JAX-DIPS: Neural bootstrapping of finite discretization methods and application to elliptic problems with discontinuities

Pouria A. Mistani* [†,a], Samira Pakravan[†,b], Rajesh Ilango[a], Frederic Gibou[b]

[a] *NVIDIA Corporation, Santa Clara, CA 95051, USA*
[b] *University of California, Santa Barbara, CA 93106-5070, USA*

Abstract

We present a scalable strategy for development of mesh-free hybrid neuro-symbolic partial differential equation solvers based on existing mesh-based numerical discretization methods. Particularly, this strategy can be used to efficiently train neural network surrogate models for the solution functions and operators of partial differential equations while retaining the accuracy and convergence properties of the state-of-the-art numerical solvers. The presented neural bootstrapping method (hereby dubbed NBM) is based on evaluation of the finite discretization residuals of the PDE system obtained on implicit Cartesian cells centered on a set of random collocation points with respect to trainable parameters of the neural network. We apply NBM to the important class of elliptic problems with jump conditions across irregular interfaces in three spatial dimensions. We show the method is convergent such that model accuracy improves by increasing number of collocation points in the domain. The algorithms presented here are implemented and [1] in a software package named `JAX-DIPS` (https://github.com/JAX-DIPS/JAX-DIPS), standing for differentiable interfacial PDE solver. `JAX-DIPS` is purely developed in `JAX`, offering end-to-end differentiability from mesh generation to the higher level discretization abstractions, geometric integrations, and interpolations, thus facilitating research into use of differentiable algorithms for developing hybrid PDE solvers.

Keywords: level-set method, free boundary problems, surrogate models, jump conditions, differentiable programming, neural networks

1. Introduction

1.1. Problem statement

Consider a closed irregular interface (Γ) that partitions the computational domain (Ω) into interior (Ω^-) and exterior (Ω^+) subdomains; *i.e.*, $\Omega = \Omega^- \cup \Gamma \cup \Omega^+$. We are interested in the solutions $\mathrm{u}^\pm \in \Omega^\pm$ to the following class of linear elliptic problems in $\mathbf{x} \in \Omega^\pm$:

$$\begin{aligned} k^\pm u^\pm - \nabla \cdot (\mu^\pm \nabla u^\pm) &= f^\pm, && \mathbf{x} \in \Omega^\pm \\ [u] &= \alpha, && \mathbf{x} \in \Gamma \\ [\mu \partial_{\mathbf{n}} u] &= \beta, && \mathbf{x} \in \Gamma \end{aligned}$$

Here $f^\pm = f(\mathbf{x} \in \Omega^\pm)$ is the spatially varying source term, $\mu^\pm = \mu(\mathbf{x} \in \Omega^\pm)$ are the diffusion coefficients, and $k^\pm$ are the reaction coefficients in the two domains. We consider Dirichlet boundary conditions in a cubic domain $\Omega = [-\mathrm{L}/2, \mathrm{L}/2]^3$.

This class of problems arise ubiquitously in describing diffusion dominated processes in physical systems and life sciences where sharp and irregular interfaces regulate transport across regions with

*Corresponding author: pmistani@nvidia.com
[1] Pending NVIDIA open sourcing procedure.
[†] These authors contributed equally to this work

different properties. Examples include Poisson-Boltzmann equation for describing electrostatic properties of membranes, colloids and solvated biomolecules with jump in dielectric permittivities [59, 47], electroporation of cell aggregates with nonlinear membrane jump conditions [49], epitaxial growth in fabrication of opto-electronic devices where atomic islands grow by surface diffusion of adatoms across freely moving interfaces [48], solidification of multicomponent alloys used for manufacturing processes with free interfaces separating different phases of matter [64, 11], directed self-assembly of diblock copolymers for next generation lithography [23, 53, 10], multiphase flows with and without phase change, and Poisson-Nernst-Planck equations for electrokinetics. Much of these processes are multiscale and the changes across interfaces must be mathematically modeled and numerically solved as sharp surfaces. Smoothing strategies introduce unphysical characteristics in the solution and lead to systemic errors.

1.2. Literature on relevant finite discretization methods

Several numerical methods have been proposed for accurate solution of this class of problems based on explicit or implicit representation of the interface. Finite element methods rely on explicit meshing of the surface that poses severe challenges [3, 13]. Implicit methods include the Immersed Interface Method (IIM) [37] and its variants [1, 41, 21, 25] that rely on Taylor expansions of the solution on both sides of the interface and modifying the local stencils to impose the jump conditions. The main challenge is evaluating high order jump conditions and surface derivatives along interface. Another method is the Ghost Fluid Method (GFM) [22] that was originally introduced to approximate two-phase compressible flows and later applied to the Poisson problem with jump conditions [42]. The basic idea is to define fictitious fluid regions across the discontinuities by adding jump conditions to the true fluid. While GFM captures the normal jump in solution accurately, the tangential jump is smeared. This was solved by the Voronoi Interface Method (VIM) [26] by applying the GFM treatment on a local Voronoi mesh by adapting a local Cartesian mesh which introduces numerical challenges. Several other approaches include the cut-cell method [17], discontinuous Galerkin and eXtended Finite Element Method (XFEM) [38, 50, 4] among others.

In this work we bootstrap the level-set based finite volume method on Cartesian grids proposed by Bochkov & Gibou (2020) [8]. This method is based on the idea of Taylor expansions in the normal direction and employing one-sided least-square interpolations for imposing jump conditions. In particular, this method offers second order accurate numerical solutions with first order accurate gradients in the L^∞-norm.

1.3. Literature on solving PDEs with neural networks

Since early 1990s, artificial neural networks have been used for solving partial differential equations by (i) mapping the algebraic operations of the discretized PDE systems onto specialized neural network architectures and minimizing the network energy, or (ii) treating the whole neural network as the basic approximation unit whose parameters are adjusted to minimize a specialized error function that includes the differential equation itself with its boundary/initial conditions.

In the first category, neurons output the discretized solution values over a set number of grid points and minimizing the network energy drives the neuronal values towards the solution of the linear system at the mesh points. In this case, the neural network energy is the residual of the finite discretization method summed over all neurons of the network [36]. Although the convergence properties of the finite discretization methods gaurantee and control quality of the obtained solutions, the computational costs grow by increasing resolution and dimensionality. Interestingly, due to regular and sparse structure of the finite discretizations, such locally connected neural network PDE solvers have been implemented on VLSI analog CMOS circuits [24, 16, 15].

The second strategy proposed by Lagaris *et al.* [35] relies on the function approximation capabilities of the neural networks. Encoding the solution everywhere in the domain within a neural network offers a mesh-free, compact, and memory efficient surrogate model for the solution function that can be utilized in subsequent inference tasks. This method has recently re-emerged as the physics-informed neural networks (PINNs) [56] and is widely used. Despite their advantages, these methods lack controllable accuracy and convergence properties of finite discretization methods.

Pursuit of hybrid solvers aims at leveraging the performance gains of neural network inference on modern accelerated hardware with the gauranteed accuracy of finite discretization methods. The hybridization efforts are algorithmic or architectural.

One important algorithmic method is the deep Galerkin method (DGM) [61] that is a neural network extension of the mesh-free Galerkin method where the solution is represented as a deep neural network rather than a linear combination of basis functions. The mesh-free nature of DGM, that stems from the underlying mesh-free Galerkin method, enables solving problems in higher dimensions by training the neural network model to satisfy the PDE operator and its initial and boundary conditions on a randomly sampled set of points rather than on an exponentially large grid. Although the number of points is huge in higher dimensions, the algorithm can process training on smaller batches of data points sequentially. Besides, second order derivatives in PDEs are calculated by a Monte Carlo method that retain scaling to higher dimensions. Another important algorithmic method is the deep Ritz method for solving variational problems [67] that implements a deep nerual network approximation of the trial function that is constrained by numerical quadrature rule for the variational functional, followed by stochastic gradient descent.

Architectural hybridization methods are based on differentiable numerical linear algebra. One emerging class involves implementing differentiable finite discretization solvers and embedding them in the neural network architectures that enable application of end-to-end differentiable gradient based optimization methods. Recently, differentiable solvers have been developed in JAX [12] for fluid dynamic problems, such as `Phi-Flow` [30], `JAX-CFD` [33], and `JAX-FLUIDS` [6]. These methods are suitable for inverse problems where an unknown field is modeled by the neural network, while the model influence is propagated by the differentiable solver into a measurable residual [54, 19, 44]. We also note the classic strategy for solving inverse problems is the adjoint method to obtain the gradient of the loss without differentiation across the solver [5]; however, deriving analytic expression for the adjoint equations is tedious, should be repeated after modification of the problem or its loss function, and can become impractical for multiphysics problems. Other important utilities of differentiable solvers are to model and correct for the solution errors of finite discretization methods [65], learning and controling PDE systems [20, 29].

Neural networks are not only universal approximators of continuous functions, but also of non-linear operators [14]. Although this fact has been leveraged using data-driven strategies for learning differential operators by many authors [43, 7, 40, 39], current authors have demonstrated utility of differentiable solvers to effectively train nonlinear operators without any data in a completely physics-driven fashion, see section on learning the inverse transforms in [54].

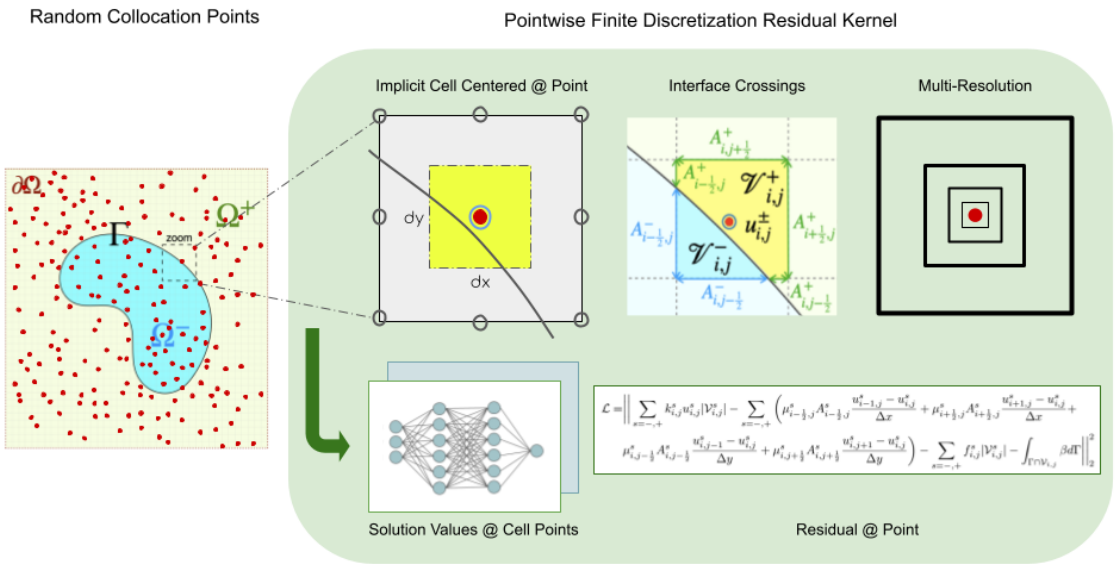
In this work we propose a novel algorithm for solving PDEs based on deep neural networks by lifting any existing mesh-based finite discretization method off of its underlying grid and extend it into a mesh-free method that can be applied to high dimensional problems on unstructured random points in an embarrasingly parallel fashion. In section 2 we present the neural bootstrapping method, next we apply it to an advanced finite volume discretization scheme for elliptic problems with jump conditions across irregular geometries in section 3. We show numerical results of the proposed framework on interfacial PDE problems in section 4 and conclude with section 5.

2. Neural Bootstrapping Method (NBM)

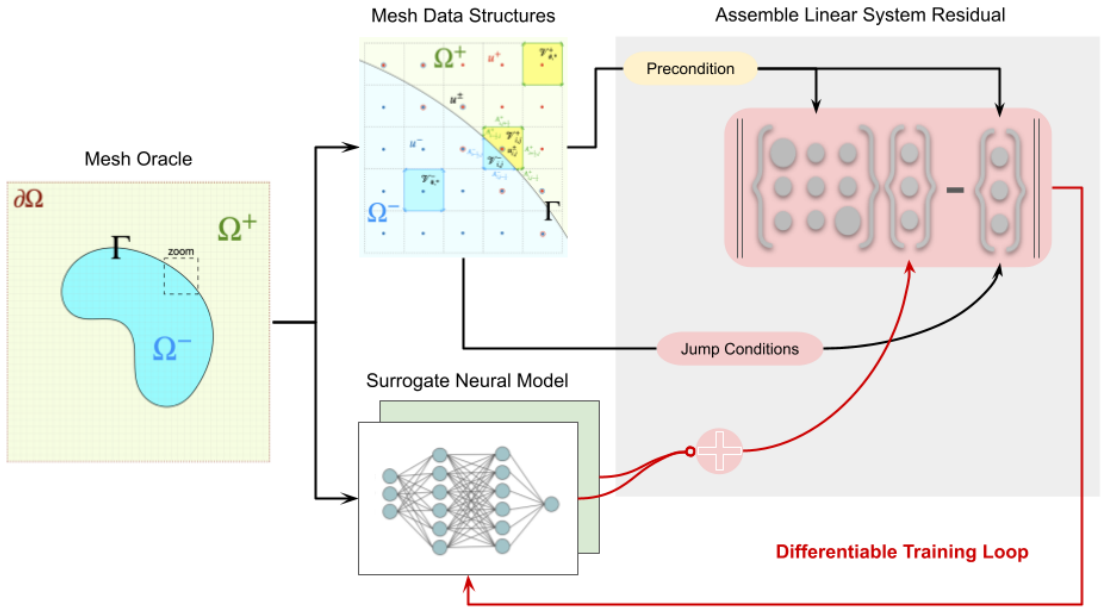
Figure 1 illustrates schematic of the proposed algorithm. Neural networks are used as surrogates for the solution function that are iteratively adjusted to minimize discretization residuals at a set of randomly sampled points and at arbitrary resolutions. The key idea is that neural netwroks can be evaluated over vertices of any discretization stencils centered at any point in the domain effectively emulating the effect of any structured mesh without ever matreializing the mesh. Therefore, we use neural networks to bootstrap mesh based finite discretization (FD) methods to compile mesh free numerical methods.

