A Neural-Preconditioned Poisson Solver for Mixed Dirichlet and Neumann Boundary Conditions

Kai Weixian Lan [1] **Elias Gueidon** [2] **Ayano Kaneda** [3] **Julian Panetta** [1] **Joseph Teran** [1]

Abstract

We introduce a neural-preconditioned iterative solver for Poisson equations with mixed boundary conditions. Typical Poisson discretizations yield large, ill-conditioned linear systems. Iterative solvers can be effective for these problems, but only when equipped with powerful preconditioners. Unfortunately, effective preconditioners like multigrid (Brandt, 1977) require costly setup phases that must be re-executed every time domain shapes or boundary conditions change, forming a severe bottleneck for problems with evolving boundaries. In contrast, we present a neural preconditioner trained to efficiently approximate the inverse of the discrete Laplacian in the presence of such changes. Our approach generalizes to domain shapes, boundary conditions, and grid sizes outside the training set. The key to our preconditioner's success is a novel, lightweight neural network architecture featuring spatially varying convolution kernels and supporting fast inference. We demonstrate that our solver outperforms state-of-the-art methods like algebraic multigrid as well as recently proposed neural preconditioners on challenging test cases arising from incompressible fluid simulations.

1. Introduction

The Poisson equation is ubiquitous in scientific computing: it governs a wide array of physical phenomena, arises as a subproblem in many numerical algorithms, and serves as a model problem for the broader class of elliptic PDEs. Solving its discretized form, a linear system of equations involving the discrete Laplacian matrix, is the bottleneck in many engineering and scientific applications. These large, symmetric positive definite and sparse systems of equations are notoriously ill-conditioned. Fast Fourier Transforms (Cooley & Tukey, 1965) are optimal for these problems when discretized on trivial geometric domains, but they are not applicable for practical domain shapes. Direct methods like Cholesky factorization (Golub & Loan, 2012) resolve conditioning issues but suffer from loss of sparsity/fill-in and are prohibitively costly in practice when per-time-step refactoring is necessary (*e.g.*, with changing domain shape or coefficients). Iterative methods like preconditioned conjugate gradient (PCG) (Saad, 2003) and multigrid (Brandt, 1977) can achieve good performance, but an optimal preconditioning strategy is not generally available. Though multigrid preconditioners can guarantee modest iteration counts, computational overhead associated with solver creation and other per-iteration costs can dominate runtimes in practice. This is especially true for problems posed on evolving domains, where multigrid hierarchies *must be rebuilt at each time step*, and for nonlinear problems that require per-iteration rebuilds. Unfortunately, there is no clear algorithmic solution.

Recently, machine learning techniques have shown promise for these problems, eliminating setup costs at runtime by training a general-purpose solver *once* on a diverse set of systems offline. Tompson et al. (2017) showed that a network (FluidNet) can be used to generate an approximate inverse across domain shapes, albeit only with Neumann boundary conditions. Kaneda et al. (2023) developed the Deep Conjugate Direction Method (DCDM), which improves on FluidNet by applying a similar network structure as a *preconditioner* for an orthogonalized gradient descent on the matrix norm of the error, enabling highly accurate solutions to be obtained. While DCDM is similar to PCG, the nonlinearity of their approximate inverse required a generalization of the PCG method. Also, their approach only supports Neumann pressure boundary conditions. We build on the DCDM approach and generalize it to domains with mixed Dirichlet and Neumann boundary conditions. Notably, these problems arise in simulating free-surface liquid flows. DCDM fails on these cases, yet we show that a novel, lighter-weight network structure can be used effectively in its iterative formalism. In contrast to DCDM, our approximate inverse is a *linear* operator and

[1]University of California, Davis, USA [2]University of California, Los Angeles, USA [3]Waseda University, Tokyo, Japan. Correspondence to: Kai Weixian Lan <kai.weixian.lan@gmail.com>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

can handle mixed boundary conditions over time-varying fluid domains. Furthermore, we demonstrate that this structure drastically improves performance over DCDM.

We design our network architecture to represent the dense nature of the inverse of a discrete Laplacian matrix. That is, the inverse matrix for a discrete Laplace operator has the property that local perturbations anywhere in the domain have non-negligible effects at all other domain points. Our network structure uses a hierarchy of grid scales to improve the resolution of this non-local behavior over what is possible with DCDM's network architecture. In effect, the process of transferring information across the hierarchy from fine grid to increasingly coarse grids and back again facilitates rapid propagation of information across the domain. This structure is similar to multigrid but has some important differences. We incorporate the effects of Dirichlet and Neumann conditions at irregular boundaries with a novel convolution design. Specifically, we use stencils that learn spatially varying weights based on a voxel's proximity to the boundary and the boundary condition types encoded there. We show that our network outperforms state-of-the-art preconditioning strategies, including DCDM, FluidNet, algebraic multigrid, and incomplete Cholesky, performing our comparisons across a number of representative free-surface liquid and fluid flow problems.

In summary, our work makes the following contributions:

- We introduce a novel light-weight architecture employing spatially varying convolutional kernels that is highly effective at approximating the inverse of structured-grid Laplacian matrices with arbitrary mixed Dirichlet and Neumann boundary conditions.
- We show that a simple loss function based on the residual of the linear system suffices for training in an unsupervised manner, producing a network that generalizes to systems not seen during training.
- We demonstrate through a comprehensive benchmark on challenging fluid-simulation test cases that, when paired with an appropriate iterative method, our neural-preconditioned solver dramatically outperforms state-of-the-art solvers like algebraic multigrid and incomplete Cholesky, as well as recent neural preconditioners like DCDM and FluidNet.

To promote reproducibility, we have released our full code and a link to our pretrained model at `https://github.com/kai-lan/MLPCG/tree/icml2024`.

2. Related Work

Many recent works leverage machine learning techniques to accelerate numerical linear algebra computations. Ackmann et al. (2020) use supervised learning to compute preconditioners from fully-connected feed-forward networks in semi-implicit time stepping for weather and climate models. Sappl et al. (2019) use convolutional neural networks (CNNs) (Lecun et al., 1998) to learn banded approximate inverses for discrete Poisson equations arising in incompressible flows discretized over voxelized spatial domains. However, their loss function is the condition number of the preconditioned operator, which is prohibitively costly at high resolution. Özbay et al. (2021) also use CNNs to approximate solutions to Poisson problems arising in incompressible flow discretized over voxelized domains, but they do not learn a preconditioner and their approach only supports two-dimensional square domains. Our approach is most similar to those of Tompson et al. (2017) and Kaneda et al. (2023), who also consider discrete Poisson equations over voxelized fluid domains, however our lighter-weight network outperforms them and generalizes to a wider class of boundary conditions. Li et al. (2023) build on the approach of Sappl et al. (2019), but use a more practical loss function based on the supervised difference between the inverse of their preconditioner times a vector and its image under the matrix under consideration. Their preconditioner is the product of easily invertible, sparse lower triangular matrices. Notably, their approach works on discretizations over unstructured meshes. Götz & Anzt (2018) learn Block-Jacobi preconditioners using deep CNNs. The choice of optimal blocking is unclear for unstructured discretizations, and they use machine learning techniques to improve upon the selection. Kopaničáková & Karniadakis (2024) develop a general collection of neural DeepONet preconditioners. The most effective of these is akin to a model reduction technique with a set of reduced basis functions taken from a trained DeepONet. However, each application of the preconditioner itself requires solving a large (usually dense) linear system.

Numerous recent works use neural network architectures that are analogous to a multigrid hierarchy. He & Xu (2019) analyzed the similarities between the structure of a convolutional network and that of the multigrid method, and proposed their novel multigrid-structured network MagNet. The UNet (Ronneberger et al., 2015) and MSNet architectures (Mathieu et al., 2016) are similar to a multigrid V-cycle in terms of data flow, as noted by Cheng et al. (2021) and Azulay & Treister (2023). Cheng et al. (2021) use the multi-scale network architecture MSNet to approximate the solution of Poisson equations arising in plasma flow problems. However, they only consider flows over a square domain in 2D. Azulay & Treister (2023) note the similarity between the multi-scale UNet architecture and a multigrid V-cycle. They use this structure to learn preconditioners for the solution of

heterogeneous Helmholtz equations. Eliasof et al. (2023) also use a multigrid-like architecture for a general class of problems. Huang et al. (2023) use deep learning to generate multigrid smoothers at each grid resolution that effectively smooth high frequencies: CNNs generate the smoothing stencils from matrix entries at each level in the multigrid hierarchy. This is similar to our boundary-condition-dependent stencils, however we note that our network is lighter-weight and allowed to vary at a larger scale during learning. Furthermore, optimal stencils are known for the problems considered in this work, and we provide evidence that our solver outperforms them.

3. Motivation: Incompressible Fluid Simulation with Mixed B.C.s

While our solver architecture can be applied to any Poisson equation discretized on a structured grid, our original motivation was to accelerate a popular method for incompressible inviscid fluid simulation based on the splitting scheme introduced by (Chorin, 1967). The fluid's velocity $\mathbf{u}(\mathbf{x}, t)$ is governed by the incompressible Euler equations:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u}\right) + \nabla p = \mathbf{f}^{\text{ext}} \quad \text{s.t.} \quad \nabla\cdot\mathbf{u} = 0 \quad \text{in } \Omega,$$

where Ω is the domain occupied by fluid, pressure p is the Lagrange multiplier for the incompressibility constraint $\nabla\cdot\mathbf{u} = 0$, ρ is the mass density, and $\mathbf{f}^{\text{ext}}$ accounts for external forces like gravity. These equations are augmented with initial conditions $\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}^0(\mathbf{x})$ and $\rho(\mathbf{x}, 0) = \rho^0$ as well as the boundary conditions discussed in Section 3.1. Incompressibility implies that mass density is conserved throughout the simulation ($\rho \equiv \rho^0$).

Chorin's scheme applies finite differences in time and splits the integration from time t^n to $t^{n+1} = t^n + \Delta t$ into two steps. First, a provisional velocity field $\mathbf{u}^*$ is obtained by an *advection step* that neglects pressure and incompressibility:

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} + (\mathbf{u}^n\cdot\nabla)\mathbf{u}^n = \frac{1}{\rho^0}\mathbf{f}^{\text{ext}}. \tag{1}$$

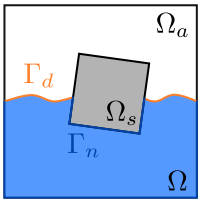
Second, a *projection step* obtains $\mathbf{u}^{n+1}$ by eliminating divergence from $\mathbf{u}^*$:

$$-\nabla\cdot\frac{1}{\rho^0}\nabla p^{n+1} = -\frac{1}{\Delta t}\nabla\cdot\mathbf{u}^*, \tag{2}$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} = -\frac{1}{\rho^0}\nabla p^{n+1}. \tag{3}$$

Equations 1-3 hold inside Ω, and we have deferred discussion of boundary conditions to Section 3.1. The bottleneck of this full process is (2), which is a Poisson equation since ρ^0 is spatially constant.

