



Neural semi-Lagrangian method for high-dimensional advection-diffusion problems

Emmanuel Franck, Victor Michel-Dansac, Laurent Navoret, Vincent Vigon

► To cite this version:

Emmanuel Franck, Victor Michel-Dansac, Laurent Navoret, Vincent Vigon. Neural semi-Lagrangian method for high-dimensional advection-diffusion problems. 2025. <hal-05051195v3>

HAL Id: hal-05051195

<https://hal.science/hal-05051195v3>

Preprint submitted on 7 Oct 2025 (v3), last revised 9 Dec 2025 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY-NC-ND 4.0 - Attribution - Non-commercial use - No Derivative Works - International License

Neural semi-Lagrangian method for high-dimensional advection-diffusion problems

Emmanuel Franck^a, Victor Michel-Dansac^{a,*}, Laurent Navoret^{b,a}, Vincent Vigon^{b,a}

^aUniversité de Strasbourg, CNRS, Inria, IRMA, F-67000, Strasbourg, France

^bIRMA, Université de Strasbourg, CNRS UMR 7501, 7 rue René Descartes, 67084, Strasbourg, France

Abstract

This work is devoted to the numerical approximation of high-dimensional advection-diffusion equations. It is well-known that classical methods, such as the finite volume method, suffer from the curse of dimensionality, and that their time step is constrained by a stability condition. The semi-Lagrangian method is known to overcome the stability issue, while recent time-discrete neural network-based approaches overcome the curse of dimensionality. In this work, we propose a novel neural semi-Lagrangian method that combines these last two approaches. It relies on projecting the initial condition onto a finite-dimensional neural space, and then solving an optimization problem, involving the backwards characteristic equation, at each time step. It is particularly well-suited for implementation on GPUs, as it is fully parallelizable and does not require a mesh. We provide rough error estimates, and present several high-dimensional numerical experiments to assess the performance of our approach, and compare it to other neural methods.

Keywords: Semi-Lagrangian method, Advection-diffusion equations, Advection equations, Scientific machine learning, Neural networks

2020 MSC: 65M25, 76R05, 65M15, 68T07

Contents

1	Introduction	2
2	Classical and neural numerical methods	3
2.1	Classical methods	4
2.1.1	Spacetime Galerkin method	4
2.1.2	Galerkin method	5
2.1.3	Semi-Lagrangian Galerkin Method	5
2.2	Neural methods	6
2.2.1	Physics-Informed Neural Networks (PINNs)	7
2.2.2	Discrete PINNs and the Neural Galerkin method	7
2.2.3	Brief comparison of space-time and time-sequential neural methods	8
3	Neural Semi-Lagrangian method	9
3.1	Methodology	9
3.2	Using the method in practice	10
3.3	Rough error estimates	12
3.4	Extension to advection-diffusion equations	16
3.5	Further improvements	16
3.5.1	Natural gradient preconditioning	17
3.5.2	Adaptive sampling	18
3.5.3	Boundary conditions	18

*corresponding author

Email addresses: emmanuel.franck@inria.fr (Emmanuel Franck), victor.michel-dansac@inria.fr (Victor Michel-Dansac), laurent.navoret@math.unistra.fr (Laurent Navoret), vincent.vigon@math.unistra.fr (Vincent Vigon)

4	Validation	19
4.1	Constant advection in 1D	19
4.1.1	Non-parametric case	19
4.1.2	Parametric case	22
4.2	Nonconstant advection in 2D	22
4.2.1	2D parametric rotating transport	23
4.2.2	1D1V Vlasov equation	24
4.3	Transport equation in a cylinder	27
4.4	Deformation of level-set functions	29
4.4.1	Deformation of a 2D level-set function	29
4.4.2	Deformation of a 3D level-set function	30
4.4.3	Deformation of a 3D level-set function with two parameters	32
4.5	High-dimensional advection-diffusion equations	33
4.5.1	Convergence study	33
4.5.2	High-dimensional periodic solution	35
4.5.3	High-dimensional Gaussian solution	36
4.5.4	Discussion	38
4.6	Extension to the Vlasov-Poisson equations	38
4.6.1	Governing equations and solving strategy	38
4.6.2	Validation: the bump-on-tail instability	39
5	Conclusion	41
	References	46
Appendix A	Algorithm to compute the approximate characteristic curve	46
Appendix B	Hyperparameters for the numerical experiments	46

1. Introduction

In this work, we are interested in solving high-dimensional advection-diffusion equations. The large dimensionality may be inherent, for instance in kinetic equations (kinetic transport, Vlasov, or Fokker-Planck equations), where the unknown typically depends on time, space and velocity. It can also be parametric (examples include equations with parametric source terms, differential operators, or boundary conditions). Traditional numerical methods, such as finite difference, finite volume, or finite element methods, applied to such high-dimensional problems, require numerous degrees of freedom to achieve a given level of accuracy. Indeed, the number of degrees of freedom grows as N^d , where N is the number of degrees of freedom in each dimension and d is the number of dimensions. Thus, N grows exponentially with d , leading to a curse of dimensionality. This exponential growth is a major obstacle to the numerical solution of high-dimensional kinetic equations, since it leads to a large computational cost and large memory requirements. To address such an issue, several methods have been developed, including model order reduction [64] and low-rank tensor decomposition [45]. These methods express the solution with a reduced number of degrees of freedom.

In recent years, neural networks have also shown their efficiency in reducing the number of degrees of freedom by enriching classical approximation spaces. For example, tensor decomposition methods have successfully been combined with neural network approximations, see e.g. [61, 35, 36]. More generally, it turns out that several classical methods have seen their neural counterparts developed. The foremost instances of this are the Physics Informed Neural Networks (PINNs) [66, 48, 22] and the Deep Ritz method [25]. In such methods, the solution itself is expressed as a neural network, whose parameters are adjusted such that the equation is satisfied in strong or weak form. These methods have produced good results for elliptic Partial Differential Equations (PDEs) in low or high physical or parametric dimensions, see e.g. [73, 3, 42]. There are also a few applications to high-dimensional transport, although these remain more limited [57, 81, 43]. Indeed, as will be detailed later, PINNs is a spacetime method, which treat space and time in the same way and specific techniques have to be considered to incorporate causality in the training to properly capture the solution, see [77]. To improve the accuracy of PINNs for transport problems, some work has been undertaken, either incorporating knowledge of the characteristic curves associated with the problem, see [41], or using a

Lagrangian formulation (which is a similar idea), see [58]. In these cases, a space-time approximation is used, contrary to the sequential-in-time methods presented below.

For time-dependent equations, another strategy is to use neural networks approximations as functions of space only, and make the neural network parameters evolve in time. This led to the development of the discrete PINNs [72, 7] or Neural Galerkin [52, 33, 11, 44] methods. These methods can be seen as extensions of the classical implicit and explicit Galerkin methods, where the finite-dimensional approximation linear subspace has been replaced with a submanifold of neural networks.

When considering advection equations and even advection-diffusion equations in high dimension, however, semi-Lagrangian approaches [24, 62, 71, 70, 32, 79, 19, 80] are known to be more efficient than standard Galerkin method. At each time step, by using the backward integration of the characteristic curves, the solution is transported exactly before being projected onto the spatial approximation space. This projection either relies on interpolation methods, like splines, Lagrange or Hermite interpolations [70, 19, 6], or is a Galerkin projection [20, 68, 63]. Dimensional splitting can be deployed to go back to one dimensional projection problems as these projections may be particularly computationally expensive in high dimension. A key feature of semi-Lagrangian schemes is that they have no time step stability constraints, even though they are explicit in time. This reduces the computational cost: the time step is then chosen only to increase the accuracy. Recently, neural network-based methods have been proposed to improve the accuracy of traditional grid-based semi-Lagrangian methods [50, 15, 14]. However, as explained above, numerical simulations for high-dimensional problems are still hard to perform due to the large number of required degrees of freedom used by the grid-based methods. To reduce this number, low-rank tensor methods have recently been developed for kinetic equations, see e.g. [46, 26, 82] or the review [27].

In order to tackle transport dynamics in large dimension, we propose in this work to combine the semi-Lagrangian approach with neural approximations. The classical projection onto a linear subspace at each time step of the transported solution is then replaced with an optimization process to fit the parameters. This approach constitutes a semi-Lagrangian variant of the discrete PINN method. Indeed, the first step is to project the initial condition onto the neural approximation space; this is nothing but a nonlinear optimization problem. Then, at each time step, the approximate solution is transported using the characteristic curves, and the parameters are updated by solving a nonlinear optimization problem.

In the following, we will apply this method to the following parametric linear *advection-diffusion* problem, describing the dynamics of an unknown $u(t, x, \mu) \in \mathbb{R}$, depending on time variable $t \in \mathbb{R}_+$, space variable $x \in \Omega \subset \mathbb{R}^d$ and $\mu \in \mathbb{M} \subset \mathbb{R}^p$ a set of physical parameters. The PDE writes:

$$\begin{cases} \partial_t u(t, x, \mu) + a(t, x, \mu) \cdot \nabla u(t, x, \mu) - \sigma \Delta u(t, x, \mu) = 0, & x \in \Omega, t \in (0, T), \\ u(t = 0, x, \mu) = u_0(x, \mu), & x \in \Omega, \\ u(t, x, \mu) = g(t, x, \mu), & x \in \partial\Omega, t \in (0, T), \end{cases} \quad (1.1)$$

where $a(t, x, \mu) \in \mathbb{R}^d$ is the advection field and $\sigma > 0$ is the constant diffusion coefficient. The parameters may appear in the boundary or initial conditions, or in the advection field. We will formally write the PDE without initial and boundary conditions as

$$\partial_t u + \mathcal{L}[u] = 0, \quad (1.2)$$

where the operator $\mathcal{L}$ is linear in u and contains no time derivatives of u , or as

$$\mathcal{T}[u] = 0, \quad (1.3)$$

where the operator $\mathcal{T}$ is linear in u and involves both time and space derivatives. For the remainder of the paper, we assume that the velocity field is known and is not determined by another equation.

The paper is organized as follows. In [Section 2](#), we introduce the classical numerical approaches for this problem and their neural extensions. This discussion illustrates that our approach naturally follows from previous work. Then, in [Section 3](#), we derive the proposed method for pure advection equations, and extend it to advection-diffusion equations. Algorithms and rough error estimates are provided. Finally, in [Section 4](#), we provide several validation experiments, for both pure advection and advection-diffusion equations, in high-dimensional settings. [Section 5](#) concludes the paper.

2. Classical and neural numerical methods

The goal of this section is to highlight the key differences between classical and neural methods, and to cast the latter into a formalism well-known in the case of classical methods. To that end, [Section 2.1](#) is

devoted to classical methods, while [Section 2.2](#) tackles neural methods. We introduce an infinite-dimensional Hilbert space V such that $u \in V$, where u is the solution to the partial differential equation (1.1). In both [Section 2.1](#) and [Section 2.2](#), this function space V will be approximated by a finite-dimensional subspace, denoted by $V_N^{\text{classical}}$ and V_N^{neural} for classical and neural methods, respectively. In the two above-defined spaces, the subscript N represents the number of degrees of freedom. The main difference between these two finite-dimensional subspaces is that, while $V_N^{\text{classical}}$ is a linear subspace spanned by some given basis functions, V_N^{neural} is no longer linear. It may become a submanifold of V , depending on the nonlinearity. As such, projecting an element of V onto $V_N^{\text{classical}}$ amounts to solving linear systems (or quadratic optimization problems), while projecting it onto V_N^{neural} requires solving nonlinear, usually non-convex, optimization problems.

First, [Section 2.1](#) introduces three families of classical methods for advection-diffusion equations. These approaches all use Galerkin projections onto a finite-dimensional subspace to represent the PDE solution. Namely, we present the spacetime Galerkin method in [Section 2.1.1](#), the Galerkin method in [Section 2.1.2](#), and the semi-Lagrangian Galerkin method in [Section 2.1.3](#). Second, we discuss neural numerical methods in [Section 2.2](#). We briefly recall that PINNs ([Section 2.2.1](#)) and their time-discrete counterparts ([Section 2.2.2](#)) can be seen as extensions of classical approaches, where the projection onto a finite-dimensional subspace (finite element space, spectral basis, etc.) is replaced by a projection onto the manifold defined by neural networks with given architectures.

For simplicity, in this section, we drop the dependence on the parameter μ in the solution u . We also do not discuss boundary and initial conditions. Moreover, throughout this section, u_N denotes the approximate solution of the PDE, obtained by the numerical method under consideration in each specific paragraph, with N dofs.

2.1. Classical methods

As mentioned above, classical approaches to solve time-dependent PDEs approximate the solution by projecting the equation onto a finite-dimensional linear subspace $V_N^{\text{classical}}$ of the Hilbert space V . To properly introduce this linear subspace, let $N \in \mathbb{N}$, and define N basis functions $\phi = \{\phi_1, \dots, \phi_N\}$. Examples of basis functions include piecewise polynomial functions for finite element methods or Fourier basis functions for spectral methods. The approximate solution u_N then belongs to the N -dimensional linear subspace $V_N^{\text{classical}}$, defined using the basis functions by

$$V_N^{\text{classical}} = \left\{ \sum_{j=1}^N \theta_j \phi_j = \langle \theta, \phi \rangle, \theta \in \Theta \subset \mathbb{R}^N \right\} = \text{Span}(\phi_1, \dots, \phi_N),$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in $\mathbb{R}^N$ and $\theta = (\theta_1, \dots, \theta_N)$. The goal is to find the degrees of freedom (dofs) $\theta \in \Theta \subset \mathbb{R}^N$ that define u_N . These dofs will depend on the choice of basis functions and on the equation to solve. Since we are considering linear equations and projections onto the linear subspace $V_N^{\text{classical}}$, the dofs θ will end up being determined by solving linear systems.

2.1.1. Spacetime Galerkin method

In the spacetime Galerkin method (see e.g. [\[65\]](#)), the basis functions depend on both space and time. The solution is approximated, for all $t \in (0, T)$ and $x \in \Omega$, by

$$u_N(t, x) = \sum_{j=1}^N \theta_j \phi_j(t, x) = \langle \theta, \phi(t, x) \rangle. \quad (2.1)$$

Now, to derive the equation for the dofs θ , we plug the spacetime Galerkin approximation (2.1) into the PDE (1.3), and integrate over the spacetime slab $(0, T) \times \Omega$ against some test function. We obtain, for all $i \in \{1, \dots, N\}$,

$$\int_{\Omega} \int_0^T \mathcal{T}[u_N](t, x) \phi_i(t, x) dt dx = 0.$$

Using (2.1), we obtain

$$\int_{\Omega} \int_0^T \sum_{j=1}^N \theta_j \mathcal{T}[\phi_j](t, x) \phi_i(t, x) dt dx = 0,$$

and so

$$\sum_{j=1}^N \theta_j \left(\int_{\Omega} \int_0^T \phi_i(t, x) \mathcal{T}[\phi_j](t, x) dt dx \right) = 0.$$

This leads to the linear system $M\theta = 0$, where the components of the mass matrix M are defined by

$$M_{ij} = \int_{\Omega} \int_0^T \phi_i(t, x) \mathcal{T}[\phi_j](t, x) dt dx.$$

The solution of this linear system then gives the dofs θ .

2.1.2. Galerkin method

Now, assume that the basis functions only depend on space, and that the dofs θ depend on time. The idea is to approximate the solution at each discrete time using these space-dependent basis functions. This is equivalent to stating that the linear combination of the basis functions used to represent the solution depends on time. Thus, the approximate solution $u_N \in V$ at time $t \in (0, T)$ and position $x \in \Omega$ is given by

$$u_N(t, x) = \sum_{j=1}^N \theta_j(t) \phi_j(x) = \langle \theta(t), \phi(x) \rangle. \quad (2.2)$$

We now plug the Galerkin approximation (2.2) into the PDE (1.2), and we integrate over Ω against some test function. We obtain, for all $i \in \{1, \dots, N\}$,

$$\int_{\Omega} (\partial_t u_N(t, x) + \mathcal{L}[u_N](t, x)) \phi_i(x) dx = 0.$$

Using the expression (2.2) yields

$$\int_{\Omega} \sum_{j=1}^N \left(\frac{d\theta_j(t)}{dt} \phi_j(x) + \theta_j(t) \mathcal{L}[\phi_j](x) \right) \phi_i(x) dx = 0,$$

which we recast as

$$\sum_{j=1}^N \frac{d\theta_j(t)}{dt} \int_{\Omega} \phi_i(x) \phi_j(x) dx + \sum_{j=1}^N \theta_j \int_{\Omega} \phi_i(x) \mathcal{L}[\phi_j](x) dx = 0.$$

We finally obtain a linear ODE on the dofs, which reads

$$M \frac{d\theta(t)}{dt} = L\theta(t),$$

where the components of the matrices M and L are defined by

$$M_{ij} = \int_{\Omega} \phi_i(x) \phi_j(x) dx \quad \text{and} \quad L_{ij} = \int_{\Omega} \phi_i(x) \mathcal{L}[\phi_j](x) dx.$$

This linear ODE is then solved using a time-stepping method. We also refer to [65] for more details.

2.1.3. Semi-Lagrangian Galerkin Method

The Galerkin semi-Lagrangian method is a subset of the Galerkin method, specifically designed for advection and advection-diffusion equations. The idea is to avoid the time step restriction by using a Lagrangian approach [24, 62]. We still write the approximate solution under the form (2.2).

We introduce the method in the specific case of a pure transport equation, i.e., when $\sigma = 0$ in (1.1). The exact solution satisfies, for all $t \in (0, T)$, $x \in \Omega$ and $s \in (0, t)$,

$$u(t, x) = u(s, \mathcal{X}(s; t, x)), \quad (2.3)$$

where $\mathcal{X}$ is the (backwards) characteristic curve implicitly defined by the ODE

$$\begin{cases} \frac{d}{ds} \mathcal{X}(s; t, x) = a(s, \mathcal{X}(s; t, x)) & \text{for all } s \in (0, t), \\ \mathcal{X}(t; t, x) = x. \end{cases} \quad (2.4)$$

Using this, we can formulate the Semi-Lagrangian Galerkin method. To do this, we approximate u by u_N in (2.3), and integrate over Ω against a test function. We thus impose that our approximate solution must satisfy, for all $t \in (0, T)$ and $s \in (0, t)$,

$$\forall i \in \{1, \dots, N\}, \quad \int_{\Omega} u_N(t, x) \phi_j(x) dx = \int_{\Omega} u_N(s, \mathcal{X}(s; t, x)) \phi_j(x) dx.$$

Plugging the Galerkin approximation (2.2) into the above equation, we obtain

$$\forall i \in \{1, \dots, N\}, \quad \sum_{j=1}^N \theta_j(t) \int_{\Omega} \phi_j(x) \phi_i(x) dx = \sum_{j=1}^N \theta_j(s) \int_{\Omega} \phi_j(\mathcal{X}(s; t, x)) \phi_i(x) dx.$$

Thus, from some known degrees of freedom $\theta(s)$ at some time $s < t$, we can compute the degrees of freedom $\theta(t)$ at time t by solving the above linear system. This is achieved without a stability condition linking t to s , contrary to the previous methods.

In practice, we define a time discretization $(t^n)_n$, with $t^{n+1} = t^n + \Delta t$, and we obtain $\theta^0 = \theta(0)$ by projecting the initial condition onto the Galerkin basis. Then, at each time step, θ^{n+1} is computed by solving

$$\forall j \in \{1, \dots, N\}, \quad \sum_{j=1}^N \theta_j^{n+1} \int_{\Omega} \phi_j(x) \phi_i(x) dx = \sum_{j=1}^N \theta_j^n \int_{\Omega} \phi_j(\mathcal{X}(t^n; t^{n+1}, x)) \phi_i(x) dx.$$

The above linear system is rewritten as

$$M(t^{n+1}, t^{n+1}) \theta^{n+1} = M(t^n, t^{n+1}) \theta^n,$$

where the elements of mass matrix $M(s, t)$ are defined by

$$M(s, t)_{ij} = \int_{\Omega} \phi_j(\mathcal{X}(s; t, x)) \phi_i(x) dx.$$

Thanks to this formulation, this method does not have a time step restriction on Δt . As an example, if a is a constant vector field, solving (2.4) is immediate, and we get $\mathcal{X}(s; t, x) = x - (t - s)a$. In this case, the components of the mass matrix simply read

$$M(t^n, t^{n+1})_{ij} = \int_{\Omega} \phi_i(x) \phi_j(x - a\Delta t) dx.$$

Moreover, if a discontinuous Galerkin basis is used, the mass matrix M becomes diagonal, rendering the method particularly efficient [20, 68].

The method can also be extended to advection-diffusion equations, see e.g. [8]. We do not detail this extension here, but the same principle applies. Namely, the feet of the characteristic curves still have to be computed as in the pure advection case. Then, in the presence of diffusion, these feet have to be split into several diffusion directions.

2.2. Neural methods

We now recall some recent neural numerical methods. In such methods, as mentioned above, the approximate function space $\mathcal{V}_N^{\text{neural}}$ is no longer a linear subspace spanned by basis functions, but a nonlinear subspace (usually, a submanifold) defined by parameterized nonlinear functions (usually, neural networks). However, it remains finite-dimensional, and the degrees of freedom are the weights of the nonlinear functions. To be clear, let

$$\begin{aligned} \mathcal{N} : \mathbb{R}^{\mathfrak{d}} \times \mathbb{R}^N &\rightarrow \mathbb{R} \\ (X, \theta) &\mapsto \mathcal{N}(X, \theta) \end{aligned} \tag{2.5}$$

be a parameterized nonlinear function, typically a neural network. Its input $X \in \mathbb{R}^{\mathfrak{d}}$ will either be $(t, x) \in (0, T) \times \Omega$, to define spacetime neural methods, or $x \in \Omega$, to define time-discrete neural methods. In the first case, $\mathfrak{d} = d + 1$; in the second one, $\mathfrak{d} = d$. Approximating the PDE solution by such nonlinear functions leads to a nonlinear approximation space. Thus, one now expects to solve nonlinear optimization problems, rather than linear systems, to find the dofs θ . Remark that nonlinear optimization problems have to be solved even when approximating linear equations.

Note that the dofs of the nonlinear function $\mathcal{N}$ remain denoted by $\theta \in \Theta \subset \mathbb{R}^N$. For instance, if $\mathcal{N}$ is a neural network, then the dofs θ are nothing but the weights and biases of the network. Similarly to [Section 2.1](#), the dependence of the dofs in space and time alters the nature of the neural method. Namely, dofs that depend on neither space nor time lead to PINNs, discussed in [Section 2.2.1](#), while time-dependent dofs lead to discrete PINNs and the neural Galerkin method, discussed in [Section 2.2.2](#). A brief comparison of both space-time and time-discrete neural methods is provided in [Section 2.2.3](#).

2.2.1. Physics-Informed Neural Networks (PINNs)

To introduce PINNs [\[66, 48, 22\]](#), let us define the nonlinear subspace

$$\mathcal{V}_N^{\text{neural}} = \{(x, t) \mapsto \mathcal{N}(x, t, \theta), \theta \in \Theta\}.$$

The function $\mathcal{N}$ depends on both space and time. To highlight this dependence, and to distinguish the parameters from the space and time variables, we introduce the notation

$$\begin{aligned} u_\theta &: (0, T) \times \Omega \rightarrow \mathbb{R} \\ (t, x) &\mapsto \mathcal{N}(t, x, \theta). \end{aligned}$$

Then, a function $u_N \in \mathcal{V}_N^{\text{neural}}$ is written as

$$u_N(t, x) = u_\theta(t, x).$$

Note that u_N depends on both space and time, similarly to the spacetime Galerkin method.

Finding the dofs θ then consists in directly minimizing, over the spacetime slab, the residual of the PDE [\(1.3\)](#) applied to the approximate solution u_N . This leads to the nonlinear optimization problem

$$\theta \in \arg \min_{\vartheta \in \Theta} \int_0^T \int_\Omega |\mathcal{T}[u_\vartheta](t, x)|^2 dt dx. \quad (2.6)$$

It is usually solved using stochastic gradient methods, discretizing the integral with the Monte-Carlo algorithm.

2.2.2. Discrete PINNs and the Neural Galerkin method

Discrete PINNs and the Neural Galerkin method take up the ideas behind the Galerkin method, but applied to the context of nonlinear function spaces. Namely, we replace the approximation space with

$$\mathcal{V}_N^{\text{neural}, t} = \{(x, t) \mapsto \mathcal{N}(x, \theta(t)), \theta(t) \in \Theta\}.$$

The function $\mathcal{N}$ still depends on space, time, and dofs, but its time dependence is no longer explicit; rather, it depends on time because the dofs themselves are time-dependent. To emphasize this dependence, we introduce the notation

$$\begin{aligned} u_{\theta(t)} &: \Omega \rightarrow \mathbb{R} \\ x &\mapsto \mathcal{N}(x, \theta(t)). \end{aligned}$$

The approximate function u_N now depends on space only, while the dofs θ (i.e., the weights and biases of the neural network if u_θ is a neural network) depend on time. This helps to avoid some issues with causality and time integration (outlined in e.g. [\[56\]](#)), and leads to the following approximation:

$$u_N(t, x) = u_{\theta(t)}(x).$$

The dofs $\theta(t)$ at time t are then determined such that the residual of equation [\(1.2\)](#),

$$\int_\Omega \left| \partial_t u_{\theta(t)}(x) + \mathcal{L}[u_{\theta(t)}](x) \right|^2 dx, \quad (2.7)$$

is minimal in L^2 norm. Of course, this minimizer is only approximated, since the true solution to the PDE lies in the infinite-dimensional Hilbert space V , while the approximate solution $u_{\theta(t)}$ is in the finite-dimensional subspace $\mathcal{V}_N^{\text{neural}, t}$. Then, to evolve the dofs in time, multiple strategies are available, two of which are detailed in the remainder of this section. Before that, we remark that the initial condition $\theta^0 := \theta(0)$ is obtained by fitting the initial condition of the PDE:

$$\theta^0 \in \arg \min_{\vartheta \in \Theta} |u_\vartheta(x) - u_0(x)|^2 dx.$$

Discrete PINNs [72, 7]. Recall the already-introduced time discretization $(t^n)_n$. Then, discrete PINNs rely on directly discretizing the time derivative in (2.7). For simplicity, we consider a simple explicit Euler scheme, but more sophisticated schemes can, and should be, used. This yields, with $\theta^n := \theta(t^n)$,

$$\partial_t u_{\theta(t)} \simeq \frac{u_{\theta^{n+1}} - u_{\theta^n}}{\Delta t} \quad \text{and} \quad \mathcal{L}[u_{\theta(t)}] \simeq \mathcal{L}[u_{\theta^n}].$$

Using the above equation, we define the updated parameters θ^{n+1} as minimizers of the residual:

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \int_{\Omega} \left| u_{\vartheta}(x) - u_{\theta^n}(x) + \Delta t \mathcal{L}[u_{\theta^n}](x) \right|^2 dx. \quad (2.8)$$

Neural Galerkin method [52, 33, 11, 44]. To derive the neural Galerkin method, we go back to (2.7). Applying the chain rule to the first term gives

$$\partial_t u_{\theta(t)}(x) = \left\langle \nabla_{\theta} u_{\theta(t)}(x), \frac{d\theta(t)}{dt} \right\rangle,$$

where $\nabla_{\theta} u_{\theta}$ is the gradient of u_{θ} with respect to θ . Therefore, we seek the dofs θ whose time derivative minimize, in the L^2 norm, the residual

$$\int_{\Omega} \left| \left\langle \nabla_{\theta} u_{\theta(t)}(x), \frac{d\theta(t)}{dt} \right\rangle + \mathcal{L}[u_{\theta(t)}](x) \right|^2 dx.$$

The time derivative is thus defined as the minimizer of the above residual:

$$\frac{d\theta(t)}{dt} \in \arg \min_{\eta \in \mathbb{R}^N} \int_{\Omega} \left| \left\langle \nabla_{\theta} u_{\theta(t)}(x), \eta \right\rangle + \mathcal{L}[u_{\theta(t)}](x) \right|^2 dx.$$

This is nothing but a quadratic optimization problem, and its exact solution is

$$M(\theta) \frac{d\theta(t)}{dt} = -L(\theta), \quad (2.9)$$

where the matrix M and the vector L are defined by

$$M(\theta) = \int_{\Omega} \nabla_{\theta} u_{\theta(t)}(x) \otimes \nabla_{\theta} u_{\theta(t)}(x) dx \quad \text{and} \quad L(\theta) = \int_{\Omega} \nabla_{\theta} u_{\theta(t)}(x) \mathcal{L}[u_{\theta(t)}](x) dx, \quad (2.10)$$

where $\otimes$ denotes the outer product of two vectors of $\mathbb{R}^N$. Therefore, finding the dofs in the neural Galerkin method amounts to solving the nonlinear ODE (2.9). This is usually done using standard ODE solvers.

