Schwarz–Schur Involution: Lightspeed Differentiable Sparse Linear Solvers

Yu Wang [1 2 3] **S. Mazdak Abulnaga** [1 2] **Yaël Balbastre** [1 4] **Bruce Fischl** [1 2]

Abstract

Sparse linear solvers are fundamental to science and engineering, applied in partial differential equations (PDEs), scientific computing, computer vision, and beyond. Indirect solvers possess characteristics that make them undesirable as stable differentiable modules; existing direct solvers, though reliable, are too expensive to be adopted in neural architectures. We substantially accelerate *direct* sparse solvers or generalized deconvolution by up to 3 orders-of-magnitude faster, violating common assumptions that direct solvers are too slow. We "condense" a sparse Laplacian matrix into a dense tensor, a compact data structure that batch-wise stores the Dirichlet-to-Neumann matrices, reducing the sparse solving to recursively merging pairs of dense matrices that are much smaller. The batched small dense systems are sliced and inverted in parallel to take advantage of dense GPU BLAS kernels, highly optimized in the era of deep learning. Our method is efficient, qualified as a strong zero-shot baseline for AI-based PDE solving and a reliable differentiable module integrable into machine learning pipelines. Our code is available at https://github.com/wangyu9/Schwarz_Schur_Involution.

1. Introduction

In this paper, we propose a conceptually simple approach that reduces solving certain sparse linear system $(\mathbf{A}, \mathbf{b})$ to "involuting" tensors $(\boldsymbol{\alpha}, \boldsymbol{\beta})$, leading to significant improvements. Global linear systems in computer vision and scientific computing, such as the Laplacian or Hessian systems on image domains, are usually very large yet sparse, as induced by pixel affinity. For a sparse $\mathbf{A}$, *direct* solvers find the *exact* solution: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, for which we present the first interactive rate algorithm on common-size images. Solving the linear system $\mathbf{A}^{-1}\mathbf{b}$ is equivalent to the generalized deconvolution of an image $\mathbf{b}$ with a spatially varying kernel (in rows of) $\mathbf{A}$. Sparse linear solvers underpin modern image processing, computer vision, graphics, as well as numerical methods for PDEs such as finite difference/element methods (FD/FEM) used in electrical/mechanical engineering and computational sciences. Users in these areas can also benefit from our approach if efficiency is a primary concern. Our direct solver—even naïvely prototyped in PyTorch—is 60 to 1000× faster than SciPy and 40 to 170× faster than CUDA (cuDSS), which are highly optimized (Table 1).

Table 1. Our substantially faster sparse solver (including the factorization and substitution stages) is the first direct method to run at interactive rates. It takes our method 10.9 ms (resp. 220 ms) to solve a Laplacian system (Dirichlet) on an image of 513×513 (resp. 2561^2). Runtime is reported in milliseconds.

Example	CUDA	SciPy	ours	speedup
2561^2	36926	253318	220.1	**168X 1151X**
2049^2	21354	143100	158.5	**135X 903X**
1025^2	4710	16512	36.45	**129X 453X**
513^2	1036	2051	10.90	**95.0X 188X**
257^2	234	355	5.82	**40.2X 61.0X**

Successes of modern deep learning are largely built on *differentiable numerical linear algebras*, particularly matrix multiplications of dense—such as attentions (Vaswani et al., 2017)—and sparse matrices—notably convolutions (LeCun et al., 1989). High performance AI systems rely on efficient implementation of these operations, including dense (Strassen, 1969; Blahut, 2010; Dao et al., 2022; Dao, 2023) and sparse multiplications (Winograd, 1980; Cook, 1966; Lavin & Gray, 2016). In contrast, matrix inversions are almost never employed in neural architectures. We believe that inversions are underexplored and can play a prominent role, since (sparse) matrix inversions encompass a large range of operations such as generalized deconvolution, geometry representation, results of physical simulations, decorrelation, and equilibrium states of localized interactions.

As detailed in §F.2, a key obstacle preventing neural architectures from adopting linear solvers is the lack of SCHWARZ—a Sparse solver like ours that is Consistent-performance, Hyperspeed, in-the-Wild, Accurate, Robust, and Zero-parameter. Indirect, a.k.a. iterative, solvers fall short of these goals due to their strong problem dependency,

[1]Athinoula A. Martinos Center, Massachusetts General Hospital, Harvard Medical School [2]Massachusetts Institute of Technology, MIT CSAIL, USA [3]Sony, USA [4]University College London, UK. Correspondence to: <{yw823,bfischl}@mgh.harvard.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

parameter sensitivity, and unpredictable runtime. The large diversity of PDEs results in laborious workflows requiring expert users in the loop to tweak parameters or preconditioners to employ indirect solvers. Iterative solvers struggle to deconvolute indefinite kernels, such as the Helmholtz PDE (Ernst & Gander, 2011). While existing direct solvers are promising candidates for meeting the desiderata, they are too slow—a limitation that our method overcomes. Catalyzed by the surge of interest in physics-informed machine learning and AI for PDE/Science (Berens et al., 2023), identification or learning of unknown physical systems open up a unique setting to solve **matrix $\mathbf{A}$ whose properties are unknown** in advance to apply appropriate iterative schemes, necessitating a general-purpose solver that is deployable **for *any* $\mathbf{A}$** and/or a large varieties of matrices from some training batch. This is similar to the generalized deconvolution with unknown kernels in computer vision. Applying unsuitable indirect solvers can yield catastrophic failure.

Our improvement is due to "condensing"—transfer GPU's capacity of dense BLAS (Dongarra et al., 1988; 2018) to sparse computation, through design choices with a compact data structure—a tensor storing Dirichlet-to-Neumann or sub-systems in batches, and a procedure we coin as "Schwarz-Schur involution"—recursively applying the Schur complement formula to block-wisely contract nodes and keep track of sparsity explicitly. Our approach amounts to a parallel implementation of Gaussian elimination under nested dissections (George, 1973) and the induced multifrontal solvers (Duff & Reid, 1983), recursively cancels variables. We take advantage of the regularity of image grids to aggressively explore parallelisms, leveraging advances in GPUs sparked by deep learning that shift the best practice towards algorithms better exploiting parallelisms.

Our sparse solvers, with speeds exceeding iterative solvers, retain the accuracy and reliability of direct solvers, efficient enough to be integrated within neural architectures. Owing to the central role of linear solvers, our method enables efficient implementations including but not limited to:

- generalized deconvolution of spatially varying kernels.
- identify and reduce gaps between AI & conventional methods: zero-shot baselines for learning-based PDE solvers.
- exact Newton solvers made tractable on image domains.
- mathematical optimization layers embedded in neural nets (Amos & Kolter, 2017), (physics) solver-in-the-loop.
- geometric deep learning & algorithms (Litany et al., 2017), shape/deformation representation, graph/mesh/FEM neural networks (Wang et al., 2019; Pfaff et al., 2020).
- eigenbases for spectral neural nets (Bruna et al., 2013), spectral clustering (Shi & Malik, 2000) made interactive.

1.1. Related Work

We refer readers to standard texts on direct solvers of dense (Davis, 2006) and sparse systems (Duff et al., 2017) for extensive surveys. Direct solvers first perform the numerical factorization—Cholesky or LDLT factorization for symmetric $\mathbf{A}$ and LU factorization for asymmetric systems—followed by a back substitution to spread out $\mathbf{b}$ to yield $\mathbf{x}$. Major backends for direct sparse solvers include SuperLU (Li, 2005), UMFPACK (Davis, 2004), CUDA (Nickolls et al., 2008), Pardiso (Schenk & Gärtner, 2004), Eigen (Guennebaud et al., 2014), and Cholmod/-SuiteSparse (Chen et al., 2008). When $\mathbf{A}$ comes from a spatially constant kernel or point spread function (PSF), the problem becomes an image deconvolution, with efficient frequency domain solvers (Sezan & Tekalp, 1990; Hansen et al., 2006); spectral solvers like FFT/IFFT (Fast Fourier Transform) (Frigo & Johnson, 2005) are limited to solving homogeneous Laplace systems. Instead, we attack PDEs or deconvolution with spatially varying coefficients/kernels.

We consider solving indefinite and nonsymmetric square linear systems (Trefethen & Bau, 1997), and thus methods requiring symmetric or positive-definite $\mathbf{A}$ do not apply, such as CG (Conjugate Gradient) (Hestenes et al., 1952), MINRES (Paige & Saunders, 1975; Liu & Roosta, 2022). Methods apply to our setting include: CGS (Conjugate Gradient Squared) (Sonneveld, 1989) and the improved variant biCGstab (Bi-Conjugate Gradient Stabilized) (Van der Vorst, 1992; Sleijpen & Fokkema, 1993), iterative sparse least-squares LSQR (Paige & Saunders, 1982b;a) that amounts to CG applied to $(\mathbf{A}^\intercal\mathbf{A})^{-1}\mathbf{A}^\intercal\mathbf{b}$ and a recent improvement LSMR (Fong & Saunders, 2011), GMRES (generalized minimal residual) (Saad & Schultz, 1986) and improvements DQGMRES (Saad & Wu, 1996) and FGMRES (Saad, 1993). Other methods include BILQ (Montoison & Orban, 2020; Fletcher, 1976), QMR (Freund & Nachtigal, 1991; 1994), FOM (Saad, 1981), and DIOM (Saad, 1984). The multigrid methods (Bramble, 1993) apply a hierarchical coarse-to-fine strategy, effective as iterative solvers or the preconditioners in CG to yield PCG, especially for elliptic PDEs. We defer discussions on solvers in vision & graphics (Barron & Poole, 2016; Krishnan et al., 2013; Bolz et al., 2003; Jeschke et al., 2009; Horvath & Geiger, 2009; Liu et al., 2016) to §A.

2. Mathematical Preliminaries

Sparse linear solver = exact generalized deconvolution. In the paper, for an $H \times W$ image with n pixels, $n := HW$, whose boundary has $b := 2W+2H-4$ pixels, we consider a sparse matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ that encodes the affinity of pixels: each row/column of $\mathbf{A}$ corresponds to one pixel and $\mathbf{A}_{ij}$ is nonzero only if pixels $i, j \in \{1, ..., n\}$ are adjacent in the image, namely their integer coordinates $(x_i, y_i), (x_j, y_j) \in \mathbb{N}^2$ satisfying that $(x_i-x_j)^2+(y_i-y_j)^2 \leq 2$. For a function $u(x, y)$ on the image grid, consider the linear operator or a 3×3 convolution centered at (x, y) with $a^{(x,y)} \in \mathbb{R}^{3\times3}$:

$$v(x, y) \leftarrow \sum\sum\nolimits_{\delta_x,\delta_y\in\{-1,0,1\}} a^{(x,y)}(\delta_x, \delta_y)u(x + \delta_x, y + \delta_y)$$

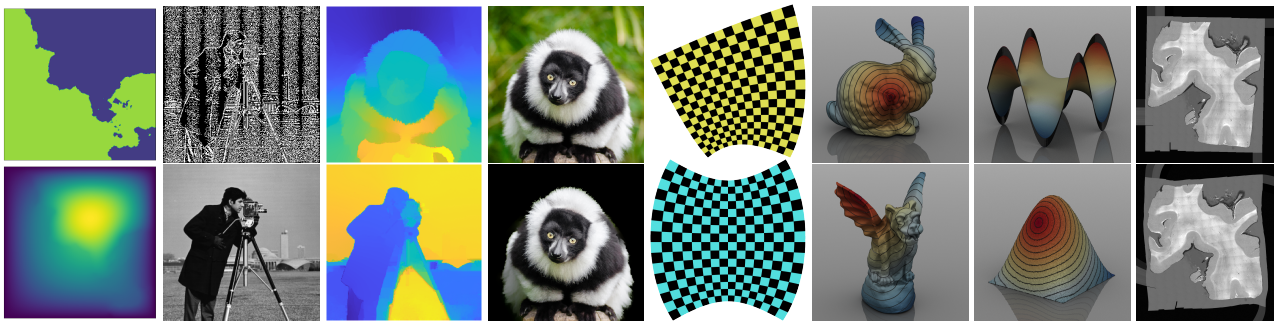


Figure 1. Our method immediately accelerates direct solvers in a large variety of tasks by orders of magnitude, including: FEM for solving PDEs, deconvolution (generalized to spatially varying kernels), eigen solver and spectral method, image segmentation and matting, physical simulation, deformation, geometry processing, shape optimization, Newton's method, and diffeomorphic image registration.

using a 9-point stencil $a^{(x,y)}$—a spatially varying kernel:

$$a^{(x,y)} := \begin{bmatrix} a^{(x,y)}(-1,-1) & a^{(x,y)}(-1,0) & a^{(x,y)}(-1,1) \\ a^{(x,y)}(0,-1) & a^{(x,y)}(0,0) & a^{(x,y)}(0,1) \\ a^{(x,y)}(1,-1) & a^{(x,y)}(1,0) & a^{(x,y)}(1,1) \end{bmatrix}. \quad (1)$$

By flattening the 2-dim array u, v into the vector $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{n\times 1}$ (see Figure 24), for $\mathbf{A}_{ij} = a^{(x_j,y_j)}(x_i - x_j, y_i - y_j)$, the convolution with $a^{(x,y)}$ can be written as the matrix-vector product $\mathbf{v} \leftarrow \mathbf{A}\mathbf{u}$. Thus, the linear solve $\mathbf{A}^{-1}\mathbf{v}$ *generalizes deconvolution* to the case with a spatially varying kernel $a^{(x,y)}$. Theoretically, our approach generalizes to larger stencils and 3D, which we leave for future work.

Differentiable and repetitive linear solvers. In the learning setting, the system $\mathbf{A}, \mathbf{b}$ come from some learnable parameters $\boldsymbol{\theta}$ to be identified with gradient descent. Thus, in addition to solving $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, we need to obtain $\partial \mathbf{x}/\partial \mathbf{b}$ and $\partial \mathbf{x}/\partial \mathbf{A}$, which have closed-form expressions that can be derived using the adjoint method; evaluating these gradients requires solving the transposed system $\mathbf{A}^\intercal$, see §C.3.

A common setting is to sequentially solve multiple pairs $(\mathbf{A}, \mathbf{b}_1)$, $(\mathbf{A}, \mathbf{b}_2)$,..., $(\mathbf{A}, \mathbf{b}_k)$, i.e., the same left-hand side $\mathbf{A}$ but different right-hand sides. Direct solvers, including ours, stand out for allowing **separating numerical factorization from back substitution and reuse** the former. Numerical factorization is the computation involving $\mathbf{A}$ only, which is the *dominant cost that can be reused* for direct solvers. This is not possible for indirect solvers, which have to start over for each right-hand side. The typical value of k is $2 \sim 20+$: 2 in differentiating linear solvers, or $k > 20$ in, e.g., being applied in eigen solvers. Thus, when compared to iterative solvers in these settings, computationally it means that iterative solvers get an extra $\times k$ slow down.

PDE and FEM preliminaries. Our method applies to any invertible $\mathbf{A}$ with the aforementioned sparsity. But first let us examine the $\mathbf{A}$ that arises from discretizing elliptic PDEs (Gilbarg et al., 1977), a situation that motivates our initial development. Surprisingly, the strategy generalizes beyond elliptic PDEs without adaptations to other linear systems in practice, even non positive-semidefinite (PSD). Let us consider solving the Laplace equation defined using the anisotropic diffusion coefficient $\mathbf{C}(\mathbf{x}) \in \mathbb{R}^{2\times 2}, \forall \mathbf{x} \in \Omega$ on the domain Ω, subject to either the Dirichlet condition (2) or the Neumann condition (3) at the boundary $\partial\Omega$:

$$-\nabla \cdot [\mathbf{C}(\mathbf{x})\nabla u(\mathbf{x})] = f(\mathbf{x}), \quad u(\mathbf{x})|_{\partial\Omega} = g(\mathbf{x}). \quad (2)$$

$$-\nabla \cdot [\mathbf{C}(\mathbf{x})\nabla u(\mathbf{x})] = f(\mathbf{x}), \quad \mathbf{n}^\intercal(\mathbf{x})\mathbf{C}(\mathbf{x})\nabla u(\mathbf{x})|_{\partial\Omega} = h(\mathbf{x}). \quad (3)$$

Discretizing with a first-order piecewise linear FEM yields:

$$\mathbf{L} = \mathbf{G}^\intercal\mathbf{C}\mathbf{G}, \quad [\mathbf{L}\mathbf{u}]|_{b+1:n} = \mathbf{f}, \quad \mathbf{u}|_{1:b} = \mathbf{g} \text{ or } [\mathbf{L}\mathbf{u}]|_{1:b} = \mathbf{h},$$

following the notation and discretization (Wang et al., 2023), in which $\mathbf{u} \in \mathbb{R}^n, \mathbf{f} \in \mathbb{R}^{n-b}, \mathbf{g}, \mathbf{h} \in \mathbb{R}^b$ discretize the solution u and right-hand side functions f, g, h; $\mathbf{n}(\mathbf{x})$ is the normal direction at the boundary point $\mathbf{x}$; $\cdot|_{1:b}$ or $\cdot|_{b+1:n}$ selects rows for boundary or interior pixels, resp.; the matrices $\mathbf{G}$, $\mathbf{C}$ discretize the gradient operator ∇ and $\mathbf{C}(\mathbf{x})$. Under Dirichlet boundary condition $g(\mathbf{x})$, the solution to (2) $u(\mathbf{x})$ is unique, which determines the Neumann boundary condition $h(\mathbf{x})$, inducing the *Dirichlet-to-Neumann* (DtN) map: $g(\partial\Omega) \to h(\partial\Omega)$. After discretization, the DtN map becomes the Schur complement of the matrix $\mathbf{L}$. Thus, the *coefficient-to-solution* operator: $\mathbf{C}(\mathbf{x}) \to u(\mathbf{x})$, under FD/FEM, can be simply viewed as a linear solver.

What exactly $\mathbf{A}$ represents—which helps to motivate—does not make any difference to our method. Our method does not assume $\mathbf{A}$ is symmetric, though this is often the case. Solving Neumann boundary value problems amounts to setting $\mathbf{A} = \mathbf{L}$; solving parabolic PDEs amounts to setting $\mathbf{A} = t\mathbf{L} + \mathbf{M}$, and setting $\mathbf{A} = \mathbf{L} - \kappa^2\mathbf{M}$ corresponds to solving the Helmholtz equation that arises in wave propagation and electromagnetism, in which $\mathbf{M}$ plays a role similar to the identity matrix—$\mathbf{M}$ is the mass matrix discretized by FEM (Allaire, 2007) with the same sparsity pattern as $\mathbf{L}$.

3. Schur Involution for Parallel Elimination

We demonstrate a procedure to convert the solution of the linear system $\mathbf{x} \leftarrow (\mathbf{A}, \mathbf{b})$ to the "involution" of the tensors $\boldsymbol{\chi} \leftarrow (\boldsymbol{\alpha}, \boldsymbol{\beta})$—slicing and inverting many small dense matrices in batches to leverage modern GPU hardware.

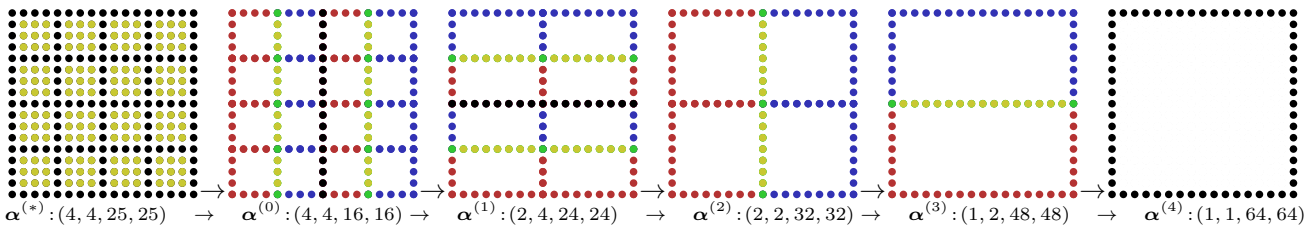


Figure 2. Schwarz–Schur involution on a 17×17 image. Our method (Algorithm 1) apply *batched linear algebras* to *parallelize* Gaussian elimination in a prescribed order—pixels marked in color correspond to nodes to be eliminated in that step. As the initialization, the Schwarz step converts the image into a wire-frame by canceling the interior pixels of each patch. Multiple Schur steps are applied to progressively simplify the wire-frame, until only pixels at the border are left. A Schur step merges every two adjacent subdomains by removing pixels on some "edges." We compact all subdomains' left- and right-hand sides in the tensors $\boldsymbol{\alpha}^{(j)}, \boldsymbol{\beta}^{(j)}$ whose shapes evolve as shown. The solutions $\boldsymbol{\chi}^{(j)}$, with the same shape as $\boldsymbol{\beta}^{(j)}$, flow reversely: $\boldsymbol{\chi}^{(*)} \leftarrow \boldsymbol{\chi}^{(0)} \leftarrow \boldsymbol{\chi}^{(1)} \ldots \leftarrow \boldsymbol{\chi}^{(3)} \leftarrow \boldsymbol{\chi}^{(4)} := (\boldsymbol{\alpha}^{(4)})^{-1}\boldsymbol{\beta}^{(4)}$.

3.1. A motivating example: sparse solvers too slow?

A 4096×4096 image can be divided into a 1024×1024 array of 4×4 patches. If adjacent patches overlap by sharing one layer of pixels, then we have a similar division: a 4097×4097 image (or 17×17 as shown in Figure 2, or 9×5 in Figure 3) can be divided into a 1024×1024 (or 4×4, or 2×1, resp.) array of small patches of size 5×5, since boundary pixels have duplicated representations. On this image, SciPy solver takes more than 20 minutes to solve a sparse system $\mathbf{A} \in \mathbb{R}^{n\times n}$, $n = 16785409$. However, inverting one million 9×9 matrices in a batch only takes 0.008 seconds using PyTorch.

```
# alpha.shape is (1024,1024,9,9)
torch.linalg.inv(alpha) #0.008 sec.
# A: sparse matrix
scipy.sparse.linalg.spsolve(A,b) #1200 sec.
```

The batched matrix inversion can be viewed as the first step of Gaussian elimination to remove the 9 pixels in the interior 3×3 block of each patch, and we already make major progress by removing approximately $9/16 \approx 56.25\%$ variables from the problem, while being $10^5\times$ faster than SciPy to solve the entire problem! This is strong evidence that the speed of sparse linear solvers has been significantly underestimated and parallelism has been under exploited.

3.2. Parallel block Gaussian elimination

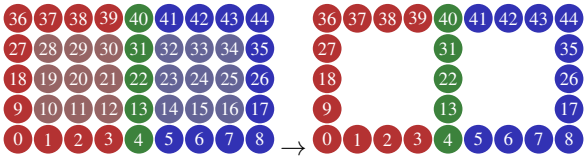


Figure 3. Schwarz step removes the interior pixels for each patch.

We introduce a procedure that we term as "Schwarz–Schur involution." For simplicity of illustration, let us first consider a simple case: with only two subdomains, the original problem (4) is first reduced to (5) by the Schwarz step, and then via (9) to (12) by a Schur step. Readers unfamiliar with numerical algebra should refer to §E.

3.2.1. Schwarz step: decompose & initialize DtN

As shown in Figure 3, let P, Q be two subdomains and divide all pixels in the image domain into five (disjoint) groups $\mathbf{r}, \mathbf{s}, \mathbf{t}, \mathbf{a}, \mathbf{b}$ that: $(\mathbf{r}, \mathbf{s})$ is the boundary of P and $(\mathbf{s}, \mathbf{t})$ is the boundary of Q—$\mathbf{s}$ is the boundary shared between P and Q, and the interior of P and Q are $\mathbf{a}$ and $\mathbf{b}$, resp. [1] The original sparse system $\mathbf{A}\mathbf{u} = \mathbf{v}$, $\mathbf{A} \in \mathbb{R}^{n\times n}$ can be written:

$$\begin{bmatrix} \mathbf{A}_{\mathbf{rr}} & \mathbf{0} & \mathbf{A}_{\mathbf{rs}} & \mathbf{A}_{\mathbf{ra}} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathbf{tt}} & \mathbf{A}_{\mathbf{ts}} & \mathbf{0} & \mathbf{A}_{\mathbf{tb}} \\ \mathbf{A}_{\mathbf{sr}} & \mathbf{A}_{\mathbf{st}} & \mathbf{A}_{\mathbf{ss}} & \mathbf{A}_{\mathbf{sa}} & \mathbf{A}_{\mathbf{sb}} \\ \mathbf{A}_{\mathbf{ar}} & \mathbf{0} & \mathbf{A}_{\mathbf{as}} & \mathbf{A}_{\mathbf{aa}} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathbf{bt}} & \mathbf{A}_{\mathbf{bs}} & \mathbf{0} & \mathbf{A}_{\mathbf{bb}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathbf{r}} \\ \mathbf{u}_{\mathbf{t}} \\ \mathbf{u}_{\mathbf{s}} \\ \mathbf{u}_{\mathbf{a}} \\ \mathbf{u}_{\mathbf{b}} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{\mathbf{r}} \\ \mathbf{v}_{\mathbf{t}} \\ \mathbf{v}_{\mathbf{s}} \\ \mathbf{v}_{\mathbf{a}} \\ \mathbf{v}_{\mathbf{b}} \end{bmatrix} \tag{4}$$

in which there are $\mathbf{0}$ blocks because the interface $\mathbf{s}$ separates the pixels into non-adjacent groups, and $\mathbf{A}_{\mathbf{ss}} = \mathbf{A}_{\mathbf{ss}}^{(P)} + \mathbf{A}_{\mathbf{ss}}^{(Q)}$, $\mathbf{v}_{\mathbf{s}} = \mathbf{v}_{\mathbf{s}}^{(P)} + \mathbf{v}_{\mathbf{s}}^{(Q)}$ can be divided into contributions from P and Q, resp. [2] Note that matrix $\mathbf{A}$ and system (4) are only for illustration purposes and never allocated as an actual matrix in our algorithm (see §E). (4) further becomes:

$$\begin{bmatrix} \mathbf{P}^{\mathbf{rr}} & \mathbf{0} & \mathbf{P}^{\mathbf{rs}} \\ \mathbf{0} & \mathbf{Q}^{\mathbf{tt}} & \mathbf{Q}^{\mathbf{ts}} \\ \mathbf{P}^{\mathbf{sr}} & \mathbf{Q}^{\mathbf{st}} & \mathbf{P}^{\mathbf{ss}} + \mathbf{Q}^{\mathbf{ss}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathbf{r}} \\ \mathbf{u}_{\mathbf{t}} \\ \mathbf{u}_{\mathbf{s}} \end{bmatrix} = \begin{bmatrix} \mathbf{p}^{\mathbf{r}} \\ \mathbf{q}^{\mathbf{t}} \\ \mathbf{p}^{\mathbf{s}} + \mathbf{q}^{\mathbf{s}} \end{bmatrix}, \tag{5}$$

by exercising Gaussian elimination to cancel $\mathbf{u}_{\mathbf{a}}, \mathbf{u}_{\mathbf{b}}$, where

$$\begin{bmatrix} \mathbf{P}^{\mathbf{rr}} & \mathbf{P}^{\mathbf{rs}} \\ \mathbf{P}^{\mathbf{sr}} & \mathbf{P}^{\mathbf{ss}} \end{bmatrix} := \begin{bmatrix} \mathbf{A}_{\mathbf{rr}} - \mathbf{A}_{\mathbf{ra}}\mathbf{A}_{\mathbf{aa}}^{-1}\mathbf{A}_{\mathbf{ar}} & \mathbf{A}_{\mathbf{rs}} - \mathbf{A}_{\mathbf{ra}}\mathbf{A}_{\mathbf{aa}}^{-1}\mathbf{A}_{\mathbf{as}} \\ \mathbf{A}_{\mathbf{sr}} - \mathbf{A}_{\mathbf{sa}}\mathbf{A}_{\mathbf{aa}}^{-1}\mathbf{A}_{\mathbf{ar}} & \mathbf{A}_{\mathbf{ss}}^{(P)} - \mathbf{A}_{\mathbf{sa}}\mathbf{A}_{\mathbf{aa}}^{-1}\mathbf{A}_{\mathbf{as}} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{Q}^{\mathbf{tt}} & \mathbf{Q}^{\mathbf{ts}} \\ \mathbf{Q}^{\mathbf{st}} & \mathbf{Q}^{\mathbf{ss}} \end{bmatrix} := \begin{bmatrix} \mathbf{A}_{\mathbf{tt}} - \mathbf{A}_{\mathbf{tb}}\mathbf{A}_{\mathbf{bb}}^{-1}\mathbf{A}_{\mathbf{bt}} & \mathbf{A}_{\mathbf{ts}} - \mathbf{A}_{\mathbf{tb}}\mathbf{A}_{\mathbf{bb}}^{-1}\mathbf{A}_{\mathbf{bs}} \\ \mathbf{A}_{\mathbf{st}} - \mathbf{A}_{\mathbf{sb}}\mathbf{A}_{\mathbf{bb}}^{-1}\mathbf{A}_{\mathbf{bt}} & \mathbf{A}_{\mathbf{ss}}^{(Q)} - \mathbf{A}_{\mathbf{sb}}\mathbf{A}_{\mathbf{bb}}^{-1}\mathbf{A}_{\mathbf{bs}} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{p}^{\mathbf{r}} \\ \mathbf{p}^{\mathbf{s}} \end{bmatrix} := \begin{bmatrix} \mathbf{v}_{\mathbf{r}} - \mathbf{A}_{\mathbf{ra}}\mathbf{A}_{\mathbf{aa}}^{-1}\mathbf{v}_{\mathbf{a}} \\ \mathbf{v}_{\mathbf{s}}^{(P)} - \mathbf{A}_{\mathbf{sa}}\mathbf{A}_{\mathbf{aa}}^{-1}\mathbf{v}_{\mathbf{a}} \end{bmatrix}, \quad \begin{bmatrix} \mathbf{q}^{\mathbf{t}} \\ \mathbf{q}^{\mathbf{s}} \end{bmatrix} := \begin{bmatrix} \mathbf{v}_{\mathbf{t}} - \mathbf{A}_{\mathbf{tb}}\mathbf{A}_{\mathbf{bb}}^{-1}\mathbf{v}_{\mathbf{b}} \\ \mathbf{v}_{\mathbf{s}}^{(Q)} - \mathbf{A}_{\mathbf{sb}}\mathbf{A}_{\mathbf{bb}}^{-1}\mathbf{v}_{\mathbf{b}} \end{bmatrix}.$$

[1] Concretely speaking, $\mathbf{s} = [4, 13, 22, 31, 40]$, $\mathbf{r} = [0, 1, 2, 3, 39, 38, 37, 36, 27, 18, 9]$, $\mathbf{a} = [10, 11, 12, 19, 20, 21, 28, 29, 30]$, $\mathbf{t} = [5, 6, 7, 8, 17, 26, 35, 44, 43, 42, 41]$, $\mathbf{b} = [14, 15, 16, 23, 24, 25, 32, 33, 34]$.