3.1. Boundary Conditions



Our primary contribution is our ability to handle both Neumann and Dirichlet boundary conditions for the Poisson equation. We assume the computational domain $\mathcal{D}$ is decomposed into $\mathcal{D} = \Omega \cup \Omega_a \cup \Omega_s$, as sketched in the inset, where Ω_a denotes free space and Ω_s the region filled with solid. This decomposition induces a partition of the fluid boundary $\partial\Omega = \Gamma_n \cup \Gamma_d$. Boundary Γ_n contains the fluid-solid interface and the intersection $\partial\Omega \cap \partial\mathcal{D}$ (*i.e.*, the region outside $\mathcal{D}$ is treated as solid); on it a free-slip boundary condition is imposed: $\mathbf{u}(\mathbf{x}, t)\cdot\hat{\mathbf{n}}(\mathbf{x}) = u_n^\Gamma(\mathbf{x}, t)$, where $\hat{\mathbf{n}}$ denotes the outward-pointing unit normal. This condition on $\mathbf{u}$ translates via (3) into a Neumann condition on (2):

$$\hat{\mathbf{n}}\cdot\nabla p^{n+1} = \frac{\rho_0}{\Delta t}(\hat{\mathbf{n}}\cdot\mathbf{u}^* - u_n^\Gamma) \quad \text{on } \Gamma_n. \tag{4}$$

Free-surface boundary Γ_d represents the interface between the fluid and free space. Ambient pressure p_a then imposes on (2) a Dirichlet condition $p^{n+1} = p_a$ on Γ_d. In our examples, we set $p_a = 0$.

The Dirichlet conditions turn out to make solving (2) fundamentally more difficult: while the DCDM paper (Kaneda et al., 2023) discovered that a preconditioner blind to the domain geometry and trained solely on an empty box is highly effective for simulations featuring pure Neumann conditions, the same is not true for Dirichlet (see Figure 6).

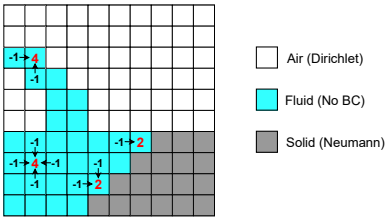
3.2. Spatial Discretization

We discretize the full domain $\mathcal{D}$ using a regular marker-and-cell (MAC) staggered grid with n_c cubic elements (Harlow, 1964). The disjoint subdomains Ω, Ω_a, and Ω_s are each represented by a per-cell rasterized indicator field; these are collected into a 3-channel image, stored as a tensor $\boldsymbol{\mathcal{I}}$. In the case of a 2D square with $n_c = N^2$, this tensor is of shape $(3, N, N)$, and summing along the first index yields a single-channel image filled with ones.

Velocities and forces are represented at the *corners* of this grid, and for smoke simulations the advection step (1) is implemented using an explicit semi-Lagrangian method (Stam, 1999; Robert, 1981). For free-surface simulations, advection is performed by interpolating fluid velocities from the grid onto particles responsible for tracking the fluid state, advecting those particles, and then transferring their velocities back to the grid. We use a PIC/FLIP blend transfer scheme with a 0.99 ratio (Zhu & Bridson, 2005).

Pressure values are stored at element *centers*, and the Laplace operator of (2) is discretized into a sparse symmetric matrix $A^{\boldsymbol{\mathcal{I}}} \in \mathbb{R}^{n_c \times n_c}$ using the standard second-order accurate finite difference stencil (with 5 points in 2D and 7 in 3D) but with modifications to ac-

count for Dirichlet and Neumann boundary conditions: stencil points falling outside Ω are dropped, and the central value (*i.e.*, the diagonal matrix entry) is determined as the number of neighboring cells belonging to either Ω or Ω_a. Examples of these stencils are visualized in 2D in the inset. Rows and columns corresponding to cells outside Ω are left empty, meaning $A^{\mathcal{I}}$ typically has a high-dimensional nullspace. These empty rows and columns are removed before solving, obtaining a smaller positive definite matrix $\tilde{A}^{\mathcal{I}} \in \mathbb{R}^{n_f \times n_f}$, where n_f is the number of fluid cells.



The right-hand side of (2) is discretized using the standard MAC divergence finite difference stencil into a vector $\mathbf{b} \in \mathbb{R}^{n_c}$, which also receives contributions from the Neumann boundary. Entries of this vector corresponding to cells outside Ω are removed to form right-hand side vector $\tilde{\mathbf{b}} \in \mathbb{R}^{n_f}$ of the reduced linear system representing the discrete Poisson equation:

$$\tilde{A}^{\mathcal{I}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}, \tag{5}$$

where $\tilde{\mathbf{x}} \in \mathbb{R}^{n_f}$ collects the fluid cells' unknown pressure values (a discretization of p^{n+1}).

The constantly changing domains and boundary conditions of a typical fluid simulation mean traditional preconditioners for (5) like multigrid or incomplete Cholesky, as well as direct sparse Cholesky factorizations, must be *rebuilt at every time step*. This prevents their high fixed costs from being amortized across frames and means they struggle to outperform a highly tuned GPU implementation of unpreconditioned CG. This motivates our neural-preconditioned solver which, after training, instantly adapts to arbitrary subdomain shapes encoded in $\mathcal{I}$.

4. Neural-preconditioned Steepest Descent with Orthogonalization

Our neural-preconditioned solver combines a carefully chosen iterative method (Section 4.1) with a preconditioner based on a novel neural network architecture (Section 4.2.1) inspired by multigrid.

4.1. Algorithm

For symmetric positive definite matrices A (like the discrete Laplacian $\tilde{A}^{\mathcal{I}}$ from (5)), the preconditioned conjugate gradient (PCG) algorithm (Shewchuk, 1994) is by far the most efficient iterative method for solving linear systems $A\mathbf{x} = \mathbf{b}$ when an effective preconditioner is available. Unfortunately, its convergence rate is known to degrade when the preconditioner itself fails to be symmetric, as is the case for our neural preconditioner (Section 4.2.2). Bouwmeester et al. (2015) have shown that good convergence can be recovered for nonsymmetric multigrid preconditioners using the "flexible PCG" variant at the expense of an additional dot product. However, this variant turns out to perform suboptimally with our neural preconditioner, as shown in Appendix A.3. Instead, we adopt the preconditioned steepest descent with orthogonalization (PSDO) method proposed in (Kaneda et al., 2023), shown in Algorithm 1, which performs well even for their *nonlinear* preconditioner.

Algorithm 1 Neural-preconditioned Steepest Descent with A-Orthogonalization (NPSDO).

Given linear system (A, $\mathbf{b}$), image $\mathcal{I}$, and trained network $\mathcal{P}^{\text{net}}$
$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
$k = 0$
while $\|\mathbf{r}_k\| \geq \epsilon$ **do**
  $k = k + 1$
  $\mathbf{d}_k = \mathcal{P}^{\text{net}}\left(\mathcal{I}, \frac{\mathbf{r}_{k-1}}{\|\mathbf{r}_{k-1}\|}\right)$
  for $k - n_{\text{ortho}} \leq i < k$ **do**
    $\mathbf{d}_k = \mathbf{d}_k - \frac{\mathbf{d}_k^\top A\mathbf{d}_i}{\mathbf{d}_i^\top A\mathbf{d}_i}\mathbf{d}_i$
  end for
  $\alpha_k = \frac{\mathbf{r}_{k-1}^\top \mathbf{d}_k}{\mathbf{d}_k^\top A\mathbf{d}_k}$
  $\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_k \mathbf{d}_k$
  $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$
end while
return $\mathbf{x}_k$

The PSDO algorithm can be understood as a modification of standard CG that replaces the residual with the preconditioned residual as the starting point for generating search directions and, consequently, cannot enjoy many of the simplifications baked into the traditional algorithm. Most seriously, A-orthogonalizing against only the previous search direction no longer suffices to achieve A-orthogonality to all past steps. Therefore, iteration k of PSDO obtains its step direction $\mathbf{d}_k$ by explicitly A-orthogonalizing the preconditioned residual against the last n_{ortho} directions (where n_{ortho} is a tunable parameter) before determining step length α_k with an exact line search. PSDO reduces to standard preconditioned steepest descent (PSD) when $n_{\text{ortho}} = 0$, and it is mathematically equivalent to unpreconditioned CG when $n_{\text{ortho}} \geq 1$ and the identity operator is used as the preconditioner. In the case of a symmetric preconditioner $P = LL^\top$, PSDO differs from PCG by taking steps that are A-orthogonal rather than $LAL^\top$-orthogonal. When combined with our neural preconditioner, we call this algorithm NPSDO, presented formally in Algorithm 1. We empirically determined

$n_{\text{ortho}} = 2$ to perform well (see Appendix A.4) and use this value in all reported experiments.

4.2. Neural Preconditioner

The ideal preconditioner for all iterative methods described in Section 4.1 is the exact inverse A^{-1}; with it, each method would converge to the exact solution in a single step. Of course, the motivation for using an iterative solver is that inverting or factorizing A is too costly (Figure 4), so instead we must seek an inexpensive approximation of A^{-1}. Examples are incomplete Cholesky, which does its best to factorize A with a limited computational budget, and multigrid, which applies iterations of a multigrid solver.

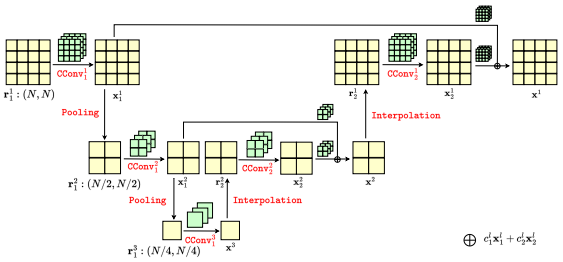
Our method approximates the map $\mathbf{r} \mapsto A^{-1}\mathbf{r}$ by our neural network $\boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{r})$. Departing from recent works like that of Kaneda et al. (2023), we use a novel architecture that both substantially boosts performance on pure-Neumann problems and generalizes to the broader class of Poisson equations with mixed boundary conditions by considering geometric information from $\boldsymbol{\mathcal{I}}$. The network performs well on 2D or 3D Poisson equations of varying sizes, but to simplify the exposition, our figures and notation describe the method on small square grids of size $N \times N$.

We note that Algorithm 1 runs on linear system $\tilde{A}^{\boldsymbol{\mathcal{I}}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, featuring vectors of smaller size n_f, but the network always operates on input vectors of full size n_c, reshaped into (N, N) tensors. Therefore, to evaluate $\tilde{\mathbf{d}} = \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \tilde{\mathbf{r}})$, $\tilde{\mathbf{r}}$ is first padded by inserting zeros into locations corresponding to cells in Ω_a and Ω_s, and then those locations of the output are removed to obtain $\tilde{\mathbf{d}} \in \mathbb{R}^{n_f}$.

