© 2022 Nicolas Nytko

LEARNING AGGREGATES AND INTERPOLATION FOR ALGEBRAIC MULTIGRID

BY

NICOLAS NYTKO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
with a concentration in Computational Science and Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Advisers:

Associate Professor Matthew West
Professor Luke Olson

ABSTRACT

Algebraic multigrid solvers are among the quickest for finding solutions to large, sparse linear systems of equations such as those arising from the discretization of partial differential equations (PDEs). Their implementation, however, often relies on constructing a coarse grid and transfer operators through the use of heuristics or other approximations; overall convergence depends on a judicious selection of parameters. In this thesis, we evaluate the use of neural networks to select such a coarse grid and transfer operators for isotropic and anisotropic diffusion problems. We show how graph neural networks can be used to output a tentative set of node groupings, followed by interpolation construction analogous to smoothed-aggregation multigrid. Difficulties in training such neural networks due to the lack of gradient information is addressed through the use of genetic evolution strategies. Finally, performance of the learned multigrid solver is compared to off-the-shelf methods from established algebraic multigrid packages.

ACKNOWLEDGMENTS

Thank you Professors Matt West, Luke Olson, and Scott MacLachlan for your endless guidance and help over these two years. I appreciate all the patience and assistance given for all the (sometimes quite obvious) questions asked. You three have also dispensed with such invaluable advice, like how to achieve proper kerning with LaTeXand to always save figures as PDF to avoid being publicly humiliated.

I thank my parents and family members for the love and support in me pursuing a graduate degree, even if they have almost no idea what it is that I work on and undoubtedly will be even more confused by reading this thesis.

I would also like to acknowledge my undergraduate Professor Mariana Silva; I thank her (or, maybe, blame her) for my developing an interest in numerical methods and the broad field of numerical analysis. I really do owe my early graduate career to her and without her nor her guidance I wouldn't currently be in the position that I am.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Iterative and multigrid methods have a long, well established history for the rapid computation of solutions to linear systems. Some of the earliest relaxation methods were first introduced in the mid-1800s by Gauss to solve linear least-squares systems arising from geodetics [1]; linear systems of approximately 20-40 unknowns were solved to triangulate land surveying data. Other simple schemes such as modern Gauss-Seidel and Jacobi relaxation were also introduced during this time to solve normal equations from similar least-squares problems. Applications of such methods towards PDEs were not explored until a paper in 1911 by Richardson [2], where he introduced both a novel way of numerically discretizing several elliptic PDEs through finite differences and an iterative method (aptly named Richardson iteration) for solving these problems.

The development of multigrid methods as a general branch of solvers is heavily rooted in the solution of numerical PDEs. The first precursor to what could be considered a multigrid solver was introduced as a relaxation scheme by Fedorenko in the early 1960s for solving Poisson's equation on a grid [3, 4]; Fedorenko established the basic foundation of a two-grid hierarchy as well as some of the theory behind targeting low-frequency error modes with the method itself. Multigrid as an individual brand of solver really started to take off in 1977 with the release of Achi Brandt's seminal paper on the topic [5], formally defining all parts of the multigrid process including inter-grid transfer operators, analysis of convergence, and introduction to unstructured domains. Multigrid was further developed into Algebraic Multigrid (AMG) by Brandt et al. [6], requiring no geometric intuition about the problem itself; then into Smoothed Aggregation AMG by Vaněk et al. [7] in 1995. Research on new techniques on smoothed aggregation is still an active area of development today.

Artificial intelligence and machine learning, on the other hand, while not deep in its history is arguably one of the most popular research areas in computing nowadays and is a very hot area even for those formally unfamiliar with computational science. Deep learning has become incredibly popular over the last decade, sparked by the then-incredible performance of AlexNet to classify the 1.2 million image dataset ImageNet [8], showing that deep convolutional networks can achieve predictive performance far superior to existing statistical techniques. Recently, research has been conducted into extending existing convolutional techniques to graph-based domains: the *Graph Convolutional Network* (GCN) and *Message Passing Neural Network* (MPNN) layers were some of the first to introduce scalable nodal convolutions on graphs [9, 10]. Nowadays, almost all operations defined on traditional convolutional networks have some analog for their graph counterparts [11, 12].

The union of these two fields: deep convolutional techniques towards learning multigrid solvers, is quite under-explored in traditional multigrid literature. Oosterlee et al. [13] in 2003 showed that genetic algorithms could be used to search for optimal parameters on fixed test problems. Recent papers have demonstrated that traditional [14] and graph convolutional networks [15] can be used to learn the interpolation operator given some coarse/fine partitioning of the space for algebraic multigrid. There have been efforts to learn this partitioning as well; Uz-Zaman et al. [16] have successfully used simulated annealing to search for this optimal partitioning over some set of test problems, Taghibakhshi et al. [17] have improved on this by using reinforcement-learning to learn a greedy algorithm for selection of this coarse/fine space. However, to our knowledge developing some machine learning method for selection of both the coarse/fine space and the resulting interpolation operator is still a problem that requires attention.

In this thesis, we will explore the usage of graph convolutional networks to output exactly this coarse/fine partitioning and resulting interpolation for *smoothed-aggregation algebraic multigrid*. Smoothed-aggregation multigrid is a branch of algebraic multigrid that forms the coarse space by collapsing nodal points into *aggregates*, then defining a transfer operators between these two spaces. Traditionally, this has been done through the use of graph heuristics such as clustering [18] or identification of strongly connected nodes [19]. We will introduce a so called machine-learned *agent* to group nodes together similarly to existing heuristics, then smooth the connections from aggregates to nodes to introduce additional flexibility in the solver procedure. This agent will be compared against a baseline random benchmark and a Lloyd aggregation method. Difficulties in training the agent with traditional gradient methods will be addressed through the use of genetic optimization techniques.

We introduce this work as a preliminary study into providing a learned method for outputting both aggregation and interpolation. The take-away from reading this is not to think "wow, ML can be used to slightly improve existing methods", but rather to think of what these sorts techniques can be used for in the future. Can we use ML to set up multigrid on problems that we do not know how to solve currently? What would a solver configuration even look like for these unknown problems? Perhaps the best way to look at this is to view it as a framework from which future ML work can be based on.

1.1 OVERVIEW

The remainder of this thesis is structured as follows: Chapter 2 will develop the concepts behind smoothed aggregation algebraic multigrid (SA-AMG) to a sufficient level such that the reader will be able to generally understand the various methods introduced; a brief in-

troduction to graph neural networks and the various GNN operations will also be included. Chapter 3 will define the sets of problems that will be testing the agent, the actual structure of the agent itself, and finally the training method itself. Then, Chapter 4 will display both tabular and graphical results comparing the ML agent against a random benchmark and Lloyd aggregation. Numerous figures detailing the raw output of the various SA-AMG methods are included. Finally, we include the obligatory conclusion in Chapter 5 to summarize results at a high level.

CHAPTER 2: BACKGROUND

In this chapter, we introduce the requisite knowledge needed to understand the bulk of this thesis. A few sections or the chapter in its entirety may be skipped if the reader is already familiar with such topics.

2.1 SMOOTHED-AGGREGATION MULTIGRID

In the realm of computational science, one of the most important operations to perform quickly is to compute solutions of linear systems of the form

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

for $\boldsymbol{A} \in \mathbb{R}^{n\times n}$; $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$. We will be focusing on the case where $\boldsymbol{A}$ is sparse and symmetric, positive definite (SPD); such systems often arise (for example) from the numerical discretization of partial differential equations. Direct solvers for such problems are readily implementable and well understood, though their performance deterioriates for large matrices and especially so for problems in higher spatial dimensions, as their operation is intrinsically linked to the number of nonzero entries in the matrix.

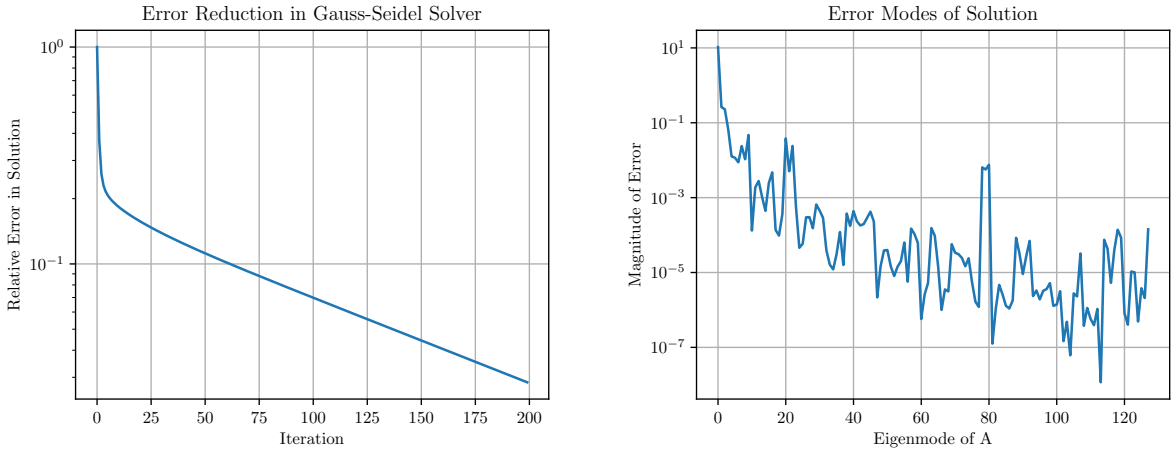
It is for this reason that *iterative methods* are often attractive for solving such systems, for they produce a series of closer approximations to $\boldsymbol{x}$ instead of directly using all entries of the matrix itself. One such family of methods that are often employed are *multigrid methods*; in their most basic form these methods construct a hierarchy of *coarser* problems (smaller-dimensional $\boldsymbol{A}$) and transfer operators between levels on this hierarchy. As one may guess, there are multiple types of multigrid methods that exist. We will be considering *smoothed-aggregation multigrid*, *algebraic* in the sense that no additional information beside the algebraic system is required and *smoothed-aggregation* by the way coarse levels are constructed: aggregating clusters of nodes together and smoothing the resultant basis functions of each aggregate.

Before SA-AMG itself is introduced, we will first motivate the relative strengths of multigrid methods as compared to stationary relaxation methods, such as Gauss-Seidel relaxation. For matrix $\boldsymbol{A}$, we can spatially decompose into its strictly lower triangular, diagonal, and upper triangular matrices such that $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$, and $\text{nonzeros}(\boldsymbol{L}) \cap \text{nonzeros}(\boldsymbol{D}) \cap \text{nonzeros}(\boldsymbol{U}) = \emptyset$, i.e. that the sparsity patterns do not overlap. We can derive the Gauss-Seidel iteration by using *only the lower diagonal portion* of $\boldsymbol{A}$ to solve the system. Letting

$\boldsymbol{x}^{(k)}$ and $\boldsymbol{x}^{(k+1)}$ be the approximations to $\boldsymbol{x}$ at the k'th and $k+1$'th iterations, we have

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{D}+\boldsymbol{L})^{-1}\left(\boldsymbol{U}\boldsymbol{x}^{(k)}+\boldsymbol{b}\right). \tag{2.2}$$

Since we have reduced the task of solving $\boldsymbol{A}$ to inverting a lower-triangular matrix, $\boldsymbol{D}+\boldsymbol{L}$, we can easily solve for $\boldsymbol{x}^{(k)}$ by a sparse forward-substitution in $\mathcal{O}(n^2)$ time. At this point, you may be wondering what the downsides to using Gauss-Seidel are. Well, as the size of the problem (and complexity) increase, the performance of Gauss-Seidel – and indeed any relaxation scheme – degrades quite heavily. As one can see in Fig 2.1a, such methods initially make rapid progress in solving $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ but then slow down significantly. After 200 iterations in the example problem given, the relative error in $\boldsymbol{x}^{(k)}$ is not reduced by even a factor of 10^{-2}. We can see an interesting phenomena arise in the eigenmodes of the error in Fig 2.1b: high frequency (large index) eigenmodes are effectively dampened while low frequency modes remain almost unscathed. Thus, we can think of relaxation methods as *relaxing the high-frequency modes* in the error, while mostly keeping *low-frequency* or *smooth* modes.



(a) Error reduction per Jacobi Iteration.

(b) Lowest 128 eigenmodes of error after iterations.

Figure 2.1: Error analysis for Gauss-Seidel relaxation method for a 3D $32\times32\times32$ Poisson problem.

Perhaps the important observation that lead to the development of multigrid as a whole, was that the approximate solution can be transferred from the original, high resolution problem into a coarser, lower resolution problem and suddenly these smooth error modes appear as higher frequency modes on the coarse level. We will refer to the original and

lower-resolution spaces as the *fine grid* and the *coarse grid*[1] The multigrid algorithm can thus be sketched by the following process:

1. Run a predefined number of pre-relaxation iterations.
2. *Transfer* solution to coarse grid.
3. Either solve directly on the coarse system or recursive further.
4. Transfer coarse solution to fine grid.
5. Run another number of post-relaxation iterations.

A two level (no recursion) form of the algorithm is formalized in Alg 2.1.

Algorithm 2.1 Two-level multigrid V-cycle

procedure MULTIGRID($\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b} \in \mathbb{R}^n$, $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{P} \in \mathbb{R}^{n \times k}$)
  $\boldsymbol{x} \leftarrow$ GAUSS-SEIDEL($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{x}$) ▷ Pre-relaxation
  $\boldsymbol{r} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$ ▷ Get residual error
  $\boldsymbol{r}_H := \boldsymbol{P}^T\boldsymbol{r}$ ▷ Restrict residual to coarse grid
  $\boldsymbol{A}_H := \boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$ ▷ Form coarse system via *Galerkin product*
  $\boldsymbol{e}_H := \boldsymbol{A}_H^{-1}\boldsymbol{r}_H$ ▷ Solve for coarse-grid correction
  $\boldsymbol{x} \leftarrow \boldsymbol{P}\boldsymbol{e}_H$ ▷ Interpolate correction to fine grid
  $\boldsymbol{x} \leftarrow$ GAUSS-SEIDEL($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{x}$) ▷ Post-relaxation
end procedure

Until now, we have carefully ignored defining how solutions are transferred between coarse and fine grids. Let $\boldsymbol{P} \in \mathbb{R}^{n \times k}$ be the linear operator that maps vectors from the k-node coarse grid to the n-node fine grid, also called the *interpolation operator*. Selection of $\boldsymbol{P}$ can determine how well — or inversely, how poorly — the resultant multigrid solver converges. There are many algorithms for generating the interpolation, we will proceed by (finally) introducing *smoothed-aggregation algebraic multigrid*.

For SA-AMG, we begin by imagining the sparse system $\boldsymbol{A}$ as a directed graph. If $\boldsymbol{A}$ has n rows and columns, define the graph $G(e, v)$ that has n nodes. The nonzero entries in $\boldsymbol{A}$ define edges such that

$$\text{nodes } j \to i \text{ are connected iff } a_{ij} \neq 0. \tag{2.3}$$

Letting our *coarsening ratio* be defined by $\alpha \in (0, 1]$, let $k := \lceil \alpha n \rceil$. Intuitively, if we have ratio α we will roughly be coarsening the fine grid by a factor of α^{-1}. We assign each node

[1]The problems are no longer constrained to be defined on grids, though the name remains for historical reasons.

