A Deep Conjugate Direction Method for Iteratively Solving Linear Systems

Ayano Kaneda* [1] **Osman Akar*** [2] **Jingyu Chen** [2] **Victoria Alicia Trevino Kala** [2] **David Hyde** [3] **Joseph Teran** [4]

Abstract

We present a novel deep learning approach to approximate the solution of large, sparse, symmetric, positive-definite linear systems of equations. Motivated by the conjugate gradients algorithm that iteratively selects search directions for minimizing the matrix norm of the approximation error, we design an approach that utilizes a deep neural network to accelerate convergence via data-driven improvement of the search direction at each iteration. Our method leverages a carefully chosen convolutional network to approximate the action of the inverse of the linear operator up to an arbitrary constant. We demonstrate the efficacy of our approach on spatially discretized Poisson equations, which arise in computational fluid dynamics applications, with millions of degrees of freedom. Unlike state-of-the-art learning approaches, our algorithm is capable of reducing the linear system residual to a given tolerance in a small number of iterations, independent of the problem size. Moreover, our method generalizes effectively to various systems beyond those encountered during training.

1. Introduction

The solution of large, sparse systems of linear equations is ubiquitous when partial differential equations (PDEs) are discretized to computationally simulate complex natural phenomena such as fluid flow (Losasso et al., 2006), thermodynamics (Chen et al., 2021), or mechanical fracture (Paluszny & Zimmerman, 2011). For linear systems arising from these diverse applications, we use the notation

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where the dimension n of the matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and the vector $\boldsymbol{b} \in \mathbb{R}^n$ correlates with spatial fidelity of the computational domain. Quality and realism of a simulation are proportional to this spatial fidelity; typical modern applications of numerical PDEs require solving linear systems with millions of unknowns. In such applications, numerical approximation to the solution of these linear systems is typically the bottleneck in overall performance; accordingly, practitioners have spent decades devising specialized algorithms for their efficient solution (Golub & Loan, 2012; Saad, 2003).

The appropriate numerical linear algebra technique depends on the nature of the problem. Direct solvers that utilize matrix factorizations (QR, Cholesky, etc. (Trefethen & Bau, 1997)) have optimal approximation error, but their computational cost is $O(n^3)$, and they typically require dense storage, even for sparse $\boldsymbol{A}$. Although Fast Fourier Transforms (Nussbaumer, 1981) can be used in limited instances (periodic boundary conditions, etc.), iterative techniques are most commonly adopted for sparse systems, which are typical for discretized PDEs. Many applications with strict performance constraints (e.g., real-time fluid simulation) utilize basic iterations (Jacobi, Gauss-Seidel, successive over relaxation (SOR), etc.) given limited computational budget (Saad, 2003). However, large approximation errors must be tolerated since iteration counts are limited by the performance constraints. This is particularly problematic since the wide elliptic spectrum of these matrices (a condition that worsens with increased spatial fidelity/matrix dimension) leads to poor conditioning and iteration counts. Iterative techniques can achieve sub-quadratic convergence if their iteration count does not grow excessively with problem size n since each iteration generally requires $O(n)$ floating point operations for sparse matrices. Discrete elliptic operators are typically symmetric positive (semi) definite, which means that the preconditioned conjugate gradients method (PCG) can be used to minimize iteration counts (Saad, 2003; Hestenes & Stiefel, 1952; Stiefel, 1952).

In the present work, we consider sparse linear systems that arise from discrete Poisson equations in incompressible flow applications (Chorin, 1967; Fedkiw et al., 2001; Bridson,

*Equal contribution [1]Department of Applied Physics, Waseda University, Tokyo, Japan [2]Department of Mathematics, University of California, Los Angeles, USA [3]Department of Computer Science, Vanderbilt University, Nashville, USA [4]Department of Mathematics, University of California, Davis, USA. Correspondence to: Ayano Kaneda <dizzy-miss-lizzy@moegi.waseda.jp>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

2008). These equations yield discrete elliptic operators, so PCG is the algorithm of choice for the associated linear systems; yet there is a subsequent question of which preconditioner to use. Preconditioners $\boldsymbol{P}$ for PCG must simultaneously: be symmetric positive definite (SPD) (and therefore admit factorization $\boldsymbol{P} = \boldsymbol{F}^2$), improve the condition number of the preconditioned system $\boldsymbol{FAFy} = \boldsymbol{Fb}$, and be computationally cheap to construct and apply; accordingly, designing specialized preconditioners for particular classes of problems is somewhat of an art. Incomplete Cholesky preconditioners (ICPCG) (Kershaw, 1978) use a sparse approximation to the Cholesky factorization and significantly reduce iteration counts; however, their inherent data dependency prevents efficient parallel implementation. Nonetheless, these are very commonly adopted for Poisson equations arising in incompressible flow (Fedkiw et al., 2001; Bridson, 2008). Multigrid (Brandt, 1977) and domain decomposition (Saad, 2003) preconditioners greatly reduce iterations counts, but they must be updated (with non-trivial cost) each time the problem changes (e.g., in computational domains with time-varying boundaries) and/or for different hardware platforms. In general, choice of an optimal preconditioner for discrete elliptic operators is an open area of research.

Recently, data-driven approaches that leverage deep learning techniques have shown promise for solving linear systems. Various researchers have investigated machine learning estimation of multigrid parameters (Greenfeld et al., 2019; Grebhahn et al., 2016; Luz et al., 2020). Others have developed machine learning methods to estimate preconditioners (Götz & Anzt, 2018; Stanaityte, 2020; Ichimura et al., 2020) and initial guesses for iterative methods (Luna et al., 2021; Um et al., 2020; Ackmann et al., 2020). Tompson et al. (2017) and Yang et al. (2016) develop *non-iterative* machine learning approximations of the inverse of discrete Poisson equations from incompressible flow.

This paper develops a novel conjugate gradients-style iterative method, enabled by deep learning, for approximating the solution of SPD linear systems, which we call the deep conjugate direction method (DCDM). CG iteratively adds $\boldsymbol{A}$-conjugate search directions while minimizing the matrix norm of the error. We instead use a convolutional neural network (CNN) as an approximation of the inverse of the matrix in order to generate more efficient search directions. We only ask that our network approximate the inverse up to an unknown scaling since this decreases the degree of nonlinearity and since it does not affect the quality of the search direction (which is scale independent). The network is similar to a preconditioner, but it is not a linear function, and our DCDM method is designed to accommodate this nonlinearity. We use self-supervised learning to train our network with a loss function equal to the L^2 difference between an input vector and a scaling of $\boldsymbol{A}$ times the output of our network. To account for this unknown scaling during training, we choose the scale of the output of the network by minimizing the matrix norm of the error. Our approach allows for efficient training and generalization to problems unseen (new matrices $\boldsymbol{A}$ and new right-hand sides $\boldsymbol{b}$). We benchmark our algorithm using the ubiquitous pressure Poisson equation (discretized on regular voxelized domains) and compare against FluidNet (Tompson et al., 2017), which is the state-of-the-art learning-based method for these types of problems.

DCDM can be viewed as an improved version of Tompson et al. (2017), because unlike the non-iterative approaches of Tompson et al. (2017) and Yang et al. (2016), our method can reduce the linear system residuals *arbitrarily*. We showcase our approach with examples that have over 16 million degrees of freedom.

2. Related Work

Several papers have focused on enhancing the solution of linear systems (arising from discretized PDEs) using learning. For instance, Götz & Anzt (2018) generate sparsity patterns for block-Jacobi preconditioners using convolutional neural networks, and Stanaityte (2020) use a CNN to predict non-zero patterns for ILU-type preconditioners for the Navier-Stokes equations (though neither work designs fundamentally new preconditioners). Ichimura et al. (2020) develop a neural-network based preconditioner where the network is used to predict approximate Green's functions (which arise in the analytical solution of certain PDEs) that in turn yield an approximate inverse of the linear system. Hsieh et al. (2019) learn an iterator that solves linear systems, performing competitively with classical solvers like multigrid-preconditioned MINRES (Paige & Saunders, 1975). Luz et al. (2020) and Greenfeld et al. (2019) use machine learning to estimate algebraic multigrid (AMG) parameters. They note that AMG approaches rely most fundamentally on effectively chosen (problem-dependent) prolongation sparse matrices and that numerous methods have attempted to automatically create them from the matrix $\boldsymbol{A}$. They train a graph neural network to learn (in an unsupervised fashion) a mapping from matrices $\boldsymbol{A}$ to prolongation operators. Grebhahn et al. (2016) note that geometric multigrid solver parameters can be difficult to choose to guarantee parallel performance on different hardware platforms. They use machine learning to create a code generator to help achieve this.

Several works consider accelerating the solution of linear systems by learning an initial guess that is close to the true solution or otherwise helpful to descent algorithms for finding the true solution. In order to solve the discretized Poisson equation, Luna et al. (2021) accelerate the convergence of GMRES (Saad & Schultz, 1986) with an initial

guess that is learned in real-time (i.e., as a simulation code runs) with no prior data. Um et al. (2020) train a network (incorporating differentiable physics, based on the underlying PDEs) in order to produce high-quality initial guesses for a CG solver. In a somewhat similar vein, Ackmann et al. (2020) use a simple feedforward neural network to predict pointwise solution components, which accelerates the conjugate residual method used to solve a relatively simple shallow-water model (a more sophisticated network and loss function are needed to handle more general PDEs and larger-scale problems).

At least two papers (Ruelmann et al., 2018; Sappl et al., 2019) have sought to learn a mapping between a matrix and an associated sparse approximate inverse. In their investigation, Ruelmann et al. (2018) propose training a neural network using matrix-inverse pairs as training data. Although straightforward to implement, the cost of generating training data, let alone training the network, is prohibitive for large-scale 3D problems. Sappl et al. (2019) seek to learn a mapping between linear system matrices and sparse (banded) approximate inverses. Their loss function is the condition number of the product of the system matrix and the approximate inverse; the minimum value of the condition number is one. Although this framework is quite simple, evaluating the condition number of a matrix is asymptotically costly ($O(n^3)$), and in general, the inverse of a sparse matrix can be quite dense. Accordingly, the method is not efficient or accurate enough for the large-scale 3D problems that arise in real-world engineering problems.