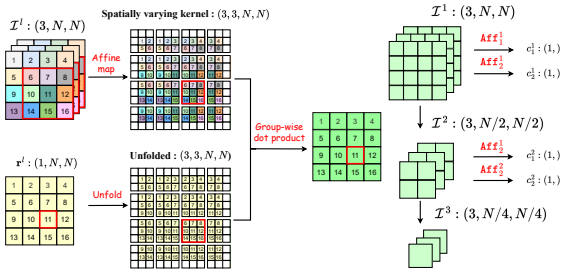
4.2.1. Architecture

Our multi-resolution network architecture (Figure 1(a)) is inspired by UNet and geometric multigrid, aiming to propagate information across the computational grid faster than the one-cell-per-iteration of unpreconditioned CG. The architecture is defined recursively, consisting of levels $1 \leq \ell \leq \mathcal{L}$ each executing the operations detailed in Algorithm 2. A given level ℓ operates on an input image $\boldsymbol{\mathcal{I}}^\ell$ of shape $(3, N^\ell, N^\ell)$ and an input vector $\mathbf{r}_1^\ell$. It performs a special image-dependent convolution operation defined by our custom convolution block `CConv` and then downsamples the resulting vector $\mathbf{x}_1^\ell$, as well as $\boldsymbol{\mathcal{I}}^\ell$, to the next-coarser level $\ell + 1$ using average pooling with kernel size 2×2. The output of the level $\ell+1$ subnetwork is then upsampled with bilinear interpolation, run through another convolution stage, and finally linearly combined with $\mathbf{x}_1^\ell$ to obtain the output. At the finest level, $\boldsymbol{\mathcal{I}}^1 = \boldsymbol{\mathcal{I}}$ and $\mathbf{r}_1^1 = \mathbf{r}$, while at the coarsest level only a single convolution operation is performed.

One crucial difference between our network and existing neural solvers like FluidNet (Tompson et al., 2017) is how geometric information from $\boldsymbol{\mathcal{I}}$ is incorporated. Past architectures treat this geometric data on the same footing as input tensor $\mathbf{r}$, *e.g.* feeding both into standard multi-channel convolution blocks. However, we note that $\boldsymbol{\mathcal{I}}$ determines the entries of $A^{\boldsymbol{\mathcal{I}}}$, and so for the convolutions to act analogously to the smoothing operations of multigrid, this data should inform the weights of convolutions applied to $\mathbf{r}$. This observation motivates our use of custom convolutional



(a) Full network architecture sketched for $\mathcal{L} = 3$ levels.



(b) Custom convolution block $\texttt{CConv}_i^\ell$ with $i = 1, 2$.

(c) Affine block $\texttt{Aff}_i^\ell$ with $i = 1, 2$.

Figure 1. Sketches of our network architecture.

Algorithm 2 Operations executed by an $\mathcal{L}$-level network in pseudocode form.

Given input vector $\mathbf{r}$ and image $\boldsymbol{\mathcal{I}}$
$\mathbf{r}_1^1 = \mathbf{r}$
$\boldsymbol{\mathcal{I}}^1 = \boldsymbol{\mathcal{I}}$
for $\ell = 1, \cdots, \mathcal{L} - 1$ **do**
  $\boldsymbol{\mathcal{I}}^{\ell+1} = \texttt{Pooling}(\boldsymbol{\mathcal{I}}^\ell)$
  $\mathbf{x}_1^\ell = \texttt{CConv}_1^\ell(\mathbf{r}_1^\ell, \boldsymbol{\mathcal{I}}^\ell)$
  $\mathbf{r}_1^{\ell+1} = \texttt{Pooling}(\mathbf{x}_1^\ell)$
end for
$\mathbf{x}^\mathcal{L} = \texttt{CConv}_1^\mathcal{L}(\mathbf{r}_1^\mathcal{L}, \boldsymbol{\mathcal{I}}^\mathcal{L})$
for $\ell = \mathcal{L} - 1, \ldots, 1$ **do**
  $\mathbf{r}_2^\ell = \texttt{Interpolation}(\mathbf{x}^{\ell+1})$
  $\mathbf{x}_2^\ell = \texttt{CConv}_2^\ell(\mathbf{r}_2^\ell, \boldsymbol{\mathcal{I}}^\ell)$
  $c_1^\ell = \texttt{Aff}_1^\ell(\boldsymbol{\mathcal{I}}^\ell)$
  $c_2^\ell = \texttt{Aff}_2^\ell(\boldsymbol{\mathcal{I}}^\ell)$
  $\mathbf{x}^\ell = c_1^\ell \mathbf{x}_1^\ell + c_2^\ell \mathbf{x}_2^\ell$
end for
return $\mathbf{x}^1$

blocks `CConv` whose *spatially varying kernels* depend on local information from $\boldsymbol{\mathcal{I}}$. We note the close connection between these varying kernels and attention (Bahdanau et al., 2014; Vaswani et al., 2017).

Each custom convolutional block (Figure 1(b)) at level ℓ learns an affine map from a 3×3 sliding window in $\boldsymbol{\mathcal{I}}^\ell$ to a 3×3 kernel $\boldsymbol{\mathcal{K}}^{i,j}$. This affine map is parametrized by a weights tensor $\boldsymbol{\mathcal{W}}$ of shape $(3^2, 3, 3, 3)$ and a bias vector $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{3^2}$. Each entry of the block's output is computed as:

$$\left[\texttt{CConv}^\ell(\mathbf{r}, \boldsymbol{\mathcal{I}}^\ell)\right]_{i,j} = \sum_{a,b=-1}^{1} \mathcal{K}^{i,j}_{a,b} r_{i+a,j+b},$$

$$\mathcal{K}^{i,j}_{a,b} := \sum_{c=1}^{3} \sum_{l,m=-1}^{1} \mathcal{W}_{3a+b,c,l,m} \mathcal{I}^\ell_{c,i+l,j+m} + \mathcal{B}_{3a+b}.$$

Out-of-bounds accesses in these formulas are avoided by padding $\boldsymbol{\mathcal{I}}^\ell$ with solid pixels (*i.e.*, the values assigned to cells in Ω_s) and $\mathbf{x}$ with zeros.

In multigrid, solutions obtained on coarser grids are *corrections* that are added to the finer grids' solutions; likewise, our network mixes in upsampled data from the lower level using the linear combination $c^\ell_1 \mathbf{x}^\ell_1 + c^\ell_2 \mathbf{x}^\ell_2$. The coefficients in this combination are defined by (i) convolving $\boldsymbol{\mathcal{I}}^\ell$ with a (spatially constant) kernel $\overline{\boldsymbol{\mathcal{W}}}$ of shape $(3, 3, 3)$; (ii) averaging to produce a scalar; and (iii) adding a scalar bias $\overline{\mathcal{B}}$. For efficiency, these evaluation steps are fused into a custom affine block (Figure 1(c)) that implements the formula:

$$\texttt{Aff}^\ell(\boldsymbol{\mathcal{I}}) = \overline{\mathcal{B}} + \frac{1}{3^2 n_c} \sum_{i,j=1}^{N^\ell} \sum_{c=1}^{3} \sum_{l,m=-1}^{1} \overline{\mathcal{W}}_{c,l,m} \mathcal{I}^\ell_{c,i+l,j+m}.$$

Our custom network architecture has numerous advantages. Its output is a linear function of the input vector (unlike the nonlinear map learned by (Kaneda et al., 2023)), making it easier to interpret as a preconditioner. The architecture is also very lightweight: a model with $\mathcal{L} = 4$ coarsening levels has only ~ 25k parameters. Its simplicity accelerates network evaluations at solve time, critical to make NPSDO competitive with the state-of-the-art solvers used in practice. We note that our solver is fully matrix free, with $\boldsymbol{\mathcal{P}}^{\text{net}}$ relying only on the image $\boldsymbol{\mathcal{I}}$ of the simulation scene to infer information about $A^{\boldsymbol{\mathcal{I}}}$. Furthermore, since all network operations are formulated in terms of local windows into $\boldsymbol{\mathcal{I}}$ and $\mathbf{r}$, it can train and run on *problems of any size divisible by* $2^{\mathcal{L}-1}$.

The 3D version of our architecture is a straightforward extension of the 2D formulas above, simply using larger tensors with additional indices to account for the extra dimension and extending the sums to run over these indices.

4.2.2. Symmetry and Positive Definiteness

Our network enforces neither symmetry nor positive definiteness, properties that would be needed to guarantee convergence of the standard PCG algorithm (motiving our use of NPSDO). However, we note that the additional operations in NPSDO vs PCG necessitated by asymmetry (*e.g.*, the extra A-orthogonalizations) do not add significant overhead compared to the cost of network evaluation, which accounts for approximately 80% of the solver time. Furthermore, the strong, albeit suboptimal, performance of the plain PCG algorithm (Appendix A.3) suggests that the deviations of our preconditioner from symmetry and positive definiteness are not severe; this is confirmed by experiments reported in Appendix A.8 that measure the magnitude of symmetry violation of our trained operator on randomly generated vectors.

Nevertheless, we experimented with enforcing positive definiteness by construction in two different ways: (i) concatenating our network with a transposed copy of itself; and (ii) constraining "post-smoothing" convolution blocks to be transposes of their corresponding "pre-smoothing" block and modifying the shortcut connection. Both architecture variants led to a significant degradation in preconditioner quality when trained using the same methodology as our proposed network (Appendix A.8).

4.2.3. Training

We train our network $\boldsymbol{\mathcal{P}}^{\text{net}}$ to approximate $A^{\boldsymbol{\mathcal{I}}} \backslash \mathbf{b}$ when presented with image $\boldsymbol{\mathcal{I}}$ and input vector $\mathbf{b}$. We calculate the loss for an example $(\boldsymbol{\mathcal{I}}, A^{\boldsymbol{\mathcal{I}}}, \mathbf{b})$ from our training dataset as the residual norm:

$$Loss = \left\| \mathbf{b} - A^{\boldsymbol{\mathcal{I}}} \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{b}) \right\|_2.$$

We found the more involved loss function used in (Kaneda et al., 2023) not to benefit our network.

Our training data set consists of 107 matrices collected from 11 different simulation scenes, some of domain shape (128, 128, 128) and others (256, 128, 128). For each matrix, we generate 800 right-hand side vectors using a similar approach to (Kaneda et al., 2023) but with far fewer Rayleigh-Ritz vectors. We first compute 1600 Ritz vectors using Lanczos iterations (Lanczos, 1950) and then generate from them 800 random linear combinations. These linear combinations are finally normalized and added to the training set. To accelerate data generation, we create the right-hand sides for different matrices in parallel; it takes between 0.5 and 3 hours to generate the data for each scene. As Ritz vector calculation is expensive, we experimented with other approaches, like picking random vectors or constructing analytical eigenmodes for the Laplacian on $\mathcal{D}$ and masking out entries outside Ω. Unfortunately these cheaper generation techniques led to degraded performance.