Operations in differentiable NBM kernels are:

1. A compute cell is implicitly constructed at any input coordinate and at a specified resolution. At the presence of discontinuities a coarse mesh encapsulates an interpolant for the level-set



(a) NBM kernels compute residual contribution by each collocation point per thread. Kernel operations involve considering implicit cells at different resolutions according to the bootstrapped finite discretization method.



(b) NBM outer training layout. Geometric information is managed by a mesh oracle that is often a structured mesh at much lower resolution that stores the level-set funcion. The pointwise evaluations at each implicit cell is locally preconditioned based on the geometry of the interfaces crossing the implicit cells. The training loop involves automatic differentiation across the assembly of the linear system performed by the NBM kernels.

Figure 1: Neural Bootstrapping Method (NBM).

function whose intersection with the implicit cell is calculated to obtain necessary geometric information for the FD kernel and preconditioner.
2. FD kernel is applied on the compute cell where the solution values are evaluated by the neural network. Each kernel contributes a local linear system L^2-norm residual $r_p = ||Au_p - b||$ at a point p.
3. Residulas are preconditioned using common preconditioners to balance relative magnitude of contributions from different points and set them on equal level of importance before summing to produce a global loss value.
4. Gradient based optimization methods used in machine learning are applied to adjust neural network parameters. The automatic differentiation loop passes across the NBM kernels, see figure 1b.

NBM is a simple technique to train neural networks with differential equations that offers several capabilities:

- FD methods offer gauranteed accuracy and controllable convergence properties for the training of neural network surrogate models. These are critical features for solving real-world complex physical systems using neural networks.
- NBM offers a straightforward path for applying mesh-based FD methods on unstructured random points. This is an important ability for augmenting observational data in the training pipelines.
- The algorithm is highly parallelizable and is ideally suited for GPU-accelerated computing paradigm.
- Multi-GPU parallel solution of PDE systems is reduced to the much simpler problem of data-parallel training using existing machine learning frameworks. Data parallelism involves distributing collocation points across multiple processors to compute gradient updates and then aggregating these locally computed updates [58].

In the remainder of this manuscript we present details of applying NBM to solving elliptic problems with discontinuities across irregular interfaces.

3. JAX-DIPS: Differentiable Interfacial PDE Solver

We developed an end-to-end differentiable library for solving the elliptic problems with discontinuities in solution and solution gradient across irregular geometries. We designed `JAX-DIPS` to be a modular open source library for the research community on the level-set methods for solving interfacial PDE problems. We invite the community to contribute to this project for further development of novel numerical algorithms for solving both forward and inverse interfacial PDE problems.

A differentiable solver for free boundary problems requires a rich data structure capable to handle several geometric operations over a given mesh. We understand the mesh as a *topological* relationship between cells that discretize a domain, the geometric information of this discretization as well as the relative configuration of the interfaces with respect to the cells need to be computed and stored in memory in traditional solvers.

In `JAX-DIPS`, we apply NBM to a finite volume discretization method to remove the need for underlying mesh data structures for the elliptic solver. The geometries are represented implicitly using the level-set method on a coarse mesh. We have implemented a uniform grid that supports operations such as interpolations, interface advection, integrations over interfaces as well as in domains. We describe the numerical algorithms for the level-set module and the elliptic solver in this section.

3.1. Interpolation methods

An important building block is the ability to interpolate field values anywhere inside a grid cell given the values on the grid points. In `JAX-DIPS` we currently support two types of interpolation schemes that have been used in the context of the level set method for achieving second-order accurate solutions by Min & Gibou (2007a)[46]: (i) trilinear interpolation, and (ii) quadratic non-oscillatory interpolation.

3.1.1. Trilinear interpolation

In a unit grid cell, rescaled to $\mathcal{C} \in [0,1]^3$, the trilinear interpolation at a point $(x,y,z) \in \mathcal{C}$ uses the grid values on the parent cell vertices according to equation 11 of [46],

$$\phi(x,y,z) = \sum_{i,j,k \in 0,1} \phi(i,j,k)(-1)^{i+j+k}(1-x-i)(1-y-j)(1-z-k)$$

Trilinear interpolation is based on polynomials of order 1 and offers accuracy of order 2 using 8 immediate vertices in a grid cell.

3.1.2. Quadratic non-oscillatory interpolation

Quadratic interpolation extends the trilinear interpolation by adding second order derivatives of the interpolating field. This is needed because trilinear interpolation is sensitive to presence of discontinuities and kinks which are abundant in the context of free boundary problems. The extension reads

$$\begin{aligned}\phi(x,y,z) = &\sum_{i,j,k \in 0,1} \phi(i,j,k)(-1)^{i+j+k}(1-x-i)(1-y-j)(1-z-k) \\ &- \phi_{xx}\frac{x(1-x)}{2} - \phi_{yy}\frac{y(1-y)}{2} - \phi_{zz}\frac{z(1-z)}{2}\end{aligned}$$

where second order derviatives are sampled as the minimum value on the parent cell vertices to enhance numerical stability of the interpolation

$$\begin{aligned}\phi_{xx} &= \min_{v \in \text{vertices}(\mathcal{C})} |D_{xx}\phi_v| \\ \phi_{yy} &= \min_{v \in \text{vertices}(\mathcal{C})} |D_{yy}\phi_v| \\ \phi_{zz} &= \min_{v \in \text{vertices}(\mathcal{C})} |D_{zz}\phi_v|\end{aligned}$$

The second order derivative operator is the familiar 5-point finite difference stencil.

3.2. Level-set method

The level-set method for solving free boundary problems was introduced by Osher & Sethian (1988) [51]. The sharp interface is described as the zero contour of a signed-distance function, ϕ, whose evolution is given by the advection equation according to some velocity field dictated by the physics of the problem, $\mathbf{v}$, that is defined over the moving boundary

$$\frac{\partial \phi}{\partial t} + \mathbf{v} \cdot \nabla \phi = 0$$

This *implicit* representation of the moving boundaries resolves the need for the challenging task of adapting the underlying grid to the yet-unknown discontinuities in the solution field. The computational simplicity of using Cartesian grids for solving free boundary problems with irregular geometries, as well as the ability to simulate freely moving discontinuities in a *sharp*-manner, that is required by the physics of these problems, are the two main offerings of the level-set method for this class of PDE problems. Besides implicit representation of the free boundaries, the level-set function can be used to compute normal vectors to the interface

$$\mathbf{n} = \nabla\phi / |\nabla\phi|$$

as well as the curvature of the interface

$$\kappa = \nabla \cdot \mathbf{n}$$

3.3. Geometric integration

3.3.1. Integration over 3D surfaces and volumes

We use uniform Cartesian grids. For computational cells that are crossed by the interface, *i.e.* $\mathcal{V}_{i,j,k} \cap \Gamma \neq 0$, we use the geometric integrations proposed by Min & Gibou (2007b) [45]. In this scheme each grid cell, $\mathcal{C}$, is decomposed into five tetrahedra by the middle-cut triangulation [57] (each cell is rescaled to $[0,1]^3$) that are described below (also see figure 1 (right) of [45]):

$$
\begin{aligned}
&\mathrm{S}_1 \equiv \mathrm{conv}(\mathrm{P}_{000};\mathrm{P}_{100};\mathrm{P}_{010};\mathrm{P}_{001}) && \mathrm{x}=0 \text{ face, } \mathrm{y}=0 \text{ face, } \mathrm{z}=0 \text{ face}\\
&\mathrm{S}_2 \equiv \mathrm{conv}(\mathrm{P}_{110};\mathrm{P}_{100};\mathrm{P}_{010};\mathrm{P}_{111}) && \mathrm{x}=1 \text{ face, } \mathrm{y}=1 \text{ face, } \mathrm{z}=0 \text{ face}\\
&\mathrm{S}_3 \equiv \mathrm{conv}(\mathrm{P}_{101};\mathrm{P}_{100};\mathrm{P}_{111};\mathrm{P}_{001}) && \mathrm{x}=1 \text{ face, } \mathrm{y}=0 \text{ face, } \mathrm{z}=1 \text{ face}\\
&\mathrm{S}_4 \equiv \mathrm{conv}(\mathrm{P}_{011};\mathrm{P}_{111};\mathrm{P}_{010};\mathrm{P}_{001}) && \mathrm{x}=0 \text{ face, } \mathrm{y}=1 \text{ face, } \mathrm{z}=1 \text{ face}\\
&\mathrm{S}_5 \equiv \mathrm{conv}(\mathrm{P}_{111};\mathrm{P}_{100};\mathrm{P}_{010};\mathrm{P}_{001}) && \text{no face exposure}
\end{aligned}
$$

Hence each 3D grid cell is the union of 5 tetrahedra (simplices) $\mathcal{C} = \cup_{i=1}^{5} S_i$, where each simplex is identified by the pre-existing vertices of the grid cell (hence not creating new grid points). Given the values of the level set function sampled at these vertices one can compute coordinates of intersection points of the interface with each of the simplices $S_i \cap \Gamma$ as well as the negative domain $S_i \cap \Omega^-$. If $P_0, \cdots, P_3$ are the four vertices of a simplex S, then Γ crosses an edge P_iP_j if and only if $\phi(P_i)\phi(P_j) < 0$ and the intersection point across this edge is given by linear interpolation:

$$P_{ij} = P_j \frac{\phi(P_i)}{\phi(P_i) - \phi(P_j)} - P_i \frac{\phi(P_j)}{\phi(P_i) - \phi(P_j)}$$

Number of negative level-set values on the 4 (in 3D) tetrahedron vertices classifies the specific configuration for intersection between simplex S and the interface through a variable $\eta(\phi, S) = n(P_i | \phi(P_i) < 0)$. In 3D, possible values are $\eta = 0, 1, 2, 3, 4$ that correspond to the four configurations for the intersection cross section enumerated below:

- $S \cap \Gamma$, see table 2 and figure 2 of [45]:
  - $\eta = 0$: tetrahedron (S) is completely in positive domain with no intersection, $S \cap \Gamma = \emptyset$.
  - $\eta = 1$: with a single vertex in negative domain and remaining three in positive domain, the tetrahedron and interface have exactly 3 intersection points, the simplex $S \cap \Gamma$ has exactly 3 vertices; *cf.*, see figure 2 (center) of [45].
  - $\eta = 2$: with two vertices in negative domain and remaining two in positive domain, the cross section has four vertices that is splitted into two 3-vertex simplices; *cf.*, see figure 2 (right) of [45].
  - $\eta = 3$: with one vertex in positive domain and remaining three vertices in negative domain, the cross section has 3 vertices that is computed by inverting the sign of the level-set values on vertices and following the instruction for case $\eta = 1$.
  - $\eta = 4$: tetrahedron is completely in negative domain with no intersection, $S \cap \Gamma = \emptyset$.
- $S \cap \Omega^-$, see table 4 and figure 4 of [45]:
  - $\eta = 0$: tetrahedron is completely in positive domain with no intersection, $S \cap \Omega^- = \emptyset$;
  - $\eta = 1$: the intersection $S \cap \Omega^-$ is characterized by a single tetrahedron with 4 vertices according to figure 4 (left) of [45]; *i.e.*, one vertex is the negative level-set vertex of the parent tetrahedron and three others are interpolated points over the three edges pertaining to the negative vertex.
  - $\eta = 2$: the intersection $S \cap \Omega^-$ is characterized by three tetrahedra with 12 vertices according to figure 4 (center) of [45]. Note that there
  - $\eta = 3$: the intersection $S \cap \Omega^-$ is characterized by three tetrahedra with 12 vertices according to figure 4 (right) of [45].

– $\eta = 4$: tetrahedron is completely in negative domain and $S \cap \Omega^- = S$;

Note that although we only need to allocate memory for at most 4 vertices to uniquely identify $S \cap \Gamma$, in `JAX-DIPS` we choose to pre-allocate memory for two 3-vertex simplex data structure per S with a total of 6 vertices to separately store information for the cross section geometry. Similarly for $S \cap \Omega^-$ we pre-allocate memory for a three 4-vertex simplex data structure per S. Altogether, in the current implementation the geometric information of intersection points for each simplex S is expressed in terms of 5 simplicies (2 three-vertex simplices for surface area and 3 four-vertex simplicies for volume) using 18 points; this is an area for future optimization.

Having the intersection points, we compute surface and volume integrals of a given field over the interface Γ and in negative domain Ω^- as a summation of integrals over the identified simplices. For each simplex (with $n = 3$ or $n = 4$ vertices) surface and volume integrals can be numerically computed by having these vertices $P_0, \cdots, P_n$ and the values of the field f at these vertices according to

$$\int_S f dx = \text{vol}(S) \cdot \frac{f(P_0) + \cdots + f(P_n)}{n+1}$$

where

$$\text{vol}(S) = \frac{1}{n!} \left| \det \begin{pmatrix} (P_1 - P_0)\hat{e}_1 & \cdots & (P_n - P_0)\hat{e}_1 \\ \vdots & \ddots & \vdots \\ (P_1 - P_0)\hat{e}_n & \cdots & (P_n - P_0)\hat{e}_n \end{pmatrix} \right|$$

with $\hat{e}_i$ being the i^{th} Cartesian unit basis vector.

