

EFFECTIVE PRECONDITIONING FOR LOBPCG

JAKE GINESIN, DANIEL YU

ABSTRACT. This paper presents a novel preconditioning approach for the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method to solve large-scale eigenvalue problems. We leverage Schur Complement and Domain Decomposition-based techniques to design an effective preconditioner, integrating concepts from Numerical Linear Algebra and Graph Theory. Our approach is tailored for large, sparse Laplacian matrices derived from graphs, which are commonly encountered in robotic mapping and other computational applications. Experimental results demonstrate that our preconditioner significantly accelerates the convergence of LOBPCG, achieving at least an order of magnitude improvement in effectiveness compared to using no preconditioner. Our findings suggest that this preconditioning strategy is efficient and robust.

CONTENTS

1. Introduction	1
2. Background	2
3. Iterative Eigenvalue Methods	6
3.1. Background on Iterative Methods	6
3.2. Related Methods	6
4. LOBPCG	7
5. LOBPCG Preconditioner Design	9
5.1. Preconditioner Effectiveness	10
5.2. Preconditioner Design Approaches	11
5.3. Maintaining Sparsity	11
6. Our LOBPCG Preconditioner	11
6.1. Domain Decomposition	11
6.2. Schur Complement	13
6.3. Construction	14
7. Experimental Results	15
7.1. Experiment Design	15
7.2. Experimental Setup and Results	16
8. Conclusion	17
9. Acknowledgements	18
References	18

1. INTRODUCTION

Many real-world systems exhibit graphical structure. Therefore, by reasoning about the underlying graph we derive from the our system of interests, we can draw conclusions about the original system. One property of interest when studying graphs is *graph connectivity*; i.e. how easily does our graph divide into smaller subgraphs? Motivated by robotics problems such as pose graph optimization in

SLAM (Simultaneous Location and Mapping) [6], we care about finding the *algebraic connectivity* of a graph using the *Fiedler value*, the second smallest eigenvalue of the Laplacian matrix, which controls the estimation error of SLAM pose graph solutions [3]. The field of study that relates graphs to its eigenvalues (*the spectrum of the graph*) is known as **Spectral Graph Theory**. The graphs we seek to reason about are (1) large, usually consisting of tens to hundreds of thousands of nodes, and (2) sparse, with each node having just a few connections. Given the nature of robotic applications, we would like our solution method to be *provably robust and computationally efficient* in practice, particularly $\leq O(n^2)$. To do so, we seek to leverage iterative methods from **Numerical Linear Algebra**, the study of how matrix operations can be used to create computer algorithms that are accurate (numerically stable) and fast (computationally efficient), to solve this Spectral Graph Theoretic problem and thus solve our robotic mapping problem of interest.

We adopt the Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG) as our eigensolver of choice. Ultimately, we seek to improve on known literature by effectively preconditioning LOBPCG in order to accelerate the rate of convergence of LOBPCG to our eigenvalue of choice. Existing preconditioners for LOBPCG are based on *algebraic* properties of the Laplacian Matrix, but we observe no preconditioner in the literature seeks to leverage the *graphical structure* of the Laplacian to improve efficiency and reduce overhead. To construct a preconditioner that leverages graphical structure, we adopt *Domain Decomposition*, a method commonly employed to solve partial differential equations, and an algebraic technique known as the *Schur Complement*. In practice, we will be performing balanced graph partitioning using the graph partitioning library Metis, which uses heuristic algorithms [7] to compute solutions efficiently in practice.¹

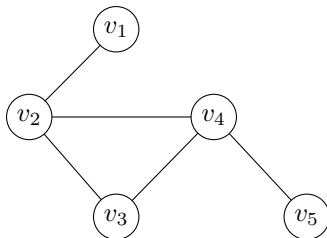
2. BACKGROUND

We begin by providing the preliminary definitions and machinery from graph theory and spectral graph theory necessary to understand LOBPCG.

Definition 2.1. A graph is a pair $G = (V, E)$, where V is a set, whose elements are called *vertices*, and E is a set of unordered pairs of vertices, whose elements are called *edges*.

We will be focusing on undirected graphs, and henceforth all definitions will be for undirected graphs (i.e. graphs where the edges are bidirectional).

Example 2.2. We define an example graph. Let $G = (V, E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5)\}$ A visualization of the defined graph is below.



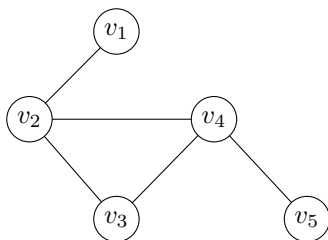
¹Min-cut bipartitioning, k-way partitioning, and balanced graph partitioning are NP-complete. Formally, NP-hard problems can be simulated in polynomial time by a nondeterministic Turing Machine, and a nondeterministic Turing Machine can be simulated in exponential time by a deterministic Turing Machine (i.e. computers in the real world).

Now, we define various matrices associated with graphs. Note that the study of eigenvalues and eigenvectors of matrices associated with graphs is referred to as *spectral graph theory*.²

Definition 2.3. An adjacency matrix A is defined in respect to a graph G and its edges E such that:

$$A_{i,j} = \begin{cases} 1, & \text{if unordered pair}(v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Example 2.4. We construct an example adjacency matrix for the graph in (2.2).



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Definition 2.5. A *sparse matrix* is a matrix in which most of the elements are zero, and the nonzero elements of a matrix form a *sparsity pattern*.

Sparse matrices can be efficiently stored as lists of tuples, with each tuple structured as (X Coordinate, Y Coordinate, value). Reasoning about large and sparse matrices in practice *requires* using a sparse representation; for example, a $10^5 \cdot 10^5$ *dense* matrix at 64-bit floating point precision consumes 80GB of memory. Moreover, graphs with few edges can be represented as a sparse adjacency matrix, which can then be efficiently stored as a list of tuples. In general, we observe certain matrix operations preserve sparsity while some do not. Sparsity is preserved under matrix-vector operations, but not matrix-matrix multiplication, naive matrix factorizations, inverses, and matrix addition & subtraction.

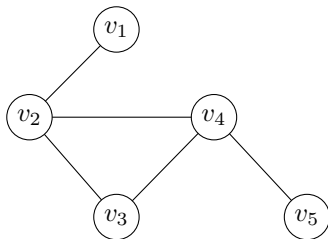
All major libraries in numerical languages such as C, C++, Python, Julia, have support for sparse data representations [12] [5]. Some common formats include Compressed Row Storage (CRS) and Compressed Column Storage (CCS). In these structures [2], we reduce the storage from n^2 to $2q + n + 1$ where q is the number of non-zero entries.

