

JAX-DIPS: Differentiable Interfacial PDE Solver

Pouria A. Mistani^{*†,a}, Samira Pakravan^{†,b}, Rajesh Ilango^a, Frederic G. Gibou^b

^aNVIDIA Corporation, Santa Clara, CA 95051, USA

^bUniversity of California, Santa Barbara, CA 93106-5070, USA

Abstract

We present an end-to-end differentiable PDE solver supporting both forward and inverse free boundary problems using the level-set method. We implemented this framework using JAX, extending support for CPU/GPU/TPU platforms. Algorithmically, our proposed framework builds on the blended inverse PDE solver architecture (BiPDE) that authors have proposed earlier [11]. JAX-DIPS is an open-source software package published is available at <https://github.com/JAX-DIPS/JAX-DIPS>.

Keywords: level-set method, free boundary problems, inverse problems, jump conditions, differentiable programming

1. Introduction

Differentiable programming ...

DTO vs OTD: we are doing DTO which is Discretize then Optimize paradigm. OTD is the traditional adjoint methods.

JAX-DIPS Poisson problem solver provides the two sides of $Au = b$ which is obtained after discretization of the governing PDE problem over a uniform 3D grid. Two solution schemes for the forwards problem are possible: (1) using usual iterative methods having left-hand-side and right-hand-side of the PDE discretization, and (2) using autodifferentiation across loss function (difference between lhs and rhs in 2-norm) with respect to a given estimate for the solution vector on the underlying grid u_{ijk} . Both approaches focus on minimizing the residual over the grid points

$$\min_{u_{ijk}} \|Au - b\|_2^2$$

The first method attempts to span the residual space in an iterative fashion by *estimating* the gradient of the minimizing function, while the second method offered in JAX-DIPS is directly computing the exact gradient of the minimizing function with respect to the current estimate for solution. Therefore, the advantage of this method is fewer iterations and faster convergence specially for irregular geometries where the condition number of the linear system leads to much difficulties that need to be resolved by complex preconditioning and increased number of iterations.

We designed JAX-DIPS as 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 library for further development of novel numerical algorithms for solving both forward and inverse interfacial PDE problems.

2. Numerical Scheme for Free Boundary Problems

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

^{*}Corresponding author: p.a.mistani@gmail.com

[†]These authors contributed equally to this work

solutions $u^\pm \in \Omega^\pm$ 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 $\Omega = [-L/2, L/2]^3$.

2.1. Interpolation methods

To support finite discretization methods for the free boundary problems (namely solver, level-set method, and geometric integrations) a widely needed 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 implement 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)[9]: (i) trilinear interpolation, and (ii) quadratic non-oscillatory interpolation.

2.1.1. 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 [9],

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

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

2.1.2. 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

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

where second order derivatives 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.

2.2. Level-set method

The level-set method for solving free boundary problems was introduced by Osher & Sethian (1988) [10]. The free boundary 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

$$\frac{\partial \phi}{\partial t} + \mathbf{v} \cdot \nabla \phi = 0 \quad (1)$$

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.

The signed-distance property of the level-set function, *i.e.* $|\nabla \phi| = 1$, deteriorates with timestepping the discretized advection equation. This is resolved by solving the Sussman reinitialization equation [16] in fictitious time (τ) every few iterations of the physical time (t),

$$\frac{\partial \phi}{\partial \tau} + \text{sgn}(\phi_0)(|\nabla \phi| - 1) = 0 \quad (2)$$

Note the asymptotic solution of the Sussman equation is the signed-distance property of the level-set function. Here $\text{sgn}(\phi_0)$ is the sign function evaluated on the level-set function before reinitialization in order to preserve the sign of grid points during the fictitious updates.

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

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

Below we describe numerical methods implemented in JAX-DIPS to solve the level-set equations.

2.2.1. Second order accurate semi-Lagrangian advection scheme

Semi-Lagrangian methods are unconditionally stable for solving advection equations, therefore avoiding the restrictive CFL condition on the timestep size from time t^n to t^{n+1} . The general procedure to solve for $\phi^{n+1}(\mathbf{x}^{n+1})$ is to first evaluate for each grid point, \mathbf{x}^{n+1} , the departure point, \mathbf{x}_d , in the upwind direction along the characteristic curve; then use interpolation methods to recover value of the solution field at the departure point; and finally updating the solution field by setting $\phi^{n+1}(\mathbf{x}^{n+1}) = \phi^n(\mathbf{x}_d)$.

The task of evaluating departure points is treated by the second-order mid-point method [17]

$$\begin{aligned} \hat{\mathbf{x}} &= \mathbf{x}^{n+1} - \frac{\Delta t}{2} \cdot \mathbf{v}^n(\mathbf{x}^{n+1}) \\ \mathbf{x}_d &= \mathbf{x}^{n+1} - \Delta t \cdot \mathbf{v}^{n+\frac{1}{2}}(\hat{\mathbf{x}}) \end{aligned}$$

The first step is trivial, but the second step demands velocity at the mid-timestep $t^{n+\frac{1}{2}}$ which can be evaluated from the velocity field at the previous two timesteps, t^n and t^{n-1} , according to

$$\mathbf{v}^{n+\frac{1}{2}} = \frac{3}{2}\mathbf{v}^n - \frac{1}{2}\mathbf{v}^{n-1}$$

this intermediate velocity field is then fed into a trilinear interpolation scheme to sample the velocities on the intermediate points $\mathbf{v}^{n+\frac{1}{2}}(\hat{\mathbf{x}})$. At this point departure points can be computed, and the advected solution is updated by sampling a nonoscillatory interpolant of the level-set function at timestep t^n over the computed departure points $\phi^n(\mathbf{x}_d)$.

