UNSTRUCTURED TO STRUCTURED: GEOMETRIC MULTIGRID ON COMPLEX DOMAINS VIA MESH REMAPPING [*]

NICOLAS NYTKO[†]

In collaboration with:
Scott MacLachlan, J. David Moulton, Luke Olson, Andrew Reisner, Matt West

Abstract. For domains with sufficient structure and regularity, geometric multigrid solvers are among the fastest for computing the numerical solution to elliptic PDEs; however, for complicated domains, constructing a suitable hierarchy of meshes becomes challenging. We propose a framework for mapping computations from such complex domains to a regular computational domain via diffeomorphism, enabling the use of black-box multigrid. This mapping facilitates regular memory accesses during solves, improving efficiency and scalability, especially on massively parallel processors such as GPUs. Moreover, we show that the diffeomorphic mapping itself may be approximately learned using an invertible neural network, enabling automated application to geometries where no analytic mapping exists.

Key words. geometric multigrid, black box, boxmg, diffeomorphic, lddmm, learning, neural ode, gpu

1. Introduction.

Geometric multigrid methods are among the fastest iterative methods for elliptic partial differential equations (PDEs), offering an optimal time complexity and rapid convergence for problems where geometric structure can be exploited. However, extending these for arbitrary complicated domains remains challenging due to the difficulty of constructing a suitable hierarchy of coarse meshes that maintains multigrid efficiency. In practice, algebraic multigrid methods [12] (AMG) are often used on complex meshes as they do not require geometric information about the problem, but rather operate on the underlying linear system itself. Despite their versability, however, AMG tends to suffer from a higher setup cost as more expensive graph algorithms must be employed to coarsen the problem. Morever, they suffer from poor data locality and irregular data accesses resulting from inefficient sparse matrix memory access patterns that hinder performance on massively parallel architectures such as GPUs [2].

To address these limitations, we propose a novel framework to extend the raw power and speed of geometric multigrid to a larger class of problems. Solutions are transferred from complex geometries to a more simple, structured mesh through a specified diffeomorphic mapping. A full multigrid hierarchy is constructed by transferring solution vectors from the unstructured mesh to the structured one and then employing a black-box multigrid solver [4, 5, 11] to quickly and efficiently solve the structured problem. This auxiliary solution is then transferred back to the unstructured mesh where only cheap local relaxation is performed. This results in a solver that leverages the regularity of the structured problem to enable an efficient solver while also allowing for essential features of the physical geometry to be preserved.

Furthermore, we show that it is possible for this diffeomorphic mapping between domains to be learned. By taking inspiration from existing work on large deformation diffeomorphic metric mappings [8] (LDDMMs), used heavily in computational

[*]This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0003963.

[†]University of Illinois Urbana-Champaign, (nnytko2@illinois.edu, https://nicknytko.github.io/)

anatomy, we demonstrate that such a mapping can be approximately learned for geometries lacking an analytic conformal mapping. We show numerical results for analytic mappings and preliminary results for learned mappings, indicating that geometric solvers constructed by our method are competitive with off the shelf algebraic multigrid solvers such as HYPRE [9].

2. Method Overview.

The PDE we are considering in this work is the elliptic diffusion problem with Dirichlet boundary conditions,

$$-\nabla \cdot \mathcal{D}\nabla u = f \quad \text{in } \Omega_p, \tag{2.1}$$

$$u = g \quad \text{on } \partial\Omega_p, \tag{2.2}$$

defined over some polygonal *physical domain* $\Omega_p = \cup_{i=1}^{|\mathcal{T}_p|} \tau_p^i \subset \mathbb{R}^d$ with triangulation $\mathcal{T}_p = \left\{\tau_p^i\right\}$, and SPD diffusion coefficient $\mathcal{D} : \mathbb{R}^d \to \mathbb{R}^d$. We will discuss the case of $d = 2$, though the method presented can be generalized to $d = 3$ as well.

In addition to the physical domain, we will define an auxiliary *computational domain*, $\Omega_c = [-1, 1]^d$, as well as a diffeomorphism $T : \Omega_p \mapsto \Omega_c$ to map functions between the two domains. We require T to be diffeomorphic to ensure the Jacobian $J_T(\boldsymbol{x})$ and its inverse exist and are continuous.