Definition 2.6. Similarly to (2.3), we define the degree matrix D in respect to a graph G :

$$D_{i,j} = \begin{cases} \deg(v_i), & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

Where $\deg(v_i)$ indicates the number of edges connected to v_i .

Example 2.7. We construct an example degree matrix for the graph in (2.2):



$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

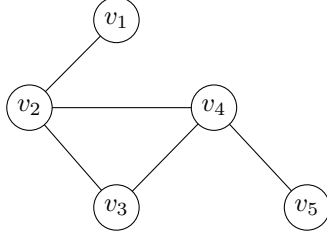
²The word "spectral" in this context is derived from the spectrum of a matrix, i.e. the set of its eigenvalues.

Definition 2.8. The Laplacian matrix L of a graph is defined as

$$L = D - A$$

where D is the degree matrix and A is the adjacency matrix. For this paper, unless otherwise noted, $L(G)$ denotes the Laplacian of G .

Example 2.9. We construct an example Laplacian matrix for the graph in (2.2):



$$L = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & -1 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

Now, we define the critical properties of the Laplacian matrix, which will in turn help us reason about the underlying graph.

Definition 2.10. Let $M_n(\mathbb{R})$ denote a ring of $n \times n$ matrices with entries in \mathbb{R} . A matrix $A \in M_n(\mathbb{R})$ is symmetric if $A = A^T$

Lemma 2.11. If $A \in M_n(\mathbb{R})$ is *symmetric*, then the eigenvalues of A are real.

Proof. A matrix $A \in M_n(\mathbb{R})$ is symmetric if $A = A^T$. Consider the characteristic polynomial of A :

$$p(\lambda) = \det(A - \lambda I)$$

This polynomial has degree n with real coefficients. By the fundamental theorem of algebra, $p(\lambda)$ has n roots in \mathbb{C} . Since $p(\lambda)$ has real coefficients, any non-real roots must occur in complex conjugate pairs. The spectral theorem states that any symmetric matrix A can be diagonalized by an orthogonal matrix Q , such that:

$$Q^T A Q = D$$

where D is a diagonal matrix with the eigenvalues of A on the diagonal. Since Q is orthogonal, the transformation $Q^T A Q$ preserves the real nature of A . Therefore, the diagonal elements of D (the eigenvalues of A) must be real numbers. Therefore, A has real eigenvalues. \square

Lemma 2.12. The Laplacian matrix is always symmetric.

Proof. Recall $L = D - A$, where D is the degree matrix and A is the adjacency matrix. The degree matrix is always symmetric because it is diagonal. The adjacency matrix is symmetric because the presence of an edge between vertices v_1 and v_2 implies the presence of an edge between vertices v_2 and v_1 ; thus, $A_{ij} = A_{ji}$. So, we see:

$$L^T = (D - A)^T = D^T - A^T = D - A = L$$

\square

Definition 2.13 (Positive Semidefinite). A matrix $A \in M_n(\mathbb{R})$ is positive semidefinite (PSD) if for any non-zero vector x ,

$$x^T A x \geq 0.$$

Lemma 2.14. Positive semidefinite matrices have non-negative eigenvalues.

Proof. Let A be a positive semidefinite matrix. By definition, for any vector $x \in \mathbb{R}^n$, $x^T Ax \geq 0$. Suppose λ is an eigenvalue of A with corresponding eigenvector v , so $Av = \lambda v$. Consider:

$$v^T Av = v^T(\lambda v) = \lambda v^T v = \lambda \|v\|^2.$$

Since A is positive semidefinite, we have $v^T Av \geq 0$. Thus, $\lambda \|v\|^2 \geq 0$. As $\|v\|^2 > 0$ (because v is a non-zero eigenvector), it follows that $\lambda \geq 0$. Therefore, all eigenvalues of A are non-negative. \square

Lemma 2.15. The Laplacian matrix L of a graph, defined as $L = D - A$ where D is the diagonal matrix of vertex degrees and A is the adjacency matrix, is positive semidefinite.

Proof. For any vector $x \in \mathbb{R}^n$, consider the quadratic form:

$$x^T Lx = x^T(D - A)x = x^T Dx - x^T Ax.$$

Here, $x^T Dx = \sum_{i=1}^n d_i x_i^2$ since D is diagonal with $D_{ii} = d_i$. Also,

$$x^T Ax = \sum_{i=1}^n \sum_{j=1}^n A_{ij} x_i x_j = \sum_{\{i,j\} \in E} 2x_i x_j,$$

where each edge $\{i, j\}$ is considered twice. Combining these, we have:

$$x^T Lx = \sum_{i=1}^n d_i x_i^2 - \sum_{\{i,j\} \in E} 2x_i x_j = \sum_{\{i,j\} \in E} (x_i - x_j)^2.$$

Each term $(x_i - x_j)^2 \geq 0$. Therefore, $x^T Lx \geq 0$ for all $x \in \mathbb{R}^n$, proving that L is positive semidefinite. \square

Lemma 2.16. The Laplacian matrix L has at least one eigenvalue equal to 0.

Proof. Let $L = D - A$ be the Laplacian matrix of a connected graph, where D is the diagonal matrix of vertex degrees and A is the adjacency matrix. Consider the vector $x = \mathbf{1}$, the all-one vector. We compute Lx :

$$(Lx)_i = \left(\sum_j D_{ij} x_j - \sum_j A_{ij} x_j \right) = d_i - \sum_{j=1}^n A_{ij} = d_i - d_i = 0,$$

where d_i is the degree of vertex i . Hence, $Lx = 0 \cdot x$, which shows that x is an eigenvector of L corresponding to the eigenvalue 0. Since $x \neq 0$, we conclude that 0 is indeed an eigenvalue of L . \square

Definition 2.17 (Fiedler Value). If L is a Laplacian matrix derived from G , then L has n real eigenvalues satisfying

$$0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \dots \leq \lambda_n$$

The **Fiedler Value** is smallest non-zero eigenvalue of L .

Lemma 2.18. A graph G is connected if and only if $\lambda_2 > 0$.

Proof. Let L be the Laplacian matrix of G with eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$. The multiplicity of the eigenvalue 0 is equal to the number of connected components of G .

(\Rightarrow) If G is connected, the number of connected components is 1, so $\lambda_1 = 0$ has multiplicity 1. Therefore, λ_2 must be greater than 0.