DCDM can also be viewed as a novel learning to optimize (L2O) method. L2O methods use learning to devise continuous optimization algorithms; for example, Andrychowicz et al. (2016) learn a gradient descent algorithm, Li & Malik (2016) provide a general reinforcement learning framework for learning optimization algorithms, Shen et al. (2019) apply L2O to minimax problems, and Liao et al. (2022) perform online meta-learning of quasi-Newton optimization methods. We refer the reader to Chen et al. (2022) for a recent review of L2O techniques.

Most relevant to the present work is FluidNet (Tompson et al., 2017). FluidNet uses a highly-tailored CNN architecture to predict the solution of a linear projection operation (specifically, for the discrete Poisson equation) given a matrix and right-hand side. The authors demonstrate fluid simulations where the linear solve is replaced by evaluating their network. Because their network is relatively lightweight and is only evaluated once per time step, their simulations run efficiently. However, their design allows the network only one opportunity to reduce the residual for the linear solve; in practice, we observe that FluidNet is able to reduce the residual by no more than about one order of magnitude. However, in computer graphics applications, at least four orders of magnitude in residual reduction are usually required for visual fidelity, while in scientific and engineering applications, practitioners prefer solutions that reduce the residual by eight or more orders of magnitude (i.e., to within machine precision). Accordingly, FluidNet's lack of convergence stands in stark contrast to classical, convergent methods like CG. Our method resolves this gap.

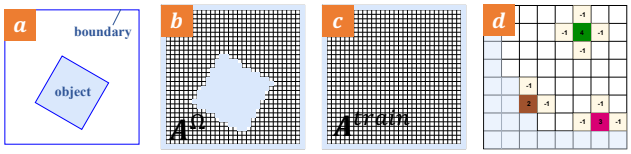


Figure 1. **(a)** We illustrate a sample flow domain $\Omega \subset (0,1)^2$ (in 2D for ease of illustration) with internal boundaries (blue lines). **(b)** We voxelize the domain with a regular grid: white cells represent interior/fluid, and blue cells represent boundary conditions. **(c)** We train using the matrix $\boldsymbol{A}^{\text{train}}$ from a discretized domain with *no* interior boundary conditions, where d is the dimension. This creates linear system with $n = (n_c + 1)^d$ unknowns, where n_c is the number of grid cells on each direction. **(d)** We illustrate the non-zero entries in an example matrix $\boldsymbol{A}^{\Omega}$ from the voxelized and labeled (white vs. blue) grid for three example interior cells (green, magenta, and brown). Each case illustrates the non-zero entries in the row associated with the example cell. All entries of $\boldsymbol{A}^{\Omega}$ in rows corresponding to boundary/blue cells are zero.

3. Motivation: Incompressible Flow

We demonstrate the efficacy of our approach with the linear systems that arise in incompressible flow applications. Specifically, we use our algorithm to solve the Poisson equation discretized on a regular grid, following the pressure projection equations that arise in Chorin's splitting technique (Chorin, 1967) for the inviscid, incompressible Euler equations. These equations are

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}\boldsymbol{u}\right) + \nabla p = \boldsymbol{f}^{ext}, \qquad \nabla \cdot \boldsymbol{u} = 0 \tag{2}$$

where $\boldsymbol{u}$ is fluid velocity, p is pressure, ρ is density, and $\boldsymbol{f}^{ext}$ accounts for external forces like gravity. The equations are assumed at all positions $\boldsymbol{x}$ in the spatial fluid flow domain Ω and for time $t > 0$. The first equation in Equation 2 enforces conservation of momentum in the absence of viscosity, and the second enforces incompressibility and conservation of mass. These equations are subject to initial conditions $\rho(\boldsymbol{x}, 0) = \rho^0$ and $\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}^0(\boldsymbol{x})$, as well as boundary conditions $\boldsymbol{u}(\boldsymbol{x}, t) \cdot \boldsymbol{n}(\boldsymbol{x}) = u^{\partial\Omega}(\boldsymbol{x}, t)$ on the boundary of the domain $\boldsymbol{x} \in \partial\Omega$ (where $\boldsymbol{n}$ is the unit outward pointing normal at position $\boldsymbol{x}$ on the boundary).

Equation 2 is discretized in both time and space. Temporally, we split the advection $\frac{\partial \boldsymbol{u}}{\partial t} + \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}\boldsymbol{u} = 0$ and body forces

terms $\rho\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{f}^{ext}$, and finally enforce incompressibility via the pressure projection $\frac{\partial \boldsymbol{u}}{\partial t} + \frac{1}{\rho}\nabla p = \boldsymbol{0}$ such that $\nabla \cdot \boldsymbol{u} = 0$; this is the standard advection-projection scheme proposed by Chorin (1967). Using finite differences in time, we can summarize this as

$$\rho^0 \left(\frac{\boldsymbol{u}^* - \boldsymbol{u}^n}{\Delta t} + \frac{\partial \boldsymbol{u}^n}{\partial \boldsymbol{x}} \boldsymbol{u}^n \right) = \boldsymbol{f}^{ext} \tag{3}$$

$$-\nabla \cdot \frac{1}{\rho^0} \nabla p^{n+1} = -\nabla \cdot \boldsymbol{u}^* \tag{4}$$

$$-\frac{1}{\rho^0} \nabla p^{n+1} \cdot \boldsymbol{n} = \frac{1}{\Delta t} \left(u^{\partial\Omega} - \boldsymbol{u}^* \cdot \boldsymbol{n} \right). \tag{5}$$

For the spatial discretization, we use a regular marker-and-cell (MAC) grid (Harlow & Welch, 1965) with cubic voxels whereby velocity components are stored on the face of voxel cells, and scalar quantities (e.g., pressure p or density ρ) are stored at voxel centers. We use backward semi-Lagrangian advection (Fedkiw et al., 2001; Gagniere et al., 2020) for Equation 3. All spatial partial derivatives are approximated using finite differences. Equations 4 and 5 describe the pressure Poisson equation with Neumann conditions on the boundary of the flow domain. We discretize the left-hand side of Equation 4 using a standard 7-point finite difference stencil. The right-hand side is discretized using the MAC grid discrete divergence finite difference stencils as well as contributions from the boundary condition terms in Equation 5. We refer the reader to Bridson (2008) for more in-depth implementation details. Equation 5 is discretized by modifying the Poisson stencil to enforce Neumann boundary conditions. We do this using a simple labeling of the voxels in the domain. For simplicity, we assume $\Omega \subset (0,1)^3$ is a subset of the unit cube, potentially with internal boundaries. We label cells in the domain as either liquid or boundary. This simple classification is enough to define the discrete Poisson operators (with appropriate Neumann boundary conditions at domain boundaries) that we focus on in the present work; we illustrate the details in Figure 1.

We use the following notation to denote the discrete Poisson equations associated with Equations 4–5:

$$\boldsymbol{A}^\Omega \boldsymbol{x} = \boldsymbol{b}^{\nabla \cdot \boldsymbol{u}^*} + \boldsymbol{b}^{u^{\partial\Omega}}, \tag{6}$$

where $\boldsymbol{A}^\Omega$ is the discrete Poisson matrix associated with the voxelized domain, $\boldsymbol{x}$ is the vector of unknown pressure, and $\boldsymbol{b}^{\nabla \cdot \boldsymbol{u}^*}$ and $\boldsymbol{b}^{u^{\partial\Omega}}$ are the right-hand side terms from Equations 4 and 5, respectively. $\boldsymbol{A}^\Omega$ in Equation 6, is a large, sparse, SPD linear system. The computational complexity of solving Equation 6 strongly depends on data (e.g., internal boundary conditions in the flow domain, see Figure 1).

We define a special case of the matrix involved in this discretization to be the Poisson matrix $\boldsymbol{A}^{\text{train}}$ associated with $\Omega = (0,1)^3$, i.e., a full fluid domain with no internal boundaries. We use this matrix for training, yet demonstrate that our network generalizes to all other matrices arising from more complicated flow domains. To be clear, the implication of this is that by training DCDM *one time*—which we have already done, and we release our pre-trained models and source code along with this paper—practitioners can immediately apply DCDM to *any* Poisson system (regardless of internal boundary conditions, etc.). Although there is a clear limitation that we only train our network to solve Poisson problems, this is a major advantage over state-of-the-art methods like FluidNet (Tompson et al., 2017), which require highly diverse training data (matrices from many fluid simulations, all with different types of obstacles and boundary conditions) in order to train a network with sufficient generalization; we only ever leverage a single training matrix (i.e., a single set of boundary conditions) $\boldsymbol{A}^{\text{train}}$.

4. Deep Conjugate Direction Method

We present our method for the deep learning acceleration of iterative approximations to the solution of linear systems of the form seen in Equation 6. We first briefly discuss relevant details of search direction methods, particularly the choice of line search directions[1]. We then present a deep learning technique for improving the quality of these search directions that ultimately reduces iteration counts required to achieve satisfactory residual reduction. Lastly, we outline the training procedures for our deep CNN.

