Computer Methods in Applied Mechanics and Engineering 418 (2024) 116486

Contents lists available at ScienceDirect

Comput. Methods Appl. Mech. Engrg.

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

A shallow physics-informed neural network for solving partial differential equations on static and evolving surfaces

Wei-Fan Hu [a,c,*], Yi-Jun Shih [b], Te-Sheng Lin [b,c], Ming-Chih Lai [b]

[a] Department of Mathematics, National Central University, Taoyuan 32001, Taiwan
[b] Department of Applied Mathematics, National Yang Ming Chiao Tung University, Hsinchu 30010, Taiwan
[c] National Center for Theoretical Sciences, National Taiwan University, Taipei 106216, Taiwan

ARTICLE INFO

Keywords:
Physics-informed neural networks
Surface partial differential equations
Laplace–Beltrami operator
Shallow neural network
Evolving surfaces

ABSTRACT

In this paper, we introduce a shallow (one-hidden-layer) physics-informed neural network (PINN) for solving partial differential equations on static and evolving surfaces. For the static surface case, with the aid of a level set function, the surface normal and mean curvature used in the surface differential expressions can be computed easily. So, instead of imposing the normal extension constraints used in literature, we write the surface differential operators in the form of traditional Cartesian differential operators and use them in the loss function directly. We demonstrate a series of performance study for the present methodology by solving Laplace–Beltrami equations and surface diffusion equations on complex static surfaces. With just a moderate number of neurons used in the hidden layer, we are able to attain satisfactory prediction results. We then extend the present methodology to solve the advection–diffusion equation on an evolving surface with a given velocity. To track the deforming surface, we additionally introduce a network, in which a prescribed hidden layer is employed to enforce the topological structure of the surface and learn the homeomorphism between the surface and the prescribed topology. The proposed network structure is designed to track the surface and solve the equation simultaneously. Again, the numerical results show comparable accuracy as the static cases. As an application, we simulate surfactant transportation on a droplet surface under shear flow and obtain some physically plausible results.

1. Introduction

Surface partial differential equations (PDEs) arise in a wide variety of scientific and engineering applications. These equations are formulated in terms of differential operators acting on curved surfaces. Mathematically, they are examples of partial differential equations on manifolds. Problems of interest include, for example, the modeling of surface-active agents [1], deforming vesicles [2], cell motility and chemotaxis [3], bio-membranes [4], restoring a damaged pattern on surfaces [5], image processing [6], and computer graphics [7].

Solving PDEs on surfaces is certainly of major interest among the scientific computing community. The fundamental difficulty comes from the numerical approximation of differential operators along a surface. This long-standing problem has been explored by many researchers for decades. For instance, the surface finite element method [8,9] is designed specifically for PDEs on discretized triangular surfaces. However, generating those triangulation nodes can be time-consuming, and the accuracy of the method is significantly affected by the quality of triangulations. For geometric PDEs and high-order PDEs, spline-based methods [10,11] can

* Corresponding author at: Department of Mathematics, National Central University, Taoyuan 32001, Taiwan.
E-mail addresses: wfhu@math.ncu.edu.tw (W.-F. Hu), tslin@math.nctu.edu.tw (T.-S. Lin), mclai@math.nctu.edu.tw (M.-C. Lai).

https://doi.org/10.1016/j.cma.2023.116486
Received 7 July 2023; Received in revised form 3 September 2023; Accepted 23 September 2023

be used to accurately discretize the partial differential equations on evolving surfaces. Using parametric representation is another natural idea [12,13]. For instance, the solution along a smooth genus-zero surface can be represented by a spherical harmonics expansion. However, it may suffer from the intrinsic singularities that are built into the PDE formulation (e.g., poles in spherical coordinates) or in the boundary integral kernel involving Green's function. Thus, it needs careful numerical treatments near the singularities. A mesh-free approach called radial basis functions (RBFs) method [14,15] works by first representing the solution as a linear combination of RBFs, and then substituting the approximation expression at some chosen collocation points into the differential equation directly. As a result, a dense linear system of coefficients must be solved which is likely to be ill-conditioned. In order to obtain a well-conditioned resultant matrix and achieve the desired accuracy, it is often necessary to manually adjust the shape parameter appearing in a certain type of radial basis functions. Nonetheless, identifying the parameters to reach optimal results remains an issue in the usage of radial basis functions.

On the other hand, embedding techniques aim to solve PDEs in a small band in the vicinity of a surface, examples including level set method [16], closest point method [17–19], or grid-based particle method [20,21]. The underlying surface PDE is alternatively represented in Eulerian coordinates and thus surface derivatives are replaced by projections of derivatives in the embedding Euclidean space. In such a way, the difficulties such as parameterized or triangulated surfaces can be avoided. Despite the geometrically flexible nature of these methods, practical implementation often requires the establishment of appropriate boundary conditions at the band's edges, which remains unclear. Besides, these methods involve the identification of surface projection points for regular Cartesian grids, which adds extra computational effort and can be challenging, especially when dealing with highly oscillatory surfaces.

To the best of our knowledge, only a few works exist using machine learning for solving PDEs on surfaces. Following the same spirit as those in embedding techniques, Fang et al. [22,23] adopt the physics-informed neural networks (PINNs) [24] framework to solve the Laplace–Beltrami equation (stationary) and diffusion equation (time-dependent) on static surfaces. The neural network solution is constrained to have zero normal derivatives at given training points along the surface. This restriction leads to an approximate normal extension solution in a narrow band of the surface, so the Laplace–Beltrami operator is replaced by the conventional Laplace operator. As a consequence, the PINNs loss function penalizes the equation residual, together with normal and second-order derivatives. In such a way, the PDE information only comes from the training points given on the surface, and thus differs significantly from those embedding techniques. Furthermore, it is completely mesh-free, unlike the aforementioned grid-based embedding methods.

While the loss function seems to be legitimate, the numerical experiments shown in these works [22,23] have relative L^2 errors more than 1% even when deep neural networks are used. So, instead of using the above loss function, in this paper, we write the surface differential operators in the form of traditional Cartesian differential operators and use them in the loss function directly. Thus, we can encode the entire embedding PDE without imposing the normal extension constraint. Besides, we adopt a completely shallow (one-hidden-layer) neural network under PINNs framework, so our model is easy to implement and train. As discussed in [25–27], a shallow neural network can theoretically approximate smooth functions and their derivatives accurately. This is the legitimate reason why it can help to solve PDEs in the first place. The shallow PINNs (or Ritz) method with augmented inputs have been proven very effective for solving elliptic interface problems with jump discontinuities, see the authors' recent papers in [28–30].

Until very recently, Tang et al. [31] proposed a methodology that exactly shares the same spirit as ours, i.e., embedding the solution into Eulerian coordinates and expressing surface differential operators by conventional Cartesian ones in the PINN loss. However, their numerical experiments (for solving the stationary advection–diffusion equation) adopt the deep network architecture (depth = 4 and width = 50), resulting in numerous parameters to be learned, to reach relative L^2 error of magnitude 10^{-4} for some smooth solutions. By contrast, we simply use shallow network structures with 60 neurons that is able to attain satisfactory prediction results with errors around 10^{-6} for the Laplace–Beltrami equation and 10^{-5} for the surface diffusion equation. Furthermore, we have extended our methodology to the evolving surface cases, whereas [31] only focuses on static surfaces.

The rest of the paper is organized as follows. In Section 2, we first describe a shallow PINNs model to solve stationary PDEs (by taking Laplace–Beltrami equation as an example) on a static surface and perform a series of numerical accuracy tests and comparisons. Then, we develop the network solver to solve time-dependent PDEs (by taking diffusion equation with a source term as an example) and also demonstrate its capability for finding solutions on complex surfaces in Section 3. In Section 4, we extend the present methodology to solve the advection–diffusion equation on a 2D evolving surface in $\mathbb{R}^3$. Some concluding remarks and future works are given in Section 5.

2. Stationary PDEs on surfaces

Denoting a regular (or smooth) surface by Γ embedded in Euclidean space $\mathbb{R}^3$, the considered PDEs take the general form

$$\mathcal{L}(u) = f \quad \text{on } \Gamma, \tag{1}$$

where, for simplicity, Γ is assumed to be a closed surface. The operator $\mathcal{L}$ may consist of common differential terms related to the surface geometry, such as surface gradient $\nabla_s u$, surface divergence $\nabla_s \cdot \mathbf{v}$ for some vector field $\mathbf{v}$, or Laplace–Beltrami (or surface Laplace) operator $\Delta_s u$. With a suitable surface parametric representation, these differential operators can be evaluated via first and second fundamental forms of differential geometry [32]. For the case where the surface is not closed, some suitable boundary conditions along $\partial\Gamma$ must be given. Nevertheless, the boundary condition does not change the main ingredient of the present methodology (see next subsection).

As aforementioned, the differential operator $\mathcal{L}$ can be computed using a local parametrization, say $u = u(\theta, \phi)$, where θ and ϕ are surface parameters. However, numerical differentiations of $\mathcal{L}(u)$ using surface parametrization might cause severe numerical instability. For instance, if the considered surface geometry is complicated (a stationary highly oscillatory surface case), or if the discretized Lagrangian points are clustered in certain parts of the surface (a time-evolving surface case), this may lead to inaccurate derivative calculations [33]. For the latter case, a reparametrization technique [33,34] is often required to redistribute those markers on the surface to maintain the numerical accuracy and stability. Alternatively, one can utilize the arbitrary Lagrangian–Eulerian formulation [35–37] to avoid node clustering.

Our goal is to develop a robust *mesh-free* numerical method for solving PDEs (1) based on a neural network learning technique. To compute $\mathcal{L}(u)$, rather than using surface parametrization, here, we adopt an alternative way using conventional differential operators. To this end, the solution u defined on the surface is now regarded as an embedded function, $u(x, y, z)$, in the Eulerian space that satisfies $u(x, y, z) = u(\theta, \phi)$ when $(x, y, z) \in \Gamma$. Although this assumption leads to the solution having one higher dimension in the variable space (surface coordinates (θ, ϕ) to Cartesian coordinates (x, y, z)), the surface differential terms in $\mathcal{L}$ can be rewritten via conventional differential operators in Eulerian coordinates. More precisely, at a given point $\mathbf{x} = (x, y, z) \in \Gamma$, we have

$$\begin{aligned} &\nabla_s u = (I - \mathbf{n}\mathbf{n}^T)\nabla u, \\ &\nabla_s \cdot \mathbf{v} = \nabla \cdot \mathbf{v} - \mathbf{n}^T(\nabla \mathbf{v})\mathbf{n}, \\ &\Delta_s u = \Delta u - 2H\partial_n u - \mathbf{n}^T(\nabla^2 u)\mathbf{n}. \end{aligned} \tag{2}$$

Here, $\mathbf{n} = \mathbf{n}(\mathbf{x})$ is the unit outward normal vector on Γ, $H = H(\mathbf{x})$ is the mean curvature, $\partial_n u = \nabla u \cdot \mathbf{n}$ denotes the normal derivative, and $\nabla^2 u$ is the Hessian matrix of u. The derivation of above identities can be found in Appendix. Note that, both normal vector and mean curvature in the above formulas can be directly computed once the level set function representation ψ of the surface Γ is available. That is, at $\mathbf{x} \in \Gamma$ (so the level set $\psi(\mathbf{x}) = 0$ represents Γ), the above two geometric quantities can be computed by

$$\mathbf{n} = \frac{\nabla \psi}{\|\nabla \psi\|} \quad \text{and} \quad 2H = \nabla \cdot \mathbf{n} = \frac{\mathrm{tr}(\nabla^2 \psi) - \mathbf{n}^T(\nabla^2 \psi)\mathbf{n}}{\|\nabla \psi\|}, \tag{3}$$

where $\mathrm{tr}(\cdot)$ gives the trace of a matrix and $\|\cdot\|$ denotes the standard Euclidean norm.