(explained *how* later) in the graph to one of k aggregates or *sets of nodes*. Usually these will be strongly connected, meaning for aggregate $\text{agg}^{(i)}$,

$$\boldsymbol{A}_{ij} \neq 0 \quad \forall i, \forall j \in \text{agg}^{(i)}. \tag{2.4}$$

We can write this mapping of nodes to aggregates with the binary assignment matrix

$$\text{Agg} \in \mathbb{R}^{n \times k}: \quad \text{Agg}_{ij} = \begin{cases} 1 & i \in \text{agg}^{(j)} \\ 0 & \text{otherwise} \end{cases}. \tag{2.5}$$

An example aggregation of a 1D graph into three aggregates can be seen in Figure 2.2. The corresponding assignment matrix can be simply written as

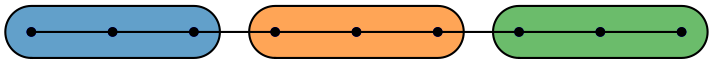


Figure 2.2: Example aggregation of a 1D problem.

$$\text{Agg}^* := \begin{bmatrix} 1 & 1 & 1 & & & & & & \\ & & & 1 & 1 & 1 & & & \\ & & & & & & 1 & 1 & 1 \end{bmatrix}^T. \tag{2.6}$$

Something to note is that we can *technically* stop here and directly use Agg as $\boldsymbol{P}$. We could interpolate $\boldsymbol{A}_H := \text{Agg}^T \boldsymbol{A} \text{Agg}$, and connections would be inserted between aggregates as

$$\left(\text{Agg}^T \boldsymbol{A} \text{Agg}\right)_{ij} = \sum_{k=1}^{n} \sum_{l=1}^{n} \text{Agg}_{li} A_{lk} \text{Agg}_{kj}, \tag{2.7}$$

meaning if nodes (i, j) are connected and are in disjoint aggregates k, l, those two aggregates would receive some contribution on the coarse grid. We can, however, extend the range of contributions by neighboring nodes by *smoothing the assignment matrix*. For all further smoothed aggregation methods, we will use a *Jacobi smoother* to smooth the basis functions.

Its operator is given by

$$\boldsymbol{S} := \boldsymbol{I} - \omega \boldsymbol{D}^{-1} \boldsymbol{A}, \tag{2.8}$$

$$\omega := \frac{4}{3\lambda_{\max}(\boldsymbol{A})}, \tag{2.9}$$

where $\boldsymbol{D}$ is the diagonal portion of $\boldsymbol{A}$. The final interpolation is therefore formed by

$$\boldsymbol{P} = \boldsymbol{S}\text{Agg}. \tag{2.10}$$

Continuing our example from above, the smoothed Agg^* is given by

$$\text{P}^* \approx \begin{bmatrix} .66 & 1 & .66 & .34 & & & & \\ & & .34 & .66 & 1 & .66 & .34 & \\ & & & & .34 & .66 & 1 & .66 \end{bmatrix}^T. \tag{2.11}$$

Additional connections are added between aggregates and the matrix itself is no longer binary as well, meaning nodes have differing interpolation weights for each aggregate.

2.2 RANDOM AND LLOYD AGGREGATION

For comparison with the ML method to be proposed, we introduce *Random aggregation* and *Lloyd aggregation* for computing the Agg assignment matrix in SA. We will begin with the random method, as Lloyd can be thought of as an enhancement to the aforementioned method. Quite simply, we define the aggregate centers $\boldsymbol{s} := \begin{bmatrix} s_1 & s_2 & \cdots & s_k \end{bmatrix}^T \in \mathbb{N}^k,\ 0 \leq s_i \leq n,\ s_i \neq s_j\ \forall i, \forall j$. This can be interpreted as assigning node s_i to be the root of aggregate i.

For the random method, the aggregate roots are, unsurprisingly, randomly drawn from the possible range of node indices. Each non-root node is then assigned to an aggregate by traversing the graph and finding the nearest (by some metric) root. This is done by a modified version of the *Bellman-Ford* algorithm, defined in Alg A.1.

Lloyd aggregation builds on the random method by attempting to evenly space out the distance between aggregates. We begin with the random seeding as before, then for each iteration we run the modified Bellman-Ford algorithm to assign the nearest root and distance to root for each node. Then, each node that lies on aggregate borders is found; this can be done in edge-linear time by checking if the indices of nonzero matrix entries belong to different roots (tentative aggregates). Bellman-Ford is then run again to get the distance

from each node to an aggregate border — the node in each aggregate with largest border distance is then selected to be the new aggregate root. This procedure is repeated for an arbitrary number of iterations or until some stopping criteria is met, i.e. roots no longer change between iterations.

2.3 GRAPH NEURAL NETWORKS

As the linear systems we will be considering are largely sparse and sparse matrices can be interpreted as graphs, one can reasonably intuit that we will be using graph-based neural network techniques in order to learn information from our diffusion problems. We will consider two operations on graphs that may be useful to us: node convolutions, which hold edge values constant and update values on nodes; edge convolutions, which hold node values constant and update values on edges. Keeping this with our graph $\Leftrightarrow$ sparse matrix analogy, if we have a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, we can consider node convolution to generate some vector $\boldsymbol{y}' \in \mathbb{R}^n$ based on $\boldsymbol{A}$ and a vector $\boldsymbol{y} \in \mathbb{R}^n$, while edge convolutions generate a matrix $\boldsymbol{A}'$ based on $\boldsymbol{y}$. We will explore later the use and importance of these two operations.

For node convolutions, we will mainly consider two layers: TAGConv[20] and MPNN[21]. The TAGConv layer learns a K-degree polynomial of the input matrix $\boldsymbol{A}$ that is multiplied by any input,

$$\boldsymbol{y}_f = \sum_c \left(\sum_k \omega_{cfk} \boldsymbol{A}^k \right) \boldsymbol{x}_f + b_f, \tag{2.12}$$

for output feature f, input feature c, and weight value ω_{ijk}. We will also use the MPNN layer, a convolutional layer based on *message passing* and has the nice benefit of allowing multiple input edge features, allowing us to convolve graphs which may have more than one value per edge. This layer has the update formula of

$$y_i = \frac{1}{|\mathcal{N}(i)|} \sum_{k \in \mathcal{N}(i)} \Theta\left(e_{j,k}\right) x_k + b_i, \tag{2.13}$$

where $\mathcal{N}(i)$ is a map returning the neighborhood of vertex i (including itself) and $\Theta : E \mapsto \mathbb{R}^{f_i \times f_{i-1}}$ is some MLP that outputs a learned weight matrix given an edge.

Along with node convolutions, there are also *edge convolution* layers. The existing literature on edge convolution is quite sparse (pun intended), so we shall introduce our own layer to achieve this. The update for the *simple edge layer* is given by

$$\boldsymbol{A}'_{ij} = \theta\left([y_i \mid y_j \mid \boldsymbol{A}_{ij}]\right), \tag{2.14}$$

where θ is some MLP, $\boldsymbol{y}$ are the input node values, and $|$ is the concatenation operator (combining multiple vectors columnwise into a matrix, for example).

Additionally, let us define the top-k operator as

$$\text{top-k}\left(\boldsymbol{x}; k\right)_i = \begin{cases} 1 & x_i \text{ is among the } k \text{ largest values of } \boldsymbol{x} \\ 0 & \text{otherwise} \end{cases}, \tag{2.15}$$

i.e., the k largest values of $\boldsymbol{x}$ are assigned the value 1 while 0 is assigned to all other values.

CHAPTER 3: METHODS

This chapter will introduce the standard data sets that the ML agent will be trained and evaluated over (3.1) as well as several more complex problems that the agent will be evaluated on (3.2). We will then describe the unsupervised loss (3.3) and the various components of the agent (3.4, 3.5) before combining the two (3.6). A slightly unorthodox method of training is then described (3.7).

3.1 ISOTROPIC AND ANISOTROPIC DIFFUSION

The main problem we will be considering is the diffusion PDE in two dimensions with variable amounts of anisotropy. Formally, we are considering equations of the type

$$-\nabla \cdot (\boldsymbol{D} \nabla u) = 0, \tag{3.1}$$

with diffusion tensor

$$\boldsymbol{D} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & \\ & \varepsilon \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^T, \tag{3.2}$$

for some rotation θ and scaling ε along the y-axis. Homogeneous Dirichlet boundary conditions are assumed.

We will separately consider two datasets of isotropic ($\theta = 0$, $\varepsilon = 1$) and anisotropic diffusion problems, each consisting of:

1. A training set of 1000 unstructured problems
2. A testing set of 300 unstructured problems

To create a problem, random points are first generated in $[0, 1]^2$, a convex hull constructed, then meshed with *gmsh* [22]. The problem is then discretized using a standard finite-element discretization with degrees-of-freedom corresponding to Dirichlet boundaries eliminated. The resulting problems have anywhere between 15 and 400 degrees of freedom in the linear system. For anisotropic problems, θ varies uniformly between $(0, 2\pi)$ while ε is log-uniformly distributed between 10^{-5} and 10^5.

Now that the set of training and testing data has been established, we will introduce the ML agent that will attempt to *learn* the interpolation in a way that is analogous to

smoothed-aggregation. We will therefore initially consider the aggregation (section 3.4) and smoothing steps (section 3.5) separately, combining the two later.

3.2 SPECIAL PROBLEMS

In addition to the above regular isotropic and anisotropic problems, we will consider a few *special problems* that may introduce more difficulty to solve. When evaluating these problems with the ML agent, we will first use the network weights trained on the *anisotropic and isotropic* problems to determine if the agent is able to generalize. Then, we will actually train the agent on the specific set of problems to establish the agent's *potential* in solving the problem if it were to be specialized. These special problems are all specific cases of the diffusion problem and are composed of:

- **Large isotropic problems**
  These problems are virtually identical to the isotropic problems introduced in section 3.1, though each mesh has a higher resolution. These have overall some number of Degrees of Freedom (DoF) between 2000 and 3000. The motivation behind trying larger problems, behind the obvious study to see if the agent can scale up, is to eliminate any performance benefit that can be exploited on the mesh boundaries due to the Dirichlet condition. A set of 100 total *large* problems was created.

- **Jumping Coefficients**
  These are diffusion problems where the coefficient has discontinuous jumps between sub-regions in the domain. These can be difficult for an AMG method to solve as region boundaries need to be found and aggregates formed along the regions [23][24].

  To generate these problems, we first output 2 or 3 random *seed points* in the domain $\Omega = [0,1]^2$; these are labeled $\boldsymbol{S} = \{\boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{s}_3\}$. We log-uniformly generate diffusion coefficients d_1, d_2, d_3, such that $\log d_i \sim U(\log 10^{-4}, \log 10^4)$ with the additional constraint that the range of d_i must be at least 10^3, enforcing large enough gaps in the diffusivity of the regions. A Voronoi graph is then constructed over the domain; each point takes the diffusion coefficient of its nearest (in L^2 norm) seed point in $\boldsymbol{S}$. A set of 250 problems with similar DoF to the standard isotropic/anisotropic problems was created.

- **Smoothly Varying Coefficients**
  Problems with smoothly-varying coefficients are generated by defining the diffusion coefficients as low-frequency cosine waves over the domain. We randomly generate the

parameters $\theta \sim U(0, 2\pi)$, $\varepsilon_x, \varepsilon_y \sim U(0.1, 10)$, $b_x, b_y \sim U(-10, 10)$. We then define the affine transform,

$$\boldsymbol{Dx} := \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^T \left(\begin{bmatrix} \varepsilon_x & \\ & \varepsilon_y \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} b_x \\ b_y \end{bmatrix} \right), \tag{3.3}$$

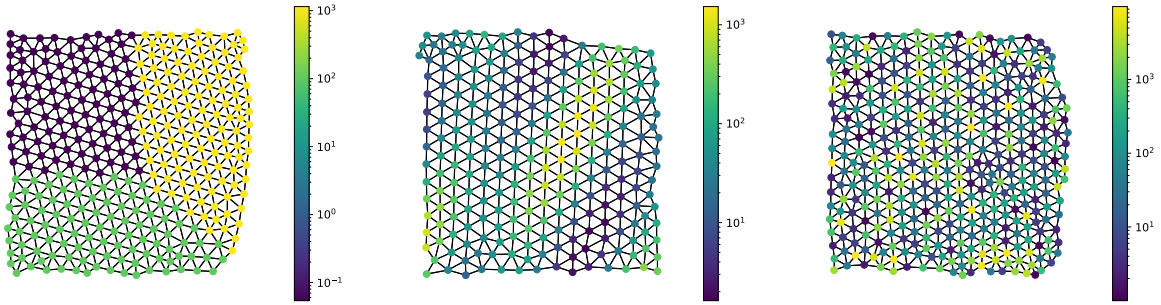
allowing random rotation, stretching, and translation of points on the domain. Let the diffusion coefficient be the function

$$d(\boldsymbol{x}) := 10^{\frac{3}{2}\left(\cos\left((\boldsymbol{Dx})_x\right)^2 + \cos\left((\boldsymbol{Dx})_y\right)^2\right) + \frac{1}{5}}. \tag{3.4}$$

Pay no attention to the magic $\frac{3}{2}$ and $\frac{1}{5}$ coefficients as they were chosen arbitrarily to obtain an affine transformation that output a range of coefficients (roughly) in the range $(1, 10^4)$. A set of 250 problems with similar DoF to the standard isotropic/anisotropic problems was created.

- **Noisy/Random Coefficients**
  As with jumping coefficients, random coefficients can be difficult problems for existing AMG methods to solve [23]. For each interior node in the problem, a random diffusion coefficient is log-uniformly generated like $\log d_i \sim U(0, \log 10^4)$. This coefficient is then bilinearly interpolated to inter-nodal values on the mesh; for example when the diffusion equation is discretized in space the quadrature rules require values of the diffusion coefficient within the element. A set of 250 problems with similar DoF to the standard isotropic/anisotropic problems was created.

A few examples of these special problems can be seen in Figure 3.1. 0



(a) Jumping coefficient problem (b) Smoothly varying coefficient problem (c) Noisy coefficient problem

Figure 3.1: Example problems for jumping, smooth, and noisy coefficient diffusion problems.

3.3 MULTIGRID LOSS

Before we get into the actual method itself, let us first define the structure of the solver that is used in this work. To solve problems of the form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.5}$$

we set up an iterative two-level multigrid V-cycle solver as outlined in 2.1. When training networks (3.6, we use a loss consisting of running the multigrid cycles to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ for some normally distributed unit initial guess $\boldsymbol{x}^{(0)} \in \mathcal{N}(0,1)\,; \|\boldsymbol{x}\| = 1$. Since we are solving for $\boldsymbol{b} = \boldsymbol{0}$, let the sequence of errors be defined as

$$\boldsymbol{e} = \begin{bmatrix} \left\|\boldsymbol{x}^{(0)}\right\| & \left\|\boldsymbol{x}^{(1)}\right\| & \cdots & \left\|\boldsymbol{x}^{(n)}\right\| \end{bmatrix}. \tag{3.6}$$

We thus approximate the factor of convergence by a simple geometric mean,

$$\mu(\boldsymbol{A}) = \left(e^{(n)} - e^{(n-k)}\right)^{1/(k-1)}, \tag{3.7}$$

$$k = \min\left\{\lceil n/3 \rceil, 10\right\}, \tag{3.8}$$

for some problem $\boldsymbol{A}$. If we have a set of problems $\mathcal{A} := \{\boldsymbol{A}_1, \boldsymbol{A}_2, \ldots, \boldsymbol{A}_n\}$, we let the loss be defined by

$$\ell(\mathcal{A}) = \frac{1}{|\mathcal{A}|} \sum_{i=1}^{|\mathcal{A}|} \mu(\boldsymbol{A}_i), \tag{3.9}$$

i.e., the average convergence over the entire data set. We will later directly optimize the value of ℓ when training the ML agent.

A *relative loss* of the form

$$\ell_{\text{rel}}(\mathcal{A}) = \frac{1}{|\mathcal{A}|} \sum_{i=1}^{|\mathcal{A}|} \frac{\mu(\boldsymbol{A}_i)}{\mu_{\text{Lloyd}}(\boldsymbol{A}_i)} \tag{3.10}$$

was also investigated, where μ_{Lloyd} is the convergence using Lloyd aggregation – this in essence formulates the loss as the relative performance from the ML and Lloyd methods with the intention of removing any bias that may be present from the problem size itself; overall convergence factor is correlated to the size and number of degrees-of-freedom in a problem. However, no noticeable difference in overall quality of the resulting ML agent was observed between the two loss functions, so ℓ was chosen to be the overall loss.

3.4 ML AGGREGATION

The purpose of the aggregation step is the selection of nodes that will form the coarse-grid. The method we will introduce selecting such coarse grids is extended from neural graph-pooling methods [25][11] whose operation consists of assigning nodes a numeric score value, then discarding nodes that fail to meet some predetermined criteria.