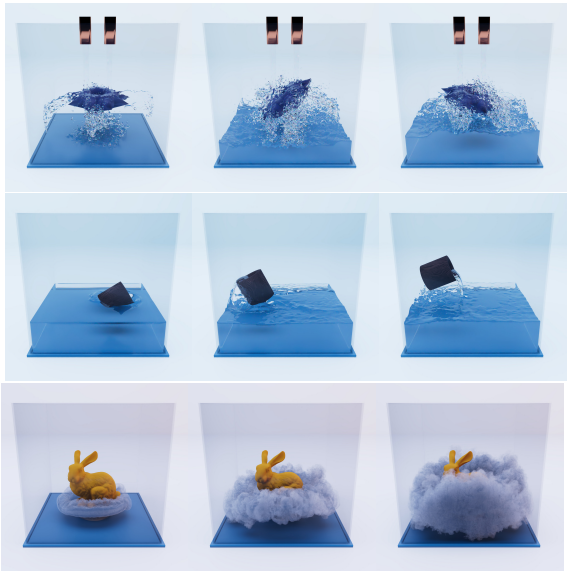
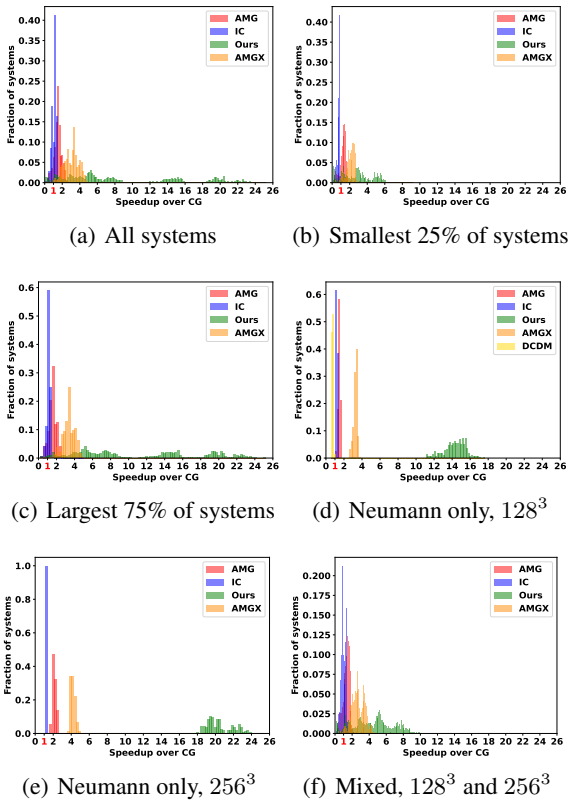


Figure 2. We demonstrate our solver with incompressible flow simulations requiring the solution of mixed Neumann/Dirichlet boundary conditions for the pressure Poisson equation.



(a) All systems (b) Smallest 25% of systems

(c) Largest 75% of systems (d) Neumann only, 128^3

(e) Neumann only, 256^3 (f) Mixed, 128^3 and 256^3

Figure 3. Histograms of solution speedup vs. a baseline of unpreconditioned CG (a) for all solves; and (b-f) for certain subsets of the systems to help tease apart different modes of the distribution.

In each epoch of training, we loop over all matrices in our dataset in shuffled order. For each matrix, we process all of its 800 right-hand sides in batches of 128, repeating five times. The full training process takes around 5 days on an AMD EPYC 9554P 64-Core Processor with an NVIDIA RTX 6000 GPU. We utilize the transfer learning technique (Pan & Yang, 2010), training first a 5-level network and using those weights to initialize a 6-level network, which is subsequently fine-tuned and used for all experiments.

4.2.4. Implementation

We built our network using PyTorch (Paszke et al., 2019), but implemented our convolutional and linear blocks as custom CUDA extensions. The neural network was trained using single-precision floating point.

5. Results and Analysis

We evaluate the effectiveness and efficiency of our neural preconditioned solver by comparing it to high-performance state-of-the-art implementations of several baseline methods: unpreconditioned CG provided by the CuPy library (Okuta et al., 2017), as well as CG preconditioned by the algebraic multigrid (AMG) and incomplete Cholesky (IC) implementations from the AMGCL library (Demidov, 2020). We furthermore compare against NVIDIA's AmgX algebraic multigrid solver (Naumov et al., 2015). All of these baseline methods are accelerated by CUDA backends running on the GPU, with the underlying IC implementation coming from NVIDIA's cuSparse library. Where appropriate, we also compare against past neural preconditioners FluidNet (Tompson et al., 2017) and DCDM (Kaneda et al., 2023). Finally, we include characteristic performance statistics of a popular sparse Cholesky solver CHOLMOD (Chen et al., 2008). In all cases, our method outperforms these baselines, often dramatically. We emphasize that a *single* trained $\mathcal{P}^{\text{net}}$ instance is used throughout, demonstrating its capacity to generalize to across simulation scenarios and grid shapes (see also Appendix A.7).

We executed all benchmarks on a workstation featuring an AMD Ryzen 9 5950X 16-Core Processor and an NVIDIA GeForce RTX 3080 GPU. We used as our convergence criterion for all methods a reduction of the residual norm by a factor of 10^6, which is sufficiently accurate to eliminate visible simulation artifacts. We evaluate our neural preconditioner in single precision floating point for efficiency but implement the rest of the NPSDO algorithm in double precision for numerical stability. We note that this use of single precision does not limit the solver's overall accuracy, as demonstrated empirically in Appendix A.9.

We benchmarked on twelve simulation scenes with various shapes—$(128, 128, 128)$, $(256, 128, 128)$, and

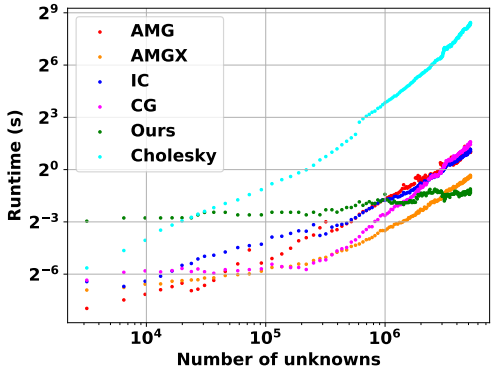


Figure 4. Solver scaling for mixed BC system matrices originating from a fixed-resolution domain ($n_c = 256^3$); matrix row/col size n_f is determined by the proportion of cells occupied by fluid. The vast majority of total solve time is contributed by the high-occupancy systems clustered to the right, where our method outperforms the rest.

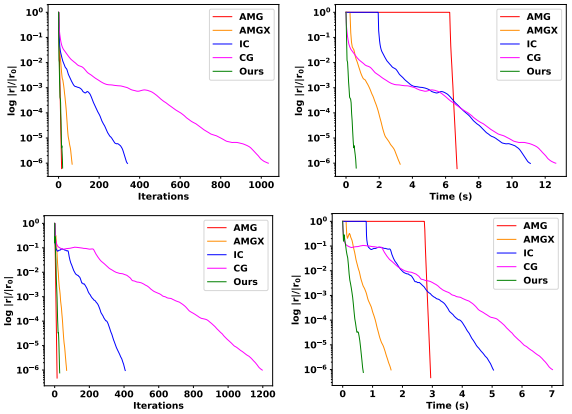


Figure 5. Comparisons among AMG, IC, CG and NSPDO (Ours) on a single frame at 256^3 with Neumann only BC (top two) and mixed BC (bottom two).

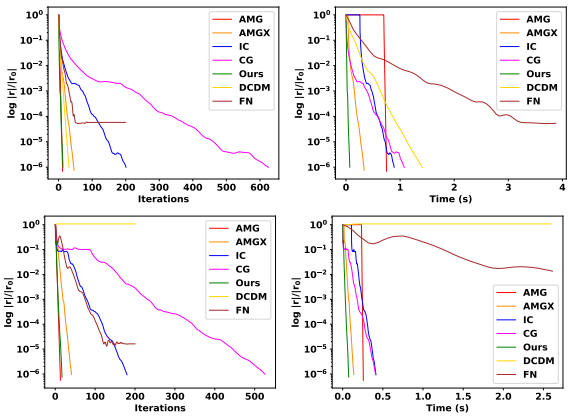


Figure 6. Comparisons among AMG, IC, CG, DCDM, FluidNet (FN) and NSPDO (Ours) on a single frame at 128^3 with Neumann only BC (top two) and mixed BC (bottom two).

$(256, 256, 256)$—each providing 200 linear systems to solve. For each solve, we recorded the number of iterations and runtime taken by each solver. These performance statistics are summarized visually in Figures 3-4 and Appendix A.1, as well as in tabular form in Appendix A.5.

Figure 3(a) summarizes timings from all solves in our benchmark suite: for each system, we divide the unpreconditioned CG solve time by the other methods' solve times to calculate their speedups and plot a histogram. We note that our method significantly outperforms the others on a majority of solves: ours is fastest on 95.6% of the systems, which account for 98.0% of our total solve time.

Our improvements are more substantial on larger problems (Figures 3(b) and 3(c)) for two reasons. First, condition numbers increase with size, impeding solvers without effective preconditioners; this is seen clearly by comparing results from two different resolutions (Figures 3(d) and 3(e)). Second, the small matrices $\tilde{A}^{\mathcal{I}}$ correspond to simulation grids with mostly non-fluid cells. While CG, AMGCL, AmgX, and IC timings shrink significantly as fluid cells are removed, our network's evaluation cost does not: it always processes all of $\mathcal{D}$ regardless of occupancy. This scaling behavior is visible in Figure 4.

Our speedups are also greater for examples with $\Gamma_d = \emptyset$. DCDM is applicable for these, and so we included in it Figure 3(d) (but not in Figure 3(e) due to the network overspilling GPU RAM). DCDM's failure to outperform CG and IC in these results, contrary to (Kaneda et al., 2023), is due to the higher-performance CUDA-accelerated implementations of those baselines used in this work. With Dirichlet conditions (Figure 3(f)), our preconditioner is less effective, and yet we still outperform the rest on 93.27% of frames, which account for 96.36% of our total solve time. Statistics are not reported in this setting for DCDM and FluidNet, which struggle to reduce the residual (Figure 6).