2.2.2. Godunov Hamiltonian for reinitialization

The Sussman equation 2 is generically discretized over its spatial dimensions to obtain

$$\frac{d\phi}{d\tau} = -\text{sgn}(\phi^0) \left[H_G(D_x^+ \phi, D_x^- \phi, D_y^+ \phi, D_y^- \phi, D_z^+ \phi, D_z^- \phi) - 1 \right] \quad (3)$$

where H_G is the so-called Gudonov Hamiltonian defined as

$$H_G(a, b, c, d, e, f) = \sqrt{\sum_{(i,j) \in (a,b),(c,d),(e,f)} \max \left(|\min(s \cdot i, 0)|^2, |\max(s \cdot j, 0)|^2 \right)} \\ s = \text{sgn}(\phi^0)$$

where the one-sided derivatives are computed using second order accurate discretizations in the bulk far from interfaces

$$(D_x^+ \phi)_{i,j,k} = \frac{\phi_{i+1,j,k} - \phi_{i,j,k}}{\Delta x} - \frac{\Delta x}{2} \text{minmod}((D_{xx}\phi)_{i,j,k}, (D_{xx}\phi)_{i+1,j,k}) \\ (D_x^- \phi)_{i,j,k} = \frac{\phi_{i,j,k} - \phi_{i-1,j,k}}{\Delta x} + \frac{\Delta x}{2} \text{minmod}((D_{xx}\phi)_{i,j,k}, (D_{xx}\phi)_{i-1,j,k})$$

where we used $\text{minmod}(a, b) \triangleq \text{median}(a, 0, b)$ slope-limiter [14]; and in the vicinity of interfaces these derivatives take into account distance to the interface using the level-set function. With similar results along y and z axes, the derivative along the positive x -axis is given by

$$(D_x^+ \phi)_{i,j,k} = \frac{0 - \phi_{i,j,k}}{s_I} - \frac{s_I}{2} c_2^+ \\ s_I = \frac{\Delta x}{2} + \begin{cases} -c_0^+/c_1^+ & \text{if } |c_2^+| < \epsilon \\ \left(-c_1^+ - \text{sgn}(\phi_{i,j,k}^0) \sqrt{(c_1^+)^2 - 4c_2^+ c_0^+} \right) / (2c_2^+) & \text{if } |c_2^+| \geq \epsilon \end{cases}$$

where

$$c_2^+ = \text{minmod}((D_{xx}\phi)_{i,j,k}, (D_{xx}\phi)_{i+1,j,k}) \\ c_1^+ = \frac{\phi_{i+1,j,k} - \phi_{i,j,k}}{\Delta x} \\ c_0^+ = \frac{\phi_{i+1,j,k} + \phi_{i,j,k}}{2} - \frac{c_2^+ (\Delta x)^2}{4}$$

And along the negative x -axis we obtain

$$(D_x^- \phi)_{i,j,k} = \frac{\phi_{i,j,k} - 0}{s_I} + \frac{s_I}{2} c_2^- \\ s_I = \frac{\Delta x}{2} + \begin{cases} c_0^-/c_1^- & \text{if } |c_2^-| < \epsilon \\ \left(c_1^- - \text{sgn}(\phi_{i,j,k}^0) \sqrt{(c_1^-)^2 - 4c_2^- c_0^-} \right) / (2c_2^-) & \text{if } |c_2^-| \geq \epsilon \end{cases}$$

where

$$c_2^- = \text{minmod}((D_{xx}\phi)_{i,j,k}, (D_{xx}\phi)_{i-1,j,k}) \\ c_1^- = \frac{\phi_{i,j,k} - \phi_{i-1,j,k}}{\Delta x} \\ c_0^- = \frac{\phi_{i-1,j,k} + \phi_{i,j,k}}{2} - \frac{c_2^- (\Delta x)^2}{4}$$

Having the right-hand-side, the TVD-RK2 method of Shu & Osher [14] is applied to the update the level-set function in fictitious time

$$\begin{aligned}\tilde{\phi}^{n+1} &= \phi^n - \Delta\tau \cdot \text{sgn}(\phi^0)[H_G^n - 1] \\ \tilde{\phi}^{n+2} &= \tilde{\phi}^{n+1} - \Delta\tau \cdot \text{sgn}(\phi^0)[\tilde{H}_G^{n+1} - 1] \\ \phi^{n+1} &= \frac{\phi^n + \tilde{\phi}^{n+2}}{2}\end{aligned}$$

with a timestep chosen by $\Delta\tau = \min(s_I, \Delta x, \Delta y, \Delta z)/3$.

