ABSTRACT

Polynomial Preconditioning with the Minimum Residual Polynomial

Jennifer A. Loe, Ph.D.

Mentor: Ronald B. Morgan, Ph.D.

Krylov subspace methods are often used to solve large, sparse systems of linear equations $Ax = b$. Preconditioning can help accelerate the Krylov iteration and reduce costs for solving the problem. We study a polynomial preconditioner $p(A)$ based upon the minimum residual polynomial from GMRES. The polynomial can improve the eigenvalue distribution of A for better convergence. We demonstrate a power basis method of generating the polynomial and how it can be unstable. Then we develop a new method to generate the polynomial which uses harmonic Ritz values as roots. We discuss sources of roundoff error and give a procedure to make the polynomial stable through adding extra copies of roots. Furthermore, we implement the polynomial into the software Trilinos. We use this implementation to test the polynomial composed with other preconditioners and give considerations for parallel computing.

Polynomial Preconditioning with the Minimum Residual Polynomial

by

Jennifer A. Loe, B.S., M.S.

A Dissertation

Approved by the Department of Mathematics

_____

Dorina Mitrea, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of

Doctor of Philosophy

Approved by the Dissertation Committee

_____

Ronald B. Morgan, Ph.D., Chairperson

_____

Robert C. Kirby, Ph.D.

_____

Johnny Henderson, Ph.D.

_____

Walter Wilcox, Ph.D.

Accepted by the Graduate School

May 2020

_____

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I want to thank Baylor University and the Department of Mathematics for their financial support. Thanks to my advisor Ron Morgan for your encouragement throughout my academic career and for always having an open office door. Thank you also for supporting the many workshops and conferences I attended and supporting me in pursuing research outside of Baylor. Thanks to Rob Kirby for pushing me to broaden my skills in mathematical software and for providing input on research questions. Thanks to Mark Embree for your encouragement and work as a collaborator.

I also want to thank Heidi Thornquist for welcoming me as her intern at Sandia National Laboratories[1] in summer of 2017 and Erik Boman for supporting my return as a visitor two summers later. Their mentorship through my time at Sandia opened my world to parallel computing and software development for mathematical applications. I am excited to continue this work in the future.

Thanks to Jennifer Bryan and Paul Howard at Oklahoma Christian University for supporting me throughout my undergraduate career and encouraging me to attend graduate school.

To Joe and Lori, Elizabeth, Louisa, and Julia: Thank you for supporting me, making me part of your family, and being my source of fun and smiles. Thanks to all the rest of my friends and church family here in Waco who have been a source of strength and joy in my life.

Finally, I want to thank my parents. Thank you for supporting me in my academic and professional endeavors, even when they led me far from home. Thanks for your active presence in my life and for always holding me up in whatever way possible.

[1]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This work was supported in part by the Department of Energy's Exascale Computing Project (ECP).

CHAPTER ONE

Introduction

We are interested in solving a large, sparse linear system $Ax = b$. Such sparse linear systems often arise in large-scale simulations of physical phenomenon. They are created from discretizations of partial differential equations (PDEs) or from other mathematical models. Possible fields of application include circuit simulation, fluid flows, ice sheet modeling, and multiphysics simulations. Real-world applications frequently require many consecutive linear solves; these solves often become a bottleneck in large-scale simulations.

Direct methods exist for solving large, sparse linear systems, but often these methods require too much computation and memory to be cost-effective. The most commonly used solvers are from a class of iterative methods called Krylov subspace methods. This thesis will focus primarily on a Krylov method for nonsymmetric linear systems called the Generalized Minimum Residual Method (GMRES) [54]. In Chapter Two we give details of the GMRES algorithm and analyze criteria for its convergence. Krylov subspace methods iterate to create a subspace from which to extract an approximate solution $\hat{x}$ via a projection. Increasing the accuracy of the solution requires more iterations, which necessitate more calculations (floating-point operations) and computer memory. Applying a preconditioner to the linear system can reduce the number of iterations and expense required to obtain an accurate solution. For right preconditioning, we solve the system $AMy = b$, where $x = My$.

The matrix M is called a preconditioner. In this work, we present a new polynomial preconditioner. Polynomial preconditioners are well-known but are not commonly used. This may be due to stability concerns or because the algorithms to generate the polynomials can be complicated.

As mathematical models become more complex, we need larger computers to provide the memory and computational power required for large-scale simulations. Manufacturers are adapting to this need by creating parallel computers with multiple processors; single-processor designs can no longer provide enough speedup within reasonable bounds of cost and power consumption. Most large-scale simulations are now run on computing clusters, which are built of several computers networked together. Communication, the act of sending information between computer processors or across a computer network, is very time-consuming in comparison to floating-point operations. Polynomial preconditioning can help Krylov solvers to avoid communication in parallel.

Our polynomial preconditioner is based on the minimum residual polynomial from GMRES. This polynomial preconditioner had previously been studied in [34, 3] for linear equations and in [56] for eigenvalue problems. In Chapter Three, we discuss how the polynomial changes the spectrum of A to improve Krylov method convergence. We demonstrate the instability of the power basis method in [34] and suggest a polynomial degree selection strategy when using this method. In Chapter Four we present a new method of generating the minimum residual preconditioning polynomial. This method uses harmonic Ritz values as roots of the polynomial. It is more stable than the power basis method and less expensive than the alternatives presented

in [34]. We also give a method to add roots to the polynomial for extra stability, and we discuss damping to help with problems that have special difficulties.

To consider larger-scale problems, we implement our minimum residual polynomial preconditioner in the software library Trilinos and study its effectiveness for several test cases. In Chapter Five, we discuss trade-offs for composing the polynomial with other preconditioners such as Incomplete LU Factorization (ILU). This combination is likely to occur in real applications. Finally, we demonstrate the preconditioner's performance in parallel and discuss how it could be used in combination with communication-avoiding Krylov methods. Many examples from Chapter Three and the content from Chapter Five were created in collaboration with Heidi Thornquist and Erik Boman from Sandia National Laboratories. Heidi helped with the Trilinos code implementation of the polynomial, and Erik provided guidance for selecting experiments. Further support at Sandia came from Christian Glusa, who helped me set up algebraic multigrid code, and from Ichitaro Yamazaki in our discussions about communication-avoiding Krylov methods. We expect that our polynomial preconditioner will become a useful addition to modern software and that it will be effectively used in real-life applications.

CHAPTER TWO

Preliminaries

We aim to solve the following linear system:

$$Ax = b,\ A \in \mathbb{R}^{n \times n},\ x, b \in \mathbb{R}^n \tag{2.1}$$

Approximate solutions will be denoted as $\hat{x}$. Throughout this dissertation, we assume that the matrix A has real-valued entries. All algorithms can be modified for complex-valued matrices. In some instances, we will make these modifications explicit.

Typically the matrix A will arise from some application. The problem matrix is likely to be large and sparse because it results from a partial differential equation (PDE) discretization (via finite elements or finite differences) or from some other model with local interactions between elements. Due to the large number of variables in real-world problems, one can easily have $n \gg 1{,}000{,}000$.

Traditional direct methods for solving linear systems are typically cost-prohibitive for such large and sparse problems. For example, traditional LU factorization requires $\mathcal{O}(n^3)$ floating-point operations and may destroy the sparsity of the matrix, requiring too much memory. Two classes of solvers are available which overcome these limitations: sparse direct solvers and sparse iterative solvers. This work focuses on a class of sparse iterative solvers called Krylov subspace methods. A Krylov solver iteratively builds a basis for a Krylov subspace and then uses a projection to extract an approximate solution $\hat{x}$ from that subspace.

This chapter is organized as follows: Section 2.1 reviews fundamental linear algebra concepts, including orthogonalization techniques and projection methods. Section 2.2 will introduce the main ideas of Krylov algorithms. In Section 2.2.1 we will present a detailed algorithm for the Generalized Minimum Residual Method (GMRES) [54] and discuss restarting. We will analyze the convergence of GMRES in Section 2.2.2, and Section 2.3 will introduce preconditioning. Then, we will discuss Rayleigh-Ritz projection and harmonic Ritz values in Sections 2.5 and 2.6. Finally, Section 2.7 will explain the cost considerations for running GMRES in serial and parallel.

Most of the linear algebra definitions and foundational algorithms in this chapter can be found in [36]. Our primary reference for Krylov methods and GMRES is [52]. Both are excellent resources for further details on the material in this chapter.

2.1 Foundations

2.1.1 Orthogonalization

Given a set of m linearly independent vectors, one can form them into an orthonormal basis for an m-dimensional subspace $\mathcal{S}$ of $\mathbb{R}^n$ via the Gram-Schmidt procedure. We will use this procedure in GMRES to form a basis for our Krylov subspace. The traditional formulation of this algorithm is called *classical Gram-Schmidt*, while *modified Gram-Schmidt* is known to be more numerically stable. Both versions are detailed in Algorithm 2.1. Implemented as written, they have the same cost. However, for high-performance computing applications, it can be advantageous to use classical Gram-Schmidt because the dot products can be combined into a single block dot product, which needs only one step of communication and synchronization. (See

Section 2.7 for more explanation.) Sometimes even the more stable modified Gram-Schmidt can lose accuracy in floating-point arithmetic. In this instance, it can be helpful to *reorthogonalize* the vectors by make a second pass of Gram-Schmidt.

Algorithm 2.1 Gram-Schmidt orthogonalization procedure

Input: $X = [x_1, x_2, \ldots, x_m]$.

1: **for** $j = 1 : m$ **do**
2: $\quad u_j \leftarrow x_j$
3: $\quad$ **for** $i = 1 : j - 1$ **do**
4: $\quad\quad \alpha_{ij} \leftarrow u_i^T x_j$ (Classical) or $\alpha_{ij} \leftarrow u_i^T u_j$ (Modified)
5: $\quad\quad u_j \leftarrow u_j - \alpha_{ij} u_i$
6: $\quad$ **end for**
7: $\quad \alpha_{jj} \leftarrow \|u_j\|_2$
8: **end for**

One can also use Householder projectors or Givens Rotations to orthogonalize vectors. These two options are more stable than either classical or modified Gram-Schmidt, but they require more floating-point operations.

2.1.2 Projections

Once we have an orthonormal basis for a Krylov subspace, we use a projection to find approximate solutions to 2.1. The following definitions make precise this notion of projection:

Definition 2.1.1. [36, p.385-6, 429] Suppose that $\mathcal{V}, \mathcal{X}$, and $\mathcal{Y}$ are subspaces of $\mathbb{R}^n$ so that $\mathcal{V} = \mathcal{X} \oplus \mathcal{Y}$. Thus, for each $v \in \mathcal{V}$ we have $v = x + y$ for some unique $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. Then x is called the *projection of* v *onto* $\mathcal{X}$ *along* $\mathcal{Y}$. The unique linear operator P defined by $Pv = x$ is called the *projector onto* $\mathcal{X}$ *along* $\mathcal{Y}$. If $\mathcal{Y}$ is

the orthogonal complement of $\mathcal{X}$, then these are called the *orthogonal projection* and *orthogonal projector*, respectively.

We will sometimes make use of a specific orthogonal projection matrix:

Theorem 2.1.1. [36, p. 430] If the columns of V form an orthonormal basis for subspace $\mathcal{S} \subseteq \mathbb{R}^n$, then the unique orthogonal projector onto $\mathcal{S}$ is $P_{\mathcal{S}} = VV^T$.

Using this orthogonal projection ensures that we find the "closest" solution from our subspace in a least-squares sense.

Theorem 2.1.2 (Best Approximation Theorem). [36, p. 435] Let $\mathcal{S}$ be a subspace of $\mathbb{R}^n$ and $x \in \mathbb{R}^n$. The unique vector in $\mathcal{S}$ that is closest to x is $v = P_{\mathcal{S}}x$, the orthogonal projection of x onto $\mathcal{S}$. In other words,

$$\min_{s \in \mathcal{S}} \|x - s\|_2 = \|x - P_{\mathcal{S}}x\|_2.$$

2.1.3 Eigenvalues

We will use eigenvalues to analyze the convergence of our Krylov subspace methods. For a matrix $A \in \mathbb{R}^{n \times n}$, scalars λ and vectors z such that $Az = \lambda z$ are called *eigenvalues* and corresponding *eigenvectors* of A, respectively. Recall that if A is symmetric, it has all real eigenvalues. If A is nonsymmetric, its complex eigenvalues occur in conjugate pairs.

To build out Krylov subspace theory, we may also assume that matrices are diagonalizable:

Definition 2.1.2. If A has a full set of linearly independent eigenvectors $z_1, z_2, \ldots, z_n$ corresponding to eigenvalues $\lambda_1 \ldots \lambda_n$, we can write

$$A = Z^{-1}\Lambda Z$$

where the columns of Z are the z_i and $\Lambda = diag\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$. In this case, we say that A is *diagonalizable*.

The convergence of Krylov subspace methods depends on the distribution of eigenvalues of A. We use the following conventions: A real-valued matrix A is called *Symmetric Positive Definite* (SPD) if it is symmetric and all its eigenvalues are greater than zero. If a matrix A has eigenvalues on both the left and right-hand sides of the complex plane, we say it is *indefinite*. Furthermore, a matrix A is called *normal* if $A^T A = AA^T$. A matrix is normal if and only if it is *unitarily diagonalizable*, that is, if it can be diagonalized with a matrix Z that is orthonormal ($Z^T Z = I$).

The condition number of A indicates the sensitivity of the linear system $Ax = b$ to small perturbations.

Definition 2.1.3. [36, p. 128] The *condition number* of a matrix with respect to the 2-norm is $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

If the condition number is high, then A is called *ill-conditioned*, and small changes in A can cause large changes in the solution x for (2.1). If $\kappa_2(A)$ is small, then A is called *well-conditioned*, and small changes in A are likely to cause a more proportional change in x.

2.2 Krylov Subspace Algorithms

Krylov solvers build an orthonormal basis for a Krylov subspace and use a projection to extract an approximate solution to (2.1) from the subspace.

Definition 2.2.1. Given a matrix A and vector b, the Krylov subspace $\mathcal{K}_m(A, b)$ is defined as

$$\mathcal{K}_m(A, b) = span\{b, Ab, A^2b, \ldots, A^{m-1}b\}.$$

Note that we never form the matrices $A^2, A^3, \ldots, A^{m-1}$ explicitly; this would require too much computation and storage. Krylov solvers are known as *matrix-free* methods because they don't require explicit storage of the matrix A. They only require a function that multiplies A times a vector. (Some methods also require a second function that applies A^T to a vector.)

The primary distinguising factors between Krylov methods include a) whether they work for symmetric or nonsymmetric problems, b) the type of projection used to extract a solution vector, and c) whether they use full orthogonalization or a 3-term recurrence. The Conjugate Gradient method (CG) [22] is standard for symmetric positive definite linear systems. It uses a Galerkin projection to minimize the error in the A-norm, i.e. it finds the approximate solution

$$\hat{x} = \underset{y \in \mathcal{K}_m(A,b)}{\arg\min} \|x - y\|_A.$$

Since CG is always applied to symmetric matrices, it can orthogonalize the Krylov basis using a 3-term recurrence. A 3-term recurrence means that each new Krylov basis vector only needs to be orthogonalized against the previous two basis vectors in order to create an orthonormal basis. In addition, CG only needs to store 3

basis vectors in memory to extract a final solution from the Krylov subspace. Thus, CG does not require restarting in order to limit its storage and orthogonalization expense. The Generalized Minimum Residual Method (GMRES) [54] is able to solve linear systems with nonsymmetric A. It minimizes the residual in the 2-norm (using a MinRes projection), finding

$$\hat{x} = \underset{y \in \mathcal{K}_m(A,b)}{\arg\min} \|b - Ay\|_2.$$

GMRES requires that each new basis vector be fully orthogonalized against previous basis vectors, so it is typically restarted to reduce storage and orthogonalization costs.

Two Krylov methods are very closely related to GMRES. The Full Orthogonalization Method (FOM) [49] is almost identical to GMRES, but it uses a Galerkin projection to obtain a solution rather than a MinRes projection. We can obtain the FOM algorithm by modifying Algorithm 2.2; simply compute $\hat{d}$ in line 13 by solving $H_m\hat{d} = \gamma e_1$. Because A might not be symmetric, FOM is not guaranteed to minimize the error in the A-norm like CG does. The Minimum Residual (MinRes) method [44] is mathematically equivalent to GMRES. It is used when A is symmetric but indefinite, so it can employ a 3-term recurrence like CG to minimize orthogonalization expense. For MinRes, the upper-Hessenberg matrix H_m will be tridiagonal. Like CG, MinRes has no need for restarting, but FOM needs to be restarted for the same reasons as GMRES.

Other Krylov methods for nonsymmetric linear systems find an approximate solution using a Petrov-Galerkin projection that extracts a solution $\hat{x}$ from $\mathcal{K}_m(A, b)$ with the residual vector orthogonal to $\mathcal{K}_m(A^T, b)$. Although these methods do not

minimize the residual over the Krylov subspace, they can still be very effective. By using two subspaces, these methods can employ a different 3-term recurrence to avoid full orthogonalization. Thus, they can build very large subspaces and do not need to be restarted. The Biconjugate Gradient (BCG) [27, 16] and Quasi-Minimum Residual (QMR) [18] methods use a Petrov-Galerkin projection while explicitly building out the subspace $\mathcal{K}_m(A^T, b)$. Other methods use this projection without explicitly forming the subspace $\mathcal{K}_m(A^T, b)$. Then no function is required to apply A^T to a vector. These methods include Biconjugate Gradients Stabilized (BiCGStab) [57], the Induced Dimension Reduction Method (IDR(s)) [55], and the Transpose-Free Quasi Minimum Residual Method (TFQMR) [17].

This thesis focuses on the method GMRES. The polynomial preconditioner presented in subsequent chapters is derived from the GMRES residual polynomial. We study it as a preconditioner for GMRES, but it can also be used as a preconditioner for any of the aforementioned Krylov methods.

2.2.1 The GMRES Algorithm

GMRES is possibly the most widely-used Krylov solver for nonsymmetric systems in today's modern software libraries. We present the full algorithm for GMRES(m) below (Algorithm 2.2) and then discuss all the pieces in detail.

GMRES begins with an initial guess x_0 for the solution vector x. (If no information is present for an initial guess, we choose $x_0 = \vec{0}$.) The recast problem is $A(x - x_0) = b - Ax_0 = r_0$, where r_0 is called the *initial residual vector*. Now the vector $\hat{x}$ that GMRES pulls from $\mathcal{K}_m(A, r_0)$ approximates $(x - x_0)$. GMRES forms an orthonormal

Algorithm 2.2 GMRES(m) (Modified Gram-Schmidt) [52, p. 172]

Input: sparse matrix $A \in \mathbb{R}^{n\times n}$, right-hand side $b \in \mathbb{R}^{n\times 1}$, initial guess $x_0 \in R^{n\times 1}$, relative residual tolerance $rTol$

Output: approximate solution x_m

1: $r_0 = b - Ax_0$,
2: $\gamma = \|r_0\|_2$ and $v_1 = r_0/\gamma$
3: **for** $j = 1 : m$ **do**
4: $\quad w_j = Av_j$
5: $\quad$ **for** $i = 1 : j$ **do**
6: $\quad\quad h_{ij} = v_i^T w_j$
7: $\quad\quad w_j = w_j - h_{ij} v_i$
8: $\quad$ **end for**
9: $\quad h_{j+1,j} = \|w_j\|_2$. (Lucky breakdown if $h_{j+1,j} = 0$.)
10: $\quad v_{j+1} = w_j / h_{j+1,j}$
11: **end for**
12: Define the $(m+1) \times m$ upper-Hessenberg matrix $\overline{H}_m = \{h_{ij}\}_{1\le i\le m+1, 1\le j\le m}$
13: Compute $\hat{d} = \arg\min_{y\in\mathbb{R}^m} \left\|\gamma e_1 - \overline{H}_m y\right\|_2$, $\hat{x} = V_m \hat{d}$, and $x_m = x_0 + \hat{x}$.
14: Compute $r_m = b - Ax_m$. If $\|r_m\|_2 / \|r_0\|_2 \le rTol$, stop. Else, set $x_0 = x_m$, $r_0 = r_m$ and go to Step 2.

basis for the Krylov subspace $\mathcal{K}_m(A, r_0)$, beginning with the vector $v_1 = r_0/\|r_0\|$. To form the next basis vector v_j, the vector Av_{j-1} is orthogonalized against the previous vectors $\{v_i\}_{i=1}^{j-1}$ and then normalized. Algorithm 2.2 gives GMRES with modified Gram-Schmidt orthogonalization, but this can be replaced by any of the orthogonalization algorithms discussed in Section 2.1.1.

Lines 2 through 12 of Algorithm 2.2 comprise the *Arnoldi Iteration*. This process creates an orthonormal matrix $V_{m+1} = [v_1, v_2, \ldots, v_{m+1}]$ of size $n \times (m+1)$ where the columns of V span the Krylov subspace $\mathcal{K}_m(A, r_0)$. (Since V is orthonormal, we have that $V_{m+1}^T V_{m+1} = I$.) The matrix H stores the coefficients from orthogonalization. It is *upper-Hessenberg*, that is, it has all zero entries below the subdiagonal. Its entries are $h_{i,j} = v_i^T A v_j$. Thus, we obtain an important relation called the *Arnoldi*

Recurrence:

$$AV_m = V_{m+1}\overline{H}_m. \tag{2.2}$$

In this notation, V_m consists of the first m columns of V_{m+1}, and $\overline{H}_m$ has size $(m+1) \times m$. Alternatively, we can write the Arnoldi recurrence as

$$AV_m = V_m H_m + \beta v_{m+1} e_m^T \tag{2.3}$$

where $\beta = h_{m+1,m}$ and H_m is the leading $m \times m$ submatrix of $\overline{H}_m$. (Let e_m denote the standard basis vector $[0, \ldots, 0, 1]^T$ of length m.) Furthermore, we have that $V_m^T A V_m = H_m$ and $V_{m+1}^T A V_m = \overline{H}_m$. These recurrence formulas are foundational for proving many properties of GMRES.

After building V and H, we use a projection to obtain the approximate solution $\hat{x}$ which minimizes the residual over the Krylov subspace. Recall that $v_1 = r_0/\gamma$ where $\gamma = \|r_0\|_2$. So we can write $r_0 = \gamma V_{m+1} e_1$. Also, since $\hat{x} \in \mathcal{K}_m(A, r_0)$, there exists some $\hat{d} \in \mathbb{R}^m$ such that $\hat{x} = V_m \hat{d}$. Then using (2.2), we can rewrite the residual norm as

$$\|r_0 - A\hat{x}\|_2 = \left\|\gamma V_{m+1} e_1 - A V_m \hat{d}\right\|_2 \tag{2.4}$$

$$= \left\|\gamma V_{m+1} e_1 - V_{m+1} \overline{H}_m \hat{d}\right\|_2 \tag{2.5}$$

$$= \left\|\gamma e_1 - \overline{H}_m \hat{d}\right\|_2 \tag{2.6}$$

The last equality holds because factoring out the orthonormal matrix V_{m+1} does not change the 2-norm. Thus, solving the over-determined least-squares problem $\overline{H}_m \hat{d} = \gamma e_1$ for $\hat{d}$ (line 13) gives the needed information to minimize the residual. Then our approximate solution for x is $x_m = x_0 + V_m \hat{d}$. Since GMRES minimizes

the residual over a subspace of increasing dimension, the residual norm is always non-increasing in exact arithmetic.

When a chosen residual norm decreases below the *residual tolerance* $rTol$ requested by the user, we say that the solver has *converged*. Most implementations of GMRES determine convergence based on the *relative residual norm* $\|r_0 - A\hat{x}\|_2/\|r_0\|_2$. (Note that if $\|r_0\|_2 = 1$, then the true and relative residual norms are equal.) Observe that the *short residual* norm of the small least squares problem in (2.6) is equivalent to the *true residual* norm $\|r_0 - A\hat{x}\|_2$. Sometimes we will monitor the short relative residual for convergence rather than the true relative residual, as the two residuals can diverge with roundoff error.

Each step through the for loop on lines 3 through 11 in Algorithm 2.2 is called an *iteration* of GMRES. One could, of course, compute an approximate solution and calculate residual norm convergence at the end of each iteration rather than waiting until the end of m iterations. In exact arithmetic, full GMRES is guaranteed to converge in at most n iterations. (A Krylov subspace of dimension n is equivalent to $\mathbb{R}^n$, and the true solution will be extracted to minimize the residual.) If ever we have $h_{m+1,m} = 0$, the approximate solution vector to be computed corresponds to the exact solution x. This is called *lucky breakdown*.

In many circumstances the orthogonalization costs and storage requirements of running full GMRES become prohibitive. In order to use an m-dimensional Krylov subspace, GMRES requires the storage of m basis vectors in memory. Additionally, the number of flops for full orthogonalization increases as we add more basis vectors. To remedy this problem, we truncate GMRES after m iterations and build a new

subspace. This algorithm is called *restarted GMRES*, or *GMRES*(m). Although restarting GMRES limits the memory and computational requirements per iteration, we are essentially "throwing out" all the information in the subspace. This slows convergence of GMRES, and it is no longer guaranteed to converge in n iterations. Some methods, such as GMRES-DR [39] work to overcome this limitation by retaining approximate eigenvectors in the Krylov subspace at the restart.

2.2.2 Convergence of GMRES

We can use polynomials to analyze the convergence of GMRES. In this section, we assume $x_0 = \vec{0}$, and thus $r_0 = b$. Assume also that A is diagonalizable. (If A is not diagonalizable, much of the following analysis still applies using a basis for $\mathbb{R}^n$ that consists of eigenvectors of A and other vectors.)

GMRES selects an approximate solution $\hat{x} \in \mathcal{K}_m(A, b)$. So we can write

$$\hat{x} = c_1 b + c_2 Ab + \cdots + c_m A^{m-1} b \tag{2.7}$$

for some constants $c_1, c_2, \ldots, c_m$. Use these constants to define the polynomial:

$$p(\alpha) = c_1 + c_2\alpha + c_3\alpha^2 + \cdots + c_m\alpha^{m-1} \tag{2.8}$$

where α is the independent variable for the polynomial. Then

$$\hat{x} = p(A)b. \tag{2.9}$$

Since the eigenvectors $z_1, \ldots, z_n$ of A form a basis for $\mathbb{R}^n$, we can rewrite the vector b as

$$b = \sum_{i=1}^{n} \beta_i z_i \tag{2.10}$$

for some scalars $\{\beta_i\}_{i=1}^n$. Thus

$$r = b - A\hat{x} = b - Ap(A)b = (I - Ap(A))b. \tag{2.11}$$

Define a new polynomial:

$$q(\alpha) \equiv 1 - \alpha p(\alpha). \tag{2.12}$$

Observe that $q(0) = 1$. Then if λ_i is the eigenvalue corresponding to z_i,

$$r = q(A)b = q(A)\sum_{i=1}^{n} \beta_i z_i = \sum_{i=1}^{n} \beta_i q(A) z_i = \sum_{i=1}^{n} \beta_i q(\lambda_i) z_i. \tag{2.13}$$

Since the value of each β_i in the above decomposition is fixed, GMRES needs the value of q to be very small at each eigenvalue λ_i in order to achieve the minimum residual.

