UGrid: An Efficient-And-Rigorous Neural Multigrid Solver for Linear PDEs

Xi Han [1] **Fei Hou** [2 3] **Hong Qin** [1]

Abstract

Numerical solvers of Partial Differential Equations (PDEs) are of fundamental significance to science and engineering. To date, the historical reliance on legacy techniques has circumscribed possible integration of big data knowledge and exhibits sub-optimal efficiency for certain PDE formulations, while data-driven neural methods typically lack mathematical guarantee of convergence and correctness. This paper articulates a mathematically rigorous neural solver for linear PDEs. The proposed ***UGrid*** solver, built upon the principled integration of ***U***-Net and Multi***Grid***, manifests a mathematically rigorous proof of both convergence and correctness, and showcases high numerical accuracy, as well as strong generalization power to various input geometry/values and multiple PDE formulations. In addition, we devise a new residual loss metric, which enables self-supervised training and affords more stability and a larger solution space over the legacy losses.

1. Introduction

Background and Major Challenges. PDEs are quintessential to various computational problems in science, engineering, and relevant applications in simulation, modeling, and scientific computing. Numerical solutions play an irreplaceable role in common practice because in rare cases do PDEs have analytic solutions, and many general-purpose numerical methods have been made available. Iterative solvers (Saad, 2003) are one of the most-frequently-used methods to obtain a numerical solution of a PDE. Combining iterative solvers with the multigrid method (Briggs & McCormick, 2000) significantly enhances the performance for large-scale problems. Yet, the historical reliance on legacy generic numerical solvers has circumscribed possible integration of big data knowledge and exhibits sub-optimal efficiency for certain PDE formulations. In contrast, recent deep neural methods have the potential to learn such knowledge from big data and endow numerical solvers with compact structures and high efficiency, and have achieved impressive results (Marwah et al., 2021). However, many currently available neural methods treat deep networks as black boxes. Other neural methods are typically trained in a fully supervised manner on loss functions that directly compare the prediction and the ground truth solution, confining the solution space and resulting in numerical oscillations in the relative residual error even after convergence. These methods generally have challenges unconquered including, a lack of sound mathematical backbone, no guarantee of correctness or convergence, and low accuracy, thus unable to handle complex, unseen scenarios.

Motivation and Method Overview. Inspired by (Hsieh et al., 2019)'s prior work on integrating the structure of multigrid V-cycles (Briggs & McCormick, 2000) and U-Net (Ronneberger et al., 2015) with convergence guarantee, and to achieve high efficiency and strong robustness, we aim to fully realize neural methods' modeling and computational potential by implanting the legacy numerical methods' mathematical backbone into neural methods in this paper. In order to make our new framework fully explainable, we propose the UGrid framework (illustrated in Fig. 1) based on the structure of multigrid V-cycles for learning the functionality of multigrid solvers. We also improve the convolutional operators originating from (Hsieh et al., 2019) to incorporate arbitrary boundary conditions and multiple differential stencils without modifying the *overall structure* of the key iteration process, and transform the iterative update rules and the multigrid V-cycles into a concrete Convolutional Neural Network (CNN) structure.

Key Contributions. The salient contributions of this paper comprise: (1) *Theoretical insight*. We introduce a novel explainable neural PDE solver founded on a solid mathematical background, affording high efficiency, high accuracy, and strong generalization power to linear PDEs; (2) *New loss metric*. We propose a residual error metric as the loss

[1]Department of Computer Science, Stony Brook University (SUNY), Stony Brook, NY 11794, USA. [2]Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, 100190, China. [3]University of Chinese Academy of Sciences, Beijing, 100049, China. Correspondence to: Xi Han <xihan1@cs.stonybrook.edu>, Fei Hou <houfei@ios.ac.cn>, Hong Qin <qin@cs.stonybrook.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

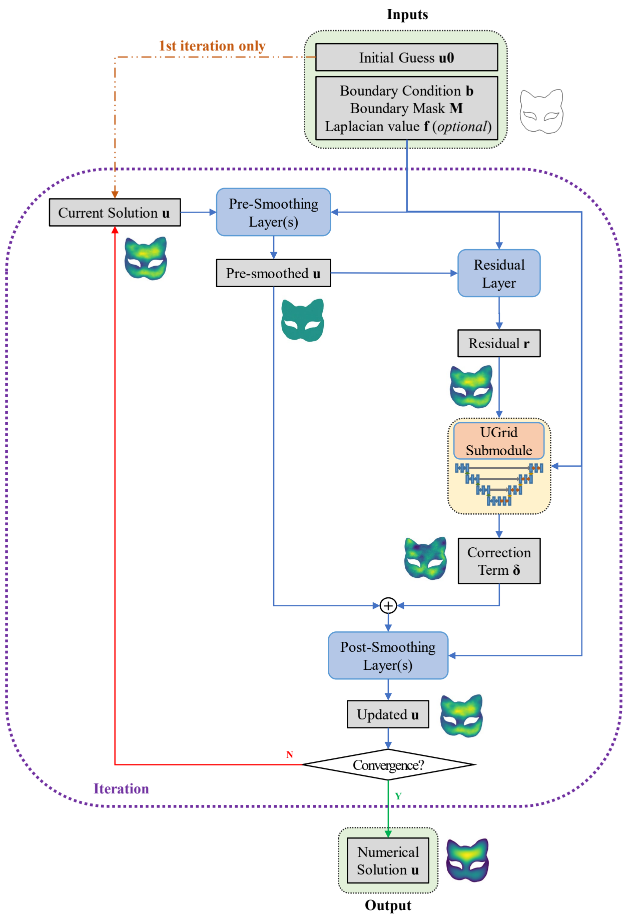


Figure 1. Overview of our novel method. Given PDE parameters and its current numerical estimation, the smoothing operations are applied multiple times first. Then, the current residual is fed into our UGrid submodule (together with the boundary mask). Next, the solution is corrected by the correction term and post-smoothed. Collectively, it comprises one iteration of the neural solver. The UGrid submodule (detailed in Fig. 2) aims to mimic the multigrid V-cycle, and its parameters are learnable, so as to endow our framework with the ability to learn from data.

function, which optimizes the residual of the prediction. Our newly-proposed error metric enables self-supervised learning and facilitates the unrestricted exploration of the solution space. Meanwhile, it eliminates the numerical oscillation on the relative residual error upon convergence, which has been frequently observed on the legacy mean-relative-error-based loss metrics; and (3) *Extensive experiments*. We demonstrate our method's capability to numerically solve PDEs by learning multigrid operators of various linear PDEs subject to arbitrary boundary conditions of complex geometries and topology, whose patterns are unseen during the training phase. Extensive experiments and comprehensive evaluations have verified all of our claimed advantages, and confirmed that our proposed method outperforms the SOTA.

2. Related Work

Black-box-like Neural PDE Solvers. Much research effort has been devoted to numerically solve PDEs with neural networks and deep learning techniques. However, most of the previous work treats neural networks as black boxes and thus come with no mathematical proof of convergence and correctness. As early as the 1990s, (Wang & Mendel, 1990a;b; 1991) applied simple neural networks to solve linear equations. Later, more effective neural-network-based methods like (Polycarpou & Ioannou, 1991; Cichocki & Unbehauen, 1992; Lagaris et al., 1998) were proposed to solve the Poisson equations. On the other hand, (Wu et al., 1994; Xia et al., 1999; Takala et al., 2003; Liao et al., 2010; Li et al., 2017) used Recurrent Neural Networks (RNNs) in solving systems of linear matrix equations. Most recently, the potential of CNNs and Generative Adversarial Networks (GANs) on solving PDEs was further explored by (Tompson et al., 2017; Tang et al., 2017; Farimani et al., 2017; Sharma et al., 2018; Özbay et al., 2021). Utilities used for neural PDE solvers also include backward stochastic differential equations (Han et al., 2018) and PN junctions (Zhang et al., 2019). On the contrary, the proposed UGrid mimics the multigrid solver, and all its contents are explainable and have corresponding counterparts in an MG hierarchy.

Physics-informed Neural PDE Solvers. Physics-informed Neural Network (PINN)-based solvers effectively optimize the residual of the solution. Physical properties, including pressure, velocity (Yang et al., 2016) and non-locality (Pang et al., 2020) are also used to articulate neural solvers. Mathematical proofs on the minimax optimal bounds (Lu et al., 2022) and structural improvements (Lu et al., 2021a;b) are also made on the PINN architecture itself, endowing physics-informed neural PDE solvers with higher efficiency and interpretability. Hinted by these, we propose the residual loss metric, which enables self-supervised training, enlarges the solution space and enhances numerical stability.

Neural PDE Solvers with Mathematical Backbones. (Zhou et al., 2009) proposed a NN-based linear system and its solving algorithm with a convergence guarantee. (Hsieh et al., 2019) modified the Jacobi iterative solver by predicting an additional correction term with a multigrid-inspired linear operator, and proposed a linear neural solver with guarantee on correctness upon convergence. (Greenfeld et al., 2019) proposed to learn a mapping from a family of PDEs to the optimal prolongation operator used in the multigrid method, which is then extended to Algebraic Multigrids (AMGs) on non-square meshes via Graph Neural Networks (GNNs) by (Luz et al., 2020). On the other hand, (Li et al., 2021) proposed a Fourier neural operator that learns mappings between function spaces by parameterizing the integral kernel directly in Fourier space. In theory, (Marwah et al., 2021) proved that when a PDE's coefficients are representable by small NNs, the number of parameters needed will increase in a polynomial fashion with the input dimension.

3. Mathematical Preliminary

For mathematical completeness, we provide readers with a brief introduction to the concepts that are frequently seen in this paper.

Discretization of 2D Linear PDEs. A linear PDE with Dirichlet boundary condition could be discretized with finite differencing techniques (Saad, 2003), and could be expressed in the following form:

$$\begin{cases} \mathcal{D}u(x,y) = f(x,y), & (x,y) \in \mathcal{I} \\ u(x,y) = b(x,y), & (x,y) \in \mathcal{B} \end{cases}, \tag{1}$$

where $\mathcal{D}$ is a 2D discrete linear differential operator, $\mathcal{S}$ is the set of all points on the discrete grid, $\mathcal{B}$ is the set of boundary points in the PDE, $\mathcal{I} = \mathcal{S} \setminus \mathcal{B}$ is the set of interior points in the PDE, $\partial\mathcal{S} \subseteq \mathcal{B}$ is the set of *trivial boundary points* of the grid.

Using $\mathcal{D}$'s corresponding finite difference stencil, Eq. 1 can be formulated into a sparse linear system of size $n^2 \times n^2$:

$$\begin{cases} (\mathbf{I}-\mathbf{M})\mathbf{A}\mathbf{u} = (\mathbf{I}-\mathbf{M})\mathbf{f} \\ \mathbf{M}\mathbf{u} = \mathbf{M}\mathbf{b} \end{cases}, \tag{2}$$

where $\mathbf{A} \in \mathbb{R}^{n^2 \times n^2}$ is the 2D discrete differential operator, $\mathbf{u} \in \mathbb{R}^{n^2}$ encodes the function values of the interior points and the non-trivial boundary points; $\mathbf{f} \in \mathbb{R}^{n^2}$ encodes the corresponding partial derivatives of the interior points; $\mathbf{b} \in \mathbb{R}^{n^2}$ encodes the non-trivial boundary values; $\mathbf{I}$ denotes the $n^2 \times n^2$ identity matrix; $\mathbf{M} \in \{0,1\}^{n^2 \times n^2}$ is a diagonal binary boundary mask defined as

$$\mathbf{M}_{k,k} = \begin{cases} 1, & (i,j) \in \mathcal{B} \setminus \partial\mathcal{S} \\ 0, & (i,j) \in \mathcal{I} \end{cases}, \; k = in+j, \; 0 \le i, \; j < n. \tag{3}$$

On the contrary of Eq. 1, both equations in Eq. 2 hold for *all* grid points.