[2] The values of $\mathbf{A}_{\mathbf{ss}}^{(P)}$ and $\mathbf{A}_{\mathbf{ss}}^{(Q)}$ can be arbitrary, as long as their summation is the correct $\mathbf{A}_{\mathbf{ss}}$. See §E for details.

using the last two equations in §4 to eliminate $\mathbf{u}_\mathrm{a}, \mathbf{u}_\mathrm{b}$:

$$\begin{bmatrix}\mathbf{u}_\mathrm{a}\\ \mathbf{u}_\mathrm{b}\end{bmatrix} = \begin{bmatrix}\mathbf{A}_{\mathrm{aa}}^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_{\mathrm{bb}}^{-1}\end{bmatrix}\left(\begin{bmatrix}\mathbf{v}_\mathrm{a}\\ \mathbf{v}_\mathrm{b}\end{bmatrix} - \begin{bmatrix}\mathbf{A}_{\mathrm{ar}} & \mathbf{0} & \mathbf{A}_{\mathrm{as}}\\ \mathbf{0} & \mathbf{A}_{\mathrm{bt}} & \mathbf{A}_{\mathrm{bs}}\end{bmatrix}\begin{bmatrix}\mathbf{u}_\mathrm{r}\\ \mathbf{u}_\mathrm{t}\\ \mathbf{u}_\mathrm{s}\end{bmatrix}\right). \tag{6}$$

Let $$\mathbf{P} \overset{\mathcal{P}}{\leftarrow} \begin{bmatrix}\mathbf{P}^{\mathrm{rr}} & \mathbf{P}^{\mathrm{rs}}\\ \mathbf{P}^{\mathrm{sr}} & \mathbf{P}^{\mathrm{ss}}\end{bmatrix} \quad \mathbf{p} \overset{\mathcal{P}}{\leftarrow} \begin{bmatrix}\mathbf{p}^{\mathrm{r}}\\ \mathbf{p}^{\mathrm{s}}\end{bmatrix} \tag{7}$$

$$\mathbf{Q} \overset{\mathcal{P}}{\leftarrow} \begin{bmatrix}\mathbf{Q}^{\mathrm{tt}} & \mathbf{Q}^{\mathrm{ts}}\\ \mathbf{Q}^{\mathrm{st}} & \mathbf{Q}^{\mathrm{ss}}\end{bmatrix} \quad \mathbf{q} \overset{\mathcal{P}}{\leftarrow} \begin{bmatrix}\mathbf{q}^{\mathrm{t}}\\ \mathbf{q}^{\mathrm{s}}\end{bmatrix} \tag{8}$$

The symbol $\mathbf{G} \overset{\mathcal{P}}{\leftarrow} \mathbf{H}, \mathbf{g} \overset{\mathcal{P}}{\leftarrow} \mathbf{h}$ means matrix $\mathbf{G}$ comes from entries of $\mathbf{H}$ but after some row/column-wise permutation $\mathbf{F} \in \mathcal{P}^{n\times n}$, so $\mathbf{G} := \mathbf{F}^{-1}\mathbf{H}\mathbf{F}, \mathbf{g} := \mathbf{F}^{-1}\mathbf{h}$, such that rows/-columns of $\mathbf{P}, \mathbf{Q}$ list boundary pixels counterclockwise in the ordering of Figure 4. See §E for details. The rationale behind dividing the domain is that once values at the boundary wire-frame $\mathbf{r}, \mathbf{s}, \mathbf{t}$ are known, the sub-problems to solve for $\mathbf{u}_\mathrm{a}$ and $\mathbf{u}_\mathrm{b}$ become independent: constructions of reduced systems $\mathbf{P}, \mathbf{Q}$ are independent of each other, concurrently computed (in a size-2 batch).

3.2.2. Schur step: merge adjacent DtNs

A Schur step merges two subdomains P and Q into a joint domain D, while converting their left-hand and right-hand sides $(\mathbf{P}, \mathbf{p})$ and $(\mathbf{Q}, \mathbf{q})$ into a new left-hand and right-hand sides $(\mathbf{D}, \mathbf{d})$. Divide the nodes in P into *contiguous* subsets $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{\epsilon}$, such that $\boldsymbol{\alpha} = [0, 1, 2, 3]$, $\boldsymbol{\beta} = [4]$, $\boldsymbol{\gamma} = [5, 6, 7]$, $\boldsymbol{\delta} = [8]$, $\boldsymbol{\epsilon} = [9, 10, 11, 12, 13, 14, 15]$. Divide the nodes in Q into contiguous subsets $\boldsymbol{\kappa}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{\nu}$, such that: $\boldsymbol{\kappa} = [0]$, $\boldsymbol{\lambda} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$, $\boldsymbol{\mu} = [12]$, $\boldsymbol{\nu} = [13, 14, 15]$. Under the indexing in Figure 4, D's boundary consists of the nodes corresponding to $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\lambda}, \boldsymbol{\delta}, \boldsymbol{\epsilon}$. After merging, $\boldsymbol{\beta}, \boldsymbol{\kappa}$ represent the same node , so are $\boldsymbol{\delta}, \boldsymbol{\mu}$; $\boldsymbol{\gamma}$ represents the same set of nodes as $\boldsymbol{\nu}$ but reversely ordered.

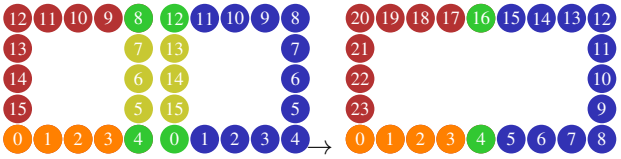


Figure 4. Schur step collapses subdomains P and Q into D.

The node grouping implies partitioning the matrix $\mathbf{P}, \mathbf{Q}$ into submatrices, by dividing the rows and columns into subsets.

$$\begin{bmatrix}\mathbf{P}_{\alpha\alpha} & \mathbf{P}_{\alpha\beta} & \mathbf{P}_{\alpha\gamma} & \mathbf{P}_{\alpha\delta} & \mathbf{P}_{\alpha\epsilon}\\ \mathbf{P}_{\beta\alpha} & \mathbf{P}_{\beta\beta} & \mathbf{P}_{\beta\gamma} & \mathbf{P}_{\beta\delta} & \mathbf{P}_{\beta\epsilon}\\ \mathbf{P}_{\gamma\alpha} & \mathbf{P}_{\gamma\beta} & \mathbf{P}_{\gamma\gamma} & \mathbf{P}_{\gamma\delta} & \mathbf{P}_{\gamma\epsilon}\\ \mathbf{P}_{\delta\alpha} & \mathbf{P}_{\delta\beta} & \mathbf{P}_{\delta\gamma} & \mathbf{P}_{\delta\delta} & \mathbf{P}_{\delta\epsilon}\\ \mathbf{P}_{\epsilon\alpha} & \mathbf{P}_{\epsilon\beta} & \mathbf{P}_{\epsilon\gamma} & \mathbf{P}_{\epsilon\delta} & \mathbf{P}_{\epsilon\epsilon}\end{bmatrix} := \mathbf{P}, \quad \begin{bmatrix}\mathbf{p}_\alpha\\ \mathbf{p}_\beta\\ \mathbf{p}_\gamma\\ \mathbf{p}_\delta\\ \mathbf{p}_\epsilon\end{bmatrix} := \mathbf{p},$$

$$\begin{bmatrix}\mathbf{Q}_{\kappa\kappa} & \mathbf{Q}_{\kappa\lambda} & \mathbf{Q}_{\kappa\mu} & \mathbf{Q}_{\kappa\nu}\\ \mathbf{Q}_{\lambda\kappa} & \mathbf{Q}_{\lambda\lambda} & \mathbf{Q}_{\lambda\mu} & \mathbf{Q}_{\lambda\nu}\\ \mathbf{Q}_{\mu\kappa} & \mathbf{Q}_{\mu\lambda} & \mathbf{Q}_{\mu\mu} & \mathbf{Q}_{\mu\nu}\\ \mathbf{Q}_{\nu\kappa} & \mathbf{Q}_{\nu\lambda} & \mathbf{Q}_{\nu\mu} & \mathbf{Q}_{\nu\nu}\end{bmatrix} := \mathbf{Q}, \quad \begin{bmatrix}\mathbf{q}_\kappa\\ \mathbf{q}_\lambda\\ \mathbf{q}_\mu\\ \mathbf{q}_\nu\end{bmatrix} := \mathbf{q}.$$

(9) is the linear system for the joint domain D. Since $\boldsymbol{\beta}, \boldsymbol{\kappa}$ represent the same node, they correspond to the same row/-column; the same applies to $\boldsymbol{\delta}, \boldsymbol{\mu}$. Now we assemble the new system $\mathbf{D}$ by Schur "involuting" the sub-systems $\mathbf{P}, \mathbf{Q}$. When eliminating the "wire-frame" nodes $\boldsymbol{\gamma}$ (i.e. $\boldsymbol{\nu}$ in reverse order), introduce the symbols to simplify notation:

$$\mathbf{X} := \begin{bmatrix}\mathbf{P}_{\alpha\alpha} & \mathbf{P}_{\alpha\beta} & \mathbf{0}_{\alpha\lambda} & \mathbf{P}_{\alpha\delta} & \mathbf{P}_{\alpha\epsilon}\\ \mathbf{P}_{\beta\alpha} & \mathbf{P}_{\beta\beta} + \mathbf{Q}_{\kappa\kappa} & \mathbf{Q}_{\kappa\lambda} & \mathbf{P}_{\beta\delta} + \mathbf{Q}_{\kappa\mu} & \mathbf{P}_{\beta\epsilon}\\ \mathbf{0}_{\lambda\alpha} & \mathbf{Q}_{\lambda\kappa} & \mathbf{Q}_{\lambda\lambda} & \mathbf{Q}_{\lambda\mu} & \mathbf{0}_{\lambda\epsilon}\\ \mathbf{P}_{\delta\alpha} & \mathbf{P}_{\delta\beta} + \mathbf{Q}_{\mu\kappa} & \mathbf{Q}_{\mu\lambda} & \mathbf{P}_{\delta\delta} + \mathbf{Q}_{\mu\mu} & \mathbf{P}_{\delta\epsilon}\\ \mathbf{P}_{\epsilon\alpha} & \mathbf{P}_{\epsilon\beta} & \mathbf{0}_{\epsilon\lambda} & \mathbf{P}_{\epsilon\delta} & \mathbf{P}_{\epsilon\epsilon}\end{bmatrix}$$

$$\mathbf{Y} := \begin{bmatrix}\mathbf{P}_{\alpha\gamma}\\ \mathbf{P}_{\beta\gamma} + \mathbf{Q}_{\kappa\nu}\mathbf{J}\\ \mathbf{Q}_{\lambda\nu}\mathbf{J}\\ \mathbf{P}_{\delta\gamma} + \mathbf{Q}_{\mu\nu}\mathbf{J}\\ \mathbf{P}_{\epsilon\gamma}\end{bmatrix} \quad \mathbf{y} := \begin{bmatrix}\mathbf{p}_\alpha\\ \mathbf{p}_\beta + \mathbf{q}_\kappa\\ \mathbf{q}_\lambda\\ \mathbf{p}_\delta + \mathbf{q}_\mu\\ \mathbf{p}_\epsilon\end{bmatrix} \quad \mathbf{x} := \begin{bmatrix}\mathbf{u}_\alpha\\ \mathbf{u}_\beta\\ \mathbf{u}_\lambda\\ \mathbf{u}_\delta\\ \mathbf{u}_\epsilon\end{bmatrix}$$

$$\mathbf{Z} := \begin{bmatrix}\mathbf{P}_{\gamma\alpha} & \mathbf{P}_{\gamma\beta} + \mathbf{J}^\intercal\mathbf{Q}_{\nu\kappa} & \mathbf{J}^\intercal\mathbf{Q}_{\nu\lambda} & \mathbf{P}_{\gamma\delta} + \mathbf{J}^\intercal\mathbf{Q}_{\nu\mu} & \mathbf{P}_{\gamma\epsilon}\end{bmatrix}$$

$$\mathbf{W} := \begin{bmatrix}\mathbf{P}_{\gamma\gamma} + \mathbf{J}^\intercal\mathbf{Q}_{\nu\nu}\mathbf{J}\end{bmatrix} \quad \mathbf{w} := \begin{bmatrix}\mathbf{p}_\gamma + \mathbf{J}^\intercal\mathbf{q}_\nu\end{bmatrix}$$

where $\mathbf{J} \equiv \mathbf{J}^\intercal$ is the reverse permutation matrix (see §E), whose action is to reverse the rows (or columns) of a matrix when being multiplied with from left (or right). By plugging in our new definitions, (9) becomes:

$$\begin{bmatrix}\mathbf{X} & \mathbf{Y}\\ \mathbf{Z} & \mathbf{W}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \mathbf{u}_\gamma\end{bmatrix} = \begin{bmatrix}\mathbf{y}\\ \mathbf{w}\end{bmatrix} \tag{10}$$

Schur eliminate $\mathbf{u}_\gamma$ using:

$$\mathbf{u}_\gamma = \mathbf{W}^{-1}(\mathbf{w} - \mathbf{Z}\mathbf{x}) \quad \text{(back-fill)}, \tag{11}$$

the system (4) to (5) to (9) finally becomes $\mathbf{D}\mathbf{x} = \mathbf{d}$, where:

$$\mathbf{D} := \left(\mathbf{X} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{Z}\right) \quad \mathbf{d} := \mathbf{y} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{w} \tag{12}$$

In summary, to solve the original problem (4) in the special case of two subdomains, after the Schwarz step that constructs the system matrices $\mathbf{P}, \mathbf{Q}$, our final algorithm of the Schur step first solves for the values $\mathbf{x} = \mathbf{D}^{-1}\mathbf{d}$, then recovers $\mathbf{u}_\gamma$—the solution at the interface—using (11), and finally recovers $\mathbf{u}_\mathrm{a}, \mathbf{u}_\mathrm{b}$—values in the interior of each patch from $\mathbf{u}_\mathrm{r}, \mathbf{u}_\mathrm{t}, \mathbf{u}_\mathrm{s}$—values at the patch's boundary.

3.2.3. General cases

Summarized in Figure 2, when there are 2^k patches of 5×5, we apply the Schwarz step once, and the Schur step k times to recursively glue every two subdomains together. These steps play the role of the standard LU factorization (Davis, 2006). See full details in §C, §E and Algorithm 1.

The Schwarz step to eliminate interior nodes in the 3×3 block is basically the same as introduced before: for each patch P the interior nodes $\mathbf{a}$ are eliminated and the system matrix for the remaining nodes $\mathbf{r}$ becomes $(\mathbf{A}_{\mathrm{rr}}^{(P)} - \mathbf{A}_{\mathrm{ra}}^{(P)}\mathbf{A}_{\mathrm{aa}}^{-1}\mathbf{A}_{\mathrm{ar}}^{(P)})$.

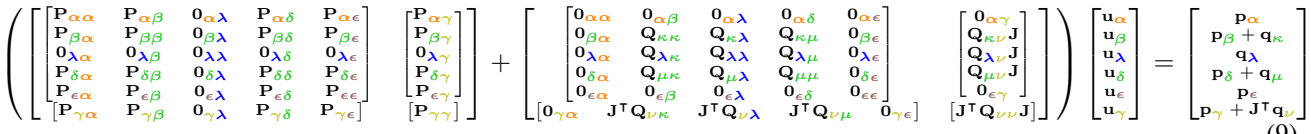


Figure 5. The joint linear system. (9)

The step transforms the domain to a wire-frame with 2^k hollow subdomains, each of which initially has one patch and is associated with a system matrix (like $\mathbf{P}, \mathbf{Q}, ...$) and a right-hand side ($\mathbf{p}, \mathbf{q}, ...$). Again, this stage takes a small fraction of overall runtime but removes the majority of pixels.

A Schur step further simplifies the wire-frame by eliminating pixels on some "edges" along the x (or y) direction for j that is odd (or even). The j-th Schur step ($j = 1, ..., k$) starts with 2^{k-j+1} subdomains that each consists of 2^{j-1} patches, and ends with 2^{k-j} subdomains that each has 2^j patches. To merge each pair of adjacent subdomains that are marked in red and blue, we gather the system matrices for them, also denoted as $\mathbf{P}, \mathbf{Q}$, and apply the formula (12).

Tensorized representations of batched reduced systems are adopted as our method's data structure: we never maintain the global system $\mathbf{A}$ as a sparse matrix like all other direct solvers. Instead, in the j-th Schur step of our algorithm, the linear system is a 4-dim tensor $\boldsymbol{\alpha}^{(j)}$ that $\boldsymbol{\alpha}^{(j)}[k, l, :, :]$ stores the left-hand side of the (k, l)-th subdomain. For $j = (2i + 1)$, the j-th Schur step takes as input $\boldsymbol{\alpha}^{(j)}$ of size $(2^{k/2-i}, 2^{k/2-i}, 16 \cdot 2^i, 16 \cdot 2^i)$, and outputs $\boldsymbol{\alpha}^{(j+1)}$ of size $(2^{k/2-i-1}, 2^{k/2-i}, 24 \cdot 2^i, 24 \cdot 2^i)$; $\boldsymbol{\alpha}^{(j)}$ contains all reduced systems; $\mathbf{P}, \mathbf{Q}$ are fetched batch-wise from $\boldsymbol{\alpha}^{(j)}$ for all red and blue subdomains, e.g., in PyTorch syntax:

```
P = a{j}[1::2, :, :, :] # odd  subdomains
Q = a{j}[2::2, :, :, :] # even subdomains
```

Batched dense linear algebras are employed for automatic parallelisms in PyTorch. For superior performance, it is critical to batch matrix operations including inversion, multiplication, sum, slicing into rows and columns, etc. For example, $\mathbf{W}, \mathbf{X}, \mathbf{Y}, \mathbf{Z}$ are 4-dim tensors obtained by batch-wise slicing into the last two dimensions of $\mathbf{P}, \mathbf{Q}$, e.g., $\mathbf{W} := \mathbf{P}[:, :, \gamma, \gamma] + \mathbf{J}^\intercal\mathbf{Q}[:, :, \nu, \nu]\mathbf{J}$, and the following code computes the merged system across all subdomains:

```
D = X - Y * torch.linalg.inv(W) * W
```

3.3. Discussion

While our method is simple to describe—recursively applying the Schur complement formula in batches, fully understanding the motivation and appreciating the superiority in performance and numerical behavior, empirically demonstrated in §4, requires familiarity with in numerical methods, elliptic PDEs, and references. We highlight a few aspects.

One question is why the matrices $\mathbf{D}, \mathbf{P}, \mathbf{Q}$ are numerically well-behaving and the sum of their sub-blocks $\mathbf{W}$ is invertible. When the system $\mathbf{A}$ comes from discretizing an elliptic PDE, including the "Laplacian matrix" considered in graph theory and computer vision, which means $\mathbf{A}$ is positive semidefinite (PSD) and $\mathbf{A}_{ii} = -\sum_{j:j\neq i} \mathbf{A}_{ij}$, the Schur complements $\mathbf{D}, \mathbf{P}, \mathbf{Q}$ are also PSD, and discretizing the Dirichlet-to-Neumann (DtN) operator, which is well-defined and enjoys numerous nice theoretical properties. See Wang et al. (2018); Levitin et al. (2023) and references therein.

DtN also plays a central role in the study of domain decomposition in numerical methods (Quarteroni & Valli, 1999). Consider the classic example of solving Laplace's equation under Dirichlet boundary condition. After dividing the image domain into subdomains, however, the usual problem arises: Dirichlet data at the interface between subdomains are *unknown*, preventing naïve divide-and-conquer. But whatever the Dirichlet data at the interface is, the solution is linear to it (through the Green's function), since we work with a linear PDE: the DtN operator encodes the solution operator that yields the solution for whatever Dirichlet data—a simple intuition why DtN is central in domain composition (Quarteroni & Valli, 1999). Unlike common practices in larger problems that implicitly realize DtN's matrix-vector-product (Liu et al., 2016), our algorithm maintains the DtN/solution operators *explicitly as dense matrix*.

A note on novelty. We remark that key ingredients of our algorithm exist in the literature. We essentially perform Gaussian elimination with a particular pivoting ordering induced by the nested dissection (George, 1973). Similar procedures were applied in early studies of distributing FEM solves among multiple processors, e.g., in civil (Farhat & Wilson, 1987; Farhat et al., 1987; Farhat & Roux, 1991) and ocean engineering (Rakowsky, 1999). However, our setting is different from scientific computing communities, which focus on much larger problems where inter-processor communication rises as a central consideration. Without recent advances in GPUs both in terms of high throughput and memory size, stirred by the needs of deep learning, applying the Schur formula to invert a massive amount of small dense matrices on a single device is either infeasible or uncompetitive compared to alternative direct solvers. Solving a massive amount of dense systems on a single device—without needing cross-device communication, is made efficiently only recently, to rise as the best practice.

Condense sparsity and restrict the number of variables. For an $10^3 \times 10^3$ image, its discrete Laplacian $\mathbf{A}$ is a sparse matrix of $10^6 \times 10^6$, which is much too large to deal with as a dense matrix. If we can instead somehow work with one (dense) $10^3 \times 10^3$ matrix and a few smaller ones, then we can leverage the efficient dense linear algebraic kernels

Table 2. Uncertain runtime of iterative solvers. When solving an anisotropic or isotropic Laplacian (Dirichlet) system, our solver is not affected by anisotropy, while AMG (Ruge-Steuben) (Bell et al., 2022) can be much slower compared to solving isotropic systems.

	anisotropic Laplacian							isotropic Laplacian		
Example	CUDA	SciPy	ours	ours-64	AMG (1e-4)	AMG (1e-7)	AMG (1e-10)	AMG (1e-4)	AMG (1e-7)	AMG (1e-10)
2049^2	21354	143100	158.5	OOM	12205	49487	53877	6559	7757	9202
1025^2	4710	16512	36.45	175.6	2678	8112	13446	1388	1754	2236
513^2	1036	2051	10.90	31.81	832.2	1658	2353	321.9	387.2	515.3
257^2	234	355	5.82	7.46	203.0	385.2	556.0	91.0	117.3	347.8

BLAS (Dongarra et al., 1988; 2018). Indeed, the largest dense matrix we have to invert is at the scale of $10^6 \times 10^6$.

Handling different boundary conditions. Beyond handling the usual Dirichlet or Neumann boundary condition, we even introduce a midpoint reflective boundary condition to handle no-disk topology such as the sphere. See §C,§D.5.

4. Results and Validations

We compare our method with direct and iterative solvers and demonstrate significant advantages in runtime.

Comparison with direct solvers. As the performance of direct solvers, especially our method is less affected by $\mathbf{A}$, Table 1 can represent the timing comparison for whatever $\mathbf{A}$. Our method supports both single and double floating-point numbers (float-32 and float-64), while existing direct solvers only support float-64, or support float-32 without major speedups (cuDSS). Implemented in float-64, our method often achieves a relative error of $10^{-16}-10^{-14}$, even better than SciPy (Table 9). Our method can use float-32 for further speedup, in which case the error range from $10^{-6}-10^{-5}$, a typical error tolerance that iterative solvers use, sufficient for many tasks. We evaluate the accuracy of the solution using the standard metric—the (relative) error tolerance:

$$\text{tol}(\mathbf{x}) := \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 / \|\mathbf{b}\|_2 \tag{13}$$

which is the stop criterion for most iterative methods. For direct solvers, we compare with SciPy (Virtanen et al., 2020) and CUDA (Nickolls et al., 2008)—the cuDSS LDU method which defaults to the METIS (nested dissection) ordering. For iterative solvers, we compare with the algebraic multigrid solver (Ruge-Stuben) from PyAMG (Bell et al., 2022). For SciPy, we use the "spsolve" function, which defaults to the COLAMD ordering and is backended by SuperLU (Li, 2005). While CUDA can be faster than SciPy, it is still on the same order of magnitude and cannot achieve interactive rates like ours. Our experiments are collected on an Nvidia GPU A6000 Ada 48GB and an Intel CPU Xeon w9-3475X.

The speedup from our approach potentially can be much more significant than what is reported in the paper. Compared to standard solvers that is already highly optimized, ours is naïvely prototyped in PyTorch—re-implementing our algorithm as low-level operators or optimization at the hardware level like CUDA solvers will result in further speed increases. Additionally, direct solvers including ours enjoy extra speedups in repetitive solves by reusing numerical factorization—impossible for iterative solvers.

Comparison with iterative solvers. §F.2 extensively discusses why iterative solvers can be undesirable modules in deep learning pipelines. Note that although we do compare with iterative methods and demonstrate advantages, direct methods remain our primary theoretical point of comparison. Direct solvers are more reliable and robust, with a predictable runtime; in contrast, iterative solvers have performance that heavily depends on $\mathbf{A}$, and in challenging cases can significantly slow down or fail.

Table 3. The time of a single linear solve on a few problems, solving to the relative tolerance of 10^{-6} for iterative solvers. No iterative solvers out-of-the-box can perform well on all tasks. "(s)": CG, MINRES benefit from the symmetry assumption when it holds, while other methods do not. "$\times k$": time for iterative solvers should be multiplied by an extra factor of $2 \sim 20+$ in some applications, since iterative solvers need to run 2 times to differentiate, and $20+$ times in eigen solving, while direct solvers do the factorization once and reuse it. "NaN": the solver diverges or fails to yield a small error. "*" indicates the method stops at a larger tolerance since it adopts a different stopping criteria: 1: 1.19e-5, 2: 1.16e-4, 3: 2.6e-6, 4: 1.41e-4, 5: 2.11e-6, 6: 1.41e-4, 7: 2.84e-6.

	Lap 1e-2	Lap 1e-4	Hel 1e-2	Hel 1.0	R-Decon	
Ours-64	35.6	35.4	35.6	35.5	35.5	
Ours-32	12.3	12.2	12.3	12.2	NaN	
SciPy-LU	2208	2487	2333	2133	3145	
CUDSS-LDU	1011	1029	1040	1020	1038	
(s) CG	65.1	459	563	6442	NaN	$\times k$
(s) MINRES	97.3	1319	1595 (*3)	11156 (*5)	NaN	
LSMR	7288	426499	76928	80617	12282	
LSQR	2896 (*1)	215758 (*2)	84442 (*4)	16158 (*6)	3290 (*7)	
biCGstab	299	1898	NaN	NaN	NaN	
CGS	58.3	922	NaN	NaN	NaN	
DIOM	228	12716	1484	13221	NaN	
FOM	1157	499930	61344	NaN	NaN	
QMR	70.8	570	463	6849	NaN	
BILQ	86.5	634	635	6878	NaN	
GMRES	954	31017	56478	4675551	NaN	
DQ-GMRES	172	1029	1391	12885	NaN	
F-GMRES	933	30793	56665	NaN	NaN	

We compare with out-of-the-box GPU linear solvers from CuPy (Okuta et al., 2017) and Julia (Bezanson et al., 2017), using the CUDA implementation in the Krylov.jl package (Montoison & Orban, 2023). We are primarily interested in indefinite and nonsymmetric square linear solvers including: LSQR, LSMR, CGS, biCGstab, DIOM. FOM, QMR, BILQ, GMRES, DQGMRES, and FGMRES, While we also report results on CG and MINRES, please note that the comparison may be slightly unfair: they benefit from the symmetry assumption that other methods do not. As shown in Table 3, out-of-the-box iterative solvers can slow down by many orders of magnitude or fail to converge for harder prob-

lems. We test all methods on solving the Laplacian equation $\mathbf{A} = \mathbf{L} + \lambda\mathbf{M}$, the Helmholtz equation $\mathbf{A} = \mathbf{L} - \kappa^2\mathbf{M}$ under different parameters, and random $\mathbf{A}$—whose entries $\mathbf{A}_{ij}$ are i.i.d., uniformly drawn from $[-1, 1]$. The iterative solvers are highly problem-dependent—changing the parameters that generate the problem can slow down them significantly; in contrast, our solver is problem-independent.