This analysis highlights another disadvantage of restarting GMRES: A subspace of dimension m can only yield a polynomial q of degree m. Since polynomials are smooth, they resist making sharp turns. Thus, low-degree GMRES polynomials can limit convergence; it is difficult for these polynomials to give $q(0) = 1$ while also making sufficient corrections to be small at the eigenvalues near the origin. This is one reason that GMRES has difficulty with problems that have lots of small eigenvalues and with indefinite spectra. A low-degree polynomial may also have trouble with a problem whose eigenvalue span a wide range; polynomials need to make lots of turns to be small over a large interval. Thus, GMRES works best when the smaller eigenvalues of A are well-separated from the origin and the larger eigenvalues are clustered together.

Let $\mathcal{P}_d$ denote the set of all the polynomials of degree d or less. GMRES picks $\hat{x}$ corresponding to the polynomial that minimizes the residual norm over all polynomi-

als in $\mathcal{P}_m$, that is,

$$\|r\|_2 = \|q(A)b\|_2 = \min_{\substack{\pi \in P_m \\ \pi(0)=1}} \|\pi(A)b\|_2.$$

More specifically, we can say that the relative residual norm improves according to the maximum value of the polynomial q over the spectrum of A:

Theorem 2.2.1. [52, p. 216] Let A be diagonalizable as in Defn. 2.1.2. Then the relative residual norm for GMRES after m iterations can be bounded by

$$\frac{\|r\|_2}{\|r_0\|_2} \leq \kappa_2(Z) \min_{\substack{\pi \in \mathcal{P}_m \\ \pi(0)=1}} \max_{i=1,\ldots,n} |\pi(\lambda_i)|. \tag{2.14}$$

Note that if A is normal, then $\kappa_2(Z) = 1$.

Chebyshev polynomials are often used to analyze convergence of GMRES. A Chebyshev polynomial $C(\alpha)$ can be shifted and scaled so that $C(0) = 1$ and the polynomial has equal oscillations over a given interval $[a, b]$. Such polynomials are optimal in the sense that $C(\alpha)$ minimizes the value $max_{x \in [a,b]} C(x)$ over all polynomials in $\mathcal{P}_m$ such that $\pi(0) = 1$. Like the minimum residual polynomials from GMRES, they can be good preconditioners because of this property.

2.3 Preconditioning

Another measure which can indicate the potential effectiveness of a Krylov solver is the condition number $\kappa_2(A)$. If A is a normal matrix, $\kappa_2(A) = |\lambda_n|/|\lambda_1|$ where λ_n and λ_1 are, respectively, the largest and smallest eigenvalues of A in terms of absolute value. This implies that a normal matrix A with both very large and very small eigenvalues will be very ill-conditioned. This often holds true for non-normal matrices as well. An ill-conditioned matrix can be challenging for GMRES both for

the difficult eigenvalue distribution and for the potentially disastrous effects of small perturbations from round-off error.

Instead of directly solving the linear system $Ax = b$, sometimes we can improve the system using *preconditioning*. We can solve a *left-preconditioned* system

$$MAx = Mb \tag{2.15}$$

or a *right-preconditioned* system

$$AMy = b \text{ with } x = My \tag{2.16}$$

which is equivalent to

$$AMM^{-1}x = b. \tag{2.17}$$

The matrix M is called a *preconditioner* and is typically chosen so that M approximates A^{-1} in some way. If M is an effective preconditioner, the preconditioned system will have a lower condition number and a more favorable spectrum for GMRES than the original system. Typically left and right preconditioning produce comparable results, but right preconditioning has the advantage that the residual norm can be monitored at no additional cost. With right preconditioning, the residual norm of the preconditioned system $\|b - AMM^{-1}\hat{x}\|_2$ is equal to the residual norm of the original system, but the residual of the left-preconditioned system $\|Mb - MA\hat{x}\|_2$ may not be. With left preconditioning, it is possible for the preconditioned linear system to converge while the original system does not, so the residual $\|b - A\hat{x}\|_2$ must be computed explicitly. For the remainder of this work, we exclusively use right preconditioning. All concepts presented in the context of right preconditioning can be extended to work with left preconditioning.

Often preconditioners are designed for a specific application using knowledge of the physical problem or matrix structure, but some preconditioners are general-purpose, generated using only the coefficient matrix A. Generally speaking, we do not form the preconditioner M explicitly. Often M is the inverse of some matrix with nice structure, so it is applied using a direct linear solve. When choosing a preconditioner, one must weigh the cost of computing and applying the preconditioner vs the improvement in Krylov solver convergence. Some preconditioners are very expensive to compute and apply. We briefly discuss some standard preconditioners:

2.3.1 ILU

One popular general-purpose preconditioner is called *Incomplete LU Factorization* or *ILU*(k). For ILU, we form a partial factorization so that $A \approx LU$, where L is lower triangular and U is upper triangular. Then we let $M = (LU)^{-1}$, but we do not compute $(LU)^{-1}$ explicitly. Rather, we use forward and backward solves to multiply $(LU)^{-1}$ by any vector as needed. The parameter k determines the "fill-in", or the sparsity of the LU factors. With ILU(0), the nonzero pattern of LU matches the nonzero pattern of A. For higher values of k, additional non-zero elements are included based on dependencies from the previous level of fill-in. Sometimes an ILU factorization cannot be computed for a matrix that is very near to being singular.

In a parallel setting, we typically do not compute an ILU factorization for the entire matrix A. Rather, we use *domain decomposition* to split the matrix into subdomains, compute an ILU factorization for each subdoman, and then combine

the overlapping parts with an Additive Schwarz method. Sometimes a parameter is available to adjust the amount of overlap between subdomains.

2.3.2 Jacobi and Block Jacobi

Often Jacobi preconditioning refers to *diagonal scaling*. (Let $M = D^{-1}$ where D is the diagonal of A.) Block Jacobi is essentially a block-diagonal variation of this. For our purposes, *block Jacobi* means that the input matrix A is decomposed into blocks and a direct solve is performed to create an LU factorization of each block. Then each block is inverted every time the preconditioner M is applied.

2.3.3 Multigrid Preconditioning

Geometric multigrid preconditioning is based upon the mesh used to discretize a PDE. It performs solves over several subsets of the original mesh; some grids are coarser and others are finer. The multigrid solver moves back and forth between the grids, applying a smoothing operation, such as a weighted Jacobi, Gauss-Seidel, or Chebyshev iteration. Solutions on the coarse grid help to obtain a solution for the fine grids and vice-versa. Algebraic multigrid (AMG) emulates the methods of geometric multigrid without any information about the underlying mesh. It can be used with only the coefficient matrix A.

2.4 Flexible GMRES

Flexible GMRES [51] is a modification of GMRES that allows one to vary the preconditioner at each step. FGMRES has no additional arithmetic over GMRES, but it needs twice the memory. RGMRES [58] is another version of GMRES that

also allows for variable preconditioners. These methods allow us, for example, to use GMRES as a preconditioner for itself. This technique has interesting parallels to our polynomial preconditioning.

2.5 Rayleigh-Ritz Procedure

The *Rayleigh-Ritz* procedure (Algorithm 2.3) projects the matrix A onto $\mathcal{K}_m(A, b)$ and then finds the eigenvalues and eigenvectors of the projected matrix.

Algorithm 2.3 Rayleigh Ritz projection

1: Compute $V_m = [v_1, v_2, \ldots, v_m]$, an orthonormal matrix whose columns span $\mathcal{K}_m(A, b)$.
2: Let $H_m = V_m^T A V_m$.
3: Solve the eigenvalue equation $H_m y_i = \theta_i g_i$.
4: The θ_i are approximate eigenvalues of A corresponding to approximate eigenvectors $y_i = V_m g_i$.

The matrix H_m has small size $m \times m$, so unlike the large problem with A, its eigenvalues and eigenvectors can be computed using standard methods such as QR-iteration. The θ_i are called *Ritz values* and the y_i are called *Ritz vectors*. The Ritz values approximate eigenvalues on the exterior of the spectrum of A.

The following steps give some intuition for how the projection process works:

1. We want to approximate solutions of $Ax = \lambda x$.
2. Use the orthogonal projector $V_m V_m^T$ (Thm 2.1.1) to project A onto $\mathcal{K}_m(A, b)$. Then we have $V_m V_m^T A x = \lambda x$.
3. Now x is in the image of V_m, so $x = V_m g$ for some $g \in \mathbb{R}^m$. So $V_m V_m^T A V_m g = \lambda V_m g$.
4. Multiplying on both sides by V_m^T gives $V_m^T A V_m g = \lambda g$.

Observe also that

$$\begin{aligned} V_m^T(Ay_i - \theta_i y_i) &= V_m^T A y_i - V_m^T \theta_i y_i \\ &= V_m^T A V_m g_i - \theta_i V_m^T V_m g_i \\ &= V_m^T A V_m g_i - \theta_i g_i \\ &= 0. \end{aligned}$$

Thus, the residual norm $Ay_i - \theta_i y_i$ is orthogonal to $\mathcal{K}_m(A, b)$. For more details and convergence theory for Ritz values, see [53, Ch. 4] and [45]. If V_m is not orthonormal, then we can instead use generalized Rayleigh-Ritz by solving

$$V_m^T A V_m g = \theta V_m^T V_m g. \tag{2.18}$$

2.6 Harmonic Ritz Values

Harmonic Rayleigh-Ritz [38] is a related method for extracting eigenvector approximations based upon a shift-and-invert approach. Observe that A has the same eigenvectors as $(A - \sigma I)^{-1}$ and that the eigenvalues are related:

$$\begin{aligned} Ax &= \lambda x \\ \implies (A - \sigma I)x &= (\lambda - \sigma)x \\ \implies \frac{1}{\lambda - \sigma} x &= (A - \sigma I)^{-1} x \end{aligned}$$

We can use generalized Rayleigh-Ritz (2.18) to approximate these eigenvalues of the shifted and inverted matrix:

$$V_m^T (A - \sigma I)^{-1} V_m g = \frac{1}{\theta - \sigma} V_m^T V_m g.$$

But we do not want to solve a problem that requires inverting a large matrix. So we replace V_m with $(A-\sigma I)V_m$. This gives the following:

$$V_m^T(A-\sigma I)^T V_m g = \frac{1}{\theta-\sigma} V_m^T(A-\sigma I)^T(A-\sigma I)V_m g$$

When we choose the shift to be $\sigma = 0$ we have:

$$V_m^T A^T V_m \tilde{g} = \frac{1}{\tilde{\theta}} V_m^T A^T A V_m \tilde{g} \tag{2.19}$$

So $(\theta, V_m g)$ is an approximate eigenpair for A. We call θ and $y = V_m g$ a *harmonic Ritz value* and *harmonic Ritz vector*, respectively. For more theory about why harmonic Ritz values give good eigenvalue approximations, see [40].

The Arnoldi Relation yields a shortcut for finding Harmonic Ritz Values [20, 43]. (We give this result in its generalized form for $A \in \mathbb{C}^{n\times n}$.)

Theorem 2.6.1. After m steps of the Arnoldi Iteration, we can compute harmonic Ritz values (with $\sigma = 0$) using the formula

$$(H_m + \beta^2 f e_m^*)g = \theta g \tag{2.20}$$

where $\beta = h_{m+1,m}$ and $f = H_m^{-*} e_j$.

Proof. Taking the conjugate transpose of both sides of (2.3), we get

$$V_m^* A^* = H_m^* V_m^* + \overline{\beta} e_m v_{m+1}^*. \tag{2.21}$$

Substituting (2.21) into (2.19) we get

$$V_m^* A^* V_m g = \frac{1}{\theta} V_m^* A^* A V_m g$$

$$\implies (H_m^* V_m^* + \overline{\beta} e_m v_{m+1}^*) V g = \frac{1}{\theta}(H_m^* V_m^* + \overline{\beta} e_m v_{m+1}^*)(V_m H_m + \beta v_{m+1} e_m^*) g$$

$$\implies H_m^* g = \frac{1}{\theta}(H_m^* H_m + \overline{\beta}\beta e_m v_{m+1}^* v_{m+1} e_m^*) g$$

$$\implies \theta g = (H_m + \beta^2 H_m^{-*} e_m e_m^*) g$$

The third equality follows from the fact that $V_m^* V_m = I$ and that v_{m+1} is orthogonal to the columns of V_m. The final step follows since $v_{m+1}^* v_{m+1} = 1$ and β is real-valued (since $h_{m+1,m} = \|w\|_2$). □

Harmonic Ritz values have one more important quality that we will exploit for polynomial preconditioning: When $\sigma = 0$, they are roots of the GMRES polynomial $q(\alpha) = 1 - p(\alpha)$ defined in (2.12). (See [43] for explanation. In [20] there is a proof of this based on the method in [60].)

2.7 Cost of GMRES in Serial and Parallel

Krylov solvers have two main sources of computational time: arithmetic and communication. Arithmetic refers to floating-point operations (flops), which include adds, multiplies, and divisions. *Communication* refers to data movement. Data can be moved between different levels of memory hierarchy (e.g. disk, DRAM, and cache), or it can be moved between different processors, as with many CPUs in a parallel computation, or from a CPU to a GPU. We will study polynomial preconditioned GMRES using both serial and parallel implementations. In both cases, we ignore the expenses of data movement between levels of memory. Thus, from this point forward, we use the term "communication" to refer to communication between processors.

To monitor the arithmetic cost of GMRES, we will tally the number of dot products, sparse matrix-vector products (SpMVs), and AXPYs performed on long vectors of length n. (An AXPY, or "vector update" has the form $v = \alpha x + y$ where v, x, y are vectors and α is a scalar.) Counting these three operations provides a rough idea of the main sources of floating-point operations (flops) for the algorithm. Length-n

matrix and vector operations dominate the algorithm; the cost of calculations with smaller matrices is typically negligible.

While arithmetic operation counts are informative, they don't give a complete picture of the expenses for modern-day computers. In the last few decades, CPU design reached the "power wall" [4]. Increasing CPU clock speed from today's norms is not sustainable because it requires so much extra power and cooling. Thus, computer manufacturers have started improving computer performance (in terms of flops/second) by squeezing in multiple processors rather than increasing the speed of a single processor. Nearly all computers today have multiple processors (cores), so algorithms need to be designed to make efficient use of this parallelism. More algorithms are beginning to incorporate GPUs (Graphics Processing Units), which are designed to efficiently execute the same instructions in parallel on multiple pieces of data. Our parallel implementations use MPI parallelism with multiple CPUs. MPI stands for *"Message Passing Interface."* MPI parallelism is a *distributed memory* model of parallel computing, meaning that each compute processor has its own memory allocation that cannot be directly accessed by any of the other processors. Thus, communication between processes becomes essential. In parallel computing, flops are less significant and communication cost begins to dominate computation time. *Latency* is, roughly speaking, the time it takes to send a piece of data from one location to another. For new computers, the maximum number of flops/second continues to grow exponentially faster than any improvements in latency (see [23] and references therein).

For working with large sparse linear systems in parallel, the large matrix and vectors are split up among the memory assigned to different processors. The most straightforward way to distribute a length-n vector or matrix among p processes is to assign the first n/p rows to Process 1 (P1), the next n/p rows to Process 2 (P2) and so on. Each process maintains its own copy of small matrices and vectors that are central to the algorithm. Different types of arithmetic operations require different levels of communication. AXPYs require no communication; each process can perform the addition locally because it stores the elements for corresponding rows of both vectors. A sparse matrix-vector product (SpMV) Ax may require local communication between neighboring processes. Since the matrix A is sparse, a process only needs the elements of x corresponding to nonzero elements in its stored rows of A in order to complete the SpMV. Figure 2.1 demonstrates a possible communication pattern for an SpMV with a simple matrix in parallel. *Load balancing* techniques can be used to equilibrate the amount of work assigned to each CPU and minimize communication time for sparse matrix-vector products. These methods use graph partitioning and other algorithms to distribute a sparse matrix among processors in the most optimal way for a SpMV.

Dot products are more costly for parallel computing than SpMVs. They require *global communication*: each process has to send data which is then aggregated and communicated to all other processes. A diagram showing a possible dot product communication pattern is in Figure 2.2. First, each processor computes its portion of the dot product locally. Then all those values are accumulated in a process called *reduction*. Local answers are communicated through a tree-like hierarchy to other processes

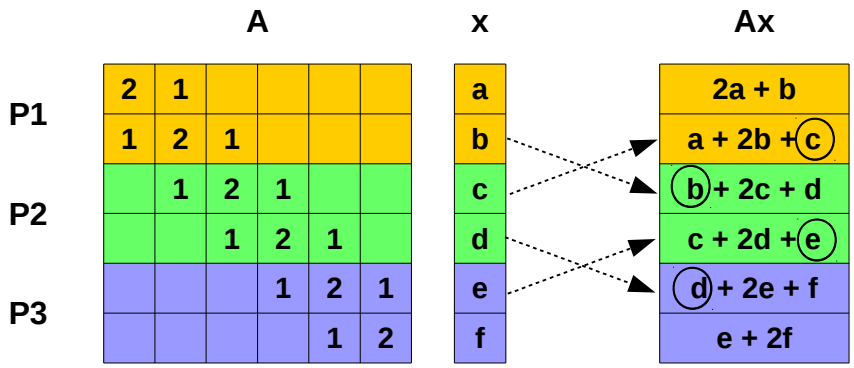


Figure 2.1: The 6×6 matrix A is distributed among 3 processes and multiplied against a vector x. Dotted lines show local communication between neighboring processes. The circled elements in the final vector Ax required communication from other processes.

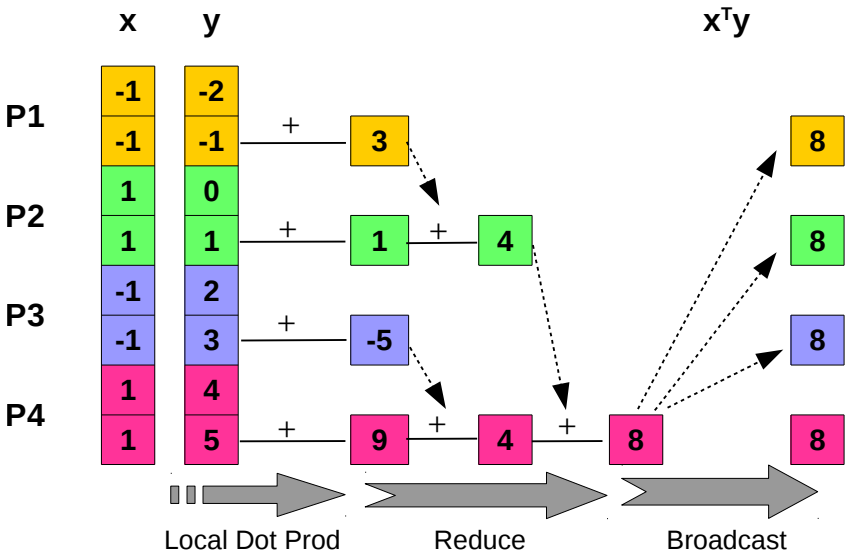


Figure 2.2: Two vectors x and y are distributed among 4 processes. Each process computes a local dot product. Solid lines indicate arithmetic. Dotted lines indicate communication. These values are summed through a reduction that requires global communication. Then the final answer is broadcast to all processes.

and added. Finally, the total for the dot product ends up on one processor, which then has to *broadcast* this value to all other processors. This broadcast often serves as a *synchronization* point for Krylov solvers. No process can continue computations until the broadcast has been completed. Note that, even with identical hardware, CPUs may need different amounts of time to complete identical tasks. Thus, a synchronization point is especially expensive because it leaves some processes idle while they wait upon the others. Parallelism also creates a potential numerical problem: the final computed value of a dot product can vary depending on the number of processes used. Recall that addition is not associative in floating-point arithmetic. Changing the number of processes alters the number of vector rows assigned to each process. Because the dot product computation begins with each process summing locally, this changes the order of addition, and, thus, the value of the dot product. These will be important considerations for studying GMRES in parallel. Because dot product calculations are inconsistent, the number of iterations required for convergence may change as we increase the number of MPI processes.

CHAPTER THREE

Minimum Residual Polynomial Preconditioning and the Power Basis Method

In this chapter, we introduce polynomial preconditioning. Then we discuss preconditioning with the GMRES polynomial and give the "power basis" formulation of the polynomial from [34, 3]. We use a new software implementation to demonstrate how the polynomial effectively transforms the spectrum of A and improves convergence for difficult problems. Then we discuss stability and starting vector issues that were unknown in [34] and give an algorithm for automating polynomial degree selection. Finally, we briefly contrast the power basis algorithm with previously known alternatives for generating the polynomial.

3.1 Introduction to Polynomial Preconditioning

To use polynomial preconditioning with (2.1), we choose a preconditioner $p(A)$ that is a polynomial of the matrix A. For right preconditioning, the new problem becomes

$$Ap(A)y = b, \tag{3.1}$$

$$x = p(A)y.$$

Then the Krylov subspace generated by GMRES is

$$\mathcal{K}_m(Ap(A), b) = span\{b, Ap(A)b, (Ap(A))^2 b, \ldots, (Ap(A))^{m-1} b\}.$$

Several authors have proposed and studied polynomial preconditioners, including [5, 6, 59, 24, 15, 25, 31, 33, 30, 34, 3, 42, 48, 29, 50, 52, 56]. There are several excellent reasons to use polynomial preconditioners. Polynomial preconditioners:

- *Help overcome limitations incurred by restarting:* When we extract an approximate solution $\hat{x}$ from $\mathcal{K}_m(A, b)$, then $\hat{x} = \pi(A)b$, where the degree of the polynomial π is $m-1$. (See (2.7) to (2.9).) Recall that in order for the residual norm to be small, the polynomial $1 - \alpha\pi(\alpha)$ needs to have small norm at the eigenvalues of A. This happens more easily when $\pi(\alpha)$ has high degree. Restarting GMRES prevents $\pi(\alpha)$ from attaining high degree. Adding a polynomial preconditioner $p(A)$ of degree *deg* gives a solution $\hat{x}$ from subspace $\mathcal{K}_m(Ap(A), b)$. Now $\hat{x} = \pi(Ap(A))b$, where π is a new polynomial of degree $m-1$. Thus, $\hat{x}$ now uses a polynomial of higher degree $deg \cdot (m-1)$. Therefore, the polynomial degree for the approximate solution is no longer limited by the subspace size.

- *Remap eigenvalues of A effectively to improve convergence:* A good preconditioner approximates A^{-1}. To improve GMRES convergence, it is best if the spectrum of the preconditioned operator $Ap(A)$ has eigenvalues clustered near one and well-spaced away from the origin. Some polynomial preconditioners, such as Chebyshev and least-squares polynomials, are created using a minimization criteria over the spectrum of A. If one has good estimates for the spectrum, this criteria demands that the polynomial $p(A)$ maps eigenvalues of A to near one.

- *Do not require explicit storage of matrix A:* Some general preconditioners require access to the actual matrix entries of A in order to compute the preconditioner. ILU, for example, uses the entries of A to compute a partial factorization. However, some real-world applications use a *matrix-free* implementation: Because the matrix A is too big or complicated to compute explicitly, they only supply a function that multiplies A (and sometimes A^*) times a vector. This is sufficient to run a Krylov solver, but such applications require that their preconditioners also be matrix-free. When applying a polynomial preconditioner, the matrix A is used only for applying matrix-vector products. No factorization is required. Thus, polynomial preconditioners can be successfully applied to matrix-free problems. If a polynomial preconditioner requires eigenvalue estimates for its construction, one can use the power method or the Arnoldi method to obtain this information without explicit matrix storage.

- *Are highly parallelizable:* When computing in parallel, preconditioners such as ILU and block Jacobi use domain decomposition to divide the work of computing and applying the preconditioner between multiple processors. While this reduces the expense of preconditioning, the preconditioner becomes less effective as the number of subdomains (parallel processes) increases. Splitting the problem into more subdomains means that the preconditioner will not be as accurate an approximation to A^{-1}. Polynomial preconditioning, on the other hand, does not use any sort of domain decomposition. In exact arithmetic, the

polynomial will be identical whether it is applied using one CPU or thousands of cores in parallel.

- *Help avoid communication in parallel computing:* Polynomial preconditioners are applied using only SpMVs and AXPYs. This makes them useful in parallel computing because they only require local communication. Thus, polynomial preconditioners can help to avoid the global communication and synchronizations that can be a bottleneck with dot products. This occurs because preconditioning allows less time to be spent in the Krylov solver's orthogonalization phase where dot products dominate. For further discussion of this communication-avoiding property, see Chapter Five.

Unfortunately, in practice, polynomial preconditioners are infrequently used. One reason may be that computing these polynomials often requires estimates of the extreme eigenvalues of A [50]. For symmetric matrices, which have all real eigenvalues, these bounds are straightforward to obtain. However, for a nonsymmetric matrix, one may need to compute a convex hull of the spectrum of A [52, p. 395], which can be very complicated and costly. Another concern which may have discouraged use is that polynomial preconditioners are often only stable for low degrees, limiting the effectiveness of the polynomial. (See [30] for details about the stability of applying a polynomial via its coefficients.) We aim to provide a polynomial preconditioner which is stable, simple to compute, and practical for a large variety of problems.

3.2 Preconditioning with the GMRES Polynomial

We advocate preconditioning with the GMRES polynomial, also known as the minimum residual polynomial. The GMRES polynomial preconditioner was used to spectrally transform eigenvalue problems in [56] and then was introduced for linear systems in [3, 34]. In [35], we further studied the preconditioner for methods IDR(s) and BiCGStab. The GMRES polynomial was also used for a hybrid GMRES solver in [41]. Our upcoming examples and analysis provide new insight into properties and behavior of the polynomial.