Error Metric And Ground-truth Solution. When using numerical solvers, researchers typically substitute the boundary mask $\mathbf{M}$ into the discrete differential matrix $\mathbf{A}$ and the partial derivative vector $\mathbf{f}$, and re-formulate Eq. 2 into the following generic sparse linear system:

$$\widetilde{\mathbf{A}}\,\mathbf{u} = \widetilde{\mathbf{f}}. \tag{4}$$

The *residual* of a numerical solution $\mathbf{u}$ is defined as

$$\mathbf{r}(\mathbf{u}) = \widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{u}. \tag{5}$$

In the ideal case, the *absolute residual error* of an exact solution $\mathbf{u}^*$ should be $r_{\mathbf{u}^*} = \|\mathbf{r}(\mathbf{u}^*)\| = 0$. However, in practice, a numerical solution $\mathbf{u}$ could only be an approximation of the exact solution $\mathbf{u}^*$. The precision of $\mathbf{u}$ is evaluated by its *relative residual error*, which is defined as

$$\varepsilon_{\mathbf{u}} = \left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{u}\right\| \Big/ \left\|\widetilde{\mathbf{f}}\right\|. \tag{6}$$

Typically, the ultimate goal of a numerical PDE solver is to seek the optimization of the relative residual error. If we have $\varepsilon_{\mathbf{u}} \le \varepsilon_{\max}$ for some small $\varepsilon_{\max}$, we would consider $\mathbf{u}$ to be a *ground-truth solution*.

Linear Iterator. A linear iterator (also called an iterative solver or a smoother) for generic linear systems like Eq. 4 could be expressed as

$$\mathbf{u}_{k+1} = \left(\mathbf{I} - \widetilde{\mathbf{P}}^{-1}\widetilde{\mathbf{A}}\right)\mathbf{u}_k + \widetilde{\mathbf{P}}^{-1}\widetilde{\mathbf{f}}, \tag{7}$$

where $\widetilde{\mathbf{P}}$ is an easily invertible approximation to the system matrix $\widetilde{\mathbf{A}}$.

4. Novel Approach

The proposed UGrid neural solver is built upon the principled integration of the U-Net architecture and the multigrid method. We observe that linear differential operators, as well as their approximate inverse in legacy iterative solvers, are analogous to convolution operators. E.g., the discrete Laplacian operator is a 3×3 convolution kernel. Furthermore, the multigrid V-cycle is also analogous to the U-Net architecture, with grid transfer operators mapped to up/downsampling layers. Moreover, the fine-tuning process of multigrid's critical components on specific PDE formulations could be completed by learning from big data. These technical observations lead to our neural implementation and optimization of the multigrid routine.

In spite of high efficiency, generalization power remains a major challenge for neural methods. Many SOTA neural solvers, e.g., (Hsieh et al., 2019), fail to generalize to new scenarios unobserved during the training phase. Such new scenarios include: (1) New problem sizes; and (2) New, complex boundary conditions and right-hand sides, which includes geometries, topology, and values (noisy inputs). Moreover, some of these methods are tested on Poisson equations only; neither mathematical reasoning nor empirical results show that they could trivially generalize to other PDEs (with or without retraining). UGrid resolves all problems above.

UGrid is comprised of the following components: (1) The *fixed* neural smoother, which consists of our proposed convolutional operators (Sec. 4.1); (2) The *learnable* neural multigrid, which consists of our UGrid submodule (Sec. 4.2); (3) A residual loss metric (Sec. 4.3) which enables the self-supervised training process.

4.1. Convolutional Operators

This subsection is organized as follows: Sec. 4.1.1 introduces the masked operators, which mimic the smoothers in a legacy multigrid routine; and Sec. 4.1.2 introduces the masked residual operators for residual calculation.

4.1.1. Masked Convolutional Iterator

A trivial linear iterator in the form of Eq. 7 does **not** fit in a neural routine. This is because in practice, the system matrix $\mathbf{A}$ for its corresponding differential operator encodes the *boundary geometry* and turns into matrix $\widetilde{\mathbf{A}}$ in Eq. 4. $\widetilde{\mathbf{A}}$'s elements are input-dependent, and is thus impossible to be expressed as a *fix-valued* convolution kernel.

We make use of the masked version of PDEs (Eq. 2) and their *masked convolutional iterators*, which are natural extensions of Eq. 7:

$$\mathbf{u}_{k+1} = (\mathbf{I} - \mathbf{M}) \left(\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A} \right) \mathbf{u}_k + \mathbf{P}^{-1}\mathbf{f} \right) + \mathbf{M}\mathbf{b}, \quad (8)$$

where $\mathbf{P}$ is an easily-invertible approximation on the discrete differential operator $\mathbf{A}$. The correctness of Eq. 8 is guaranteed by the following Theorem (proved as Theorem A.3 in the appendix):

Theorem 4.1. *For a PDE in the form of Eq. 2, the masked iterator Eq. 8 converges to its ground-truth solution when its prototype Jacobi iterator converges and* $\mathbf{P}$ *is full-rank diagonal.*

Matrix $\mathbf{A}$ has different formulations for different linear PDEs. Without loss of generality, we consider the following problem formulations. Other possible PDE formulations are essentially combinations of differential primitives available in the three problems below, and our convolutional operators could be trivially extended to higher orders of differentiation.

2D **Poisson Problem**. Under Dirichlet boundary condition, a Poisson problem could be expressed as follows:

$$\begin{cases} \nabla^2 u(x,y) = f(x,y), & (x,y) \in \mathcal{I} \\ u(x,y) = b(x,y), & (x,y) \in \mathcal{B} \end{cases}, \quad (9)$$

where u is the unknown scalar field, f is the Laplacian field, and b is the Dirichlet boundary condition.

Matrix $\mathbf{A}$ could be assembled by the five-point finite difference stencil for 2D Laplace operators (Saad, 2003), and we could simply let $\mathbf{P} = -4\mathbf{I}$, where $\mathbf{I}$ denotes the identity matrix. The update rule specified in Eq. 8 thus becomes

$$\begin{aligned} \mathbf{u}_{k+1}(i,j) = \frac{1}{4}(\mathbf{I} - \mathbf{M})(&\mathbf{u}_k(i-1,j) + \mathbf{u}_k(i+1,j) \\ &+ \mathbf{u}_k(i,j-1) + \mathbf{u}_k(i,j+1) - \mathbf{f}) + \mathbf{M}\mathbf{b}. \end{aligned} \quad (10)$$

To transform the masked iterator into a convolution layer, we reorganize the column vectors $\mathbf{u}$, $\mathbf{b}$, $\mathbf{M}$ and $\mathbf{f}$ into $n \times n$ matrices with their semantic meanings untouched. Then, the neural *smoother* could be expressed as

$$\begin{aligned} \mathbf{u}_{k+1} &= \text{smooth}(\mathbf{u}_k) \\ &= (\mathbf{1} - \mathbf{M})(\mathbf{u}_k * \mathbf{J} - 0.25\mathbf{f}) + \mathbf{M}\mathbf{b}\,, \\ \mathbf{J} &= \begin{pmatrix} 0 & 0.25 & 0 \\ 0.25 & 0 & 0.25 \\ 0 & 0.25 & 0 \end{pmatrix}, \end{aligned} \quad (11)$$

where $\mathbf{1}$ is an $n \times n$ matrix whose elements are all equal to 1, and $*$ denotes 2D convolution.

2D **Helmholtz Problem**. Under Dirichlet boundary condition, an inhomogeneous Helmholtz equation with spatially-varying wavenumber may be expressed as follows:

$$\begin{cases} \nabla^2 u(x,y) + k^2(x,y)u(x,y) = f(x,y), & (x,y) \in \mathcal{I} \\ u(x,y) = b(x,y), & (x,y) \in \mathcal{B} \end{cases}, \quad (12)$$

where u is the unknown scalar field, k^2 is the spatially-varying wavenumber, f is the (non-zero) right hand side, and b is the Dirichlet boundary condition.

For our proposed UGrid solver, we could naturally extend Eq. 11 into the following form to incorporate Eq. 12:

$$\mathbf{u}_{k+1} = \frac{1}{\mathbf{4} - \mathbf{k^2}}(\mathbf{1} - \mathbf{M})(\mathbf{u}_k * 4\mathbf{J} - \mathbf{f}) + \mathbf{M}\mathbf{b}\,, \quad (13)$$

where all notations retain their meanings as in Eq. 11.

2D **Steady-state Convection-diffusion-reaction Problem**. Under Dirichlet boundary condition, an inhomogeneous steady-state convection-diffusion-reaction equation may be expressed as follows:

$$\begin{cases} \mathbf{v}(x,y) \cdot \boldsymbol{\nabla} u(x,y) - \alpha\nabla^2 u(x,y) + \beta u(x,y) = f(x,y), \\ \hfill (x,y) \in \mathcal{I} \\ \hfill u(x,y) = b(x,y), \\ \hfill (x,y) \in \mathcal{B} \end{cases}, \quad (14)$$

where u is the unknown scalar field, $\mathbf{v} = (v_x, v_y)^\top$ is the vector velocity field, α, β are constants, f is the (non-zero) right-hand side, and b is the Dirichlet boundary condition.

For our proposed UGrid solver, we could naturally extend Eq. 11 into the following form to incorporate Eq. 14:

$$\begin{aligned} \mathbf{u}_{k+1} = \frac{1}{4\alpha + \beta}&(\mathbf{1} - \mathbf{M}) \\ &(\alpha \mathbf{u}_k * 4\mathbf{J} + \mathbf{v_x}(\mathbf{u}_k * \mathbf{J_x}) + \mathbf{v_y}(\mathbf{u}_k * \mathbf{J_y}) + \mathbf{f}) \\ &+ \mathbf{M}\mathbf{b}\,, \end{aligned} \quad (15)$$

$$\mathbf{J_x} = \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0 & -0.5 \\ 0 & 0 & 0 \end{pmatrix} , \quad \mathbf{J_y} = \begin{pmatrix} 0 & -0.5 & 0 \\ 0 & 0 & 0 \\ 0 & 0.5 & 0 \end{pmatrix} ,$$

where $\mathbf{J_x}$ and $\mathbf{J_y}$ are two convolution kernels introduced for the gradient operator in Eq. 14, and all notations retain their meanings as in Eq. 11.

4.1.2. Convolutional Residual Operator

Except for the smoother, the multigrid method also requires the calculation of the residual in each iteration step. In practice, the *residual operator* (Eq. 5) can also be seamlessly implemented as a convolution layer. Because our masked iterator (Eq. 8) guarantees that $\mathbf{u}$ satisfies $\mathbf{Mu} = \mathbf{Mb}$ at any iteration step, the residual operator for Poisson problems could be simplified into

$$\mathbf{r}(\mathbf{u}) = (\mathbf{1} - \mathbf{M})\,(\mathbf{f} - \mathbf{u} * \mathbf{L}) \ , \quad \mathbf{L} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} . \tag{16}$$

For Helmholtz problems, Eq. 16 could be naturally extended into

$$\mathbf{r}(\mathbf{u}) = (\mathbf{1} - \mathbf{M})\left(\mathbf{f} - \mathbf{u} * \mathbf{L} - \mathbf{k^2 u}\right) , \tag{17}$$

where all notations retain their meanings as in Eq. 16.

For steady-state convection-diffusion-reaction problems, we could extend Eq. 16 into

$$\mathbf{r}(\mathbf{u}) = (\mathbf{1} - \mathbf{M}) \\ \left(\mathbf{f} + \mathbf{v_x}(\mathbf{u} * \mathbf{J_x}) + \mathbf{v_y}(\mathbf{u} * \mathbf{J_y}) + \alpha \mathbf{u} * \mathbf{L} - \beta \mathbf{u}\right) , \tag{18}$$

where all notations retain their meanings as in Eq. 16.