Throughout the rest of this section, we will only focus on the Laplace–Beltrami equation as

$$\Delta_s u(\mathbf{x}) = f(\mathbf{x}) \quad \text{on } \Gamma, \tag{4}$$

where we deliberately put the variable $\mathbf{x}$ to clarify that the differential equation is defined in Eulerian coordinates. In addition, it is important to mention that, there exists infinitely many embedded functions $u(\mathbf{x})$ whose restriction on Γ serves as a solution to Eq. (4) (or more generally, Eq. (1)), so that such embedded solutions can be representable in a wide range of neural network approximator, thanks to the expressive power of universal approximation theory [25,26].

2.1. Physics-informed learning machinery using shallow neural network approximation

With the expressive capabilities of neural networks [26], we hereby construct a simple feedforward, fully-connected, shallow (one-hidden-layer) neural network solution $u_{\mathcal{N}}$ as

$$u_{\mathcal{N}}(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(W_j \mathbf{x}^T + b_j). \tag{5}$$

Here, σ is the activation function, and N is the number of employed neurons in that hidden layer. The total number of learnable (or trainable) parameters N_p is the sum of the numbers of weights (denoted by $W_j \in \mathbb{R}^{1\times 3}$, $\alpha_j \in \mathbb{R}$) and biases (denoted by $b_j \in \mathbb{R}$). Since only one hidden layer is employed, the total number of trainable parameters is $N_p = 5N$. Notice that, the output layer in the present network structure is assumed to be unbiased, so the network output can be concisely written in the form of a finite linear combination of activation functions.

Let us describe the methodology of physics-informed learning machinery [24] for solving Eq. (4) as follows. With a given training set $\{\mathbf{x}^i = (x^i, y^i, z^i) \in \Gamma\}_{i=1}^{M}$, the neural net parameters (weights and biases in Eq. (5)) are learned via minimizing the mean squared error of the differential equation residual

$$\mathrm{Loss}(\mathbf{p}) = \frac{1}{M} \sum_{i=1}^{M} \left[\Delta_s u_{\mathcal{N}}(\mathbf{x}^i; \mathbf{p}) - f(\mathbf{x}^i)\right]^2,$$

where $\mathbf{p}$ is a vector collecting all trainable parameters (of dimension N_p) . Using the third identity in Eq. (2), it is natural to choose the loss function as

$$\mathrm{Loss}_{\Delta_s}(\mathbf{p}) = \frac{1}{M} \sum_{i=1}^{M} \left[\Delta u_{\mathcal{N}}(\mathbf{x}^i) - 2H(\mathbf{x}^i)\partial_n u_{\mathcal{N}}(\mathbf{x}^i) - \mathbf{n}(\mathbf{x}^i)^T \left(\nabla^2 u_{\mathcal{N}}(\mathbf{x}^i)\right) \mathbf{n}(\mathbf{x}^i) - f(\mathbf{x}^i)\right]^2, \tag{6}$$

where we have dropped the notation $\mathbf{p}$ in $u_{\mathcal{N}}$ for succinct purpose. The first- and second-order partial derivatives of $u_{\mathcal{N}}$ involved in the above loss can be evaluated via auto-differentiation [38], or, derived explicitly through the network expression (5) thanks to the simplicity of the shallow network structure. We remark that the explicit evaluations of partial derivatives can be done more efficiently than the auto-differentiation since the latter requires multiple runs of backpropagation.

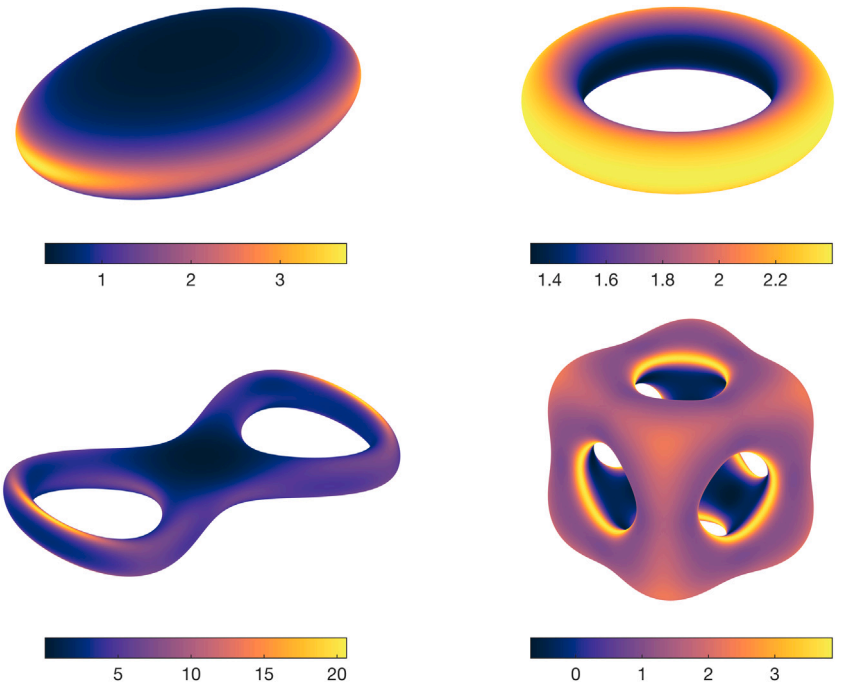

Fig. 1. Shapes of ellipsoid and torus (top row), genus-2 torus and cheese-like (bottom row). The color code denotes the magnitude of mean curvature H.

Here, we should point out that the following loss function is used in [23]

$$\text{Loss}_{\Delta}(\mathbf{p}) = \frac{1}{M}\sum_{i=1}^{M}\left[\Delta u_{\mathcal{N}}(\mathbf{x}^i) - f(\mathbf{x}^i)\right]^2 + \frac{1}{M}\sum_{i=1}^{M}\left[\partial_n u_{\mathcal{N}}(\mathbf{x}^i)\right]^2 + \frac{1}{M}\sum_{i=1}^{M}\left[\mathbf{n}(\mathbf{x}^i)^T\left(\nabla^2 u_{\mathcal{N}}(\mathbf{x}^i)\right)\mathbf{n}(\mathbf{x}^i)\right]^2, \tag{7}$$

which is inspired by the inequality

$$|\Delta_s u - f| \le |\Delta u - f| + |2H||\partial_n u| + |\mathbf{n}^T(\nabla^2 u)\mathbf{n}|, \tag{8}$$

that is also a direct result from the third identity in Eq. (2). One can see that their idea of designing loss_{Δ} in Eq. (7) is to penalize each term on the right-hand side of the above inequality. This will generally result in a normal extension solution (in a very narrow region) since it is attempted to enforce $\partial_n u_{\mathcal{N}} = 0$ on Γ. Like our proposed Loss_{Δ_s} in Eq. (6), second partial derivatives are still required in Eq. (7). One favorable feature of Loss_{Δ} is to avoid computing the local mean curvatures at training points which will save some computational efforts. However, since the normal derivative term $\partial_n u$ in the inequality (8) is multiplied by the factor $|2H|$, this will significantly influence the upper bound of the actual differential equation residual using loss_{Δ} (7). In next subsection, we will demonstrate that our proposed loss function (6) indeed outperforms the splitting residual loss (7) in the sense of higher predictive accuracy regardless of the surface geometries. We also point out that since only one hidden layer with moderate number of neurons employed in the present network, the computational complexity and learning workload can be significantly reduced without sacrificing the accuracy.

2.2. Numerical results

Here, we use the established network model to perform a series of numerical tests for Laplace–Beltrami equation. We consider four different surfaces Γ which can be represented by the zero level sets of ψ as follows.

- Ellipsoid : $\psi(x, y, z) = (x/1.5)^2 + (y)^2 + (z/0.5)^2 - 1$
- Torus: $\psi(x, y, z) = (\sqrt{x^2 + y^2} - 1)^2 + z^2 - 1/16$
- Genus-2 torus: $\psi(x, y, z) = \left[(x + 1)x^2(x - 1) + y^2\right]^2 + z^2 - 0.01$
- Cheese-like surface: $\psi(x, y, z) = (4x^2 - 1)^2 + (4y^2 - 1)^2 + (4z^2 - 1)^2 + 16(x^2 + y^2 - 1)^2 + 16(x^2 + z^2 - 1)^2 + 16(y^2 + z^2 - 1)^2 - 16$

As mentioned before, the normal vector and mean curvature used in the computation of Laplace–Beltrami operator Δ_s can be exactly obtained through symbolic differentiation in Eq. (3). The shapes of these surfaces and corresponding local mean curvatures are shown in Fig. 1.

We should note that the solution to Laplace–Beltrami equation is unique up to an arbitrary additive constant. To assess the accuracy of our method, the obtained network solution $u_{\mathcal{N}}$ is shifted to have the same value of the exact solution at a given point.

Table 1
The average relative L^2 errors with different sizes of neurons N in the hidden layer. For each case the number of training points is fixed by $M = 400$.

N	Loss_{Δ_s} (6), present work			Loss_{Δ} (7), proposed in [23]		
	ADAM	L-BFGS	LM	ADAM	L-BFGS	LM
20	2.070E−04	9.500E−05	6.841E−06	7.148E−02	9.453E−02	9.774E−02
30	2.959E−04	1.009E−04	1.837E−06	5.199E−02	5.422E−02	4.390E−02
40	1.260E−04	9.376E−05	3.780E−07	4.300E−02	4.218E−02	3.304E−02

Throughout all numerical tests in this paper, we choose sigmoid as the activation function. We generate a set of collecting points on Γ via the usage of `DistMesh` package developed in [39] wherein level set function related to target surface is required as an input. We then randomly pick training points $\{\mathbf{x}^i\}$ in that point cloud set. To train the network model, we adopt the Levenberg–Marquardt (LM) method [40] (except the below discussion on the comparison between different popular optimizers) that can effectively find the optimal parameters for losses of mean squared type. After the training process is finished, we measure the accuracy of the solution using the test error instead of the training error. That is, we randomly choose M_{test} testing points on Γ by computing the relative error in L^2 norm as

$$\frac{\|u_{\mathcal{N}} - u\|_2}{\|u\|_2} = \sqrt{\sum_{i=1}^{M_{test}} \left(u_{\mathcal{N}}(\mathbf{x}^i) - u(\mathbf{x}^i)\right)^2} \Big/ \sqrt{\sum_{i=1}^{M_{test}} \left(u(\mathbf{x}^i)\right)^2}.$$

For each case, we set $M_{test} \sim 10^4$. Notice that we initialize the network parameters in LM method by the standard normal distribution. It was found that the randomness in the initialization only leads to slightly different prediction results and nearly the same computational training time for each experiment. For each test, we repeat the numerical runs five times so the test error reported here is the averaged one. All trials are ran on an iMac equipped with an intel i7 CPU. We implement the proposed PINN model using MATLAB.

In the following, we aim to analyze the performance of our proposed method. We quantify the prediction accuracy through a series of virtual experiments, including the comparisons of loss functions and optimizers, and single and double precision computations. We also study the effects on the number of training points and the depth of network architecture. In the above tests, the ellipsoidal surface is considered, along which the exact solution is chosen as $u(x, y, z) = \sin(x)\cos(y - z)$, so the corresponding right-hand side function $f(x, y, z)$ can be computed directly by substituting u into Eq. (4). Furthermore, we also apply the present method to a non-closed surface case (the boundary condition is taken into account) and other more complex surfaces described earlier.