(\Leftarrow) Conversely, if $\lambda_2 > 0$, the multiplicity of $\lambda_1 = 0$ is 1, implying that G has exactly one connected component. Hence, G is connected.

Thus, G is connected if and only if $\lambda_2 > 0$. \square

The Fiedler value is critical because it tells us how *connected* the underlying graph is via Cheeger’s inequality [4], and in turn is critical in various practical applications [3].

3. ITERATIVE EIGENVALUE METHODS

In this section we briefly introduce iterative methods for solving the eigenvalue problem and discuss iterative methods related to LOBPCG.

3.1. Background on Iterative Methods. For the rest of the paper we will denote $A = L(G)$, the Laplacian matrix derived from a *large and sparse graph*, G . Likewise, A will be large and sparse; in fact, it is *critical* we preserve the sparsity of A when performing computations. It is infeasible to store A as a dense matrix.

We seek to compute λ_2 by solving the eigenvalue problem:

$$(1) \quad Ax = \lambda x$$

Naively, computing λ_2 of A can be done by computing the character polynomial of A , e.g. $\text{char}(A) = \det(A - \lambda I)$. However, computing a determinant is $O(n!)$ via expansion by minors where $n = \dim(A)$. Alternatively, computing a complete eigendecomposition of the matrix through a method such as the QR method is $O(n^3)$. In fact, the `numpy.linalg.eig` function in the numpy library for python uses a modified QR algorithm for their eigendecomposition. Furthermore, these methods do not preserve the sparsity pattern of a matrix as they require various factorization or matrix-matrix multiplications. This makes these naive methods infeasible ($> O(n^2)$) for computing λ_2 of A .

Thus, we turn to *iterative* methods to compute eigenpairs, where we view matrices as operators on vectors, and only compute matrix-vector products. This is much more computationally feasible than naive methods as our storage cost reduces from $n \times n$ to just n and our complexity reduces from $O(n^2) \rightarrow O(nnz)$ where nnz is the number of non-zero entries.

$$A\vec{x} \in \mathbb{R}^n \ll A \in \mathbb{R}^{n \times n}$$

Or in the case of a block of vectors, where $X \in \mathbb{R}^{n \times k}$, our storage costs reduce from $n \times n$ to $n \times k$ and our complexity reduces from $O(n^2k) \rightarrow O(nnz \cdot k)$.

$$A\vec{X} \in \mathbb{R}^{n \times k} \ll A \in \mathbb{R}^{n \times n}$$

Henceforth, our discussion should solely focus on iterative methods.

3.2. Related Methods. Several iterative eigenpair recovery methods exist. This includes the *power* and *subspace iteration* methods, which excel at finding eigenvalues that are far from other eigenvalues (i.e. *dominant* eigenvalues), and the *Lanczos* method, which excels at computing *extremal* eigenvalues. Notably, the convergence rates of these methods generally depends on the *distribution* of eigenvalues, particularly how separated the eigenvalue of choice is. This is not ideal for finding the Fiedler value, as the Fiedler value is not guaranteed to be well-isolated. Additionally, it is not straightforward to accelerate these iterative methods. Given the sensitivity of the power, subspace iteration, and Lanczos methods to eigenvalue distributions, naively one could try to precondition the aforementioned methods by applying a *spectral transformation* that separates the Fiedler value from the other eigenvalues. However, spectral transformations in general do not take advantage of the inherent graphical structure of Laplacian matrices. Given these limitations, we apply the *locally optimal block preconditioned conjugate gradient* (LOBPCG) method as our iterative method of choice.

4. LOBPCG

In this section we build up to and review the LOBPCG method, which forms the basis of our approach. We highlight some of the features of LOBPCG that make it particularly well-suited for our problem.

Definition 4.1. The symmetric eigenvalue problem³ we wish to solve is defined as:

$$(2) \quad Ax = \lambda x$$

Recall that A will be sparse and PSD with non-negative real eigenvalues.

Definition 4.2 (Rayleigh Quotient). The Rayleigh quotient $R(x)$ for a symmetric matrix A and a non-zero vector x is defined as:

$$R(x) = \frac{x^T Ax}{x^T x}$$

To understand the significance of the Rayleigh quotient, consider its derivation from the symmetric eigenvalue problem.

Definition 4.3 (Rayleigh Quotient Derivation). We derive the Rayleigh quotient from the symmetric eigenvalue problem.

$$(3) \quad Ax = \lambda x \text{ (symmetric eigenvalue problem)}$$

$$(4) \quad x^T Ax = x^T \lambda x \text{ (multiplying both sides by } x^T \text{)}$$

$$(5) \quad x^T Ax = \lambda x^T x \text{ (pulling out the scalar } \lambda \text{ from the inner product)}$$

$$(6) \quad \lambda = \frac{x^T Ax}{x^T x} \text{ (dividing by } x^T x \text{)}$$

$$(7) \quad R(x) = \frac{x^T Ax}{x^T x} \text{ (letting the RHS be in terms of } R(x) \text{)}$$

Which completes our derivation.

We now prove three important properties about the Rayleigh Quotient:

Lemma 4.4. The Rayleigh quotient $R(x)$ provides an estimate of the eigenvalue associated with x .

Proof. Let x be a non-zero vector. The Rayleigh quotient $R(x)$ is defined as:

$$R(x) = \frac{x^T Ax}{x^T x}$$

If x is an eigenvector corresponding to eigenvalue λ , then:

$$Ax = \lambda x$$

Substituting this into the Rayleigh quotient, we get:

$$R(x) = \frac{x^T(\lambda x)}{x^T x} = \lambda \frac{x^T x}{x^T x} = \lambda$$

Thus, when x is an eigenvector, $R(x)$ exactly equals the eigenvalue λ . For other vectors, $R(x)$ provides an approximation to the eigenvalue associated with the direction of x . \square

Lemma 4.5. As x converges to an eigenvector v , $R(x)$ converges to the corresponding eigenvalue λ .

³There is also a generalized eigenvalue problem $Ax = \lambda Bx$ common in Differential equations and Control Theory. Here, we set $B = I$.

Proof. Let v be an eigenvector corresponding to eigenvalue λ , so $Av = \lambda v$. Consider a sequence of vectors $\{x_k\}$ such that x_k converges to v . We need to show that $R(x_k)$ converges to λ .

