will not be compared in this work; the goal is to compare the accuracy of the resulting models. Performance measurements of different training procedures will not be given here since the code used to perform the numerical experiments in this work was not written with computational efficiency in mind, and since training was not performed on recent hardware. However, derivative fitting is expected to be computationally more efficient due to the fact that it does not require differentiating through the ODE solution process. A performance comparison between different implementations for optimise-then-discretise and discretise-then-optimise approaches is given by Ma et al. [19]. The performance difference between derivative fitting and trajectory fitting will be taken into account when making recommendations in Section 5.

As for accuracy, the discretise-then-optimise approach is expected to yield more accurate gradients than optimise-then-discretise, due to the absence of temporal discretisation errors in the gradient computation. The accuracy of derivative fitting is not easily compared to that of the other methods. Like optimise-then-discretise, it suffers from the fact that the training does not take the temporal discretisation error of the ODE solver into account.

Onken and Ruthotto [24] compare discretise-then-optimise and optimise-then-discretise approaches for two problems, including a time series regression similar to the trajectory fitting problem described earlier in this section. Their findings indicate that training with discretise-then-optimise results in computationally more efficient training (less time required per epoch), as well as faster convergence (fewer epochs required to reach a given level of accuracy). However, the trajectory regression test performed there only considers a small and relatively simple ODE system over just two variables. Furthermore, the use of neural networks as closure terms introduces some additional challenges that need to be overcome in some cases, including solving the ODE (2) efficiently for stiff ODEs, and choosing the value of N_t for chaotic systems.

3. Theory concerning training approaches

As described in the previous section, different training approaches are available for neural ODEs and neural closure models. The training approach used will generally have an effect on both the short-term accuracy and the long-term accuracy of the trained model. This is supported by the following two theorems, which use the short-term error of a model to provide upper bounds on the long-term error. Here, ‘short-term’ refers to the predictions and prediction errors in the time-derivatives (for derivative fitting), or after a single time step (for trajectory fitting). The term ‘long-term’ refers to the predictions after multiple time steps, i.e. when predicting over a time interval of $T > \Delta t$. Although neither of these theorems are new, they are interpreted in a novel way that gives insight regarding the long-term accuracy of models based on their accuracy during training.

3.1. Derivative fitting

For models trained using derivative fitting, a relation between the error of the right-hand sides of the ODE and the error of the ODE solutions is given by Theorem 10.2 of Hairer et al. [11]. This theorem is referred to there as the ‘fundamental lemma’ and is repeated here in a simplified form:

Theorem 3.1 *(Hairer et al. [11]). Let $\mathbf{u}_{\text{ref}}(t) \in \mathbb{R}^{N_x}, t \geq 0$ be given, let $\mathbf{u}(t) \in \mathbb{R}^{N_x}, t \geq 0$ be the solution of the ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = g(\mathbf{u}; \vartheta)$ with initial condition $\mathbf{u}(0) = \mathbf{u}_{\text{ref}}(0)$, and let $\|\cdot\|$ be a norm on $\mathbb{R}^{N_x}$. If the following holds:*

a) $\left\| \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{u}_{\text{ref}}(t) - g(\mathbf{u}_{\text{ref}}(t); \vartheta) \right\| \leq \varepsilon,$

b) $\|g(\mathbf{a}; \vartheta) - g(\mathbf{b}; \vartheta)\| \leq C \|\mathbf{a} - \mathbf{b}\|,$

for fixed Lipschitz constant $C > 0$ and fixed $\varepsilon > 0$. Then the following error bound holds:

$$\|\mathbf{u}_{\text{ref}}(t) - \mathbf{u}(t)\| \leq \frac{\varepsilon}{C}\left(e^{Ct} - 1\right).$$

Theorem 3.1 can be interpreted as follows: suppose that a machine learning model is trained to predict the time-derivatives of trajectories $\mathbf{u}_{\text{ref}}(t)$. The result is a function $g(\cdot; \vartheta)$ that computes the right-hand side of an ODE. Then, achieving an error $\leq \varepsilon$ in the short term does not guarantee a low error in the long term. The error at time t *can* grow exponentially in t, with the exponential growth factor being dependent on the Lipschitz constant of the trained model.

A possible mitigation for this problem is to add a term to the loss function that penalises large Lipschitz constants in the neural network. However, in the case of neural closure models (1), penalising the Lipschitz constant of the neural network only is not expected to help much. Theorem 3.1 concerns the Lipschitz constant of the total right-hand side $g(\cdot; \vartheta)$, which equals $f(\mathbf{u}) + \text{NN}(\mathbf{u}; \vartheta)$ for neural closure models. In such cases, the Lipschitz constant of $g(\cdot; \vartheta)$ cannot be kept small by bounding the Lipschitz constant of the neural network term only. Furthermore, for closure models for stiff ODEs the function f itself has a large Lipschitz constant by definition, meaning that the Lipschitz constant of the total right-hand side will inevitably be large as well. For many problems the function f is not Lipschitz continuous at all, for example because it contains a quadratic term. In such cases, the above theorem does not provide an upper bound for the error. Additionally, even if reducing the Lipschitz constant of the entire model is an option, this may come at the expense of lower short-term accuracy (i.e. a larger value for ε).

An example where the potential exponential error increase seems to occur is encountered by Beck et al. [3]: they train neural networks to predict the closure term in three-dimensional fluid flows, and find that neural networks that predict this closure term with high accuracy can nonetheless result in inaccurate predicted trajectories. Similarly, Park and Choi [26] find that neural closure models with high accuracy on derivatives do not always produce accurate trajectories. MacArt et al. [20] encounter the same issue and instead train using optimise-then-discretise to obtain models that produce accurate solutions. It should be noted, however, that derivative fitting does not always lead to poor models. For example, Guan et al. [9] do not encounter this problem when training neural networks for LES closure terms.

3.2. Trajectory fitting

A theorem similar to Theorem 3.1 exists for models trained using trajectory fitting, either by discretise-then-optimise or by optimise-then-discretise. This theorem is likely not new, but an independently derived proof is given here. The theorem applies to all models that are used to advance a solution $\mathbf{u}(t)$ by a fixed time step Δt. As such, the theorem is not written specifically for neural closure models.

Theorem 3.2. *Let $\mathbf{u}_{\mathrm{ref}}(t_i), i = 0, 1, \dots, \dots$ be a sequence of vectors in $\mathbb{R}^{N_x}$ where $t_i = i\Delta t$, let $\|\cdot\|$ be a norm on $\mathbb{R}^{N_x}$, and let $G(\cdot;\vartheta) : \mathbb{R}^{N_x} \to \mathbb{R}^{N_x}$ be a function such that:*

a) $\left\|\mathbf{u}_{\mathrm{ref}}(t_{i+1}) - G(\mathbf{u}_{\mathrm{ref}}(t_i);\vartheta)\right\| \leq \varepsilon$ *for all* $i = 1, 2, \dots, N_t$,

b) $\|G(\mathbf{a};\vartheta) - G(\mathbf{b};\vartheta)\| \leq C\,\|\mathbf{a} - \mathbf{b}\|$ *for all* $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{N_x}$,

for fixed Lipschitz constant $C > 0, C \neq 1$ and fixed $\varepsilon > 0$. Define the sequence $\mathbf{u}(t_{i+1}) = G(\mathbf{u}(t_i);\vartheta)$ with $\mathbf{u}(0) = \mathbf{u}_{\mathrm{ref}}(0)$. Then the following error bound holds:

$$\left\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| \leq \varepsilon \frac{C^k - 1}{C - 1} \text{ for } k = 0, 1, 2, \dots.$$

Proof. For arbitrary $k \geq 1$, the following holds:

$$\begin{aligned}
&\left\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| \\
&\quad = \left\|G(\mathbf{u}(t_{k-1});\vartheta) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| \\
&\quad \leq \left\|G(\mathbf{u}(t_{k-1});\vartheta) - G(\mathbf{u}_{\mathrm{ref}}(t_{k-1});\vartheta)\right\| + \left\|G(\mathbf{u}_{\mathrm{ref}}(t_{k-1});\vartheta) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| \\
&\quad \leq C\left\|\mathbf{u}(t_{k-1}) - \mathbf{u}_{\mathrm{ref}}(t_{k-1})\right\| + \varepsilon.
\end{aligned}$$

Now, define $r_k = \left\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| + \frac{\varepsilon}{C-1}$. Then $r_0 = \frac{\varepsilon}{C-1}$ and for $k \geq 1$:

$$\begin{aligned}
r_k &= \left\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| + \frac{\varepsilon}{C-1} \\
&\leq C\left\|\mathbf{u}(t_{k-1}) - \mathbf{u}_{\mathrm{ref}}(t_{k-1})\right\| + \varepsilon + \frac{\varepsilon}{C-1} \\
&= C\left(r_{k-1} - \frac{\varepsilon}{C-1}\right) + \frac{C\varepsilon}{C-1} \\
&= Cr_{k-1}.
\end{aligned}$$

As a result, $r_k \leq C^k r_0 = \frac{C^k \varepsilon}{C-1}$, so:

$$\left\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| = r_k - \frac{\varepsilon}{C-1} \leq \varepsilon \frac{C^k - 1}{C - 1}. \qquad \square$$

The proof does not work in the case that $C = 1$ due to the use of the term $C - 1$ as a numerator, but using similar reasoning it can be shown that $\left\|\mathbf{u}(t_k) - \mathbf{u}_{\mathrm{ref}}(t_k)\right\| \leq \varepsilon k$ holds when $C = 1$. Also note that if $C < 1$ then $G(\cdot;\vartheta)$ is a contraction and the sequence $\mathbf{u}(t_k)$ will converge to a fixed point. This also results in a bounded error r_k, since properties *a*) and *b*) combined imply that the sequence $\mathbf{u}_{\mathrm{ref}}(t_k)$ is bounded as well. However, in this case the model may still be qualitatively poor, since the actual and predicted sequences may converge to different fixed points and may have different transient dynamics. As such, even in the case that $C < 1$, a low error after one time step does not imply that the behaviour over multiple time steps is accurate.