Comparisons of loss functions and optimizers. First, we perform the accuracy comparison between our proposed model and existing method in [23] (i.e., the usage of loss function (7)). We fix $M = 400$ training points and train the model using several popular optimizers, such as ADAM [41], L-BFGS [42], and LM method. The results are reported in Table 1, in which the relative L^2 errors are shown for $N = 20, 30, 40$ neurons used in the hidden layer. From the left panel, one can see that the testing accuracy of the present loss model (Loss_{Δ_s}) is quite satisfactory (at least 0.01% predictive accuracy) among all optimizers, showing a good approximation capability to the solution for the network model. One can also see that only the results obtained by LM algorithm show convergence tendency with increasing N. This is because the LM algorithm, a quadratic convergence method particularly designed for nonlinear least squares problems, generally seeks a local minimum in a faster decaying rate than the other two methods. As a result, the local minimum found by LM optimizer in general has smaller training loss, and thus achieves higher prediction accuracy. See the time history of training loss for these three optimizers in Fig. 2.

We also check the testing accuracy using the loss function Loss_{Δ} in (7) proposed in [23], and show the results in the right panel of Table 1. One can immediately see how significantly different those relative L^2 errors are compared with the results in left panel (10^{-2} versus 10^{-7} for the case of $N = 40$ with LM optimizer). And all errors obtained by Loss_{Δ} are apparently greater than 1% no matter which optimizer is adopted. This result indicates that, the requirement $\partial_n u_{\mathcal{N}} = 0$ at points along Γ gives rise to a locally normal extension solution (in a small neighborhood) which might be complicated, and thus the network model may require more neurons or deeper network structure to be employed to have an accurate prediction. We further run a series of tests with various exact solutions following the same setup in Table 1. It turns out that same tendency is observed for both models (not shown here). When other complex surfaces are considered, our model is still able to achieve good predictive accuracy (see later in this subsection) whereas the loss function seeking normal extension predicts much less accurate solution (these results are not shown here). Based on this finding, we conclude that, with the full expression of differential operators in the loss function, the embedded solution can be accurately expressed under the present shallow neural network.

Comparison of single and double precision computations. Table 2 reports an extensive study on the comparison between the single and double precision computations. We vary the number of neurons in the hidden layer and evaluate the relative L^2 error between the exact and predicted solutions, terminal loss values, and CPU time (in seconds). In each run, the network model is trained up to 4000 steps, while the number of training points is fixed by $M = 400$. The results show that for both floating-point representations, given enough training points, the prediction accuracy increases with the number of neurons used.

When $N = 20$, the loss value obtained using single precision reaches around 10^{-9}, which is the limit of single precision calculation. Further increasing the number N does not reduce the loss so the error is stuck around 10^{-5}. For double precision

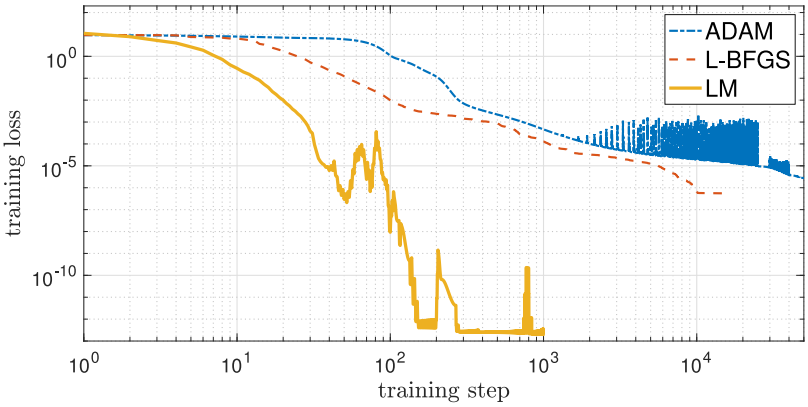

Fig. 2. Time history of training Loss_{Δ_s} with $N = 40$ using different optimizers. Dash–dotted line: ADAM, dashed line: L-BFGS, solid line: LM. All training processes use $M = 400$ training points.

Table 2
The average relative L^2 errors, loss values, and CPU time (in seconds) with different number of neurons N in the hidden layer. For each case, the number of training points is fixed by $M = 400$ and the training procedure is terminated at 4000 steps.

(N, N_p)	Double precision			Single precision		
	Error	Loss	CPU time (s)	Error	Loss	CPU time (s)
(5, 25)	6.524E−04	1.799E−05	27	6.967E−04	2.309E−05	26
(10, 50)	6.237E−05	7.285E−07	31	9.746E−05	6.320E−07	27
(20, 100)	6.841E−07	4.875E−11	36	1.349E−05	4.899E−09	30
(40, 200)	3.780E−07	1.429E−12	43	1.314E−05	2.300E−09	38
(80, 400)	2.082E−07	1.384E−13	67	1.398E−05	2.093E−09	49
(160, 800)	1.267E−07	1.622E−13	134	9.047E−06	2.179E−09	84

Table 3
The average relative L^2 errors for the shallow neural network with $N = 40$ using different number of training points M.

M	Error	Loss
100	1.423E−04	3.427E−08
200	1.958E−06	9.978E−11
300	5.228E−07	4.777E−12
400	3.780E−07	1.429E−12
500	2.218E−07	1.942E−13

computation, increasing the network complexity beyond $N = 20$ gives a low convergence rate. This is because we stop the training process at 4000 iterations, and the loss may not reach its theoretical minimum. We observe that when the loss is small, its value decays slowly during training, thus requires much more training steps to make the loss smaller.

Effect on the number of training points. Next, we investigate the effect on the number of training points. In Table 3, we deploy $N = 40$ neurons in the hidden layer, and minimize the loss model with the number of training points ranging from $M = 100$ to $M = 500$ (this can be roughly regarded as increasing the spatial resolution in traditional numerical methods). As can be seen, given a small bunch of training data $M = 100$ only results in the accuracy around 10^{-4} with the loss value around 10^{-8}. When the loss model is given by the enough information, namely, sufficient number of training points, the network is capable of reaching higher predictive accuracy around 10^{-7} with the loss around 10^{-12}.

Effect on the depth of network architecture. We investigate the performance of multiple-hidden-layer network architectures. With fixed number of training point $M = 400$, we investigate the prediction results using the two-hidden-layer network, which employs N neurons per hidden layer, written as

$$u_{\mathcal{N}}(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(W_j^{[2]} \sigma(W^{[1]}\mathbf{x}^T + \mathbf{b}^{[1]}) + b_j^{[2]}),$$

where the weights $\alpha_j \in \mathbb{R}$, $W^{[1]} \in \mathbb{R}^{N\times 3}$ and $W_j^{[2]} \in \mathbb{R}^{1\times N}$, the biases $\mathbf{b}^{[1]} \in \mathbb{R}^N$ and $b_j^{[2]} \in \mathbb{R}$. The total number of learnable parameters is thus counted as $N_p = N^2 + 6N$. From Table 4 we can see that, for the number of learnable parameters $N_p = 160, 520, 1840$, the two-hidden-layer network attains equally good accuracy around 10^{-7} in comparison to the shallow one with

Table 4
The average relative L^2 errors for the two-hidden-layer neural network. In each case the total number of learnable parameters is counted by $N_p = N^2+6N$ and the training data points is fixed by $M = 400$.

(N, N_p)	Error	Loss
(10, 160)	2.347E−07	2.054E−12
(20, 520)	3.418E−07	2.617E−13
(40, 1840)	1.316E−07	1.170E−13
(80, 6880)	5.445E−08	2.582E−14

Table 5
The average relative L^2 errors. For each case the number of training points is fixed by $M = 400$ and $M_b = 100$.

(N, N_p)	Error
(5, 25)	8.903E−04
(10, 50)	6.598E−05
(20, 100)	1.828E−06
(40, 200)	1.735E−07
(80, 400)	8.158E−08
(160, 800)	7.697E−08

$N_p = 200$, refer the case $N = 40$ and $M = 400$ in Table 3. When $N = 80$ is used in each hidden layer, the prediction accuracy reaches around 10^{-8} and the loss value decays as low as 10^{-14}. However, this small improvement of accuracy requires a large number of parameters $N_p = 6880$ needed to be trained. Thus, the usage of shallow neural network representation is readily able to encode smooth solutions, and the accuracy performance is equally well compared to the two-hidden-layer network.

Application to a non-closed surface. When the considered surface is not closed, the underlying PDE must be subject to an additional boundary condition along $\partial\Gamma$. Here, we consider the Dirichlet-type boundary condition $u(\mathbf{x}) = u_b(\mathbf{x})$ for $\mathbf{x} \in \partial\Gamma$, so, it is straightforward to simultaneously enforce mean squared errors for both differential equation and boundary condition in a loss function. That is, given training sets $\{\mathbf{x}^i \in \Gamma\}_{i=1}^{M}$ and $\{\mathbf{x}^j_{\partial\Gamma} \in \partial\Gamma\}_{j=1}^{M_b}$, the loss function (6) is thus slightly modified with an additional penalty term as

$$\begin{aligned}\mathrm{Loss}_{\Delta_s}(\mathbf{p}) = &\frac{1}{M}\sum_{i=1}^{M}\left[\Delta u_{\mathcal{N}}(\mathbf{x}^i) - 2H(\mathbf{x}^i)\partial_n u_{\mathcal{N}}(\mathbf{x}^i) - \mathbf{n}(\mathbf{x}^i)^T\left(\nabla^2 u_{\mathcal{N}}(\mathbf{x}^i)\right)\mathbf{n}(\mathbf{x}^i) - f(\mathbf{x}^i)\right]^2\\ &+ \frac{1}{M_b}\sum_{j=1}^{M_b}\left[u_{\mathcal{N}}(\mathbf{x}^j_{\partial\Gamma}) - u_b(\mathbf{x}^j_{\partial\Gamma})\right]^2.\end{aligned}$$

We run a test example whose solution is chosen as $u(x, y, z) = \sin(x)\exp(\cos(y-z))$ and the hemi-ellipsoid $\psi(x, y, z) = (x/1.5)^2 + (y)^2 + (z/0.5)^2 - 1$ with $z > 0$. In Table 5, with fixed number of training points $M = 400$ and $M_b = 100$, we show the prediction accuracy with different number of neurons N in the hidden layer. As seen, with the presence of boundary conditions, the proposed model is still able to attain satisfactory accuracy. Again, given a sufficient number of training points M and M_b, the prediction accuracy increases as the number N increases.

Numerical results of more complex surfaces. In the previous tests, we only focus on the surface geometry as simple as an ellipsoid (or hemi-ellipsoid). Here, we present the numerical accuracy results for our proposed neural network using the loss function (6) with more complex geometries such as torus, genus-2 surface, and cheese-like surface (see Fig. 1).

In Table 6 we show the average relative L^2 errors and CPU time (in seconds) for those different surfaces. Again, we choose the exact solution as $u(x, y, z) = \sin(x)\exp(\cos(y - z))$; we fix $M = 400$ training points which are randomly deployed along each surface and train the model up to 500 steps. The number of neurons used in the hidden layer is varied from $N = 20, 30, 40, 50, 60$. We see that for all those different surface geometries, using just $N = 20$ neurons (learnable parameters $N_p = 100$) is sufficient to encode the solutions with at least 0.01% predictive accuracy. Although the numerical convergence is not rigorously verified, the increase of neurons generally leads to better accuracy for all these cases shown in the table. Besides, for a fixed N, it is interesting to see that the overall training procedures appear to have almost the same computational performance among the three different surfaces, showing the robustness of the present model for handling the surface PDE on different surface geometries.

The predicted network solution $u_{\mathcal{N}}$ (with $N = 60$ and $M = 400$) and the absolute error $|u_{\mathcal{N}} - u|$ for these surfaces are depicted in Fig. 3. One can see that, regardless of the surface geometries, our designed network model is able to obtain equally accurate prediction for all cases (the largest absolute error does not necessarily occur at the points with large mean curvatures). In addition, these results are obtained by randomly sampled training points on the underlying surfaces, highlighting the robustness feature of the mesh-free nature of the neural network model.