Since $x_k \rightarrow v$, for large k , x_k can be expressed as $x_k = v + \epsilon_k$ where ϵ_k is a small perturbation vector. The Rayleigh quotient for x_k is:

$$R(x_k) = \frac{x_k^T A x_k}{x_k^T x_k}$$

Substituting $x_k = v + \epsilon_k$ into the Rayleigh quotient:

$$R(x_k) = \frac{(v + \epsilon_k)^T A (v + \epsilon_k)}{(v + \epsilon_k)^T (v + \epsilon_k)}$$

Expanding the numerator:

$$(v + \epsilon_k)^T A (v + \epsilon_k) = v^T A v + \epsilon_k^T A v + v^T A \epsilon_k + \epsilon_k^T A \epsilon_k$$

Since $Av = \lambda v$:

$$= \lambda v^T v + \epsilon_k^T \lambda v + \lambda v^T \epsilon_k + \epsilon_k^T A \epsilon_k$$

Similarly, expanding the denominator:

$$(v + \epsilon_k)^T (v + \epsilon_k) = v^T v + \epsilon_k^T v + v^T \epsilon_k + \epsilon_k^T \epsilon_k$$

As ϵ_k becomes small, higher-order terms $\epsilon_k^T A \epsilon_k$ and $\epsilon_k^T \epsilon_k$ become negligible. Thus:

$$R(x_k) \approx \frac{\lambda v^T v}{v^T v} = \lambda$$

Therefore, as $x_k \rightarrow v$, $R(x_k) \rightarrow \lambda$, as desired. \square

Lemma 4.6. For a symmetric matrix A , the Rayleigh quotient reaches its minimum and maximum at the smallest and largest eigenvalues of A , respectively.

Proof. Let A be a symmetric matrix with eigenvalues $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ and corresponding orthonormal eigenvectors v_1, v_2, \dots, v_n . Any vector x can be expressed as a linear combination of the eigenvectors:

$$x = \sum_{i=1}^n \alpha_i v_i$$

The Rayleigh quotient for x is:

$$R(x) = \frac{x^T A x}{x^T x}$$

Substituting x :

$$x^T A x = \left(\sum_{i=1}^n \alpha_i v_i \right)^T A \left(\sum_{i=1}^n \alpha_i v_i \right) = \sum_{i=1}^n \alpha_i^2 \lambda_i$$

and

$$x^T x = \left(\sum_{i=1}^n \alpha_i v_i \right)^T \left(\sum_{i=1}^n \alpha_i v_i \right) = \sum_{i=1}^n \alpha_i^2$$

Thus, the Rayleigh quotient becomes:

$$R(x) = \frac{\sum_{i=1}^n \alpha_i^2 \lambda_i}{\sum_{i=1}^n \alpha_i^2}$$

The expression $R(x)$ is a weighted average of the eigenvalues λ_i with weights $\alpha_i^2 / \sum_{j=1}^n \alpha_j^2$. The minimum value of $R(x)$ is λ_1 when $x = v_1$ (i.e., all weight is on the smallest eigenvalue), and the maximum value of $R(x)$ is λ_n when $x = v_n$ (i.e., all weight is on the largest eigenvalue).

Hence, the Rayleigh quotient reaches its minimum and maximum at the smallest and largest eigenvalues of A , respectively. \square

Thus, as a corollary to the previous lemma, to find the Fiedler value λ_2 of a large, sparse Laplacian A , we can reformulate the Rayleigh quotient as an optimization problem where the Karush-Kuhn-Tucker (KKT) points, which indicates optimality, are precisely the eigenpairs of the symmetric eigenvalue problem. We then solve for the smallest non-zero eigenvalue.

$$(8) \quad \lambda = \min_{x \in \mathbb{R}^n} \frac{x^T A x}{x^T x}$$

We can take the Lagrangian of $R(x)$, which we denote as $L(x, y)$ as opposed to $L(x, \lambda)$ because λ already denotes eigenvalues. This results in:

$$(9) \quad L(x, y) = \frac{x^T A x}{x^T x} - y(x^T x - 1)$$

And, a straightforward calculation of the gradient of $L(x, y)$ with respect to x is:

$$(10) \quad \nabla_x L(x, y) = 2Ax - 2yx$$

We observe the direction of steepest descent is positively proportional to (9); thus we can define the residual:

$$(11) \quad r := Ax - R(x)x$$

Thus, an iterative method to continually approach a fixed-point can be defined as:

$$(12) \quad x_{i+1} := x_i + Ax_i - R(x_i)x_i$$

Definition 4.7 (LOBPCG). The locally optimal block preconditioned gradient method (LOBPCG) makes various improvements to (11), principal of which involves the inclusion of a chosen *preconditioner* T :

$$(13) \quad x_{i+1} := x_i - T(Ax_i - R(x_i)x_i)$$

Notice that the eigenvalue problem (2) has turned into solving a series of linear systems (13), one at each iteration. This allows one to apply the large body of work done for iterative linear system solves such as GMRES, Jacobi, Gauss-Siedel. Also, notice that at every step we only compute matrix vector products, preserving the sparsity pattern.

One benefit of LOBPCG is that we can expand the algorithm to easily take in block-vectors $X \in \mathbb{R}^{n \times k}$ by replacing x with X in (11)-(13). This allows for the simultaneous recovery of the k eigenvalues. Thus, to maximize descent speed, we seek to choose our preconditioner T such that $T \approx A^{-1}$ and $k(TA) \ll k(A)$, which can be thought of as undistorting the contours of the gradient descent [8].

Recall that computing inverses is not feasible in practice as the computation is $O(n^3)$, and taking inverses also destroys sparsity. Therefore, our research question becomes: how do we efficiently create a preconditioner for LOBPCG, given that A is a Laplacian derived from a graph?

5. LOBPCG PRECONDITIONER DESIGN

Thus, our goal is to efficiently construct T such that $T \approx A^{-1}$, where A is a large and sparse Laplacian matrix. In this section, we define condition numbers, which allows us to measure the effectiveness of an inverse, previous preconditioners used for LOBPCG, and our designed preconditioner.