4.2. Neural Network Design

UGrid Iteration. We design the iteration step of our neural iterator as a sequence of operations as follows (which is illustrated in Fig. 1):

$$\begin{aligned}
\mathbf{u} &= \mathrm{smooth}^{\nu_1}(\mathbf{u}) && \text{(Pre-smooth for } \nu_1 \text{ times);} \\
\mathbf{r} &= \mathbf{r}(\mathbf{u}) && \text{(Calculate the current residual);} \\
\delta &= \mathrm{UGrid}(\mathbf{r}, \mathbf{1} - \mathbf{M}) && \text{(UGrid submodule recursion);} \\
\mathbf{u} &= \mathbf{u} + \delta && \text{(Apply the correction term);} \\
\mathbf{u} &= \mathrm{smooth}^{\nu_2}(\mathbf{u}) && \text{(Post-smooth for } \nu_2 \text{ times).}
\end{aligned} \tag{19}$$

The entire iteration process is specifically designed to emulate the multigrid iteration (Saad, 2003): We use the pre-smoothing and post-smoothing layers (as specified in Eq. 11) to eliminate the high-frequency modes in the residual $\mathbf{r}$, and invoke the *UGrid submodule* to eliminate the low-frequency modes.

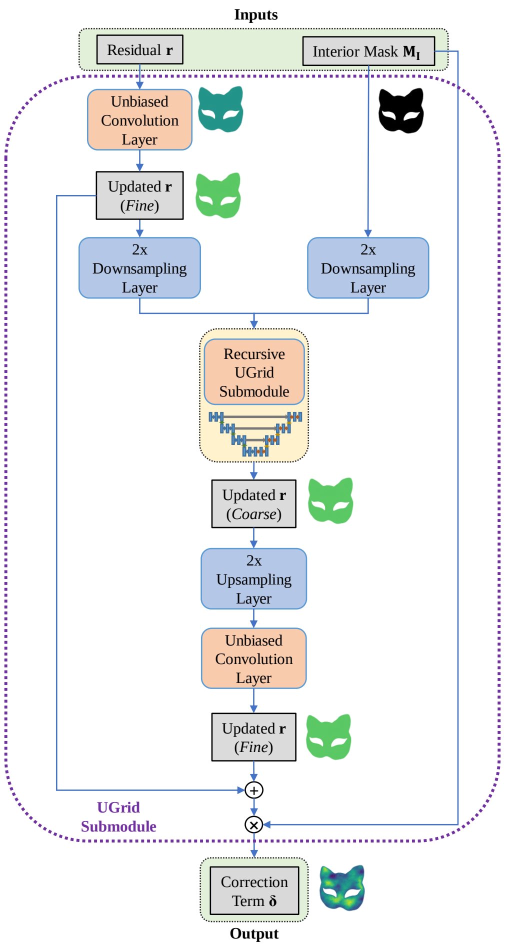


Figure 2. An overview of our recursive UGrid submodule. The residual is smoothed by *unbiased* convolution layers, downsampled to be *recursively updated* by a 2x-coarser UGrid submodule (note the orange recursive invocation of UGrid submodule in the middle), then upsampled back to the fine grid, smoothed, and added with the initial residual by skip-connection. Boundary values are enforced by interior mask via element-wise multiplication. The convolution layers (shown in orange) are learnable; other layers (shown in blue) are the fixed mathematical backbone.

UGrid Submodule. Our UGrid submodule is also implemented as a fully-convolutional network, whose structure is highlighted in Fig. 2. The overall structure of UGrid is built upon the principled combination of U-Net (Ronneberger et al., 2015) and multigrid V-cycle, and could be considered a "V-cycle" with skip connections. Just like the multigrid method, our UGrid submodule is also invoked recursively, where each level of recursion would coarsen the mesh grid by 2x.

To approximate the linearity of the multigrid iteration (note that we are learning to invert a system matrix), we implement the smoothing layers in the legacy multigrid V-cycle (not to be confused with the pre-smoother and the postsmoother in Eq. 19, which are *outside* of the V-cycle hierarchy) as learnable 2D convolution layers *without* any bias. For the same reason, and inspired by (Hsieh et al., 2019), we also drop many commonly seen neural layers which would introduce non-linearity, such as normalization layers and activation layers.

4.3. Loss Function Design

Legacy Loss Metric. We refer the equivalents of the mean relative error between the prediction and a ground-truth solution as the *legacy loss metric*. The following Theorem (proved in Sec. A.5) shows that though intuitive, the legacy loss is unstable:

Theorem 4.2. *When a neural network converges on a legacy loss metric such that its prediction* $\mathbf{x}$ *satisfies* $\mathcal{L}_{\text{legacy}}(\mathbf{x}, \mathbf{y}) = \text{mean}\left(|\mathbf{x} - \mathbf{y}|/|\mathbf{y}|\right) \le l_{\max}$*, where* $\mathbf{y}$ *denotes the ground truth value, the upper and lower bounds of* $\mathbf{x}$*'s relative residual error are still dependent on the input.*

Theorem 4.2 explains our experimental observations that: (1) Optimizing the legacy loss metric does **not** increase the precision in terms of the relative residual error; and (2) The legacy loss metric restricts the solution space: A valid numerical solution with low relative residual error may have a large relative difference from another valid solution (the one selected as the ground truth value). As a result, many valid solutions are *unnecessarily rejected* by the legacy loss metric.

Proposed Residual Loss Metric. To overcome the shortcomings of the legacy loss metric, we propose to optimize the neural solver directly to the residual in a self-supervised manner:

$$\mathcal{L}_{\mathbf{r}_{\text{abs}}}(\mathbf{x}) = \mathbb{E}_{\mathbf{x}}[\|(\mathbf{1} - \mathbf{M})(\mathbf{f} - \mathbf{A}\,\mathbf{x})\|_2] \,. \tag{20}$$

We have conducted ablation studies on our relative loss metric and the legacy loss (Sec. 5 and Sec. A.6). The results showcase that the proposed residual loss performs better than the legacy loss in terms of both efficiency and accuracy. Heuristically, the proposed residual loss metric is closer to the fundamental definition of the precision of a PDE's solution, and is more robust and stable, because upon convergence, it guarantees an input-independent final accuracy.

Moreover, the self-supervised training process endows our method with the following merits: (1) Easier data generation (compared to other neural routines which are trained on the legacy loss), and thus achieve better numerical performance; (2) For a specific PDE formulation, we could easily get a decent neural multigrid solver optimized for that specific formulation (which outperforms existing general-purpose legacy routines), simply by training UGrid on the data generated. On the contrary, fine-tuning legacy solvers is a non-trivial task requiring a solid mathematical background as well as non-trivial programming effort.

5. Experiments and Evaluations

Experiments Overview. For each of the three PDEs mentioned in Sec. 4.1: (i) Poisson problem, (ii) inhomogeneous Helmholtz problem with varying wave numbers, and (iii) inhomogeneous steady-state diffusion-convection-reaction problem, we train one UGrid model specialized for its formulation. We apply our model and the baselines to the task of 2.5D freeform surface modeling. These surfaces are modeled by the three types of PDEs as 2D height fields, with non-trivial geometry/topology. Each surface is discretized into: (1) Small-scale problem: A linear system of size $66,049 \times 66,049$; (2) Large-scale problem: A linear system of size $1,050,625 \times 1,050,625$; (3) XL-scale problem: A linear system of size $4,198,401 \times 4,198,401$; and (4) XXL-scale problem: A linear system of size $16,785,409 \times 16,785,409$. UGrid is trained **on the large scale only**. Other problem sizes are designed to evaluate UGrid's generalization power and scalability.

In addition, we have conducted an ablation study on the residual loss metric (v.s. legacy loss) as well as the UGrid architecture itself (v.s. vanilla U-Net).

Data Generation and Implementation Details. Our new neural PDE solver is trained in a self-supervised manner on the residual loss. Before training, we synthesized a dataset with 16000 $(\mathbf{M}, \mathbf{b}, \mathbf{f})$ pairs. For Helmholtz and diffusion problems, we further random-sample their unique coefficient fields (more details available in Sec. A.8 and Sec. A.9.) To examine the generalization power of UGrid, the geometries of boundary conditions in our training data are limited to "Donuts-like" shapes as shown in Fig. 3 (h). Moreover, all training data are restricted to zero $\mathbf{f}$-fields *only*, i.e., $\mathbf{f} \equiv \mathbf{0}$. Our UGrid model has 6 recursive Multigrid submodules. We train our model and perform all experiments on a personal computer with 64GB RAM, AMD Ryzen 9 3950x 16-core processor, and NVIDIA GeForce RTX 2080 Ti GPU. We train our model for 300 epochs with the Adam optimizer. The learning rate is initially set to 0.001, and decays by 0.1 for every 50 epochs. We initialize all learnable parameters with PyTorch's default initialization policy. Our code is available as open-source at `https://github.com/AXIHIXA/UGrid`.

Experimental Results. We compare our model with two state-of-the-art legacy solvers, AMGCL (Demidov, 2019), and NVIDIA AmgX (NVIDIA Developer, 2022), as well as one SOTA neural solver proposed by (Hsieh et al., 2019).

Our testcases as shown in Fig. 3. These are all with complex geometry and topology, and **none** of which are present in the training data, except the geometry of Fig. 3 (h). Testcase(s)

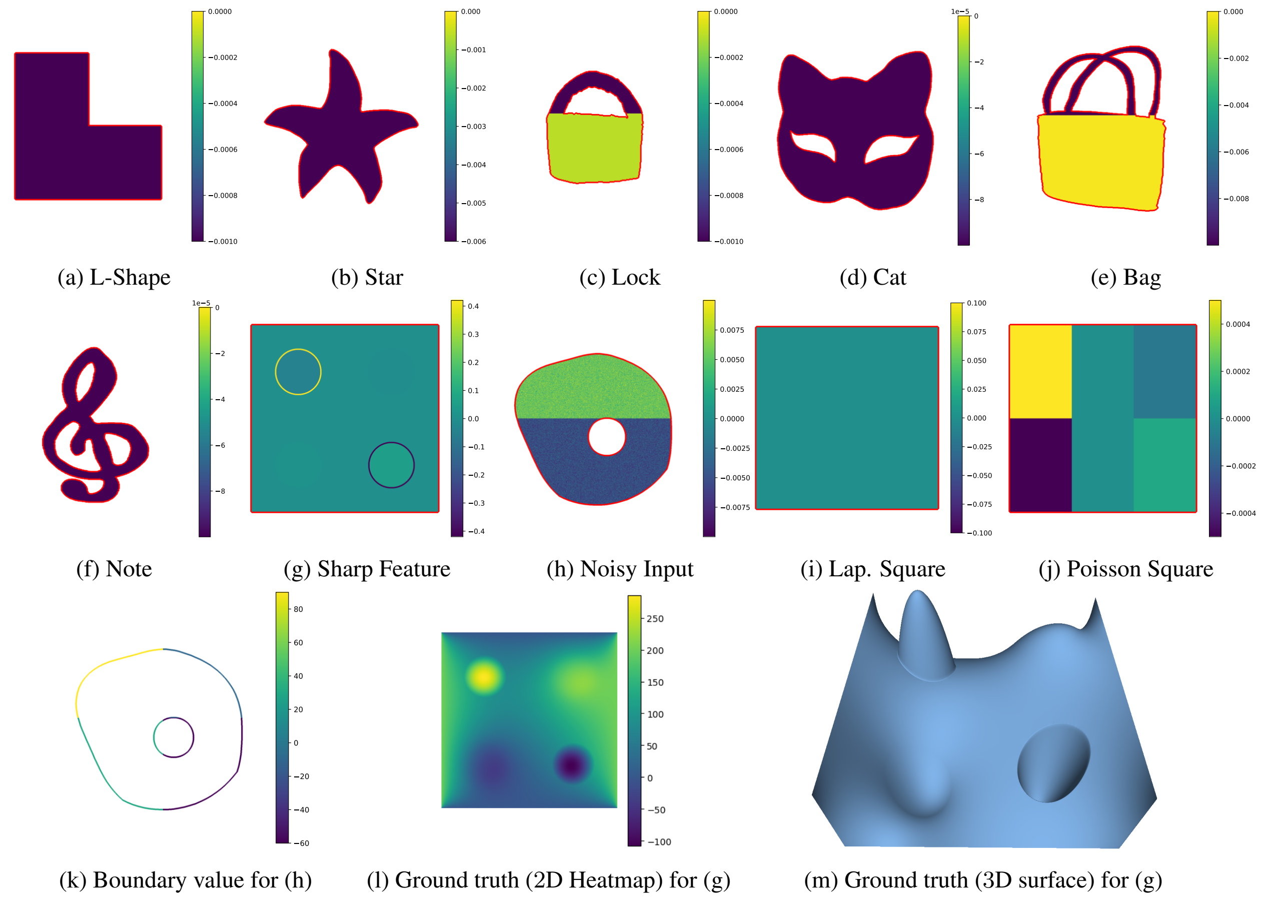


Figure 3. (a-j) illustrates the **f field distributions** of our testcases. The boundaries are shown in bold red lines for a better view. (k-m) are self-explanatory. The complete illustration of **boundary values** is available in Sec. A.2.