Table 6
The average relative L^2 errors and CPU time (in seconds) with different number of neurons N in the hidden layer. For each case, the number of training points is fixed by $M = 400$ and the training procedure is terminated at 500 steps.

(N, N_p)	Torus	CPU time (s)	Genus-2	CPU time (s)	Cheese-like	CPU time (s)
(20, 100)	2.774E−05	2.16	1.816E−06	2.25	1.522E−04	2.28
(30, 150)	5.568E−06	2.72	9.150E−07	2.78	2.897E−05	2.76
(40, 200)	2.181E−06	3.42	6.100E−07	3.40	1.018E−05	3.37
(50, 250)	1.708E−06	4.00	4.731E−07	4.08	7.176E−06	4.13
(60, 300)	1.139E−06	4.60	5.169E−07	4.55	5.617E−06	4.65

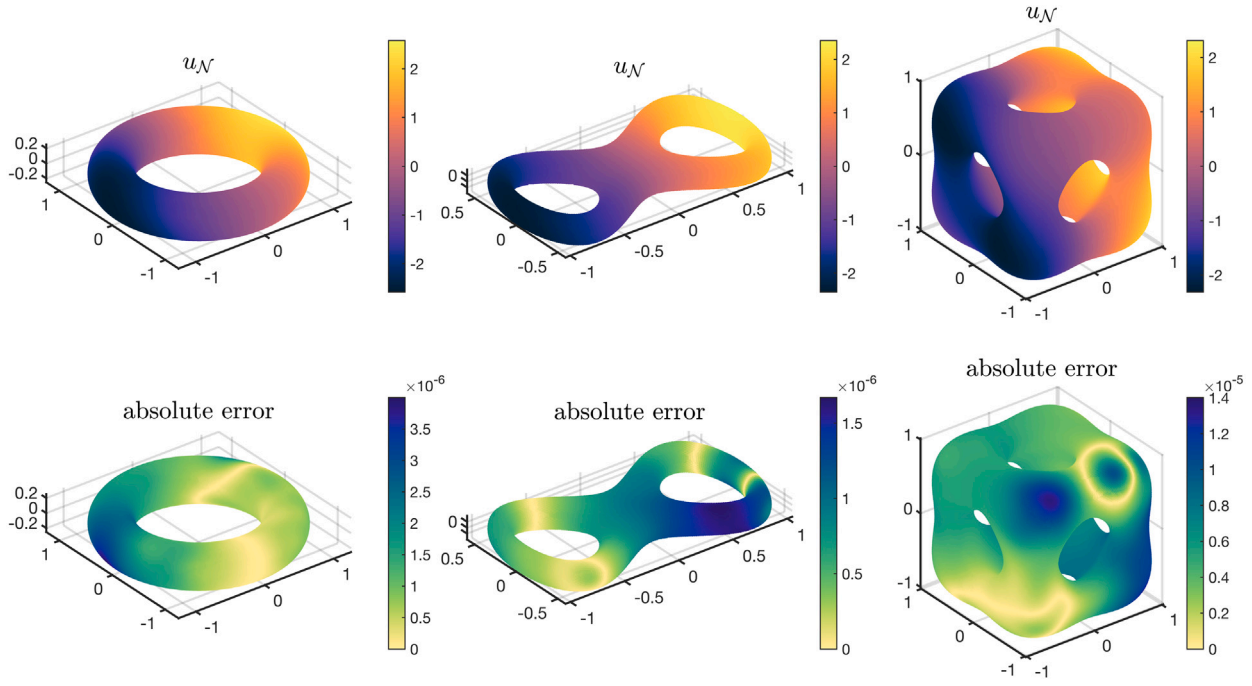

Fig. 3. Prediction solution $u_{\mathcal{N}}$ and corresponding absolute error $|u_{\mathcal{N}} - u|$ with $N = 60$ neurons employed. From left to right: torus, genus-2 surface, cheese-like surface.

3. Time-dependent PDEs on static surfaces

In this section, we turn our attention to solve time-dependent PDEs on static surfaces. Given a regular and closed surface Γ, along which we consider the PDEs of the general form

$$\partial_t u(\mathbf{x}, t) = \mathcal{L}(u(\mathbf{x}, t)) + f(\mathbf{x}, t) \quad \text{on } \Gamma,\ t \in (0, T], \tag{9}$$

where t denotes the time variable and T is the terminal time; f is a source term defined on Γ. Again $\mathcal{L}(u)$ may contain the surface gradient $\nabla_s u$, surface diffusion $\Delta_s u$, or $\nabla_s \cdot \mathbf{v}$ for some known vector field $\mathbf{v}$.

In this section, we shall concentrate on solving the surface diffusion equation ($\mathcal{L} = \Delta_s$) as

$$\partial_t u(\mathbf{x}, t) = \Delta_s u(\mathbf{x}, t) + f(\mathbf{x}, t) \quad \text{on } \Gamma,\ t \in (0, T]. \tag{10}$$

The above PDE is subjected to an initial condition

$$u(\mathbf{x}, t = 0) = u_0(\mathbf{x}) \quad \text{on } \Gamma. \tag{11}$$

To solve this time-dependent PDE, we follow the pioneering framework of physics-informed neural networks proposed in [24], i.e., the above diffusion equation is solved by continuous-time or discrete-time model.

Continuous-time model. It is natural to encapsulate both spatial and time variables as the input of neural network function. Thus, the approximate solution to Eq. (10) now can be written as

$$u_{\mathcal{N}}(\mathbf{x}, t) = \sum_{j=1}^{N} \alpha_j \sigma(W_j(\mathbf{x}, t)^T + b_j). \tag{12}$$

Differing from the stationary case (see Eq. (5)) due to the time variable augmentation, the dimension of weights becomes $W_j \in \mathbb{R}^{1\times 4}$, so the total number of learnable parameters is increased as $N_p = 6N$.

To learn those parameters, as in stationary case, we train the neural net model using the identities in Eq. (2) to compute differential terms appeared in Eq. (10). Thus, it is straightforward to employ the physics-informed learning method to minimize the mean squared residual for both differential Eq. (10) and initial condition (11). For given training points $\{(\mathbf{x}^i, t^i)|\mathbf{x}^i \in \Gamma, t^i \in (0,T]\}_{i=1}^{M_T}$ and $\{\mathbf{x}_0^j \in \Gamma\}_{j=1}^{M_0}$, the natural training loss is chosen as

$$\text{Loss}_c(\mathbf{p}) = \frac{1}{M_T}\sum_{i=1}^{M_T}\left[\partial_t u_{\mathcal{N}}(\mathbf{x}^i,t^i) - \Delta_s u_{\mathcal{N}}(\mathbf{x}^i,t^i) - f(\mathbf{x}^i,t^i)\right]^2 + \frac{1}{M_0}\sum_{j=1}^{M_0}\left[u_{\mathcal{N}}(\mathbf{x}_0^j,0) - u_0(\mathbf{x}_0^j)\right]^2, \tag{13}$$

where $\Delta_s u_{\mathcal{N}}(\mathbf{x}^i,t^i) = \Delta u_{\mathcal{N}}(\mathbf{x}^i,t^i) - 2H(\mathbf{x}^i)\partial_n u_{\mathcal{N}}(\mathbf{x}^i,t^i) - \mathbf{n}(\mathbf{x}^i)^T\left(\nabla^2 u_{\mathcal{N}}(\mathbf{x}^i,t^i)\right)\mathbf{n}(\mathbf{x}^i)$.

Discrete-time model. In contrast to the continuous-time model, in discrete-time model, the PDE (10) is alternatively solved by a semi-discretization scheme as in classical numerical methods [24]. That is, we obtain the numerical solution via the q-stage time-stepping implicit Runge–Kutta (RK) scheme:

$$u^{n+c_j} = u^n + \Delta t\sum_{k=1}^{q} a_{jk}(\Delta_s u^{n+c_k} + f^{n+c_k}), \quad j = 1,2,\dots,q, \tag{14}$$

$$u^{n+1} = u^n + \Delta t\sum_{k=1}^{q} b_k(\Delta_s u^{n+c_k} + f^{n+c_k}), \tag{15}$$

where Δt is the time step size, $u^{n+c_j} = u(\mathbf{x},(n+c_j)\Delta t)$ and $f^{n+c_j} = f(\mathbf{x},(n+c_j)\Delta t)$ are the intermediate solution and source term correspondingly, and $u^{n+1} = u(\mathbf{x},(n+1)\Delta t)$ is the numerical solution at the next time level. Here we adopt Gauss–Legendre method so the temporal discretization error of above q-stage Runge–Kutta scheme is $O(\Delta t^{2q})$, where the parameters $\{a_{jk}, b_k, c_k\}$ are given from Butcher tableau [43]. By taking sufficiently large q, this high-order scheme allows us to obtain an accurate numerical solution u^{n+1} even with large Δt. Meanwhile, the numerical stability can be retained due to the fully implicit expression in Eq. (14).

To obtain u^{n+1}, we need to learn those intermediate network solutions, $u_{\mathcal{N}}^{n+c_j}$, again via physics-informed learning technique. We proceed by placing a multi-output neural network $\mathbf{u}_{\mathcal{N}}(\mathbf{x}) = [u_{\mathcal{N}}^{n+c_1}(\mathbf{x}), u_{\mathcal{N}}^{n+c_2}(\mathbf{x}), \dots, u_{\mathcal{N}}^{n+c_q}(\mathbf{x}), u_{\mathcal{N}}^{n+1}(\mathbf{x})]^T$ and it can be compactly expressed by

$$\mathbf{u}_{\mathcal{N}}(\mathbf{x}) = W^{[2]}\sigma(W^{[1]}\mathbf{x}^T + \mathbf{b}^{[1]}),$$

where $W^{[1]} \in \mathbb{R}^{N\times 3}$ and $W^{[2]} \in \mathbb{R}^{(q+1)\times N}$ are the weight matrices and $\mathbf{b}^{[1]} \in \mathbb{R}^N$ is the bias (so all $u_{\mathcal{N}}^{n+c_j}$ and $u_{\mathcal{N}}^{n+1}$ are learned in a single network). In this network, there are $N_p = (5+q)N$ parameters needed to be learned. The loss function is thereby designed to simultaneously enforce all discretization equations (14) together with the updating step (15). That is, given a set of training points $\{\mathbf{x}^i \in \Gamma\}_{i=1}^{M}$, we have

$$\begin{aligned}\text{Loss}_d(\mathbf{p}) = &\frac{1}{M}\sum_{j=1}^{q}\sum_{i=1}^{M}\left[u_{\mathcal{N}}^{n+c_j}(\mathbf{x}^i) - u^n(\mathbf{x}^i) - \Delta t\sum_{k=1}^{q} a_{jk}(\Delta_s u_{\mathcal{N}}^{n+c_k}(\mathbf{x}^i) + f^{n+c_k}(\mathbf{x}^i))\right]^2 \\ &+ \frac{1}{M}\sum_{i=1}^{M}\left[u_{\mathcal{N}}^{n+1}(\mathbf{x}^i) - u^n(\mathbf{x}^i) - \Delta t\sum_{k=1}^{q} b_k(\Delta_s u_{\mathcal{N}}^{n+c_k}(\mathbf{x}^i) + f^{n+c_k}(\mathbf{x}^i))\right]^2.\end{aligned} \tag{16}$$

After finishing the training of the above loss model, we use this prediction $u_{\mathcal{N}}^{n+1}$ as the initial condition to advance to the next time level $u_{\mathcal{N}}^{n+2}$ by proceeding to the same training process. Eventually, we obtain the numerical solution at the target terminal time.