3.3.2. Cross sections of interface with grid cell faces

For the numerical discretizations considered in this work we also need the surface areas for simplices created at the intersection of Γ with each of the 6 faces of a grid cell $\mathcal{C}$. In `JAX-DIPS` for each face we reuse two corresponding simplices exposed to that face that were calculated in the geometric integrations module, explicitly:

- $x = 0$ face has contributions from (S_1, S_4)
- $x = 1$ face has contributions from (S_2, S_3)
- $y = 0$ face has contributions from (S_1, S_3)
- $y = 1$ face has contributions from (S_2, S_4)
- $z = 0$ face has contributions from (S_1, S_2)
- $z = 1$ face has contributions from (S_3, S_4)

For each face, we extract vertices from (S_i, S_j)-pair that lie on the considered face and sum the surface areas in (negative domain) contributed from each simplex on that face; therefore, the portion of the face surface area in the positive domain is simply the complementing value $\text{area}^+ = \text{area}_{\text{face}} - \text{area}^-$; *i.e.*, this ensures sum of areas adds up to the exact face area in downstream computations.

3.4. Neural network approximators for the solution

In 1987, Hecht and Nielson [27] applied an improved version of Kolmogorov's 1957 superposition theorem [34], due to Sprecher [62], to the field of neurocomputing and demonstrated that a 3-layer feedforward neural network (one input layer with n inputs, one hidden layer with $2n+1$ neurons, one output layer) are universal approximators for all *continuous* functions from the n-dimensional cube to a finite m-dimensional real vector space; *i.e.*, $f : [0, 1]^n \to \mathbb{R}^m$. Recently, Ismailov (2022) [31] demonstrated existence of neural networks implementing *discontinuous* functions, however efficient learning algorithms for such networks are not still available.

The solutions of interfacial PDE problems are discontinuous, with jumps appearing not only in the solution but also in the solution gradient. In light of above considerations, we define two separate neural networks to represent solution in Ω^- and Ω^+ regions:

$$u^+ = \mathcal{N}^+(\mathbf{x}) : \mathbb{R}^3 \cap \Omega^+ \to \mathbb{R} \qquad\qquad u^- = \mathcal{N}^-(\mathbf{x}) : \mathbb{R}^3 \cap \Omega^- \to \mathbb{R}$$

We use SIREN neural networks, where we implement fully connected feedforward architecture with `sin` activation function and the output layer is a single linear neuron. Note that piecewise differentiable nonlinearities such as the `ReLU` function are inappropriate choices for representing solutions to differential equations. Weights and biases are initialized from a truncated normal distribution with zero mean and unit variance.

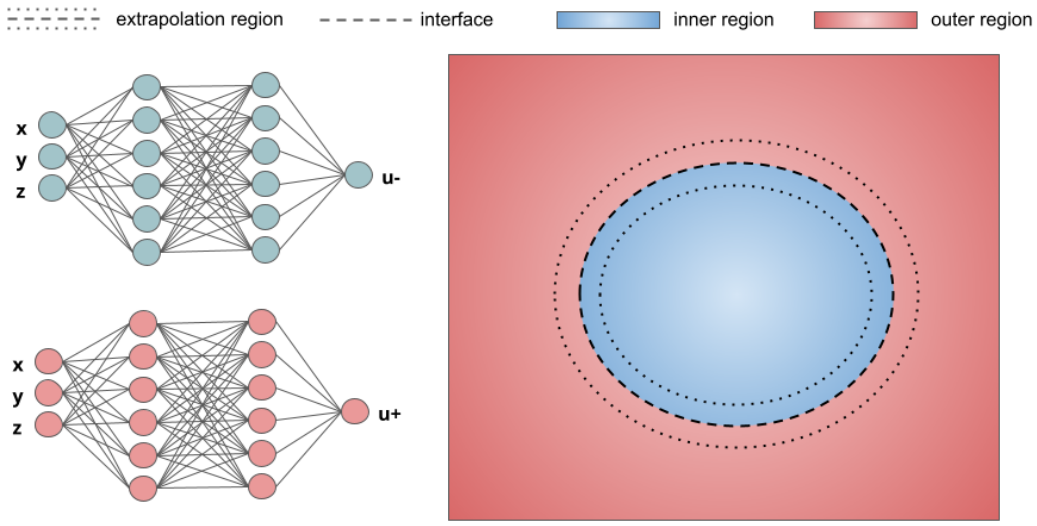


Figure 2: Two neural networks are defined for the two regions of the computational domain.

Solution networks are evaluated on sampled points in the domain while the parameters of these networks are optimized using the loss function. We define the loss function by the mean-squared-error (MSE) of the residual of the discretized partial differential equation with jump conditions derived in section 3.5 that is evaluated on the grid points:

$$\mathcal{L}(u) = ||A\hat{u}(\mathbf{x}_{ijk} \in \Omega) - b||_2^2$$

`JAX-DIPS` allows for computation of the gradient of the loss function using automatic differentiation, *i.e.* $\nabla_u \mathcal{L}(u)$. Therefore, our strategy is to leverage this capability and use more sophisticated optimizers developed in the deep learning community (*e.g.* Adam [32], RMSProp, *etc*) to minimize the afforementioned loss function with the desired solution vector u^* of the PDE problem.

We emphasize the main motivation for using first order gradient based optimization algorithms over the existing methods such as GMRES and Conjugate Gradient is that optimizers in the field of machine learning are suitable for large scale optimiziation problems (very large number of parameters of the deep neural networks) because they have better memory efficiency. This is critical for training large scale neural network models that can encode multiscale simulations.

3.5. Approach I. Finite discretization method fused with regression-based extrapolation

For spatial discretizations at the presence of jump conditions we employ the numerical algorithm proposed by Bochkov and Gibou (2020) [9]. Method of [9] produces second-order accurate solutions

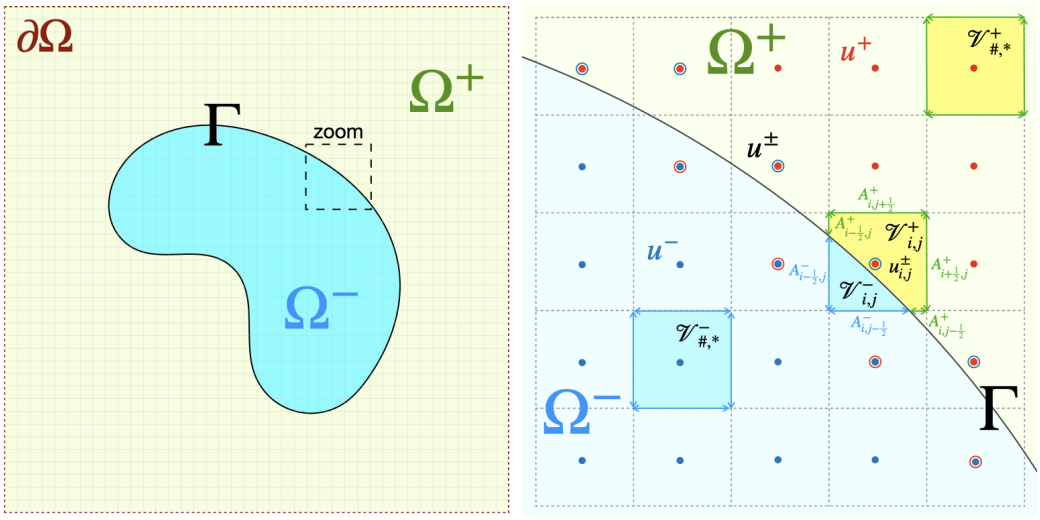


Figure 3: Notation used in this paper. Close to the interface where finite volumes are crossed by the interface, there are extra degrees of freedom (open circles) that are extrapolations of solutions from each domain to the opposite domain. Jump conditions are implicitly encoded in these extrapolated values.

and first-order accuracte gradients in the L^∞-norm, while having a compact stencil that makes it a good candidate for parallelization. Moreover, treatment of the interface jump conditions do not introduce any augmented variables, this preserves the homogeneous structure of the linear system. Most importantly, jump conditions only appear on the right-hand-side of the discretization and do not pollute the matrix term, this is beneficial for accelerating the solver.

Here we use a background uniform 2D grid only for presentation of the finite volume discretization equations; we will not use this grid in the actual implementation but instead assume a local 3D cell around random points spanning in the domain during optimization.

At points where the finite volumes are crossed by Γ we have

$$\sum_{s=-,+}\int_{\Omega^s\cap\mathcal{V}_{i,j}} k^s u^s d\Omega - \sum_{s=-,+}\int_{\Omega^s\cap\partial\mathcal{V}_{i,j}} \mu^s \partial_{\mathbf{n}^s} u^s d\Gamma = \sum_{s=-,+}\int_{\Omega^s\cap\mathcal{V}_{i,j}} f^s d\Omega + \int_{\Gamma\cap\mathcal{V}_{i,j}} [\mu\partial_{\mathbf{n}} u] d\Gamma$$

following standard treatment of volumetric integrals and using central differencing for derivatives we obtain in 2D (with trivial 3D extension)

$$\begin{aligned}
&\sum_{s=-,+} k^s_{i,j} u^s_{i,j} |\mathcal{V}^s_{i,j}| - \sum_{s=-,+}\left(\mu^s_{i-\frac{1}{2},j} A^s_{i-\frac{1}{2},j}\frac{u^s_{i-1,j}-u^s_{i,j}}{\Delta x} + \mu^s_{i+\frac{1}{2},j} A^s_{i+\frac{1}{2},j}\frac{u^s_{i+1,j}-u^s_{i,j}}{\Delta x} + \right.\\
&\left.\mu^s_{i,j-\frac{1}{2}} A^s_{i,j-\frac{1}{2}}\frac{u^s_{i,j-1}-u^s_{i,j}}{\Delta y} + \mu^s_{i,j+\frac{1}{2}} A^s_{i,j+\frac{1}{2}}\frac{u^s_{i,j+1}-u^s_{i,j}}{\Delta y}\right)\\
&= \sum_{s=-,+} f^s_{i,j}|\mathcal{V}^s_{i,j}| + \int_{\Gamma\cap\mathcal{V}_{i,j}} \beta d\Gamma + \mathcal{O}(\max(\Delta x,\Delta y)^{\mathcal{D}})
\end{aligned}$$

where $\mathcal{D}$ is the problem dimensionality.

Note that far from interface either $s=-$ (for $\mathbf{x}\in\Omega^-$) or $s=+$ (for $\mathbf{x}\in\Omega^+$) is retained. This is automatically considered through zero values for sub-volumes $|\mathcal{V}^+_{i,j}|$ and $|\mathcal{V}^-_{i,j}|$ as well as their face areas. Note that $\mu^-_{i-1/2,j}$ (or $\mu^+_{i-1/2,j}$) corresponds to the value of diffusion coefficient at the middle of segment $A^-_{i-1/2,j}$ (or $A^+_{i-1/2,j}$) respectively, same is true for other edges as well. However, there are extra degrees of freedom on grid points whose finite volumes are crossed by the interface; *i.e.*, see double circles in figure 3. [9] derived analytical expressions for the extra degrees of freedom (u^+

in Ω^- and u^- in Ω^+) in terms of the original degrees of freedom (u^- in Ω^- and u^+ in Ω^+) as well as the jump conditions, this preserves the original $\mathrm{N_x} \times \mathrm{N_y}$ system size.

In this scheme the basic idea is to extrapolate the jump at grid point from jump condition at the projected point onto the interface using a Taylor expansion: $u^+_{i,j} - u^-_{i,j} = [u]_{\mathbf{r}^{pr}_{i,j}} + \delta_{i,j}(\partial_{\mathbf{n}} u^+(\mathbf{r}^{pr}_{i,j}) - \partial_{\mathbf{n}} u^-(\mathbf{r}^{pr}_{i,j}))$. The unknown value ($u^-_{i,j}$ or $u^+_{i,j}$) is obtained based on approximation of the normal derivatives (*i.e.* $\partial_{\mathbf{n}} u^{\pm}(\mathbf{r}^{pr}_{i,j})$) which are computed using a least squares calculation on neighboring grid points that are in the fast-diffusion region (referred to as "Bias Fast") or in the slow diffusion region (referred to as "Bias Slow"). This makes two sets of rules for unknown values $u^{\pm}_{i,j}$.