Theorem 3.2 concerns the case that a model is trained by predicting a single time step, i.e. with $N_t = 1$. However, it can be generalised to models trained with $N_t > 1$. For example, a model trained with $N_t = 2$ that achieves an error $\leq \varepsilon_2$ after two time steps can be seen as a single model that performs two applications of the inner model, meaning that Theorem 3.2 can be applied to the model $\mathbf{u} \mapsto G(G(\mathbf{u};\vartheta);\vartheta)$, which has some Lipschitz constant C_2. Then, the error at time step k is bounded by $\varepsilon_2(C_2^{k/2} - 1)/(C_2 - 1)$, since the approximation at $t = t_k$ only requires $k/2$ applications of the model.

Theorem 3.2 implies that when training models by trajectory fitting, training with a small number of predicted time steps N_t may result in models that produce poor predictions in the long term. While training with larger values of N_t still results in an exponential upper bound for the error, it is expected that a model will yield more accurate solutions in the long term if long-term errors are penalised during training.

However, if the underlying ODE or PDE is chaotic, N_t must not be too large, either. Initially similar solutions to a chaotic system diverge from each other as $\exp(\lambda_{\max} t)$ where $\lambda_{\max}$ is the Lyapunov exponent of the system, which will be further explained in Section 4.2. Then, the sensitivity of the ODE solution after time t with respect to ϑ will also grow as $\exp(\lambda_{\max} t)$. This means that when the loss function is a simple mean-square error between the predicted and actual trajectories, the gradient of this loss function with respect to the model parameters is mostly determined by the solution of the neural ODE for large t. The result is that the optimisation procedure works to decrease the long-term error instead of first decreasing the short-term error. For non-closure models (i.e. pure neural ODEs of the form $g(\mathbf{u};\vartheta) = \mathrm{NN}(\mathbf{u};\vartheta)$), this may result in poor models (see Section 5.6.2 of Melchers [22]). This issue does not appear to be as severe for neural closure models. Nonetheless it can be helpful to compensate for the exponentially increasing sensitivity by exponentially weighing the loss function:

$$\mathrm{Loss}_c\left(\vartheta, \mathbf{u}_{\mathrm{ref}}\right) = \frac{1}{N_x N_p Z} \sum_{i=1}^{N_t} \sum_{j=1}^{N_p} \exp(-2c\lambda_{\max} t_i) \cdot \left\|\mathbf{u}^{(j)}(t_i) - \mathbf{u}_{\mathrm{ref}}^{(j)}(t_i)\right\|_2^2, \tag{8a}$$

$$\text{where } Z = \sum_{i=1}^{N_t} \exp(-2c\lambda_{\max} t_i). \tag{8b}$$

This loss function generalises the mean square error (5) by allowing the prediction error at time t_i to be weighted by a factor $\exp(-2c\lambda_{\max} t_i)$. Here, the constant c can be chosen arbitrarily. Taking $c = 0$ recovers the standard mean-square error (MSE). Note that the sum is over squared errors, which grow as $\exp(2\lambda_{\max} t)$, meaning that the reasonable choice according to the above reasoning is $c = 1$. Weighted loss functions with a number of choices for c will be evaluated in Section 4.2.2, as well as training on shorter trajectories.

Similar to the continuous case (Theorem 3.1), the exponential increase in error can be mitigated by training the model in a way that penalises large Lipschitz constants. Again, for neural closure models it is not sufficient to limit the Lipschitz constant of just the neural network. Another mitigation approach is to train on a larger number of time steps, i.e. to increase the value of N_t in the loss function (5). Note that the fact that models trained to predict time series perform poorly when trained on single steps is already known, and methods to mitigate this problem have already been studied. In research such as that done by List et al. [18], unrolling multiple time steps is found to be crucial in obtaining models that make accurate predictions. Similarly, Pan and Duraisamy [25] find that in the discrete case, the long-term accuracy of the model can be improved by adding a regularisation term to the loss function based on the Frobenius norm of the Jacobian of the neural network. Another way to obtain more accurate models is to add noise to the neural network's inputs (the vectors $\mathbf{u}_{\mathrm{ref}}(\cdot)$) during training, which makes the model less sensitive to small perturbations in its input (see for example Chapter 7.4 of Goodfellow et al. [8]). Since the input to the model is equal to the output of the previous step, this regularisation technique is therefore expected to decrease the rate at which the approximation error increases per step, meaning that it has a similar effect to reducing the Lipschitz constant C.

The exponentially increasing sensitivity with respect to the parameters, combined with Theorem 3.2, shows that choosing the number of time steps N_t for trajectory fitting is not trivial as both too small and too large values of N_t may result in poor models.

4. Numerical experiments

In this section, neural networks will be trained in different ways to predict solutions of discretised PDEs of the form

$$\frac{\partial u}{\partial t}(x,t) = F(u)(x,t). \tag{9}$$

This is a scalar PDE on a one-dimensional spatial domain. The boundary conditions are periodic, i.e. $u(x,t) = u(x + L, t)$ for some domain length

L, so that the PDE is translation-invariant. Two PDEs of the form (9) are used: **Burgers' equation** and the **Kuramoto-Sivashinsky equation**. These equations are described in more detail in Appendix A.1 and Appendix A.2, respectively.

In order to generate training data from these equations, they are discretised using the finite volume method with a large number of finite volumes. The resulting ODEs are solved, and the solutions are downsampled by averaging the resulting vectors $\mathbf{u}(t)$ to obtain a coarser discretisation of the original PDE. More details regarding the data generation are given in Appendix A.

4.1. Burgers' equation

The first test equation is Burgers' equation:

$$\frac{\partial u}{\partial t} = -\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) + \nu\frac{\partial^2 u}{\partial x^2},$$

with $\nu = 0.0005$, for $x \in [0,1]$ with periodic boundary conditions. Solutions of this PDE contain waves that travel either left or right through the domain (depending on the sign of u), resulting in shock waves that are then dissipated. Training data is generated by discretising the PDE into 4096 finite volumes, solving the resulting ODE for $t \in [0, 0.5]$ using the standard fourth-order accurate Runge-Kutta method, and downsampling the solutions to $N_x = 64$ finite volumes. The resulting training data consists of $N_p = 96$ trajectories used for training and 32 for testing. Each trajectory consists of an initial condition followed by 64 additional snapshots with a time interval of $\Delta t = 2^{-7}$ between snapshots, meaning the total number of snapshots per trajectory is $N_s = 65$. More information about the data generation is given in Appendix A.1.

Three ML models are trained on data obtained from Burgers' equation. These are closure models of the form $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \mathrm{NN}(\mathbf{u};\vartheta)$. For all three models, the neural network is a Convolutional Neural Network (CNN) with two convolutional layers; more details about the neural network are given in Appendix B.1. The convolutional structure is chosen to ensure that the models satisfy the translational invariance present in Burgers' equation. The models are trained in the three ways outlined in Section 2.

4.1.1. Derivative fitting

One neural closure model is trained using derivative fitting, meaning the loss function is as given in (4). The training data consists of $N_p N_s = 6240$ input-output pairs $\left(\mathbf{u}_{\mathrm{ref}}, \left(\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right)_{\mathrm{ref}}\right)$ on which the model is trained. This model is trained for 10000 epochs with a batch size of 64. No regularisation term is added to the loss function as this not is found to meaningfully improve the resulting model. The choice not to add a regularisation term is motivated in more detail in Section 4.1.4.

4.1.2. Discretise-then-optimise

For the discretise-then-optimise approach, the neural closure model is embedded in `Tsit5`, a fourth-order ODE solver due to Tsitouras [36], with fixed time step. The loss function is given by equation (5) where $N_p = 96$, $N_t = 64$, and $t_i = i\Delta t$ with $\Delta t = 2^{-7}$. In words, the loss function is the mean-square error of the trajectory prediction, averaged over all snapshots of the training data except the initial condition. This model is trained for 20000 epochs, since trajectory fitting is found to converge more slowly than derivative fitting. The training is done with a smaller batch size of 8, since an input-output-pair that can be used for training is now one of the 96 trajectories, instead of one of the 6240 snapshots.

4.1.3. Optimise-then-discretise

A third model is trained using optimise-then-discretise. The ODE solver, loss function, batch size, and number of epochs are the same as those for the discretise-then-optimise model. The main difference is that for the optimise-then-discretise model, the time step of the ODE solver is not fixed, but is determined by the solver's internal error control mechanism. To compute the gradients required for training, the adjoint ODE is solved using the same ODE solver as the forward ODE.

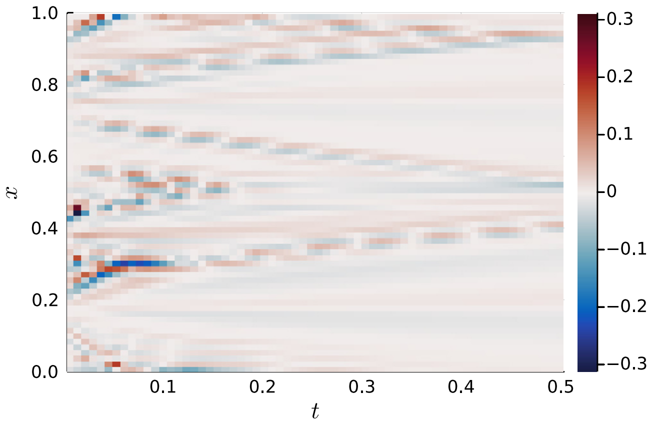
Table 2
The RMSE for each of the tested models on the 32 testing trajectories of the Burgers' equation.