Table 1. Qualitative results for large-scale Poisson problems. "Time" denotes the total time (ms) to reach relative residual errors $\leq 10^{-4}$, and "Assembly" and "Iteration" denotes time (ms) for the two phases for AMG solvers; "Error" denotes the final relative residual errors, divided by 10^{-5}. Convergence maps are also available in the appendix.

Testcase	**UGrid**	**AMGCL**	**AmgX**	**Hsieh et al.**
Poisson (L)	Time / Error	Time / Assembly / Iteration / Error	Time / Assembly / Iteration / Error	Time / Error
Bag	**18.66** / 2.66	202.95 / 192.89 / 10.06 / 4.80	92.14 / 23.55 / 68.59 / 4.26	58.09 / 420
Cat	**10.09** / 2.70	270.33 / 261.75 / 8.58 / 6.63	113.92 / 25.58 / 88.35 / 6.60	49.79 / 14.6
Lock	**10.55** / 9.88	140.13 / 133.06 / 7.07 / 4.05	67.60 / 17.90 / 49.69 / 4.87	49.92 / 55.78
N. Input	**10.16** / 2.64	260.92 / 254.36 / 6.55 / 3.52	116.11 / 24.38 / 91.72 / 9.22	51.07 / 2654
Note	**10.31** / 4.06	128.96 / 121.81 / 7.16 / 3.00	64.44 / 16.55 / 47.89 / 4.80	20.26 / 8.67
S. Feat.	**20.01** / 3.80	424.41 / 413.66 / 10.75 / 4.13	174.76 / 38.85 / 135.92 / 3.87	51.22 / 24883
L-shape	**15.26** / 8.43	224.23 / 216.57 / 7.66 / 6.75	110.52 / 24.52 / 86.00 / 4.92	50.53 / 96.1
Lap. Squ.	**15.10** / 3.88	420.35 / 407.60 / 12.75 / 3.63	164.71 / 40.14 / 124.57 / 0.01	31.43 / 9.03
P. Squ.	**15.07** / 9.37	420.93 / 407.60 / 12.74 / 5.24	161.40 / 39.57 / 121.83 / 0.01	50.57 / 974
Star	**15.18** / 7.50	154.29 / 146.03 / 8.26 / 6.65	71.35 / 19.01 / 52.35 / 5.17	50.45 / 384

(a-f) examines the strong generation power and robustness of UGrid for irregular boundaries with complex geometries and topology **unobserved** during the training phase; (g) is designed to showcase UGrid's power to handle both sharp and smooth features in one scene (note that there are two sharp-feature circles on the top-left and bottom-right corners, as well as two smooth-feature circles on the opposite corners); (h) examines UGrid's robustness against noisy

Table 2. Qualitative results for large-scale Helmholtz problems.

Testcase	**UGrid**	**AMGCL**	**AmgX**
Helmholtz (L)	Time / Error	Time / Assembly / Iteration / Error	Time / Assembly / Iteration / Error
Bag	**20.03** / 8.08	199.34 / 191.97 / 7.37 / 4.80	93.67 / 24.01 / 69.67 / 6.12
Cat	**16.85** / 0.51	274.12 / 266.14 / 7.98 / 6.63	116.02 / 26.48 / 89.64 / 4.54
Lock	**12.83** / 6.82	140.21 / 133.48 / 6.73 / 4.05	68.32 / 18.02 / 50.30 / 4.99
N. Input	**11.98** / 6.79	247.65 / 241.00 / 6.65 / 3.51	116.98 / 24.71 / 92.28 / 9.16
Note	**12.18** / 6.48	128.56 / 121.93 / 6.63 / 3.00	65.91 / 16.89 / 49.02 / 8.22
S. Feat.	**57.68** / 9.86	412.93 / 403.02 / 9.92 / 4.13	Diverge
L-shape	**23.14** / 4.68	211.86 / 204.38 / 7.49 / 6.75	111.80 / 24.54 / 87.26 / 5.37
Lap. Squ.	**44.91** / 9.98	401.24 / 388.40 / 12.84 / 3.63	165.90 / 41.32 / 124.58 / 6.91
P. Squ.	**43.55** / 8.31	402.97 / 389.34 / 13.62 / 5.24	167.35 / 40.93 / 126.42 / 9.14
Star	**15.18** / 7.50	150.84 / 142.86 / 7.98 / 6.65	72.69 / 19.53 / 53.16 / 5.78

Table 3. Qualitative results for large-scale steady-state diffusion-convection-reaction problems.

Testcase	**UGrid**	**AMGCL**	**AmgX**
Diffusion (L)	Time / Error	Time / Assembly / Iteration / Error	Time / Assembly / Iteration / Error
Bag	**41.89** / 4.77	197.55 / 190.33 / 7.22 / 5.29	104.53 / 23.83 / 80.71 / 5.12
Cat	**100.68** / 9.06	273.65 / 265.80 / 7.85 / 9.21	124.11 / 26.43 / 97.69 / 5.63
Lock	**58.79** / 4.78	141.53 / 134.62/ 6.91 / 4.78	141.53 / 134.62/ 6.91 / 4.78
N. Input	**84.29** / 8.75	260.69 / 253.78 / 6.90 / 4.40	127.31 / 25.77 / 101.54 / 0.08
Note	**25.24** / 7.42	127.19 / 121.74 / 5.45 / 6.73	61.26 / 16.73 / 44.53 / 4.78
S. Feat.	**33.80** / 7.90	412.85 / 402.80 / 10.05 / 4.26	182.50 / 40.38 / 142.12 / 0.46
L-shape	**30.09** / 4.70	223.98 / 216.29 / 7.69 / 6.29	111.92 / 24.76 / 87.16 / 4.57
Lap. Squ.	**60.31** / 6.62	422.53 / 409.46 / 13.07 / 4.56	196.37 / 43.44 / 152.93 / 7.83
P. Squ.	**48.60** / 7.89	418.10 / 405.11 / 12.99 / 5.11	210.96 / 48.09 / 162.86 / 5.63
Star	**25.59** / 9.38	158.00 / 150.00 / 8.02 / 6.02	75.57 / 19.92 / 55.65 / 4.09

Table 4. Qualitative results for XL-scale Poisson problems.

Testcase	**UGrid**	**AMGCL**	**AmgX**
Poisson (XL)	Time / Error	Time / Assembly / Iteration / Error	Time / Assembly / Iteration / Error
Bag	**36.52** / 6.95	893.28 / 869.92 / 23.35 / 4.64	280.91 / 86.22 / 194.69 / 2.80
Cat	**36.41** / 7.06	1224.02 / 1198.05 / 25.97 / 6.54	339.78 / 94.98 / 244.80 / 2.65
Lock	**36.18** / 3.09	642.52 / 622.94 / 19.58 / 4.47	191.26 / 54.01 / 137.24 / 1.99
N. Input	**36.29** / 6.16	1142.47 / 1120.98 / 21.49 / 3.61	322.05 / 93.25 / 228.80 / 0.04
Note	**36.23** / 2.70	568.23 / 550.23 / 18.00 / 2.62	168.90 / 48.42 / 120.48 / 3.17
S. Feat.	**89.50** / 7.32	1781.23 / 1748.57 / 32.67 / 5.18	Diverge
L-shape	**108.06** / 8.07	997.72 / 973.48 / 24.23 / 9.64	282.22 / 83.25 / 198.97 / 3.34
Lap. Squ.	**58.23** / 6.57	1782.50 / 1745.55 / 36.95 / 7.29	495.05 / 146.40 / 348.64 / 70.8
P. Squ.	**215.16** / 9.46	1791.26 / 1750.34 / 40.92 / 5.60	498.01 / 146.22 / 351.78 / 71.3
Star	**90.25** / 4.55	679.37 / 657.72 / 21.66 / 8.34	204.33 / 56.80 / 147.53 / 2.72

input (boundary values, boundary geometries/topology, and Laplacian distribution); (i-j) are two baseline surfaces.

In Table 1, for Poisson problems, UGrid reaches the desirable precision 10-20x faster than AMGCL, 5-10x faster than NVIDIA AmgX, and 2-5x faster than (Hsieh et al., 2019) (note that (Hsieh et al., 2019) **diverged** for most of the testcases, those cases are **not** counted). This shows the efficiency and accuracy of our method. Moreover, the testcase "Noisy Input" showcases UGrid's robustness. Furthermore, among all the ten testcases, only (the geometry of) the "Noisy Input" case is observed in the training phase of UGrid. This shows that UGrid converges to unseen scenarios whose boundary conditions are of complex geome-

Table 5. Qualitative results for XXL-scale Poisson problems.

Testcase	**UGrid**	**AMGCL**	**AmgX**
Poisson (XXL)	Time / Error	Time / Assembly / Iteration / Error	Time / Assembly / Iteration / Error
Bag	**214.16** / 6.71	3646.92 / 3570.03 / 76.88 / 7.66	828.96 / 269.75 / 559.20 / 1.15
Cat	**210.52** / 7.56	4970.15 / 4881.97 / 88.18 / 8.77	1063.20 / 359.20 / 704.00 / 1.72
Lock	**142.23** / 4.36	2466.58 / 2399.21 / 67.36 / 4.89	617.33 / 203.30 / 414.03 / 1.28
N. Input	**145.77** / 7.62	4469.40 / 4396.84 / 72.55 / 4.91	Diverge
Note	**142.67** / 2.06	2267.48 / 2211.69 / 55.78 / 6.01	538.91 / 180.52 / 358.38 / 1.23
S. Feat.	**1131.88** / 11.2	7107.18 / 6960.41 / 146.76 / 4.46	Diverge
L-shape	**1131.77** / 14.7	4055.08 / 3959.96 / 95.11 / 4.17	Diverge
Lap. Squ.	**243.03** / 7.44	7194.95 / 7049.11 / 145.84 / 6.77	1746.84 / 598.24 / 1148.60 / 38.8
P. Squ.	**1137.95** / 13.9	7376.67 / 7183.54 / 193.13 / 5.65	1813.83 / 603.60 / 1210.23 / 52.3
Star	**1118.97** / 9.36	2862.27 / 2771.75 / 90.51 / 6.05	Diverge

try (e.g., "Cat", "Star", and "L-shape") and topology (e.g., "Note", "Bag", and "Cat"). On the contrary, (Hsieh et al., 2019) **failed** to converge in most of our testcases, which verifies one of their claimed limitations (i.e., **no** guarantee of convergence to unseen cases), and showcases the strong generalization power of our method. In addition, even for those converged cases, our method is still faster than (Hsieh et al., 2019).

In Table 2, for large-scale Helmholtz problems, UGrid reaches the desirable precision 7-20x faster than AMGCL and 5-10x faster than NVIDIA AmgX.

In Table 3, for large-scale steady-state diffusion-convection-reaction problems, UGrid reaches the desirable precision 2-12x faster than AMGCL and on average 2-6x faster than NVIDIA AmgX.

Scalability. We further conducted two more sets of experiments on XL and XXL Poisson problems **without** retraining UGrid. The results are available in Tables 4 and 5. UGrid still delivers a performance boost similar to that observed in large-scale Poisson problems. These experimental results collectively validate the strong scalability of UGrid (without the need for retraining). Due to the page limit, results for "small-scale" problems are available in the appendix.

Ablation Study. The results are available in Table 6 (more details are available in Sec. A.6). The residual loss endows our UGrid model with as much as 2x speed up versus the legacy loss. The residual loss also endows UGrid to converge to the failure cases of its counterpart trained on legacy loss. These results demonstrate the claimed merits of the residual loss. On the other hand, it will **diverge** if we naively apply the vanilla U-Net architecture directly to Poisson's equations. This showcases the significance of UGrid's mathematically-rigorous network architecture.