2.3. Integration over 3D surfaces and volumes

We use uniform Cartesian grids. For computational cells that are crossed by the interface, *i.e.* $\mathcal{V}_{i,j,k} \cap \Gamma \neq \emptyset$, we use the geometric integrations proposed by Min & Gibou (2007b) [8]. In this scheme each grid cell, \mathcal{C} , is decomposed into five tetrahedra by the middle-cut triangulation [13] (each cell is rescaled to $[0, 1]^3$) that are described below (also see figure 1 (right) of [8]):

$S_1 \equiv \text{conv}(P_{000}; P_{100}; P_{010}; P_{001})$	x = 0 face, y = 0 face, z = 0 face
$S_2 \equiv \text{conv}(P_{110}; P_{100}; P_{010}; P_{111})$	x = 1 face, y = 1 face, z = 0 face
$S_3 \equiv \text{conv}(P_{101}; P_{100}; P_{111}; P_{001})$	x = 1 face, y = 0 face, z = 1 face
$S_4 \equiv \text{conv}(P_{011}; P_{111}; P_{010}; P_{001})$	x = 0 face, y = 1 face, z = 1 face
$S_5 \equiv \text{conv}(P_{111}; P_{100}; P_{010}; P_{001})$	no face exposure

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, \dots, P_3 are the four vertices of a simplex S , then Γ crosses an edge $P_i P_j$ 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 [8]:
 - $\eta = 0$: 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 [8].
 - $\eta = 2$: 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 [8].
 - $\eta = 3$: 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 [8]:
 - $\eta = 0$: 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 [8]; *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 [8]. Note that there
- $\eta = 3$: the intersection $S \cap \Omega^-$ is characterized by three tetrahedra with 12 vertices according to figure 4 (right) of [8].
- $\eta = 4$: 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 simplices (2 three-vertex simplices for surface area and 3 four-vertex simplices 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, \dots, P_n and the values of the field f at these vertices according to

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

where

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

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

2.4. 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.

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 $N_x \times N_y$ 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: $u_{i,j}^+ - u_{i,j}^- = [u]_{\mathbf{r}_{i,j}^{pr}} + \delta_{i,j}(\partial_{\mathbf{n}} u^+(\mathbf{r}_{i,j}^{pr}) - \partial_{\mathbf{n}} u^-(\mathbf{r}_{i,j}^{pr}))$. The unknown value ($u_{i,j}^-$ or $u_{i,j}^+$) is obtained based on approximation of the normal derivatives (*i.e.* $\partial_{\mathbf{n}} u^\pm(\mathbf{r}_{i,j}^{pr})$) 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 $u_{i,j}^\pm$.

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

$$(\nabla u^\pm)_{i,j} = \mathbf{D}_{i,j}^\pm \begin{bmatrix} u_{i-1,j-1} - u_{i,j}^\pm \\ u_{i,j-1} - u_{i,j}^\pm \\ \vdots \\ u_{i+1,j+1} - u_{i,j}^\pm \end{bmatrix} \quad \mathbf{D}_{i,j}^\pm = (X_{i,j}^T W_{i,j}^\pm X_{i,j})^{-1} (W_{i,j}^\pm X_{i,j})^T$$

and

$$W_{i,j}^\pm = \begin{bmatrix} \omega_{i,j}^\pm(-1, -1) & & & \\ & \omega_{i,j}^\pm(0, -1) & & \\ & & \ddots & \\ & & & \omega_{i,j}^\pm(1, 1) \end{bmatrix} \quad X_{i,j} = \begin{bmatrix} -h_x & -h_y \\ 0 & -h_y \\ h_x & -h_y \\ -h_x & 0 \\ 0 & 0 \\ h_x & 0 \\ -h_x & h_y \\ 0 & h_y \\ h_x & h_y \end{bmatrix}$$

and

$$\omega_{i,j}^\pm(p, q) = \begin{cases} 1 & (p, q) \in N_{i,j}^\pm \\ 0 & \text{else} \end{cases} \quad (4)$$

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

$$\mathbf{D}_{i,j}^\pm = [\quad d_{i,j,-1,-1}^\pm \quad d_{i,j,0,-1}^\pm \quad d_{i,j,1,-1}^\pm \quad d_{i,j,-1,0}^\pm \quad d_{i,j,0,0}^\pm \quad d_{i,j,1,0}^\pm \quad d_{i,j,-1,1}^\pm \quad d_{i,j,0,1}^\pm \quad d_{i,j,1,1}^\pm]$$

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

$$c_{i,j,p,q}^\pm = \mathbf{n}_{i,j}^T d_{i,j,p,q}^\pm$$

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

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

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

$$\begin{aligned} \zeta_{i,j,p,q}^\pm &:= \delta_{i,j} \frac{[\mu]}{\mu^\mp} c_{i,j,p,q}^\pm & \zeta_{i,j}^\pm &:= - \sum_{(p,q) \in N_{i,j}^\pm} \zeta_{i,j,p,q}^\pm \\ \gamma_{i,j,p,q}^\pm &:= \frac{\zeta_{i,j,p,q}^\pm}{1 \pm \zeta_{i,j}^\pm} & \gamma_{i,j}^\pm &:= - \sum_{(p,q) \in N_{i,j}^\pm} \gamma_{i,j,p,q}^\pm \end{aligned}$$

where the set of neighboring grid points are

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

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 $\partial_{\mathbf{n}} u^+(\mathbf{r}_{i,j}^{pr})$:

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j}(1 - \gamma_{i,j}^-) - \sum_{(p,q) \in N_{i,j}^-} \gamma_{i,j,p,q}^- u_{i+p,j+q} - (\alpha + \frac{\delta_{i,j}\beta}{\mu^+})(1 - \gamma_{i,j}^-) & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (5)$$

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

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

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

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

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ (1 + \mathbf{\Gamma}_{i,j}^- : \mathbf{N}_{i,j}^-) u_{i,j} - (\mathbf{\Gamma}_{i,j}^- \odot \mathbf{N}_{i,j}^-) : \mathbf{U}_{i,j} - (\alpha + \delta_{i,j} \frac{\beta}{\mu^+})(1 + \mathbf{\Gamma}_{i,j}^- : \mathbf{N}_{i,j}^-) & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (7)$$