Our approach iteratively improves approximations to the solution $\boldsymbol{x}$ of Equation 6. We build on the method of CG, which requires the matrix $\boldsymbol{A}^\Omega$ in Equation 6 to be SPD. SPD matrices $\boldsymbol{A}^\Omega$ give rise to the matrix norm $\|\boldsymbol{y}\|_{\boldsymbol{A}^\Omega} = \sqrt{\boldsymbol{y}^T \boldsymbol{A}^\Omega \boldsymbol{y}}$. CG can be derived in terms of iterative line search improvement based on optimality in this norm. That is, an iterate $\boldsymbol{x}_{k-1} \approx \boldsymbol{x}$ is updated along search direction $\boldsymbol{d}_k$ by a step size α_k that is chosen to minimize the matrix norm of the error between the updated iterate and $\boldsymbol{x}$:

$$\begin{aligned} \alpha_k &= \arg\min_\alpha \frac{1}{2} \left\| \boldsymbol{x} - (\boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}_k) \right\|_{\boldsymbol{A}^\Omega}^2 \\ &= \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A}^\Omega \boldsymbol{d}_k}, \end{aligned} \tag{7}$$

where $\boldsymbol{r}_{k-1} = \boldsymbol{b} - \boldsymbol{A}^\Omega \boldsymbol{x}_{k-1}$ is the $(k-1)^{\text{th}}$ residual (see Appendix A.2 for details). Different search directions $\boldsymbol{d}_k$ result in different algorithms. A natural choice is the negative gradient of the matrix norm of the error (evaluated at the current iterate), $\boldsymbol{d}_k = -\frac{1}{2}\nabla \left\| \boldsymbol{x}_{k-1} \right\|_{\boldsymbol{A}^\Omega}^2 = \boldsymbol{r}_{k-1}$, since this will point in the direction of steepest decrease. This is the gradient descent method (GD). Unfortunately, this approach requires many iterations in practice. CG modifies GD into a

[1] For a comprehensive background on CG, see Appendix A.1.

more effective strategy by instead choosing directions that are $\boldsymbol{A}$-orthogonal (i.e., $\boldsymbol{d}_i^T \boldsymbol{A}^\Omega \boldsymbol{d}_j = 0$ for $i \neq j$). More precisely, the search direction $\boldsymbol{d}_k$ is chosen as follows:

$$\boldsymbol{d}_k = \boldsymbol{r}_{k-1} - \sum_{i=1}^{k-1} h_{ik} \boldsymbol{d}_i, \qquad h_{ik} = \frac{\boldsymbol{d}_i^T \boldsymbol{A}^\Omega \boldsymbol{r}_{k-1}}{\boldsymbol{d}_i^T \boldsymbol{A}^\Omega \boldsymbol{d}_i},$$

which guarantees A-orthogonality. The magic of CG is that $h_{ik} = 0$ for $i < k-1$, hence this iteration can be performed without the need to store all previous search directions $\boldsymbol{d}_i$ and without the need for computing all previous h_{ik}.

While the residual is a natural choice for generating $\boldsymbol{A}$-orthogonal search directions (since it points in the direction of the steepest local decrease), it is not the optimal search direction. Optimality is achieved when $\boldsymbol{d}_k$ is parallel to $(\boldsymbol{A}^\Omega)^{-1}\boldsymbol{r}_{k-1}$, whereby $\boldsymbol{x}_k$ will be equal to $\boldsymbol{x}$ since α_k (computed from Equation 7) will step directly to the solution. We can see this by considering the residual and its relation to the search direction:

$$\begin{aligned}\boldsymbol{r}_k &= \boldsymbol{b} - \boldsymbol{A}^\Omega \boldsymbol{x}_k = \boldsymbol{b} - \boldsymbol{A}^\Omega \boldsymbol{x}_{k-1} - \alpha_k \boldsymbol{A}^\Omega \boldsymbol{d}_k \\ &= \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{A}^\Omega \boldsymbol{d}_k.\end{aligned}$$

In light of this, we use deep learning to create an approximation $\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r})$ to $(\boldsymbol{A}^\Omega)^{-1}\boldsymbol{r}$, where $\boldsymbol{c}$ denotes the network weights and biases. This is analogous to using a preconditioner in PCG; however, our network is not SPD (nor even a linear function). We simply use this data-driven approach as our means of generating better search directions $\boldsymbol{d}_k$. Furthermore, we only need to approximate a vector parallel to $(\boldsymbol{A}^\Omega)^{-1}\boldsymbol{r}$ since the step size α_k will account for any scaling in practice. In other words, $\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}) \approx s_{\boldsymbol{r}} (\boldsymbol{A}^\Omega)^{-1}\boldsymbol{r}$, where the scalar $s_{\boldsymbol{r}}$ is not defined globally; it only depends on $\boldsymbol{r}$, and the model does not learn it. Lastly, as with CG, we enforce $\boldsymbol{A}$-orthogonality, yielding search directions

$$\boldsymbol{d}_k = \boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}_{k-1}) - \sum_{i=1}^{k-1} h_{ik} \boldsymbol{d}_i, h_{ik} = \frac{\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}_{k-1})^T \boldsymbol{A}^\Omega \boldsymbol{d}_i}{\boldsymbol{d}_i^T \boldsymbol{A}^\Omega \boldsymbol{d}_i}.$$

We summarize our approach in Algorithm 1. Note that we introduce the variable i_{start}. To guarantee $\boldsymbol{A}$-orthogonality between all search directions, we must have $i_{\text{start}} = 1$. However, this requires storing all prior search directions, which can be costly. We found that using $i_{\text{start}} = k-2$ worked nearly as well as $i_{\text{start}} = 1$ in practice (in terms of our ability to iteratively reduce the residual of the system). We demonstrate this in Figure 4c.

Algorithm 1 DCDM

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}^\Omega \boldsymbol{x}_0$
$k = 1$
while $\|\boldsymbol{r}_{k-1}\| \geq \epsilon$ **do**
  $\boldsymbol{d}_k = \boldsymbol{f}(\boldsymbol{c}, \frac{\boldsymbol{r}_{k-1}}{\|\boldsymbol{r}_{k-1}\|})$
  for $i_{\text{start}} \leq i < k$ **do**
    $h_{ik} = \frac{\boldsymbol{d}_k^T \boldsymbol{A}^\Omega \boldsymbol{d}_i}{\boldsymbol{d}_i^T \boldsymbol{A}^\Omega \boldsymbol{d}_i}$
    $\boldsymbol{d}_k$-=$h_{ik}\boldsymbol{d}_i$
  end for
  $\alpha_k = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A}^\Omega \boldsymbol{d}_k}$
  $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{d}_k$
  $\boldsymbol{r}_k = \boldsymbol{b} - \boldsymbol{A}^\Omega \boldsymbol{x}_k$
  $k = k + 1$
end while

5. Model Architecture, Datasets, and Training

Efficient performance of our method requires effective training of our deep convolutional network for weights and biases $\boldsymbol{c}$ such that $\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}) \approx s_{\boldsymbol{r}} (\boldsymbol{A}^\Omega)^{-1}\boldsymbol{r}$ (for arbitrary scalar $s_{\boldsymbol{r}}$). We design a model architecture, loss function, and self-supervised training approach to achieve this. Our approach has modest training requirements and allows for effective residual reduction while generalizing well to problems not seen in the training data.

5.1. Loss Function and Self-supervised Learning

Although we generalize to arbitrary matrices $\boldsymbol{A}^\Omega$ from Equation 6 that correspond to domains $\Omega \subset (0,1)^3$ that have internal boundaries (see Figure 1), we train using just the matrix $\boldsymbol{A}^{\text{train}}$ from the full cube domain $(0,1)^3$. "the full cube domain $(0,1)^3$" is just the unit cube discretized on regular intervals, see e.g. Figure 1(c).

In contrast, other similar approaches (Tompson et al., 2017; Yang et al., 2016) train using matrices $\boldsymbol{A}^\Omega$ and right-hand sides $\boldsymbol{b}^{\nabla \cdot \boldsymbol{u}^*} + \boldsymbol{b}^{u^{\partial\Omega}}$ that arise from flow in many domains with internal boundaries. We train our network by minimizing the L^2 difference $\|\boldsymbol{r} - \alpha \boldsymbol{A}^{\text{train}} \boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r})\|_2$, where $\alpha = \frac{\boldsymbol{r}^T \boldsymbol{f}(\boldsymbol{c},\boldsymbol{r})}{\boldsymbol{f}(\boldsymbol{c},\boldsymbol{r})^T \boldsymbol{A}^{\text{train}} \boldsymbol{f}(\boldsymbol{c},\boldsymbol{r})}$ from Equation 7. This choice of α accounts for the unknown scaling in the approximation of $\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r})$ to $\left(\boldsymbol{A}^{\text{train}}\right)^{-1} \boldsymbol{r}$. We use a self-supervised approach and train the model by minimizing

$$\text{Loss}(\boldsymbol{f}, \boldsymbol{c}, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{r} \in \mathcal{D}} \|\boldsymbol{r} - \frac{\boldsymbol{r}^T \boldsymbol{f}(\boldsymbol{c},\boldsymbol{r})}{\boldsymbol{f}(\boldsymbol{c},\boldsymbol{r})^T \boldsymbol{A}^{\text{train}} \boldsymbol{f}(\boldsymbol{c},\boldsymbol{r})} \boldsymbol{A}^{\text{train}} \boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r})\|_2$$

for a given dataset $\mathcal{D}$ consisting of training vectors $\boldsymbol{b}^i$. In Algorithm 1, the normalized residuals $\frac{\boldsymbol{r}_k}{\|\boldsymbol{r}_k\|}$ are passed as inputs to the model. Unlike in e.g. FluidNet (Tompson et al., 2017), only the first residual $\frac{\boldsymbol{r}_0}{\|\boldsymbol{r}_0\|}$ is directly related to the problem-dependent original right-hand side $\boldsymbol{b}$. Hence we consider a broader range of training vectors than those

expected in a given problem of interest, e.g., incompressible flows. We observe that generally the residuals $\boldsymbol{r}_k$ in Algorithm 1 are skewed to the lower end of the spectrum of the matrix $\boldsymbol{A}^{\Omega}$. Since $\boldsymbol{A}^{\Omega}$ is a discretized elliptic operator, lower end modes are of lower frequency of spatial oscillation. We create our training vectors $\boldsymbol{b}^i \in \mathcal{D}$ using $m \ll n$ approximate eigenvectors of the training matrix $\boldsymbol{A}^{\text{train}}$. We use the Rayleigh-Ritz method to create approximate eigenvectors $\boldsymbol{q}_i$, $0 \leq i < m$. This approach allows us to effectively approximate the full spectrum of $\boldsymbol{A}^{\text{train}}$ without computing the full eigendecomposition, which can be expensive ($O(n^3)$) at high resolution. Note that generating the dataset has $O(m^2 N)$ complexity, N being the resolution (e.g., 64^3 or 128^3), due to reorthogonalization of Lanczos vectors (see Appendix A.4). Hence we tried values like m = 1K, 5K, 10K, and 20K, and chose the smallest value (m = 10,000) that gave a viable model after training.