Limitations. UGrid is currently designed for linear PDEs *only*, as Theorem. 4.1 does not hold for non-linear PDEs.

Table 6. Ablation study on large-scale Poisson problems. Columns from left to right: UGrid trained with residual loss, UGrid trained with legacy loss, and vanilla U-Net trained with residual loss.

Testcase	**UGrid**	**UGrid (L)**	**U-Net**
Poi. (L)	Time / Error	Time / Error	Time / Error
Bag	**18.66** / 2.66	28.81 / 4.86	Diverge
Cat	**10.09** / 2.70	23.80 / 1.43	Diverge
Lock	**10.55** / 9.88	Diverge	Diverge
N. Input	**10.16** / 2.64	20.65 / 2.42	Diverge
Note	**10.31** / 4.06	Diverge	Diverge
S. Feat.	**20.01** / 3.80	31.34 / 5.14	Diverge
L-shape	**15.26** / 8.43	Diverge	Diverge
Lap. Squ.	**15.10** / 3.88	30.72 / 2.76	Diverge
P. Squ.	**15.07** / 9.37	31.52 / 3.33	Diverge
Star	**15.18** / 7.50	Diverge	Diverge

Another limitation lies in the fact that there is no mathematical guarantee on *how fast* UGrid will converge. As a consequence, UGrid does not necessarily converge faster on small-scale testcases.

6. Conclusion and Future Work

This paper has articulated a novel efficient-and-rigorous neural PDE solver built upon the U-Net and the Multigrid method, naturally combining the mathematical backbone of correctness and convergence as well as the knowledge gained from data observations. Extensive experiments validate all the claimed advantages of our proposed approach. Our future research efforts will be extending the current work to non-linear PDEs. The critical algorithmic barrier between our approach and non-linear PDEs is the limited expressiveness of the convolution semantics. We would like to explore more alternatives with stronger expressive power.

Acknowledgements. This work was partially supported by: (1) USA NSF IIS-1715985 and USA NSF IIS-1812606 (awarded to Hong QIN); and (2) The National Key R&D Program of China under Grant 2023YFB3002901, and the Basic Research Project of ISCAS under Grant ISCAS-JCMS-202303.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Briggs, W. L. and McCormick, S. F. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics (SIAM), 2000.

Cichocki, A. and Unbehauen, R. Neural Networks for Solving Systems of Linear Equations and Related Problems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39(2):124–138, 1992.

Demidov, D. AMGCL: An Efficient, Flexible, and Extensible Algebraic Multigrid Implementation. *Lobachevskii Journal of Mathematics*, 40(5):535–546, 2019.

Demmel, J. W. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), 1997.

Farimani, A. B., Gomes, J., and Pande, V. S. Deep Learning the Physics of Transport Phenomena. *CoRR*, abs/1709.02432, 2017.

Greenfeld, D., Galun, M., Kimmel, R., Yavneh, I., and Basri, R. Learning to Optimize Multigrid PDE Solvers. In *Proceedings of Machine Learning Research: International Conference on Machine Learning (ICML)*, volume 97, pp. 2415–2423, 2019.

Han, J., Jentzen, A., and E, W. Solving High-dimensional Partial Differential Equations using Deep Learning. *Proceedings of the National Academy of Sciences*, 115(34): 8505–8510, 2018. doi: 10.1073/pnas.1718942115.

Hsieh, J.-T., Zhao, S., Eismann, S., Mirabella, L., and Ermon, S. Learning Neural PDE Solvers with Convergence Guarantees. In *International Conference on Learning Representations (ICLR)*, 2019.

Lagaris, I. E., Likas, A., and Fotiadis, D. I. Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *IEEE Transactions on Neural Networks and Learning Systems*, 9(5):987–1000, 1998.

Li, Z., Cheng, H., and Guo, H. General Recurrent Neural Network for Solving Generalized Linear Matrix Equation. *Complexity*, 2017:545–548, 2017.

Li, Z., Kovachki, N. B., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Fourier Neural Operator for Parametric Partial Differential Equations. In *International Conference on Learning Representations (ICLR)*, 2021.

Liao, W., Wang, J., and Wang, J. A Discrete-time Recurrent Neural Network for Solving Systems of Complex-valued Linear Equations. *International Conference in Swarm Intelligence (LCSI)*, 1(2):315–320, 2010.

Lu, L., Jin, P., Pang, G., Zhang, Z., and Karniadakis, G. E. Learning Nonlinear Operators via DeepONet Based on The Universal Approximation Theorem of Operators. *Nature Machine Intelligence*, 3(3):218–229, 2021a.

Lu, L., Meng, X., Mao, Z., and Karniadakis, G. E. DeepXDE: A deep learning library for solving differential equations. *SIAM Review (SIREV)*, 63(1):208–228, 2021b. doi: 10.1137/19M1274067.

Lu, Y., Chen, H., Lu, J., Ying, L., and Blanchet, J. Machine Learning For Elliptic PDEs: Fast Rate Generalization Bound, Neural Scaling Law and Minimax Optimality. In *International Conference on Learning Representations (ICLR)*, 2022.

Luz, I., Galun, M., Maron, H., Basri, R., and Yavneh, I. Learning Algebraic Multigrid Using Graph Neural Networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, 2020.

Marwah, T., Lipton, Z., and Risteski, A. Parametric Complexity Bounds for Approximating PDEs with Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, pp. 15044–15055, 2021.

NVIDIA Developer. AmgX. https://developer.nvidia.com/amgx, 2022.

Pang, G., D'Elia, M., Parks, M., and Karniadakis, G. nPINNs: Nonlocal Physics-informed Neural Networks for a Parametrized Nonlocal Universal Laplacian Operator. Algorithms and Applications. *Journal of Computational Physics*, 422(5):109760, 2020.

Polycarpou, M. M. and Ioannou, P. A. A Neural-type Parallel Algorithm for Fast Matrix Inversion. *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 5:108–113, 1991.

Ronneberger, O., Fischer, P., and Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351, pp. 234–241, 2015.

Saad, Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics (SIAM), second edition, 2003.

Sharma, R., Farimani, A. B., Gomes, J., Eastman, P. K., and Pande, V. S. Weakly-Supervised Deep Learning of Heat Transport via Physics Informed Loss. *CoRR*, abs/1807.11374, 2018.

Takala, J., Burian, A., and Ylinen, M. A Comparison of Recurrent Neural Networks for Inverting Matrices. *Proceedings of IEEE International Symposium on Signals, Circuits and Systems (ISSCS)*, 2:545–548, 2003.

Tang, W., Shan, T., Dang, X., Li, M., Yang, F., Xu, S., and Wu, J. Study on a Poisson's Equation Solver based on Deep Learning Technique. In *IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*, volume 16, pp. 1–3, 2017.

Tompson, J., Schlachter, K., Sprechmann, P., and Perlin, K. Accelerating Eulerian Fluid Simulation with Convolutional Networks. *Computer Animation and Virtual Worlds*, 70(3/4):3424–3433, 2017.

Wang, L. and Mendel, J. M. Structured Trainable Networks for Matrix Algebra. *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, 2:125–132, 1990a.

Wang, L. and Mendel, J. M. Three-Dimensional Structured Networks for Matrix Equation Solving. *IEEE Transactions on Computers*, 40(12):1337–1346, 1991.

Wang, L. X. and Mendel, J. M. Matrix Computations and Equation Solving using Structured Networks and Training. *IEEE Conference on Decision and Control (CDC)*, 40(12):1747–1750, 1990b.

Wu, G., Wang, J., and Hootman, J. A Recurrent Neural Network for Computing Pseudo-inverse Matrices. *Mathematical and Computer Modelling*, 20(1):13–21, 1994.

Xia, Y., Wang, J., and Hung, D. L. Recurrent Neural Networks for Solving Linear Inequalities and Equations. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 46(4):452–462, 1999.

Yang, C., Yang, X., and Xiao, X. A Comparison of Recurrent Neural Networks for Inverting Matrices. *Computer Animation and Virtual Worlds*, 27(3/4):415–424, 2016.

Zhang, Z., Zhang, L., Sun, Z., Erickson, N., From, R., and Fan, J. Solving Poisson's Equation using Deep Learning in Particle Simulation of PN Junction. In *2019 Joint International Symposium on Electromagnetic Compatibility, Sapporo and Asia-Pacific International Symposium on Electromagnetic Compatibility (EMC Sapporo & APEMC)*, pp. 305–308, 2019.

Zhou, Z., Chen, L., and Wan, L. Neural Network Algorithm for Solving System of Linear Equations. *Proceedings of International Conference on Computational Intelligence and Natural Computing (ICCIC)*, 1:7–10, 2009.

Özbay, A. G., Hamzehloo, A., Laizet, S., Tzirakis, P., Rizos, G., and Schuller, B. Poisson CNN: Convolutional Neural Networks for The Solution of The Poisson Equation on A Cartesian Mesh. *Data-Centric Engineering*, 2:e6, 2021.

A. Appendix

This supplemental material is provided to readers in the interest of our paper's theoretical and experimental completeness.

A.1. More Specifications on Our Training Data

UGrid is trained **only** with pairs of boundary masks and boundary values as shown in Fig. 4 (h). To be specific, throughout the whole training phase, UGrid is exposed only to *zero* $\mathbf{f}$-fields and piecewise-constant Dirichlet boundary conditions with the "Donut-like" geometries. UGrid is **unaware** of all other complex geometries, topology, as well as the irregular/noisy distribution of boundary values/Laplacians observed in our testcases. This showcases the strong generalization power of our UGrid neural solver.

A.2. More Specifications on Our Testcases

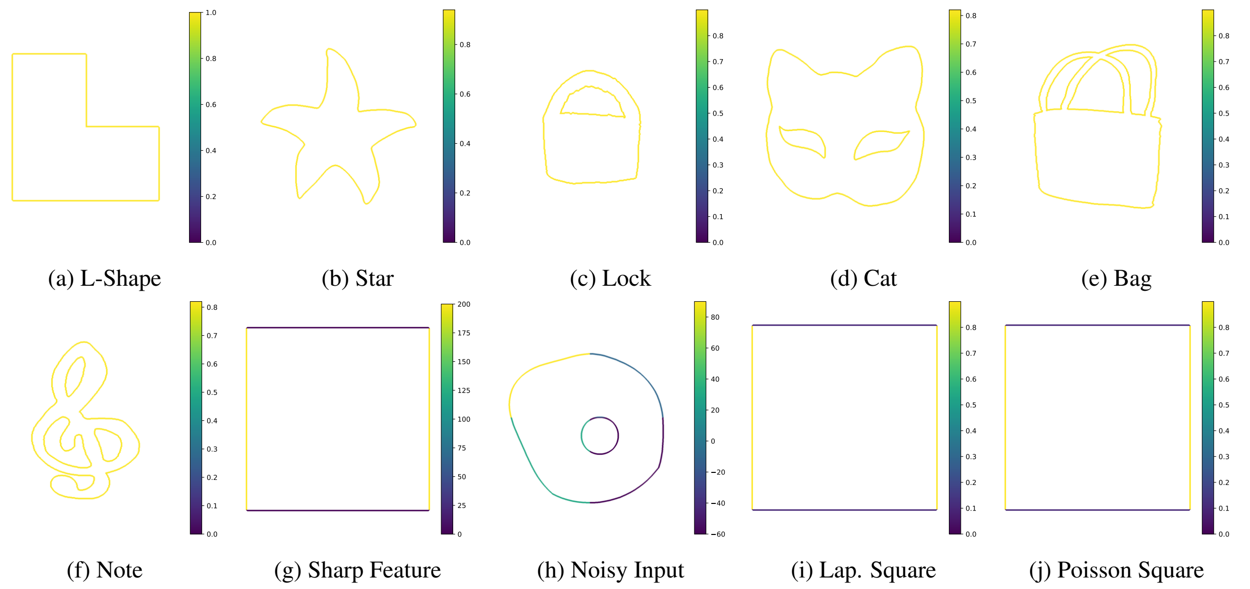


Figure 4. Illustration of the Dirichlet boundary values of our ten testcases. Again, all boundaries are shown in bold for a better view. Note that these boundaries are **not** required to be constant and could have *discontinuities*, which could be observed in testcases (g-j).