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

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

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j}(1 - \zeta_{i,j}^+) - \sum_{(p,q) \in N_{i,j}^+} \zeta_{i,j,p,q}^+ u_{i+p,j+q} - \alpha - \delta_{i,j} \frac{\beta}{\mu^-} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (9)$$

$$u_{i,j}^+ = \begin{cases} u_{i,j}(1 - \gamma_{i,j}^+) - \sum_{(p,q) \in N_{i,j}^+} \gamma_{i,j,p,q}^+ u_{i+p,j+q} + (\alpha + \delta_{i,j} \frac{\beta}{\mu^-})(1 - \gamma_{i,j}^+) & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (10)$$

in matrix notation we have

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ (1 + \zeta_{i,j}^+ : \mathbf{N}_{i,j}^+) u_{i,j} - (\zeta_{i,j}^+ \odot \mathbf{N}_{i,j}^+) : \mathbf{U}_{i,j} - \alpha - \delta_{i,j} \frac{\beta}{\mu^-} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (11)$$

$$u_{i,j}^+ = \begin{cases} (1 + \mathbf{\Gamma}_{i,j}^+ : \mathbf{N}_{i,j}^+) u_{i,j} - (\mathbf{\Gamma}_{i,j}^+ \odot \mathbf{N}_{i,j}^+) : \mathbf{U}_{i,j} + (\alpha + \delta_{i,j} \frac{\beta}{\mu^-})(1 + \mathbf{\Gamma}_{i,j}^+ : \mathbf{N}_{i,j}^+) & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (12)$$

```

1: procedure BIAS SLOW
2:   if  $\Gamma \cap \mathcal{C}_{i,j} = \emptyset$  then
3:      $B_{i,j}^\pm = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^\pm = 0$ 
4:   else
5:     if  $\mu_{i,j}^- > \mu_{i,j}^+$  then
6:       if  $\phi_{i,j} \geq 0$  then
7:          $B_{i,j}^+ = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^+ = 0$ 
8:          $B_{i,j}^- = \begin{bmatrix} -\gamma_{i,j,-1,1} & -\gamma_{i,j,0,1} & -\gamma_{i,j,1,1} \\ -\gamma_{i,j,-1,0} & 1 - \gamma_{i,j} & -\gamma_{i,j,1,0} \\ -\gamma_{i,j,-1,-1} & -\gamma_{i,j,0,-1} & -\gamma_{i,j,1,-1} \end{bmatrix}$ ;  $r_{i,j}^- = -(\alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{i,j}^+})(1 - \gamma_{i,j}^-)$ 
9:       else
10:         $B_{i,j}^+ = \begin{bmatrix} -\zeta_{i,j,-1,1} & -\zeta_{i,j,0,1} & -\zeta_{i,j,1,1} \\ -\zeta_{i,j,-1,0} & 1 - \zeta_{i,j} & -\zeta_{i,j,1,0} \\ -\zeta_{i,j,-1,-1} & -\zeta_{i,j,0,-1} & -\zeta_{i,j,1,-1} \end{bmatrix}$ ;  $r_{i,j}^+ = \alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{i,j}^+}$ 
11:         $B_{i,j}^- = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^- = 0$ 
12:      else
13:        if  $\phi_{i,j} \geq 0$  then
14:           $B_{i,j}^+ = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^+ = 0$ 
15:           $B_{i,j}^- = \begin{bmatrix} -\zeta_{i,j,-1,1}^+ & -\zeta_{i,j,0,1}^+ & -\zeta_{i,j,1,1}^+ \\ -\zeta_{i,j,-1,0}^+ & 1 - \zeta_{i,j}^+ & -\zeta_{i,j,1,0}^+ \\ -\zeta_{i,j,-1,-1}^+ & -\zeta_{i,j,0,-1}^+ & -\zeta_{i,j,1,-1}^+ \end{bmatrix}$ ;  $r_{i,j}^- = \alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{i,j}^-}$ 
16:        else
17:           $B_{i,j}^+ = \begin{bmatrix} -\gamma_{i,j,-1,1}^+ & -\gamma_{i,j,0,1}^+ & -\gamma_{i,j,1,1}^+ \\ -\gamma_{i,j,-1,0}^+ & 1 - \gamma_{i,j}^+ & -\gamma_{i,j,1,0}^+ \\ -\gamma_{i,j,-1,-1}^+ & -\gamma_{i,j,0,-1}^+ & -\gamma_{i,j,1,-1}^+ \end{bmatrix}$ ;  $r_{i,j}^+ = (\alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{i,j}^-})(1 - \gamma_{i,j}^+)$ 
18:           $B_{i,j}^- = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^- = 0$ 

```

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 $u_{i,j}^\pm = B_{i,j}^\pm : \mathbf{U}_{i,j} + r_{i,j}^\pm$.

2.6. Neural network approximators for the solution

In 1987, Hecht and Nielson [3] applied an improved version of Kolmogorov’s 1957 superposition theorem [7], due to Sprecher [15], 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.*, $f : [0, 1]^n \rightarrow \mathbb{R}^m$. Recently, Ismailov (2022) [5] 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:

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

In each network we implement two hidden layers with 10 neurons per layer using `tanh` 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. There are a total of 322 trainable parameters.

Solution networks are evaluated on grid points while the parameters of these networks are optimized using the loss function.