Further insights can be obtained by consulting Figures 5 and 6, which show the convergence behavior of each iterative solver on characteristic example problems. AMG is clearly the most effective preconditioner, but this comes at the high cost of rebuilding the multigrid hierarchy before each solve: its iterations cannot even start until long after our solver already converged. Our preconditioner is the second most effective and, due to its lightweight architecture, achieves the fastest solves. DCDM is also quite effective at preconditioning Neumann-only problems but is slowed by costly network evaluations. IC's setup time is shorter than AMG but still substantial, and it is much less effective as a preconditioner; the same holds for AmgX.

We note that the smoke example (Figure 6) also includes a comparison to FluidNet *applied as a preconditioner* for

PSDO. In the original paper, FluidNet was presented as a standalone solver, to be run just once per simulation frame. However, in this form it cannot produce highly accurate solutions. Incorporating it as a preconditioner as we do here in theory allows the system to be solved to controlled accuracy, but this solver ended up stalling before reaching a 10^6 reduction in our experiments; it was omitted from Figure 3 for this reason.

On average, our solver spends 79.4% of its time evaluating $\boldsymbol{\mathcal{P}}^{\text{net}}$, 4.4% of its time in orthogonalization, and the remaining 16.2% in other CG operations. In contrast, AMG takes a full 90% of its time in its setup stage. IC's quicker construction and slower convergence mean it takes only 23% in setup. Our architecture also confers GPU memory usage benefits: for 128^3 grids, our solver uses 1.5GiB of RAM, while FluidNet and DCDM consume 5.2GiB and 8.5GiB, respectively (Appendix A.6).

5.1. Ablation Studies

To demonstrate the necessity of each component of our solver, we performed several ablation studies. Regarding the network architecture, we evaluated a simple CNN with fixed kernels and masked output, concatenating image and input vectors into separate channels. However, this approach performed poorly compared to our network. Specifically, the residual obtained by the masked CNN solver increased after the first few iterations and remained high even after reaching the 100 iteration cap applied in this experiment. Table 1 summarizes this comparison. We also investigated the effectiveness of several PCG variants, discussed in Appendix A.3.

5.2. Hyperparameter Selection and More Experiments

The ideal number of network levels $\mathcal{L}$ and orthogonalizations n_{ortho} were determined empirically as detailed in Appendix A.2 and Appendix A.4, respectively.

To ensure fairness of our comparisons, we ran additional experiments on examples from FluidNet's original training dataset processed according to the instructions provided with their open-source code release. Naturally, these examples contain only pure-Neumann boundary conditions. We compared our model to DCDM and FluidNet, both as a standalone solver and as a preconditioner. For all iterative methods, except unpreconditioned Conjugate Gradient (CG), we capped the maximum iteration count at 100. Notably, FluidNet, when used as a preconditioner, failed to converge to a residual tolerance of 10^{-6}. As a standalone solver, FluidNet runs quickly but is highly ineffective at reducing the residual. These results are reported in Table 2.

Table 1. Comparison among our current network and CNN with masked output and CG

Methods	Iteration count	Time	Residual
CG	390	0.1125	9.72×10^{-7}
Standard CNN	100	0.1025	3.23×10^{-2}
Ours	12	0.027	4.61×10^{-7}

Table 2. Comparison among FluidNet (as a solver and as a preconditioner), DCDM, ours and CG

Methods	Iteration count	Time	Final residual
CG	790	0.5615	1.00×10^{-6}
Ours	19	0.05237	8.77×10^{-7}
FluidNet	1	0.03084	2.035
FluidNet PSDO	100	3.116	2.10×10^{-5}
DCDM	39	12.35	9.81×10^{-7}

6. Conclusions

The neural-preconditioned solver we propose not only addresses more general boundary conditions than past machine learning approaches for the Poisson equation (Tompson et al., 2017; Kaneda et al., 2023) but also dramatically outperforms these solvers. It even surpasses state-of-the art high-performance implementations of standard methods like algebraic multigrid and incomplete Cholesky. It achieves this through a combination of its strong efficacy as a preconditioner and its fast evaluations enabled by our novel lightweight architecture.

Nevertheless, we see several opportunities to improve and extend our solver in future work. Although we implemented our spatially-varying convolution block in CUDA, it remains the computational bottleneck of the network evaluation and is not yet fully optimized. We are excited to try porting our architecture to special-purpose acceleration hardware like Apple's Neural Engine; not only could this offer further speedups, but also it would free up GPU cycles for rendering the results in real-time applications like visual effects and games. For applications where fluid occupies only a small portion of the computational domain, we would like to develop techniques to exploit sparsity for better scaling (Figure 4). Finally, we look forward to extending our ideas to achieve competitive performance for problems posed on unstructured grids as well as equations with non-constant coefficients, vector-valued unknowns (*e.g.*, elasticity), and nonlinearities.

Acknowledgements

Kai Lan was supported on a UC Multiple Campus Award (MCA) MCA-001639-M23PL6076. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsor.

Impact Statement

This paper aims to advance the field of computational fluid simulation. We do not foresee any potential societal consequences resulting from our work that warrant discussion.

References

Ackmann, J., Düben, P. D., Palmer, T. N., and Smolarkiewicz, P. K. Machine-learned preconditioners for linear solvers in geophysical fluid flows. *arXiv preprint arXiv:2010.02866*, 2020.

Azulay, Y. and Treister, R. Multigrid-augmented deep learning preconditioners for the helmholtz equation. *SIAM Journal on Scientific Computing*, 45(3):S127–S151, 2023. doi: 10.1137/21M1433514. URL https://doi.org/10.1137/21M1433514.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Bouwmeester, H., Dougherty, A., and Knyazev, A. Nonsymmetric preconditioning for conjugate gradient and steepest descent methods. *Procedia Computer Science*, 51:276–285, 2015. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2015.05.241. URL https://www.sciencedirect.com/science/article/pii/S1877050915010492. International Conference On Computational Science, ICCS 2015.

Brandt, A. Multi-level adaptive solutions to boundary-value problems. *Math Comp*, 31(138):333–390, 1977.

Chen, Y., Davis, T., Hager, W., and Rajamanickam, S. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3), oct 2008. ISSN 0098-3500. doi: 10.1145/1391989.1391995. URL https://doi.org/10.1145/1391989.1391995.

Cheng, L., Illarramendi, E., Bogopolsky, G., Bauerheim, M., and Cuenot, B. Using neural networks to solve the 2d poisson equation for electric field computation in plasma fluid simulations. *arXiv preprint arXiv:2109.13076*, 2021.

Chorin, A. A numerical method for solving incompressible viscous flow problems. *J Comp Phys*, 2(1):12–26, 1967.

Cooley, J. and Tukey, J. An algorithm for the machine calculation of complex fourier series. *Math Comp*, 19(90):297–301, 1965.

Demidov, D. Amgcl —a c++ library for efficient solution of large sparse linear systems. *Software Impacts*, 6:100037, 2020. ISSN 2665-9638. doi: https://doi.org/10.1016/j.simpa.2020.100037. URL https://www.sciencedirect.com/science/article/pii/S2665963820300282.

Eliasof, M., Ephrath, J., Ruthotto, L., and Treister, E. Mgic: Multigrid-in-channels neural network architectures. *SIAM Journal on Scientific Computing*, 45(3):S307–S328, 2023. doi: 10.1137/21M1430194. URL https://doi.org/10.1137/21M1430194.

Golub, G. and Loan, C. V. *Matrix computations*, volume 3. JHU Press, 2012.

Götz, M. and Anzt, H. Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, pp. 49–56, 2018. doi: 10.1109/ScalA.2018.00010.

Harlow, F. The particle-in-cell method for numerical solution of problems in fluid dynamics. *Meth Comp Phys*, 3:319–343, 1964.

He, J. and Xu, J. Mgnet: A unified framework of multigrid and convolutional neural network. *Science china mathematics*, 62:1331–1354, 2019.

Huang, R., Li, R., and Xi, Y. Learning optimal multigrid smoothers via neural networks. *SIAM Journal on Scientific Computing*, 45(3):S199–S225, 2023. doi: 10.1137/21M1430030. URL https://doi.org/10.1137/21M1430030.

Kaneda, A., Akar, O., Chen, J., Kala, V., Hyde, D., and Teran, J. A deep conjugate direction method for iteratively solving linear systems. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 15720–15736. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/kaneda23a.html.

Kopaničáková, A. and Karniadakis, G. Deeponet based preconditioning strategies for solving parametric linear systems of equations. *arXiv preprint arXiv:2401.02016*, 2024.

Lanczos, C. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. 1950.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Li, Y., Chen, P., Du, T., and Matusik, W. Learning preconditioners for conjugate gradient PDE solvers. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19425–19439. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/li23e.html.

Mathieu, M., Couprie, C., and LeCun, Y. Deep multi-scale video prediction beyond mean square error, 2016.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., and Strzodka, R. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015. doi: 10.1137/140980260.

Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. URL http://learningsys.org/nips17/assets/papers/paper_16.pdf.

Özbay, A., Hamzehloo, A., Laizet, S., Tzirakis, P., Rizos, G., and Schuller, B. Poisson cnn: Convolutional neural networks for the solution of the poisson equation on a cartesian mesh. *Data-Centric Engineering*, 2:e6, 2021. doi: 10.1017/dce.2021.7.

Pan, S. J. and Yang, Q. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22 (10):1345–1359, 2010. doi: 10.1109/TKDE.2009.191.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Robert, A. A stable numerical integration scheme for the primitive meteorological equations. *Atm Ocean*, 19(1): 35–46, 1981.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation, 2015.

Saad, Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003. ISBN 0898715342.

Sappl, J., Seiler, L., Harders, M., and Rauch, W. Deep learning of preconditioners for conjugate gradient solvers in urban water related problems, 2019. URL https://arxiv.org/abs/1906.06925.

Shewchuk, J. An introduction to the conjugate gradient method without the agonizing pain. Technical report, USA, 1994.

Stam, J. Stable fluids. In *Siggraph*, volume 99, pp. 121–128, 1999.

Tompson, J., Schlachter, K., Sprechmann, P., and Perlin, K. Accelerating Eulerian fluid simulation with convolutional networks. In Precup, D. and Teh, Y. (eds.), *Proc 34th Int Conf Mach Learn*, volume 70 of *Proc Mach Learn Res*, pp. 3424–3433. PMLR, 06–11 Aug 2017.