The minimum residual polynomial is constructed from information that corresponds to an initial GMRES run with a random right-hand side. Then the polynomial is applied as a preconditioner to GMRES as in system (3.2). Let v_0 denote the random right-hand side vector for generating the polynomial. (In Section 3.6, we discuss why a random starting vector is preferable to using $v_0 = b$.) When iterating to solve $Ax = v_0$, GMRES picks an approximate solution $\hat{x}$ from the Krylov subspace $\mathcal{K}_{deg+1}(A, v_0) = span\{v_0, Av_0, A^2v_0, \ldots, A^{deg}v_0\}$. Then there is a polynomial $p(A)$ of degree deg such that $\hat{x} = p(A)v_0$. This $p(A)$ is the polynomial that we choose be our preconditioner. It corresponds to the *minimum residual polynomial* $q(A) = I - Ap(A)$, which was first introduced in (2.12). The polynomial $q(A)$ will help us to analyze the properties of $p(A)$ and of the preconditioned operator $Ap(A)$.

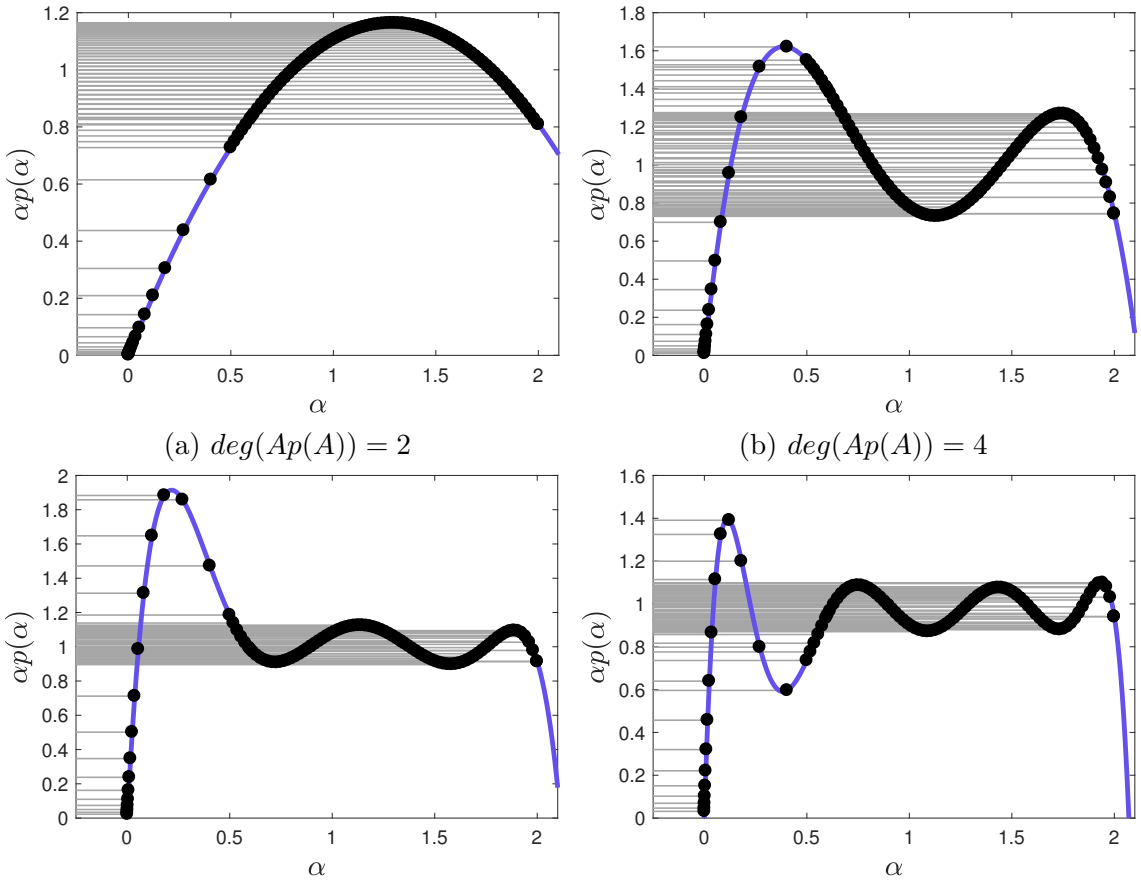
The minimum residual polynomial provides several advantages over other polynomial preconditioners. Unlike Chebyshev and least-squares polynomials, it is straightforward because no explicit eigenvalue estimates or bounds need to be constructed.

Thus, especially for nonsymmetric problems with complex spectra, it can be cheaper and simpler to obtain. Furthermore, because $p(A)$ is generated using GMRES, it can easily be composed with another preconditioner. When combined with a standard preconditioner M (such as ILU or block-Jacobi), the polynomial preconditioned system becomes

$$AMp(AM)y = b, \tag{3.2}$$
$$x = Mp(AM)y.$$

In this dissertation, we study two methods of generating the minimum residual polynomial preconditioner. The "power basis" method will be presented in the following section and the "roots" method is introduced in Chapter Four. For both methods, storage costs are minimal because the polynomial only requires storing a small vector of coefficients or roots in memory. For parallel computations with distributed memory, each processor can maintain its own copy of the polynomial information.

The GMRES minimum residual polynomial effectively remaps eigenvalues to improve convergence of GMRES. The initial GMRES run minimizes the residual norm $\|r\|_2 = \|v_0 - A\hat{x}\|_2 = \|(I - Ap(A))v_0\|_2$, so as $\|r\|_2$ goes to zero, we have that $Ap(A) \approx I$. Therefore $p(A)$ approximates A^{-1}. The spectrum of the preconditioned operator $Ap(A)$ will typically be better for convergence of GMRES than that of the original matrix A: The small eigenvalues of A will be mapped to well-separated eigenvalues of $Ap(A)$, and other eigenvalues will be clustered near one. Figure 3.1 shows polynomials $\alpha p(\alpha)$ of various degrees. The polynomials are generated using a diagonal matrix A which has 20 eigenvalues logarithmically spaced between 10^{-3} and 2 and 80 more eigenvalues evenly spaced between 0.5 and 2. The x-axis represents the



(a) $deg(Ap(A)) = 2$

(b) $deg(Ap(A)) = 4$

(c) $deg(Ap(A)) = 6$

(d) $deg(Ap(A)) = 8$

Figure 3.1: Plots of several minimum residual polynomials $\alpha p(\alpha)$ of varying degree. The x-axis spans the spectrum of the original matrix, while the y-axis spans the spectrum of the preconditioned matrix. Solid dots and horizontal gray lines show how eigenvalues of A are mapped by the polynomial to form the spectrum of the preconditioned operator $Ap(A)$.

original eigenvalues of A, and the y-axis shows where the eigenvalues are mapped by the polynomial to new eigenvalues of $Ap(A)$. Notice how the polynomials separate the tightly clustered eigenvalues near the origin. As the polynomial degree increases, the slope near the origin becomes steeper, providing even more separation. Furthermore, the larger eigenvalues of A are clustered near one, giving more improvement needed for faster GMRES convergence.

3.3 The Power Basis Method

We now introduce the *power basis* method of generating the polynomial as was used in [35, 34, 3]. To construct the polynomial preconditioner $p(A)$ of degree deg, first build the matrix of power basis vectors $V = [v_0, Av_0, \ldots, A^{deg}v_0]$. (To create a polynomial composed with a standard preconditioner M, simply form V with the operator AM instead of A.) Then solve the normal equations

$$(AV)^*AVy = (AV)^*v_0. \tag{3.3}$$

(As before, use the operator AM in place of A if combining with a standard preconditioner.) The elements of y are the coefficients of $p(A)$, that is,

$$p(A) = y_{deg}A^{deg} + y_{deg-1}A^{deg-1} + \cdots + y_1A + y_0. \tag{3.4}$$

We multiply $p(A)$ by a vector using Algorithm 3.1. (We use Matlab-style notation, so $y(1)$ indicates the first element of vector y corresponding to y_0, etc.) An alternative would be to apply the polynomial using a Horner iteration scheme. This would avoid the storage requirement of the temporary vector for multiplying by A and could be more stable. However, as our storage is not strictly limited, and Algorithm 3.1 does not appear to be the source of observed instability, we do not investigate this option further.

For higher polynomial degrees, the columns of V begin to converge to a single eigenvector of A and lose linear independence. This means that the normal equations (3.3) are very ill-conditioned, and computing the polynomial coefficients becomes unstable. Even so, the power basis method can give low-degree polynomials that significantly improve convergence of GMRES. In the next sections, we will demonstrate

Algorithm 3.1 $p(A)$ times z via coefficients

Input: sparse matrix $A \in \mathbb{R}^{n \times n}$, $z \in \mathbb{R}^{n \times 1}$, vector y of length $deg + 1$ containing the coefficients of $p(\alpha)$.

1: $prod = y(1) * z$
2: $temp = z$
3: **for** $i = 1$ to deg **do**
4: $\quad temp = A * temp$
5: $\quad prod = prod + y(i + 1) * temp$
6: **end for**
7: Return $prod$.

the effectiveness of polynomial preconditioned (PP)-GMRES at reducing costs and creating a better spectrum for convergence. Then we will discuss specifics of how stability problems can manifest and present a possible solution.

The results presented in this chapter are obtained using an implementation of the power basis polynomial preconditioner written in Trilinos [21]. (For more information about Trilinos, see Section 5.1.) The polynomial creation routine first forms the matrices for (3.3) and then uses the LAPACK [1] functions POTRF and POTRS to compute a Cholesky factorization and solve the system of normal equations. The implementation also provides a function to apply $p(A)$ to a vector z as in Algorithm 3.1. All experiments in this chapter are run in serial, except as indicated otherwise in Section 3.7.

3.4 Effectiveness of the Polynomial

Examples for the remainder of this chapter run GMRES(m) to a relative residual tolerance of 1×10^{-8}. We use two steps of classical Gram-Schmidt orthogonalization per iteration (denoted in Trilinos as ICGS(2)). The dot products from each orthogonalization step are combined into a single block dot product, as mentioned in Section

2.1.1. Thus, each orthogonalization step requires two block inner products and one norm, which we count as three "dot" products per iteration in the figures to follow. The algorithm needs $deg + 1$ SpMVs and two block inner products to create the polynomial, which are also included in the cost counts below.

For all experiments in this section, we use the matrix e20r0100 from Matrix Market [2], a real nonsymmetric matrix of size $n = 4241$ from fluid dynamics. This matrix is sparse with about 0.73% nonzeros and an average of 31 nonzeros per row. This matrix has a high condition number, estimated by Matrix Market at 2.15×10^{10}, and proves to be extremely difficult for GMRES. In practice, e20r0100 may be best addressed using a direct solver, but it is representative of the difficulties that one may encounter in GMRES.

Example 3.4.1. In this example, we choose subspace size $m = 50$ and a right-hand side b with random entries. We generate the polynomial using $v_0 = b$. Figure 3.2 shows residual norm convergence for unpreconditioned GMRES (indicated by $deg = 0$) and polynomial preconditioned GMRES for $deg = 3, 5, 7, 10$. Convergence in relation to SpMVs is shown on the top and convergence relative to inner products is shown on the bottom. It is important to observe that even though applying a polynomial requires more SpMVs at each iteration, the polynomial preconditioner often reduces the total number of SpMVs needed to attain convergence.

Figure 3.2 shows that without preconditioning, the relative residual norm only improves by one order of magnitude before stalling out. We see a distinct improvement when adding a polynomial preconditioner and when raising the degree. Unlike

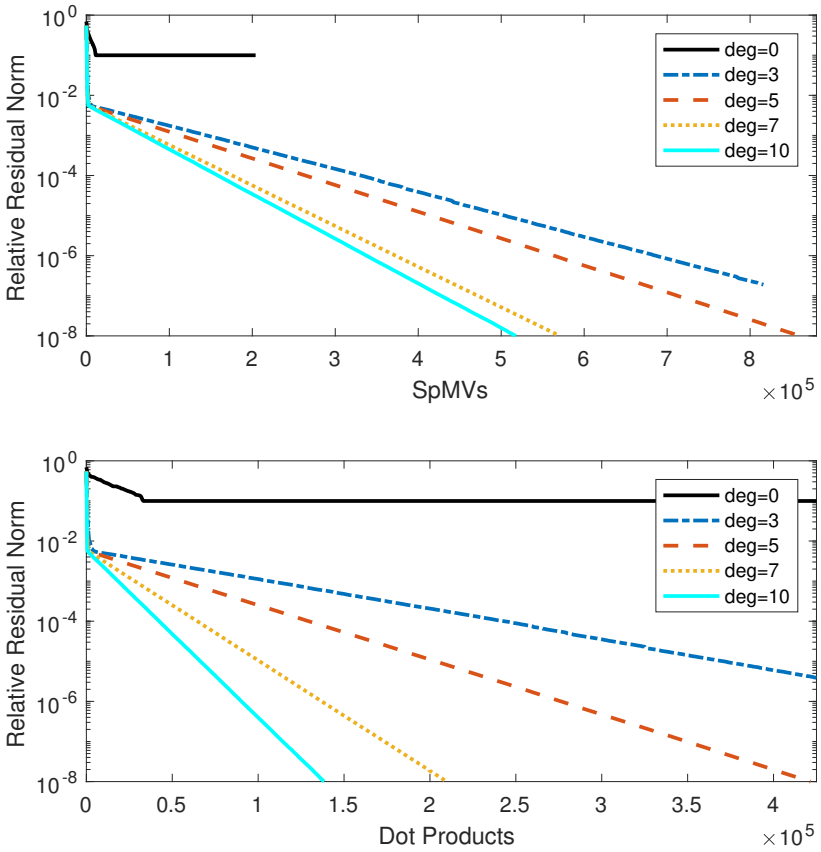
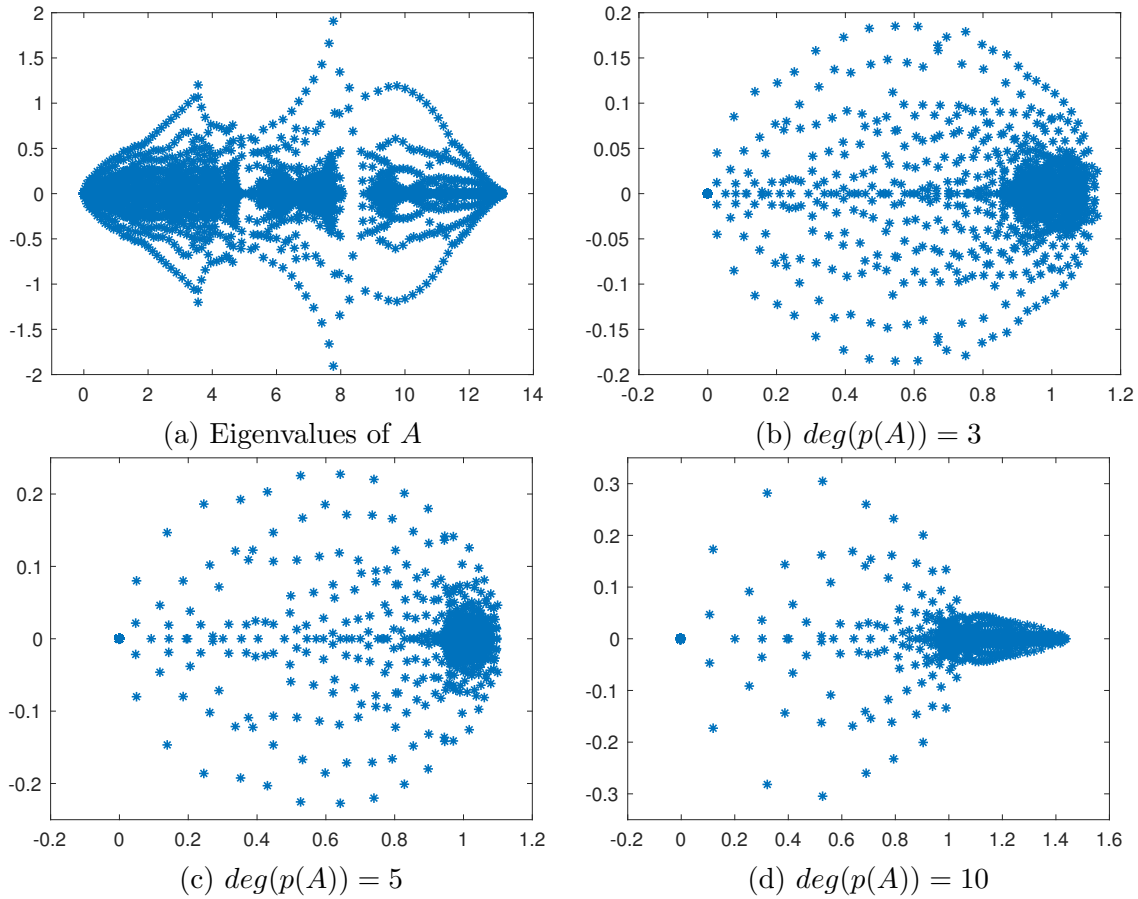


Figure 3.2: Residual norm convergence for the matrix e20r0100 with a random right-hand side in terms of SpMVs (top) and dot products (bottom). Subspace size is $m = 50$. Degree 0 indicates no preconditioning. All tests were run to 200,000 maximum iterations.

the original problem, the preconditioned problems will all converge. The degree 10 polynomial gives convergence in 517,452 SpMVs, a 40% decrease over the SpMVs needed for degree 5. The improvement in inner (dot) products is more substantial. The degree 5 problem converges in 421,559 dot products, while the degree 10 problem only requires 138,350 dot products. Thus, the degree 10 polynomial converges in less than one-third the number of dot products required for degree 5. This reduction could prove significant for parallel architectures.

Considering the spectra of A and $Ap(A)$ helps to explain the improvement from polynomial preconditioning. Figure 3.3 shows the eigenvalues of A and the new preconditioned spectrum after applying $p(A)$ of degrees $3, 5$, and 10. The matrix A is indefinite, with 1199 eigenvalues that lie in the left half of the complex plane. The largest eigenvalue in the left half plane has magnitude 0.0013, and the smallest has magnitude 1.4×10^{-6}. While the polynomials do not create a significant change in these eigenvalues, they do help to cluster eigenvalues on the right-half plane away from the origin. We find that the ratio of largest to smallest magnitudes of eigenvalues on the right half plane (a very rough approximation of the condition number) is 357.1 without preconditioning. With only a degree 3 polynomial, this ratio improves almost ten times to 39.2. When the polynomial is degree 10, the ratio has improved to 12.4. This suggests that the preconditioned problem does, in fact, have a much better spectrum for GMRES.

Example 3.4.2. In this example, we use the same problem and polynomials from Example 3.4.1, but we increase the subspace size to $m = 100$. Though unpreconditioned GMRES no longer stalls with the larger subspace, convergence is too slow to run to completion. We estimate that about 1.3 million SpMVs and 3.9 million dot products are needed to converge. Figure 3.4 shows the improvement with preconditioning. The polynomial of degree 5 helps to attain convergence at a cost of 147,421 SpMVs and 72,974 dot products. With degree 10, the cost is 40,008 SpMVs and 10,799 dot products. Thus, if we performed this computation in parallel, we could reduce global communication calls by about 2.5 orders of magnitude over no precon-



(a) Eigenvalues of A

(b) $deg(p(A)) = 3$

(c) $deg(p(A)) = 5$

(d) $deg(p(A)) = 10$

Figure 3.3: Eigenvalues of the preconditioned matrix $Ap(A)$ where A is the matrix e20r0100. As the degree of $p(A)$ increases, the eigenvalues are more clustered around 1 and more loosely scattered near zero.

ditioning. With that comes approximately 1.5 orders of magnitude improvement in matrix-vector products, reducing local communication as well.

3.5 Stability Issues and Degree Selection

It may be difficult for the user to determine when it is best to stop raising the polynomial degree. Raising the degree often results in a better preconditioner, but it can reach a point of diminishing returns. The polynomial preconditioner can decrease both the number of inner products and SpMVs required to converge, but sometimes

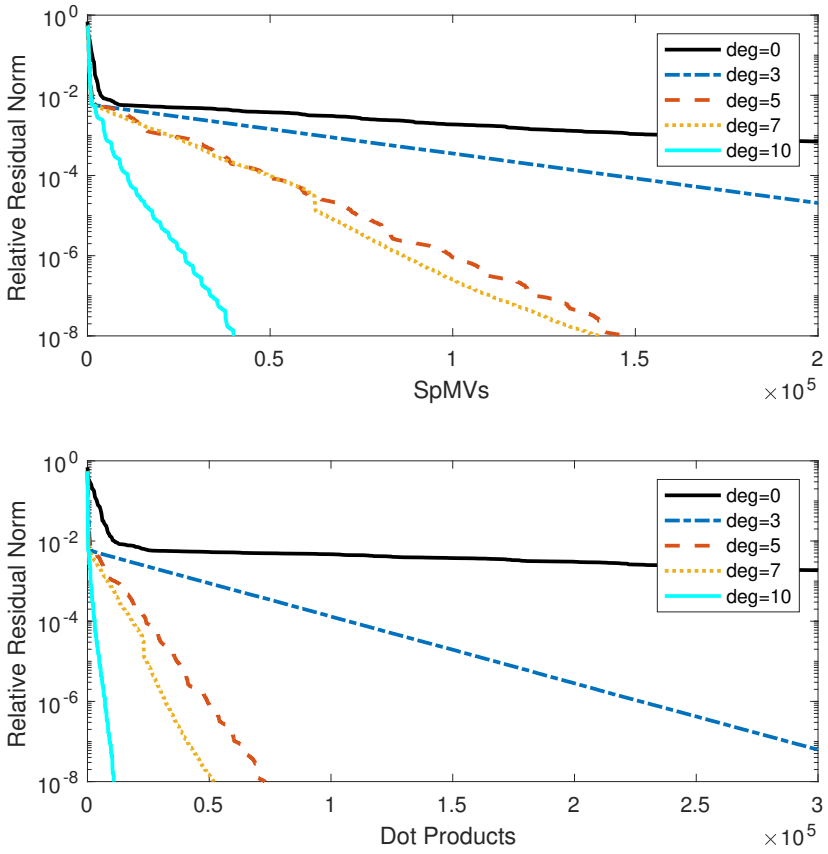


Figure 3.4: Residual norm convergence for the matrix e20r0100 with a random right-hand side in terms of SpMVs (top) and dot products (bottom). Subspace size is $m = 100$. Deg 0 indicates no preconditioning.

the iteration count and inner products are reduced at the expense of more SpMVs. Fortunately, many matrices are stored so that SpMVs only require communication with neighboring processors. Thus, for communication reduction, it may be beneficial to perform extra SpMVs in order to avoid inner products that require global communication and synchronization. In this case, polynomial preconditioning may still reduce total solve time even if the number of SpMVs is increased. See Chapter Five for further discussion.

We ran serial tests on several different matrices to determine the effects of raising the polynomial degree. All tests were performed with a right-hand side $b = Ax$ where

x was a randomly generated solution vector. We let $v_0 = b$ and choose a maximum subspace size of 50. Matrices bwm2000, orsirr1, s1rmq4m1, and e20r0100 can be obtained via Matrix Market [2]. The matrix BiDiag1, with $n = 2000$, has the values $1, 2, \ldots, 2000$ on the diagonal and 0.05 for all elements of the superdiagonal. Matrix BiDiag2, with $n = 5000$, has $0.1, 0.2, \ldots, 0.9, 1, 2, \ldots, 4991$ on the diagonal and 0.2 on all elements of the superdiagonal.

Figure 3.5 shows the number of SpMVs required to reach a relative residual tolerance of 1×10^{-8} for polynomials $p(A)$ of degrees $3, 5, 7, 10, 12, 15, 17, 20$. Results for no preconditioning correspond to degree 0 on the plot. Figure 3.6 shows the corresponding number of inner products required for convergence, where one inner product is counted for each of two passes of Gram-Schmidt orthogonalization and one more for the norm. For matrices bwm2000 and e20r0100, convergence stagnates with no preconditioning. The problem e20r0100 first converges with the preconditioner of degree 3, and bwm2000 first converges with degree 10.

The results suggest that preconditioning is more likely to reduce matrix-vector products for difficult problems than for simpler ones. For the easiest problem, BiDiag1, matrix-vector products increase with preconditioning, even for degree 3. For matrices e20r0100, bwm2000, and s1rmq4m1, which were most difficult, the expense of SpMVs decreases with preconditioning up until degree 10. For the other two problems, the number of SpMVs decreases slightly for very low-degree polynomials and begins to rise again, with no further savings after degree 10. In spite of this, for the matrix s1rmq4m1, the number of SpMVs with preconditioning is still less than that of no preconditioning up through degree 20.

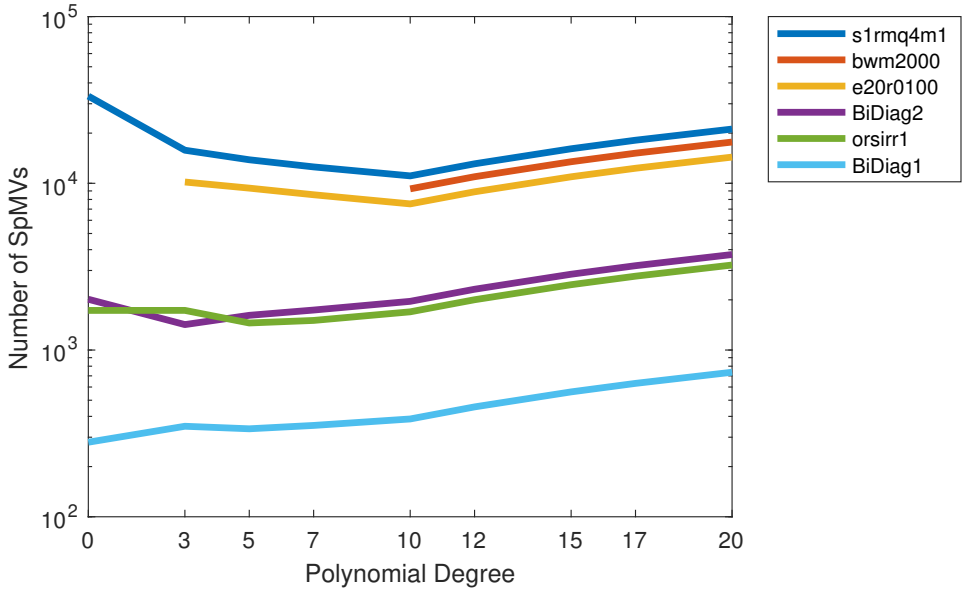


Figure 3.5: The number of SpMVs required to attain convergence for several preconditioned matrices with different polynomial degrees. Subspace size is 50. Degree 0 indicates no preconditioning.

