Title

Novel Applications of Neural Networks in Physics-Based Simulations

Author

Akar, Osman

Publication Date

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Novel Applications of Neural Networks in Physics-Based Simulations

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Mathematics

by

Osman Akar

2024

ABSTRACT OF THE DISSERTATION

Novel Applications of Neural Networks in Physics-Based Simulations

by

Osman Akar
Doctor of Philosophy in Mathematics
University of California, Los Angeles, 2024
Professor Joseph Michael Teran, Co-Chair
Professor Chenfanfu Jiang, Co-Chair

Machine learning, particularly deep neural networks, has become a powerful tool in scientific research. In this thesis, we demonstrate how neural networks can enhance performance and reduce memory requirements in physics-based simulation applications for computer graphics. In the first part of the thesis 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 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 train the network using self-supervised 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, 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.

In the second part we present learning-based implicit shape representations designed for real-time avatar collision queries arising in the simulation of clothing. Signed distance functions (SDFs) have been used for such queries for many years due to their computational efficiency. Recently deep neural networks have been used for implicit shape representations (DeepSDFs) due to their ability to represent multiple shapes with modest memory requirements compared to traditional representations over dense grids. However, the computational expense of DeepSDFs prevents their use in real-time clothing simulation applications. We design a learning-based representation of SDFs for human avatars whose bodies change shape kinematically due to joint-based skinning. Rather than using a single DeepSDF for the entire avatar, we use a collection of extremely computationally efficient (shallow) neural networks that represent localized deformations arising from changes in body shape induced by the variation of a single joint. This requires a stitching process to combine each shallow SDF in the collection together into one SDF representing the signed closest distance to the boundary of the entire body. To achieve this we augment each shallow SDF with an additional output that resolves whether or not the individual shallow SDF value is referring to a closest point on the boundary of the body, or to a point on the interior of the body (but on the boundary of the individual shallow SDF). Our model is extremely fast and accurate and

we demonstrate its applicability with real-time simulation of garments driven by animated characters.

The dissertation of Osman Akar is approved.

Guido Francisco Montufar Cuartas

Christopher R. Anderson

Stanley J. Osher

Chenfanfu Jiang, Committee Co-Chair

Joseph Michael Teran, Committee Co-Chair

University of California, Los Angeles

2024

To my beautiful wife Sena Nur,

my loving parents Serpil and Nail,

my beloved sisters Ceren and Asya Nehir,

and my kind-hearted grandmas Emine and Havva

To my precious family . . .

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Joseph Teran, for his constant support and insightful guidance. His mentorship has been invaluable in my journey. I would also like to extend my thanks to my doctoral committee members, Professor Chenfanfu Jiang, Professor Stanley Osher, Professor Christopher Anderson, and Professor Guido Montufar for their invaluable feedback and inspiration for improving my work.

I am very thankful to my fellow lab members: Yizhou Chen, Yushan Han, Victoria Kala, Ayano Kaneda, Jingyu Chen, and Elias Gueidon-for their collaboration, and discussions which made my research journey more enjoyable and fulfilling. Additionally, I owe a special thanks to Professor David Hyde, whose guidance and generosity in granting me access to his supercomputer and GPU resources were instrumental to my research.

I am deeply grateful to my mentors at Epic Games, Benn Gallagher and Michael Lentine, for sharing their expertise and providing invaluable guidance. Benn's support was especially critical in helping me integrate my project into Unreal Engine. I would also love to thank Epic Games for sponsoring my research.

Finally, I would like to express my endless gratitude to my friends and family for their unconditional love and encouragement throughout these years. A special acknowledgment goes to my father, whose hard work, unwavering dedication and work ethics have always been a source of inspiration to me.

VITA

2019	BS in Mathematics of Computation, UCLA
2019	MA in Mathematics, UCLA
2019-2024	Teaching Assistant, UCLA Mathematics Department
2023 Spring	Instructor, UCLA Mathematics Department
2022-2024	Research Intern, Epic Games, Inc.

PUBLICATIONS

Hallee E Wong, Osman Akar, Emmanuel Antonio Cuevas, Iuliana Tabian, Divyaa Ravichandran, Iris Fu, Cambron Carter. 2018. Markerless Augmented Advertising for Sports Videos. Computer Vision–ACCV 2018 Workshops: 14th Asian Conference on Computer Vision, Perth, Australia, December 2–6, 2018, Revised Selected Papers 14

Ayano Kaneda, Osman Akar, Jingyu Chen, Victoria A. T. Kala, David Hyde, Joseph Teran. 2023. A Deep Conjugate Direction Method for Iteratively Solving Linear Systems. Proceedings of the 40th International Conference on Machine Learning, in Proceedings of Machine Learning Research 202:15720-15736

Osman Akar, Yushan Han, Yizhou Chen, Weixian Lan, Benn Gallagher, Ronald Fedkiw, Joseph Teran. 2024. Shallow Signed Distance Functions for Kinematic Collision Bodies.

arXiv preprint. arXiv:2411.06719

CHAPTER 1

Introduction of Linear Systems

1.1 Linear Systems

Large sparse linear systems often arises in many problems in applied mathematics, especially physics based simulations. These systems can be simply written as

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} \tag{1.1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{b} \in \mathbb{R}^n$ and n corresponds to the spatial fidelity of the computational domain, which can be millions depending on the application. One particular example comes from incompressible flow applications (see 2.3). In general it is very costly (if possible) to find exact solutions of the linear systems with many unknowns, instead, it is leveraged with iterative methods for an approximate solution. Such methods include but not limited to gradient descent method, conjugate gradients (CG) methods, and preconditioned conjugate gradient (PCG) methods. These methods are examples of the *line search method*.

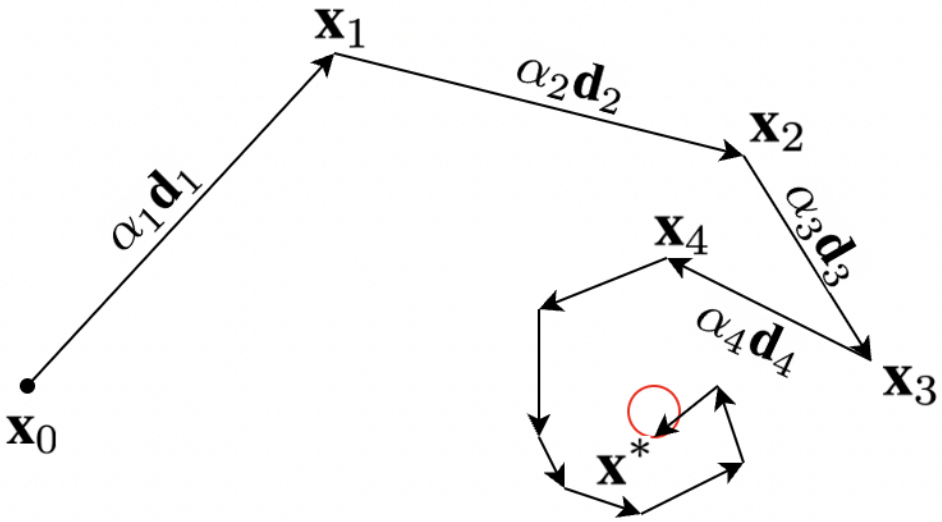


Figure 1.1: Visualization of Line Search Method. Starting from initial guess $\boldsymbol{x}_0$, the iterate is being updated towards the direction $\boldsymbol{d}_k$ with step size α_k until the approximated solution is close enough to the exact solution $\boldsymbol{x}^*$.

1.1.1 Overview Of Iterative Line Search Methods

Line search method is an iterative optimization technique to find the global minima of an objective (e.g. energy) function $h(\boldsymbol{x})$. At each step, the current approximation $\boldsymbol{x}_{k-1}$ is refined towards to the direction $\boldsymbol{d}_k$ with step size α_k. In other words, the k^{th} iterate is chosen as

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

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

$$\alpha_k = \arg\min_{\alpha} h(\mathbf{x}_{k-1} + \alpha \mathbf{d}_k)$$

For the linear systems of interest Equation 1.1, we define

$$h(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \mathbf{A}\boldsymbol{x} - \boldsymbol{x}^T \boldsymbol{b}$$

The global minima of $h(\boldsymbol{x})$ satisfies $\nabla h(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} - \boldsymbol{b} = 0$. Since the hessian matrix of h is $\mathbf{A}$ which positive definite, the global minimum of h solves the equation 1.1.

Natural choice of the search direction is the negative gradient of the objective function $\boldsymbol{d}_k = -\nabla h(\boldsymbol{x}_{k-1}) = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_{k-1}$ which is when the steepest descent occurs. This results in gradient descent algorithm.

Lemma: α_k can be calculated directly as follows:

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

where $\boldsymbol{r}_{k-1} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_{k-1}$ is the $(k-1)^{\text{th}}$ residual.

Proof: Define the another objective function $g(\alpha) : \mathbb{R} \to \mathbb{R}$

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

Taking the derivative with respect to α, we have $g'(\alpha) = \alpha \boldsymbol{d}_k^T \mathbf{A} \boldsymbol{d}_k - \boldsymbol{d}_k^T \boldsymbol{r}_{k-1} = 0$ and $g''(\alpha) = \boldsymbol{d}_k^T \mathbf{A} \boldsymbol{d}_k >= 0$ following from semi positive definiteness of the the matrix $\mathbf{A}$. This yields $\alpha = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \mathbf{A} \boldsymbol{d}_k}$ is a minimizer of $g(\alpha)$.

1.1.2 Conjugate Gradients (CG) Algorithm

Definition: Two vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ are called $\mathbf{A} - orthogonal$ if $\boldsymbol{v}^T \mathbf{A} \mathbf{w} = 0$.

Definition: $\mathbf{A} - norm$ of a vector is defined as

$$||\boldsymbol{v}||_{\mathbf{A}} = \sqrt{\boldsymbol{v}^T \mathbf{A} \boldsymbol{v}}$$

The conjugate gradients method is a special case of the line search method when the $\mathbf{A}$ is

symmetric positive definite matrix. In this method the search directions are chosen to be **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 **A**-orthogonal to all previous search directions:

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

Indeed, the choice of h_{ik} guarantees the new search direction is $\boldsymbol{d}_k$ is $\mathbf{A} - orthogonal$ to all the remaining search direction. 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, and $\boldsymbol{x}^*$ is the exact solution to the equation 1.1. Furthermore, since the search directions are **A**-orthogonal, the optimal step sizes α_k at each iteration satisfy

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

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 [GL12], e.g., compared to gradient descent algorithm. 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 $\boldsymbol{d}_i^T \mathbf{A} \boldsymbol{r}_{k-1} = 0$.

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

Proof:

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

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

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

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 **A**-orthogonal to all previous residuals, i.e., $\boldsymbol{d}_i^T \mathbf{A} \boldsymbol{r}_{k-1} = 0$ for all $i < k-1$.

Proof: $\boldsymbol{r}_i^T \boldsymbol{r}_{k-1} = (\boldsymbol{r}_{i-1}^T - \alpha_i \mathbf{A}\boldsymbol{d}_i)^T \boldsymbol{r}_{k-1}$, hence $\boldsymbol{d}_i \mathbf{A} \boldsymbol{r}_{k-1} = \boldsymbol{r}_{i-1}^T \boldsymbol{r}_{k-1} - \boldsymbol{r}_i^T \boldsymbol{r}_{k-1} = 0$ for all $i < k-1$, using the lemma above. This proves the sparsity of the h_{ik}. Hence CG method has "memoryless" property in a sense that at any iterate only previous two search directions has to be stored in the memory. This "memoryless" idea has also been inherited by **DCDM** algorithm in the second chapter, and enables the efficiency of the method (Figure 2.5-(c) shows enforcing search directions $\mathbf{A} - orhogonal$ to previous directions result in faster convergence). Finally, the CG algorithm can be summarized as follows. The algorithm iterates until the residual is less than ϵ.

Algorithm 1 Conjugate Gradients Algorithm [TB97]

$\boldsymbol{r}_0 = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_0$

$\boldsymbol{d}_1 = \boldsymbol{r}_0$ (Initial Search Direction)

$k = 1$

while $\|\boldsymbol{r}_{k-1}\| \geq \epsilon$ **do**

$\alpha_k = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}{\boldsymbol{d}_k^T \mathbf{A} \boldsymbol{d}_k}$ (Step Length)

$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{d}_k$ (Solution Update)

$\boldsymbol{r}_k = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_k$ (Residual Update)

$\beta_k = \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}$ (Search Direction Correction)

$\boldsymbol{d}_{k+1} = \boldsymbol{r}_k + \beta_k \boldsymbol{d}_k$ (Search Direction Update)

$k = k + 1$

end while

Definition: Condition number for positive definite matrix **A** is defined as

$$\kappa(\mathbf{A}) = \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(A)}$$

where λ_{min} and λ_{max} represents the maximum and minimum eigenvalues.
The error norm after k iterations satisfies [TB97]

$$||\boldsymbol{x}^* - \boldsymbol{x}_k||_{\mathbf{A}} \leq 2 \left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} - 1} \right)^k ||\boldsymbol{x}^* - \boldsymbol{x}_0||_{\mathbf{A}} \tag{1.2}$$

This inequality shows that, although CG performs better than GD, the convergence is slow for matrices with a large condition number. To overcome this, *preconditioners* are used to reduce the condition number.

1.1.3 Preconditioned Conjugate Gradients Algorithm

For any invertible matrix $\boldsymbol{F}$, the solution of the following linear system

$$\boldsymbol{F}^T \mathbf{A} \boldsymbol{F} \boldsymbol{y} = \boldsymbol{F}^T \boldsymbol{b}$$

also solves 1.1 with $\boldsymbol{x} = \boldsymbol{F}\boldsymbol{y}$. Note that the matrix $\boldsymbol{F}^T \mathbf{A} \boldsymbol{F}$ in the new linear system is also positive definite. In PCG algorithm, $\boldsymbol{F}$ is chosen to reduce the condition number of this new equivalent linear system ($\kappa(\boldsymbol{F}^T \mathbf{A} \boldsymbol{F}) < \kappa(\mathbf{A})$), making the CG method converge faster in theory and practice. One would like to choose the matrix $\boldsymbol{F}$ that satisfies $\boldsymbol{F}^T \boldsymbol{F} \approx \mathbf{A}^{-1}$. Although $\boldsymbol{F}$ is present in the linear system, one does not need to have define $\boldsymbol{F}$ explicitly. PCG algorithm only uses the $\boldsymbol{F}^T \boldsymbol{F} = \boldsymbol{M} \approx \mathbf{A}^{-1}$ in the calculation of the search direction, making the algorithm convenient to use. Hence preconditioners can be taught as approximate inverses (implicit or explicit). Some of the well know preconditioners are diagonal (Jacobi) preconditioner, incomplete cholesky, or multigrid preconditioners.

Algorithm 2 Preconditioned Conjugate Gradient Algorithm (Wikipedia)

$\boldsymbol{r}_0 = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_0$

$\mathbf{z}_0 = \boldsymbol{M}\boldsymbol{r}_0$

$\boldsymbol{d}_1 = \mathbf{z}_0$

$k = 1$

while $\|\boldsymbol{r}_{k-1}\| > \epsilon$ **do**

$\alpha_k = (\boldsymbol{r}_{k-1}^T \mathbf{z}_{k-1})/(\boldsymbol{d}_k^T \mathbf{A}\boldsymbol{d}_k)$

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

$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{d}_k \quad (= \boldsymbol{b} - \mathbf{A}\boldsymbol{x}_k)$

$\mathbf{z}_k = \boldsymbol{M}^{-1}\boldsymbol{r}_k$

$\beta_n = (\boldsymbol{r}_k^T \mathbf{z}_k)/(\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1})$

$\boldsymbol{d}_{k+1} = \boldsymbol{r}_k + \beta_k \boldsymbol{d}_k$

$k = k + 1$

end while

CHAPTER 2

A Deep Conjugate Direction Method for Iteratively Solving Linear Systems

2.1 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 [LFO06], thermodynamics [CKM21], or mechanical fracture [PZ11]. In this work, we consider sparse linear systems that arise from discrete Poisson equations in incompressible flow applications [Cho67, FSJ01, Bri08]. For linear systems arising from these diverse applications, we use the notation

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2.1}$$

where the dimension n of the matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ 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 [GL12, Saa03].