The Rayleigh-Ritz vectors are orthonormal and satisfy $\boldsymbol{Q}_m^T \boldsymbol{A}^{\text{train}} \boldsymbol{Q}_m = \boldsymbol{\Lambda}_m$, where $\boldsymbol{\Lambda}_m$ is a diagonal matrix with nondecreasing diagonal entries λ_i referred to as Ritz values (approximate eigenvalues) and $\boldsymbol{Q}_m = [\boldsymbol{q}_0, \boldsymbol{q}_1, \ldots, \boldsymbol{q}_{m-1}] \in \mathbb{R}^{n \times m}$. We pick $\boldsymbol{b}^i = \frac{\sum_{j=0}^{m-1} c_j^i \boldsymbol{q}_j}{\left\| \sum_{j=0}^{m-1} c_j^i \boldsymbol{q}_j \right\|}$, where the coefficients c_j^i are picked from a standard normal distribution

$$c_j^i = \begin{cases} 9 \cdot \mathcal{N}(0,1) & \text{if } \tilde{j} \leq j \leq \frac{m}{2} + \theta \\ \mathcal{N}(0,1) & \text{otherwise} \end{cases}$$

where θ is a small number (we used $\theta = 500$), and $\tilde{j}$ is the first index that $\lambda_{\tilde{j}} > 0$. This choice creates 90% of $\boldsymbol{b}^i$ from the lower end of the spectrum, with the remaining 10% from the higher end. The Riemann-Lebesgue Lemma states the Fourier spectrum of a continuous function will decay at infinity, so this specific choice of $\boldsymbol{b}_i$'s is reasonable for the training set. In practice, we also observed that the right-hand sides of the pressure system that arose in flow problems (in the empty domain) tended to be at the lower end of the spectrum. Notably, even though this dataset only uses Rayleigh-Ritz vectors from the training matrix $\boldsymbol{A}^{\text{train}}$, our network can be effectively generalized to flows in irregular domains, e.g., smoke flowing past a rotating box and flow past a bunny (see Figure 3).

We generate the Rayleigh-Ritz vectors by first tridiagonalizing the training matrix $\boldsymbol{A}^{\text{train}}$ with m Lanczos iterations (Lanczos, 1950) to form $\boldsymbol{T}^m = {\boldsymbol{Q}_m^L}^T \boldsymbol{A}^{\text{train}} \boldsymbol{Q}_m^L \in \mathbb{R}^{m \times m}$. We then diagonalize $\boldsymbol{T}^m = \hat{\boldsymbol{Q}}^T \boldsymbol{\Lambda}_m \hat{\boldsymbol{Q}}$. While asymptotically costly, we note that this algorithm is performed on the comparably small $m \times m$ matrix $\boldsymbol{T}^m$ (rather than on the $\boldsymbol{A}^{\text{train}} \in \mathbb{R}^{n \times n}$). This yields the Rayleigh-Ritz vectors as the columns of $\boldsymbol{Q}_m = \boldsymbol{Q}_m^L \hat{\boldsymbol{Q}}$. The Lanczos vectors are the columns of the matrix $\boldsymbol{Q}_m^L$ and satisfy a three-term recurrence whereby the next Lanczos vector can be iteratively

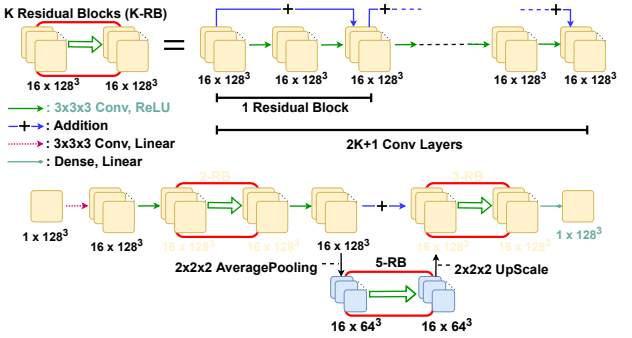


Figure 2. Architecture for training with $\boldsymbol{A}^{\text{train}}$ on a 128^3 grid.

computed from previous two as

$$\beta_j \boldsymbol{q}_{j+1}^L = \boldsymbol{A}^{\text{train}} \boldsymbol{q}_j^L - \beta_{j-1} \boldsymbol{q}_{j-1}^L - \alpha_j \boldsymbol{q}_j^L,$$

where α_j and β_j are diagonal and subdiagonal entries of $\boldsymbol{T}^k$. β_j is computed so that $\boldsymbol{q}_{j+1}^L$ is a unit vector, and $\alpha_{j+1} = \boldsymbol{q}_{j+1}^T \boldsymbol{A}^{\text{train}} \boldsymbol{q}_{j+1}$. We initialize the iteration with a random $\boldsymbol{q}_0^L \in \text{span}(\boldsymbol{A}^{\text{train}})$. The Lanczos algorithm can be viewed as a modified Gram-Schmidt technique to create an orthonormal basis for the Krylov space associated with $\boldsymbol{q}_0^L$ and $\boldsymbol{A}^{\text{train}}$, and it therefore suffers from rounding error sensitivities manifested as loss of orthonormality with vectors that do not appear in the recurrence. We found that the simple strategy described in Paige (1971) of orthogonalizing each iterate with respect to all previous Lanczos vectors to be sufficient for our training purposes. Dataset creation takes 5–7 hours for a 64^3 computational grid, and 2–2.5 days for a 128^3 grid (see Appendix A.4 for more detail).

We reiterate that since DCDM generalizes to various Poisson systems (see Sections 5.2 and 6) despite only using data corresponding to an empty fluid domain, practitioners do not need to generate new data in order to apply our method. Moreover, we show in the examples that it is possible to use trained model weights from a lower-resolution grid for higher-resolution problems, so practitioners may not need to generate new data even if running problems at different resolutions than what we consider.

5.2. Model Architecture

The internal structure of our CNN architecture for a 128^3 grid is shown in Figure 2. It consists of a series of convolutional layers with residual connections. The upper left of Figure 2 (K Residual Blocks) shows our use of multiple blocks of residually connected layers. Notably, within each block, the first layer directly affects the last layer with an addition operator. All non-input or output convolutions use a $3 \times 3 \times 3$ filter, and all layers consist of 16 feature maps. In the middle of the first level, a layer is downsampled (via the average pooling operator with $(2 \times 2 \times 2)$ pool size)

and another set of convolutional layers is applied with residual connection blocks. The last layer in the second level is upscaled and added to the layer that is downsampled. The last layer in the network is dense with a identity function. The activation functions in all convolutional layers are ReLU, except for the first convolution, which uses a linear activation function.

Initially we tried a simple deep feedforward convolutional network with residual connections (motivated by He et al. (2016)). Although such a simple model works well for DCDM, it requires a high number of layers, which results in higher training and inference times. We found that creating parallel layers of CNNs with downsampling reduced the number of layers required. In summary, our goal was to first identify the simplest network architecture that provided adequate accuracy for our target problems, and subsequently, we sought to make architectural changes to minimize training and inference time. We are interested in a more thorough investigation of potential network architectures, filter sizes, etc., to better characterize the tradeoff curves between accuracy and efficiency; as a first step in this direction, we included a brief ablation study in Appendix A.4.

Differing resolutions use differing numbers of convolutions, but the fundamental structure remains the same. More precisely, the number of residual connections is changed for different resolutions. For example, a 64^3 grid uses one residual block on the left, two on the right on the upper level, and three on the lower level. Furthermore, the weights trained on a lower resolution grid can be used effectively with higher resolutions. Figure 4d shows convergence results for a 256^3 grid, using a model trained for a 64^3 grid and a 128^3 grid. The model that we use for 256^3 grids in our final examples was trained on a 128^3 grid; however, as the shown in the figure, even training with a 64^3 grid allows for efficient residual reduction. Table 1 shows results for three different resolutions, where DCDM uses 64^3 and 128^3 trained models. Since we can use the same weights trained over a 64^d domain and/or 128^d domain, the number of parameters does not depend on the spatial fidelity. It depends on d for the kernel size.

5.3. Training

Using the procedure explained in Section 5.1, we create the training dataset $\mathcal{D} \in \text{span}(\boldsymbol{A}^{\text{train}}) \cap \mathcal{S}^{n-1}$ of size 20,000 generated from 10,000 Rayleigh-Ritz vectors. $\mathcal{S}^{n-1}$ is the unit sphere, i.e., all training vectors are scaled to have unit length. We train our model with TensorFlow (Abadi et al., 2015) on a single NVIDIA RTX A6000 GPU with 48GB memory. Training is done with standard deep learning techniques—more precisely, back-propagation and the ADAM optimizer (Kingma & Ba, 2015) (with starting learning rate 0.0001). Training takes approximately 10 minutes and 1 hour per epoch for grid resolutions 64^3 and 128^3, respectively. We trained our model for 50 epochs; however, the model from the thirty-first epoch was optimal for 64^3, and the model from the third epoch was optimal for 128^3.

Figure 3. DCDM for simulating a variety of incompressible flow examples. Left: smoke plume at $t = 6.67, 13.33, 20$ seconds. Middle: smoke passing a bunny at $t = 5, 10, 15$ seconds. Right: smoke passing a spinning box (time-dependent Neumann boundary conditions) at $t = 2.67, 6, 9.33$ seconds.