We use this auxiliary domain to transfer both functions in the weak form and discrete vectors in the solution process. This has several benefits, mainly:

1. Geometric multigrid on a square domain is cheap to setup and run, and
2. memory access patterns are much more regular than on an unstructured mesh, giving a speedup on compute platforms that have performance bottlenecks resulting from memory transfers, rather than raw compute power, such as GPUs.

We will first consider example domains where T is known analytically, though in section 3 we will show how T can be learned for more complex domains.

The PDE is discretized on the computational domain by finite elements through a change of coordinates defined by T, in a similar fashion to how a standard finite element assembly is performed on, e.g., a triangular mesh. We begin with the regular weak form for the diffusion equation, we wish to find $\hat{u} \in H_0^1(\Omega_p)$ for the solution $u(\boldsymbol{x}) = \hat{u}(\boldsymbol{x}) + \hat{g}(\boldsymbol{x})$ that satisfies

$$\int_{\Omega_p} \mathcal{D}\nabla u \cdot \nabla v \; dA = \int_{\Omega_p} f v \; dA \qquad \forall v \in H_0^1(\Omega_p), \tag{2.3}$$

where $H^1(\Omega_p)$ is the standard Hilbert space of continuous functions over Ω_p, $H_0^1(\Omega_p)$ is the restriction of $H^1(\Omega_p)$ to functions that vanish on the boundary, and $\hat{g} \in H^1(\Omega_p)$ satisfies $\hat{g}(\boldsymbol{x}) = g$ on $\partial\Omega_p$. We then transfer this weak form to Ω_c with J_T to obtain

$$\int_{\Omega_c} (\mathcal{D} J_T^T \nabla u) \cdot (J_T^T \nabla v) \; |J_T^{-1}| \; dA = \int_{\Omega_c} f v \; |J_T^{-1}| \; dA \qquad \forall v \in H_0^1(\Omega_c). \tag{2.4}$$

The computational domain is then "meshed" into a logically structured grid of regular elements $\mathcal{T}_c = \left\{\tau_c^i\right\}$. Letting the reference element be defined by $\hat{\tau}_c = [0, 1]^d$ and the Jacobian of element τ_i^c by $J_{\tau_i^c}$, we obtain the final weak form

$$\sum_{i=1}^{|\mathcal{T}_c|} |J_{\tau_c^i}| \int_{\hat{\tau}_c} (\mathcal{D} J_T^T J_{\tau_c^i}^{-T} \nabla u) \cdot (J_T^T J_{\tau_c^i}^{-T} \nabla v) \; |J_T^{-1}| \; dA = \sum_{i=1}^{|\mathcal{T}_c|} |J_{\tau_c^i}| \int_{\hat{\tau}_c} f v \; |J_T^{-1}| \; dA \qquad \forall v \in H_0^1(\hat{\tau}_c). \tag{2.5}$$

Remark. *In the above weak form, we have inverse element Jacobians as we transform from Ω_c to the reference element space (opposite of the element transform), and we have a regular Jacobian of T as we transform from Ω_p to Ω_c, which follows the definition of T.*

Equation (2.5) is then discretized by restricting the test and trial function spaces to a standard Q1 bilinear basis,

$$\boldsymbol{\phi}(x,y) = \begin{bmatrix}(1-x)(1-y) & x(1-y) & (1-x)y & xy\end{bmatrix}^T . \tag{2.6}$$

We then approximate the integrals in (2.5) and assemble a linear system $\boldsymbol{A}_c$ as is standard for finite elements; in practice the quadrature rule for integration does not need to be of very high order (midpoint or trapezoidal rule in each dimension is often sufficient) in order to obtain a convergent method.

While in the above formulation it is possible to have rectangular meshes with variable grid spacing (for example, a Shishkin mesh [13]), we will for simplicity use computational meshes with uniform grid spacing $\Delta x, \Delta y$, giving element Jacobians $J_{\tau_1^c} = J_{\tau_2^c} = \cdots = J_c$.

