A Deep Gradient Correction Method for Iteratively Solving Linear Systems

Ayano Kaneda  
Department of Pure and Applied Physics  
Waseda University  
Tokyo, Japan  
dizzy-miss-lizzy@moegi.waseda.jp

Osman Akar  
Department of Mathematics  
UCLA  
Los Angeles, CA, USA  
oak@math.ucla.edu

Jingyu Chen  
Department of Mechanical and Aerospace Engineering  
UCLA  
Los Angeles, CA, USA  
chenjy@ucla.edu

Victoria Kala  
Department of Mathematics  
UCLA  
Los Angeles, CA, USA  
victoriakala@ucla.edu

David Hyde  
Department of Computer Science  
Vanderbilt University  
Nashville, TN, USA  
david.hyde.1@vanderbilt.edu

Joseph Teran  
Department of Mathematics  
UC Davis  
Davis, CA, USA  
jteran@math.ucdavis.edu

Abstract

We present a novel deep learning approach to approximate the solution of large, sparse, symmetric, positive-definite linear systems of equations. These systems arise from many problems in applied science, e.g., in numerical methods for partial differential equations. Algorithms for approximating the solution to these systems are often the bottleneck in problems that require their solution, particularly for modern applications that require many millions of unknowns. Indeed, numerical linear algebra techniques have been investigated for many decades to alleviate this computational burden. Recently, data-driven techniques have also shown promise for these problems. 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 directions. 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 train the network using unsupervised learning with a loss function equal to the $L^2$ difference between an input and the system matrix times the network evaluation, where the unspecified constant in the approximate inverse is accounted for. We demonstrate the efficacy of our approach on spatially discretized Poisson equations with millions of degrees of freedom arising in computational fluid dynamics applications. 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.

Preprint. Under review.
1 Introduction

The solution of large, sparse systems of linear equations is ubiquitous when partial differential equations (PDEs) are discretized to simulate complex natural phenomena such as fluid flow [29], thermodynamics [8], mechanical fracture [36], etc. Discretization often leads to a linear system

\[ Ax = b \]  

where the dimension \( n \) of the matrix \( A \in \mathbb{R}^{n \times n} \) and the vector \( b \in \mathbb{R}^n \) correlate with spatial fidelity of the computational domain. Overall quality and realism of a simulation is proportional to this spatial fidelity and typical modern applications of numerical PDEs require the solution of linear systems with many 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 [14, [39].

The appropriate numerical linear algebra technique depends on the nature of the problem. In the present work, we consider sparse linear systems that arise from discrete Poisson equations in incompressible flow applications [9, 10, 6]. Direct solvers that utilize matrix factorizations (QR, Cholesky, etc. [47]) have optimal approximation error, but their computational cost is \( O(n^3) \) and they typically require dense storage, even for sparse \( A \). Although Fast Fourier Transforms [33] can be used in limited instances (periodic boundary conditions, etc.), iterative techniques are most commonly adopted for these systems given their sparsity. 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 [39]. 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 and the preconditioned conjugate gradients method (PCG) can be used to minimize iteration counts [39, 19, 45]. Preconditioners \( P \) for PCG must simultaneously: be symmetric positive definite (and therefore admit factorization \( P = F^2 \)), improve the condition number of the preconditioned system \( FAFy = Fb \), 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) [25] use a sparse approximation to the Cholesky factorization and significantly reduce iteration counts in practice; however, their inherent data dependency prevents efficient parallel implementation. Nonetheless, these are very commonly adopted for Poisson equations arising in incompressible flow [10, 6]. Multigrid [5] and domain decomposition [39] 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 [17, 16, 32]. Others have developed machine learning methods to estimate preconditioners [15, 44, 22] and initial guesses for iterative methods [31, 49, 2]. Thompson et al. [46] and Yang et al. [52] develop non-iterative machine learning approximations of the inverse of discrete Poisson equations from incompressible flow. We leverage deep learning and develop a novel accelerated conjugate gradients iterative method for approximating the solution of symmetric positive definite linear systems which we call the deep gradient correction method (DGCM). CG iteratively adds \( A \)-conjugate search directions while minimizing the matrix norm of the error. We use a convolutional neural network 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 modified conjugate gradients approach is designed to accommodate this nonlinearity. We use unsupervised learning to train our network with a loss function equal to the \( L^2 \) difference between an input vector and a scaling of \( A \) 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 (matrices \(A\) and right hand sides \(b\)). We benchmark our algorithm using the ubiquitous pressure Poisson equation (discretized on regular voxelized domains) and compare against FluidNet \([46]\), which is the state of the art learning-based method for these types of problems. Unlike the non-iterative approaches of Thompson et al. \([46]\) and Yang et al. \([52]\), our method can reduce the linear system residuals \emph{arbitrarily}. We showcase our approach with examples that have over 16 million degrees of freedom.