6. Results and Analysis

We demonstrate DCDM on three increasingly difficult examples and provide numerical evidence for the efficient convergence of our method. All examples were run on a workstation with dual stock AMD EPYC 75F3 processors, and an NVIDIA RTX A6000 GPU with 48GB memory.The grid resolutions we evaluate are the same as used in e.g. Tompson et al. (2017) and are common for graphics papers.

Figure 3 showcases DCDM for incompressible smoke simulations. In each simulation, inlet boundary conditions are set in a circular portion of the bottom of the cubic domain, whereby smoke flows around potential obstacles and fills the domain. We show a smoke plume (no obstacles), flow past a complex static geometry (the Stanford bunny), and flow past a dynamic geometry (a rotating cube). Visually plausible and highly-detailed results are achieved for each simulation (see supplementary material for larger videos). The plume example uses a computational grid with resolution 128^3, while the other two uses grids with resolution 256^3 (representing over 16 million unknowns). For each linear solve, DCDM was run until the residual was reduced by four orders of magnitude[2]. In our experience, production-grade solvers (e.g., 3D smoke simulators for movie visual effects) use resolutions of 128^3 or more, and as computing resources improve we are seeing more problems solved at huge scales like 512^3 and above, where a learning-enhanced

[2]Computer graphics experts have found that solving Poisson equations until a four orders-of-magnitude reduction in residual is achieved is enough for visual realism (any further computational effort does not yield easily perceptible differences) (McAdams et al., 2010; Panuelos et al., 2023).

method like DCDM will have a more dramatic impact.

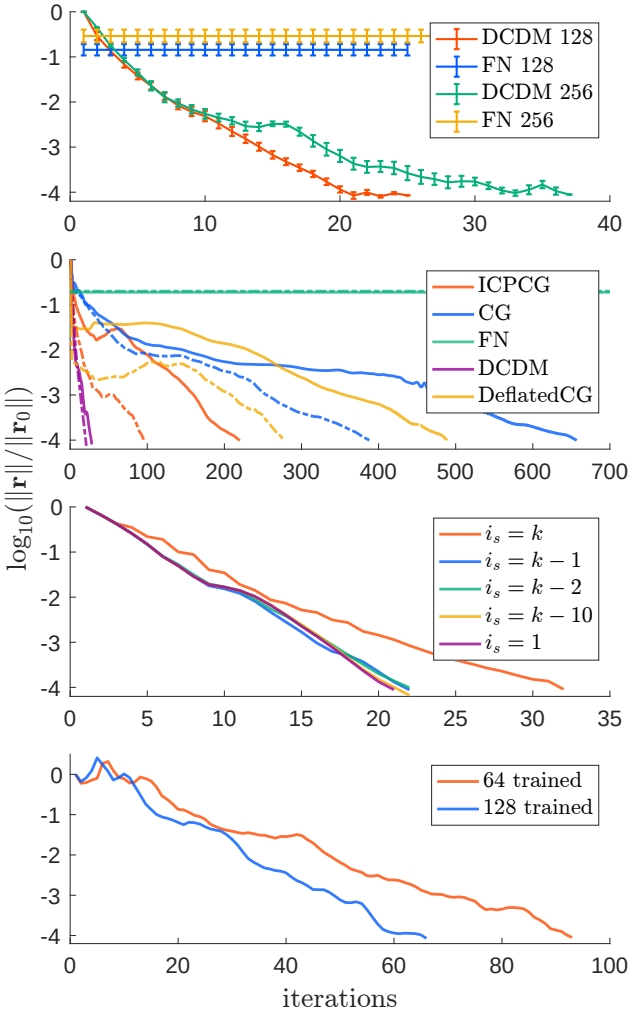


Figure 4. Convergence data for the bunny example (see also Table 1). **(a)** Mean and std. dev. (over all 400 frames in the simulation) of residual reduction during linear solves (with 128^3 and 256^3 grids) using FluidNet (FN) and DCDM. **(b)** Residual plots with ICPCG, CG, FN, DCDM, and Deflated CG at frame 150. Dashed and solid lines represent results for 128^3 and 256^3, respectively. **(c)** Decrease in residuals with varying degrees of $\boldsymbol{A}$-orthogonalization ($i_s = i_{\text{start}}$) in the 128^3 case. **(d)** Reduction in residuals when the network is trained with a 64^3 or 128^3 grid for the 256^3 grid simulation shown in Figure 3 Middle.

For the bunny example, Figures 4a–b demonstrate how residuals decrease over the course of a linear solve, comparing DCDM with other methods. Figure 4a shows the mean results (with standard deviations) over the course of 400 simulation frames, while in Figure 4b, we illustrate behavior on a particular frame (frame 150). For FluidNet, we use the optimized implementation provided by fluidnetsc22 (2022). This implementation includes pre-trained models that we use without modification. In both subfigures, it is evident that the FluidNet residual never changes, since the method is not iterative; FluidNet reduces the initial residual by no more than one order of magnitude. On the other hand, with DCDM, we can continually reduce the residual (e.g., by four orders of magnitude) as we apply more iterations of our method, just as with classical CG. In Figure 4b, we also visualize the convergence of three other classical methods, CG, Deflated CG (Saad et al., 2000), and incomplete Cholesky preconditioned CG (ICPCG); clearly, DCDM reduces the residual in the fewest number of iterations (e.g., approximately one order of magnitude fewer iterations than ICPCG). Since FluidNet is not iterative and lacks a notion of residual reduction, we treat $\boldsymbol{r}_0$ for FluidNet as though an initial guess of zero is used (as is done in our solver).

To clarify these results, Table 1 reports convergence statistics for DCDM compared to standard iterative techniques, namely, CG, Deflated CG, and ICPCG. For all 64^3, 128^3, and 256^3 grids with the bunny example, we measure the time t_r and the number of iterations n_r required to reduce the initial residual on a particular time step of the simulation by four orders of magnitude. DCDM achieves the desired results in by far the fewest number of iterations at all resolutions. At 256^3, DCDM performs approximately 6 times faster than CG, suggesting a potentially even wider performance advantage at higher resolutions. Inference is the dominant cost in an iteration of DCDM; the other linear algebra computations in an iteration of DCDM are comparable to those in CG. The nice result of our method is that despite the increased time per iteration, the number of required iterations is reduced so drastically that DCDM materially outperforms classical methods like CG. Although ICPCG successfully reduces number of iterations (Figure 4b), we found the runtime to scale prohibitively with grid resolution. We used SciPy's (Virtanen et al., 2020) `sparse.linalg.spsolve_triangular` function for forward and back substitution in our ICPCG implementation, and we also used a precomputed $\boldsymbol{L}$ that is not accounted for in the table results (though this took no more than 4 seconds at the highest resolution); Appendix A.3 includes further details on ICPCG.

Notably, even though Deflated CG and DCDM are based on approximate Ritz vectors, DCDM performs far better, indicating the value of using a neural network.

We performed three additional sets of tests. First, we tried low resolutions, 16^3 and 32^3, which are such small problems that we would expect CG to win due to the relatively high overhead of evaluating a neural network: indeed, DCDM and CG take 0.377sec/15iter and 0.008sec/48iter at 16^3, respectively, and 0.717sec/16iter and 0.063sec/53iter at 32^3. Note that we used the model (and parameters) tailored for 64^3 resolution to obtain these results; a lighter model, trained specifically for 16^3 and 32^3 resolutions, would give better timings, though likely still behind CG. Second, we

Method	64^3 Grid			128^3 Grid			256^3 Grid		
	t_r	n_r	tp_r	t_r	n_r	tp_r	t_r	n_r	tp_r
DCDM-64	2.71s	**16**	0.169s	**22s**	27	0.814 s	**261s**	58	4.50s
DCDM-128	5.37s	19	0.283 s	26s	**25**	1.083s	267s	**44**	6.07s
CG	**1.77**s	168	0.0105s	26s	465	0.0559s	1548s	1046	1.479s
Deflated CG	771.6s	117	6.594s	3700s	277	13.357s	21030s	489	43.00s
ICPCG	164s	43	3.813s	2877s	94	30.60s	54714s	218	250.98s

Table 1. Timing and iteration comparison for different methods on the bunny example. t_r, n_r and tp_r represents time, iteration and time per iteration. DCDM-{64,128} calls a model whose parameters are trained over a $\{64^3, 128^3\}$ grid. All computations are done using only CPUs; model inference does not use GPUs. All implementation is done in Python. See Appendix A.3 for convergence plots.

tested cases where d = 2, at resolutions 256^2 and 512^2. For this setup, running the smoke plume test (2D analogue of 3 Left) at 256^2, DCDM and CG take 2.18sec/64iter and 0.59sec/536iter, respectively. Again, since the system for this resolution is much smaller than those reported in Table 1, we expect CG to be more efficient. However, at 512^2, the system is big enough where we actually do outperform CG in time as well: 3.87sec/126iter for DCDM vs. 5.60sec/1146iter for CG. Third, we performed comparisons between DCDM and a more recent work, Sappl et al. (2019). Since Sappl et al. (2019) requires many asymptotically expensive computations (see Section 2), we expected a significant performance advantage with DCDM. For the 256^2 smoke plume example, using matrices from frame 10 of the simulation, Sappl et al. (2019) requires 1024 iterations for convergence (15.41s), vs. only 50 for DCDM (1.50s).

7. Conclusions

We presented DCDM, incorporating CNNs into a CG-style algorithm that yields efficient, convergent behavior for solving linear systems. Our method effectively acts as a preconditioner, albeit a nonlinear one[3]. Our method is evaluated on linear systems with over 16 million degrees of freedom and converges to a desired tolerance in merely tens of iterations. Furthermore, despite training the underlying network on a single domain (per resolution) without obstacles, our network is able to successfully predict search directions that enable efficient linear solves on domains with complex and dynamic geometries. Moreover, the training data for our network does not require running fluid simulations or solving linear systems ahead of time; our Rayleigh-Ritz vector approach enables us to quickly generate very large training datasets, unlike other works. We release our code, data, and pre-trained models so users can immediately apply DCDM to Poisson systems without further dataset generation or training, especially due to the feasibility of pre-trained weights for inference at different grid resolutions: `https://github.com/ayano721/2023_DCDM`.