In two dimensions and on uniform grids, the gradient operator at the grid cell (i, j) that is crossed by an interface is estimated by a least squares solution given by

$$(\nabla u^{\pm})_{i,j} = \mathbf{D}^{\pm}_{i,j} \begin{bmatrix} u_{i-1,j-1} - u^{\pm}_{i,j} \\ u_{i,j-1} - u^{\pm}_{i,j} \\ \vdots \\ u_{i+1,j+1} - u^{\pm}_{i,j} \end{bmatrix} \qquad \mathbf{D}^{\pm}_{i,j} = \left(X^T_{i,j} W^{\pm}_{i,j} X_{i,j}\right)^{-1} \left(W^{\pm}_{i,j} X_{i,j}\right)^T$$

and

$$W^{\pm}_{i,j} = \begin{bmatrix} \omega^{\pm}_{i,j}(-1,-1) & & & \\ & \omega^{\pm}_{i,j}(0,-1) & & \\ & & \ddots & \\ & & & \omega^{\pm}_{i,j}(1,1) \end{bmatrix} \qquad X_{i,j} = \begin{bmatrix} -h_x & -h_y \\ 0 & -h_y \\ h_x & -h_y \\ -h_x & 0 \\ 0 & 0 \\ h_x & 0 \\ -h_x & h_y \\ 0 & h_y \\ h_x & h_y \end{bmatrix}$$

and

$$\omega^{\pm}_{i,j}(p,q) = \begin{cases} 1 & (p,q) \in N^{\pm}_{i,j} \\ 0 & \text{else} \end{cases} \tag{1}$$

In this case, $D^{\pm}_{i,j}$ is a 2×9 matrix and we denote each of its 2×1 columns with $d^{\pm}_{i,j,p,q}$

$$\mathbf{D}^{\pm}_{i,j} = \begin{bmatrix} d^{\pm}_{i,j,-1,-1} & d^{\pm}_{i,j,0,-1} & d^{\pm}_{i,j,1,-1} & d^{\pm}_{i,j,-1,0} & d^{\pm}_{i,j,0,0} & d^{\pm}_{i,j,1,0} & d^{\pm}_{i,j,-1,1} & d^{\pm}_{i,j,0,1} & d^{\pm}_{i,j,1,1} \end{bmatrix}$$

The least square coefficients are then obtained by dot product of normal vector with these columns

$$c^{\pm}_{i,j,p,q} = \mathbf{n}^T_{i,j} d^{\pm}_{i,j,p,q}$$

and normal derivative can be computed (noting that $c^{\pm}_{i,j} = -\sum_{(p,q)\in N^{\pm}_{i,j}} c^{\pm}_{i,j,p,q}$)

$$\partial_n u^{\pm}(\mathbf{r}^{proj}_{i,j}) = c^{\pm}_{i,j} u^{\pm}_{i,j} + \sum_{(p,q)\in N^{\pm}_{i,j}} c^{\pm}_{i,j,p,q} u^{\pm}_{i+p,j+q} + \mathcal{O}(h)$$

At this point we can define a few intermediate variables at each grid point to simplify the presentation of the method,

$$\zeta^{\pm}_{i,j,p,q} := \delta_{i,j} \frac{[\mu]}{\mu^{\mp}} c^{\pm}_{i,j,p,q} \qquad \zeta^{\pm}_{i,j} := -\sum_{(p,q)\in N^{\pm}_{i,j}} \zeta^{\pm}_{i,j,p,q}$$

$$\gamma^{\pm}_{i,j,p,q} := \frac{\zeta^{\pm}_{i,j,p,q}}{1 \pm \zeta^{\pm}_{i,j}} \qquad \gamma^{\pm}_{i,j} := -\sum_{(p,q)\in N^{\pm}_{i,j}} \gamma^{\pm}_{i,j,p,q}$$

where the set of neighboring grid points are

$$N_{i,j}^{\pm} = \{(p,q): \quad p=-1,0,1, \quad q=-1,0,1, \quad (p,q)\neq(0,0), \quad \mathbf{x}_{i+p,j+q}\in\Omega^{\pm}\}$$

and $\delta_{i,j}$ is the signed distance from $\mathbf{x}_{i,j}$ that is computed from the level-set function $\phi(\mathbf{x})$

$$\delta_{i,j} = \frac{\phi(\mathbf{x}_{i,j})}{|\nabla\phi(\mathbf{x}_{i,j})|}$$

- Rules based on approximating $\partial_{\mathbf{n}}u^{+}(\mathbf{r}_{i,j}^{pr})$:

$$u_{i,j}^{-} = \begin{cases} u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{-} \\ u_{i,j}(1-\gamma_{i,j}^{-}) - \sum_{(p,q)\in N_{i,j}^{-}}\gamma_{i,j,p,q}^{-}u_{i+p,j+q} - (\alpha+\frac{\delta_{i,j}\beta}{\mu^{+}})(1-\gamma_{i,j}^{-}) & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{2}$$

$$u_{i,j}^{+} = \begin{cases} u_{i,j}(1-\zeta_{i,j}^{-}) - \sum_{(p,q)\in N_{i,j}^{-}}\zeta_{i,j,p,q}^{-}u_{i+p,j+q} + \alpha + \delta_{i,j}\frac{\beta}{\mu^{+}} & \mathbf{x}_{i,j}\in\Omega^{-} \\ u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{3}$$

It is useful to cast this in the form of matrix kernel operations through defining intermediate tensors:

$$\mathbf{\Gamma}_{i,j} := \begin{bmatrix} \gamma_{i-1,j+1}^{-} & \gamma_{i,j+1}^{-} & \gamma_{i+1,j+1}^{-} \\ \gamma_{i-1,j}^{-} & \gamma_{i,j}^{-} & \gamma_{i+1,j}^{-} \\ \gamma_{i-1,j-1}^{-} & \gamma_{i,j-1}^{-} & \gamma_{i+1,j-1}^{-} \end{bmatrix}, \qquad \boldsymbol{\zeta}_{i,j} := \begin{bmatrix} \zeta_{i-1,j+1}^{-} & \zeta_{i,j+1}^{-} & \zeta_{i+1,j+1}^{-} \\ \zeta_{i-1,j}^{-} & \zeta_{i,j}^{-} & \zeta_{i+1,j}^{-} \\ \zeta_{i-1,j-1}^{-} & \zeta_{i,j-1}^{-} & \zeta_{i+1,j-1}^{-} \end{bmatrix}$$

$$\mathbf{U}_{i,j} := \begin{bmatrix} u_{i-1,j+1} & u_{i,j+1} & u_{i+1,j+1} \\ u_{i-1,j} & u_{i,j} & u_{i+1,j} \\ u_{i-1,j-1} & u_{i,j-1} & u_{i+1,j-1} \end{bmatrix}, \quad \mathbf{N}_{i,j}^{\pm} := \begin{bmatrix} \omega_{i,j}^{\pm}(-1,1) & \omega_{i,j}^{\pm}(0,1) & \omega_{i,j}^{\pm}(1,1) \\ \omega_{i,j}^{\pm}(-1,0) & 0 & \omega_{i,j}^{\pm}(1,0) \\ \omega_{i,j}^{\pm}(-1,-1) & \omega_{i,j}^{\pm}(0,-1) & \omega_{i,j}^{\pm}(1,-1) \end{bmatrix}$$

where $\mathbf{N}^{-}$ is a masking filter that passes the values in the negative neighborhood of node (i,j).

We also introduce the Hadamard product $\odot$ between two identical matrices that creates another identical matrix with each entry being elementwise products. Moreover, double contraction of two tensors A and B is defined by $A:B=\sum A\odot B$ which is a scalar value and equals the sum of all entries of the Hadamard product of the tensors; *i.e.*, note $A:A$ is square of Frobenius norm of A. Using these notations, the substitution rules read

$$u_{i,j}^{-} = \begin{cases} u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{-} \\ \big(1+\mathbf{\Gamma}_{i,j}^{-}:\mathbf{N}_{i,j}^{-}\big)u_{i,j} - \big(\mathbf{\Gamma}_{i,j}^{-}\odot\mathbf{N}_{i,j}^{-}\big):\mathbf{U}_{i,j} - (\alpha+\delta_{i,j}\frac{\beta}{\mu^{+}})\big(1+\mathbf{\Gamma}_{i,j}^{-}:\mathbf{N}_{i,j}^{-}\big) & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{4}$$

$$u_{i,j}^{+} = \begin{cases} \big(1+\boldsymbol{\zeta}_{i,j}^{-}:\mathbf{N}_{i,j}^{-}\big)u_{i,j} - \big(\boldsymbol{\zeta}_{i,j}^{-}\odot\mathbf{N}_{i,j}^{-}\big):\mathbf{U}_{i,j} + \alpha + \delta_{i,j}\frac{\beta}{\mu^{+}} & \mathbf{x}_{i,j}\in\Omega^{-} \\ u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{5}$$

- Rules based on approximating $\partial_{\mathbf{n}}u^{-}(\mathbf{r}_{i,j}^{pr})$:

$$u_{i,j}^{-} = \begin{cases} u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{-} \\ u_{i,j}(1-\zeta_{i,j}^{+}) - \sum_{(p,q)\in N_{i,j}^{+}}\zeta_{i,j,p,q}^{+}u_{i+p,j+q} - \alpha - \delta_{i,j}\frac{\beta}{\mu^{-}} & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{6}$$

$$u_{i,j}^{+} = \begin{cases} u_{i,j}(1-\gamma_{i,j}^{+}) - \sum_{(p,q)\in N_{i,j}^{+}}\gamma_{i,j,p,q}^{+}u_{i+p,j+q} + (\alpha+\delta_{i,j}\frac{\beta}{\mu^{-}})(1-\gamma_{i,j}^{+}) & \mathbf{x}_{i,j}\in\Omega^{-} \\ u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{7}$$

in matrix notation we have

$$u_{i,j}^{-} = \begin{cases} u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{-} \\ \big(1+\boldsymbol{\zeta}_{i,j}^{+}:\mathbf{N}_{i,j}^{+}\big)u_{i,j} - \big(\boldsymbol{\zeta}_{i,j}^{+}\odot\mathbf{N}_{i,j}^{+}\big):\mathbf{U}_{i,j} - \alpha - \delta_{i,j}\frac{\beta}{\mu^{-}} & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{8}$$

$$u_{i,j}^{+} = \begin{cases} \big(1+\mathbf{\Gamma}_{i,j}^{+}:\mathbf{N}_{i,j}^{+}\big)u_{i,j} - \big(\mathbf{\Gamma}_{i,j}^{+}\odot\mathbf{N}_{i,j}^{+}\big):\mathbf{U}_{i,j} + (\alpha+\delta_{i,j}\frac{\beta}{\mu^{-}})\big(1+\mathbf{\Gamma}_{i,j}^{+}:\mathbf{N}_{i,j}^{+}\big) & \mathbf{x}_{i,j}\in\Omega^{-} \\ u_{i,j} & \mathbf{x}_{i,j}\in\Omega^{+} \end{cases} \tag{9}$$

procedure BIAS SLOW

if $\Gamma \cap \mathcal{C}_{i,j} = \emptyset$ **then**

$$B^{\pm}_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \quad r^{\pm}_{i,j} = 0$$

else

if $\mu^{-}_{i,j} > \mu^{+}_{i,j}$ **then**

if $\phi_{i,j} \geq 0$ **then**

$$B^{+}_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \quad r^{+}_{i,j} = 0$$

$$B^{-}_{i,j} = \begin{bmatrix} -\gamma^{-}_{i,j,-1,1} & -\gamma^{-}_{i,j,0,1} & -\gamma^{-}_{i,j,1,1} \\ -\gamma^{-}_{i,j,-1,0} & 1-\gamma^{-}_{i,j} & -\gamma^{-}_{i,j,1,0} \\ -\gamma^{-}_{i,j,-1,-1} & -\gamma^{-}_{i,j,0,-1} & -\gamma^{-}_{i,j,1,-1} \end{bmatrix}; \quad r^{-}_{i,j} = -(\alpha^{proj}_{i,j} + \delta_{i,j}\frac{\beta^{proj}_{i,j}}{\mu^{+}_{proj}})(1-\gamma^{-}_{i,j})$$

else

$$B^{+}_{i,j} = \begin{bmatrix} -\zeta^{-}_{i,j,-1,1} & -\zeta^{-}_{i,j,0,1} & -\zeta^{-}_{i,j,1,1} \\ -\zeta^{-}_{i,j,-1,0} & 1-\zeta^{-}_{i,j} & -\zeta^{-}_{i,j,1,0} \\ -\zeta^{-}_{i,j,-1,-1} & -\zeta^{-}_{i,j,0,-1} & -\zeta^{-}_{i,j,1,-1} \end{bmatrix}; \quad r^{+}_{i,j} = \alpha^{proj}_{i,j} + \delta_{i,j}\frac{\beta^{proj}_{i,j}}{\mu^{+}_{proj}}$$

$$B^{-}_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \quad r^{-}_{i,j} = 0$$

else

if $\phi_{i,j} \geq 0$ **then**

$$B^{+}_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \quad r^{+}_{i,j} = 0$$

$$B^{-}_{i,j} = \begin{bmatrix} -\zeta^{+}_{i,j,-1,1} & -\zeta^{+}_{i,j,0,1} & -\zeta^{+}_{i,j,1,1} \\ -\zeta^{+}_{i,j,-1,0} & 1-\zeta^{+}_{i,j} & -\zeta^{+}_{i,j,1,0} \\ -\zeta^{+}_{i,j,-1,-1} & -\zeta^{+}_{i,j,0,-1} & -\zeta^{+}_{i,j,1,-1} \end{bmatrix}; \quad r^{-}_{i,j} = \alpha^{proj}_{i,j} + \delta_{i,j}\frac{\beta^{proj}_{i,j}}{\mu^{-}_{proj}}$$

else

$$B^{+}_{i,j} = \begin{bmatrix} -\gamma^{+}_{i,j,-1,1} & -\gamma^{+}_{i,j,0,1} & -\gamma^{+}_{i,j,1,1} \\ -\gamma^{+}_{i,j,-1,0} & 1-\gamma^{+}_{i,j} & -\gamma^{+}_{i,j,1,0} \\ -\gamma^{+}_{i,j,-1,-1} & -\gamma^{+}_{i,j,0,-1} & -\gamma^{+}_{i,j,1,-1} \end{bmatrix}; \quad r^{+}_{i,j} = (\alpha^{proj}_{i,j} + \delta_{i,j}\frac{\beta^{proj}_{i,j}}{\mu^{-}_{proj}})(1-\gamma^{+}_{i,j})$$

$$B^{-}_{i,j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \quad r^{-}_{i,j} = 0$$

Algorithm 1: Bias Slow approximation of the non-existing solution value on a grid point based on existing solution values in its neighborhood. The notation is used for $u^{\pm}_{i,j} = B^{\pm}_{i,j} : \mathbf{U}_{i,j} + r^{\pm}_{i,j}$.

3.6. Approach II. Finite discretization method fused with neural extrapolation

Our proposed hybrid PDE solver is based on the observation that the neural network models for negative and positive domains can be explicitly used as interpolation and extrapolation functions within the finite discretization schemes.

The basic idea of the finite discretization method of section 3.5 is to impose the jump conditions on the already-available grid points in the computational cells that are crossed by the interface, instead of aspiring to impose the jump conditions exactly on the interface itself. This is achieved by a Taylor expansion to project the interfacial jump conditions onto adjacent grid points. Using the neural network models for solutions, we are able to naturally evaluate extrapolations of the solution functions in a banded region around the interface as illustrated in figure 2.