The appropriate numerical linear algebra technique depends on the nature of the problem. Direct solvers that utilize matrix factorizations (QR, Cholesky, etc. [TB97]) have op-

timal approximation error, but their computational cost is $O(n^3)$, and they typically require dense storage, even for sparse $\mathbf{A}$. Although Fast Fourier Transforms [Nus81] 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 [Saa03]. 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 [Saa03, HS52, Sti52].

In the present work, we consider sparse linear systems that arise from discrete Poisson equations in incompressible flow applications [Cho67, FSJ01, Bri08]. 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 $\boldsymbol{F}\mathbf{A}\boldsymbol{F}\boldsymbol{y} = \boldsymbol{F}\boldsymbol{b}$, 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) [Ker78] 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 [FSJ01, Bri08]. Multigrid [Bra77] and domain

decomposition [Saa03] 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 [GGB19, GSK16, LGM20]. Others have developed machine learning methods to estimate preconditioners [GA18, Sta20, IFH20] and initial guesses for iterative methods [LKB21, UBF20, ADP20]. [TSS17] and [YYX16] develop *non-iterative* machine learning approximations of the inverse of discrete Poisson equations from incompressible flow.

This chapter 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 $\mathbf{A}$-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 $\mathbf{A}$ 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 $\mathbf{A}$ 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

[TSS17], which is the state-of-the-art learning-based method for these types of problems.

DCDM can be viewed as an improved version of [TSS17], because unlike the non-iterative approaches of [TSS17] and [YYX16], our method can reduce the linear system residuals *arbitrarily*. We showcase our approach with examples that have over 16 million degrees of freedom.

2.2 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 [BHJ20, GHF19, KKL21]). From learning how to discretize PDEs [BHH19] to upsampling the results of low-resolution simulations [JGG20, TKC21], 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 [UPT20, SGP20, WBT19], 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., [LLK19, HKU20, SMF20]). We note the popular physics-informed neural network framework of Karniadakis and colleagues [CMW22, RPK19], which uses automatic differentiation to represent all PDE operators and incorporates physical constraints like conservation laws into the network's loss function. While all of these approaches may produce visually plausible results (e.g., when solving the Poisson equation for a fluid flow simulation), they may not be computationally efficient (e.g., slower than a traditional CFD code), they are inherently limited by their generalizability, and they may not converge to the true solution that would be obtained with a classical algorithm like CG.

Several papers have focused on enhancing the solution of linear systems (arising from

discretized PDEs) using learning. For instance, [GA18] generate sparsity patterns for block-Jacobi preconditioners using convolutional neural networks, and [Sta20] use a CNN to predict non-zero patterns for ILU-type preconditioners for the Navier-Stokes equations (though neither work designs fundamentally new preconditioners). [IFH20] 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. [HZE19] learn an iterator that solves linear systems, performing competitively with classical solvers like multigrid-preconditioned MINRES [PS75]. [LGM20] and [GGB19] 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 $\mathbf{A}$. They train a graph neural network to learn (in an unsupervised fashion) a mapping from matrices $\mathbf{A}$ to prolongation operators. [GSK16] 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, [LKB21] accelerate the convergence of GMRES [SS86] with an initial guess that is learned in real-time (i.e., as a simulation code runs) with no prior data. [UBF20] 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, [ADP20] 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 [RGT18, SSH19] have sought to learn a mapping between a matrix

and an associated sparse approximate inverse. In their investigation, [RGT18] 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. [SSH19] 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, [ADG16] learn a gradient descent algorithm, [LM16] provide a general reinforcement learning framework for learning optimization algorithms, [SCH19] apply L2O to minimax problems, and [LDF22] perform online meta-learning of quasi-Newton optimization methods. We refer the reader to [CCC22] for a recent review of L2O techniques.

Most relevant to the present work is FluidNet [TSS17]. 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

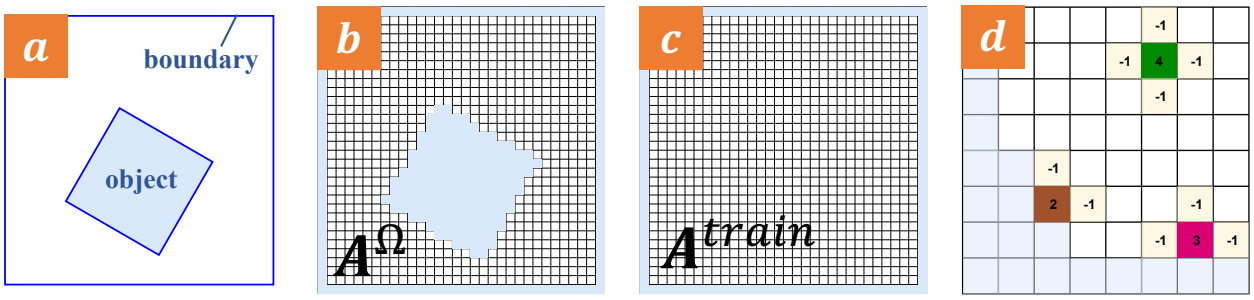


Figure 2.1: **(a)** We illustrate a sample flow domain $\Omega \subset (0,1)^2$ (in 2D for ease of illustration) with internal boundaries (blue lines). **(b)** We voxelize the domain with a regular grid: white cells represent interior/fluid, and blue cells represent boundary conditions. **(c)** We train using the matrix $\mathbf{A}^{\text{train}}$ 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 $\mathbf{A}^{\Omega}$ 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 $\mathbf{A}^{\Omega}$ in rows corresponding to boundary/blue cells are zero. The numbers shown are for the 2D case, in 3D case a box with 6 neighboring white cells has one 6 and six -1s in its corresponding row.

precision). Accordingly, FluidNet's lack of convergence stands in stark contrast to classical, convergent methods like CG. Our method resolves this gap.

2.3 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 [Cho67] for the inviscid, incompressible Euler equations. These equations

are

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t}+\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}\boldsymbol{u}\right)+\nabla p=\boldsymbol{f}^{ext},\qquad \nabla\cdot\boldsymbol{u}=0 \tag{2.2}$$

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 2.2 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 2.2 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 [Cho67]. Using finite differences in time, we can summarize this as

$$\rho^0\left(\frac{\boldsymbol{u}^*-\boldsymbol{u}^n}{\Delta t}+\frac{\partial \boldsymbol{u}^n}{\partial \boldsymbol{x}}\boldsymbol{u}^n\right)=\boldsymbol{f}^{ext} \tag{2.3}$$

$$-\nabla\cdot\frac{1}{\rho^0}\nabla p^{n+1}=-\nabla\cdot\boldsymbol{u}^* \tag{2.4}$$

$$-\frac{1}{\rho^0}\nabla p^{n+1}\cdot\boldsymbol{n}=\frac{1}{\Delta t}\left(u^{\partial\Omega}-\boldsymbol{u}^*\cdot\boldsymbol{n}\right). \tag{2.5}$$

For the spatial discretization, we use a regular marker-and-cell (MAC) grid [HW65] 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 [FSJ01, GHM20] for Equation 2.3. All spatial partial derivatives are approximated using finite differences. Equations 2.4 and 2.5 describe the pressure Poisson equation with Neumann conditions on the boundary of the flow domain. We discretize the left-hand side of Equation 2.4 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 2.5. We refer the reader to [Bri08] for more in-depth implementation details. Equation 2.5 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 2.1.

We use the following notation to denote the discrete Poisson equations associated with Equations 2.4–2.5:

$$\mathbf{A}^{\Omega}\boldsymbol{x} = \boldsymbol{b}^{\nabla\cdot\boldsymbol{u}^*} + \boldsymbol{b}^{u^{\partial\Omega}}, \tag{2.6}$$

where $\mathbf{A}^{\Omega}$ 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 2.4 and 2.5, respectively. $\mathbf{A}^{\Omega}$ in Equation 2.6, is a large, sparse, SPD linear system. These linear system require many number of iterations to solve with traditional iterative solvers to produce an adequately small residual. The computational complexity of solving Equation 2.6 strongly depends on data (e.g., internal boundary conditions in the flow domain, see 2.1).

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 [TSS17], 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) $\boldsymbol{A}^{\text{train}}$.

2.4 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 2.6. 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 2.6. We build on the method of CG, which requires the matrix $\mathbf{A}^{\Omega}$ in Equation 2.6 to be SPD. SPD matrices $\mathbf{A}^{\Omega}$ give rise to the matrix norm $\|\boldsymbol{y}\|_{\mathbf{A}^{\Omega}} = \sqrt{\boldsymbol{y}^T \mathbf{A}^{\Omega} \boldsymbol{y}}$. 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 $\boldsymbol{x}$:

$$\begin{aligned} \alpha_k &= \arg\min_{\alpha} \frac{1}{2} \left\| \boldsymbol{x} - (\boldsymbol{x}_{k-1} + \alpha \boldsymbol{d}_k) \right\|_{\mathbf{A}^{\Omega}}^2 \\ &= \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \mathbf{A}^{\Omega} \boldsymbol{d}_k}, \end{aligned} \tag{2.7}$$

[1]For a comprehensive background on CG, see Section 1.1.2.

where $\boldsymbol{r}_{k-1} = \boldsymbol{b} - \mathbf{A}^{\Omega}\boldsymbol{x}_{k-1}$ is the $(k-1)^{\text{th}}$ residual (see section 1.1.1 for details). Note that this objective function is equivalent to the one defined in 1.1.1. 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), $\boldsymbol{d}_k = -\frac{1}{2}\nabla \|\boldsymbol{x}_{k-1}\|^2_{\mathbf{A}^{\Omega}} = \boldsymbol{r}_{k-1}$, 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 more effective strategy by instead choosing directions that are $\mathbf{A}$-orthogonal (i.e., $\boldsymbol{d}_i^T \mathbf{A}^{\Omega} \boldsymbol{d}_j = 0$ for $i \neq j$). More precisely, the search direction $\boldsymbol{d}_k$ is chosen as follows:

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

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 $\mathbf{A}$-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 $(\mathbf{A}^{\Omega})^{-1}\boldsymbol{r}_{k-1}$, whereby $\boldsymbol{x}_k$ will be equal to $\boldsymbol{x}$ since α_k (computed from Equation 2.7) will step directly to the solution. We can see this by considering the residual and its relation to the search direction:

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

In light of this, we use deep learning to create an approximation $\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r})$ to $(\mathbf{A}^{\Omega})^{-1}\boldsymbol{r}$, 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 $(\mathbf{A}^{\Omega})^{-1}\boldsymbol{r}$ since the step size α_k will account

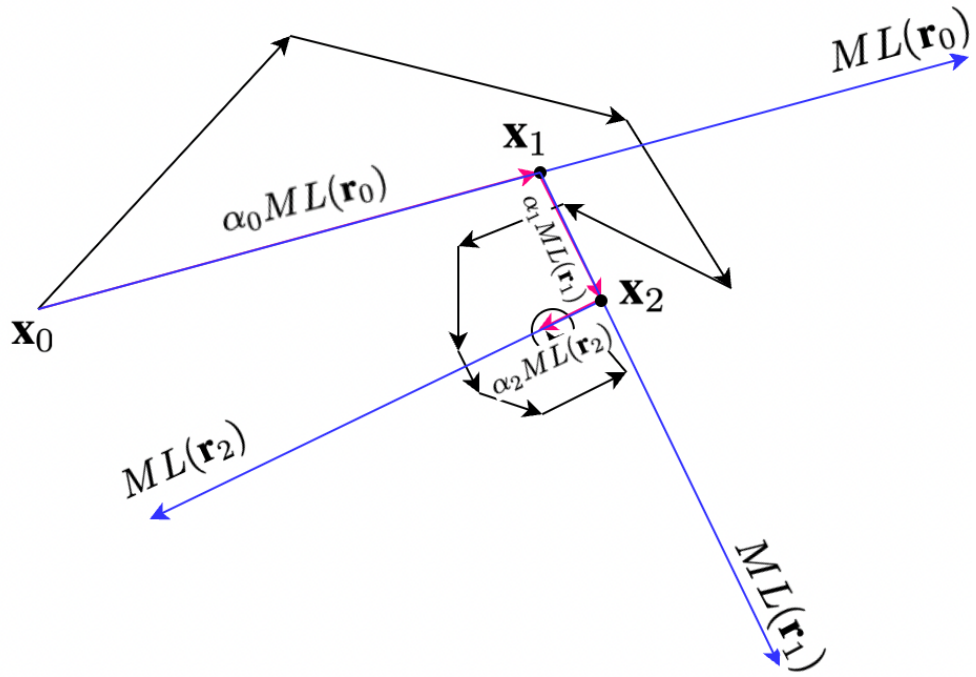


Figure 2.2: Visualization of DCDM and regular Line Search Methods. The blue arrows represents the neural network outputs for the search direction at the current iterate ($ML(\boldsymbol{r}_k) = \boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}_k)$). Search directions generated by the network results in faster convergence compared to CG method.

for any scaling in practice. In other words, $\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}) \approx s_{\boldsymbol{r}}(\mathbf{A}^{\Omega})^{-1}\boldsymbol{r}$, 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 $\mathbf{A}$-orthogonality, yielding search directions

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

We summarize our approach in Algorithm 3. Note that we introduce the variable i_{start}. To guarantee $\mathbf{A}$-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 2.5c. This choice significantly reduces the required number of search directions to be stored, making our algorithms memory requirement comparable to the CG.

Algorithm 3 DCDM

$\boldsymbol{r}_0 = \boldsymbol{b} - \mathbf{A}^{\Omega}\boldsymbol{x}_0$

$k = 1$

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**

$h_{ik} = \frac{\boldsymbol{d}_k^T \mathbf{A}^{\Omega} \boldsymbol{d}_i}{\boldsymbol{d}_i^T \mathbf{A}^{\Omega} \boldsymbol{d}_i}$

$\boldsymbol{d}_k$-=$h_{ik}\boldsymbol{d}_i$

end for

$\alpha_k = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T \mathbf{A}^{\Omega} \boldsymbol{d}_k}$

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

$\boldsymbol{r}_k = \boldsymbol{b} - \mathbf{A}^{\Omega}\boldsymbol{x}_k$

$k = k + 1$

end while

2.5 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

$$\boldsymbol{f}(\boldsymbol{c}, \boldsymbol{r}) \approx s_{\boldsymbol{r}} (\mathbf{A}^{\Omega})^{-1} \boldsymbol{r}$$

(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.

2.5.1 Loss Function and Self-supervised Learning

Although we generalize to arbitrary matrices $\mathbf{A}^\Omega$ from Equation 2.6 that correspond to domains $\Omega \subset (0,1)^3$ that have internal boundaries (see 2.1), 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. 2.1(c).

In contrast, other similar approaches [TSS17, YYX16] train using matrices $\mathbf{A}^\Omega$ 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 2.7. 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

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

for a given dataset $\mathcal{D}$ consisting of training vectors $\boldsymbol{b}^i$. In Algorithm 3, the normalized residuals $\frac{\boldsymbol{r}_k}{\|\boldsymbol{r}_k\|}$ are passed as inputs to the model. Unlike in e.g. FluidNet [TSS17], 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 3 are skewed to the lower end of the spectrum of the matrix $\mathbf{A}^\Omega$. Since $\mathbf{A}^\Omega$ 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 2.8.2). 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 2.4).

We generate the Rayleigh-Ritz vectors by first tridiagonalizing the training matrix $\boldsymbol{A}^{\text{train}}$ with m Lanczos iterations [Lan50] 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 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 [Pai71] 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 2.8.2 for more detail).

We reiterate that since DCDM generalizes to various Poisson systems (see Sections 2.5.2 and 2.6) 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.