Our network was designed for and trained exclusively using data related to the discrete Poisson matrix, which likely limits the generalizability of our present *model*. However, we believe our *method* is readily applicable to other classes of PDEs (or general problems with graph structure) that give rise to large, sparse, symmetric linear systems. To that end, we briefly applied DCDM to matrices arising from discretized heat equations (a similar class of large, sparse matrices; hence expected to work well with DCDM). We found that we can achieve convergence (reducing the initial residual by four orders of magnitude) using DCDM trained only on Poisson matrices—even though our test heat equation used Dirichlet boundary conditions, unlike the Neumann boundary conditions used with the Poisson equation systems we solved before. For a heat equation matrix at $N = 64$, DCDM can converge in only 14 iterations. Future work will extend this analysis. We note that our method is unlikely to work well for matrices that have high computational cost to evaluate $\boldsymbol{A} * x$ (e.g., dense matrices), since training relies on efficient $\boldsymbol{A} * \boldsymbol{x}$ evaluations. An interesting question is how well our method and current models would apply to discrete Poisson matrices arising from non-uniform grids, e.g., quadtrees or octrees (Losasso et al., 2004).

[3]Algebraically, any preconditioner P is attempting to learn an inverse of A^{Ω}, which is equivalent to what DCDM achieves for purposes of CG (learning the action of the inverse of the matrix on a vector $\boldsymbol{x}$). We initially tried learning a low-rank linear preconditioner, but our explorations were not successful; the approach was not efficient for higher resolutions because it required a large k.

Acknowledgements

This work is supported by DARPA through contract number FA8750-20-C-0537. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsor. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was also supported under DoE ORNL contract 4000171342. We would like to thank Professor Shigeo Morishima for his advice.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

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.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. Learning to learn by gradient descent by gradient descent. *Advances in Neural Information Processing Systems*, pp. 29, 2016.

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

Bridson, R. *Fluid simulation for computer graphics*. Taylor & Francis, 2008.

Chen, J., Kala, V., Marquez-Razon, A., Gueidon, E., Hyde, D. A. B., and Teran, J. A momentum-conserving implicit material point method for surface tension with contact angles and spatial gradients. *ACM Trans. Graph.*, 40(4), jul 2021. ISSN 0730-0301. doi: 10.1145/3450626.3459874. URL `https://doi.org/10.1145/3450626.3459874`.

Chen, T., Chen, X., Chen, W., Heaton, H., Liu, J., Wang, Z., and Yin, W. Learning to optimize: a primer and a benchmark. *Journal of Machine Learning Research 23*, pp. 8562–8620, 2022.

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

Fedkiw, R., Stam, J., and Jensen, H. Visual simulation of smoke. In *SIGGRAPH*, pp. 15–22. ACM, 2001.

fluidnetsc22. fluidnetsc22/fluidnet_sc22: v0.0.1, April 2022. URL `https://doi.org/10.5281/zenodo.6424901`. doi: 10.5281/zenodo.6424901, URL: `https://doi.org/10.5281/zenodo.6424901`.

Gagniere, S., Hyde, D., Marquez-Razon, A., Jiang, C., Ge, Z., Han, X., Guo, Q., and Teran, J. A hybrid Lagrangian/Eulerian collocated velocity advection and projection method for fluid simulation. *Computer Graphics Forum*, 39(8):1–14, 2020. doi: https://doi.org/10.1111/cgf.14096.

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. IEEE, 2018.

Grebhahn, A., Siegmund, N., Köstler, H., and Apel, S. Performance prediction of multigrid-solver configurations. In *Software for Exascale Computing-SPPEXA 2013-2015*, pp. 69–88. Springer, 2016.

Greenfeld, D., Galun, M., Basri, R., Yavneh, I., and Kimmel, R. Learning to optimize multigrid PDE solvers. In *Int Conf Mach Learn*, pp. 2415–2423. PMLR, 2019.

Harlow, F. and Welch, E. Numerical calculation of time dependent viscous flow of fluid with a free surface. *Phys Fluid*, 8(12):2182–2189, 1965.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Hestenes, M. R. and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409, 1952.

Hsieh, J.-T., Zhao, S., Eismann, S., Mirabella, L., and Ermon, S. Learning neural PDE solvers with convergence guarantees, 2019. URL `https://arxiv.org/abs/1906.01200`.

Ichimura, T., Fujita, K., Hori, M., Maddegedara, L., Ueda, N., and Kikuchi, Y. A fast scalable iterative implicit solver with Green's function-based neural networks. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pp. 61–68, 2020. doi: 10.1109/ScalA51936.2020.00013.

Kershaw, D. The incomplete Cholesky conjugate gradient method for the iterative solution of systems of linear equations. *J Comp Phys*, 26(1):43–65, 1978.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

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

Li, K. and Malik, J. Learning to optimize. *arXiv preprint*, 2016.

Liao, I., Dangovski, R. R., Foerster, J. N., and Soljačić, M. Learning to optimize quasi-newton methods. *arXiv preprint*, 2022.

Losasso, F., Gibou, F., and Fedkiw, R. Simulating water and smoke with an octree data structure. *ACM Trans. Graph.*, 23(3):457–462, 2004.

Losasso, F., Fedkiw, R., and Osher, S. Spatially adaptive techniques for level set methods and incompressible flow. *Computers & Fluids*, 35(10):995–1010, 2006.

Luna, K., Klymko, K., and Blaschke, J. P. Accelerating GMRES with deep learning in real-time, 2021. URL https://arxiv.org/abs/2103.10975.

Luz, I., Galun, M., Maron, H., Basri, R., and Yavneh, I. Learning algebraic multigrid using graph neural networks. In *Int Conf Mach Learn*, pp. 6489–6499. PMLR, 2020.

McAdams, A., Sifakis, E., and Teran, J. A parallel multigrid poisson solver for fluids simulation on large grids. In *Proc 2010 ACM SIGGRAPH/Eurograph Symp Comp Anim*, pp. 65–74. Eurographics Association, 2010.

Nussbaumer, H. The fast Fourier transform. In *Fast Fourier Transform and Convolution Algorithms*, pp. 80–111. Springer, 1981.

Paige, C. C. *The computation of eigenvalues and eigenvectors of very large sparse matrices*. PhD thesis, University of London, 1971.

Paige, C. C. and Saunders, M. A. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis*, 12(4):617–629, 1975.

Paluszny, A. and Zimmerman, R. W. Numerical simulation of multiple 3d fracture propagation using arbitrary meshes. *Computer Methods in Applied Mechanics and Engineering*, 200(9):953–966, 2011. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2010.11.013. URL https://www.sciencedirect.com/science/article/pii/S0045782510003373.

Panuelos, J., Goldade, R., Grinspun, E., Levin, D., and Batty, C. PolyStokes: A polynomial model reduction method for viscous fluid simulation. *ACM Trans Graph (TOG)*, 42(4), 2023.

Ruelmann, H., Geveler, M., and Turek, S. On the prospects of using machine learning for the numerical simulation of PDEs: Training neural networks to assemble approximate inverses, 2018. URL http://dx.doi.org/10.17877/DE290R-18778.

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

Saad, Y. and Schultz, M. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci Stat Comp*, 7(3):856–869, 1986.

Saad, Y., Yeung, M., Erhel, J., and Guyomarc'h, F. A deflated version of the conjugate gradient algorithm. *SIAM Journal on Scientific Computing*, 21:1909–1926, 2000.

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.

Shen, J., Chen, X., Heaton, H., Chen, T., Liu, J., Yin, W., and Wang, Z. Learning a minimax optimizer: A pilot study. *International Conference on Learning Representations*, 2019.

Stanaityte, R. *ILU and Machine Learning Based Preconditioning For The Discretized Incompressible Navier-Stokes Equations*. PhD thesis, University of Houston, 2020.

Stiefel, E. Über einige methoden der relaxationsrechnung. *Zeitschrift für angewandte Mathematik und Physik ZAMP*, 3(1):1–33, 1952.

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.

Trefethen, L. and Bau, D. *Numerical Linear Algebra*, volume 50. SIAM, 1997.

Um, K., Brand, R., Fei, Y., Holl, P., and Thuerey, N. Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 6111–6122. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/43e4e6a6f341e00671e123714de019a8-Paper.pdf.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P.,

Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Yang, C., Yang, X., and Xiao, X. Data-driven projection method in fluid simulation. *Comp Anim Virt Worlds*, 27 (3-4):415–424, 2016.

A. Appendix

A.1. Conjugate Gradients Method

In this appendix, we provide a review of the conjugate gradients (CG) method, which is the inspiration for DCDM as presented in this paper. The conjugate gradients method is a special case of the line search method, where the search directions are $\boldsymbol{A}$-orthogonal to each other. It can also be viewed as a modification of gradient descent (GD) where the search direction is chosen as the component of the residual (equivalently, the negative gradient of the matrix norm of the error) that is $\boldsymbol{A}$-orthogonal to all previous search directions:

$$\boldsymbol{d}_k = \boldsymbol{r}_{k-1} - \sum_{i=1}^{k-1} h_{ik} \boldsymbol{d}_i, \qquad h_{ik} = \frac{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{r}_{k-1}}{\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{d}_i}.$$

With this choice, the search directions form a basis for $\mathbb{R}^n$ so that the initial error can be written as $\boldsymbol{e}_0 = \boldsymbol{x} - \boldsymbol{x}_0 = \sum_{i=1}^{n} e_i \boldsymbol{d}_i$, where e_i are the components of the initial error written in the basis. Furthermore, since the search directions are $\boldsymbol{A}$-orthogonal, the optimal step sizes α_k at each iteration satisfy

$$\begin{aligned} \alpha_k &= \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k} = \frac{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{e}_{k-1}}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k} \\ &= \frac{\boldsymbol{d}_k^T \boldsymbol{A} \left(\sum_{i=1}^{n} e_i \boldsymbol{d}_i - \sum_{j=1}^{k-1} \alpha_j \boldsymbol{d}_j \right)}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k} = e_k. \end{aligned}$$

That is, the optimal step sizes are chosen to precisely eliminate the components of the error on the basis defined by the search directions. Thus, convergence is determined by the (at most n) non-zero components e_i in the initial error. Although rounding errors prevent this from happening exactly in practice, this property greatly reduces the number of required iterations (Golub & Loan, 2012).