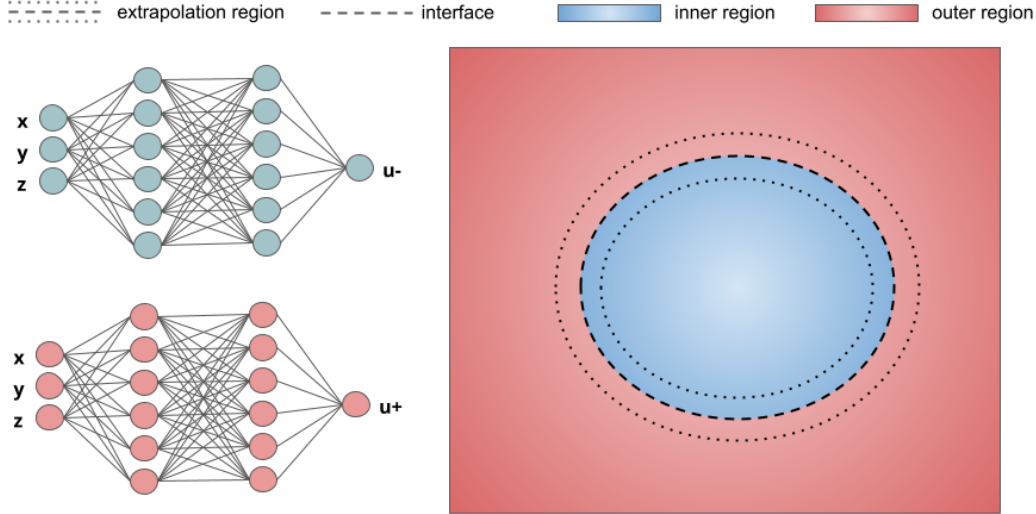


Figure 2: Two neural networks are defined for the two regions of the computational domain.

2.7. Differentiable solution strategy

We define the loss function by the 2-norm of the residual of the discretized partial differential equation with jump conditions derived in section 2.5 that is evaluated on the grid points:

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

JAX-DIPS allows for exact computation of the gradient of the loss function using automatic differentiation, *i.e.* $\nabla_u \mathcal{L}(u)$. This is advantageous over existing approximate iterative methods (with

preconditioning) such as GMRES, Conjugate Gradient, *etc.* Therefore, our strategy is to leverage this capability and use more sophisticated optimizers developed in the deep learning community (*e.g.* Adam [6], RMSProp, *etc.*) to minimize the aforementioned loss function with the desired solution vector u^* of the PDE problem.

Solving an interfacial PDE problem on a $128 \times 128 \times 128$ grid is therefore equivalent to a deep neural network architecture with 2,097,152 trainable parameters. Besides number of trainable parameters, the computational complexity of the algebraic operations in solving a PDE system with provided discretizations closely resembles computational complexity of convolutional operations with kernel sizes equivalent to the stencil size of the discretization method (in this case $3 \times 3 \times 3$).

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, RMSProp, *etc.* Second order methods such as Newton or BFGS are certainly faster convergence 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 build a basis vectors by computing gradients that are conjugate to each other $\mathbf{p}_j^T \mathbf{A} \mathbf{p}_i = 0$ and will converge to the solution in at most n steps; *i.e.*, at most the solution vector is spanned in the full basis.

2.7.1. Gradient based optimization with learning rate annealing

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 [4] to control the learning rate in the Adam [6] 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) [12] 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.

2.7.2. Improving convergence with preconditioning

Preconditioners are a powerful technique to accelerate convergence of traditional 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 . This 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 preconditioner which amounts to using the diagonal part of A as the preconditioner, $M = \text{diag}(A)$.

2.8. Interface induced penalty terms

Our proposed physics-aware 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 2.5 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 extensions 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

$$\begin{aligned} u^+ - u^- &= \alpha \\ \mu^+ \partial_n u^+ - \mu^- \partial_n u^- &= \beta \end{aligned}$$

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

$$u_{i,j}^+ - u_{i,j}^- = [u]_{\mathbf{r}_{i,j}^{pr}} + \delta_{i,j} (\partial_n u^+(\mathbf{r}_{i,j}^{pr}) - \partial_n u^-(\mathbf{r}_{i,j}^{pr})) \quad (13)$$

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

$$\begin{aligned} \partial_n u^+(\mathbf{r}_{i,j}^{pr}) &= \frac{\mu^-}{\mu^+} \partial_n u^-(\mathbf{r}_{i,j}^{pr}) + \frac{\beta}{\mu^+} \\ \partial_n u^-(\mathbf{r}_{i,j}^{pr}) &= \frac{\mu^+}{\mu^-} \partial_n u^+(\mathbf{r}_{i,j}^{pr}) - \frac{\beta}{\mu^-} \end{aligned}$$

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

$$\begin{aligned} u_{i,j}^+ - u_{i,j}^- &= \alpha(\mathbf{r}_{i,j}^{pr}) + \delta_{i,j} \left(\left(\frac{\mu^-}{\mu^+} - 1 \right) \partial_n u^-(\mathbf{r}_{i,j}^{pr}) + \frac{\beta(\mathbf{r}_{i,j}^{pr})}{\mu^+} \right) \\ u_{i,j}^+ - u_{i,j}^- &= \alpha(\mathbf{r}_{i,j}^{pr}) + \delta_{i,j} \left(\left(1 - \frac{\mu^+}{\mu^-} \right) \partial_n u^+(\mathbf{r}_{i,j}^{pr}) + \frac{\beta(\mathbf{r}_{i,j}^{pr})}{\mu^-} \right) \end{aligned}$$