We will formally define two networks, Θ_{agg}, and Θ_{soc}, whose purpose are to output nodal score values and strength-of-connection matrices, respectively. The aggregation network, Θ_{agg} takes as input a sparse matrix (that is converted to a graph), and outputs a binary $\{0,1\}^n$ vector assigning certain nodes to be *aggregate roots*. This network begins with a constant $1/n$ value for each node and then convolves nodal values with TAGConv layers interspersed with binary top-k layers to convert real values into aggregate centers. A graphical overview can be seen in Figure 3.2; this architecture was chosen to mimic the several iterations used by Lloyd aggregation that are used to select a set of aggregate centers. These layers are repeated several times until a set of aggregate centers is chosen.

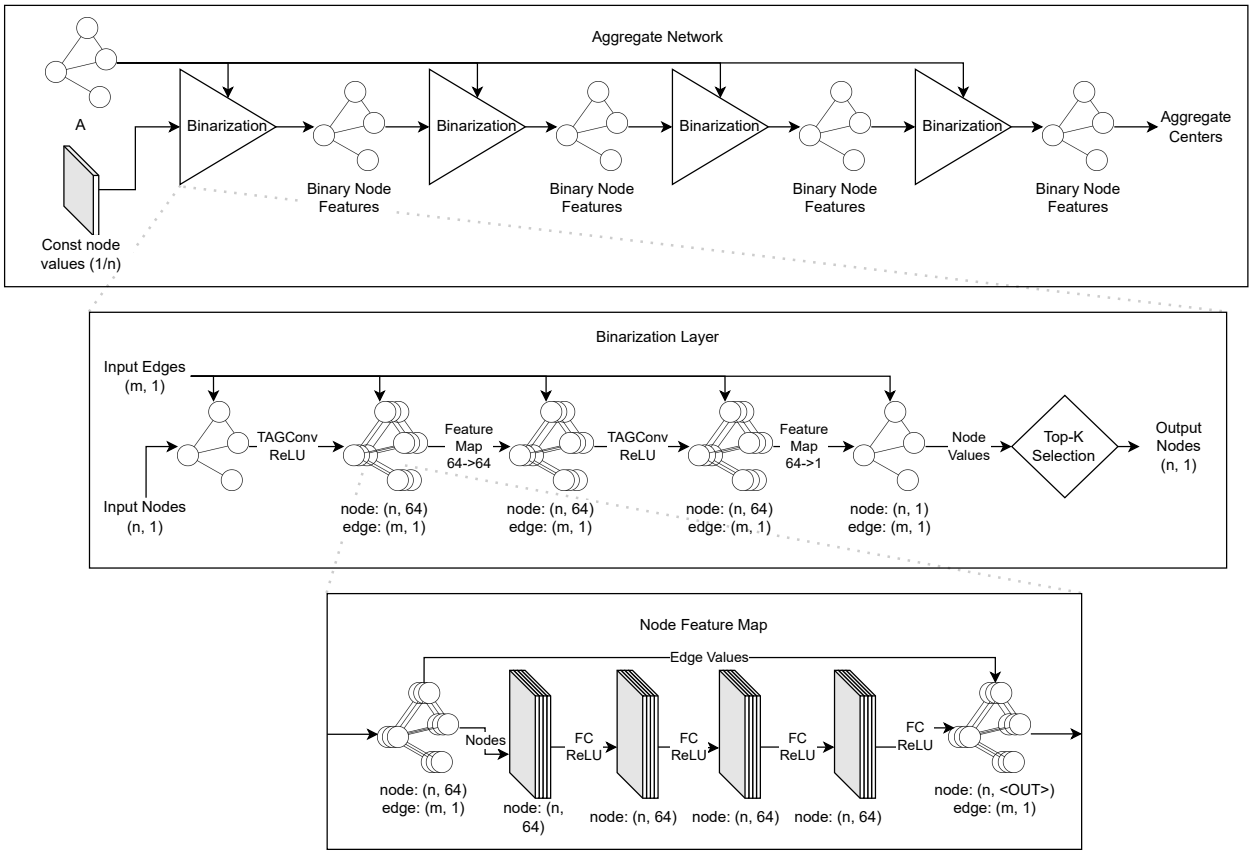


Figure 3.2: Architecture for the aggregate-picking network. This consists of 4 binarization layers, each having two node convolution passes.

After aggregate roots are chosen, the strength-of-connection network is used to output a set of edge weights. This is done by means of numerous edge convolutions on the network $\boldsymbol{A}$, with a set of constant node values, a graphical overview is given in Figure 3.3. Denoting the output values from Θ_{soc} as $\boldsymbol{C}$, a binary matrix $\text{Agg} \in \mathbb{R}^{n \times c}$ assigning nodes to aggregates can be formed using a modified Bellman-Ford algorithm (A.1) then assigning nodes to unique aggregates (A.2).

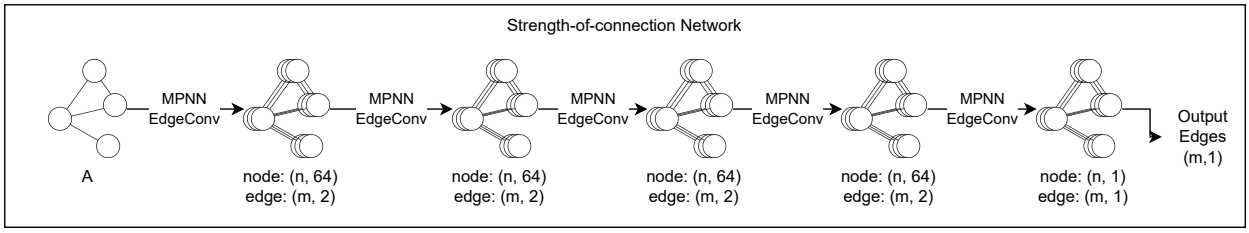


Figure 3.3: Architecture for the strength of connection network. This consists of several MPNN layers interspersed with edge convolutional layers.

To summarize, Θ_{agg} is used to select k aggregate roots, while Θ_{soc} is input into a modified Bellman-Ford algorithm to assign each node to an aggregate. These two are used to output a tentative aggregate mapping, $\text{Agg} \in \mathbb{R}^{n \times c}$. We will make the remark that we have also tried the network architecture of combining the functions of both Θ_{agg} and Θ_{soc} into one network that outputs both modified nodal values and edge weights. Performance was, however, ultimately disappointing as the resultant network was unable to effectively learn neither aggregation nor smoothing.

3.5 ML INTERPOLATION

We now consider the task of finding some interpolation operator $\boldsymbol{P}$ given a tentative interpolation operator, Agg. Our main motivation for this is to introduce connections between aggregates on the coarse grid. As we cannot change the sparsity pattern with our edge convolution layers, we thus learn a network Θ_S that can be thought of as *smoothing* the values of Agg.

We first construct an auxiliary edge feature between nodes in the graph given by

$$\boldsymbol{B}_{ij} = \begin{cases} 0 & i \text{ and } j \text{ are in the same aggregate} \\ 1 & \text{otherwise} \end{cases}, \tag{3.11}$$

i.e., an indicator whose value is one between aggregate boundaries. Note that this step can be efficiently done in edge-linear time by comparing aggregate roots of each node using the

nearest-neighbor information computed from Bellman-Ford in 3.4. This feeds the network with information on differing aggregates without needing to specifically enumerate each aggregate.

As with finding the strength-of-connection matrix, we define a new network Θ_S that similarly uses edge convolutions interspersed with MPNN convolutions (as we have multi-dimensional edge features) to output a tentative smoother, $\hat{\boldsymbol{S}}$, with the same sparsity pattern as $\boldsymbol{A}$. Again, a graphical overview is given in Figure 3.4. Because a ReLU activation is used for the network, the entries of $\hat{\boldsymbol{S}}$ are non-negative.

We can finally combine $\hat{\boldsymbol{S}}$ and Agg to form the interpolation operator $\boldsymbol{P} := \hat{\boldsymbol{S}}\text{Agg}$. As with the typical smoothed-aggregation smoother, this will force connections between aggregates in the coarse grid.

Theorem 3.1. Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be a sparse matrix whose graph does not have disconnected regions and $\text{Agg} \in \mathbb{R}^{n\times c}$ be the assignment of nodes to (also fully connected) aggregates. Forming $\boldsymbol{P} := \hat{\boldsymbol{S}}\text{Agg}$ results in aggregates that have connections to each node they are composed of, as well as the neighbors of each node on aggregate boundaries.

Proof. Let $\boldsymbol{P} := \hat{\boldsymbol{S}}\text{Agg}$; $p_{ij} \neq 0$ iff node i is connected to aggregate j. Looking at the columns of $\boldsymbol{p}_j$, we have

$$\boldsymbol{p}_j = \sum_{k=1}^{n} \hat{\boldsymbol{s}}_k \text{Agg}_{kj} = \sum_{k\in \text{agg}_j} \hat{\boldsymbol{s}}_k. \tag{3.12}$$

Because of the ReLU activation used, $a_{ij} \neq 0$ implies $\hat{s}_{ij} \geq 0$, thus if nodes i, j are connected in $\boldsymbol{A}$ then they remain connected in $\hat{\boldsymbol{S}}$ and so the nonzero entries in $\hat{\boldsymbol{s}}_k$ represent the connectivity of node k. Since $\hat{s}_{ij}$ are strictly nonnegative, the sparsity of column $\boldsymbol{p}_j$ is the *union* of the sparsity of the $\hat{\boldsymbol{s}}_k$, meaning $\boldsymbol{p}_j$ is connected to each node in agg_j as well as the neighboring nodes on the boundary. QED.

3.6 FULL ML AGENT

Having defined Θ_{agg}, Θ_{soc}, Θ_S, we now sketch the full method from start to finish of generating an interpolation operator given a sparse system $\boldsymbol{A}$ and $\alpha \in (0, 1]$ a parameter that determines the ratio of aggregates to vertices (i.e., roughly coarsening the system by $1/\alpha$.) The steps to be taken are formalized in Algorithm 3.1.

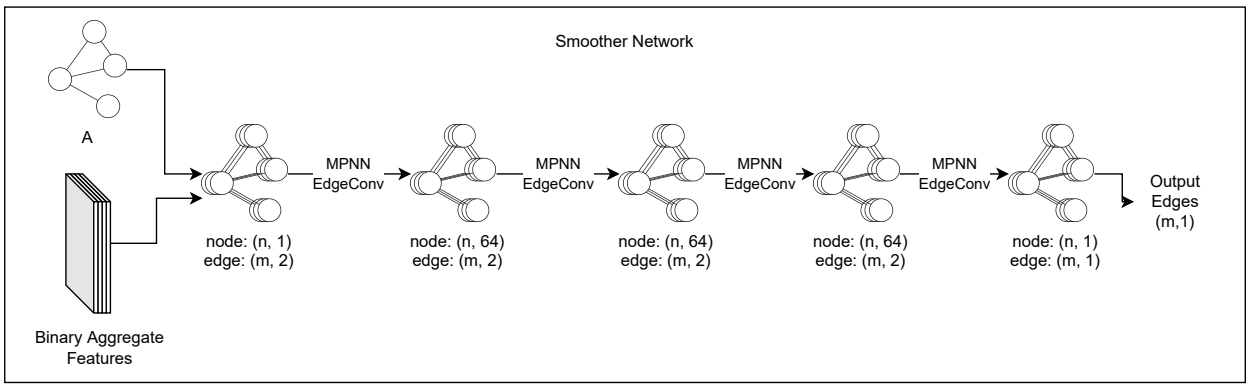


Figure 3.4: Architecture for the smoother network. This consists of several MPNN layers interspersed with edge convolutional layers.

Algorithm 3.1 Outputting Aggregates and Interpolation

procedure FULLAGGNET($\boldsymbol{A} \in \mathbb{R}^{n \times n}, \alpha$)
  $k := \lceil \alpha n \rceil$ ▷ Number of aggregates to choose
  $\boldsymbol{C} := \Theta_{\text{soc}}(\boldsymbol{A})$ ▷ Obtain strength-of-connection matrix

  $\boldsymbol{s} := \Theta_{\text{agg}}(\boldsymbol{A}, \boldsymbol{C}, k)$ ▷ Get aggregate scores for each node
  $\boldsymbol{r} := \text{top-k}(\boldsymbol{s}; k)$ ▷ Get indices of k largest nodes to be centers
  Agg := BELLMAN-FORD($\boldsymbol{C}, \boldsymbol{r}$) ▷ Get tentative aggregate assignment matrix

  $\hat{\boldsymbol{S}} := \Theta_S(\boldsymbol{A}, \text{Agg})$ ▷ Aggregate smoother
  return $\hat{\boldsymbol{S}}$Agg ▷ Smooth aggregate assignment
end procedure

3.7 GENETIC TRAINING

Because of the use of the top-k operators in the network, the agent used is inherently non-differentiable and thus gradient-free optimizers are required to update network weight and bias values. The optimizer we have chosen to use is a custom genetic algorithm [26]. All three networks in the agent are combined into one "module" that is trained concurrently with the genetic algorithm. The genetic training is initially seeded with a population of 20 randomly-generated agents. There are then four steps during the training that are repeated in order to find an optimal set of network weights, mimicking biological evolution:

1. **Selection**
   Individual networks in the population are selected for *breeding*. For this step, we use *steady-state selection* for approximately 200 training iterations, then switch to a so-called *greedy* selection. For steady-state selection, the top $\frac{2}{3}$ fittest individuals are selected for breeding and the bottom $\frac{1}{4}$ are discarded (the remaining $\frac{1}{12}$ is simply

ignored). When switched over to the *greedy selection*, the single most fit individual is retained and all other networks are discarded; this is to approximate traditional gradient-based optimizers that take an optimal step at each iteration.

2. **Crossover**
   This is the process of combining two individuals for breeding and generating two offspring networks. We use a so-called *folded crossover* [27] in that weights corresponding to entire layers are grouped together into "folds". When breeding networks, one child will randomly receive half of its layer weights from the first parent and the other half of its layer weights from the other parent. The second child is then generated as the complement of the first child, receiving the opposing set of layer weights from the parents. Thus, the two parents weights are dis-jointly distributed to the new children.

3. **Mutation**
   Random mutations are now inserted into each network in order to maintain population diversity and allow exploration towards optima. Each weight value has a $p = 0.5$ probability of being perturbed by $\delta \in [-1, 1]$. When the greedy selection is employed, this is modified to $p = 1$ and $\delta \in [-0.1, 0.1]$.

4. **Fitness Evaluation**
   Finally, fitness of the entire population is evaluated. In ML terms, the loss of each network is evaluated by taking the averaged 2-level multigrid convergence over the training set. As this is the most computationally intensive step, this step is both multi-threaded for parallel evaluation and stochastically batched, selecting only 32 problem samples from the training set per evaluation iteration. Batching is often avoided in genetic algorithms as it ruins any sense of caching loss values for unchanged networks, though here the speedup in reducing computational cost far outweighs potential caching benefits.

We thus train the agent on an *unsupervised loss*, allowing it to obtain a method for finding aggregates and a smoothing operator that overall reduces multigrid convergence; the actual multigrid cycle used is defined in section 3.3. The method is completely unsupervised in that existing aggregation or interpolation methods are not used to seed or somehow influence the agent beforehand.

3.7.1 Simultaneous Perturbation Stochastic Approximation

The use alternative gradient approximations was also considered in this work. Classical finite difference approximations were immediately ruled out as too costly: the agent has

somewhere around the order of several thousand network parameters and the loss itself is already expensive to compute once. However, simultaneous perturbation stochastic approximation [28] (SPSA) is a method to approximate the gradient to some function with only two evaluations.

For some step-size h, let the gradient by approximated by

$$\nabla_\theta \ell(\theta) \approx \frac{\ell(\theta + h\boldsymbol{\Delta}) - \ell(\theta - h\boldsymbol{\Delta})}{2h\boldsymbol{\Delta}}, \tag{3.13}$$

letting $\ell : \mathbb{R}^n \to \mathbb{R}$ be the function that is optimized, $\theta \in \mathbb{R}^n$ the parameters, and $\boldsymbol{\Delta} \in \mathbb{R}^n$ some simultaneous perturbation to the parameter space, where $(2\Delta_i - 1) \sim \text{Bernoulli}\left(\frac{1}{2}\right)$; the values of Δ_i can randomly take either 1 or -1 with mean 0. Assume entry-wise division by the vector $\boldsymbol{\Delta}$. This distribution for the perturbation is necessary for the inverse moment of Δ_i to be bounded and the approximation to hold.