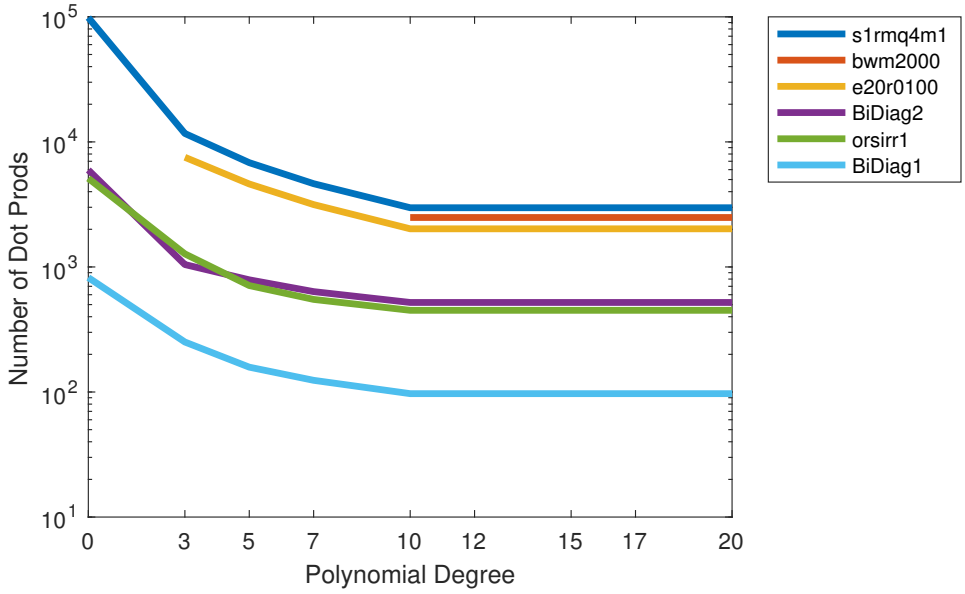


Figure 3.6: Total number of dot products (inner products plus norms) from orthogonalization for several preconditioned matrices with different polynomial degrees. Subspace size is 50. Degree 0 indicates no preconditioning.

Unlike with SpMVs, polynomial preconditioning is consistent in reducing the number of inner products for all problems in Figure 3.6, regardless of difficulty. By the time the polynomial degree is increased to 10, the number of inner products has decreased by approximately an order of magnitude or more for all problems. However, after degree 10, the number of inner products remains constant while the number of SpMVs is increasing. Since the number of inner products varies directly with the iteration count, this means that the iteration count for GMRES is also not being reduced any more. Further investigation reveals that this is due to the polynomial coefficients.

We found that after the polynomial degree gets large enough, increasing it failed to produces any changes of significant magnitude in the coefficients. Example coefficients for the matrix s1rmq4m1 are shown in Table 3.1, arranged with the constant term on top and higher-order coefficients descending. Note first that even though the higher-order coefficients for the degree 7 and 10 polynomials are well below machine epsilon, the changes in the coefficients still make a difference in the convergence of the problem. Then notice that for degrees $10, 12$, and 15, the first eleven polynomial coefficients remain the same (shaded boxes). After degree 10, the polynomial coefficients are "frozen." The lower-order coefficients are not changing, and the new higher-order coefficients in the degree 12 and 15 polynomials are so near zero that they do not provide any additional information. This explains why the degree 12 and degree 15 preconditioners give no improvement in cost over the degree 10 preconditioner. In fact, they are more expensive due to the extra SpMVs incurred with near-zero coefficients. Polynomials for the other matrices tested followed a similar trend.

Table 3.1: Coefficients for the polynomial $p(A)$ for the matrix s1rmq4m1, ordered with the constant term on top and the highest-order coefficient on bottom.

Deg 7	Deg 10	Deg 12	Deg 15
3.70798e-05	6.08343e-05	6.08343e-05	6.08343e-05
-5.2613e-10	-1.45759e-09	-1.45759e-09	-1.45759e-09
3.82756e-15	1.86807e-14	1.86807e-14	1.86807e-14
-1.59154e-20	-1.45193e-19	-1.45193e-19	-1.45193e-19
3.93091e-26	7.29453e-25	7.29453e-25	7.29453e-25
-5.69776e-32	-2.44543e-30	-2.44543e-30	-2.44543e-30
4.47271e-38	5.52107e-36	5.52107e-36	5.52107e-36
-1.46677e-44	-8.28868e-42	-8.28868e-42	-8.28868e-42
	7.92937e-48	7.92937e-48	7.92937e-48
	-4.3729e-54	-4.3729e-54	-4.3729e-54
	1.05777e-60	1.05777e-60	1.05777e-60
		7.47962e-208	7.47962e-208
		6.051e-237	6.051e-237
			-3.85124e-257
			-1.25918e-274

We attribute this problem to two causes: 1) The columns of the power basis matrix V begin to lose linear independence as more powers of A are applied. Therefore, appending columns to V really does give less new information as the polynomial degree increases. 2) The normal equations in 3.3 are highly ill-conditioned. (Solving normal equations with an ill-conditioned matrix AV can square an already high condition number.) This ill-conditioning means that polynomial coefficients may be computed inaccurately.

We observe that the appearance of near-zero and "frozen" coefficients corresponds with a positive return value *info* in the LAPACK function POTRF when forming the polynomial. This warning code means that the matrix $(AV)^*(AV)$ is not positive definite (it is numerically singular) and the Cholesky factorization could not be completed. Of the six matrices discussed in this section, three first give LAPACK warn-

ings starting with degree 10, and the other three examples begin to give warnings at degree 12. Despite the warning, all coefficients are still computed. In examples with other matrices, zeros or NaNs were computed after the LAPACK error occurred and the polynomial degree was raised too high. Attempts to solve (3.3) more accurately failed. We tried using a more stable LAPACK routine POSVX, which equilibrates the system before Cholesky factorization and has an additional option to improve the solution using iterative refinement. Neither of these options resulted in better polynomial coefficients. Another possible solution would be to find a QR factorization for solving the normal equations, but in [34] this resulted in less accurate polynomial coefficients. Thus, we did not consider it here.

These LAPACK warnings do not always correspond exactly to the polynomial degree where the SpMV count begins to increase, and sometimes they do not occur until well after the polynomial coefficients "freeze". However, the warnings still may be a good starting point for choosing an optimal polynomial. We suggest automating the polynomial degree selection by constructing the highest degree polynomial possible without a warning from LAPACK. For difficult examples that benefit from higher polynomial degrees, this procedure seems likely to minimize the number of inner products and norms while avoiding extra SpMVs.

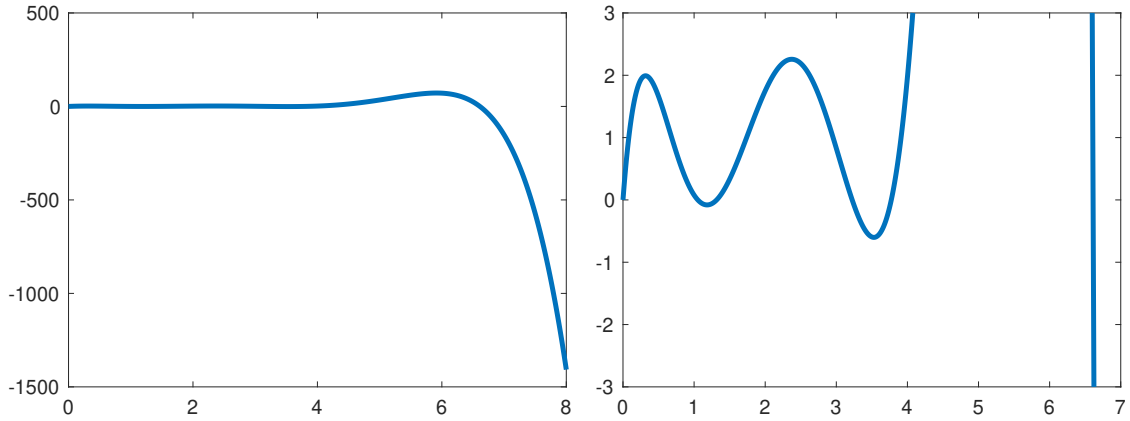
It is worth noting that there are matrices, such as Sherman5 from Matrix Market, where this degree selection strategy fails. This indefinite matrix is an extremely difficult problem for GMRES. (See its spectrum plotted in Figure 4.23.) Our polynomial of degree 7 was a very successful preconditioner because the spectrum of $Ap(A)$ was entirely in the one side of the complex plane. With higher degree polynomials, the

matrix $Ap(A)$ was once again indefinite and GMRES did not converge. However, LAPACK did not give any warnings until degree 15. In such instances, it may be best to take the auto-selection degree as an upper bound and try to obtain results with lower-degree polynomials.

3.6 Choosing a Vector to Generate the Polynomial

All experiments thus far have successfully generated the polynomial preconditioner using $v_0 = b$, the problem right-hand side, as suggested in [34]. This choice worked well in the previous sections because the problem right-hand side was generated using randomization. More structured right-hand sides may yield a poor polynomial preconditioner.

Example 3.6.1. Consider the discretized Laplacian equation $-\nabla^2 u = f$ over a square domain, with constant source function $f(x) \equiv 1$ and zero boundary conditions (Example 1.1.1 [13]). The matrix and right-hand side are generated with Firedrake [46] software using a function space of continuous piecewise-linear polynomials. The mesh size is $N = 200$, so the matrix size is $n = 40{,}401$. The eigenvalues of this matrix are all real-valued and lie in the interval $[0, 8]$, with several eigenvalues very close to 8. All values of the right-hand side vector b are very close to either 0 or -1×10^{-4}. Figure 3.7 shows the polynomial $\alpha p(\alpha)$, where $p(\alpha)$ has degree 5 and is generated with $v_0 = b$. The x-axis corresponds to the spectrum of A, and the y-axis shows the range of eigenvalues of $Ap(A)$. Recall that if $p(A)$ is a good preconditioner, the large eigenvalues of A will be mapped close to 1 and the small eigenvalues of A will be well-separated between 0 and 1. This polynomial does nothing of the sort.



(a) Polynomial from bad start vector (b) Closeup of bad polynomial

Figure 3.7: Polynomial $\alpha p(\alpha)$ for the Laplacian matrix, where the degree of $p(\alpha)$ is 5 and $v_0 = b$. Plotted over the interval $[0, 8]$ which contains the eigenvalues of A. Closeup on the right.

The largest eigenvalues near 8 are mapped to near -1400, and the eigenvalues in the middle of the spectrum are mapped to values as small as $-1/2$ up to larger than 2. The smallest eigenvalues are mapped to the interior of the spectrum.

The preconditioned matrix $Ap(A)$ is highly indefinite and is much harder for GMRES than the original problem. After 2550 iterations of GMRES(50), the relative residual almost stalls out at 0.856. The vector b appears to have very small components in the directions of eigenvectors of A that correspond to large eigenvalues. (So some of the β_i values in (2.10) and (2.13) are very tiny.) Thus, the GMRES minimum residual polynomial effectively ignores those large eigenvalues; it does not need to be small in those directions.

We now generate a random vector v_0 with uniformly distributed elements in $[-1, 1]$. The new polynomial $\alpha p(\alpha)$, where $p(\alpha)$ has degree 5, is shown in Figure 3.8. The preconditioner works very well; GMRES(50) reaches a relative residual

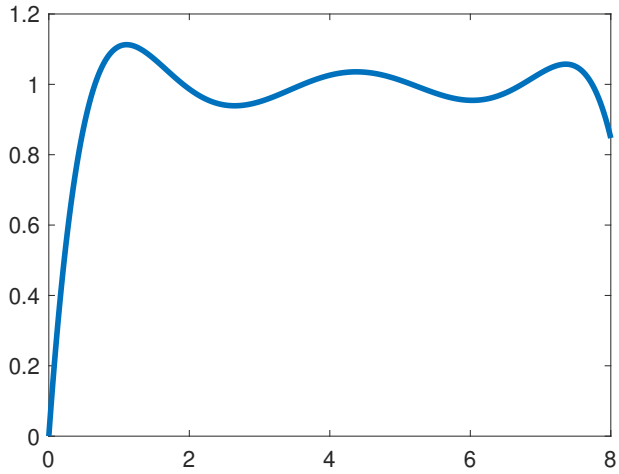


Figure 3.8: Polynomial $\alpha p(\alpha)$ for the 2D Laplacian, where $p(\alpha)$ has degree 5, generated with v_0 as a random vector.

norm of 1.0×10^{-8} in only 148 iterations. The plot of the polynomial shows a steep slope near zero which is able to separate the small eigenvalues of A. The rest of the spectrum is mapped between 0.8 and 1.2. It appears that a random vector helps the polynomial to address all parts of the spectrum better than a structured right-hand side vector. For the remaining experiments in this chapter, we let v_0 be a random vector.

3.7 Some Parallel Numerical Results

Experiments in this section were performed using the Kodiak cluster at Baylor University. We used the Cray compute nodes, which each had dual 18-core Intel E5-2695 V4 (Broadwell) processors and 256GB RAM. All tests were performed using only one compute node.

Example 3.7.1. The example that follows tests a finite element discretization of the convection-diffusion equation

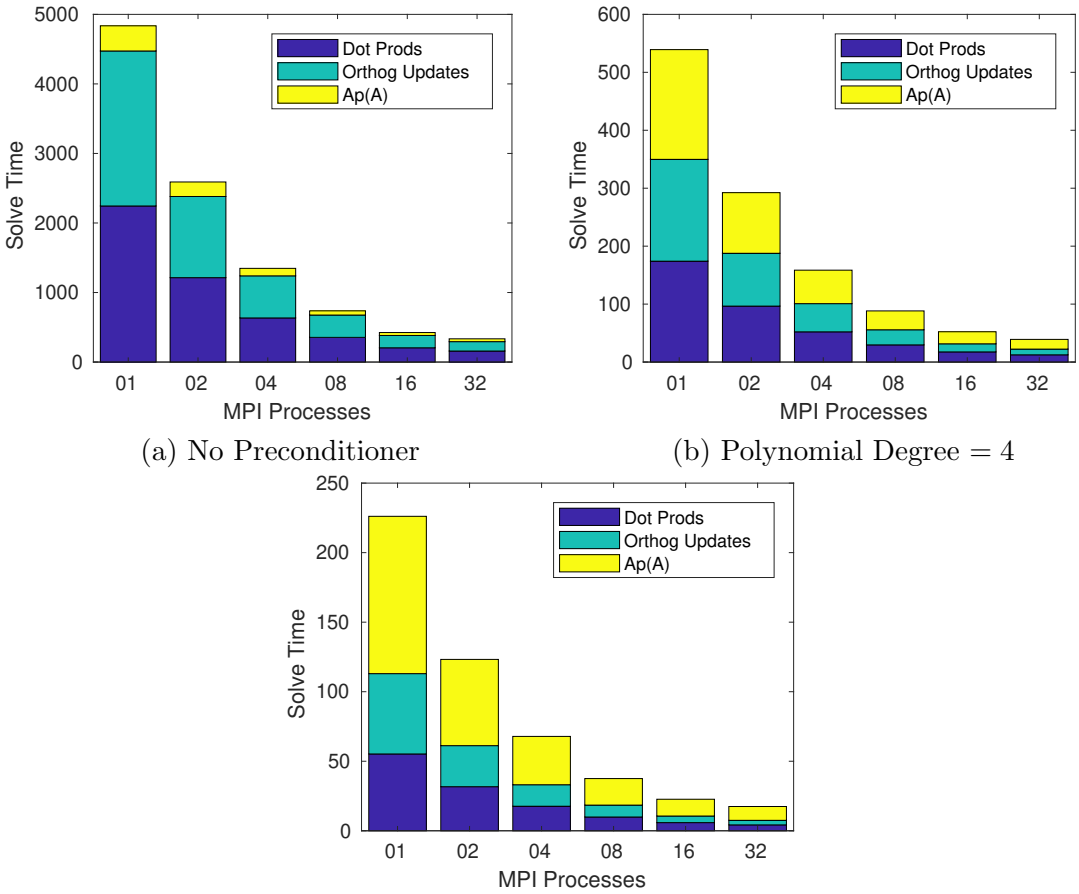
$$-\epsilon\nabla^2 u + \vec{w} \cdot \nabla u = f.$$

The matrix and right-hand side are again generated with Firedrake [46] software using a function space of continuous piecewise-linear polynomials. The domain is a 2D unit square mesh centered at the origin with $N = 1024$, yielding a matrix of size $n = 1{,}050{,}625$. Similar to Example 6.1.4 in [13], we let $f \equiv 0$, $\epsilon = 1/2$, and

$$\vec{w} = (2y(1 - x^2), -2x(1 - y^2)).$$

We use Dirichlet boundary conditions, where $u = 1$ on boundary $x = 1$, and $u = 0$ on the remaining boundaries. Because Trilinos gives a different random vector depending on the number of MPI processes used, we generate a random vector v_0 in Matlab and read it in from file to generate each polynomial in the following tests. Thus, the starting vector v_0 is the same for all polynomials created, regardless of degree or number of MPI processes.

Without preconditioning, GMRES(50) does eventually converge. The bar graphs in Figure 3.9 show solve times over increasing numbers of MPI processes for no preconditioning and polynomial preconditioners of degrees 4 and 9. On one processor, the autodegree selection algorithm chooses degree 9 as optimal. The bars are split to show three different timings (in seconds). Time spent in the orthogonalization kernel is indicated by the bottom and middle parts of the bar. The dot products (including norms) are in the bottom portion and the vector updates (AXPYs) for orthogonalization are in the middle. The top part of the bar indicates time spent applying $Ap(A)$



(a) No Preconditioner

(b) Polynomial Degree = 4

(c) Polynomial Degree = 9

Figure 3.9: Solve times (in seconds) for a convection-diffusion problem using no preconditioning and polynomials of degrees 4 and 9 over increasing numbers of MPI Processes. The bottom section of each bar gives time spent in dot products for orthogonalization, the middle section gives time for AXPYs for orthogonalization, and the top section gives time for applying $Ap(A)$.

to a vector (SpMVs and AXPYs). Timings for other operations, including polynomial construction, were negligible.

Notice first the differences in scaling on the y-axes. The polynomial preconditioner of degree 4 gives almost 10 times improvement in solve time over no preconditioning, and degree 9 gives over 20 times improvement in solve time over no preconditioning. Strong scaling is roughly the same with the preconditioned and unpreconditioned

problems: Solve time decreases by about half as we go from 1 to 2 MPI processes and by a little less than a half each time we double the number of processes again. Although running on a single compute node means that communication consists only of reading shared memory, orthogonalization dominates solve time when no preconditioning is used. In particular, with no preconditioning, dot products and norms require almost half of the total solve time. With a degree 9 preconditioner, less than one-fourth of the compute time is used for dot products and norms, while a much greater proportion of time is used applying the preconditioned matrix using SpMVs and vector updates. This shows potential to further reduce solve time by implementing the SpMVs for the polynomial application using a communication-avoiding algorithm such as the Matrix Powers Kernel. See Section 5.4 for further discussion.

We observed that for a fixed degree, the polynomial coefficients changed as the number of MPI processes increased, even with the starting vector v_0 held constant. This is likely because results of dot products for forming the normal equations in (3.3) were inconsistent as the number of processes increased. We discussed this phenomenon in Section 2.7. Thus, the number of iterations required for GMRES convergence varied with the number of MPI processes, even for a fixed polynomial degree. While it would be ideal to have consistent convergence behavior when increasing the number of MPI processes, all the polynomial preconditioners generated here greatly improve convergence.

3.8 More Stable Polynomial Implementations

In [34], Morgan and coauthors present several more stable alternatives for computing and applying the minimum residual polynomial preconditioner. First, they investigate several options for computing the polynomial coefficients in a more stable manner. None of these techniques gives significant improvement, so they start to investigate methods of applying the polynomial coefficients indirectly using a different basis. The two most promising tecniques are the "Newton basis" method based on [7] and the "Arnoldi basis" method, a variation of which was first introduced in [56]. Both are more stable than the power basis method, but they both require significantly more expense to create and apply the polynomial.

3.8.1 Newton Basis Method

Instead of building power basis vectors, the Newton method builds power basis vectors with Ritz value shifts at each step. The following vectors are constructed and normalized:

$$[v_0, (A-\theta_1 I)v_0, (A-\theta_2 I)(A-\theta_1 I)v_0, (A-\theta_3 I)(A-\theta_2 I)(A-\theta_1 I)v_0, \ldots, \\ (A-\theta_{deg} I)(A-\theta_{deg-1} I)\cdots(A-\theta_2 I)(A-\theta_1 I)v_1] \tag{3.5}$$

The Newton basis method then forms and solves a new set of normal equations similar to (3.3). The resulting values in the solution vector y are coefficients of $p(A)$ with respect to the Newton basis. See [35, p. 12-13] for precise details of the algorithm.

The Newton basis method has the disadvantage of having to build not one, but two GMRES bases of length *deg* to generate a degree *deg* polynomial. The first is the GMRES basis for finding the upper-Hessenberg matrix H and the Ritz values. This

basis is orthonormalized. The second is the Newton basis, which is normalized but not orthogonalized. This means that in order to set up the polynomial, we need *deg* more dot products and double the SpMVs than when using the power basis method. Furthermore, even though normal equations with a Newton basis will undoubtedly be more stable than with a power basis, they still may be prone to being ill-conditioned. Applying the polynomial to a vector requires roughly one SpMV and two AXPYs per degree, which is similar to the new method presented in Chapter Four.

3.8.2 Arnoldi Basis Method

The Arnoldi basis vectors method is also more stable than the power basis method. To set up the polynomial, it first creates one orthonormalized Krylov basis per the usual GMRES iteration. It then solves a least-squares problem to get polynomial coefficients with respect to the GMRES basis and then works backwards to undo the basis vectors when applying the polynomial. The Arnoldi basis method is as follows:

1. Choose $v_0 = b/\|b\|_2$ or v_0 random with norm 1. Run $deg+1$ steps of the Arnoldi iteration with starting vector v_0 to build the upper-Hessenberg matrix $\overline{H}_{deg+1}$ from the Arnoldi recurrence (2.3). Solve the least squares equation for

$$\hat{d} = \underset{y \in \mathbb{R}^{deg+1}}{\arg\min} \left\| \gamma e_1 - \overline{H}_{deg+1} y \right\|_2,$$

   just as when finding an approximate solution with GMRES.

2. Use the upper-Hessenberg matrix $\overline{H}_{deg+1}$ and short solution vector $\hat{d}$ to apply the polynomial $p(A)$ as needed using Algorithm 3.2.

Algorithm 3.2 $p(A)$ times z Arnoldi basis method [56, p. 69]

Input: sparse matrix A, upper-Hessenberg matrix $\overline{H}$ and least-squares solution vector $\hat{d}$ from an initial GMRES run, vector z

Output: $p(A)z$

1: $v_1 = z$, $V_1 = [v_1]$;
2: **for** $j = 1$ to deg **do**
3: $\quad w = Av_j$, $\tilde{h} = \overline{H}(1{:}j, j)$
4: $\quad f = w - V_j\tilde{h}$
5: $\quad v_{j+1} = f/h_{j+1,j}$, $V_{j+1} = [V_j, v_{j+1}]$
6: **end for**
7: Return $V_{deg+1}\hat{d}$.

(Algorithm 3.2 uses Matlab-style notation in that $H(i{:}j, j)$ denotes the ith to jth rows of column j in matrix H.)

Since the Arnoldi basis method only needs one set of Krylov vectors to set up the polynomial, polynomial creation is much cheaper than in the Newton method (which needs two sets of Krylov vectors). However, when applying the polynomial to a vector, the Arnoldi basis method is significantly more expensive than the Newton method: For each polynomial degree, it needs one SpMV with A, one AXPY, and one matrix-vector product with the tall-skinny matrix V_j. This matrix-vector product with V_j can be expensive with a large matrix; see Example 5.2.5. This algorithm also has the disadvantage that it requires storing $deg + 1$ vectors to apply the polynomial. This is a substantial increase in storage requirement over the Newton basis method, which only required storing a length $deg + 1$ basis in the polynomial setup phase.

3.8.3 Towards a Better Polynomial

While the GMRES "power basis" polynomial can be effective using low degrees to reduce cost to convergence and create a better spectrum, high degree polynomials

were unstable. Both the Newton and Arnoldi basis methods allow us to create higher degree polynomials than with the power basis method. However, [34] shows that even these methods are unstable when a problem has a large outstanding eigenvalue. Furthermore, the Newton basis method incurs significant extra cost in generating the polynomial, and the Arnoldi basis method has significant cost in applying it. We need an implementation that is less costly in terms of floating-point operations and is stable for problems with an outstanding eigenvalue. Our alternative is developed in the upcoming chapter.

CHAPTER FOUR

A New Stable and Effective Polynomial Implementation

In this chapter, we introduce a new, more stable implementation of the GMRES polynomial as a preconditioner for linear equations. Instead of computing the coefficients of the polynomial, we use harmonic Ritz values to implement it via its roots. We analyze stability problems caused by outstanding eigenvalues and give an algorithm to mitigate the issue by adding extra copies of roots. Our investigation determines the source of error in the polynomial application and leads to additional methods to check for stability. Finally, we discuss special situations where damping may be useful and give an example of double polynomial preconditioning.

There are four concepts in this chapter that we first presented in [14] in the context of eigenvalue problems: 1) the application of $Ap(A)$ using harmonic Ritz values as roots, 2) adding extra copies of roots for stability, 3) damping of polynomials with high oscillations, and 4) double polynomial preconditioning. Since it is outside the scope of this dissertation to discuss the details of the Arnoldi method and eigenvalue problems, we will present each of these concepts anew within the context of solving linear equations. The following ideas originated in our study of the new polynomial for linear systems: 1) the new formula for $p(A)$ in Section 4.2, 2) the stability analysis in Section 4.6, and 3) the applications of damping in Section 4.9.

4.1 Applying the Preconditioned Operator $Ap(A)$

Recall that we want to solve the polynomial preconditioned problem

$$Ap(A)y = b, \tag{4.1}$$

$$x = p(A)y \tag{4.2}$$

where $p(A)$ is the related to the GMRES minimum residual polynomial $q(A) = I - Ap(A)$ as discussed in Section 3.2. Again, we assume that A is a real-valued matrix. We will note modifications for matrices with complex entries. From this point forward, the degree d of the polynomial preconditioner indicates the degree of the preconditioned operator $Ap(A)$ instead of the degree of $p(A)$.