Each of the two versions of the constraints offers an extra residual term to penalize the loss function

$$\mathcal{L}_{interface} = \left\| u_{i,j}^+ - u_{i,j}^- - \alpha(\mathbf{r}_{i,j}^{pr}) - \delta_{i,j} \left(\left(\frac{\mu^-}{\mu^+} - 1 \right) \partial_n u^-(\mathbf{r}_{i,j}^{pr}) + \frac{\beta(\mathbf{r}_{i,j}^{pr})}{\mu^+} \right) \right\|_2^2 \quad (14)$$

or

$$\mathcal{L}_{interface} = \left\| u_{i,j}^+ - u_{i,j}^- - \alpha(\mathbf{r}_{i,j}^{pr}) - \delta_{i,j} \left(\left(1 - \frac{\mu^+}{\mu^-} \right) \partial_n u^+(\mathbf{r}_{i,j}^{pr}) + \frac{\beta(\mathbf{r}_{i,j}^{pr})}{\mu^-} \right) \right\|_2^2 \quad (15)$$

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 accessible 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:

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

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

$$\begin{aligned} \mathcal{L}_{bulk} &= \left\| k^\pm u^\pm - \nabla \cdot (\mu^\pm \nabla u^\pm) - f^\pm \right\|_2^2 \\ \mathcal{L}_{boundary} &= \left\| u(\mathbf{r}_{bc}) - \hat{u}(\mathbf{r}_{bc}) \right\|_2^2 \end{aligned}$$

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

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

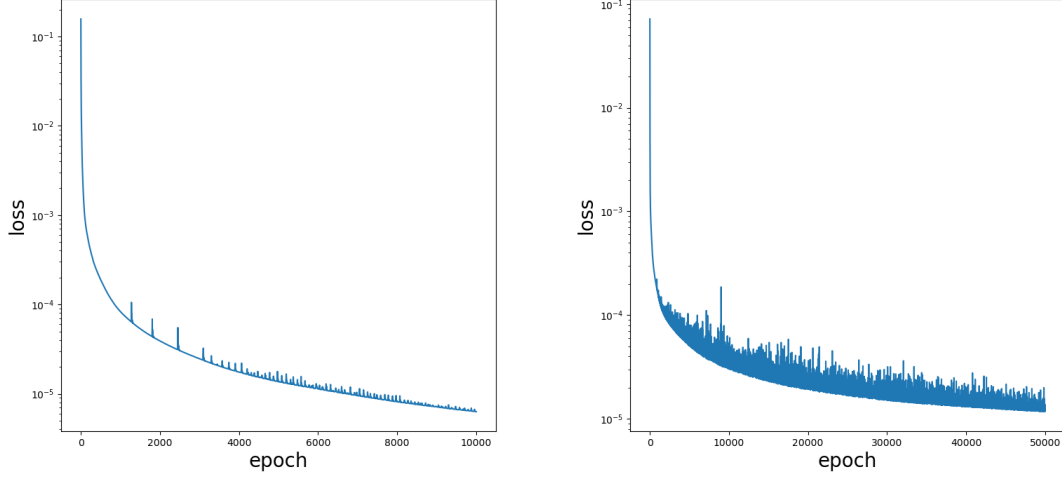


Figure 3: Loss evolution with epochs for the sphere (left) and star (right) examples.

3. Numerical Examples

We consider solution to the problem

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

In both cases a pair of neural networks with architecture $3 \times 20 \times 20 \times 20 \times 20 \times 1$ are used with `tanh` activation functions for hidden layers and linear output neuron. There are 3,562 trainable parameters (*i.e.*, weights and biases) in the model.

3.1. Case I.

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 the diffusion coefficient

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

that imply

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

After 10,000 epochs on a grid of $16 \times 16 \times 16$ the L^∞ -norm error of the solution is 1.72×10^{-1} , corresponding to a 10% relative error in the solution.

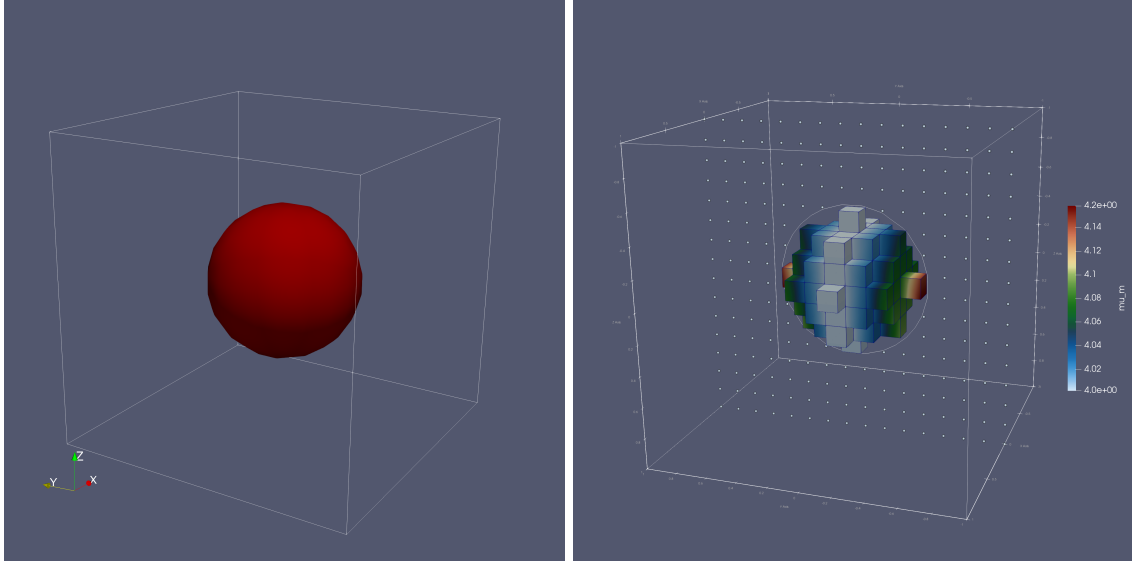


Figure 4: Illustration of three dimensional interface used (left), and μ^\pm on the $16 \times 16 \times 16$ grid (right).