This can then be used in a standard descent algorithm, i.e.

$$\theta^{(i+1)} = \theta^{(i)} - \frac{\ell\left(\theta^{(i)} + h\boldsymbol{\Delta}\right) - \ell\left(\theta^{(i)} - h\boldsymbol{\Delta}\right)}{2h\boldsymbol{\Delta}}. \tag{3.14}$$

The performance of using an SPSA based optimizer was ultimately disappointing; however, it is possible that further investigation could be necessary and the idea revisited in the future.

CHAPTER 4: RESULTS

We will be comparing the ML agent against a *random aggregation* and *Lloyd aggregation* methods, readily taken from PyAMG [29]. This algorithm is *seeded* via a random set of aggregate centers which are then iteratively refined by an implementation of Lloyd's method (also known as K-Means) reformulated for graph inputs [18]. One may conject that the performance of Lloyd aggregation may be dependent on the initial seeding; to reduce any affects of randomness the initial seeding has been fixed for all comparisons.

To allow the method to output something sensible for anisotropic and generally more difficult problems, the distance between nodes is defined by a heuristic *strength-of-connection* measure. Here, we use a combination of an evolution-based measure [30] and the inverse of the magnitude of the entries of the system. Formally, let the strength measure being used be defined as

$$\boldsymbol{C} := \boldsymbol{S} + \frac{1}{|\boldsymbol{A}|}, \tag{4.1}$$

where $\boldsymbol{S}$ is the evolution measure, $\boldsymbol{A}$ is the original system, and $\frac{1}{|\cdot|}$ is applied entry-wise to $\boldsymbol{A}$. This strength measure is hereby referred to as the *Olson* strength-measure[1].

Once a set of aggregate centers is found, each remaining node is assigned to an aggregate by finding its nearest (by means of $\boldsymbol{C}$) center node. This is done by a modified Bellman-Ford algorithm [18], whose implementation is given in Alg. A.1. After each nearest center node is found, a tentative aggregate operator is returned by allotting each aggregate a column in $\text{Agg} \in \mathbb{R}^{n \times c}$, where $\text{Agg}_{ij} = 1$ iff node i belongs in aggregate j.

We then form the interpolation operator using the usual Jacobi smoother (2.8). We will also use the *random method* outlined in 2.2.

The ML method was compared against the baseline using a coarsening ratio of $\alpha = 0.1$. Performance was measured by *average rate of convergence* for some set of problems; a two-level multigrid solver was run for a set tolerance to approximate asymptotic convergence rate. Detailed information on convergence is given in tables 4.1, 4.2. In Table 4.1, Each ML agent is trained for that specific class of problem, i.e. *isotropic* or *anisotropic*; table 4.2 contains a single general agent trained on both classes of problems.

[1] Thanks to Prof. Scott MacLachlan for the name

Problem Type	Data Set	Lloyd Conv.	ML Conv.
Isotropic	Train	0.4208	0.4109
Isotropic	Test	0.4177	0.4075
Anisotropic	Train	0.7705	0.7451
Anisotropic	Test	0.7978	0.7697

Table 4.1: Convergence of ML vs baseline Lloyd+SA on various sets of problems, individual networks trained on problem type.

Additionally, an agent was trained on datasets of both isotropic and anisotropic problems (the union of the problems in Table 4.1. The results of this new network are given in 4.2, as well as an ablation performed by replacing one of the aggregation and smoothing networks with its respective baseline component.

Problem Type	Data Set	Baseline	Lloyd	Full ML	ML Agg.	ML Int.
Isotropic	Train	0.4652	0.4208	0.3956	0.3974	0.4190
Isotropic	Test	0.4623	0.4177	0.3913	0.3922	0.4172
Anisotropic	Train	0.7680	0.7705	0.7462	0.7532	0.7614
Anisotropic	Test	0.7902	0.7978	0.7727	0.7776	0.7910

Table 4.2: Convergence of ML vs baseline, Lloyd on various sets of problems, one agent trained on both isotropic and anisotropic problems. Columns with **ML Agg.** and **ML Int.** are ablation studies and refer to ML aggregation + Jacobi smoothing, Lloyd aggregation + ML smoothing, respectively.

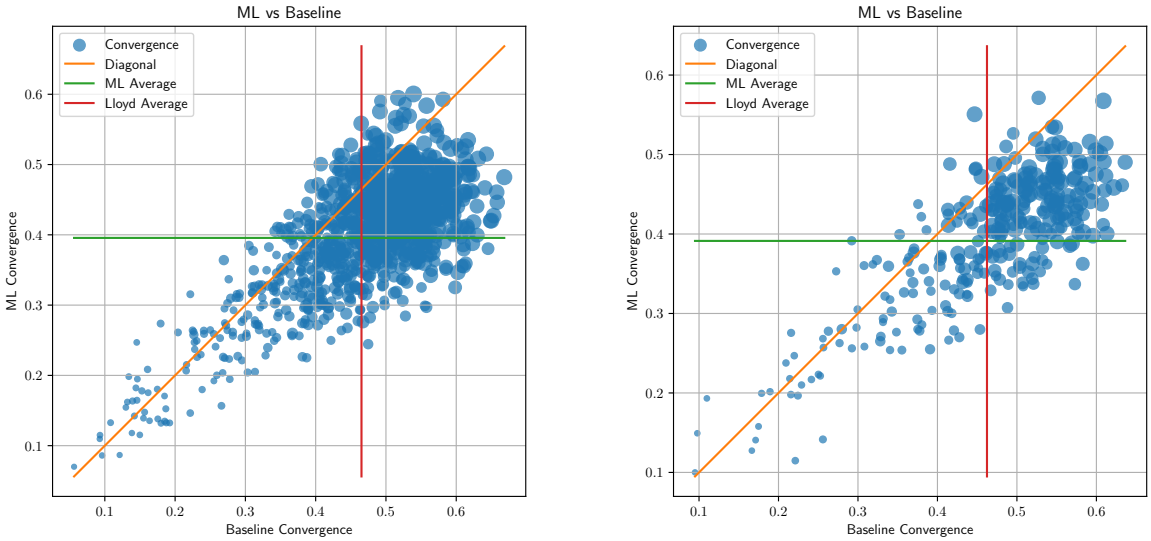
The methods are also run on a few test problems and their outputs displayed (Figures 4.4, 4.5, 4.6, 4.7, 4.12, 4.13, 4.14). As the meaning of the figures may be somewhat ambiguous, we will explain how to read them:

- Thin blue lines around multiple nodes indicates an *aggregate*.
- A *red star with yellow background* on a node indicates the *aggregate root*.
- Colors on edges range from dark blue (low magnitude) to bright yellow (high magnitude) and indicate strength values that are used.
- A *red star without background* indicates mean-point for aggregate interpolation.
- Colorful lines from aggregate mean-points to nodes depict interpolation strength; high weights have high opacity while low weights have low opacity.

All following numerical results were generated from the model trained on both sets of problems, i.e. the one for which data is given in Table 4.2.

4.1 ISOTROPIC DIFFUSION

Overall convergence of the ML agent versus baseline and Lloyd aggregation is given in Figure 4.1. Results indicate that both Lloyd and the ML agent are able to improve the solver significantly as compared to the random baseline. Convergence is also plotted against problem size itself in Figures 4.2 and 4.3, indicating the agent tends to do slightly worse on larger problems.

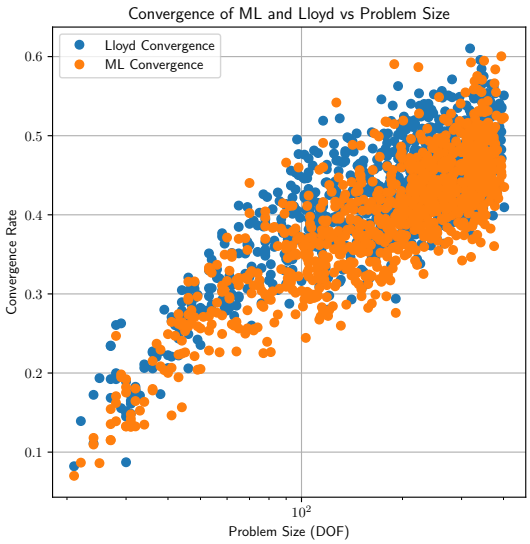


(a) Training convergence

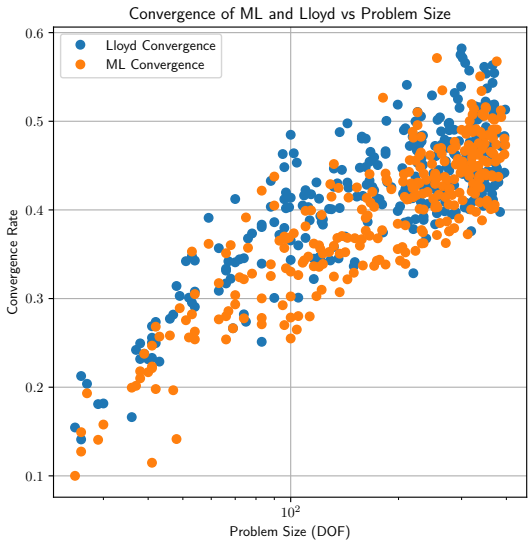
(b) Testing convergence

Figure 4.1: Convergence data for the ML AMG method vs the Lloyd method. Values below the diagonal indicate a better convergence for the ML. Markers are scaled by problem size.

A few test problems are shown in Figures 4.4, 4.5, 4.6, 4.7. It is interesting to note how the ML agent uses the problem boundary to its advantage, creating larger aggregates along the boundary as higher resolution is not needed (since we have homogeneous Dirichlet conditions).

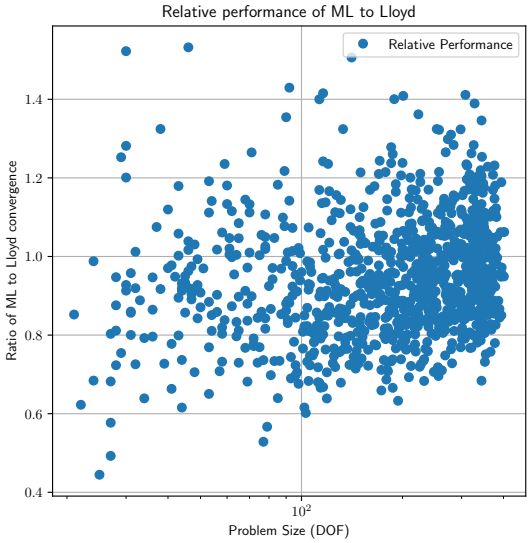


(a) Training convergence

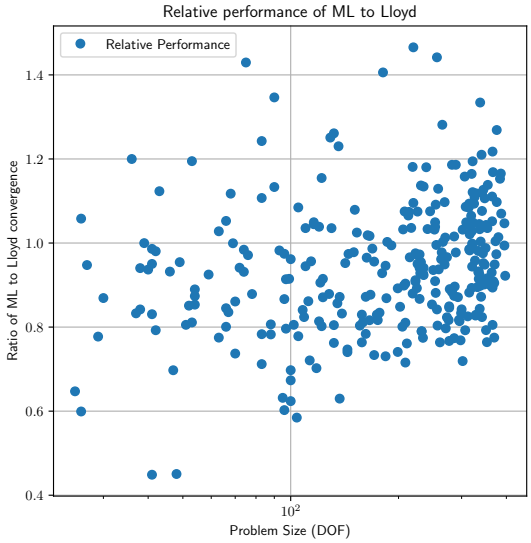


(b) Testing convergence

Figure 4.2: Convergence data for the two methods plotted against problem size (DOF).

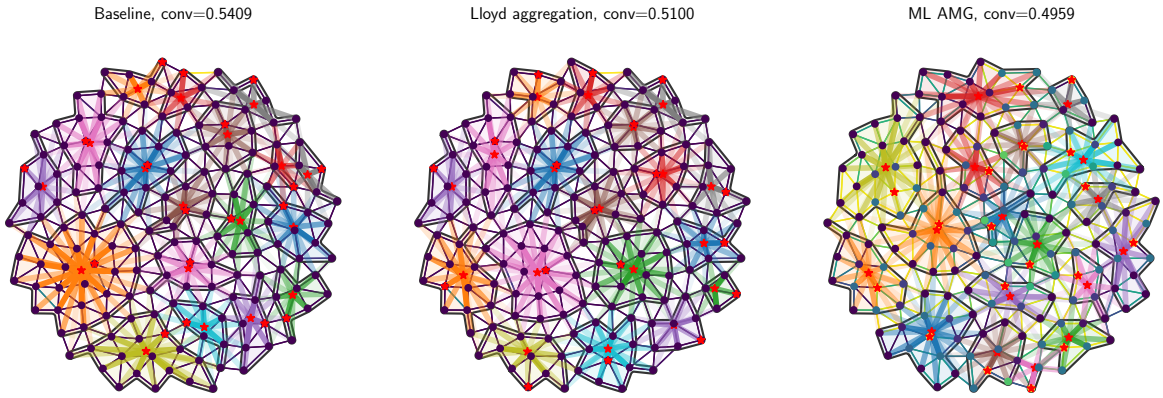


(a) Relative training performance

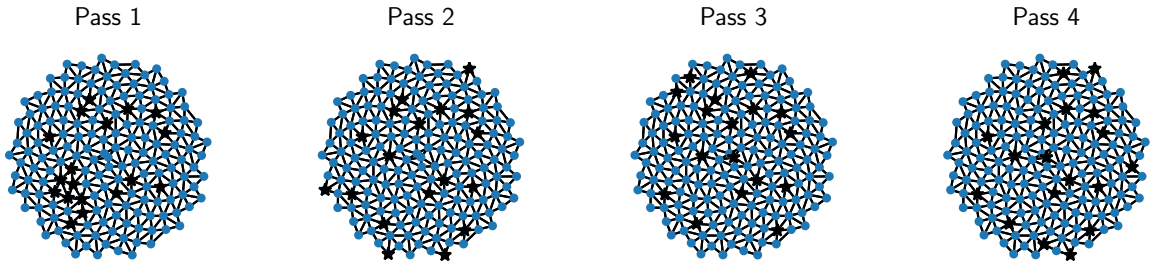


(b) Relative testing performance

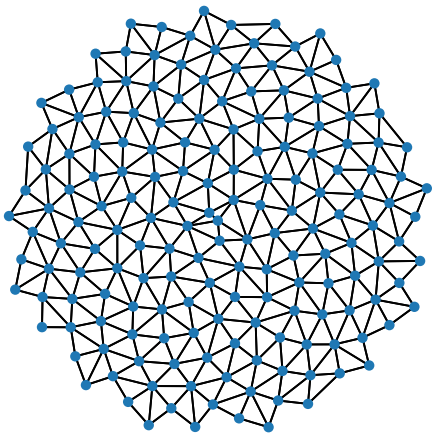
Figure 4.3: Relative performance of the ML to the Lloyd method, plotted against problem size. Relative performance is obtained by dividing the ML convergence by the Lloyd convergence for each problem. Values below 1 indicate better ML performance, while values above 1 indicate better baseline performance.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

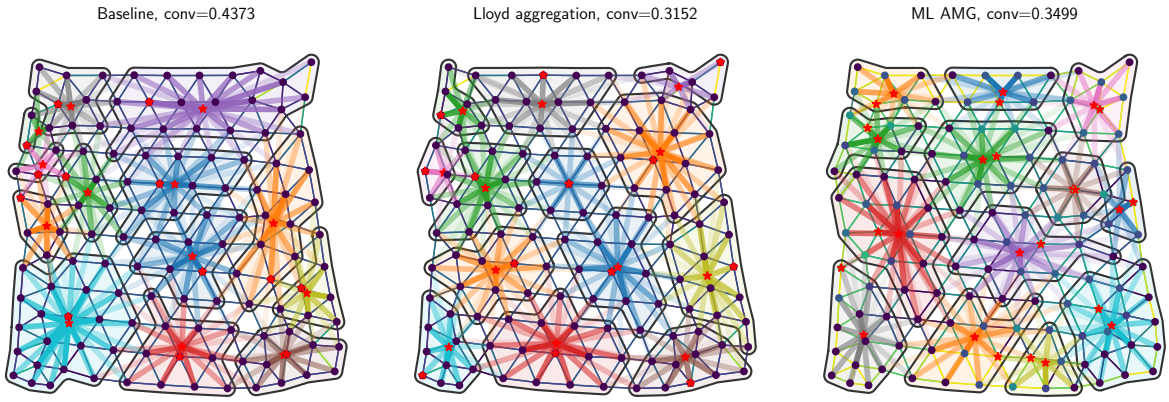


(d) Tentative aggregate centers after each binarization "pass".