Definition 2.1 (Effective anisotropy). *Let the effective anisotropy, $\mathcal{D}_{\mathit{eff}} : \Omega_c \to \mathbb{R}^{d\times d}$, mapping coordinates on the reference quad to a diffusion tensor, be defined by*

$$\mathcal{D}_{\mathit{eff}}(\boldsymbol{x}) := J_c^{-1} J_T \mathcal{D}(T^{-1}(\boldsymbol{x})) J_T^T J_c^{-T} = J_T \mathcal{D}(T^{-1}(\boldsymbol{x})) J_T^T \odot (\boldsymbol{j}_c \boldsymbol{j}_c^T), \tag{2.7}$$

where $\boldsymbol{j}_c := \begin{bmatrix}\frac{1}{\Delta x} & \frac{1}{\Delta y}\end{bmatrix}^T$, $J_c := \operatorname{diag}(\boldsymbol{j}_c)^{-1}$ is the diagonal anisotropy resulting from the aspect ratio of the discretization of the computational domain, and $\odot$ is the entrywise Hadamard product.

Remark. *If the Jacobian of the map T is poorly conditioned, the overall conditioning of $\mathcal{D}_{\mathit{eff}}$ can potentially be improved by changing the grid aspect ratio, as the element size will scale the rows and columns accordingly.*

2.1. Interpolation. The diffeomorphism T allows one to transfer continuous functions between the physical and computational domains, though a discrete analog is now needed to transfer solution vectors to numerically solve the PDE. Let N_c and N_p denote the number of vertices in the computational and physical meshes, respectively. To transfer discrete solution vectors, we will define an interpolation operator $\boldsymbol{P} \in \mathbb{R}^{N_c \times N_p}$, analogously to coarse-to-fine interpolation in multigrid. We then define the entries of $\boldsymbol{P}$ by mapping the vertices of the physical mesh, $\boldsymbol{x}_i^p$, to the computational domain using T, then computing interpolation coefficients by evaluating local bilinear basis functions, (2.6).

For example, if the image of $T(\boldsymbol{x}_p^i)$ lies within element τ_j on the computational domain, column $\boldsymbol{P}_i$ would have nonzero entries corresponding to the vertices of τ_j, the values of which would be given by evaluating bilinear basis functions $\boldsymbol{\phi}$ on $T(\boldsymbol{x}_p^i)$, see Figure 2.1 for a visual diagram. We then form restriction as the transpose of prolongation, $\boldsymbol{R} = \boldsymbol{P}^T$, as is standard for a symmetric problem. Note that even if $N_c = N_p$, surjectivity onto the computational degrees of freedom is not guaranteed.

Remark. *The type of basis function on the physical discretization does not need to be the same as on the computational system. For example, the physical system can be discretized using P1 triangles and the computational domain using Q1 quads. Moreover, while it is possible to use mismatching polynomial degrees, for example, P2 triangles on the physical domain and Q1 on the computational, we have not tested this and leave careful analysis of this to a future work.*

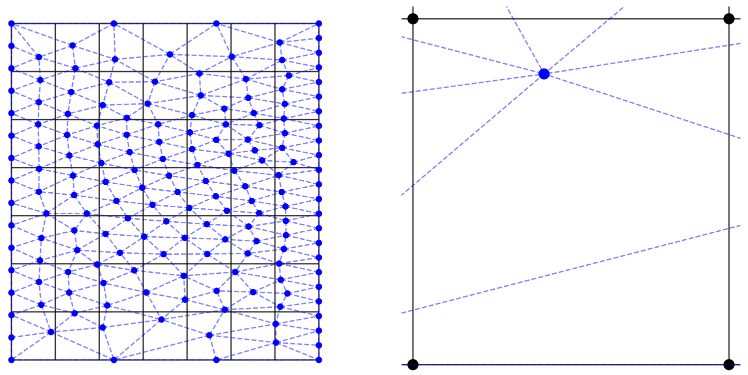

FIG. 2.1. *A diagram of the interpolation process. On the left, vertices of the physical mesh are remapped to the computational domain via T, this transformed mesh is displayed in blue with dashed lines. On the right is a zoomed view of computational element $(4, 1)$: a physical degree of freedom is interpolated to the enclosing computational element by evaluating its mapped position with bilinear basis functions to give interpolation coefficients.*

2.2. Multigrid Hierarchy. To construct a full multigrid hierarchy to solve (2.1), we will assume we have access to the following:

1. A linear system for the physical PDE, $\boldsymbol{A}_p\boldsymbol{u}_p = \boldsymbol{f}_p$, whether that be through an explicit matrix or some method to compute matrix-vector products.
2. A stationary solver S to cheaply relax residuals defined on $\boldsymbol{A}_p$.
3. A computational linear system, $\boldsymbol{A}_c$, as from section 2.
4. An interpolation operator, $\boldsymbol{P}$, as from subsection 2.1.