Anisotropic Laplacian. We solve Poisson equation with random or constant right-hand sides, with an isotropic or anisotropic kernel. As shown in Table 2, iterative solvers can slow down significantly with anisotropic kernels, while direct solvers are less affected. Table 2 reports the time to solve Laplace equation with Dirichlet condition with either the isotropic kernel $\mathbf{K}_1$, or the anisotropic $\mathbf{K}_2$ in Figure 9. The number of iterations increases for anisotropic kernels by a factor of $6\times$, with the same number of non-zeros (0s in $\mathbf{K}_1$ are treated as non-zero entries with value 0.0).

Unlike general-purpose sparse solvers, we do not aim at a solver with minimal floating-point operations or memory consumption and reducing fill-in sparsity during the solving, for very large-scale problems with billions of variables. Instead, we focus on runtime for common image sizes, such as 1024×1024 with around 1 million variables. As a limitation: our method is more demanding in memory (see §D.4).

Parabolic PDEs and high-precision diffusion kernels. The parabolic PDE $\frac{du(x,t)}{dt} = \Delta u(x,t)$ is solved by our method. The standard practice to approximate $u(x,t)$ is the implicit Euler scheme that solves the system $\mathbf{A} \leftarrow t\mathbf{L} + \mathbf{M}$. Heat geodesics is a situation in which the **high precision of direct solvers within a tolerance of** 10^{-10} becomes critical. Crane et al. (2017) approximate geodesic distance from a point x_0, using the solution under the initial condition $u(x, 0) = \delta(x - x_0)$. The right-hand side is a delta function that is nonzero only at one pixel. The solution $u(x,t)$, known as the heat kernel, is a Gaussian-like function centered at x_0, with infinite support. Then geodesics are recovered from $\log u(x,t)$—logarithm is of direct interests. $\mathbf{x}$ has to be computed within a tolerance of 10^{-10} or even 10^{-15} to produce non-negative heat kernel values to apply the heat method (Crane et al., 2017). Requiring a tiny tolerance makes iterative methods incompetent, while direct solvers can be simply applied. Figure 8 shows the result of our method. Our method (float-64) solves the system to an error tolerance of 10^{-16}, smaller than that of SciPy. Our method can succeed under extreme numerics even when SciPy fails, by decreasing the stepsize $t \to 0$.

4.1. A zero-shot baseline of efficient PDE solvers

Learned PDE solvers emerge in scientific tasks (Lu et al., 2019; 2021; Raissi et al., 2019; Lagaris et al., 1998). In addition to surrogate modeling (*e.g.*, when the governing PDEs are unknown and must be inferred from data), a major advantage of learned solvers over conventional numerical methods is speed: the former can be 3 orders-of-magnitude faster, the main inspiration for our work. Surprisingly, we demonstrate a *zero-shot* approach also capable of improving runtime by only leveraging GPUs, without needing training.

A successful line of research achieves orders-of-magnitude speedup in solving the PDE (2) which is also known as a Darcy flow, by learning the *coefficient-to-solution* operator (Li et al., 2020; Wang et al., 2021; Li et al., 2024). Our solver can implement FEM to realize the same operator, which becomes the mapping: $\mathbf{c} \to \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ where $\mathbf{A} := \mathbf{G}^\intercal \mathrm{diag}([\mathbf{c}, \mathbf{c}])\mathbf{G}$. We compare with the state-of-the-art learning solver on the task of Darcy flows. We test on 1024 Darcy flow examples available with the Python package "neuraloperator" from Li et al. (2024) resampled from 421^2 to 257^2. Our method as a direct solver does not need any training and solves the linear system within a tolerance of 10^{-12} (float 64) on all examples. For our method the relative "errors" are $0.869\% \pm 0.0927\%$, smaller than that of 1.56% in Li et al. (2024). Note that the "error" of our method, i.e., differences between our result and the ground truth, is due solely to the use of different numerical schemes. Our method, if used as the simulation engine, *is the ground truth*, achieving comparable "inference" speed without any training. Ours at 257^2 has a running time of $5 - 7$ ms, comparable with many neural architectures—Darcy flows take a few milliseconds for some recent methods (Li et al., 2025).

The results suggest the suitability of our method as an efficient solver-in-the-loop importable by existing architectures, such as solution super-resolution (Ren et al., 2023). As learning-based solvers increasingly import modules from numerical PDEs, it is promising to generalize our involution to procedures with learnable parameters for data-driven discretizations (Wang et al., 2019; Bar-Sinai et al., 2019).

5. Applications

Featured in Figure 1, due to the foundational role of the linear solver, numerous applications immediately benefit from our method, removing the need to design problem-specific solvers. For $\mathbf{A}$ coming from different applications, our solver consistently demonstrates speedup over other direct solvers to a level similar to the case of solving PDEs in §4. Examples of sparse $\mathbf{A}$ encompass:

Mathematical optimization. When $\mathbf{A}, \mathbf{b}$ are the Hessian and gradient, our solver implements the **Newton's method**. See §B.1 for detailed discussions and experiments. Despite theoretical superiority, Newton's method has very limited applications in practice due to the expensive Hessian solves, an obstacle that our orders-of-magnitude speedup removes. Similar use cases include shape optimization and inverse rendering (Nicolet et al., 2021).

The Laplacian paradigms for graph theory (Chung, 1997), spectral clustering, manifold learning, image/geometry processing, and vision have been extremely successful; see Wang & Solomon (2019); Chen et al. (2021b) for a survey. Early examples include: Laplace equations in optical flow (Horn & Schunck, 1981), semi-supervised learning with harmonic label propagation (Zhu et al., 2003), Perona–Malik model with anisotropic diffusion (Perona & Malik, 1990), and normalized cuts for segmentation (Shi & Malik, 2000). These are examples where PDE or linear algebraic solvers play a central role and Δ^{-1} or $(t\Delta+\mathbf{I})^{-1} \approx e^{-t\Delta}$, instead of the forward operator Δ, is of primary interest. Sparse solvers play an even more prominent role in geometry modeling, where almost every algorithm solves some form of Laplacian systems (Solomon et al., 2014).

Image processing: matting, segmentation, and editing. $\mathbf{A}$ can be a graph or matting Laplacian $\mathbf{L}$ that $\mathbf{L}_{ij}$ encodes the similarity between adjacent pixels. In this setting, $\mathbf{L}$ can be viewed as discretizing an anisotropic Laplace equation. Our solver $\mathbf{L}^{-1}\mathbf{b}$ can be used for spectral segmentation (Shi & Malik, 2000): see §B.4. Often a constraint term is added to the linear system $\mathbf{A} := \mathbf{L}+\lambda\mathbf{R}\mathbf{R}^\intercal$, in image matting, and Poisson image editing (Pérez et al., 2003). While the constraint term $\mathbf{R}\mathbf{R}^\intercal$ makes the problem's difficulty unevenly distributed across the image and a large λ makes the system ill-conditioned, our method in double precision consistently yields an error smaller than SciPy. See details in §B.3.

Diffeomorphic image registration. Diffeomorphisms, or smooth bijective maps between image domains (Younes, 2010), are not only a central concept in differential manifolds and geometry, but also foundational in image registration (Beg et al., 2005). Recent results in variational quasi-harmonic maps (Wang et al., 2023) demonstrate that the map $\mathbf{x} \to (u(\mathbf{x}), v(\mathbf{x}))$ is diffeomorphic if and only if $\nabla \cdot [\mathbf{C}(\mathbf{x})\nabla u(\mathbf{x})] = 0$ (resp. $v(\mathbf{x})$) for some $\mathbf{C}(\mathbf{x})$, with appropriate boundary conditions. Intuitively, this condition states that each pixel must be placed about some weighted average of its neighbors' positions, which is enforced by a sparse linear system. To compute diffeomorphic image registration in Figure 7, we use Wang et al. (2023), which recursively solves anisotropic Laplacian systems. Replacing their linear solver with ours as the backend again leads to orders-of-magnitude speedups. Registering a pair of images of size 1024×768 requires solving 150 sparse systems sequentially and takes 5 seconds with our solver, compared to 8 minutes in Wang et al. (2023) if using a SciPy backend.

Geometry algorithms. §4 shows how to reduce geometric PDEs to anisotropic PDEs: evaluate $\mathbf{L}, \mathbf{M}$ using a curved surface's metric, pushed back to the plane. §B.5 handles non-disk topology with appropriate boundary conditions.

Generalized deconvolution. As shown in Figure 6, we recover an image from its convolution with a spatially varying kernel in the form (1), which reduces to solving a linear system (Hansen et al., 2006). To our knowledge, the only algorithm applicable to this setting is to use linear solvers (since solving PDE reduces to this problem). Our method is again orders-of-magnitude faster than SciPy: on 257^2 images, SciPy has an average runtime of 494.4ms, slowing down compared to solving the Laplacian in Table 2, while it takes 7.51ms for ours (float-64). As expected, PyAMG fails for kernel $a_2^{(x,y)}$ since it is not symmetric.

(Differentiable) physics simulation. When $\mathbf{A}$ comes from discretizing a physics-based energy, our method immediately accelerates the physical simulator. §B.2 shows an example. Robot learning engines (Du et al., 2021; Hu et al., 2019) and scientific machine learning (Sanchez-Gonzalez et al., 2020) are increasingly relying on solver-in-the-loop (Amos & Kolter, 2017) using CUDA sparse solvers, which can be replaced with ours.

6. Conclusion and Future Work

It is observed that in many areas—from medical image analysis to learning for solving PDEs—traditional optimization-based methods can be orders-of-magnitude slower compared to deep learning. We identify that this is partially because the implementation of conventional methods is not specially tailored to a new setting of HPC (high performance computing) on a single workstation, enabled by GPUs. In interactive vision and graphics, considerable effort has been devoted to designing numerical schemes that avoid direct linear solvers, with the assumption that exact linear solvers are too slow. Whether or not the computation time is within tens of milliseconds makes an *essential* difference for the deployability of algorithms in video conferencing, self-driving cars, and virtual reality environment. For example, Bro-Nielsen & Cotin (1996) note *"speed is everything"* in virtual surgery. Our method eliminates the need for customizing solvers, and revives methods that used to be slow due to (repetitive) linear solvers in a straightforward manner. Our method can serve as a strong baseline for learning-based PDEs, or an efficient solver/simulator-in-the-loop in robot learning or vision pipelines (Barron & Poole, 2016).

Due to the foundational role of sparse linear solvers, our method can potentially impact a broad range of applications beyond what is presented. Many classical algorithms can be viewed as one or more steps of the Schwarz–Schur involution—solving a linear Laplacian-like system (Barron & Poole, 2016; Pérez et al., 2003; Germer et al., 2020; Shi & Malik, 2000). For future work, we plan to design neural architectures based on the SS-involution, generalizing optimization-based Laplacian paradigms, and extend our method to work with an arbitrary mesh/graph. Our exact solver can be converted to an approximate solver by downsampling the DtN and integrated with iterative schemes.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning, Optimization, Computer Vision, and Scientific Computing. We call attention to the fact that even for a fundamental and well-studied task—solving linear systems, speedups of up to $1000\times$ are still achievable, demonstrating the potential for substantial performance gains by developing novel methods like our solver to best leverage hardware capacity. We suspect that perhaps the advances in GPUs have been underutilized in terms of improving conventional methods, with or without relying on deep learning. Possible societal impacts of our work include *improved energy efficiency* and reduced carbon footprint: significantly reducing computational time, methods like ours lower power consumption and contribute to a more *sustainable and environmentally friendly computing* landscape.

Acknowledgements

We thank Justin Solomon and Mike Taylor for insightful discussions. We thank the anonymous reviewers, especially for their comments on improving clarity, self-containment, and accessibility to the broader ICML community. YB is supported by a fellowship from the Royal Society (NIF-R1-232460). Support for this research was provided in part by the BRAIN Initiative Cell Atlas Network (BICAN) grants U01MH117023 and UM1MH130981, the Brain Initiative Brain Connects consortium (U01NS132181, 1UM1NS132358-01), the National Institute for Biomedical Imaging and Bioengineering (1R01EB023281, R21EB018907, R01EB019956, P41EB030006), the National Institute on Aging (R21AG082082, 1R01AG064027, R01AG016495, 1R01AG070988), the National Institute of Mental Health (UM1MH130981, R01 MH123195, R01 MH121885, 1RF1MH123195), the National Institute for Neurological Disorders and Stroke, (1U24NS135561-01, R01NS070963, 2R01NS083534, R01NS105820, R25NS125599), and was made possible by the resources provided by Shared Instrumentation Grants 1S10RR023401, 1S10RR019307, and 1S10RR023043. Much of the computation resources required for this research was performed on computational hardware generously provided by the Massachusetts Life Sciences Center (https://www.masslifesciences.com/). In addition, BF is an advisor to DeepHealth, a company whose medical pursuits focus on medical imaging and measurement technologies. BF's interests were reviewed and are managed by Massachusetts General Hospital and Partners HealthCare in accordance with their conflict of interest policies. Support for this research was provided in part by the National Institute of Diabetes and Digestive and Kidney Diseases (1-R21-DK-108277-01).

Algorithm 1 Schwarz–Schur Involution.
(Overall solve: numerical factorization + back substitution.)

Require: $\boldsymbol{\alpha}^{(*)} \in \mathbb{R}^{2^{k/2}\times 2^{k/2}\times 25\times 25}$
Require: $\boldsymbol{\beta}^{(*)} \in \mathbb{R}^{2^{k/2}\times 2^{k/2}\times 25\times 1}$
Require: $\boldsymbol{\chi}^{(k)} \in \mathbb{R}^{1\times 1\times b\times 1}$ only for Dirichlet condition.
$b := 2H + 2W - 4$ is the number of boundary pixels.
The Schwarz forward step Algorithm 2:

$$\boldsymbol{\alpha}^{(0)}, \boldsymbol{\beta}^{(0)} \leftarrow \boldsymbol{\alpha}^{(*)}, \boldsymbol{\beta}^{(*)}$$

for i in range($k/2$): **do**
The Schur forward step (horizontal)—Algorithm 4:

$$\boldsymbol{\alpha}^{(2i+1)}, \boldsymbol{\beta}^{(2i+1)} \leftarrow \boldsymbol{\alpha}^{(2i)}, \boldsymbol{\beta}^{(2i)}$$

The Schur forward step (vertical):

$$\boldsymbol{\alpha}^{(2i+2)}, \boldsymbol{\beta}^{(2i+2)} \leftarrow \boldsymbol{\alpha}^{(2i+1)}, \boldsymbol{\beta}^{(2i+1)}$$

end for
if Dirichlet boundary condition **then**
$\boldsymbol{\chi}^{(k)}$ is given

$$\boldsymbol{\chi}^{(k)} \leftarrow \ldots$$

else
either use the improved implementation of Dirichlet or other non-disk topology boundary conditions (see §C.2)

$$\boldsymbol{\chi}^{(k)} \leftarrow \ldots$$

or the naïve slow approach: $\boldsymbol{\chi}^{(k)}$ is solved by matrix inversion using: $\boldsymbol{\alpha}^{(k)} \in \mathbb{R}^{1\times 1\times b\times b}$, $\boldsymbol{\beta}^{(k)} \in \mathbb{R}^{1\times 1\times b\times 1}$

$$\boldsymbol{\chi}^{(k)} \leftarrow (\boldsymbol{\alpha}^{(k)})^{-1}\boldsymbol{\beta}^{(k)}$$

end if
for i in range($k/2$, 0, -1) **do**
The Schur backward step (vertical):

$$\boldsymbol{\chi}^{(2i-1)} \leftarrow \boldsymbol{\chi}^{(2i)}$$

The Schur backward step (horizontal)—Algorithm 5:

$$\boldsymbol{\chi}^{(2i-1)} \leftarrow \boldsymbol{\chi}^{(2i)}$$

end for
The Schwarz backward step Algorithm 3:

$$\boldsymbol{\chi}^{(*)} \leftarrow \boldsymbol{\chi}^{(0)}$$

return $\boldsymbol{\chi}^{(*)}$ of size $(2^{k/2}, 2^{k/2}, 25, 25)$.

References

Allaire, G. *Numerical analysis and optimization: an introduction to mathematical modelling and numerical simulation*. OUP Oxford, 2007.

Amos, B. and Kolter, J. Z. Optnet: Differentiable optimization as a layer in neural networks. In *International conference on machine learning*, pp. 136–145. PMLR, 2017.

Arisaka, S. and Li, Q. Principled acceleration of iterative numerical methods using machine learning. In *International Conference on Machine Learning*, pp. 1041–1059. PMLR, 2023.

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

Barron, J. T. and Poole, B. The fast bilateral solver. In *European conference on computer vision*, pp. 617–632. Springer, 2016.

Bebendorf, M. *Hierarchical matrices*. Springer, 2008.

Beg, M. F., Miller, M. I., Trouvé, A., and Younes, L. Computing large deformation metric mappings via geodesic flows of diffeomorphisms. *International journal of computer vision*, 61:139–157, 2005.

Bell, N., Olson, L. N., and Schroder, J. Pyamg: Algebraic multigrid solvers in python. *Journal of Open Source Software*, 7(72):4142, 2022.

Belongie, S., Fowlkes, C., Chung, F., and Malik, J. Spectral partitioning with indefinite kernels using the nyström extension. In *Computer Vision—ECCV 2002: 7th European Conference on Computer Vision Copenhagen, Denmark, May 28–31, 2002 Proceedings, Part III 7*, pp. 531–542. Springer, 2002.

Berens, P., Cranmer, K., Lawrence, N. D., von Luxburg, U., and Montgomery, J. Ai for science: an emerging agenda. *arXiv preprint arXiv:2303.04217*, 2023.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

Blahut, R. E. *Fast algorithms for signal processing*. Cambridge University Press, 2010.

Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM transactions on graphics (TOG)*, 22(3):917–924, 2003.

Bouaziz, S., Martin, S., Liu, T., Kavan, L., and Pauly, M. Projective dynamics: Fusing constraint projections for fast simulation. *Acm Transactions On Graphics*, 33(4): 154, 2014.

Bramble, J. H. *Multigrid Methods*, volume 294. CRC Press, 1993.

Bro-Nielsen, M. and Cotin, S. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. In *Computer graphics forum*, volume 15, pp. 57–66. Wiley Online Library, 1996.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Chen, J., Schäfer, F., Huang, J., and Desbrun, M. Multiscale cholesky preconditioning for ill-conditioned problems. *ACM Transactions on Graphics (TOG)*, 40(4):1–13, 2021a.

Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):1–14, 2008.

Chen, Y., Chi, Y., Fan, J., Ma, C., et al. Spectral methods for data science: A statistical perspective. *Foundations and Trends® in Machine Learning*, 14(5):566–806, 2021b.

Chung, F. R. *Spectral graph theory*, volume 92. American Mathematical Soc., 1997.

Cook, S. On the minimum computation time for multiplication. *Doctoral diss., Harvard U., Cambridge, Mass*, 1, 1966.

Crane, K., Weischedel, C., and Wardetzky, M. The heat method for distance computation. *Communications of the ACM*, 60(11):90–99, 2017.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Davis, T. A. Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004.

Davis, T. A. *Direct methods for sparse linear systems*. SIAM, 2006.

Dongarra, J., Duff, I., Gates, M., Haidar, A., Hammarling, S., Higham, N. J., Hogg, J., Lara, P. V., Luszczek, P., Zounon, M., et al. Batched blas (basic linear algebra subprograms) 2018 specification. Technical report, University of Tennessee, 2018.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

Du, T., Wu, K., Ma, P., Wah, S., Spielberg, A., Rus, D., and Matusik, W. Diffpd: Differentiable projective dynamics. *ACM Transactions on Graphics (TOG)*, 41(2):1–21, 2021.

Duff, I. S. and Reid, J. K. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983.

Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct methods for sparse matrices*. Oxford University Press, 2017.

Ernst, O. G. and Gander, M. J. Why it is difficult to solve helmholtz problems with classical iterative methods. *Numerical analysis of multiscale problems*, pp. 325–363, 2011.

Farhat, C. and Roux, F.-X. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International journal for numerical methods in engineering*, 32(6):1205–1227, 1991.

Farhat, C. and Wilson, E. A new finite element concurrent computer program architecture. *International Journal for Numerical Methods in Engineering*, 24(9):1771–1792, 1987.

Farhat, C., Wilson, E., and Powell, G. Solution of finite element systems on concurrent processing computers. *Engineering with Computers*, 2:157–165, 1987.

Fletcher, R. Conjugate gradient methods for indefinite system. *Springer-Verlag Berlin. Heidelberg. New York*, pp. 73, 1976.

Fong, D. C.-L. and Saunders, M. Lsmr: An iterative algorithm for sparse least-squares problems. *SIAM Journal on Scientific Computing*, 33(5):2950–2971, 2011.

Freund, R. W. and Nachtigal, N. M. Qmr: a quasi-minimal residual method for non-hermitian linear systems. *Numerische mathematik*, 60(1):315–339, 1991.

Freund, R. W. and Nachtigal, N. M. An implementation of the qmr method based on coupled two-term recurrences. *SIAM Journal on Scientific Computing*, 15(2):313–337, 1994.

Frigo, M. and Johnson, S. G. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

George, A. Nested dissection of a regular finite element mesh. *SIAM journal on numerical analysis*, 10(2):345–363, 1973.

Germer, T., Uelwer, T., Conrad, S., and Harmeling, S. Pymatting: A python library for alpha matting. *Journal of Open Source Software*, 5:2481, 2020.

Gilbarg, D., Trudinger, N. S., Gilbarg, D., and Trudinger, N. *Elliptic partial differential equations of second order*, volume 224. Springer, 1977.

Grady, L., Schiwietz, T., Aharon, S., and Westermann, R. Random walks for interactive alpha-matting. In *Proceedings of VIIP*, volume 2005, pp. 423–429. Citeseer, 2005.

Grementieri, L. and Galeone, P. Towards neural sparse linear solvers. *arXiv preprint arXiv:2203.06944*, 2022.

Guennebaud, G., Jacob, B., et al. Eigen: a c++ linear algebra library. *URL http://eigen. tuxfamily. org, Accessed*, 22, 2014.

Hansen, P. C., Nagy, J. G., and O'leary, D. P. *Deblurring images: matrices, spectra, and filtering*. SIAM, 2006.

He, K., Sun, J., and Tang, X. Fast matting using large kernel matting laplacian matrices. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2165–2172. IEEE, 2010.

Hestenes, M. R., Stiefel, E., et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

Horn, B. K. and Schunck, B. G. Determining optical flow. *Artificial intelligence*, 17(1-3):185–203, 1981.

Horvath, C. and Geiger, W. Directable, high-resolution simulation of fire on the gpu. *ACM Transactions on Graphics (TOG)*, 28(3):1–8, 2009.

Hovland, P. and Hückelheim, J. Differentiating through linear solvers. *arXiv preprint arXiv:2404.17039*, 2024.

Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., and Durand, F. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

Jeschke, S., Cline, D., and Wonka, P. A gpu laplacian solver for diffusion curves and poisson image editing. *ACM Transactions on Graphics*, 28(5):1–8, 2009.

Krishnan, D., Fattal, R., and Szeliski, R. Efficient preconditioning of laplacian matrices for computer graphics. *ACM transactions on Graphics (tOG)*, 32(4):1–15, 2013.

Lagaris, I. E., Likas, A., and Fotiadis, D. I. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5): 987–1000, 1998.

Lavin, A. and Gray, S. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4013–4021, 2016.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Levin, A., Lischinski, D., and Weiss, Y. A closed-form solution to natural image matting. *IEEE transactions on pattern analysis and machine intelligence*, 30(2):228–242, 2007.

Levitin, M., Mangoubi, D., and Polterovich, I. *Topics in spectral geometry*, volume 237. American Mathematical Society, 2023.

Li, H., Miao, Y., Khodaei, Z. S., and Aliabadi, M. An architectural analysis of deeponet and a general extension of the physics-informed deeponet model on solving nonlinear parametric partial differential equations. *Neurocomputing*, 611:128675, 2025.

Li, M., Lian, X.-C., Kwok, J. T., and Lu, B.-L. Time and space efficient spectral clustering via column sampling. In *CVPR 2011*, pp. 2297–2304. IEEE, 2011.

Li, X. S. An overview of superlu: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.

Li, Y., Chen, P. Y., Du, T., and Matusik, W. Learning preconditioners for conjugate gradient pde solvers. In *International Conference on Machine Learning*, pp. 19425–19439. PMLR, 2023.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., and Anandkumar, A. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

Li, Z., Zheng, H., Kovachki, N., Jin, D., Chen, H., Liu, B., Azizzadenesheli, K., and Anandkumar, A. Physics-informed neural operator for learning partial differential equations. *ACM/JMS Journal of Data Science*, 1(3):1–27, 2024.

Litany, O., Remez, T., Rodola, E., Bronstein, A., and Bronstein, M. Deep functional maps: Structured prediction for dense shape correspondence. In *Proceedings of the IEEE international conference on computer vision*, pp. 5659–5667, 2017.

Liu, H., Mitchell, N., Aanjaneya, M., and Sifakis, E. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics (TOG)*, 35(6):1–12, 2016.

Liu, Y. and Roosta, F. Minres: from negative curvature detection to monotonicity properties. *SIAM Journal on Optimization*, 32(4):2636–2661, 2022.

Lu, L., Jin, P., and Karniadakis, G. E. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.

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, 2021.

Montoison, A. and Orban, D. Bilq: An iterative method for nonsymmetric linear systems with a quasi-minimum error property. *SIAM Journal on Matrix Analysis and Applications*, 41(3):1145–1166, 2020.

Montoison, A. and Orban, D. Krylov. jl: A julia basket of hand-picked krylov methods. *Journal of Open Source Software*, 8(89):5187, 2023.

Mullen, P., Tong, Y., Alliez, P., and Desbrun, M. Spectral conformal parameterization. In *Computer Graphics Forum*, volume 27, pp. 1487–1494. Wiley Online Library, 2008.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., et al. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5): S602–S626, 2015.

Négiar, G., Mahoney, M. W., and Krishnapriyan, A. S. Learning differentiable solvers for systems with hard constraints. In *ICLR*, 2023.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

Nicolet, B., Jacobson, A., and Jakob, W. Large steps in inverse rendering of geometry. *ACM Transactions on Graphics (TOG)*, 40(6):1–13, 2021.

Nocedal, J. and Wright, S. J. *Numerical optimization*. Springer, 1999.

Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of workshop on machine learning systems (LearningSys) in the thirty-first annual conference on neural information processing systems (NIPS)*, volume 6, 2017.

Ovsjanikov, M., Ben-Chen, M., Solomon, J., Butscher, A., and Guibas, L. Functional maps: a flexible representation of maps between shapes. *ACM Transactions on Graphics (ToG)*, 31(4):1–11, 2012.

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

Paige, C. C. and Saunders, M. A. Algorithm 583: Lsqr: Sparse linear equations and least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 8(2): 195–209, 1982a.

Paige, C. C. and Saunders, M. A. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software (TOMS)*, 8(1): 43–71, 1982b.

Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. *ACM Transactions on Graphics*, 22(3):313–318, 2003.

Perona, P. and Malik, J. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on pattern analysis and machine intelligence*, 12(7):629–639, 1990.

Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. Learning mesh-based simulation with graph networks. In *International conference on learning representations*, 2020.

Praun, E. and Hoppe, H. Spherical parametrization and remeshing. *ACM transactions on graphics (TOG)*, 22(3): 340–349, 2003.

Quarteroni, A. and Valli, A. *Domain decomposition methods for partial differential equations*. Oxford University Press, 1999.

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

Rakowsky, N. The schur complement method as a fast parallel solver for elliptic partial differential equations in oceanography. *Numerical linear algebra with applications*, 6(6):497–510, 1999.

Ren, P., Rao, C., Liu, Y., Ma, Z., Wang, Q., Wang, J.-X., and Sun, H. Physr: Physics-informed deep super-resolution for spatiotemporal data. *Journal of Computational Physics*, 492:112438, 2023.

Reuter, M., Wolter, F.-E., and Peinecke, N. Laplace–beltrami spectra as 'shape-dna'of surfaces and solids. *Computer-Aided Design*, 38(4):342–366, 2006.

Saad, Y. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of computation*, 37 (155):105–126, 1981.

Saad, Y. Practical use of some krylov subspace methods for solving indefinite and nonsymmetric linear systems. *SIAM journal on scientific and statistical computing*, 5 (1):203–228, 1984.