A.3. More Specifications on Our Qualitative Evaluations

Baseline Specifications. AMGCL is a C++ multigrid library with multiple GPU backends, we are comparing with its CUDA backend, with CG solver at the coarsest level. AmgX is part of NVIDIA's CUDA-X GPU-Accelerated Library, and we adopt its official `AMGX_LEGACY_CG` configuration. (Hsieh et al., 2019)'s code is available at `https://github.com/ermongroup/Neural-PDE-Solver`. They did not release a pre-trained model, so we train their model with configurations as-is in their training and data-generation scripts, with minimal changes to make the program run.

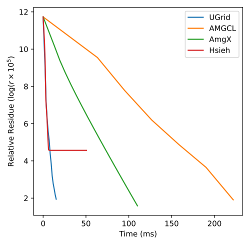
Testing. All of our testcases are tested for 100 times and the results are averaged. For UGrid and (Hsieh et al., 2019), we set the maximum number of iterations as 64, and the iteration is terminated immediately upon reaching this threshold, no matter whether the current numerical solution has reached the desirable precision. AmgX has no direct support for relative residual errors, so we set tolerance on absolute residual errors case-by-case to achieve similar precision.

In our qualitative results, the "time" columns for AMGCL and AmgX include MG hierarchy building time and do not include training time for UGrid. This is because: (1) Training is required only once for one type of PDEs, and could be ignored over the solver's lifespan; and (2) AMGCL/AmgX must reconstruct their MG hierarchy when input grid or boundary geometry changes; UGrid doesn't need retraining for these cases. When MG hierarchy is constructed already, and only RHS changes, AMGCL/AmgX performs better. However, in fields like PDE-based CAD, grid and boundary-geometry changes as frequently as RHS, and UGrid will be a better choice.

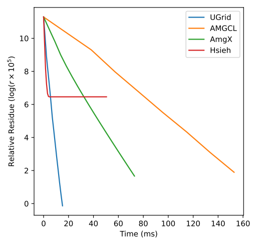
For experimental completeness, for AMGCL and AMGX, we also separately report the MG hierachy building time as well as iteration time in Tables 1, 2 and 3. The results validate that AMGX is slower than UGrid even without assembly time, and compared to UGrid, AMGCL has slightly better efficiency (without the assembly phase), yet its assembly phase takes 10-20x time compared to UGrid's overall time consumption.

Convergence Maps. For the experimental completeness of this paper, we also provide readers with the convergence maps of UGrid and the three SOTA solvers we compare with. The convergence maps are plotted for all of our ten testcases, each for its two different scales. We further plot these convergence maps as functions of time, and functions of iterations.

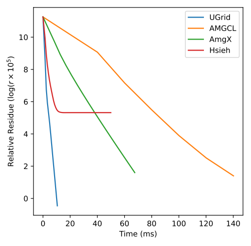
The convergence maps for large-scale problems are as follows:



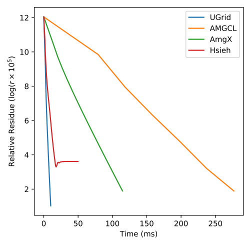
(a) L-Shape



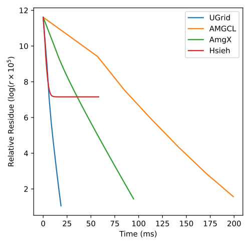
(b) Star



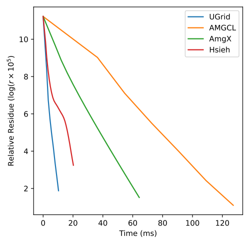
(c) Lock



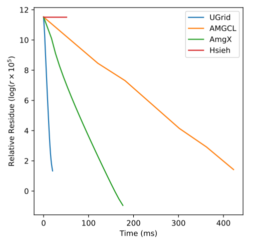
(d) Cat



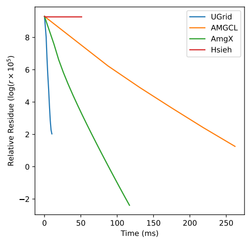
(e) Bag



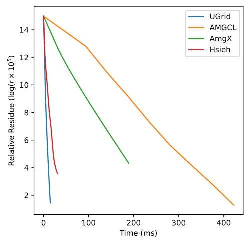
(f) Note



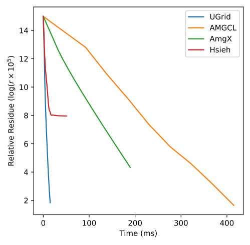
(g) Sharp Feature



(h) Noisy Input

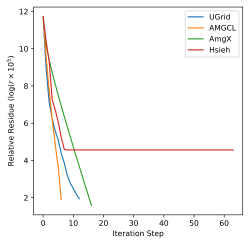


(i) Lap. Square

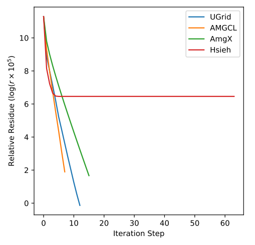


(j) Poisson Square

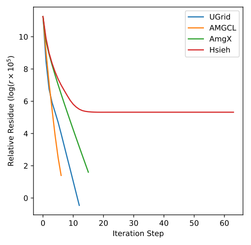
Figure 5. Convergence map on large-scale Poisson problem. The x coordinates are time(s), shown in ms; the y coordinates are the relative residual errors, shown in logarithm ($\log\left(r \times 10^5\right)$) for a better view.



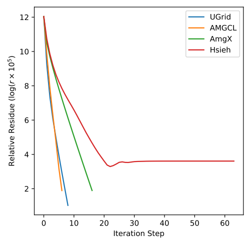
(a) L-Shape



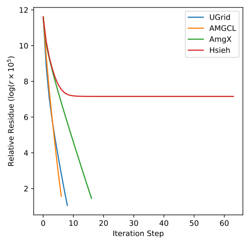
(b) Star



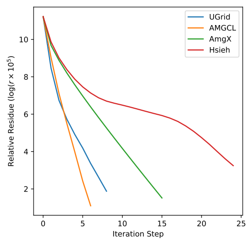
(c) Lock



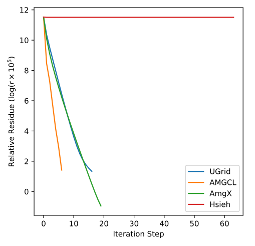
(d) Cat



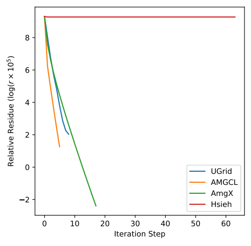
(e) Bag



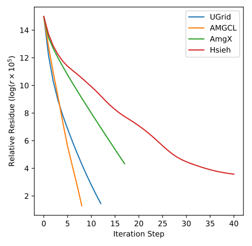
(f) Note



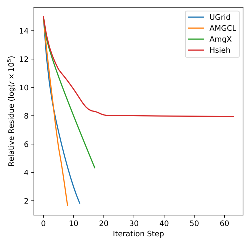
(g) Sharp Feature



(h) Noisy Input



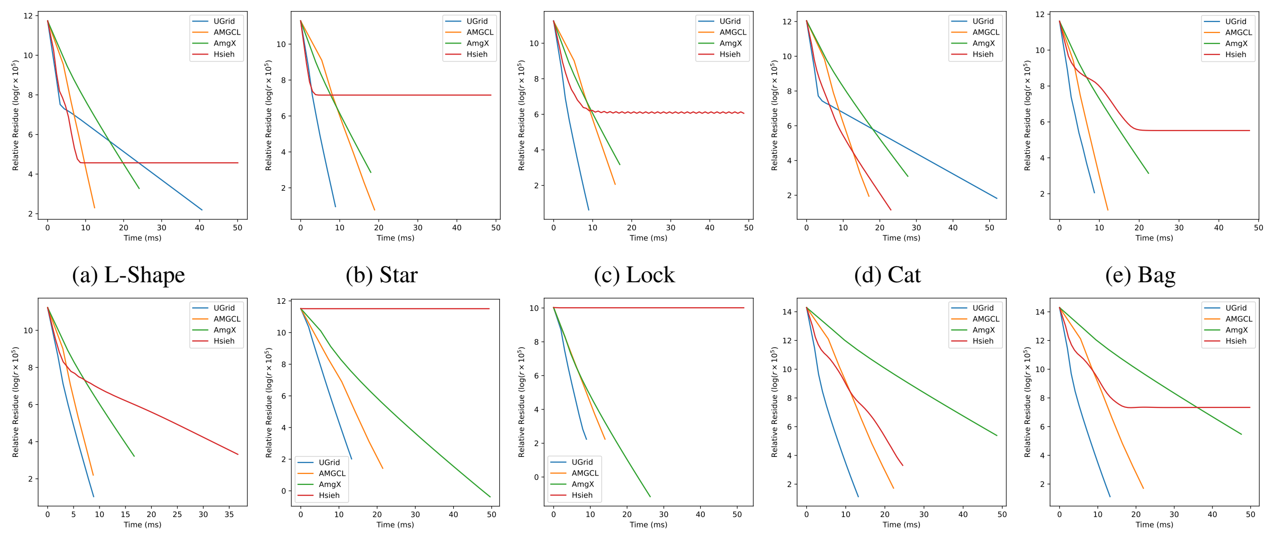
(i) Lap. Square



(j) Poisson Square

Figure 6. Convergence map on large-scale Poisson problem. The x coordinates are the iteration steps; the y coordinates are the relative residual errors, shown in logarithm ($\log\left(r \times 10^5\right)$) for a better view.

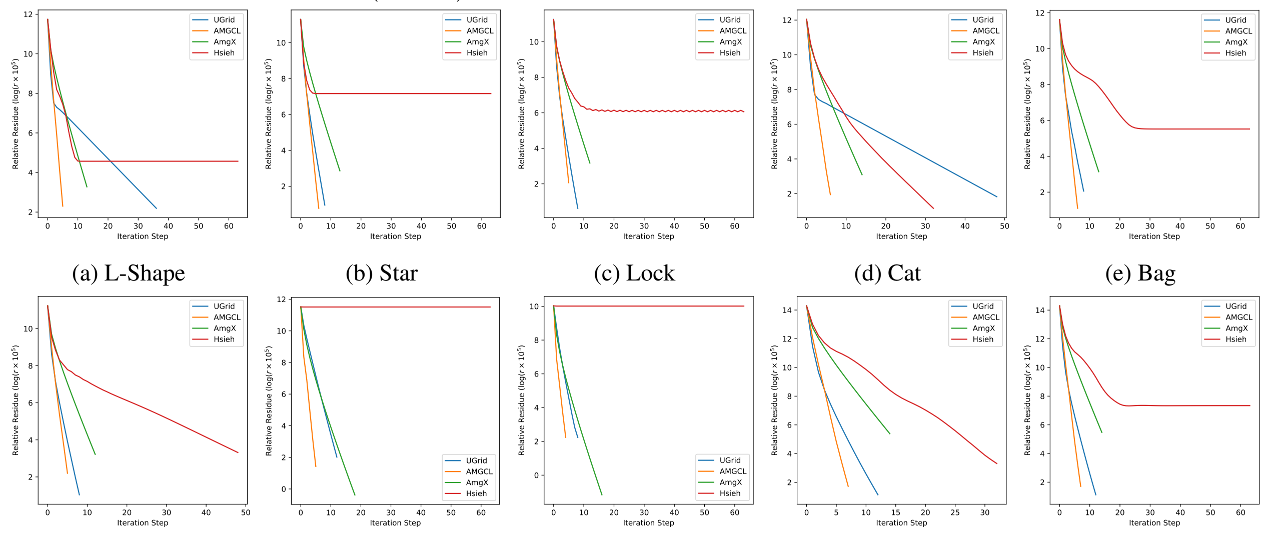
We also provide the convergence maps for small-scale problems as follows:



(a) L-Shape (b) Star (c) Lock (d) Cat (e) Bag

(f) Note (g) Sharp Feature (h) Noisy Input (i) Lap. Square (j) Poisson Square

Figure 7. Convergence map on small-scale Poisson problem. The x coordinates are time(s), shown in ms; the y coordinates are the relative residual errors, shown in logarithm ($\log(r \times 10^5)$) for a better view.