Training approach	Error on test data
Coarse ODE without closure term	0.104
Derivative fitting	2.67
Discretise-then-optimise	0.0264
Optimise-then-discretise	0.0312

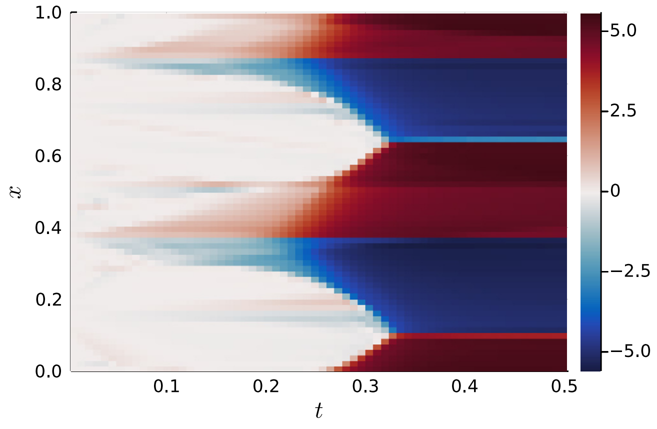
4.1.4. Results

After training, all models are evaluated on test data that is not used during training. For each of the evaluated models, 32 initial conditions $\mathbf{u}^{(j)}_{\mathrm{ref}}(0)$ for $j = 1, \dots, 32$ are given to the ML model. The model then uses these initial conditions to make predictions for the trajectories $\mathbf{u}^{(j)}_{\mathrm{predict}}(t_i)$ for $j = 1, \dots, 32$ and $i = 1, \dots, N_t$. These are then compared to the actual trajectories $\mathbf{u}^{(j)}_{\mathrm{ref}}(t_i)$ by taking the root-mean-square error (RMSE), summing over all components of the vector $\mathbf{u}$, all time points t_i, and all 32 testing trajectories. Therefore, the RMSE is simply the square root of the expression in (5), except over a different set of trajectories. In addition to the neural closure models, the RMSE is also computed when solving the coarse discretisation of Burgers' equation without a closure term, i.e. with $\mathrm{NN}(\mathbf{u};\vartheta) \equiv 0$. The RMSEs on the test data for all models are shown in Table 2. While the two models trained on trajectory fitting achieve significantly lower error than the coarse ODE solved without a closure term, the model trained using derivative fitting produces highly inaccurate results. As is visible in Fig. 2b, the model trained on derivative fitting produces a trajectory prediction that has completely different behaviour to the training data. The error in the prediction grows steadily as t increases, and eventually reaches a steady-state with high error. This indicates that the derivative-fitting trained model creates a prediction that converges to a significantly different steady state to the steady state of the reference solution. To investigate this result further, more models are trained using derivative fitting with L2-regularisation, and indeed adding a regularisation term to the loss function does improve the accuracy of the resulting models. Nevertheless, these results are not included here. The reason for this is that adding a regularisation term only helps by reducing the magnitude of the neural network parameters, thereby also reducing the magnitude of the neural network output. The result is that training with a strong regularisation brings the accuracy of the neural closure model closer to that of the coarse ODE, without surpassing the coarse ODE in accuracy. Therefore, even with a regularisation term, derivative fitting is not effective at producing accurate neural closure models. Note that the prediction made by the model trained by discretise-then-optimise, which is shown in Fig. 2a, has lower error for large t than for small t. The reason for this is that solutions to Burgers' equation show relatively complex transient behaviour before converging to a simple steady state, making the long-term behaviour easier to predict than the short-term behaviour.

Fig. 3 shows how the RMSE on the training data of the two trajectory fitting models improves during training. From this figure, it is clear that while both models start out with approximately the same RMSE (slightly above 0.1), the model trained with discretise-then-optimise improves slightly faster and starts converging to a measurably lower error than the other model after approximately 8000 epochs. The lower training error is not due to overfitting, as the model trained with discretise-then-optimise also produces measurably more accurate results on the test data as shown in Table 2. Instead, the reason for the higher training error in optimise-then-discretise is due to the inaccuracy of the gradient of the loss function: while the exact minimiser of the optimisation problem, i.e. the optimal parameters ϑ, satisfy $\frac{\mathrm{dLoss}}{\mathrm{d}\vartheta} = 0$, the gradient of the loss function is not computed exactly when using



(a) Prediction error by the model trained by discretise-then-optimise.



(b) Prediction error by the model trained by derivative fitting.

Fig. 2. The errors in the trajectory predictions made by the models trained by discretise-then-optimise and derivative fitting, respectively.

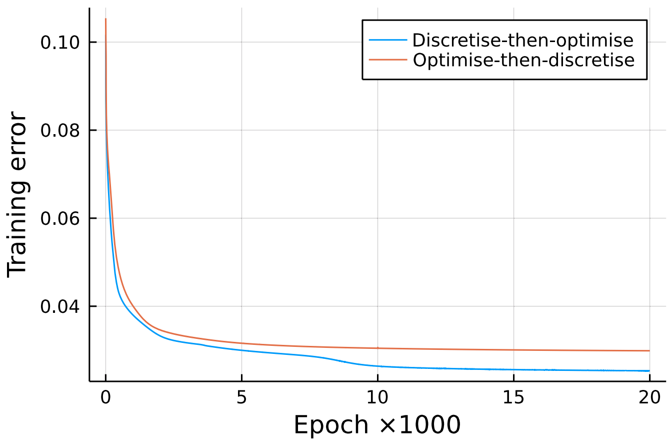


Fig. 3. The RMSE history during training of the two models that are trained using trajectory fitting on the Burgers' test case.

the adjoint ODE formulation. As a result, training by optimise-then-discretise does not converge to the truly optimal parameters or even to a local optimiser, but to a different parameter vector that satisfies $\frac{\mathrm{dLoss}}{\mathrm{d}\vartheta} + (\text{adjoint ODE error}) = 0$.

4.2. The Kuramoto-Sivashinsky equation

Since Burgers' equation is relatively predictable (resulting in shock waves which then dissipate over time while moving through the domain), we also consider a more challenging case. In this section, experiments are performed using the Kuramoto-Sivashinsky equation:

$$\frac{\partial u}{\partial t} = -\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4},$$

for $x \in [0, 64]$ with periodic boundary conditions. For a short description of the Kuramoto-Sivashinsky equation and its terms and solution behaviour, see Appendix A.2. Training data for the Kuramoto-Sivashinsky equation was generated by discretising the PDE into 1024 finite volumes, solving the resulting ODE using a third-order stiff ODE solver for $t \in [0, 256]$, and down-sampling the resulting solution to $N_x = 128$ finite volumes. The resulting training data consists of $N_p = 80$ trajectories used for training and 10 for testing. Each trajectory consists of an initial state followed by $N_t = 512$ additional snapshots with $\Delta t = \frac{1}{2}$ between snapshots. All models trained in this section use the 'large' neural network shown in Table B.5, which is a CNN with six convolutional layers.

The Kuramoto-Sivashinsky equation is chaotic, meaning that arbitrarily small differences in the initial state $\mathbf{u}(0)$ will eventually lead to a completely different solution $\mathbf{u}(t)$ for large t. As a result, all methods of approximating this PDE are expected to diverge from the training data at some point, meaning that simply taking the RMSE between the predicted and actual trajectories is not a very useful metric for accuracy. Instead, the accuracy of a method will be evaluated using the valid prediction time (VPT), which is the time until the error between the approximation and the training data exceeds some pre-defined threshold. Here, the VPT of a prediction $\mathbf{u}(t)$ to a real trajectory $\mathbf{u}_{\mathrm{ref}}(t)$ is computed following the procedure used by Pathak et al. [27]. First, the average energy of the real trajectory is computed as

$$E_{\mathrm{avg}} = \sqrt{\frac{1}{N_t}\sum_{i=1}^{N_t} \left\|\mathbf{u}_{\mathrm{ref}}(t_i)\right\|^2}.$$

Then, the valid prediction time is given by

$$\mathrm{VPT}(\mathbf{u}_{\mathrm{ref}}, \mathbf{u}, t_{1,2,\ldots,N_t}) = \min\left\{t_i \;\middle|\; \left\|\mathbf{u}(t_i) - \mathbf{u}_{\mathrm{ref}}(t_i)\right\| \geq 0.4 E_{\mathrm{avg}}\right\}. \tag{10}$$

One property shared among chaotic processes is that the difference between two solutions $\mathbf{u}_{\mathrm{ref}}, \mathbf{u}$ grows approximately exponentially in time:

$$\left\|\mathbf{u}_{\mathrm{ref}}(t) - \mathbf{u}(t)\right\| \approx e^{\lambda_{\max} t}\left\|\mathbf{u}_{\mathrm{ref}}(0) - \mathbf{u}(0)\right\|. \tag{11}$$

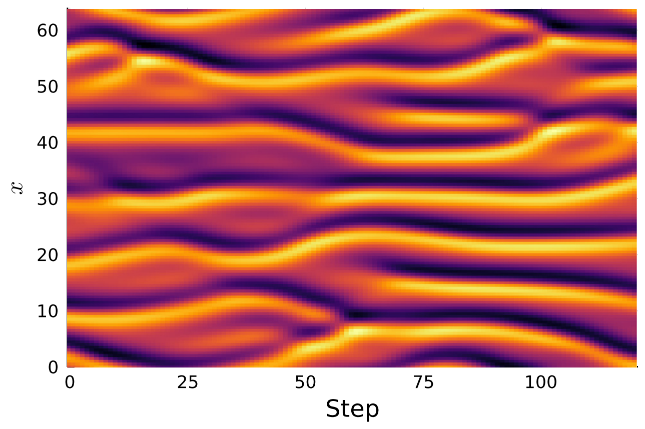
In this equation, $\lambda_{\max}$ is the Lyapunov exponent of the ODE, which determines the growth rate of the error between two solutions. The inverse of the Lyapunov exponent is the Lyapunov time $T_{\mathrm{Lyap}} = \lambda_{\max}^{-1}$, which can be interpreted as the time it takes for the difference between two similar trajectories to increase by a factor e. Since the Lyapunov time represents the time scale on which trajectories diverge, the valid prediction times of models will be expressed as multiples of T_{Lyap}.