Saad, Y. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2): 461–469, 1993.

Saad, Y. and Schultz, M. H. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

Saad, Y. and Wu, K. Dqgmres: A direct quasi-minimal residual algorithm based on incomplete orthogonalization. *Numerical linear algebra with applications*, 3(4):329–343, 1996.

Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., and Battaglia, P. Learning to simulate complex physics with graph networks. In *International conference on machine learning*, pp. 8459–8468. PMLR, 2020.

Schenk, O. and Gärtner, K. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004.

Sezan, M. I. and Tekalp, A. M. Survey of recent developments in digital image restoration. *Optical Engineering*, 29(5):393–404, 1990.

Shi, J. and Malik, J. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

Sleijpen, G. L. and Fokkema, D. R. Bicgstab (l) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis*, 1(11):2000, 1993.

Solomon, J. *Numerical algorithms: methods for computer vision, machine learning, and graphics*. CRC press, 2015.

Solomon, J., Crane, K., and Vouga, E. Laplace-beltrami: The swiss army knife of geometry processing. In *Symposium on Geometry Processing Graduate School (Cardiff, UK, 2014)*, volume 2, 2014.

Sonneveld, P. Cgs, a fast lanczos-type solver for nonsymmetric linear systems. *SIAM journal on scientific and statistical computing*, 10(1):36–52, 1989.

Strassen, V. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

Trefethen, L. N. and Bau, III, D. Numerical linear algebra, 1997.

Van der Vorst, H. A. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

Vladymyrov, M. and Carreira-Perpinan, M. The variational nystrom method for large-scale spectral problems. In *International Conference on Machine Learning*, pp. 211–220. PMLR, 2016.

Vladymyrov, M., Von Oswald, J., Miller, N. A., and Sandler, M. Efficient linear system solver with transformers. In *AI for Math Workshop@ ICML*, 2024.

Wang, S., Wang, H., and Perdikaris, P. Learning the solution operator of parametric partial differential equations with physics-informed deeponets. *Science advances*, 7(40): eabi8605, 2021.

Wang, Y. and Solomon, J. Intrinsic and extrinsic operators for shape analysis. In *Handbook of numerical analysis*, volume 20, pp. 41–115. Elsevier, 2019.

Wang, Y. and Solomon, J. Fast quasi-harmonic weights for geometric data interpolation. *ACM Transactions on Graphics (TOG)*, 40(4):1–15, 2021.

Wang, Y., Ben-Chen, M., Polterovich, I., and Solomon, J. Steklov spectral geometry for extrinsic shape analysis. *ACM Transactions on Graphics (TOG)*, 38(1):1–21, 2018.

Wang, Y., Kim, V., Bronstein, M., and Solomon, J. Learning geometric operators on meshes. In *Representation Learning on Graphs and Manifolds 2019 (ICLR workshop)*, 2019.

Wang, Y., Guo, M., and Solomon, J. Variational quasi-harmonic maps for computing diffeomorphisms. *ACM Transactions on Graphics (TOG)*, 42(4):1–26, 2023.

Winograd, S. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

Yi, L., Su, H., Guo, X., and Guibas, L. J. Syncspeccnn: Synchronized spectral cnn for 3d shape segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2282–2290, 2017.

Younes, L. *Shapes and diffeomorphisms*, volume 171. Springer, 2010.

Zhu, X., Ghahramani, Z., and Lafferty, J. D. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pp. 912–919, 2003.

Table of Contents

A. Extended Discussions on Related Work

We focus on direct solvers—exact solutions that are computed up to numerical errors accumulated due to machine precisions, rather than iterative ones that only yield approximate solutions under a given error bound. The performance of direct solvers is more stable, whereas iterative solvers can greatly slow down for difficult $\mathbf{A}$ and may fail to converge if used without care. In contrast to direct solvers, iterative methods such as multigrid solvers only yield solutions within a prescribed error tolerance. Even so, they are still much slower than ours in scenarios that they are specifically designed for, and many orders of magnitude slower on challenging examples. Direct solvers are praised for their robustness and reliability, while being much slower. We implement a direct solver to achieve both accuracy/reliability and speed, combining the best of both worlds.

Despite a well-studied problem, general-purpose sparse direct solvers, introduced decades ago and having remained stable since, are not specifically designed for today's vision and learning applications. We focus on problems that fit on a single GPU, that nonetheless encompass common image sizes, avoiding considerations in cross-device communication that arise in larger scale problems in scientific computing. In addition, advances in GPUs sparked by deep learning shift the best practice towards algorithms exploiting parallelisms and that modern GPU BLAS kernels are extremely good at solving a large number of small problems. We observe that the high throughput offered by modern GPUs has been under-exploited in direct sparse solvers at the common scale of problems in vision.

Sparse linear solvers in vision & graphics. In computer vision and graphics, previous developments have focused on iterative solvers, and we are not aware of a prior effort to design *direct* solvers for Laplacian-like systems—a missing setting we address. A major component of contributions in optimization-based methods is to develop problem-specific iterative solvers. Our approach can bypass the need for iterative schemes in many tasks.

Hierarchical approaches or multigrid methods are popular especially when $\mathbf{A}$ arises from elliptic PDEs (Allaire, 2007). However, in situations where the solution is non-smooth, such as the Helmholtz equation, common preconditioned iterative solvers can work very poorly (Ernst & Gander, 2011). Incomplete LU factorizations are another family of preconditioning schemes. For positive semi-definite systems, incomplete Cholesky factorization can be applied (Chen et al., 2021a). Depending on the properties of $\mathbf{A}$, popular choices of iterative methods include Jacobi methods, Gauss–Seidel, Krylov subspace including (preconditioned) conjugate gradients (PCG) for positive-definite $\mathbf{A}$, and generalized minimal residual (GMRES) for un-symmetric $\mathbf{A}$ (Solomon, 2015).

Bolz et al. (2003) pioneer the application of GPUs to accelerate iterative solvers for computer graphics applications. Barron & Poole (2016) apply preconditioned conjugate gradients for fast bilateral filtering; their solvers operate in a lower dimensional bilateral space, unlike our method which aims for pixel-space solvers. Krishnan et al. (2013) use a Schur complement formula to construct a Laplacian preconditioning scheme. For very small-scale problems, it is well-known that dense direct solvers can be faster (Bro-Nielsen & Cotin, 1996). Problem-specific iterative solvers have been designed for image processing (Jeschke et al., 2009), physical simulation (Horvath & Geiger, 2009; Liu et al., 2016), and geometry processing (Krishnan et al., 2013). Direct solvers such as CHOLMOD have been used in differentiable rendering and inverse geometry design (Nicolet et al., 2021).

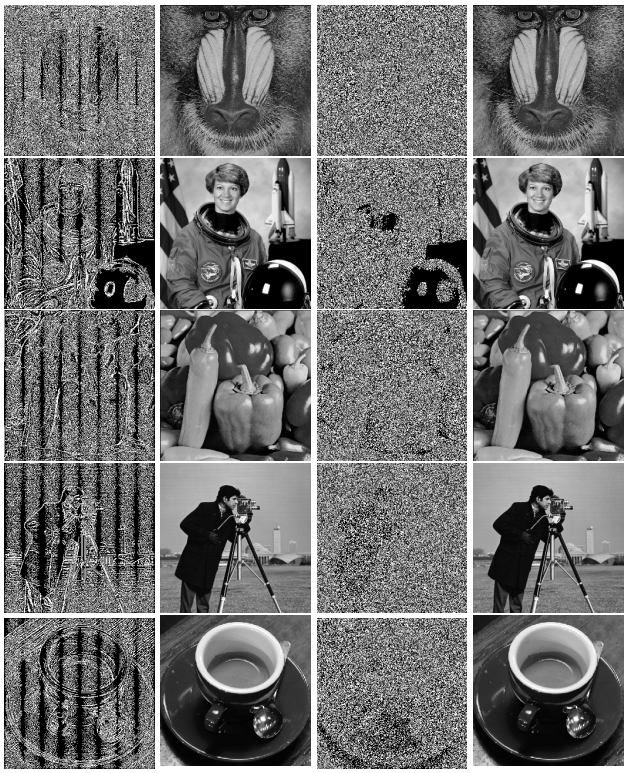
Li et al. (2023) apply a preconditioner learned from data, and Arisaka & Li (2023) develop learning-based acceleration of iterative methods under a meta-learning framework.

Domain decomposition. Our approach conceptually follows the divide-and-conquer strategy, dating back to the celebrated Schwarz alternating method in domain decomposition (Quarteroni & Valli, 1999). The Dirichlet-to-Neumann (DtN) operator is a standard object used to "glue" solutions between subdomains (Quarteroni & Valli, 1999). Typically, it is implicitly maintained through its matrix-vector product, realized by solving sparse systems on subdomains (Liu et al., 2016). In contrast, we make a different design choice to explicitly maintain the discrete DtNs as dense matrices.

Sparse solvers: missing in differentiable programming. Differentiable sparse linear or eigen solvers have been as popular requested feature in deep learning packages: please refer to §F.3 for a list of examples of community requests and discussions on sparse linear or eigen solvers. In a related but different effort, recent works (Grementieri & Galeone, 2022; Négiar et al., 2023; Vladymyrov et al., 2024) learn to solve linear systems, for which our method can serve as a strong baseline or an efficient training engine. Our analysis and discussion of FEM in §2 suggest that the learning-based PDE solvers and learning-based linear solvers are in fact attacking the *same* underlying problem—yet they are currently studied as separate fields with little cross-referencing.

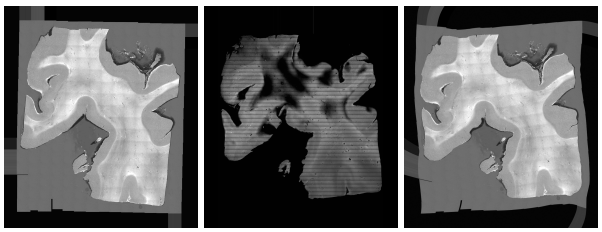
B. Visualization, Applications, and Experiments

Note that our goal in this section is *not* to compete on every task with state-of-the-arts, but to pick classic methods already requiring a solution to some Laplacian-like systems, collecting sparse matrices $\mathbf{A}$ covering a broad array of cases to test our methods with. Details on the timing are provided in §B.6.



Input $\mathbf{b_1}$: convolved image. | Recovered image: $\mathbf{x_1} = \mathbf{A}_1^{-1}\mathbf{b_1}$. | Input $\mathbf{b_2}$: convolved image. | Recovered image: $\mathbf{x_2} = \mathbf{A}_2^{-1}\mathbf{b_2}$.

Figure 6. Our solver is applied to image deconvolution with $\mathbf{A}$. We test with two cases of spatially varying kernels: $a_1^{(x,y)} = \sin(8\pi x)^2\mathbf{K}_1 + y^3\mathbf{K}_2$, $\mathbf{K}_1, \mathbf{K}_2 \in \mathbb{R}^{3\times 3}$; $a_2^{(x,y)}$ is randomly drawn at (x, y). where $\mathbf{K}_1, \mathbf{K}_2$ are defined in Figure 9.



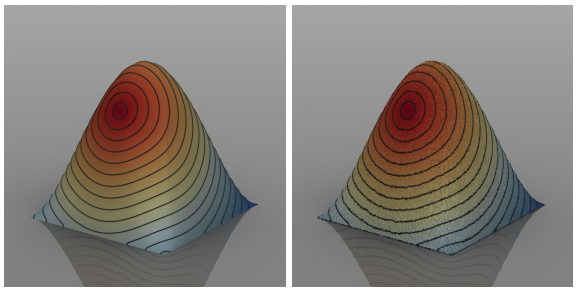
(a) Moving image. (b) Target image. (c) Moved (ours).

Figure 7. For computing diffeomorphisms and image registration, previous optimization-based methods can take a few minutes on high-res images. Switching the backend to our solver immediately reduces the runtime to a few seconds.

B.1. Newton's method and interactive graphics

As an immediate benefit, our fast solver potentially unlocks the capacity of Newton's method (Nocedal & Wright, 1999) relying on Hessian solves for applications in interactive physics, computer vision, and graphics.

When $\mathbf{A} = \mathbf{H}$ is the Hessian and $\mathbf{b} = \mathbf{g}$ is the gradient of some energy of a function on the image domain, the solution



The kernel on a smooth surface. The kernel on a noise surface.

Figure 8. Our solver accelerates by orders of magnitude the computation of geodesic distances. The heat kernel computed by our method is shown (in log scale). This is a task where iterative methods struggle: often the kernel solution must be within a tolerance of 10^{-10} for accurate geodesic distances (Crane et al., 2017).

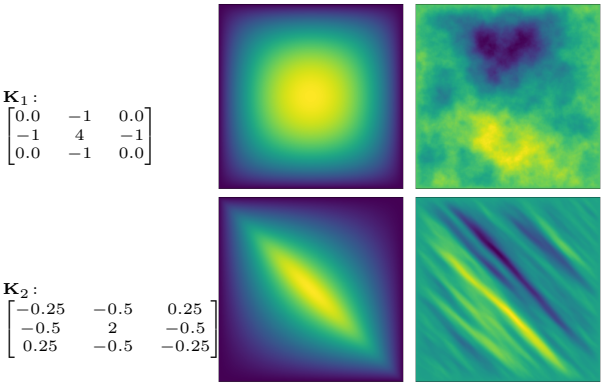
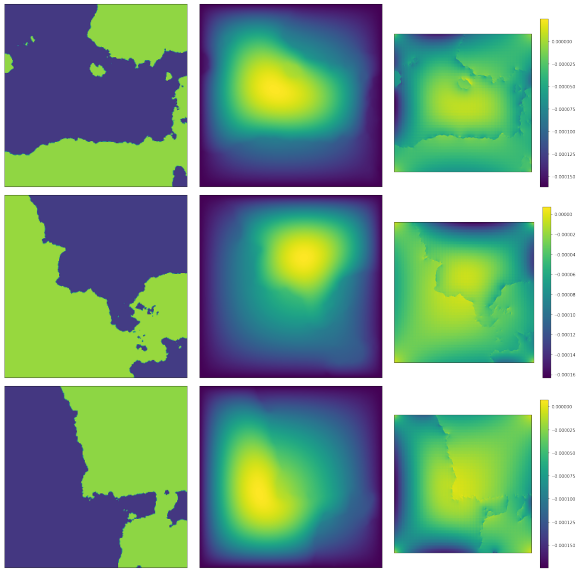


Figure 9. Our solutions to isotropic and anisotropic Poisson. For fair comparison, 0s in $\mathbf{K}_1$ are treated as non-zero entries with value 0.0 in all solvers.

$\mathbf{H}^{-1}\mathbf{g}$ yields the descent direction in Newton's method. It is very common that $\mathbf{H}$ and $\mathbf{g}$ come from an energy in which pixels interact locally with their neighbors, rather than all other pixels in the image. In this setting, the Hessian $\mathbf{H}$ has the same sparsity as the Laplacian matrix to apply our method; in fact, for some physics-based energy, $\mathbf{H}$ equals exactly to an anisotropic Laplacian, $\mathbf{H} \leftarrow \mathbf{L}$. In this case, the sparse linear solver is the sole bottleneck in applying Newton's method: constructing entries in $\mathbf{H}, \mathbf{g}$ involves only per-element/pixel computation, which can happen in parallel.

400× faster Hessian solvers for minimal surfaces. As an application, in Figure 11, we apply our solver to Newton's method to compute the surface that minimizes the area with a fixed boundary—the minimal surface problem. The surface is parameterized as a height field $z = u(x, y)$, and the goal is to find the one minimizing the total area, which is a nonlinear objective. Minimal surface area can be used as a



Coefficients $\mathbf{c}$: $\mathbf{A} \leftarrow \mathbf{G}^\intercal \mathrm{diag}(\mathbf{c}, \mathbf{c}) \mathbf{G}$ | Solution $\mathbf{x}$: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ | Difference with the "ground truth"

Figure 10. Our solvers realize the FEM coefficient-to-solution map.

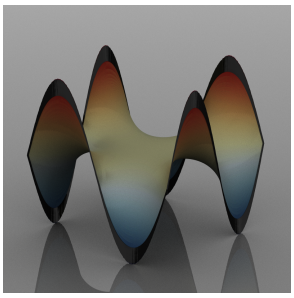


Figure 11. The minimal surface problem searches for the surface whose area is minimized under a prescribed boundary, having been a prior in, e.g. shape completion. The minimal surface computed by Newton's method, which is accelerated by $400\times$ with our Hessian solver compared to the SciPy backend.

prior in shape completion. Newton's method sequentially solves $\mathbf{H}^{-1}\mathbf{g}$. Thanks to the second-order convergence, using Newton's method with our solver as the backend, it converges within 20 iterations, while gradient descent using L-BFGS takes more than 1000 iterations to get to the same accuracy. Replacing SciPy in Newton's method with our solver immediately yields a speedup of $400\times$.

Unlock Newton's method? Despite superiority in the convergence rate, we find that Newton's method has very limited applications in computer vision. Perhaps it is partially because of the expensive Hessian solves, a barrier that our method removes.

This phenomenon is even more prominent in interactive computer graphics. In the seminal work Projective Dynamics (PD) (Bouaziz et al., 2014), as the central assumption when it comes to fast simulators, it is made explicit that Newton's method requires orders-of-magnitude fewer iterations while solving the Hessian in each iteration is expensive, making it less competitive.

The assumption that linear solve is expensive necessitates the development of alternative methods. A common pattern of the alternative fast methods is to reuse the factorization of the Laplacian matrix and run tens of thousands of such cheaper iterations, while Newton's method can converge in tens of iterations. This assumption also governs the best practice in differentiable simulation, e.g. in robot learning (Du et al., 2021).

Our method challenges this long-standing assumption, potentially unlocks the capacity of Newton's method, and eases the design of efficient algorithms.

B.2. Physical simulation and shape optimization

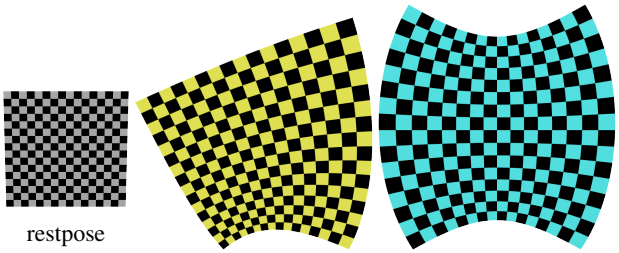


Figure 12. Some shape deformation produced by our method.

Physical simulation already can benefit from our method: Laplace equation governs physical phenomena in electromagnetism, gravity, heat diffusion, fluid dynamics, and shape deformation. Incorporating the coefficients $\mathbf{C}(\mathbf{x})$ allows pushing forward the PDEs in some irregular geometric domain onto a canonical domain, and accounts for the distortion induced by the geometric mapping—like how we compute distances on surfaces in §4. In addition, a straightforward way to utilize our solver is to write down some quadratic objective whose minimizer yields a linear system solve. The quadratic objective may come from the linearization of some nonlinear energy, such as the Hessian. Our solver currently only supports one variable per pixel; we plan to generalize it to multiple variables per pixel, e.g. two variables can represent the x, y coordinates of a deformation. This can be done by generalizing the "DtN matrix" to be twice as large and viewing our method simply as a block Gaussian elimination. Instead, we present a simple trick to directly leverage our current solver in the setting of two variables per pixel. We put both the x-,y-components in a complex number $u_z = u_x + \mathrm{i}u_y$. For instance, we can squeeze the $2n \times 2n$ block matrix arising in a conformal deformation energy (Mullen et al., 2008)

into a single $n \times n$ complex-valued Hermitian matrix that $\mathbf{A} = \text{conj}(\mathbf{A}^\intercal)$. Without modifying the code, our solver applies to complex-valued linear systems. Figure 12 shows the deformation obtained in this way.

B.3. Image matting and segmentation

In image matting (Grady et al., 2005; He et al., 2010; Levin et al., 2007), the matrix $\mathbf{A}$ encodes both an affinity matrix $\mathbf{L}$, a Laplacian with values decided by pixels' color, and a soft constraint term. It typically minimizes for an alpha function $\mathbf{x}$ that is smooth as measured by $\mathbf{L}$, and constrained to be 0 or 1 (stored in $\mathbf{k}$) in the foreground or background, respectively. $\mathbf{R}$ is a binary matrix selecting those constrained pixels.

$$\mathbf{x}^\intercal \mathbf{L} \mathbf{x} + \lambda \|\mathbf{R}^\intercal \mathbf{x} - \mathbf{k}\|^2 + \text{constant} \tag{14}$$

In this case, $\mathbf{A} := \mathbf{L} + \lambda \mathbf{R}\mathbf{R}^\intercal, \mathbf{b} := \lambda \mathbf{R}\mathbf{k}$. Most of the change in $\mathbf{x}$ is restricted to the band between two layers of the user-provided masks (often called the trimap). We validate using the linear system obtained by PyMatting on the examples shown in Figure 13. We use the image matting framework PyMatting (Germer et al., 2020) and its implementation of the random walk Laplacian (Grady et al., 2005) with a small 3×3 stencil as $\mathbf{L}$. The error of our method in double precision is on the scale of 10^{-16}, and is slightly smaller than SciPy in every image.

We choose to test it with our method for a particular reason: the difficulty of solving concentrates at the band, not evenly distributed across the image. The problem can be converted to an easy task for geometry-aware PDEs if solving only within the band region with Dirichlet condition at known pixels. But (14) represents a common way to incorporate the constraint softly: add a penalty term weighted by a very large coefficient λ. It makes some entries $\mathbf{A}_{ij}$ at constrained pixels very large compared to unconstrained pixels. Uncareful direct solvers potentially follow a numerically unstable order when performing Gaussian elimination.

A purpose of this validation is to emphasize that viewed as a Gaussian elimination, our method cancels variables in an order that *is dependent* on the entries in $\mathbf{A}$, avoiding many numerical issues. A common misperception of our method is that it performs a Gaussian elimination that is blind to data $\mathbf{A}$. This is not the case. Our method does prescribes some block structures, which are independent of $\mathbf{A}$. However, within each block, the canceling order of nodes are still permuted, leading to robust numerical behaviors. This is a critical detail hidden in how to compute matrix inversions—we use "torch.linalg.inv" function which is backended by cuBLAS. Thus, when inverting a batch of dense matrices, it performs careful numerical schemes on each matrix individually to avoid usual pitfalls in numerical operations.

Figure 13. The input image is used to generate an affinity matrix $\mathbf{A}$, the trimap for $\mathbf{A}$, $\mathbf{b}$, and the output alpha is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

B.4. Fast eigen solver for spectral segmentation

Often it is the low-frequency components of the Laplacian that are relevant for practical applications, rather than the high-frequency ones: $\mathbf{x} \leftarrow \mathbf{A}^{-1}\mathbf{x}$ magnifies the low-frequency components while $\mathbf{x} \leftarrow \mathbf{A}\mathbf{x}$ does the opposite. This observation makes our method relevant for many vision tasks.

In many applications, matrix $\mathbf{A}$ comes from pixel values and can be considered as a surrogate representation of an image, but it is not clear what constitutes a relevant $\mathbf{b}$. Then, the eigenvalue problem $\mathbf{A}\mathbf{x} = \lambda \mathbf{x}$ becomes the right computational model to extract information from $\mathbf{A}$.

The design of fast eigen solvers can be reduced to that of linear solvers, immediately benefiting from our method. Instead of designing a standalone hierarchical eigen solver, our fast linear solver can be used as the inverse iteration in implementing a fast eigen solver. Our linear solver can serve as the atomic linear operator $\mathbf{x} \leftarrow \mathbf{A}^{-1}\mathbf{x}$ for iterative eigen solvers. For example, our solver can immediately speed up the spectral image segmentation method, normalized cuts (Shi & Malik, 2000). In this case, we choose the matrix $\mathbf{A}$ as $(\mathbf{I} - \hat{\mathbf{L}})$, where $\mathbf{L}$ is the matting Laplacian with the 3×3 kernel in §B.3 that becomes $\hat{\mathbf{L}}$ after having its rows

and columns normalized, following Shi & Malik (2000).

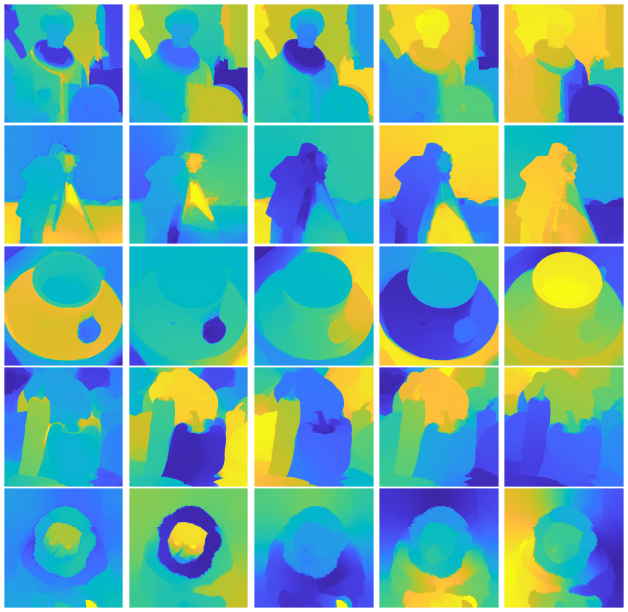


Figure 14. Our linear solver, when being used to realize the inverse iteration in iterative eigen solvers, is immediately transferred into an orders-of-magnitude faster eigen solver—455× faster than MATLAB's sparse eigen solver for 513×513 images, with applications to spectral image segmentation (Shi & Malik, 2000). The top 5 eigenfunctions of the astronaut, cameraman, coffee, pepper, and lemur images, computed by 20 calls to $\mathbf{A}^{-1}\mathbf{x}$, are shown.

For examples shown in Figure 14, we found iterative eigen solvers require only $13 \sim 20$ *calls* to $\mathbf{A}^{-1}\mathbf{x}$, to compute the top 6 eigenvectors within the tolerance of $10^{-3} \sim 10^{-2}$. For an image of size 513^2, it takes 25 seconds to solve the top-6 eigenvalue problem for MATLAB, on which the code of Shi & Malik (2000) relies, while it takes as short as 55 milliseconds for our method, yielding an acceleration of **455**×. Our method is many orders of magnitude faster compared to the Nyström methods that only approximate the spectrum (Vladymyrov & Carreira-Perpinan, 2016; Li et al., 2011; Belongie et al., 2002). For our method, the eigenvalue problem requires 1 step of numerical factorization (11 ms for our method), and tens of the back substitution steps.

Recall that the back substitution can be much faster than the numerical factorization since it skips calculating any new left-hand sides. Thus, our solver can immediately accelerate by orders of magnitude the spectral segmentation in the *full* pixel space. This is different from previous methods like Barron & Poole (2016), where a major source of speedup comes from reducing the computation to a lower-dimensional space. Future work might combine our approach with the dimension-reduction strategy for further improvement, and leverage our solver as a parameter-free differentiable layer similarly to Barron & Poole (2016).

B.5. PDEs on domains with a non-disk topology

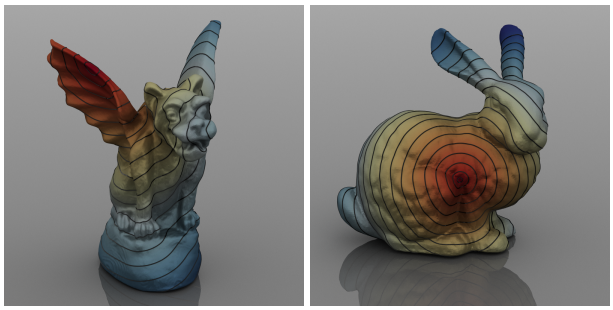


Figure 15. Our solver applies to shape analysis and geometry processing for curved surfaces as well. The heat kernel computed on two surfaces with the spherical topology are shown, accelerated by orders of magnitude using our method. Using a corner reflective boundary condition and an octahedral parameterization (Praun & Hoppe, 2003), our method can solve a geometric PDE that is defined on a surface with the spherical topology (§B.5).

We can solve PDEs on a surface with a non-disk topology, by leveraging the surface parameterization techniques, such as the octahedral parameterization (Praun & Hoppe, 2003) to cut and map a spherical topology surface onto an image domain. Our method can easily incorporate a midpoint reflective boundary condition induced by the octahedral parameterization (Praun & Hoppe, 2003), by modifying the last step to fill in the Dirichlet boundary condition in the Neumann solver (§C.2). It allows us to process surfaces with spherical parameterization. Figure 15 shows the heat kernel computed on curved surfaces that have the spherical topology using our method.

Adding the ability to handle curved surfaces also makes our solver applicable to shape analysis and geometric deep learning, such as spectral methods for shape correspondence (Ovsjanikov et al., 2012; Litany et al., 2017; Yi et al., 2017), Laplacian eigenvalues for shape classification (Reuter et al., 2006), and many other applications (Solomon et al., 2014).