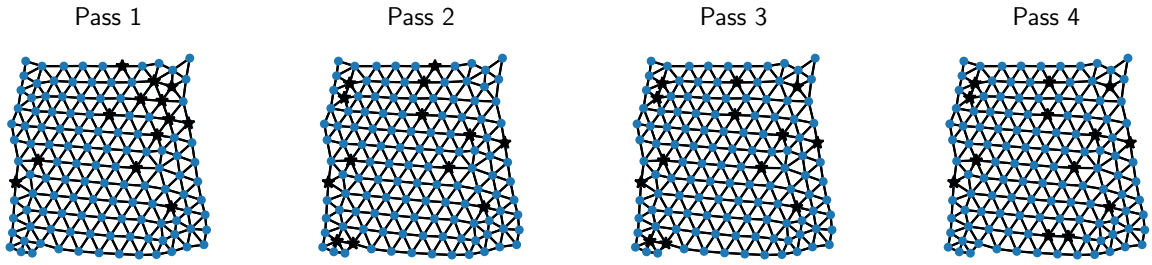


(e) Visualization of degrees-of-freedom in problem.

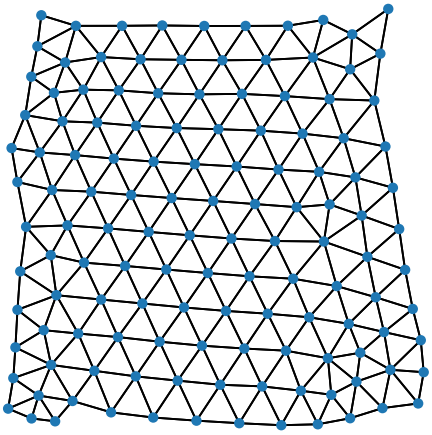
Figure 4.4: Aggregate and interpolation data for a circular unstructured mesh. This particular mesh has 173 DoF.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

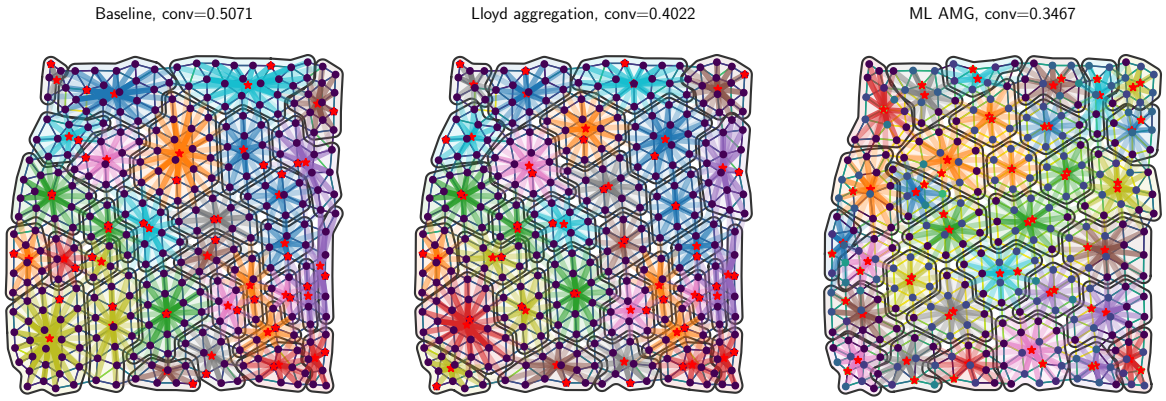


(d) Tentative aggregate centers after each binarization "pass".

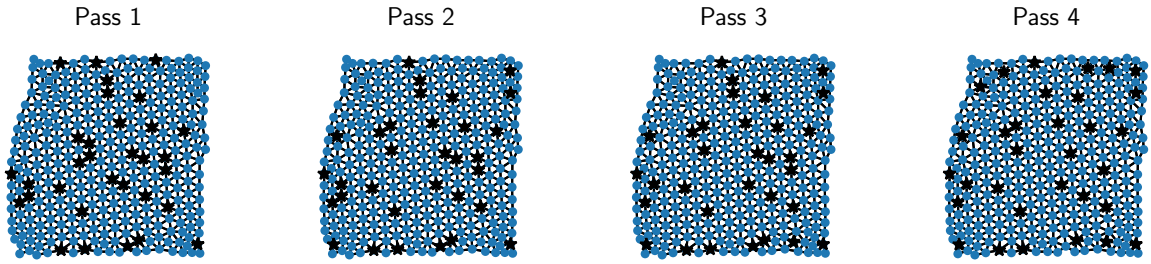


(e) Visualization of degrees-of-freedom in problem.

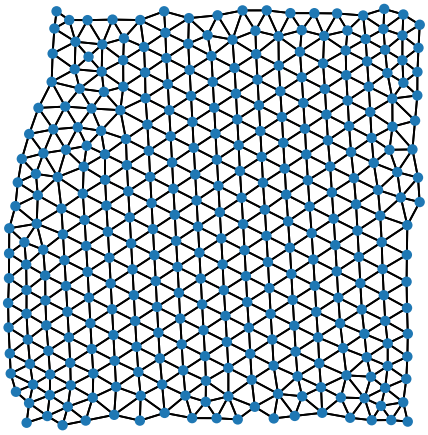
Figure 4.5: Aggregate and interpolation data for a random unstructured mesh. This particular mesh has 220 DoF.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

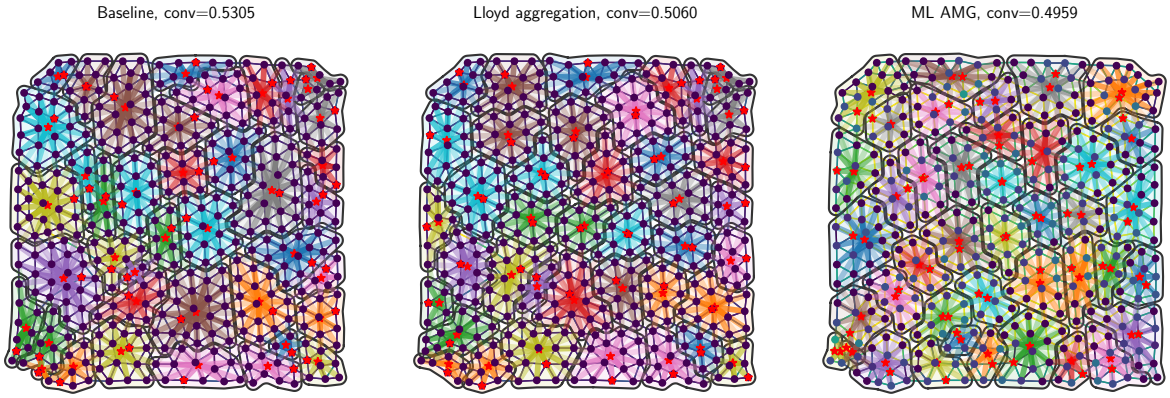


(d) Tentative aggregate centers after each binarization "pass".

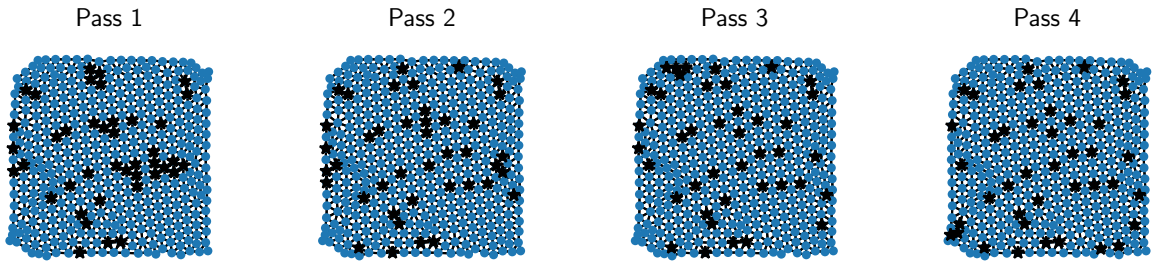


(e) Visualization of degrees-of-freedom in problem.

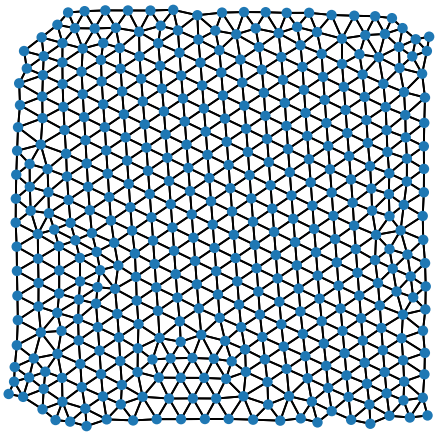
Figure 4.6: Aggregate and interpolation data for a random unstructured mesh. This particular mesh has 273 DoF.



(a) Random Baseline (b) Lloyd Aggregation (c) ML



(d) Tentative aggregate centers after each binarization "pass".



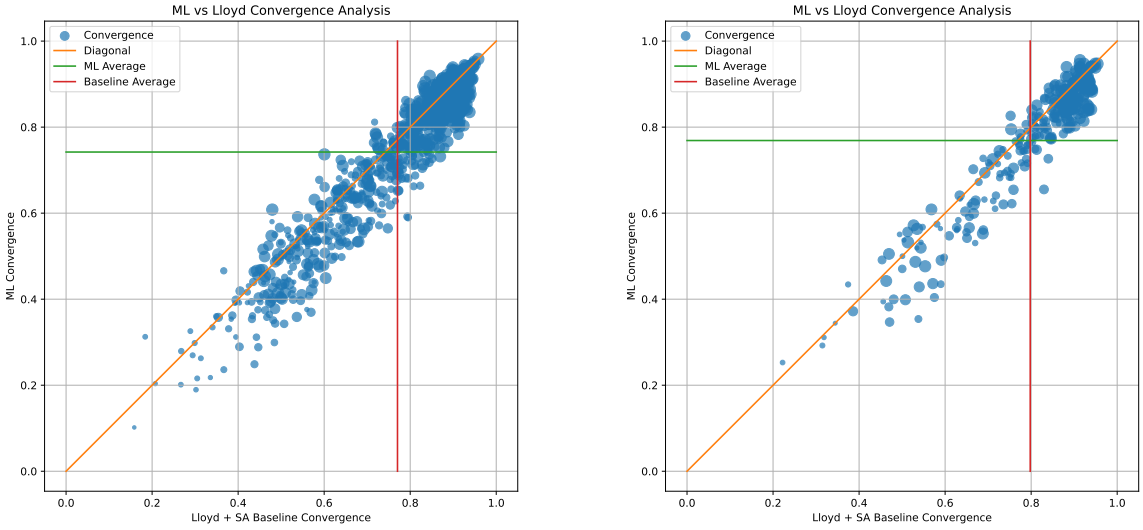
(e) Visualization of degrees-of-freedom in problem. Irregularities in element size are visible on left and bottom of domain.

Figure 4.7: Aggregate and interpolation data for the largest mesh in the *training set*. This particular mesh has 410 DoF.

4.2 ANISOTROPIC DIFFUSION

Again, overall convergence of the ML agent versus baseline and Lloyd aggregation is given in Figure 4.8. Results are no longer favorable for the Lloyd aggregation, indicating that on average it is unable to improve upon the random aggregates, while the agent provides some benefit. It is probable that a more optimal strength-of-connection measure is required for the Lloyd aggregation method to remain competitive; the ML method is advantageous in that it can select the strength-of-connection. Overall convergence for both the Lloyd and ML methods is also plotted against problem size itself in Figures 4.9 and 4.11.

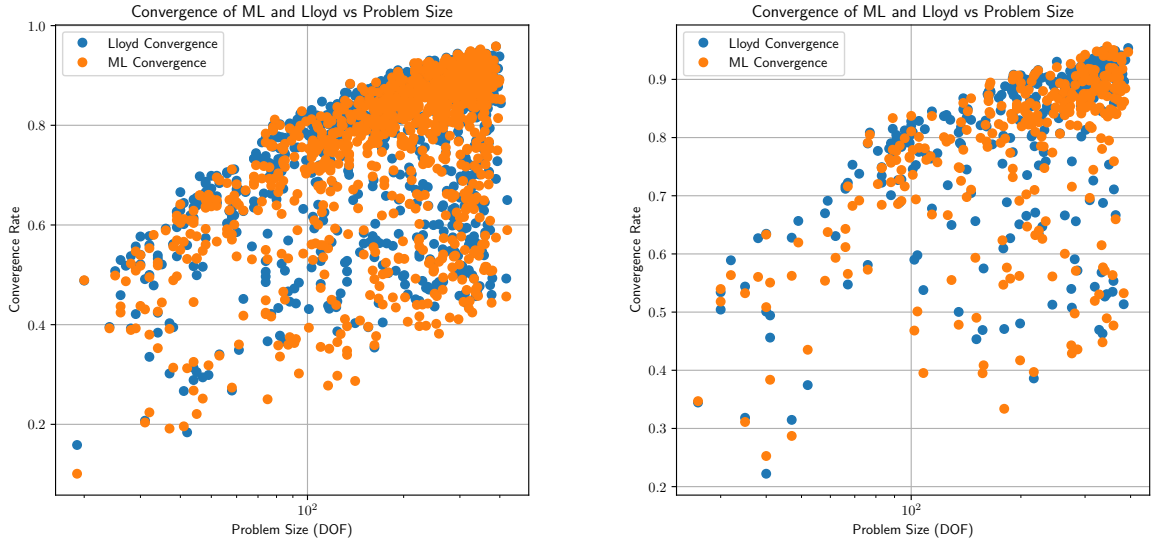
A few test problems are shown in Figures 4.12, 4.13, 4.14. Rotation and scaling anisotropy values are given in both text format in captions and graphically as a "compass" in the ML sub-figures. The same interesting boundary behavior can be seen by the agent in these anisotropic examples.



(a) Training convergence

(b) Testing convergence

Figure 4.8: Convergence data for the ML AMG method vs a baseline Lloyd and Jacobi SA method. Values below the diagonal indicate a better convergence for the ML. Markers are scaled by problem size.



(a) Training convergence

(b) Testing convergence

Figure 4.9: Convergence data for the two methods plotted against problem size (DOF).