Starting from the jump conditions, for points on the interface, $\mathbf{x} \in \Gamma$, we have

$$u^+ - u^- = \alpha$$
$$\mu^+ \partial_n u^+ - \mu^- \partial_n u^- = \beta$$

and after Taylor expansion in the normal direction we obtain on the adjacent grid points (i, j)

$$u^+_{i,j} - u^-_{i,j} = [u]_{\mathbf{r}^{pr}_{i,j}} + \delta_{i,j}(\partial_{\mathbf{n}} u^+(\mathbf{r}^{pr}_{i,j}) - \partial_{\mathbf{n}} u^-(\mathbf{r}^{pr}_{i,j})) \tag{10}$$

which explicitly incorporates the jump condition in the solutions. To incorporate the jump condition in fluxes we can rewrite either of the normal gradients in terms of the other

$$\partial_{\mathbf{n}} u^+(\mathbf{r}^{pr}_{i,j}) = \frac{\mu^-}{\mu^+} \partial_{\mathbf{n}} u^-(\mathbf{r}^{pr}_{i,j}) + \frac{\beta}{\mu^+}$$
$$\partial_{\mathbf{n}} u^-(\mathbf{r}^{pr}_{i,j}) = \frac{\mu^+}{\mu^-} \partial_{\mathbf{n}} u^+(\mathbf{r}^{pr}_{i,j}) - \frac{\beta}{\mu^-}$$

which leads to two relationships among predictions of the two neural networks at each grid point in the banded extrapolation region

$$u^+_{i,j} - u^-_{i,j} = \alpha(\mathbf{r}^{pr}_{i,j}) + \delta_{i,j}\left((\frac{\mu^-}{\mu^+} - 1)\partial_{\mathbf{n}} u^-(\mathbf{r}^{pr}_{i,j}) + \frac{\beta(\mathbf{r}^{pr}_{i,j})}{\mu^+} \right) \tag{11}$$

$$u^+_{i,j} - u^-_{i,j} = \alpha(\mathbf{r}^{pr}_{i,j}) + \delta_{i,j}\left((1 - \frac{\mu^+}{\mu^-})\partial_{\mathbf{n}} u^+(\mathbf{r}^{pr}_{i,j}) + \frac{\beta(\mathbf{r}^{pr}_{i,j})}{\mu^-} \right) \tag{12}$$

Note that we are representing solution functions, $\hat{u}^{\pm}(\mathbf{r})$, with neural networks where computing the normal derivatives is trivial using automatic differentiation of the network along the normal directions. In contrast to finite discretization methods, solutions at off-grid points is readily available by simply evaluating the neural network function at any desired points. Note that we can compute the projected location on the interface starting from each grid point (i, j) using the level-set function:

$$\mathbf{r}^{proj}_{ij} = \mathbf{r}_{ij} - \delta_{i,j}\mathbf{n}_{i,j}$$

In the second approach, the loss function remains as before, except the unknown $u^{\pm}$ values are derived using equations 11–12, instead of computing a regression-based extrapolation function based on the points in the neighborhood of interface cells:

$$\mathcal{L} = \Bigg|\Bigg| \sum_{s=-,+} k^s_{i,j} u^s_{i,j} |\mathcal{V}^s_{i,j}| - \sum_{s=-,+} \Bigg(\mu^s_{i-\frac{1}{2},j} A^s_{i-\frac{1}{2},j} \frac{u^s_{i-1,j} - u^s_{i,j}}{\Delta x} + \mu^s_{i+\frac{1}{2},j} A^s_{i+\frac{1}{2},j} \frac{u^s_{i+1,j} - u^s_{i,j}}{\Delta x} +$$
$$\mu^s_{i,j-\frac{1}{2}} A^s_{i,j-\frac{1}{2}} \frac{u^s_{i,j-1} - u^s_{i,j}}{\Delta y} + \mu^s_{i,j+\frac{1}{2}} A^s_{i,j+\frac{1}{2}} \frac{u^s_{i,j+1} - u^s_{i,j}}{\Delta y} \Bigg) - \sum_{s=-,+} f^s_{i,j} |\mathcal{V}^s_{i,j}| - \int_{\Gamma \cap \mathcal{V}_{i,j}} \beta d\Gamma \Bigg|\Bigg|^2_2$$

There is a major downside with this approach that during training the automatic differentiation has to be applied on the network one more time and therefore we need to compute second-order derivatives of the network during training. This slows down convergence, and the time-to-solution increases with square of depth of the neural network while in the regression based cost grows linearly in the network depth by restricting to only first order automatic differentiation.

3.7. Optimization scheme

Here we describe the optimization protocols for training neural models in `JAX-DIPS`.

3.7.1. Preconditioners are ideal network regularizers

Finite discretization methods lead to solving a linear algebraic system with gaurantees on convergence and accuracies. The geometric irregularities and fine-grain details of the system around interfaces often lead to bad condition number for the linear system, which can be remedied by applying preconditioners. Intuitively, condition number is caused by a separation of scales for geometric lengthscales or material properties that underly the solution patterns. One of the strengths of the presented approach is to readily enable usage of preconditioners for training neural network surrogate models.

Preconditioners are a powerful technique to accelerate convergence of tradional numerical linear algebraic solvers. Given a poorly conditioned linear system $Ax = b$ one can obtain an equivalent system $\hat{A}\hat{x} = \hat{b}$ with accelerated convergence rate when using iterative gradient based methods. For the conjugate gradient method convergence iteration is proportional to $\sqrt{\kappa(A)}$ where $\kappa(A)$ is the condition number of matrix A. Preconditioning is achieved by mapping the linear system with a nonsingular matrix M into a new space $M^{-1}Ax = M^{-1}b$ where $M^{-1}A$ has more regular spread of eigenvalues, hence a better condition number. The precondition matrix M should approximate A^{-1} such that $|I - M^{-1}A| < 1$. The simplest choice is the Jacobi prconditioner which amounts to using the diagonal part of A as the preconditioner, $M = diag(A)$. Note that the diagonal term is locally available at each point and it is straightforward to parallelize.

In this work we use the Jacobi pre-conditioner. Basically, every element of the left-hand-side (Au) and right-hand-side (b) vectors are divided by the coefficient of the diagonal term of the matrix given by:

$$a_{ii} = \sum_{s=-,+} \left(k^s_{i,i} |\mathcal{V}^s_{i,i}| + (\mu^s_{i-\frac{1}{2},i} A^s_{i-\frac{1}{2},i} + \mu^s_{i+\frac{1}{2},i} A^s_{i+\frac{1}{2},i})/\Delta x + (\mu^s_{i,i-\frac{1}{2}} A^s_{i,i-\frac{1}{2}} + \mu^s_{i,i+\frac{1}{2}} A^s_{i,i+\frac{1}{2}})/\Delta y \right)$$

Note that for memory efficiency we never explicitly compute the matrix, instead we compute the effect of matrix product of Au.

3.7.2. Learning rate scheduling

First order methods are slow but cheap; second order methods are fast but expensive. In `JAX-DIPS` we primarily utilize first order optimization methods such as Adam [32]. Second order methods such as Newton or BFGS certainly offer convergence in less iterations but require much more memory. Traditionally used GMRES or Conjugate Gradient methods for sparse linear systems are somewhere between first order and second order optimization methods that are based on building basis vectors by computing gradients that are conjugate to each other $\mathbf{p}_j^T A \mathbf{p}_i = 0$ and will converge to the solution in at most n steps; *i.e.*, at most the solution vector is spanned in the full basis.

We found that starting from a zero guess for the solution it is important to start from a large learning rate and gradually decay the learning rate in a process of exponential annealing. For this purpose, we use the exponential decay scheduler provided by `Optax` [28] to control the learning rate in the Adam optimizer:

$$r_k = r_0 \alpha^{k/T}$$

where r_k is the learning rate at step k of optimization, $\alpha < 1$ is the decay rate, and T is the decay count-scale. By default, we set $T = 100$, $\alpha = 0.975$, starting from an initial value of $r_0 = 10^{-2}$ and clip gradients by maximum global gradient norm (to a value 1) [55] before applying the Adam updates in each step. We note a larger decay rate, *e.g.* $\alpha = 0.98$, leads to small oscillations after 10000 steps and although similar levels of accuracy can be achieved at much less iterations, here we report results with the more robust decay rate.

3.7.3. Domain switching optimization scheme

The linear system suffers from worse condition number in the domain with more variability in diffusion coefficient, or where diffusion coefficient is larger; *i.e.,* the fast region. This leads to regionally unbalanced solution error where the overall error is systematically lopsided by the faster diffusion region.

We found this problem can be drastically improved by interleaving region-specific optimization epochs in the training pipeline, where only one of the networks is updated based on the loss computed in its region. See algorithm 2 for details of the algorithm.

procedure DOMAIN SWITCHING OPTIMIZATION
  for *epoch* in $0 \cdots$ N **do**
    $region = Region(epoch)$
    if $region > 0$ **then**
      if $\mu^- > \mu^+$ **then**
        optimize u^-_{NN} in Ω^- given fixed u^+_{NN}
      else
        optimize u^+_{NN} in Ω^+ given fixed u^-_{NN}

    if $region == 0$ **then**
      optimize both networks in $\Omega^- \cup \Omega^+$

    if $region < 0$ **then**
      if $\mu^- < \mu^+$ **then**
        optimize u^-_{NN} in Ω^- given fixed u^+_{NN}
      else
        optimize u^+_{NN} in Ω^+ given fixed u^-_{NN}

procedure REGION(epoch)
  if *mode* == whole region $\rightarrow$ fast region **then**
    $region = epoch \ \% \ \tau$
  if *mode* == fast region $\rightarrow$ whole region $\rightarrow$ slow region **then**
    $region = \tau//2 - epoch \ \% \ \tau$

Algorithm 2: Domain switching method. Switching interval is τ.

3.7.4. Multi-GPU parallelization with model parallel training

The NBM is embarrasingly parallel and residual evaluation at each point is independent from other points. Therefore, multi-GPU parallelization does not involve inter-GPU communication for evaluating the residuals per point. We partition the training points and distribute them along with copies of neural network parameters among multiple GPUs to compute gradient updates per batch. Then we aggregate these updates by averaging the values on different GPUs. The updates are then broadcasted and model parameters are updated on each device.

The ability to batch over grid points is one of the key enabling factors for reaching higher resolutions and higher dimensions. With NBM it is straightforward to scale finite discretization methods on GPU clusters. It is extremely important to randomly shuffle and redistribute the points among the batches at the beginning of each epoch, we found without shuffling the accuracy deteriorates and increasing resolution does not yield better solutions; see figure 4.

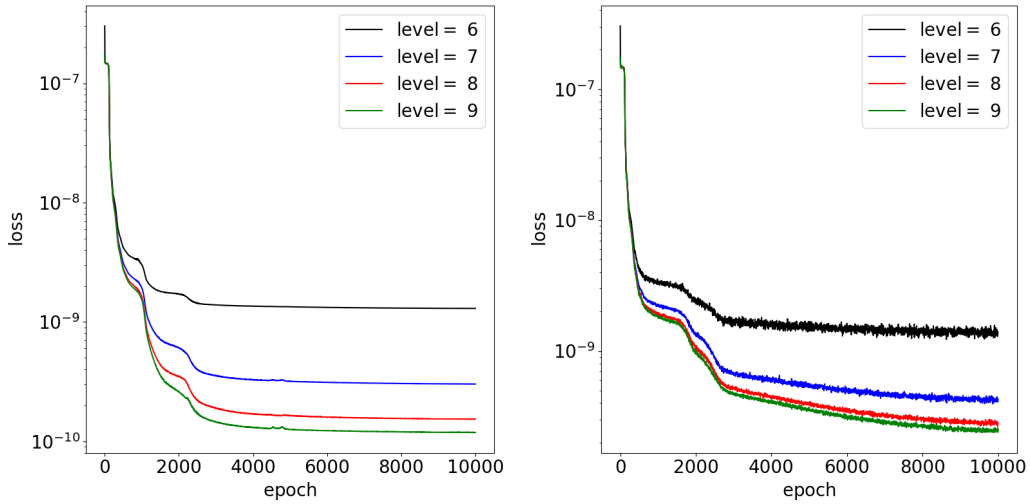


Figure 4: Effect of shuffling on the star example of 4.2; (left) without shuffling and (right) with shuffling. Shuffling has to be applied on the batches when training on batched data points.

4. Numerical Results

We consider examples for solution to elliptic problems of the form

$$\begin{aligned} k^{\pm}u^{\pm} - \nabla\cdot(\mu^{\pm}\nabla u^{\pm}) &= f^{\pm}, && \mathbf{x}\in\Omega^{\pm}\\ [u] &= \alpha, && \mathbf{x}\in\Gamma\\ [\mu\partial_{\mathbf{n}}u] &= \beta, && \mathbf{x}\in\Gamma \end{aligned}$$

Using different features of `JAX-DIPS` one can compose solvers with different training configurations; *i.e.*, single/multi-resolution, single/multi-batch, and single/multi-GPU, and domain alternating training. Moreover, the neural extrapolation method discussed in section 3.6 provides an alternative solver. Below we implement and compare numerical accuracy and performance of these strategies.