B.6. Timing details

We provide details on the timing comparison. Again, unlike the baseline solvers that are already highly optimized with official support from Nvidia, our method is simply prototyped in PyTorch with great potential for further acceleration. We have used torch.compile to reduce the overhead of PyTorch, which makes PyTorch code 1.2 to 1.8× faster. In addition to Table 2, Table 8 and 9 report the average runtime (in milliseconds) and the error tolerance to solve an isotropic or anisotropic system with the Dirichlet condition, respectively. The runtime of direct solvers including CUDA and our method are not affected by the anisotropic coefficients that make the problem much harder for iterative

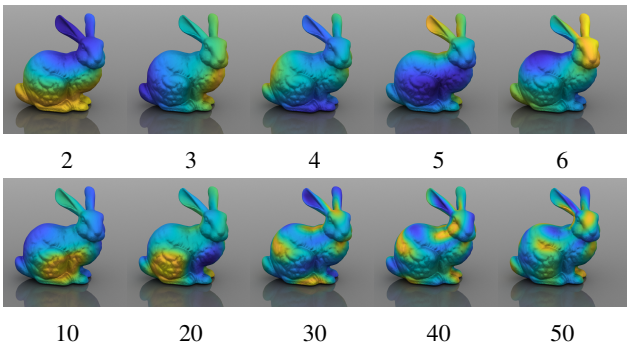


Figure 16. Our linear solver with the midpoint reflective boundary condition can be used as the reverse iteration for computing eigen functions of a surface with the the octahedral parameterization (Praun & Hoppe, 2003). Some Laplace-Beltrami eigen functions of the Stanford bunny are shown. Eigen functions of the Laplace-Beltrami operator are foundational for manifold data analysis and geometric deep learning.

solvers as demonstrated in Table 2. SciPy does slow down for the anisotropic coefficients, but is still comparable to the isotropic case. In all experiments, we do not treat any matrix as symmetric even though they are.

Table 9 and 10 compare the average runtime when solving the Dirichlet or Neumann boundary condition. Since we reuse the Dirichlet solver to the Neumann problem, it can take slightly more time for the Neumann problem, but still comparable to the Dirichlet case. Future work may implement §C.2 for a better Neumann solver.

Surprisingly, the error tolerance of our method in float 64 is consistently smaller than that of SuperLU from SciPy when solving elliptic PDEs, though our method does not optimize for accuracy like the SuperLU method does. We had suspected that the more careful pivoting in SuperLU might help to achieve a better precision, which nonetheless is not observed for solving Laplacian systems.

Table 11 reports the runtime when solving the matting system. Interestingly, SuperLU from SciPy significantly slows down in the matting example, due to the unbalanced concentration in spatial difficulty, as we have discussed in §B.3. However, our method (float 64) still has an error smaller than SciPy.

B.7. Complexity analysis

To emphasize, unlike the standard numerical analysis, in our setting, we are less concerned about the asymptotic complexity of the numerical algorithm, but rather the actual runtime on common image sizes. These are very small-scale problems by the standard of scientific computing, so having a small constant in the complexity can be equally or even more important than the asymptotic rate. The constant

Table 4. The experiments same to the ones in Table 3 but for 1025×1025 images.

	Lap 1e-2	Lap 1e-4	Hel 1e-2	Hel 1.0	R-Decon	
Ours-64	175	175	175	175	175	
Ours-32	36.4	36.3	36.5	36.4	36.4	
SciPy-LU	16001	15882	15899	16008	23175	
CUDSS-LDU	4566	4592	5095	4554	4687	
(s) CG	92.1	682	2280	21473	NaN	$\times k$
(s) MINRES	113	1292	5037	33864	NaN	
LSMR	8244	1042012	749481	600585	1871 (*)	
LSQR	4520	NaN	NaN	118307 (*)	827 (*)	
biCGstab	356	823	NaN	NaN	NaN	
CGS	95	538	NaN	NaN	NaN	
DIOM	2263	17549	39340	347352	NaN	
FOM	5934	866361	667058	NaN	NaN	
QMR	147	4268	2444	NaN	NaN	
BILQ	150	4228	2694	NaN	NaN	
GMRES	1028	279031	285348	NaN	NaN	
DQGMRES	279	13018	5172	49156	NaN	
FGMRES	1051	40096	NaN	NaN	NaN	

Table 5. The error as measured by relative tolerance corresponding to Table 4. Note that random deconvolution (R-Decon) may not be a very well-defined task to evaluate accuracy, since the matrix $\mathbf{A}$ can be close to singular due to randomness.

Error	Lap 1e-2	Lap 1e-4	Hel 1e-2	Hel 1.0	R-Decon
Ours-32	1.52E-05	1.71E-05	1.94E-04	2.30E-03	NaN
Ours-64	2.71E-14	3.03E-14	5.45E-13	4.30E-12	3.39E-08
SciPy-LU	4.06E-14	4.19E-14	7.22E-13	1.03E-12	2.06E-15
CUDSS-LDU	2.97E-14	3.04E-14	1.56E-13	2.86E-12	6.24E-12

Table 6. The error as measured by relative tolerance corresponding to Table 3. Note that random deconvolution (R-Decon) may not be a very well-defined task to evaluate accuracy, since the matrix $\mathbf{A}$ can be close to singular due to randomness.

Error	Lap 1e-2	Lap 1e-4	Hel 1e-2	Hel 1.0	R-Decon
Ours-32	1.52E-05	1.71E-05	1.94E-04	2.30E-03	NaN
Ours-64	2.71E-14	3.03E-14	5.45E-13	4.30E-12	3.39E-08
SciPy-LU	4.06E-14	4.19E-14	7.22E-13	1.03E-12	2.06E-15
CUDSS-LDU	2.97E-14	3.04E-14	1.56E-13	2.86E-12	6.24E-12

Table 7. The back substitution time for our method to solve a Neumann problem. Surprisingly, the speed for a 1025^2 image is similar to that of a 257^2 image, suggesting that there are probably a large room for low-level optimization for the scale of 257^2.

	Resolution	Time (milliseconds)
Ours (float-64)	257	2.56
Ours (float-64)	513	3.01
Ours (float-64)	1025	3.13

and thus the best algorithm are hardware dependent, so the actual runtime as we have reported in the paper should be the only criterion, in addition to which we still provide an asymptotic complexity analysis.

The time complexity of our algorithm is dominated by the parallel matrix inversion in the Schur steps:

$$\begin{aligned}\sum_{j=1}^{\log n}((2^j)^{1/2})^3 =& n^{3/2} + \frac{1}{2^{1.5}}n^{3/2} + \left(\frac{1}{2^{1.5}}\right)^2 n^{3/2} + \cdots \\ =& \mathcal{O}(n^{1.5})\end{aligned} \tag{15}$$

Table 8. Average runtime (in milliseconds) to solve an isotropic Laplacian with Dirichlet condition.

	time				error tolerance		
Example	CUDA	SciPy	ours	ours-64	SciPy	ours-64	ours
2049^2	21297	138219	158.5	OOM	1.92e-14	1.07e-14	6.86e-6
1025^2	4663	15526	37.4	172.9	1.27e-14	7.56e-15	5.28e-6
513^2	1053	2223	10.9	31.9	8.52e-15	5.43e-15	3.87e-6
257^2	235	334	5.11	7.55	5.49e-15	3.79e-15	2.53e-6

Table 9. Average runtime (in milliseconds) to solve an anisotropic system with Dirichlet condition.

	time				error tolerance		
Example	CUDA	SciPy	ours	ours-64	SciPy	ours-64	ours
2049^2	21354	143100	159.3	OOM	3.43e-14	2.03e-14	1.07e-5
1025^2	4710	16512	37.5	171.3	2.54e-14	1.57e-14	7.97e-6
513^2	1036	2051	10.9	31.77	1.36e-14	8.85e-15	4.44e-6
257^2	234	355	5.82	7.43	2.24e-16	1.57e-16	2.45e-6

Table 10. Average runtime (in milliseconds) to solve an anisotropic system with Neumann condition.

	time				error tolerance		
Example	CUDA	SciPy	ours	ours-64	SciPy	ours-64	ours
2049^2	21396	186231	171.6	OOM	2.41e-15	1.47e-15	6.73e-7
1025^2	4779	17511	42.2	187.4	2.40e-15	1.47e-15	6.74e-7
513^2	1038	2174	12.2	35.4	2.37e-15	1.47e-15	6.61e-7
257^2	235	389	6.47	8.82	2.28e-15	1.47e-15	6.56e-7

Table 11. Average runtime (in milliseconds) to solve the matting system with Neumann condition.

	time			error tolerance	
Example	CUDA	SciPy	ours-64	SciPy	ours-64
2049^2	22601	244125	OOM	2.89e-16	1.80e-16
1025^2	4721	16969	187.6	2.73e-16	1.77e-16
513^2	1046	6223	35.8	2.50e-16	1.65e-16
257^2	232	1045	8.79	2.24e-16	1.57e-16

where we have assumed the cubic cost of inverting dense matrices and that the parallelized inversions take the same time as inverting one matrix of the same size.

C. Method: Extended Discussions

In summary, our "Schwarz–Schur involution" recursively merges subdomains and eliminates nodes/pixels at the interface between subdomains, and updates the maintained system matrix by applying a Schur complement formula.

Following the nested dissection hierarchy (George, 1973), we recursively divide the domain into two subdomains and reduce the degrees of freedom: nodes at the interface between subdomains have duplicated copies in each subdomain. This yields a quad-tree in 2D: as shown in Figure 17, recursively applying this step reduces the number of subdomains following the sequence: $(512, 512) \to (256, 256) \to (128, 128) \to (64, 64) ... \to (1, 1)$. This procedure visually

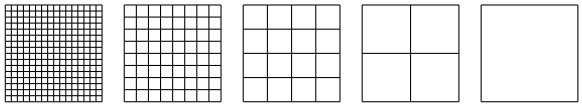


Figure 17. After the Schwarz step that eliminates interior nodes/pixels for each of the subdomains, the domain becomes a "wire-frame" that is progressively simplified in the Schur steps. The Schur step recursively collapses adjacent hollow subdomains by applying a Schur formula to merge two DtN matrices.

corresponds to Figure 17 but in the reverse order.

C.1. Dirichlet-to-Neumann factorization

We call the Schwarz and Schur steps as the *Dirichlet-to-Neumann factorization*, a numerical factorization procedure similar to the LU factorization. The role of the dense matrices we saved in $\boldsymbol{\alpha}^{(j)}$ and the overall hierarchy is analogous to the L and U factors in standard LU factorization (Davis, 2006). In the Schur step, small patches are recursively merged, during which the new left-hand side—the DtN matrices for the merged subdomains are constructed by inverting sub-block of the DtN matrices $\mathbf{P}, \mathbf{Q}$ at subdomains using the equations in §3.2.2; every matrix there will be saved for later usage—in the back substitution stage to apply to the right-hand side.

The output in the last Schur step is the tensor $\boldsymbol{\alpha}^{(k)}$ of size $(1, 1, b, b)$, where again b is the number of pixels on the border of the image domain. Denoting $\boldsymbol{\alpha}^{(k)}[0, 0, :, :]$ as the dense matrix $\mathbf{D} \in \mathbb{R}^{b\times b}$, the original linear system $\mathbf{Ax} = \mathbf{b}$ at the last level of the hierarchy becomes $\mathbf{Du} = \mathbf{d}$, where $\mathbf{D} \in \mathbb{R}^{b\times b}$ and $\mathbf{d} \in \mathbb{R}^{b\times 1}$.

Back substitution is the standard step to fill in solutions for all rows of $\mathbf{x}$. For Dirichlet boundary condition, we directly fill in rows that correspond to boundary pixels, $\mathbf{x}|_{1:b} \leftarrow \mathbf{g}$. Then values at the interface can be recursively backed filled by solving (11), so the values in $\mathbf{x}$ at the wire-frame of the image domains can be obtained. Finally, we recover the values in $\mathbf{x}$ that correspond to the interior 3×3 block of each patch.

C.2. Solvers for Neumann boundary condition

For Neumann boundary condition, there are multiple ways to directly apply our Dirichlet solvers: 1) solve some DtN system for the final domain to fill in the missing Dirichlet data and call the Dirichlet solver (§C.2.1); 2) the actual option we use in the paper for better performance: modify the first Schwarz step to also eliminate pixels at the boundary of the domain (§C.2.3).

We use the Dirichlet boundary value problem to explain our method as it arises naturally: our method is a hierarchical construction of DtN matrices that are directly applicable under the known Dirichlet condition. The linear condition (11) that we enforced at the interface says that the Dirichlet data should be chosen so that the resulting Neumann data on either side of the interface must match each other—flux into the interface should equal to flux outward on the other side of the interface.

Most tasks in computer vision have the natural (Neumann) boundary condition. The Neumann boundary value problem can be solved in a similar fashion. In principle, one can recursively work with the *Neumann-to-Dirichlet* matrix instead of the Dirichlet-to-Neumann matrix, which are (pseudo) inverses of each other. Then we can fill in the missing Neumann boundary condition in a hierarchical fashion. Instead, we present a simple solution that directly leverages the Dirichlet solver. The goal is to minimize effort and reuse Dirichlet solvers.

C.2.1. Solution 1

The original linear system $\mathbf{Ax} = \mathbf{b}$ at the last level of the hierarchy becomes $\mathbf{Du} = \mathbf{d}$, where $\mathbf{D} \in \mathbb{R}^{b\times b}$ and $\mathbf{d} \in \mathbb{R}^{b\times 1}$. From the Gaussian elimination viewpoint, the DtN matrix $\mathbf{D}$ is nothing more than the linear system matrix with the interior nodes in the image eliminated. Thus, the Dirichlet boundary condition that is missing in order to apply the Dirichlet solver for the Neumann problem can be simply found by solving $\mathbf{D}^{-1}\mathbf{d}$.

Indeed, this immediately gives a Neumann solver. However, there is one drawback with this strategy of reusing the Dirichlet solver in a Neumann problem. The system matrix $\mathbf{D}$ is $16\times$ as large as the largest matrix that one has to solve in the Dirichlet problem. ($\mathbf{D}$ is $4\times$ as large in rows and columns so a total of 16). This makes the Neumann solver for large-scale problem noticeably slower than the Dirichlet solver, even though in principle they should be equally difficult.

C.2.2. Solution 2

As shown in Figure 18, to avoid the need to invert a larger matrix, we design a pre-processing step to further eliminate nodes at the beginning.

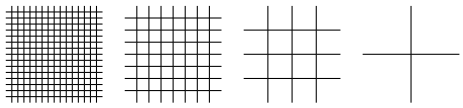


Figure 18. The wire-frame structure resulted in the Neumann elimination, obtained by modifying the Dirichlet elimination in Figure 17.

In general, it is better to eliminate nodes as early as possible, since that can happen in parallel, rather than deferring it to the final stage to increase the size of the large dense matrix. In some sense, the Neumann boundary value problem is easier than the Dirichlet counterpart, since the former allows node elimination at an earlier stage.

C.2.3. Solution 3

While Solution 2 should be the theoretically optimal way in this paper, it requires implementing a new edge collapse procedure and handling boundary patches differently. In order to reuse the Dirichlet solver like Solution 1 does, we propose a further simplification: at the boundary of the im-

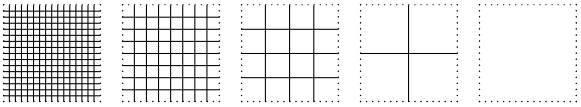


Figure 19. The final hierarchy for Neumann solver.

age domain we only eliminate wire-frames without two end points. The nodes at the boundary of the image domain are still kept, though any rows and columns corresponding to them in the left-hand side are zeros, and any rows corresponding to them in the right-hand side are also zeros. Then, we can reuse the same equations in §3.2.2. This solution has the advantage that at the last level, the system to solve will be smaller than $\mathbf{D}$: while $\mathbf{D}$ has the same size, we know that most rows and cols in $\mathbf{D}$ are zeros and can be thrown away. So we only need to solve for, e.g., $\mathbf{D}^{-1}_{1:4:b,1:4,b}\mathbf{d}_{1:4:b}$ when patch size is 4×4 enlarged to 5×5 so $(\cdot)_{1:4:b}$ selects one from every 4 pixels along the border. The new system matrix $\mathbf{D}^{-1}_{1:4:b,1:4,b} \in \mathbb{R}^{b/4 \times b/4}$ records the interplay among the scatter points on the border of the image domain.

C.3. Differentiable linear solvers and derivatives

Differentiable linear solvers are useful in many scientific applications (Hovland & Hückelheim, 2024). Following notations of Wang & Solomon (2021), let us derive the closed-form formulas for the partial derivatives $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$: we need to obtain $\partial \mathbf{x}/\partial \mathbf{b}$ and $\partial \mathbf{x}/\partial \mathbf{A}$.

It is convenient to introduce the notation to flatten the matrix $\mathbf{A}$ into a vector:

$$\mathbf{a} := \textsc{Flatten}(\mathbf{A}) \in \mathbb{R}^{n^{(nz)}} \tag{16}$$

which stores the the nonzero entries of $\mathbf{A}$ in a vector $\mathbf{a}$, where $n^{(nz)}$ is the number of nonzero entries, such that the matrix product

$$\mathbf{A} = \mathbf{J}_1^\intercal \textsc{Diag}(\mathbf{a}) \mathbf{J}_2 \tag{17}$$

recovers the matrix $\mathbf{A}$ in the coordinate format, where $\mathbf{J}_1, \mathbf{J}_2 \in \mathbb{R}^{n^{(nz)} \times n}$. Here $\textsc{Diag}(\cdot)$ converts a vector into a diagonal matrix and notice that

$$\textsc{Diag}(\mathbf{u})\mathbf{v} = \textsc{Diag}(\mathbf{v})\mathbf{u} = \mathbf{u} \odot \mathbf{v} \tag{18}$$

for two vectors $\mathbf{u}, \mathbf{v}$, where $\mathbf{u} \odot \mathbf{v}$ is the element-wise multiplication of two vectors. Namely, $\mathbf{J}_1(i, 0)$ and $\mathbf{J}_2(i, 0)$ is a one-hot sparse vector putting $\mathbf{a}_i$ at the correct row and column.

Then, any infinitesimal deviation $(\delta\mathbf{A}, \delta\mathbf{b}, \delta\mathbf{x})$ from the stationary $(\mathbf{A}, \mathbf{b}, \mathbf{x})$ such that $\mathbf{A}\mathbf{x} = \mathbf{b}$ must satisfy that:

$$\begin{aligned}(\delta\mathbf{A})\mathbf{x} + \mathbf{A}\delta\mathbf{x} &= \delta\mathbf{b} \\ \mathbf{J}_1^\intercal \textsc{Diag}(\delta\mathbf{a})\mathbf{J}_2\mathbf{x} + \mathbf{J}_1^\intercal \textsc{Diag}(\mathbf{a})\mathbf{J}_2\delta\mathbf{x} &= \delta\mathbf{b} \\ \mathbf{J}_1^\intercal \textsc{Diag}(\mathbf{J}_2\mathbf{x})\delta\mathbf{a} + \mathbf{A}\delta\mathbf{x} &= \delta\mathbf{b}\end{aligned}$$

So we have:

$$\partial \mathbf{x}/\partial \mathbf{a} = -\mathbf{A}^{-1}\mathbf{J}_1^\intercal \textsc{Diag}(\mathbf{J}_2\mathbf{x}) \tag{19}$$

$$\partial \mathbf{x}/\partial \mathbf{b} = \mathbf{A}^{-1} \tag{20}$$

Thus, provided with $\nabla_{\mathbf{x}} E$, the gradient of the energy $E(\mathbf{x})$ w.r.t. $\mathbf{x}$, the chain rule gives the gradients $\nabla_{\mathbf{a}} E$ and $\nabla_{\mathbf{b}} E$.

$$\nabla_{\mathbf{a}} E = -(\mathbf{J}_2\mathbf{x}) \odot \left(\mathbf{J}_1 (\mathbf{A}^\intercal)^{-1} \nabla_{\mathbf{x}} E\right) \tag{21}$$

$$\nabla_{\mathbf{b}} E = (\mathbf{A}^\intercal)^{-1} \nabla_{\mathbf{x}} E \tag{22}$$

Remark: to evaluate the gradients, we need to solve the system $\mathbf{A}^\intercal$ instead of $\mathbf{A}$ in general, unless $\mathbf{A}$ is symmetric.

C.4. Two settings in linear solvers

We clarify that there are two different settings when solving (multiple instances of) $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$:

- $\mathbf{A}$ is unchanged and fixed. This is when the common trick can be applied, e.g. in interactive computer graphics (Bouaziz et al., 2014; Du et al., 2021): pre-factorize $\mathbf{A} = \mathbf{B}^\intercal\mathbf{B}$ once and at runtime reuse the fixed factorization for speedup, only performing back substitution.
- $\mathbf{A}$ is changed over iterations, solving systems with coefficients unknown in advance. This is a more common situation for nonlinear systems. For example, in Newton's method, $\mathbf{A}$ is the Hessian that changes over iterations.

In this paper, we focus on the second setup, where the system must be solved from scratch and the numerical factorization stage—which is the main bottleneck—is therefore included. Nonetheless, both settings can benefit from our method significantly.

C.5. Involution: exact inverse convolution (spatially varying kernel)

Many classical algorithms can be viewed as a single step of Schwarz–Schur involution—solving a linear Laplacian-like system. Image filtering (Barron & Poole, 2016), editing (Pérez et al., 2003), matting (Germer et al., 2020), and segmentation (Shi & Malik, 2000) are classical examples. The linear solving in these methods can be viewed as the generalized deconvolution with a spatially varying kernel, but they are usually not called a deconvolution task. To distinguish from the common setting of deconvolution, we term our process Schwarz–Schur (SS) involution, the exact inversion of the convolution with a spatial kernel. We call the set of dense matrices that are applied across all subdomains as the *Schwarz–Schur (SS) involution kernel* at that level, which are stored in the tensor $\boldsymbol{\alpha}^{(*)}$.

Theoretically, our method can enjoy further improvements when the kernel $a^{(x,y)}$ is a spatially constant matrix (the timing reported in the paper does not take advantage of

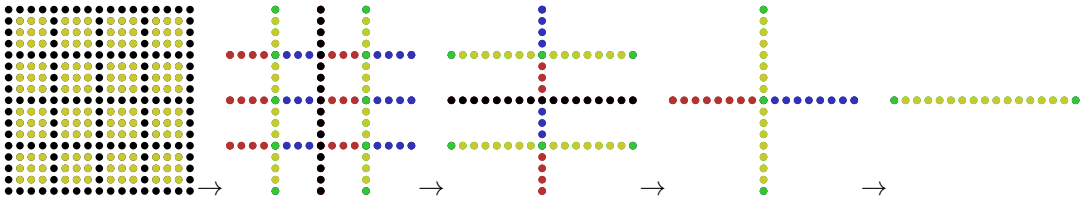


Figure 20. The node elimination order corresponds to Figure 18. An ideal elimination order for the Neumann boundary conditions.

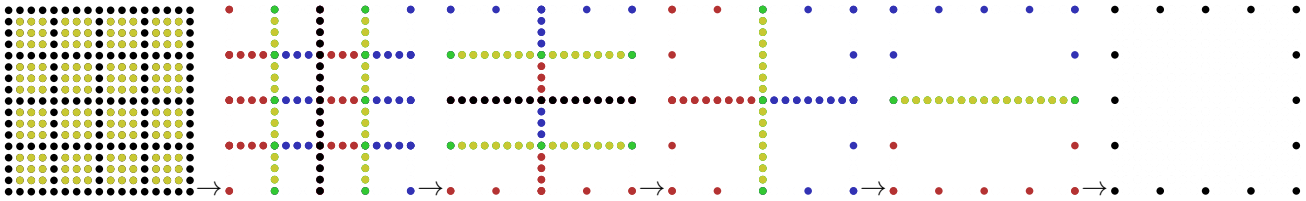


Figure 21. The node elimination order corresponds to Figure 19. The elimination order we actually use for the Neumann boundary conditions for the ease of implementation.

this: we always treat $a^{(x,y)}$ as spatially varying even if it is constant for fair comparisons). In this case, at each subdivision level, the SS-involution kernels are the same across the domain. Thus, there is no need to maintain a dense matrix for each subdomain individually; instead, we can maintain one SS-involution kernel that applies to every subdomain.

What is even better: for a prescribed kernel, the SS-involution kernel has a closed-form formula that can be derived and calculated ahead of time, bypassing the need of numerical factorization. When $\mathbf{A}$ is the isotropic Laplacian, the resulting DtN matrix approximates a continuous DtN operator which has a known low-rank approximation, the structure that may be leveraged in future work (Bebendorf, 2008).

C.6. Inverse problems, optimal control of PDEs, and PDE-constrained optimization

Our method accelerates not only the forward linear solver, but also the "backward" process of differentiating into the linear solver, which is useful in, e.g. inverse problems, optimal control of PDEs, and PDE-constrained optimization.

For $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, back-propagating the gradient w.r.t. $\mathbf{x}$ to that w.r.t. $\mathbf{b}$ is straightforward since $\delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b}$, so we will focus on how to differentiate into the coefficients of $\mathbf{A}$. There are two ways:

- In fact, our method implemented in PyTorch, is already end-to-end differentiable w.r.t. entries of $\mathbf{A}$. However, it is generally advised against differentiating through the low-level implementation details: see Hovland & Hückelheim (2024) empirical studies and references therein for discussions. We also find it that indeed this naïve approach is inefficient in the backward step to obtain gradients.
- Alternatively, as described in §C.3, without relying on the automatic differentiation feature of PyTorch, we can explicitly calculate the gradient by solving the adjoint system $\mathbf{A}^{-\intercal}\mathbf{c}$, where $\mathbf{c}$ depends on $\mathbf{x}$, similarly to the differentiable bilateral solver (Barron & Poole, 2016). When $\mathbf{A}$ is symmetric, this involves solving a linear system with the same left-hand side, see, e.g., Wang et al. (2023).

Thus, our solver immediately speeds up the optimization of some distributed parameters $\mathbf{A}$, which—written in the equivalent form $a^{(x,y)}$—can represent a convolutional kernel, local material property, or the Jacobian of a deformation field. In the literature of inverse PDEs, this is often referred to as the problem of distributed parameter identification.

In §5, we apply the differentiable solver to search for the $\mathbf{A}$ that minimizes an objective $E(\cdot)$. This is an inverse problem

$$\begin{aligned} &\min_{\mathbf{A}} E(\mathbf{x}) \\ &\text{s.t. } \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \end{aligned} \tag{23}$$

In §5, the linear system comes from FEM: $\mathbf{A} = \mathbf{G}^\intercal\mathbf{C}\mathbf{G}$ to yield a PDE-constrained optimization (Wang & Solomon, 2021; Wang et al., 2023), and the goal is to optimize $\mathbf{C}$ that encodes a deformation field that minimizes the image matching loss.

C.7. Generalize to larger kernels

Our approaches can be generalized to a larger kernel size of 5×5 or 7×7 with a more complicated implementation. Currently, one layer of boundary pixels can separate two subdomains P, Q, and it will require two layers of boundary pixels in the case of 5×5 to separate two subdomains P,Q (or 3 layers of boundary pixels in the case of 7×7).

Similarly, the boundary pixels (at the border of the whole image) in the last Schur step will have two layers of pixels. The parallel elimination procedure will be similar, but will have to account for the fact that the "wire-frame" has two layers of pixels.

In fact, in some sense, our current method **already supports** kernels larger than 3×3. The actual constraint that we have is that every pixel can only contribute to pixels in the same 5×5 patch (and recall that pixels at the patch boundary belong to multiple patches). Thus, the convolution window for a pixel can cover the entire 1/2/4 patches it belongs to: for example, pixels at the patch boundary can use a local window of 9×9 or 9×5, and it is 5×5 for an interior pixel, though the window may not be centered at it. Also, recall that the 5×5 patch size is a hyperparameter that we are free to change arbitrarily in the current method.

D. Discussions and Implementation Details

D.1. Details on the algorithm

A hyperparameter in our method is the size of patches, which we choose as 4×4 (enlarged to 5×5 for duplicating boundary pixels at interfaces), except for the image of size 2561^2 where we choose the size to be 5×5 (enlarged to 6×6).

D.2. Distributed representations of sparse systems

In our method, we never have to construct a global sparse matrix as other direct sparse solvers do, leading to extra saving in time. In the initialization step the Schur complements are stored in a distributed fashion.

D.3. Details on experiment setups

Compared to Python, the Julia ecosystem has much better support for sparse linear algebraic CUDA APIs that are critical in scientific computing. For this reason, when comparing our method with CUDA, we use the up-to-date Julia binding that provides APIs backended by the most recently released cuDSS with cuSolver, cuSparse and CUDA. cuDSS is the state-of-the-art official CUDA library that supports sparse linear solvers including sparse LU (LDU) factorization, becoming our direct point of comparison.

D.4. Memory complexity

We emphasize that the reported memory for our solver is the maximum GPU usage for its workspace storing all intermediate matrices and can be released after the solution is obtained: if the solver is included as a layer in the neural networks, the memory can be released after the forward pass (though storing the workspace can skip the numerical factorization stage in the back-propagation step to improve performance). For example, if the solver layer is included 10 times in the neural architecture, the peak memory usage for the model is same to including the solver only 1 time. Our solver currently treats symmetric $\mathbf{A}$ as asymmetric ones: future work can leverage the symmetry to save half of the memory usage.