While at first it seems natural to apply the polynomial $p(A)$ using its coefficients, we have seen that this approach is limited; algorithms using coefficients are unstable for problems with outstanding eigenvalues and can be costly. An alternative is to apply the polynomial using its roots. Recall (Section 2.6) that after a cycle of GMRES, the roots of the minimum residual polynomial $q(\alpha) = I - \alpha p(\alpha)$ are the harmonic Ritz values $\{\theta_i\}_{i=1}^{d}$. To set up the polynomial, we first run d steps of GMRES with a random right-hand side v_0 to create the matrix $H_{d+1,d}$ from the Arnoldi recurrence (2.2). Then we use formula (2.20) to compute the roots $\{\theta_i\}_{i=1}^{d}$ of $q(\alpha)$. Since $q(0) = 1$, we can write

$$q(\alpha) = \prod_{i=1}^{d} \left(1 - \frac{\alpha}{\theta_i}\right). \tag{4.3}$$

Then because $q(A) = I - Ap(A)$, we apply the preconditioned operator as

$$Ap(A) = I - q(A) = I - \prod_{i=1}^{d} \left(1 - \frac{A}{\theta_i}\right). \tag{4.4}$$

To improve numerical stability and mitigate rounding errors when applying the polynomial, it is best to order the roots θ_i with a modified Leja ordering [7]. The modified Leja ordering maximizes the product of distances between one entry and the previous entries. It also orders complex conjugate pairs consecutively. (If A has complex-valued entries, use a Leja ordering [10]. The Leja ordering is exactly like the modified Leja, except that complex conjugate pairs are not assumed to exist and so are not ordered consecutively.) Our modified Leja ordering in Algorithm 4.1 has a slight modification over the version from [7]: we construct the logarithm of the product of distances between rather than the actual product (line 9). This logarithm operation does not change the ordering of the θ_i values, but it scales down the value of *prod* to avoid overflow and underflow problems. Beginning in Section 4.5, we will further modify Algorithm 4.1 to order repeated harmonic Ritz values. In [47] the authors present alternative methods to avoid over/underflow and to sort repeated values. See [23, p. 266-268] for a good summary of challenges with a modified Leja ordering and possible solutions.

When multiplying a vector by $Ap(A)$ using (4.4), we apply the product (4.3) one factor at a time. Since the modified Leja ordering orders complex conjugates consecutively, we can avoid complex arithmetic by combining conjugate pairs. Suppose that for some k we have $\theta_k = a + bi$ and $\theta_{k+1} = a - bi$. Then

$$\left(1 - \frac{\alpha}{\theta_k}\right)\left(1 - \frac{\alpha}{\theta_{k+1}}\right) = 1 + \frac{\alpha^2 - 2\alpha a}{a^2 + b^2}. \tag{4.5}$$

Thus, the next two factors of (4.4) can be computed using only real arithmetic and then applied to the vector in the same step. Full details are in Algorithm 4.2.

Algorithm 4.1 Modified Leja ordering

Input: Vector *theta* of d values to be ordered.
Output: Vector of sorted values.

1: $j = 1$
2: $sorted(1) = theta(k)$ such that $theta(k)$ has the largest absolute value of all $theta(i)$
3: **if** $sorted(1)$ is complex **then**
4: $sorted(2)$ = complex conjugate of $sorted(1)$
5: $j = j + 1$
6: **end if**
7: **while** $j <= d$ **do**
8: **for** $i = 1 : d$ **do**
9: $prod(i) = 1$
10: **for** $k = 1 : j - 1$ **do**
11: $prod(i) = prod(i) + log(abs(theta(i) - sorted(k)))$
12: **end for**
13: **end for**
14: $sorted(j)$ is the $theta(i)$ with the largest value of $prod(i)$
15: **if** $sorted(j)$ is complex **then**
16: $sorted(j + 1)$ = complex conjugate of $sorted(j)$
17: $j = j + 1$
18: **end if**
19: $j = j + 1$
20: **end while**
21: Return *sorted*.

4.2 Method for Generating the Polynomial $p(A)$

We first used the polynomial implementation from (4.3) in [14] to precondition nonsymmetric eigenvalue problems for the Arnoldi method. Polynomial preconditioning for linear systems presents a different set of challenges than for eigenvalue problems. For preconditioned eigenvalue problems, we only needed to apply the polynomial $Ap(A)$; we never needed to apply $p(A)$ by itself. For PP-GMRES, we need to apply $p(A)$ alone to use (4.2) and obtain a final solution at the end of the GMRES iteration. We cannot directly use the factorization of $Ap(A)$ in (4.4). In this section, we present a new factorization of the polynomial $p(\alpha)$ which can be used for

Algorithm 4.2 $Ap(A)$ times z via (4.4)

Input: sparse matrix $A \in \mathbb{R}^{n\times n}$, $z \in \mathbb{R}^{n\times 1}$, vector *theta* of d harmonic Ritz values
Output: $Ap(A)z$

$poly = z$, $i = 1$
while $i <= d$ **do**
    if $imag(theta(i)) == 0$ **then**
        $product = A * poly$
        $poly = poly - (1/theta(i)) * product$
        $i = i + 1$
    else
        $a = real(theta(i)),\ b = imag(theta(i))$
        $sum = a * a + b * b$
        $product = A * poly$
        $temp = A * product - 2 * a * product$
        $poly = poly + (1/sum) * temp$
        $i = i + 2$
    end if
end while
$poly = z - poly$
Return $poly$.

(4.2). It is important that the implementations of $Ap(A)$ and $p(A)$ be numerically compatible so that the solution obtained from (4.2) will match the preconditioned system.

To find the factored form of p, first divide the polynomial $\alpha p(\alpha) = 1 - q(\alpha)$ by α. Substituting (4.3) for $q(\alpha)$ gives

$$p(\alpha) = \frac{1}{\alpha} - \frac{1}{\alpha}\prod_{i=1}^{d}\left(1 - \frac{\alpha}{\theta_i}\right).$$

Then distribute the $-1/\alpha$ into the last term of the product and split it into two products, obtaining

$$p(\alpha) = \frac{1}{\alpha} - \frac{1}{\alpha}\prod_{i=1}^{d-1}\left(1 - \frac{\alpha}{\theta_i}\right) + \frac{1}{\theta_d}\prod_{i=1}^{d-1}\left(1 - \frac{\alpha}{\theta_i}\right).$$

Continue distributing the $-1/\alpha$ into the last term of the adjacent product and splitting the products. Cancel the remaining $1/\alpha$ terms at the end to get that

$$p(\alpha) = \sum_{k=1}^{d} \frac{1}{\theta_k} u_k \quad \text{where} \quad u_k \equiv \prod_{i=1}^{k-1} \left(1 - \frac{\alpha}{\theta_i}\right). \tag{4.6}$$

We use this factorization to implement $p(A)$ using harmonic Ritz values.

The algorithm for multiplying a vector by $p(A)$ alternates between building out the product u_k and adding u_k/θ_k to the polynomial sum. As before, for real-valued matrices, we can avoid complex arithmetic by combining complex conjugates. Suppose that all values θ_i are real for $i < k$ and then $\theta_k = a + bi$ with $\theta_{k+1} = a - bi$. The next two terms of the polynomial are

$$\begin{aligned}
\frac{1}{\theta_k} u_k &= \frac{1}{a+bi} \left(1 - \frac{\alpha}{\theta_1}\right) \left(1 - \frac{\alpha}{\theta_2}\right) \cdots \left(1 - \frac{\alpha}{\theta_{k-1}}\right) \\
\text{and} \quad \frac{1}{\theta_{k+1}} u_{k+1} &= \frac{1}{a-bi} \left(1 - \frac{\alpha}{\theta_1}\right) \left(1 - \frac{\alpha}{\theta_2}\right) \cdots \left(1 - \frac{\alpha}{\theta_{k-1}}\right) \left(1 - \frac{\alpha}{a+bi}\right) \\
&= \left(1 - \frac{\alpha}{\theta_1}\right) \left(1 - \frac{\alpha}{\theta_2}\right) \cdots \left(1 - \frac{\alpha}{\theta_{k-1}}\right) \left(\frac{1}{a-bi} - \frac{\alpha}{a^2+b^2}\right).
\end{aligned}$$

We can combine the two terms as:

$$\begin{aligned}
\frac{1}{\theta_k} u_k + \frac{1}{\theta_{k+1}} u_{k+1} &= \left(1 - \frac{\alpha}{\theta_1}\right) \left(1 - \frac{\alpha}{\theta_2}\right) \cdots \left(1 - \frac{\alpha}{\theta_{k-1}}\right) \left(\frac{1}{a+bi} + \frac{1}{a-bi} - \frac{\alpha}{a^2+b^2}\right) \\
&= \left(\frac{2a}{a^2+b^2} - \frac{\alpha}{a^2+b^2}\right) u_k.
\end{aligned} \tag{4.7}$$

Use (4.7) to add the u_k/θ_k and u_{k+1}/θ_{k+1} terms to the final sum. Then use (4.5) to compute the product u_{k+2} as

$$\begin{aligned}
u_{k+2} &= \left(1 - \frac{\alpha}{\theta_1}\right) \left(1 - \frac{\alpha}{\theta_2}\right) \cdots \left(1 - \frac{\alpha}{\theta_{k-1}}\right) \left(1 - \frac{\alpha}{a+bi}\right) \left(1 - \frac{\alpha}{a-bi}\right) \\
&= \left(1 + \frac{\alpha^2 - 2\alpha a}{a^2+b^2}\right) u_k.
\end{aligned}$$

If θ_{k+2} is real, continue building out the polynomial as usual. Otherwise, repeat the above procedure. Algorithm 4.3 details the full process for applying $p(A)$ to a vector while avoiding complex arithmetic. We use Matlab notation in that $zeros(n, 1)$ indicates a vector of length n filled with zeros.

Algorithm 4.3 $p(A)$ times z

Input: sparse matrix $A \in \mathbb{R}^{n \times n}$, $z \in \mathbb{R}^{n \times 1}$, vector *theta* of d harmonic Ritz values.
Output: $p(A)z$

$product = z$, $poly = zeros(n, 1)$, $i = 1$
while $i <= d - 1$ **do**
    if $imag(theta(i)) == 0$ **then**
        $poly = poly + (1/theta(i)) * product$
        $product = product - (1/theta(i)) * A * product$
        $i = i + 1$
    else
        $a = real(theta(i))$, $b = imag(theta(i))$
        $mod = a * a + b * b$
        $temp = 2 * a * product - A * product$
        $poly = poly + (1/mod) * temp$
        if $i <= d - 2$ **then**
            $product = product - (1/mod) * A * temp$
        end if
        $i = i + 2$
    end if
end while
if $imag(theta(d)) == 0$ **then**
    $poly = poly + (1/theta(d)) * product$
end if
Return *poly*.

Throughout this chapter, we use Algorithm 4.2 whenever possible to apply $Ap(A)$ and Algorithm 4.3 as needed to apply $p(A)$ alone. We refer to this implementation as the "roots method" for preconditioning with the GMRES polynomial. At points, we will discuss differences that arise when we choose to apply $Ap(A)$ and $p(A)$ using only Algorithm 4.3. We refer to this variation as the "one formula" implementation

of the roots method. This implementation is used for the new Trilinos polynomial preconditioning code in Chapter Five.

4.3 Cost of Computing and Applying the Polynomial

When evaluating the cost of a polynomial implementation, we can consider two measures: preconditioner setup expense and preconditioner application expense. The latter may be more important for large-scale applications because for a problem with multiple right-hand sides, the preconditioner will be reused.

Setting up the polynomial with the "roots" implementation requires d sparse matrix-vector products and several dot products for orthogonalization. This is more expensive than setting up the power basis polynomial, which needs $d-1$ SpMVs and one block dot product for the normal equations. Even so, the roots method setup cost is less than or equal to that of the other more stable methods presented in Section 3.8. Setup for the new method is much cheaper than with the Newton basis polynomial because we do not need to create a second set of basis vectors. The setup cost for the roots method is the same as with the Arnoldi basis method.

The cost of polynomial application depends on the choice of formulas. Applying $Ap(A)$ via Algorithm 4.2 always requires d matrix-vector products and d AXPYs. This is the same cost as with the power basis method from Chapter Three. The Newton and Arnoldi basis methods are both more expensive. The Newton method requires about twice the number of AXPYs. The Arnoldi method needs matrix-vector products with a tall, skinny matrix and storage of several vectors, making it the most expensive method for applying the polynomial.

As expected, applying $p(A)$ via Algorithm 4.3 requires $d-1$ SpMVs, but it uses more AXPYs than applying $Ap(A)$ via Algorithm 4.2. If r is the number of real harmonic Ritz values and c is the number of non-real harmonic Ritz values, then Algorithm 4.3 uses $2r + (3/2)c - 1$ AXPYs. Since $r + c = d$ and we could have $c = 0$, this formula may require up to two times the AXPYs of Algorithm 4.2. This gives one reason not to use "one formula" implementation of the roots method. When we only use Algorithm 4.3 once at the very end of running GMRES, the additional cost is not significant. Using Algorithm 4.3 exclusively to apply $p(A)$ in GMRES requires twice as many AXPYs as the power basis method, has about the same cost as the Newton method, and is still significantly cheaper than the Arnoldi basis method. Example 5.2.5 in the next chapter will present a comparison of a power basis, Arnoldi basis, and the one formula roots implementation for a problem in Trilinos.

4.4 Initial Experiments

All experiments in this chapter were run using Matlab version 2017a under Ubuntu version 16.04. The desktop computer has a quad-core Intel i5-6400 CPU at 3.20 GHz. Our GMRES(m) code uses modified Gram-Schmidt orthogonalization and no reorthogonalization. Experiments are run in serial, except where Matlab has automatic multithreading. Unless otherwise stated, the following hold: 1) For the right-hand side, we generate a random normal vector and scale it to have norm 1. (Thus the relative residual norm is equal to the true residual norm.) 2) We use a maximum subspace size of $m = 50$. 3) The relative residual tolerance is 1×10^{-8}.

Convergence is determined using the short residual vector for the preconditioned problem, scaled by $\|b\|_2$.

Example 4.4.1. Our first example uses the matrix bwm2000, a chemical engineering problem from Matrix Market [2]. It is nonsymmetric and has size $n = 2000$. This matrix was also used in Section 3.5, but here we directly generate a random right-hand side b instead of computing a random $\tilde{x}$ and forming $b = A\tilde{x}$. (Using $b = A\tilde{x}$ means that b will have small components in the directions of eigenvectors corresponding to small eigenvalues; these components are damped out due to multiplication by A, making a much easier problem for GMRES.)

Without preconditioning, convergence stalls at a residual norm of 8.08×10^{-2}. We test polynomial degrees 2 through 80. The first polynomial to give convergence before 50,000 iterations is the degree 9 polynomial, requiring 7,554 iterations and 68,012 SpMVs. The minimum number of matrix-vector products is 3,179 and occurs with degree 60. Figure 4.1 shows residual norm convergence in terms of matrix-vector products for a few polynomials. Table 4.1 shows more details of how the costs of convergence decrease as we raise the polynomial degree. In particular, the number of dot products improves drastically. Going from the degree 20 polynomial to the degree 60 polynomial, they improve by a factor of over 17. After degree 60, the number of matrix-vector products starts increasing, even though the iteration count is going down. The number of dot products and AXPYs also begins to increase due to the cost of creating a high-degree polynomial. Even though raising the polynomial degree has

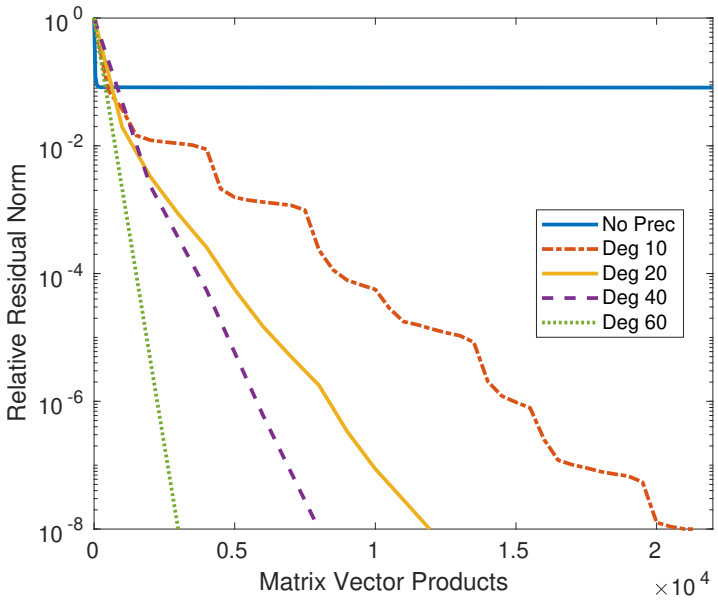


Figure 4.1: Residual norm convergence for matrix bwm2000 in terms of sparse matrix-vector products. The top line indicates no preconditioning, and the other lines use polynomial preconditioners with degrees from 10 to 60.

Table 4.1: Costs for the matrix bwm2000 with different polynomial degrees

Degree	10	20	40	60	80
Iterations	2101	599	198	50	39
SpMVs	21039	12039	8039	3179	3359
AXPYs	83014	29881	14651	6498	7580
Dots	55761	16092	6064	3217	4141

limits for improving total cost, the overall benefit from polynomial preconditioning is substantial.

We now compare the new "roots" method to the power basis method. Figure 4.2 shows the iteration counts required for convergence for increasing polynomial degrees. The dotted line shows the iteration counts for polynomials generated by the power basis method, and the solid line corresponds to the roots method. The power basis method can only generate polynomials up to degree 30; for higher degrees, it

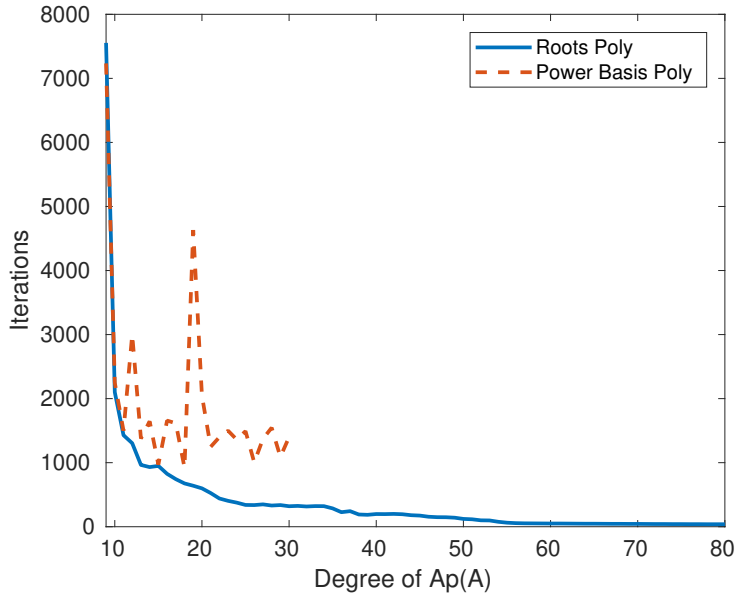


Figure 4.2: Iterations to convergence for the bwm2000 matrix, using polynomials generated by the new roots method (solid line) and by the power basis method (dotted line).

gives NaNs for coefficients. Though the "roots" and "power basis" polynomials of same degree should be identical in exact arithmetic, roundoff errors create a vast difference in their behavior. As the polynomial degree rises, the "roots" polynomials consistently reduce the iteration count, obtaining a minimum of 39 iterations. The power basis polynomials, on the other hand, give very inconsistent convergence; the iteration count goes up for several degrees, and the minimum is 923 iterations with degree 18. This is more than 23 times the minimum number of iterations needed for the roots method. Clearly the new method is inherently more stable than the power basis method. For the remainder of this chapter, we exclusively consider the roots method.

Figure 4.3 shows the eigenvalues for bwm2000 (left) and the new spectrum after a degree 60 polynomial is applied (right). Both the preconditioned and unprecondi-

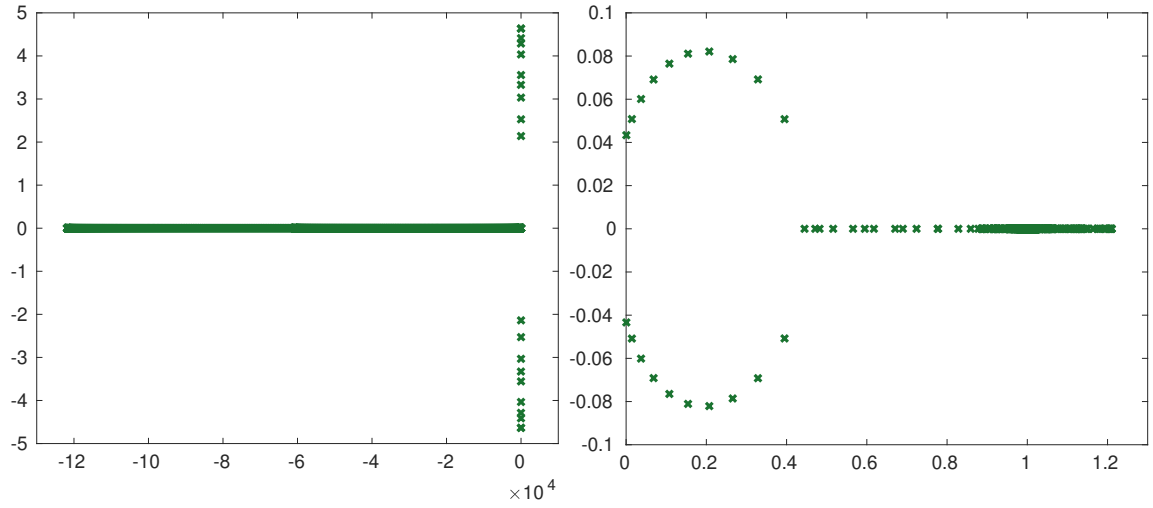


Figure 4.3: Original eigenvalues of bwm2000 (left) and eigenvalues of the preconditioned matrix with a degree 60 polynomial (right).

tioned matrices have several small non-real eigenvalues making a semicircle near the origin. The unpreconditioned matrix has eigenvalues with magnitude as small as 2.14 and as large as 1.22×10^5. For the preconditioned spectrum, the smallest eigenvalue has magnitude 0.043, but the largest eigenvalue has magnitude 1.21, so there are only two orders of magnitude difference rather than five. The degree 60 polynomial actually pushes the smaller eigenvalues closer to the origin, but it clusters the large eigenvalues so that GMRES can now deal with the problem effectively.

The experiments with bwm2000 give two confirmations that Algorithms 4.2 and 4.3 are sufficiently numerically compatible: 1) Throughout experiments with the new "roots" polynomial, the true residual norm for the preconditioned problem matches the residual norm after $\hat{x}$ is computed using (4.2). Therefore, using a different formula to apply $p(A)$ at the end of the GMRES iteration does not seem to affect the accuracy of the solution. 2) We also tested the "one formula" implementation of the roots polynomial which uses Algorithm 4.3 to apply $p(A)$ everywhere. For all polynomials

except degree 9, the iteration counts remained the same as in the original roots method implementation. Results have been similar for other problems.

These initial experiments show the new roots polynomial implementation to be very effective. The next section will discuss stability problems that may still arise and how we can fix them.

4.5 Adding Roots for Stability

Even though the new polynomial formula with roots is inherently more stable than the power basis formulation, it can be very unstable when a matrix has a large outlying eigenvalue. In the following examples, we study this phenomenon and offer a possible solution.

Example 4.5.1. We an unstable polynomial with a bidiagonal matrix of size $n =$ 10,000. The matrix has diagonal values of

$$0.05, 0.1, 0.2, \ldots, 9.9, 10, 11, 12, \ldots, 9906, 9907, 12{,}000, 20{,}000.$$

It also has values of 0.15 on the superdiagonal elements. The matrix is difficult for GMRES because of its 101 relatively small eigenvalues. It has one somewhat outlying eigenvalue at 12,000 and another extremely separated eigenvalue at 20,000. Without preconditioning, the residual norm stalls at 0.0243.

We first consider a degree 30 polynomial preconditioner. The preconditioning makes convergence worse. According to the short residual norm from (2.6), the preconditioned system converges to 1×10^{-8} in only 182 iterations. However, the true preconditioned residual $\|b - Ap(A)y\|_2$ increases and stalls out near 17.3. Figure 4.4 shows the convergence curves. For other polynomial degrees, the true residual

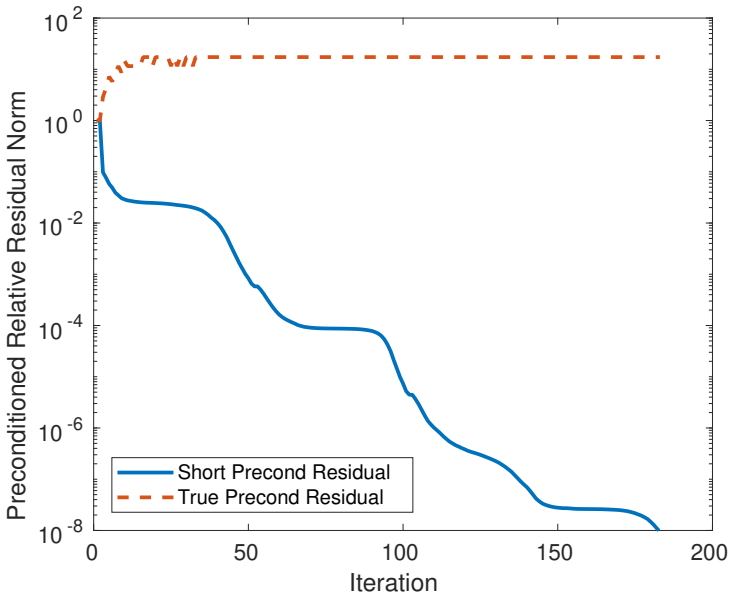


Figure 4.4: Residual norm convergence for a bidiagonal matrix with outstanding eigenvalues. The solid line indicates the residual from the small least squares problem, while the true residual for the preconditioned problem (dotted) is based on the full computed solution vector.

similarly stalls out well before convergence. The ending value of the true residual norm tends to increase as the polynomial degree increases. Ending true residual norm values for polynomial degrees divisible by five are plotted in the middle line in Figure 4.5. For degrees 45 and 50, the values plotted are deceiving- the final residual recorded was near 1, but convergence was not improving. For some iterations with degree 50, the residual norm spiked as high as 10^{23}.