Furthermore, $h_{ik} = 0$ for $i < k - 1$, and thus iteration can be performed without the need to store all previous search directions $\boldsymbol{d}_i$ and without the need for computing all previous h_{ik}. To see this, it is sufficient to show $\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{r}_{k-1} = 0$.

Lemma In the CG method, residuals are orthogonal, i.e., $\boldsymbol{r}_k^T \boldsymbol{r}_j = 0$ for all $j < k$.

Proof

$$\begin{aligned} \boldsymbol{r}_k^T \boldsymbol{r}_j &= (\boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{A} \boldsymbol{d}_k)^T \boldsymbol{r}_j \\ &= \boldsymbol{r}_{k-1}^T \boldsymbol{r}_j - \alpha_k \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{r}_j \\ &= \boldsymbol{r}_{k-1}^T \boldsymbol{r}_j - \alpha_k \boldsymbol{d}_k^T \boldsymbol{A} \left(\boldsymbol{d}_{j+1} + \sum_{i=1}^{j} h_{i(j+1)} \boldsymbol{d}_i \right) \end{aligned}$$

For $j < k-1$, $\boldsymbol{r}_{k-1}^T \boldsymbol{r}_j = 0$ follows from induction. $\boldsymbol{d}_k^T \boldsymbol{A} (\boldsymbol{d}_{j+1} + \sum_{i=1}^{j} h_{i(j+1)} \boldsymbol{d}_i) = \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_{j+1} + \sum_{i=1}^{j} h_{i(j+1)} \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_i$=0

because $\boldsymbol{d}_i$ are $\boldsymbol{A}$-orthogonal by their definition. For $j = k-1$,

$$\begin{aligned}
\boldsymbol{r}_k^T \boldsymbol{r}_{k-1} &= \boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{r}_{k-1} \\
&= \boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1} - \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k} \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{r}_{k-1} \\
&= \boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1} - \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k} \boldsymbol{d}_k^T \boldsymbol{A} \left(\boldsymbol{d}_k + \sum_{i=1}^{k-1} h_i \boldsymbol{d}_k \right) \\
&= \boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1} - \boldsymbol{r}_{k-1}^T \boldsymbol{d}_k \qquad \text{(by } \boldsymbol{A}\text{-orthogonality of } \boldsymbol{d}_k) \\
&= \boldsymbol{r}_{k-1}^T (\boldsymbol{r}_{k-1} - \boldsymbol{d}_k) \\
&= \boldsymbol{r}_{k-1}^T \left(\sum_{i=1}^{k-1} h_{ik} \boldsymbol{d}_i \right) \\
&= \sum_{i=1}^{k-1} h_{ik} \boldsymbol{r}_{k-1}^T \boldsymbol{d}_i
\end{aligned}$$

So proving $\boldsymbol{r}_{k-1}^T \boldsymbol{d}_i = 0$ for $i < k$ would finish the proof. However, by the definition of $\boldsymbol{d}_i = \boldsymbol{r}_{i-1} - \sum_{j=1}^{i-1} h_{ij} \boldsymbol{d}_k$, induction proves $\boldsymbol{d}_i \in \text{span}(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{i-1})$. Hence, $\boldsymbol{r}_{k-1}^T \boldsymbol{d}_i = 0$ for all $i \leq k-1$, which proves the lemma.

Claim In the CG method, search directions are $\boldsymbol{A}$-orthogonal to all previous residuals, i.e., $\boldsymbol{d}_i^T \boldsymbol{A} \boldsymbol{r}_{k-1} = 0$ for all $i < k-1$.

Proof $\boldsymbol{r}_i^T \boldsymbol{r}_{k-1} = (\boldsymbol{r}_{i-1}^T - \alpha_i \boldsymbol{A} \boldsymbol{d}_i)^T \boldsymbol{r}_{k-1}$, hence $\boldsymbol{d}_i \boldsymbol{A} \boldsymbol{r}_{k-1} = \boldsymbol{r}_{i-1}^T \boldsymbol{r}_{k-1} - \boldsymbol{r}_i^T \boldsymbol{r}_{k-1} = 0$ for all $i < k-1$, using the lemma above.

This proves the sparsity of the h_{ik}. As discussed in the main body of the paper, this "memoryless" property of CG is inherited by DCDM and enables the efficiency of our method.

A.2. Choice of α

Line search is an iterative method to find a local minimum of an objective function $h : \mathbb{R}^n \to \mathbb{R}$. In the context of variationally solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b}$, and the k^{th} iterate is computed by

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_k \mathbf{d}_k.$$

One desires a step size α_k that yields $h(\boldsymbol{x}_k) < h(\boldsymbol{x}_{k-1})$. More specifically, the optimal choice is

$$\alpha_k = \arg\min_{\alpha} h(\mathbf{x}_{k-1} + \alpha \mathbf{d}_k) = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k},$$

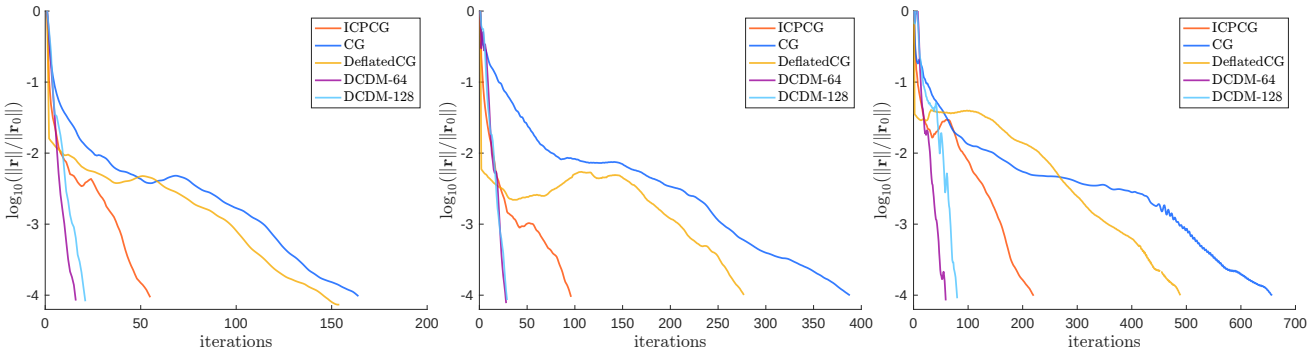
where $\boldsymbol{r}_{k-1} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{k-1}$ is the $(k-1)^{\text{th}}$ residual. To see that this choice of α_k is indeed the minimizer, we can define the objective function as $g(\alpha)$ and write

$$\begin{aligned}
g(\alpha) &= h(\boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}_k) \\
&= \frac{1}{2}(\boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}_k)^T \boldsymbol{A} (\boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}_k) - \boldsymbol{b}^T (\boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}_k) \\
&= \frac{1}{2}\alpha^2 \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k + \alpha (\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{x}_{k-1} - \boldsymbol{d}_k^T \boldsymbol{b}) + \left(\frac{1}{2} \boldsymbol{x}_{k-1}^T \boldsymbol{A} \boldsymbol{x}_{k-1} + \boldsymbol{x}_{k-1}^T \boldsymbol{b} \right) \\
&= \frac{1}{2}\alpha^2 \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k - \alpha \boldsymbol{d}_k^T \underbrace{\boldsymbol{r}_{k-1}}_{\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{k-1}} + (\text{constant terms}).
\end{aligned}$$

Taking the derivative with respect to α, we have $g'(\alpha) = \alpha \boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k - \boldsymbol{d}_k^T \boldsymbol{r}_{k-1} = 0$, yielding $\alpha = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \boldsymbol{A} \boldsymbol{d}_k}$ as the minimizer of $g(\alpha)$.

A.3. Additional Convergence Results

We include additional convergence results, similar to those shown in Figure 4b, in Figure 5. Specifically, these plots show the convergence of all the methods reported in Table 1 at each of the resolutions reported there. The figure visually demonstrates the significant reduction in iteration count achieved by DCDM.



(a) $N = 64$ (b) $N = 128$ (c) $N = 256$

Figure 5. Convergence of different methods on the 3D bunny example for $N = 64$, 128, 256; summary results, as well as timings, are reported in Table 1. DCDM-{64,128} calls a model whose parameters are trained over a $\{64^3, 128^3\}$ grid.

We remark on ICPCG since it is a popular preconditioner and closest in performance to DCDM. When using ICPCG with matrices that arise in a domain with moving internal boundaries (such as our bunny examples), the approximate factorization of $\boldsymbol{A}$ must be recomputed. Recomputation is also required in the approach of Tompson et al. (2017) in examples like these. Moreover, as Figure 4d shows, DCDM does not require full A-orthogonality. Hence the algorithm only stores two previous vectors, just like CG, and unlike the much more significant memory requirements of ICPCG. For example, the L and D matrices for 128^3 take about 18.7MB in `scipy.sparse` format, while our network can be stored in less than 500KB.