Table 12. Memory usage (MB) comparison. Note we do not optimize our memory usage and there is likely large room for improvement. (*: while it fits in memory to run the code, using "torch.compile" for an optimized runtime exceeds the limit.)

	GPU memory usage (MB)		
Example	CUDA	ours-32	ours-64
4097^2	33330	OOM	OOM
2049^2	8104	16094	32188 (*)
1025^2	2234	3880	7244
513^2	880	890	1736
257^2	548	242	420

Our method achieves speed at the cost of memory usage. Table 12 compares the GPU memory usage with the CUDA solver. Note that our current implementation is generous in memory usage and we believe there is likely plenty of room for improvement. We save every intermediate dense matrices to allow quick experimentation of ideas, and do not use any PyTorch in-place operations, which we find may cause counterintuitive behaviors for sliced matrices, making debugging challenging.

In addition, the memory usage can be significantly reduced or even eliminated, if the kernel $a^{(x,y)}$ is constant spatially; future implementation may take advantage of this. In this case, all dense matrices are the same across the domain, and even have known closed form that can be derived ahead of time.

D.5. Midpoint reflective boundary condition

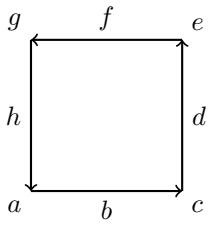


Figure 22. Name pixels at the boundary of the image domain in counter-clockwise ordering.

To handle spherical topology, we introduce a *midpoint reflective* boundary condition, as induced by the octahedral parameterization (Praun & Hoppe, 2003) that maps from a surface with spherical topology to the image domain. Shown

in Figure 22, the boundary of the image domain is split into the chain $a \to b \to c \to d \to e \to f \to g \to h \to a$. The problem becomes

$$\begin{aligned} & \min_{\mathbf{u}} \frac{1}{2}\mathbf{u}^\intercal \mathbf{D}\mathbf{u} - \mathbf{d}^\intercal \mathbf{u} \\ \text{s.t.} \quad & \mathbf{u}_a = \mathbf{u}_c = \mathbf{u}_e = \mathbf{u}_g \\ & \mathbf{u}_{(a\to b)} = \mathbf{u}_{(c\to b)}, \\ & \mathbf{u}_{(c\to d)} = \mathbf{u}_{(e\to d)}, \\ & \mathbf{u}_{(e\to f)} = \mathbf{u}_{(g\to f)}, \\ & \mathbf{u}_{(g\to h)} = \mathbf{u}_{(a\to h)}. \end{aligned} \tag{24}$$

The sub-vector $\mathbf{u}_{(a\to b)}$ does not include two endpoints a, b. The constraints come from how the octahedral parameterization (Praun & Hoppe, 2003) specifically sets the boundary condition to enforces the spherical topology. To solve the problem, we stack the unique values in $\mathbf{u}$ into a vector $\mathbf{v}$, such that

$$\mathbf{u} = \begin{bmatrix} \mathbf{u}_a \\ \mathbf{u}_{(a\to b)} \\ \mathbf{u}_b \\ \mathbf{u}_{(b\to c)} \\ \mathbf{u}_c \\ \mathbf{u}_{(c\to d)} \\ \mathbf{u}_d \\ \mathbf{u}_{(d\to e)} \\ \mathbf{u}_e \\ \mathbf{u}_{(e\to f)} \\ \mathbf{u}_f \\ \mathbf{u}_{(f\to g)} \\ \mathbf{u}_g \\ \mathbf{u}_{(g\to h)} \\ \mathbf{u}_h \\ \mathbf{u}_{(h\to a)} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_a \\ \mathbf{u}_{(a\to b)} \\ \mathbf{u}_b \\ \text{REV}(\mathbf{u}_{(a\to b)}) \\ \mathbf{u}_a \\ \mathbf{u}_{(c\to d)} \\ \mathbf{u}_d \\ \text{REV}(\mathbf{u}_{(c\to d)}) \\ \mathbf{u}_a \\ \mathbf{u}_{(e\to f)} \\ \mathbf{u}_f \\ \text{REV}(\mathbf{u}_{(e\to f)}) \\ \mathbf{u}_a \\ \mathbf{u}_{(g\to h)} \\ \mathbf{u}_h \\ \text{REV}(\mathbf{u}_{(g\to h)}) \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} \mathbf{u}_a \\ \mathbf{u}_{(a\to b)} \\ \mathbf{u}_b \\ \mathbf{u}_{(c\to d)} \\ \mathbf{u}_d \\ \mathbf{u}_{(e\to f)} \\ \mathbf{u}_f \\ \mathbf{u}_{(g\to h)} \\ \mathbf{u}_h \end{bmatrix}$$

in which $\text{REV}(\cdot)$ indicates flipping a vector, i.e. the vector with the same elements appearing in the reverse order. In this case, we can reduce the number of variables using $\mathbf{u} = \mathbf{F}\mathbf{v}$. $\mathbf{F}$ is a sparse binary matrix, describing how entries in $\mathbf{u}$ can be copied from entries in $\mathbf{v}$, such that $\mathbf{1} = \mathbf{F}\mathbf{1}$.

System (24) then can be solved via $\mathbf{v} = (\mathbf{F}^\intercal \mathbf{D}\mathbf{F})^{-1}\mathbf{F}^\intercal \mathbf{d}$ and the solution can be recovered from $\mathbf{u} \leftarrow \mathbf{F}\mathbf{v}$.

E. Detailed Method Description with Necessary Background Information

In this section, we provide a detailed description of our method along with the necessary background, supplementing the brief explanation in the main text (kept short due to space constraints). Nonetheless, understanding the internal details is typically unnecessary for most users, as our method is designed to be used as a black-box module.

Plug-and-play for end users Just as convolution and matrix multiplication are accessed via CUDA (cuDNN/cuBLAS), we that envision our solver can be similarly exposed through low-level APIs (e.g., via a cuDSS-like interface) with further opportunities of optimization, allowing users to easily integrate it without needing to implement anything themselves.

Notation: matrix slicing For a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, we use the matrix slicing notation $\mathbf{A}_{rs}$ to denote the submatrix of $\mathbf{A}$ that takes the rows specified by r and columns specified by s. For example, for $r = [2, 1, 3]$, $s = [6, 7]$, $\mathbf{A}_{rs}$ refers to the 3×2 submatrix:

$$\mathbf{A}_{rs} := \begin{bmatrix} \mathbf{A}_{2,6} & \mathbf{A}_{2,7} \\ \mathbf{A}_{1,6} & \mathbf{A}_{1,7} \\ \mathbf{A}_{3,6} & \mathbf{A}_{3,7} \end{bmatrix} \tag{25}$$

that is obtained by selecting the first three rows, in the order of $2, 1, 3$, and 6-th, 7-th columns from matrix $\mathbf{A}$. The lower scripted $\mathbf{A}_{rs}$ should not be confused with upper scripted $\mathbf{A}^{rs}$, which is just a variable that contains r, s as part of its name and does not necessarily have anything to do with slicing.

Reverse permutation matrix

$$\mathbf{J} := \begin{bmatrix} & & & 1 \\ & & \iddots & \\ & 1 & & \\ 1 & & & \end{bmatrix} \equiv \mathbf{J}^\intercal \equiv \mathbf{J}^{-1} \tag{26}$$

Schur complement Schur complement reduces solving the 2×2 block matrix,

$$\begin{bmatrix} \mathbf{X} & \mathbf{Y} \\ \mathbf{Z} & \mathbf{W} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \mathbf{w} \end{bmatrix} \tag{27}$$

to solving a smaller matrix $\mathbf{D}\mathbf{x} = \mathbf{d}$, assuming an invertible $\mathbf{W}$, where

$$\mathbf{D} := \left(\mathbf{X} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{Z}\right) \quad \mathbf{d} := \mathbf{y} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{w} \tag{28}$$

by canceling $\mathbf{z}$ using the second line of the equation that rewrites $\mathbf{z} = \mathbf{W}^{-1}[\mathbf{w} - \mathbf{Z}\mathbf{x}]$ as a function of $\mathbf{x}$.

While equations in the paper might look a bit dense, at the high level the overall procedure is quite simple: recursively applying the Schur complement to the system many times to reduce the problem to a smaller system. There are a few extra considerations: 1) implicitly or explicitly re-order rows and columns of the matrix as necessary, so that we know the part we want to eliminate is located at the sub-block $\mathbf{W}$ (or anywhere we prescribed); 2) apply many Schur complements in parallel.

E.1. Case study: two subdomains

We review some standard concepts before explaining our method for accessibility to a larger audience. The reader may refer to §E.6, a smaller 3×7 image for clarity of illustration.

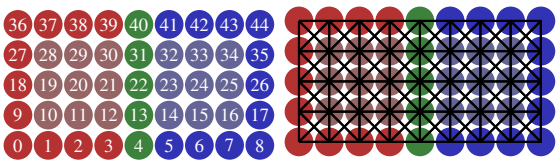


Figure 23. The connectivity graph resulted from the use of the 3×3 kernel size, for a 5×9 image under the lexicographic sweeping order.

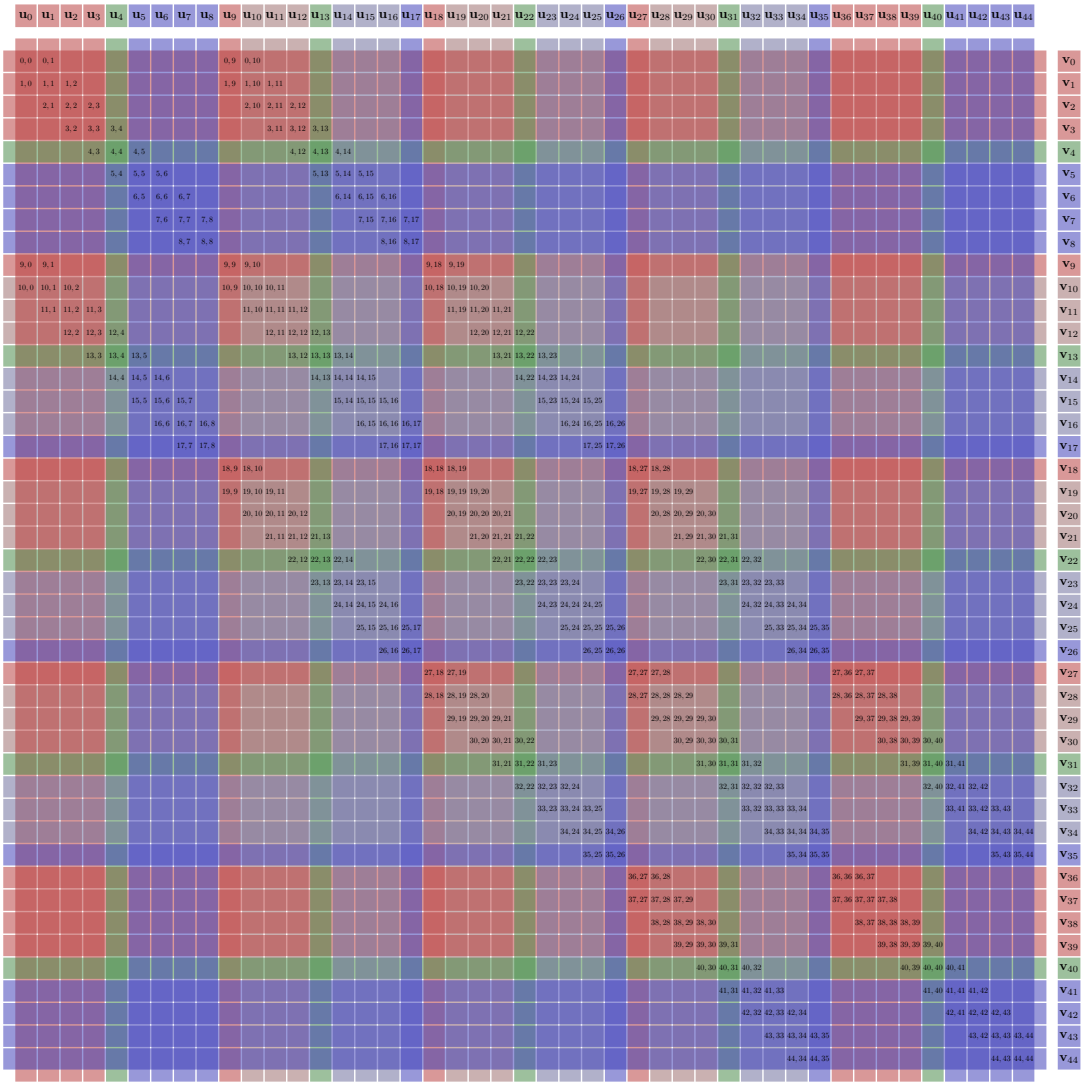


Figure 24. For the 5×9 image, using the pixel indexing in Figure 23, visualization of the sparsity pattern of $\mathbf{A} \in \mathbb{R}^{45\times 45}$, for the original sparse linear system $\sum_j \mathbf{A}_{ij}\mathbf{u}_j = \mathbf{v}_i, \forall i$. As we can see, the original Laplacian-like $\mathbf{A}$ is a **banded matrix** with three banded diagonals.

For the 5×9 image shown in Figure 23, each pixel corresponds to a node in the graph and is assigned with an index $i \in \{0, 1, ..., 45-1\}$. Two nodes i and j, where $i, j \in \{0, 1, ..., 45-1\}$, are connected by an edge if the 3×3 kernel centered at one node will cover the other node. $\mathbf{A}_{ij} = 0$ if node i and node j are not connected by an edge in the graph.

For the original sparse linear system, $\mathbf{A} \in \mathbb{R}^{45\times 45}$:

$$\sum_{j=0}^{45-1} \mathbf{A}_{ij}\mathbf{u}_j = \mathbf{v}_i, \forall i \in \{0, 1, ..., 45-1\}. \tag{29}$$

Figure 24 visualizes the sparsity pattern of $\mathbf{A} \in \mathbb{R}^{45\times 45}$. In this visualization of $\mathbf{A}$, only where there is a (i, j) corresponds an entry that $\mathbf{A}_{ij} \neq 0$—namely i and j are connected by an edge in Figure 23, and all other entries that we omitted are zeros. The original $\mathbf{A}$ is a Laplacian-like matrix with a 9-point stencil. $\mathbf{A}$ is a **banded matrix** with three banded diagonals. Previous direct solvers have to explicitly maintain the sparse matrix $\mathbf{A}$ shown in Figure 24, while our solver does not. Instead, we work with a more compact representation that takes advantage of the regular grid structure: we start with the dense block submatrices shown in Figure 25.

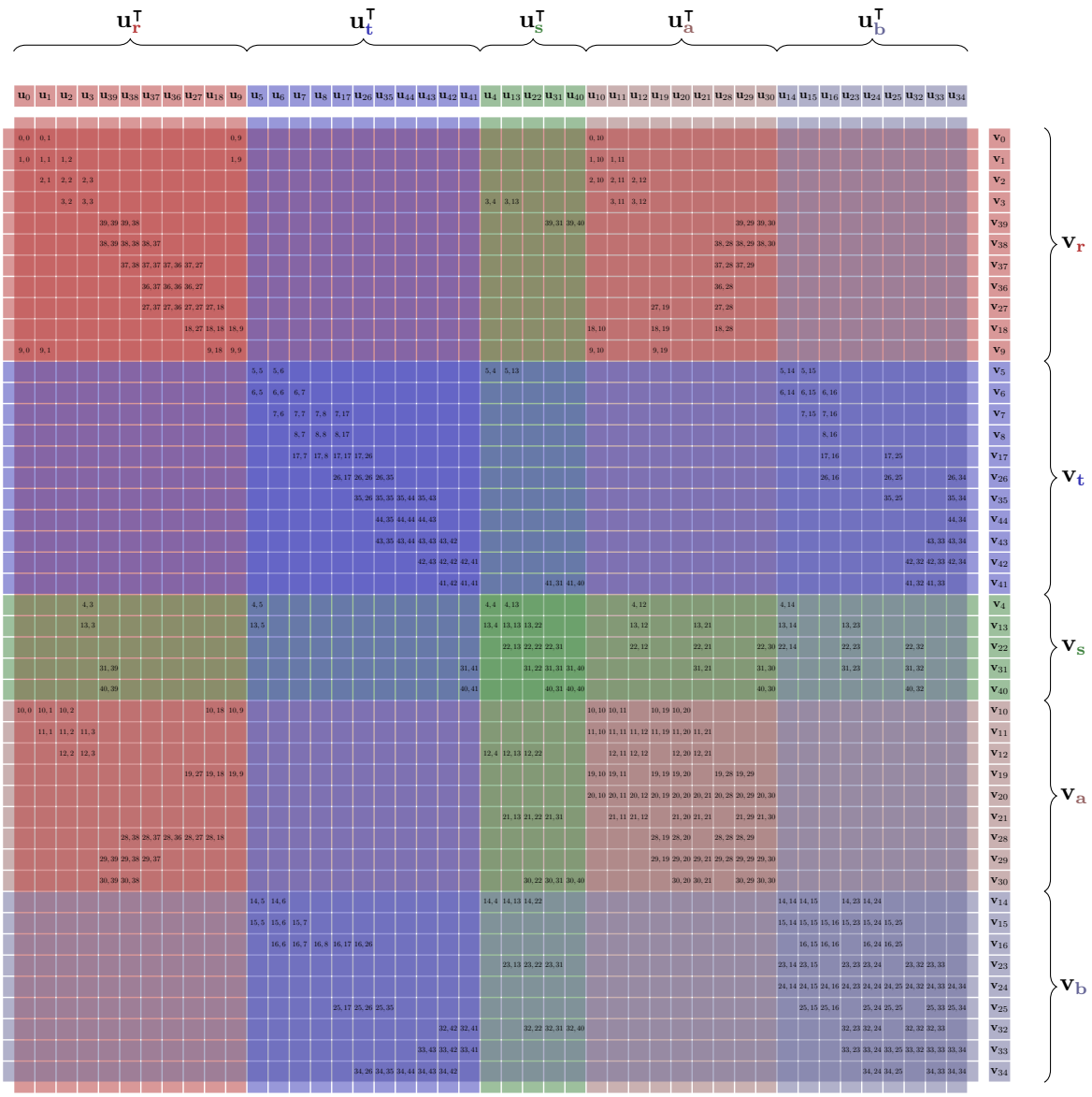


Figure 25. After reordering rows and columns, the original system becomes the 5×5 block system:

$$\begin{bmatrix} \mathbf{A}_{rr} & \mathbf{0} = \mathbf{A}_{rt} & \mathbf{A}_{rs} & \mathbf{A}_{ra} & \mathbf{0} = \mathbf{A}_{rb} \\ \mathbf{0} = \mathbf{A}_{tr} & \mathbf{A}_{tt} & \mathbf{A}_{ts} & \mathbf{0} = \mathbf{A}_{ta} & \mathbf{A}_{tb} \\ \mathbf{A}_{sr} & \mathbf{A}_{st} & \mathbf{A}_{ss} & \mathbf{A}_{sa} & \mathbf{A}_{sb} \\ \mathbf{A}_{ar} & \mathbf{0} = \mathbf{A}_{at} & \mathbf{A}_{as} & \mathbf{A}_{aa} & \mathbf{0} = \mathbf{A}_{ab} \\ \mathbf{0} = \mathbf{A}_{br} & \mathbf{A}_{bt} & \mathbf{A}_{bs} & \mathbf{0} = \mathbf{A}_{ba} & \mathbf{A}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{u}_r \\ \mathbf{u}_t \\ \mathbf{u}_s \\ \mathbf{u}_a \\ \mathbf{u}_b \end{bmatrix} = \begin{bmatrix} \mathbf{v}_r \\ \mathbf{v}_t \\ \mathbf{v}_s \\ \mathbf{v}_a \\ \mathbf{v}_b \end{bmatrix}.$$

By pivoting the rows and columns of $\mathbf{A}$, we arrive at the linear system in Figure 25. It is easy to verify that the pivoting does not change the problem: it only re-orders the equations and variables. From Figure 25, now it is visually clear that after the row- and column- pivoting, some submatrices are zero. We treat the nonzero block matrices as dense matrices though they can be sparse.

Now, let us start with:

$$\begin{bmatrix} \mathbf{A}_{\mathrm{rr}} & \mathbf{0} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{ra}} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathrm{tt}} & \mathbf{A}_{\mathrm{ts}} & \mathbf{0} & \mathbf{A}_{\mathrm{tb}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{st}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sa}} & \mathbf{A}_{\mathrm{sb}} \\ \mathbf{A}_{\mathrm{ar}} & \mathbf{0} & \mathbf{A}_{\mathrm{as}} & \mathbf{A}_{\mathrm{aa}} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathrm{bt}} & \mathbf{A}_{\mathrm{bs}} & \mathbf{0} & \mathbf{A}_{\mathrm{bb}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \mathbf{u}_{\mathrm{t}} \\ \mathbf{u}_{\mathrm{s}} \\ \mathbf{u}_{\mathrm{a}} \\ \mathbf{u}_{\mathrm{b}} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{\mathrm{r}} \\ \mathbf{v}_{\mathrm{t}} \\ \mathbf{v}_{\mathrm{s}} \\ \mathbf{v}_{\mathrm{a}} \\ \mathbf{v}_{\mathrm{b}} \end{bmatrix}. \tag{30}$$

where

$$\begin{aligned} \mathrm{s} &= [4, 13, 22, 31, 40], \\ \mathrm{r} &= [0, 1, 2, 3, 39, 38, 37, 36, 27, 18, 9], \\ \mathrm{a} &= [10, 11, 12, 19, 20, 21, 28, 29, 30], \\ \mathrm{t} &= [5, 6, 7, 8, 17, 26, 35, 44, 43, 42, 41], \\ \mathrm{b} &= [14, 15, 16, 23, 24, 25, 32, 33, 34]. \end{aligned} \tag{31}$$

Patch division and patch-wise finite element methods Recall that

$$\mathbf{A}_{\mathrm{ss}} = \mathbf{A}_{\mathrm{ss}}^{(P)} + \mathbf{A}_{\mathrm{ss}}^{(Q)}, \tag{32}$$

$$\mathbf{v}_{\mathrm{s}} = \mathbf{v}_{\mathrm{s}}^{(P)} + \mathbf{v}_{\mathrm{s}}^{(Q)} \tag{33}$$

can be divided into contributions from P and Q, resp. With this partition, the computations for patch P and Q are made independent from each other, and thus can be done in parallel. The values of $\mathbf{A}_{\mathrm{ss}}^{(P)}$ and $\mathbf{A}_{\mathrm{ss}}^{(Q)}$ can be arbitrary, as long as their summation is the correct $\mathbf{A}_{\mathrm{ss}}$. In fact, their values are never used standalone, only their sum $\mathbf{A}_{\mathrm{ss}}^{(P)} + \mathbf{A}_{\mathrm{ss}}^{(Q)}$ is used (they become parts of the matrices $\mathbf{P}, \mathbf{Q}$ which are summed later).

Especially for applications like PDEs (discretized with first-order piecewise linear FEM), each patch P and Q will contribution to the value of $\mathbf{A}_{\mathrm{ss}}$ independently: their portions of contributions $\mathbf{A}_{\mathrm{ss}}^{(P)}$ and $\mathbf{A}_{\mathrm{ss}}^{(Q)}$ only depend on information within the patch P and Q, respectively. This makes the computation within P and Q exactly independent from each other, easing parallel implementation.

System partitioning

$$\begin{bmatrix} \mathbf{A}_{\mathrm{rr}} & \mathbf{0} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{ra}} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathrm{tt}} & \mathbf{A}_{\mathrm{ts}} & \mathbf{0} & \mathbf{A}_{\mathrm{tb}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{st}} & \mathbf{A}_{\mathrm{ss}}^{(P)} + \mathbf{A}_{\mathrm{ss}}^{(Q)} & \mathbf{A}_{\mathrm{sa}} & \mathbf{A}_{\mathrm{sb}} \\ \mathbf{A}_{\mathrm{ar}} & \mathbf{0} & \mathbf{A}_{\mathrm{as}} & \mathbf{A}_{\mathrm{aa}} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathrm{bt}} & \mathbf{A}_{\mathrm{bs}} & \mathbf{0} & \mathbf{A}_{\mathrm{bb}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \mathbf{u}_{\mathrm{t}} \\ \mathbf{u}_{\mathrm{s}} \\ \mathbf{u}_{\mathrm{a}} \\ \mathbf{u}_{\mathrm{b}} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{\mathrm{r}} \\ \mathbf{v}_{\mathrm{t}} \\ \mathbf{v}_{\mathrm{s}}^{(P)} + \mathbf{v}_{\mathrm{s}}^{(Q)} \\ \mathbf{v}_{\mathrm{a}} \\ \mathbf{v}_{\mathrm{b}} \end{bmatrix} \tag{34}$$

$$\begin{bmatrix} \mathbf{A}_{\mathrm{rr}} & \mathbf{0} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{ra}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{0} & \mathbf{A}_{\mathrm{ss}}^{(P)} & \mathbf{A}_{\mathrm{sa}} & \mathbf{0} \\ \mathbf{A}_{\mathrm{ar}} & \mathbf{0} & \mathbf{A}_{\mathrm{as}} & \mathbf{A}_{\mathrm{aa}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \mathbf{u}_{\mathrm{t}} \\ \mathbf{u}_{\mathrm{s}} \\ \mathbf{u}_{\mathrm{a}} \\ \mathbf{u}_{\mathrm{b}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathrm{tt}} & \mathbf{A}_{\mathrm{ts}} & \mathbf{0} & \mathbf{A}_{\mathrm{tb}} \\ \mathbf{0} & \mathbf{A}_{\mathrm{st}} & \mathbf{A}_{\mathrm{ss}}^{(Q)} & \mathbf{0} & \mathbf{A}_{\mathrm{sb}} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathrm{bt}} & \mathbf{A}_{\mathrm{bs}} & \mathbf{0} & \mathbf{A}_{\mathrm{bb}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \mathbf{u}_{\mathrm{t}} \\ \mathbf{u}_{\mathrm{s}} \\ \mathbf{u}_{\mathrm{a}} \\ \mathbf{u}_{\mathrm{b}} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{\mathrm{r}} \\ \mathbf{0} \\ \mathbf{v}_{\mathrm{s}}^{(P)} \\ \mathbf{v}_{\mathrm{a}} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{v}_{\mathrm{t}} \\ \mathbf{v}_{\mathrm{s}}^{(Q)} \\ \mathbf{0} \\ \mathbf{v}_{\mathrm{b}} \end{bmatrix} \tag{35}$$

Instead of working with the sparse matrix $\mathbf{A}$ and a right-hand side $\mathbf{b}$, our method works with the dense blocks—that are compactly put in two tensors $\boldsymbol{\alpha}, \boldsymbol{\beta}$ as we will introduce later—that come from the nonzero parts of $\mathbf{A}$:

$$\begin{bmatrix} \mathbf{A}_{\mathrm{rr}} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{ra}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}}^{(P)} & \mathbf{A}_{\mathrm{sa}} \\ \mathbf{A}_{\mathrm{ar}} & \mathbf{A}_{\mathrm{as}} & \mathbf{A}_{\mathrm{aa}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{r}} \\ \mathbf{u}_{\mathrm{s}} \\ \mathbf{u}_{\mathrm{a}} \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} \mathbf{v}_{\mathrm{r}} \\ \mathbf{v}_{\mathrm{s}}^{(P)} \\ \mathbf{v}_{\mathrm{a}} \end{bmatrix} \tag{36}$$

$$\begin{bmatrix} \mathbf{A}_{\mathrm{tt}} & \mathbf{A}_{\mathrm{ts}} & \mathbf{A}_{\mathrm{tb}} \\ \mathbf{A}_{\mathrm{st}} & \mathbf{A}_{\mathrm{ss}}^{(Q)} & \mathbf{A}_{\mathrm{sb}} \\ \mathbf{A}_{\mathrm{bt}} & \mathbf{A}_{\mathrm{bs}} & \mathbf{A}_{\mathrm{bb}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_{\mathrm{t}} \\ \mathbf{u}_{\mathrm{s}} \\ \mathbf{u}_{\mathrm{b}} \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} \mathbf{v}_{\mathrm{t}} \\ \mathbf{v}_{\mathrm{s}}^{(Q)} \\ \mathbf{v}_{\mathrm{b}} \end{bmatrix} \tag{37}$$

The symbol "$\stackrel{?}{=}$" indicates the above systems cannot be solved directly, since our assumption is that the values $\mathbf{A}_{ss}^{(P)}$ and $\mathbf{A}_{ss}^{(Q)}$ can be arbitrary, and only their summation is a known value $\mathbf{A}_{ss}$. In fact, the solution depends on every patch so any local computation cannot yield the correct solution. But we can still eliminate $\mathbf{u}_a, \mathbf{u}_b$ by writing their values as a function of $\mathbf{u}_r, \mathbf{u}_s, \mathbf{u}_t$.