4.1. Accuracy on spherical interface: single-resolution, single batch, single GPU

We use a single uniform grid and train on all the points in a single batch. We consider a sphere $\phi(\mathbf{x}) = \sqrt{x^2+y^2+z^2} - 0.5$ centered in a box $\Omega : [-1,1]^3$ with the exact solution

$$\begin{aligned} u^{-}(x,y,z) &= e^{z}, && \phi(\mathbf{x}) < 0\\ u^{+}(x,y,z) &= \cos(x)\sin(y), && \phi(\mathbf{x}) \geq 0 \end{aligned}$$

and variable diffusion coefficients

$$\begin{aligned} \mu^{-}(x,y,z) &= y^2\ln(x+2)+4 && \phi(\mathbf{x}) < 0\\ \mu^{+}(x,y,z) &= e^{-z} && \phi(\mathbf{x}) \geq 0 \end{aligned}$$

that imply variable source terms

$$\begin{aligned} f^{-}(x,y,z) &= -[y^2\ln(x+2)+4]e^{z} && \phi(\mathbf{x}) < 0\\ f^{+}(x,y,z) &= 2\cos(x)\sin(y)e^{-z} && \phi(\mathbf{x}) \geq 0 \end{aligned}$$

The network has 5 hidden layers with 10 neurons in each layer using sine activation functions. Table 1 reports convergence results for the solution in the L^{∞}-norm and root-mean-squared-error of the solution. Order of convergence, denoted by p, is computed by doubling the number of grid

points in every dimension and measuring the L^∞ error of solution and its gradient over all the grid points in the domain:

$$\frac{\text{err}(2h)}{\text{err}(h)} = 2^p \rightarrow p = \log_2\left(\frac{\text{err}(2h)}{\text{err}(h)}\right) \qquad\qquad h = \min(h_x, h_y, h_z)$$

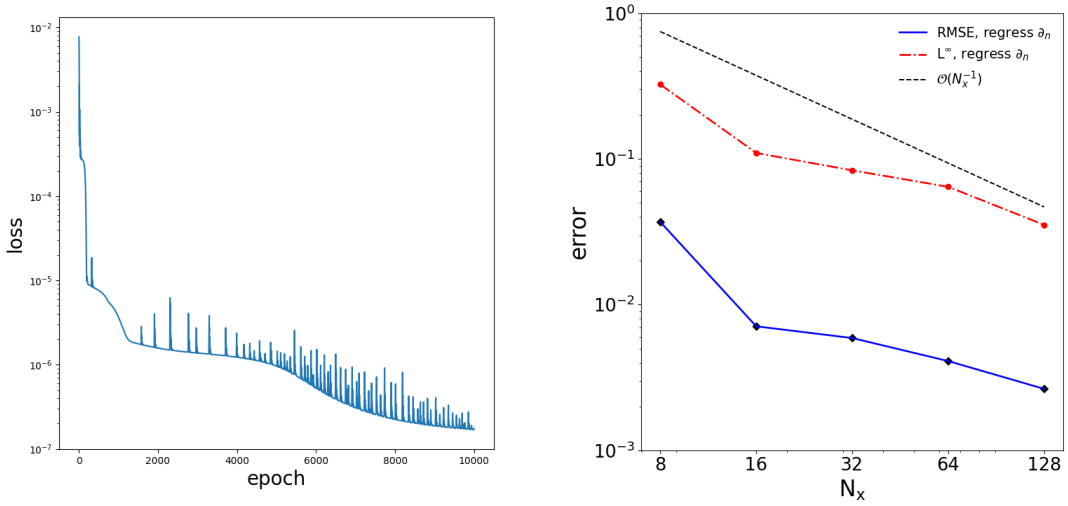


Figure 5: Loss evolution with epochs for the sphere of $16 \times 16 \times 16$ grid (left) and different accuracy measures, RMSE and L^∞, at 5 different resolutions (right).

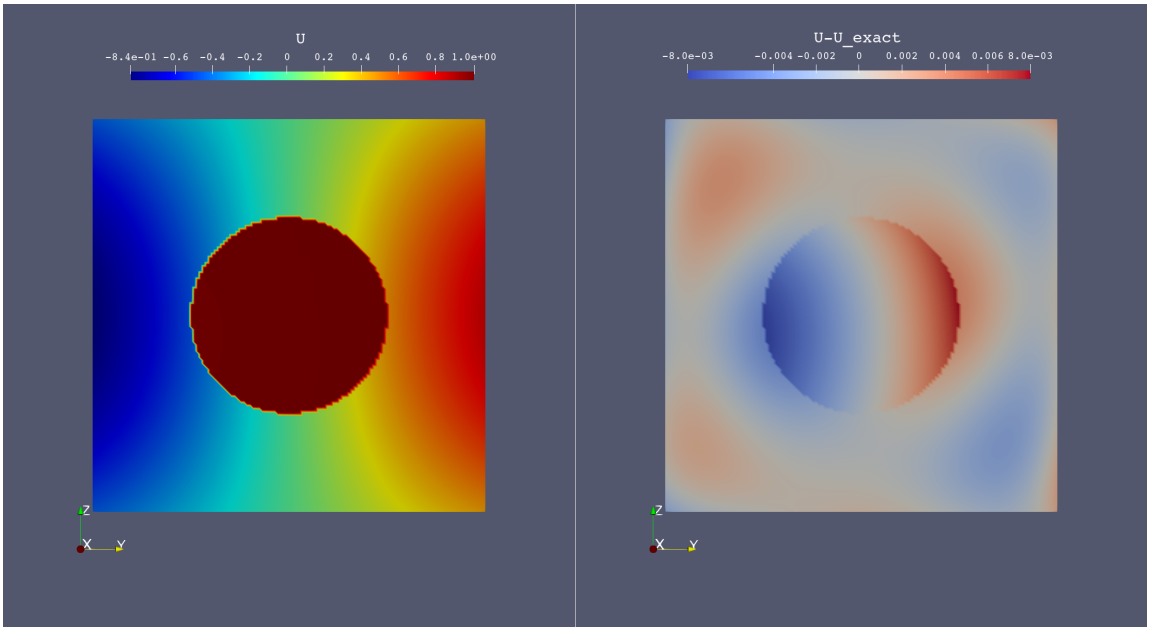
Table 1: Convergence and timings for the sphere example averaged over $10,000$ epochs. Timings include the initial compilation time. Measurements are on a single NVIDIA A6000 GPU. The regression-based method has 5 hidden layers with 10 neurons each, overall 928 trainable parameters.

	RMSE		L^∞		GPU Statistics	
$N_{x,y,z}$	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
2^3	3.7×10^{-2}	-	3.25×10^{-1}	-	0.0306	1.05
2^4	7.1×10^{-3}	2.38	1.10×10^{-1}	1.56	0.056	1.72
2^5	5.9×10^{-3}	0.27	8.36×10^{-2}	0.4	0.053	2.15
2^6	4.1×10^{-3}	0.53	6.44×10^{-2}	0.38	0.287	5.57
2^7	2.64×10^{-3}	0.64	3.53×10^{-2}	0.87	2.125	32.1

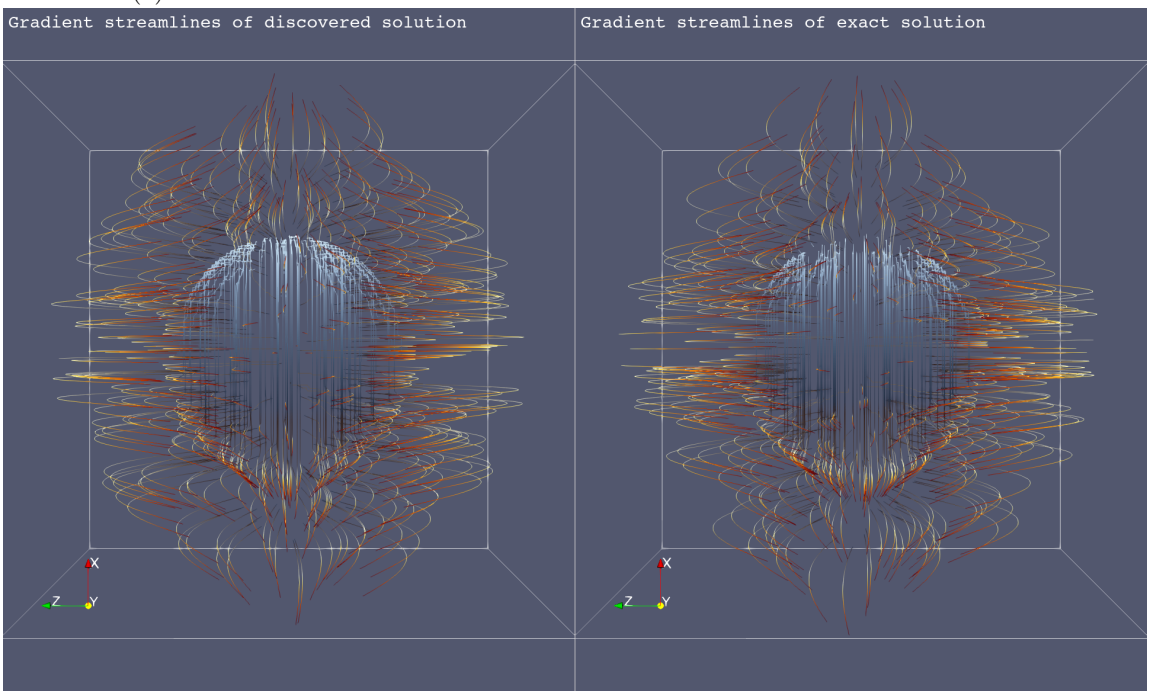
4.2. Accuracy on star interface: single GPU, domain switching, neural extrapolation, and batching

We use a pair of fully connected feedforward neural networks, each composed of 1 hidden layer and 100 neurons with `sine` activation function, followed by an output layer with 1 linear neuron. There are a total of $1,002$ trainable parameters in the model. We consider a star-shaped interface with inner and outer radii $r_i = 0.151$ and $r_e = 0.911$ that is immersed in a box $\Omega : [-1, 1]^3$ described by the level-set function

$$\phi(\mathbf{x}) = \sqrt{x^2 + y^2 + z^2} - r_0\left(1 + \left(\frac{x^2 + y^2}{x^2 + y^2 + z^2}\right)^2 \sum_{k=1}^{3} \beta_k \cos\left(n_k\left(\arctan\left(\frac{y}{x}\right) - \theta_k\right)\right)\right)$$



(a) Illustration of numerical solution and absolute error on a cross section of the domain.



(b) Streamlines of solution gradient for (left) the surrogate neural model colored by model solution value, (right) exact streamlines colored by exact solution values.

Figure 6: The neural network surrogate model trained on a 128^3 grid using a single NVIDIA A6000 GPU.

with the parameters

$$r_0 = 0.483, \qquad \begin{pmatrix} n_1 \\ \beta_1 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} 3 \\ 0.1 \\ 0.5 \end{pmatrix}, \begin{pmatrix} n_2 \\ \beta_2 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 4 \\ -0.1 \\ 1.8 \end{pmatrix}, \qquad \begin{pmatrix} n_3 \\ \beta_3 \\ \theta_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 0.15 \\ 0 \end{pmatrix}$$

Considering an exact solution

$$u^-(x,y,z) = \sin(2x)\cos(2y)e^z, \qquad \phi(\mathbf{x}) < 0$$
$$u^+(x,y,z) = \left[16\left(\frac{y-x}{3}\right)^5 - 20\left(\frac{y-x}{3}\right)^3 + 5\left(\frac{y-x}{3}\right)\right]\ln(x+y+3)\cos(z), \qquad \phi(\mathbf{x}) \geq 0$$

and the diffusion coefficient

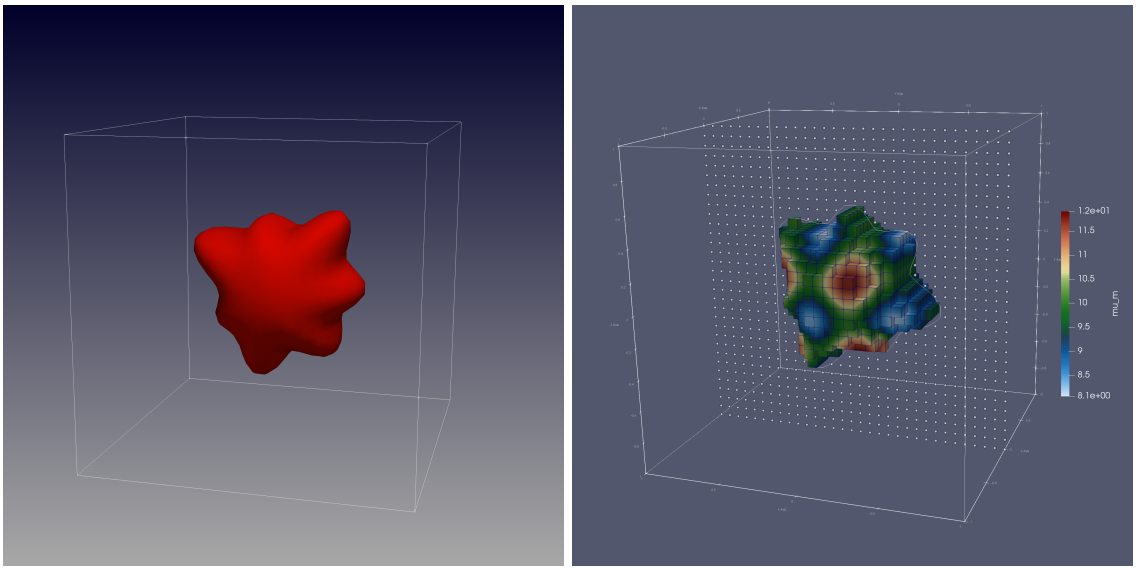
$$\mu^-(x,y,z) = 10\Big[1 + 0.2\cos(2\pi(x+y))\sin(2\pi(x-y))\cos(z)\Big] \qquad \phi(\mathbf{x}) < 0$$
$$\mu^+(x,y,z) = 1 \qquad \phi(\mathbf{x}) \geq 0$$

Table 2: Convergence in solution of the star geometry using the single-resolution regression-based solver with domain switching. We report L^∞-norm error as well as root-mean-squared-error (RMSE) of the solution field evaluated everywhere in the domain. Timings are averaged over $10,000$ epochs in each case and include the initial compilation time for jaxpressions. The neural network pair have 1 hidden layer each with 100 neurons, overall $1,002$ trainable parameteres. Domain switching scheme follows the whole region → fast region → fast region sequence.