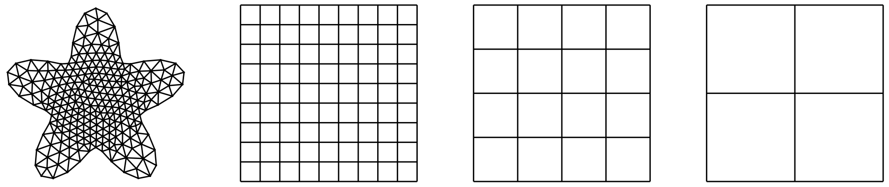

FIG. 2.2. *An example multigrid hierarchy. The finest level is the original physical system (left-most mesh), followed by the hierarchy of structured grids (right three meshes).*

We solve the computational system, $\boldsymbol{A}_c$, using black box multigrid implemented in the Cedar software package [4, 5, 11], though any geometric multigrid solver can be used. This applies a standard coarsening by two in each dimension until a computational hierarchy is formed. The full hierarchy is formed by prepending the original physical system to the computational hierarchy, see Figure 2.2. Discrete solutions are transferred between the physical and computational systems using $\boldsymbol{P}$. We sketch out the solver algorithm in the following subsections.

2.2.1. Setup phase. Inputs: $\boldsymbol{A}_p$, relaxation S, physical mesh $\mathcal{T}_p$, diffeomorphic mapping T, weak form of PDE.

1. Discretize $\boldsymbol{A}_c$ by mapping weak form using T (2).

2. Compute interpolation $\boldsymbol{P}$ by mapping vertices of $\mathcal{T}_p$ via T (2.1).
3. Setup solver on $\boldsymbol{A}_c$ using geometric multigrid.

2.2.2. Solve phase. Inputs: $\boldsymbol{A}_p$, relaxation S, $\boldsymbol{A}_c$, right-hand-side $\boldsymbol{f}_p$, approximate solution $\boldsymbol{u}_p$.

1. Pre-relaxation: $\boldsymbol{u}_p \leftarrow \omega(\boldsymbol{f}_p - \boldsymbol{A}_p\boldsymbol{u}_p)$.
2. Restrict residual: $\boldsymbol{r}_c \leftarrow \boldsymbol{R}(\boldsymbol{f}_p - \boldsymbol{A}_p\boldsymbol{u}_p)$.
3. Computational solve with geometric multigrid: $\boldsymbol{u}_c = \boldsymbol{A}_c^{-1}\boldsymbol{r}_c$.
4. Prolongate solution: $\boldsymbol{u}_p \leftarrow \boldsymbol{P}\boldsymbol{u}_c$.
5. Post-relaxation: $\boldsymbol{u}_p \leftarrow \omega(\boldsymbol{f}_p - \boldsymbol{A}_p\boldsymbol{u}_p)$.

3. Learning the mapping.

Thus far, we have developed a solver framework for problems in which an analytic mapping T between the physical and computational domains is known and can be defined a priori. However, for more complicated geometry this is usually not the case. We will show how an approximate mapping $T_\theta : \Omega_p \to \Omega_c$, parameterized by some values θ, can be learned.

In the machine learning sphere, Neural Ordinary Differential Equations [3] (Neural ODEs, or NODEs), are a continuous generalization of feedforward networks wherein the layer update is given by some continuous function $f(\boldsymbol{x}, t) : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$; evaluation of the neural network is then computed by integrating f in time. In this context, we use Neural ODEs as they guarantee a function that is smooth and invertible – sufficient for use as a diffeomorphic mapping.

Formally, we have

$$T_\theta(\boldsymbol{x}) = \boldsymbol{z}(1) = \int_0^1 \boldsymbol{f}_\theta(\boldsymbol{z}(t), t)\, dt, \tag{3.1}$$

where $\boldsymbol{z}(0) = \boldsymbol{x}$ and $\boldsymbol{f}_\theta(\boldsymbol{x}, t)$ is a learned vector field parameterized by θ. Derivatives of the network evaluation, for both backpropagation and finite element assembly, can be efficiently computed by integration of an adjoint equation [10, 3]. The inverse of the function can be computed simply by integrating backwards in time.