2.2.3. Brief comparison of space-time and time-sequential neural methods

Space-time vs time-sequential methods. Time-sequential methods are less commonly used in the literature than standard PINNs, likely because they are somewhat more challenging to implement. However, they have consistently produced significantly more accurate results as soon as the PDE becomes even moderately difficult. Comparisons have been carried out in particular in [72, 7, 11]. This is consistent with classical numerical methods, where time-sequential approaches are much more commonly used than space-time methods (though this is less true for certain linear PDEs). In practice, the main limitation to the accuracy of neural methods lies in the optimization process. For moderately difficult PDEs, optimization becomes significantly more challenging for space-time approaches (although this can be improved with progressive time training) compared to sequential methods, where optimization is simpler: for instance, a straightforward projection in the case of an explicit discrete PINN, or a least-squares problem for neural Galerkin. The numerical experiments from Section 4 include a comparison of these time-sequential methods with space-time PINNs.

Discrete PINNs vs Neural Galerkin. The neural Galerkin and discrete PINNs methods can be interpreted within the same framework of time-sequential methods. Discrete PINNs can be viewed as a “discretize-then-optimize” strategy, in which the time variable is first discretized, and the solution at the following time step is then obtained via optimization (see [16]). In contrast, the Neural Galerkin method follows an “optimize-then-discretize” paradigm. In some cases, it can be derived as a linearization of the discrete PINNs for small time

steps. In our experiments, in the explicit setting, we observed that the neural Galerkin method is more accurate than discrete PINNs trained with Adam, but less accurate than discrete PINNs trained using a natural gradient approach (see [16]). This suggests that, in the case of Adam, the optimization error may be larger than the error introduced by the linearization in the neural Galerkin method. For discrete PINNs, boundary conditions can be enforced either strongly or weakly, as in standard PINNs. However, in the Neural Galerkin framework, weak enforcement of boundary conditions would introduce a non-convex loss, which would undermine the benefits of the method. Therefore, only strong enforcement is generally applicable, which may, in some cases, reduce efficiency. Both methods share the same neural architectures, geometry handling, collocation point sampling, and high-dimensional capabilities. As such, the choice between the two approaches will depend primarily on computational cost and accuracy requirements. In the implicit setting, discrete PINNs reduce to solving an elliptic PINN problem at each time step, whereas the neural Galerkin method leads to solving a Galerkin problem in the tangent space. The linearization inherent in the neural Galerkin method may limit its performance in the implicit case. Several test cases comparing the two approaches are available in [16].

3. Neural Semi-Lagrangian method

In this paper, we propose a new approach to solve advection-diffusion equations. Similarly to how space-time and Galerkin methods have their neural equivalents, discussed in Section 2.2, we propose to extend the semi-Lagrangian Galerkin method from Section 2.1.3 to the neural domain.

This approach is quite different from the ones proposed in [50, 15, 14]. In these works, the idea is to maintain a grid-based interpolation while using a neural network to correct the classical interpolation. However, in all cases, the approach still relies on grid-based interpolation, which inherits the challenges of classical methods in high dimensions. Even if such methods make it possible to use coarser grids, in high dimensions the grid size remains large, and both the number of degrees of freedom and the computational cost can still grow explosively.

Similarly to Section 2.2.2, the solution will be approximated by a nonlinear function whose dofs depend on time:

$$u_N(t, x) = u_{\theta(t)}(x). \quad (3.1)$$

In practice, we will use neural networks to represent u_{θ} .

This extension has several advantages compared to traditional methods and to other traditional and neural methods:

- compared to traditional methods, it is able to tackle high-dimensional problems with a comparatively low cost, since the number of degrees of freedom does not increase exponentially;
- compared to traditional PINNs, it remains sequential in time, thus avoiding issues observed with PINNs and highlighted in e.g. [56];
- it eliminates the (sometimes very restrictive, and for the moment unknown) stability condition on the time step that discrete PINNs and the neural Galerkin method retain (if explicit time integration is used);
- its optimization problem avoids the need to compute an additional PDE residual, compared to the discrete PINN and neural Galerkin methods with implicit time-stepping.

However, it should be noted that the semi-Lagrangian approach is predominantly suited for advection-diffusion or Fokker-Planck problems like (1.1). This restricts its applicability to a broader range of phenomena. Moreover, compared to traditional semi-Lagrangian methods, the use of a nonlinear representation makes proving convergence estimates much harder, and pretty much unattainable at this state.

3.1. Methodology

Recall that the solution to the advection equation satisfies (2.3). Hence, we seek an approximate solution u_N that also satisfies it:

$$\forall t \in (0, T), \forall s \in (0, t), \forall x \in \Omega, \quad u_N(t, x) \simeq u_N(s, \mathcal{X}(s; t, x)).$$

Plugging (3.1), we obtain

$$\forall t \in (0, T), \forall s \in (0, t), \forall x \in \Omega, \quad u_{\theta(t)}(x) \simeq u_{\theta(s)}(\mathcal{X}(s; t, x)). \quad (3.2)$$

The discrete version of (3.2) reads, using the time discretization $(t^n)_n$,

$$\forall n \geq 0, \forall x \in \Omega, \quad u_{\theta(t^{n+1})}(x) \simeq u_{\theta(t^n)}(\mathcal{X}(t^n; t^{n+1}, x)).$$

We wish to approximately satisfy the above equation in the L^2 norm. Therefore, we define the dofs $\theta^{n+1} = \theta(t^{n+1})$ at time t^{n+1} as the minimizer of the following optimization problem:

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \int_{\Omega} |u_{\vartheta}(x) - u_{\theta^n}(\mathcal{X}(t^n; t^{n+1}, x))|^2 dx. \quad (3.3)$$

Just like the semi-Lagrangian Galerkin method, this method does not have a stability condition on the time step. The newly introduced neural (meshless) semi-Lagrangian scheme is visually compared to a classical (mesh-based) one in Figure 1.

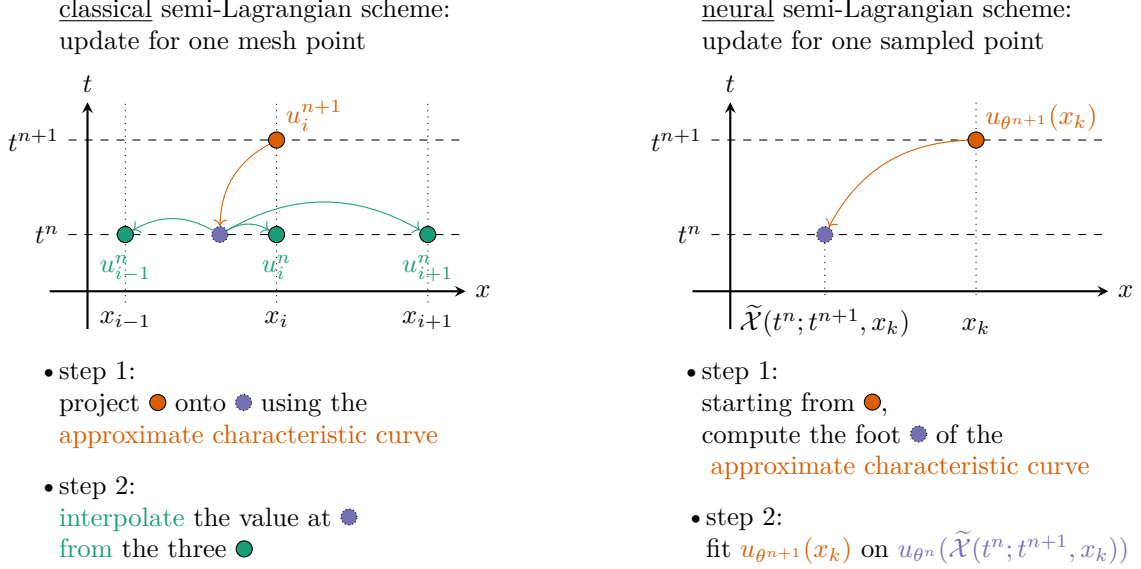


Figure 1: Visual comparison of the classical (left) and neural (right) semi-Lagrangian schemes in one space dimension. In the mesh-based classical scheme, the update is performed on a mesh, and thus an interpolation onto the mesh is needed after computing the foot of the characteristic curve. Conversely, in the meshless neural scheme, the approximation at the previous time step is directly evaluated at the foot of the characteristic curve.

In the linear advection case, we recall that the characteristic curve is given by $\mathcal{X}(s; t, x) = x - (t - s)a$. Therefore, the optimization problem reads

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \int_{\Omega} |u_{\vartheta}(x) - u_{\theta^n}(x - a\Delta t)|^2 dx.$$

In other cases, the ODE (2.4) defining the characteristic curves has to be solved, either exactly or approximately, to compute $\mathcal{X}$. This is detailed in the following section.

3.2. Using the method in practice

In practice, we do not exactly solve the optimization problem (3.3). Indeed, we have to make three adjustments to make this problem computationally feasible.

1. As usual, the integrals are approximated using the Monte-Carlo method. Given a number N_c of samples, called “collocation points” and denoted by $(x_k)_{k \in \{1, \dots, N_c\}}$, we write

$$\frac{1}{N_c} \sum_{k=1}^{N_c} |u_{\vartheta}(x_k) - u_{\theta^n}(\mathcal{X}(t^n; t^{n+1}, x_k))|^2 \simeq \int_{\Omega} |u_{\vartheta}(x) - u_{\theta^n}(\mathcal{X}(t^n; t^{n+1}, x))|^2 dx.$$

This approximation comes with an error in $\mathcal{O}(1/\sqrt{N_c})$.

2. A time-stepping algorithm (e.g. a Runge-Kutta method) is employed to solve the ODE (2.4) describing the characteristic curves. This leads to an approximate characteristic curve, which is denoted by

$$\tilde{\mathcal{X}} \approx \mathcal{X}.$$

Note that $\tilde{\mathcal{X}} = \mathcal{X}$ as soon as the ODE has a closed-form solution. Because of the Monte-Carlo integration, this characteristic solver is applied to integration points that are randomly sampled. When the velocity field has an analytic expression, this part is fully vectorizable and easily parallelizable by assigning subsets of points to different processes as needed. If it does not have an analytic expression (e.g. given by another differential equation, see [Section 4.6](#)), it has to be computable at any point in the domain to benefit from the vectorization and parallelization. For instance, it could be given by a spline or a neural network. In practice, solving the ODE represents a negligible part of the overall computation cost.

3. The optimization problem is solved using a stochastic gradient descent method. This is done by computing the gradient of the objective function with respect to θ . This gradient is computed using the chain rule, and it requires computing the gradient of u_θ with respect to θ , which is done using automatic differentiation.

We now summarize the Neural Semi-Lagrangian (NSL) method in the following algorithm, in the case of a parametric advection equation. It is detailed in [Algorithm 1](#), while the computation of the approximate characteristic curve is more classical, and recalled in [Algorithm 3](#) for completeness.

Algorithm 1 Neural Semi-Lagrangian (NSL) method for advection equations

- 1: **Input:** Initial condition $u_0(x, \mu)$, time discretization $(t^n)_n$, neural network architecture, final time T , number of collocation points N_c
- 2: **Output:** Approximate solution $u_N(t^n, x, \mu)$ for all t^n , all $x \in \Omega$, and all $\mu \in \mathbb{M}$
- 3: **Initialization:** Compute the initial dofs θ^0 by randomly sampling collocation points $(x_k)_k \in \Omega^{N_c}$ and collocation parameters $(\mu_k)_k \in \mathbb{M}^{N_c}$ and solving

$$\theta^0 \in \arg \min_{\vartheta \in \Theta} \sum_{k=1}^{N_c} |u_\vartheta(x_k, \mu_k) - u_0(x_k, \mu_k)|^2$$

- 4: **while** $t < T$ **do**
- 5: Solve the nonlinear optimization problem

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \sum_{k=1}^{N_c} |u_\vartheta(x_k, \mu_k) - u_{\theta^n}(\tilde{\mathcal{X}}(t^n; t^{n+1}, x_k, \mu_k))|^2$$

- 6: **for** each iteration of the nonlinear optimization algorithm **do**
 - 7: Randomly sample collocation points $(x_k)_k \in \Omega^{N_c}$ and collocation parameters $(\mu_k)_k \in \mathbb{M}^{N_c}$
 - 8: Compute the approximate foot $\tilde{\mathcal{X}}(t^n; t^{n+1}, x_k, \mu_k)$ of the characteristic curve using [Algorithm 3](#)
 - 9: **end for**
 - 10: Update the time step: $t \leftarrow t + \Delta t$
 - 11: **end while**
 - 12: **Return:** $u_N(t, x, \mu) = u_{\theta(t)}(x)$
-

Remark 1. [Algorithm 1](#) corresponds to the backwards semi-Lagrangian method, where the characteristic curves are integrated backwards in time. While the backwards method is well-known within the realm of traditional numerical methods, the forward version of the semi-Lagrangian method also exists, see [\[21\]](#). It consists in integrating the characteristic curves forwards in time rather than backwards. It would also be possible to develop a forwards version of the present method, simply by replacing [\(3.3\)](#) with

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \int_{\Omega} |u_\vartheta(\mathcal{X}^{fwd}(t^{n+1}; t^n, x)) - u_{\theta^n}(x)|^2 dx,$$

where $\mathcal{X}^{fwd}$ is the (forwards) characteristic curve originating in point x at time t^n , and $\mathcal{X}^{fwd}(t^{n+1}; t^n, x)$ is the position of the head at time t^{n+1} . Both forwards and backwards versions of the present method should perform the same, and we focus on the backwards version to simplify the presentation.

3.3. Rough error estimates

In this section, we present a rough estimate of the error made by the NSL method in order to decompose it into several contributions: integration, optimization, approximation and characteristic errors. Indeed, at each iteration, a projection problem onto the neural network function space is solved. The error thus depends on the approximation capacity of neural networks (approximation error) as well as the error at the previous step. Other sources of error come from the fact that the projection problem is actually modified: the integral is replaced by a Monte Carlo estimate (integration error) and the characteristic curves are solved numerically if they are not explicitly known (characteristic error). Finally, the optimization problem is only solved approximately (optimization error). Errors then accumulate over the course of iterations. We do not expect the resulting estimate to give practical control over the error; rather, we aim at a better understanding of the error sources.

We do not consider the dependence in the parameters μ , for simplicity. Moreover, we consider the pure transport case, i.e., we take $\sigma = 0$ in (1.1). We seek to estimate

$$\mathcal{E}^{n+1} = \|u_{\theta^{n+1}} - u(t^{n+1}, \cdot)\|_{L^2(\Omega)},$$

where u is the exact PDE solution, and where $u_{\theta^{n+1}}$ is the approximate solution at time t^{n+1} . It is obtained by solving, in an approximate manner, the nonlinear optimization problem from Algorithm 1, which reads

$$\arg \min_{u \in \mathcal{V}_N^{\text{neural}, t^{n+1}}} \frac{1}{N_c} \sum_{k=1}^{N_c} |u(x_k) - u_{\theta^n}(\tilde{\mathcal{X}}(t^n; t^{n+1}, x_k))|^2, \quad (3.4)$$

where $\mathcal{V}_N^{\text{neural}, t^{n+1}} \subset L^2(\Omega)$ is the nonlinear approximation space, whose solution is denoted $u_{\theta_*^{n+1}}$. Let us emphasize that, up to an optimization error to be discussed later, we have

$$u_{\theta^{n+1}} \approx u_{\theta_*^{n+1}},$$

since $u_{\theta^{n+1}}$ is an approximate minimizer of (3.4), while $u_{\theta_*^{n+1}}$ is the true minimizer. Further, note that

$$u^0 \in \arg \min_{u \in \mathcal{V}_N^{\text{neural}, 0}} \frac{1}{N_c} \sum_{k=1}^{N_c} |u(x_k) - u_0(x_k)|^2.$$

To bound the error $\mathcal{E}^{n+1}$, we define three quantities, respectively corresponding to the integration, optimization and approximation errors. In these definitions, $(x_k)_{k \in \{1, \dots, N_c\}} \in \Omega^{N_c}$ form a set of N_c collocation points. For any f in V , the *integration error* is defined by

$$\varepsilon_{\text{int}}(f; (x_k)_k) = \left| \|f\|_{L^2(\Omega)} - \frac{1}{N_c} \sum_{k=1}^{N_c} |f(x_k)|^2 \right|. \quad (3.5)$$

This is nothing but the error made when approximating the integral of f using the Monte-Carlo method. We next define the (discrete) *optimization error* between the exact minimizer f_N^* and the approximate minimizer f_N of the same optimization problem:

$$\varepsilon_{\text{opt}}(f_N, f_N^*; (x_k)_k) = \frac{1}{N_c} \sum_{k=1}^{N_c} |f_N(x_k) - f_N^*(x_k)|^2. \quad (3.6)$$

This measures how close the approximate minimizer is to the exact minimizer of the optimization problem, using a Monte-Carlo approximation. The *approximation error* is defined by

$$\varepsilon_{\text{approx}}(f_N, f) = \|f_N - f\|_{L^2(\Omega)}. \quad (3.7)$$

This measures how well a function in the nonlinear subspace $\mathcal{V}_N^{\text{neural}, t}$ approximates a given function in the infinite-dimensional space V . It is expected to decrease as the number N of dofs increases.

Equipped with these definitions, we are almost ready to state the main result of this section. Before that, we state and prove the following lemma.

Lemma 2. Let $n \geq 0$, and define the function $\mathcal{X}^{n+1} : x \mapsto \mathcal{X}(t^n; t^{n+1}, x)$. Then, for all $f \in V$,

$$\|f \circ \mathcal{X}^{n+1}\|_{L^2(\Omega)} \leq \frac{1}{d_{\mathcal{X}}^{n+1}} \|f\|_{L^2(\Omega)}, \quad \text{with } d_{\mathcal{X}}^{n+1} = \sqrt{\inf_{y \in \Omega} |\det(D\mathcal{X}^{n+1})(y)|},$$

Proof. The function $\mathcal{X}^{n+1}$ is a diffeomorphism from Ω to itself, since it is the flow of the ODE (2.4), see e.g. [39]. Therefore, we have

$$\int_{\Omega} f(\mathcal{X}^{n+1}(x))^2 dx = \int_{\mathcal{X}^{n+1}(\Omega)} f(y)^2 |\det(D\mathcal{X}^{n+1})^{-1}(y)| dy = \int_{\Omega} f(y)^2 \frac{dy}{|\det(D\mathcal{X}^{n+1})(y)|}.$$

Introducing $d_{\mathcal{X}}^{n+1}$ and taking the square root of the above equation gives the desired result. $\square$

Proposition 3. Let u be the exact solution of the transport equation (i.e., equation (1.1) with $\sigma = 0$) on a bounded domain Ω , and let u_{θ^n} be the approximate solution at time t^n . Let $(x_k)_{k \in \{1, \dots, N_c\}} \in \Omega^{N_c}$ be a set of N_c collocation points. Let p and n_{τ} be, respectively, the order and the number of sub-time steps of the characteristic curve solver. We have, for all n such that $t^n < T$,

$$\|u_{\theta^n} - u(t^n, \cdot)\|_{L^2(\Omega)} \leq \sum_{n=0}^n (\varepsilon_{int}^n + \varepsilon_{opt}^n + \varepsilon_{approx}^n) \mathcal{D}^n + \left(\sum_{n=1}^n \mathcal{C}^n \mathcal{D}^n \right) \left(\frac{\Delta t}{n_{\tau}} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_{\tau}} \right)^{2p+2} \right), \quad (3.8)$$

which involves the error terms at each iteration

$$\begin{aligned} \varepsilon_{int}^n &= \varepsilon_{int}(u_{\theta^n} - u_{\theta_*^n}; (x_k)_k), & \varepsilon_{approx}^n &= \begin{cases} \varepsilon_{approx}(u_{\theta_*^n}, u_{\theta^{n-1}} \circ \tilde{\mathcal{X}}(t^{n-1}; t^n, \cdot)) & \text{if } n > 0, \\ \varepsilon_{approx}(u_{\theta_*^n}, u_0) & \text{otherwise,} \end{cases} \\ \varepsilon_{opt}^n &= \varepsilon_{opt}(u_{\theta^n}, u_{\theta_*^n}; (x_k)_k), \end{aligned}$$

where the errors are defined by (3.5)-(3.6)-(3.7) and the terms

$$\mathcal{C}^n = \frac{M_p}{d_{\mathcal{X}}^n} \|(\mathcal{X}^n)^{(p+1)}\|_{L^\infty(\Omega)}, \quad \text{and} \quad \mathcal{D}^n = \prod_{m=n+1}^n \frac{1}{d_{\mathcal{X}}^m},$$

where M_p is a constant associated with the characteristic curve solver and $d_{\mathcal{X}}^n$ is defined in Lemma 2.

Proof. We wish to bound the error $\mathcal{E}^{n+1} = \|u_{\theta^{n+1}} - u(t^{n+1}, \cdot)\|_{L^2(\Omega)}$. We first note that, since u is the exact solution of the advection equation, with characteristic curve $\mathcal{X}$,

$$\mathcal{E}^{n+1} = \|u_{\theta^{n+1}} - u(t^{n+1}, \cdot)\|_{L^2(\Omega)} = \|u_{\theta^{n+1}} - u(t^n, \cdot) \circ \mathcal{X}^{n+1}\|_{L^2(\Omega)},$$

where we have defined $\mathcal{X}^{n+1} : x \mapsto \mathcal{X}(t^n; t^{n+1}, x)$. Now, we split the error into a telescopic sum and obtain:

$$\begin{aligned} \mathcal{E}^{n+1} &\leq \|u_{\theta^{n+1}} - u_{\theta_*^{n+1}}\|_{L^2(\Omega)} \\ &\quad + \|u_{\theta_*^{n+1}} - u_{\theta^n} \circ \tilde{\mathcal{X}}^{n+1}\|_{L^2(\Omega)} \\ &\quad + \|u_{\theta^n} \circ \tilde{\mathcal{X}}^{n+1} - u_{\theta^n} \circ \mathcal{X}^{n+1}\|_{L^2(\Omega)} \\ &\quad + \|u_{\theta^n} \circ \mathcal{X}^{n+1} - u(t^n, \cdot) \circ \mathcal{X}^{n+1}\|_{L^2(\Omega)} = \alpha + \beta + \gamma + \delta, \end{aligned}$$

with $\alpha, \beta, \gamma, \delta$ respectively referring to the four terms of the upper bound. We now bound each of them separately.

First, with a triangle inequality, α satisfies

$$\alpha \leq \left\| \|u_{\theta^{n+1}} - u_{\theta_*^{n+1}}\|_{L^2(\Omega)} - \frac{1}{N_c} \sum_{k=1}^{N_c} |u_{\theta^{n+1}}(x_k) - u^{n+1}(x_k)|^2 \right\| + \frac{1}{N_c} \sum_{k=1}^{N_c} |u_{\theta^{n+1}}(x_k) - u^{n+1}(x_k)|^2.$$

Thus, α is directly bounded by the integration and optimization errors, and we obtain

$$\alpha \leq \varepsilon_{int}^{n+1} + \varepsilon_{opt}^{n+1}.$$

Furthermore, we directly note that $\beta = \varepsilon_{approx}^{n+1}$.