(a) L-Shape (b) Star (c) Lock (d) Cat (e) Bag

(f) Note (g) Sharp Feature (h) Noisy Input (i) Lap. Square (j) Poisson Square

Figure 8. Convergence map on small-scale Poisson problem. The x coordinates are the iteration steps; the y coordinates are the relative residual errors, shown in logarithm ($\log(r \times 10^5)$) for a better view.

A.4. Proof of Correntness of Eq. 8

The proof of correctness of Eq. 8 originates from (Hsieh et al., 2019). For the completeness of this paper, we will also go through the mathematical proof. We start with the following Lemmas and Theorems:

Lemma A.1. *For a fixed linear iterator in the form of*

$$\mathbf{u}_{k+1} = \mathbf{G} \cdot \mathbf{u}_k + \mathbf{c}\ , \tag{21}$$

with a square update matrice $\mathbf{G}$ *having a spectral radius* $\rho(\mathbf{G}) < 1$, $\mathbf{I} - \mathbf{G}$ *is non-singular, and Eq. 21 converges for any constant* $\mathbf{c}$ *and initial guess* $\mathbf{u}_0$. *Conversely, if Eq. 21 converges for any* $\mathbf{c}$ *and* $\mathbf{u}_0$, *then* $\rho(\mathbf{G}) < 1$.

Proof. Proved as Theorem 4.1 in (Saad, 2003). □

Lemma A.2. *For all operator norms $\|\cdot\|_k$, $k = 1, 2, \ldots, \infty$, the spectral radius of a matrix $\mathbf{G}$ satisfies $\rho(\mathbf{G}) \leq \|\mathbf{G}\|_k$.*

Proof. Proved as Lemma 6.5 in (Demmel, 1997). □

Theorem A.3. *For a PDE in the form of Eq. 2, the masked iterator Eq. 8 converges to its ground-truth solution when its prototype Jacobi iterator converges and $\mathbf{P}$ is full-rank diagonal.*

Proof. To prove this Theorem, we only need to prove: (1) Eq. 8 converges to a fixed point $\mathbf{u}$; and (2) The fixed point $\mathbf{u}$ satisfies Eq. 2.

To prove (1), we only need to prove that for the update matrix

$$\mathbf{G} = (\mathbf{I} - \mathbf{M})\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right),$$

its spectral radius $\rho(\mathbf{G}) < 1$. Given that the prototype Jacobi iterator converges, we have $\rho\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right) < 1$. From Lemma A.2, taking the *spectral norm* $\|\cdot\|_2$ (i.e., $k = 2$), we have

$$\rho(\mathbf{G}) \leq \left\|(\mathbf{I} - \mathbf{M})\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\right\|_2 \leq \|\mathbf{I} - \mathbf{M}\|_2 \left\|\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right\|_2 .$$

Furthermore, because $\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}$ is symmetric, we have $\rho\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right) = \left\|\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right\|_2$. On the other hand, because $\mathbf{I} - \mathbf{M} \in \{0, 1\}^{n^2 \times n^2}$ is a binary diagonal matrix, we have $\|\mathbf{I} - \mathbf{M}\|_2 = 1$. This yields $\rho(\mathbf{G}) < 1$.

To prove (2), we first notice that the fixed point $\mathbf{u} = \mathbf{u}_{k+1} = \mathbf{u}_k$ of Eq. 8 satisfies

$$\begin{aligned} \mathbf{u} &= (\mathbf{I} - \mathbf{M})\left(\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\mathbf{u} + \mathbf{P}^{-1}\mathbf{f}\right) + \mathbf{M}\mathbf{b}, \quad \text{i.e.,} \\ (\mathbf{I} - \mathbf{M})\,\mathbf{u} + \mathbf{M}\mathbf{u} &= (\mathbf{I} - \mathbf{M})\left(\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\mathbf{u} + \mathbf{P}^{-1}\mathbf{f}\right) + \mathbf{M}\mathbf{b}. \end{aligned}$$

Again, since $\mathbf{M} \in \{0, 1\}^{n^2 \times n^2}$ is a binary diagonal matrix, we have

$$\begin{cases} (\mathbf{I} - \mathbf{M})\,\mathbf{u} = (\mathbf{I} - \mathbf{M})\left(\left(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}\right)\mathbf{u} + \mathbf{P}^{-1}\mathbf{f}\right) \\ \qquad \mathbf{M}\mathbf{u} = \mathbf{M}\mathbf{b} \end{cases} . \tag{22}$$

The second equation in Eq. 22 is essentially the second equation in Eq. 2. Furthermore, the first equation in Eq. 22 could be simplified into $(\mathbf{I} - \mathbf{M})\mathbf{P}^{-1}(\mathbf{A}\mathbf{u} - \mathbf{f}) = \mathbf{0}$. Since $\mathbf{P}$ is full-rank diagonal, $\mathbf{P}^{-1}$ should also be full-rank diagonal. Then we have $(\mathbf{I} - \mathbf{M})(\mathbf{A}\mathbf{u} - \mathbf{f}) = \mathbf{0}$, which means that $\mathbf{u}$ also satisfies the first equation in Eq. 2. □

A.5. Proof of Theorem. 4.2

Proof. Denote $\varepsilon_\mathbf{x}$ as $\mathbf{x}$'s relative residual error, then we have:

$$\begin{aligned} \varepsilon_\mathbf{x} &= \frac{\left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{x}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} \approx \frac{\left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,(\mathbf{y} \pm l_{\max}\,\mathbf{y})\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} = \frac{\left\|\left(\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{y}\right) \mp \left(l_{\max}\,\widetilde{\mathbf{A}}\,\mathbf{y}\right)\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} \\ &\leq \frac{\left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{y}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} + \frac{\left\|l_{\max}\,\widetilde{\mathbf{A}}\,\mathbf{y}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} = \varepsilon_\mathbf{y} + \frac{\left\|l_{\max}\,\widetilde{\mathbf{A}}\,\mathbf{y}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|}, \end{aligned}$$

where $\varepsilon_\mathbf{y}$ denotes the relative residual error of the "ground-truth" value $\mathbf{y}$. $\varepsilon_\mathbf{y} \neq 0$ because in most cases, a PDE's ground-truth solution could only be a numerical approximation with errors. The upper bound is input-dependent because $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{f}}$ are input-dependent.

Similarly, we could prove that the lower bound of $\varepsilon_{\mathbf{x}}$ is also input-dependent:

$$\varepsilon_{\mathbf{x}} = \frac{\left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{x}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} \approx \frac{\left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\left(\mathbf{y} \pm l_{\max}\,\mathbf{y}\right)\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} = \frac{\left\|\left(\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{y}\right) \mp \left(l_{\max}\,\widetilde{\mathbf{A}}\,\mathbf{y}\right)\right\|}{\left\|\widetilde{\mathbf{f}}\right\|}$$

$$\geq \left| \frac{\left\|\widetilde{\mathbf{f}} - \widetilde{\mathbf{A}}\,\mathbf{y}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} - \frac{\left\|l_{\max}\,\widetilde{\mathbf{A}}\,\mathbf{y}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} \right| = \left| \varepsilon_{\mathbf{y}} - \frac{\left\|l_{\max}\,\widetilde{\mathbf{A}}\,\mathbf{y}\right\|}{\left\|\widetilde{\mathbf{f}}\right\|} \right| > 0\,.$$

□

A.6. Ablation Study

For experimental completeness, we conduct ablation studies with respect to our residual loss and the UGrid architecture itself. In addition to the UGrid model trained with residual loss (as proposed in the main content), we also train another UGrid model with legacy loss, as well as one vanilla U-Net model with residual loss. This U-Net model has the same number of layers as UGrid, and has non-linear layers as proposed in (Ronneberger et al., 2015). We let the U-Net directly regress the solutions to Poisson's equations. All these models are trained with the same data in the same manner (except for the loss metric), as detailed in Section 5.

We conduct qualitative experiments on the same set of testcases as detailed in Section 5, and the results are as follows:

Table 7. Ablation study on large-scale Poisson problems.

Testcase	**UGrid**	**UGrid (L)**	**U-Net**
Poisson (L)	Time / Error	Time / Error	Time / Error
Bag	**18.66** / 2.66	28.81 / 4.86	81.71 / 1384131
Cat	**10.09** / 2.70	23.80 / 1.43	70.09 / 2539002
Lock	**10.55** / 9.88	Diverge	70.92 / 1040837
N. Input	**10.16** / 2.64	20.65 / 2.42	73.05 / 21677
Note	**10.31** / 4.06	Diverge	69.97 / 614779
S. Feat.	**20.01** / 3.80	31.34 / 5.14	70.08 / 222020
L-shape	**15.26** / 8.43	Diverge	74.67 / 1800815
Lap. Squ.	**15.10** / 3.88	30.72 / 2.76	72.24 / 30793035
P. Squ.	**15.07** / 9.37	31.52 / 3.33	71.74 / 31043896
Star	**15.18** / 7.50	Diverge	70.01 / 1138821

In Table 7, the residual loss endows our UGrid model with as much as 2x speed up versus the legacy loss. The residual loss also endows UGrid to converge to the failure cases of its counterpart trained on legacy loss. These results demonstrate the claimed merits of the residual loss. On the other hand, it will **diverge** if we naively apply the vanilla U-Net architecture directly to Poisson's equations. For experimental completeness only, we list the diverged results in the last column. (The "time" column measures the time taken for 64 iterations; the iterators are shut down once they reach this threshold.) This showcases the significance of UGrid's mathematically-rigorous network architecture.

In Table 8, for small-scale problems, the residual loss still endows UGrid with as much as 2x speedup and stronger generalization power against its counterpart trained with legacy loss. Once again, the vanilla U-Net model diverged for all testcases, and we list its diverged results for experimental completeness only.

A.7. Qualitative Evaluations on Small-scale Poisson Problems

In Table 9, even on small-scale problems that hinder our solver with a compact multigrid-like hierarchy from delivering its full power, the UGrid model is still faster than or exhibits comparable efficiency with respect to the three SOTA legacy/neural solvers. Again, this shows the high efficiency as well as the strong generalization power of our new method. The testcases "Cat" and "L-shape" showcase that the generalization power (in terms of problem size) does come with a price of potentially downgraded efficiency. Thus, for the sake of the best efficiency, we still recommend re-training UGrid for problems of different sizes.

Table 8. Ablation study on small-scale Poisson problems.

Testcase	**UGrid**	**UGrid (L)**	**U-Net**
Poisson (S)	Time / Error	Time / Error	Time / Error
Bag	**8.76** / 8.05	17.89 / 4.50	71.86 / 678141
Cat	**51.96** / 6.21	Diverge	68.89 / 1317465
Lock	**9.00** / 2.11	18.32 / 2.83	69.47 / 189412
Noisy Input	**8.94** / 6.00	17.88 / 6.58	69.54 / 21666
Note	**8.87** / 2.75	17.79 / 3.06	69.59 / 24715
Sharp Feature	**13.31** / 7.52	26.64 / 1.91	70.57 / 191499
L-shape	**40.60** / 7.09	Diverge	69.71 / 1011364
Laplacian Square	**13.21** / 3.27	22.23 / 9.55	73.80 / 15793109
Poisson Square	**13.21** / 2.88	22.13 / 9.76	71.56 / 15393069
Star	**8.92** / 2.36	17.60 / 5.69	73.72 / 502993

Table 9. Comparison of UGrid and state-of-the-art on small-scale Poisson problems.