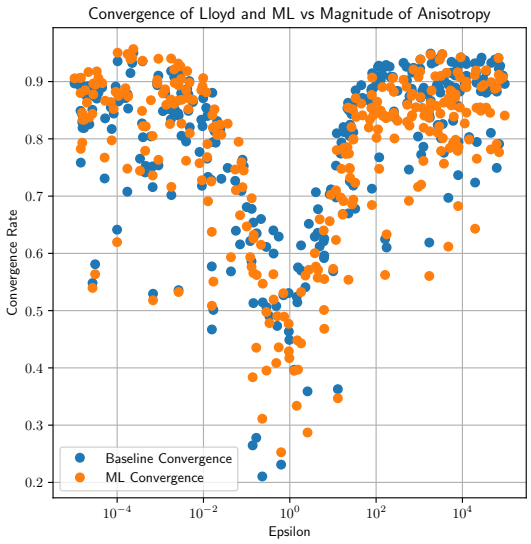
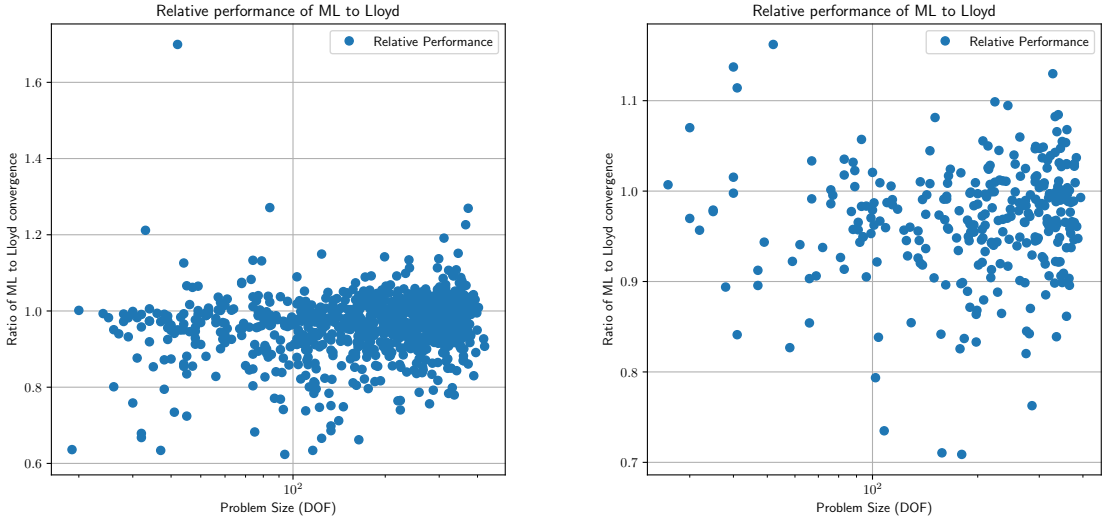
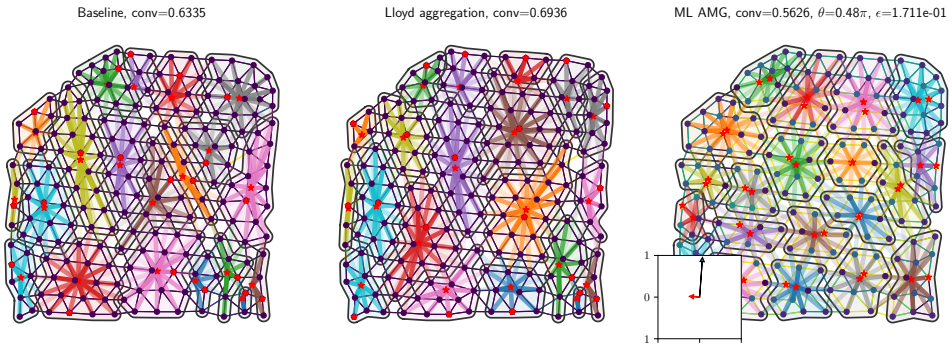


Figure 4.10: Convergence on the testing set per value of anisotropic scaling, ε.

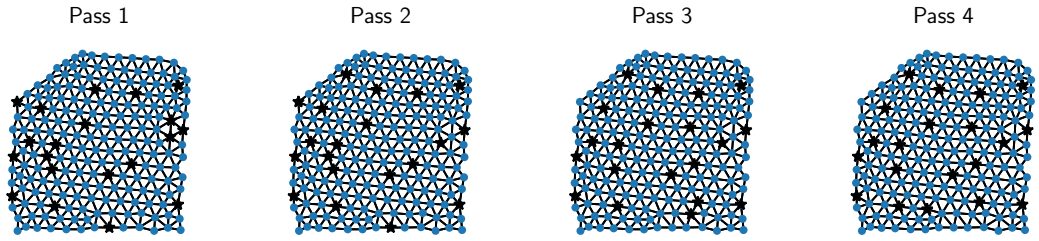


(a) Relative training performance (b) Relative testing performance

Figure 4.11: Relative performance of the ML to the Lloyd method, plotted against problem size. Relative performance is obtained by dividing the ML convergence by the Lloyd convergence for each problem. Values below 1 indicate better ML performance, while values above 1 indicate better baseline performance.

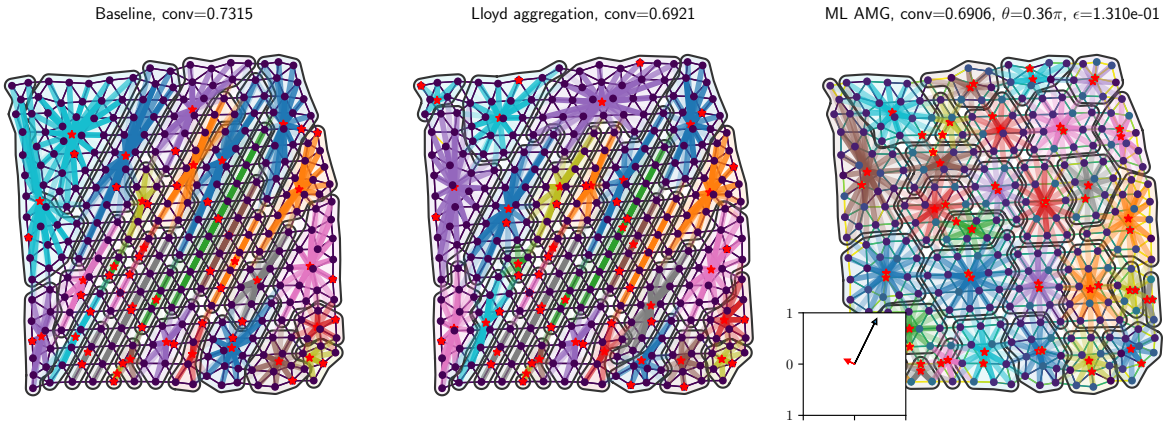


(a) Random Baseline (b) Lloyd Aggregation (c) ML

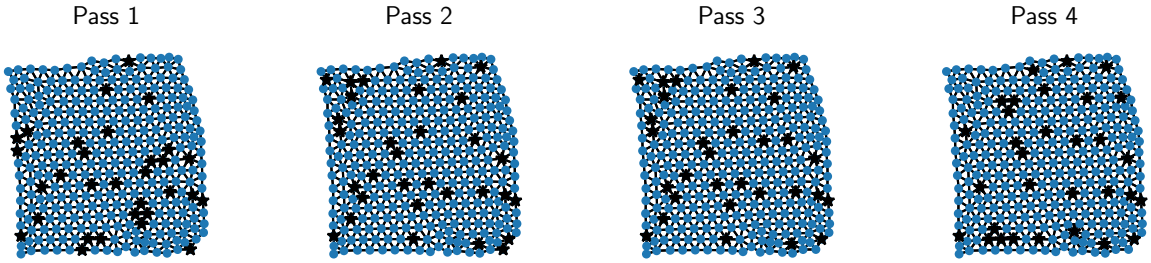


(d) Tentative aggregate centers after each binarization "pass".

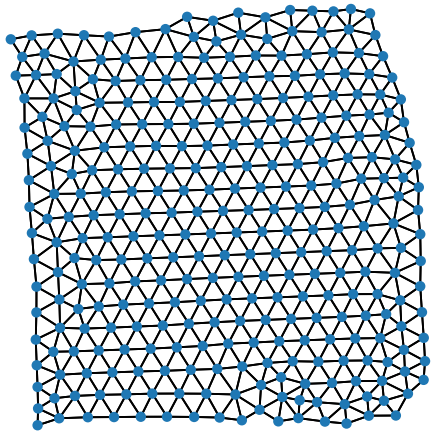
Figure 4.12: Aggregate and interpolation data for an anisotropic problem with 198 DoF. The problem is stretched by a factor of $\varepsilon = 0.177$ along $\theta = 0.98\pi$.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

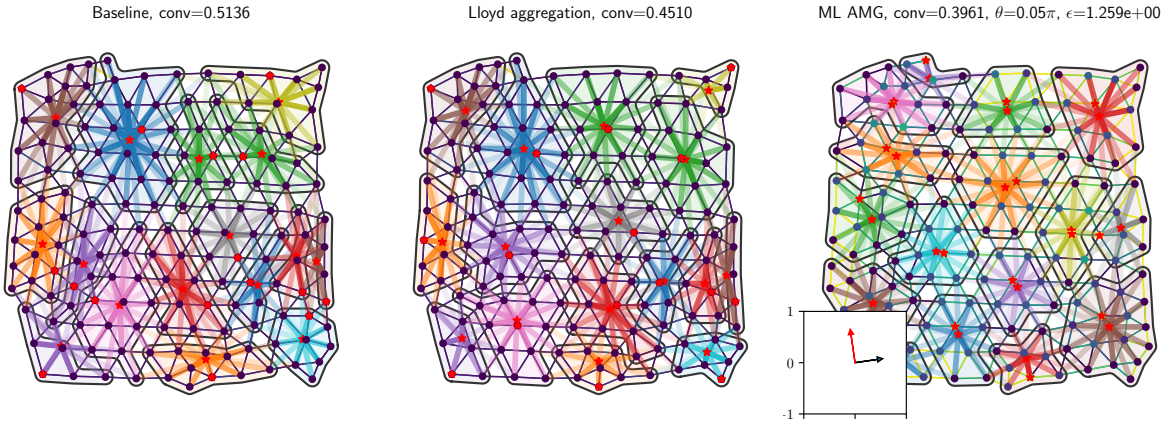


(d) Tentative aggregate centers after each binarization "pass".

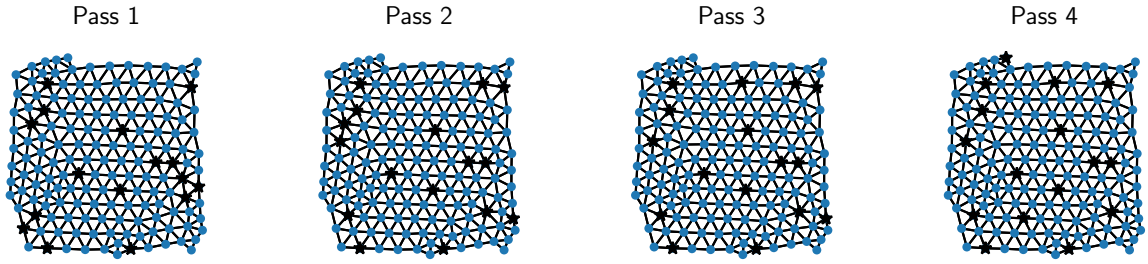


(e) Visualization of degrees-of-freedom in problem. A small "knot" of nodes is visible on the bottom of the mesh that can provide difficulties for the aggregation method.

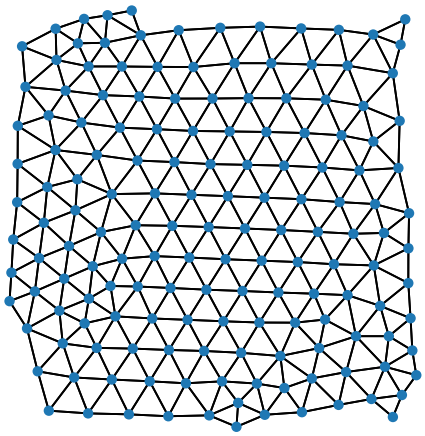
Figure 4.13: Aggregate and interpolation data for an anisotropic problem with 309 DoF. The problem is stretched by a factor of $\varepsilon = 0.131$ along $\theta = 0.86\pi$.



(a) Random Baseline (b) Lloyd Aggregation (c) ML



(d) Tentative aggregate centers after each binarization "pass".



(e) Visualization of degrees-of-freedom in problem.

Figure 4.14: Aggregate and interpolation data for an anisotropic problem with 157 DoF. The problem is stretched by a factor of $\varepsilon = 1.259$ along $\theta = 0.55\pi$.

4.3 LARGE ISOTROPIC PROBLEMS

The ML agent was also evaluated on a set of *larger* isotropic diffusion problems whose overall DoF were in the range of between 2000 and 3000 variables. This dataset was evaluated on both the generic agent trained in sections 4.1, 4.2 and also a agent specially trained on these large problems. Overall convergence data can be seen in Table 4.3.

Baseline Conv.	**Lloyd Conv.**	**ML (Untrained)**	**ML (Trained)**
0.6539	0.5663	0.6214	0.5860

Table 4.3: Convergence of ML vs baseline, Lloyd on large isotropic problems.

It is interesting to note that with special training, the ML method is able to approach (but not exceed) convergence comparable to Lloyd aggregation. Perhaps one reason one reason for the large drop in performance by the ML agent is due to the boundaries having less influence over the problem; the agent was able to exploit the boundary on smaller problems to reduce convergence and such tricks are no longer beneficial on these larger problems.

This also suggests that in a sense, Lloyd aggregation is optimal for these sorts of larger, isotropic diffusion problems. Due to the relative simplicity of the diffusion present, aggregates need only be evenly spaced in the domain.

4.4 SPECIAL COEFFICIENT PROBLEMS

For the remaining special problems (defined in 3.2), we first evaluate each problem set on the agent trained on anisotropic and isotropic problems (sec 4.1, 4.2) — this is done to establish how the agent can generalize to problem types it has not seen before, and these results will be referred to as the *untrained agent*. Afterwards, the agent was trained on each specific dataset to determine optimal performance: these results are enumerated in Table 4.4. The results seem to indicate that even on problems the agent has not been specifically trained on, we can expect to at least match or exceed the performance for the random baseline. The benefit that we see from the ML agent vs. the baseline or Lloyd methods seems to degrade as the complexity increases, however. We also see an improvement in every problem set if the agent is specially trained on the specific set of problems.

We have taken special care to look at the jumping-coefficients problem in further detail; Table 4.5 separates convergence data for problems containing two or three regions. While the convergence benefit degrades in problems with three regions, in all cases the agent is able to outperform both the baseline and Lloyd aggregation. One can note upon inspecting

Problem Set	Baseline Conv.	Lloyd Conv.	ML (Untrained)	ML (Trained)
Smooth	0.5515	0.6058	0.4919	0.4610
Jumping	0.5451	0.5889	0.5211	0.4965
Noisy	0.8853	0.9147	0.8770	0.8510

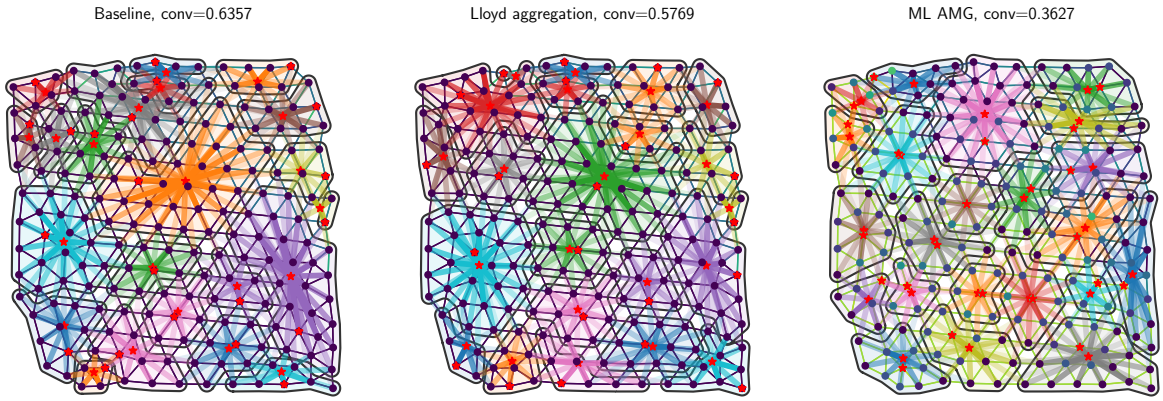
Table 4.4: Convergence of ML vs baseline, Lloyd on specially problems.

the output (Figure 4.15) that the strength of connection visibily shows the distinct regions present in the problem.

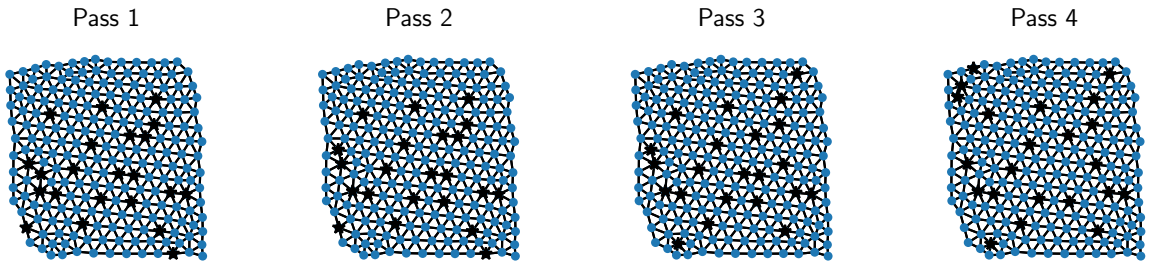
Regions	Baseline Conv.	Lloyd Conv.	ML (Untrained)	ML (Trained)
All	0.5451	0.5889	0.5211	0.4965
2	0.5575	0.5778	0.5229	0.4792
3	0.5336	0.5992	0.5195	0.5126

Table 4.5: Convergence of ML vs baseline, Lloyd on Jumping-coefficents problem.