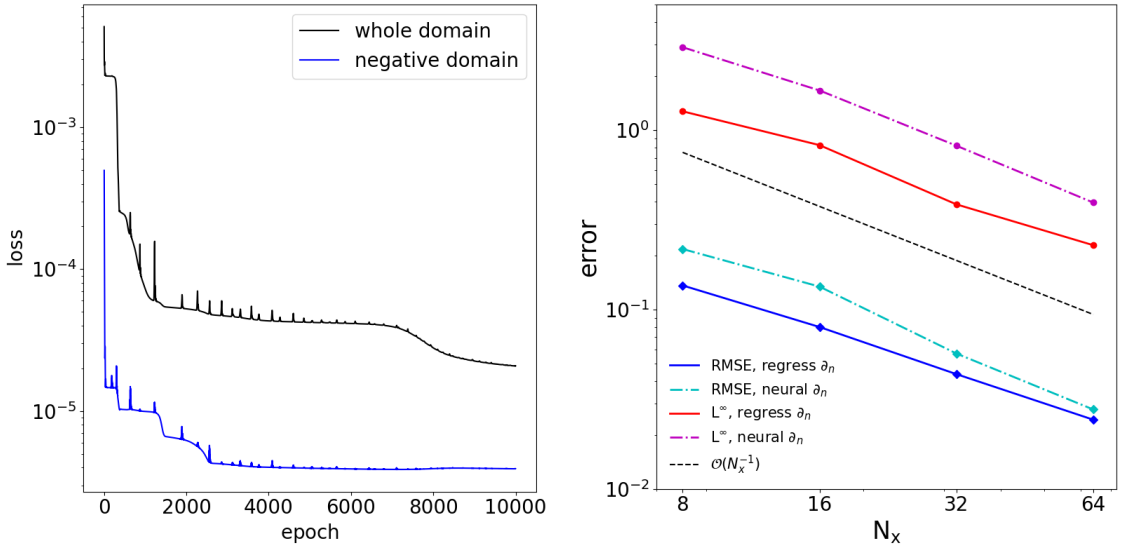
	RMSE		L^∞		GPU Statistics	
$N_{x,y,z}$	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
regress ∂_n						
2^3	1.36×10^{-1}	-	1.27	-	0.019	0.98
2^4	7.98×10^{-2}	0.77	8.23×10^{-1}	0.63	0.022	1.01
2^5	4.36×10^{-2}	0.87	3.85×10^{-1}	1.10	0.032	1.30
2^6	2.43×10^{-2}	0.84	2.28×10^{-1}	0.76	0.200	3.7
neural ∂_n						
2^3	2.17×10^{-1}	-	2.89	-	0.0259	0.93
2^4	1.34×10^{-1}	0.70	1.66	0.80	0.0408	1.19
2^5	5.68×10^{-2}	1.24	8.17×10^{-1}	1.02	0.0712	2.96
2^6	2.77×10^{-2}	1.03	3.94×10^{-1}	1.05	0.334	13.6

Table 3: Convergence in solution of the star geometry using the multi-resolution regression-based solver with batching. We use a multi-resolution training protocol that refines to 4 levels at each collocation point. Batch size is the minimum of $64 \times 64 \times 32$ and number of collocation points, which ensures memory saturation at 30 GB.

regress ∂_n	RMSE		L^∞		GPU Statistics	
$N_{x,y,z}$	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
2^3	1.05×10^{-1}	-	1.29	-	0.0225	1.27
2^4	5.52×10^{-2}	0.93	6.22×10^{-1}	1.05	0.0411	1.27
2^5	2.44×10^{-2}	1.18	2.66×10^{-1}	1.23	0.1814	8.3
2^6	2.33×10^{-2}	0.07	2.24×10^{-1}	0.25	1.889	29.6
2^7	8.62×10^{-2}	−1.88	3.80×10^{-1}	−0.76	9.649	29.7



(a) Illustration of three dimensional interface used (left), and $\mu^{\pm}$ on the $32 \times 32 \times 32$ grid (right).



(b) Loss evolution with epochs for the star of $64 \times 64 \times 64$ grid using domain switching training (left), and decrease in error by increasing resolutions (right).

Figure 7: The neural network model trained with different configurations and resolutions.

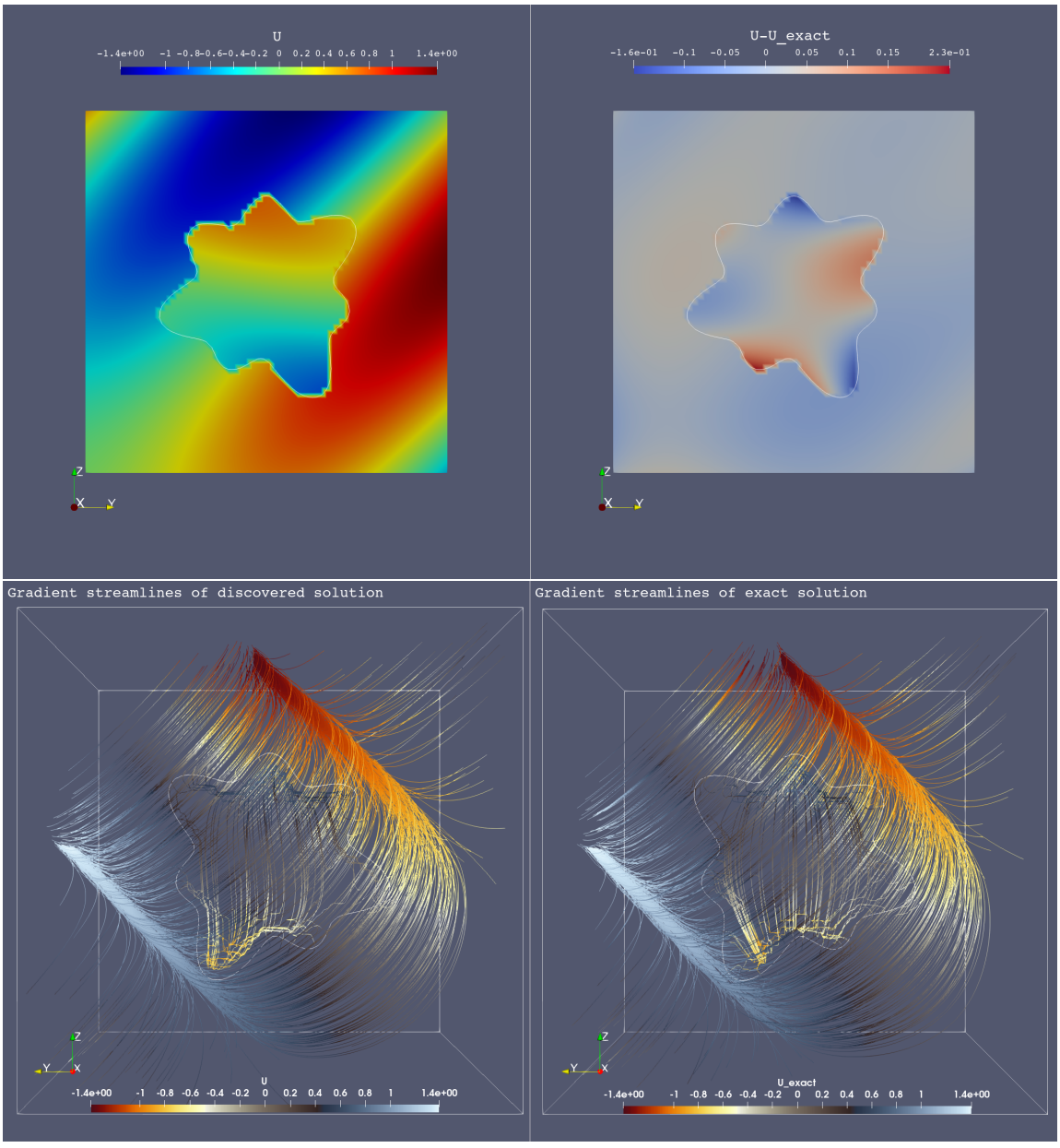


Figure 8: Illustration of exact and numerical solutions (top row) and gradient streamlines (bottom row) on a $64 \times 64 \times 64$ grid.

4.3. Time complexity and parallel scaling on GPU clusters

We adopt the problem setup presented in 4.2, however with a considerably more challenging geometry of the Dragon presented in [18]. In this case we used the signed-distance function produced by SDFGen, and initiated an interpolant based on its values.

The results are shown in figure 9, with a L^∞-error of 0.5 and RMSE of 0.06 after 1000 epochs on a base resolution of 64^3 and implicitly refined onto multi-resolutions $128^3, 256^3, 512^3$. The neural network pair have only 1 hidden layer with 100 sine-activated neurons, although investigating more complex networks (transformers, symmetry preserving, *etc.*) would likely improve accuracy.

In table 4 we report scaling results on NVIDIA A100 GPUs at four base resolutions with three levels of implicit refinement. We used a batchsize of $32 \times 32 \times 16$ in all cases. At fixed number of GPUs, training time scales linearly (*i.e.*, optimal scaling) with the number of grid points. At a fixed resolution, increasing the number of GPUs accelerates training roughly with epoch time $\sim 1/\sqrt{\#\text{ GPUs}}$, although the advantage is more effective at higher resolutions. Compile time increases with resolution and decreases with number of GPUs. A maximum grid size of 1024^3 at multi-resolutions 1024^3, $2048^3, 4096^3$, 8192^3 was simulated on one NVIDIA DGX with 8 A100 GPUs. The results are shown in figure 9.

Table 4: Scaling test. Time per epoch (sec) and JAX compile time for different configurations.

base resolution:	64^3		128^3		256^3		512^3	
A100 GPUs	epoch	compile	epoch	compile	epoch	compile	epoch	compile
1	0.908	9.027	6.960	9.288	55.287	12.164	438.45	49.020
2	0.657	7.575	5.893	7.823	47.360	10.045	378.98	39.815
4	0.405	7.480	3.629	7.863	28.261	9.129	226.73	27.405
8	0.384	7.983	3.340	7.901	26.799	9.154	204.88	20.632

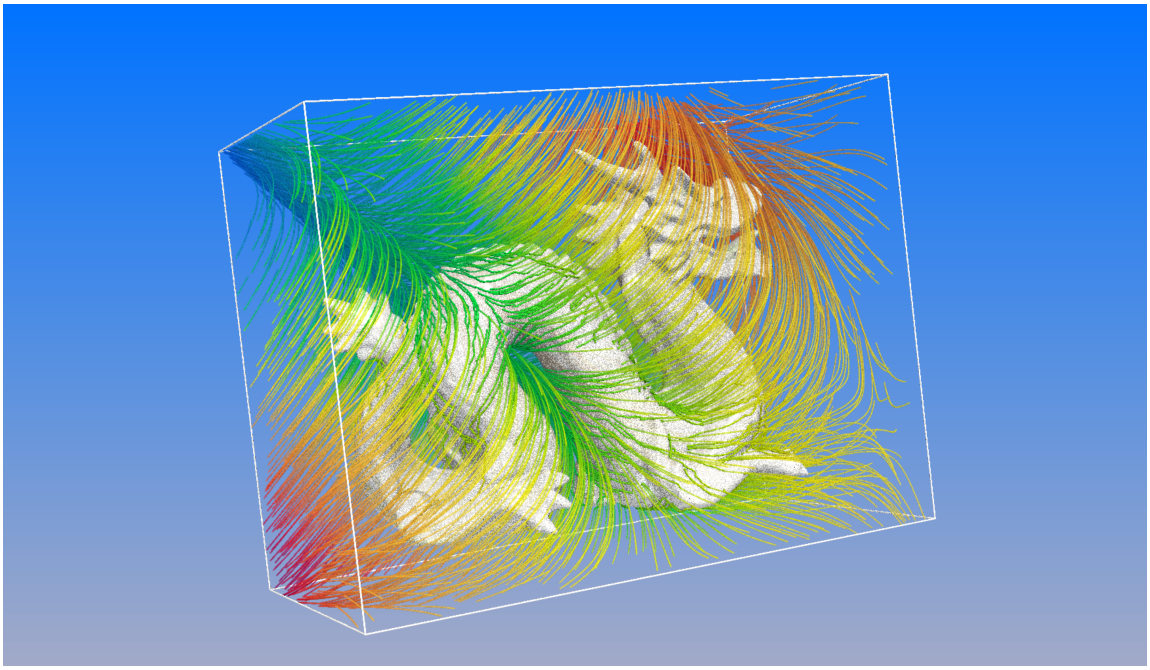
5. Conclusion

We developed a differentiable GPU-based framework for solving partial differential equations with jump conditions across irregular interfaces in three spatial dimensions. Solutions in each domain are represented by a simple multi-layer perceptron (MLP) and Cartesian grid points of the underlying numerical discretization scheme are treated as collocation points for optimizing the unknown parameters of the MLPs.