2.5.2 Model Architecture

The internal structure of our CNN architecture for a 128^3 grid is shown in 2.3. It consists of a series of convolutional layers with residual connections. The upper left of 2.3 (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

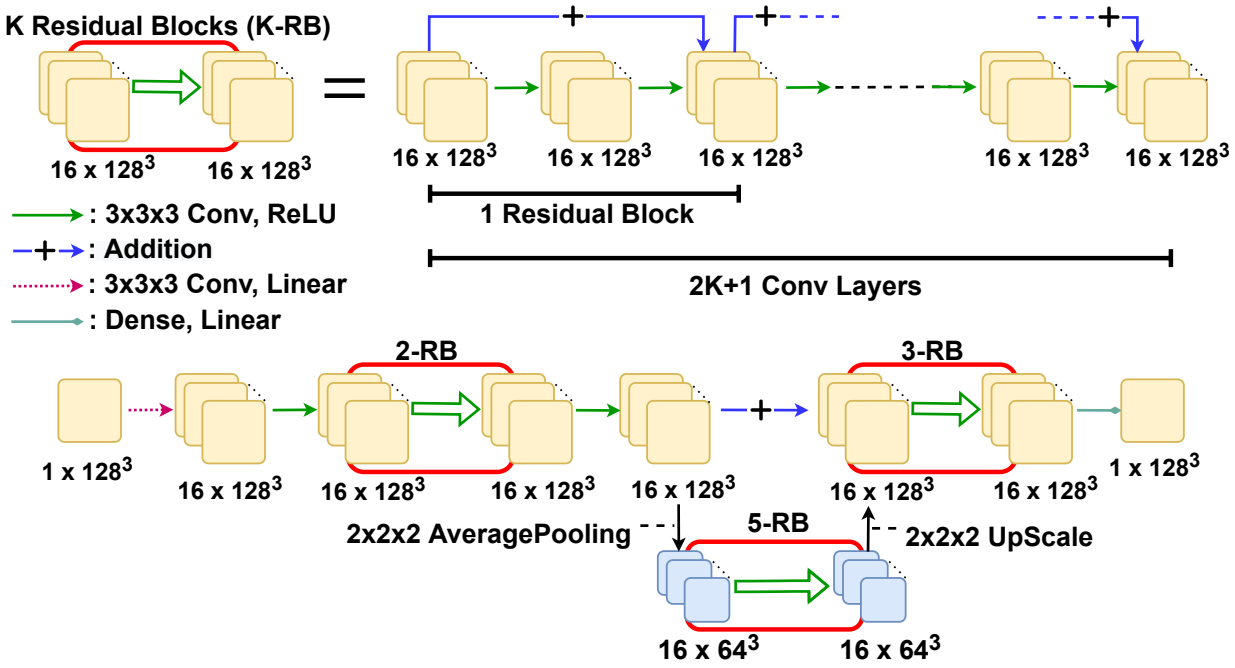


Figure 2.3: Architecture for training with $\boldsymbol{A}^{\text{train}}$ on a 128^3 grid.

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 [HZR16]). 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 direc-

tion, we included a brief ablation study in Appendix 2.8.2. 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. 2.5d 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. 2.1 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.

2.5.3 Training

Using the procedure explained in Section 2.5.1, 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. Hence input of the model is regularized, and note that the DCDM algorithm also normalizes all residual inputs. We train our model with TensorFlow [AAB15] on a single NVIDIA RTX A6000 GPU with 48GB memory. Training is done with standard deep learning techniques—more precisely, with back-propagation and the ADAM optimizer [KB15] (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.

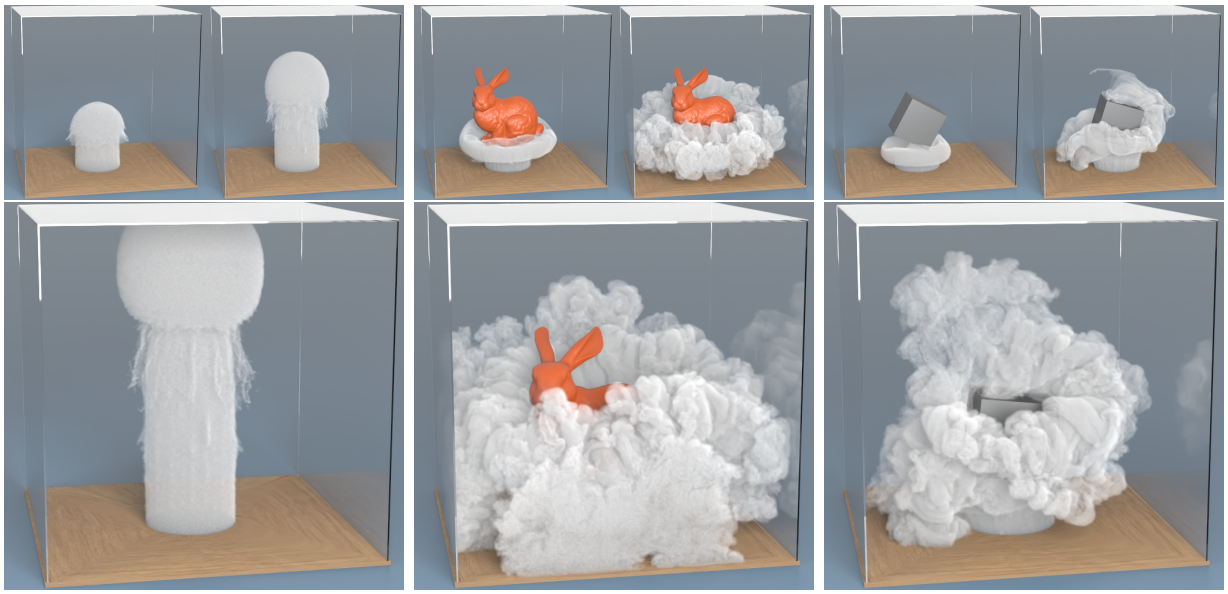


Figure 2.4: DCDM for simulating a variety of incompressible flow examples. Left: smoke plume at $t = 6.67, 13.33, 20$ seconds. Middle: smoke passing a bunny at $t = 5, 10, 15$ seconds. Right: smoke passing a spinning box (time-dependent Neumann boundary conditions) at $t = 2.67, 6, 9.33$ seconds.

2.6 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. [TSS17] and are common for graphics papers.

Figure 2.4 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 method like DCDM will have a more dramatic impact.

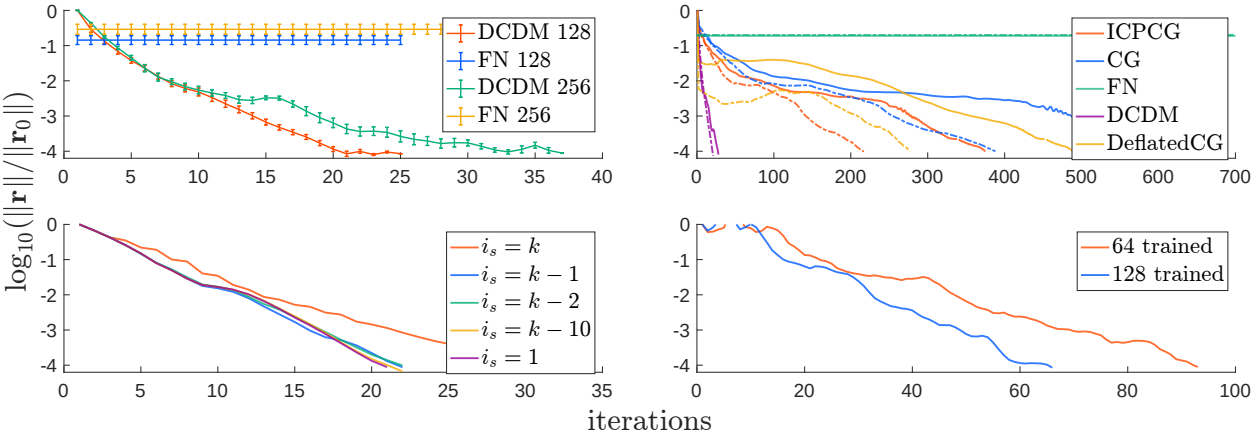


Figure 2.5: Convergence data for the bunny example (see also Table 2.1). **(a)** Mean and std. dev. (over all 400 frames in the simulation) of residual reduction during linear solves (with 128^3 and 256^3 grids) using FluidNet (FN) and DCDM. **(b)** Residual plots with ICPCG, CG, FN, DCDM, and Deflated 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 **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 2.4 Middle.

For the bunny example, Figures 2.5a–b demonstrate how residuals decrease over the

[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) [MST10, PGG23].

	64^3 Grid			128^3 Grid			256^3 Grid		
Method	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 2.1: 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 2.8.1 for convergence plots.

course of a linear solve, comparing DCDM with other methods. Figure 2.5a shows the mean results (with standard deviations) over the course of 400 simulation frames, while in Figure 2.5b, we illustrate behavior on a particular frame (frame 150). For FluidNet, we use the optimized implementation provided by [flu22]. 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 2.5b, we also visualize the convergence of three other classical methods, CG, Deflated CG [SYE00], 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 2.1 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 (2.5b), we found the runtime to scale prohibitively with grid resolution. We used SciPy's [VGO20] `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 2.8.1 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 tested cases where d = 2, at resolutions 256^2 and 512^2. For this setup, running the smoke plume test (2D analogue of 2.4 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 2.1, 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, [SSH19]. Since [SSH19] requires many asymptotically expensive computations (see Section 2.2), we expected a significant performance advantage with DCDM. For the 256^2 smoke plume example, using matrices from frame 10 of the simulation, [SSH19] requires 1024 iterations for convergence (15.41s), vs. only 50 for DCDM (1.50s).

2.7 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

[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.

at different grid resolutions: `https://github.com/ayano721/2023_DCDM`.

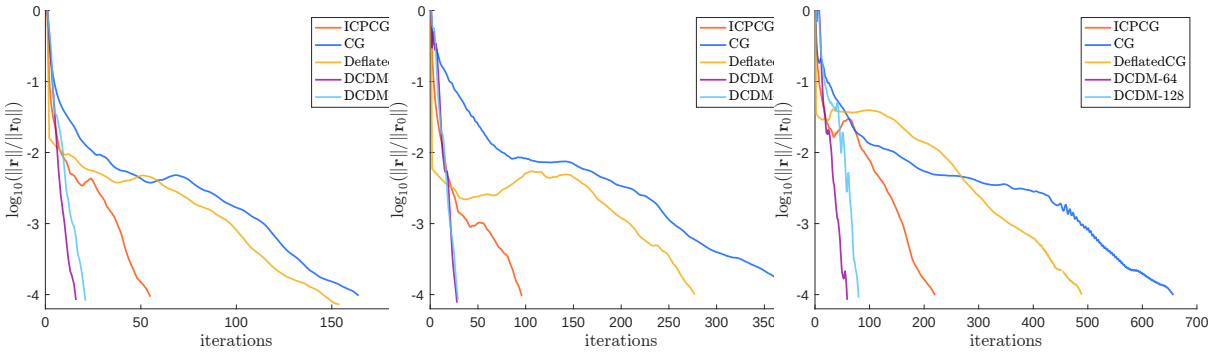
Our network was designed for and trained exclusively using data related to the discrete Poisson matrix, which likely limits the generalizability of our present *model*. However, we believe our *method* is readily applicable to other classes of PDEs (or general problems with graph structure) that give 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 $\mathbf{A} * x$ (e.g., dense matrices), since training relies on efficient $\mathbf{A} * \boldsymbol{x}$ 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 [LGF04].

2.8 Additional Results and Model Architecture Discussion

2.8.1 Additional Convergence Results

We include additional convergence results, similar to those shown in Figure 2.5b, in Figure 2.6. Specifically, these plots show the convergence of all the methods reported in Table 2.1 at each of the resolutions reported there. The figure visually demonstrates the significant reduction in iteration count achieved by DCDM.

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 $\mathbf{A}$ must be re-



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

Figure 2.6: Convergence of different methods on the 3D bunny example for $N = 64$, 128, 256; summary results, as well as timings, are reported in 2.1. DCDM-{64,128} calls a model whose parameters are trained over a $\{64^3, 128^3\}$ grid.

computed. Recomputation is also required in the approach of [TSS17] in examples like these. Moreover, as 2.5d 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.

2.8.2 Ablation Study and Runtime Analysis

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

Table 2.2: 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 (2.9a to 2.9e), several of which are modifications of the

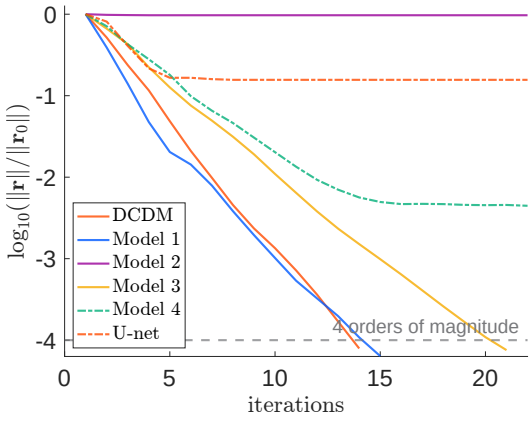


Figure 2.7: Residual plot for the bunny example at $N = 64$ with each trained model. The dashed line represents a four-orders-of-magnitude reduction in residual, which is the convergence criterion we use throughout our examples.

model we ultimately used to generate our results (2.9a). The models we considered include one without ResNet connections (2.9b), one with simple downsampling and upsampling (a U-Net-like structure) (2.9c), a minimal CNN (2.9d), and a model with different filter sizes of the blocks (2.9e). We compared how these models perform on the same bunny example considered in the main part of the paper (at resolution 64^3). 2.7 shows that the architecture we ultimately selected for DCDM yields the best results.

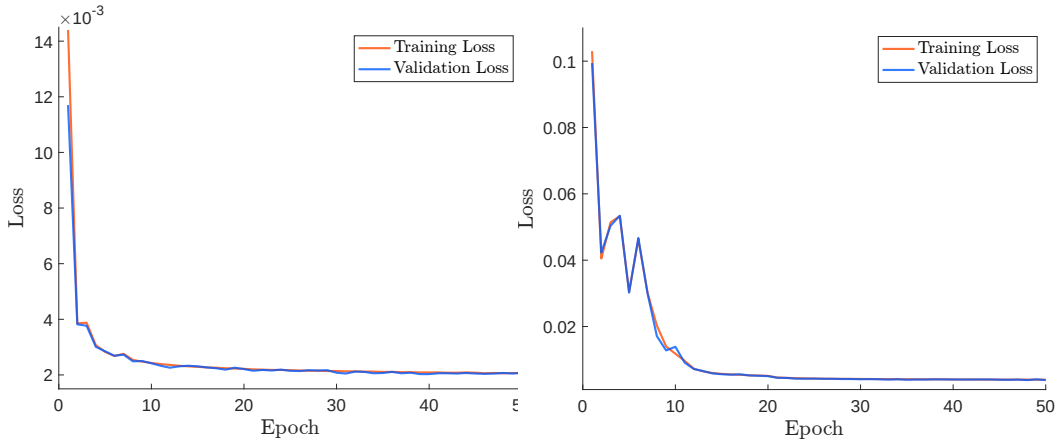
Each model's parameter count is listed in 2.2. Compared to a basic CNN or U-Net architecture (like the one used in [TSS17]), our DCDM network is actually quite light. For example, the U-Net architecture in [TSS17] uses 3,527,505 parameters (at $N = 64$ in 3D), while our network (at the same resolution) 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 2.5.1) to take $O(n^3m^2)$ time, where m is the number of Lanczos vectors to be created and n is the 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 2.5.2.) 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.

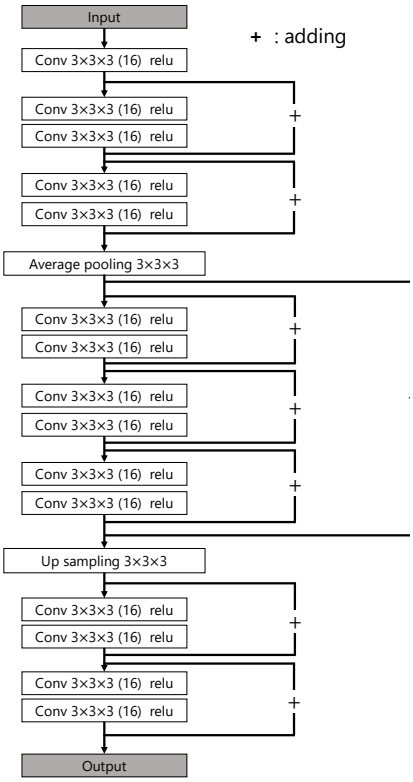
2.8.3 Model training

Figure 2.8 shows the decrease in training and validation losses observed when training the neural network used for DCDM. As mentioned in Section 2.5.3, 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 2.6. 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.