Then, the term γ corresponds to the error made when approximating the characteristic curve. The local truncation error of the numerical solver writes:

$$\tilde{\mathcal{X}}(t^n; t^{n+1}, x) = \mathcal{X}(t^n; t^{n+1}, x) + M_p \mathcal{X}^{(p+1)}(t^n; t^{n+1}, x) \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right)$$

where M_p is a positive real constant and $(\mathcal{X}^{n+1})^{(p+1)}$ is the $(p+1)$ -th derivative of the function $t \mapsto \mathcal{X}(t; t^{n+1}, x)$. Thus, performing a Taylor expansion, we have

$$\begin{aligned} \gamma &= \left\| u_{\theta^n} \circ \tilde{\mathcal{X}}^{n+1} - u_{\theta^n} \circ \mathcal{X}^{n+1} \right\|_{L^2(\Omega)} \\ &= \left\| u_{\theta^n} \left(\mathcal{X}^{n+1} + M_p \mathcal{X}^{(p+1)}(t^n; t^{n+1}, \cdot) \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right) \right) - u_{\theta^n} \circ \mathcal{X}^{n+1} \right\|_{L^2(\Omega)} \\ &\leq \left\| \left(\nabla u_{\theta^n} \circ \mathcal{X}^{n+1} \right) M_p \mathcal{X}^{(p+1)}(t^n; t^{n+1}, \cdot) \left(\frac{\Delta t}{n_\tau} \right)^{p+1} \right\|_{L^2(\Omega)} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right) \\ &\leq \left\| \nabla u_{\theta^n} \circ \mathcal{X}^{n+1} \right\|_{L^2(\Omega)} M_p \left\| \mathcal{X}^{(p+1)}(t^n; t^{n+1}, \cdot) \right\|_{L^\infty(\Omega)} \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right), \end{aligned}$$

where the higher order terms are lumped into the last term. Arguing [Lemma 2](#), we further bound the gradient norm:

$$\left\| \nabla u_{\theta^n} \circ \mathcal{X}^{n+1} \right\|_{L^2(\Omega)} \leq \frac{1}{d_{\mathcal{X}}^{n+1}} \left\| \nabla u_{\theta^n} \right\|_{L^2(\Omega)}.$$

All in all, we obtain the bound

$$\gamma \leq \frac{M_p}{d_{\mathcal{X}}^{n+1}} \left\| (\mathcal{X}^{n+1})^{(p+1)} \right\|_{L^\infty(\Omega)} \left\| \nabla u_{\theta^n} \right\|_{L^2(\Omega)} \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right).$$

Lastly, using [Lemma 2](#) on δ , we obtain

$$\delta = \left\| u_{\theta^n} \circ \mathcal{X}^{n+1} - u(t^n, \cdot) \circ \mathcal{X}^{n+1} \right\|_{L^2(\Omega)} \leq \frac{1}{d_{\mathcal{X}}^{n+1}} \left\| u_{\theta^n} - u(t^n, \cdot) \right\|_{L^2(\Omega)} = \frac{1}{d_{\mathcal{X}}^{n+1}} \mathcal{E}^n.$$

Putting everything together, we obtain

$$\mathcal{E}^{n+1} \leq \varepsilon_{\text{int}}^{n+1} + \varepsilon_{\text{opt}}^{n+1} + \varepsilon_{\text{approx}}^{n+1} + \mathcal{C}^{n+1} \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right) + \frac{\mathcal{E}^n}{d_{\mathcal{X}}^{n+1}},$$

where we have defined

$$\mathcal{C}^{n+1} = \frac{M_p}{d_{\mathcal{X}}^{n+1}} \left\| (\mathcal{X}^{n+1})^{(p+1)} \right\|_{L^\infty(\Omega)} \left\| \nabla u_{\theta^n} \right\|_{L^2(\Omega)}.$$

Using the discrete version of Grönwall's lemma, we obtain

$$\mathcal{E}^n \leq \sum_{n=1}^n \left(\varepsilon_{\text{int}}^n + \varepsilon_{\text{opt}}^n + \varepsilon_{\text{approx}}^n + \mathcal{C}^n \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right) \right) \mathcal{D}^n + \mathcal{D}^1 \mathcal{E}^0,$$

where we have set

$$\mathcal{D}^n = \prod_{m=n+1}^n \frac{1}{d_{\mathcal{X}}^m}.$$

Noting that $\mathcal{E}^0 \leq \varepsilon_{\text{int}}^0 + \varepsilon_{\text{opt}}^0 + \varepsilon_{\text{approx}}^0$, we finally obtain

$$\mathcal{E}^n \leq \sum_{n=0}^n (\varepsilon_{\text{int}}^n + \varepsilon_{\text{opt}}^n + \varepsilon_{\text{approx}}^n) \mathcal{D}^n + \left(\sum_{n=1}^n \mathcal{C}^n \mathcal{D}^n \right) \left(\frac{\Delta t}{n_\tau} \right)^{p+1} + \mathcal{O} \left(\left(\frac{\Delta t}{n_\tau} \right)^{2p+2} \right),$$

which is the desired result. $\square$

Corollary 4. *Under the hypotheses of Proposition 3, assume that the velocity field a is divergence-free. Let $C = \max_{n \in \{1, \dots, n\}} C^n$. Then*

$$\mathcal{E}^{n+1} \leq \sum_{n=0}^n \varepsilon_{int}^n + \sum_{n=0}^n \varepsilon_{opt}^n + \sum_{n=0}^n \varepsilon_{approx}^n + \frac{CT}{n_\tau} \left(\frac{\Delta t}{n_\tau} \right)^p.$$

Proof. If the velocity field a is divergence-free, then invoking Liouville's theorem, the determinant of the Jacobian matrix of the characteristic curve is constant and equal to one. Therefore, for all n , $\mathcal{D}^n = 1$. Plugging this into Proposition 3 gives the desired result. $\square$

Compared to classical error estimates, the interpolation error in the semi-Lagrangian schemes (see e.g. [29, 13, 5, 31]) or the projection error in the semi-Lagrangian discontinuous Galerkin schemes (see e.g. [67, 68, 63]) is replaced with the approximation error, and we have additional optimization and integration errors (which are already present in semi-Lagrangian Galerkin methods). The characteristic error, depending on the accuracy of the characteristic curve solver, is also present in classical schemes. Let us discuss these four errors in more detail.

- The integration error depends on the chosen quadrature method. In our case, since we choose the Monte-Carlo method, it is bounded by $1/\sqrt{N_c}$, where N_c is the number of collocation points, up to a constant independent of N_c . Thus, this error can be made as small as desired. In addition, we can further simplify (3.8) by remarking that

$$\sum_{n=1}^n \varepsilon_{int}^n \leq \frac{C_{int}T}{\Delta t \sqrt{N_c}},$$

where C_{int} is a constant independent of Δt and N_c .

- The optimization error is the threshold at which the optimization algorithm is stopped. Depending on the algorithm used, this threshold may or may not be computationally reachable. Moreover, it may strongly depend on time since the optimization problem may be harder to solve at some time steps than at others.
- The approximation error depends only on the expressiveness of the approximation space $u_{\theta_{*+1}} \in \mathcal{V}_N^{\text{neural}, t}$. To decrease this error, we must increase the number of dofs (i.e., the number of neurons in the neural network), but no generic quantitative estimates are available for the moment. Note also that, if the approximation error is supposed to be bounded by ε_N , then the corresponding error satisfies

$$\sum_{n=1}^n \varepsilon_{approx}^n \leq \frac{T\varepsilon_N}{\Delta t}.$$

In classical semi-Lagrangian schemes, ε_N is generally replaced with an approximation error of the form $\mathcal{O}(\Delta x^{q+1})$, where q is the spatial order of approximation. Here, on the contrary, we might expect that ε_N is very small compared with Δt .

- The characteristic error directly depends on Δt and on the characteristic curve solver. As such, it is the easiest to control, either by improving the order of the solver, or by adding more sub-time steps. Let us not that, as soon as an exact expression of the characteristic curve is known, M_p is equal to zero, and so C_n vanishes for all $n \geq 0$, leading to the whole characteristic error vanishing, as expected.

All in all, supposing that these three errors can be controlled, they must be at least in $O(\Delta t^2)$ to ensure the convergence of the method. In practice, we do not have fine control over these three errors. If these errors become large, larger time steps are recommended to limit the error accumulating at each time step. Note, however, that it would also lead to more difficult optimization problems. This is expanded upon in Remark 5.

Remark 5. *Proposition 3 might suggest that, in the case of pure advection, more time steps would lead to errors accumulating, and therefore one should take a time step as large as possible. But things are not that simple: the most computationally expensive step of our algorithm is to find a ϑ that allows u_ϑ (the target) to best fit $u_{\theta^n}(\mathcal{X}(t^n; t^{n+1}, \cdot))$ (the source). To achieve this, we perform a gradient descent starting from $\vartheta = \theta^n$. When the time step $\Delta t = t^{n+1} - t^n$ is very small, the characteristic curve $\mathcal{X}(t^n; t^{n+1}, \cdot)$ is very short, so the source and the target are very close: this implies that the gradient descent is fast and leads to a very good fit. Conversely, if we take an extremely large Δt , the source and target are very far apart, and the minimization problem then becomes difficult. For problems where the complexity of the target solution increases with time*

(such as the Vlasov equation, which leads to filamentation), the fitting between a “simple” source and a “very complex” target could be technically very challenging. The choice of the time step is therefore necessarily a compromise. In the case of advection-diffusion, as explained in the next section, the error on the diffusion scales as $\sqrt{\Delta t}$. Therefore, it outweighs the fitting errors of the neural network, and the time step should be taken as small as possible.

3.4. Extension to advection-diffusion equations

Equipped with the NSL method applied to advection equations with a space-, time- and parameter-dependent advection field, we extend it to the full advection-diffusion equation (1.1). In this context, recall that σ represents a constant diffusion coefficient.

To perform this extension, it turns out that it is sufficient to modify the computation of the foot of the characteristic curves. Indeed, after e.g. [30], the advection-diffusion equation can be reformulated as a stochastic differential equation (SDE) with a Gaussian noise, whose variance depends on the dimension d , the time step Δt , and the diffusion coefficient σ . To discretize this SDE, an appropriate discretization of the Gaussian noise must be provided, see e.g. [37, 10]. For instance, on a 2D Cartesian mesh, it is shown in [9] that 4 directions are required for a scheme of order 1 and 9 directions for a scheme of order 2. The first-order scheme is easily extended to higher dimensions, and thus requires $2d$ directions in dimension d .

In this work, we follow the same approach, and define the $2d$ diffusion directions by

$$\forall i \in \{1, \dots, d\}, \quad v_i = e_i \quad \text{and} \quad v_{d+i} = -e_i,$$

where for all $i \in \{1, \dots, d\}$, e_i is the i^{th} vector of the canonical basis of $\mathbb{R}^d$.

Equipped with the $2d$ directions, we can now compute the foot of the characteristic curves in the presence of a constant diffusion, with a scheme of time order 1. In this case, the nonlinear optimization problem in step 5 of Algorithm 1 is replaced with

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \sum_{k=1}^{N_c} \left| u_{\vartheta}(x_k, \mu_k) - \sum_{i=1}^{2d} u_{\theta^n}(\tilde{\mathcal{X}}_i(t^n; t^{n+1}, x_k, \mu_k)) \right|^2.$$

Compared to the pure advection case, this problem contains $2d$ characteristic feet $(\tilde{\mathcal{X}}_i)_{i \in \{1, \dots, 2d\}}$ to be computed instead of one. They are calculated, using the $2d$ directions, as follows:

$$\forall i \in \{1, \dots, 2d\}, \quad \tilde{\mathcal{X}}_i(t^n; t^{n+1}, x, \mu) = \tilde{\mathcal{X}}(t^n; t^{n+1}, x, \mu) + \sqrt{2d\sigma\Delta t} v_i,$$

where $\tilde{\mathcal{X}}(t^n; t^{n+1}, x)$ is the result of Algorithm 3, i.e., the approximate foot of the characteristic curve in the presence of advection only.

We remark that our approach is compatible with any already-developed strategy to improve the approximation of diffusion in stochastic differential equations or semi-Lagrangian schemes. For instance, using higher-order spherical quadrature rules could provide finer and more uniform angular coverage than the canonical basis of $\mathbb{R}^d$, see e.g. [74]. Similarly, adaptive strategies in which directions are dynamically aligned with the local eigenspectrum of the diffusion tensor could better capture anisotropic effects, see e.g. [8]. These variants could be integrated into our method without altering its general structure.

3.5. Further improvements

In this section, motivated by the error estimate from Proposition 3, we present several techniques to further improve the results of the NSL method. Namely, we first focus on improving the resolution of the optimization problem (3.3) in Section 3.5.1 via natural gradient preconditioning, to decrease the optimization error ε_{opt} . Then, we introduce an adaptive sampling strategy in Section 3.5.2 that, while not directly related to the NSL method, allows us to improve the quality of the solution by improving the Monte-Carlo approximation of the integral in (3.3), thus decreasing the integration error ε_{int} . Finally, we briefly mention in Section 3.5.3 how to tackle boundary conditions.

3.5.1. Natural gradient preconditioning

To properly introduce natural gradient descent, we go back to the optimization problem (3.3) and rewrite it, dropping the time indices for simplicity, as

$$\theta \in \arg \min_{\vartheta \in \Theta} \int_{\Omega} |\mathcal{N}(x, \vartheta) - u^*(x)|^2 dx. \quad (3.9)$$

In (3.9), we have reused the nonlinear parametric function $\mathcal{N}$ defined in (2.5), typically a neural network, and we have introduced the function u^* to concisely represent the approximate foot of the characteristic curve. It should be noted that (3.9) is nothing but a nonlinear least-squares problem with objective function u^* .

Natural gradient descent was introduced in [1]. Now, we present the method, alongside its main advantages over standard gradient descent, and why it is particularly well-suited to solving problems such as (3.9).

To solve (3.9), one usually constructs a sequence of iterates $(\vartheta^n)_n$, which converges to the solution θ . These iterates are updated using the gradient descent method¹. This method amounts to finding a so-called descent direction η , and writing $(\vartheta^n)_n$ as the time discretization of the ordinary differential equation $\dot{\vartheta} = \eta$. To find the descent direction, it is convenient to rewrite the optimization problem (3.9) as

$$\theta \in \arg \min_{\vartheta \in \Theta} \mathfrak{l}(\mathbf{n}(\vartheta)),$$

with $\mathfrak{l}$ the loss function depending on the neural network $\mathbf{n}$, which itself depends on the dofs ϑ . More precisely, we have

$$\begin{aligned} \mathfrak{l}: \mathcal{H} &\rightarrow \mathbb{R} & \mathbf{n}: \mathbb{R}^N &\rightarrow \mathcal{H} \\ f &\mapsto \int_{\Omega} |f(x) - u^*(x)|^2 dx & \text{and} & \vartheta \mapsto (x \mapsto \mathcal{N}(x, \vartheta)). \end{aligned} \quad (3.10)$$

In (3.10), similarly to (3.4), we have defined the finite-dimensional nonlinear function space

$$\mathcal{H} = \{x \in \Omega \mapsto \mathcal{N}(x, \vartheta) \in \mathbb{R}, \vartheta \in \Theta\} \subset L^2(\Omega).$$

Equipped with this notation, the gradient descent (GD) methods corresponds to the steepest descent in terms of the vector ϑ , i.e., the descent direction is given by

$$\eta^{\text{GD}} = -\nabla_{\vartheta} \mathfrak{l}(\mathbf{n}(\vartheta)),$$

where $\nabla_{\vartheta} \mathfrak{l}(\mathbf{n}(\vartheta))$ is the gradient of $\vartheta \mapsto \mathfrak{l}(\mathbf{n}(\vartheta))$. This disregards the fact that $\mathcal{H}$ is a nonlinear space.

Conversely, natural gradient descent (NGD) corresponds to an approximation of the steepest descent direction in terms of the nonlinear function $\mathbf{n}$. In the general case, one needs to consider the tangent space of the manifold $\mathcal{H}$ at the point $\mathbf{n}(\vartheta)$. However, in our specific case, the system under consideration is the nonlinear least-squares problem (3.9). Thus, NGD amounts to modifying the gradient descent direction to

$$\eta^{\text{NGD}} = G(\vartheta)^{\dagger} \eta^{\text{GD}},$$

where $G(\vartheta)$ is the so-called *Fisher information matrix*, with $G(\vartheta)^{\dagger}$ its pseudo-inverse. Note that this matrix is identical to the Neural Galerkin mass matrix, defined in (2.10). The (i, j) coefficient of this matrix is defined by

$$G_{i,j}(\vartheta) = \langle \partial_{\vartheta_i} \mathbf{n}(\vartheta), \partial_{\vartheta_j} \mathbf{n}(\vartheta) \rangle_{L^2(\Omega)} = \int_{\Omega} \partial_{\vartheta_i} \mathcal{N}(x, \vartheta) \partial_{\vartheta_j} \mathcal{N}(x, \vartheta) dx.$$

In the end, NGD amounts to preconditioning the gradient descent direction using the gradient of the neural network $\mathcal{N}$ with respect to its dofs.

More sophisticated methods have been proposed to further improve NGD, most notably in the context of physics-informed learning, see for instance [60, 59, 69]. In all of these cases, information from the loss function is used to improve the convergence of the optimization problem. In physics-informed learning, the loss function involves the residual of the PDE, like (2.6) for PINNs. As a result, the preconditioner becomes much more complicated to compute, since derivatives with respect to ϑ of the space and time derivatives of the residual become involved. One major advantage of the proposed method is that the optimization problem (3.3) is nothing but a nonlinear least-squares problem, for which classical natural gradient preconditioning can directly be applied with minimal implementation effort and computational cost.

¹Usually, a stochastic version of the gradient method is used; the deterministic version is presented here for simplicity.

3.5.2. Adaptive sampling

We now introduce a technique to decrease the integration error ε_{int} . Namely, rather than uniformly sampling the collocation points in step 7 of [Algorithm 1](#), we propose an adaptive sampling strategy, in the framework of rejection sampling. For simplicity, we assume that we wish to sample additional points in the vicinity of the zeros of some function $f : \Omega \rightarrow \mathbb{R}$. This is motivated by e.g. the transport of level-set functions, or the approximation of localized functions (where f could be the inverse of the norm of the gradient of the function to be approximated).

[Algorithm 2](#) details the adaptive sampling strategy. It relies on three hyperparameters σ_1 , σ_2 and σ_3 , whose specific choice turns out not to be crucial. The algorithm performs three iterations, each corresponding to a different value of the parameter $\sigma \in \{\sigma_1, \sigma_2, \sigma_3\}$. In each iteration, it uniformly samples $10N_c$ candidate points from the domain $\Omega \times \mathbb{M}$, evaluates the function f at those points, and computes weights ω_k that reflect proximity to the zero set of f . From these, it selects up to $N_c/4$ points where $\omega_k > 0.75$, which are close to the zeros of f , and adds them to the growing set of collocation points. After the three passes, it supplements the set with up to N_c additional uniformly sampled points to ensure some coverage of the entire domain. The final output is a collection of N_c points in $\Omega \times \mathbb{M}$, strategically biased towards regions where f is close to 0.

Algorithm 2 Adaptively sampling N_c points concentrated around the zeros of f

- 1: **Input:** Number N_c of collocation points, function f , parameters $\sigma_1, \sigma_2, \sigma_3$
 - 2: **Output:** N_c points $(x_k, \mu_k)_{k \in \{1, \dots, N_c\}} \in \Omega^{N_c} \times \mathbb{M}^{N_c}$
 - 3: **for** $\sigma \in \{\sigma_1, \sigma_2, \sigma_3\}$ **do**
 - 4: Uniformly sample $10N_c$ points in $\Omega \times \mathbb{M}$, denoted by $(\bar{x}_k, \bar{\mu}_k)_{k \in \{1, \dots, 10N_c\}}$
 - 5: Compute $\omega_k = \exp(-f(\bar{x}_k, \bar{\mu}_k)^2 / (2\sigma^2))$ for all $k \in \{1, \dots, 10N_c\}$
 - 6: Add at most $N_c/4$ random points $(\bar{x}_k, \bar{\mu}_k)$ such that $\omega_k > 0.75$ to the set of collocation points (x_k, μ_k)
 - 7: **end for**
 - 8: Add at most N_c uniformly sampled points to the set of collocation points (x_k, μ_k)
 - 9: **Return:** $(x_k, \mu_k)_{k \in \{1, \dots, N_c\}}$
-

Let us emphasize that more sophisticated adaptive sampling could be used to further decrease ε_{int} . The proposed strategy was chosen for its simplicity. One idea could be to advect the collocation points using the characteristic curves. However, since the optimization problem (3.3) is a simple nonlinear least-squares problem and does not involve the residual of the PDE (compared to e.g. discrete PINNs), we would not need more complex physics-informed sampling relying on the PDE residual, see e.g. the review paper [78]. In our case, another possibility would be to use optimal least-squares sampling [17].

3.5.3. Boundary conditions

In neural methods such as PINNs, we distinguish two main approaches to handling boundary conditions.

Weak boundary conditions. In the so-called weak approach, initially proposed in [66], a loss term enforcing the boundary conditions is added to the loss function. The same strategy is applicable to our neural semi-Lagrangian framework. In (1.1), the boundary condition is given by $u = g$ on $\partial\Omega$. Then, at each step, we solve:

$$\theta^{n+1} \in \arg \min_{\vartheta \in \Theta} \int_{\Omega} |u_{\vartheta}(x) - u_{\theta^n}(\mathcal{X}(t^n; t^{n+1}, x))|^2 dx + \omega^{\text{BC}} \int_{\partial\Omega} |B(u_{\vartheta}(x)) - g(x)|^2 dx$$

instead of (3.3), where ω^{BC} is a hyperparameter that controls the weight of the boundary condition loss term. Neumann and periodic boundary conditions are handled in a similar way.

Strong boundary conditions. The strong approach, generally preferable, consists in enforcing the boundary conditions directly within the model. It was originally proposed in [49], and it is the one we used in the paper. It can be done by using a representation of the domain with a level-set function, see [73], or with eigenmodes of the Laplace operator associated with the domain, see [44]. For a Dirichlet boundary condition ($u = g$ on $\partial\Omega$), given a level-set function Φ for the domain, the boundary condition is imposed by replacing u_{θ} with $\Phi u_{\theta} + g$. This formulation clearly equals g on the boundary. Neumann boundary conditions are treated similarly; the reader is referred to [73] for a detailed breakdown. To impose periodic boundary conditions for rectangular domains in the NSL method, we simply place the foot of the characteristic curve back into the domain Ω when

it leaves through a periodic boundary. This amounts to adding a step between steps 14 and 15 of [Algorithm 3](#), replacing $\mathcal{X}$ with $x_{\min} + (\mathcal{X} - x_{\min}) \% X$, where $x_{\min}$ is the lower bound of the domain Ω , X is its extent, and $\%$ denotes the modulo operator. For example, in the case of a domain $\Omega = [0, 1] \times [-1, 1]$ with periodic boundary conditions, $x_{\min} = (0, -1)$ and $X = (1, 2)$.

4. Validation

This section is dedicated to the presentation of some numerical experiments, to validate the proposed method.

We first test the method on an advection equation with constant advection coefficient in one space dimension in [Section 4.1](#). Then, we move on to nonconstant advection coefficients in a multidimensional setting in [Section 4.2](#). In these cases, we compare the proposed Neural Semi-Lagrangian (NSL) method from [Section 3](#) to other neural methods, namely PINNs from [Section 2.2.1](#), discrete PINNs (dPINNs) from [Section 2.2.2](#), and the neural Galerkin (NG) method, also from [Section 2.2.2](#). In all these cases, we deliberately avoid showing the variability of the results with respect to the random initialization and optimization of the neural networks. Indeed, the method has a very low variability, and showing it would only clutter the figures.

Afterwards, in [Section 4.3](#), we test the several avenues of improvement proposed in [Section 3.5](#). The improved NSL scheme is then used on challenging level-set transport problems in [Section 4.4](#). Lastly, the scheme is tested on high-dimensional advection-diffusion problems in [Section 4.5](#).

Unless otherwise mentioned, all ODEs (in e.g. dPINN and NG, or in the NSL method when the foot of the characteristic curve is unknown) are solved using the Runge-Kutta 4 (RK4) method. Moreover, all nonlinear optimization problems in [Sections 4.1](#) and [4.2](#) are solved by first applying the Adam optimizer, and then switching to LBFGS for the last 10% of the epochs. Afterwards, natural gradient preconditioning is applied. The problems are solved using the `PyTorch` library [\[2\]](#) and the `ScimBa`² scientific machine learning library; to report computation times, we use a single AMD Instinct MI210 GPU.

4.1. Constant advection in 1D

The first numerical experiment we run consists in solving

$$\begin{cases} \partial_t u + a \partial_x u = 0, & x \in (0, 2), t \in (0, 1), \mu \in \mathbb{M}, \\ u(0, x, \mu) = u_0(x, \mu), & x \in (0, 2), \mu \in \mathbb{M}, \\ u(t, 0, \mu) = u(t, 2, \mu), & t \in (0, 1), \mu \in \mathbb{M}, \end{cases} \quad (4.1)$$

where we have introduced two parameters: the variance ν of the Gaussian pulse, and the advection coefficient a . The two parameters are grouped into a single vector $\mu = (\nu, a) \in \mathbb{M}$, and we set the parameter space to be $\mathbb{M} = [0.05, 0.15] \times [0.5, 1] \subset \mathbb{R}^2$. The parametric initial condition is defined by

$$u_0(x, \mu) = \exp\left(-\frac{(x - 0.5)^2}{2\nu^2}\right),$$

and the exact solution is given by $u_{\text{ex}}(t, x, \mu) = u_0(x - at, \mu)$. Since a is constant in space, the first branch of [Algorithm 3](#) is applied.

4.1.1. Non-parametric case

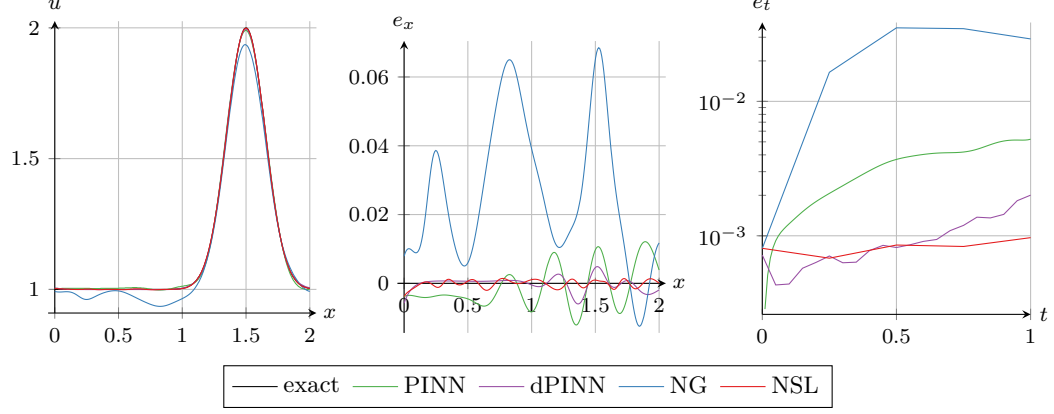
We first focus on the non-parametric case, where the advection coefficient is fixed to $a = 1$ in [\(4.1\)](#). The variance ν is also set to a constant value. We compare the PINN, dPINN, NG and NSL methods. All hyperparameters are given in [Table B.9](#). Note that the PINN solves a 2D problem (1D in space and 1D in time), while the neural network inherent in dPINN, NG and NLS solves a 1D problem in space. For comparison purposes, we denote the pointwise error by e_x and the L^2 error by e_t . They are respectively given by

$$e_x(x) = u(1, x) - u_N(1, x) \quad \text{and} \quad e_t(t) = \int_0^2 |u(t, x) - u_N(t, x)|^2 dx.$$

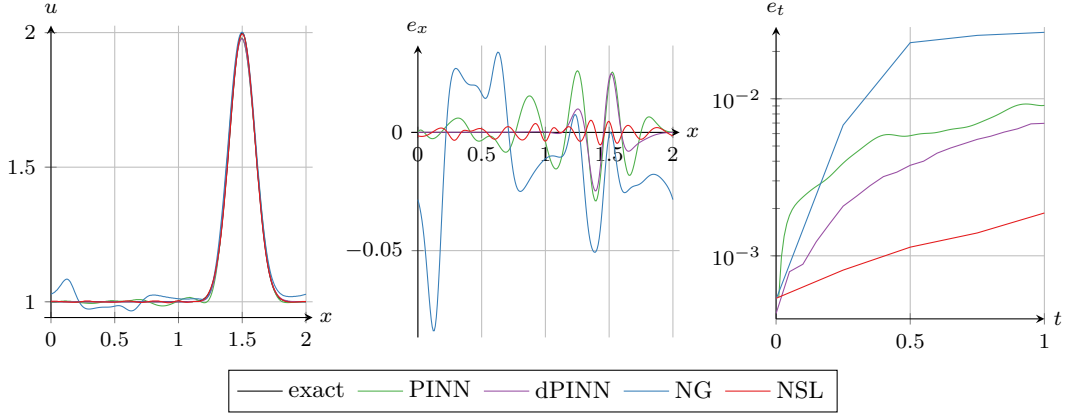
They are computed using analytic solutions and a Monte-Carlo estimate of the L^2 error using many more points than during the training process.