4.2.1. Derivative fitting

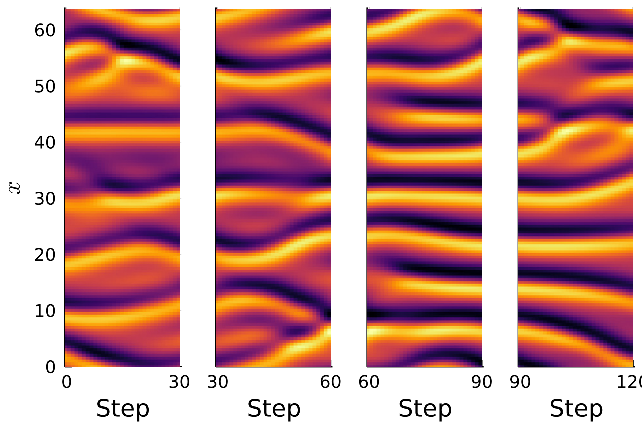
As is the case for Burgers' equation, the simplest training approach available is to train using derivative fitting. Since adding a penalty term to the neural network is not found to help significantly, this penalty term is omitted for derivative fitting for the Kuramoto-Sivashinsky equation, meaning only one model is trained with derivative fitting. This model is trained on all snapshots of the 80 trajectories of the training data. The model is trained for 1000 epochs with the batch size set to 128.

4.2.2. Optimise-then-discretise

The optimise-then-discretise training approach is tested on the same neural network. With this approach, the parameter N_t, the number of snapshot predictions computed from the initial condition, must be chosen. For the models trained on Burgers' equation, N_t was equal to the number of snapshots in the training data, but this is not expected



(a) A 'long' trajectory consisting of the initial state and 120 additional snapshots.



(b) Four 'short' trajectories, each consisting of an initial condition and 30 additional snapshots.

Fig. 4. A visual representation of how training data from the Kuramoto-Sivashinsky equation is split into smaller trajectories. The last snapshot of one trajectory is equal to the initial condition of the next trajectory.

to yield accurate models for the chaotic Kuramoto-Sivashinsky equation as described in Section 3.2. Here, two models are trained with the optimise-then-discretise approach with two different choices for N_t. The first model is trained on the first 25 snapshots from each trajectory (i.e. an initial condition and $N_t = 24$ additional snapshots, corresponding to one Lyapunov time), and the other is trained on the first 145 snapshots from each trajectory (i.e. with $N_t = 144$, corresponding to six Lyapunov times). This is done so that the effect of the trajectory length N_t on the model accuracy can also be studied. For both models, the forward and adjoint ODEs are both solved using `KenCarp47`, a 4th-order accurate additive Runge-Kutta method due to Kennedy and Carpenter [14] that is implicit in f but explicit in the neural network term. Using this ODE solver was found to be computationally more efficient than using either a fully explicit or fully implicit Runge-Kutta method (see sections 5.5.3 and 5.6.1 of [22]).

In order to avoid very large gradients causing the training to fail, gradient clipping is applied to the gradients before applying the optimiser. This way, gradients of which the norm exceeds some constant r are scaled such that their norms are exactly equal to r, thereby avoiding very large gradients while leaving small gradients unchanged. Here, gradient clipping is used with $r = 10^{-2}$. The model trained on short trajectories is trained for 1000 epochs with a batch size of 10. The model trained on long trajectories is only trained for 100 epochs since the longer trajectories make the training much slower. While this means that the second model is not trained until convergence, the effect on the accuracy of the resulting models is found to be small compared to the difference in accuracy reported in Section 4.2.4. Both models are trained with a batch size of 8.

As described in Section 3.2, trajectory fitting on chaotic systems can cause problems due to exploding gradients, especially when training on long trajectories. To see if weighing the loss function as in (8a) mitigates this problem, four more models are trained using optimise-then-discretise on long trajectories, using the weighted MSE as loss function with $c \in \{0.5, 1.0, 1.5, 2.0\}$. The number of epochs, batch size, and other parameters for these models is the same as those for the model trained on long trajectories with a simple MSE loss function.

4.2.3. Discretise-then-optimise

To test the discretise-then-optimise method for the Kuramoto-Sivashinsky equation, the PDE is solved in the pseudospectral domain using an exponential integrator. More information about this approach is given in Appendix B.2.

In order to be able to compare more directly with the previous models, the closure term is given by the same neural network as used in earlier experiments, meaning that its input and output are in the physical domain. As such, the neural network term is preceded by an inverse Fourier transform and followed by a Fourier transform:

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{\mathbf{u}} = \left(\boldsymbol{\Lambda}^2 - \boldsymbol{\Lambda}^4\right)\hat{\mathbf{u}} - \frac{i}{2}\boldsymbol{\Lambda}\mathcal{F}\left(\left(\mathcal{F}^{-1}\hat{\mathbf{u}}\right)^2\right) + \mathcal{F}\left(\mathrm{NN}\left(\mathcal{F}^{-1}\hat{\mathbf{u}}; \vartheta\right)\right). \tag{12}$$

As is the case for the optimise-then-discretise tests, one must choose N_t, the number of time steps that are predicted with the model. In Section 4.1, the number of time steps to predict was 64, equal to the number of snapshots in the training data, with good results. In their experiments with neural closure models for two-dimensional incompressible fluid flow problems, List et al. [18] refer to this as the number of unrolled steps, and find that this has a significant effect on the stability of the resulting model. As such, a number of different models are trained by unrolling with different numbers of time steps. Specifically, 11 different models are trained with $N_t \in \{1, 2, 4, 8, 15, 30, 60, 90, 100, 110, 120\}$. The model with $N_t = 120$ is trained on the first 121 snapshots of each of the 80 training trajectories. For the models with $N_t \in \{90, 100, 110\}$, the same 80 trajectories are truncated to the desired number of steps. For the remaining models, the training trajectories are split into multiple shorter trajectories as shown in Fig. 4. In this way, each training trajectory of 121 snapshots (an initial condition followed by 120 additional time steps) can be split into 4 trajectories of 31 snapshots each, or 15 trajectories of 9 snapshots each, and so on.

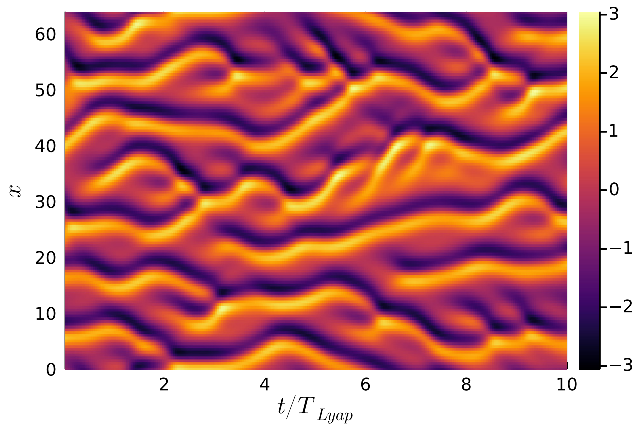
This way, all discretise-then-optimise models are trained on the first 121 snapshots of each trajectory in the training data (or slightly fewer). All these models are trained for 5000 epochs with a batch size of 8.

4.2.4. Results

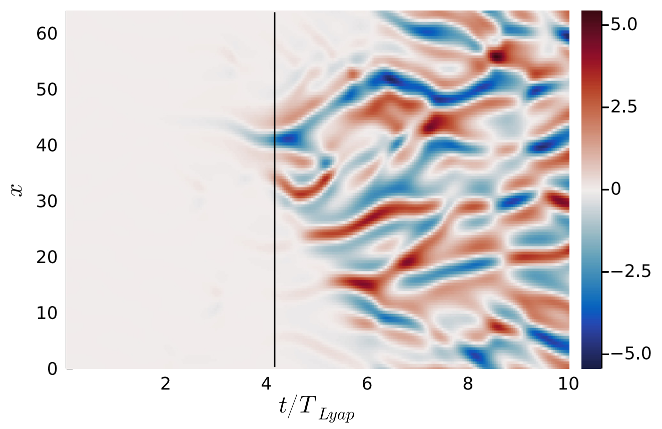
Table 3 shows the minimum, average, and maximum VPTs of neural ODEs trained with derivative fitting and optimise-then-discretise, and some example trajectories and predictions are shown in Fig. 5. Note that the VPTs in Figs. 5b and 5c only show one of the ten test trajectories, meaning that the VPTs in these figures do not correspond to the averages shown in Table 3. From this table, it can be seen that optimise-then-discretise fitting works best on short trajectories, where the resulting models slightly outperform models trained by derivative fitting. Training on long trajectories results in worse performance, even if the loss function is exponentially weighted as in (8a) to mitigate the exploding gradients problem. This is likely due to the fact that the adjoint ODE methods introduce an error in the gradients used during training. As is generally the case for ODE solutions, this error increases with the time interval over which the ODE is solved. Hence, training on longer trajectories introduces a larger error in the gradients, which reduces the accuracy of the resulting model even if the

Table 3
An overview of the performance of different models tested on the Kuramoto-Sivashinsky equation, sorted by training approach. Fig. 7 summarises all results for the Kuramoto-Sivashinsky equation.