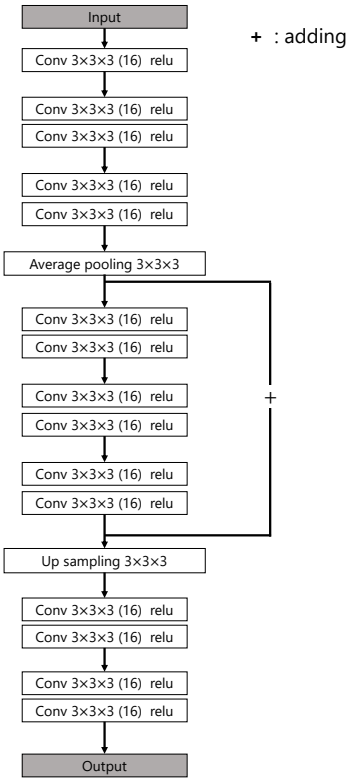


(a) $N = 64$ (b) $N = 128$

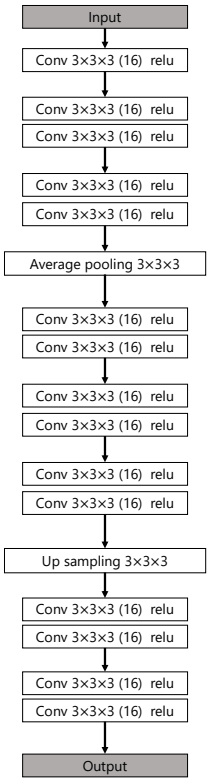
Figure 2.8: Training and validation loss for the networks used in DCDM at resolutions $N = 64$ and $N = 128$.



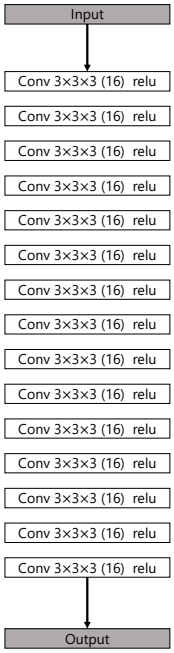
(a) DCDM (our model)



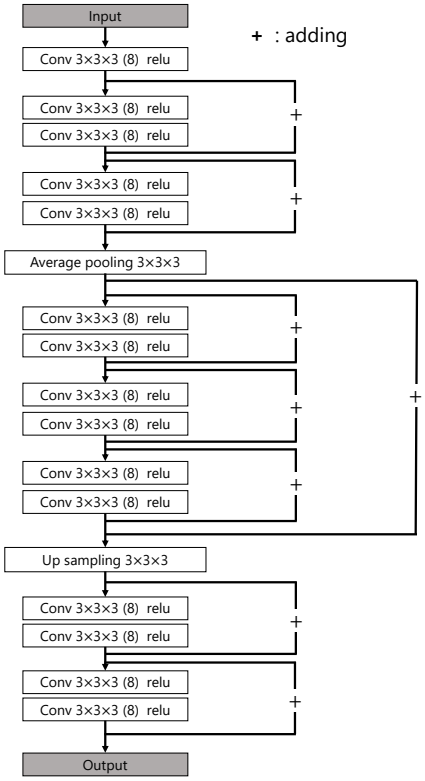
(b) Model 1



(c) Model 2



(d) Model 3



(e) Model 4

Figure 2.9: Network architectures considered for our ablation study.

CHAPTER 3

MLLevelSets, or Shallow Signed Distance Functions for Kinematic Collision Bodies

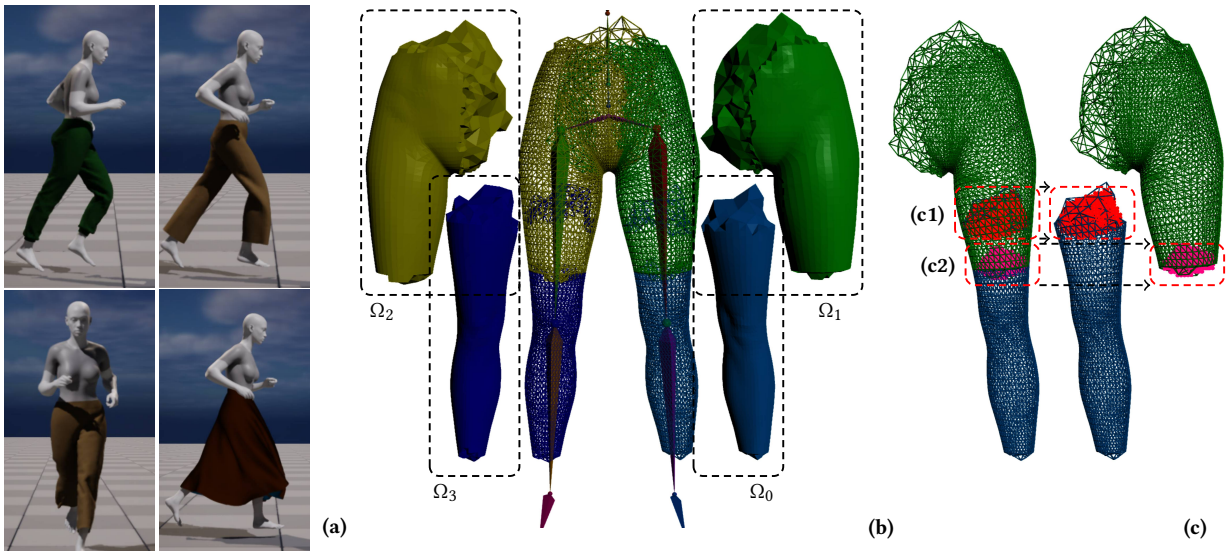


Figure 3.1: **Overview**. **(a)** Learning based SDFs are used in cloth simulations in real time. **(b)** Our method partitions the character domain into subregions, and each region is represented by very fast shallow generalized Multi Layer Perceptron (MLP) neural networks. **(c)** Combining multiple SDFs requires additional information if the queried point is closer to interior boundary or the correct boundary. **(c1,c2)** highlights the region for the knee and the thigh in which the points inside are closer to the interior boundary than the avatars' boundary. The regions are also determined by a separate neural network.

3.1 Introduction and Related Work

Simulation of deformable objects is ubiquitous in modern computer graphics applications. Whether it is the intricate stretching and folding of cloth, or the squash, stretch and contraction of soft tissues in virtual characters, elastic deformation is essential for creating satisfactory visual realism in modern visual effects and video games. Simulation of this type is complex and computationally expensive in general. The most challenging aspects are generally self-collision detection/resolution [BFA02, BW98, Wan14, WWY20] and the rapid solution of large systems of nonlinear equations [TWT16, WLF18, WY16]. However, another important aspect is detection and resolution of collision constraints with kinematic geometric objects in the scene. These kinematic objects are not influenced by the deformable objects in the scene (e.g. due to large mass ratios), however their motion often determines the deformation of the elastic objects of interest. Clothing draping and interacting with a kinematic body shape is perhaps the most important example of this, and it is the focus of our approach. An example of this is a walking human avatar with garments on such as pants or skirt (See Figure 3.1-a and 3.12).

In most simulation techniques, elastic object collision with kinematic collision bodies is imposed as a constraint on the governing equations. These constraints are detected and enforced using a variety of geometric descriptions of the collision body. We focus on the use of machine learning techniques for accelerating this process. Although many recent methods have investigated the use of these techniques in clothing simulation, most replace the simulation process altogether with a neural network. While fast, learning techniques are still limited in accuracy compared to simulation, particularly in the case of free flowing cloth with significant inertia. Our aim is to avoid these limitations by replacing just the kinematic body collision portion of a typical cloth simulation pipeline with machine learning enhancement. Our proposed approach is unique in this way, however we briefly discuss a number or recent techniques that utilize machine learning in relevant ways. Romero et al.

[RCC23] use a neural network to learn a displacement mapping for resolving elastic object collisions against kinematic rigid bodies with reduced models for deformable objects. Tan et al. [TZW22] note that purely-learning based cloth techniques suffer greatly from self and kinematic body collision artifacts. Santesteban et al. [STO21] also address this problem by adding a repulsion term into their loss function so that trained cloth will be less likely to penetrate the body. Betiche et al. [BME20, BMT21] also add body-collision based loss terms into training to discourage cloth/body penetration. Gundogdu et al. [GCS19] do this as well.

Signed distance functions (SDFs) generally have constant query time (e.g. when pre-computed and stored over dense grids) and are very useful when rapidly detecting and resolving collisions between a kinematic animated body and simulated clothing [OF03]. However, SDF calculation (e.g. over dense grid nodes) is expensive and is therefore usually done as an offline/pre-computation. Furthermore, while pre-computed SDFs over regular grids are effective, there are some notable drawbacks. First, SDFs are usually pre-computed at frame-rate time intervals since sub-frame time steps would require even more excessive computation of the SDFs (even as a pre-computation). Temporal interpolation of SDFs can be used for sub-frame time queries [SLF08]. Dynamic time step sizes (e.g. resulting from CFL conditions with explicit or semi-implicit time stepping) cannot be known a priori and require this sub-frame interpolation of precomputed SDFs (or excessive non-pre-computation of SDFs on the fly). Also, the storage cost of SDFs at each frame in an animation for each character in the scene quickly becomes excessive, especially for denser grids. Lastly, the motion of the character must be known completely before the simulation is carried out if pre-computation is to be used. While this is a reasonable assumption for some applications (e.g. offline visual effects), it is not possible in real-time simulation environments where the user is actively redefining the character motion on the fly.

Recently, a variety of neural network models for SDFs have been proposed. [OCD22, KFB23]

use neural networks to approximate signed distance functions for scene reconstruction for robotics in real time.[GCV19] learn a general shape template from data. [SCT20] use meta-learning to perform the same task as [PFS19], representing multiple 3D shapes. [LWY23] use unsigned distance function for shape construction for volume rendering. Ma et al. [BZY21] use neural networks to learn SDF representations of point clouds. Chabra et al. [CLI20] utilize local shape patches to increase the variety of shapes representable with neural SDFs. The Deep SDF approach of Park et al. [PFS19] is particularly powerful. In this case, a network is trained to represent a discrete collection of shapes by training over their individual SDFs (sampled over regular grids). By using encoder-decoder network architecture, encoded with a representative shape vector for each object is learned in the process. These functions can represent a wide range of shapes and utilize dramatically less memory than a collection of SDFs defined over regular grids. For example, Deep SDF can store shape information of thousands of 3D chair shapes in a neural network model that use only 7.4 MB (megabytes), whereas each chair requires 16.8 MB (the grid size of the uncompressed 3D bitmap of a SDF of a single shape is 512^3). In the context of representing the body shape of a kinematic animated character, DeepSDFs could be used to model the SDF of the skin surface rigged with joint-based skinning (e.g. linear blend skinning) over a *discrete* collection of joint states. However, real-time simulation in this context requires collision queries against the shape of the kinematic avatar skin at *continuous* samples of the joint state since animation states cannot be discretely sample a priori.

To enable application of learning-based SDF techniques in real-time clothing simulation, we design a neural network SDF that depends continuously on both the collision query point and the kinematic joint-state vector. Rather than using a single DeepSDF defined over the entire body, we use a collection of extremely shallow and computationally efficient networks that represent the skin surface very accurately near individual joints. Figure 3.1-(b) shows an example of partitioning of the lower body into four subregions corresponding to the fol-

lowing joints :left knee, left thigh, right thigh and right knee. This joint-local approach efficiently focuses network degrees of freedom where they are needed and allows for additive scaling complexity of training data burden (in the number of joints). That is, each joint network can be trained separately without the need to couple the effect of distant joints. However, by decoupling into joint-centric shallow SDFs we lose some information about the signed distance to the surface of the complete skinned character since each joint SDF refers to only a portion of the character. This means that the joint-centric SDF zero-isocontours may coincide with the true boundary or may coincide with an internal boundary specific to the joint (see Figure 3.2). We correct for this by training our networks to know whether or not the signed-distance value is associated with a true boundary or an internal boundary. More precisely, each joint is associated with two neural networks, one approximates SDF for the local subregion, and the other flags if the computed signed distance is true signed distance of the entire body. This knowledge allows us to blend the join-centric SDFs into an efficient and accurate SDF for the skin of the character.

Linear blend skinning (LBS) [MLT89] is an effective and widely-used means for defining the skin surface of animated characters from a kinematic joint state. However, the LBS surface is not guaranteed to be self-intersection free which complicates the definition of a SDF representation (e.g. near joints with large ranges of motion like the elbow and knee). We compensate for this by training on surfaces that have had LBS self-collisions resolved in a simulation post-process. We demonstrate the accuracy and efficiency of our approach with real-time simulation of clothing colliding against representative animated skin surfaces of human avatars. In summary, our contributions can be listed as:

- Learning-based SDFs that vary continuously with the kinematic joint state of animated characters.
- Shallow joint-centric neural networks trained to represent local skin deformation.

- A boolean variable returned by each joint-centric shallow SDF that indicates whether a query point is associated with a fictitious joint-internal surface or the global skin boundary.
- A blending mechanism for computing the SDF to the union of the regions defined by each joints shallow SDF.
- Resolution of self-collision artifacts in the SDF of LBS surfaces.

3.2 Character Kinematics

We assume that the kinematics of the animated character are define by a joint-state vector $\boldsymbol{\theta} \in \mathbb{R}^{\hat{N}_J}$ where $\hat{N}_J$ denotes the number of joint degrees of freedom in the animation rig. We use $\boldsymbol{\theta}_i \in \mathbb{R}^{D_i}$, $1 \leq i \leq N_J$ to denote the individual the joint-state vector components where $1 \leq D_i \leq 3$ depending on the type of joint. N_J is the total number of joints in the character. For simple pin joints (e.g. knee), $D_i = 1$ but for a general revolute joint $D_i = 3$. Our neural network model is able to capture all three degrees of freedom, but that requires deeper models which increases the inference time. Note that we did not consider the fully-general case of a 6-degree of freedom rigid joint in this work, and note that we assumed that the distance between joints do not change over time (e.g. there is no transformation). With this convention we have the joint state vector of the entire body

$$\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_{N_J})^T \in \mathbb{R}^{\hat{N}_J}$$

where $\hat{N}_J = \sum_{i=1}^{N_J} D_i \leq 3N_J$. We further assume that the kinematics of the character motion are defined in terms of a deformation mapping $\boldsymbol{\phi} : \Omega \times \mathbb{R}^{\hat{N}_J} \to \mathbb{R}^3$ where $\Omega \subset \mathbb{R}^3$ is the three dimensional domain of the interior of the character in a reference pose. We use

$$\Omega^{\boldsymbol{\theta}} = \left\{ \boldsymbol{x} \in \mathbb{R}^3 | \exists \mathbf{X} \in \Omega \text{ such that } \boldsymbol{\phi}(\mathbf{X}, \boldsymbol{\theta}) = \boldsymbol{x} \right\}$$

to denote the interior region of the animated state of the body (given joint state $\boldsymbol{\theta}$). In our examples we define $\boldsymbol{\phi}(\mathbf{X}, \boldsymbol{\theta}) = \boldsymbol{\phi}^{\mathrm{C}}(\mathbf{X}, \boldsymbol{\phi}^{\mathrm{LBS}}(\mathbf{X}, \boldsymbol{\theta}))$ where $\boldsymbol{\phi}^{\mathrm{LBS}}(\mathbf{X}, \boldsymbol{\theta}) : \Omega \times \mathbb{R}^{\hat{N}_J} \to \mathbb{R}^3$ is the

standard linear blend skinning and $\boldsymbol{\phi}^{\mathrm{C}} : \Omega \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is a collision corrective that resolves collision/pinching in $\boldsymbol{\phi}^{\mathrm{LBS}}$ so that $\boldsymbol{\phi}(\cdot, \boldsymbol{\theta}) : \Omega \rightarrow \Omega^{\boldsymbol{\theta}}$ is always bijective. The $\boldsymbol{\phi}^{\mathrm{C}}$ corrective on LBS is defined in Section 3.5 and assures that the SDF of the animated state has ample room for clothing to surround the character body.

We partition the reference character domain $\Omega \subset \mathbb{R}^3$ into subregions $\Omega_i \subset \Omega$ associated with each joint $0 \leq i < N_J$ such that $\Omega = \cup_{i=1}^{N_J} \Omega_i$. Each subregion Ω_i is the portion of the interior of the character (in the reference pose) that is most deformed by changes in the joint state. Generally, each Ω_i is determined based on proximity to the joint transformation center. Note that these subsets are not intersection free in general $\Omega_i \cap \Omega_j \neq 0$ as nearby joints will influence similar regions. We use

$$\Omega_i^{\boldsymbol{\theta}} = \left\{ \boldsymbol{x} \in \mathbb{R}^3 | \exists \mathbf{X} \in \Omega_i \text{ such that } \boldsymbol{\phi}(\mathbf{X}, \boldsymbol{\theta}) = \boldsymbol{x} \right\}$$

to denote the animated state of the joint sub-region.