²<https://gitlab.inria.fr/scimba/scimba>

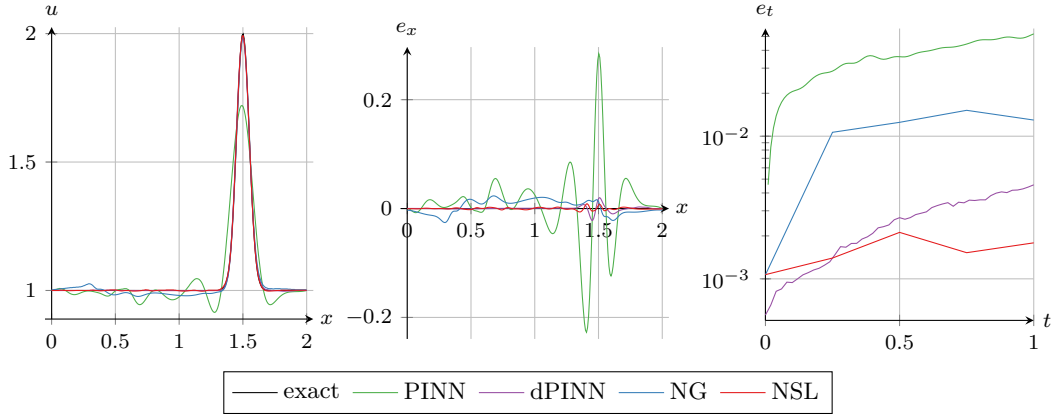
Figure 2 displays the results of the four approaches, for four distinct values of the variance v . Note that each value of the variance corresponds to a new problem to solve, with a new neural network to train. The truly parametric case is treated in the following section. For dPINN, we take $\Delta t = 0.05$ (corresponding to 20 time steps) for $\nu \geq 0.1$, and $\Delta t = 0.02$ (corresponding to 50 time steps) otherwise. For NG and NSL, we take $\Delta t = 0.25$ (corresponding to 4 time steps). We have chosen 150 epochs for the inner optimization problems (3.3) of the NSL method, since the problem converged quickly and more epochs did not bring noticeable improvements.



(a) 1D constant advection: $\nu = 0.15$.



(b) 1D constant advection: $\nu = 0.1$.



(c) 1D constant advection: $\nu = 0.05$.

Figure 2: 1D constant advection from Section 4.1.1: comparison of the different methods for several values of the variance ν , and for a fixed time step ($\Delta t = 0.05$, i.e., 20 time steps in the dPINN method; $\Delta t = 0.25$, i.e., 4 time steps in the NG and SL methods). From left to right: prediction of the solution at $t = 1$, pointwise errors between the predicted solutions and the exact solution at $t = 1$, and time evolution of the L^2 error.

We observe that the NSL method outperforms the other methods, by almost an order of magnitude in some cases. Indeed, the NSL method is able to capture both the peak of the Gaussian pulse and the constant state far away from the peak, with good accuracy.

On the contrary, the other methods tend to either diffuse the peak or create oscillations in the constant state, or both. Notably, the dPINN method required quite a restrictive time step to perform well; larger time steps led to numerical instabilities and oscillations destroying the simulation quality. Moreover, it also required better solving the optimization problems, which led to an increased number of epochs during the initialization phase and the inner epochs, compared to the NG and NSL methods. Therefore, even if the results of the dPINN method are quite good, they come at a large computational cost, at least 10 times as large as the NG and NSL methods.

To better understand the role of Δt , we display in Figure 3 the results of the methods for $\nu = 0.1$ and for several values of Δt . Namely, we show the L^2 error e_t as a function of time, for $\Delta t \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$, which corresponds to 100, 50, 20, 10, 5, and 2 time steps, respectively. For time steps $\Delta t \leq 0.05$, we perform 300 epochs for the inner optimization problems instead of 150, to help convergence. The results of the dPINN method are only shown for $\Delta t \leq 0.05$, since otherwise it does not converge.

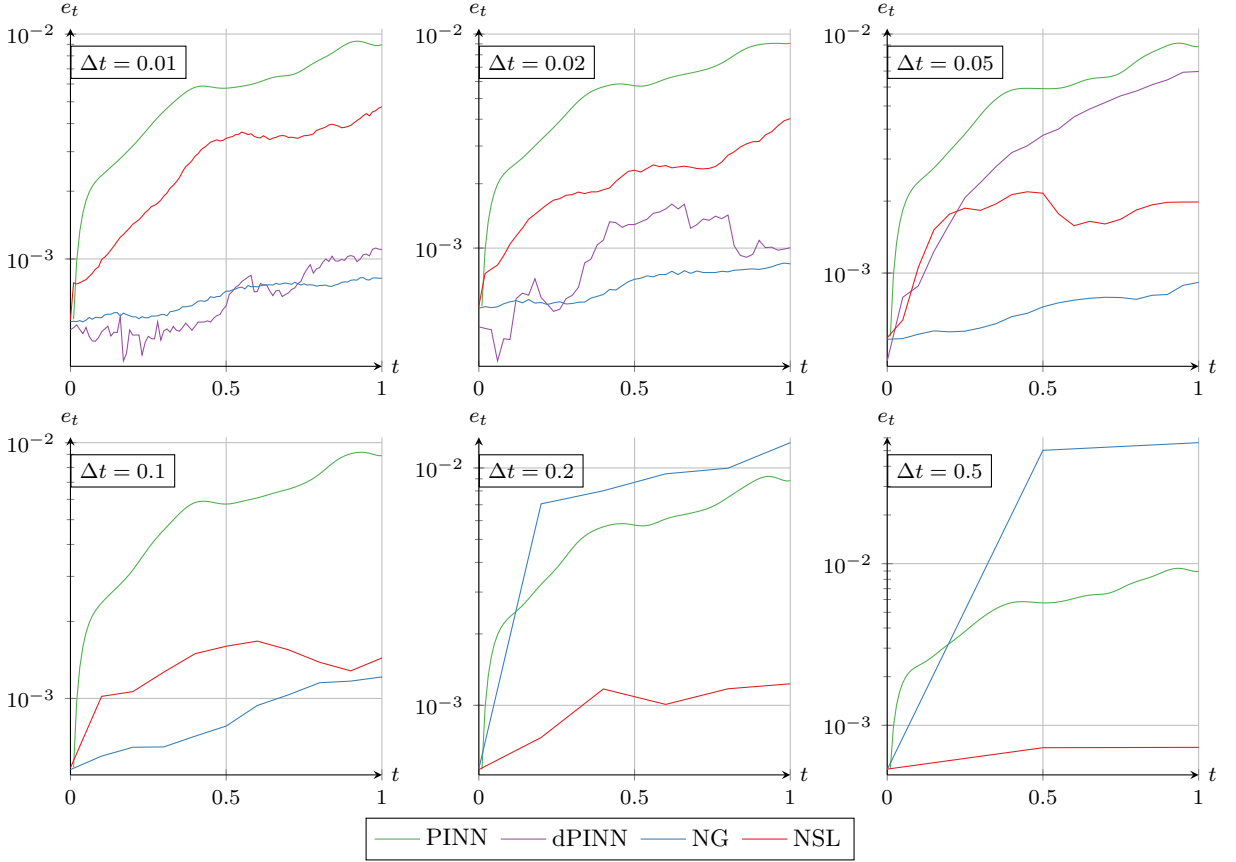


Figure 3: 1D constant advection from Section 4.1.1: comparison of the different methods for several values of the time step Δt in the NG and SL methods and a fixed variance, equal to 0.1. From top to bottom and left to right, we take $\Delta t \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$, and we display the time evolution of the L^2 error e_t .

On the one hand, for both the NG and dPINN methods, we observe that the accuracy is improved when using smaller time steps, overtaking the PINN for $\Delta t \leq 0.1$. On the other hand, interestingly, we note that smaller time steps do not necessarily lead to better results for the NSL method, at least in this simpler case. Indeed, the exact characteristic curves are known for constant advection, and so the error due to the characteristic solver vanishes. We are thus left with only the integration, optimization, and approximation errors. In this simple 1D example, the integration and approximation errors remain very low, and so the whole error is driven by the optimization one. These optimization errors will accumulate during the time iterations, because each optimization problem is not solved exactly. This naturally means that more iterations will lead to a larger total error, unless they are also accompanied by a reduction of the optimization error

at each iteration. In short, this expected behavior is observed since the optimization errors are much larger than the time errors; this was already the subject of [Remark 5](#). We expect this observation to change when considering time-dependent advection coefficients, or advection-diffusion equations, or even pure advection equations whose solutions exhibit strong space variations. Further, note that, in this case, we have considered a fixed number N_e of epochs for each time step. Since, for smaller time steps, the optimization problems are easier to solve, we could have reduced N_e and obtained almost the same results for a smaller computational cost. However, this reduction is not enough to justify taking small time steps, because it remains cheaper to solve a few hard optimization problems.

4.1.2. Parametric case

We now turn to the parametric version of (4.1), where this time the variance ν and the advection coefficient a are both parameters. The hyperparameters are defined in [Table B.10](#). Compared to the previous case, the PINN is now solving a 4D problem (1D in space, 1D in time, and 2D in the parameter space), and the dPINN, NG and NSL neural networks are solving 3D problems (1D in space and 2D in the parameter space). This makes the problem quite a bit more complex, which explains the increased number of epochs, collocation points and neurons in the hidden layers.

An added difficulty for the dPINN, NG and NSL methods is that the initial condition does not depend on a , so the network has to learn to ignore the second parameter for learning the initial condition, but to use it for the time evolution. To overcome that difficulty, on the one hand, we set a rather small time step for NG and dPINN ($\Delta t = 0.025$). On the other hand, the NSL time step is still large ($\Delta t = 0.25$), but the number of epochs in the inner optimization problems has been increased to 2500. With this setup, on the one hand, for NG and NSL, it takes 15 seconds to train the network approximating the initial condition, while the whole time stepping takes about 1 minute (meaning that NSL iterations are about 10 times slower than NG ones). This can be compared to the non-parametric case, where the NG and NSL methods took about 10 seconds to train the initial condition and 5 seconds for the time-stepping. This is due to the larger network and higher number of epochs and collocation points. On the other hand, it takes about 5 minutes to train the PINN. The dPINN method is much slower than the other ones: indeed, training and solving the optimization problems totals about 20 minutes. This is due to the larger dimension of the problem, requiring additional collocation points.

As a first test, we take three values of the parameters $\mu \in \mathbb{M}$, and we draw the solutions (top panels) and associated errors (bottom panels) in [Figure 4](#). We observe that NSL consistently yields a better approximation than the other methods; namely, it is less oscillatory.

To get a more quantitative estimation of the error with respect to the parameter set $\mathbb{M}$, we compute the L^2 norm of e_x (computed at the final time) for 2000 randomly sampled parameters in $\mathbb{M}$. The results are presented in [Table 1](#), where we report the minimum, average and maximum of the L^2 norm of e_x over $\mathbb{M}$, as well as its standard deviation. We observe that, over the whole parameter set $\mathbb{M}$, the NSL method consistently gives very good results. On average, the NSL results are quite close to the PINN. However, this comes at a much lower computational cost, as the NSL method is about 5 times faster than the PINN method.

	PINN	dPINN	NG	NSL
min	2.92×10^{-3}	4.47×10^{-3}	1.63×10^{-2}	5.03×10^{-3}
avg	9.00×10^{-3}	2.69×10^{-2}	5.31×10^{-2}	1.08×10^{-2}
max	4.29×10^{-2}	9.76×10^{-2}	5.00×10^{-1}	9.09×10^{-2}
std	5.78×10^{-3}	1.36×10^{-2}	1.14×10^{-1}	5.10×10^{-3}

Table 1: 1D parametric advection from [Section 4.1.2](#): statistics on the errors e_x for 2000 randomly sampled parameters in $\mathbb{M}$.

4.2. Nonconstant advection in 2D

We now provide some numerical experiments on advection equations with nonconstant advection coefficients, in two and three space dimensions, and with several parameters. The first test case consists in a 2D advection equation with a rotating transport, presented in [Section 4.2.1](#). The second one, presented in [Section 4.2.2](#) is a 2D advection equation, without parameters, mimicking the Vlasov equation representing the movement of charged particles in an electric field.

In this section, we do not consider the dPINN method any longer, since it is not competitive with NSL and NG for the problems at hand, especially in terms of stability condition, and to simplify the presentation.

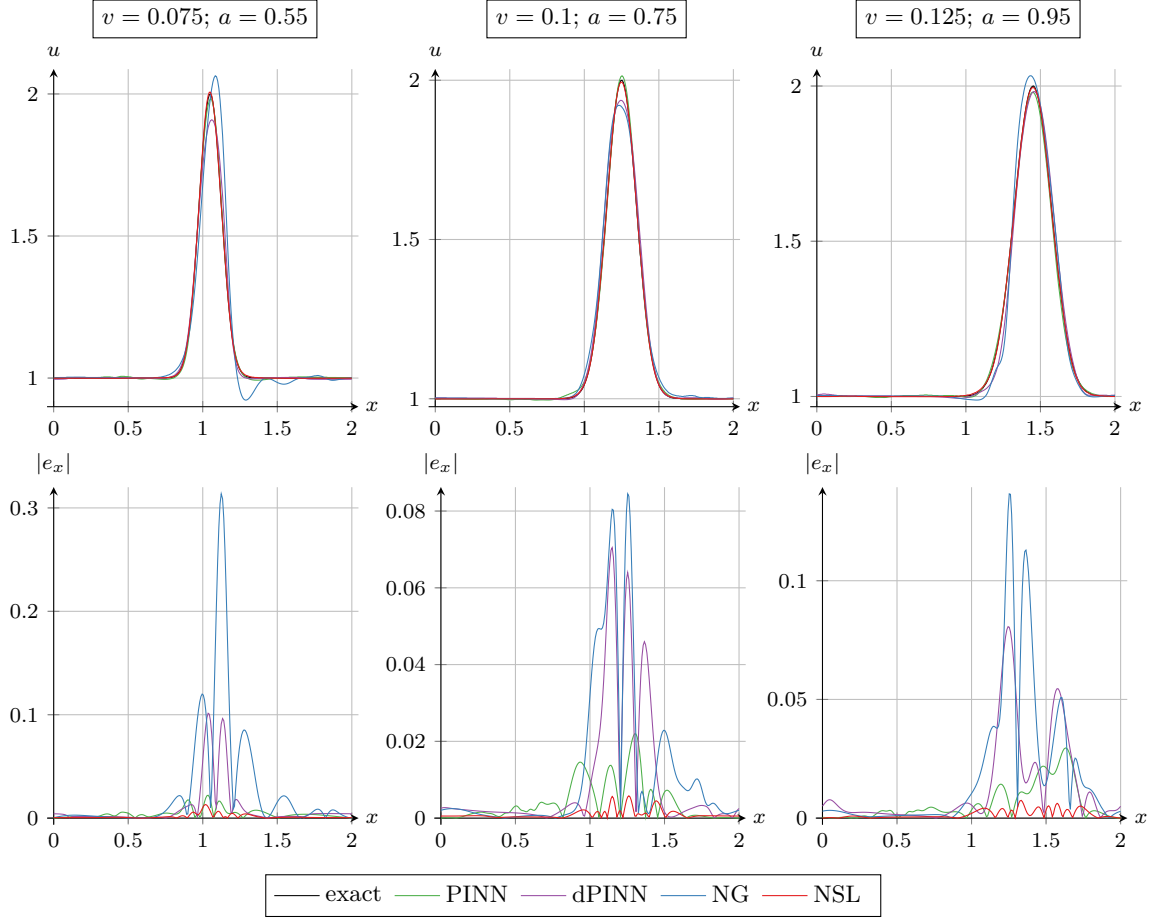


Figure 4: 1D parametric advection from Section 4.1.2: comparison of the different methods for several values of the parameters and with a fixed time step. Top panels: approximate solutions; bottom panels: absolute values of the errors. From left to right: $\mu = (0.075, 0.55)$, $\mu = (0.1, 0.75)$, and $\mu = (0.125, 0.95)$.

4.2.1. 2D parametric rotating transport

We first consider a 2D advection equation with a nonconstant advection coefficient, leading to a rotating advection equation within the unit disk $\Omega = \mathbb{D}^1$. The advection equation reads

$$\begin{cases} \partial_t u + a(x) \cdot \nabla u = 0, & x \in \Omega, t \in (0, 1), \mu \in \mathbb{M}, \\ u(0, x, \mu) = u_0(x, \mu), & x \in \Omega, \mu \in \mathbb{M}, \\ u(t, x, \mu) = 0, & x \in \partial\Omega, t \in (0, 1), \mu \in \mathbb{M}, \end{cases}$$

with the divergence-free advection vector field given by

$$a(x) = \begin{pmatrix} 2\pi x_2 \\ -2\pi x_1 \end{pmatrix}. \quad (4.2)$$

The parametric exact solution is given by

$$u_{\text{ex}}(t, x, \mu) = 1 + \exp\left(-\frac{1}{2v^2} \left((x_1 - cx_1^0(t))^2 + (x_2 - cx_2^0(t))^2\right)\right),$$

where $x_1^0(t) = \cos(2\pi t)$ and $x_2^0(t) = \sin(2\pi t)$, and with the parameters $\mu = (v, c)$, where μ is in the parameter set $\mathbb{M} = [0.05, 0.1] \times [0.2, 0.4]$. Equipped with the exact solution, the initial condition is then simply given by $u_0(x, \mu) = u_{\text{ex}}(0, x, \mu)$.

Note that the advection field (4.2) is no longer constant in space, although it still leads to an exact solution to the ODE (2.4) governing the characteristic curves. This exact solution is given by

$$\mathcal{X}_{\text{ex}}(s; t, x) = \begin{pmatrix} k_2(s; x) \cos(2\pi t) + k_1(s; x) \sin(2\pi t) \\ k_1(s; x) \cos(2\pi t) - k_2(s; x) \sin(2\pi t) \end{pmatrix},$$

where we have set

$$k_1(s; x) = x_1 \sin(2\pi s) + x_2 \cos(2\pi s) \quad \text{and} \quad k_2(s; x) = x_1 \cos(2\pi s) - x_2 \sin(2\pi s).$$

This allows us to apply the second branch of [Algorithm 3](#) when computing the characteristic curves in the NSL method.

The hyperparameters are given in [Table B.11](#). We note that, for this experiment, more collocation points are used to approximate the integrals. Indeed, the solution is a Gaussian function, highly localized in both parameter space and physical space, which requires a better Monte-Carlo approximation.

We run the PINN, NG and NSL methods on this problem. On the one hand, we set a fixed time step $\Delta t = 0.02$ for the NG method, leading to 50 time steps and $\Delta t = 0.5$ for the NSL method, leading to 2 time steps. Since the characteristic curves are known exactly, like in [Section 4.1.1](#), we have chosen a small number of time steps to avoid the accumulation of optimization errors. With this setup, it takes about 10 seconds to train the network approximating the initial condition, and about 20 seconds to run both the NG and the NSL methods. On the other hand, since the PINN is now solving a 5D problem, a larger network has to be used, leading to a training time of about 5 minutes.

The results are displayed on [Figure 5](#), where we show the approximate solutions (top panels) and the associated errors (bottom panels). We observe a very good agreement of the NSL solution, especially close to the peak of the Gaussian bump, which is much better approximated with the NSL method than with the other methods.

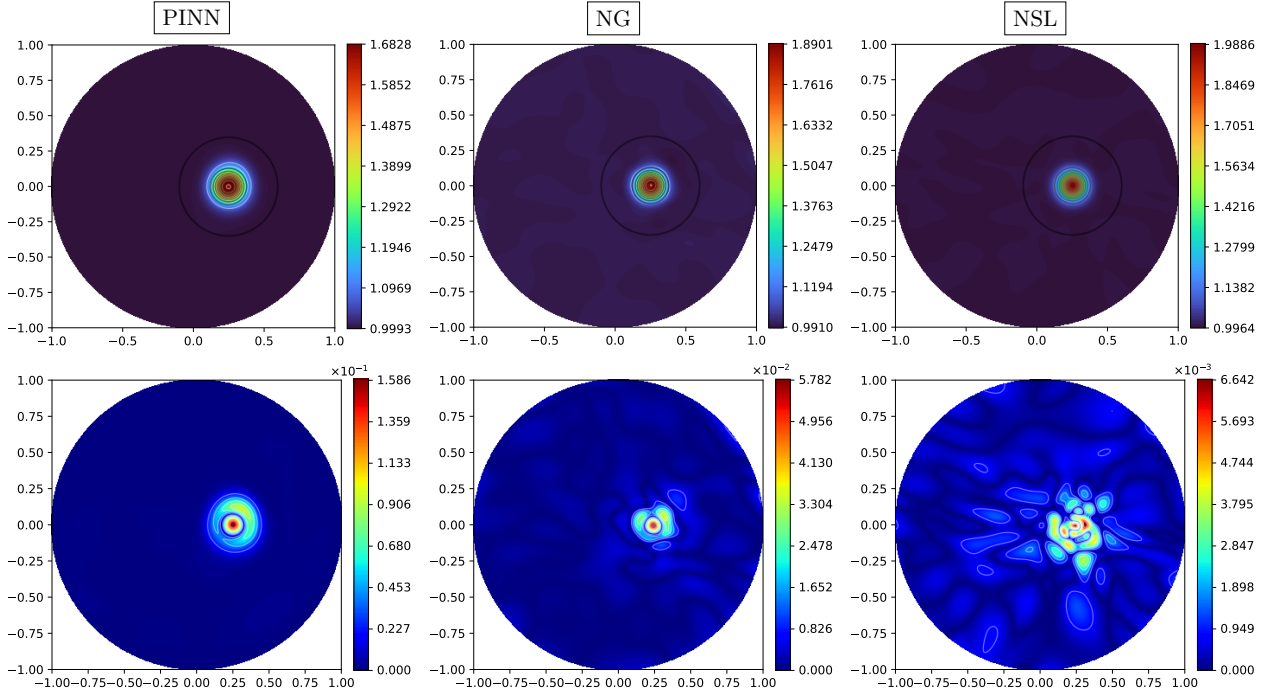


Figure 5: 2D parametric rotating advection from [Section 4.2.1](#): from left to right, results of the PINN, NG and NSL schemes for $\mu = (0.06, 0.25)$, with a fixed time step for NG ($\Delta t = 0.02$) and NSL ($\Delta t = 0.5$). Top panels: approximate solutions; bottom panels: absolute value of the pointwise error.

For a refined comparison of the methods, we run the three methods on 2000 randomly sampled parameters in $\mathbb{M}$, and report the statistics on the errors in [Table 2](#). In both L^2 and L^∞ norms, we observe a significant gain in accuracy for the NG and NSL methods over the PINN. Furthermore, the NSL method outperforms the NG method, by a factor of 3 in L^2 norm, and up to 5 in L^∞ norm. For all cases, we note that the maximum errors are concentrated close to the boundaries of the parameter space.

4.2.2. 1D1V Vlasov equation

The next test case in this series is a Vlasov equation in one space dimension and one velocity dimension. It is nothing but a 2D advection equation without parameters, whose setup is described in e.g. [\[4\]](#). It is given

	NSL		NG		PINN	
	L^2 error	L^∞ error	L^2 error	L^∞ error	L^2 error	L^∞ error
min	3.79×10^{-4}	5.73×10^{-3}	1.05×10^{-3}	1.89×10^{-2}	1.20×10^{-3}	5.25×10^{-2}
avg	4.99×10^{-4}	1.08×10^{-2}	1.57×10^{-3}	5.78×10^{-2}	2.73×10^{-3}	1.36×10^{-1}
max	9.47×10^{-4}	4.28×10^{-2}	2.80×10^{-3}	1.52×10^{-1}	5.26×10^{-3}	2.70×10^{-1}
std	9.76×10^{-5}	6.24×10^{-3}	3.72×10^{-4}	2.89×10^{-2}	7.97×10^{-4}	5.20×10^{-2}

Table 2: 2D parametric rotating advection from [Section 4.2.1](#): statistics on the errors e_x for 2000 randomly sampled parameters in $\mathbb{M}$.

by the following set of equations, with periodic boundary conditions in x and v :

$$\begin{cases} \partial_t u + v \partial_x u + \sin(x) \partial_v u = 0, & x \in (0, 2\pi), v \in (-6, 6), t \in (0, 4.5), \\ u(0, x, v) = u_0(x, v), & x \in \Omega, v \in (-6, 6), \\ u(t, 0, v) = u(t, 2\pi, v), & v \in (-6, 6), t \in (0, 4.5), \\ u(t, x, -6) = u(t, x, 6), & x \in (0, 2\pi), t \in (0, 4.5), \end{cases} \quad (4.3)$$

with the initial condition

$$u_0(x, v) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{v^2}{2}\right).$$

This time, neither the advection equation (4.3) nor the characteristic ODE (2.4) have a closed-form solution. To obtain a reference solution, we numerically solve (4.3) using a classical semi-Lagrangian scheme³ on a Cartesian grid, with 512 in space and 8192 in velocity, and with 200 time steps per second (leading to e.g. 900 time steps for $t = 4.5$). Moreover, to solve the characteristic ODE, we use the third branch of [Algorithm 3](#), with $n_\tau = 5$ sub-time steps.

Since this problem is more complex than the previous ones, in that it leads to a sheared solution with filaments, we ran the simulation in double precision. Moreover, the hyperparameters are adjusted accordingly. Namely, more collocation points are used to approximate the integrals. All hyperparameters are given in [Table B.12](#). We elect to use a fixed time step $\Delta t = 10^{-2}$ for the NG method, corresponding to 450 time iterations, and $\Delta t = 1.5$ for the NSL method, corresponding to 3 time iterations. This leads to a computation time of about 5 minutes for the PINN, 1.5 minute for the NSL method and 4 minutes for the NG method (plus 30 seconds for the initial condition). The results are displayed in [Figures 6 to 8](#).

This experiment shows the limits of the PINN when approximating a solution with fine structures. Indeed, the results of the PINN (displayed in the left panels of the figures) are quite poor, and the solution is diffused away. Since the PINN learns all times at once, there is no reason why the early times should be better approximated than the later times, and this is confirmed by observing the results.

The NG scheme, on the other hand, provides a better approximation of the solution than the PINN. However, even though the solution is less diffusive, it remains more oscillatory, especially close to the filaments. In particular, we see that the solution becomes negative in some regions, which is not physical.