2 Related Work

Deep learning has been applied in various ways to physical simulation and the numerical solution of PDEs (see, e.g., the reviews in \([4, 13, 24]\)). From learning how to discretize PDEs \([3]\) to upsampling the results of low-resolution simulations \([23, 48]\), practitioners have sought to incorporate learning into most aspects of numerical PDE pipelines.

A number of recent works have sought to eschew the numerical solution of PDEs by using neural networks and appropriate representations of physical quantities \([50, 41, 51]\), often taking into account PDEs or physics while training. Many other works have hybridized traditional techniques and governing equations with neural networks in order to improve the accuracy and/or efficiency of solving PDEs, without replacing PDEs wholesale (e.g., \([28, 20, 43]\)). We note the popular physics-informed neural network framework of Karniadakis and colleagues \([7, 37]\), which uses automatic differentiation to represent all PDE operators and incorporates physical constraints like conservation laws into the network’s loss function.

Notably, several papers have focused on enhancing the solution of linear systems (arising from discretized PDEs) using learning. For instance, \([15]\) generate sparsity patterns for block-Jacobi preconditioners using convolutional neural networks (CNNs), and \([44]\) use a CNN to predict non-zero patterns for ILU-type preconditioners for the Navier-Stokes equations (though neither work designs fundamentally new preconditioners). \([22]\) 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. \([21]\) learn an iterator that solves linear systems, performing competitively with classical solvers like multigrid-preconditioned MINRES \([35]\). Luz et al. \([32]\) and Greenfield et al. \([17]\) 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 \(A\). They train a graph neural network to learn (in an unsupervised fashion) a mapping from matrices \(A\) to prolongation operators. Gebbahn et al. \([16]\) 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, \([31]\) accelerate the convergence of GMRES \([40]\) with an initial guess that is learned in real-time (i.e., as a simulation code runs) with no prior data. \([49]\) 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, \([2]\) 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 \([48, 42]\) have sought to learn a mapping between a matrix and an associated sparse approximate inverse. In their investigation, \([38]\) 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. \([42]\) 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, which is achieved exactly when an exact inverse is obtained. Although this framework is quite simple, evaluation of the condition number of a matrix is asymptotically costly, and in general, the inverse of a sparse matrix can be quite dense. Accordingly,
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 matrix $A^{(0,1)^d}$ from a discretized domain with no interior boundary conditions, where $d$ is the dimension. (d) We illustrate the non-zero entries in an example matrix $A^\Omega$ from the voxelized and labeled (white vs. blue) grid for three example interior cells (green, purple, and brown). Each case illustrates the non-zero entries in the row associated with the example cell. All entries in rows corresponding to boundary/blue cells are zero.

the method is not efficient or accurate enough for the large-scale 3D problems that arise in real-world engineering problems.

Most relevant to the present work is FluidNet [46]. FluidNet develops a highly-tailored CNN architecture that is used 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.

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 discrete Poisson equations in regular-grid-based discretization of the pressure projection equations that arise in Chorin’s splitting technique [9] for the inviscid, incompressible Euler equations. These equations are

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

(2)

where $\mathbf{u}$ is fluid velocity, $p$ is pressure, $\rho$ is density, and $f^{ext}$ accounts for external forces like gravity. The equations are assumed at all positions $\mathbf{x}$ in the spatial fluid flow domain $\Omega$ and for time $t > 0$. The left term in Equation (2) enforces conservation of momentum in the absence of viscosity, and the second part enforces incompressibility and conservation of mass. These equations are subject to initial conditions $\rho(\mathbf{x}, 0) = \rho^0$ and $\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}^0(\mathbf{x})$ as well as boundary conditions $\mathbf{u}(\mathbf{x}, t) \cdot \mathbf{n}(\mathbf{x}) = u^{\partial \Omega}(\mathbf{x}, t)$ on the boundary of the domain $\mathbf{x} \in \partial \Omega$ (where $\mathbf{n}$ is the unit outward pointing normal at position $\mathbf{x}$ on the boundary). Equation (2) is discretized in both time and space. Temporally, we split the advection $\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \mathbf{u} = 0$ and body forces terms $\rho \frac{\partial \mathbf{u}}{\partial t} = f^{ext}$, and finally enforce incompressibility via the pressure projection $\frac{\partial \mathbf{u}}{\partial t} + \frac{1}{\rho} \nabla p = 0$ such that $\nabla \cdot \mathbf{u} = 0$; this is the standard advection-projection scheme proposed by Chorin [9]. Using finite differences in time, we
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 discuss relevant details of the method of conjugate gradients, particularly line search and $A$-orthogonal search directions. 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 convolutional neural network.