3.3 Signed Distance Function

We define the signed distance to the surface of the animated character as $\phi : \mathbb{R}^3 \times \mathbb{R}^{\hat{N}_J} \rightarrow \mathbb{R}$. Here $|\phi(\boldsymbol{x}, \boldsymbol{\theta})|$ denotes the distance from a point $\boldsymbol{x} \in \mathbb{R}^3$ to the closest point on the skin surface of the character in the animated state defined by the joint state vector $\boldsymbol{\theta}$. The sign of $\phi(\boldsymbol{x}, \boldsymbol{\theta})$ indicates whether the point $\boldsymbol{x}$ is inside the skin surface or outside. The closest point on the skin surface to the point $\boldsymbol{x}$ is determined as

$$\boldsymbol{x} - \phi(\boldsymbol{x}, \boldsymbol{\theta}) \nabla^{\boldsymbol{x}} \phi(\boldsymbol{x}, \boldsymbol{\theta})$$

where $\nabla^{\boldsymbol{x}} \phi(\boldsymbol{x}, \boldsymbol{\theta})$ is the gradient of $\phi(\boldsymbol{x}, \boldsymbol{\theta})$ with respect to $\boldsymbol{x}$. Our idea is to approximate signed distance function $\phi(\boldsymbol{x}, \boldsymbol{\theta})$ in terms of a collection of joint-wise augmented local signed distance functions

$$\phi_i : \mathbb{R}^3 \times \mathbb{R}^{D_i} \rightarrow \mathbb{R}$$

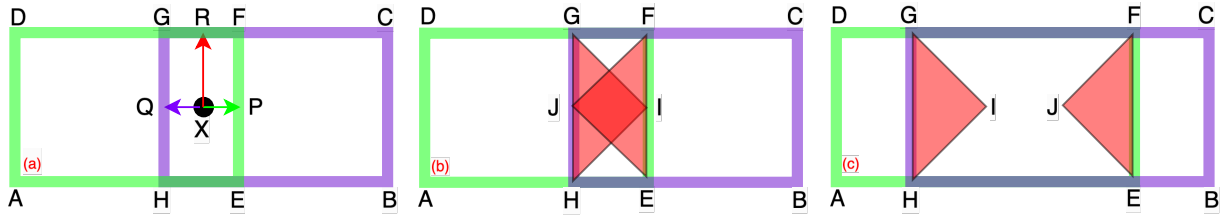


Figure 3.2: Example of combining SDF of two subregions into one. The rectangle body $\Omega = ABCD$ is divided into two subregions $\Omega_1 = AEFD$ and $\Omega_2 = BHGC$. **(a)** Point X is closer to the interior boundary than the true boundary for both subregions. Hence the local signed distance at point X of subregion Ω_1 is not true signed distance value. $\phi_1(X) = -|XP| \neq -|XR| = \phi(X)$. **(b)** Red colored regions JFE and HIG highlights the points with incorrect boundary information for $AEFD$ and $BHGC$. This partition leads to undesired region (intersection of the triangles) where the correct SDF cannot be computed. **(c)** The subregions selected farther enough so that the incorrect boundary regions (red) do not intersect, therefore SDF can be computed for all interior points. In other words, for each interior point, there exist a subregion with true boundary information (the closest boundary point is on the true boundary).

Note that the only components of the i^{th} joint is given as an input the signed distance function. In reality, subregion $\Omega_i^{\boldsymbol{\theta}}$ depends on the total joint state $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{N_J})^T \in \mathbb{R}^{\hat{N}_J}$, although the only $\boldsymbol{\theta}_i$ has a major affect on the shape of Ω_i. Hence we disregard the other joints for the sake of performance, and fix their joint states to the reference pose. In this sense, we define

$$\Omega_i^{\boldsymbol{\theta}_i} = \left\{ \boldsymbol{x} \in \mathbb{R}^3 | \exists \mathbf{X} \in \Omega_i \text{ such that } \boldsymbol{\phi}_i(\mathbf{X}, \boldsymbol{\theta}_i) = \boldsymbol{x} \right\}$$

as an approximation of $\Omega_i^{\boldsymbol{\theta}}$. Furthermore, for each joint i is equipped with another boolean function

$$\mathbb{b} : \mathbb{R}^3 \times \mathbb{R}^{D_i} \to \mathbb{B}$$

that indicates whether the closest point to $\boldsymbol{x}$ on the boundary of $\Omega_i^{\boldsymbol{\theta}_i}$ is on the true boundary of animated character $\Omega^{\boldsymbol{\theta}}$ ($\mathbb{b}_i(\boldsymbol{x}, \boldsymbol{\theta}_i) = 1$) or whether it is on the interior of $\Omega^{\boldsymbol{\theta}}$ ($\mathbb{b}_i(\boldsymbol{x}, \boldsymbol{\theta}_i) = 0$). Here $\mathbb{B} = \{0, 1\}$. In the example depicted in Figure 3.2-(a), the closest point on the boundary of the subregion $\Omega_1 = AEFD$ from the query point X lies on the edge GH, which is in the interior of the entire body $\Omega = ABCD$ (hence $\mathbb{b}_i(\boldsymbol{x}, \boldsymbol{\theta}_i) = 0$). A portion of the boundary of $\Omega_i^{\boldsymbol{\theta}_i}$ coincides with the boundary of $\Omega^{\boldsymbol{\theta}}$ and another portion is interior to $\Omega^{\boldsymbol{\theta}}$ and this boolean is used to resolve the true signed distance when a point $\boldsymbol{x}$ is in multiple joint subregions. See Figure 3.2-(b,c) for an illustration of the boolean values and choice of the subregions to avoid undesired cases. We use the notation $\boldsymbol{\phi}_i(\mathbf{X}, \theta_i) = (\phi_i(\mathbf{X}, \theta_i), \mathbb{b}_i(\mathbf{X}, \theta_i))$ to represent our augmented signed distance convention. With this formalism, the signed distance function is defined as

$$\phi(\boldsymbol{x}, \boldsymbol{\theta}) = \min_{i \in S(\boldsymbol{x}, \boldsymbol{\theta})} \phi_i(\boldsymbol{x}, \boldsymbol{\theta}_i) \tag{3.1}$$

where $S(\boldsymbol{x}, \boldsymbol{\theta}) \subset \{1, 2, \ldots, N_J\}$ is the collection of joint indices i such that $\mathbb{b}_i(\boldsymbol{x}, \boldsymbol{\theta}_i) = 1$. Note that $S(\boldsymbol{x}, \boldsymbol{\theta}) \neq \emptyset$, for any $\boldsymbol{x}$ there is at least one subregion so that closest point from $\boldsymbol{x}$ to Ω_i is on the true boundary. We enforce this by defining the subregions to be analogous to Figure 3.2(c) so that for any $\boldsymbol{x}$ the set $S(\boldsymbol{x}, \boldsymbol{\theta})$ is nonempty when we create them.

3.4 Shallow Joint Signed Distance Functions

We define each joint-wise signed distance function $\boldsymbol{\phi}_i : \mathbb{R}^3 \times \mathbb{R}^{D_i} \to \mathbb{R} \times \mathbb{B}$ in terms of a pair of shallow neural networks that can be evaluated efficiently in real-time and are accurate enough to represent the deformed shape of the animated joint region $\Omega_i^{\boldsymbol{\theta}_i}$. We named our models *shallow* to emphasize its simplicity and compactness. To enhance the ability of our neural network parameters to capture the signed distance values over a range of animated states, we find it helpful to define each signed distance in the canonical space associated with each joint. We define this space in terms of joint-wise rigid transforms

$$\boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta}) = \mathbf{R}_i(\boldsymbol{\theta})\boldsymbol{x} + \mathbf{t}_i(\boldsymbol{\theta})$$

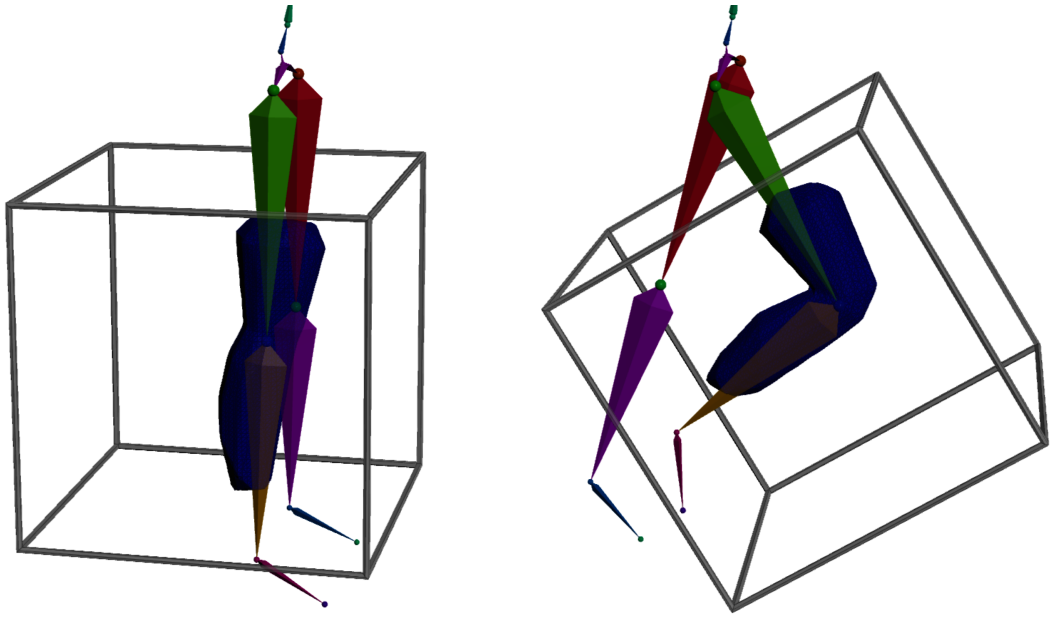


Figure 3.3: The canonical space (bounding box) of SSDF for the joint knee is determined by thigh, the parent of knee in the skeletal hierarchy. The canonical spaces moves with the parent joint as shown in the right example.

for rotations $\mathbf{R}_i(\boldsymbol{\theta})$ and translations $\mathbf{t}_i(\boldsymbol{\theta})$ as well as a Multi Layer Perceptron (MLP, [Hay94]) neural network $SSDF : \mathbb{R}^3 \times \mathbb{R}^{N_W} \times \mathbb{R}^{N_B} \times \mathbb{R}^{D_i} \to \mathbb{R}$ where N_W is the number of weights and N_B is the number of biases as

$$\phi_i(\boldsymbol{x}, \boldsymbol{\theta}) = SSDF_i(\boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta}), \mathbf{W}_i, \mathbf{B}_i, \boldsymbol{\theta}_i). \tag{3.2}$$

Here $\mathbf{W}_i$ and $\mathbf{B}_i$ represents the weights and biases of the model. The transform $\boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta})$ is chosen according to the parent transform joint in the hierarchy so that deformation near the joint is isolated from rigid motion in the rig. See Figure 3.3 for illustration.

3.4.1 Model Architecture and Joint Depended Weights

In this section we discuss the design of our neural network model $SSDF_i(\boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta}), \mathbf{W}_i, \mathbf{B}_i, \boldsymbol{\theta}_i)$ for the subregion Ω_i. For simplicity, we use notation $\mathbf{X}^i = \boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta})$ for the location input

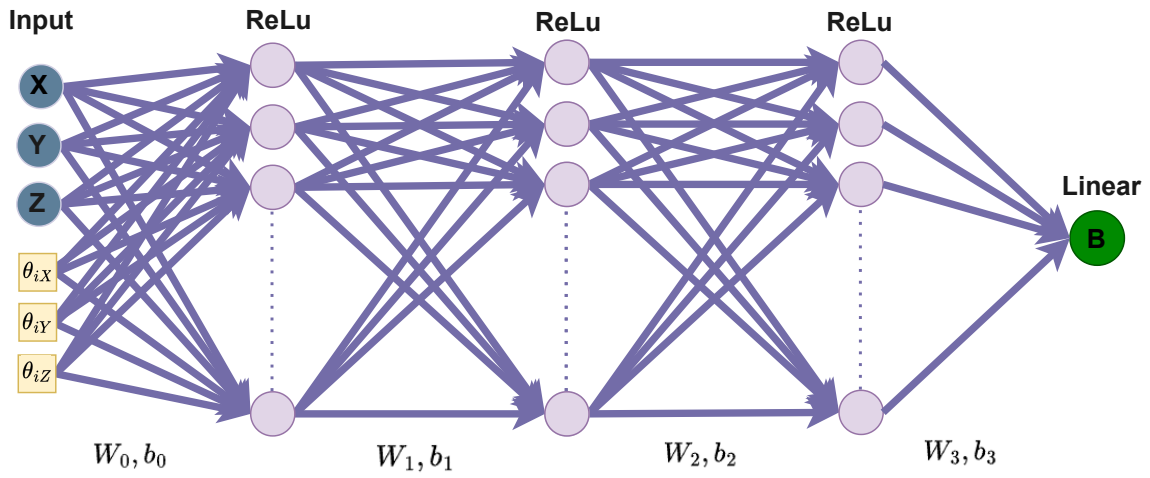
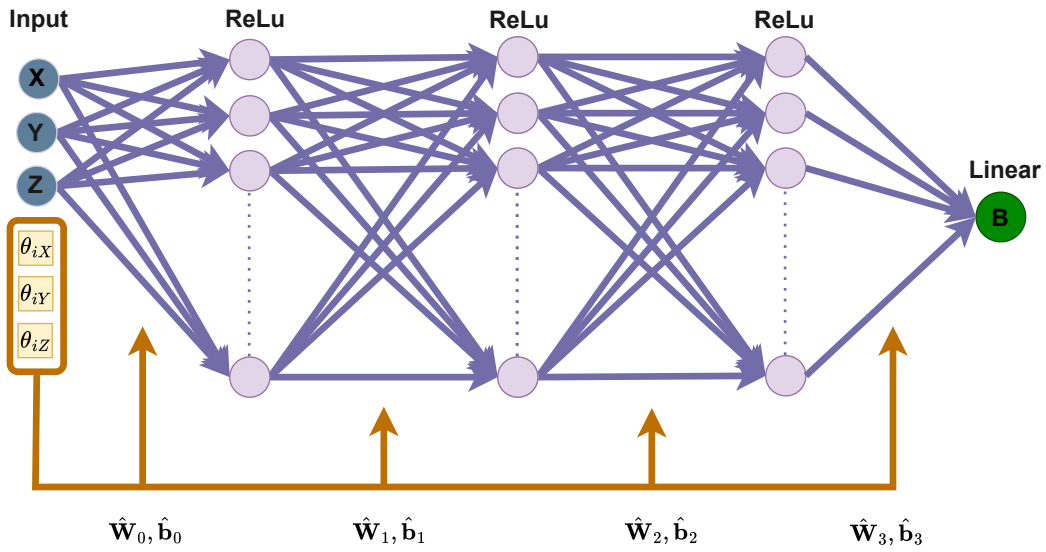


Figure 3.4: Basic MLP architecture with three hidden layers. Inputs are location of the query point wrt canonical space $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$, and joint state in terms of rotational degrees $\theta_{iX}, \theta_{iY}, \theta_{iZ}$. Note that in this example $D_i = 3$. Weights and biases $\mathbf{W}_0, \boldsymbol{b}_0, \mathbf{W}_1, \boldsymbol{b}_1, ..., \mathbf{W}_3, \boldsymbol{b}_3$ are trained and kept fixed during inference.