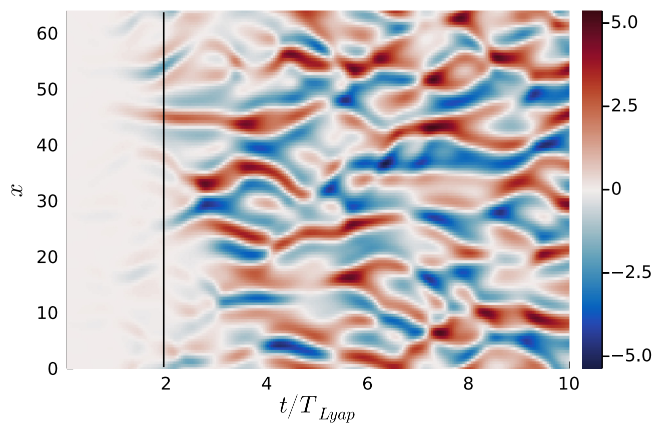
Training approach		VPT on test data		
		Min	Avg	Max
Coarse ODE		1.17	1.93	3.00
Derivative fitting		4.17	5.36	7.54
Optimise-then-discretise	Short trajectories ($N_t = 24$)	4.08	5.84	8.29
	Long trajectories ($N_t = 144$)	2.38	3.38	4.67
Optimise-then-discretise, $N_t = 144$ decaying error weights	$c = 0.5$	2.42	4.20	5.38
	$c = 1.0$	2.96	4.38	6.29
	$c = 1.5$	3.29	4.58	5.88
	$c = 2.0$	2.71	4.29	5.75
Discretise-then-optimise	Short trajectories ($N_t = 30$)	4.92	7.10	9.12
	Long trajectories ($N_t = 120$)	4.12	5.33	7.38



(a) A trajectory of the K-S equation from the test data.



(b) The prediction error for this trajectory of the closure model trained on short trajectories.



(c) The prediction error for this trajectory of the closure model trained on long trajectories.

Fig. 5. An example trajectory from the test data, and the prediction error of two different models obtained for this trajectory. Notice how the model trained on long trajectories produces significantly greater error in the short-term, and also achieves a lower VPT for this trajectory as a result (indicated by the vertical black lines in Panels b and c).

error function is weighted to mitigate the exploding gradients problem.

Fig. 6 shows the minimum, average, and maximum VPTs of the 11 models trained using discretise-then-optimise on different numbers of unrolling steps. It can be seen that models trained on 60 or more steps perform worse than models trained on fewer steps, although performance continues to improve after 1000 epochs. After 5000 epochs, the model trained on 30 steps performs the best. The model trained on 120 steps is found to perform very badly after 1000 epochs, although performance is similar to that of other models after 5000 epochs. The reason for the poor performance after 1000 epochs is described in Section 3.2: the long time interval used for training, combined with the chaotic nature of the Kuramoto-Sivashinsky equation, means that the loss function is most sensitive to the behaviour for large t. This initially prevents the model from becoming more accurate in the short term, resulting in a very short VPT.

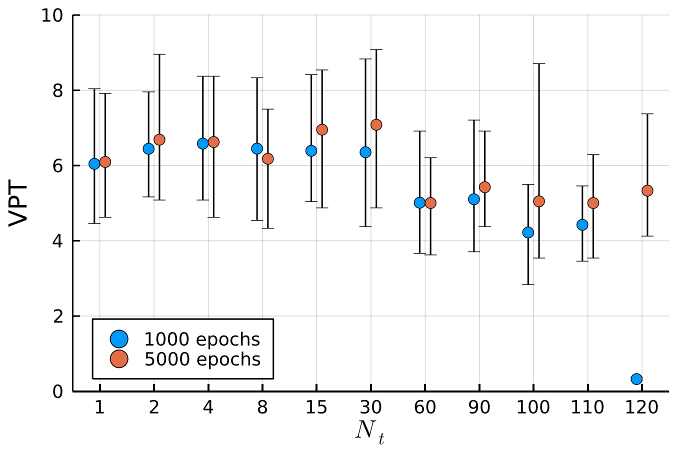


Fig. 6. The minimum, average, and maximum VPT across the 10 test trajectories for models trained for the given number of time steps N_t on the Kuramoto-Sivashinsky equation with discretise-then-optimise.

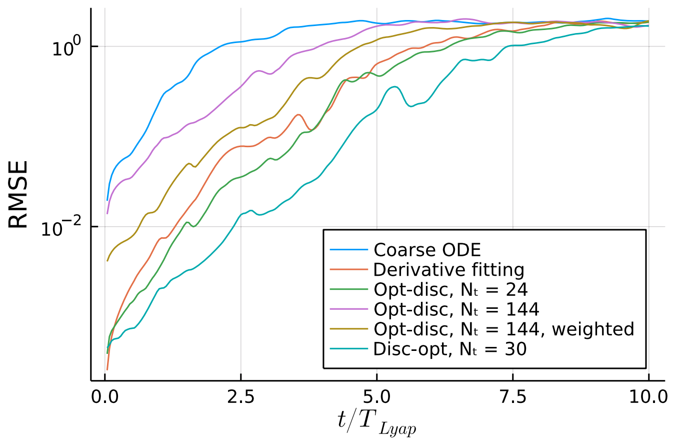


Fig. 7. A plot of the RMSE on the Kuramoto-Sivashinsky equation of some of the different models in Table 3 as a function of t.

5. Conclusions

The goal of this paper is to make a general comparison of approaches for neural closure models, and make recommendations based on the results of this comparison. To this end, a variety of different training approaches were evaluated on two different test cases derived from partial differential equations. Furthermore, two theorems were given that provide upper bounds on the long-term error of a neural closure model based on the short-term error that is used as the objective function during training.

The simplest approach, derivative fitting, is found to perform approximately as well as trajectory fitting on the Kuramoto-Sivashinsky equation, but performs significantly worse on Burgers' equation. Theorem 3.1 implies that in general, models that are trained with derivative fitting may indeed produce inaccurate trajectories, even if they achieve high accuracy during training. It is not entirely clear what causes derivative fitting to perform well on the Kuramoto-Sivashinsky equation and not on Burgers' equation. The performance may depend on the ODE for which the neural network acts as a closure term, or on the time integration method used to solve the ODE. However, the performance of derivative fitting may also be sensitive to specific parameters of the training procedure. For example, the use of derivative fitting for LES closure models produces accurate models in the work of Guan et al. [9], but not in the work of MacArt et al. [20]. Since derivative fitting is computationally much cheaper than trajectory fitting, it is generally worthwhile to try training a neural closure model with derivative fitting first. If derivative fitting produces poor models as it does in the Burgers' equation tests, one can still switch to trajectory fitting. Furthermore, if derivative fitting gives stable but inaccurate models, another approach is to train with derivative fitting for a relatively small number of epochs, and to continue training the resulting model with trajectory fitting (see Section 5.1.2 of Melchers [22] for details). This approach produces accurate models like trajectory fitting, and is computationally more efficient by reducing the number of iterations of the slower trajectory fitting that have to be performed in order to train the model to convergence.

Regarding trajectory fitting, the preferable approach is to use the discretise-then-optimise strategy, rather than optimise-then-discretise, which is in agreement with the results obtained by Onken and Ruthotto [24]. The reason for this is likely that the discretise-then-optimise approach computes gradients more accurately, allowing the model to be trained to a higher accuracy. This results in slightly faster convergence during training as well as in a smaller error overall. This also means that a systematic comparison between optimise-then-discretise and discretise-then-optimise is not always straightforward; in the Kuramoto-Sivashinsky case the pseudospectral method used for the discretise-then-optimise tests is a more accurate spatial discretisation of the PDE than the finite volume method used in the derivative fitting and optimise-then-discretise tests. This is a fundamental issue in testing training approaches for neural closure models, as many test problems require specific ODE solvers to efficiently obtain accurate solutions.

When training by trajectory fitting, the length of trajectories used for training must be chosen carefully, both for optimise-then-discretise and discretise-then-optimise. This is in line with our Theorem 3.2, stating that models that produce accurate predictions in the short term may still produce inaccurate predictions in the long-term. However, this is not always the case as Theorem 3.2 only provides an upper bound for the error. On the Kuramoto-Sivashinsky data, for example, the models trained by derivative fitting and by discretise-then-optimise with unrolling a single time step both produce good long-term predictions.

For optimise-then-discretise, training on long trajectories results in less accurate models due to the exploding gradients problem. This problem can be mitigated by introducing a weighted error function, although the resulting approach still produces less accurate models than discretise-then-optimise trajectory fitting due to the increased error in the gradient computation. For the Kuramoto-Sivashinsky test case, derivative fitting produces better results than optimise-then-discretise on long trajectories. For discretise-then-optimise, fairly accurate models can be obtained by training on long trajectories, although training this way converges much more slowly than when training on shorter trajectories.

As machine learning techniques become more popular as a way to learn models from data, the existence of general recommendations and rules of thumb becomes increasingly important to reduce the amount of trial and error required to obtain accurate models. Overall, the work presented here provides a set of recommendations for neural closure models. Notwithstanding, we point to a few remaining open questions. Most notably, it is not fully clear yet what properties of an ODE system determine whether or not derivative fitting produces accurate models. Furthermore, for trajectory fitting it was argued that the number N_t of time steps computed during training should be chosen carefully, but in this work good values for N_t were found by trial and error. It is not clear how a good value for N_t can be chosen ahead of time.

Funding statement

This publication is part of the project "Discretize first, reduce next" (with project number VI.Vidi.193.105) of the research programme Vidi which is (partly) financed by the Dutch Research Council (NWO).

CRediT authorship contribution statement

Hugo Melchers: Conceptualization, Formal analysis, Investigation, Methodology, Software, Visualization, Writing – original draft. **Daan Crommelin:** Conceptualization, Project administration, Supervision, Writing – review & editing. **Barry Koren:** Conceptualization, Project administration, Supervision, Writing – review & editing. **Vlado Menkovski:** Conceptualization, Project administration, Supervision, Writing – review & editing. **Benjamin Sanderse:** Conceptualization, Funding acquisition, Project administration, Supervision, Writing – review & editing.

Data availability

The code and data associated with this work is available online at: https://github.com/HugoMelchers/neural-closure-models.

Acknowledgements

Large parts of this work were completed as part of a master thesis on neural closure models at the Department of Mathematics and Computer Science at Eindhoven University of Technology (TU/e) and at the Centrum Wiskunde & Informatica (CWI) in Amsterdam.