3.1. Numerical accuracy

We perform the capability of continuous- and discrete-time neural network model, corresponding to Loss_c in (13) and Loss_d in (16), for encoding the diffusion equation on the cheese-like surface. We check the prediction accuracy by considering the exact solution

$$u(x,y,z,t) = \sin(x+\sin(t))\exp(\cos(y-z)),$$

so the source term f can be obtained accordingly. We set the terminal time $T = 1$. For continuous-time model we use $M_0 = 100$ and $M_T = 800$ spatial–temporal training points, in which the surface points $\mathbf{x}^i$ are randomly sampled while temporal points t^i are chosen based on Latin Hypercube Sampling strategy [44]. In discrete-time model we set $M = 200$ spatial training points and adopt 6-stage implicit Runge–Kutta scheme with time step size $\Delta t = 1$ (so the network solution at terminal time $T = 1$ is obtained under a single time update). In both models we terminate the training procedure up to 500 steps. The average relative L^2 errors and CPU time (in seconds) at $T = 1$ for the network models with various neurons of the hidden layer N are shown in Table 7. Again, both models can obtain accurate predictive results. Furthermore, as expected, the increase of the number of neurons generally leads to better accuracy as well. It is apparent that the performance of discrete-time model is much more costly than the continuous-time model. Despite the number of trainable parameters of discrete-time model is just a little bit larger than the one of continuous-time model when the same number N is used, the multi-output network employed in the discrete-time model results in a tremendous workload in the computation of auto-differentiation and LM update step.

Table 7
The average relative L^2 errors and CPU time (in seconds) at $T = 1$ for continuous-time and discrete-time model with various neurons of hidden layer N. In each test, we fix $M_T = 800$ and $M_0 = 100$ for the continuous-time model; $M = 200$ and $\Delta t = 1$ for the 6-stage RK discrete-time model. Both models are ran up to 500 training steps.

(N, N_p)	Continuous-time model	CPU time (s)	(N, N_p)	Discrete-time model	CPU time (s)
(20, 120)	1.400E−03	4.05	(20, 220)	6.448E−04	60
(30, 180)	1.975E−04	5.61	(30, 330)	5.013E−05	111
(40, 240)	1.390E−04	7.08	(40, 440)	1.627E−05	176
(50, 300)	5.984E−05	8.54	(50, 550)	7.920E−06	241
(60, 360)	3.661E−05	9.89	(60, 660)	6.446E−06	346

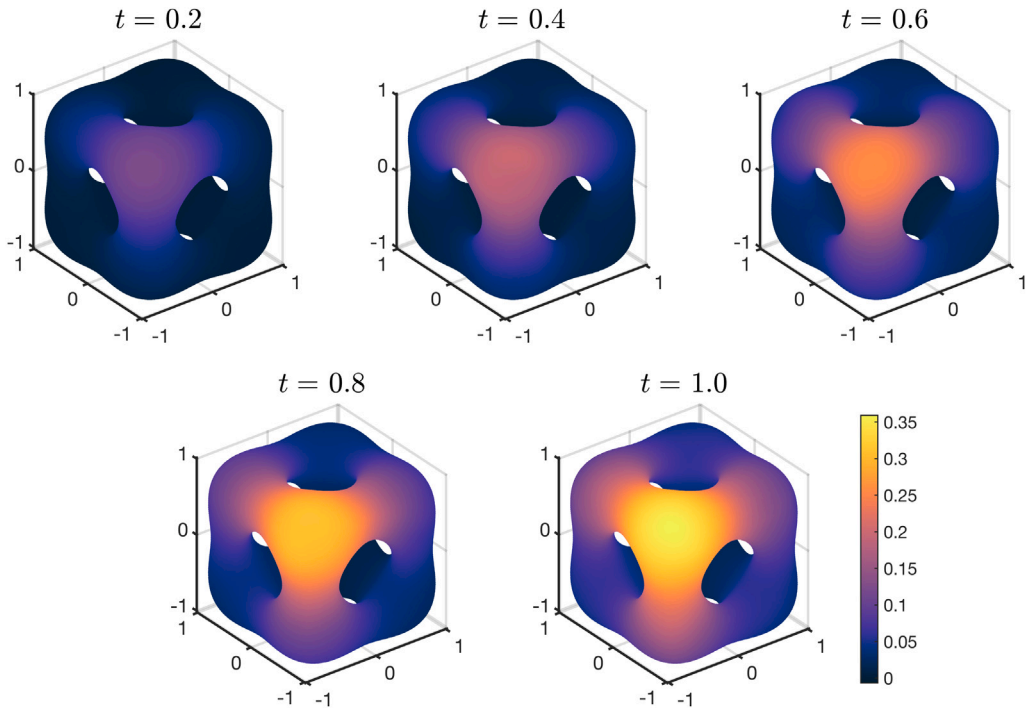

Fig. 4. The snapshots of solution distribution for heating up the cheese-like surface at different times. The color code ranging from 0 to 0.36 indicates the magnitude of the solution.

3.2. A surface heating up application

We perform an application simulation by mimicking the process of heating up a surface. The initial condition is set to be zero everywhere and the heating source is a time-independent Gaussian bump given by

$$f(x, y, z) = \exp(-((x + 1)^2 + (y + 1)^2 + (z - 1)^2)),$$

so the majority of the source accumulates in the vicinity of the point $(-1, -1, 1)$. The diffusion equation is solved using the discrete-time model with the 4-stage RK scheme, in which we set the time step $\Delta t = 0.1$ and compute the solution up to the terminal time $T = 1$. We use $M = 500$ training points and $N = 100$ neurons. Since there is no analytical solution available in this case, we are unable to measure the relative L^2 error quantitatively. We simply train the network to get the loss value to the order of magnitude 10^{-8} which is roughly matched with the temporal discretization error $(\Delta t)^8 = 10^{-8}$. The successive snapshots of time-evolutional solution are displayed in Fig. 4. As we can see, near the source of Gaussian bump, the magnitude of prediction solution becomes larger as time evolves. At the same time, the heat distribution becomes wider due to the diffusion mechanism in the PDE model. Therefore, the predictive solution generated by our network model presents some visually plausible results.

4. PDEs on evolving surfaces

In this section, we extend the proposed methodology to solve PDEs on evolving surfaces. Here, the considered surface $\Gamma(t)$ evolves with a prescribed velocity field $\mathbf{v}(\mathbf{x}, t)$ so its configuration follows the evolutional equation

$$\partial_t \mathbf{x} = \mathbf{v}(\mathbf{x}(t), t), \quad \mathbf{x}(t) \in \Gamma(t),\ t \in (0, T], \tag{17}$$

together with an initial configuration $\Gamma(0)$ represented by $\mathbf{x}(t=0)=\mathbf{x}_0$. For simplicity, we assume that $\Gamma(t)$ remains a regular surface under the velocity field $\mathbf{v}$.

Throughout this section, we consider the following advection–diffusion equation on the evolving surface $\Gamma(t)$ as

$$\partial_t u + \mathbf{v}\cdot\nabla u + (\nabla_s\cdot\mathbf{v})u = \Delta_s u + f \quad \text{on } \Gamma(t),\ t\in(0,T], \tag{18}$$

where $\partial_t u+\mathbf{v}\cdot\nabla u$ denotes the material derivative of u, and the term $(\nabla_s\cdot\mathbf{v})u$ represents the surface stretching effect on the quantity u. The term $f=f(\mathbf{x},t)$ is again a given source term defined on $\Gamma(t)$. For completeness, the above equation must be accompanied by a given initial condition $u(\mathbf{x}_0,t=0)=u_0(\mathbf{x}_0)$ on $\Gamma(t=0)$. One should note that the above Eq. (18) is popularly used in modeling certain physical applications; for instance, the insoluble surfactant concentration along a droplet surface in fluid flows [34,45]. The major challenge of solving Eq. (18) arises from the time-dependent computation of surface geometrical quantities such as mean curvatures $H(\mathbf{x},t)$ and normal vectors along the evolving surface (which are involved in those conventional differential terms as seen in Eq. (2)). We aim to solve the PDE system (17)–(18) under a unified continuous-time neural network framework, as stated as follows.

4.1. Neural network solver for PDEs on evolving surfaces

To track this time evolving surface using neural network representation, we adopt the surface parametrization as

$$\Gamma(t)=\{\mathbf{x}(\theta,\phi,t)\in\mathbb{R}^3\,|\,(\theta,\phi)\in[0,\pi]\times[0,2\pi), t\in[0,T]\}.$$

The key observation is that a closed surface (with genus zero) is homeomorphic to a unit sphere $\mathbb{S}^2$ in three-dimensional space. Therefore, there exists a continuous and invertible mapping between the surface $\Gamma(t)$ and $\mathbb{S}^2$. We hereby propose a two-hidden-layer neural network structure to represent the surface. We first map the input variables (θ,ϕ) to the unit sphere $\mathbb{S}^2$ (as the output of the first hidden layer), and then use a fully-connected neural network to learn the homeomorphism between $\mathbb{S}^2$ and the surface. More precisely, let $S^2(\theta,\phi)=(\sin\theta\cos\phi,\sin\theta\sin\phi,\cos\theta)$, then the homeomorphic network can be written as

$$\mathbf{x}_{\mathcal{N}}(\theta,\phi,t)=W^{[2]}\sigma(W^{[1]}(S^2(\theta,\phi),t)^T+\mathbf{b}^{[1]}), \tag{19}$$

where the weight matrices $W^{[1]}\in\mathbb{R}^{N\times 4}$ and $W^{[2]}\in\mathbb{R}^{3\times N}$, and the bias $\mathbf{b}^{[1]}\in\mathbb{R}^N$ (so the total number of training parameters $N_p=8N$). We should emphasize the features of the above surface network representation: (i) There are no parameters needed to be trained from the input layer to the first hidden layer, i.e., the output of the first hidden layer is directly computed through the nonlinear map S^2. Whereas, the remaining parameters of the network ($W^{[1]}$, $W^{[2]}$, and $\mathbf{b}^{[1]}$) need to be trained. (ii) This representation automatically fulfills 2π-periodicity in ϕ-direction while the pole conditions at $\theta=0$ and π are taken care by the parametrization of S^2. (iii) When a genus g surface is considered, following the same idea, we can adopt the mapping from the parametric domain to a g-torus so that the homeomorphism can be learned using neural network representation.