### 4 Deep Gradient Correction 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 discuss relevant details of the method of conjugate gradients, particularly line search and $A$-orthogonal search directions. 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 convolutional neural network.

#### 4.1 Line search and $A$-orthogonality

Our approach iteratively improves approximations to the solution $x$ of Equation 6. We build on the method of conjugate gradients (CG), which requires the matrix $A^\Omega$ in Equation 6 to be symmetric positive definite (SPD). SPD matrices $A^\Omega$ give rise to the matrix norm $\|y\|_{A^\Omega} = \sqrt{y^T A^\Omega y}$. The conjugate gradients method can be derived in terms of iterative line search improvement based on optimality in this norm. That is, an iterate $x_{k-1} \approx x$ is updated in search direction $d_k$ by a step size $\alpha_k$ that is chosen to minimize the matrix norm of the error:

$$\alpha_k = \arg\min_{\alpha} \frac{1}{2} \|x - (x_{k-1} + \alpha d_k)\|_{A^\Omega}^2 = \frac{r_{k-1}^T d_k}{d_k^T A^\Omega d_k},$$

where $r_{k-1} = b - A^\Omega x_{k-1}$ is the $(k-1)$th residual. Different search directions result in different $\alpha_k$. A natural choice is the negative gradient of the matrix norm of the error (evaluated at the current iterate), since this will point in the direction of steepest decrease $d_k = -\frac{1}{2} \nabla \|x_{k-1}\|_{A^\Omega}^2 = \nabla \|x_{k-1}\|_{A^\Omega} = \nabla \frac{1}{2} \|x_{k-1}\|_{A^\Omega}^2$. However, this choice is undesirable since the gradient information is non-local and may not be readily available or computationally impractical to compute. Instead, we propose to train a deep convolutional neural network to compute the search direction $d_k$ given the residual $r_{k-1}$.
We demonstrate this in Figure 4c. We summarize our approach in Algorithm 1. Note that we introduce the variable \( A \) while the residual is a natural choice for generating \( A \)-orthogonal directions. For example, well as \( A \)-orthogonality between all search directions, we must have \( A \)-orthogonality between all 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.

4.2 Learning Optimal Search Directions

While the residual is a natural choice for generating \( A \)-orthogonal search directions (since it points in the direction of the steepest local decrease), it is not the optimal search direction. If \( d_k \) is parallel to \((A^\Omega)^{-1}r_{k-1}\), then \( x_k \) will be equal to \( x \) since \( \alpha_k \) (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:

\[
\mathbf{r}_k = \mathbf{b} - A^\Omega\mathbf{x}_k = \mathbf{b} - A^\Omega\mathbf{x}_{k-1} - \alpha_k A^\Omega \mathbf{d}_k = \mathbf{r}_{k-1} - \alpha_k A^\Omega \mathbf{d}_k.
\]

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

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

We summarize our approach in Algorithm 1. Note that we introduce the variable \( i_{\text{start}} \). To guarantee \( A \)-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.
5 Model Architecture, Datasets, and Training

Efficient performance of our method requires effective training of our deep convolutional network for weights and biases c such that \( f(c, r) \approx s_k(A^{\Omega})^{-1}r \) (for arbitrary scalar \( s_k \)). We design a model architecture, loss function, and unsupervised 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 Unsupervised Learning