$$\begin{bmatrix}\mathbf{u}_a\\ \mathbf{u}_b\end{bmatrix} = \begin{bmatrix}\mathbf{A}_{aa}^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_{bb}^{-1}\end{bmatrix}\left(\begin{bmatrix}\mathbf{v}_a\\ \mathbf{v}_b\end{bmatrix} - \begin{bmatrix}\mathbf{A}_{ar} & \mathbf{0} & \mathbf{A}_{as}\\ \mathbf{0} & \mathbf{A}_{bt} & \mathbf{A}_{bs}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix}\right) \tag{38}$$

and plugging it into the joint system:

$$\begin{bmatrix}\mathbf{A}_{rr} & \mathbf{0} & \mathbf{A}_{rs}\\ \mathbf{0} & \mathbf{A}_{tt} & \mathbf{A}_{ts}\\ \mathbf{A}_{sr} & \mathbf{A}_{st} & \mathbf{A}_{ss}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix} + \begin{bmatrix}\mathbf{A}_{ra} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_{tb}\\ \mathbf{A}_{sa} & \mathbf{A}_{sb}\end{bmatrix}\begin{bmatrix}\mathbf{u}_a\\ \mathbf{u}_b\end{bmatrix} = \begin{bmatrix}\mathbf{v}_r\\ \mathbf{v}_t\\ \mathbf{v}_s\end{bmatrix} \tag{39}$$

which leads to:

$$\begin{bmatrix}\mathbf{A}_{rr} & \mathbf{0} & \mathbf{A}_{rs}\\ \mathbf{0} & \mathbf{A}_{tt} & \mathbf{A}_{ts}\\ \mathbf{A}_{sr} & \mathbf{A}_{st} & \mathbf{A}_{ss}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix} + \begin{bmatrix}\mathbf{A}_{ra} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_{tb}\\ \mathbf{A}_{sa} & \mathbf{A}_{sb}\end{bmatrix}\begin{bmatrix}\mathbf{A}_{aa}^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_{bb}^{-1}\end{bmatrix}\left(\begin{bmatrix}\mathbf{v}_a\\ \mathbf{v}_b\end{bmatrix} - \begin{bmatrix}\mathbf{A}_{ar} & \mathbf{0} & \mathbf{A}_{as}\\ \mathbf{0} & \mathbf{A}_{bt} & \mathbf{A}_{bs}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix}\right) = \begin{bmatrix}\mathbf{v}_r\\ \mathbf{v}_t\\ \mathbf{v}_s\end{bmatrix} \tag{40}$$

which further simplifies to:

$$\begin{aligned}&\begin{bmatrix}\mathbf{A}_{rr} - \mathbf{A}_{ra}\mathbf{A}_{aa}^{-1}\mathbf{A}_{ar} & \mathbf{0} & \mathbf{A}_{rs} - \mathbf{A}_{ra}\mathbf{A}_{aa}^{-1}\mathbf{A}_{as}\\ \mathbf{0} & \mathbf{A}_{tt} - \mathbf{A}_{tb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bt} & \mathbf{A}_{ts} - \mathbf{A}_{tb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bs}\\ \mathbf{A}_{sr} - \mathbf{A}_{sa}\mathbf{A}_{aa}^{-1}\mathbf{A}_{ar} & \mathbf{A}_{st} - \mathbf{A}_{sb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bt} & \mathbf{A}_{ss} - \mathbf{A}_{sa}\mathbf{A}_{aa}^{-1}\mathbf{A}_{as} - \mathbf{A}_{sb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bs}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix}\\ &= \begin{bmatrix}\mathbf{v}_r - \mathbf{A}_{ra}\mathbf{A}_{aa}^{-1}\mathbf{v}_a\\ \mathbf{v}_t - \mathbf{A}_{tb}\mathbf{A}_{bb}^{-1}\mathbf{v}_b\\ \mathbf{v}_s - \mathbf{A}_{sa}\mathbf{A}_{aa}^{-1}\mathbf{v}_a - \mathbf{A}_{sb}\mathbf{A}_{bb}^{-1}\mathbf{v}_b\end{bmatrix}\end{aligned} \tag{41}$$

solving which yields the solution to the original problem.

$$\begin{aligned}&\begin{bmatrix}\mathbf{A}_{rr} - \mathbf{A}_{ra}\mathbf{A}_{aa}^{-1}\mathbf{A}_{ar} & \mathbf{0} & \mathbf{A}_{rs} - \mathbf{A}_{ra}\mathbf{A}_{aa}^{-1}\mathbf{A}_{as}\\ \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{A}_{sr} - \mathbf{A}_{sa}\mathbf{A}_{aa}^{-1}\mathbf{A}_{ar} & \mathbf{0} & \mathbf{A}_{ss}^{(P)} - \mathbf{A}_{sa}\mathbf{A}_{aa}^{-1}\mathbf{A}_{as}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix} + \begin{bmatrix}\mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_{tt} - \mathbf{A}_{tb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bt} & \mathbf{A}_{ts} - \mathbf{A}_{tb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bs}\\ \mathbf{0} & \mathbf{A}_{st} - \mathbf{A}_{sb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bt} & \mathbf{A}_{ss}^{(Q)} - \mathbf{A}_{sb}\mathbf{A}_{bb}^{-1}\mathbf{A}_{bs}\end{bmatrix}\begin{bmatrix}\mathbf{u}_r\\ \mathbf{u}_t\\ \mathbf{u}_s\end{bmatrix} =\\ &\begin{bmatrix}\mathbf{v}_r - \mathbf{A}_{ra}\mathbf{A}_{aa}^{-1}\mathbf{v}_a\\ \mathbf{0}\\ \mathbf{v}_s^{(P)} - \mathbf{A}_{sa}\mathbf{A}_{aa}^{-1}\mathbf{v}_a\end{bmatrix} + \begin{bmatrix}\mathbf{0}\\ \mathbf{v}_t - \mathbf{A}_{tb}\mathbf{A}_{bb}^{-1}\mathbf{v}_b\\ \mathbf{v}_s^{(Q)} - \mathbf{A}_{sb}\mathbf{A}_{bb}^{-1}\mathbf{v}_b\end{bmatrix}\end{aligned} \tag{42}$$

The derivations demonstrate that the computation can be split into contributions from each subdomain.

E.2. Parallel Schwarz elimination step

Previous discussions of the Schwarz step in §3.2.1 are only for illustrative purposes and do not match our implementation of the Schwarz step. Instead, in this section we provide full details for the Schwarz step that eliminates the interior of each patch.

The boundary-first ordering. As shown in Figure 26, we adopt a boundary-first ordering that uses $\eta = \{0 \sim 15\}$ to index the boundary nodes, and $\omega = \{16 \sim 24\}$ for interior nodes.

$$\eta := [0, 1, 2, ..15] \tag{43}$$
$$\omega := [16, 17, 18, ...24] \tag{44}$$

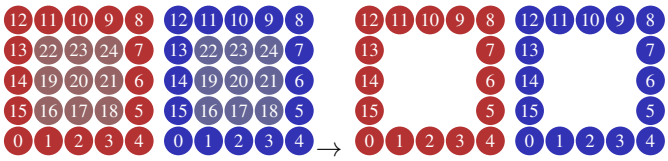


Figure 26. The boundary-first ordering. Schwarz step removes the interior pixels for each patch in parallel.

In contrast, Figure 27 shows the lexicographic sweeping order.

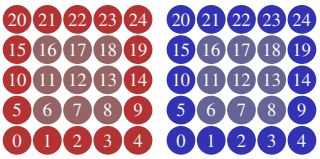


Figure 27. The lexicographic sweeping ordering.

Define the permutation vector $\boldsymbol{\mu}$ that is useful to map array stored in the lexicographic sweeping order to the boundary-first ordering, and its inverse permutation vector $\boldsymbol{\nu}$:

$$\boldsymbol{\mu} := [0, 1, 2, 3, 4, 9, 14, 19, 24, 23, 22, 21, 20, 15, 10, 5, 6, 7, 8, 11, 12, 13, 16, 17, 18] \tag{45}$$
$$\boldsymbol{\nu} := [0, 1, 2, 3, 4, 15, 16, 17, 18, 5, 14, 19, 20, 21, 6, 13, 22, 23, 24, 7, 12, 11, 10, 9, 8] \tag{46}$$

Patch-wise subsystems. This step directly constructs the matrices $\mathbf{P}, \mathbf{Q} \in \mathbb{R}^{16\times 16}$. The input to the Schwarz step is the sparse system matrices $\mathbf{H}^{(P)}, \mathbf{H}^{(Q)} \in \mathbb{R}^{25\times 25}$ and right-hand sides $\mathbf{h}^{(P)}, \mathbf{h}^{(Q)} \in \mathbb{R}^{25\times 1}$. Although $\mathbf{H}^{(P)}, \mathbf{H}^{(Q)}$ are sparse, we treat them as dense matrices in our algorithms. $(\mathbf{H}^{(P)}, \mathbf{h}^{(P)})$ and $(\mathbf{H}^{(Q)}, \mathbf{h}^{(Q)})$ play the role of sub-systems in previous derivations.

$$\mathbf{H}^{(P)} :\overset{\mathcal{P}}{\leftarrow} \begin{bmatrix} \mathbf{A}_{\mathbf{rr}} & \mathbf{A}_{\mathbf{rs}} & \mathbf{A}_{\mathbf{ra}} \\ \mathbf{A}_{\mathbf{sr}} & \mathbf{A}_{\mathbf{ss}}^{(P)} & \mathbf{A}_{\mathbf{sa}} \\ \mathbf{A}_{\mathbf{ar}} & \mathbf{A}_{\mathbf{as}} & \mathbf{A}_{\mathbf{aa}} \end{bmatrix} \quad \mathbf{h}^{(P)} :\overset{\mathcal{P}}{\leftarrow} \begin{bmatrix} \mathbf{v}_{\mathbf{r}} \\ \mathbf{v}_{\mathbf{s}}^{(P)} \\ \mathbf{v}_{\mathbf{a}} \end{bmatrix} \tag{47}$$

$$\mathbf{H}^{(Q)} :\overset{\mathcal{P}}{\leftarrow} \begin{bmatrix} \mathbf{A}_{\mathbf{tt}} & \mathbf{A}_{\mathbf{ts}} & \mathbf{A}_{\mathbf{tb}} \\ \mathbf{A}_{\mathbf{st}} & \mathbf{A}_{\mathbf{ss}}^{(Q)} & \mathbf{A}_{\mathbf{sb}} \\ \mathbf{A}_{\mathbf{bt}} & \mathbf{A}_{\mathbf{bs}} & \mathbf{A}_{\mathbf{bb}} \end{bmatrix} \quad \mathbf{h}^{(Q)} :\overset{\mathcal{P}}{\leftarrow} \begin{bmatrix} \mathbf{v}_{\mathbf{t}} \\ \mathbf{v}_{\mathbf{s}}^{(Q)} \\ \mathbf{v}_{\mathbf{b}} \end{bmatrix} \tag{48}$$

In fact, our algorithm always adopts $\mathbf{H}$ as the direct representation of $\mathbf{A}$, and never explicitly constructs $\mathbf{A}$ as a matrix. Using $\mathbf{H}$ is a much more natural choice to work with parallel linear solvers.

So, our method directly constructs matrix $\mathbf{H}, \mathbf{h}$ in the following way: the entry $\mathbf{H}_{ij}^{(P)}$ represents some interaction or affinity between node i and node j in the subdomain P, under the *boundary-first ordering* shown in Figure 26. For example, it is

common that the patch-wise sub-system is given in the lexicographic sweeping order shown in Figure 27 as a matrix and vector $\mathbf{E}, \mathbf{e}$ of the same size. in this case, we should first map it to the boundary-first ordering using:

$$\mathbf{H} := \mathbf{E}[\boldsymbol{\mu}, \boldsymbol{\mu}], \quad \mathbf{h} := \mathbf{e}[\boldsymbol{\mu}, :] \tag{49}$$

The symbol "$:\overset{\mathcal{P}}{\leftarrow}$" amounts to using some permutation vector in this way.

We will also be able to map it back to the lexicographic sweeping order if necessary using the inverse permutation vector:

$$\mathbf{E} = \mathbf{H}[\boldsymbol{\nu}, \boldsymbol{\nu}], \quad \mathbf{e} = \mathbf{h}[\boldsymbol{\nu}, :] \tag{50}$$

We can store the patch-wise sub-systems in the tensors $\boldsymbol{\alpha}^{(*)} \in \mathbb{R}^{2\times 1\times 25\times 25}$, $\boldsymbol{\beta}^{(*)} \in \mathbb{R}^{2\times 1\times 25\times 1}$:

$$\begin{aligned} \boldsymbol{\alpha}^{(*)}[0,0,:,:] &:= \mathbf{H}^{(P)}, \quad \boldsymbol{\beta}^{(*)}[0,0,:,:] := \mathbf{h}^{(P)} \\ \boldsymbol{\alpha}^{(*)}[1,0,:,:] &:= \mathbf{H}^{(Q)}, \quad \boldsymbol{\beta}^{(*)}[1,0,:,:] := \mathbf{h}^{(Q)} \end{aligned}$$

Then, we call Algorithm 2 for a parallel implementation of the Schwarz step:

Algorithm 2 The parallel Schwarz step. Forward pass.

Require: $\boldsymbol{\alpha}^{(*)} \in \mathbb{R}^{d_1\times d_2\times 25\times 25}, \boldsymbol{\beta}^{(*)} \in \mathbb{R}^{d_1\times d_2\times 25\times 1}$
(d_1, d_2) is $(2^{k/2}, 2^{k/2})$, or $(2, 1)$ in the 2-patch illustrative example.

$$\begin{aligned} \mathbf{X} &:= \boldsymbol{\alpha}[:,:,\eta,\eta] & \mathbf{Y} &:= \boldsymbol{\alpha}[:,:,\eta,\omega] & \mathbf{y} &:= \boldsymbol{\beta}[:,:,\eta,:], \qquad (51) \\ \mathbf{Z} &:= \boldsymbol{\alpha}[:,:,\omega,\eta] & \mathbf{W} &:= \boldsymbol{\alpha}[:,:,\omega,\omega] & \mathbf{w} &:= \boldsymbol{\beta}[:,:,\omega,:]. \qquad (52) \end{aligned}$$

$$\mathbf{D} := \left(\mathbf{X} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{Z}\right) \quad \mathbf{d} := \mathbf{y} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{w} \tag{53}$$

$$\boldsymbol{\alpha} := \mathbf{D}, \quad \boldsymbol{\beta} := \mathbf{d} \tag{54}$$

return $\boldsymbol{\alpha}^{(0)} \in \mathbb{R}^{d_1\times d_2\times 16\times 16}, \boldsymbol{\beta}^{(0)} \in \mathbb{R}^{d_1\times d_2\times 16\times 1}$

Algorithm 2 simultaneously calculates the new sub-systems for all subdomains—$\mathbf{P}$ and $\mathbf{Q}$ in the case of two subdomains. and the sub-systems can be fetched from the tensor $\boldsymbol{\alpha}$.

$$\begin{aligned} \mathbf{P} &= \boldsymbol{\alpha}[0,0,:,:] \\ \mathbf{Q} &= \boldsymbol{\alpha}[1,0,:,:] \end{aligned}$$

Algorithm 2 still applies when there is an $d_1 \times d_2$ array of patches, instead of an 2×1 array of patches that we used as the illustration example. Algorithm 2 conducts batch-wise computation that simultaneously calculates the new sub-systems for all patches.

E.3. Details on Schur step

Now we provide details on the Schur step that merges every two adjacent sub-systems, to supplement §3.2.2.

The Schur step assembles the new system $\mathbf{D}$ by Schur "involuting" the sub-systems $\mathbf{P}, \mathbf{Q}$. The step takes as input the left-hand and right-hand sides $(\mathbf{P}, \mathbf{p})$ and $(\mathbf{Q}, \mathbf{q})$, and outputs a new left-hand and right-hand sides $(\mathbf{D}, \mathbf{d})$.

We store the sub-systems in a four dimensional tensor $\boldsymbol{\alpha}^{(j)}$ that $\boldsymbol{\alpha}^{(j)}[c,d,:,:]$ stores the system matrix at (c,d) in the array of subdomains.

Algorithm 3 The parallel Schwarz step. Backward pass.

Require: $\boldsymbol{\chi}^{(0)}$: tensor of size $(d_1, d_2, 16, 16)$.

$$\mathbf{x} := \mathbf{u}_{\eta} \tag{55}$$

$$\mathbf{u}_{\omega} := \mathbf{W}^{-1}\left(\mathbf{w} - \mathbf{Z}\mathbf{x}\right) \quad \text{(back-substitution)}, \tag{56}$$

$$\boldsymbol{\chi}^{(*)} := \text{ZEROS}(2^{k/2}, 2^{k/2}, 25 \cdot 2^i, 1). \tag{57}$$

$$\boldsymbol{\chi}^{(*)}[:, :, \eta, :] := \mathbf{u}_{\eta} \tag{58}$$

$$\boldsymbol{\chi}^{(*)}[:, :, \omega, :] := \mathbf{u}_{\omega} \tag{59}$$

return $\boldsymbol{\chi}^{(*)}$ tensor of size $(d_1, d_2, 25, 25)$.

$\boldsymbol{\alpha}^{(j)}$ contains all reduced systems. For $j = (2i+1)$, the j-th Schur step takes as input $\boldsymbol{\alpha}^{(j)}$ of size $(2^{k/2-i}, 2^{k/2-i}, 16 \cdot 2^i, 16 \cdot 2^i)$, and outputs $\boldsymbol{\alpha}^{(j+1)}$ of size $(2^{k/2-i-1}, 2^{k/2-i}, 24 \cdot 2^i, 24 \cdot 2^i)$;

For $j = (2i+1), i = 0, 1, ..., k-1$, the j-th Schur step merges subdomains in the horizontal direction.

For $j = (2i+2), i = 0, 1, ..., k-1$, the j-th Schur step merges subdomains in the vertical direction.

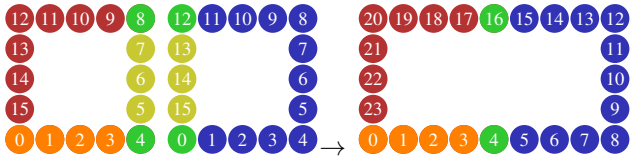


Figure 28. Figure 4: Schur step collapses subdomains P and Q into D.

Horizontal merge Divide the nodes in P into contiguous subsets $\alpha, \beta, \gamma, \delta, \epsilon$, such that

$$\alpha = [0, 1, 2, 3], \beta = [4], \gamma = [5, 6, 7], \delta = [8], \epsilon = [9, 10, 11, 12, 13, 14, 15]. \tag{60}$$

Divide the nodes in Q into contiguous subsets $\kappa, \lambda, \mu, \nu$, such that:

$$\kappa = [0], \lambda = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], \mu = [12], \nu = [13, 14, 15]. \tag{61}$$

Under the indexing in Figure 4, D's boundary consists of the nodes corresponding to $\alpha, \beta, \lambda, \delta, \epsilon$. β, κ represent the same node after merging, so are δ, μ.

It makes sense to define their indices in the merged domain.

$$\hat{\alpha} = [0, 1, 2, 3], \hat{\beta} = [4], \hat{\lambda} = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], \hat{\delta} = [16], \hat{\epsilon} = [17, 18, 19, 20, 21, 22, 23]. \tag{62}$$

Note that the values of these indices depend on the shape of the patches: we only give their values in the first Schur step.

The node grouping implies partitioning the matrix $\mathbf{P}, \mathbf{Q}$ into submatrices, by dividing the rows and columns into subsets.

$$\left(\begin{bmatrix}\begin{bmatrix}\mathbf{P}_{\alpha\alpha} & \mathbf{P}_{\alpha\beta} & \mathbf{0}_{\alpha\lambda} & \mathbf{P}_{\alpha\delta} & \mathbf{P}_{\alpha\epsilon}\\ \mathbf{P}_{\beta\alpha} & \mathbf{P}_{\beta\beta} & \mathbf{0}_{\beta\lambda} & \mathbf{P}_{\beta\delta} & \mathbf{P}_{\beta\epsilon}\\ \mathbf{0}_{\lambda\alpha} & \mathbf{0}_{\lambda\beta} & \mathbf{0}_{\lambda\lambda} & \mathbf{0}_{\lambda\delta} & \mathbf{0}_{\lambda\epsilon}\\ \mathbf{P}_{\delta\alpha} & \mathbf{P}_{\delta\beta} & \mathbf{0}_{\delta\lambda} & \mathbf{P}_{\delta\delta} & \mathbf{P}_{\delta\epsilon}\\ \mathbf{P}_{\epsilon\alpha} & \mathbf{P}_{\epsilon\beta} & \mathbf{0}_{\epsilon\lambda} & \mathbf{P}_{\epsilon\delta} & \mathbf{P}_{\epsilon\epsilon}\end{bmatrix} & \begin{bmatrix}\mathbf{P}_{\alpha\gamma}\\ \mathbf{P}_{\beta\gamma}\\ \mathbf{0}_{\lambda\gamma}\\ \mathbf{P}_{\delta\gamma}\\ \mathbf{P}_{\epsilon\gamma}\end{bmatrix}\\ \begin{bmatrix}\mathbf{P}_{\gamma\alpha} & \mathbf{P}_{\gamma\beta} & \mathbf{0}_{\gamma\lambda} & \mathbf{P}_{\gamma\delta} & \mathbf{P}_{\gamma\epsilon}\end{bmatrix} & \begin{bmatrix}\mathbf{P}_{\gamma\gamma}\end{bmatrix}\end{bmatrix}\right.$$
$$\left.+\begin{bmatrix}\begin{bmatrix}\mathbf{0}_{\alpha\alpha} & \mathbf{0}_{\alpha\beta} & \mathbf{0}_{\alpha\lambda} & \mathbf{0}_{\alpha\delta} & \mathbf{0}_{\alpha\epsilon}\\ \mathbf{0}_{\beta\alpha} & \mathbf{Q}_{\kappa\kappa} & \mathbf{Q}_{\kappa\lambda} & \mathbf{Q}_{\kappa\mu} & \mathbf{0}_{\beta\epsilon}\\ \mathbf{0}_{\lambda\alpha} & \mathbf{Q}_{\lambda\kappa} & \mathbf{Q}_{\lambda\lambda} & \mathbf{Q}_{\lambda\mu} & \mathbf{0}_{\lambda\epsilon}\\ \mathbf{0}_{\delta\alpha} & \mathbf{Q}_{\mu\kappa} & \mathbf{Q}_{\mu\lambda} & \mathbf{Q}_{\mu\mu} & \mathbf{0}_{\delta\epsilon}\\ \mathbf{0}_{\epsilon\alpha} & \mathbf{0}_{\epsilon\beta} & \mathbf{0}_{\epsilon\lambda} & \mathbf{0}_{\epsilon\delta} & \mathbf{0}_{\epsilon\epsilon}\end{bmatrix} & \begin{bmatrix}\mathbf{0}_{\alpha\gamma}\\ \mathbf{Q}_{\kappa\nu}\mathbf{J}\\ \mathbf{Q}_{\lambda\nu}\mathbf{J}\\ \mathbf{Q}_{\mu\nu}\mathbf{J}\\ \mathbf{0}_{\epsilon\gamma}\end{bmatrix}\\ \begin{bmatrix}\mathbf{0}_{\gamma\alpha} & \mathbf{J}^\intercal\mathbf{Q}_{\nu\kappa} & \mathbf{J}^\intercal\mathbf{Q}_{\nu\lambda} & \mathbf{J}^\intercal\mathbf{Q}_{\nu\mu} & \mathbf{0}_{\gamma\epsilon}\end{bmatrix} & \begin{bmatrix}\mathbf{J}^\intercal\mathbf{Q}_{\nu\nu}\mathbf{J}\end{bmatrix}\end{bmatrix}\right)\begin{bmatrix}\mathbf{u}_\alpha\\ \mathbf{u}_\beta\\ \mathbf{u}_\lambda\\ \mathbf{u}_\delta\\ \mathbf{u}_\epsilon\\ \mathbf{u}_\gamma\end{bmatrix}=\begin{bmatrix}\mathbf{p}_\alpha\\ \mathbf{p}_\beta+\mathbf{q}_\kappa\\ \mathbf{q}_\lambda\\ \mathbf{p}_\delta+\mathbf{q}_\mu\\ \mathbf{p}_\epsilon\\ \mathbf{p}_\gamma+\mathbf{J}^\intercal\mathbf{q}_\nu\end{bmatrix} \tag{63}$$

is the linear system for the joint domain D. Since β, κ represent the same node, they correspond to the same row/column; the same applies to δ, μ. Note that γ represents the same set of nodes as ν but in reverse order. Let $\mathbf{J} \equiv \mathbf{J}^\intercal$ to be the reverse permutation matrix (see §E), whose action is to reverse the rows (or columns) of a matrix when being multiplied with from left (or right).

E.4. Final algorithm

Algorithm 1 specifies the overall algorithm with both the numerical factorization and back substitution stages.

The numerical factorization variant can be implemented by only keeping the computations involving $\boldsymbol{\alpha}$, and returns and saves the values of $\boldsymbol{\alpha}^{(0)}$, $\boldsymbol{\alpha}^{(1)}$, ..., $\boldsymbol{\alpha}^{(k)}$ for the use of the back substitution step.

The back substitution variant can be realized by only keeping the lines involving $\boldsymbol{\beta}, \boldsymbol{\chi}$, and using the $\boldsymbol{\alpha}^{(0)}$, $\boldsymbol{\alpha}^{(1)}$, ..., $\boldsymbol{\alpha}^{(k)}$ cached in the numerical factorization step.

Algorithm 4 The parallel Schur step, j-th step, $j = (2i+1)$. Forward pass. Horizontal.