Now, with the proper surface network representation (19), we proceed to solve the surface evolving equation (17). Given sets of training points $\{(\theta^i,\phi^i,t^i)|(\theta^i,\phi^i)\in[0,\pi]\times[0,2\pi), t^i\in(0,T]\}_{i=1}^{M_T}$ and $\{(\theta_0^j,\phi_0^j)\in[0,\pi]\times[0,2\pi)\}_{j=1}^{M_0}$, the surface configuration at any instantaneous time is found by minimizing the continuous-time loss model as

$$\mathrm{Loss}_{\mathbf{x}}(\mathbf{p})=\frac{1}{M_T}\sum_{i=1}^{M_T}\left[\partial_t\mathbf{x}_{\mathcal{N}}(\theta^i,\phi^i,t^i)-\mathbf{v}(\mathbf{x}_{\mathcal{N}}(\theta^i,\phi^i,t^i),t^i)\right]^2+\frac{1}{M_0}\sum_{j=1}^{M_0}\left[\mathbf{x}_{\mathcal{N}}(\theta_0^j,\phi_0^j,0)-\mathbf{x}_0(\theta_0^j,\phi_0^j)\right]^2. \tag{20}$$

Here, both (θ^i,ϕ^i) and (θ_0^j,ϕ_0^j) are chosen so that S^2 acting on those points are randomly distributed on $\mathbb{S}^2$. This strategy shall effectively avoid local cluster of sample points on $\Gamma(t)$. After the termination of the training process, we use the network solution $\mathbf{x}_{\mathcal{N}}(\theta,\phi,t)$ to build up the training sets $\{(\mathbf{x}^i,t^i)|\mathbf{x}^i\equiv\mathbf{x}_{\mathcal{N}}(\theta^i,\phi^i,t^i)\in\Gamma(t^i)\}_{i=1}^{M_T}$ and $\{\mathbf{x}_0^j\equiv\mathbf{x}_{\mathcal{N}}(\theta_0^j,\phi_0^j,0)\in\Gamma(0)\}_{j=1}^{M_0}$, and find the normal vectors $\mathbf{n}(\mathbf{x}^i,t^i)$ and mean curvatures $H(\mathbf{x}^i,t^i)$ via the first and second fundamental forms in differential geometry [32]. As a consequence, finding the solution to the surface PDE (18) is a straightforward application of the present method. Namely, expressing the shallow neural network solution by Eq. (12), we minimize the loss function

$$\begin{aligned}\mathrm{Loss}_u(\mathbf{p})=&\frac{1}{M_T}\sum_{i=1}^{M_T}\left[\partial_t u_{\mathcal{N}}(\mathbf{x}^i,t^i)+\mathbf{v}(\mathbf{x}^i,t^i)\cdot\nabla u(\mathbf{x}^i,t^i)+(\nabla_s\cdot\mathbf{v}(\mathbf{x}^i,t^i))u(\mathbf{x}^i,t^i)-\Delta_s u_{\mathcal{N}}(\mathbf{x}^i,t^i)-f(\mathbf{x}^i,t^i)\right]^2\\&+\frac{1}{M_0}\sum_{j=1}^{M_0}\left[u_{\mathcal{N}}(\mathbf{x}_0^j,0)-u_0(\mathbf{x}_0^j)\right]^2,\end{aligned} \tag{21}$$

where

$$\Delta_s u_{\mathcal{N}}(\mathbf{x}^i,t^i)=\Delta u_{\mathcal{N}}(\mathbf{x}^i,t^i)-2H(\mathbf{x}^i,t^i)\partial_n u_{\mathcal{N}}(\mathbf{x}^i,t^i)-\mathbf{n}(\mathbf{x}^i,t^i)^T\left(\nabla^2 u_{\mathcal{N}}(\mathbf{x}^i,t^i)\right)\mathbf{n}(\mathbf{x}^i,t^i),$$

and $\nabla_s\cdot\mathbf{v}(\mathbf{x}^i,t^i)=\nabla\cdot\mathbf{v}(\mathbf{x}^i,t^i)-\mathbf{n}(\mathbf{x}^i,t^i)^T\nabla\mathbf{v}(\mathbf{x}^i,t^i)\mathbf{n}(\mathbf{x}^i,t^i)$.

It is worth mentioning that in other Eulerian grid-based embedding methods [21,45], an operator splitting strategy is required in order to solve the advection and diffusion parts separately. By contrast, the present method (21) deals with the surface PDE at the instantaneous time $t=t_i$ directly; thus, the implementation is simple and straightforward.

Since the surface configuration $\mathbf{x}_{\mathcal{N}}$ and the underlying solution $u_{\mathcal{N}}$ change simultaneously as time proceeds, it is more practical to obtain them in a time sequential manner [46,47], especially for longer time T. In this way, we divide the time interval $[0,T]$

Table 8
The average relative L^2 errors for the surface configuration $\mathbf{x}$, normal vector $\mathbf{n}$, mean curvature H, and CPU time (in seconds) at $T = 2$. For each case, the number of training points is fixed by $M = 800$ and $M_0 = 100$ and the training task is ran up to 500 steps. The total number of learnable parameters for the network expression (19) is $N_p = 8N$.

(N, N_p)	$\|\mathbf{x}_{\mathcal{N}} - \mathbf{x}\|_2/\|\mathbf{x}\|_2$	$\|\mathbf{n}_{\mathcal{N}} - \mathbf{n}\|_2/\|\mathbf{n}\|_2$	$\|H_{\mathcal{N}} - H\|_2/\|H\|_2$	CPU time (s)
(10, 80)	4.574E−04	3.317E−04	7.360E−04	98
(20, 160)	8.678E−05	3.305E−05	1.037E−04	122
(30, 240)	4.206E−06	1.980E−06	7.156E−06	148
(40, 320)	1.116E−06	1.892E−06	7.393E−06	181

into n uniform subintervals as $[0, T] = \cup_{k=1}^{n}[T_{k-1}, T_k]$, and apply the above learning machinery to obtain the solutions of $\mathbf{x}_{\mathcal{N}}$ and $u_{\mathcal{N}}$ in each time interval $[T_{k-1}, T_k]$ starting at $k = 1$. We repeatedly use the loss functions Eqs. (20) and (21) by resuming the initial data that is obtained from the trained results in the previous time interval. Unless otherwise stated, we use the notation M (instead of M_T) to denote the number of training points used in parametric domain $(\theta, \phi, t) \in [0, \pi] \times [0, 2\pi) \times (T_{k-1}, T_k]$, and M_0 to denote the number of training points used in the initial data at T_{k-1}. Here, we randomly choose $M_{test} = 100M$ test points and repeat the numerical experiments five times, so the averaged relative L^2 error is computed based on these five runs on different training points. The numerical results are shown in the following subsections.

4.2. Numerical results

This example aims to demonstrate the capability and accuracy of the proposed method for solving the advection–diffusion equation on a 2D evolving surface in $\mathbb{R}^3$. We consider the case of an oscillating ellipsoid [18,45] whose configuration is described by

$$\Gamma(t) = \left\{ (x, y, z) \middle| \left(\frac{x}{1.5a(t)}\right)^2 + y^2 + \left(\frac{z}{0.5}\right)^2 = 1 \right\}.$$

The associated velocity field is $\mathbf{v} = \left(\frac{a'(t)}{a(t)}x, 0, 0\right)$ and we set $a(t) = \sqrt{1 + 0.95\sin(\pi t)}$. One should note that the above Cartesian representation for $\Gamma(t)$ can be easily rewritten as the parametric form in terms of (θ, ϕ) using unit sphere representation S^2 in the previous subsection. The exact solution to the surface advection–diffusion Eq. (18) is again chosen as $u(x, y, z, t) = \sin(x + \sin(t))\exp(\cos(y - z))$, and the source function f can be obtained accordingly. We divide the time interval $[0, 2]$ into 10 uniform subintervals, and sequentially calculate both the neural network solutions $\mathbf{x}_{\mathcal{N}}$ and $u_{\mathcal{N}}$ up to time $T = 2$, so a total of 10 steps of time integration are needed to reach the terminal time.

Predictive accuracy for tracking the surface. The surface evolutional differential equation (17) is solved using the loss model (20). Given the number of training points $M = 800$ and $M_0 = 100$, we train the network model with different number of neurons in the hidden layer up to 500 training steps. Table 8 reports the relative L^2 errors of the network predictive surface configuration $\mathbf{x}_{\mathcal{N}}$, the normal vector $\mathbf{n}_{\mathcal{N}}$, the mean curvature $H_{\mathcal{N}}$, and the CPU time (in seconds) at $T = 2$. It can be seen that the network solution (19) can accurately predict not only the surface configuration, but also the normal vector and mean curvature at the test points. Those relative L^2 errors range from 10^{-4} to 10^{-6} using merely 10–40 neurons in the hidden layer. The history of training loss (20) for $N = 40$ is shown in Fig. 5. Recall that the time integration is processed sequentially in 10 temporal subintervals, so we use black dotted line to divide the loss history in each training task. In the first time interval, the trainable parameters are initialized by standard normal distribution, while in the subsequent time intervals we initialize those parameters by inheriting the ones obtained in the previous training task. As can be seen, this strategy would somehow provide proper initialization for each training task. In addition, we depict the snapshots of the predictive surface configuration $\mathbf{x}_{\mathcal{N}}$ and mean curvature $H_{\mathcal{N}}$ for $N = 40$ in Fig. 6. We should also point out that the present method is mesh-free and the implementation is much easier in comparison with the traditional grid based methods [45].

Predictive accuracy for solving the advection–diffusion equation on an evolving surface. The loss model (21) is used to find the solution to the advection–diffusion equation (18). Again, using $M = 800$ and $M_0 = 100$ training points with 500 training steps, we first train the network with $N = 40$ neurons to predict $\mathbf{x}_{\mathcal{N}}$, $\mathbf{n}_{\mathcal{N}}$, and $H_{\mathcal{N}}$ at the training points (i.e., at (θ^i, ϕ^i, t^i), we set $\mathbf{x}^i_{\mathcal{N}} = \mathbf{x}^i$, $\mathbf{n}^i_{\mathcal{N}} = \mathbf{n}(\mathbf{x}^i, t^i)$, and $H^i_{\mathcal{N}} = H(\mathbf{x}^i, t^i)$), and then use them as the inputs in the loss function (21). Table 9 shows that, simply using the shallow network representation, our proposed solver can indeed achieve high accuracy predictions for different number of neurons $N = 10, 20, 30, 40$ used in the hidden layer. The trajectory of training loss (21) is depicted in Fig. 7. Again, we use black dotted line to divide the loss history in each training task, and use the same strategy as in the previous experiment to initialize the trainable parameters. We must emphasize that the present method enjoys the advantage of being mesh-free, and thus can be easily implemented to find the embedded solution u.

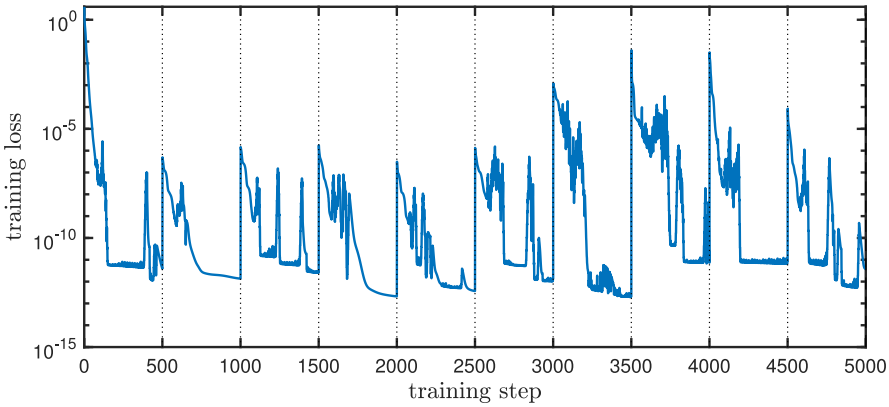

Fig. 5. The history of training loss (20) for $N = 40$. Notice that the time integration is processed sequentially in 10 temporal subintervals, so we use black dotted line to divide the loss history in each training task.

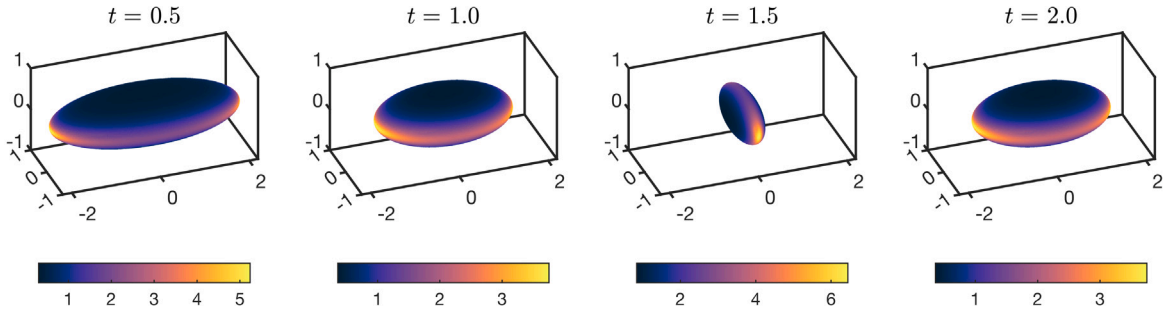

Fig. 6. The snapshots of the network solution $\mathbf{x}_{\mathcal{N}}$ with $N = 40$ neurons at different times. The color code indicates the magnitude of the mean curvature $H_{\mathcal{N}}$.