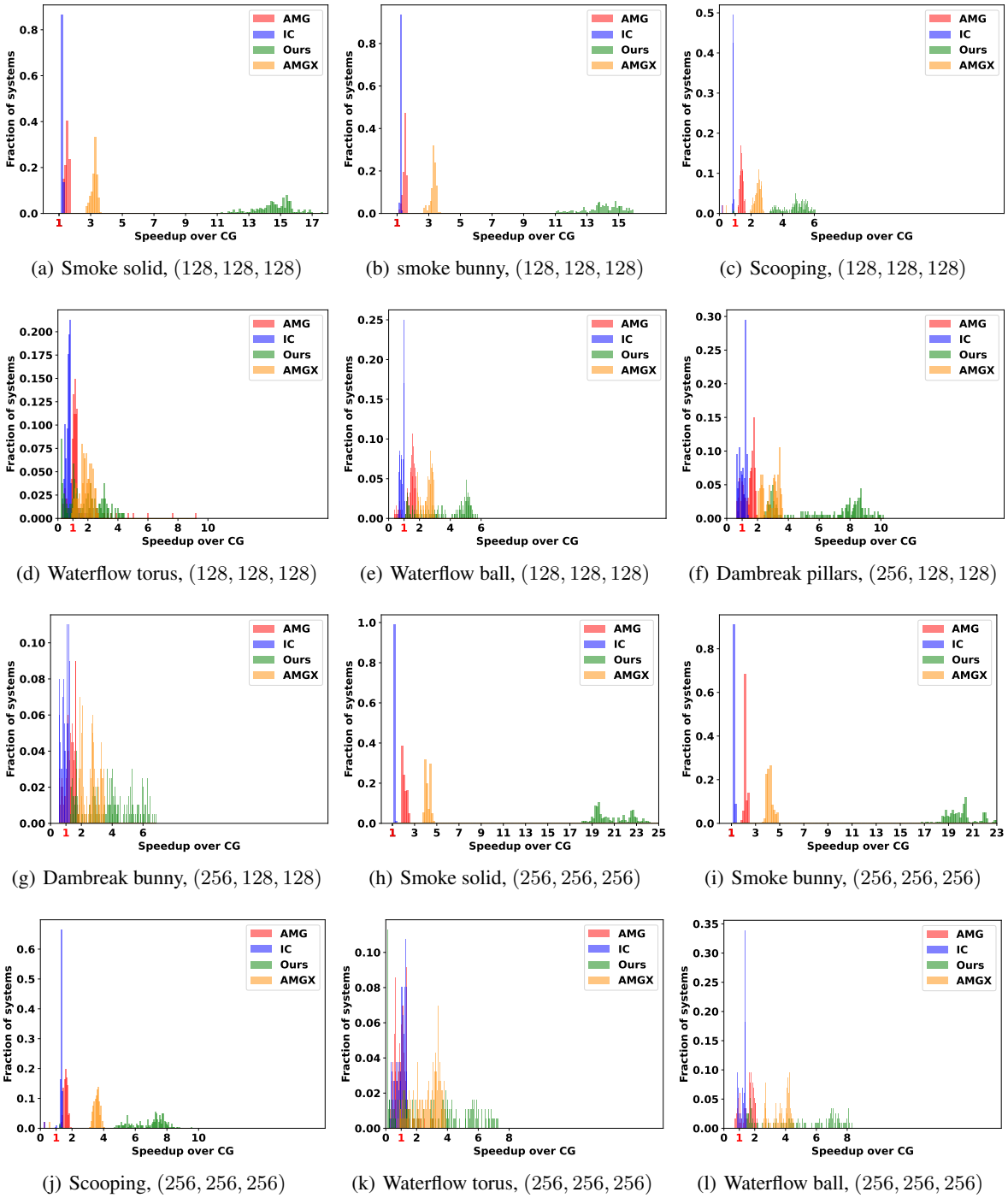
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Zhu, Y. and Bridson, R. Animating sand as a fluid. *ACM Trans Graph*, 24(3):965–972, 2005.

A. Appendix

A.1. Additional Histograms

The following histograms offer additional views into the data presented in Figure 3, focusing on the linear systems arising from simulations with mixed boundary conditions (*i.e.*, featuring both Dirichlet and Neumann conditions).



(a) Smoke solid, $(128, 128, 128)$ (b) smoke bunny, $(128, 128, 128)$ (c) Scooping, $(128, 128, 128)$

(d) Waterflow torus, $(128, 128, 128)$ (e) Waterflow ball, $(128, 128, 128)$ (f) Dambreak pillars, $(256, 128, 128)$

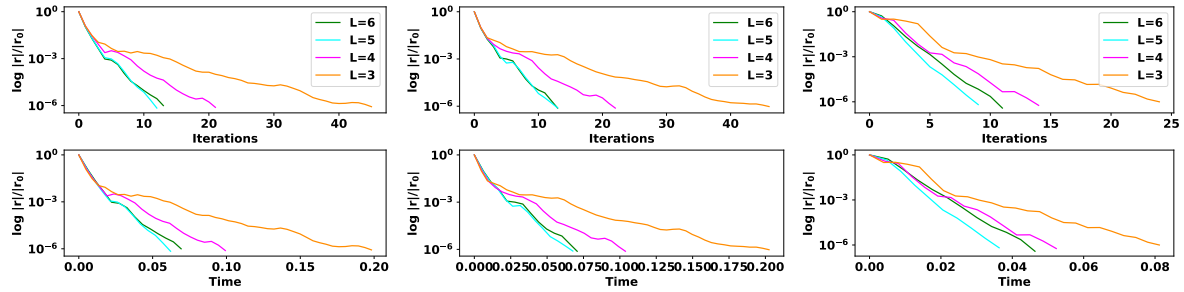
(g) Dambreak bunny, $(256, 128, 128)$ (h) Smoke solid, $(256, 256, 256)$ (i) Smoke bunny, $(256, 256, 256)$

(j) Scooping, $(256, 256, 256)$ (k) Waterflow torus, $(256, 256, 256)$ (l) Waterflow ball, $(256, 256, 256)$

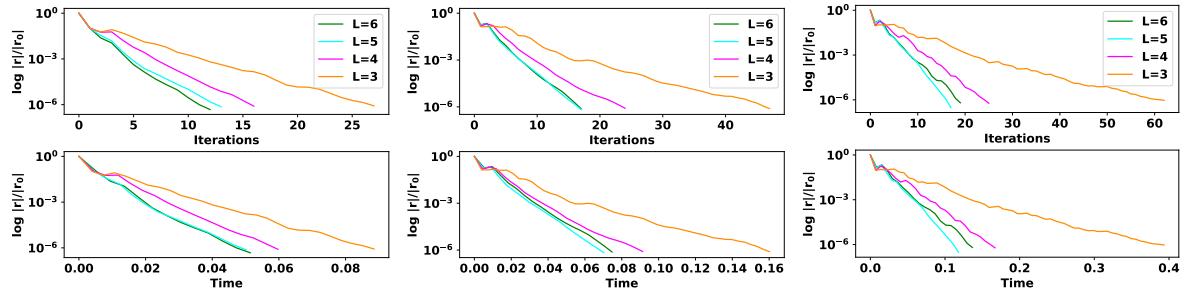
Figure 7. Mixed BC examples split by linear system size into smallest 25% and largest 75%.

A.2. Analysis on depth of the network

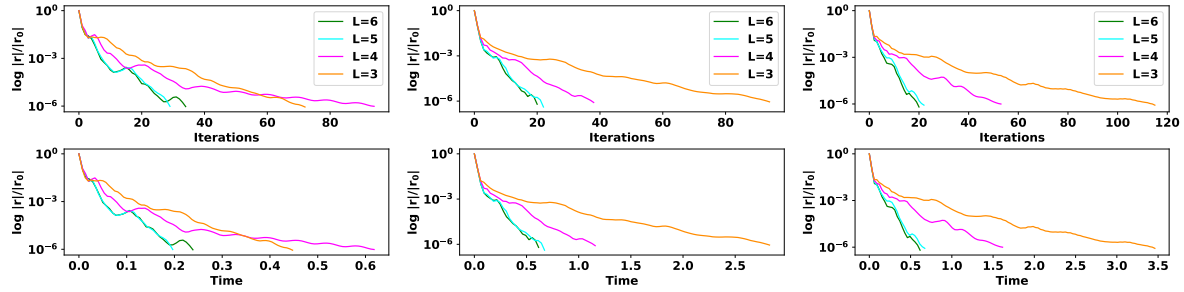
The depth $\mathcal{L}$ of our network model is a hyperparameter. To study its impact on the performace of our solver, we compared the models for $\mathcal{L} = 3, 4, 5, 6$ on a few examples.



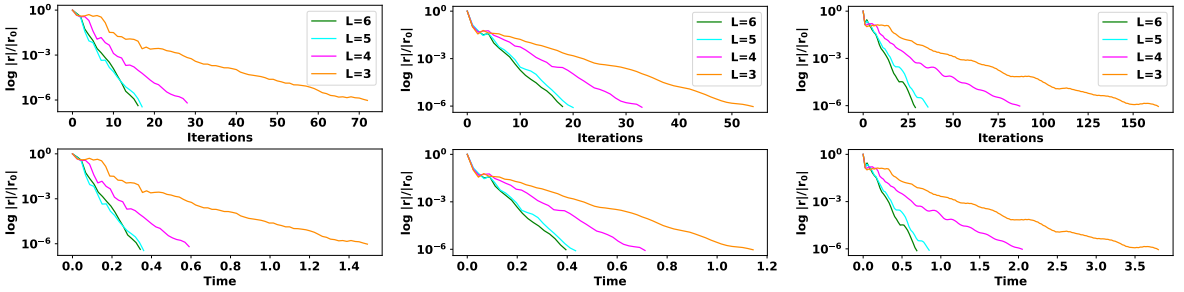
(a) Smoke solid, $(128, 128, 128)$ (b) smoke bunny, $(128, 128, 128)$ (c) Scooping, $(128, 128, 128)$



(d) Waterflow torus, $(128, 128, 128)$ (e) Waterflow ball, $(128, 128, 128)$ (f) Dambreak pillars, $(256, 128, 128)$



(g) Dambreak bunny, $(256, 128, 128)$ (h) Smoke solid, $(256, 256, 256)$ (i) Smoke bunny, $(256, 256, 256)$

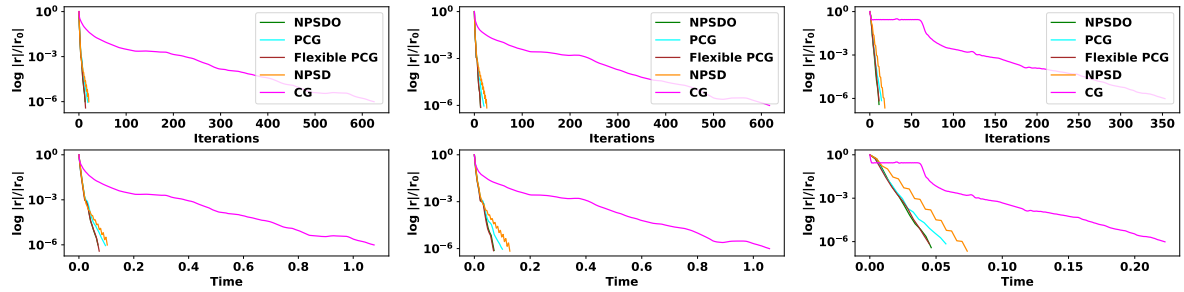


(j) Scooping, $(256, 256, 256)$ (k) Waterflow torus, $(256, 256, 256)$ (l) Waterflow ball, $(256, 256, 256)$

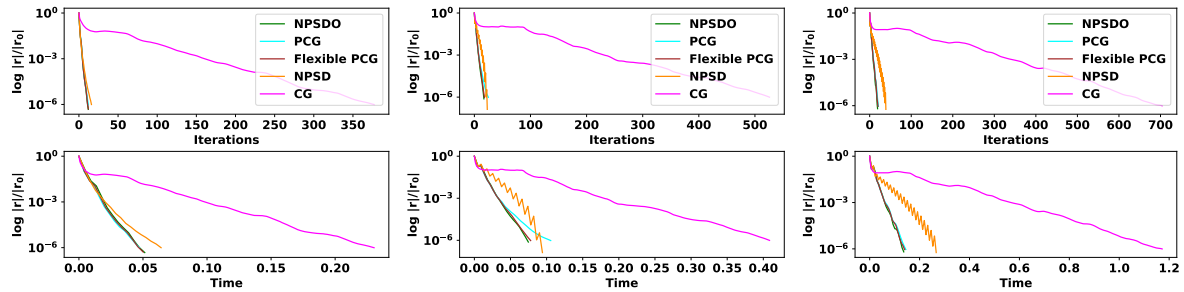
Figure 8. Comparisons among models with different depth $\mathcal{L}$ on 12 frames from all scenes at varying resolutions.