Finally, the NSL method provides the best approximation among the three tested methods. Namely, the fine filaments present in the solution are well-captured, although the approximation quality decreases with time since the filamentation process increases. This decrease in quality is not improved when decreasing the time step, even locally in time (e.g. taking two time steps between $t = 3$ and $t = 4.5$). This motivates the next section, where we will introduce preconditioning and other techniques to further improve the accuracy and efficiency of our method.

Lastly, we report in [Table 3](#) the L^2 and L^∞ errors of each method, bearing in mind that the L^∞ error may not be very informative in this case, since we are comparing solutions with sharp gradients. These errors confirm our observations from [Figures 6 to 8](#), namely that PINN is the least accurate method, while NSL is the most accurate one, improving the L^2 error by a factor of about 10 and the L^∞ error by a factor of about 5 compared to the NG method, all for a third of the computational cost.

³The semi-Lagrangian code is inspired from the one developed by Pierre Navaro, available at <https://pnavaro.github.io/python-notebooks/19-LandauDamping.html>.

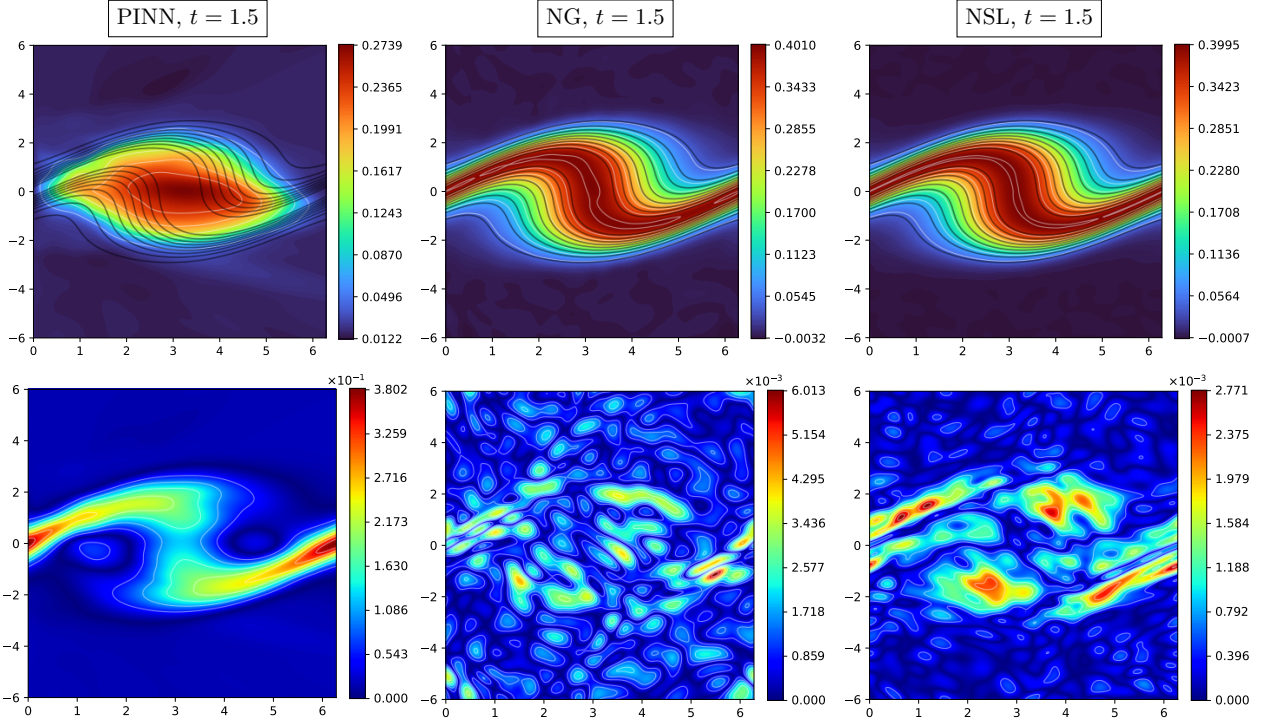


Figure 6: 2D Vlasov equation from [Section 4.2.2](#): from left to right, results of the PINN, NG and NSL schemes at time $t = 1.5$. Top panels: approximate solutions and contour lines of the exact solution (in black); bottom panels: absolute value of the pointwise error.

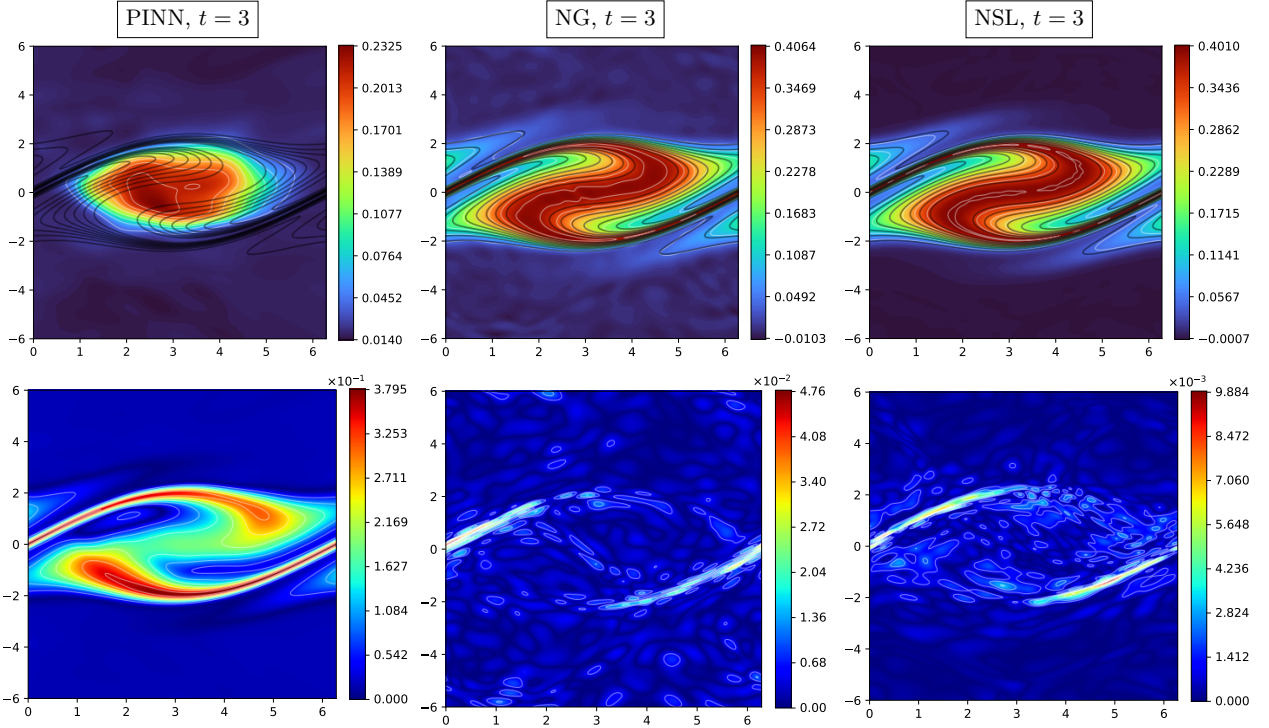


Figure 7: 2D Vlasov equation from [Section 4.2.2](#): from left to right, results of the PINN, NG and NSL schemes at time $t = 3$. Top panels: approximate solutions and contour lines of the exact solution (in black); bottom panels: absolute value of the pointwise error.

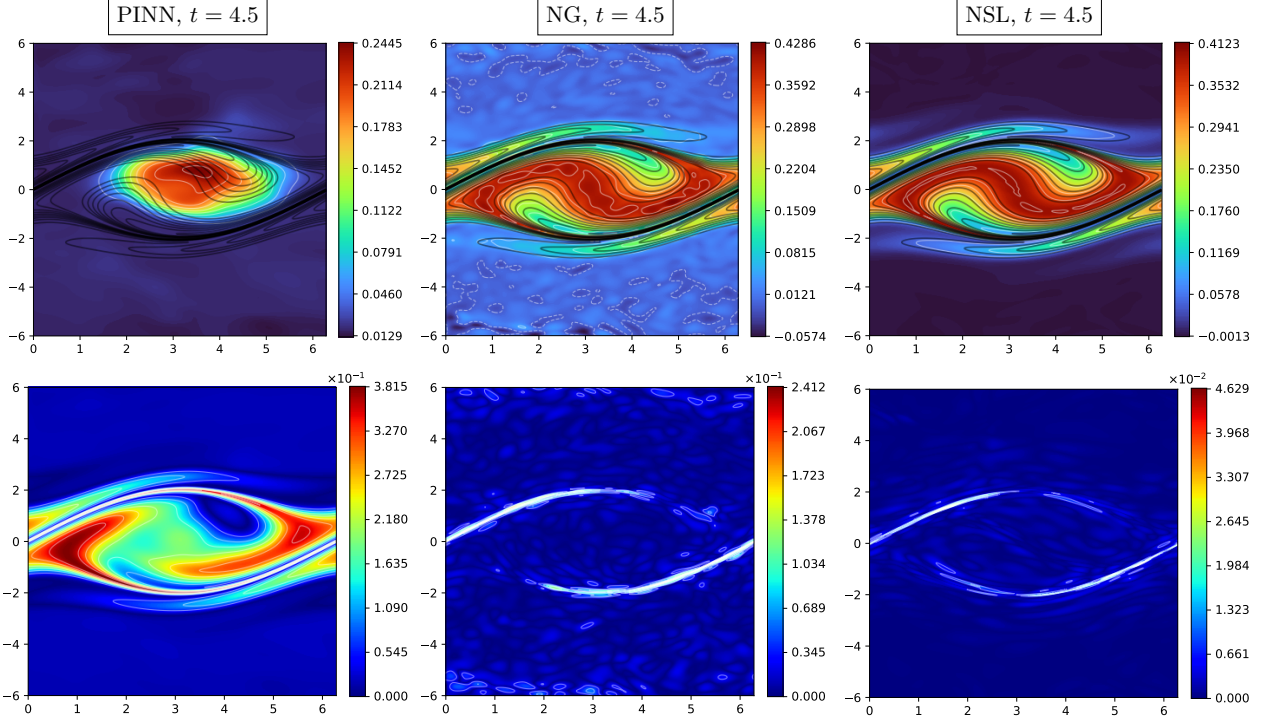


Figure 8: 2D Vlasov equation from Section 4.2.2: from left to right, results of the PINN, NG and NSL schemes at time $t = 4.5$. Top panels: approximate solutions and contour lines of the exact solution (in black); bottom panels: absolute value of the pointwise error.

time t	PINN error		NG error		NSL error	
	L^2 error	L^∞ error	L^2 error	L^∞ error	L^2 error	L^∞ error
1.5	7.98×10^{-2}	3.80×10^{-1}	1.19×10^{-3}	6.01×10^{-3}	6.76×10^{-4}	2.77×10^{-3}
3	1.06×10^{-1}	3.80×10^{-1}	3.39×10^{-3}	4.76×10^{-2}	8.22×10^{-4}	9.88×10^{-3}
4.5	1.17×10^{-1}	3.82×10^{-1}	1.48×10^{-2}	2.41×10^{-1}	1.50×10^{-3}	4.63×10^{-2}

Table 3: Comparison of L^2 and L^∞ errors for the PINN and the NG and NSL methods at different times.

4.3. Transport equation in a cylinder

From this section onwards, we will only consider the NSL method, and check it on several benchmark problems. The present section is devoted to testing the improvements proposed in Section 3.5. To that end, we consider an advection equation in three space dimensions and two parameter dimensions. The space domain is a cylinder $\Omega = \mathbb{D}^1 \times [0, 2]$. The solution u is governed by the advection equation

$$\partial_t u + a(x) \cdot \nabla u = 0, \quad x = (x_1, x_2, x_3)^\top \in \Omega, \quad t \in (0, 2), \quad \mu \in \mathbb{M},$$

supplemented with periodic boundary conditions in the third variable, and a suitable initial condition. Namely, we set the advection field to

$$a(x) = \begin{pmatrix} -2\pi x_2 \\ 2\pi x_1 \\ x_3 \end{pmatrix}, \quad (4.4)$$

corresponding to a rotation in the first two variables and a constant advection in the third one. Moreover, the (parameter-dependent) exact solution is a bump function, which reads

$$u_{\text{ex}}(t, x, \mu) = 1 + \exp\left(-\frac{1}{2v^2} \left((x_1 - c x_1^0(t))^2 + (x_2 - c x_2^0(t))^2 + x_3^0(t)^2 \right)\right),$$

where we have set

$$x_1^0(t) = \cos(2\pi t), \quad x_2^0(t) = \sin(2\pi t), \quad x_3^0(t) = (x_3 - t) \% 2 - 1,$$

where $a \% b$ denotes the modulo operation. The two parameters $c \in (0.3, 0.5)$ and $v \in (0.05, 0.15)$ respectively represent the position of the center of the Gaussian bump, and its variance. The exact solution corresponds to a rotation of the bump around the x_3 -axis, with a constant advection in the x_3 direction. We take a time step $\Delta t = 0.5$, corresponding to 4 time steps.

We compare the results of the NSL method, as described before, to the improved version equipped with natural gradient preconditioning (Section 3.5.1) and adaptive sampling (Section 3.5.2). The adaptive sampling is based on the zeroes of the gradient of the approximate solution, i.e.,

$$f = \frac{1}{10^{-2} + \|\nabla u_\theta\|^2}$$

in Algorithm 2, where 10^{-2} is a safety factor to avoid division by zero. Furthermore, we take $\sigma_1 = 0.5$, $\sigma_2 = 10$ and $\sigma_3 = 500$ in Algorithm 2. We test three configurations for the improved NSL method: first with a small network (configuration (a)), then with a larger network but few epochs (configuration (b)), and lastly with a larger network, as well as additional epochs and collocation points (configuration (c)). The hyperparameters are summarized in Table 4, for each configuration.

Hyperparameter	NSL	improved NSL		
		configuration (a)	configuration (b)	configuration (c)
N_e (init.)	1 000	150	150	500
N_e (iter.)	1 000	50	50	200
N_c	50 000	50 000	50 000	150 000
ℓ	[60, 60, 60]	[20, 40, 20]	[60, 60, 60]	[60, 60, 60]

Table 4: Hyperparameters (number of epochs N_e and layers ℓ) used for initialization (init.) and NSL iterations (iter.) in Section 4.3. In every case, we use a tanh activation function.

To report the results, we perform a parametric study over 2 000 randomly sampled parameters in $\mathbb{M}$, and report the statistics in Table 5, along with the computation time. The first configuration of the improved NSL method decreases the computation time by a factor of roughly 10, and decreases the average error by a factor of 10 in L^2 norm, 20 in L^∞ norm. These substantial increases in accuracy and efficiency make the improved NSL method even more competitive with other neural methods. We remark that the network is really quite small, and natural gradient preconditioning shows that solving the optimization problem is an important bottleneck in the training of the neural network. Moving on to larger networks and additional epochs, we observe a further decrease of the error, alongside a (substantial) increase in the computation time. We observe that, in configuration (b), the increase in epochs is not sufficient to improve the accuracy and solve the optimization problems more efficiently than in configuration (a). However, in configuration (c), the further increase in both epochs and collocation points allows us to further decrease the error by a factor of around 4. The downside is that the computation time is increased by a factor of about 45. All in all, we see that the improved NSL method is able to provide a very accurate solution for a small computational cost, and that this accuracy can be improved further should one be willing to pay the computation time.

To give an idea of the adaptively sampled points, we represent in Figures 9 and 10 the predictions (left panels) and associated errors (right panels) of the improved method, in configuration (c), at the final time $t = 2$, for the physical parameters $c = 0.4$ and $v = 0.1$, and in two planes where the Gaussian bump is localized ($x_3 = 1$ for Figure 9 and $x_2 = 0$ for Figure 10). We have displayed 5 000 adaptively sampled points out of the 150 000 collocation points actually used in this test case. In both cases, in addition to noticing that the errors produced by the improved NSL method are quite low, we observe that the adaptively sampled points are concentrated around the peak of the Gaussian bump, where the solution is the most challenging to approximate, and where the error is the largest, even though the adaptive sampling procedure is not aware of the exact solution.

Remark 6. For the remainder of the numerical experiments, we elect to use hyperparameters similar to configuration (c). This leads to higher computation times, but better accuracy for the optimization problems. Using hyperparameters similar to configuration (a) would lead to significantly lower computation times, at the cost of a decrease in accuracy.

		min	avg	max	std
non-improved	L^2 error	2.72×10^{-3}	5.22×10^{-3}	1.08×10^{-2}	1.69×10^{-3}
	L^∞ error	5.55×10^{-2}	1.91×10^{-1}	3.75×10^{-1}	9.97×10^{-2}
	computation time (init.)		121.78 s		
	computation time (iter.)		296.58 s		
configuration (a)	L^2 error	3.44×10^{-4}	4.36×10^{-4}	9.39×10^{-4}	1.23×10^{-4}
	L^∞ error	3.01×10^{-3}	1.03×10^{-2}	7.06×10^{-2}	1.23×10^{-2}
	computation time (init.)		19.06 s		
	computation time (iter.)		26.35 s		
configuration (b)	L^2 error	5.85×10^{-4}	7.21×10^{-4}	1.26×10^{-3}	1.30×10^{-4}
	L^∞ error	3.16×10^{-3}	8.49×10^{-3}	6.03×10^{-2}	9.58×10^{-3}
	computation time (init.)		110.20 s		
	computation time (iter.)		149.87 s		
configuration (c)	L^2 error	1.04×10^{-4}	1.36×10^{-4}	3.10×10^{-4}	3.33×10^{-5}
	L^∞ error	9.84×10^{-4}	2.74×10^{-3}	3.00×10^{-2}	4.08×10^{-3}
	computation time (init.)		774.73 s		
	computation time (iter.)		1 267.12 s		

Table 5: 3D parametric rotating advection from [Section 4.3](#): statistics on the errors e_x for 2000 randomly sampled parameters in $\mathbb{M}$.

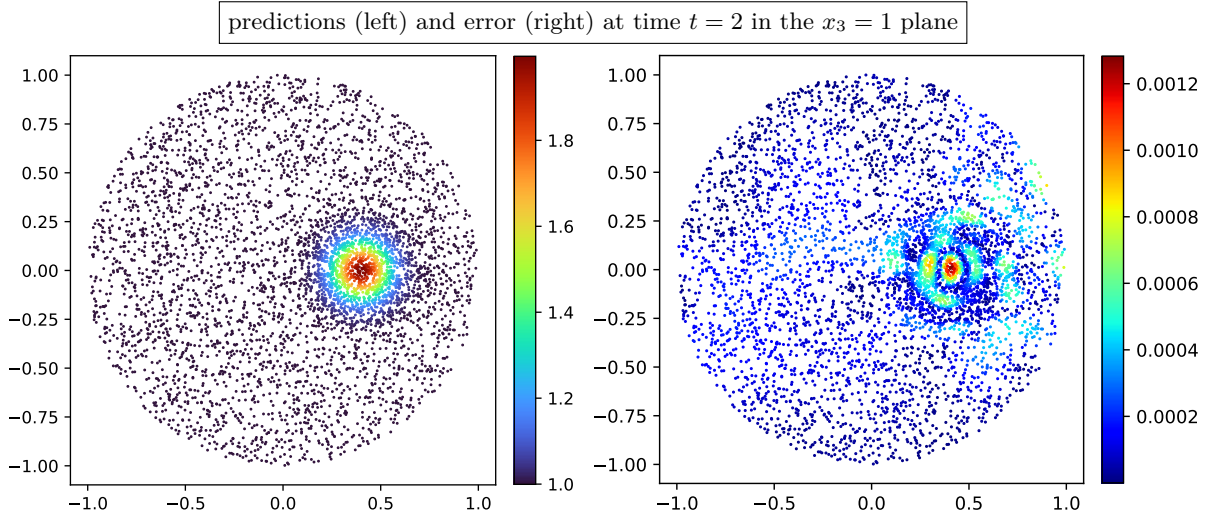


Figure 9: 3D parametric advection from [Section 4.3](#), in configuration (c): approximate solution (left) and error (right) in the $x_3 = 1$ plane at the final time $t = 2$, for the physical parameters $c = 0.4$ and $v = 0.1$.

4.4. Deformation of level-set functions

Equipped with the improved version of NSL, we now test it on two challenging test cases, namely the deformation of level-set functions, in 2D ([Section 4.4.1](#)), in 3D ([Section 4.4.2](#)), and in 3D with 2 parameters ([Section 4.4.3](#)). These test cases, described in e.g. [51, 12], consist in the transport of a level-set function by a time-dependent advection field. They are particularly interesting since, while the transient solution is not known, the final solution is the initial condition itself, which allows for a better assessment of the accuracy of the method. Since we are approximating a level-set function, we use $f = u_\theta$ in [Algorithm 2](#); we take $\sigma_1 = 2 \times 10^{-3}$, $\sigma_2 = 1 \times 10^{-2}$ and $\sigma_3 = 5 \times 10^{-2}$. In this section, we take $n_\tau = 10$ sub-time steps in the characteristic curve solver.

4.4.1. Deformation of a 2D level-set function

The initial condition of this two-dimensional level-set advection test case is, accordingly, a level-set function. It is defined in the space domain $\Omega = [0, 1]^2$, and it represents the disk of radius 0.15 centered at (0.5, 0.75),

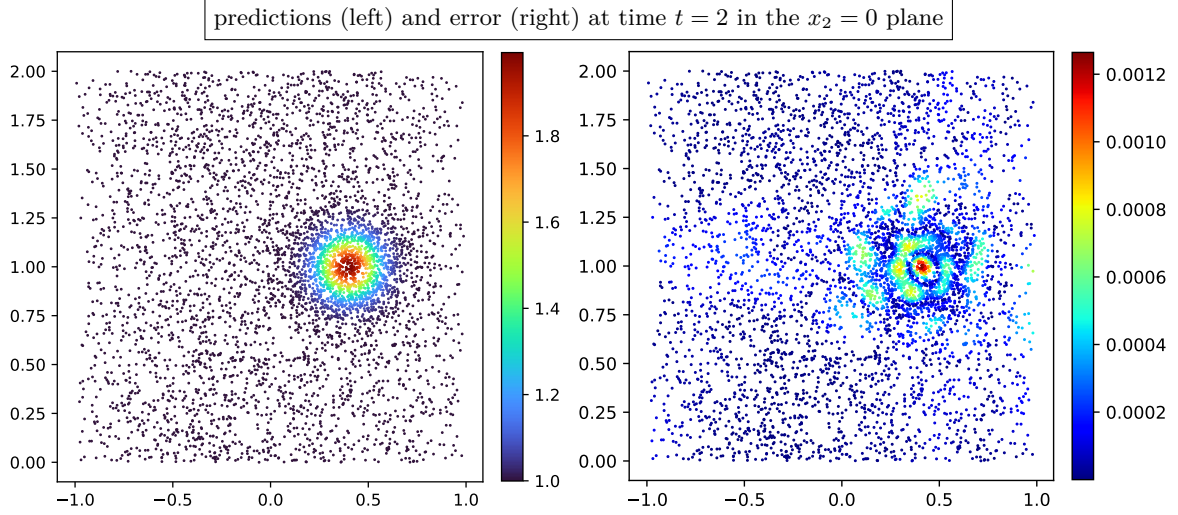


Figure 10: 3D parametric advection from [Section 4.3](#), in configuration (c): approximate solution (left) and error (right) in the $x_2 = 0$ plane at the final time $t = 2$, for the physical parameters $c = 0.4$ and $v = 0.1$.

i.e.,

$$u_0(x) = \left(x_1 - \frac{1}{2}\right)^2 + \left(x_2 - \frac{3}{4}\right)^2 - 0.15^2.$$

The shape then undergoes a deformation, according to the following time-dependent advection field:

$$a(t, x) = \begin{pmatrix} -\sin^2(\pi x_1) \sin(2\pi x_2) \cos(\frac{\pi t}{T}) \\ \sin^2(\pi x_2) \sin(2\pi x_1) \cos(\frac{\pi t}{T}) \end{pmatrix},$$

where T is the final time. The shape is most deformed at time $t = T/2$, and goes back to the initial shape at time $t = T$. The hyperparameters for this experiment are given in [Table B.13](#). We take a time step $\Delta t = 0.2$, corresponding to 40 time steps. Note that additional time steps are performed compared to previous experiments; this is to obtain a good discretization of the time-dependent advection field. With this configuration, the full computation takes about one hour.

[Figure 11](#) depicts the zero contour of the approximate level-set function at several times. We observe a good agreement with the reference solution from [\[51, 12\]](#). Namely, the deformation of the shape is well-captured, even at its most deformed, for $t = T/2 = 4$.

To get a better view of the accuracy of the NSL method, we compare the NSL solution at $t = 8$ to the exact solution, which is nothing but the initial condition. Namely, the zero contour of the exact solution is the disk of radius 0.15 centered at $(0.5, 0.75)$. This comparison is carried out in [Figure 12](#), where we observe a very good agreement between the two solutions, further validating the accuracy of the NSL method. For a more quantitative assessment, we compute the error between the true volume ($0.15^2\pi \approx 0.070685$) and the approximate one, computed by the Monte-Carlo method. We obtain

$$\int_{\Omega} \mathbb{1}_{\{u_{\theta} < 0\}} dx \approx 0.070924,$$

which leads to a relative error of about 0.338 %.