To "train" T_θ to approximate the mapping between our physical and computational domains, we will create a loss function inspired by large deformation diffeomorphic metric mappings (LDDMM) [8]. The LDDMM suite of algorithms is used in applications such as medical imagery to map and transfer images between different coordinate bases; here, we will use it to map continuous functions between domains.

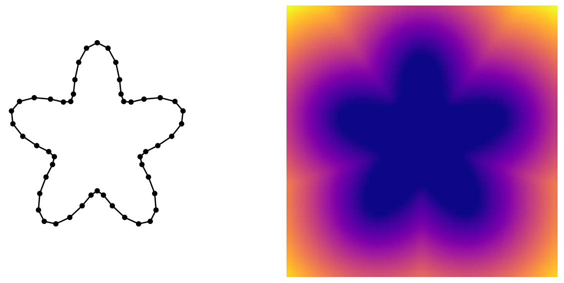

Fig. 3.1. *An example star-shaped domain (left) with corresponding unsigned distance function (right). Even if the underlying geometry is non-convex it is possible to construct a distance function to represent it.*

Let U_Ω be an unsigned distance function defined like

$$U_\Omega(\boldsymbol{x}) = \begin{cases} \|\boldsymbol{x} - P_{\partial\Omega}(\boldsymbol{x})\|_2 & \boldsymbol{x} \notin \Omega \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

where $P_{\partial\Omega}(\boldsymbol{x}) = \arg\min_{\boldsymbol{y}\in\partial\Omega} \|\boldsymbol{y} - \boldsymbol{x}\|_2^2$ is the projector onto the boundary of Ω. We define the *images* of the physical and computational domains simply by

$$I_p(\boldsymbol{x}) = U_{\Omega_p}(\boldsymbol{x}), \tag{3.3}$$
$$I_c(\boldsymbol{x}) = U_{\Omega_c}(\boldsymbol{x}). \tag{3.4}$$

These can be interpreted as encoding the interior volume and boundary of the underlying geometry, as shown in Figure 3.1. Note that both of these functions are defined over all $\mathbb{R}^d$, not simply in their respective domains. Following the LDDMM framework, we can then define the loss by

$$\ell(\theta) = \|I_p \circ T^{-1} - I_c\|_{L^2}^2, \tag{3.5}$$

which attempts to match the image of the transformed physical domain to that of the computational domain. The integral in (3.5) is then approximated by a Monte Carlo scheme of the form

$$\ell(\theta) \approx \frac{1}{N_s} \sum_{i=1}^{N_s} (I_p(T^{-1}(\boldsymbol{x}_i)) - I_c(\boldsymbol{x}_i))^2, \tag{3.6}$$

where the points $\{\boldsymbol{x}_i\}$ are drawn uniformly from $\mathcal{U}(-1.5, 1.5)^2$, a region extending slightly past the computational domain. We choose this loss over, e.g., a pointwise loss such as $\sum_i \|T_\theta(\boldsymbol{x}_i) - \boldsymbol{x}_i\|$, as points on one domain are automatically moved to their closest equivalent position in the other; there is no need to explicitly identify nearest neighbors nor is there any explicit need to treat domain boundaries.

An interesting consequence of the fact that this learning is performed on domains and not meshes, is that the mapping needs only to be computed once if the underlying domain remains the same. For example, if adaptive mesh refinement or a moving mesh method is used, then T_θ remains a valid mapping as the domain itself does not change. This also allows for the training cost to be amortized for, e.g., time-stepped simulations.

4. Numerical Results.

To evaluate the proposed framework, we have implemented a solver routine in MFEM [1]; the physical operator is discretized using $P1$ triangular finite elements while the structured computational domain is discretized with $Q1$ quadrilateral elements. For our diffeomorphic geometric solver, we use the existing Gauss-Seidel implementation for relaxation on the fine grid and Cedar [11] and BoxMG [4, 5] for the geometric multigrid solve on the computational domain. This is compared against HYPRE's BoomerAMG [9, 7] implementation to perform a solve on the physical operator outright. The tests were run on an ARM-based M1 pro CPU running at 3.2GHz. Both cases were run on a single thread/process, though our solver can easily be modified to run on parallel machines as it is using mainly off-the-shelf solver components besides the discretization code.