Upon analyzing the degree 30 polynomial, we find that its descent is extremely steep near the outstanding eigenvalue of 20,000. (See Figure 4.6.) The slope of the tangent line at $(20{,}000, 0)$ is approximately -4.5×10^{14}. Recall that the harmonic Ritz values, the roots of the polynomial $1 - \alpha p(\alpha)$, approximate eigenvalues of the matrix A, and that they converge very quickly to well-separated eigenvalues on the

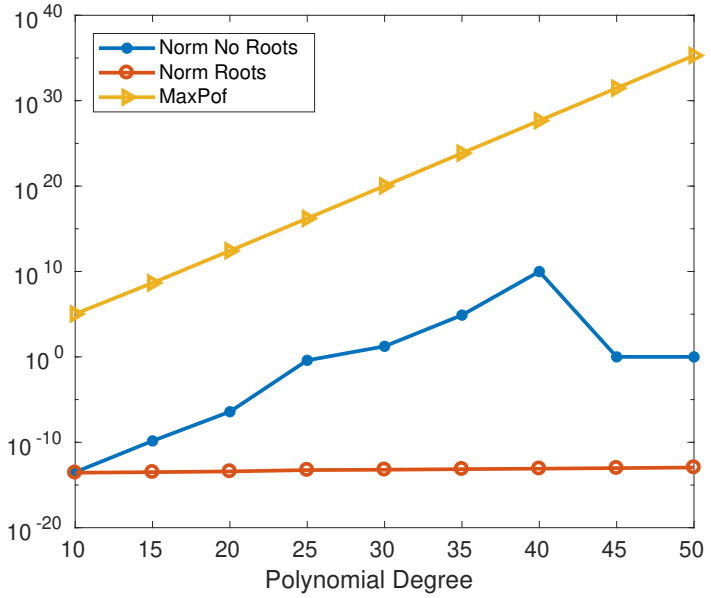


Figure 4.5: Top line: Maximum pof values for polynomials of various degrees for the bidiagonal matrix. Middle line: Point at which the true residual stalls without added roots. Bottom line: Accuracy attained for the true residual norm with added roots.

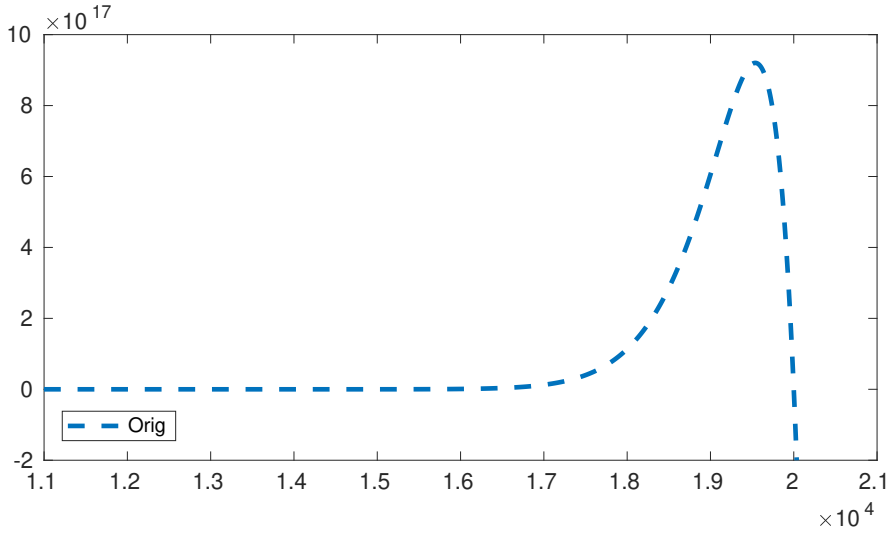


Figure 4.6: Polynomial of degree 30 for the bidiagonal matrix with outstanding eigenvalues at 12,000 and 20,000.

edge of the spectrum. Examining the harmonic Ritz values, we see that the eigenvalues of 20,000 and 12,000 have converged. The polynomial gets very big between 12,000 and 20,000 because it has no roots between these values. The steep slope of the polynomial can give significant numerical error when applying the polynomial to a vector; small error in the vector component corresponding to eigenvalue 20,000 will become large error when multiplied by the polynomial $Ap(A)$.

In order to estimate the slope of the polynomial near outlying eigenvalues, we compute a 'product of other factors' or 'pof' The value is computed for each harmonic Ritz value θ_j as

$$pof(j) \equiv \prod_{\substack{i=1 \\ i \neq j}}^{d} |1 - \theta_j/\theta_i|. \tag{4.8}$$

Define $s(\alpha) \equiv \alpha p(\alpha)$. The quantity $pof(j)$ varies with the slope of s at θ_j, since

$$|s'(\theta_j)| = pof(j)/|\theta_j|.$$

Unlike $s'(\theta_j)$, $pof(j)$ does not change with scaling of the matrix. The maximum pof values for our bidiagonal matrix for various polynomial degrees are plotted in the top line of Figure 4.5. Overall, the residual norm grows with the pof. We aim to decrease the large pof values of the polynomials in hopes that the residual norms will follow.

To resolve our stability problem, if $pof(k)$ is large, we expand $\alpha p(\alpha)$ to have extra copies of the term $(1 - \alpha/\theta_k)$, i.e. to have a root of higher multiplicity at θ_k. This flattens the polynomial near θ_k and makes the preconditioner application more stable. Experiments in [14] suggested that one should add one extra copy of root θ_k when $pof(k) > 10^4$ and another copy for every factor of 10^{14} beyond that. Algorithm 4.4 gives the full procedure to automate adding roots for stability. The Modified Leja

Algorithm 4.4 Adding roots to $\pi(\alpha)$ for stability [14]

1. **Setup:** Assume the d roots $(\theta_1, \ldots, \theta_d)$ have been computed and then sorted according to the modified Leja ordering. (Alg. 4.1)
2. **Compute** $pof(j)$**:** For $j = 1, \ldots, d$, compute $pof(j) = \prod_{i \neq j} |1 - \theta_j/\theta_i|$.
3. **Add roots:** For each j, compute the least integer greater than $(\log_{10}(pof(j)) - 4)/14$. Add that number of θ_j values to the list of roots, spacing the repeated values.

Ordering in Algorithm 4.1 is not designed to order sets with repeated values. To attempt to maintain a similar ordering, we perturb the extra roots by a multiple of some very small number, say $k(1 \times 10^{-5})$, where k is 1 for the first root copy, 2 for the second, and so on. Then we use the Modified Leja ordering to order the set of original roots with the perturbed roots. Finally, we go back and replace the perturbed roots with the unperturbed value.

For the bidiagonal matrix with a degree 30 polynomial, our algorithm says that three extra roots are needed, one for the eigenvalue at 12,000 and two for the eigenvalue at 20,000. The pof values are 1.33×10^9 at 12,000 and 1.06×10^{20} at 20,000. Figure 4.7 shows the degree 30 polynomial with one root added at 20,000 (red line) and then with all three roots added (yellow line). The added root at 20,000 forms a double root, giving the tangent line slope zero at that point. Adding another root makes for a triple root, and the polynomial becomes even flatter. The Figure 4.8 shows the polynomials with and without added roots zoomed in at the valuesα = 12,000 (left) and α = 20,000 (right). The original polynomial makes a dip before coming back to 12,000, but adding an extra root makes it flat there also.

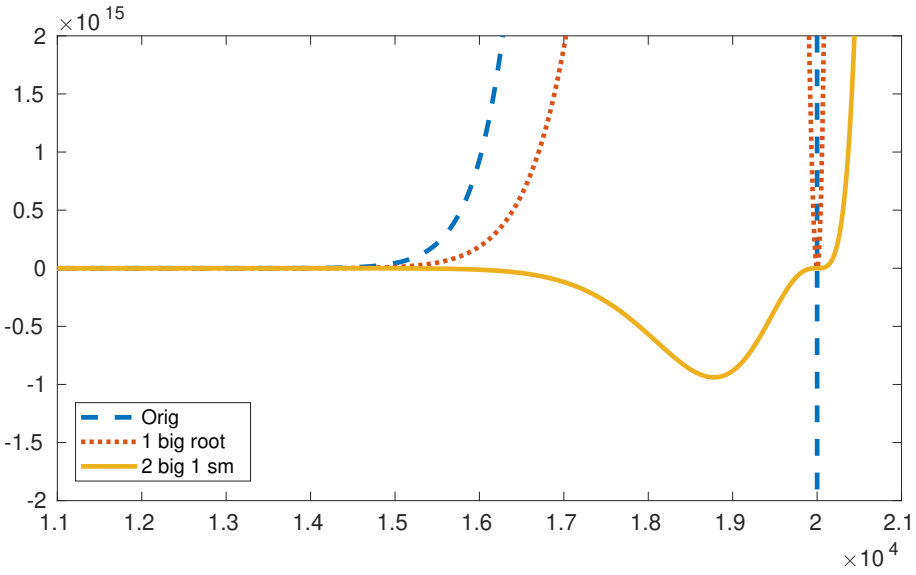


Figure 4.7: Polynomials of degree 30, with and without added roots, for the bidiagonal matrix with an outstanding eigenvalue at 20,000. "Orig" denotes the original degree 30 polynomial. The "1 big root" polynomial has one additional root at 20,000. The "2 big 1 sm" polynomial has two extra roots at 20,000 and one at 12,000.

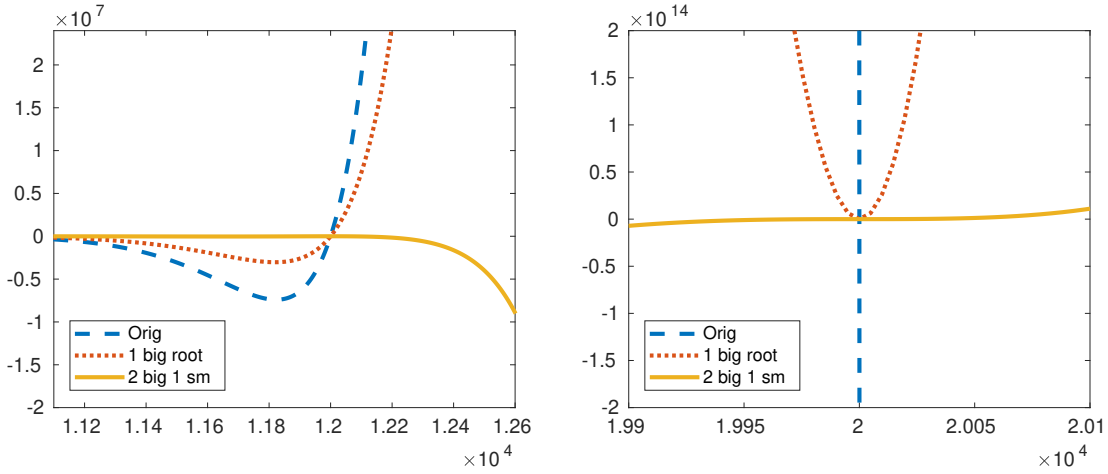


Figure 4.8: Polynomials of degree 30 for the bidiagonal matrix with and without added roots. Zoomed at 12,000 (left) and 20,000 (right).

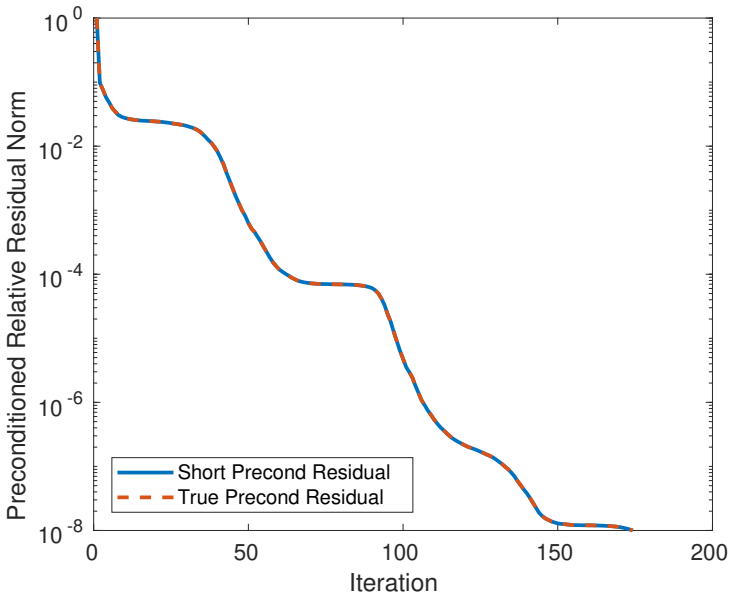


Figure 4.9: Residual norm convergence for the bidiagonal matrix with outstanding eigenvalues. Now roots are added, and the true residual matches the short residual.

Adding all three roots fixes the convergence problem for the degree 30 polynomial. The short and true preconditioned residuals now align, and the problem converges in about 175 iterations. The new convergence curves are plotted in Figure 4.9. We use Algorithm 4.4 to add roots to all the tested polynomials and push convergence as far as possible. The new attained values of the true preconditioned residual norms are plotted in the bottom line of Figure 4.5. All polynomial preconditioned problems up to degree 50 converge successfully. True residual norms end near 10×10^{-13}.

It turns out that the root-adding criteria in Algorithm 4.4 is overly cautious for the degree 30 polynomial. Adding only one root at 20,000 is sufficient for the true residual to converge. The root at 12,000 and the second root at 20,000 are not really needed. One objective for future study is to further refine the root-adding criteria so that we are less likely to incur extra matrix-vector products by adding roots unnecessarily. For

the degree 40 polynomial, Algorithm 4.4 also suggests adding one root at 12,000 and two roots at 20,000. The corresponding pof values are 8.08×10^{12} and 4.57×10^{27}, respectively. The shape of the degree 40 polynomials with and without added roots are very similar to those of the degree 30 and related polynomials. This time all three added roots are necessary for convergence. Thus, the pof root-adding criteria is not universally hyper-sensitive. A third case occurs when our root-adding criteria adds too few roots to attain convergence. This happens for polynomials of degree 70 and higher applied to the bidiagonal matrix. Added roots improve the true residuals for these degrees, but the residuals still stall before converging. We need another method to determine before running GMRES whether the added roots have made our polynomial stable. After roots are added at θ_j, the value $pof(j)$ is zero, so recomputing the pof provides no further information about whether the problem has been fixed. Our analysis in the next section will lead to a possible solution.

4.6 Error Analysis

In this section, we continue analyzing the bidiagonal matrix in Example 4.5.1 and the degree 30 polynomial that was unstable without added roots. We will give an additional perspective on the source of the error between the true and short residual norms as seen in Figure 4.4.

4.6.1 Finding the Error

First, we must point out another subtlety of the GMRES algorithm: When computing the new residual vector at the restart (Algorithm 2.2 line 14), one can either

compute it directly as

$$r_m = b - Ax_m, \tag{4.9}$$

or one can use the short residual vector and compute

$$r_m = V_{m+1}s. \tag{4.10}$$

where $s = \gamma e_1 - \overline{H}_m\hat{d}$.

In our Matlab implementation of GMRES, we employ the latter computation because it can be cheaper for very large A. This method, however, has a disadvantage. If we had computed (4.9), any components of error in the true residual would have been included in the new subspace upon the restart, and GMRES would have had a chance to remove the error in those directions. Using (4.10) maintains the error and GMRES will not correct for it. We now modify our GMRES code to use (4.9) at the restart. Then we rerun the problem from Example 4.5.1 using the degree 30 polynomial without added roots. (We also adjust the convergence tolerance to 1×10^{-12}.) The new true and short residuals for the right-preconditioned problem are shown in Figure 4.10. Though changing the residual computation gives a small improvement, GMRES still needs added roots. (Compare to Figure 4.4.)

With this change in our code, we can now discern the source of the error between the true and short preconditioned residuals. We propose that much of the error in computing the solution of the preconditioned problem results from the following loss of associativity of multiplication:

$$(Ap(A))(V_m\hat{d}) \neq (Ap(A)V_m)\hat{d}. \tag{4.11}$$

The term on the left corresponds to the true preconditioned residual computed at the end of each iteration. (To compute that residual, first we compute $x_m = x_0 + V_m\hat{d}$, and

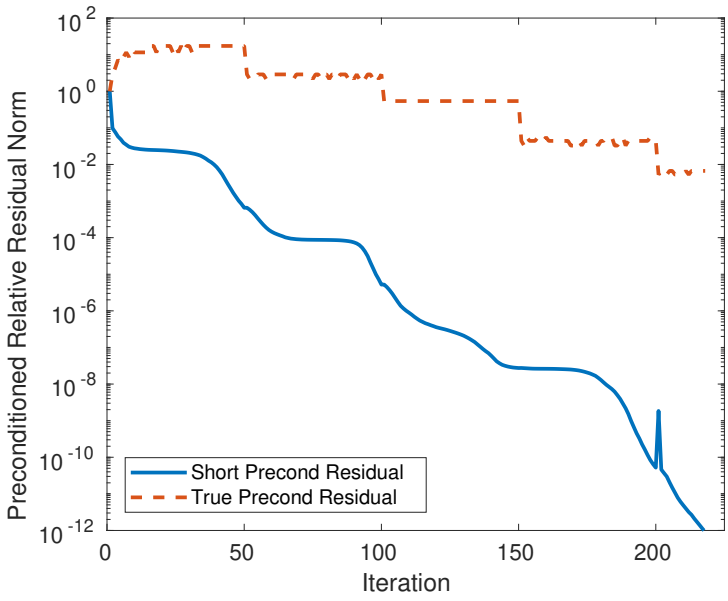


Figure 4.10: True and short preconditioned residual norms for the bidiagonal matrix with a degree 30 polynomial and no added roots. The problem is the same as in Figure 4.4, but this time GMRES uses (4.9) as the starting vector at the restart instead of (4.10).

then $\|b - Ap(A)x_m\|_2$.) For the term on the right, recall that $Ap(A)V_m = V_{m+1}\overline{H}_m$, so this term corresponds to the short residual problem which is solved to obtain $\hat{d}$.

We can monitor this error during GMRES. We begin by computing

$$StChk1 = \|(Ap(A)V_m)\hat{d} - Ap(A)(V_m\hat{d})\|_2 \tag{4.12}$$

at the end of each iteration. (Here, m denotes the size of the subspace at the current iteration.) We use Algorithm 4.2 to apply $Ap(A)$ for both terms of $StChk1$, so the error is not due to difference in polynomial applications. We maintain our earlier modification, restarting GMRES with the vector (4.9). For the first 50 iterations of GMRES, $StChk1$ matches the error in the true preconditioned residual. However, after the first restart it becomes much smaller. This is because $StChk1$ does not take into account the error in the solution x_0 carried over from previous cycles. To account

for the error in the previous solution, we monitor a new error value:

$$StChk2 = \|Ap(A)x_0 + (Ap(A)V_m)\hat{d} - Ap(A)(x_0 + V_m\hat{d})\|_2, \tag{4.13}$$

which is plotted in the top of Figure 4.11. The new error check $StChk2$ matches the error in the true residual exactly. When we add roots to stabilize the degree 30 polynomial, both error check measures fall to well below machine precision; $StChk2$ is plotted again in the bottom of Figure 4.11. Thus, we conclude that the source of instability is, indeed, the loss of associativity in (4.11).

Now that we have identified the source of the error, we ask whether there is a method to detect unstable polynomials without relying on the pof. We propose the following: After computing the Krylov basis for the polynomial, solve the small least-squares system for $\hat{d}$. Then compute

$$\tau = \|(Ap(A)V_d)\hat{d} - Ap(A)(V_d\hat{d})\|_2, \tag{4.14}$$

applying $Ap(A)$ for both terms via Algorithm 4.2. Although this will not directly correspond to the error when we begin the preconditioned GMRES iteration, it gives an estimate of how much the Krylov basis may be perturbed when the polynomial is applied. We suggest that when τ is significantly larger than machine precision, the polynomial is still unstable. A good rule of thumb is that if τ is more than one order of magnitude greater than the requested convergence tolerance, then the polynomial likely needs more roots. Figure 4.12 plots the value τ versus the value at which the true residual norm stalled for polynomials of various degrees with the bidiagonal matrix. For this example, τ aligns closely with the residual error.

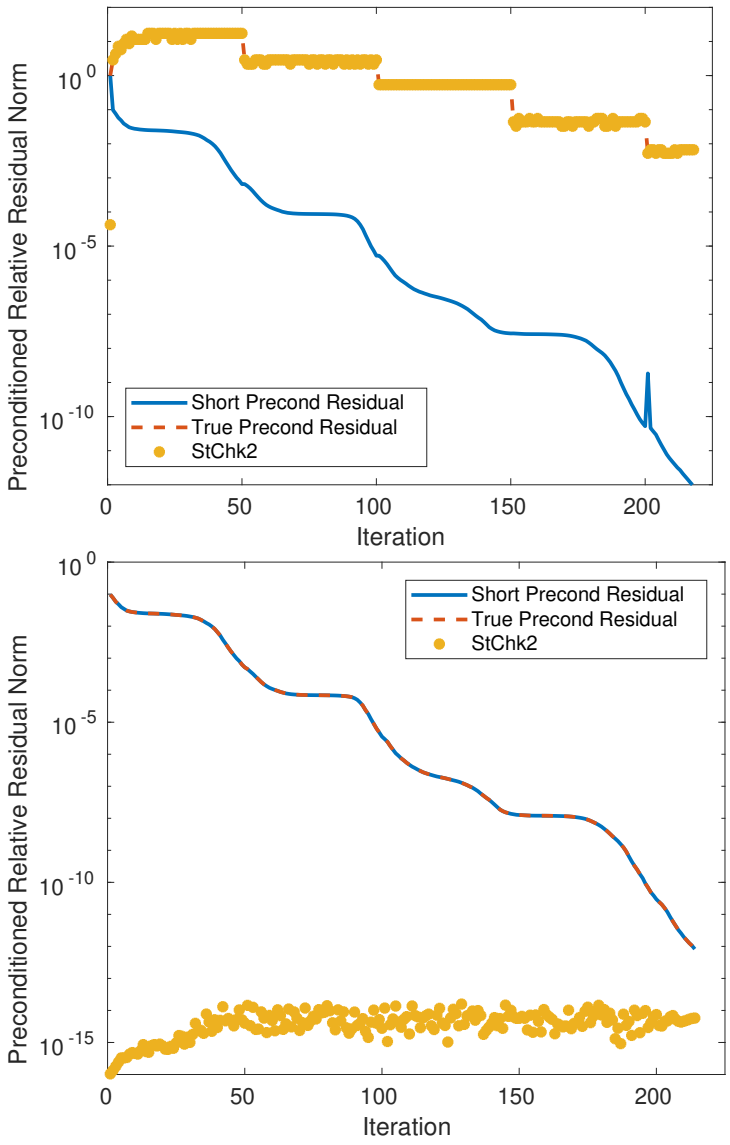


Figure 4.11: Residual norms for the bidiagonal matrix with a degree 30 polynomial preconditioner. The polynomial has no added roots for stability in the top plot and three added roots in the bottom plot. We use (4.9) for the starting vector at the restart. Furthermore, a stability check is computed according to (4.13).

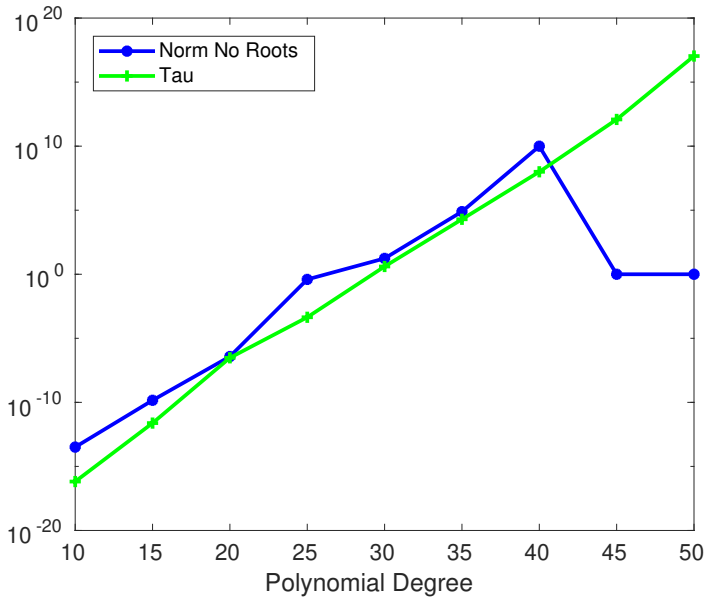


Figure 4.12: The stability check value τ versus the stalled true residual norm values for the bidiagonal matrix. The polynomials have no extra roots.

Computing τ is somewhat expensive, especially if the polynomial has high degree. Another possibility, is to compute

$$\psi = \|(b - Ap(A)b) - (b - Ap(A)b)\|_2 \tag{4.15}$$

where the application of $p(A)$ on the left term is done using Algorithm 4.3 and applying $Ap(A)$ in the right term uses Algorithm 4.2. This computation takes into account cancellation error with the residual computation and any possible numerical differences between Algorithms 4.2 and 4.3. The value ψ is often not as sensitive as τ because it doesn't consider higher order Krylov basis vectors. Even so, the computations with b can cheaply give a reasonable indication of stability. In Examples 4.7.1 and 4.7.2, we give a comparison between ψ and τ for detecting unstable polynomials. The measures τ and ψ both, in some sense, indicate the sensitivity of the basis vectors to loss of accuracy under polynomial application.

4.6.2 Relating to Other Methods

Another way to discuss the error in the residual norms is in terms of linear independence of basis vectors. It is well-known that as powers of a matrix A are applied to a vector v, the resulting vector converges to the eigenvector of A corresponding to the eigenvalue of largest magnitude. Numerically, this can cause Krylov basis vectors to lose linear independence in floating-point arithmetic where they should have been independent in exact arithmetic. This was one of the problems we encountered with the power basis implementation. We see it again for the roots implementation as we raise the polynomial degree for the bidiagonal matrix. For a degree 80 polynomial with no added roots, we actually have breakdown of GMRES (Algorithm 2.2, line 9). Unfortunately, in this case, it is not so 'lucky' because the basis vectors have lost linear independence without producing an exact solution. In fact, the resulting solution vector gives no improvement from the initial residual.

We believe this is related to the stability problems that occur in communication-avoiding (CA)-GMRES and other s-step GMRES variants. (See [23] and references therein.) In these methods, s basis vectors for GMRES are created one after another without orthogonalizing in between. The new basis vectors are then orthogonalized all at once using a Tall-Skinny QR (TSQR) or other algorithm, but sometimes they lose linear independence in the meantime. Users of s-step GMRES may employ a Newton Basis as in (3.5) to help maintain linear independence of the basis vectors. (This is similar to how our harmonic Ritz values implementation made the polynomial preconditioner more stable than with a power basis.) Considerations for choosing the

best step size s may be related to polynomial preconditioning. Future work will consider whether techniques from s-step GMRES can be applied to help identify and construct stable polynomials. Furthermore, we will consider whether root-adding techniques could help improve the stability of s-step GMRES.