4.4.2. Deformation of a 3D level-set function

After the 2D level-set deformation, we move on to a more complex three-dimensional test case. This time, the initial condition is a level-set function, defined in $\Omega = [0, 1]^3$, of the sphere of radius 0.15 centered at $(0.35, 0.35, 0.35)$. This initial condition is given by

$$u_0(x) = (x_1 - 0.35)^2 + (x_2 - 0.35)^2 + (x_3 - 0.35)^2 - 0.15^2.$$

It is then advected with the time-dependent advection field

$$a(t, x) = \begin{pmatrix} \sin^2(\pi x_1) \sin(2\pi x_2) \sin(2\pi x_3) \cos(\frac{\pi t}{3}) \\ \sin^2(\pi x_2) \sin(2\pi x_1) \sin(2\pi x_3) \cos(\frac{\pi t}{3}) \\ \sin^2(\pi x_3) \sin(2\pi x_1) \sin(2\pi x_2) \cos(\frac{\pi t}{3}) \end{pmatrix}. \quad (4.5)$$

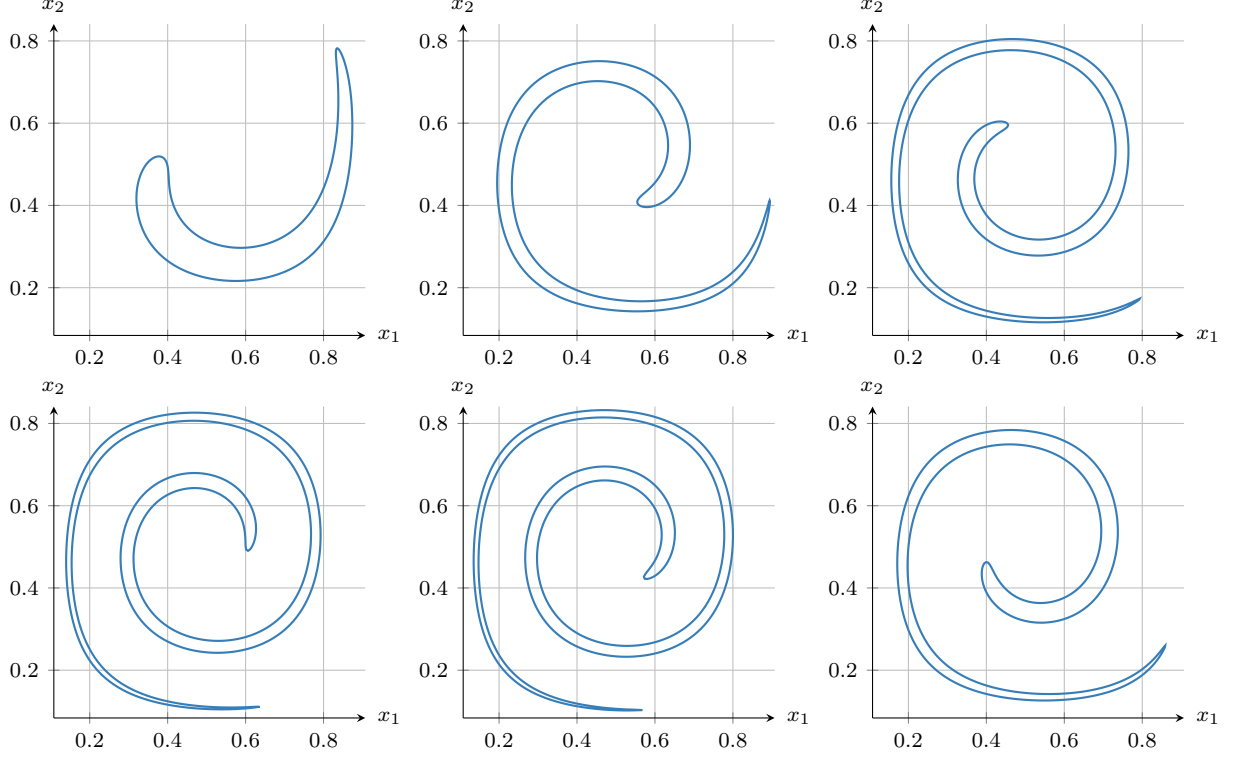


Figure 11: 2D level-set deformation from [Section 4.4.1](#): approximate solution at several times (from left to right and top to bottom, $t = 0.8$, $t = 1.6$, $t = 2.4$, $t = 3.2$, $t = 4$ and $t = 6$). Only the zero contour of the level-set function is displayed.

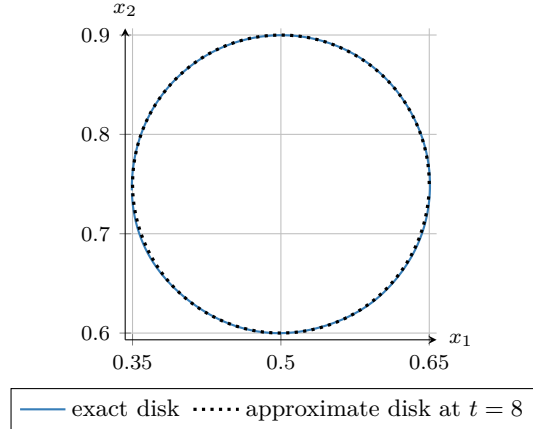


Figure 12: 2D level-set deformation from [Section 4.4.1](#): comparison of the exact solution (dotted black line) and the NSL solution (solid blue line) at the final time $t = T$. Only the zero contour of the level-set function is displayed.

Similarly to [Section 4.4.1](#), we recover the initial condition for $t = 3$. We take a time step $\Delta t = 0.3$, which corresponds to 10 time steps. This leads, together with the hyperparameters reported in [Table B.13](#), lead to a computation time of about 90 minutes. Same as the 2D level-set deformation, additional points are sampled around the zeroes of the approximate solution.

First, in [Figure 13](#), we display the zero contour of the approximate level-set function at all 10 computed times between 0 and 3, with a step of Δt . The approximate zero contour is well-captured by the NSL scheme, as can be seen by comparing it to the reference solution from [\[51, 12\]](#).

Second, [Figure 14](#) shows the zero contour of the approximate level-set function, comparing it to the exact solution u_0 . To do so, we display the 3D sphere sliced by the planes $x = 0.35$, $y = 0.35$ and $z = 0.35$, leading to three 2D graphs. Once again, we observe a very good agreement between the NSL solution and the exact solution, even for this more complex three-dimensional problem. Quantitatively, the exact volume of

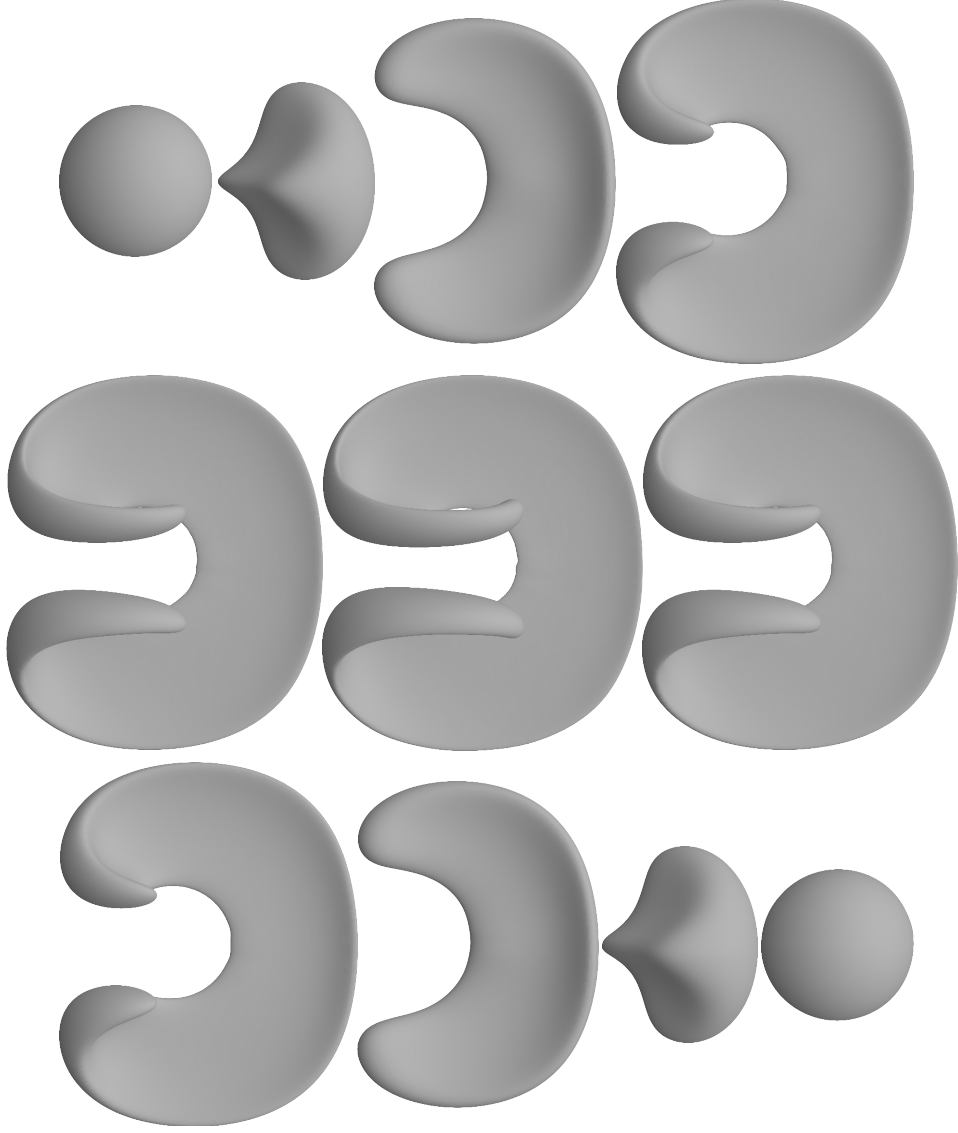


Figure 13: 3D level-set deformation from [Section 4.4.2](#): zero contour of the approximate solution at all 10 computed times (from left to right and top to bottom, starting from $t = 0$ on the top left and finishing with $t = 3$ on the bottom right).

the sphere is $\frac{4}{3}\pi(0.15)^3 \approx 0.014137$, while the approximate volume computed by the NSL method is roughly equal to 0.014108. The relative error is thus of about 0.209%.

4.4.3. Deformation of a 3D level-set function with two parameters

Lastly, we add two parameters to the previous 3D test case. The initial condition becomes a level-set function of a perturbed ellipsoid centered at $(0.35, 0.35, 0.35)$. The space domain remains $\Omega = [0, 1]^3$, and the parameters $\mu = (\alpha_1, \epsilon)$ live in the parameter domain $\mathbb{M} = [0.8, 1.2] \times [0, 0.25]$. The first parameter, α_1 , is a measure of the x -major axis of the ellipsoid, while the second parameter, ϵ , dictates how much the ellipsoid is perturbed. To give the initial condition, let us first define

$$x_{1,c} = x_1 - 0.35, \quad x_{2,c} = x_2 - 0.35, \quad x_{3,c} = x_3 - 0.35.$$

The radius and polar angle are defined by

$$r^2 = \left(\frac{x_{1,c}}{\alpha_1}\right)^2 + \left(\frac{x_{2,c}}{\alpha_1^{-1}}\right)^2 + \left(\frac{x_{3,c}}{1.2}\right)^2 \quad \text{and} \quad \varphi = \text{atan2}\left(\sqrt{x_{1,c}^2 + x_{2,c}^2}, x_{3,c}\right),$$

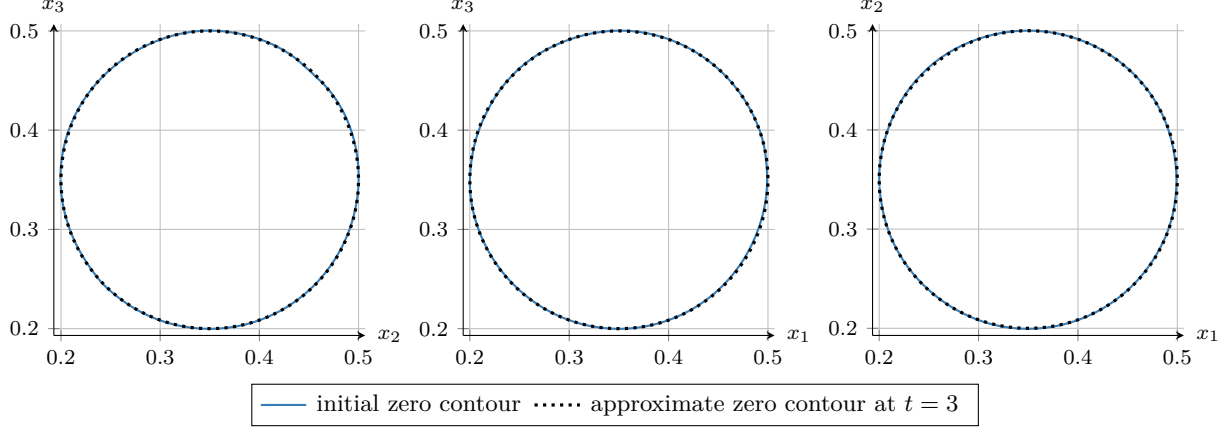


Figure 14: 3D level-set deformation from Section 4.4.2: comparison of the exact solution (dotted black line) and the NSL solution (solid blue line) at the final time $t = 3$. Only the zero contour of the level-set function is displayed. From left to right, we display the sphere sliced by planes $x_1 = 0.35$, $x_2 = 0.35$ and $x_3 = 0.35$.

and the initial condition is then

$$u_0(x, \mu) = 2 \left(r^2 \left(1 + \epsilon \sin(9\varphi) e^{-1.5(\varphi - \frac{\pi}{2})^2} \right) - 0.15^2 \right).$$

This initial condition is advected with the same velocity field (4.5) as in Section 4.4.2, and the initial condition is recovered at $t = 3$. We still take a time step $\Delta t = 0.3$ and 10 time steps. The hyperparameters are given in Table B.13. The total computation time is around 3 hours, twice as long as the non-parametric case. Same as the other two level-set deformations, the adaptive sampling focuses on the zeroes of the approximate solution.

The results are depicted in Figure 15. First, like in the previous case, we observe a very good agreement between zero contours of the approximate and exact solutions, as evidenced by the closeness of the solid blue line and the dotted black line. Of course, this problem is made harder by the parametrized perturbation of the solution. Hence, the approximate and exact solutions are not as nicely superimposed as in Figure 14, especially for highly perturbed solutions (represented in the right panels). Second, the positive and negative contours (solid and dashed red lines) emphasize that the approximate solution remains a level-set function, even after having undergone the deformation. Third, the volume preservation remains good: across 128 instances of parameters in $\mathbb{M}$, the average relative error on the volume is 0.468 %, with a standard deviation of 3.39×10^{-3} and a maximal error of 1.24 %. The method is thus about half as accurate (in terms of volume preservation) as for the non-parametric problem, which remains well within an acceptable range for such a jump in complexity.

4.5. High-dimensional advection-diffusion equations

This last section is dedicated to the approximation of advection-diffusion equations, following Section 3.4. In a d -dimensional space domain $\Omega \subset \mathbb{R}^d$, we define a constant advection field $a = (1, 1, \dots, 1)^\top$ and a constant diffusion coefficient σ . The advection-diffusion equation (1.1) then reads

$$\partial_t u + a \cdot \nabla u - \sigma \Delta u = 0, \quad x \in \Omega, \quad t \geq 0.$$

We first check the convergence of the scheme in Section 4.5.1, and then we test it on two high-dimensional problems: a periodic solution in Section 4.5.2 and a Gaussian solution in Section 4.5.3. In particular, we perform comparisons with a classical semi-Lagrangian scheme to evaluate the performance of the method, both in terms of accuracy and computational cost. Note, however, that the comparison will naturally favor the proposed method in a high-dimensional setting, since classical methods such as the semi-Lagrangian scheme suffer from the curse of dimensionality. A discussion of the results is finally provided in Section 4.5.4.

4.5.1. Convergence study

For this study, the space domain $\Omega = (-1, 1)^d$ is supplemented with periodic boundary conditions. We define the exact solution, for all $x \in \Omega$ and $t \geq 0$, by

$$u_{\text{ex}}(t, x) = 2 + \sin(\pi \langle x - at - s, a \rangle) e^{-\sigma \pi^2 \langle a, a \rangle t},$$

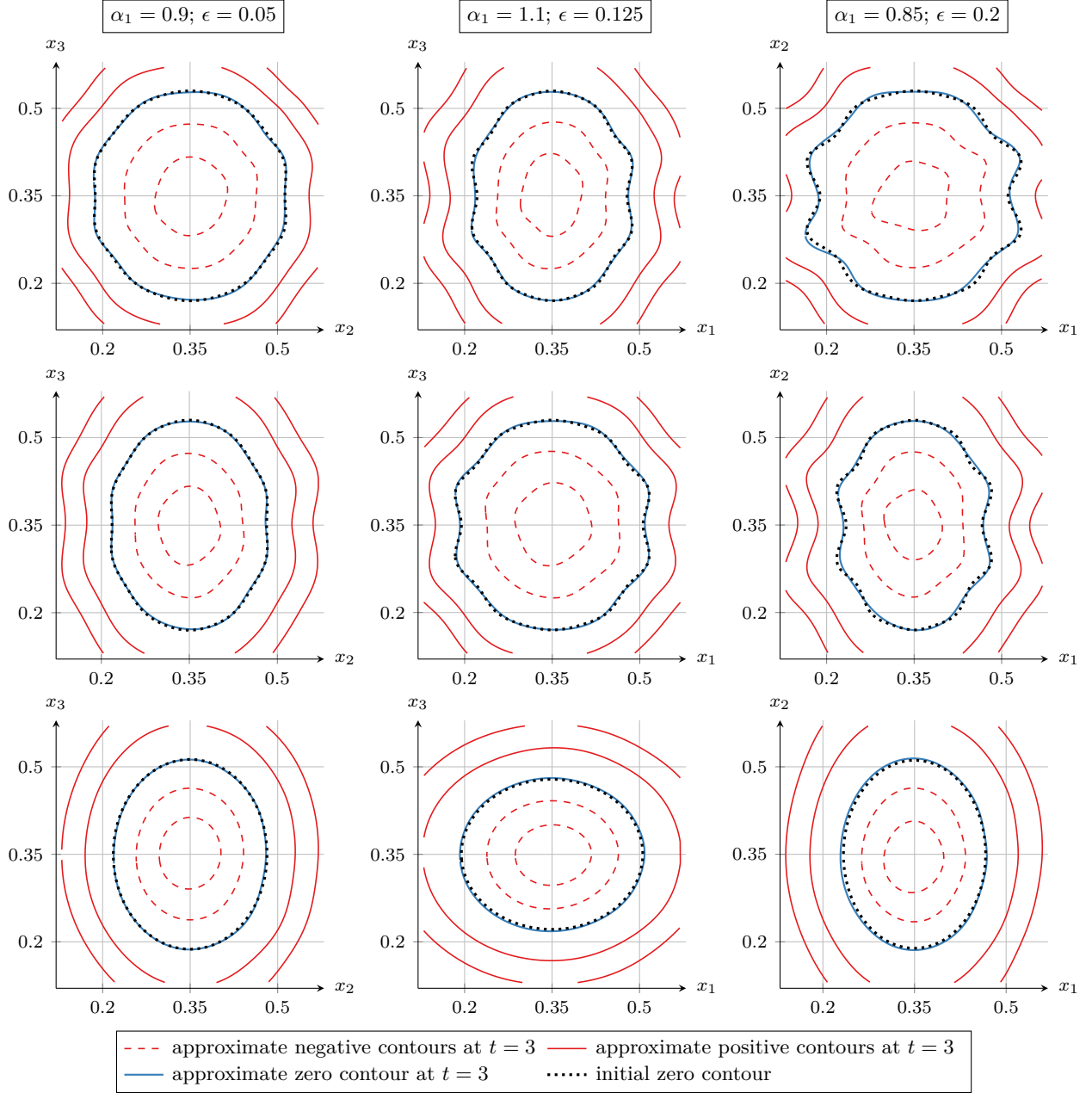


Figure 15: Deformation of a 3D level-set function with two parameters from [Section 4.4.3](#): comparison of the exact solution (dotted black line) and the NSL solution (solid blue line; solid and dashed red lines) at the final time $t = 3$. The solid blue line corresponds to the zero contour of the final solution; the solid and dashed red lines respectively correspond to positive and negative contours of the final solution. From top to bottom, we display the sphere sliced by planes $x_1 = 0.35$, $x_2 = 0.35$ and $x_3 = 0.35$. From left to right, three values of the parameters μ are considered.

with $s = (s_i)_{i=1}^d \in \mathbb{R}^d$, where for all $i = 1, \dots, d$, $s_i = (i - 1)/d$, is used to make the problem harder by introducing a different phase shift in each dimension. The rank of the solution, given by trigonometric formulas, is 2^{d-1} . In practice, the diffusion coefficient is set to $\sigma = 0.1$. For this experiment, natural gradient preconditioning is used, but not adaptive sampling, since there are no obvious areas in which additional points should be sampled. Moreover, sin activation functions are employed.

To test the convergence of the scheme as Δt decreases, we select dimension $d = 3$ to avoid unnecessary computational costs but still have a reasonably high-dimensional problem. According to [Remark 6](#), we choose to use hyperparameters corresponding to a higher accuracy, at the cost of computation time. This configuration is used to ensure a good spatial accuracy, thus making it possible to observe the convergence in time. This leads us to taking $\ell = [30, 30]$ layers, $N_e = 400$ epochs for the initialization, $N_e = 15$ epochs per time step,

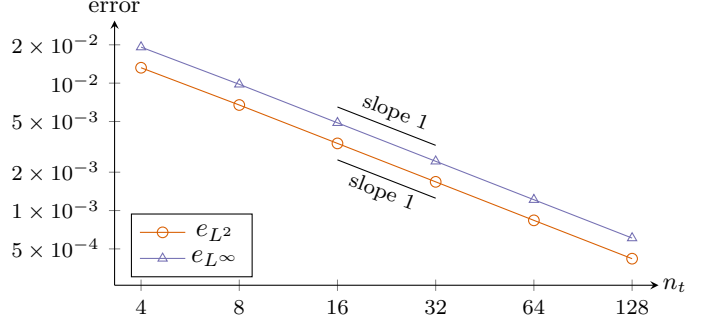
and $N_c = 100\,000$ collocation points. We take the final time $T = 1$, and we vary Δt by running the scheme with several numbers n_t of time steps. To that end, we compute the following L^2 and L^∞ relative errors

$$e_{L^2} = \sqrt{\frac{1}{N'_c} \sum_{k=1}^{N'_c} \frac{(u(T, x_i) - u_{\text{ex}}(T, x_i))^2}{u_{\text{ex}}(T, x_i)^2}} \quad \text{and} \quad e_{L^\infty} = \max_{i \in \{1, \dots, N'_c\}} \frac{|u(T, x_i) - u_{\text{ex}}(T, x_i)|}{|u_{\text{ex}}(T, x_i)|},$$

for $(x_i)_{i \in \{1, \dots, N'_c\}}$ a set of N'_c collocation points uniformly drawn in $(-1, 1)^d$. In practice, we use about 10 times as many collocation points to compute the error than to compute the approximate solution: $N'_c \approx 10N_c$. Note that the true L^2 error would have required a multiplication by the volume of the domain, which is 2^d . The errors are reported in Figure 16, where we observe, as expected, that the error decreases with order one as the number of time steps increases. Moreover, the scheme is indeed stable for all time steps, even very large ones.

n_t	e_{L^2}	e_{L^∞}
4	1.32×10^{-2}	1.91×10^{-2}
8	6.73×10^{-3}	9.79×10^{-3}
16	3.36×10^{-3}	4.88×10^{-3}
32	1.68×10^{-3}	2.43×10^{-3}
64	8.37×10^{-4}	1.21×10^{-3}
128	4.18×10^{-4}	6.08×10^{-4}

(a) Errors for advection-diffusion (in dimension $d = 3$) across different numbers of time steps.



(b) Relative errors vs. number of time steps (in log-log scale).

Figure 16: Advection-diffusion equation from Section 4.5: convergence of the error with respect to the number of time steps n_t in dimension $d = 3$.

4.5.2. High-dimensional periodic solution

Then, we check the error and the computation time with respect to the dimension d , making sure that they do not explode with the dimension. We still consider the periodic solution from Section 4.5.1. We fix the dimension-dependent final time

$$T = \frac{\log 2}{\sigma \pi^2 \langle a, a \rangle},$$

which corresponds to the time at which the diffusion halves the amplitude of the solution. The interval $[0, T]$ is discretized into 20 time steps.

Compared to the previous case, we still use sin activation functions, but the number of layers depends on the dimension d : namely, we take $\ell = [7d, 7d, 7d]$. The other hyperparameters are the same with respect to the dimension: the number of collocation points is $N_c = 75\,000$, the number of epochs is $N_e = 250$ for the initialization and $N_e = 5$ for each iteration.

This time, we also compare our neural semi-Lagrangian scheme to a classical one, also implemented on GPU using the PyTorch library. The classical SL scheme uses a uniform mesh with n_x points per spatial direction. The characteristic curves are then computed to shift the mesh, which is then interpolated on the original mesh in a directionwise manner, using a Lagrange interpolator of degree 3. Contrary to the NSL case, the error is computed directly at the mesh points rather than at randomly sampled collocation points.

The results are reported in Table 6 and Figure 17. To determine the number n_x of space points per direction in the classical SL scheme, we have used the following rule:

- for lower dimensions ($d \leq 4$), choose n_x to have roughly the same L^2 error as the NSL scheme (represented by a red area on Figure 17);
- for higher dimensions ($d \geq 5$), choose n_x to have roughly the same computation time as the NSL scheme (represented by a green area on Figure 17).

This leads to the memory consumption and computation time being comparable for high dimensions, where the classical scheme struggles the most. Otherwise, reaching similar errors in high dimensions would have been prohibitively expensive in terms of computation time, or even unreachable in terms of the 64 GB memory of our GPU.

dimension d	classical SL				neural SL			
	n_x	Time	relative error		Time		relative error	
			e_{L^2}	e_{L^∞}	init.	iter.	e_{L^2}	e_{L^∞}
1	36	0.96 s	1.57×10^{-3}	2.71×10^{-3}	20.88 s	8.96 s	1.56×10^{-3}	2.73×10^{-3}
2	38	2.19 s	1.63×10^{-3}	2.70×10^{-3}	22.02 s	11.05 s	1.54×10^{-3}	2.79×10^{-3}
3	40	3.31 s	1.56×10^{-3}	2.56×10^{-3}	25.49 s	13.96 s	1.54×10^{-3}	3.40×10^{-3}
4	40	17.09 s	1.49×10^{-3}	2.47×10^{-3}	31.98 s	18.31 s	1.55×10^{-3}	3.06×10^{-3}
5	30	50.64 s	9.11×10^{-3}	1.33×10^{-2}	49.33 s	27.43 s	1.52×10^{-3}	6.17×10^{-3}
6	19	110.62 s	3.91×10^{-3}	8.72×10^{-3}	72.50 s	42.72 s	1.53×10^{-3}	1.11×10^{-2}
7	13	158.15 s	1.00×10^{-2}	2.99×10^{-2}	113.48 s	59.05 s	1.54×10^{-3}	6.60×10^{-3}
8	10	271.33 s	1.66×10^{-2}	8.85×10^{-2}	170.27 s	84.11 s	1.57×10^{-3}	1.94×10^{-2}

Table 6: Advection-diffusion equation (4.5) in a periodic domain from Section 4.5.2: comparison between the classical and neural semi-Lagrangian schemes. Some performance metrics are reported in different dimensions. For the classical SL scheme, we report the number n_x of points per spatial direction, the computation time, and the L^2 and L^∞ errors at the final time T . For the neural SL scheme, we report the computation time for all the time iterations (iter.) and initialization (init.), with associated L^2 and L^∞ errors.