We tested performance on domains where the diffeomorphic mapping is analytically known: a unit square (46 686 DOF) and a quarter annulus (110 083 DOF, see Figure 4.1) – both were generated with Gmsh [6] to create unstructured meshes. The

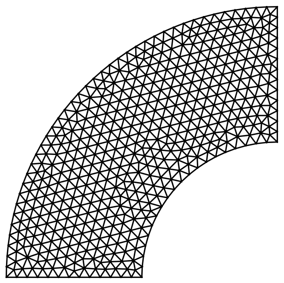

FIG. 4.1. *A low-poly version of the quarter annulus domain, extending from $\pi/2$ to π radians.*

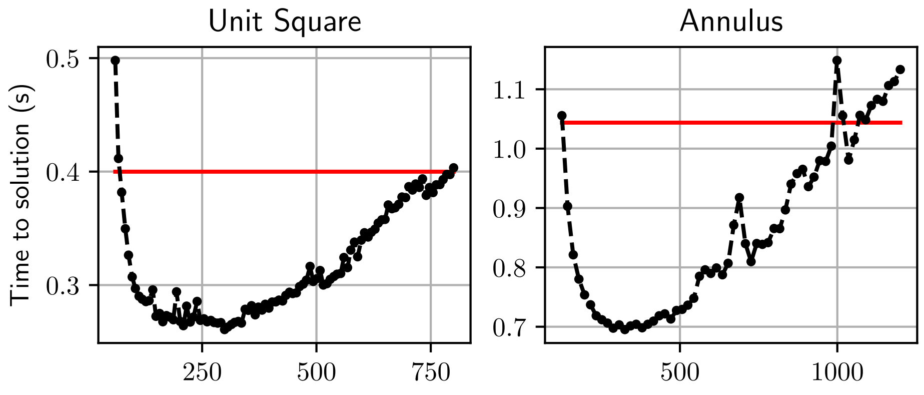

FIG. 4.2. *Time-to-solution for the diffeomorphic GMG method as a function of computational grid size (in one dimension). This is compared to BoomerAMG in HYPRE, displayed as the horizontal red line in each plot.*

first domain uses an identity mapping to the computational domain and serves as a baseline sanity check. The second has a mapping defined by a composition of trigonometric functions. A random right-hand-side $\boldsymbol{b}$ and initial solution $\boldsymbol{x}_0$ was generated for each problem and overall wall clock time was recorded for the residual norm to be sufficiently minimized to below 10^{-10}. The diffeomorphic solver was tested for various sizes of (square) computational domains; this is shown in Figure 4.2. Intuitively, reaching an optimal time-to-solution requires a compromise of computational mesh fidelity and total work performed: too small of a computational mesh will fail to sufficiently capture medium- and long-range phenomena, while too large of a mesh will incur extra work at no benefit. For the unit square, the overall time to solution was 0.40 seconds with BoomerAMG and 0.26 seconds with the geometric solver. For the quarter annulus, the time to solution was 1.04 seconds with BoomerAMG and 0.70 seconds with the geometric solver. In both cases, a roughly 33% speedup was achieved by using the diffeomorphic geometric solver.

For the timing results above, the aspect ratio of the computational mesh was keep square, meaning that each dimension had the same number of elements. For domains that have roughly the square aspect ratio, this seems like a natural choice; however, one might expect for the quarter annulus that perhaps a different choice of