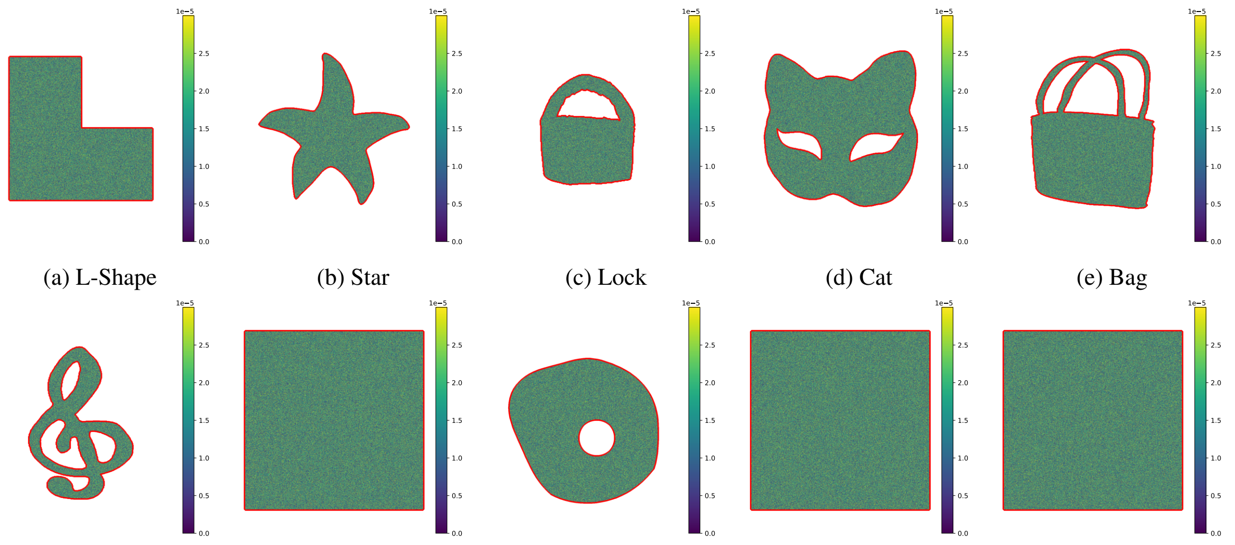
Testcase	**UGrid**	**AMGCL**	**AmgX**	**Hsieh et al.**
Poisson (S)	Time / Error	Time / Error	Time / Error	Time / Error
Bag	**8.76** / 8.05	12.14 / 3.00	22.34 / 8.20	47.69 / 252
Cat	51.96 / 6.21	**17.03** / 6.98	27.66 / 4.83	23.02 / 9.95
Lock	**9.00** / 2.11	15.77 / 7.89	16.96 / 9.36	48.72 / 117.9
Noisy Input	**8.94** / 6.00	14.00 / 9.39	26.30 / 3.14	51.79 / 5576
Note	8.87 / 2.75	**8.79** / 9.02	16.68 / 7.23	36.66 / 8.28
Sharp Feature	**13.31** / 7.52	21.47 / 4.15	49.59 / 6.85	49.31 / 24876
L-shape	40.60 / 7.09	**12.36** / 9.97	24.08 / 9.35	50.06 / 96.44
Laplacian Square	**13.21** / 3.27	22.22 / 5.60	48.60 / 3.98	24.57 / 6.54
Poisson Square	**13.21** / 2.88	21.93 / 5.51	47.56 / 4.03	49.77 / 473
Star	**8.92** / 2.36	18.93 / 2.17	17.96 / 9.42	48.68 / 456

A.8. Specifications on Evaluations of Inhomogeneous Helmholtz Problems with Spatially-varying Wavenumbers

We train UGrid with the same training data and residual loss as mentioned in Section 5. As one exception, we also input randomly-sampled k^2 during training, evaluation, and testing.

The randomly-sampled k^2s we used are illustrated in Fig. 9. For qualitative experiments, we use the same boundary conditions and Laplacian distributions as shown in Fig. 4 and Fig. 3, and we randomly initialize the wavenumber field k^2 across the whole domain, resulting in a noisy distribution.

The qualitative results for large-scale problems are available in the main paper as Table 2. Qualitative results for small-scale problems are available in Table 10. We could observe that even on small-scale problems that hinder our solver with a compact multigrid-like hierarchy from delivering its full power, UGrid is still faster than or exhibits comparable efficiency with respect to the SOTA.



(a) L-Shape (b) Star (c) Lock (d) Cat (e) Bag

(f) Note (g) Sharp Feature (h) Noisy Input (i) Lap. Square (j) Poisson Square

Figure 9. Illustration of the wavenumber distributions of our testcases. The boundaries are shown in bold red lines for a better view. (Boundary values are shown in Fig. 4.)

Table 10. Qualitative results for small-scale Helmholtz problems.

Testcase	**UGrid**	**AMGCL**	**AmgX**
Helmholtz (S)	Time / Error	Time / Error	Time / Error
Bag	14.70 / 6.29	**12.92** / 3.00	25.82 / 1.79
Cat	**16.63** / 7.86	16.82 / 6.98	28.37 / 3.03
Lock	**9.78** / 5.87	16.23 / 7.88	19.75 / 2.02
Noisy Input	14.95 / 0.76	**14.34** / 9.40	28.92 / 0.04
Note	14.37 / 8.28	**9.01** / 9.02	18.76 / 2.55
Sharp Feature	**19.46** / 1.18	21.37 / 4.21	52.82 / 0.13
L-shape	14.64 / 0.88	**12.29** / 9.99	26.90 / 2.10
Laplacian Square	**14.60** / 4.60	22.43 / 5.59	43.68 / 17.8
Poisson Square	**15.27** / 6.53	22.35 / 5.50	43.57 / 17.2
Star	**9.77** / 5.96	19.09 / 2.16	20.89 / 2.18

A.9. Specifications on Evaluations of Inhomogeneous Steady-state Convection-diffusion-reaction Problems

We train UGrid in the same manner as for Helmholtz equations. As one exception, we input randomly-sampled $\mathbf{v}$s, αs, and βs during training, evaluation, and testing. These values are sampled using the same routine as for Helmholtz equations, resulting in noisy velocity fields like Fig. 9 as well as randomized α, β coefficients ($4\alpha + \beta \neq 0$). The qualitative results for large-scale problems are available in the main paper as Table 3. Qualitative results for the small-scale problem are available in Table 11.

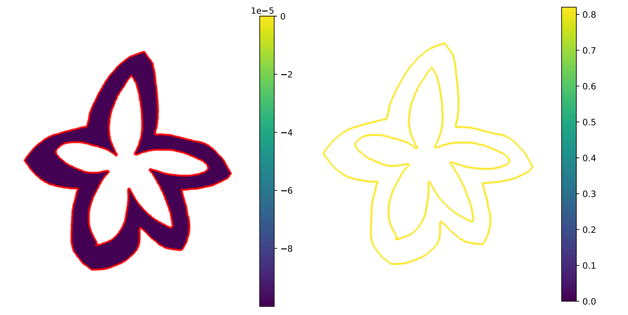
Table 11. Qualitative results for small-scale Diffusion problems.

Testcase Diffusion (S)	**UGrid** Time / Error	**AMGCL** Time / Error	**AmgX** Time / Error
Bag	16.99 / 3.79	**12.34** / 9.94	22.58 / 3.66
Cat	69.12 / 8.76	**16.57** / 9.79	27.44 / 6.74
Lock	17.07 / 1.43	**16.07** / 6.34	18.34 / 4.06
Noisy Input	22.84 / 6.47	**15.94** / 2.74	24.35 / 0.42
Note	22.76 / 1.17	**9.38** / 2.67	19.25 / 3.79
Sharp Feature	**17.05** / 5.15	21.41 / 4.29	41.67 / 0.72
L-shape	35.18 / 6.25	**13.02** / 2.30	25.51 / 3.97
Laplacian Square	90.73 / 64.5	**22.14** / 8.17	50.49 / 3.50
Poisson Square	50.95 / 5.01	**22.05** / 7.34	50.06 / 3.28
Star	**17.06** / 3.69	18.70 / 7.55	18.88 / 4.71

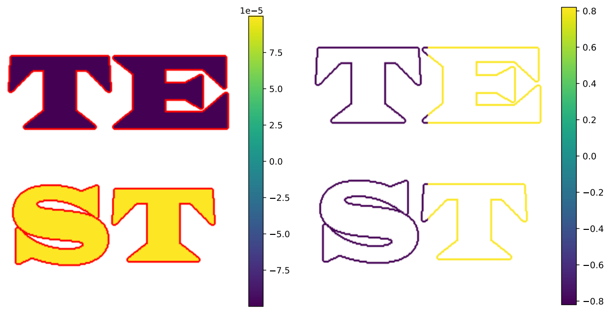
In Table 11, again, even on small-scale problems that hinder our solver with a compact multigrid-like hierarchy from delivering its full power, UGrid is still exhibits comparable efficiency with respect to the SOTA. This demonstrates UGrid's generalization power over problem sizes, though possibly at the price of relatively lower efficiency compared to the size it is trained on.

A.10. Additional Experiments on Poisson Problems

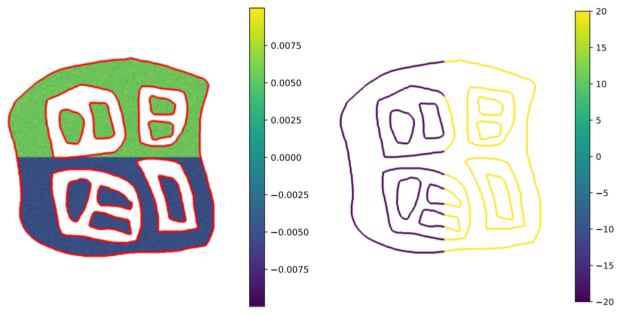
We have conducted four additional experiments to showcase the generalization power of the proposed UGrid solver. The four testcases are illustrated as follows:



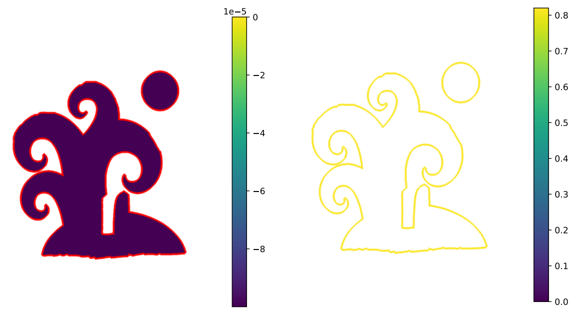
(a) Flower (Left: Laplacian; Right: Boundary)



(b) Word (Left: Laplacian; Right: Boundary)



(c) Topo (Left: Laplacian; Right: Boundary)



(d) Tree (Left: Laplacian; Right: Boundary)

Figure 10. Illustration of four additional testcases.

The qualitative results are as follows:

Table 12. Qualitative results for additional large-scale Poisson problems. "Time" denotes the total time (ms) to reach relative residual errors $\leq 10^{-4}$, and "Assembly" and "Iteration" denotes time (ms) for the two phases for AMG solvers; "Error" denotes the final relative residual errors, divided by 10^{-5}.

Testcase	**UGrid**	**AMGCL**	**AmgX**	**Hsieh et al.**
Poisson (L)	Time / Error	Time / Assembly / Iteration / Error	Time / Assembly / Iteration / Error	Time / Error
Flower	**18.52** / 1.60	144.91 / 138.00 / 6.91 / 3.07	69.14 / 18.11 / 51.03 / 5.41	27.23 / 7.16
Word	**41.14** / 5.60	188.59 / 178.34 / 10.25 / 4.15	108.51 / 21.44 / 87.06 / 4.62	50.76 / 2612
Topo	**10.30** / 4.29	210.06 / 203.86 / 6.20 / 4.86	97.22 / 24.29 / 72.93 / 2.94	52.29 / 3933
Tree	**10.05** / 2.52	173.70 / 166.86 / 6.84 / 4.61	78.73 / 20.98 / 57.76 / 6.88	16.05 / 7.60

In Table 12, UGrid still delivers similar efficiency, accuracy and generalization power like the ten testcases covered in the main contents of the paper. Note that Hsieh et al. **diverged** for testcases ”Word” and ”Topo”; their time for these two cases is the time to reach the maximum number of iterations.