Appendix A. Data generation

Solutions to the Burgers and Kuramoto-Sivashinsky equations are computed using the finite volume method. For both equations, the initial states $\mathbf{u}(0)$ are randomly generated as the sum of random sine and cosine waves with wave numbers $1 \leq k \leq 10$:

$$\mathbf{u}_i(0) = \mathrm{Re}\left(\sum_{k=1}^{10} \hat{u}_k \exp(2\pi i k/N_x) + \sum_{k=1}^{10} \hat{u}_{-k} \exp(-2\pi i k/N_x)\right), \tag{A.1}$$

where N_x is the number of finite volumes and the $\hat{u}_k, k = \pm 1, \ldots, \pm 10$ are independent and distributed according to a unit Gaussian distribution. The resulting initial conditions are then multiplied by a constant so that $\max_i |\mathbf{u}_i(0)| = 2$.

A.1. Burgers' equation

Burgers' equation is a PDE over one variable, the momentum $u(x,t)$, as a function of space and time. Its general form is as follows:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} - \frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right), \tag{A.2}$$

where $\nu \geq 0$ is a constant. The PDE can be seen as a highly simplified one-dimensional version of a fluid flow problem, with a linear second-order diffusion term that models the effects of fluid viscosity, and a quadratic convection term that resembles the convection term in the Navier-Stokes equation. In this section, Burgers' equation will be used with periodic boundary conditions and a domain length of 1, i.e. $u(x+1,t) = u(x,t)$ for all x, t.

The Burgers equation (A.2) is solved with $\nu = 0.0005$. The spatial computational domain is discretised using the first-order accurate spatial discretisation given by Jameson [12]:

$$\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t} = f(\mathbf{u})_i = \frac{\nu}{\Delta x^2}\left(\mathbf{u}_{i-1} - 2\mathbf{u}_i + \mathbf{u}_{i+1}\right) - \frac{1}{\Delta x}\left(\mathbf{f}_{i+1/2} - \mathbf{f}_{i-1/2}\right), \tag{A.3a}$$

$$\text{where } \mathbf{f}_{i+1/2} = \frac{1}{6}\left(\mathbf{u}_i^2 + \mathbf{u}_i\mathbf{u}_{i+1} + \mathbf{u}_{i+1}^2\right) - \alpha_{i+1/2}\left(\mathbf{u}_{i+1} - \mathbf{u}_i\right), \tag{A.3b}$$

$$\text{and } \alpha_{i+1/2} = \frac{1}{4}\left|\mathbf{u}_i + \mathbf{u}_{i+1}\right| - \frac{1}{12}\left(\mathbf{u}_{i+1} - \mathbf{u}_i\right). \tag{A.3c}$$

The resulting ODEs are solved for $t \in [0, 0.5]$ using the `Tsit5` algorithm [36], which is a fourth-order, five-stage Runge-Kutta method with embedded error estimator. This ODE solver was chosen following the recommendations of the DifferentialEquations.jl documentation [23]. In total, 128 solutions are obtained, each from a random initial state according to (A.1). Of these solutions, 96 are used for training and the remaining 32 are used for testing. The solutions are then down-sampled by a factor of 64 in space. This way, training data is created to allow a neural network to work on the low-fidelity (downsampled) initial conditions, but to still approximate the original high-fidelity solution. The downsampling is performed by averaging the solution over chunks of 64 finite volumes. This is a necessary step, since training data obtained through solving an ODE $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u})$ would trivially allow a neural closure model $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = f(\mathbf{u}) + \mathrm{NN}(\mathbf{u};\vartheta)$ to produce very accurate predictions when $\mathrm{NN}(\mathbf{u};\vartheta) \approx 0$. The coarse-grid solution is saved with a time step of $\Delta t = 2^{-7}$ between snapshots, meaning that each coarse-grid solution consists of an initial condition followed by 64 additional snapshots.

Note that when numerically solving PDEs, choosing a first-order accurate spatial discretisation and a fourth-order accurate ODE solver would typically be a bad choice. The error in the resulting solution would then be dominated by the error in the spatial discretisation, meaning that more accurate solutions could be obtained by using a finer spatial discretisation and a lower-order ODE solver. In the experiments performed in this work the goal is to train neural networks to compensate for the spatial discretisation error. This means that choosing a relatively high-order accurate ODE solver is necessary to ensure that the temporal discretisation error does not contribute significantly to the overall error.

This process is illustrated in Fig. A.8. This figure also shows the result of solving the coarsely discretised ODE starting from the downsampled initial state $\mathbf{u}(0) \in \mathbb{R}^{64}$ (Fig. A.8c). Crucially, this is not equal to the downsampled fine-grid solution (Fig. A.8b). The downsampled fine-grid solution is by definition the most accurate low-fidelity approximation to the original high-fidelity data. The solution of the low-fidelity ODE introduces additional errors (Fig. A.8d), and is therefore a less accurate approximation to the original data than the downsampled high-fidelity solution.

A.2. Kuramoto-Sivashinsky

The Kuramoto-Sivashinsky equation is named after the two researchers who independently derived the equation in Kuramoto [17] and Sivashinsky [34]. This PDE is taken with periodic boundary conditions as well:

$$\frac{\partial u}{\partial t} = -\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right) - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4}, \tag{A.4a}$$

$$u(x+L,t) = u(x,t). \tag{A.4b}$$

The Lyapunov exponent of (A.4a) depends on the length L of the domain. Edson et al. [6] found the following approximation for the Lyapunov eigenvalues for varying L:

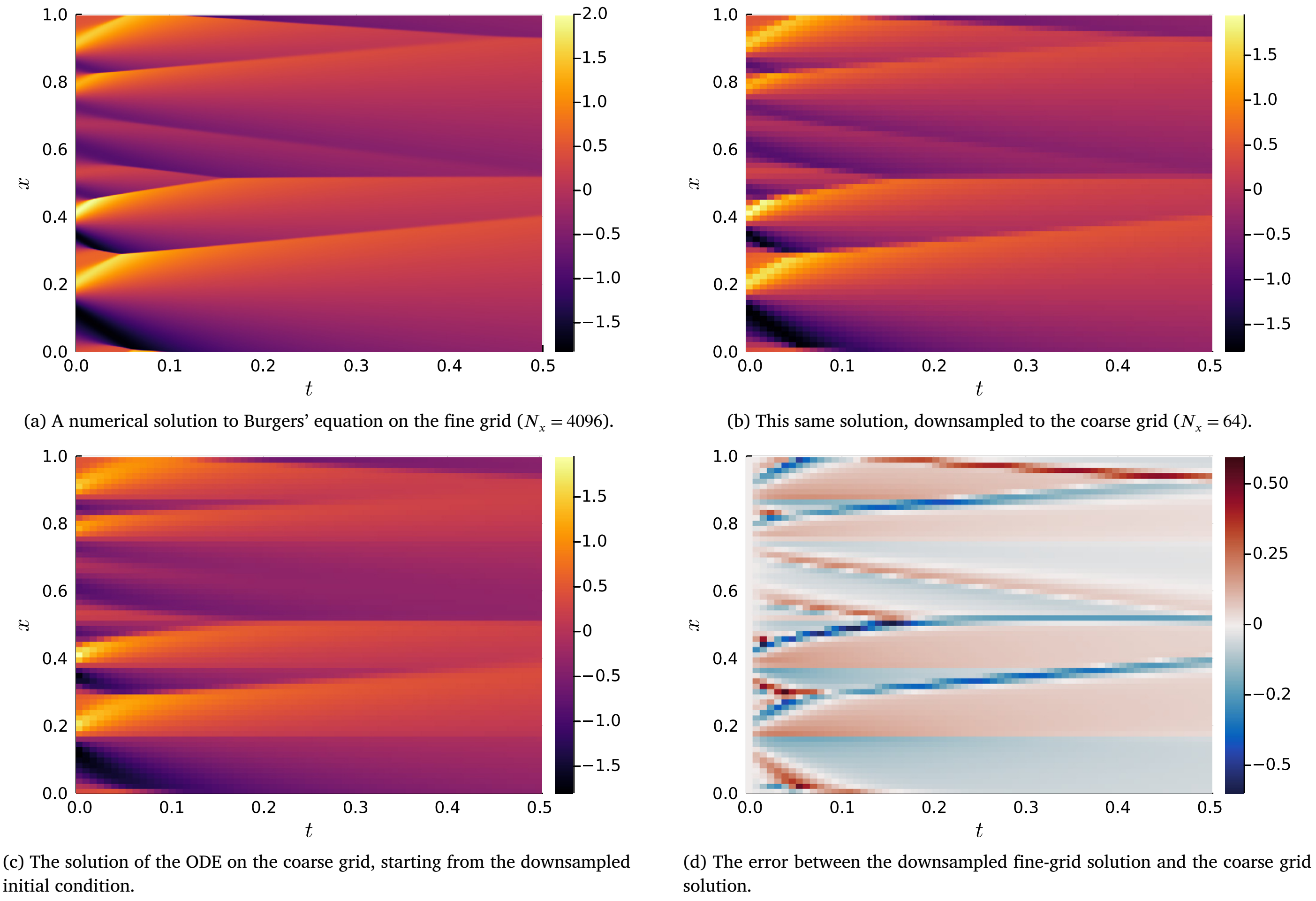
$$\lambda_i(L) \approx 0.093 - \frac{0.94}{L}(i - 0.39), \; i = 1, 2, \ldots$$

$$\implies \lambda_{\max}(L) \approx 0.093 - \frac{0.57}{L}.$$

The training data is created with $L = 64$, leading to a Lyapunov exponent of $\lambda_{\max} \approx 0.084$. The inverse of the Lyapunov exponent is the Lyapunov time T_{Lyap}, which can be loosely interpreted as the time it takes for the error between two similar trajectories to grow by a factor e. Taking $L = 64$ yields $T_{\mathrm{Lyap}} \approx 12$.