5.1. Preconditioner Effectiveness.

Definition 5.1 (Condition Number). A linear system of equations $Ax = b$ has error δ from numerical instability. That is,

$$(14) \quad (A + \delta A)(x + \delta x) = (b + \delta b)$$

We observe we can rewrite (14) as

$$(15) \quad \frac{\|\delta x\|}{\|x\|} \leq \|A^{-1}\| \|A\| \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right)$$

That is, we define the instability of x in terms of the instability of A and b . Thus, we define our *condition number* to be precisely the maximum ratio of relative error in x relative to the relative error in b . That is, for any consistent norm we will have:

$$(16) \quad k(A) = \|A^{-1}\| \|A\| \geq \|A^{-1}A\| = 1$$

As a corollary, the condition number is *in general* defined as

$$(17) \quad k(A) = \frac{\lambda_1}{\lambda_n} = \frac{\lambda_{max}}{\lambda_{min}}$$

Where $\lambda_{max}, \lambda_{min} \neq 0$. Thus, we observe that an *effective* preconditioner for A is some T that minimizes the condition number,

$$1 = K(AA^{-1}) < k(TA) \ll k(A)$$

Recall, LOBPCG is a gradient descent-based method; thus, the behavior of convergence of LOBPCG exhibits a level set-based structure. Also recall gradient descent-based methods continually move perpendicularly to the contour lines of the solution space contour plot. That is, the more circular the contour plot of the solution space of LOBPCG is, the faster LOBPCG will approach a solution to the symmetric eigenvalue problem. We observe the *lower* the condition number of the preconditioner, the more circular the contour plot of the solution space for LOBPCG is.

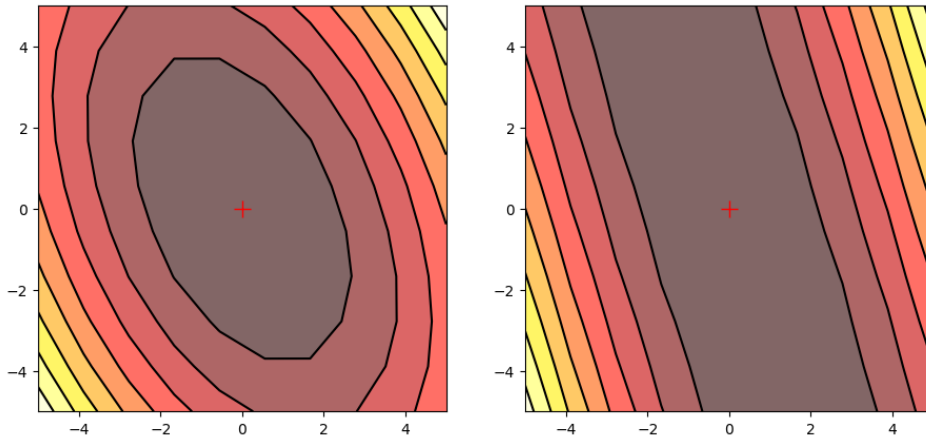


FIGURE 1. The plot on the left shows a LOBPCG solution space contour plot with a low condition number (3.03), while the plot on the right shows a LOBPCG solution space contour plot with a high condition number (58.48). The more circular the contours, the faster the gradient descent (in our case conjugate gradient).

5.2. Preconditioner Design Approaches. Previous work decomposes A into *factors* before taking the inverse, that is:

$$A = Q_1 Q_2 \Rightarrow A^{-1} = Q_2^{-1} Q_1^{-1}$$

Factorizations that are used in practice include the *LU factorization*, where A is decomposed into a lower and upper triangular matrix $A = LU$, and the *Cholesky Decomposition*, which takes advantage of the positive semidefinite A by Lemma 2.20 to decompose $A = LDL^T$ (where L is lower-triangular and D is diagonal). Both approaches are expensive to compute and, notably, do not take advantage of the graphical structure of A .

5.3. Maintaining Sparsity. The biggest problem with decomposition-based preconditioners is that they significantly change the sparsity pattern of the original matrix; in general, taking inverses ruins the sparsity pattern of matrices. Thus, preconditioners *in practice* employ various techniques to reduce density by simply removing dense sections of the resulting matrix. Because our preconditioners only seek to approximate an inverse, maintaining sparsity while losing an exact inverse is a worthwhile tradeoff.

Incomplete LU Factorization. Instead of computing $A = LU$, we compute $A = LU - R$, where $R \in \mathbb{R}^{n \times n}$ a chosen matrix representing the sparsity pattern of LU . Such a preconditioner has been proven effective in preconditioning linear systems [11].

Fill-Reducing Orderings. Instead of computing, $A = LU$ or $A = LDL^T$, we choose a *permutation matrix* P and compute $PAP^T = LU$ or $PAP^T = LDL^T$. A Cholesky Decomposition-based preconditioner with fill-reducing orderings has been shown to produce a preconditioner with a condition number of 1, as shown in [8].

6. OUR LOBPCG PRECONDITIONER

We observe that the preconditioners outlined in the previous section are purely algebraic constructs. They do not take advantage of the graphical structure of the matrix A . We ask: how can we leverage the graphical structure of A to efficiently construct a preconditioner $T \approx A^{-1}$? In this section, we outline our approach to construct a preconditioner T that leverages the graphical structure of A .

6.1. Domain Decomposition. Recall that we seek to reframe our original problem, the canonical example of which is a robotic mapping problem, as a graph optimization problem. Given the geometric nature of the problem, we can expect that our graph possesses a geometric interpretation which can be thought of as easily decomposing into clusters or subgraphs. We would wish to somehow leverage the graph-theoretic methods to partition a graph into subgraphs, solve each subgraph as an independent system, then recombine to solve the original graph and the global system.

It turns out that there is a related concept in Partial Differential Equations known as Domain Decomposition. Domain Decomposition is used to solve a boundary value problem by splitting into smaller boundary value problems on subdomains and coordinating the couplings between subdomains to piece back together a global solution to the original boundary value problem. This can be thought of as *divide and conquering* the original system. In our case, the graph represents a linear system of equations. This is clear when viewing the graph through the Laplacian matrix, $A = L(G)$, where it acts a matrix operator on the initial values of the nodes (variables) to produce some new value for each node.

We will specifically look at balanced graph partitioning. Recall that graph partitioning is NP-hard and that henceforth in practice would be done *heuristically*. Balanced graph partitioning has been shown to be NP-hard by K. Andreev [1].

Definition 6.1. Given a graph $G = (V, E)$, we seek to partition vertex set V into k disjoint subsets V_1, V_2, V_3, \dots , such that:

- (i) the size of each partition V_i is approximately $\frac{|V|}{k}$
- (ii) the number of edges $(u, v) \in E$ is minimized where $u \in V_i$ and $v \in V_j$ for $i \neq j$ (i.e. edges that cross partitions).

Each disjoint vertex set can be thought of as forming a subgraph. This subgraph naturally can be thought of as a subdomain. From the perspective of the subdomains, variables (nodes) from each subdomain (subgraph) should be grouped together. Algebraically, this results in a block structure forming in the matrix.