Although we generalize to arbitrary matrices \( A^\Omega \) 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 \( A^{(0,1)^3} \) from the full cube domain \((0, 1)^3\). In contrast, other similar approaches \([46, 52]\) train using matrices \( A^\Omega \) and right hand sides \( bV^\Omega u^* + b^{\text{orig}} \) that arise from flow in many domains with internal boundaries. We train our network by minimizing the \( L^2 \) difference \( \| r - \alpha A^{(0,1)^3} f(c, r) \|_2 \),

\[
\text{Loss}(f, c, D) = \frac{1}{|D|} \sum_{r \in D} \| r - \frac{r^T f(c, r)}{f(c, r)^T A^{(0,1)^3} f(c, r)} A^{(0,1)^3} f(c, r) \|_2
\]

for given dataset \( D \) consisting of training vectors \( b^i \). In Algorithm 1 the normalized residuals \( \frac{r_k}{|r_k|} \) are passed as inputs to the model. Unlike in e.g. FluidNet \([46]\), only the first residual \( \frac{r_0}{|r_0|} \) is directly related to the problem-dependent original right hand side \( b \). 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 \( r_k \) in Algorithm 1 are skewed to the lower end of the spectrum of the matrix \( A^\Omega \). Since \( A^\Omega \) is a discretized elliptic operator, this means that lower end modes are of lower frequency of spatial oscillation. We create our training vectors \( b^i \in D \) using \( N \ll n \) approximate eigenvectors of the training matrix \( A^{(0,1)^3} \). We use the Rayleigh-Ritz method to create approximate eigenvectors \( q_i \), \( 0 \leq i < N \). This approach allows us to effectively approximate the full spectrum of \( A^{(0,1)^3} \) without computing the full eigendecomposition, which can be expensive (\( O(n^3) \)) at high resolution. We found that using \( N = 10000 \) worked well in practice. The Rayleigh-Ritz vectors are orthonormal and satisfy \( Q^T_N A^{(0,1)^3} Q_N = A_N \), where \( A_N \) is a diagonal matrix with nondecreasing diagonal entries \( \lambda_i \) referred to as Rayleigh values (approximate eigenvalues) and \( Q_N = [q_0, q_1, \ldots, q_{N-1}] \in \mathbb{R}^{n \times N} \). We pick \( b^i = \frac{\sum_{j=1}^N c_j q_j}{\| \sum_{j=1}^N c_j q_j \|} \), where the coefficients \( c_j \) are picked from a normal distribution

\[
c_j = \begin{cases} 9 \cdot \mathcal{N}(0, 1) & \text{if } j \leq \frac{N}{2} + \theta \\ \mathcal{N}(0, 1) & \text{otherwise} \end{cases}
\]

where \( \theta \) is a small number (we used \( \theta = 500 \)), and \( j \) is the first index that \( \lambda_j > 0 \). This choice creates \( 90\% \) of \( b^i \) from the lower end of the spectrum with the remaining \( 10\% \) from the higher end. Notably, even though this dataset only uses Rayleigh-Ritz vectors from the training matrix \( A^{(0,1)^3} \), our network can be effectively generalized to flows in irregular domains, e.g., smoke flow past a rotating box and flow past a bunny (see Figure 3).

We generate the Rayleigh-Ritz vectors by first tridiagonalizing the training matrix \( A^{(0,1)^3} \) with \( N \) Lanczos iterations \([27, 34]\) to form \( T^N = Q^T_N A^{(0,1)^3} Q^T_N \in \mathbb{R}^{n \times N} \). We then diagonalize \( T^N = Q^T \Lambda_N Q \) using the QR eigenvalue algorithm with implicit Wilkinson shifts \([27]\). While costly, we note that this algorithm is performed on the comparably small \( N \times N \) matrix \( T^N \) (rather than

---

Algorithm 1 DGCM

1: \( r_0 = b - A^\Omega x_0 \)
2: \( d_1 = r_0 \)
3: \( k = 1 \)
4: while \( \| r_{k-1} \| \geq \epsilon \) do
5: \( d_k = f(c, r_{k-1}) \)
6: for \( i = k \) to \( \epsilon \) do
7: \( h_{ik} = \frac{d_k^T A^\Omega d_i}{d_i^T A^\Omega d_i} \)
8: \( d_{k+1} = h_{ik} d_i \)
9: end for
10: \( \alpha_k = \frac{d_k^T d_k}{\| d_k \|_2^2} \)
11: \( x_k = x_{k-1} + \alpha_k d_k \)
12: \( r_k = b - A^\Omega x_k \)
13: \( k = k + 1 \)
14: end while
on the $A^{(0,1)^3} \in \mathbb{R}^{n \times n}$). This yields the Rayleigh-Ritz vectors as the columns of $Q_N = Q_L^T \hat{Q}$. The Lanczos vectors are the columns of the matrix $Q_L^T$ and satisfy a three-term recurrence whereby we can iteratively compute the next Lanczos vector from previous two as