Table 9
The average relative L^2 errors and CPU time (in seconds) for the solution u at $T = 2$. For each case, the number of training points is fixed by $M = 800$ and $M_b = 100$ and the training task is ran up to 500 steps. The total number of learnable parameters for the network expression is $N_p = 6N$.

(N, N_p)	$\|u_{\mathcal{N}} - u\|_2/\|u\|_2$	CPU time (s)
(10, 60)	1.968E−03	25
(20, 120)	6.557E−04	43
(30, 180)	3.903E−05	60
(40, 240)	2.885E−05	78

4.3. Surfactant transport on a droplet surface under shear flow

As an application, we mimic the simulation of surfactant transport on a droplet surface [34,45] that has been extensively studied using various numerical methods in literature. Here we neglect the fluid effect but simply apply the known shear flow $\mathbf{v}(x, y, z, t) = (z, 0, 0)$ to the droplet. The initial shape of the droplet surface is set as a unit sphere located at the origin, and will be elongated by the shear flow along the x-direction. One can simply derive the exact surface configuration under this flow as

$$\Gamma(t) = \left\{ (x, y, z) \middle| (x - tz)^2 + y^2 + z^2 = 1 \right\}.$$

The initial surfactant concentration u is set to be uniform as $u(x, y, z, 0) = 1$ while the source term is $f(x, y, z, t) = 0$. We construct the network representation for $\mathbf{x}_{\mathcal{N}}$ and $u_{\mathcal{N}}$ with $N = 50$ and $N = 100$ neurons respectively, and use $M = 1000$ and $M_0 = 500$ training points in the loss models. The simulation is performed up to time $T = 3$ sequentially by dividing the time interval $[0, 3]$ into 10 uniform subintervals so overall 10 steps of time integration are needed to reach the terminal time. The snapshots for the droplet configuration $\mathbf{x}_{\mathcal{N}}$ along with the surfactant concentration $u_{\mathcal{N}}$ are shown in Fig. 8. As seen, due to the presence of the applied shear flow, the surfactant is swept toward the both tips of the droplet surface as time evolves, leading to high concentration at the tips while low concentration at the sides of the surface. This concentration distribution is commonly observed in the presence of shear

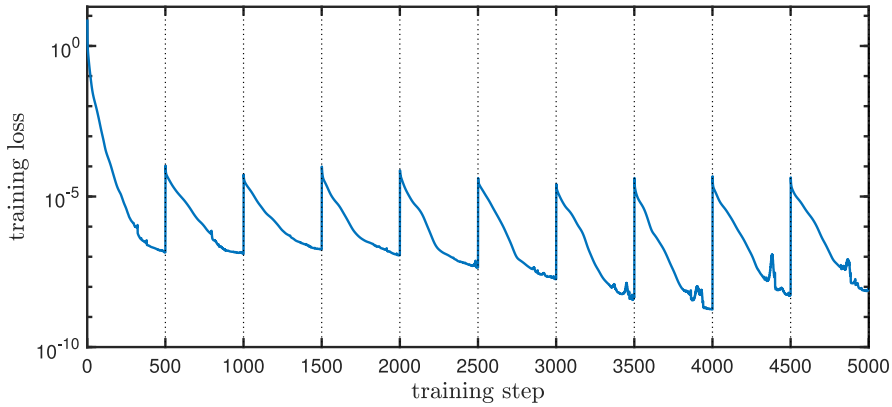

Fig. 7. The history of training loss (21). Notice that the time integration is processed sequentially in 10 temporal subintervals, so we use black dotted line to divide the loss history in each training task.

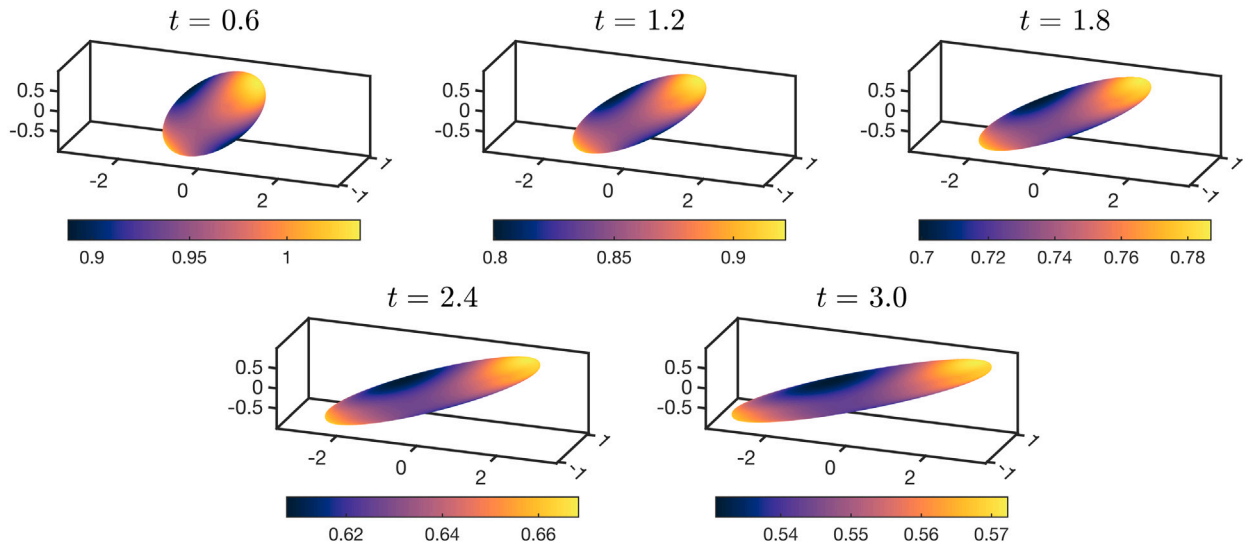

Fig. 8. The snapshots of the droplet surface configuration $\mathbf{x}_{\mathcal{N}}$ ($N = 50$) and surfactant concentration $u_{\mathcal{N}}$ ($N = 100$) at different times. The color code indicates the magnitude of $u_{\mathcal{N}}$.

flow even with the fluid effect [45]. We should point out that, the present mesh-free neural network method has no difficulty to handle the scenario of large surface distortion (see $t = 3$ in the figure), while in traditional numerical methods, the droplet surface must be re-meshed from time to time to keep accurate and stable computations.

Since the given flow $\mathbf{v}$ is incompressible, the droplet volume $V(t) = \frac{1}{3}\int_{\Gamma(t)} \mathbf{x}\cdot\mathbf{n}\,\mathrm{d}S$ should be conserved as a constant $V(0) = 4\pi/3$. Meanwhile, without the additional source ($f = 0$), the total surfactant mass $m(t) = \int_{\Gamma(t)} u\,\mathrm{d}S$ is also conserved as its initial value $m(0) = 4\pi$. Despite the present method does not guarantee the numerical conservation for these two quantities, we plot the evolutions of the relative error for droplet volume $|V(t) - V(0)|/V(0)$ and total surfactant mass $|m(t) - m(0)|/m(0)$ in Fig. 9. These two surface integrations for $V(t)$ and $m(t)$ are performed by Gauss–Legendre quadrature rule in θ-direction and midpoint rule in ϕ-direction. One can see that both error plots are discontinuous at the endpoint of each time subinterval since the initial conditions for $\mathbf{x}_{\mathcal{N}}$ and $u_{\mathcal{N}}$ are resumed in our loss models. The relative volume error reaches as low as approximately 10^{-6} even when the surface is highly distorted at $t = 3$, and the total surfactant mass error reaches around 10^{-5}. This results outperform the ones obtained in [45].

5. Conclusion and future works

In this paper, a completely shallow physics-informed neural network is developed to solve Laplace–Beltrami and diffusion equations on static surfaces, and advection–diffusion equation on evolving surfaces. Those surface PDEs are written in Eulerian coordinates in which geometrical differentiations are calculated by conventional differential operators. For the static surface case, with the aid of the level set function, the surface geometrical quantities such as the normal and mean curvature of the surface

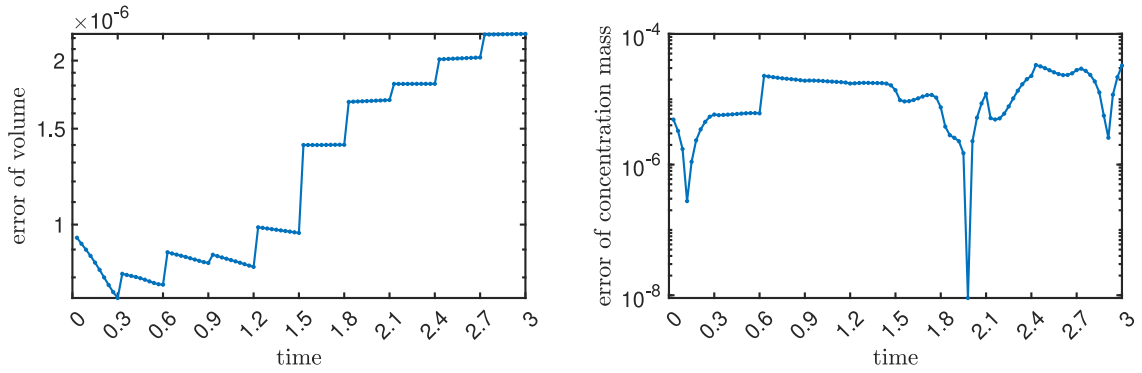

Fig. 9. The time plots of droplet volume error $|V(t) - V(0)|/V(0)$ (left), and total surfactant mass error $|m(t) - m(0)|/m(0)$ (right).

can be computed directly and used in our surface differential expressions. The loss function hereby penalizes the equation residual written in the form of Cartesian differential operators instead of imposing normal extension constraints used in literature. As for the evolving surface, we additionally introduce a prescribed hidden layer to enforce the topological structure of the surface and use the network to learn the homeomorphism between the surface and the prescribed topology. The proposed network structure is designed to track the surface and solve the equation simultaneously. Since the present neural network uses only one hidden layer, the model is easy to implement and train. Numerical results show high predictive accuracy using just a moderate number of neurons in the hidden layer. We have to point out that, with different input sources or surface geometries, one needs to train the associated loss model to find the network solution accordingly.

The traditional mesh-free method represents the solution by a linear combination of some chosen radial basis functions (RBFs; for instance, Gaussian), and enforces the solution to satisfy the PDE directly at some chosen points. In fact, one can regard the present shallow neural network solution as a linear combination of activation basis in which the weights and bias must be determined via learning. It would be nice to make a fair performance comparison (including the computational cost and accuracy) between the RBFs method and the present neural network method. But this is beyond the scope of the paper which we shall leave it as our future work.

As known, PINN performs well when the solution is smooth. However, if the solution is highly oscillatory, one can implement a multi-scale deep neural network structure developed in [48] to expedite the training process and achieve fast convergence over different scales. On the other hand, if the solution has low regularity such as the one comes from corner singularity problems, to the best of our knowledge, so far there is no effective remedy to improve the accuracy except putting more training points near the corner singularity. Since the main themes of the paper are to compare the difference of two different loss functions used in approximating the surface Laplace operator in PINNs framework and extend the developed methodology to evolving surface case, the detailed investigation on how to obtain non-smooth solutions in PINNs for solving PDEs is beyond the scope of the paper.