Remark 6.2. A graph is unique up to relabelling of the nodes. This corresponds to a permutation of the associated adjacency and Laplacian matrix. $PAP^T = A'$ where $A \sim A'$ i.e. A and A' are *similar* and possess the same eigenvalues.

For each subdomain i (represented as subgraph V_i). We want to re-order the variables (nodes) in order of:

- (i) **Local Variables:** variables (nodes) coupled (connected) only with other local variables (i.e. other nodes in the subgraph).
- (ii) **Interface Variables:** variables coupled with both external and local variables.
- (iii) **Boundary Variables:** variables from other subdomains that are coupled with local variables inside the subdomain.

Intuitively, we seek to decouple local (nodes) variables from interface (nodes) variables.

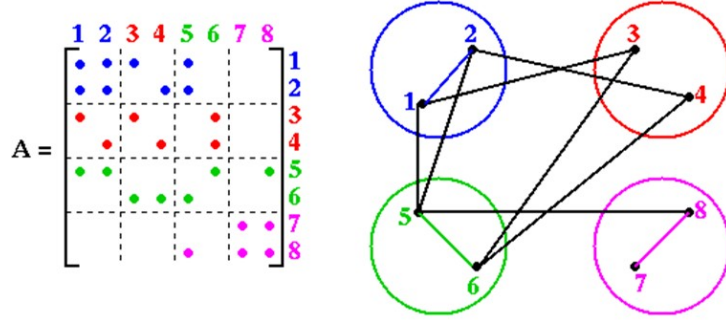


FIGURE 2. Graph Partitioning. Each circle: $(1, 2), (3, 4), (5, 6), (7, 8)$ forms a subdomain, seen in the block structure of the matrix to the left (each dot represents an edge).

Notice that for each subdomain $i \in \{(1, 2), (3, 4), (5, 6), (7, 8)\}$, they are of the form:

$$\begin{pmatrix} B_i & E_i \\ E_i^T & C_i \end{pmatrix}$$

where the block matrices on the main diagonals represent the connections within subdomains, and each block E_i represents the connection between subdomains.

For example, for the matrix above, we achieve this block structure.

$$\begin{pmatrix} B_1 & E_{12} & E_{13} & 0 \\ E_{12}^T & B_2 & E_{23} & 0 \\ E_{13}^T & E_{23}^T & B_3 & E_{34} \\ 0 & 0 & E_{34}^T & B_4 \end{pmatrix}$$

By using graph partitioning with a slight bit of linear algebra we have successfully generated matrices with a certain sparsity pattern and structure.

6.2. Schur Complement. We will use an algebraic technique to leverage this matrix structure.

Consider the system $Ax = b$ of the following form, where A is split into the block matrices, A_s, B, C, D :

$$\begin{pmatrix} A_s & B \\ C & D \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

Initial Equations:

$$\begin{aligned} A_s x + B y &= a \\ C x + D y &= b \end{aligned}$$

Solving for y :

$$(18) \quad y = D^{-1}(b - Cx)$$

We can substitute (18) into $A_s x + B y = a$:

$$(19) \quad A_s x + B(D^{-1}(b - Cx)) = a$$

This results in the reduced system:

$$(20) \quad (A_s - BD^{-1}C)x = a - BD^{-1}C$$

Definition 6.3. Given an $m \times m$ matrix $A = \begin{pmatrix} A_s & B \\ C & D \end{pmatrix}$, the Schur complement in general is defined as: $S = A_s - BD^{-1}C$.

Notice that this computation requires matrix inversion and multiplication which destroys sparsity patterns. Thus, S is generically dense but of much smaller rank. This allows us to write (20) as:

$$(21) \quad Sx = a - BD^{-1}C$$

We can solve (20), (21):

$$(22) \quad x = S^{-1}(a - BD^{-1}C)$$

Finally, we can back-substitute into (18) to solve for y . Computationally this means we only have to solve a much smaller system (21) where $S \ll A$ then another system (22) also much smaller than the original where the sizes of the systems realizes on the block matrix structure. This is exactly what we are looking for in Domain Decomposition.

In our case, A is a real symmetric matrix and using convention, unless otherwise specified, henceforth $A = \begin{pmatrix} B & E^T \\ E & C \end{pmatrix}$

Recall that S is generically dense, so in order to compute S^{-1} quickly, we want to reduce the dimension of S as much as possible through good *partition schemes*. We also observe direct computation of the Schur complement $S = A_s - BD^{-1}C$ is expensive and breaks sparsity. To maintain efficiency and sparsity, we approximate S with a low-rank update: $S \approx S_0 + UV^T$, where S_0 is an initial approximation, and U and V are matrices with a small number of columns representing the most significant changes needed to correct S_0 . ParGeMSLR employs this technique to maintain sparsity [10].

6.3. Construction.

6.3.1. Schur Complement Preconditioner.

Lemma 6.4. The Schur Complement system implicitly constructs an ideal preconditioner when solving $Ax = b$.

Proof. Consider the system $Ax = b$ with the two properties:

- (i) $z = Mx$, a matrix-vector product
- (ii) $Az = x$, another matrix-vector product

They imply: $Ax = b \iff A(Az) = b \iff A(A(Mx)) = b$.

- (i) We want $M \approx A^{-1}$
- (ii) By solving the $Ax = b$ system with Schur Complement **we implicitly** solve the above equation $A(Mx) = x \rightarrow M = A^{-1}$ exactly **without explicitly storing or computing** A^{-1} !

Therefore, LOBPCG acts as a preconditioner in the sense that instead of supplying a whole matrix T such that $k(TA) = 1$, we supply a matrix operator (similar to a function) that allows us to perform operations such that $k(T(A)) = 1$. \square

Remark 6.5. In fact, in order to optimization efficiency, many algebraic preconditioners are commonly stored as matrix operators in practice to improve efficiency. This is because one can work with the block matrices within the factorization so that we can avoid storing and computing the whole matrix.

6.3.2. *Schur Complement Global System.* Consider the global system of linear equations:

$$Ax = b$$

where $A = L(G)$, x represents a vector of the initial values for the nodes, and b is the optimal solution.

We can break down the global system into a global Schur Complement System where u_i represents the local unknowns for the i th subdomain and y_i represents the interface unknowns.

$$(23) \quad \begin{pmatrix} B_1 & & & E_1 \\ & B_2 & & E_2 \\ & & \ddots & \vdots \\ & & & B_p & E_p \\ E_1^T & E_2^T & \dots & E_p^T & C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ g \end{pmatrix}$$

Recall for each subdomain i we can construct a matrix with a block structure like so.

$$\begin{pmatrix} B_i & E_i \\ E_i^T & C_i \end{pmatrix}$$

This naturally corresponds to breaking down the system to:

$$(24) \quad \begin{pmatrix} B_i & E_i \\ E_i^T & C_i \end{pmatrix} \cdot \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}$$

Here N_i corresponds to all the neighboring subdomains to i , so $E_{ij} y_j$ reflects the contribution from other subdomains to this subdomain.