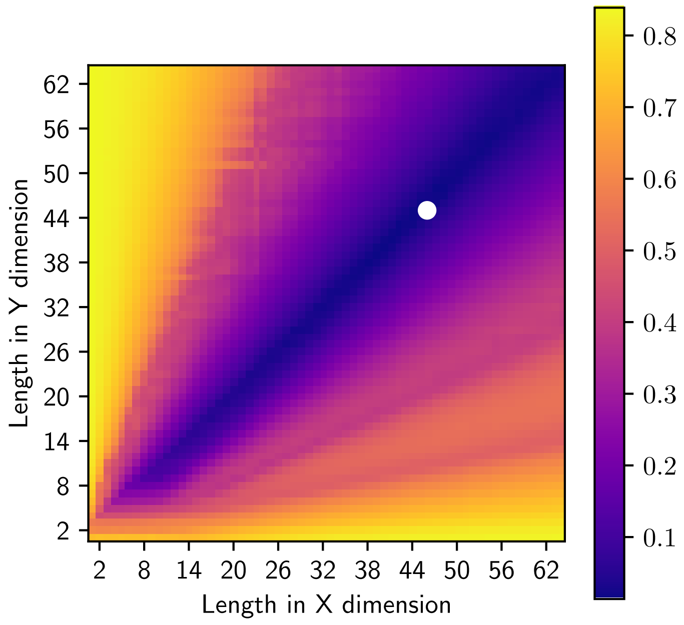

FIG. 4.3. *The effect of computational mesh aspect ratio on solver convergence on the annulus domain, for a small problem of $N = 265$ DOF. Optimal convergence (lower is better) is achieved by an equal aspect ratio, denoted by the grey circle at* $(45, 45)$.

aspect ratio would result in faster convergence. A sweep of possible grid sizes was performed, the results of which are displayed in Figure 4.3. Optimal convergence for the quarter annulus is in fact achieved when a square aspect ratio is used. Different domains, however, would likely need further analysis on the computational mesh size and aspect ratio needed for convergence.

Results for learned diffeomorphic mappings are currently preliminary; the learning framework from section 3, was tested to learn the diffeomorphic mappings for a restricted channel (a box with cuts on top and bottom) and a rounded star domain. To visually identify the performance of the mappings, vertices from one mesh were mapped onto the other; i.e., computational mesh was mapped onto the physical and vice versa. Vertices should line up in their respective domains and not, for example, depart from the boundary – indicating that the two domains are mapped well. Another consideration is that points should not be irregularly distributed in any part of the domain, as this may present eventual difficulties in discretizing the underlying PDE and remapping discrete solutions in that part of the domain.

The forward and inverse mappings of the restricted channel geometry can be seen in Figure 4.4. Vertices are mapped (more-or-less) uniformly from one domain to another and except for small areas near the edges of the cutout, the domain boundary is mapped well. For the rounded star, Figure 4.5, the learned mapping is much looser in terms of both satisfying mapped point uniformity and boundaries of the domains. For example, there is a large clustering of points in the middle of the star, while mapped points are much sparser towards the points. It is also visually apparent that the method has some difficulty in matching, e.g., corners with smoothed edges.

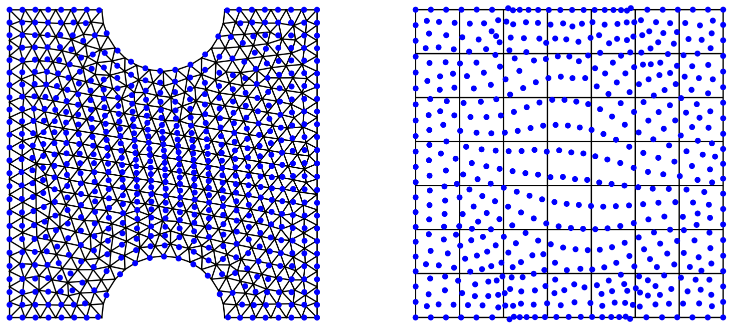

Fig. 4.4. *Vertices of the unit square mesh mapped onto the restricted channel domain (left), and vertices of the restricted channel mesh mapped onto the unit square (right).*

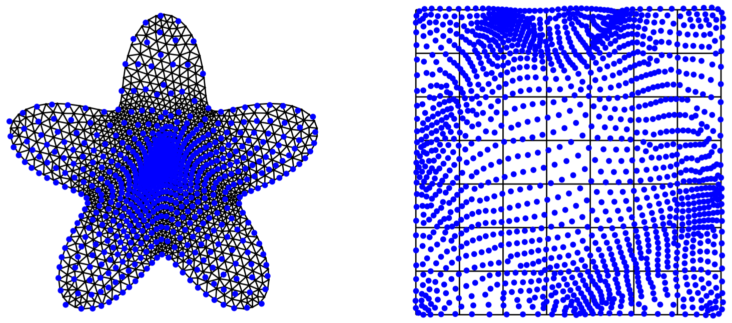

Fig. 4.5. *Vertices of the unit square mesh mapped onto the rounded star domain (left), and vertices of the rounded star mesh mapped onto the unit square (right). A large distribution of points can be seen clustered in the middle of the star on the left, perhaps indicating potential Jacobian irregularity.*