4.7 Experiments with Root-Adding and Stability Checking

For all remaining Matlab experiments in this chapter, we return to computing the residual vector at the GMRES restart using (4.10).

Example 4.7.1. To further demonstrate the effectiveness and necessity of added roots, we consider the matrix Goodwin_23 from SuiteSparse, a finite element discretization of a Navier-Stokes type equation. Its size is $n = 6005$, and it has 83 eigenvalues near the origin with nonzero imaginary part. There are 369 eigenvalues at 1 which are somewhat outstanding, and the remainder of the eigenvalues lie on the real line between 0 and 0.72. We use a random right-hand side with norm 1 and request a convergence tolerance of 1×10^{-10}. GMRES(50) converges eventually with no preconditioning, requiring 122,470 iterations and 3,247,500 dot products. We consider polynomial preconditioners with degrees that are multiples of 5. The degree 5 and 10 polynomials give significant improvement, converging in 6928 and 392 iterations, respectively.

Beginning with degree 15, stability problems arise. The solid black line in Figure 4.13 shows the true preconditioned residual norms at the point when the short preconditioned residual has converged. All polynomials above degree 10, except for degrees 65 and 90, are unstable. After adding roots, all polynomial preconditioned

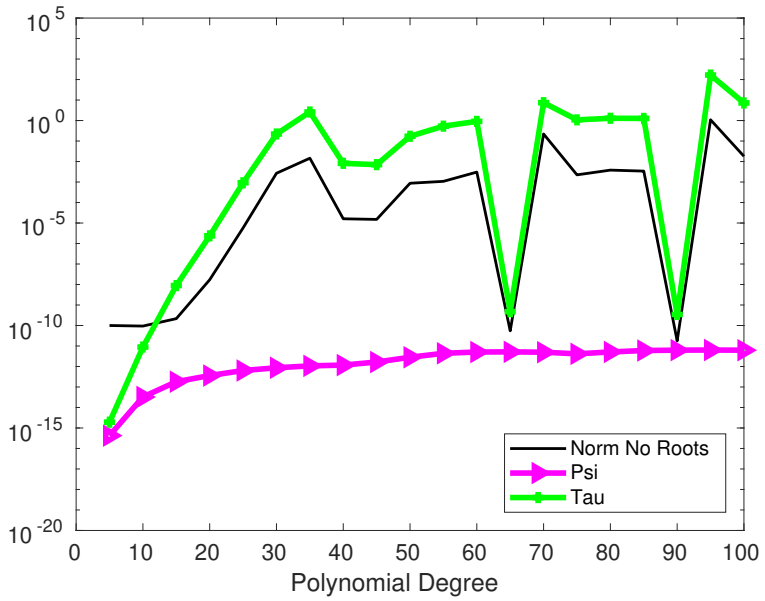


Figure 4.13: Ending true residual norms (black line) for the polynomial preconditioned matrix Goodwin_23 without added roots for stability. Also plotted are the stability check values τ (green) and ψ (pink).

problems successfully converge. Ultimately, the best solve time comes with the degree 20 polynomial, using 49 iterations, 1091 SpMVs, and 1506 dot products. Figure 4.14 shows cost counts for polynomials of degree 10 and up. These counts include cost to generate the polynomial preconditioner, so cost begins to increase with the polynomial degree, even when iteration counts are decreasing.

Figure 4.15 shows the maximum pof value for each polynomial and the number of roots added per Algorithm 4.4. We make a few observations about the added roots: 1) Here the pof values are not strictly increasing as we saw in Example 4.5.1. For this particular problem, raising the polynomial degree does not always give steeper slope at the outstanding eigenvalue. 2) All of the extra roots are added near the outstanding eigenvalue at 1, but sometimes these added roots are not exactly at the eigenvalue. For example, the degree 80 polynomial adds 1 root at 1.000000000000001,

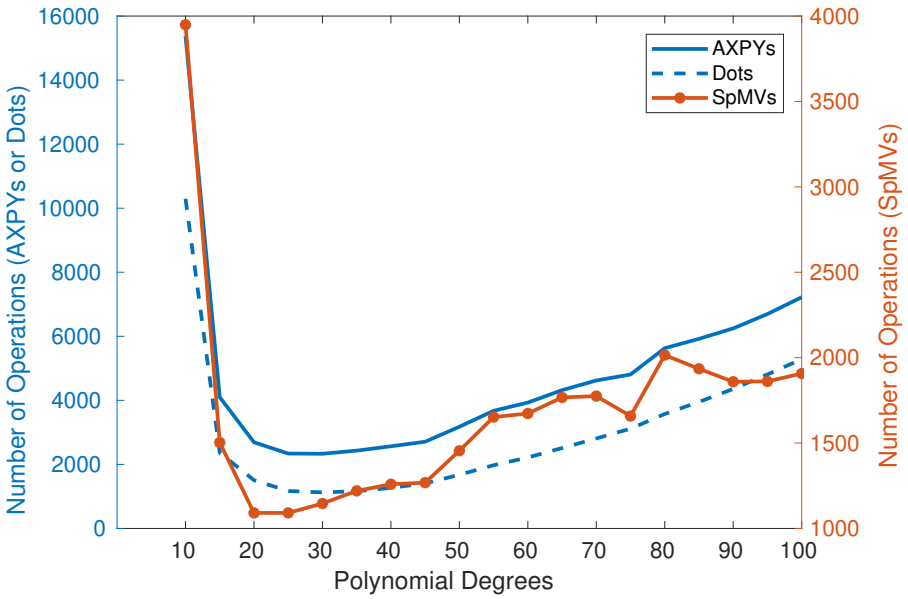


Figure 4.14: Cost for convergence for various degree polynomials applied to matrix Goodwin_23. All polynomials degree 15 and up have added roots for stability. AXPYs and dot products are represented by the left axis, and SpMVs by the right axis. The x-axis gives the degree of the polynomial before added roots.

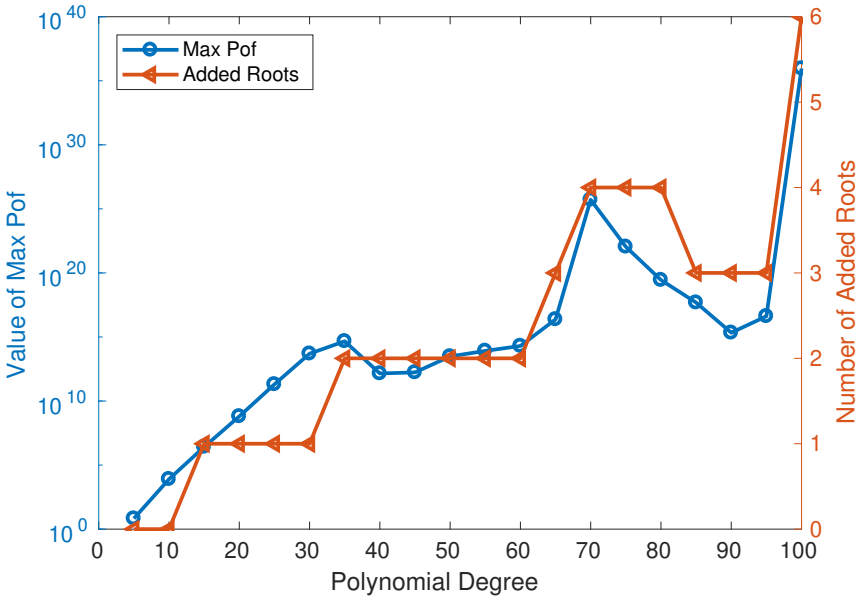


Figure 4.15: Maximum *pof* value (left axis) and number of added roots (right axis) for various polynomials applied to matrix Goodwin_23.

2 roots at 0.9999999997949012, and 1 root at 0.9999999999999996. 3) Even though the degree 65 and 90 polynomials are stable without added roots, they still have very high pof values. The maximum pof for degree 65 was 2.37×10^{16}, and the maximum pof for degree 90 was 2.15×10^{15}. Thus, Algorithm 4.4 cannot detect that the polynomials are stable and adds 3 extra roots, increasing the polynomial application expense unnecessarily.

The stability check values provide insight as to why the polynomials of degrees 65 and 90 were stable in spite of having high pof value. We compute the values τ and ψ for each polynomial before adding roots; results are plotted in Figure 4.13. For this example, τ is typically two orders of magnitude higher than the ending true residual norm, so it detects all of the unstable polynomials. For all of the stable polynomials, τ is less than 5×10^{-10}. Thus, in spite of the high pof value, it seems that the polynomials of degrees 65 and 90 do not cause large perturbations in Krylov basis vectors.

The τ values correspond closely with polynomial stability. After adding roots via Algorithm 4.4, τ values for all polynomials drop to 7.11×10^{-10} or less. The value ψ, on the other hand, is completely unhelpful in detecting unstable polynomials for this matrix. Even without added roots, its maximum value is 6.36×10^{-12}. To further investigate, we re-ran the results using the "one formula" polynomial implementation with only Algorithm 4.3 for $p(A)$. For all of the unstable polynomials, the true residual stalls at exactly the same value. Thus, the choice of polynomial application algorithm seems to have no impact on stability, which may explain why the ψ values were unhelpful for this example.

Example 4.7.2. The next matrix is OLM1000, a hydrodynamics problem from Matrix Market. We combine polynomial preconditioning with an ILU(0) preconditioner. The ILU preconditioner is very effective, converging in 55 iterations and 0.03 seconds, so the problem does not need polynomial preconditioning. Nevertheless, it makes an interesting test case for root-adding because the ILU preconditioned problem has one very outstanding eigenvalue at 16.31. We request a convergence tolerance of 1×10^{-15} for the short residual to push the limits of accuracy, and we test polynomials of degrees 2 up to 45.

First, we consider the case with no added roots. For degrees 2 through 22, the short preconditioned residual reaches the requested tolerance, but for polynomials of higher degree, the short residual stalls out near 0.9. The ending values of the true residuals are much higher; see the solid black line in Figure 4.16. Beginning with degree 7, the polynomials are unstable, and the true residuals cannot attain a tolerance of 1×10^{-10}. Also plotted in Figure 4.16 are the values of the stability checks τ and ψ. At first, the τ values align closely with the stalled true residuals. Then, beginning with degree 24, the true residuals hover near 6×10^{11}, while the τ values continue to grow rapidly. Unlike the last example, the ψ values are also helpful in detecting unstable polynomials, but they lag behind the τ values and do no detect instability until about degree 16.

Adding roots via Algorithm 4.4 is very helpful for most of the polynomials. See the new ending true residuals and stability checks plotted in Figure 4.17. This time, the short residuals converge up to degree 43. The true residuals, with the exception of degrees 24 to 26, attain a tolerance of 1×10^{-10} up through degree 31. This time, the

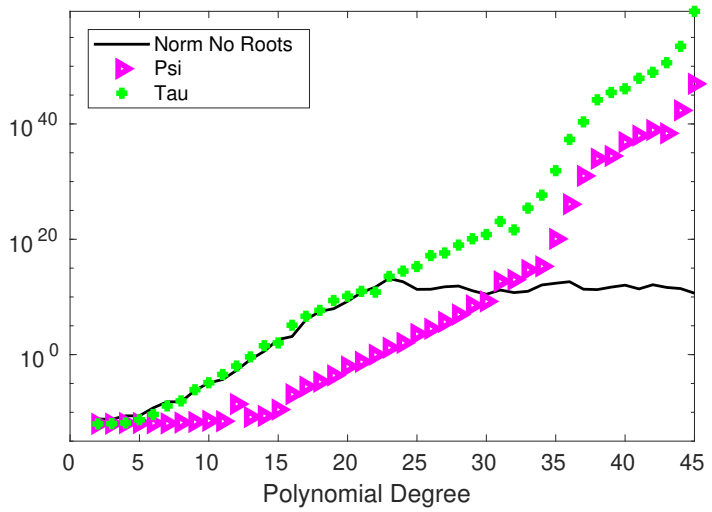


Figure 4.16: Polynomials of various degrees are applied to the matrix OLM1000 along with ILU(0) preconditioning. No extra roots are added for stability. The solid black line indicates at what value the true preconditioned residual norm stalls. Two stability checks are indicated by crosses (for τ) and triangles (for ψ).

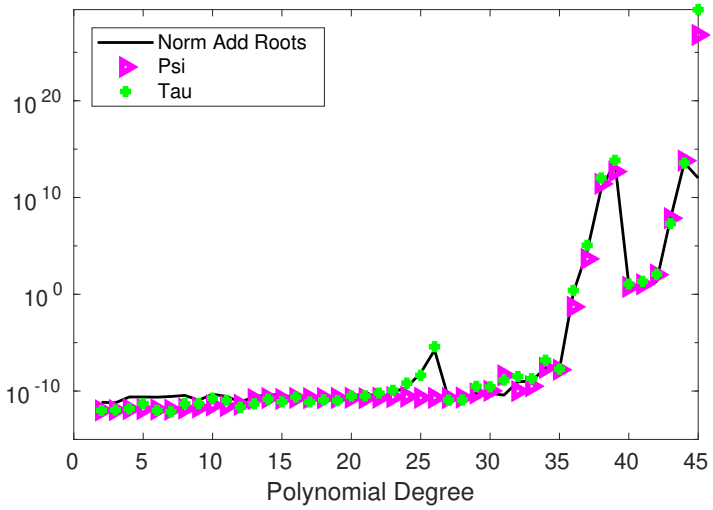


Figure 4.17: Polynomials of various degrees with added roots for stability are applied to the matrix OLM1000 along with ILU(0) preconditioning. The solid black line indicates at what value the true preconditioned residual norm stalls. Two stability checks are indicated by crosses (for τ) and triangles (for ψ).

two stability checks perform almost equally in detecting good and bad polynomials. The one exception is that τ detects the unstable polynomials from degree 24 to 26, while ψ does not.

The significant values of ψ for this problem lead us to speculate that there may be numerical differences in applying $Ap(A)$ using Algorithm 4.2 versus Algorithm 4.3. Thus, we ran tests (with no added roots) using only Algorithm 4.3 to see if this improved the stability. Of the 45 polynomials tested, 50% of the true residuals were smaller with the original algorithm, and 50% were better with the "one formula" polynomial implementation. The results were truly a toss-up. Though choice of algorithm for applying $Ap(A)$ did make a difference in convergence, the difference between Algorithms 4.2 and 4.3 does not seem to be the source of instability.

4.7.1 More Observations on Stability Checking

The previous three examples lead us to make a few broad recommendations for stability checking. Since root-adding is generally successful and the cost from unnecessary roots is usually minimal, we recommend always adding roots as per Algorithm 4.4 based on the pof values. Then, since ψ is much less expensive to compute than τ, use it to determine whether is still unstable after added roots. Compute τ only for small polynomial degrees or when certainty that a polynomial is stable is worth the large overhead. (This might be the case for a difficult matrix that has multiple right-hand sides to solve.)

The close alignment with τ and the stalled residual norms supports the evidence from Section 4.6 that our stability problems result from loss of information when

applying the polynomial to Krylov basis vectors. However, there are several areas where stability analysis of polynomials could benefit from further research. Because numerics can vary with different processors and underlying code libraries, it will be important to test these stability checks on other architectures to verify that they have the same predictive behavior. We also need to consider which algorithm for $Ap(A)$ is best for computing the stability check value τ. Some experiments have suggested that computing τ via Algorithm 4.3 is more sensitive than using Algorithm 4.2. More investigation is needed here to determine if one method gives more information than the other.

Finally, the previous three examples studying stability checks have a significant limitation: the random vector used to generate the polynomial was the same as the random vector for the problem right-hand side. Further testing is needed to see if comparable results will hold when the polynomial is generated by a vector that is unrelated to the problem right-hand side. (Initial experiments with the bidiagonal matrix suggest that results will be similar.) The next section demonstrates that, even with more stable roots polynomial implementation, the choice of starting vector for generating polynomials can be crucial.

4.8 The Starting Vector for the Polynomial

When generating a polynomial preconditioner, it is typically best to compute the Arnoldi recurrence for the harmonic Ritz values using a random starting vector rather than the problem right-hand side. The following experiment demonstrates

how polynomials generated using the problem right-hand side might ignore certain eigenvalues and give bad preconditioners.

Example 4.8.1. We consider the electronic circuit matrix Memplus and its corresponding right-hand side available on Matrix Market. The matrix A is of size $n = 17{,}758$. We let b_{prob} denote the problem right-hand side and b_{rand} denote a vector generated from a random normal $(0, 1)$ distribution. We generate three polynomial preconditioners of degree $d = 15$. The first is created by running GMRES(d) with b_{rand} as a starting vector and the second by using b_{prob} as a starting vector. For the second polynomial, additional roots are needed for stability, so our third polynomial is generated with b_{prob} as a starting vector and the four roots are added. Figure 4.18 shows residual norm convergence for the unpreconditioned problem and the three preconditioned problems, using GMRES(50) with a requested tolerance of 1×10^{-10}. While the problem does converge in 41 cycles without a preconditioner, the b_{rand} polynomial gives a 90% decrease in AXPYs and a 94% decrease in dot products. This comes with a 1% increase in SpMVs. However, the polynomials generated with b_{prob} stall GMRES convergence and make the problem much worse than with no preconditioning.

This inconsistent preconditioner behavior can be better explained by considering how the polynomials remap the eigenvalues of A. All eigenvalues of A lie in the right half of the complex plane. The two non-real eigenvalues have very small magnitude and were minimally affected by the polynomial preconditioners. Of the real eigenvalues, four lie near 1.5, and the remainder have magnitude less than or equal to 0.5.

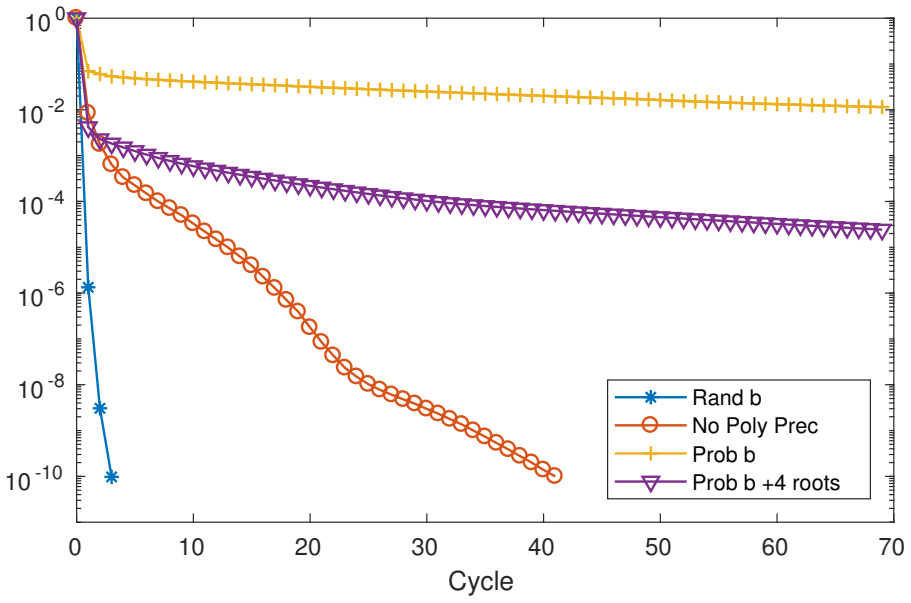
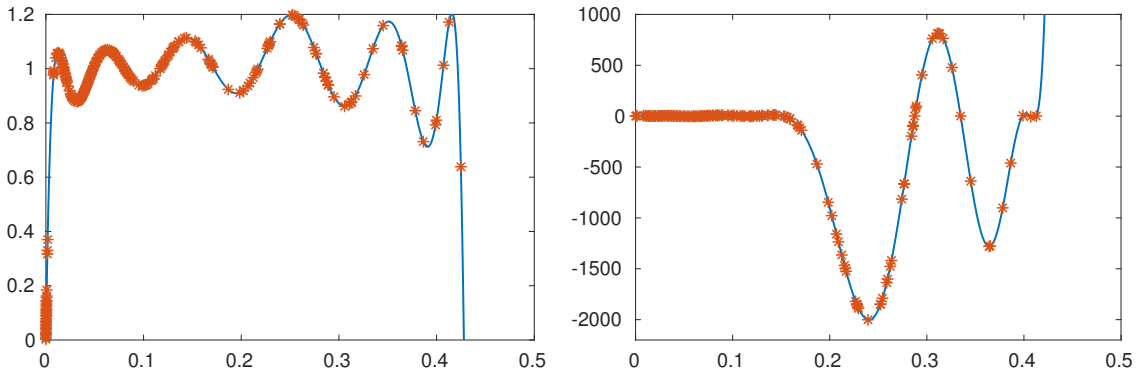


Figure 4.18: Relative residual convergence vs number of cycles for GMRES(50) on the Memplus matrix. Circles indicate no polynomial preconditioning. The preconditioned problems are indicated by: asterisks for the b_{rand} polynomial, plus for the b_{prob} polynomial, and triangles for the polynomial with added roots. All polynomials have degree 15 before adding roots for stability, if needed.

Figure 4.19 shows the effect of the first two polynomial preconditioners on this subset of real eigenvalues. (Though the spectrum is slightly complex, the polynomial is only graphed on the real axis.) While the b_{rand} polynomial effectively maps most of the larger eigenvalues to near 1, the b_{prob} polynomial neglects these eigenvalues. It creates a more difficult spectrum by making the problem highly indefinite. Outside of this figure, the b_{rand} polynomial moves the eigenvalues of magnitude 1.5 closer to 1, but the b_{prob} polynomial effectively ignores those eigenvalues, mapping them to near 10^6. In Figure 4.20, a cumulative distribution function provides further insight into the eigenvalue distributions of the preconditioned operators.

Upon further examination of the vector b_{prob}, it appears that b_{prob} has components of significant magnitude in the direction of only a handful of the eigenvectors of A.



(a) Polynomial generated with b_{rand} (b) Polynomial generated with b_{prob}

Figure 4.19: Polynomials of degree 15 plotted over the real axis on $[0, 0.5]$. Stars indicate eigenvalues of the Memplus matrix (horizontal axis) mapped to the eigenvalues of the preconditioned matrix (vertical axis). Observe the large difference in scaling between the two vertical axes.

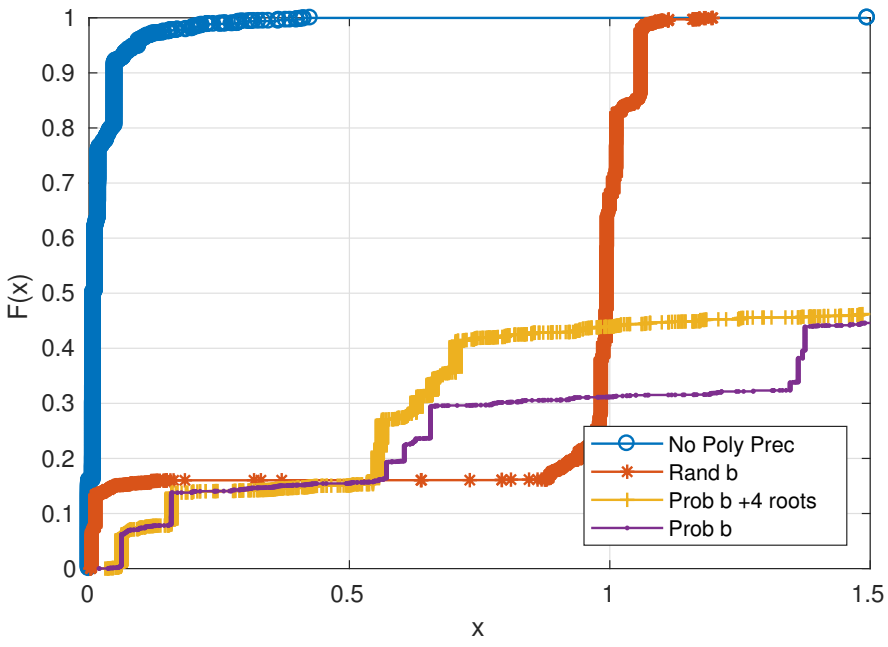


Figure 4.20: Cumulative distribution of absolute values of eigenvalues for the preconditioned and non-preconditioned Memplus matrix. Vertical axis shows the probability that an eigenvalue has magnitude less than or equal to a given value on the x-axis. The line with circles shows how the unpreconditioned eigenvalues are mostly clustered near zero. The polynomial generated by b_{rand} (*s) clusters more eigenvalues near 1. The polynomials generated by b_{prob} uncluster the eigenvalues and create a difficult spectrum.

The components of b_{rand} are more evenly distributed among the eigenvectors. This explains why the b_{prob} polynomial ignored many eigenvalues while the b_{rand} polynomial adapted for all of them. Experiments in generating polynomials with other vectors yielded comparable results: A uniformly distributed random vector worked almost as well as the b_{rand} vector, but a starting vector of all ones again stalled convergence (though not quite as badly as b_{prob}). This phenomenon has been observed in a number of other matrix problems; thus, we recommend always using a random vector to generate the polynomial preconditioner. If there is concern that a particular random starting vector will not generate an effective polynomial, it is possible to use two starting vectors to generate the polynomial; see [14]. There are some instances when a random starting vector cannot guarantee a well-behaved polynomial. In the next section, we show other ways that ill-behaved polynomials can manifest and present a possible solution.

4.9 Damped Polynomials

We now introduce damping. Damping can reduce erratic behavior and oscillations in minimum residual polynomials. It can also be useful for indefinite problems. To create a damped polynomial, first generate a random vector v_0, as usual. Then, instead of running GMRES(d) with vector v_0, use the starting vector Av_0. This creates the RRGMRES [9] subspace $\mathcal{K}_d(A, Av_0)$ rather than the usual GMRES subspace. The starting vector Av_0 is "damped" in the sense that the components of v_0 in the directions of eigenvectors with small eigenvalues are muted. After calculating the matrix H_d from the Arnoldi recurrence, solve for the harmonic Ritz values and use

them to apply $p(A)$ as usual. This technique seems contrary to the advice in Section 4.8, since small eigenvalues are ignored, but ignoring small eigenvalues produces a very different outcome from neglecting the large ones. In a number of special cases, it can be extremely helpful. In general, a damped polynomial that works less to separate small eigenvalues will be a less effective preconditioner, but we will only use this technique to correct polynomials that are initially ineffective.