A.4. Ablation Study and Runtime Analysis

Method	DCDM	Model 1	Model 2	Model 3	Model 4	U-Net
Number of Parameters	97,457	97,457	97,457	97,457	24,537	3,527,505

Table 2. Number of parameters for each network architecture considered in the ablation study.

Here, we provide results of a small ablation study on network architecture in order to justify some of the architectural choices we made in constructing the DCDM network. We considered a few different models (Figure 8a to Figure 8e), several of which are modifications of the model we ultimately used to generate our results (Figure 8a). The models we considered include one without ResNet connections (Figure 8b), one with simple downsampling and upsampling (a U-Net-like structure) (Figure 8c), a minimal CNN (Figure 8d), and a model with different filter sizes of the blocks (Figure 8e). We compared how these models perform on the same bunny example considered in the main part of the paper (at resolution 64^3). Figure 6 shows that the architecture we ultimately selected for DCDM yields the best results.

Each model's parameter count is listed in Table 2. Compared to a basic CNN or U-Net architecture (like the one used in Tompson et al. (2017)), our DCDM network is actually quite light. For example, the U-Net architecture in Tompson et al. (2017) uses 3,527,505 parameters (at $N = 64$ in 3D), while our network (at the same resolution) requires only 97,457 parameters (a 36x reduction). In addition, one advantage of our method is that DCDM only needs to be trained once (and data only generated once) per problem class (and possibly size). So if a user desired to solve Poisson systems (which are quite common in computer graphics and engineering), they could use our pre-trained models off the shelf; though we readily concede that new classes of matrices or new resolutions could require new data generation or retraining.

Dataset generation is a key step in using the DCDM model we selected. We found that we needed to include orthogonalization to previous vectors in the Lanczos problem in practice (a well-known limitation of the method). This causes the creation of a dataset (cf. Section 5.1) to take $O(n^3m^2)$ time, where m is the number of Lanczos vectors to be created and n is the

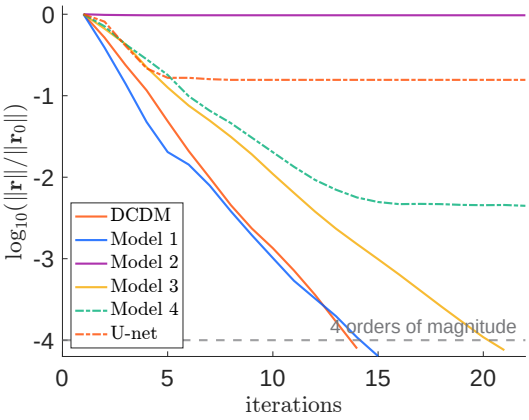
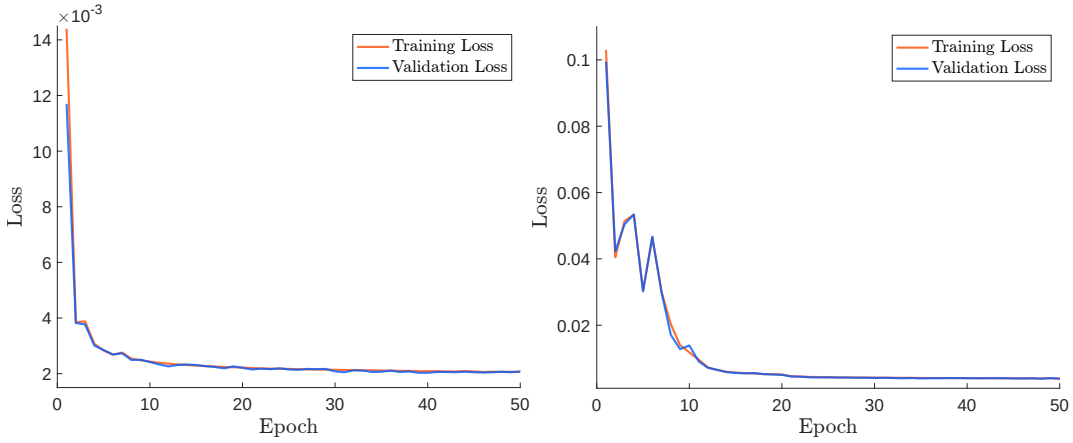


Figure 6. Residual plot for the bunny example at $N = 64$ with each trained model. The dashed line represents a four-orders-of-magnitude reduction in residual, which is the convergence criterion we use throughout our examples.

resolution. Hence increasing resolution from 64^3 to 128^3 increases the time by a factor of 8, which scales 5–7 hours to 2–2.5 days. (However, since we can use low resolution models on higher resolution problems, this scaling can be mitigated, cf. Section 5.2.) In addition, the orthogonalization step makes dataset generation have complexity $O(n^3m^2)$, instead of the $O(n^3m)$ complexity of classical Lanczos processes. If we can find any other solution for the numerical problems of classical Lanczos iteration besides orthogonalization, we can drastically reduce the time to generate the dataset (such a task is outside the scope of the present work). We note that storing the training dataset has asymptotic cost $O(kn^3)$; for instance, the dataset of $k = 20{,}000$ synthetic data takes 23GB and 159GB of storage for resolutions 64^3 and 128^3, respectively.

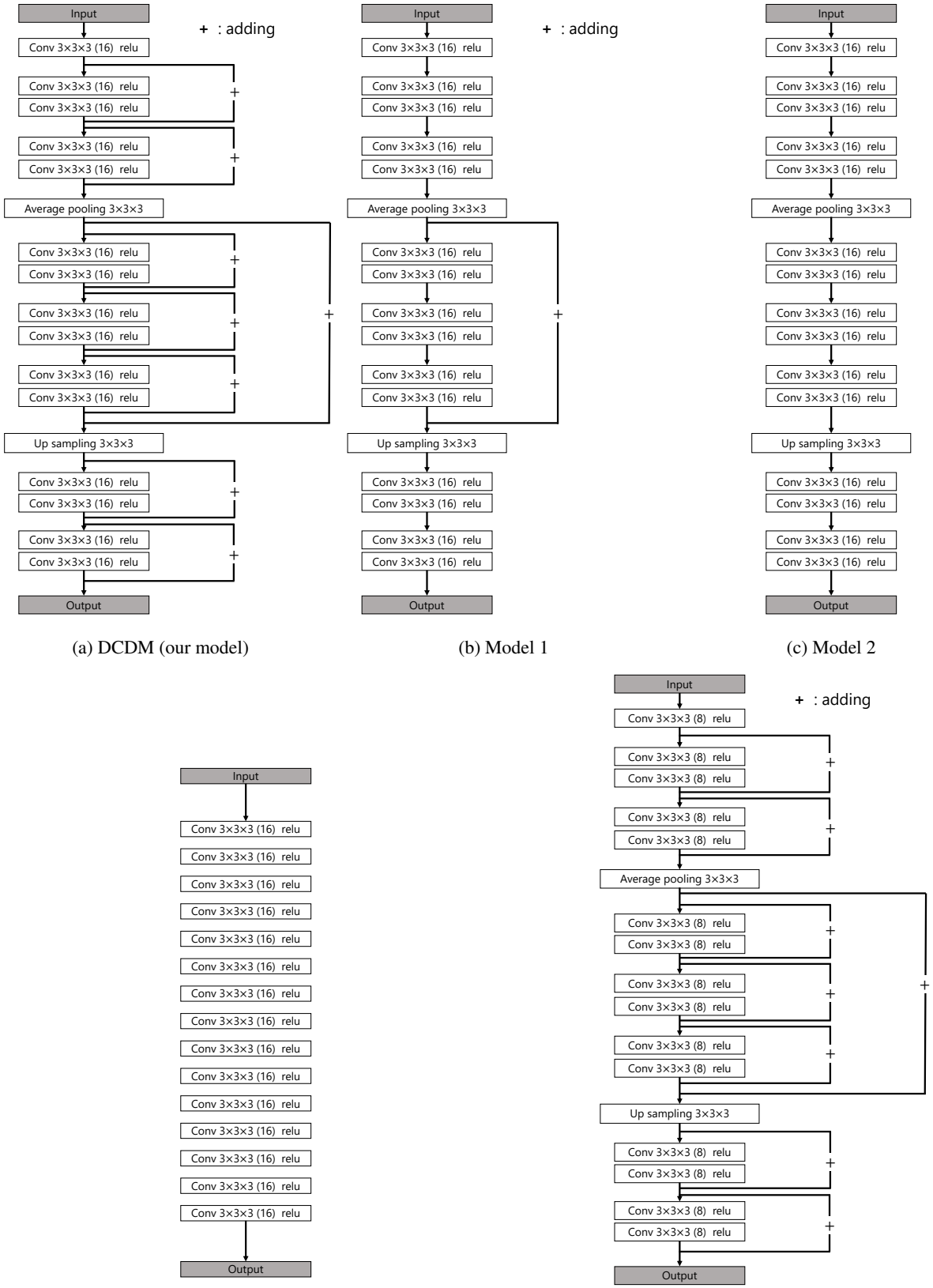
A.5. Model training

Figure 7 shows the decrease in training and validation losses observed when training the neural network used for DCDM. As mentioned in Section 5.3, for DCDM, we selected the model after epoch 31 for $N = 64$ and epoch 3 for $N = 128$. The plots clearly demonstrate that training and validation loss seem to decrease after these epochs. However, we found that our epoch selections yielded the best performance on our test data, namely, the examples we showed in Section 6. Accordingly, we conjecture that our model overfit relatively quickly to both training and validation data, and that perhaps training and validation data were much more similar to each other compared to the test data. We are interested in exploring this further in future work. Of course, philosophically, choosing a model by comparing its performance from different epochs on test data essentially makes that test data part of the validation data, but this is a broader discussion for the learning community.



(a) $N = 64$ (b) $N = 128$

Figure 7. Training and validation loss for the networks used in DCDM at resolutions $N = 64$ and $N = 128$.



(a) DCDM (our model) (b) Model 1 (c) Model 2

(d) Model 3 (e) Model 4

Figure 8. Network architectures considered for our ablation study.