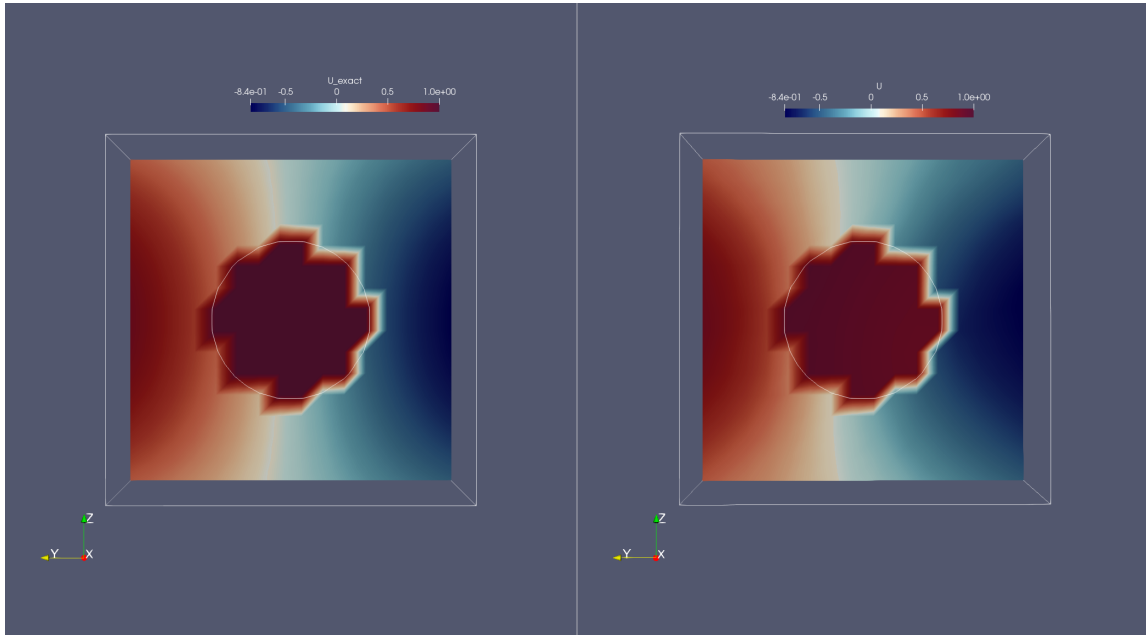


Figure 5: Illustration of exact and numerical solutions.

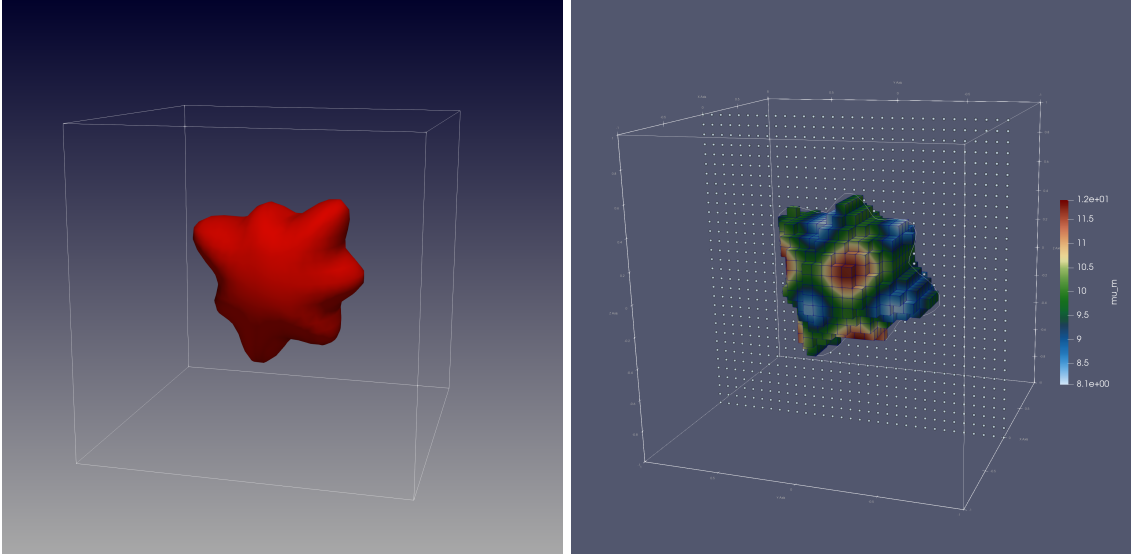


Figure 6: Illustration of three dimensional interface used (left), and μ^\pm on the $32 \times 32 \times 32$ grid (right).