for the model in the canonical space. The most basic MLP architecture concatenates the location input $\mathbf{X}^i \in \mathbb{R}^3$ with the joint state input $\boldsymbol{\theta}_i \in \mathbb{R}^{D_i}$ to create an input layer and then the input later is connected with the next hidden layer (See Figure 3.4). This network architecture is indeed successful for approximating SDFs, however, it does not use the fact that the joint state inputs remains the same at inference – only the location input varies. More precisely, at fixed joint state (e.g. stationary timestep in the simulation), multiple cloth particles at different locations are queried against the body to resolve collision. Hence, we propose another type of MLP architecture which takes account the this practical information. More precisely, we find that allowing the effective weights and biases in each layer to depend



$$\hat{\mathbf{W}}_n = W_n^X \theta_{iX} + W_n^Y \theta_{iY} + W_n^Z \theta_{iZ} + W_n^C$$

$$\hat{\mathbf{b}}_n = b_n^X \theta_{iX} + b_n^Y \theta_{iY} + b_n^Z \theta_{iZ} + b_n^C$$

Figure 3.5: **Model Architecture:** The illustrated model assumes $D_i = 3$ degrees of freedom. Model trains the weight matrices $W_n^X, W_n^Y, W_n^Z, W_n^C$ and biases $b_n^X, b_n^Y, b_n^Z, b_n^C$. At inference, model first updates its weights according to the linear equation presented (Equation 3.3), then model becomes classical MLP (as in Figure 3.4 without rotational degree input). The figure illustrates the architecture for $N_J = 5$ (3 hidden layers) and each hidden layer has $N_H = 8$ neurons. The hidden layers have rectified linear unit (ReLU) activation function, the output layer has linear activation function.

linearly on the joint degrees of freedom $\boldsymbol{\theta}_i = (\theta_{i1}, \ldots, \theta_{iD_i})^T$ as

$$\hat{\mathbf{w}}_{ij}^n(\boldsymbol{\theta}_i) = \sum_{\alpha=1}^{D_i} \mathbf{w}_{ij\alpha}^n \theta_{i\alpha} + \mathbf{w}_{ij0}^n, \tag{3.3}$$

$$\hat{b}_{ij}^n(\boldsymbol{\theta}_i) = \sum_{\alpha=1}^{D_i} b_{ij\alpha}^n \theta_{i\alpha} + b_{ij0}^n \tag{3.4}$$

improved expressivity across ranges of joint rotations and inference performance. Here n indicates that the weight connects layers n and $n+1$ in the MLP (where $1 \leq n \leq N_L - 1$) and j refers to the neuron in the layer $n+1$. N_L refers to the total number of layers in the MLP with the convention that all but the first (input) and the last (output) are the so called hidden layers. N_H refers to the number of neurons in the hidden layers. In general MLP models can have different number of neuron in different layers, but in our case all hidden layers have the same number of neurons. These feed into that activation functions $\sigma : \mathbb{R} \to \mathbb{R}$ to define $SSDF(\mathbf{X}, \mathbf{W}_i, \mathbf{C}_i, \boldsymbol{\theta}_i)$ in terms of the per-layer neuron outputs y_j^{n+1} as

$$y_j^{n+1} = \sigma(\hat{\mathbf{w}}_{ij}^n(\boldsymbol{\theta}_i)^T \boldsymbol{y}^n + \hat{b}_{ij}^n(\boldsymbol{\theta}_i)), \ 0 \leq n < N_L \tag{3.5}$$

where $\boldsymbol{y}^1 = (y_1^1, y_2^1, y_3^1)^T = \mathbf{X}^i = \boldsymbol{T}_i(\boldsymbol{x}, \theta) \in \mathbb{R}^3$ and $\boldsymbol{y}^n = (y_1^n, y_2^n, ..., y_{N_H}^n)$ for $2 \leq n \leq N_L - 1$. The model outputs

$$SSDF(\mathbf{X}, \mathbf{W}_i, \mathbf{B}_i, \boldsymbol{\theta}_i) = y_1^{N_L}$$

See Figure 3.5 an illustration of this network structure. Note that with this convention, the weights relating the input and second (first hidden) layers have $\hat{\mathbf{w}}_{ij}^1(\boldsymbol{\theta}_i) \in \mathbb{R}^3$, $1 \leq j \leq N_H$. For all other choices of $1 < n < N_L$, $\hat{\mathbf{w}}_{ij}^n(\boldsymbol{\theta}_i) \in \mathbb{R}^{N_H}$. Furthermore for $1 \leq n < N_L$, the neuron index has $1 \leq j \leq N_H$ however in the weights and biases connecting the last hidden layer with the output layer ($n = N_L - 1$), there is only one output neuron and the index is simply $j = 1$.

We use $\mathbf{W}_i = \{\mathbf{w}_{ij\alpha}^n\} \in \mathbb{R}^{N_W}$ and $\mathbf{B}_i = \{b_{ij\alpha}^n\} \in \mathbb{R}^{N_B}$ Equation equation 3.2 to represent the collection of all learnable weights and biases in the MLP network and note that the effective weights and biases in Equation equation 3.3 are chosen in terms of them in

this way to suit cloth simulation and collision against dynamic avatars. In particular, the $SSDF$ is evaluated at each particle in the simulation mesh at each time step, but the dependence of the model on the joint state $\boldsymbol{\theta}$ happens only once over the time step. The formula in Equation equation 3.3 updates the weights whenever the joint state changes and once complete inference only with recomputes based on the positional inputs (with the effective weights held fixed). This allows the network to have $(D_i + 1)$ times more parameters in training, compared to what it is required at the inference. In practice we used 3 hidden layers ($N_L = 5$) and 8 channels per hidden layer ($N_H = 8$).

We use the same network structure with $N_L = 4$ hidden layers and $N_H = 8$ neurons per hidden layer for the joint-wise boolean function

$$\mathbb{b}_i(\boldsymbol{x}, \boldsymbol{\theta}) = bool(SSDF(\boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta}), \mathbf{W}_i^{bool}, \mathbf{B}_i^{bool}, \boldsymbol{\theta}_i)) \tag{3.6}$$

and also evaluate it through the joint-local transform $\mathbf{X}^i = \boldsymbol{T}_i(\boldsymbol{x}, \boldsymbol{\theta})$. Note that this network has its own learnable weights $\mathbf{W}_i^{bool}$ and biases $\mathbf{B}_i^{bool}$. Also note that $bool(\cdot)$ in Equation equation 3.6 returns false for negative values of the neural network and true for positive values.

3.5 Training and Dataset Creation

To train the joint-wise neural networks we first partition the reference pose of the body Ω (which we take to be an A-pose with the characters arms at their side) into the joint-based regions Ω_i. We assume that the character skin surface vertices have weights associated with each transform in the rig so that LBS can be applied. We then tetrahedralize the interior of the skin surface and generate skinning weights on newly created interior tetrahedron vertices by solving a Poisson equation. Dirichlet boundary conditions are applied on the surface of the tetrahedron mesh and set to the values of the LBS weights. We then associate all vertices with weight values above a threshold with the parent transform of each joint to define the seed region for the Ω_i (see top left image in Figure 3.6). We then assign vertices that did

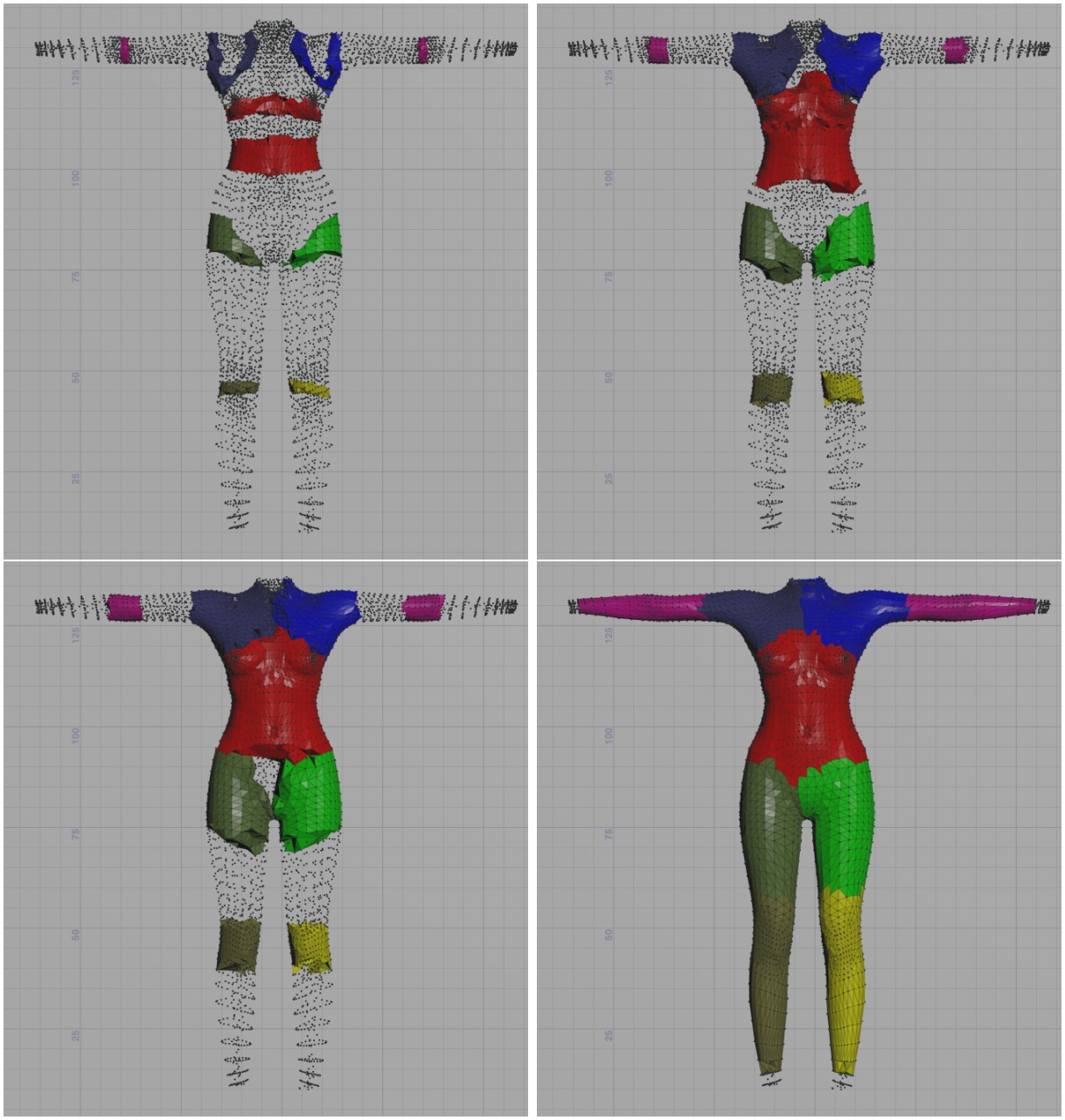


Figure 3.6: Subregion selection in the reference pose. First entire body is tetrahedralized. Each subregion is initialized as a subset of tetrahedrons that are closest to the joint of interest. The each subset is eagerly grows to their neighboors until two subregion intersect. Above figures shows inital step, step 1, step 2, and step 7.

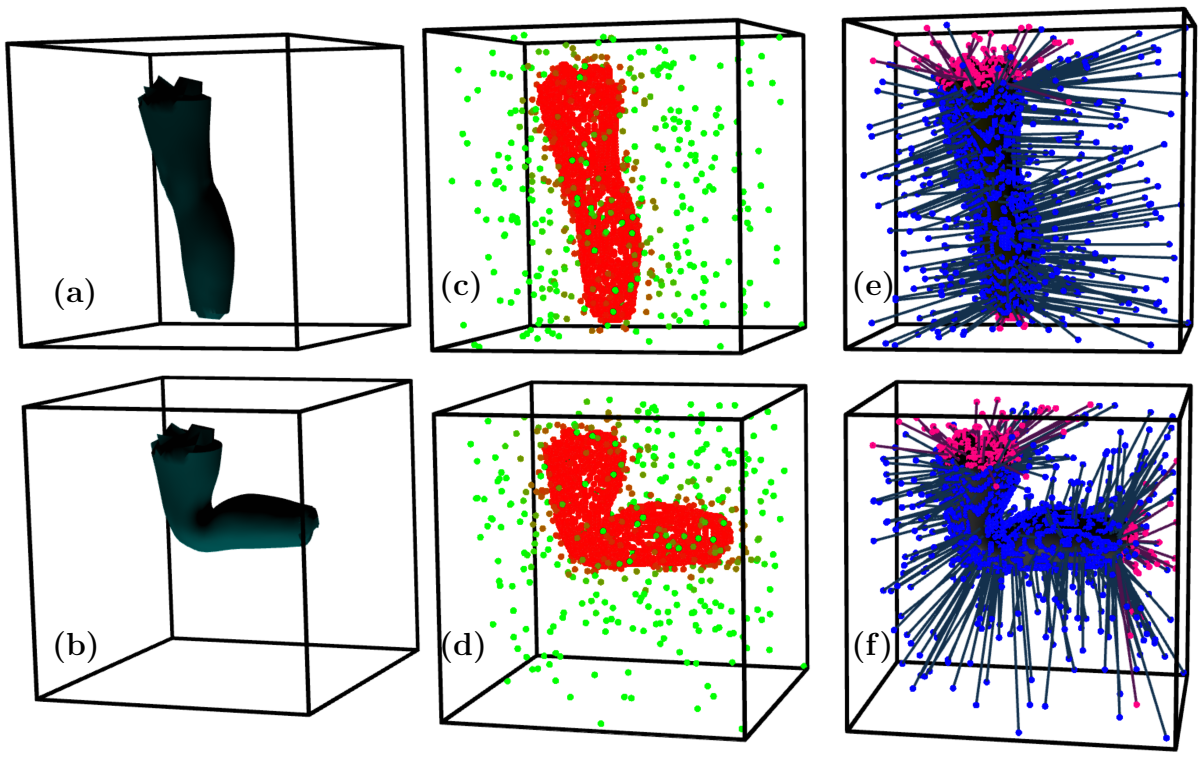


Figure 3.7: Left: Example of deformation of knee surrounded by the training grid. Knee in the upper figure is the in the reference pose (joint rotation angles are (0,0,0)), and the knee in the below image rotated 90 degrees in negative Z direction (joint rotation angles are (0,0,-90)). Middle: Training data generation illustration. Color of the point represents the distance to the boundary (red: close, green: distant). Note that almost all points near the isocounter are selected. Right: Blue points have correct signed distance as the closest point lies on the origional boundary, the purple points have incorrect signed distance values.

not have a weight above a threshold for any Ω_i greedily to regions associated with vertices connected in the tetrahedron mesh. This region growing is continued until all vertices are assigned to a region Ω_i. We then grow each region slightly to make sure that there is sufficient overlap to apply the logic of Figure 3.2(c). Figure 3.6 illustrates this process in a representative example. Figure 3.1-(a) illustrates the final partition.

To train the SSDF for each Ω_i, grid based SDFs are generated for a range of joint poses. For a particular joint pose $\boldsymbol{\theta}_i$, Ω_i is deformed to $\Omega_i^{\hat{\boldsymbol{\theta}}_i}$ (where $\hat{\boldsymbol{\theta}}_i \in \mathbb{R}^{N_J}$ has all joint variables except $\boldsymbol{\theta}_i$ set to the A-pose) where the mesh-based elastic material point method (MPM) of Jiang et al. [JSS15] is used to prevent any collisions associated with LBS. Specifically, the vertices of the tetrahedron mesh overlapping bones in the rig are tracked with Dirichlet displacement boundary conditions in a quasistatic elasticity solve using mesh-based MPM. This defines the LBS collision correction mapping in $\boldsymbol{\phi}^C$ in Section 3.2. Once this collision-free version of the LBS mapping is defined, we define SDF values over a regular grid using exact geometric distances on cut grid cells which are swept to the remaining grid nodes using the Fast Marching Method [Set96].

For the joint i with D_i degrees of freedom, we create training range as a product space of evenly spaced values. More precisely, the training range for joint i is the product space

$$\bigotimes_{\alpha=1}^{D_i} [\boldsymbol{\theta}_{i\alpha;min} : \boldsymbol{\theta}_{i\alpha;inc} : \boldsymbol{\theta}_{i\alpha;max}]$$

where

$$[\boldsymbol{\theta}_{i\alpha;min} : \boldsymbol{\theta}_{i\alpha;inc} : \boldsymbol{\theta}_{i\alpha;max}] = \{\boldsymbol{\theta}_{i\alpha;min} + k * \boldsymbol{\theta}_{i\alpha;inc} | 0 \leq k \leq \frac{\boldsymbol{\theta}_{i\alpha;max} - \boldsymbol{\theta}_{i\alpha;min}}{\boldsymbol{\theta}_{i\alpha;inc}}\}$$

All values are in euler angles. For the examles we provide, the following range of motion are used:

- Knees: 2 degrees of freedom with product space $[-20:10:20]\bigotimes[-150:10:30]$. Note that we are not using the rotation input that associates with twisting, which does not make much difference for the clothing other than very tight clothes.