A.3. PCG Solver Variant Comparisons

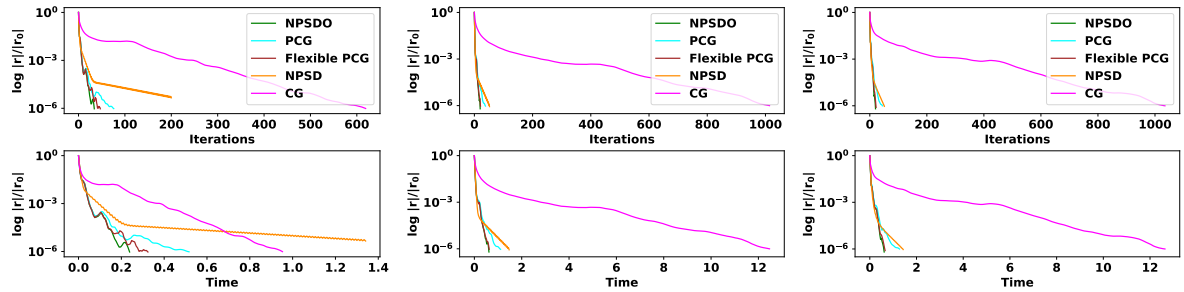
The following plots compare the performances of various iterative solvers preconditioned by $\mathcal{P}^{\text{net}}$. Statistics for unpreconditioned CG are also included for reference. While PCG and Flexible PCG both perform reasonably, PSDO achieves a modest speedup over them.



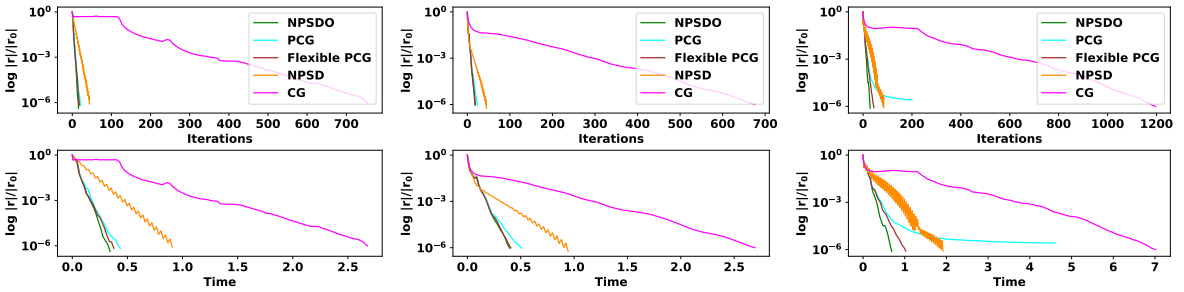
(a) Smoke solid, $(128, 128, 128)$ (b) smoke bunny, $(128, 128, 128)$ (c) Scooping, $(128, 128, 128)$



(d) Waterflow torus, $(128, 128, 128)$ (e) Waterflow ball, $(128, 128, 128)$ (f) Dambreak pillars, $(256, 128, 128)$



(g) Dambreak bunny, $(256, 128, 128)$ (h) Smoke solid, $(256, 256, 256)$ (i) Smoke bunny, $(256, 256, 256)$

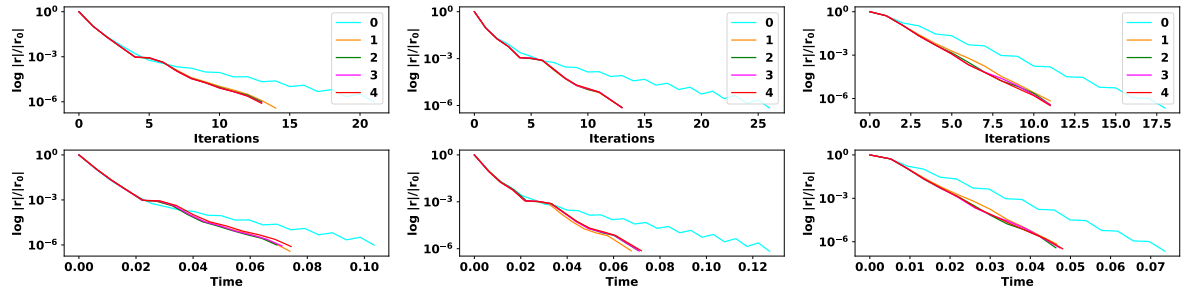


(j) Scooping, $(256, 256, 256)$ (k) Waterflow torus, $(256, 256, 256)$ (l) Waterflow ball, $(256, 256, 256)$

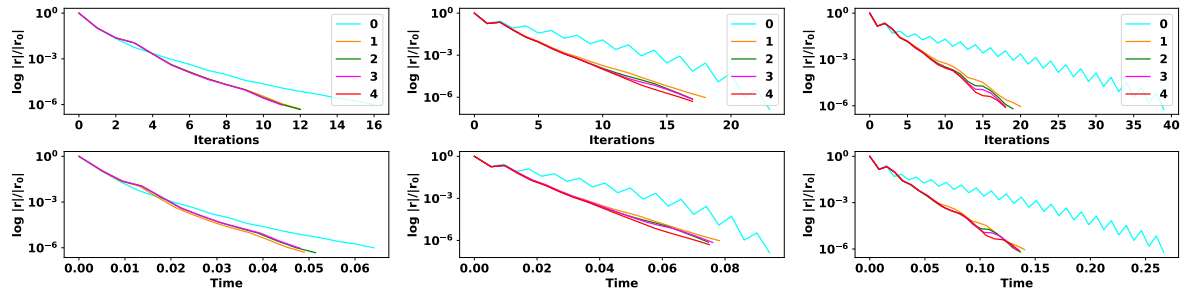
Figure 9. Performance statistics for several solver variants on 12 frames from all scenes

A.4. Number of Orthogonalizations

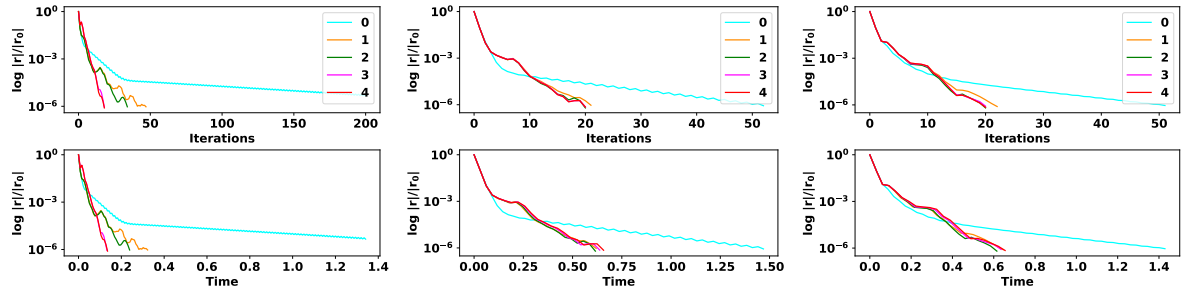
The following plots compare the performances of our NPSDO solver with different numbers of orthogonalization steps (n_{ortho} in Algorithm 1).



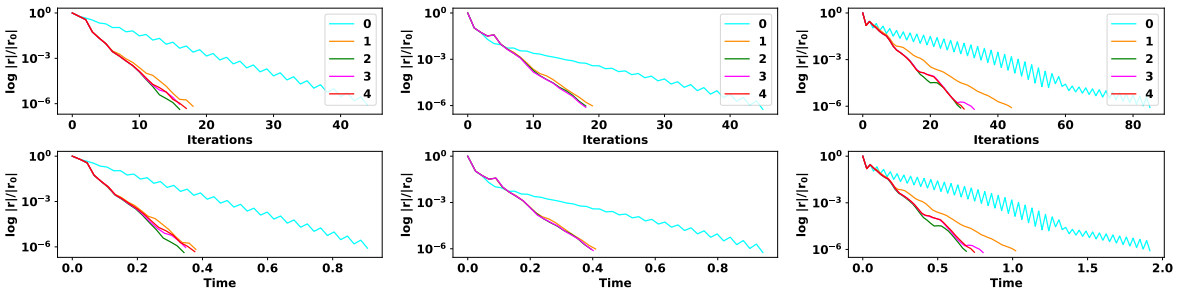
(a) Smoke solid, $(128, 128, 128)$ (b) smoke bunny, $(128, 128, 128)$ (c) Scooping, $(128, 128, 128)$



(d) Waterflow torus, $(128, 128, 128)$ (e) Waterflow ball, $(128, 128, 128)$ (f) Dambreak pillars, $(256, 128, 128)$



(g) Dambreak bunny, $(256, 128, 128)$ (h) Smoke solid, $(256, 256, 256)$ (i) Smoke bunny, $(256, 256, 256)$



(j) Scooping, $(256, 256, 256)$ (k) Waterflow torus, $(256, 256, 256)$ (l) Waterflow ball, $(256, 256, 256)$

Figure 10. Performance statistics for different n_{ortho} on 12 frames from all scenes

A.5. Timing Benchmarks

This section lists the average iteration count and runtime for each test case.