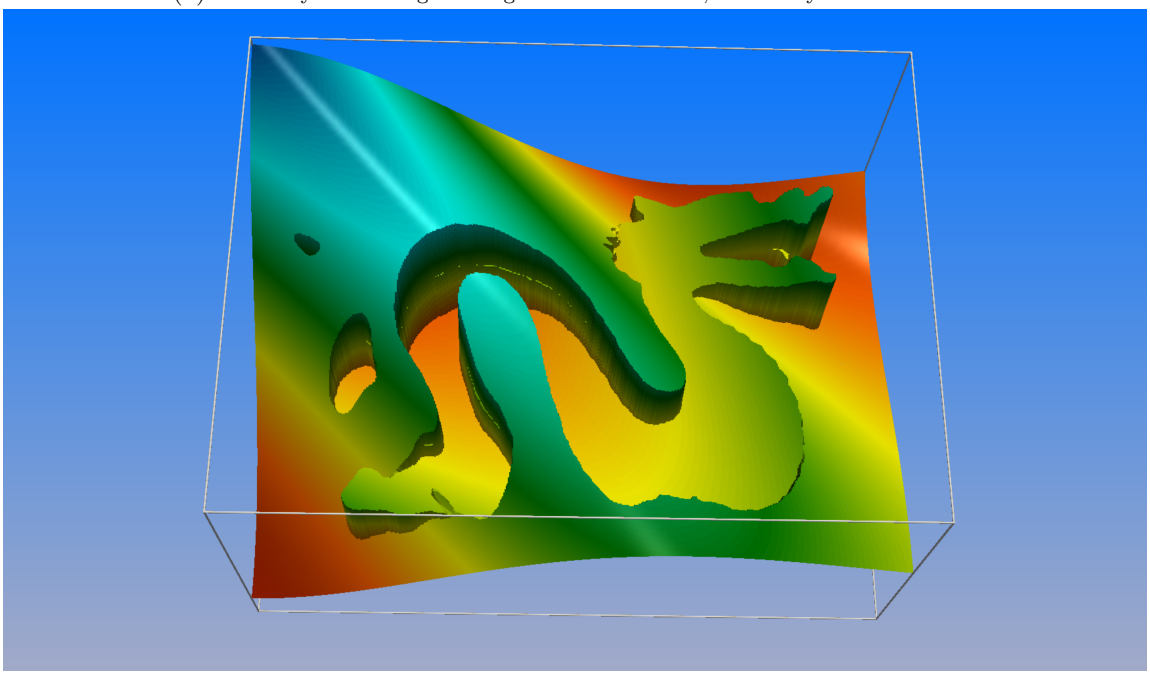
There are many improvements for `JAX-DIPS` that we will pursue for future development:

- More sophisticated neural architectures can be considered in `JAX-DIPS` by adding to the model class of the library. We only considered MLPs, however in recent years there have been a plethora of deep neural network models that have shown great promise such as transformers, graph neural networks, *etc.* An important class are symmetry preserving neural networks.
- Extension to adaptive grids with enhanced resolutions closer to the interfaces while coarsening the grid cells in the bulk.

Acknowledgement



(a) Geometry of the dragon and gradient streamlines, colored by solution values.



(b) Jump in solution and its gradient.

Figure 9: The NBM approach enables a 1024^3 effective resolution on a single NVIDIA A6000 GPU.

References

[1] L. Adams and Z. Li. The immersed interface/multigrid methods for interface problems. *SIAM Journal on Scientific Computing*, 24(2):463–479, 2002.

[2] D. Andrienko. Introduction to liquid crystals. *Journal of Molecular Liquids*, 267:520–541, 2018. Special Issue Dedicated to the Memory of Professor Y. Reznikov.

[3] I. Babuška. The finite element method for elliptic equations with discontinuous coefficients. *Computing*, 5(3):207–213, 1970.

[4] T. Belytschko, N. Moës, S. Usui, and C. Parimi. Arbitrary discontinuities in finite elements. *International Journal for Numerical Methods in Engineering*, 50(4):993–1013, 2001.

[5] J. Berg and K. Nyström. Neural network augmented inverse problems for pdes. *arXiv preprint arXiv:1712.09685*, 2017.

[6] D. A. Bezgin, A. B. Buhendwa, and N. A. Adams. Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *arXiv preprint arXiv:2203.13760*, 2022.

[7] K. Bhattacharya, B. Hosseini, N. B. Kovachki, and A. M. Stuart. Model reduction and neural networks for parametric pdes. *arXiv preprint arXiv:2005.03180*, 2020.

[8] D. Bochkov and F. Gibou. Solving elliptic interface problems with jump conditions on cartesian grids. *Journal of Computational Physics*, 407:109269, 2020.

[9] D. Bochkov and F. Gibou. Solving elliptic interface problems with jump conditions on cartesian grids. *Journal of Computational Physics*, 407:109269, 2020.

[10] D. Bochkov and F. Gibou. A non-parametric shape optimization approach for solving inverse problems in directed self-assembly of block copolymers. *arXiv preprint arXiv:2112.09615*, 2021.

[11] D. Bochkov, T. Pollock, and F. Gibou. Sharp-interface simulations of multicomponent alloy solidification. *arXiv preprint arXiv:2112.08650*, 2021.

[12] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[13] J. H. Bramble and J. T. King. A finite element method for interface problems in domains with smooth boundaries and interfaces. *Advances in Computational Mathematics*, 6(1):109–138, 1996.

[14] T. Chen and H. Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[15] L. O. Chua and L. Yang. Cellular neural networks: Applications. *IEEE Transactions on circuits and systems*, 35(10):1273–1290, 1988.

[16] L. O. Chua and L. Yang. Cellular neural networks: Theory. *IEEE Transactions on circuits and systems*, 35(10):1257–1272, 1988.

[17] R. Crockett, P. Colella, and D. T. Graves. A cartesian grid embedded boundary method for solving the poisson and heat equations with discontinuous coefficients in three dimensions. *Journal of Computational Physics*, 230(7):2451–2469, 2011.

[18] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996.

[19] N. Dal Santo, S. Deparis, and L. Pegolotti. Data driven approximation of parametrized pdes by reduced basis and neural networks. *Journal of Computational Physics*, 416:109550, 2020.

[20] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31, 2018.

[21] R. E. Ewing, Z. Li, T. Lin, and Y. Lin. The immersed finite volume element methods for the elliptic interface problems. *Mathematics and Computers in Simulation*, 50(1-4):63–76, 1999.

[22] R. P. Fedkiw, T. Aslam, B. Merriman, and S. Osher. A non-oscillatory eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of computational physics*, 152(2):457–492, 1999.

[23] K. Galatsis, K. L. Wang, M. Ozkan, C. S. Ozkan, Y. Huang, J. P. Chang, H. G. Monbouquette, Y. Chen, P. Nealey, and Y. Botros. Patterning and templating for nanoelectronics. *Advanced Materials*, 22(6):769–778, 2010.

[24] D. Gobovic and M. Zaghloul. Design of locally connected cmos neural cells to solve the steady-state heat flow problem. In *Proceedings of 36th Midwest Symposium on Circuits and Systems*, pages 755–757. IEEE, 1993.

[25] Y. Gong, B. Li, and Z. Li. Immersed-interface finite-element methods for elliptic interface problems with nonhomogeneous jump conditions. *SIAM Journal on Numerical Analysis*, 46(1):472–495, 2008.

[26] A. Guittet, M. Lepilliez, S. Tanguy, and F. Gibou. Solving elliptic problems with discontinuities on irregular domains–the voronoi interface method. *Journal of Computational Physics*, 298:747–765, 2015.

[27] R. Hecht-Nielsen. Kolmogorov's mapping neural network existence theorem. In *Proceedings of the international conference on Neural Networks*, volume 3, pages 11–14. IEEE Press New York, NY, USA, 1987.

[28] M. Hessel, D. Budden, F. Viola, M. Rosca, E. Sezener, and T. Hennigan. Optax: composable gradient transformation and optimisation, in jax!, 2020.

[29] P. Holl, V. Koltun, and N. Thuerey. Learning to control pdes with differentiable physics. *arXiv preprint arXiv:2001.07457*, 2020.

[30] P. Holl, V. Koltun, K. Um, and N. Thuerey. phiflow: A differentiable pde solving framework for deep learning via physical simulations. In *NeurIPS Workshop*, volume 2, 2020.

[31] V. Ismailov. A three layer neural network can represent any multivariate function, 2020.

[32] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.

[34] A. N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In *Doklady Akademii Nauk*, volume 114, pages 953–956. Russian Academy of Sciences, 1957.

[35] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[36] H. Lee and I. S. Kang. Neural algorithm for solving differential equations. *Journal of Computational Physics*, 91(1):110–131, 1990.

[37] R. J. LeVeque and Z. Li. The immersed interface method for elliptic equations with discontinuous coefficients and singular sources. *SIAM Journal on Numerical Analysis*, 31(4):1019–1044, 1994.

[38] A. J. Lew and G. C. Buscaglia. A discontinuous-galerkin-based immersed boundary method. *International Journal for Numerical Methods in Engineering*, 76(4):427–454, 2008.

[39] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[40] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*, 2020.

[41] Z. Li, T. Lin, and X. Wu. New cartesian grid methods for interface problems using the finite element formulation. *Numerische Mathematik*, 96(1):61–98, 2003.

[42] X.-D. Liu, R. P. Fedkiw, and M. Kang. A boundary condition capturing method for poisson's equation on irregular domains. *Journal of computational Physics*, 160(1):151–178, 2000.

[43] L. Lu, P. Jin, and G. E. Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.

[44] P. Y. Lu, S. Kim, and M. Soljačić. Extracting interpretable physical parameters from spatiotemporal systems using unsupervised learning. *Physical Review X*, 10(3):031056, 2020.

[45] C. Min and F. Gibou. Geometric integration over irregular domains with application to level-set methods. *Journal of Computational Physics*, 226(2):1432–1443, 2007.

[46] C. Min and F. Gibou. A second order accurate level set method on non-graded adaptive cartesian grids. *Journal of Computational Physics*, 225(1):300–321, 2007.

[47] M. Mirzadeh, M. Theillard, and F. Gibou. A second-order discretization of the nonlinear Poisson-Boltzmann equation over irregular geometries using non-graded adaptive Cartesian grids. *Journal of Computational Physics*, 230(5):2125–2140, Mar. 2011.

[48] P. Mistani, A. Guittet, D. Bochkov, J. Schneider, D. Margetis, C. Ratsch, and F. Gibou. The island dynamics model on parallel quadtree grids. *Journal of Computational Physics*, 361:150–166, 2018.

[49] P. Mistani, A. Guittet, C. Poignard, and F. Gibou. A parallel voronoi-based approach for mesoscale simulations of cell aggregate electropermeabilization. *Journal of Computational Physics*, 380:48–64, 2019.

[50] N. Moës, J. Dolbow, and T. Belytschko. A finite element method for crack growth without remeshing. *International journal for numerical methods in engineering*, 46(1):131–150, 1999.

[51] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations. *Journal of computational physics*, 79(1):12–49, 1988.

[52] P. Oswald and P. Pieranski. *Smectic and columnar liquid crystals: concepts and physical properties illustrated by experiments*. CRC press, 2005.

[53] G. Y. Ouaknin, N. Laachi, K. Delaney, G. H. Fredrickson, and F. Gibou. Level-set strategy for inverse dsa-lithography. *Journal of Computational Physics*, 375:1159–1178, 2018.

[54] S. Pakravan, P. A. Mistani, M. A. Aragon-Calvo, and F. Gibou. Solving inverse-pde problems with physics-aware neural networks. *Journal of Computational Physics*, 440:110414, 2021.

[55] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2012.

[56] M. Raissi, P. Perdikaris, and G. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[57] J. F. Sallee. The middle-cut triangulations of the n-cube. *SIAM Journal on Algebraic Discrete Methods*, 5(3):407–419, 1984.

[58] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

[59] K. A. Sharp and B. Honig. Calculating total electrostatic energies with the nonlinear poisson-boltzmann equation. *Journal of Physical Chemistry*, 94(19):7684–7692, 1990.

[60] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of computational physics*, 77(2):439–471, 1988.

[61] J. Sirignano and K. Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[62] D. A. Sprecher. On the structure of continuous functions of several variables. *Transactions of the American Mathematical Society*, 115:340–355, 1965.

[63] M. Sussman, P. Smereka, and S. Osher. A level set approach for computing solutions to incompressible two-phase flow. *Journal of Computational Physics*, 114(1):146–159, 1994.

[64] M. Theillard, F. Gibou, and T. Pollock. A sharp computational method for the simulation of the solidification of binary alloys. *Journal of scientific computing*, 63(2):330–354, 2015.

[65] K. Um, R. Brand, Y. R. Fei, P. Holl, and N. Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 33:6111–6122, 2020.

[66] D. Xiu and G. E. Karniadakis. A semi-lagrangian high-order method for navier–stokes equations. *Journal of Computational Physics*, 172(2):658–684, 2001.

[67] B. Yu et al. The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1):1–12, 2018.

Appendix A. Solving interface problems with physics-informed neural networks

Physics-informed neural networks (PINNs) are based on minimizing a penalty function on evaluated on a point cloud, where the actual PDE constitutes the residual and automatic differentiation is used to compute the spatial derivaties in the PDE. Each of the two relations 11–12 offers an extra residual term that can be used to penalize the loss function in PINNs

$$\mathcal{L}_{interface} = \left|\left| u^+_{i,j} - u^-_{i,j} - \alpha(\mathbf{r}^{pr}_{i,j}) - \delta_{i,j}\left((\frac{\mu^-}{\mu^+} - 1)\partial_{\mathbf{n}} u^-(\mathbf{r}^{pr}_{i,j}) + \frac{\beta(\mathbf{r}^{pr}_{i,j})}{\mu^+} \right) \right|\right|^2_2 \tag{A.1}$$

or

$$\mathcal{L}_{interface} = \left|\left| u^+_{i,j} - u^-_{i,j} - \alpha(\mathbf{r}^{pr}_{i,j}) - \delta_{i,j}\left((1 - \frac{\mu^+}{\mu^-})\partial_{\mathbf{n}} u^+(\mathbf{r}^{pr}_{i,j}) + \frac{\beta(\mathbf{r}^{pr}_{i,j})}{\mu^-} \right) \right|\right|^2_2 \tag{A.2}$$

Far from interface, the usual procedure for physics-informed neural networks is applicable, namely,

$$\mathcal{L}_{bulk} = \left|\left| k^{\pm}u^{\pm} - \nabla \cdot (\mu^{\pm}\nabla u^{\pm}) - f^{\pm} \right|\right|_2^2$$

$$\mathcal{L}_{boundary} = \left|\left| u(\mathbf{r}_{bc}) - \hat{u}(\mathbf{r}_{bc}) \right|\right|_2^2$$

hence, the overall loss function for this class of problems shall be

$$\mathcal{L} = \mathcal{L}_{bulk} + \mathcal{L}_{boundary} + \mathcal{L}_{interface}$$