$$
\beta_j q_j^{L+1} = A^{(0,1)^3} q_j^L - \beta_j q_{j-1}^L - \alpha_j q_j^L,
$$

where $\alpha_j$ and $\beta_j$ are diagonal and subdiagonal entries of $T^k$. $\beta_j$ is computed so that $q_j^{L+1}$ is a unit vector, and $\alpha_{j+1} = q_j^{L+1} A^{(0,1)^3} q_{j+1}^L$. We initialize the iteration with a random $q_0^L \in \text{span}(A^{(0,1)^3})$. The Lanczos algorithm can be viewed as a modified Gram-Schmidt technique to create an orthonormal basis for the Krylov space associated with $q_0^L$ and $A^{(0,1)^3}$, 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 [34] of orthogonalizing each iterate with all previous Lanczos vectors to be sufficient for our training purposes.

### 5.2 Model Architecture

The internal structure of our CNN architecture for a $128^3$ grid can be seen in Figure 2. Our model 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. Importantly, 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, one 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 up-scaled and added to the layer that is downsampled. The last layer in the network is dense with a linear activation function. The activation functions in all convolutional layers are Rectified Linear Units (ReLU), except for the first convolution, which uses a linear activation function.

Differing resolutions use differing numbers of convolutions, but the fundamental structure remains the same. More precisely, the number of residual connections are changed for different resolutions. For example, a $64^3$ grid uses 1 residual block on the left and 2 residual blocks on the right on the upper level, and 3 residual blocks on the lower level. Furthermore, the weights trained on a lower resolution grid can be used effectively with higher resolutions. Figure 4 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 shown in the figure, even training with a $64^3$ grid allows for efficient residual reduction.

### 5.3 Training

Using the procedure explained in Section 5.1, we create the training dataset $D \in \text{span}(A^{(0,1)^3}) \cap S^{n-1}$ of size 20000 generated from 10000 Rayleigh-Ritz vectors. We train our model with TensorFlow 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 [26] (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 $128^3$. 
6 Results and Analysis

We demonstrate DGCM 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 AMD EPYC 75F3 processors, 512GB RAM, and an NVIDIA RTX A6000 GPU with 48GB memory.

Figure 3 showcases DGCM 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 (with 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 examples use grids with resolution $256^3$ (representing over 16 million unknowns). For each linear solve, DGCM was run until the residual was reduced by four orders of magnitude.

For the bunny example, Figures 4a–b demonstrate how residuals decrease over the course of a linear solve, comparing DGCM 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 implementation provided by [11] (available under the CeCILL-B Free Software License). This implementation includes pre-trained models (usable under the same license) 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 DGCM, 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 two other classical methods, CG and incomplete Cholesky preconditioned CG (ICPCG); clearly, DGCM reduces the residual in the fewest number of iterations (e.g., approximately one order of magnitude fewer iterations than ICPCG). Since FluidNet is not an iterative method and lacks a notion of residual reduction, we treat $r_0$ 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 DGCM compared to standard iterative techniques CG and ICPCG. For both $128^3$ 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. DGCM achieves the desired results in by far the fewest number of iterations at both resolutions. At $256^3$, DGCM performs approximately one order of magnitude faster than CG, suggesting a potentially even wider performance advantage at higher resolutions. ICPCG performs poorly for both resolutions due to its inherent serial nature.

7 Conclusions

We presented DGCM, incorporating CNNs into a CG-style algorithm that yields efficient, convergent behavior for solving linear systems. 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 domains 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
Figure 4: Convergence data for the bunny example. (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 DGCM. (b) Residual plots with CG, ICPCG, FN, and DGCM at frame 150. Dashed and solid lines represent results for $128^3$ and $256^3$, respectively. (c) Decrease in residuals with varying degrees of $A$-orthogonalization ($\ell_s = \ell_{\text{start}}$). (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.

simulations or solving linear systems ahead of time; our Rayleigh-Ritz vector approach enables us to quickly generate very large training datasets, unlike approaches seen in other works.

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. We note that our method is unlikely to work well for dense $A$ since that would likely entail using an impractical number of convolutional layers. An interesting question to consider is how well our method and current models would apply to discrete Poisson matrices arising from non-uniform grids, e.g., quadtrees or octrees.

| Method   | $128^3$ Grid | $256^3$ Grid |
|----------|--------------|--------------|
| DGCM     | 29s          | 58s          |
| CG       | 34s          | 592s         |
| ICPCG    | 59s          | 1459s        |


ded to discrete Poisson matrices arising from non-uniform grids, e.g., quadtrees or octrees.