Thus in theory, while we solve the global Schur complement system, what we compute in practice is the local Schur complement system and reconstruct the global solution from those local systems! This greatly simplifies our computations!

6.3.3. Schur Complement Algorithm. Preprocessing

- (i) we preprocess the Laplacian $A = L(G)$ once using the graph partitioning library METIS [7] to partition and permute the laplacian into block matrix format. While partitioning, we operate on the Laplacian matrix as input. Internally, metis constructs a graph, relabels the nodes, and returns the laplacian. It does not permute the matrix through matrix multiplication (more costly).
- (ii) we store the resulting Laplacian in sparse format in blocks E_j, B_j, C_j for each subdomain. Recall that B_i will be diagonal and E_i will be block diagonal matrices.
- (iii) compute C_j^{-1} .
- (iv) compute the Schur Complement matrix $S_j = B_j - E_j^T C_j^{-1} E_j$ for each subdomain.
- (v) we compute the exact Schur Complement inverse S_j^{-1} or we approximate the inverse using *Schur Low Rank Correction*.

LOBPCG Recall that on the update step of LOBPCG, (13), there is a system of equations that needs to be solved at each iteration. The matrix operator T will take in a vector (or block of vectors) and apply the precomputed block matrices in

order to recover x_{i+1} . Let $x_i = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ \vec{y} \end{pmatrix}$ and $x_{i,j} = u_j$ and $y_{i,j} = y_j$ for some x_i at

iteration i :

For all subdomains j :

- (i) at each j , pop E_j, C_j^{-1}
- (ii) Calculate $x_{i+1,j} = S_j^{-1}(x_{i,j} - E_j^T C_j^{-1} E_j x_{i,j})$
- (iii) back substitute for $y_{i+1,j} = C_j^{-1}(y_{i,j} - E_j x_{i+1,j})$

Each subdomain j can be solved in parallel.

6.3.4. Schur Complement Preconditioner Summary. The advantage of a Schur Complement Preconditioner is that by breaking the global system into smaller systems, we reduce the computational strain since we could rely on modern distributed and parallel computing methods on a reduced problem, making such a solution extremely computationally effective. Furthermore, we reduce overhead for the preconditioner construction time (compared to direct factorization) while improving solve speed at each step while maintaining a theoretically ideal preconditioner. We could also leverage the literature on linear system solvers and apply such techniques on the reduced Schur Complement system, $Sx = c$. Another idea is to implement a multilevel Schur Complement, which entails recursively reducing the the system until the overhead construction costs exceed computational savings.

The disadvantages are that given a bad partition scheme, we could end up with block matrices that are still too large for computation, particularly during the matrix inversion and computation steps.

7. EXPERIMENTAL RESULTS

7.1. Experiment Design. After some sleuthing, we found that there was already a C++ library ParGeMSLR (Parallelized Generalized Multilevel Schur Complement

Low Rank Preconditioner) that implemented Schur Complement Domain Decomposition for solving large and sparse symmetric and non-symmetric linear systems [10]⁴.

The library is design to handle large-scale numerical experiments even on a magnitude of order greater than typically seen in robotics applications. For this reason, the ParGeMSLR algorithm introduces machinery for parallelization, Multilevel Schur Complement, and a modified Schur Low Rank Correction [13]. The Parallel GeMSLR relies on ParMETIS (parallelized version of METIS) to decouple the solution of the linear system to a solution associate with interior variables and another associated to interface variables. At each level, the first system can be solved using some form of ILU preconditioning while a Schur Complement decoupling is then recursively applied on the second system hence Multilevel.

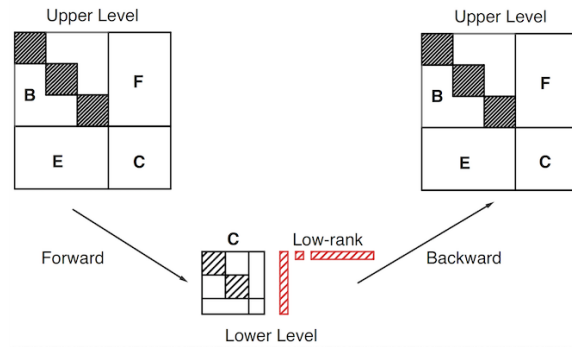


FIGURE 3. 2-level GeMSLR

Due to the approximation of the Schur inverse, the preconditioner loses its theoretical guarantee of an exact solution and thus ideal condition number, but in return computational speed is greatly increased.

In our experiments, ParGeMSLR without an accelerating linear system solver such as Flex-GMRES [9].

The second library we used for our analysis was HYPRE, a canonical C library for numerical linear algebra, which contained implementations for eigensolvers such as LOBPCG as well as algebraic and non-algebraic classes of preconditioners.

Our contribution consisted of integrating the two libraries so that we could pass ParGeMSLR as a preconditioner for LOBPCG. This included exposing C/C++ interfaces and other implementation details.

7.2. Experimental Setup and Results.

7.2.1. Setup.

- Randomly generated sparse Laplacian matrices for a 3d-grid of dimensions $10^3, 32^3, 50^3, 100^3$
- Ran a sample of 10 experiments for each matrix sample size with residual error tolerance at $1 \cdot 10^{-8}$
- Evaluated each experiment with 3 cases:
 - (i) LOBPCG without Preconditioner
 - (ii) LOBPCG with a Euclid, a highly optimized HYPRE pre-built Preconditioner that performs parallelized ILU factorization
 - (iii) LOBPCG with ParGeMSLR

⁴“Generalized” refers to the ability to extend these Schur Complement ideas to non-symmetric indefinite matrices

We decided to use Euclid as a benchmark for our ParGeMSLR preconditioner because it performs ILU factorization which is a canonical preconditioner. Note that we used standard drop pre-built drop tolerances for Euclid and ParGeMSLR.

All experiments were conducted running on a Ubuntu 24.04 LTS x8664 on Intel i7-8650U (8) @ 4.200GHz and took a few hours to complete.

7.2.2. *Results.* The experiment ran is a comparison between LOBPCG iterations and matrix size. Thus, we are disregarding preconditioner construction time as well as solve time.