5. Conclusions.

We have presented a framework of enabling the use of geometric multigrid solvers on more complex geometries via the use of diffeomorphic mappings to transfer computations to structured domains. This not only preserves the efficiency and overall scalability of geometric multigrid methods but also allows their use on problems that are more traditionally solved by algebraic methods. For domains with analytic mappings, our method achieved a 33% speedup over BoomerAMG on the same problem, underscoring its practicality. Moreover, for geometries where analytic mappings are unavailable, we propose a methodology for learning said mapping using invertible neural networks in the form of neural ODEs. While results for such mappings are still in development, preliminary findings suggest promise to generalize this framework to irregular, complex domains.

Possible future directions can further focus and validate the use of learning mappings in this solver framework, perhaps providing rigorous convergence guarantees under which a learned mapping sufficient under some metric can guarantee a geometric solver that will be convergent. Another interesting direction would be to consider applying this framework in a domain decomposition setting: geometric multigrid is already used in practice for semi-structured meshes, and allowing more flexibility in the types of (sub)-domains would allow such solvers to be much more competitive with existing algebraic methods in terms of the geometry that it can be applied to.

REFERENCES

[1] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, *MFEM: A modular finite element methods library*, Computers & Mathematics with Applications, 81 (2021), pp. 42–74, https://doi.org/10.1016/j.camwa.2020.06.009.

[2] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM Journal on Scientific Computing, 34 (2012), https://doi.org/10.1137/110838844.

[3] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural ordinary differential equations*, NIPs, 109 (2018), pp. 31–60, https://doi.org/10.48550/arxiv.1806.07366, https://arxiv.org/abs/1806.07366v5.

[4] J. Dendy, *Black box multigrid*, Journal of Computational Physics, 48 (1982), pp. 366–386, https://doi.org/10.1016/0021-9991(82)90057-2, https://linkinghub.elsevier.com/retrieve/pii/0021999182900572.

[5] J. E. Dendy and J. D. Moulton, *Black box multigrid with coarsening by a factor of three*, Numerical Linear Algebra with Applications, 17 (2010), pp. 577–598, https://doi.org/10.1002/nla.705, https://onlinelibrary.wiley.com/doi/10.1002/nla.705.

[6] C. Geuzaine and J. Remacle, *Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities*, International Journal for Numerical Methods in Engineering, 79 (2009), p. 1309–1331, https://doi.org/10.1002/nme.2579, http://dx.doi.org/10.1002/nme.2579.

[7] V. E. Henson and U. M. Yang, *Boomeramg: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), p. 155–177, https://doi.org/10.1016/s0168-9274(01)00115-5, http://dx.doi.org/10.1016/S0168-9274(01)00115-5.

[8] M. Hernandez and U. R. Julvez, *Insights into traditional large deformation diffeomorphic metric mapping and unsupervised deep-learning for diffeomorphic registration and their evaluation*, Computers in Biology and Medicine, 178 (2024), https://doi.org/10.1016/j.compbiomed.2024.108761.

[9] *hypre: High performance preconditioners*. https://llnl.gov/casc/hypre, https://github.com/hypre-space/hypre.

[10] L. S. Pontryagin, *Mathematical theory of optimal processes: Mathematical theory of optimal processes L.s. pontryagin selected works volume 4*, Classics of Soviet Mathematics, Harwood Academic, Amsterdam, Netherlands, Mar. 1987.

[11] A. Reisner, L. N. Olson, and J. D. Moulton, *Scaling structured multigrid to 500k+ cores through coarse-grid redistribution*, (2018).

[12] J. W. Ruge and K. Stüben, *4. Algebraic Multigrid*, Society for Industrial and Applied Mathematics, Jan. 1987, p. 73–130, https://doi.org/10.1137/1.9781611971057.ch4, http://dx.doi.org/10.1137/1.9781611971057.ch4.

[13] G. Shishkin, *A difference scheme on a non-uniform mesh for a differential equation with a small parameter in the highest derivative*, USSR Computational Mathematics and Mathematical Physics, 23 (1983), p. 59–66, https://doi.org/10.1016/s0041-5553(83)80102-5, http://dx.doi.org/10.1016/S0041-5553(83)80102-5.