Acknowledgments and Disclosure of Funding

The authors would like to thank Guido Montúfar for the suggestion to use the weights and biases from networks trained for lower-resolution grids to make predictions on higher-resolution grids.

References

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

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

[3] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner. Learning data-driven discretizations for partial differential equations. Proceedings of the National Academy of Sciences, 116(31):15344–15349, 2019.

[4] C. Beck, M. Hutzenthal, A. Jentzen, and B. Kuckuck. An overview on deep learning-based approximation methods for partial differential equations, 2020.
[5] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Math Comp*, 31(138):333–390, 1977.

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

[7] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: A review. *Acta Mechanica Sinica*, pages 1–12, 2022.

[8] J. Chen, V. Kala, A. Marquez-Razon, E. Gueidon, D. A. B. Hyde, and J. Teran. A momentum-conserving implicit material point method for surface tension with contact angles and spatial gradients. *ACM Trans. Graph.*, 40(4), jul 2021.

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

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

[11] fluidnetsc22. fluidnetsc22/fluidnet_sc22: v0.0.1, April 2022. doi: 10.5281/zenodo.6424901, URL: [https://doi.org/10.5281/zenodo.6424901](https://doi.org/10.5281/zenodo.6424901).

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

[13] F. Gibou, D. Hyde, and R. Fedkiw. Sharp interface approaches and deep learning techniques for multiphase flows. *Journal of Computational Physics*, 380:442–463, 2019.

[14] G. Golub and C. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[15] M. Götz and H. Anzt. 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)*, pages 49–56. IEEE, 2018.

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

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

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

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

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

[21] J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon. Learning neural PDE solvers with convergence guarantees, 2019.

[22] T. Ichimura, K. Fujita, M. Hori, L. Maddegedara, N. Ueda, and Y. Kikuchi. 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)*, pages 61–68, 2020.

[23] J. Jakob, M. Gross, and T. Günther. A fluid flow data set for machine learning and its application to neural flow map interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1279–1289, 2020.

[24] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

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

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

[27] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. 1950.
[28] J. Liang, M. Lin, and V. Koltun. Differentiable cloth simulation for inverse problems. In Advances in Neural Information Processing Systems, volume 32. Curran Associates, Inc., 2019.

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

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

[31] K. Luna, K. Klymko, and J. P. Blaschke. Accelerating GMRES with deep learning in real-time, 2021.

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

[33] H. Nussbaumer. The fast Fourier transform. In Fast Fourier Transform and Convolution Algorithms, pages 80–111. Springer, 1981.

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

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

[36] A. Paluszny and R. W. Zimmerman. Numerical simulation of multiple 3d fracture propagation using arbitrary meshes. Computer Methods in Applied Mechanics and Engineering, 200(9):953–966, 2011.

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

[38] H. Ruelmann, M. Geveler, and S. Turek. On the prospects of using machine learning for the numerical simulation of PDEs: Training neural networks to assemble approximate inverses, 2018.

[39] Y. Saad. Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003.

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

[41] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia. Learning to simulate complex physics with graph networks. In International Conference on Machine Learning, pages 8459–8468. PMLR, 2020.

[42] J. Sappl, L. Seiler, M. Harders, and W. Rauch. Deep learning of preconditioners for conjugate gradient solvers in urban water related problems, 2019.

[43] J. Sirignano, J. F. MacArt, and J. B. Freund. DPM: A deep learning PDE augmentation method with application to large-eddy simulation. Journal of Computational Physics, 423:109811, 2020.

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

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

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

[47] L. Trefethen and D. Bau. Numerical Linear Algebra, volume 50. SIAM, 1997.

[48] E. Tumanov, D. Korobchenko, and N. Chentanez. Data-driven particle-based liquid simulation with deep learning utilizing sub-pixel convolution. Proceedings of the ACM on Computer Graphics and Interactive Techniques, 4(1):1–16, 2021.

[49] K. Um, R. Brand, Y. Fei, P. Holl, and N. Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, Advances in Neural Information Processing Systems, volume 33, pages 6111–6122. Curran Associates, Inc., 2020.
[50] B. Ummenhofer, L. Prantl, N. Thuerey, and V. Koltun. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*, 2020.

[51] S. Wiewel, M. Becher, and N. Thuerey. Latent space physics: Towards learning the temporal evolution of fluid flow. In *Computer graphics forum*, volume 38, pages 71–82. Wiley Online Library, 2019.

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