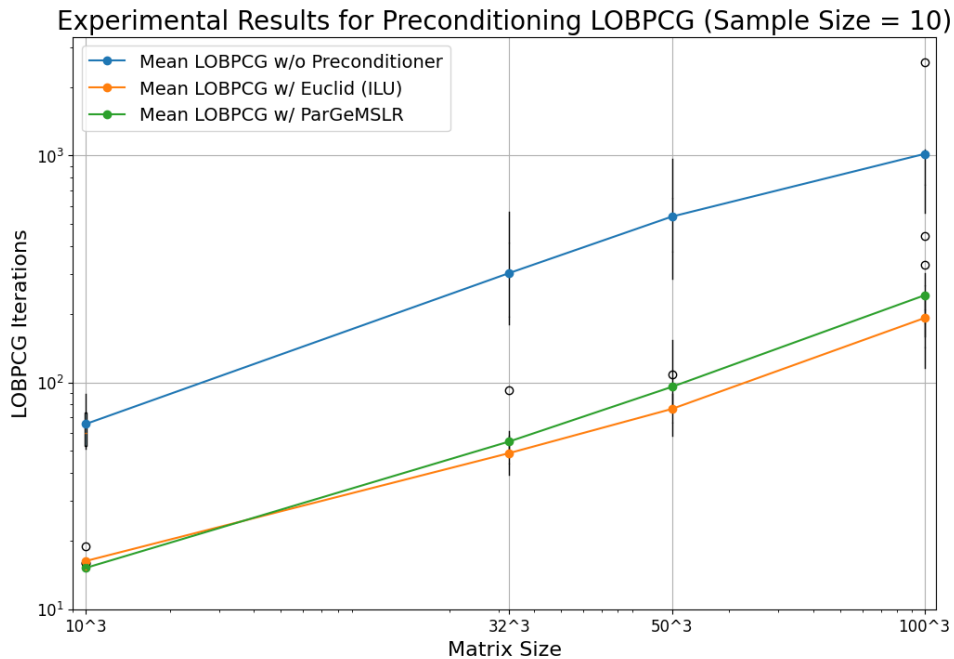


FIGURE 4. Preconditioned LOBPCG Results
LOBPCG with either Euclid and ParGeMSLR as a preconditioner perform an order of magnitude better than without. But Euclid slightly outperforms ParGeMSLR at all sample sizes.

We are not considering runtime in our analysis and purely the strength of the preconditioner.

7.2.3. *Analysis.* The difference in expected results and numerical experiments could be due to the algebraic construction of Euclid and the approximation performed by the SLR in ParGeMSLR at each system and each level which while improves solve speed, increases the "error" of the preconditioning. Additionally, we are not using an accelerator ParGeMSLR which would further increase computational speed whereas Euclid is already highly-optimized with linear system solvers.

One final note is that we are evaluating these experiments on regular grids which have much simpler forms than would appear in real-world robotic mapping problems. We predict that ParGeMSLR would scale better with such problems.

8. CONCLUSION

We have shown the feasibility of Schur Complement based Domain Decomposition methods for preconditioning LOBPCG in the case that the input matrix A is a

Laplacian matrix from a graph. We demonstrate that in general, this preconditioner requires an order of magnitude less iterations than without a preconditioner. As Saad et. al have shown in their work with ParGeMSLR, a Schurs Complement-based preconditioner can be used much more *generally* to solve linear systems beyond those with graphical structure given the algebraic (Domain Decomposition) and graph-theoretic (graph partitioning) duality. Thus, our work naturally extends to preconditioning LOBPCG with Schur Complement based Domain Decomposition methods where A is purely *algebraic*.

Recall, that the nature of the specific robotic mapping problems we are interested in forms a graph that is very large and very sparse with a geometric structure. While we were unable to test our algorithm on these examples, further work will be focused on robustly evaluating Schur Complement-based preconditioners on example pose graphs stemming from SLAM problems incorporating analysis on overhead construction time, solve time, and overall runtime in addition to number of iterations.

9. ACKNOWLEDGEMENTS

We would like to thank the Northeastern University College of Science, Northeastern University Department of Mathematics, and NSF-RTG grant “Algebraic Geometry and Representation Theory at Northeastern University” (DMS-1645877) for funding and supporting this research experience. We would also like to thank our mentors Forrest Miller and Professor David Rosen.

REFERENCES

- [1] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 120–124, New York, NY, USA, 2004. Association for Computing Machinery.
- [2] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, page 233–244, Calgary AB Canada, August 2009. ACM.
- [3] Kevin J. Doherty, David M. Rosen, and John J. Leonard. Spectral measurement sparsification for pose-graph SLAM. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 6412–6419, Kyoto, Japan, October 2022.
- [4] Chris Godsil and Gordon Royle. *Algebraic Graph Theory*. Springer, 2nd edition, 2013.
- [5] Julia Developers. Sparse arrays in julia. <https://docs.julialang.org/en/v1/stdlib/SparseArrays/>, 2024. Accessed: 2024-07-02.
- [6] A. Jurić, F. Kendeš, I. Marković, and I. Petrović. A comparison of graph optimization approaches for pose estimation in slam. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1113–1118, Opatija, Croatia, 2021.
- [7] Karypis Lab. Metis: Family of multilevel partitioning algorithms, 2024. Accessed: 2024-06-30.
- [8] David M. Rosen. Accelerating certifiable estimation with preconditioned eigensolvers, May 2022.
- [9] Yousef Saad et al. ParGeMSLR documentation. <https://www-users.cse.umn.edu/~saad/software/ParGeMSLR/Documentation.pdf>, 2015. Accessed: 2024-07-02.
- [10] Yousef Saad et al. ParGeMSLR: Parallel generalized minimum spanning tree for low rank approximation. <https://www-users.cse.umn.edu/~saad/software/ParGeMSLR/index.html>, 2015. Accessed: 2024-07-02.
- [11] Yousef Saad and Jun Zhang. Bilum: Block versions of multielimination and multilevel ilu preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 20(6):2103–2121, January 1999.
- [12] SciPy Developers. Compressed sparse row matrix (csr). In *SciPy Documentation*, 2024. Accessed: 2024-07-02.
- [13] Tianshi Xu, Vassilis Kalantzis, Ruipeng Li, Yuanzhe Xi, Geoffrey Dillon, and Yousef Saad. pargemslr: A parallel multilevel schur complement low-rank preconditioning and solution package for general sparse matrices. *Parallel Computing*, 113:102956, October 2022.