The surfaces considered in this paper are defined by given level set representations. Within the current model implementation, both the normal vector and mean curvature are required at each training point, and these can be readily computed through the level set function. As a forthcoming extension, we shall consider PDEs on a point cloud of closed surface in which the level set function is not available. One potential and feasible way to determine the mean curvature of a point cloud is to leverage methods of point cloud parametrization [49,50]. For instance, a point cloud surface with genus-zero can be mapped onto a unit sphere via spherical conformal parametrization [49]. Along this way, the mean curvature can be computed directly using the cotangent formula [51] over a high-quality triangular mesh. Computing the normal vector and mean curvature at these training points is clearly a challenging task as the surface evolves. We shall leave it for our future work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgment

W.-F. Hu, T.-S. Lin and M.-C. Lai acknowledge the supports by National Science and Technology Council, Taiwan, under the research grant 111-2115-M-008-009- MY3, 111-2628-M-A49-008-MY4 and 110-2115-M-A49-011-MY3, respectively.

Appendix

Here, we present the derivations of the relation between surface differential operators and conventional differential operators in Euclidean space. We begin by considering the surface gradient operator ∇_s, which describes the changing rate along a regular surface (tangent to the surface) by removing the normal component in conventional gradient

$$\nabla_s u = \nabla u - \partial_n u\,\mathbf{n} = (I - \mathbf{n}\mathbf{n}^T)\nabla u,$$

where both $\mathbf{n}$ and ∇u are aligned as column vectors. On the other hand, the surface divergence operator reads

$$\nabla_s \cdot \mathbf{v} = \left[(I - \mathbf{n}\mathbf{n}^T)\nabla\right]^T \mathbf{v} = \nabla^T (I - \mathbf{n}\mathbf{n}^T)\mathbf{v} = \nabla \cdot \mathbf{v} - \mathbf{n}^T(\nabla \mathbf{v})\mathbf{n}.$$

Combining the above identities, we compute the Laplace–Beltrami operator by

$$\begin{aligned}
\Delta_s u &= \nabla_s \cdot (\nabla_s u) = \nabla_s \cdot (\nabla u - \partial_n u\,\mathbf{n}) \\
&= \nabla \cdot (\nabla u - \partial_n u\,\mathbf{n}) - \mathbf{n}^T \left(\nabla(\nabla u - \partial_n u\,\mathbf{n})\right)\mathbf{n} \\
&= \Delta u - (\nabla \cdot \mathbf{n})\partial_n u - (\nabla \partial_n u)\cdot \mathbf{n} - \mathbf{n}^T(\nabla^2 u)\mathbf{n} + \mathbf{n}^T(\nabla(\partial_n u\,\mathbf{n}))\mathbf{n} \\
&= \Delta u - 2H\partial_n u - \mathbf{n}^T(\nabla^2 u)\mathbf{n},
\end{aligned}$$

where we have used the fact that $\nabla \cdot \mathbf{n} = 2H$ and $\mathbf{n}^T \nabla \mathbf{n} = \mathbf{0}$.

References

[1] W.-F. Hu, M.-C. Lai, C. Misbah, A coupled immersed boundary and immersed interface method for interfacial flows with soluble surfactant, Comput. & Fluids 168 (2018) 201–215.
[2] G. Ayton, J. McWhirter, P. McMurty, G. Voth, Coupling field theory with continuum mechanics: A simulation of domain formation in giant unilamellar vesicles, Biophys. J. 88 (2005) 3855–3869.
[3] C. Elliott, B. Stinner, C. Venkataraman, Modelling cell motility and chemotaxis with evolving surface finite elements, J. R. Soc. Interface (2012) 20120276.
[4] C. Elliott, B. Stinner, Modeling and computation of two phase geometric biomembranes using surface finite elements, J. Comput. Phys. 229 (2010) 6585–6612.
[5] M. Bertalmío, A. Bertozzi, G. Sapiro, Navier–Stokes, fluid dynamics, and image and video inpainting, in: Proceedings of IEEE-CVPR, 2001, pp. 355–362.
[6] P. Tang, F. Qiu, H. Zhang, Y. Yang, Phase separation patterns for diblock copolymers on spherical surfaces: A finite volume method, Phys. Rev. E 72 (2005) 016710.
[7] S. Auer, R. Westermann, A semi-Lagrangian closest point method for deforming surfaces, Comput. Graph. Forum 32 (2013) 207–214.
[8] G. Dziuk, C.M. Elliott, Finite elements on evolving surfaces, IMA J. Numer. Anal. 27 (2007) 262–292.
[9] G. Dziuk, C.M. Elliott, Finite element methods for surface PDEs, Acta Numer. 22 (2013) 289–396.
[10] C. Zimmermann, D. Toshniwal, C. Landis, T. Hughes, K. Mandadapu, R. Sauer, An isogeometric finite element formulation for phase transitions on deforming surfaces, Comput. Methods Appl. Mech. Engrg. 351 (2019) 441–477.
[11] N. Valizadeh, T. Rabczuk, Isogeometric analysis for phase-field models of geometric PDEs and high-order PDEs on stationary and evolving surfaces, Comput. Methods Appl. Mech. Engrg. 351 (2019) 599–642.
[12] M. O'Neil, Second-kind integral equations for the Laplace–Beltrami problem on surfaces in three dimensions, Adv. Comput. Math. 44 (2018) 1385–1409.
[13] B. Gross, P.J. Atzberger, Spectral numerical exterior calculus methods for differential equations on radial manifolds, J. Sci. Comput. 76 (2018) 145–165.
[14] D. Álvarez, P. González-Rodríguez, M. Moscoso, A closed-form formula for the RBF-based approximation of the Laplace–Beltrami operator, J. Sci. Comput. 77 (2018) 1115–1132.
[15] H. Wendland, J. Künemund, Solving partial differential equations on (evolving) surfaces with radial basis functions, Adv. Comput. Math. 46 (64).
[16] M. Bertalmío, L.-T. Chen, S. Osher, Variational problems and partial differential equations on implicit surfaces, J. Comput. Phys. 174 (2001) 759–780.
[17] S.J. Ruuth, B. Merriman, A simple embedding method for solving partial differential equations on surfaces, J. Comput. Phys. 227 (2008) 1943–1961.
[18] A. Petras, S. Ruuth, PDEs on moving surfaces via the closest point method and a modified grid based particle method, J. Comput. Phys. 312 (2016) 139–156.
[19] A. Petras, L. Ling, C. Piret, S. Ruuth, A least-squares implicit RBF-FD closest point method and applications to PDEs on moving surfaces, J. Comput. Phys. 381 (2019) 146–161.
[20] S. Leung, H. Zhao, A grid based particle method for moving interface problems, J. Comput. Phys. 228 (2009) 2993–3024.
[21] S. Leung, J. Lowengrub, H. Zhao, A grid based particle method for solving partial differential equations on evolving surfaces and modeling high order geometrical motion, J. Comput. Phys. 230 (2011) 2540–2561.
[22] Z. Fang, J. Zhan, A physics-informed neural network framework for PDEs on 3D surfaces: Time independent problems, IEEE Access 8 (2019) 26328–26335.
[23] Z. Fang, J. Zhang, X. Yang, A physics-informed neural network framework for partial differential equations on 3d surfaces: time-dependent problems, 2021, arXiv:2103.13878.
[24] M. Raissi, P. Perdikaris, 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 (2019) 686–707.
[25] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signal Syst. 2 (4) (1989) 303–314.
[26] H. Hornik, Approximation capabilities of multilayer feedforward networks, Neural Netw. 4 (1991) 251–257.
[27] H.N. Mhaskar, Neural networks for optimal approximation of smooth and analytic functions, Neural Comput. 8 (1996) 164–177.
[28] W.-F. Hu, T.-S. Lin, M.-C. Lai, A discontinuity capturing shallow neural network for elliptic interface problems, J. Comput. Phys. 469 (2022) 111576.
[29] M.-C. Lai, C.-C. Chang, W.-S. Lin, W.-F. Hu, T.-S. Lin, A shallow Ritz method for elliptic problems with singular sources, J. Comput. Phys. 469 (2022) 111547.
[30] Y.-H. Tseng, T.-S. Lin, W.-F. Hu, M.-C. Lai, A cusp-capturing PINN for elliptic interface problems, J. Comput. Phys. 491 (2023) 112359.
[31] Z. Tang, Z. Fu, S. Reutskiy, An extrinsic approach based on physics-informed neural networks for PDEs on surfaces, Mathematics 10 (2022) 2861.
[32] S.W. Walker, The Shapes of Things: a Practical Guide to Differential Geometry and the Shape Derivative, SIAM, 2015.
[33] S. Veerapaneni, A. Rahimian, G. Biros, D. Zorin, A fast algorithm for simulating vesicle flows in three dimensions, J. Comput. Phys. 230 (2011) 5610–5634.
[34] Y. Seol, S.-H. Hsu, M.-C. Lai, An immersed boundary method for simulating interfacial flows with insoluble surfactant in three dimensions, Commun. Comput. Phys. 23 (2018) 640–664.
[35] A. Torres-Sánchez, D. Millán, M. Arroyo, Modelling fluid deformable surfaces with an emphasis on biological interfaces, J. Fluid Mech. 872 (2019) 218–271.

[36] A. Sahu, Y. Omar, R. Sauer, K. Mandadapu, Arbitrary Lagrangian-Eulerian finite element method for curved and deforming surfaces: I. General theory and application to fluid interfaces, J. Comput. Phys. 407 (2020) 109253.
[37] S. Reuther, I. Nitschke, A. Voigt, A numerical approach for fluid deformable surfaces, J. Fluid Mech. 900 (2020) R8.
[38] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, Automatic differentiation in machine learning: A survey, J. Mach. Learn. Res. 18 (2018) 1–43.
[39] P.-O. Persson, G. Strang, A simple mesh generator in MATALB, SIAM Rev. 46 (2004) 329–345.
[40] D. Marquardt, An algorithm for least-squares estimation of nonlinear parameters, SIAM J. Appl. Math. 11 (1963) 431–441.
[41] B. Hanin, M. Sellke, Adam: A method for stochastic optimization, 2018, arXiv:1710.11278.
[42] D. Liu, J. Nocedal, On the limited memory BFGS method for large scale optimization, Math. Program. 45 (1989) 503–528.
[43] A. Iserles, A First Course in the Numerical Analysis of Differential Equations, Cambridge University Press, 2009.
[44] M. Stein, Large sample properties of simulations using latin hypercube sampling, Technometrics 29 (1987) 143–151.
[45] S.-H. Hsu, J. Chu, M.-C. Lai, R. Tsai, A coupled grid based particle and implicit boundary integral method for two-phase flows with insoluble surfactant, J. Comput. Phys. 395 (2019) 747–764.
[46] A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, M. Mahoney, Characterizing possible failure modes in physics-informed neural networks, Adv. Neural Inf. Process. Syst. 34 (2021) 26548–26560.
[47] M. Penwarden, A. Jagtap, S. Zhe, G. Karniadakis, R. Kirby, A unified scalable framework for causal sweeping strategies for Physics-Informed Neural Networks (PINNs) and their temporal decompositions. arXiv:2302.14227.
[48] Z. Liu, W. Cai, Z.-Q.J. Xu, Multi-scale deep neural network (MscaleDNN) for solving Poisson–Boltzmann equation in complex domains, Commun. Comput. Phys. 28 (5) (2020) 1970–2001.
[49] P.-T. Choi, K.T. Ho, L.M. Lui, Spherical conformal parameterization of genus-0 point clouds for meshing, SIAM J. Imaging Sci. 9 (2016) 1582–1618.
[50] P.-T. Choi, Y. Liu, L.M. Lui, Free-boundary conformal parameterization of point clouds, J. Sci. Comput. 90 (2022) 14.
[51] A. Bobenko, J. Sullivan, P. Schröder, G. Ziegler, Discrete Differential Geometry, Springer, 2008.