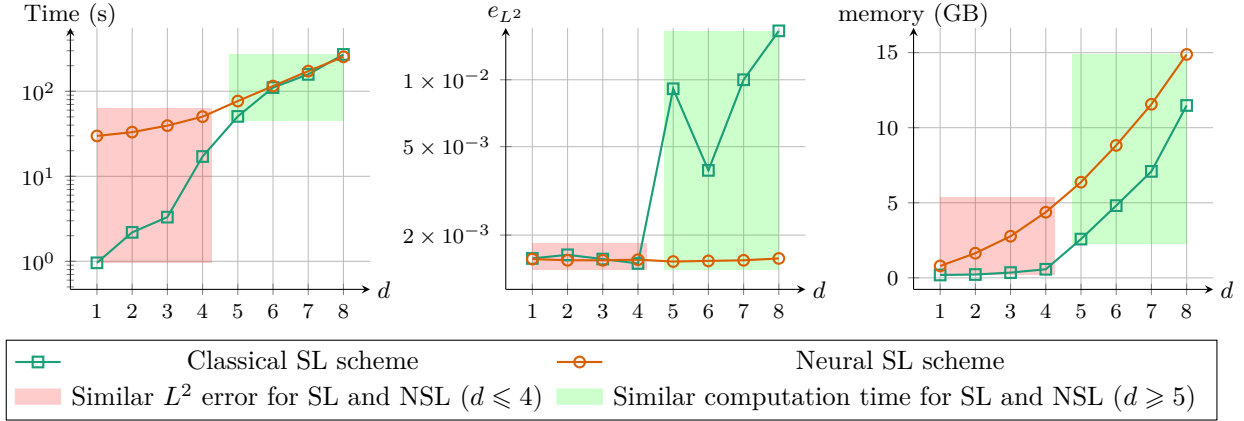


Figure 17: Advection-diffusion equation (4.5) in a periodic domain from Section 4.5.2: comparison between the classical (green line) and neural (orange line) semi-Lagrangian schemes. Some performance metrics (from left to right: computation time, L^2 error, and memory consumption) are displayed with respect to the dimension d for both schemes. The faded red and green areas represent the regions where the classical SL scheme has roughly the same L^2 error and computation time as the NSL scheme, respectively.

With this configuration, we observe that the L^2 error remains roughly constant with the dimension for the NSL scheme, while it sharply increases for the classical SL scheme when higher dimensions are considered. For both L^2 and L^∞ errors with roughly the same resource usage (computation time and memory), we observe that the NSL scheme outperforms the classical SL scheme by a factor of around 10 in dimensions $d \geq 5$. Moreover, for dimensions $d \leq 4$, we note that the NSL scheme has an error similar to that of a classical SL scheme with around 40 points per spatial direction. Reaching this level of refinement becomes untenable in higher dimensions. These findings are in line with e.g. [34], where PINNs were proven to be inferior to standard finite element methods in low dimensions ($d \leq 3$).

4.5.3. High-dimensional Gaussian solution

We now consider a more localized solution. Namely, in a non-periodic domain, the exact solution is given by

$$u_{\text{ex}}(t, x) = 1 + \sqrt{\frac{\det \Sigma(0)}{\det \Sigma(t)}} \exp \left(-\frac{1}{2} (x - M(t))^T \Sigma(t)^{-1} (x - M(t)) \right),$$

where the mean M is defined by $M(t) = at$ and the covariance Σ satisfies $\Sigma(t) = \Sigma(0) + 2\sigma t I_d$, with I_d the identity matrix in dimension d . The symmetric positive-definite initial covariance $\Sigma(0)$ satisfies

$$\Sigma(0) = 2d \sigma_0^2 (2d I_d + \tilde{\Sigma}), \quad \text{with} \quad \sigma_0 = 0.05 \quad \text{and} \quad \tilde{\Sigma}_{ij} = d - |i - j|,$$

and we take the diffusion coefficient $\sigma = 5 \times 10^{-2}$. This solution corresponds to a nonsymmetric Gaussian pulse being advected and diffused along the diagonal of the domain. To have a good representation of this solution, we take the space domain $\Omega = (-3, 3)^d$, and the final time is $T = 1$. Note that, in this section, we only consider dimensions $d > 1$, to have a true non-separable and anisotropic solution.

Contrary to an isotropic Gaussian bump, the tensor rank of this exact solution is not one. For example, in dimension 2, its rank is around 15 to 20 (depending on time). In this case, an approximation with precision 10^{-3} requires a rank of 5 or 6. So, like in the previous case, a low rank method is not well-suited to this test case. Indeed, the approximate rank value combined with the dimension, which can go up to 8, means that the number of 1D interpolations that must be performed at each step of the time-stepping algorithm becomes significant, making the whole computation much more expensive.

This time, we use both the natural gradient preconditioning and the adaptive sampling. For the latter, we sample using the gradient of the solution, like in (4.4), and we set $\sigma_1 = 20$, $\sigma_2 = 100$ and $\sigma_3 = 5000$. Regarding the other hyperparameters, we take 5 time steps, regularized hat activation functions as defined in (B.1), and the number of layers is $\ell = [7d, 7d, 7d]$. The number of epochs is $N_e = 150$ for the initialization and $N_e = 15$ for each iteration, and we take $N_c = 75\,000$ collocation points.

dimension d	classical SL				neural SL			
	n_x	Time	relative error		Time		relative error	
			e_{L^2}	e_{L^∞}	init.	iter.	e_{L^2}	e_{L^∞}
2	64	0.86 s	9.39×10^{-4}	1.15×10^{-2}	21.27 s	10.84 s	8.75×10^{-4}	1.14×10^{-2}
3	64	1.27 s	3.08×10^{-4}	1.09×10^{-2}	23.97 s	14.23 s	3.44×10^{-4}	1.10×10^{-2}
4	64	27.16 s	1.15×10^{-4}	9.42×10^{-3}	29.76 s	17.86 s	1.47×10^{-4}	9.26×10^{-3}
5	32	62.91 s	1.03×10^{-4}	1.58×10^{-2}	41.45 s	25.12 s	7.35×10^{-5}	8.02×10^{-3}
6	23	85.59 s	1.12×10^{-4}	1.75×10^{-2}	55.86 s	33.86 s	3.93×10^{-5}	4.09×10^{-3}
7	15	106.87 s	1.19×10^{-4}	3.82×10^{-2}	79.70 s	47.69 s	3.62×10^{-5}	5.34×10^{-3}
8	11	139.95 s	2.47×10^{-4}	1.08×10^{-1}	109.83 s	64.34 s	3.69×10^{-5}	3.70×10^{-3}

Table 7: Advection-diffusion equation (4.5) of a Gaussian pulse Section 4.5.3: comparison between the classical and neural semi-Lagrangian schemes. Some performance metrics are reported in different dimensions. For the classical SL scheme, we report the number n_x of points per spatial direction, the computation time, and the L^2 and L^∞ errors at the final time T . For the neural SL scheme, we report the computation time for all the time iterations (iter.) and initialization (init.), with associated L^2 and L^∞ errors.

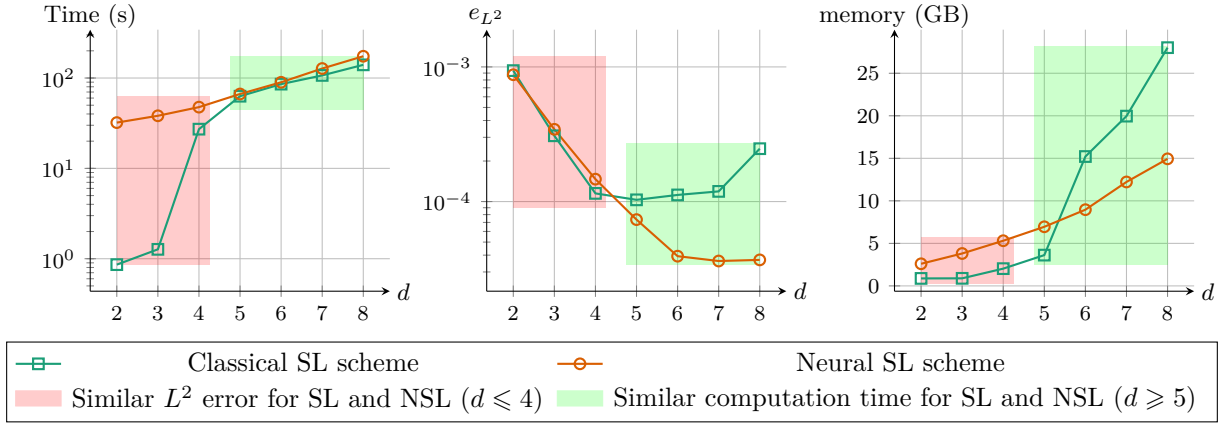


Figure 18: Advection-diffusion equation (4.5) of a Gaussian pulse Section 4.5.3: comparison between the classical (green line) and neural (orange line) semi-Lagrangian schemes. Some performance metrics (from left to right: computation time, L^2 error, and memory consumption) are displayed with respect to the dimension d for both schemes. The faded red and green areas represent the regions where the classical SL scheme has roughly the same L^2 error and computation time as the NSL scheme, respectively.

The results are reported in Table 7 and Figure 18. Similar comments as in the previous case apply. Namely, we observe that the NSL scheme is much more efficient than the classical SL scheme in high dimensions. This effect is further increased by the fact that the NSL scheme is roughly equivalent to a classical SL scheme with $n_x = 64$ points per spatial direction, which becomes even more prohibitive in high dimensions than the

previous case. Therefore, the NSL scheme reaches much lower L^2 and L^∞ errors than the classical SL scheme, up to a factor of 50 on the L^∞ error in dimension $d = 8$ for a similar computation time. Moreover, the memory consumption of the classical SL scheme increases much faster in this case, mostly when computing the initial condition. Indeed, taking $n_x = 12$ points per spatial direction in dimension $d = 8$ leads to almost 64 GB of memory, almost filling the GPU and taking over five minutes to compute the approximate solution.

4.5.4. Discussion

The results of the two previous sections allow us to draw some conclusions regarding the comparison between the classical and neural semi-Lagrangian schemes.

Memory consumption. The analysis of memory complexity is particularly challenging in our setting. In the classical SL method, memory usage scales predictably with the number of dofs, which grows exponentially with the dimension. For example, using N dofs (i.e., grid points) per dimension results in N^d dofs in d dimensions, quickly making storage intractable. In contrast, our approach relies on small neural networks (here, with between 161 dofs for $d = 1$ and 6776 dofs for $d = 8$), leading to negligible storage cost even if each time step is stored. However, the runtime memory footprint and number of operations per dof can still be significant, primarily due to the training process. This makes training significantly more memory-intensive than inference. Estimating the theoretical memory usage for neural networks is very challenging, and thus we only provided an empirical comparison.

Computation time. For both considered test cases, we observed that the neural semi-Lagrangian scheme is more efficient than the classical one for dimensions $d \geq 5$, leading to lower errors for the same computation time. Moreover, even if one was willing to pay an extremely high computational cost, the classical scheme would probably not be able to reach the same accuracy as the NSL one due to the aforementioned memory constraints. Indeed, to decrease the error by a factor of 8, one needs to increase the number of points per spatial direction by a factor of 2 (since the classical scheme is third-order accurate in space, and provided the time-stepping is accurate enough). In dimension $d = 8$, this means increasing the total number of dofs from 11^8 to $22^8 = 10^8 N^8 \approx 5 \times 10^{10}$, which is clearly unreachable on a single GPU. Even if it were reachable, the computation time would be multiplied by a factor of roughly 2^8 , leading to a total computation time of around 10h.

Applicability of the neural semi-Lagrangian method. As shown in our comparisons, we are able to solve problems in high dimensions, where, as expected, the standard semi-Lagrangian method become intractable due to the explosion in dofs, and associated memory and computational costs. This confirms that, while our method involves a higher per-iteration memory and computational cost, its global complexity in high dimensions is often far more manageable thanks to its mesh-free nature.

4.6. Extension to the Vlasov-Poisson equations

As a last experiment, we propose to extend the NSL method to a coupled nonlinear system: the Vlasov-Poisson equations. In this experiment, the goal is not to solve these equations efficiently. Rather, it is to show that the proposed method is able to handle such a system, despite it being naturally well-suited to linear problems. We first describe the system and how to solve it in [Section 4.6.1](#). Then, we present an approximation of the well-known bump-on-tail instability in [Section 4.6.2](#).

4.6.1. Governing equations and solving strategy

The Vlasov-Poisson equations (see e.g. [\[47\]](#)) are a system of kinetic equations, governed in one space and one velocity dimension by

$$\begin{cases} \partial_t u(t, x, v) + v \partial_x u(t, x, v) + E(t, x) \partial_v u(t, x, v) = 0, & \forall t \in [0, T], \forall x \in \Omega_x, \forall v \in \Omega_v, \\ \Delta \Psi(t, x) = \int_{\Omega_v} u(t, x, v) dv - \langle u \rangle, & \forall t \in [0, T], \forall x \in \Omega_x, \\ E(t, x) = \partial_x \Psi(t, x), & \forall t \in [0, T], \forall x \in \Omega_x, \end{cases}$$

where $t \in [0, T]$ is the time variable (with T a final time), $x \in \Omega_x$ is the space variable, and $v \in \Omega_v$ is the velocity variable. Periodic boundary conditions are prescribed everywhere. The function u represents

the distribution function, $\langle u \rangle = \frac{1}{|\Omega_x|} \int_{\Omega_x} \int_{\Omega_v} u(t, x, v) dv dx$ its average, and E is an electric field, with Ψ its potential. This equation comprises the Vlasov equation from [Section 4.2.2](#) and a Poisson equation giving E , the v -component of the advection field.

Classical semi-Lagrangian method. Because of this coupling, classical semi-Lagrangian strategies are not directly applicable. Usual, mesh-based numerical methods rely on an operator splitting, alternating between advection in the x -direction and advection in the v -direction, effectively dividing the dimension by two. Strang operator splitting makes it possible to achieve second-order accuracy in time. To describe the algorithm, assume that values u^n at time t^n are known on the grid. We seek updated values u^{n+1} at time t^{n+1} , still on the grid. The algorithm relies on the following operations, presented here in semi-discrete form:

1. solve $\partial_t u + v \partial_x u = 0$ with initial data u^n , from t^n to $t^n + 0.5 \Delta t$, to get intermediate values u^* ;
2. solve $\Delta \Psi^* = \int_{\Omega_v} u^* dv - \langle u^* \rangle$ to get an intermediate electric field $E^* = \partial_x \Psi^*$;
3. solve $\partial_t u + E^* \partial_v u = 0$ with initial data u^* , from t^n to t^{n+1} , to get new intermediate values u^{**} ;
4. solve $\partial_t u + v \partial_x u = 0$ with initial data u^{**} , from $t^n + 0.5 \Delta t$ to t^{n+1} , to get the updated values u^{n+1} .

Neural semi-Lagrangian method. Similarly, our neural semi-Lagrangian method has to be adapted to this coupling. In our case, there is no need to split the two advection operators to lower the dimension, since our meshless algorithm can handle higher dimensions than classical, mesh-based ones. In our case, the algorithm uses a predictor-corrector method to reach second-order accuracy in time. It reads as follows:

1. solve $\Delta \Psi^n = \int_{\Omega_v} u^n dv - \langle u^n \rangle$ to get the electric field $E^n = \partial_x \Psi^n$ at time t^n ;
2. solve $\partial_t u + v \partial_x u + E^n \partial_v u = 0$ with initial data u^n , from t^n to t^{n+1} , to get a prediction u^* ;
3. solve $\Delta \Psi^* = \int_{\Omega_v} u^* dv - \langle u^* \rangle$ to get a predicted electric field $E^* = \partial_x \Psi^*$, and set $\bar{E} = \frac{E^n + E^*}{2}$;
4. solve $\partial_t u + v \partial_x u + \bar{E} \partial_v u = 0$ with initial data u^n , from t^n to t^{n+1} , to get the updated values u^{n+1} .

The second and fourth steps are the same as in the previous sections, but the first and third ones involve solving a Poisson equation. Naturally, a PINN is well-suited for this task, since it gives the potential Ψ^n as a function that can be automatically differentiated to get the electric field E^n . Concretely, to solve the Poisson equation, the PINN uses hard-constrained homogeneous Dirichlet boundary conditions on Ψ , to fix the free constant in the Poisson equation with periodic boundary conditions. Classical schemes (based on e.g. a discrete Fourier transform) can also be employed to solve the Poisson equation in dimension $d = 1$. We still decide to use a PINN for this task, as a preparation for higher dimensions, and since both approaches gave the same results. All in all, our algorithm consists in alternating a PINN with the NSL method at each time step. More complex algorithms, potentially higher-order accurate in time, could also be employed. We do not investigate this direction further here, since our goal is to provide a proof of concept for such coupled nonlinear systems.

4.6.2. Validation: the bump-on-tail instability

The bump-on-tail instability consists in perturbing an initial equilibrium distribution function, thus creating vortices that grow over time. In lower dimensions, it is well-solved by classical schemes. The space domain is $\Omega_x = [0, 10\pi]$ and the velocity domain is $\Omega_v = [-9, 9]$, while the final time is $T = 32$. The initial condition is given by

$$u(0, x, v) = \left(\frac{1 - \varepsilon_{\text{bot}}}{\sqrt{2\pi}} e^{-\frac{v^2}{2}} + \frac{2 \varepsilon_{\text{bot}}}{\sqrt{2\pi T_{\text{bot}}}} e^{-\frac{(v - v_{\text{bot}})^2}{2 T_{\text{bot}}}} \right) (1 + \varepsilon_{\text{pert}} \cos(k_x x)),$$

where $\varepsilon_{\text{bot}} = 0.1$, $v_{\text{bot}} = 3.8$, $T_{\text{bot}} = 0.2$, $\varepsilon_{\text{pert}} = 0.03$, and $k_x = 0.4$.

Since Ψ is a PINN approximating the periodic solution to a one-dimensional equation, we use the rectangle method for integrating the loss function, for added efficiency. Moreover, the integral over Ω_v in the Poisson equation is also approximated using the rectangle method (using 196 points) since u is a periodic function. In addition, to keep the distribution function u nonnegative, we add a nonnegative activation function (ReLU^4) on the last layer. We take $\Delta t = 0.2$, leading to 160 time steps. With this choice of hyperparameters and time discretization, the computation time is around four hours. This is prohibitively long for such a two-dimensional test case (the reference solution runs in two hours on a single CPU core), but we recall that this simulation is merely a proof of concept, intended to show that the proposed NSL method is able to handle coupled and nonlinear problems.

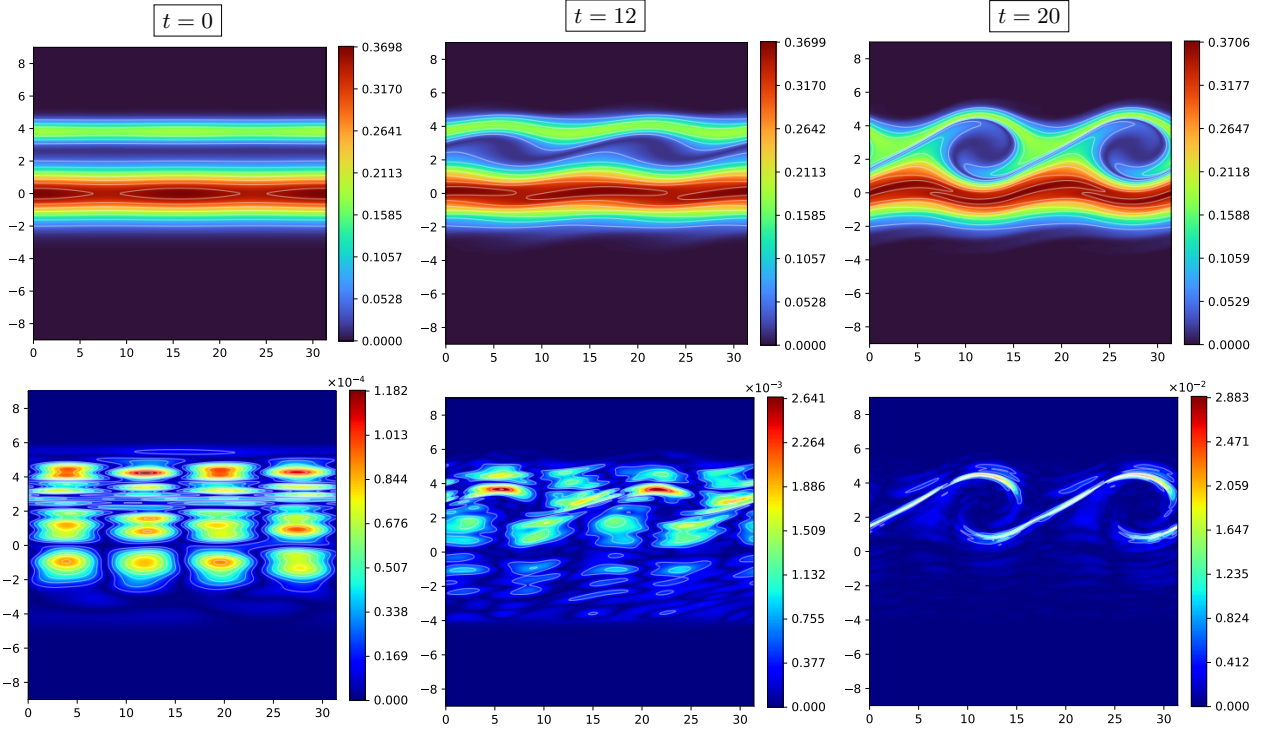


Figure 19: Bump-on-tail instability from [Section 4.6.2](#): evolution of the distribution function u at different times (from left to right: $t = 0$, $t = 12$ and $t = 20$) The top panels display u (in color) and its contours (in white). The bottom panels display the pointwise error between u and the reference solution.

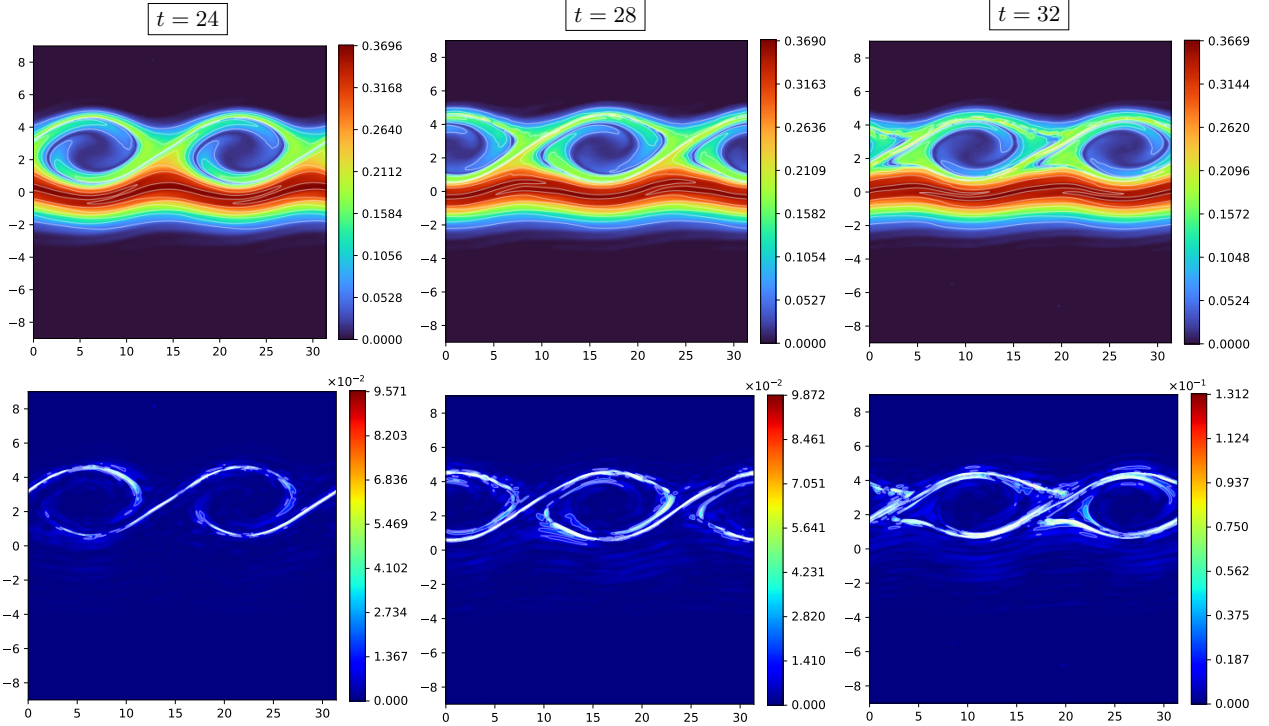


Figure 20: Bump-on-tail instability from [Section 4.6.2](#): evolution of the distribution function u at different times (from left to right: $t = 24$, $t = 28$ and $t = 32$) The top panels display u (in color) and its contours (in white). The bottom panels display the pointwise error between u and the reference solution.

	$t = 0$	$t = 12$	$t = 20$	$t = 24$	$t = 28$	$t = 32$
relative L^2 error	2.25×10^{-4}	3.41×10^{-3}	1.49×10^{-2}	2.56×10^{-2}	3.97×10^{-2}	6.23×10^{-2}
relative L^∞ error	3.20×10^{-4}	7.14×10^{-3}	7.79×10^{-2}	2.59×10^{-1}	2.67×10^{-1}	3.55×10^{-1}

Table 8: Bump-on-tail instability from Section 4.6.2: relative L^2 and L^∞ errors at different times.

The results are displayed in Figure 19 for $t \in \{0, 12, 20\}$ and in Figure 20 for $t \in \{24, 28, 32\}$. The reference solution, used to compute the error, is obtained using the algorithm described in Section 4.6.1 on a very fine space-time grid of 2048×1024 points and 8000 time steps (leading to $\Delta t = 5 \times 10^{-3}$). Relative L^2 and L^∞ errors are reported in Table 8. In all cases, we observe a good agreement with the reference solution. On the one hand, for times $t \leq 20$, the relative L^2 error remains below roughly 1%, and increases slowly over time. Of course, as time advances, the instability grows and becomes harder to approximate, and thus the relative L^2 error also increases over time, up to around 6% at $t = 32$. On the other hand, the L^∞ error is naturally larger than usual, and may not be a very relevant measure of error. This is due to two main reasons: first, the time grid of the NSL method is quite coarse; second, even with a finer grid, there would be an $\mathcal{O}(\Delta t^2)$ dispersion error. Indeed, the classical and neural methods do not use the same time splitting, even though both are second-order accurate in time. This leads to a slight shift between the two solutions, exacerbated by the long-time simulation. We nevertheless included the pointwise error in the figures, as it provides a more detailed view of the discrepancies. We also remark that the optimization problems become harder and harder to solve with time. Indeed, at the beginning of the simulation, the optimization problems are solved up to a mean squared error of 10^{-10} for the problem on u , and 10^{-12} for the problem on Ψ . This increases to 10^{-7} and 5×10^{-9} , respectively, at the final time.

5. Conclusion

In this work, we have introduced a neural version of the well-known semi-Lagrangian method. This method has the advantage of being able to solve advection-diffusion in high dimensions, without any time step requirements for stability. It is based on modelling the approximate solution using a neural network, and then using nonlinear optimization to solve the resulting backwards transport equation. It is an iterative process creating a sequence of neural networks $(u_{\theta^n})_n$ approximating the solution at time $t^n = n\Delta t$. This sequence is constructed using a simple algorithm, whose steps each consist in solving a nonlinear optimization problem. The rough algorithm is as follows:

- *step 0*: fit u_{θ^0} to the initial condition u_0 ;
- *step $n + 1$* : given the network u_{θ^n} at time t^n , fit $u_{\theta^{n+1}}$ to $u_{\theta^n}(\tilde{\mathcal{X}}(t^n; t^{n+1}, \cdot))$, where $\tilde{\mathcal{X}}(t^n; t^{n+1}, \cdot)$ is an approximation of the backwards characteristic curve.