- Thighs: 2 degrees of freedom with product space $[-80:10:10]\bigotimes[-90:10:90]$.

We use $100 \times 100 \times 100$ uniform grid and compute signed distance values for all 100^3 nodes for each pose and use them to generate the training data. The grid can be thought as a bounding box that covers subregion Ω_i for all deformations of $\boldsymbol{\theta}_i$ in the training range. Note that the points near the 0-isocontour are the most important points to determing the 0-levelset [PFS19]. We devise a probabilistic selection method to create the training data from the grid based SDFs (so that we do not use all 100^3 points). Point $\boldsymbol{x}$ is selected to appear in the dataset for joint i if either

$$|SDF[\boldsymbol{\theta}_i](\boldsymbol{x})| < \epsilon$$

or

$$|SDF[\boldsymbol{\theta}_i](\boldsymbol{x})| \cdot \mathbf{I} < \beta$$

Here $\mathbf{I}$ is a uniform random variable distributed over interval [0,1], ϵ is the boundary selection bound, and β is the randomized selection bound.

For our training data, we choose $\epsilon = 0.025 * L_i$ and $\beta = 0.001 * L_i$ where L_i is the side length of the bounding box associated with the i^{th} joint. Figure 3.7-(b,e) shows an example of chosen grid points. This probabilistic process selects about 30000 grid nodes among 100000. As it can be seen from the Figure 3.7, all points near boundary are selected, whereas the points further away from boundary have less chance to be chosen. Furthermore we use the same point cloud to generate the training data for the boolean network. For each selected point, we assign a float value 1 if the closest point is on the correct boundary and -1 if on

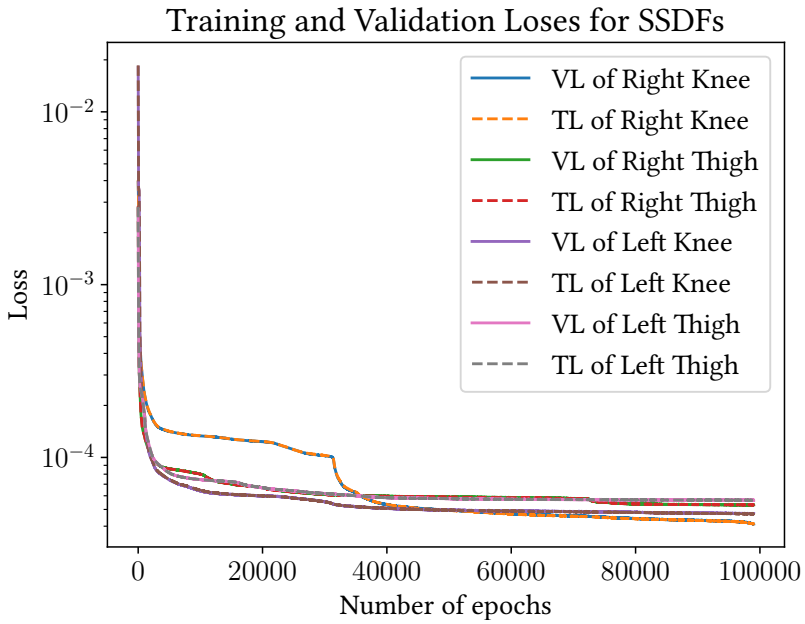


Figure 3.8: Training and validation losses for the Shallow SDF networks for various subregions. $y - axis$ is log scaled.

the interior boundary. Figure 3.7(c,f) shows boolean labeling for knee. Purple points are closer to the interior boundary, and the blue points are closer to the true boundary.
We use a modified version of clamped loss function as suggested in [PFS19].

$$L(\phi_i(\boldsymbol{x}, \boldsymbol{\theta}_i), s) = |clamp((\phi_i(\boldsymbol{x}, \boldsymbol{\theta}_i), \delta)) - clamp(s, \delta)|^2$$

where s is the ground truth signed distance value and clamping function is defined as $clamp(s, \delta) = min\{\delta, max\{-\delta, s\}\}$. Smaller clamping values allows network to focus on the boundary. In our experiments we saw that $L2$ error creates visually better results compared to $L1$. We choose $\delta = 0.2 * L_G$. We train our model with TensorFlow [AAB15] on a single NVIDIA RTX A6000 GPU with 48GB memory. We use $3GB$ of a shared memory, allowing us to train multiple models at once. Training is done with back-propagation and the ADAM [KB15] optimizer with learning rate 0.0001. We train our network for 100K epochs. Figure 3.9 shows how 0-levelset of the SSDFs evolve after 1K, 10K, 50K and 100K epochs. Training takes approximately 4 hours for 100K epochs for our network with $N_L = 5$ layers

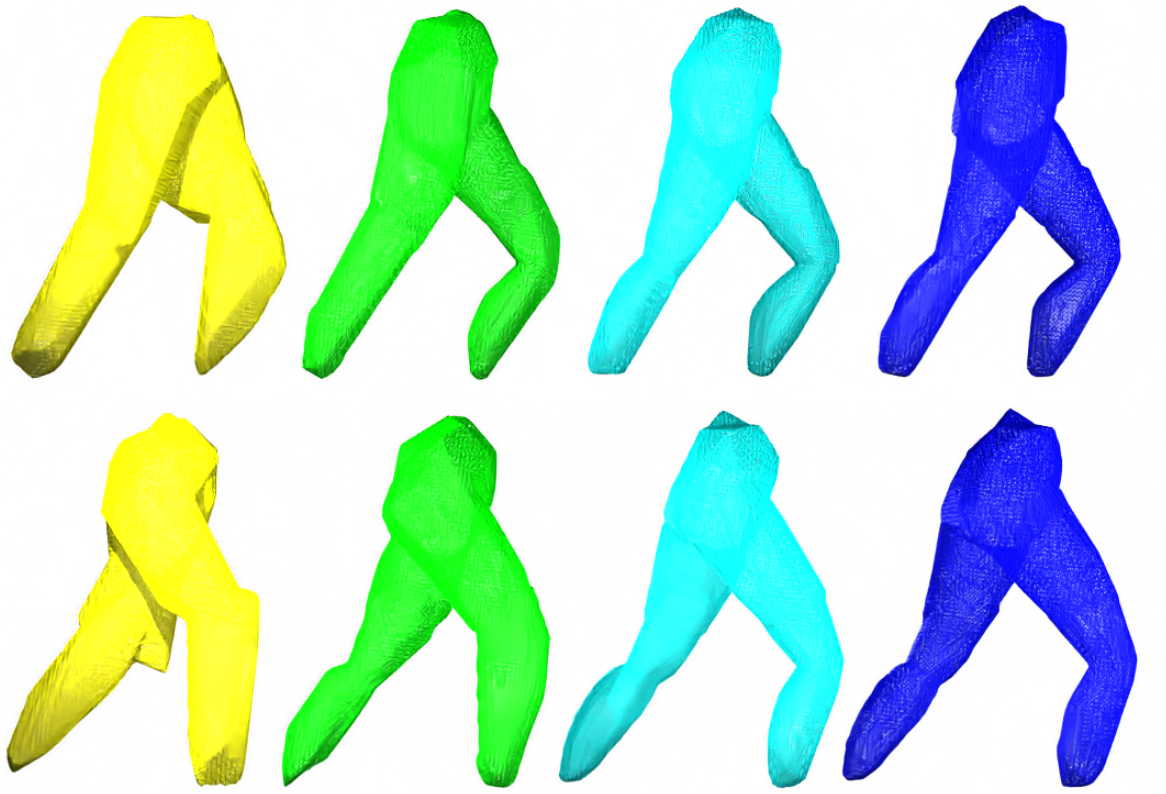


Figure 3.9: Zero-levelsets of the trained SDFs after 1K, 10K, 50K and 100K epochs.

Table 3.1: **Training and Evaluation Loss**.

Network Choices	P_T	P_I	L_T	L_V	T_{train}
$N_L = 5, N_H = 4$	183	61	2.227×10^{-4}	2.222×10^{-4}	3 hours
$N_L = 5, N_H = 8$	555	185	4.117×10^{-5}	4.119×10^{-5}	4 hours
$N_L = 5, N_H = 16$	1875	625	2.793×10^{-5}	2.812×10^{-5}	5 hours
$N_L = 5, N_H = 32$	6819	2273	5.518×10^{-6}	5.751×10^{-6}	7 hours
$N_L = 7, N_H = 32$	13155	4385	3.517×10^{-6}	3.505×10^{-6}	10 hours

Table 3.2: P_T = Number of Parameters of the network for training. P_I = Number of paramaters of the network for inference. L_T = Training loss after 100K epochs. L_V = Validation loss after 100K epochs. T_{train} = Time it takes to train for 100K epochs. The results shown are for the SSDF of the right knee decribed in Figure 3.10. The degrees of freedom for the knee is $D_i = 2$.

and $N_H = 5$ neurons at each hidden later.

3.6 Results and Examples

We demonstrate the efficacy of our approach in practical cloth simulation examples over different types of garments. These are show in Figures 3.1-(a) and 3.12. We use the SDF in a standard way to resolve collisions during simulation. Specifically, the SDF is queried to determine if a cloth particle is inside of the body and push it outwards in negative gradient direction if it is inside. We use finite forward differencing to compute the gradient normal, which requires 3 more queries (one per direction) per particle inside the body. For optimization we batch queries whenever possible, because increasing the batch size for the

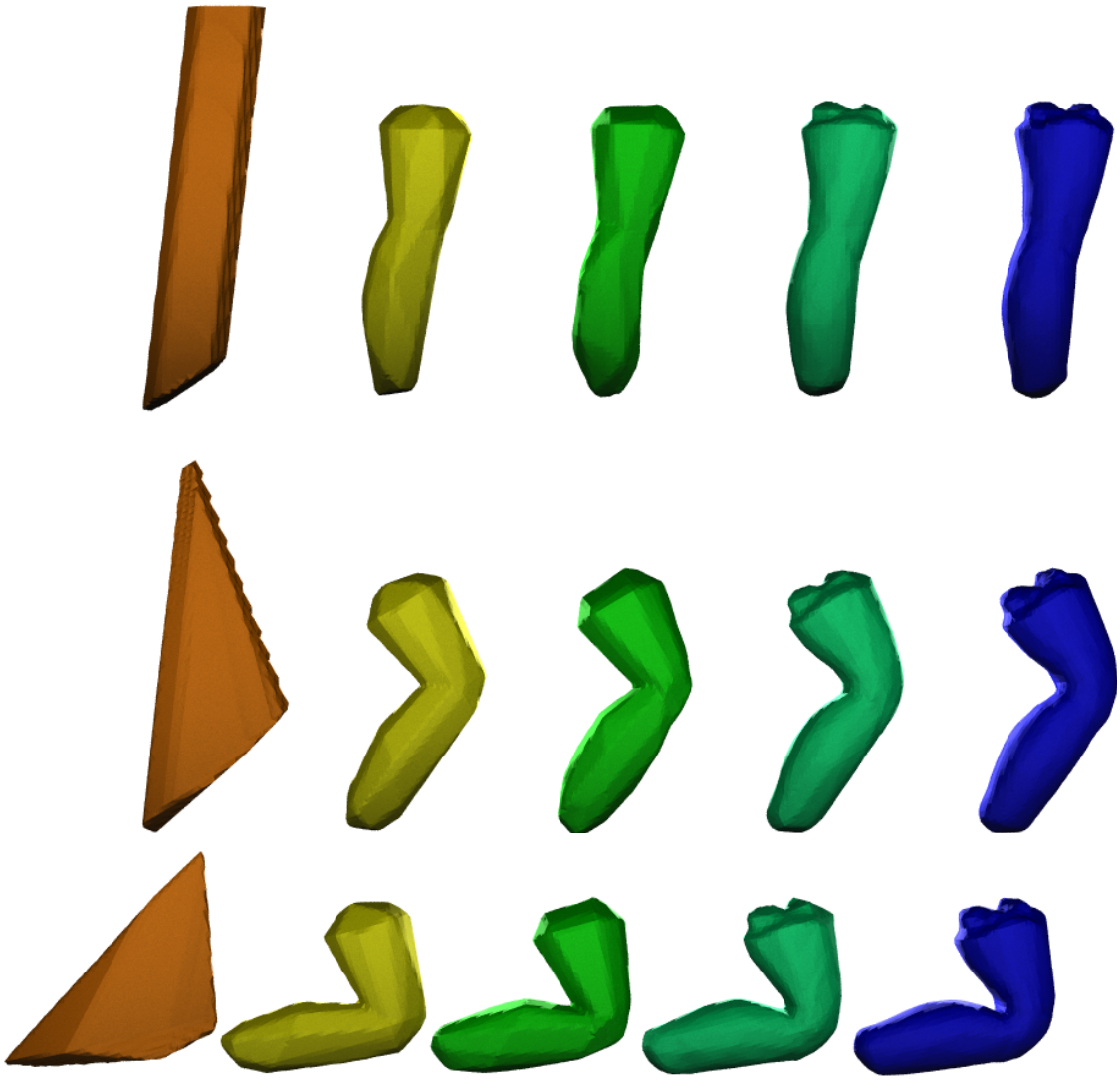


Figure 3.10: Example of SSDFs with different network structures trained for the right knee. Top to bottom 0-levelset of the deformed object for three different joints states are illustrated. From left to right the following network structures are used: $(N_L = 5, N_H = 4)$, $(N_L = 5, N_H = 8)$, $(N_L = 5, N_H = 16)$, $(N_L = 7, N_H = 2)$ and $(N_L = 7, N_H = 32)$. All networks are trained for 100K epochs. We choose the network structure in yellow for the balance between speed and performance.

Table 3.3: **Simulation Timing**

Example	N_P	T_{sim}	T_{SDF}	Per_{SDF}
Green Pants	4305	$13.7ms$	$2.55ms$	19%
Yellow Pants	4169	$14.2ms$	$3.2ms$	22.5%
Skirt	6056	$37.2ms$	$3.78ms$	10%

Table 3.4: N_P = Number of particles on the garment cloth. T_{sim} = Simulation time per frame. T_{SDF} = Total time for SDF computation. Per_{SDF} is the percentage of the time used for collision detection using our learned SDF in total simulation.

model inference reduces average time cost per particle. We achieve real-time performance with clothing meshes consisting of $4 - 6K$ particles. Collision detection/resolution takes between 10 to 25 percent of the total simulation time (See Table 3.3).

We also explicitly illustrate that our learning-based SDFs successfully predict the avatar skin boundary geometry for the any pose in a continuous motion. Figure 3.11 shows the 0-levelset of the learned SDFs in a jogging sequence. The result is remarkably comparable to the ground truth. Figure 3.9 illustrates the effect of training convergence. Improved geometric detail clearly arises with increased training epochs.