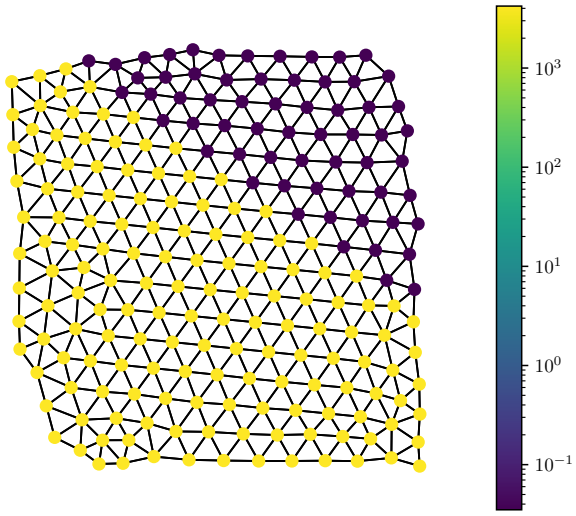
Examples of all three methods are shown for two specific problems in each set: Figures 4.15, 4.16 display jumping coefficients problems; Figures 4.17, 4.18 show problems with smoothly-varying coefficients; Figures 4.19, 4.20 are for problems with random/noisy coefficients.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

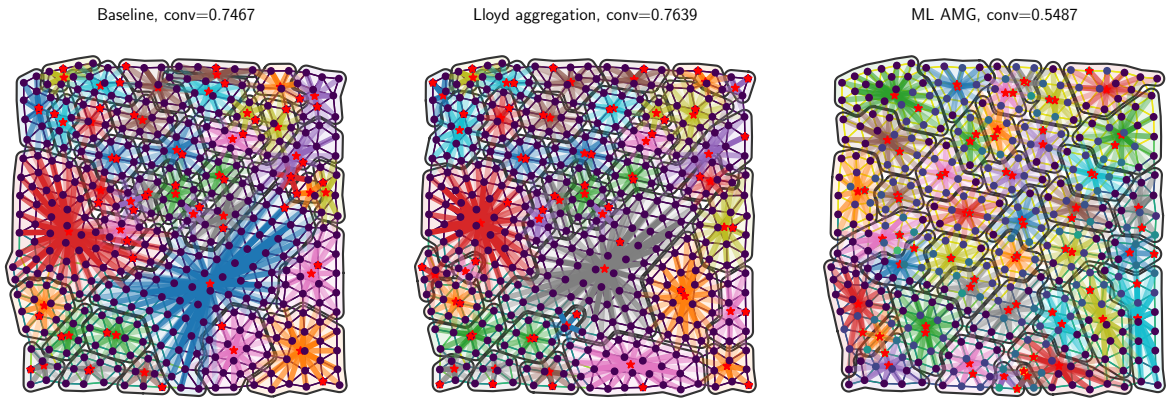


(d) Tentative aggregate centers after each binarization "pass".

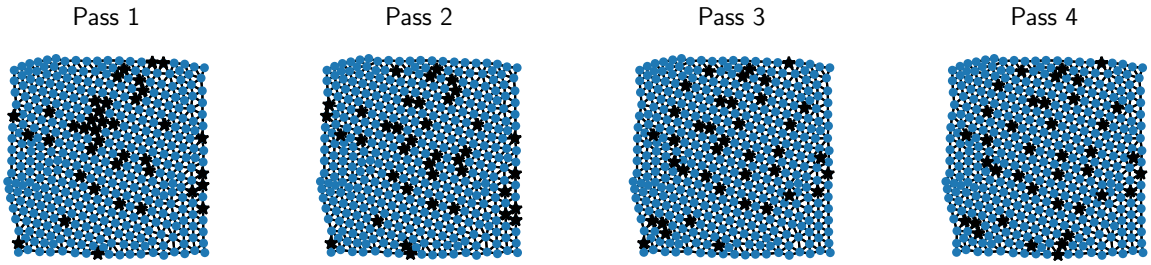


(e) Visualization of problem with jumps present.

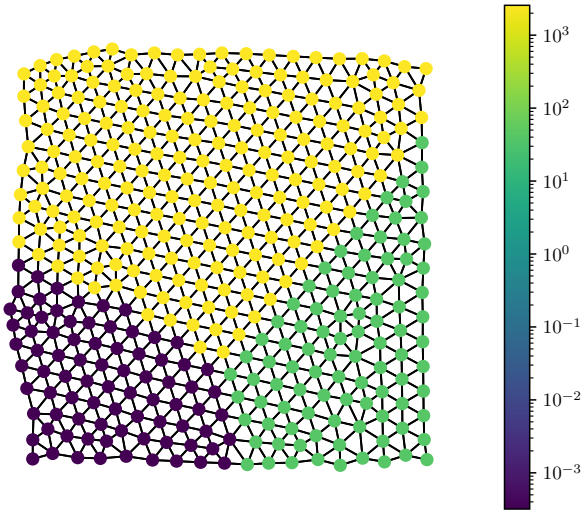
Figure 4.15: Aggregate and interpolation data for a jumping coefficients problem with 211 DoF. The ML method forms aggregates along the jump boundary.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

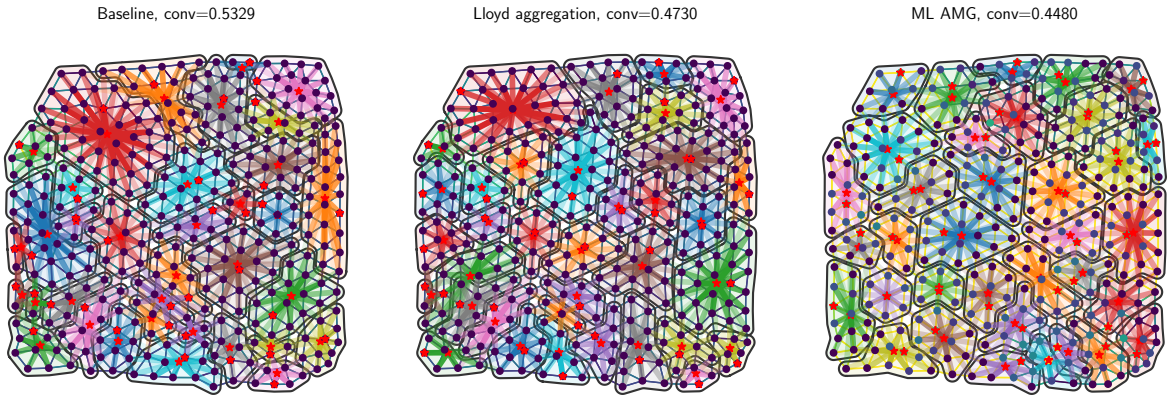


(d) Tentative aggregate centers after each binarization "pass".

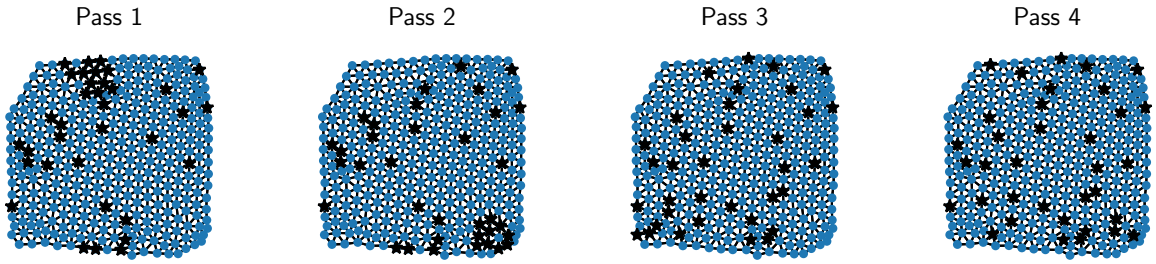


(e) Visualization of problem with jumps present.

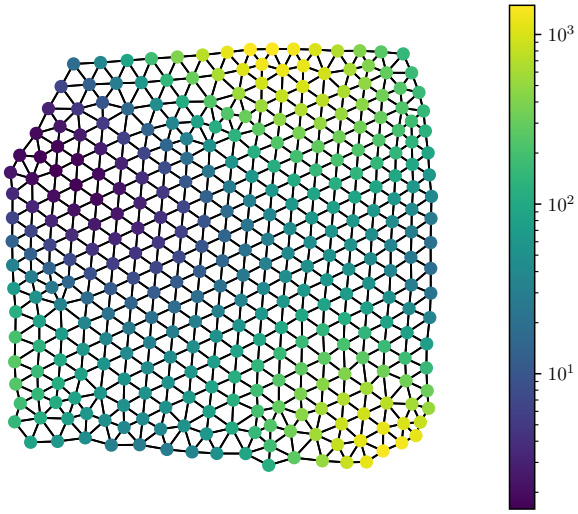
Figure 4.16: Aggregate and interpolation data for a jumping coefficients problem with 390 DoF and three distinct regions. The ML method again forms aggregates along the jump boundary.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

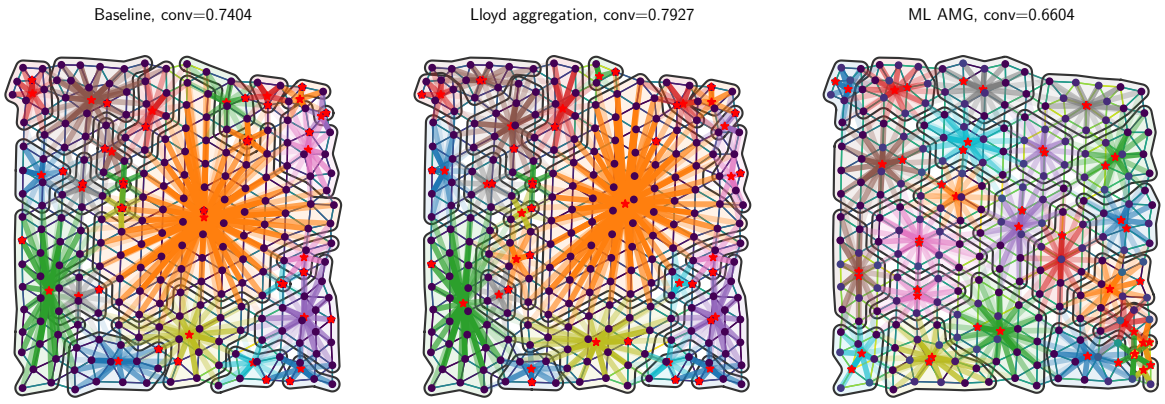


(d) Tentative aggregate centers after each binarization "pass".

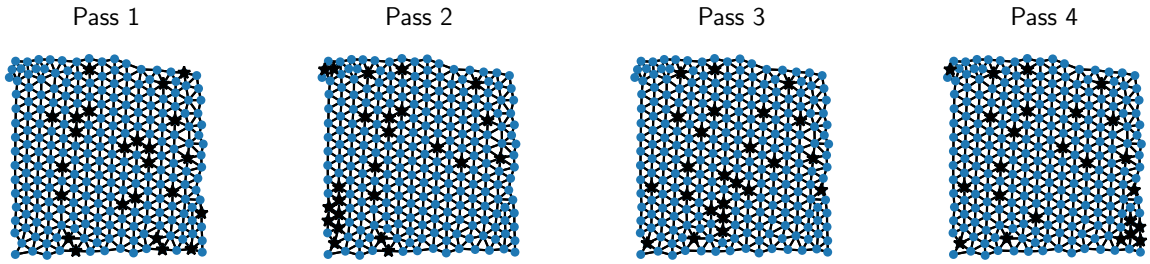


(e) Visualization of problem with varying diffusion coefficients.

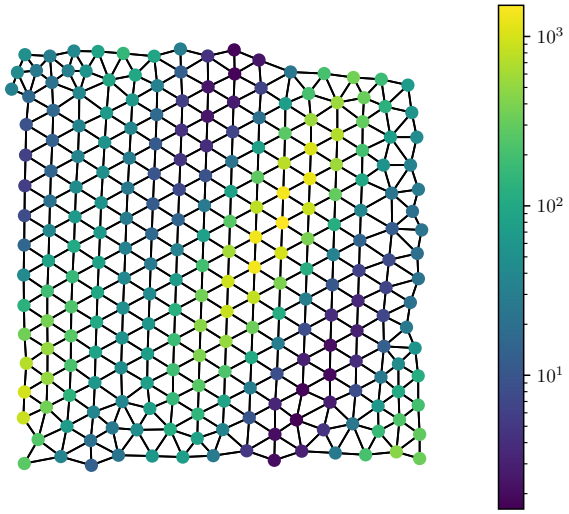
Figure 4.17: Aggregate and interpolation data for smoothly-varying coefficients problem with 348 DoF. The ML method provides higher resolution in regions with higher diffusivity (top, bottom right).



(a) Random Baseline (b) Lloyd Aggregation (c) ML

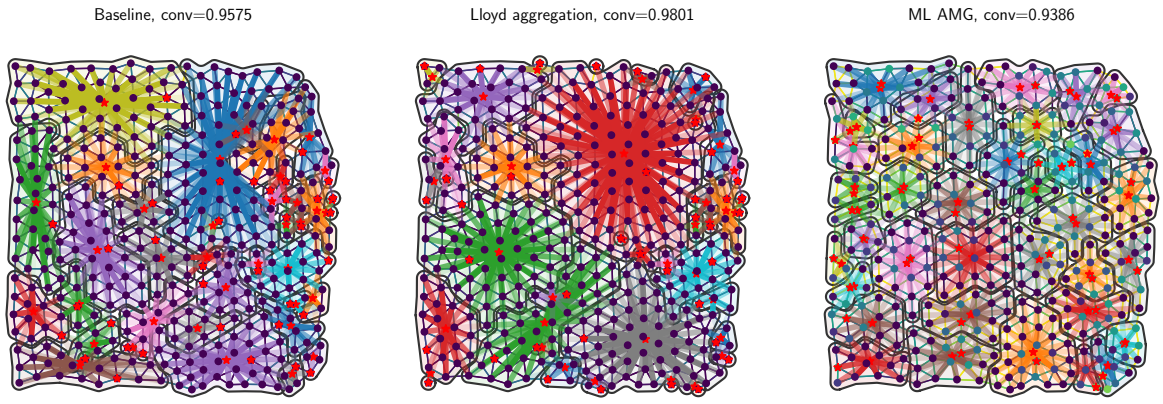


(d) Tentative aggregate centers after each binarization "pass".

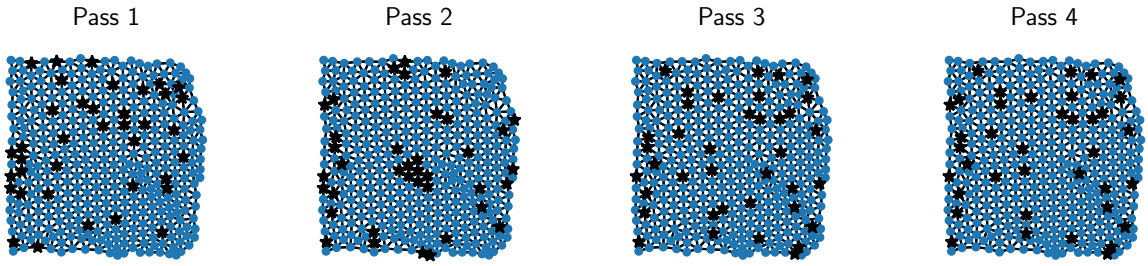


(e) Visualization of problem with varying diffusion coefficients.