Numerical results show the performance of the method, especially compared to (albeit more general) methods from the literature. Namely, we show that a good accuracy is retained in high dimensions, for a comparatively low computational cost.

To improve Vlasov simulations, the next step consists in exploring improved network architectures, e.g. using Fourier features [75], PirateNets [76], or Kolmogorov-Arnold Networks (KAN, see [54]). Domain decomposition and parallelism will also be employed to optimize computational efficiency, in the spirit of e.g. [23]. We will also propose a better adaptive sampling, to further reduce the computational cost, especially at high dimensions. Work like [55] could be adapted. We already provided a proof of concept on the Vlasov-Poisson equations, but our neural strategy will also be extended to more complex systems. For instance, Vlasov-Poisson simulations will be improved, and Vlasov-Maxwell [53] or gyrokinetic [28] models will be tackled, going towards high-dimensional applications in plasma physics using advanced splitting methods like in [18]. Our method could also be adapted to further improve the results obtained on the level-set deformation, by introducing strong or weak constraints on the volume preservation.

Acknowledgments

As part of the “France 2030” initiative, this work has benefited from a national grant managed by the French National Research Agency (Agence Nationale de la Recherche) attributed to the Exa-MA project of the NumPEx PEPR program, under the reference ANR-22-EXNU-0002. This work was also supported

by the French National Research Agency through projects ANR-23-PEIA-0004 (PDE-AI, all four authors), ANR-21-CE46-0014 (Milk, E. F. and L. N.), and ANR-22-CE25-0017 (OptiTrust, V. M.-D.).

References

- [1] S. Amari and S. C. Douglas. Why natural gradient? In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 2 of *ICASSP-98*, pages 1213–1216. IEEE, 1998.
- [2] J. Ansel and E. Yang et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ASPLOS, Volume 2*, volume 5 of *ASPLOS '24*, pages 929–947. ACM, 2024.
- [3] A. Beltran-Pulido, I. Bilionis, and D. Aliprantis. Physics-Informed Neural Networks for Solving Parametric Magnetostatic Problems. *IEEE Trans. Energy Convers.*, 37(4):2678–2689, 2022.
- [4] J. Berman and B. Peherstorfer. Randomized Sparse Neural Galerkin Schemes for Solving Evolution Equations with Deep Networks. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
- [5] N. Besse and M. Mehrenberger. Convergence of classes of high-order semi-Lagrangian schemes for the Vlasov–Poisson system. *Math. Comput.*, 77(261):93–123, 2008.
- [6] N. Besse and E. Sonnendrücker. Semi-Lagrangian schemes for the Vlasov equation on an unstructured mesh of phase space. *J. Comput. Phys.*, 191(2):341–376, 2003.
- [7] V. Biesek and P. H. de Almeida Konzen. Burgers’ PINNs with implicit Euler Transfer Learning. *Rev. Mundi Eng., Tecnol. Gest.*, 9(4), 2024.
- [8] O. Bokanowski and G. Simarmata. Semi-Lagrangian discontinuous Galerkin schemes for some first- and second-order partial differential equations. *ESAIM: M2AN*, 50(6):1699–1730, 2016.
- [9] L. Bonaventura, E. Calzola, E. Carlini, and R. Ferretti. Second Order Fully Semi-Lagrangian Discretizations of Advection-Diffusion-Reaction Systems. *J. Sci. Comput.*, 88(1), 2021.
- [10] L. Bonaventura, R. Ferretti, and L. Rocchi. A fully semi-Lagrangian discretization for the 2D incompressible Navier-Stokes equations in the vorticity-streamfunction formulation. *Appl. Math. Comput.*, 323:132–144, 2018.
- [11] J. Bruna, B. Peherstorfer, and E. Vanden-Eijnden. Neural Galerkin schemes with active learning for high-dimensional evolution equations. *J. Comput. Phys.*, 496:112588, 2024.
- [12] C. Bui, C. Dapogny, and P. Frey. An accurate anisotropic adaptation method for solving the level set advection equation. *Int. J. Numer. Meth. Fl.*, 70(7):899–922, 2011.
- [13] F. Charles, B. Després, and M. Mehrenberger. Enhanced Convergence Estimates for Semi-Lagrangian Schemes Application to the Vlasov–Poisson Equation. *SIAM J. Numer. Anal.*, 51(2):840–863, 2013.
- [14] Y. Chen, W. Guo, and X. Zhong. A learned conservative semi-Lagrangian finite volume scheme for transport simulations. *J. Comput. Phys.*, 490:112329, 2023.
- [15] Y. Chen, W. Guo, and X. Zhong. Conservative semi-Lagrangian finite difference scheme for transport simulations using graph neural networks. *J. Comput. Phys.*, 526:113768, 2025.
- [16] Z. Chen, J. McCarran, E. Vizcaino, L. Soljagic, and D. Luo. TENG: Time-Evolving Natural Gradient for Solving PDEs With Deep Neural Nets Toward Machine Precision. In R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 7143–7162. PMLR, 21–27 Jul 2024.
- [17] A. Cohen and G. Migliorati. Optimal weighted least-squares methods. *SMAI J. Comput. Math.*, 3:181–203, 2017.

- [18] N. Crouseilles, L. Einkemmer, and E. Faou. Hamiltonian splitting for the Vlasov-Maxwell equations. *J. Comput. Phys.*, 283:224–240, 2015.
- [19] N. Crouseilles, M. Mehrenberger, and É. Sonnendrücker. Conservative semi-Lagrangian schemes for Vlasov equations. *J. Comput. Phys.*, 229(6):1927–1953, 2010.
- [20] N. Crouseilles, M. Mehrenberger, and F. Vecil. Discontinuous Galerkin semi-Lagrangian method for Vlasov-Poisson. *ESAIM: Proceedings*, 32:211–230, 2011.
- [21] N. Crouseilles, T. Respaud, and É. Sonnendrücker. A forward semi-Lagrangian method for the numerical solution of the Vlasov equation. *Comput. Phys. Commun.*, 180(10):1730–1745, 2009.
- [22] T. De Ryck and S. Mishra. Error analysis for physics-informed neural networks (PINNs) approximating Kolmogorov PDEs. *Adv. Comput. Math.*, 48(6), 2022.
- [23] V. Dolean, A. Heinlein, S. Mishra, and B. Moseley. Multilevel domain decomposition-based architectures for physics-informed neural networks. *Comput. Method. Appl. M.*, 429:117116, 2024.
- [24] J. Douglas, Jr and T. F. Russell. Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element or finite difference procedures. *SIAM J. Numer. Anal.*, 19(5):871–885, 1982.
- [25] W. E and B. Yu. The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems. *Commun. Math. Stat.*, 6(1):1–12, 2018.
- [26] L. Einkemmer and I. Joseph. A mass, momentum, and energy conservative dynamical low-rank scheme for the vlasov equation. *J. Comput. Phys.*, 443:110495, 2021.
- [27] L. Einkemmer, K. Kormann, J. Kusch, R. G. McClarren, and J.-M. Qiu. A review of low-rank methods for time-dependent kinetic simulations. *J. Comput. Phys.*, 538:114191, 2025.
- [28] V. Grandgirard et al. A 5D gyrokinetic full- f global semi-Lagrangian code for flux-driven ion turbulence simulations. *Comput. Phys. Commun.*, 207:35–68, 2016.
- [29] M. Falcone and R. Ferretti. Convergence analysis for a class of high-order semi-Lagrangian advection schemes. *SIAM J. Numer. Anal.*, 35(3):909–940, 1998.
- [30] R. Ferretti. A Technique for High-Order Treatment of Diffusion Terms in Semi-Lagrangian Schemes. *Commun. Comput. Phys.*, 8(2):445–470, 2010.
- [31] R. Ferretti and M. Mehrenberger. Stability of semi-Lagrangian schemes of arbitrary odd degree under constant and variable advection speed. *Math. Comput.*, 89(324):1783–1805, 2020.
- [32] F. Filbet, É. Sonnendrücker, and P. Bertrand. Conservative Numerical Schemes for the Vlasov Equation. *J. Comput. Phys.*, 172(1):166–187, 2001.
- [33] M. A. Finzi, A. Potapczynski, M. Choptuik, and A. G. Wilson. A Stable and Scalable Method for Solving Initial Value PDEs with Neural Networks. In *The Eleventh International Conference on Learning Representations*, 2023.
- [34] T. G. Grossmann, U. J. Komorowska, J. Latz, and C.-B. Schönlieb. Can physics-informed neural networks beat the finite element method? *IMA J. Appl. Math.*, 89(1):143–174, 2024.
- [35] J. Guo, G. Domel, C. Park, H. Zhang, O. Can Gumus, Y. Lu, G. J. Wagner, D. Qian, J. Cao, T. J. R. Hughes, and W. K. Liu. Tensor-decomposition-based A Priori Surrogate (TAPS) modeling for ultra large-scale simulations. *Comput. Methods Appl. Mech. Eng.*, 444, 2025.
- [36] J. Guo, X. Xie, C. Park, H. Zhang, M. J. Politis, G. Domel, and W. K. Liu. Interpolating neural network-tensor decomposition (INN-TD): a scalable and interpretable approach for large-scale physics-based problems. In *Forty-second International Conference on Machine Learning*, 2025.
- [37] A. Hamiaz, M. Mehrenberger, H. Sellama, and É. Sonnendrücker. The semi-Lagrangian method on curvilinear grids. *Commun. Appl. Ind. Math.*, 7(3):99–137, 2016.

- [38] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, 2016.
- [39] M. W. Hirsch, S. Smale, and R. L. Devaney. *Differential Equations, Dynamical Systems, and an Introduction to Chaos*. Elsevier, 2013.
- [40] Q. Hong, J. W. Siegel, Q. Tan, and J. Xu. On the Activation Function Dependence of the Spectral Bias of Neural Networks. *preprint*, 2022.
- [41] S. Hu, M. Liu, S. Zhang, S. Dong, and R. Zheng. Physics-informed neural network combined with characteristic-based split for solving Navier-Stokes equations. *Eng. Appl. Artif. Intel.*, 128:107453, 2024.
- [42] Z. Hu, K. Shukla, G. E. Karniadakis, and K. Kawaguchi. Tackling the curse of dimensionality with physics-informed neural networks. *Neural Netw.*, 176:106369, 2024.
- [43] S. Jin, Z. Ma, and K. Wu. Asymptotic-Preserving Neural Networks for Multiscale Time-Dependent Linear Transport Equations. *J. Sci. Comput.*, 94(3), 2023.
- [44] M. Kast and J. S. Hesthaven. Positional embeddings for solving PDEs with evolutionary deep neural networks. *J. Comput. Phys.*, 508:112986, 2024.
- [45] T. G. Kolda and B. W. Bader. Tensor Decompositions and Applications. *SIAM Rev.*, 51(3):455–500, 2009.
- [46] K. Kormann. A Semi-Lagrangian Vlasov Solver in Tensor Train Format. *SIAM J. Sci. Comput.*, 37(4):B613–B632, 2015.
- [47] K. Kormann, K. Reuter, and M. Rampp. A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov-Poisson equation. *Int. J. High Perform. C.*, 33(5):924–947, 2019.
- [48] A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, and M. W. Mahoney. Characterizing possible failure modes in physics-informed neural networks. In *Thirty-fourth Conference on Neural Information Processing Systems*, 2021.
- [49] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. Neural Netw.*, 9(5):987–1000, 1998.
- [50] L. Á. Larios-Cárdenas and F. Gibou. Error-correcting neural networks for semi-Lagrangian advection in the level-set method. *J. Comput. Phys.*, 471:111623, 2022.
- [51] R. J. LeVeque. High-Resolution Conservative Algorithms for Advection in Incompressible Flow. *SIAM J. Numer. Anal.*, 33(2):627–665, 1996.
- [52] L. Li and C. Yang. APFOS-Net: Asymptotic preserving scheme for anisotropic elliptic equations with deep neural network. *J. Comput. Phys.*, 453:110958, 2022.
- [53] H. Liu, X. Cai, G. Lapenta, and Y. Cao. Conservative semi-Lagrangian kinetic scheme coupled with implicit finite element field solver for multidimensional Vlasov Maxwell system. *Commun. Nonlinear Sci.*, 102:105941, 2021.
- [54] Z. Liu and Y. Wang et al. KAN: Kolmogorov-Arnold Networks. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [55] Z. Mao and X. Meng. Physics-informed neural networks with residual/gradient-based adaptive sampling methods for solving partial differential equations with sharp solutions. *Appl. Math. Mech.*, 44(7):1069–1084, 2023.
- [56] X. Meng, Z. Li, D. Zhang, and G. E. Karniadakis. PPINN: Parareal physics-informed neural network for time-dependent PDEs. *Comput. Method. Appl. M.*, 370:113250, 2020.
- [57] S. Mishra and R. Molinaro. Physics informed neural networks for simulating radiative transfer. *J. Quant. Spectrosc. Radiat. Transfer*, 270:107705, 2021.

- [58] R. Mojjani, M. Balajewicz, and P. Hassanzadeh. Kolmogorov n -width and Lagrangian physics-informed neural networks: A causality-conforming manifold for convection-dominated PDEs. *Comput. Method. Appl. M.*, 404:115810, 2023.
- [59] J. Müller and M. Zeinhofer. Achieving high accuracy with PINNs via energy natural gradient descent. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 25471–25485. PMLR, 23–29 Jul 2023.
- [60] L. Nurbekyan, W. Lei, and Y. Yang. Efficient Natural Gradient Descent Methods for Large-Scale PDE-Based Optimization Problems. *SIAM J. Sci. Comput.*, 45(4):A1621–A1655, 2023.
- [61] C. Park, S. Saha, J. Guo, H. Zhang, X. Xie, M. A. Bessa, D. Qian, W. Chen, G. J. Wagner, J. Cao, and W. K. Liu. Interpolating neural network: A novel unification of machine learning and interpolation theory. *arXiv preprint arXiv:2404.10296*, 2024.
- [62] O. Pironneau. On the transport-diffusion algorithm and its applications to the Navier-Stokes equations. *Numer. Math.*, 38:309–332, 1982.
- [63] J.-M. Qiu and C.-W. Shu. Positivity preserving semi-Lagrangian discontinuous Galerkin formulation: theoretical analysis and application to the Vlasov–Poisson system. *J. Comput. Phys.*, 230(23):8386–8409, 2011.
- [64] A. Quarteroni, A. Manzoni, and F. Negri. *Reduced Basis Methods for Partial Differential Equations*. Springer International Publishing, 2016.
- [65] A. Quarteroni and A. Valli. *Numerical approximation of partial differential equations*, volume 23. Springer Science & Business Media, 2008.
- [66] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.*, 378:686–707, 2019.
- [67] M. Restelli, L. Bonaventura, and R. Sacco. A semi-Lagrangian discontinuous Galerkin method for scalar advection by incompressible flows. *J. Comput. Phys.*, 216(1):195–215, 2006.
- [68] J. A. Rossmann and D. C. Seal. A positivity-preserving high-order semi-Lagrangian discontinuous Galerkin scheme for the Vlasov–Poisson equations. *J. Comput. Phys.*, 230(16):6203–6232, 2011.
- [69] N. Schwenke and C. Furtlehner. ANaGRAM: A natural gradient relative to adapted model for efficient PINNs learning. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [70] É. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo. The Semi-Lagrangian Method for the Numerical Resolution of the Vlasov Equation. *J. Comput. Phys.*, 149(2):201–220, 1999.
- [71] A. Staniforth and J. Côté. Semi-Lagrangian Integration Schemes for Atmospheric Models—A Review. *Mon. Weather Rev.*, 119(9):2206–2223, 1991.
- [72] J. Stiasny and S. Chatzivasileiadis. Physics-informed neural networks for time-domain simulations: Accuracy, computational cost, and flexibility. *Electr. Pow. Syst. Res.*, 224:109748, 2023.
- [73] N. Sukumar and A. Srivastava. Exact imposition of boundary conditions with distance functions in physics-informed deep neural networks. *Comput. Method. Appl. M.*, 389:114333, 2022.
- [74] S. Särkkä and A. Solin. *Applied Stochastic Differential Equations*. Cambridge University Press, 2019.
- [75] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, and R. Ng. Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 7537–7547. Curran Associates, Inc., 2020.
- [76] S. Wang, B. Li, Y. Chen, and P. Perdikaris. PirateNets: Physics-informed Deep Learning with Residual Adaptive Networks. *J. Mach. Learn. Res.*, 25:1–51, 2024.

- [77] S. Wang, S. Sankaran, and P. Perdikaris. Respecting causality for training physics-informed neural networks. *Comput. Method. Appl. M.*, 421:116813, 2024.
- [78] C. Wu, M. Zhu, Q. Tan, Y. Kartha, and L. Lu. A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks. *Comput. Methods Appl. Mech. Engrg.*, 403:115671, 2023.
- [79] D. Xiu and G. E. Karniadakis. A Semi-Lagrangian High-Order Method for Navier-Stokes Equations. *J. Comput. Phys.*, 172(2):658–684, 2001.
- [80] C. Yang and M. Mehrenberger. Highly accurate monotonicity-preserving Semi-Lagrangian scheme for Vlasov-Poisson simulations. *J. Comput. Phys.*, 446:110632, 2021.
- [81] B. Zhang, G. Cai, H. Weng, W. Wang, L. Liu, and B. He. Physics-informed neural networks for solving forward and inverse Vlasov-Poisson equation via fully kinetic simulation. *Mach. Learn.: Sci. Technol.*, 4(4):045015, 2023.
- [82] N. Zheng, D. Hayes, A. Christlieb, and J.-M. Qiu. A Semi-Lagrangian Adaptive-Rank (SLAR) Method for Linear Advection and Nonlinear Vlasov-Poisson System. *J. Comput. Phys.*, page 113970, 2025.

Appendix A. Algorithm to compute the approximate characteristic curve

In this section, we explain how to actually compute the approximate characteristic curve $\tilde{\mathcal{X}}$, in Algorithm 3. In this algorithm, a time-stepping algorithm must be chosen to approximate the characteristic curve. We elect to use the RK4 (fourth-order Runge-Kutta) method, a well-known and widely used ODE solver. For some time step $\delta\tau$, some initial condition x at time τ , and some parameters μ , we denote by $\text{RK4}(\tau, \delta\tau, x, \mu)$ the RK4 approximation of the solution to the ODE (2.4) at time $\tau - \delta\tau$.

Algorithm 3 Computation of the characteristic curve for advection equations

- 1: **Input:** Final position x , parameters μ , final time t^{n+1} , initial time $t^n = t^{n+1} - \Delta t$, advection field a
 - 2: **Output:** Approximate foot $\tilde{\mathcal{X}}(t^n; t^{n+1}, x, \mu)$ of the characteristic curve
 - 3: **if** $a(\mu)$ is constant in space **then**
 - 4: Set $\tilde{\mathcal{X}}(t^n; t^{n+1}, x, \mu) = x - a(\mu)\Delta t$
 - 5: **else if** $a(x, \mu)$ is such that the ODE (2.4) has a closed-form solution $\mathcal{X}$ **then**
 - 6: Set $\tilde{\mathcal{X}}(t^n; t^{n+1}, x, \mu) = \mathcal{X}(t^n; t^{n+1}, x, \mu)$
 - 7: **else**
 - 8: Numerically solve the ODE (2.4) with initial condition x at time t^{n+1}
 - 9: **Initialization:** set $\tilde{\mathcal{X}} = x$, set $\delta\tau$ and n_τ such that $n_\tau\delta\tau = \Delta t$, set $\tau = t^{n+1}$
 - 10: **while** $\tau > t^n$ **do**
 - 11: Solve (2.4) using the RK4 method on the sub-time step $\delta\tau$: $\tilde{\mathcal{X}} \leftarrow \text{RK4}(\tau, \delta\tau, \tilde{\mathcal{X}}, \mu)$
 - 12: Update the sub-time step: $\tau \leftarrow \tau - \delta\tau$
 - 13: **end while**
 - 14: **end if**
 - 15: **Return:** $\tilde{\mathcal{X}}(t^n; t^{n+1}, x, \mu)$
-

Appendix B. Hyperparameters for the numerical experiments

This section regroups all the hyperparameters used in the numerical experiments. We denote by:

- N_c the number of collocation points, i.e., the number of points uniformly sampled to discretize the space domain Ω and approximate the integrals in the optimization problems, see Algorithm 1;
- N_e the number of epochs;
- Ξ the activation function;
- ω^{BC} (resp. ω^{IC}) the boundary (resp. initial) loss function coefficient, i.e., the coefficient by which the boundary (resp. initial) loss function is multiplied when weakly imposing the boundary (resp. initial) conditions in the PINN;

- N_e^{BC} (resp. N_e^{IC}) the number of collocation points in the boundary (resp. initial) integrals in the PINN;
- ℓ the list of layer sizes,
- λ the learning rate (note that no learning rate is reported when natural gradient preconditioning is deployed, since the default one of 1×10^{-3} is used).

Note that ω^{BC} or ω^{IC} being set to 0 means that the corresponding loss function is not used, and that the boundary or initial conditions are strongly imposed. Moreover, the activation function $\hat{h}$ corresponds to the following regularized hat function, see [40]:

$$\hat{h} : x \mapsto \exp \left(-12 \tanh \left(\frac{x^2}{2} \right) \right). \quad (\text{B.1})$$

For each experiment, we report the hyperparameters used:

- to train the PINN;
- to compute the initial dofs θ^0 for the dPINN, NG and NSL methods (in the column called “initialization”);
- to iterate the dPINN method, i.e., to solve the nonlinear optimization problem (2.8) at each time step;
- to iterate the NSL method, i.e., to solve the nonlinear optimization problem (3.3) at each time step.

While we did not run a full ablation or sensitivity analysis, we nevertheless provide some insights into the choice of hyperparameters for the NSL method. For both the initialization and the iterations, standard neural network training strategies are used to affect the different error terms ε_{int} , ε_{opt} and $\varepsilon_{\text{approx}}$. More collocation points will lower ε_{int} , while more epochs will lower ε_{opt} and additional layers will lower $\varepsilon_{\text{approx}}$.

- Correctly training the network to approximate the initial condition is the most important step for the NSL method. If the initial condition is not well-approximated, then the time iterations will also not provide a good approximation. This is why a larger number N_e of epochs is often used for the initialization than for the iterations.
- The number of collocation points N_c is usually similar for the initialization and the iterations, except when the solution becomes more complex as time progresses. In this case, N_c may be increased for the iterations.
- Especially when natural gradient preconditioning is used, the number of layers can be kept small, as even small networks are very expressive if they are well-trained.
- Finally, the activation function Ξ is usually chosen to be a tanh function, except for oscillatory solutions where a sin function is used, or for localized solutions benefiting from a regularized hat function $\hat{h}$.

Hyperparameter	PINN	initialization	dPINN iterations	NSL iterations
λ	9×10^{-3}	1.5×10^{-2}	1.5×10^{-2}	1.5×10^{-2}
N_e	1 000	1 500	200	150
N_c	2 000	3 000 (5 000 for dPINN)	5 000	1 000
ℓ	[40, 40, 40]	[40, 40, 40]	N/A	N/A
Ξ	tanh	tanh	N/A	N/A
ω^{BC}	50	N/A	N/A	N/A
N_e^{BC}	1 000	N/A	N/A	N/A
ω^{IC}	0	N/A	N/A	N/A
N_e^{IC}	0	N/A	N/A	N/A

Table B.9: Hyperparameters used in [Section 4.1.1](#).

Hyperparameter	PINN	initialization	dPINN iterations	NSL iterations
λ	9×10^{-3}	1.5×10^{-2}	5×10^{-3}	5×10^{-3}
N_e	3 000	2 500	2 500	2 500
N_c	10 000	3 000 (10 000 for dPINN)	1 000	10 000
ℓ	[40, 60, 80, 60, 40]	[40, 60, 80, 60, 40]	N/A	N/A
Ξ	tanh	tanh	N/A	N/A
ω^{BC}	50	N/A	N/A	N/A
N_c^{BC}	1 000	N/A	N/A	N/A
ω_c^{IC}	0	N/A	N/A	N/A
N_c^{IC}	0	N/A	N/A	N/A

Table B.10: Hyperparameters used in [Section 4.1.2](#).

Hyperparameter	PINN	NG and NSL initialization	NSL iterations
λ	1×10^{-2}	1×10^{-3}	5×10^{-3}
N_e	5 000	700	500
N_c	25 000	15 000	15 000
ℓ	[80, 80, 80, 80]	[40, 40, 40, 40, 40]	N/A
Ξ	$\hat{h}$	$\hat{h}$	N/A
ω^{BC}	0	N/A	N/A
N_c^{BC}	0	N/A	N/A
ω_c^{IC}	50	N/A	N/A
N_c^{IC}	15 000	N/A	N/A

Table B.11: Hyperparameters used in [Section 4.2.1](#).

Hyperparameter	PINN	NG and NSL initialization	NSL iterations
λ	1×10^{-2}	1×10^{-3}	5×10^{-3}
N_e	1 000	250	500
N_c	15 000	5 000	15 000
ℓ	[80, 80, 80, 80]	[40, 40, 40, 40, 40]	N/A
Ξ	$\hat{h}$	$\hat{h}$	N/A
ω^{BC}	50	N/A	N/A
N_c^{BC}	5 000	N/A	N/A
ω_c^{IC}	50	N/A	N/A
N_c^{IC}	1 000	N/A	N/A

Table B.12: Hyperparameters used in [Section 4.2.2](#).

Hyperparameter	Section 4.4.1	Section 4.4.2	Section 4.4.3
N_e	250	250	500
N_c	60 000	64^3	48^3
ℓ	[35, 50, 35]	[35, 50, 50, 35]	see legend
Ξ	tanh	tanh	tanh

Table B.13: Hyperparameters used in [Sections 4.4.1](#) to [4.4.3](#). For the 3D level-set deformation without and with parameters ([Sections 4.4.2](#) and [4.4.3](#)), the number of epochs in the first time step is increased to 500 and 1000 respectively. For the 3D level-set deformation with parameters, a residual network (see [\[38\]](#)) is employed, with 9 layers of 26 neurons each, with skip connections between layers 1 and 3, 4 and 6, and 7 and 9.

Hyperparameter	Network for u	Network for Ψ
$N_e^{\text{init.}}$	500	350
$N_e^{\text{iter.}}$	100	50
N_c	112^2	768
ℓ	$[30] \times 9$	$[20] \times 6$
Ξ	sin	sin

Table B.14: Hyperparameters used in [Section 4.6.2](#). For both residual networks, skip connections skip one layer (they run from layers 1 to 3 and 4 to 6 for both u and Ψ , and 7 to 9 for u).