In this PDE, the time-dependent behaviour of u is governed by three terms:

- A non-linear convection term $-\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right)$, the same as in Burgers' equation.
- A destabilising anti-diffusion term $-\frac{\partial^2 u}{\partial x^2}$. Note that this term appears on the right-hand with a minus sign, which is unusual for diffusion terms.



(a) A numerical solution to Burgers' equation on the fine grid ($N_x = 4096$).

(b) This same solution, downsampled to the coarse grid ($N_x = 64$).

(c) The solution of the ODE on the coarse grid, starting from the downsampled initial condition.

(d) The error between the downsampled fine-grid solution and the coarse grid solution.

Fig. A.8. An example showing how solving a coarse-grid solution to Burgers' equation does not yield the same solution as down-sampling the fine-grid solution to the equation. A closure model can be added to the coarse discretisation to bring its solution (Panel c) closer to the fine-grid solution (Panel a).

- A stabilising hyper-diffusion term $-\frac{\partial^4 u}{\partial x^4}$. Without this term, the equation would be ill-posed since the anti-diffusion term would cause solutions to blow up in a finite amount of time.

Solutions to the Kuramoto-Sivashinsky equation are created by solving the PDE with domain length $L = 64$. As is the case for Burgers' equation, for the Kuramoto-Sivashinsky equation the function $u(x,t)$ is discretised as a time-dependent vector $\mathbf{u}(t)$. Since the non-linear convection term is the same as in Burgers' equation, it is discretised in the same way (see (A.3a)) with the exception that the artificial diffusion term (given by the vector $\boldsymbol{\alpha}$ in (A.3b)) is no longer needed since the Kuramoto-Sivashinsky equation produces smooth solutions. The linear diffusion and hyper-diffusion terms are discretised using simple 3-wide and 5-wide stencils, respectively, leading to the following discretisation for the three different terms:

$$\left(\frac{1}{2}\frac{\partial}{\partial x}\left(u^2\right)\right)(\mathbf{x}_i) \to \frac{1}{\Delta x}\left(\mathbf{f}_{i+1/2} - \mathbf{f}_{i-1/2}\right),$$

$$\text{where } \mathbf{f}_{i+1/2} = \frac{1}{6}\left(\mathbf{u}_i^2 + \mathbf{u}_i\mathbf{u}_{i+1} + \mathbf{u}_{i+1}^2\right),$$

$$\frac{\partial^2 u}{\partial x^2}(\mathbf{x}_i) \to \frac{1}{\Delta x^2}\left(\mathbf{u}_{i-1} - 2\mathbf{u}_i + \mathbf{u}_{i+1}\right),$$

$$\frac{\partial^4 u}{\partial x^4}(\mathbf{x}_i) \to \frac{1}{\Delta x^4}\left(\mathbf{u}_{i-2} - 4\mathbf{u}_{i-1} + 6\mathbf{u}_i - 4\mathbf{u}_{i+1} + \mathbf{u}_{i+2}\right).$$

Since the PDE is chosen with periodic boundary conditions, the stencils shown here are also applied with periodic boundary conditions, i.e. $\mathbf{u}_{i+N_x} = \mathbf{u}_i$. Written out fully, the resulting ODE is:

$$\frac{\mathrm{d}\mathbf{u}_i}{\mathrm{d}t} = f(\mathbf{u})_i = -\frac{1}{6\Delta x}\left(\mathbf{u}_{i+1}^2 - \mathbf{u}_{i-1}^2 + \mathbf{u}_i\left(\mathbf{u}_{i+1} - \mathbf{u}_{i-1}\right)\right) - \frac{1}{\Delta x^2}\left(\mathbf{u}_{i-1} - 2\mathbf{u}_i + \mathbf{u}_{i+1}\right) - \frac{1}{\Delta x^4}\left(\mathbf{u}_{i-2} - 4\mathbf{u}_{i-1} + 6\mathbf{u}_i - 4\mathbf{u}_{i+1} + \mathbf{u}_{i+2}\right). \tag{A.5}$$

The PDE is discretised with $N_x = 1024$ finite volumes, and solved from $t = 0$ to $t = 256$. The initial conditions are generated in the same way as for Burgers' equation, see (A.1). The ODEs are solved using the 3rd-order accurate stiff ODE solver `Rodas4P` [35], again following the recommendations from the DifferentialEquations.jl documentation [23]. The resulting solutions are downsampled to 128 finite volumes in space with a time step of $\Delta t = \frac{1}{2}$ in between snapshots. To avoid the effects of initial transients, the first 32 snapshots of the trajectories are not used for training. An example trajectory from the resulting training data is shown in Fig. A.9.

Of the 100 generated trajectories, the first 80 are used for training and the last 10 are used for testing. The remaining 10 trajectories are used to evaluate models while they are being trained, to ensure models are trained for enough iterations but without overfitting.

Appendix B. Training details

B.1. Neural network architectures

To ensure that the experiments only test the effect of including prior knowledge, only two different neural networks are used for all numerical experiments: a 'small' convolutional neural network with 57 parameters, and a 'large' convolutional neural network with 533 parameters.

Table B.4
A description of the small neural network structure used for the experiments on Burgers' equation.

Layer	Description	σ	Parameters
1	$\mathbf{u} \mapsto \left[(\mathbf{u}_i)_i \quad (\mathbf{u}_i^2)_i \right]$	–	0
2	9-wide conv, $2 \to 2$ channels	tanh	38 ($9 \times 2 \times 2$ weights and 2 biases)
3	9-wide conv, $2 \to 1$ channels	–	19 ($9 \times 2 \times 1$ weights and 1 bias)
8	$\mathbf{u} \mapsto \Delta_{\text{fwd}} \mathbf{u}$	–	0
Total			57

Table B.5
A description of the large neural network structure used for the experiments on the Kuramoto-Sivashinsky equation.

Layer	Description	σ	Parameters
1	$\mathbf{u} \mapsto \left[(\mathbf{u}_i)_i \quad (\mathbf{u}_i^2)_i \right]$	–	0
2	5-wide conv, $2 \to 4$ channels	tanh	44 ($5 \times 2 \times 4$ weights and 4 biases)
3	5-wide conv, $4 \to 6$ channels	tanh	126 ($5 \times 4 \times 6$ weights and 6 biases)
4	5-wide conv, $6 \to 6$ channels	tanh	186 ($5 \times 6 \times 6$ weights and 6 biases)
5	5-wide conv, $6 \to 4$ channels	tanh	124 ($5 \times 6 \times 4$ weights and 4 biases)
6	5-wide conv, $4 \to 2$ channels	tanh	42 ($5 \times 4 \times 2$ weights and 2 biases)
7	5-wide conv, $2 \to 1$ channels	–	11 ($5 \times 2 \times 1$ weights and 1 bias)
8	$\mathbf{u} \mapsto \Delta_{\text{fwd}} \mathbf{u}$	–	0
Total			533

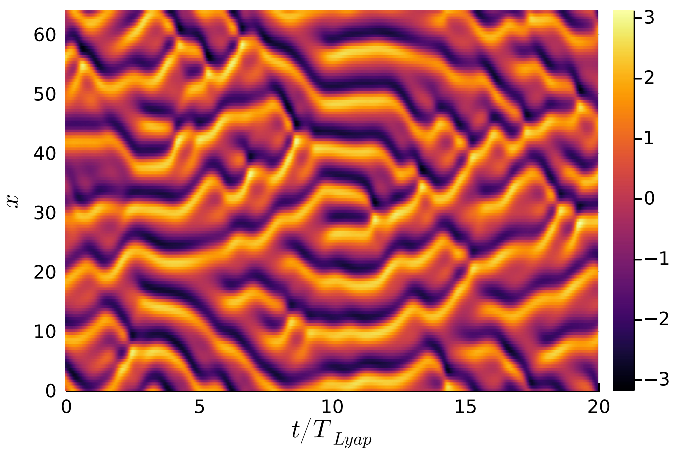


Fig. A.9. An example trajectory from the Kuramoto-Sivashinsky equation used for training models.

Experiments for the Burgers equation are done only with the smaller of the two neural networks. Experiments for the Kuramoto-Sivashinsky equation are done only with the larger neural network. Note that both neural networks, especially the small neural network, are very small compared to networks used in modern machine learning applications. However, as seen from the results, both neural networks are large enough to significantly improve the accuracy over the coarse ODE without closure term.

The neural network structures are summarised in Tables B.4 and B.5. Note that both neural networks have an initial layer that extends the input vector $\mathbf{u}$ with its component-wise square $\mathbf{u}_i^2$. This is done since the true right-hand sides of both PDEs (A.2) and (A.4a) contain an advection term that depends directly on u^2. As such, passing the values $\mathbf{u}_i^2$ to the first convolutional layer is expected to improve the ability of the network to learn the closure term. Also note that the single bias parameter in the last convolutional layer of both models is actually meaningless: its value does not affect the output of the model due to the Δ_{fwd} layer. This layer ensures that the entries of the neural network output always sum to zero, which is a property that is also satisfied by the training data. Enforcing this property in the neural network was found to result in a small but consistent improvement in accuracy, see Chapter 4 of Melchers [22].

All neural networks are trained using the Adam optimiser [15] with a learning rate of 10^{-3}.

B.2. Discretise-then-optimise for the Kuramoto-Sivashinsky equation

As mentioned in Section 2.2, the discretise-then-optimise approach requires the use of a differentiable ODE solver. This is not a problem for Burgers' equation, but does pose a problem for the stiff Kuramoto-Sivashinsky equation since stiff equations are typically solved using implicit methods, which are not trivial to back-propagate through. Note that back-propagating through implicit methods is possible, since the gradient of an implicitly defined function (i.e. a function whose output is defined as the solution of a system of equations) can be computed by the implicit function theorem, as demonstrated by Kolter et al. [16]. Nevertheless, explicit ODE solvers are preferable over implicit methods whenever they are applicable, due to their simplicity and speed, as well as the property that back-propagation through explicit methods is comparatively easy.