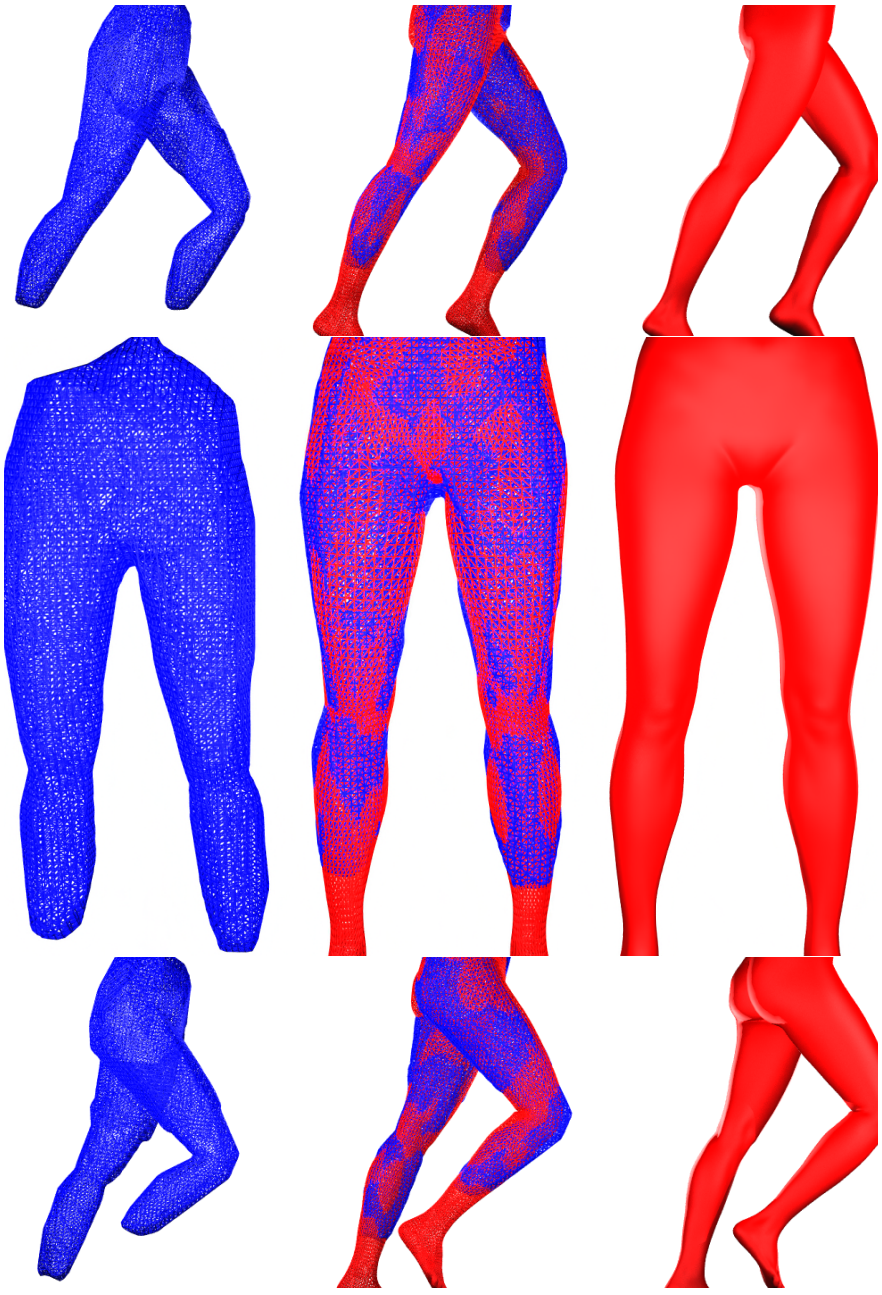


Figure 3.11: Zero-levelset derived from learned SDF with 4-SSDFs in 3 different joint states. From left to right: learned SDF, learned SDF and Ground Truth combined, and Ground Truth are presented.

Figure 3.12: Cloth simulation using our network-based SDFs. The character runs in real time with different garments on. Simulation is performed using Unreal Engine 5.

3.7 Discussion and Future Work

Our method allows for real-time SDF queries in practical simulation of clothing collision against deformable avatar skin surfaces. The use of shallow networks is crucial for ensuring efficient evaluation times, as their localized structure minimizes input queries, thereby significantly reducing the network size. While we show that this can be done accurately in the context of pants and dress collisions with the lower torso, there is still room for improvement. Increasing the number of weights and biases greatly enhances the expressivity of the network, as shown in Figure 3.10. Due to performance constraints, we focus on using very shallow networks. However, increasing the model size allows for capturing additional details. In future work, we would like to achieve higher accuracy for a given performance constraint. This can be done with the investigation of novel network architectures, or using methods such as knowledge distillation where a smaller (student) network is trained to learn a larger (teacher) network's output (or the last hidden layer). Lastly, we assume that joint degrees of freedom $\boldsymbol{\theta}_i$ will not deform the skin too dramatically as in regions Ω_j far from the joint. This will not always be the case and our method could be adjusted to better resolve these cases. More precisely, with the current design, neighboring subregions may not align perfectly with each other, leading to discontinuities near their intersections. This can be seen in the knee-tight intersection in upper rightmost image in Figure 3.9. We also would like to investigate models that accepts the neighboring joint states as input.

REFERENCES

[AAB15] 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.

[ADG16] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and Nando de Freitas. "Learning to learn by gradient descent by gradient descent." *Advances in Neural Information Processing Systems*, p. 29, 2016.

[ADP20] 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.

[BFA02] R. Bridson, R. Fedkiw, and J. Anderson. "Robust Treatment of Collisions, Contact and Friction for Cloth Animation." *ACM Trans Graph*, **21**(3):594–603, 2002.

[BHH19] 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.

[BHJ20] C. Beck, M. Hutzenthaler, A. Jentzen, and B. Kuckuck. "An overview on deep learning-based approximation methods for partial differential equations.", 2020.

[BME20] H. Bertiche, M. Madadi, and S. Escalera. "Cloth3d: clothed 3d humans." In *European Conf Comp Vision (ECCV)*, pp. 344–359. Springer, 2020.

[BMT21] H. Bertiche, M. Madadi, E. Tylson, and S. Escalera. "DeePSD: Automatic deep skinning and pose space deformation for 3D garment animation." In *Proc EEE/CVF International Conference on Computer Vision*, pp. 5471–5480, 2021.

[Bra77] A. Brandt. "Multi-level adaptive solutions to boundary-value problems." *Math Comp*, **31**(138):333–390, 1977.

[Bri08] R. Bridson. *Fluid simulation for computer graphics*. Taylor & Francis, 2008.

[BW98] D. Baraff and A. Witkin. "Large Steps in Cloth Simulation." In *Proc ACM SIGGRAPH*, SIGGRAPH '98, pp. 43–54, 1998.

[BZY21] M. Baorui, H. Zhizhong, L. Yu-Shen, and Z. Matthias. "Neural-Pull: Learning Signed Distance Functions from Point Clouds by Learning to Pull Space onto Surfaces." In *International Conference on Machine Learning (ICML)*, 2021.

[CCC22] T. Chen, X. Chen, W. Chen, H. Heaton, J. Liu, Z. Wang, and W. Yin. "Learning to optimize: a primer and a benchmark." *Journal of Machine Learning Research 23*, pp. 8562–8620, 2022.

[Cho67] A. Chorin. "A numerical method for solving incompressible viscous flow problems." *J Comp Phys*, **2**(1):12–26, 1967.

[CKM21] 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.

[CLI20] R. Chabra, J. Lenssen, E. Ilg, T. Schmidt, J. Straub, S. Lovegrove, and R. Newcombe. "Deep local shapes: Learning local sdf priors for detailed 3d reconstruction." In *Computer Vision–ECCV 2020*, pp. 608–625. Springer, 2020.

[CMW22] 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*, pp. 1–12, 2022.

[flu22] fluidnetsc22. "fluidnetsc22/fluidnet_sc22: v0.0.1.", April 2022. doi: 10.5281/zenodo.6424901, URL: `https://doi.org/10.5281/zenodo.6424901`.

[FSJ01] R. Fedkiw, J. Stam, and H. Jensen. "Visual simulation of smoke." In *SIGGRAPH*, pp. 15–22. ACM, 2001.

[GA18] 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)*, pp. 49–56. IEEE, 2018.

[GCS19] E. Gundogdu, V. Constantin, A. Seifoddini, M. Dang, M. Salzmann, and P. Fua. "Garnet: A two-stream network for fast and accurate 3d cloth draping." In *Proc IEEE/CVF Int Conf Comp Vision*, pp. 8739–8748, 2019.

[GCV19] K. Genova, F. Cole, D. Vlasic, A. Sarna, W. T. Freeman, and T. Funkhouser. "Learning shape templates with structured implicit functions." In *Proc IEEE/CVF ICCV*, pp. 7154–7164, 2019.

[GGB19] D. Greenfeld, M. Galun, R. Basri, I. Yavneh, and R. Kimmel. "Learning to optimize multigrid PDE solvers." In *Int Conf Mach Learn*, pp. 2415–2423. PMLR, 2019.

[GHF19] 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.

[GHM20] 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.

[GL12] G. Golub and C. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[GSK16] A. Grebhahn, N. Siegmund, H. Köstler, and S. Apel. "Performance prediction of multigrid-solver configurations." In *Software for Exascale Computing-SPPEXA 2013-2015*, pp. 69–88. Springer, 2016.

[Hay94] S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[HKU20] P. Holl, V. Koltun, K. Um, and N. Thuerey. "phiflow: A differentiable pde solving framework for deep learning via physical simulations." In *NeurIPS Workshop*, 2020.

[HS52] 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.

[HW65] 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.

[HZE19] J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon. "Learning Neural PDE Solvers with Convergence Guarantees.", 2019.

[HZR16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[IFH20] 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)*, pp. 61–68, 2020.

[JGG20] 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.

[JSS15] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin. "The Affine Particle-In-Cell Method." *ACM Trans Graph*, **34**(4):51:1–51:10, 2015.

[KB15] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization." *CoRR*, **abs/1412.6980**, 2015.

[Ker78] 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.

[KFB23] M. Koptev, N. Figueroa, and A. Billard. "Neural Joint Space Implicit Signed Distance Functions for Reactive Robot Manipulator Control." *IEEE Robotics and Automation Letters*, **8**(2):480–487, 2023.

[KKL21] 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.

[Lan50] C. Lanczos. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators." 1950.

[LDF22] I. Liao, Rumen R. Dangovski, Jakob N. Foerster, and M. Soljačić. "Learning to Optimize Quasi-Newton Methods." *arXiv preprint*, 2022.

[LFO06] F. Losasso, R. Fedkiw, and S. Osher. "Spatially adaptive techniques for level set methods and incompressible flow." *Computers & Fluids*, **35**(10):995–1010, 2006.

[LGF04] F. Losasso, F. Gibou, and R. Fedkiw. "Simulating water and smoke with an octree data structure." *ACM Trans. Graph.*, **23**(3):457–462, 2004.

[LGM20] I. Luz, M. Galun, H. Maron, R. Basri, and I. Yavneh. "Learning algebraic multigrid using graph neural networks." In *Int Conf Mach Learn*, pp. 6489–6499. PMLR, 2020.

[LKB21] K. Luna, K. Klymko, and J. P. Blaschke. "Accelerating GMRES with Deep Learning in Real-Time.", 2021.

[LLK19] 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.

[LM16] K. Li and J. Malik. "Learning to Optimize." *arXiv preprint*, 2016.

[LWY23] Y. Liu, L. Wang, J. Yang, W. Chen, X. Meng, Bo. Yang, and L. Gao. "NeUDF: Leaning Neural Unsigned Distance Fields With Volume Rendering." In *Proc IEEE/CVF CVPR*, pp. 237–247, June 2023.

[MLT89] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. "Joint-Dependent Local Deformations for Hand Animation and Object Grasping." In *ProcGraph Int '88*, pp. 26–33. Canadian Information Processing Society, 1989.

[MST10] A. McAdams, E. Sifakis, and J. Teran. "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.

[Nus81] H. Nussbaumer. "The fast Fourier transform." In *Fast Fourier Transform and Convolution Algorithms*, pp. 80–111. Springer, 1981.

[OCD22] J. Ortiz, A. Clegg, J. Dong, E. Sucar, D. Novotny, M. Zollhoefer, and M. Mukadam. "iSDF: Real-Time Neural Signed Distance Fields for Robot Perception." In *Robotics: Science and Systems*, 2022.

[OF03] S. Osher and R. Fedkiw. *Level set methods and dynamic implicit surfaces*. Applied mathematical science. Springer, New York, N.Y., 2003.

[Pai71] C. C. Paige. *The computation of eigenvalues and eigenvectors of very large sparse matrices*. PhD thesis, University of London, 1971.

[PFS19] J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. "DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation." In *Proc IEEE/CVF CVPR*, June 2019.

[PGG23] J. Panuelos, R. Goldade, E. Grinspun, D.I.W. Levin, and C. Batty. "PolyStokes: A Polynomial Model Reduction Method for Viscous Fluid Simulation." *ACM Trans Graph (TOG)*, **42**(4), 2023.

[PS75] 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.

[PZ11] 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.

[RCC23] C. Romero, D. Casas, M. Chiaramonte, and M. Otaduy. "Learning Contact Deformations with General Collider Descriptors." In *SIGGRAPH Asia 2023 Conf Papers*, SA '23. ACM, 2023.

[RGT18] 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.

[RPK19] 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.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003.

[SCH19] J. Shen, X. Chen, H. Heaton, T. Chen, J. Liu, W. Yin, and Z. Wang. "Learning a minimax optimizer: A pilot study." *International Conference on Learning Representations*, 2019.

[SCT20] V. Sitzmann, E. Chan, R. Tucker, N. Snavely, and G. Wetzstein. "MetaSDF: Meta-Learning Signed Distance Functions." In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pp. 10136–10147. Curran Associates, Inc., 2020.

[Set96] J. Sethian. "A fast marching level set method for monotonically advancing fronts." *Proc Nat Acad Sci*, **93**(4):1591–1595, 1996.

[SGP20] 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*, pp. 8459–8468. PMLR, 2020.

[SLF08] A. Selle, M. Lentine, and R. Fedkiw. "A Mass Spring Model for Hair Simulation." *ACM Trans Graph*, **27**(3):64:1–64:11, 2008.

[SMF20] 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.

[SS86] 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.

[SSH19] J. Sappl, L. Seiler, M. Harders, and W. Rauch. "Deep Learning of Preconditioners for Conjugate Gradient Solvers in Urban Water Related Problems.", 2019.

[Sta20] R. Stanaityte. *ILU and Machine Learning Based Preconditioning For The Discretized Incompressible Navier-Stokes Equations*. PhD thesis, University of Houston, 2020.

[Sti52] E. Stiefel. "Über einige methoden der relaxationsrechnung." *Zeitschrift für angewandte Mathematik und Physik ZAMP*, **3**(1):1–33, 1952.

[STO21] I. Santesteban, N. Thuerey, M. Otaduy, and D. Casas. "Self-supervised collision handling via generative 3d garment models for virtual try-on." In *Proc IEEE/CVF Conf Comp Vision and Pattern Recognition*, pp. 11763–11773, 2021.

[SYE00] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h. "A Deflated Version of the Conjugate Gradient Algorithm." *SIAM Journal on Scientific Computing*, **21**:1909–1926, 2000.

[TB97] L. Trefethen and D. Bau. *Numerical Linear Algebra*, volume 50. SIAM, 1997.

[TKC21] 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.

[TSS17] 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*, pp. 3424–3433. PMLR, 06–11 Aug 2017.

[TWT16] M. Tang, H. Wang, L. Tang, R. Tong, and D. Manocha. "CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-based Cloth Simulation." *Comp Graph Forum*, **35**(2):511–521, 2016.

[TZW22] Q. Tan, Y. Zhou, T. Wang, D. Ceylan, X. Sun, and D. Manocha. "A repulsive force unit for garment collision handling in neural networks." In *European Conf Comp Vision (ECCV)*, pp. 451–467. Springer, 2022.

[UBF20] 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, pp. 6111–6122. Curran Associates, Inc., 2020.

[UPT20] B. Ummenhofer, L. Prantl, N. Thuerey, and V. Koltun. "Lagrangian Fluid Simulation with Continuous Convolutions." In *International Conference on Learning Representations*, 2020.

[VGO20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa,

Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods*, **17**:261–272, 2020.

[Wan14] H. Wang. "Defending Continuous Collision Detection Against Errors." *ACM Trans Graph*, **33**(4):122:1–122:10, 2014.

[WBT19] S. Wiewel, M. Becher, and N. Thuerey. "Latent space physics: Towards learning the temporal evolution of fluid flow." In *Computer graphics forum*, volume 38, pp. 71–82. Wiley Online Library, 2019.

[WLF18] Z. Wang, L.Wu, M. Fraftarcangeli, M. Tang, and H. Wang. "Parallel Multigrid for nonlinear cloth simulation." *Computer Graphics Forum*, **37**(7):131–141, 2018.

[WWY20] L. Wu, B. Wu, Y. Yang, and H. Wang. "A Safe and Fast Repulsion Method for GPU-Based Cloth Self Collisions." *ACM Trans. Graph.*, **40**(1), dec 2020.

[WY16] H. Wang and Y. Yang. "Descent methods for elastic body simulation on the GPU." *ACM Trans Graph*, **35**(6):1–10, Nov 2016.

[YYX16] C. Yang, X. Yang, and X. Xiao. "Data-driven projection method in fluid simulation." *Comp Anim Virt Worlds*, **27**(3-4):415–424, 2016.