3.2. Case II.

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

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

with the parameters

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

Considering an exact solution

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

and the diffusion coefficient

$$\begin{aligned} \mu^-(x, y, z) &= 10 \left[1 + 0.2 \cos(2\pi(x+y)) \sin(2\pi(x-y)) \cos(z) \right] & \phi(\mathbf{x}) < 0 \\ \mu^+(x, y, z) &= 1 & \phi(\mathbf{x}) \geq 0 \end{aligned}$$

4. Conclusion & Future Directions

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:

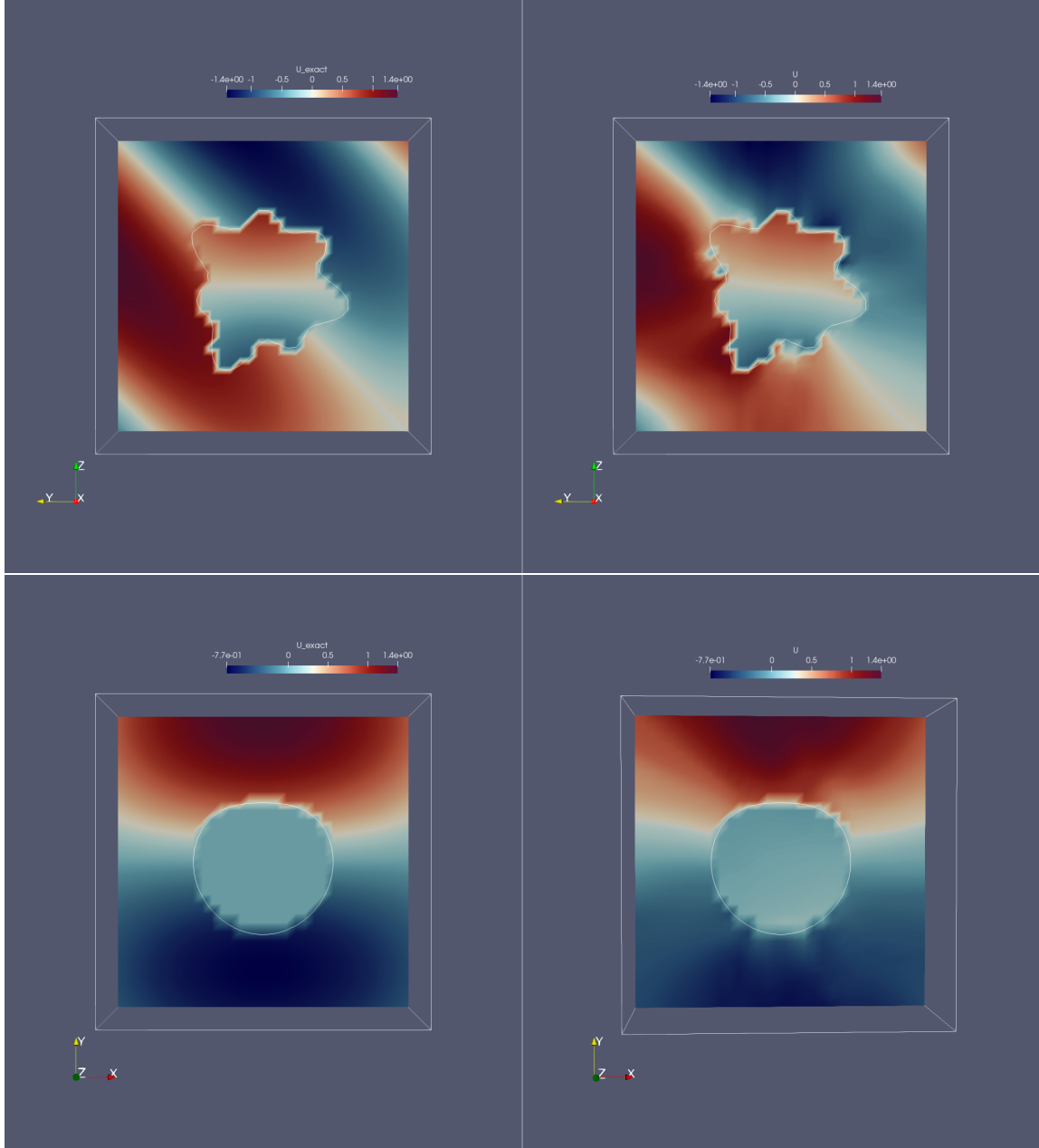


Figure 7: Illustration of exact and numerical solutions.

- 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 a plethora of deep neural network models that have shown great promise such as transformers, graph neural networks, *etc.*
- Multi-GPU training using domain decomposition and distribution using the `pmap` primitive in JAX is another very immediate improvement that will enable higher resolution simulations.
- Modification of the numerical discretization scheme in order to operate over unstructured point clouds rather than the uniform Cartesian grids.
- Extension to adaptive grids with enhanced resolutions closer to the interfaces while coarsening the grid cells in the bulk.

Acknowledgement

References

- [1] D. Bochkov and F. Gibou. Solving elliptic interface problems with jump conditions on cartesian grids. *Journal of Computational Physics*, 407:109269, 2020.
- [2] 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.
- [3] 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.
- [4] M. Hessel, D. Budden, F. Viola, M. Rosca, E. Sezener, and T. Hennigan. Optax: composable gradient transformation and optimisation, in jax!, 2020.
- [5] V. Ismailov. A three layer neural network can represent any multivariate function, 2020.
- [6] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2012.
- [13] J. F. Sallee. The middle-cut triangulations of the n-cube. *SIAM Journal on Algebraic Discrete Methods*, 5(3):407–419, 1984.
- [14] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of computational physics*, 77(2):439–471, 1988.
- [15] D. A. Sprecher. On the structure of continuous functions of several variables. *Transactions of the American Mathematical Society*, 115:340–355, 1965.
- [16] 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.
- [17] 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.