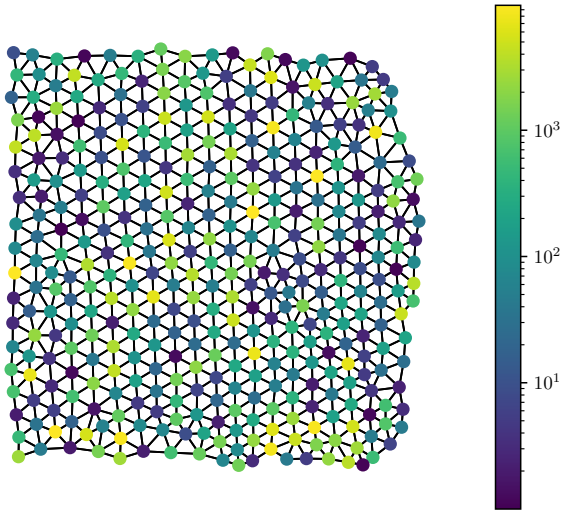
Figure 4.18: Aggregate and interpolation data for smoothly-varying coefficients problem with 238 DoF. The ML method provides roughly evenly-shaped aggregates that encapsulate both high and low values of diffusivity.



(a) Random Baseline (b) Lloyd Aggregation (c) ML

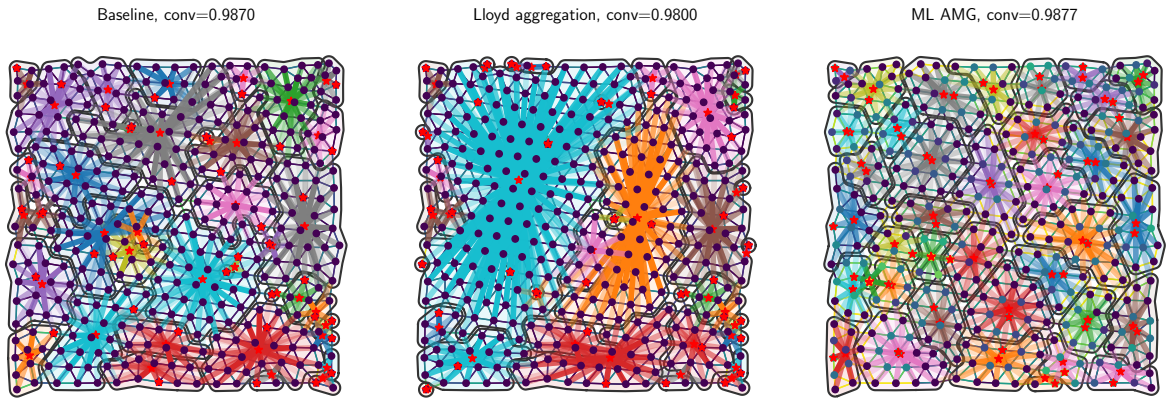


(d) Tentative aggregate centers after each binarization "pass".

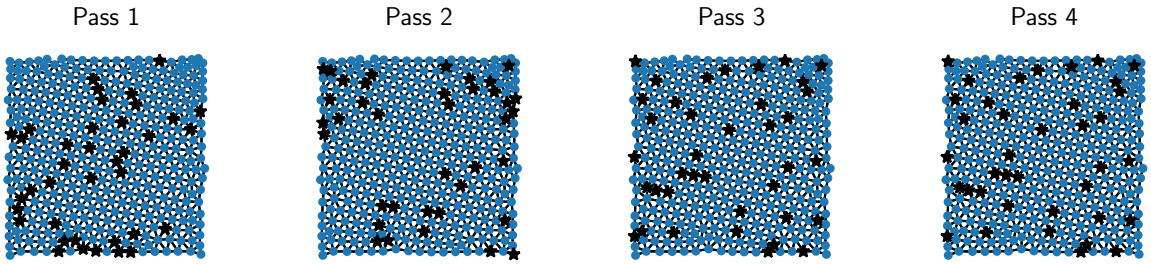


(e) Visualization of problem with varying diffusion coefficients.

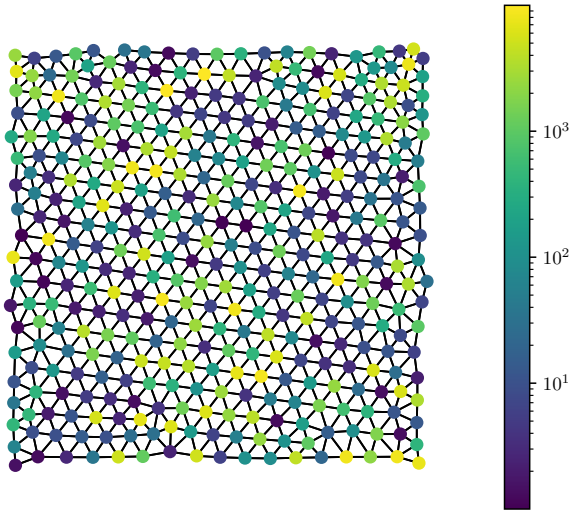
Figure 4.19: Aggregate and interpolation data for noisy coefficients problem with 373 DoF. The ML method forms regular aggregates as opposed to the erraticly-sized aggregates given by Lloyd.



(a) Random Baseline (b) Lloyd Aggregation (c) ML



(d) Tentative aggregate centers after each binarization "pass".



(e) Visualization of problem with varying diffusion coefficients.

Figure 4.20: Aggregate and interpolation data for noisy coefficients problem with 375 DoF. The ML method once again forms regular aggregates as opposed to the erratic aggregates output by Lloyd.

CHAPTER 5: CONCLUSIONS

In this work, we have developed an ML *agent* to output aggregates and a smoothing operator for smoothed-aggregation multigrid. Somewhat surprisingly, gradient-free genetic algorithms are useful and can provide good results in training such networks that could not otherwise be trained with traditional descent-based methods. Additionally, genetic algorithms avoid the problem of becoming *stuck* in local minima in the optimization space.

Methods based on graph networks can actually be used to learn smoothed-aggregation. On average, the ML method can meet or exceed the performance of the Lloyd-based method. Average convergence over both isotropic and anisotropic datasets is lower than average convergence for both the baseline and Lloyd methods. Additionally, this agent can provide something akin to a black box: a strength-of-connection and tweaking to the parameters is not needed and some reasonable set of parameters is automatically built-in (learned) by the method.

There are definitely more directions in which this work can be taken: could we try out more difficult PDEs, maybe trying problems in which conventional AMG does not do well in setting up a solver? There is also the possibility of reformulating the ML method so that gradient information is readily available and can be trained with traditional methods; one such way to do this would be to perhaps re-cast as a reinforcement-learning problem. In the future, one could also explore looking at giving the ML agent more autonomy in picking the aggregate and final interpolation operators — it is currently limited to only selecting aggregate centers and a smoother operator. More parts of the SA procedure could be replaced with ML components to potentially learn deeper or overall more optimal interpolation.

APPENDIX A: GRAPH ALGORITHMS

A modified implementation of Bellman-Ford used to compute shortest paths from every vertex to every root node is presented in Algorithm A.1. This implementation is the same as presented by Bell [18].

Algorithm A.1 Modified Bellman-Ford

```
procedure BellmanFord(C ∈ ℝ^{n×n}, centers ∈ ℝ^n)
    dist := [∞, ∞, ..., ∞] ∈ ℝ^n
    nearest := [0, 0, ..., 0] ∈ ℝ^n

    for c ∈ centers do
        dist_c ← 0
        nearest_c ← c
    end for

    finished := True
    while ¬finished do
        finished ← True
        for (i, j) such that C_ij ≠ 0 do
            if distance_i + C_ij < distance_j then
                distance_j ← distance_i + C_ij
                nearest_j ← nearest_i
                finished ← False
            end if
        end for
    end while

    return distance, nearest
end procedure
```

Additionally, a simple algorithm to assign nodes in an assignment matrix is presented in Algorithm A.2. This runs in linear time assuming the inputs are passed directly from Bellman-Ford.

Algorithm A.2 Assignment of nodes to aggregates

procedure AssignAggregates($\texttt{centers} \in \mathbb{R}^c, \texttt{nearest} \in \mathbb{R}^n$)
    $\texttt{center\_number} := \{\}$
    $\text{Agg} \in \mathbb{R}^{n \times c}$

    for $i \leftarrow 1, c$ **do**
        $\texttt{center\_number}\,[\texttt{centers}_i] \leftarrow i$ ▷ Assign each center a column in Agg
    end for
    for $i \leftarrow 1, n$ **do**
        $j := \texttt{center\_number}\,[\texttt{centers}_i]$ ▷ Find the correct column based on the center
        $\text{Agg}_{ij} = 1$
    end for
    return Agg
end procedure

APPENDIX B: PROOFS

B.1 MULTIGRID COLUMN-SCALING INVARIANCE

The multigrid cycle is invariant to any nonzero scaling of the columns of the interpolation operator.

Proof. Let $\boldsymbol{P}$ be the unscaled interpolation operator, and $\bar{\boldsymbol{P}} = \boldsymbol{P}\boldsymbol{\Sigma}$ be some operator whose columns are scaled by a square, diagonal scaling matrix $\boldsymbol{\Sigma}$. We assume $\boldsymbol{\Sigma}$ to have nonzero values on the diagonal, i.e. that $\boldsymbol{\Sigma}$ is full rank and thus invertible.

Using the scaled interpolator, define the multigrid coarse grid solve as

$$\boldsymbol{A}_H \boldsymbol{e}_H = \boldsymbol{r}_H = \bar{\boldsymbol{P}}^T \boldsymbol{r}_h. \tag{B.1}$$

This implies that

$$\begin{aligned}
\boldsymbol{e}_H &= \boldsymbol{A}_H^{-1} \bar{\boldsymbol{P}}^T \boldsymbol{r}_h && \text{(B.2)}\\
&= \left(\bar{\boldsymbol{P}}^T \boldsymbol{A}_h \bar{\boldsymbol{P}}\right)^{-1} \bar{\boldsymbol{P}}^T \boldsymbol{r}_h && \text{(B.3)}\\
&= \left(\boldsymbol{\Sigma}^T \boldsymbol{P} \boldsymbol{A}_h \boldsymbol{P}^T \boldsymbol{\Sigma}\right)^{-1} \boldsymbol{\Sigma}^T \boldsymbol{P}^T \boldsymbol{r}_h. && \text{(B.4)}
\end{aligned}$$

If we interpolate (B.4) to the fine grid, we obtain

$$\begin{aligned}
\boldsymbol{e}_h &= \bar{\boldsymbol{P}} \boldsymbol{e}_H && \text{(B.5)}\\
&= \boldsymbol{P}\boldsymbol{\Sigma} \left(\boldsymbol{\Sigma}^T \boldsymbol{P}^T \boldsymbol{A}_h \boldsymbol{P} \boldsymbol{\Sigma}\right)^{-1} \boldsymbol{\Sigma}^T \boldsymbol{P}^T \boldsymbol{r}_h && \text{(B.6)}\\
&= \boldsymbol{P}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-1} \left(\boldsymbol{P}^T \boldsymbol{A}_h \boldsymbol{P}\right)^{-1} \boldsymbol{\Sigma}^{-T} \boldsymbol{\Sigma}^T \boldsymbol{P}^T \boldsymbol{r}_h && \text{(B.7)}\\
&= \boldsymbol{P} \left(\boldsymbol{P}^T \boldsymbol{A}_h \boldsymbol{P}\right)^{-1} \boldsymbol{P}^T \boldsymbol{r}_h. && \text{(B.8)}
\end{aligned}$$

Thus, the column scalings drop out and we recover the regular multigrid cycle. QED.

REFERENCES

[1] Y. Saad, "Iterative methods for linear systems of equations: A brief historical journey," 2019. [Online]. Available: https://arxiv.org/abs/1908.01083

[2] "The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam," *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, vol. 210, no. 459-470, pp. 307–357, Jan. 1911. [Online]. Available: https://doi.org/10.1098/rsta.1911.0009

[3] R. Fedorenko, "A relaxation method for solving elliptic difference equations," *USSR Computational Mathematics and Mathematical Physics*, vol. 1, no. 4, pp. 1092–1096, 1962. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0041555362900319

[4] R. Fedorenko, "The speed of convergence of one iterative process," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 3, pp. 227–235, 1964. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0041555364902538

[5] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977. [Online]. Available: http://www.jstor.org/stable/2006422

[6] A. Brandt, S. McCormick, and J. Ruge, "Algebraic multigrid (amg) for sparse matrix equations," 01 1984.

[7] P. Vaněk, J. Mandel, and M. Brezina, "Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems," *Computing*, vol. 56, no. 3, pp. 179–196, Sep. 1996. [Online]. Available: https://doi.org/10.1007/bf02238511

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[9] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 9 2016. [Online]. Available: https://arxiv.org/abs/1609.02907v4

[10] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *CoRR*, vol. abs/1704.01212, 2017. [Online]. Available: http://arxiv.org/abs/1704.01212

[11] H. Gao and S. Ji, "Graph u-nets," 2019. [Online]. Available: https://arxiv.org/abs/1905.05178

[12] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," *CoRR*, vol. abs/1904.08082, 2019. [Online]. Available: http://arxiv.org/abs/1904.08082

[13] C. W. Oosterlee and R. Wienands, "A genetic search for optimal multigrid components within a fourier analysis setting," *SIAM Journal on Scientific Computing*, vol. 24, pp. 924–944, 2003. [Online]. Available: https://epubs.siam.org/terms-privacy

[14] D. Greenfeld, M. Galun, R. Kimmel, I. Yavneh, and R. Basri, "Learning to optimize multigrid pde solvers," 2 2019. [Online]. Available: http://arxiv.org/abs/1902.10248

[15] I. Luz, M. Galun, H. Maron, R. Basri, and I. Yavneh, "Learning algebraic multigrid using graph neural networks," 2020.

[16] T. Uz-Zaman, S. P. MacLachlan, L. N. Olson, and M. West, "Coarse-grid selection using simulated annealing," *ArXiv*, vol. abs/2105.13280, 2021.

[17] A. Taghibakhshi, S. MacLachlan, L. Olson, and M. West, "Optimization-based algebraic multigrid coarsening using reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[18] N. Bell, "Algebraic multigrid for discrete differential forms," Ph.D. dissertation, University of Illinois Urbana-Champaign, Aug. 2008.

[19] P. Vaněk, "Fast multigrid solver," *Applications of Mathematics*, vol. 40, no. 1, pp. 1–20, 1995. [Online]. Available: https://doi.org/10.21136/am.1995.134274

[20] J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar, "Topology adaptive graph convolutional networks," 2017. [Online]. Available: https://arxiv.org/abs/1710.10370

[21] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," 2017. [Online]. Available: https://arxiv.org/abs/1704.01212

[22] Geuzaine, Christophe and Remacle, Jean-Francois, "Gmsh." [Online]. Available: http://http://gmsh.info/

[23] P. Kumar, C. Rodrigo, F. J. Gaspar, and C. W. Oosterlee, "On local fourier analysis of multigrid methods for pdes with jumping and random coefficients," 2018. [Online]. Available: https://arxiv.org/abs/1803.08864

[24] H. Zheng and J. Wu, "Convergence analysis on multigrid methods for elliptic problems with large jumps in coefficients," *IMA Journal of Numerical Analysis*, vol. 35, no. 4, pp. 1888–1912, 12 2014. [Online]. Available: https://doi.org/10.1093/imanum/dru055

[25] E. Ranjan, S. Sanyal, and P. P. Talukdar, "ASAP: adaptive structure aware pooling for learning hierarchical graph representations," *CoRR*, vol. abs/1911.07979, 2019. [Online]. Available: http://arxiv.org/abs/1911.07979

[26] M. Mitchell, *An introduction to genetic algorithms*, ser. Complex Adaptive Systems. Cambridge, MA: Bradford Books, Mar. 1998.

[27] P. Esfahanian and M. Akhavan, "GACNN: training deep convolutional neural networks with genetic algorithm," *CoRR*, vol. abs/1909.13354, 2019. [Online]. Available: http://arxiv.org/abs/1909.13354

[28] S. Bhatnagar, H. Prasad, and L. Prashanth, *Stochastic Recursive Algorithms for Optimization*. Springer London, 2013. [Online]. Available: https://doi.org/10.1007/978-1-4471-4285-0

[29] L. N. Olson and J. B. Schroder, "Pyamg: Algebraic multigrid solvers in python v4.0," https://github.com/pyamg/pyamg, 2018, release 4.0.

[30] L. N. Olson, J. Schroder, and R. S. Tuminaro, "A new perspective on strength measures in algebraic multigrid," *Numerical Linear Algebra with Applications*, vol. 17, no. 4, pp. 713–733, Nov. 2009. [Online]. Available: https://doi.org/10.1002/nla.669