Require: $\boldsymbol{\alpha}^{(j)}$ of size $(2^{k/2-i}, 2^{k/2-i}, 16 \cdot 2^i, 16 \cdot 2^i)$,
Require: $\boldsymbol{\beta}^{(j)}$ of size $(2^{k/2-i}, 2^{k/2-i}, 16 \cdot 2^i, 1)$.
$\mathbf{P}, \mathbf{Q}$ are fetched in batches from $\alpha^{(j)}$ for all red and blue subdomains.

$$\mathbf{P} := \boldsymbol{\alpha}^{(j)}[0 :: 2, :, :, :], \quad \mathbf{p} := \boldsymbol{\beta}^{(j)}[0 :: 2, :, :, :] \tag{64}$$

$$\mathbf{Q} := \boldsymbol{\alpha}^{(j)}[1 :: 2, :, :, :], \quad \mathbf{q} := \boldsymbol{\beta}^{(j)}[1 :: 2, :, :, :] \tag{65}$$

$$\begin{bmatrix} \mathbf{P}_{\alpha\alpha} & \mathbf{P}_{\alpha\beta} & \mathbf{P}_{\alpha\gamma} & \mathbf{P}_{\alpha\delta} & \mathbf{P}_{\alpha\epsilon} \\ \mathbf{P}_{\beta\alpha} & \mathbf{P}_{\beta\beta} & \mathbf{P}_{\beta\gamma} & \mathbf{P}_{\beta\delta} & \mathbf{P}_{\beta\epsilon} \\ \mathbf{P}_{\gamma\alpha} & \mathbf{P}_{\gamma\beta} & \mathbf{P}_{\gamma\gamma} & \mathbf{P}_{\gamma\delta} & \mathbf{P}_{\gamma\epsilon} \\ \mathbf{P}_{\delta\alpha} & \mathbf{P}_{\delta\beta} & \mathbf{P}_{\delta\gamma} & \mathbf{P}_{\delta\delta} & \mathbf{P}_{\delta\epsilon} \\ \mathbf{P}_{\epsilon\alpha} & \mathbf{P}_{\epsilon\beta} & \mathbf{P}_{\epsilon\gamma} & \mathbf{P}_{\epsilon\delta} & \mathbf{P}_{\epsilon\epsilon} \end{bmatrix} := \mathbf{P}, \quad \begin{bmatrix} \mathbf{p}_{\alpha} \\ \mathbf{p}_{\beta} \\ \mathbf{p}_{\gamma} \\ \mathbf{p}_{\delta} \\ \mathbf{p}_{\epsilon} \end{bmatrix} := \mathbf{p}, \tag{66}$$

$$\begin{bmatrix} \mathbf{Q}_{\kappa\kappa} & \mathbf{Q}_{\kappa\lambda} & \mathbf{Q}_{\kappa\mu} & \mathbf{Q}_{\kappa\nu} \\ \mathbf{Q}_{\lambda\kappa} & \mathbf{Q}_{\lambda\lambda} & \mathbf{Q}_{\lambda\mu} & \mathbf{Q}_{\lambda\nu} \\ \mathbf{Q}_{\mu\kappa} & \mathbf{Q}_{\mu\lambda} & \mathbf{Q}_{\mu\mu} & \mathbf{Q}_{\mu\nu} \\ \mathbf{Q}_{\nu\kappa} & \mathbf{Q}_{\nu\lambda} & \mathbf{Q}_{\nu\mu} & \mathbf{Q}_{\nu\nu} \end{bmatrix} := \mathbf{Q}, \quad \begin{bmatrix} \mathbf{q}_{\kappa} \\ \mathbf{q}_{\lambda} \\ \mathbf{q}_{\mu} \\ \mathbf{q}_{\nu} \end{bmatrix} := \mathbf{q}. \tag{67}$$

$$\mathbf{X} := \begin{bmatrix} \mathbf{P}_{\alpha\alpha} & \mathbf{P}_{\alpha\beta} & \mathbf{0}_{\alpha\lambda} & \mathbf{P}_{\alpha\delta} & \mathbf{P}_{\alpha\epsilon} \\ \mathbf{P}_{\beta\alpha} & \mathbf{P}_{\beta\beta} + \mathbf{Q}_{\kappa\kappa} & \mathbf{Q}_{\kappa\lambda} & \mathbf{P}_{\beta\delta} + \mathbf{Q}_{\kappa\mu} & \mathbf{P}_{\beta\epsilon} \\ \mathbf{0}_{\lambda\alpha} & \mathbf{Q}_{\lambda\kappa} & \mathbf{Q}_{\lambda\lambda} & \mathbf{Q}_{\lambda\mu} & \mathbf{0}_{\lambda\epsilon} \\ \mathbf{P}_{\delta\alpha} & \mathbf{P}_{\delta\beta} + \mathbf{Q}_{\mu\kappa} & \mathbf{Q}_{\mu\lambda} & \mathbf{P}_{\delta\delta} + \mathbf{Q}_{\mu\mu} & \mathbf{P}_{\delta\epsilon} \\ \mathbf{P}_{\epsilon\alpha} & \mathbf{P}_{\epsilon\beta} & \mathbf{0}_{\epsilon\lambda} & \mathbf{P}_{\epsilon\delta} & \mathbf{P}_{\epsilon\epsilon} \end{bmatrix} \tag{68}$$

$$\mathbf{Y} := \begin{bmatrix} \mathbf{P}_{\alpha\gamma} \\ \mathbf{P}_{\beta\gamma} + \mathbf{Q}_{\kappa\nu}\mathbf{J} \\ \mathbf{Q}_{\lambda\nu}\mathbf{J} \\ \mathbf{P}_{\delta\gamma} + \mathbf{Q}_{\mu\nu}\mathbf{J} \\ \mathbf{P}_{\epsilon\gamma} \end{bmatrix} \quad \mathbf{y} := \begin{bmatrix} \mathbf{p}_{\alpha} \\ \mathbf{p}_{\beta} + \mathbf{q}_{\kappa} \\ \mathbf{q}_{\lambda} \\ \mathbf{p}_{\delta} + \mathbf{q}_{\mu} \\ \mathbf{p}_{\epsilon} \end{bmatrix} \tag{69}$$

$$\mathbf{Z} := \begin{bmatrix} \mathbf{P}_{\gamma\alpha} & \mathbf{P}_{\gamma\beta} + \mathbf{J}^{\intercal}\mathbf{Q}_{\nu\kappa} & \mathbf{J}^{\intercal}\mathbf{Q}_{\nu\lambda} & \mathbf{P}_{\gamma\delta} + \mathbf{J}^{\intercal}\mathbf{Q}_{\nu\mu} & \mathbf{P}_{\gamma\epsilon} \end{bmatrix} \tag{70}$$

$$\mathbf{W} := \begin{bmatrix} \mathbf{P}_{\gamma\gamma} + \mathbf{J}^{\intercal}\mathbf{Q}_{\nu\nu}\mathbf{J} \end{bmatrix} \quad \mathbf{w} := \begin{bmatrix} \mathbf{p}_{\gamma} + \mathbf{J}^{\intercal}\mathbf{q}_{\nu} \end{bmatrix} \tag{71}$$

$$\mathbf{D} := \left(\mathbf{X} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{Z}\right) \quad \mathbf{d} := \mathbf{y} - \mathbf{Y}\mathbf{W}^{-1}\mathbf{w} \tag{72}$$

$$\boldsymbol{\alpha} := \mathbf{D}, \quad \boldsymbol{\beta} := \mathbf{d} \tag{73}$$

return $\boldsymbol{\alpha}^{(j+1)}$ of size $(2^{k/2-i-1}, 2^{k/2-i}, 24 \cdot 2^i, 24 \cdot 2^i)$,
$\boldsymbol{\beta}^{(j+1)}$ of size $(2^{k/2-i-1}, 2^{k/2-i}, 24 \cdot 2^i, 1)$

Algorithm 5 The parallel j-th Schur step, $j = (2i+1)$. Backward pass. Horizontal.

Require: $\boldsymbol{\chi}^{(j+1)}$ of size $(2^{k/2-i}, 2^{k/2-i}, 24 \cdot 2^i, 1)$

$$\begin{bmatrix} \mathbf{u}_{\hat{\alpha}} \\ \mathbf{u}_{\hat{\beta}} \\ \mathbf{u}_{\hat{\lambda}} \\ \mathbf{u}_{\hat{\delta}} \\ \mathbf{u}_{\hat{\epsilon}} \end{bmatrix} := \boldsymbol{\beta}^{(j)} \text{ or } \begin{aligned} \mathbf{u}_{\hat{\alpha}} &:= \boldsymbol{\beta}^{(j)}[:,:,\hat{\alpha},:] \\ \mathbf{u}_{\hat{\beta}} &:= \boldsymbol{\beta}^{(j)}[:,:,\hat{\beta},:] \\ \mathbf{u}_{\hat{\lambda}} &:= \boldsymbol{\beta}^{(j)}[:,:,\hat{\lambda},:] \\ \mathbf{u}_{\hat{\delta}} &:= \boldsymbol{\beta}^{(j)}[:,:,\hat{\delta},:] \\ \mathbf{u}_{\hat{\epsilon}} &:= \boldsymbol{\beta}^{(j)}[:,:,\hat{\epsilon},:] \end{aligned} \tag{74}$$

$$\mathbf{x} := \begin{bmatrix} \mathbf{u}_{\hat{\alpha}} \\ \mathbf{u}_{\hat{\beta}} \\ \mathbf{u}_{\hat{\lambda}} \\ \mathbf{u}_{\hat{\delta}} \\ \mathbf{u}_{\hat{\epsilon}} \end{bmatrix} \tag{75}$$

$$\boldsymbol{\chi}^{(j)} := \text{ZEROS}(2^{k/2-i}, 2^{k/2-i}, 16 \cdot 2^i, 1). \tag{76}$$

$$\mathbf{u}^{\gamma} := \mathbf{W}^{-1}(\mathbf{w} - \mathbf{Z}\mathbf{x}) \quad \text{(back-substitution)}, \tag{77}$$

where we use upper script to emphasize that it does not correspond to slice into a vector.

$$\boldsymbol{\chi}^{(j)}[0::2,:,:,:] := \begin{bmatrix} \mathbf{u}_{\hat{\alpha}} \\ \mathbf{u}_{\hat{\beta}} \\ \mathbf{u}^{\gamma} \\ \mathbf{u}_{\hat{\delta}} \\ \mathbf{u}_{\hat{\epsilon}} \end{bmatrix} \text{ or } \begin{aligned} \boldsymbol{\chi}^{(j)}[0::2,:,\alpha,:] &:= \mathbf{u}_{\hat{\alpha}} \\ \boldsymbol{\chi}^{(j)}[0::2,:,\beta,:] &:= \mathbf{u}_{\hat{\beta}} \\ \boldsymbol{\chi}^{(j)}[0::2,:,\gamma,:] &:= \mathbf{u}^{\nu}, \\ \boldsymbol{\chi}^{(j)}[0::2,:,\delta,:] &:= \mathbf{u}_{\hat{\delta}} \\ \boldsymbol{\chi}^{(j)}[0::2,:,\epsilon,:] &:= \mathbf{u}_{\hat{\epsilon}} \end{aligned} \quad \begin{aligned} \boldsymbol{\chi}^{(j)}[1::2,:,\kappa,:] &:= \mathbf{u}_{\hat{\beta}} \\ \boldsymbol{\chi}^{(j)}[1::2,:,\lambda,:] &:= \mathbf{u}_{\hat{\lambda}} \\ \boldsymbol{\chi}^{(j)}[1::2,:,\mu,:] &:= \mathbf{u}_{\hat{\delta}} \\ \boldsymbol{\chi}^{(j)}[1::2,:,\nu,:] &:= \mathbf{J}^{\intercal}\mathbf{u}^{\gamma} \end{aligned} \tag{78}$$

return $\boldsymbol{\chi}^{(j)}$ of size $(2^{k/2-i}, 2^{k/2-i}, 16 \cdot 2^i, 1)$.

E.5. Graph algorithm perspective

Gaussian elimination, including our method, can be simply understood as a *graph algorithm* that removes nodes from a graph, while updating the edge weights accordingly. The graph algorithm perspective can make the overall algorithm significantly easier to understand.

Weighted graph. The matrix $\mathbf{A}$ plays the same role as the adjacency matrix in graph theory. As in Figure 21, initially each pixel in the image is a node in the graph, and two nodes are connected by an edge with weights $\mathbf{A}_{ij}$, if the pixels i and j are adjacent (as defined in Sec. 2).

Node elimination Sequential Gaussian elimination removes nodes one by one from the graph. When removing a node k from the current graph, we only need to apply a simple modification to matrix $\mathbf{A}$: for all pair of nodes i, j that are both adjacent to k, update $\mathbf{A}_{ij}$ by subtracting the term $\mathbf{A}_{ik}\mathbf{A}_{kk}^{-1}\mathbf{A}_{kj}$ from it.

$$\begin{aligned} &\mathbf{A}_{ij} \leftarrow \mathbf{A}_{ij} - \mathbf{A}_{ik}\mathbf{A}_{kk}^{-1}\mathbf{A}_{kj}, \quad \forall i, j \\ &\mathbf{A}_{ik} \leftarrow 0, \quad \forall i \\ &\mathbf{A}_{ki} \leftarrow 0, \quad \forall i \\ &\mathbf{A}_{kk} \leftarrow 0. \end{aligned} \qquad \text{(weights updating)} \tag{79}$$

Since $\mathbf{A}_{k,:}, \mathbf{A}_{:,k}$—the row and column that correspond to node k become zero after the step, we can actually delete them from the matrix; note we will need to relabel and nodes after the deletion. That is, when deleting a node k, the indirect influence of node j on i via k, is attributed through a direct influence of node j on i.

While the sequential Gaussian elimination is sufficient to solve the linear system and is mathematical equivalent to our method, it is inefficient. As an improvement, we can remove multiple nodes in a single elimination, to leverage dense CUDA BLAS kernel.

Block (multiple nodes) elimination The updating formula generalizes to the case when i, j, k each is not a single node, but each consists of a set of nodes. Then, we have block Gaussian elimination: first divide the domain into three sets of nodes as r, s, and t, such that any node in r is not connected to any node in t as separated by s. The 3×3 block:

$$\begin{bmatrix} \mathbf{A}_{rr} & \mathbf{A}_{rs} & 0 = \mathbf{A}_{rt} \\ \mathbf{A}_{sr} & \mathbf{A}_{ss} & \mathbf{A}_{st} \\ 0 = \mathbf{A}_{tr} & \mathbf{A}_{ts} & \mathbf{A}_{tt} \end{bmatrix}$$

becomes the 2×2 block:

$$\begin{bmatrix} \mathbf{A}_{ss} - \mathbf{A}_{sr}\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs} & \mathbf{A}_{st} \\ \mathbf{A}_{ts} & \mathbf{A}_{tt} \end{bmatrix}$$

Namely, the update rule is to subtract the adjustment term $\mathbf{A}_{sr}\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}$.

$$\begin{aligned} &\mathbf{A}_{rr} \leftarrow \mathbf{A}_{rr} - \mathbf{A}_{sr}\mathbf{A}_{rr}^{-1}\mathbf{A}_{rs}, \quad \forall i, j \\ &\mathbf{A}_{rs} \leftarrow \mathbf{0}, \quad \forall i \\ &\mathbf{A}_{sr} \leftarrow \mathbf{0}, \quad \forall i \\ &\mathbf{A}_{rr} \leftarrow \mathbf{0}. \end{aligned} \qquad \text{(weights updating)} \tag{80}$$

Parallel block elimination For efficiency, we apply many Schur complements in parallel. In §3, we simply do two block Gaussian elimination at the same time. Then our algorithm is basically a parallel block Gaussian elimination in which many groups of nodes (marked in yellow in Figure 3) are removed by concurrently subtracting many adjustment terms. The major effort in the code is "index-tracking": carefully track what the indices of remaining nodes become after some nodes are removed.

E.6. Illustration on a 3x7 image

An example with detailed visualization Let us first consider a smaller 3×7 image for the convenience of illustration, as shown in Figure 23. Each pixel corresponds to a node in the graph and is assigned with an index $i \in \{0, 1, ..., 21-1\}$. Two nodes i and j, where $i, j \in \{0, 1, ..., 21-1\}$, are connected by an edge if the 3×3 kernel centered at one node will cover the other node. $\mathbf{A}_{ij} = 0$ if node i and node j are not connected by an edge in the graph. Each of the color includes the following subsets of nodes: $\mathbf{r} = [0, 1, 2, 7, 14, 15, 16]$, $\mathbf{s} = [3, 10, 17]$, $\mathbf{t} = [4, 5, 6, 13, 18, 19, 20]$, $\mathbf{a} = [8, 9]$, $\mathbf{b} = [11, 12]$.

The original sparse linear system, $\mathbf{A} \in \mathbb{R}^{21 \times 21}$:

$$\sum_{j=0}^{21-1} \mathbf{A}_{ij}\mathbf{u}_j = \mathbf{v}_i, \forall i \in \{0, 1, ..., 21-1\}. \tag{81}$$

Figure 29. The connectivity graph resulted from the use of the 3×3 kernel size, for a 3×7 image under the lexicographic sweeping order.

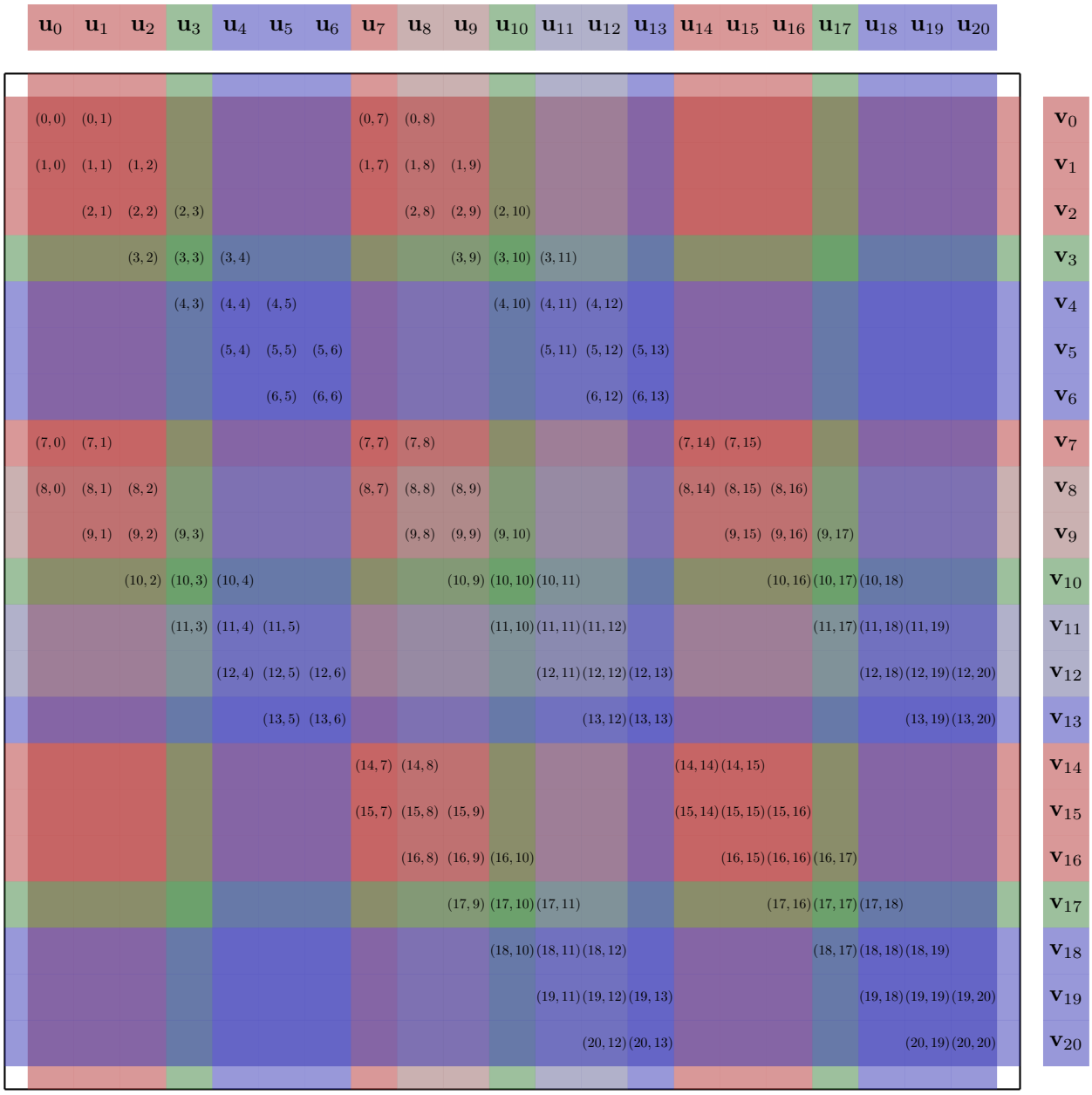


Figure 30. Same as Figure 24 but for a 3×7 image.

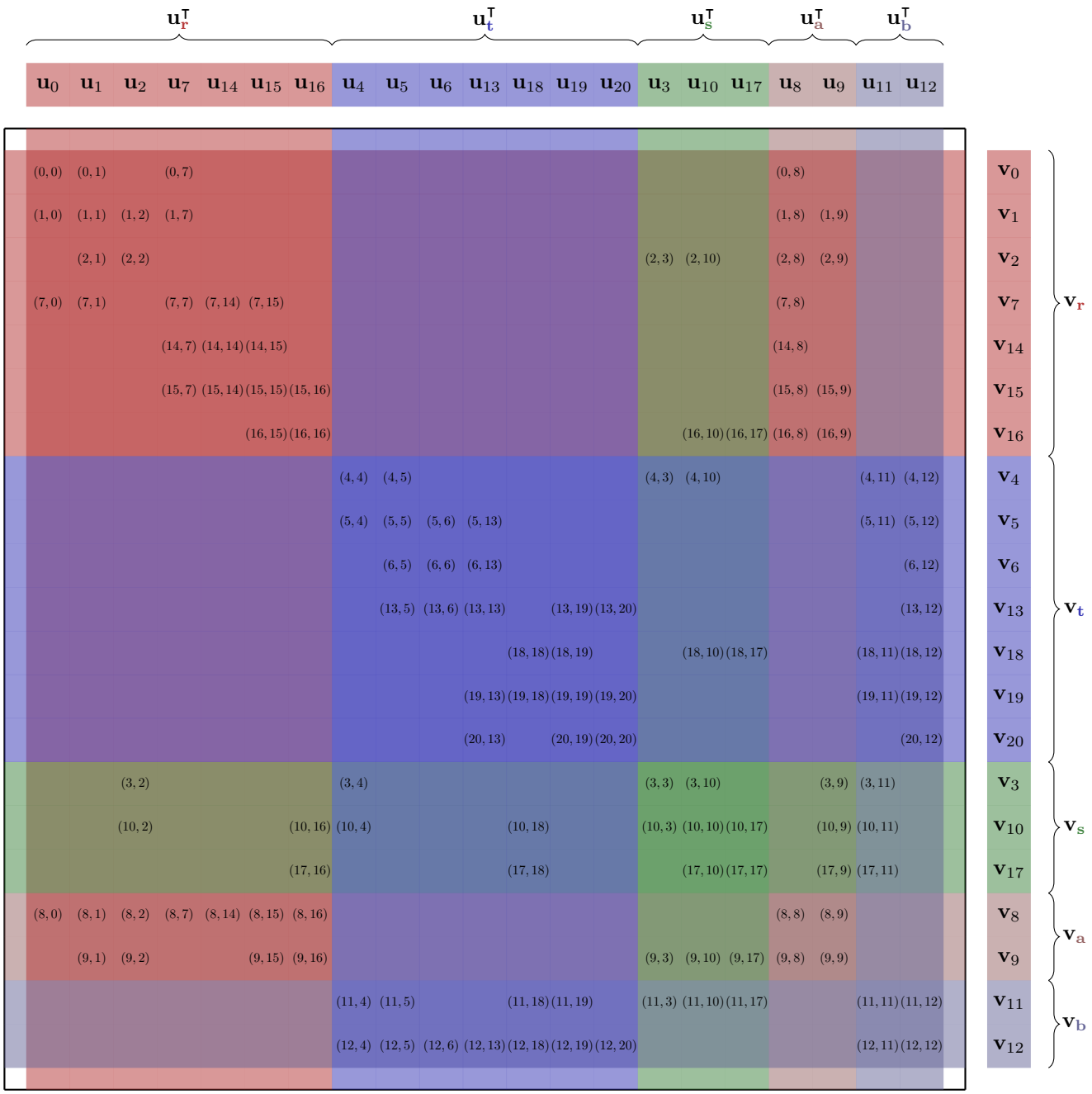


Figure 31. Same as Figure 25 but for a 3×7 image.

F. Extra Information

Please note that our use of the term "involution" is not related to the mathematical concept that refers to a self-inverse function as an "involution."

F.1. Details on FEM and PDE discretization: addibility of parallel linear elements

Although where the matrix $\mathbf{A}$ or tensor $\boldsymbol{\alpha}$ comes from does not matter to apply our sparse solver, as long as $\mathbf{A}$ is invertible, we supplement with implementation details for reproducibility. Despite that theoretically one could use, e.g., bilinear bases as well, in this paper, we use the first-order piecewise-linear finite element method (Wang et al., 2023) to discretize the partial differential equations. There are multiple advantages: 1) The discretized system preserved certain algebraic properties of the continuous PDEs that are particularly valuable in geometry tasks (Wang et al., 2023). 2) Most importantly, the **addibility of parallel linear elements**. It is easy to verify that if each of the patch (i, j) puts in $\boldsymbol{\alpha}^{(*)}[i, j, :, :]$ the Laplacian matrix $\mathbf{L}^{(i,j)}$ that is discretized with the Neumann boundary condition, then **assembling together these patch sub-matrices yields the Laplacian matrix $\mathbf{L}$ for the whole image domains**, under the same Neumann boundary condition. Here "assembling patch sub-matrices" means summing together the sub-matrices to a large sparse matrix $\mathbf{L}$ by indexing into the rows and columns of $\mathbf{L}$ that the patch nodes correspond to. In other words, we know it holds that:

$$\mathbf{L} \equiv \sum_{(i,j)} (\mathbf{J}^{(i,j)})^\intercal \mathbf{L}^{(i,j)} \mathbf{J}^{(i,j)} \in \mathbb{R}^{n \times n} \tag{82}$$

where $\mathbf{L}^{(i,j)} \in \mathbb{R}^{25 \times 25}$ comes from the Laplacian matrix discretizing an elliptic PDE with the Neumann boundary condition, and $\mathbf{L} \in \mathbb{R}^{n \times n}$ comes from discretizing the same PDE & boundary condition but over the entire image domain. Here $\mathbf{J}^{(i,j)} \in \mathbb{R}^{25 \times n}$ is the binary matrix, playing the role of putting the patch sub-system $\mathbf{L}^{(i,j)}$ in the correct rows and columns of the large sparse system.

The fact can be verified by noticing that using first-order piecewise-linear FEM (Wang et al., 2023) to discretize the elliptic PDE (3) which stores the anisotropic tensor $\mathbf{C}(\mathbf{x})$ as a 2×2 matrix per triangle, each triangle will contribute to the entries of $\mathbf{A}$ independently using information only within that triangle. The fact that patch sub-systems can be "seamlessly" added together thanks to the use of the Neumann boundary condition—the *natural boundary condition*. Using certain high-order finite element methods might necessitate information exchange between adjacent patches, which is still doable but requires extra care in the implementation.

A double covered FEM mesh layout for images. During the tessellation of the regular grid as triangular meshes, to avoid bias in choosing the orientation of the long edge of the triangles, we simply use a double-covered triangular mesh—each square has four cross-edge triangles whose edges connect pixels.

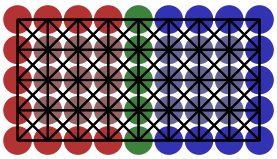


Figure 32. Our double-covered piecewise-linear triangle mesh layout.

F.2. Issues with incorporating iterative solvers in deep learning

In this section, we identify a key obstacle preventing neural architectures from adopting linear solvers: the lack of **SCHWARZ** such as our method—a **S**parse solver that is **C**onsistent-performance, **H**yperspeed, in-the-**W**ild, **A**ccurate, **R**obust, and **Z**ero-parameter. We explain why indirect, a.k.a. iterative, solvers fall short of these goals.

Deploying iterative solvers appropriately comes with a tedious workflow, requiring users in the loop with PhD-level expertise in numerical analysis, PDEs, GPU optimization, or image processing. Iterative solvers require problem-specific solvers with many parameters, preconditioners which also have parameters. Although solvers such as LSMR, LSQR seem to be applicable to all matrices, we observe that they can be inefficient compared to our method in §4.

The large diversity of possible PDEs makes it difficult for one user to master all the methods well and implement the entire arsenal of numerical PDEs. Especially, we lack an automatic pipeline to analyze the type of $\mathbf{A}$ to deploy the appropriate

iterative solvers, and to automatically set the parameters, preconditioner, and the parameters of the preconditioner for good performance by analyzing entries of $\mathbf{A}$. In the setup of PDE learning, the exact form of PDE is unknown. To algorithmically search for the PDE (that is, the matrix A represents it), one will have to implement iterative solvers for *all* possible PDEs, which is an infeasible task for unstudied PDEs. Iterative solvers can perform poorly for highly singularly, stiff, anisotropic coefficients, or oscillatory solutions such as the Helmholtz equation under a relatively large frequency (Ernst & Gander, 2011).

Even restricting to solving some well-studied PDEs, so there are efficient iterative numerical schemes, the best values of these parameters also depend on the entries and properties of $\mathbf{A}$. The many parameters in iterative solvers must be appropriately set to yield a correct result: sigma shift values, preconditioning schemes, and parameters such as error tolerance, maximum iterations. Below is an example taken from the official example of Nvidia AMGX (Naumov et al., 2015).

```
cfg = pyamgx.Config().create_from_dict({
    "config_version": 2,
    "determinism_flag": 1,
    "exception_handling" : 1,
    "solver": {
        "monitor_residual": 1,
        "solver": "FGMRES",
        "convergence": "RELATIVE_INI_CORE",
        "tolerance": 1e-6,
        "max_iters": 10000,
        "gmres_n_restart": 20,
        "norm": "L2",
        "preconditioner": {
            "solver": "AMG",
            "algorithm": "CLASSICAL",
            "max_iters": 2,
            "presweeps": 1,
            "postsweeps": 1,
            "cycle" : "V",
        },
    },
})
```

That said, even to solve a single instance of PDE with iterative solvers, the user may have to manually repeat the trial-and-error steps until convergence: tweak the parameters and then run the iterative solver with them; if not converged or the result is not desirable, the user has to go back to change the parameters. The tedious user-in-the-loop workflows prevent training with a solver in the loop on a massive amount of data in scientific machine learning in a way similar to the practice of computer vision and natural language processing.

Unpredictable runtime: When the input problem changes, the runtime can change dramatically, which is a critical drawback for interactive applications such as self-driving cars. Conservative parameter settings significantly slow down iterative solvers, and aggressive parameter settings risk insufficient iterations or unconverged results.

Catastrophic failures can incur when applying unsuitable iterative solvers: it can obtain totally wrong results to ruin trained models, calling for user intervention: manually fixes for certain training examples and restoration from trained checkpoints.

F.3. Differentiable sparse solvers: widely desired, yet absent

Here are links to examples of community requests and discussions on differentiable linear solvers. Despite of the frequent requests, linear solvers have only very limited support in mainstream deep learning packages, which perhaps makes sense because existing sparse linear solvers are quite limited in applicability and inefficient compared to other differentiable modules and our method. Porting existing solvers to, e.g., PyTorch makes them the sole bottleneck in the training pipelines.

https://github.com/pytorch/pytorch/issues/58828,

https://github.com/pytorch/pytorch/issues/108977,

https://github.com/pytorch/pytorch/issues/69538,

https://discuss.pytorch.org/t/linear-solver-for-sparse-matrices/200289/6,

https://github.com/tensorflow/addons/pull/2396,

https://github.com/tensorflow/addons/issues/2387,

https://github.com/tensorflow/tensorflow/issues/27380,

https://discuss.pytorch.org/t/solving-ax-b-for-sparse-tensors-preferably-with-backward/102331,

https://discuss.pytorch.org/t/differentiable-sparse-linear-solver-with-cupy-backend-unsupported-tensor-layout-sparse-in-gradcheck/141309.