For some problems including the Kuramoto-Sivashinsky equation, exponential time differencing Runge-Kutta methods are suitable. These methods assume a stiff but linear term which can be solved exactly using exponentials, combined with a non-stiff non-linear term that can be taken into account using multiple stages, similar to how standard explicit Runge-Kutta methods achieve higher orders of accuracy. Exponential integrators of orders 2, 3, and 4 were derived by Cox and Matthews [5]. A numerically stable way to compute the coefficients required by these methods was presented by Kassam and Trefethen [13] and demonstrated on the four-stage fourth-order accurate method `ETDRK4`. The resulting algorithm was found to perform very well on a variety of problems including the Kuramoto-Sivashinsky equation and will therefore be used here.

Exponential integrators for the Kuramoto-Sivashinsky equation are most efficient when the PDE is solved in the pseudo-spectral domain, meaning that the ODE is not over the variables $\mathbf{u}(t)$, but over their discrete Fourier transform $\hat{\mathbf{u}}(t) := \mathcal{F}\mathbf{u}(t)$ (the Fourier transform is only performed over space, not over time). Transforming the equation in this way yields the following ODE system:

$$\frac{\mathrm{d}}{\mathrm{d}t}\hat{\mathbf{u}} = \left(\mathbf{\Lambda}^2 - \mathbf{\Lambda}^4\right)\hat{\mathbf{u}} - \frac{i}{2}\mathbf{\Lambda}\mathcal{F}\left(\left(\mathcal{F}^{-1}\hat{\mathbf{u}}\right)^2\right), \tag{B.1}$$

where $\mathbf{\Lambda}$ is a diagonal matrix $\mathbf{\Lambda} = \operatorname{diag}\left(\lambda_0, \lambda_1, \dots, \lambda_{N_x-1}\right)$ where

$$\lambda_k = \begin{cases} \frac{2\pi k}{L} & \text{for } 0 \le k < \frac{N_x}{2}, \\ 0 & \text{for } k = \frac{N_x}{2}, \\ \frac{2\pi}{L}(k - N_x) & \text{for } \frac{N_x}{2} < k \le N_x - 1. \end{cases}$$

References

[1] Shady E. Ahmed, Suraj Pawar, Omer San, Adil Rasheed, Traian Iliescu, Bernd R. Noack, On closures for reduced order models – a spectrum of first-principle to machine-learned avenues, Phys. Fluids 33 (2021), https://doi.org/10.1063/5.0061577.

[2] Giancarlo Alfonsi, Reynolds-averaged Navier-Stokes equations for turbulence modeling, Appl. Mech. Rev. 62 (2009), https://doi.org/10.1115/1.3124648.

[3] Andrea D. Beck, David G. Flad, Claus-Dieter Munz, Deep neural networks for data-driven LES closure models, J. Comput. Phys. 398 (2019), https://doi.org/10.1016/j.jcp.2019.108910.

[4] Ricky T.Q. Chen, Yulia Rubanova, Jesse Bettencourt, David K. Duvenaud, Neural ordinary differential equations, Adv. Neural Inf. Process. Syst. 31 (2018) 6571–6583.

[5] Steven M. Cox, Paul C. Matthews, Exponential time differencing for stiff systems, J. Comput. Phys. 176 (2002) 430–455, https://doi.org/10.1006/jcph.2002.6995.

[6] Russell A. Edson, Judith E. Bunder, Trent W. Mattner, Anthony J. Roberts, Lyapunov exponents of the Kuramoto-Sivashinsky PDE, ANZIAM J. 61 (2019) 270–285, https://doi.org/10.1017/S1446181119000105.

[7] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, Adam M. Oberman, How to train your neural ODE: the world of Jacobian and kinetic regularization, arXiv:2002.02798, 2020.

[8] Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.

[9] Yifei Guan, Adam Subel, Ashesh Chattopadhyay, Pedram Hassanzadeh, Learning physics-constrained subgrid-scale closures in the small-data regime for stable and accurate LES, arXiv:2201.07347, 2022.

[10] Abhinav Gupta, Pierre F.J. Lermusiaux, Neural closure models for dynamical systems, Proc. R. Soc. A 477 (2021).

[11] Ernst Hairer, Syvert Paul Nørsett, Gerhard Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, Springer, 1993.

[12] Antony Jameson, Energy estimates for nonlinear conservation laws with applications to solutions of the Burgers equation and one-dimensional viscous flow in a shock tube by central difference schemes, in: Proceedings of the 18th AIAA Computational Fluid Dynamics Conference, 2007.

[13] Aly-Khan Kassam, Lloyd N. Trefethen, Fourth-order time-stepping for stiff PDEs, SIAM J. Sci. Comput. 26 (2005) 1214–1233, https://doi.org/10.1137/S1064827502410633.

[14] Christopher A. Kennedy, Mark H. Carpenter, Higher-order additive Runge–Kutta schemes for ordinary differential equations, Appl. Numer. Math. 136 (2019) 183–205.

[15] Diederik P. Kingma, Jimmy Ba Adam, A method for stochastic optimization, arXiv:1412.6980, 2015.

[16] Zico Kolter, David Duvenaud, Matt Johnson, Deep implicit layers – neural ODEs, deep equilibirum models, and beyond, http://implicit-layers-tutorial.org/. (Accessed 25 March 2022).

[17] Yoshiki Kuramoto, Diffusion-induced chaos in reaction systems, Prog. Theor. Phys. Suppl. 64 (1978) 346–367, https://doi.org/10.1143/PTPS.64.346.

[18] Björn List, Li-Wei Chen, Nils Thuerey, Learned turbulence modelling with differentiable fluid solvers, arXiv:2202.06988, 2022.

[19] Yingbo Ma, Vaibhav Dixit, Michael J. Innes, Xingjian Guo, Chris Rackauckas, A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions, in: 2021 IEEE High Performance Extreme Computing Conference (HPEC), 2021, pp. 1–9.

[20] Jonathan F. MacArt, Justin Sirignano, Jonathan B. Freund, Embedded training of neural-network subgrid-scale turbulence models, Phys. Rev. Fluids 6 (2021).

[21] Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, Hajime Asama, Dissecting neural ODEs, Adv. Neural Inf. Process. Syst. 33 (2020) 3952–3963.

[22] Hugo Melchers, Machine learning for closure models, Master's thesis, Eindhoven University of Technology, June 2022, https://ir.cwi.nl/pub/31972.

[23] ODE solvers > Recommended methods, https://docs.sciml.ai/DiffEqDocs/v7.1/solvers/ode_solve/. (Accessed 18 January 2022).

[24] Derek Onken, Lars Ruthotto, Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows, arXiv:2005.13420, 2020.

[25] Shaowu Pan, Karthik Duraisamy, Long-time predictive modeling of nonlinear dynamical systems using neural networks, Complexity (2018), https://doi.org/10.1155/2018/4801012.

[26] Jonghwan Park, Haecheon Choi, Toward neural-network-based large eddy simulation: application to turbulent channel flow, J. Fluid Mech. 914 (2021), https://doi.org/10.1017/jfm.2020.931.

[27] Jaideep Pathak, Alexander Wikner, Rebeckah Fussell, Sarthak Chandra, Brian R. Hunt, Michelle Girvan, Edward Ott, Hybrid forecasting of chaotic processes: using machine learning in conjunction with a knowledge-based model, Chaos 28 (2018), https://doi.org/10.1063/1.5028373.

[28] Suraj Pawar, S.M. Rahman, H. Vaddireddy, Omer San, Adil Rasheed, Prakash Vedula, A deep learning enabler for nonintrusive reduced order modeling of fluid flows, Phys. Fluids 31 (2019) 085101, https://doi.org/10.1063/1.5113494.

[29] Elisabeth Roesch, Christopher Rackauckas, Michael P.H. Stumpf, Collocation based training of neural ordinary differential equations, Stat. Appl. Genet. Mol. Biol. 20 (2021) 37–49, https://doi.org/10.1515/sagmb-2020-0025.

[30] Pierre Sagaut, Large Eddy Simulation for Incompressible Flows: An Introduction, Springer Science & Business Media, 2006.

[31] Filippo Salmoiraghi, Francesco Ballarin, Giovanni Corsi, Andrea Mola, Marco Tezzele, Gianluigi Rozza, et al., Advances in geometrical parametrization and reduced order models and methods for computational fluid dynamics problems in applied sciences and engineering: overview and perspectives, in: VII European Congress on Computational Methods in Applied Sciences and Engineering, vol. 1, 2016, pp. 1013–1031.

[32] Omer San, Romit Maulik, Neural network closures for nonlinear model order reduction, Adv. Comput. Math. 44 (2018) 1717–1750, https://doi.org/10.1007/s10444-018-9590-z.

[33] Justin Sirignano, Jonathan F. MacArt, Jonathan B. Freund, DPM: a deep learning PDE augmentation method with application to large-eddy simulation, J. Comput. Phys. 423 (2020), https://doi.org/10.1016/j.jcp.2020.109811.

[34] Gregory I. Sivashinsky, Nonlinear analysis of hydrodynamic instability in laminar flames—I. Derivation of basic equations, Acta Astronaut. 4 (1977) 1177–1206, https://doi.org/10.1016/0094-5765(77)90096-0.

[35] Gerd Steinebach, Order-reduction of ROW-methods for DAEs and method of lines applications, Preprint, Technische Hochschule Darmstadt, Fachbereich Mathematik, 1995.

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

[37] Kiwon Um, Robert Brand, Yun (Raymond) Fei, Philipp Holl, Nils Thuerey, Solver-in-the-loop: learning from differentiable physics to interact with iterative PDE-solvers, in: H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, vol. 33, Curran Associates, Inc., 2020, pp. 6111–6122.