Table 3. Average iteration count and runtime across all frames for each test suite

	AMGCL		AMGX		IC		CG		Ours	
Examples	**Iteration**	**Time**	**Iteration**	**Time**	**Iteration**	**Time**	**Iteration**	**Time**	**Iteration**	**Time**
Smoke solid 128^3	11.2	0.696	44.9	0.324	196.1	0.852	612.5	1.050	13.3	0.072
Smoke solid 256^3	14.3	6.433	64.8	3.156	343.7	11.1	1076.8	13.245	20.5	0.636
Smoke bunny 128^3	11.3	0.713	43.7	0.319	200.5	0.862	615.8	1.048	14.0	0.075
Smoke bunny 256^3	14.3	6.046	63.7	3.119	345.7	10.9	1069.7	13.042	21.0	0.648
Scooping 128^3	12.2	0.170	34.6	0.097	128.1	0.270	392.8	0.234	11.8	0.051
Scooping 256^3	12.2	1.643	50.5	0.738	246.0	1.968	749.2	2.593	18.1	0.388
Waterflow torus 128^3	9.4	0.086	20.6	0.046	66.9	0.129	208.6	0.095	12.0	0.048
Waterflow torus 256^3	10.4	1.024	30.1	0.346	130.0	0.946	401.2	1.121	16.6	0.338
Waterflow ball 128^3	11.2	0.173	32.7	0.098	136.5	0.275	405.4	0.257	16.1	0.068
Waterflow ball 256^3	12.1	2.442	57.6	1.167	328.4	3.559	969.7	4.875	28.5	0.650
Dambreak pillars $256 \cdot 128^2$	11.2	0.476	39.9	0.229	167.7	0.625	523.2	0.704	14.9	0.103
Dambreak bunny $256 \cdot 128^2$	11.1	0.395	36.7	0.183	148.2	0.514	452.1	0.522	19.3	0.129
Average	11.7	1.691	43.3	0.819	203.2	2.667	623.1	3.232	17.2	0.267

A.6. Memory Usage

The following peak memory usage (in MiB) statistics were recorded with the command `nvidia-smi` on GeForce RTX 3080. Examples that threw out-of-memory error are marked as `NA`.

Table 4. Peak memory usage on a smoke simulation.

Resolution	**AMGCL**	**AMGX**	**IC**	**CG**	**NPSDO**	**DCDM**	**FN**
128^3	1248	2150	1668	1418	1548	8532	5170
256^3	5032	7370	8214	3716	4776	NA	NA

A.7. New Grid Sizes

The following plots demonstrate the ability of our solver to generalize to domains of different resolutions *without re-training the preconditioner*: it maintains consistently strong convergence behavior when applied to simulations with grid dimensions not seen during training. We note that 256^3 resolution simulations reported on in the main paper also were not present in the training set.

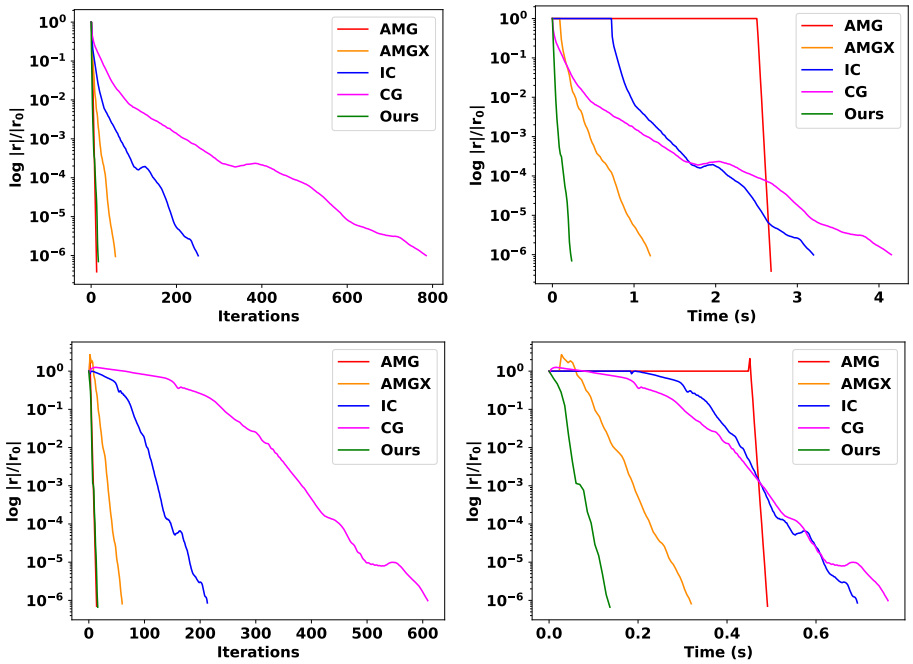


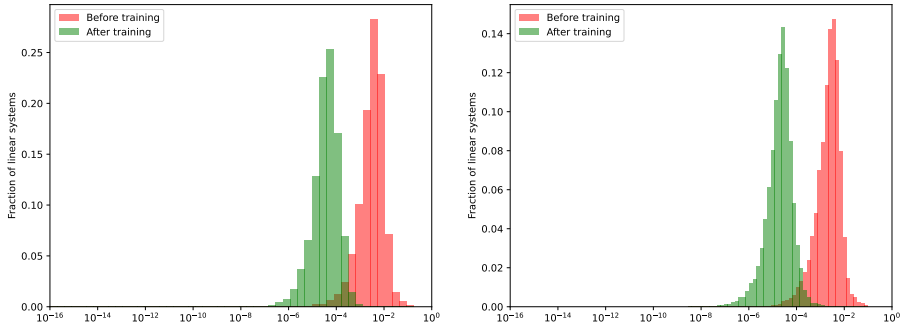
Figure 11. Convergence on a representative frame of a 192^3-resolution Neumann-only simulation (top pair) and a $(384, 128, 128)$-resolution mixed BC simulation (bottom pair).

A.8. Analysis of and Enforcement of Symmetry

To analyze the deviation of our preconditioner from symmetry, we applied our network to pairs of test vectors $\mathbf{x}$ and $\mathbf{y}$ and evaluated the following relative asymmetry measure:

$$\mathcal{S}(\mathbf{x}, \mathbf{y}; \boldsymbol{\mathcal{I}}) := \frac{\left|\mathbf{x} \cdot \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{y}) - \mathbf{y} \cdot \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{x})\right|}{\sqrt{\left|\mathbf{x} \cdot \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{x})\right| \left|\mathbf{y} \cdot \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{y})\right|}}.$$

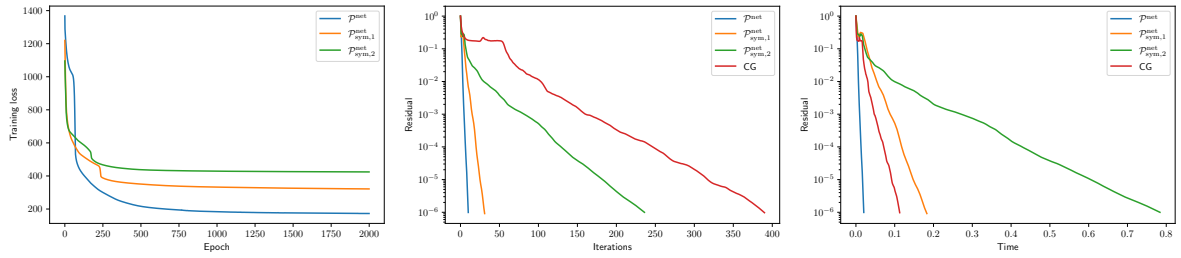
We did this using 100 test vector pairs for each system (image) in our training set, reporting the aggregate statistics in Figure 13(b) and Figure 13(c) for two different types of test vectors: (a) random pairs of vectors from the training set, and (b) fully random vectors from a normal distribution. Not only is the trained network's asymmetry small, it is several orders of magnitude smaller than that of the initial network, suggesting that training the network to approximate the inverse of the discrete Laplace operator (a symmetric matrix) naturally promotes symmetry.



(a) Random pairs of training RHS vectors. (b) Random vectors from normal distribution.

Figure 12. Histograms of relative symmetry metric $\mathcal{S}(\mathbf{x}, \mathbf{y}; \boldsymbol{\mathcal{I}})$ across 100 test vector pairs $(\mathbf{x}, \mathbf{y})$ for each image $\boldsymbol{\mathcal{I}}$ in the training set.

We furthermore constructed two variants of our network architecture that enforce symmetry and positive definiteness by construction. The first concatenates the $\boldsymbol{\mathcal{P}}^{\text{net}}$ architecture with its transpose $\boldsymbol{\mathcal{P}}^{\text{net}\top}$, obtaining $\boldsymbol{\mathcal{P}}^{\text{net}}_{\text{sym},1}(\boldsymbol{\mathcal{I}}, \mathbf{r}) := \boldsymbol{\mathcal{P}}^{\text{net}\top}(\boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{P}}^{\text{net}}(\boldsymbol{\mathcal{I}}, \mathbf{r}))$, while the second constrains the "pre-smoothing" and "post-smoothing" blocks to be transposes of each other via weight sharing, obtaining $\boldsymbol{\mathcal{P}}^{\text{net}}_{\text{sym},2}$. However, even when training on a single frame, both of these variants exhibit suboptimal performance: the training loss is not sufficiently reduced compared to our proposed model, and both symmetric networks perform worse in terms of wallclock time than unpreconditioned CG.



(a) Training loss (b) Residual vs. iterations (c) Residual vs. time

Figure 13. Comparison between our proposed model and two symmetric models, with baseline method unpreconditioned CG.

A.9. Capacity for Residual Reduction

To confirm that our solver can achieve high accuracy despite the single-precision arithmetic used in evaluating $\mathcal{P}^{\text{net}}$, we disabled the convergence test and ran for a fixed number of iterations (100), recording the final relative residual achieved for each system in our test set in Figure 14. The median relative residual is 4.01×10^{-15}.

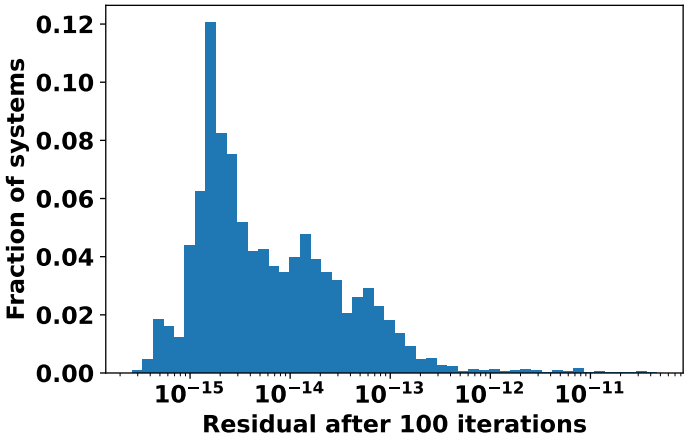


Figure 14. Histogram of the final relative residual achieved after running 100 iterations without a convergence test.