4.9.1 Overenthusiastic Polynomials

We begin with the matrix s1rmq4m1 from Matrix Market. This matrix comes from a finite element problem. It has size 5489 and is symmetric positive definite. Its condition number estimate is 3.21×10^6. The problem right-hand side is generated random normal and then normed to one. We run GMRES(50) to a tolerance of 1×10^{-8}. This matrix is very difficult in spite of being symmetric positive definite. Its eigenvalues lie in the interval $(0, 7 \times 10^5)$, but they are far from being evenly spaced. Almost 50% of the eigenvalues are less than 1000, so they lie in the first 0.15% of the interval. Without preconditioning, the problem requires 5141 cycles, 257,009 SpMVs, and 477 seconds to converge.

We test polynomial degrees divisible by 5. Polynomials of requested degree 55 and higher each used 1 to 3 extra roots for stability, per Algorithm 4.4. The blue line with asterisks in Figure 4.21 gives the solve times for the un-damped polynomials. Cost in terms of SpMVs, AXPYs and dot products varied in proportion to solve time. At first the polynomial preconditioning seems to work well. The polynomial of degree 25 reduces solve time to 10.5 seconds, with 20,949 SpMVs. However, the degree 30

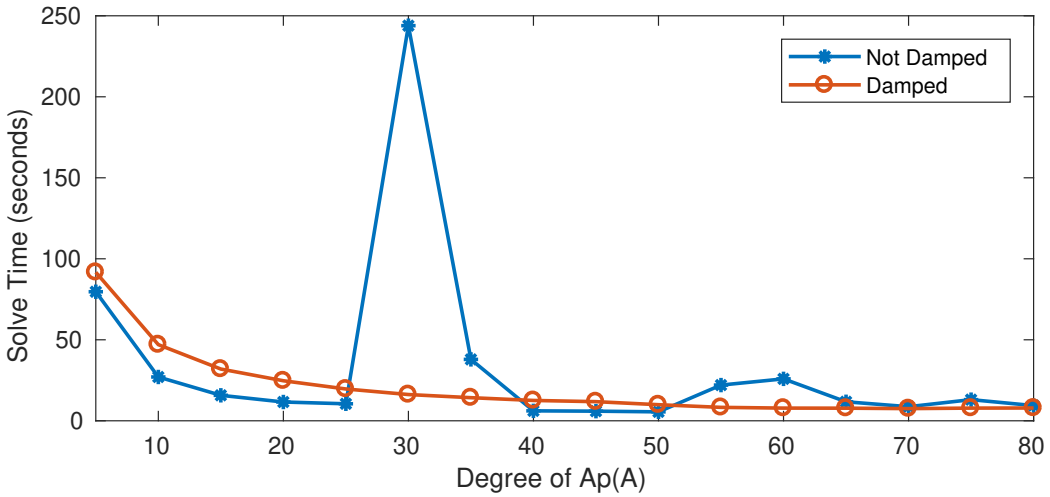
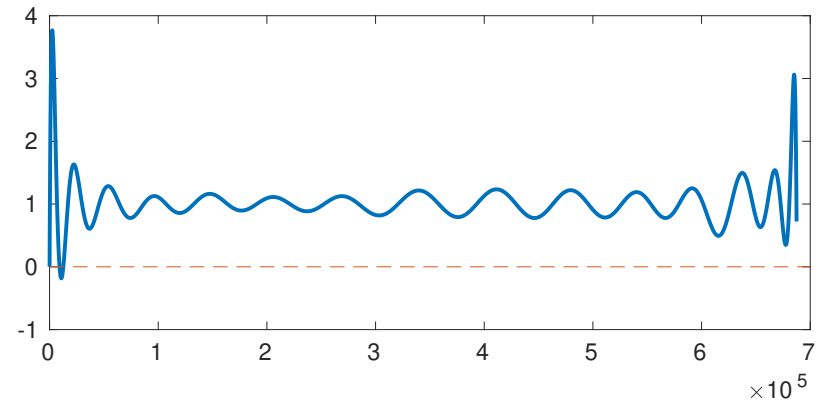


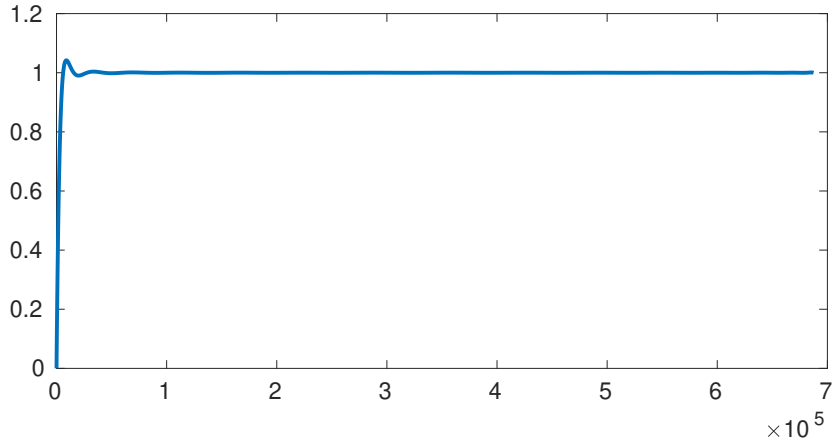
Figure 4.21: Solve times for various polynomial degrees for s1rmq4m1 with and without damping

polynomial goes bad: It needs 244 seconds and 503,880 SpMVs to converge. The degree 30 polynomial is pictured in Figure 4.22a. It is plotted along the interval containing the spectrum of s1rmq4m1. Notice that the polynomial has a steep slope at the beginning and then turns back to dip below zero. The polynomial needs such a steep slope to separate the many small eigenvalues that it cannot correct fast enough for the larger eigenvalues. In [14] these polynomials are called "overenthusiastic." They work so hard on the tightly clustered small eigenvalues that they cannot help but create a difficult spectrum elsewhere.

The degree 30 polynomial is problematic because it turns the SPD problem into an indefinite problem; it maps 22 eigenvalues to less than zero. Indefinite problems can be extremely difficult for GMRES. Other polynomials of different degree also create indefinite spectra. The degree 25 polynomial still performs well since it only maps one eigenvalue below zero. The polynomials of degrees 60 and 35 each map 15 eigenvalues below zero and perform badly compared to the other polynomials. The



(a) Degree 30 polynomial



(b) Degree 30 polynomial damped

Figure 4.22: Damped and un-damped polynomials of degree 30 for the matrix s1rmq4m1, plotted over its spectrum. The un-damped polynomial of degree 30 dips below zero.

polynomials of degrees $55, 65, 70, 75$, and 80 also make the problem indefinite. They each map between 7 and 12 eigenvalues below zero.

Next we try damping the polynomials. None of the damped polynomials need added roots for stability. Solve times are given with the orange line and open circles in Figure 4.21. The damped polynomial of degree 30 is plotted in Figure 4.22b. This polynomial has no oscillations of high magnitude (no "overenthusiastic" behavior) and does not make the spectrum indefinite. Solve times reflect this behavior. The

degree 30 damped problem needs only 16 seconds and 33,899 SpMVs to converge, almost 15 times less than the expense of the un-damped degree 30 polynomial.

Ultimately, the un-damped polynomial of degree 50 is the best, with a solve time of 5.5 seconds and 12,199 SpMVs. It is able to have a steep slope at the small eigenvalues without over-correcting. However, the success of un-damped polynomials as the degree increases is somewhat unpredictable. In addition, the effectiveness of un-damped polynomials varies greatly with the choice of random vector v_0. For instance, if we re-create our un-damped polynomials with a different random number seed for v_0, the degree 30 polynomial now only maps 1 eigenvalue below zero, but the degree 50 polynomial, which was previously best, now maps 10 eigenvalues to below zero. Using the damped polynomials presents less of a gamble: the solve time decreases consistently as the degree increases, reaching a minimum at 7.5 seconds with degree 70. If a sequence of matrices was prone to generating overenthusiastic polynomials and one did not have time to select the best degree for each matrix, the damped polynomials might give more consistent performance. If damping once isn't enough, one can damp again by using higher powers of A when creating the polynomial generating vector.

While the damped polynomials with starting vector Av_0 do work better in several instances, the ideal solution may be to 'partially' damp the polynomial. For eigenvalue problems, [14] suggests creating a partially damped polynomial using the starting vector $Av_0 + \alpha v_0$ where α is a constant, perhaps near 1000. Then the polynomial can be adjusted via the parameter α to avoid dipping below zero while maintaining the separation of the smallest eigenvalues as much as possible. This should give better

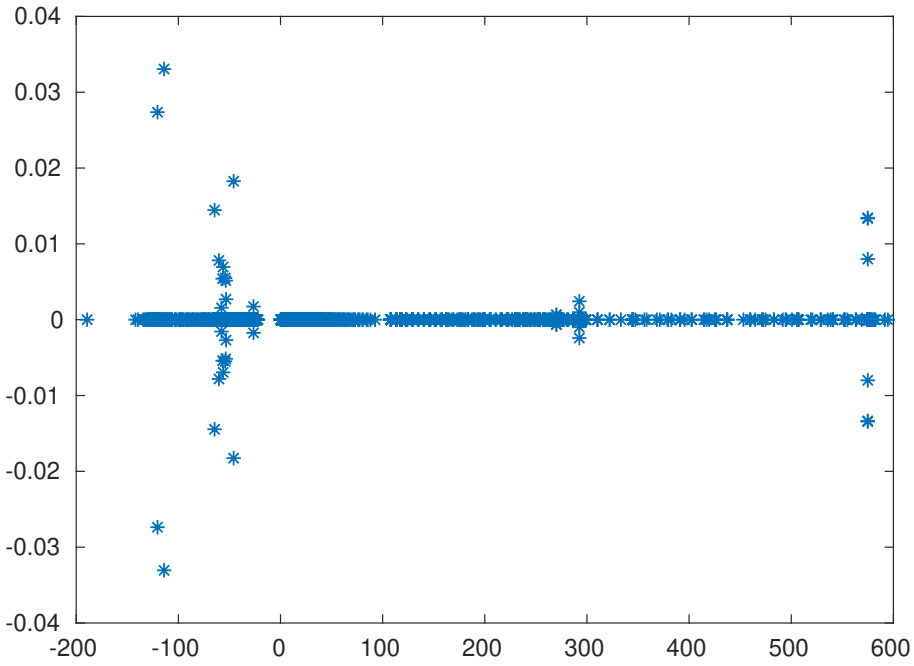


Figure 4.23: Eigenvalues of the matrix Sherman5.

solve times than the original damped polynomial. Unlike with the eigenvalue problems in [14], for linear equations it does not matter if the preconditioner changes the order of the eigenvalues, so long as the problem does not become indefinite. We need further investigation to see if overenthusiastic polynomials can be detected automatically and 'partially damped' effectively.

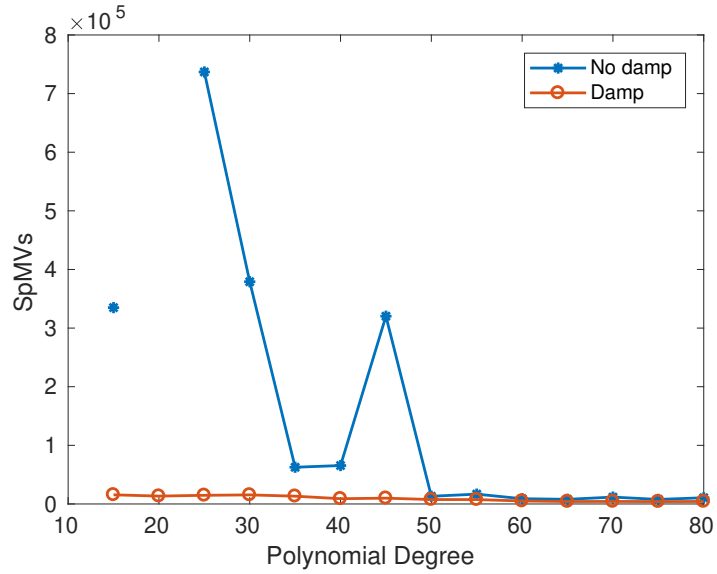
4.9.2 Damped Polynomials for Indefinite Problems

The next matrix is Sherman5, also from Matrix Market. This matrix comes from an oil reservoir simulation and has size $n = 3312$. It is nonsymmetric and indefinite, with 546 eigenvalues (16%) in the left half of the complex plane. The spectrum of Sherman5 is plotted in Figure 4.23. The end of Section 3.5 suggested that polynomial preconditioning might be useful for this difficult indefinite problem. Here we investigate further. All polynomial preconditioners in this section use added

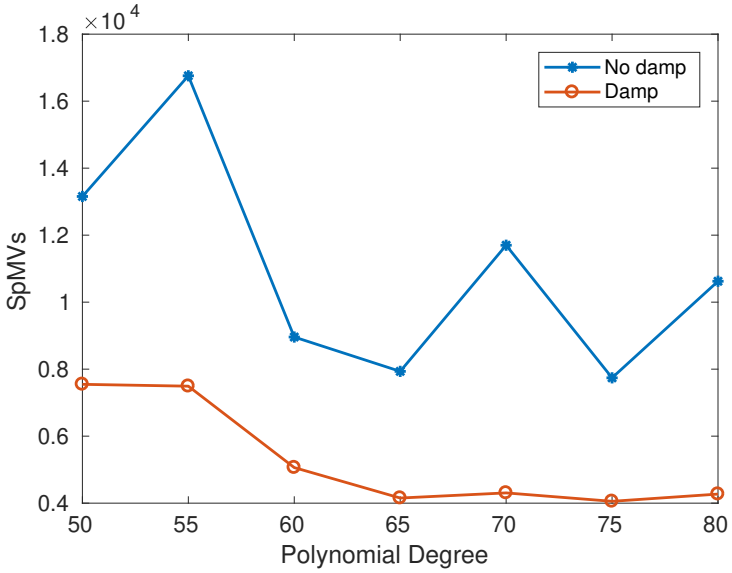
roots for stability as needed, determined by Algorithm 4.4. The polynomial degree indicates the requested degree of $Ap(A)$ before any extra roots are added. We consider polynomial degrees divisible by 5.

GMRES(50) does converge to a tolerance of 1×10^{-8} without any preconditioning. It needs 25,649 SpMVs and about 21 seconds to converge. Adding an un-damped polynomial of degree $5, 10$ or 20 makes the problem worse; it no longer converges. Higher degree polynomials, however, start to give improvement. Next, we apply damping. The damped polynomials of degrees 5 and 10 also fail, and the problem does not converge. Beginning with degree 15, the damped polynomials consistently outperform the un-damped polynomials. See a comparison of matrix-vector products for both damped and un-damped polynomials in Figure 4.24.

To determine why the damped polynomials are better, we begin by considering the polynomials of degree 20. This was the only degree where the damped polynomial was successful and the un-damped polynomial made convergence stall. When we look at the spectra of both preconditioned operators (Figure 4.25), we see that the un-damped preconditioned spectrum is still indefinite, with 31 eigenvalues on the left half of the complex plane. The spectrum for the damped polynomial is contained in the right half of the complex plane. Since indefinite spectra are particularly difficult for GMRES, this explains why the damped polynomial gave the best convergence. It turns out that all of the un-damped polynomials maintain the indefiniteness of the problem, while the damped polynomials, starting with degree 15, move all eigenvalues to the right-half plane. The least indefinite un-damped polynomial is degree 80, which leaves only 3 eigenvalues in the left half plane. One might expect that a higher degree

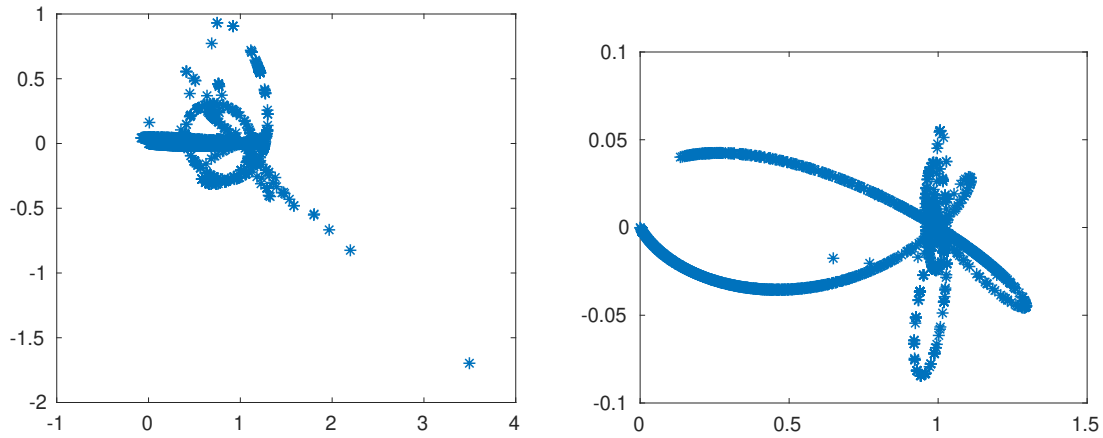


(a) SpMVs for Sherman5



(b) Close-up view of above

Figure 4.24: SpMVs for Sherman5 for multiple polynomial degrees, damped and undamped.



(a) Eigenvalues remapped by degree 20 polynomial

(b) Eigenvalues remapped by damped degree 20 polynomial

Figure 4.25: Eigenvalues of the Sherman5 matrix as remapped by polynomials of degree 20.

polynomial could overcome the indefiniteness completely, but raising the degree to 85 makes the problem worse again, with 33 eigenvalues in the left-half plane. The damped polynomials (of high enough degree) conquer the indefinite spectrum where the un-damped polynomials do not.

Damped polynomials may also be more stable than un-damped polynomials. Most polynomials (damped and un-damped) of degree 20 and above needed between 1 and 4 extra roots. The un-damped polynomials of degrees 70 and 80 needed 8 and 5 added roots respectively, and the damped polynomial of degree 80 needed 9 extra roots. With the exception of degrees 55 and 80, the damped polynomials needed fewer added roots for stability than the un-damped polynomials of corresponding degree. Given the preconditioned spectra of the degree 20 polynomials in Figure 4.25, this is not surprising: the spectrum for the un-damped polynomial has one very

outstanding eigenvalue near $3.5 - 1.7i$, and the damped spectra has no outstanding eigenvalues. The other preconditioned spectra may be similar.

Damping polynomials may also help with a different manifestation of stability problems. Even when a problem converges to the requested tolerance, we still may detect tiny increases in the residual norm during the GMRES iteration. (This should not happen in exact arithmetic since GMRES minimizes the residual over an ever-expanding subspace.) For Sherman5, these tiny residual increases occurred for all polynomials up to degree 45, even with the stability control of added roots. For the damped polynomials, on the other hand, there were no residual norm increases for degrees 15 and higher.

To conclude, we note that not all indefinite matrices will require damped polynomials. For instance, the matrix e20r0100 from Chapter Three is indefinite, and un-damped polynomials still worked well. The next section shows how discernment is required when choosing whether to damp a polynomial.

4.9.3 Why Damping is Not Universally Beneficial

After reading the previous two examples, one might be tempted to conclude that it is better to be safe than sorry and always use damped polynomials. The following example demonstrates why this strategy will often result in greater computational cost. The example is a Laplacian of size $n = 40{,}401$, the same matrix and right-hand side as in Example 3.6.1. We use a random right-hand side vector to generate our polynomial preconditioners. As before, we run GMRES(50) to a tolerance of 1×10^{-8}. A cost comparison of SpMVs for polynomials with degrees divisible by

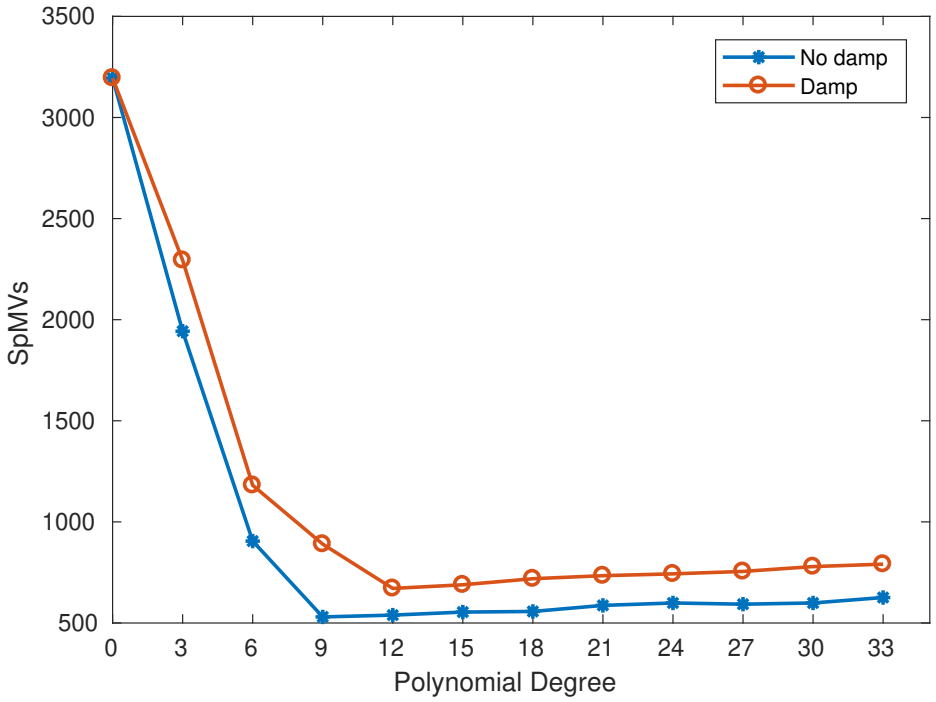


Figure 4.26: Laplacian SpMVs with and without Damping

3, with and without damping, is given in Figure 4.26. No polynomials need added roots for stability. The unpreconditioned problem (indicated by degree 0) converges in 3195 SpMVs and 19.5 seconds. While the SpMVs start to increase after degree 9 (un-damped) and degree 12 (damped), the total solve time continues to decrease as polynomial degrees increase. The un-damped timings reach a minimum at 0.456 seconds with degree 30, and the minimum solve timed for damped polynomials is 0.584 seconds with degree 27.

The un-damped polynomials consistently outperform the damped polynomials in SpMVs and solve time. To explain this behavior, we consider the damped and un-damped polynomials of degree 6, plotted over the spectrum of A (Figure 4.27). Both polynomials effectively map the bulk of the spectrum to near 1, but the un-

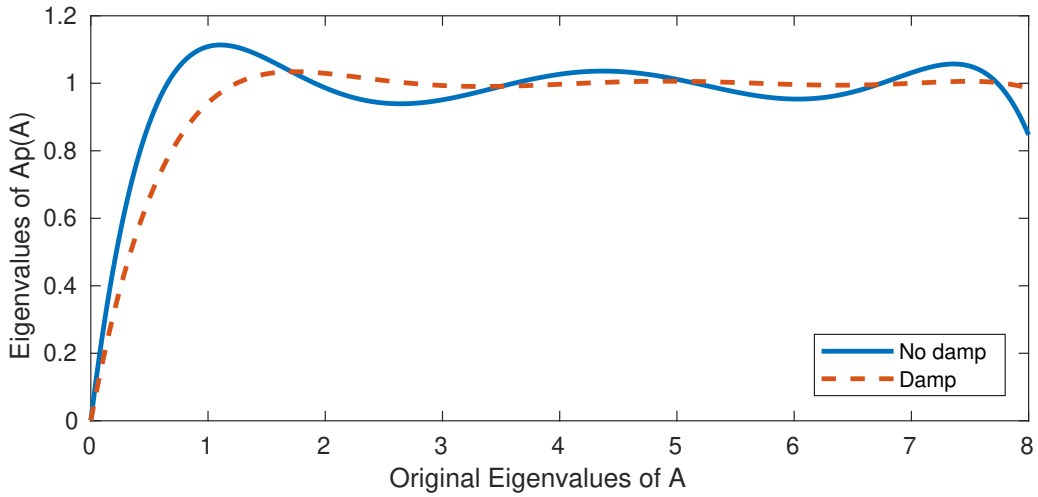


Figure 4.27: Laplacian polynomials damped and un-damped

damped polynomial has a steeper slope near the origin and does a better job of separating the small eigenvalues from zero. The un-damped polynomial also gives better conditioning. The original condition number of A is 16,211. With an un-damped polynomial of degree 6, the new condition number of $Ap(A)$ is 806.4, and the smallest eigenvalue is 1.4×10^{-3}. For the damped polynomial, the condition number is 1129.4, and the smallest eigenvalue is 9.2×10^{-4}. Damped polynomials are designed to ignore small eigenvalues, so we do not typically expect them to improve the spectrum as much as un-damped polynomials.

For this example, the differences in solve times for the damped and un-damped polynomials were small but consistent. For instance, the solve time with an un-damped degree 30 polynomial was 0.456 seconds, and this is only slightly increased with damping to 0.606 seconds. We expect that for more difficult problems with a more expensive matrix-vector product, the extra solve time for using damped polynomials will often be substantial. One such example is given in Section 5.2.3 with

the matrix G3_circuit. For that problem, using a degree 40 damped polynomial gives a 60% increase in solve time over the un-damped degree 40 polynomial. Thus, we recommend to avoid damping unless there is a clear reason that it is needed. Future work will attempt to automate polynomial damping based upon the harmonic Ritz values for the polynomial.

4.10 Summarizing the Polynomial Preconditioning Algorithm

Algorithm 4.5 summarizes the new polynomial preconditioned GMRES and its possible modifications.

4.11 Double Polynomial Preconditioning

Sometimes a matrix can benefit from a very high-degree polynomial preconditioner. Unfortunately, creating a high-degree polynomial has a very large expense in terms of orthogonalization cost and storage of basis vectors. The dots products for orthogonalization could be especially expensive for a large parallel system.

To create high-degree polynomials without the large expense, we can use double polynomial preconditioning. In this case we precondition using a composite polynomial, and the new problem becomes:

$$Ap_1(A)p_2(Ap_1(A))y = b, \tag{4.16}$$

$$x = p_1(A)p_2(Ap_1(A))y.$$

We will use d_1 to denote the degree of $\alpha p_1(\alpha)$ and d_2 for the degree of $\alpha p_2(\alpha)$. Thus, the double preconditioned problem has degree $d_1 \times d_2$.

Algorithm 4.5 GMRES with polynomial preconditioner of degree d

1. Construction of the polynomial preconditioner:
   (a) Choose v_0 random. Run d steps of the Arnoldi iteration with starting vector v_0 to build the upper-Hessenberg matrix $H_{d+1,d}$ and orthonormal matrix V_d from the Arnoldi recurrence $AV_d = V_{d+1}H_{d+1,d}$ (2.3). (To combine the polynomial with a standard preconditioner M and right preconditioning, run the initial Arnoldi iteration with the matrix AM.)
   (b) Find the harmonic Ritz values, $\{\theta_i\}_{i=1}^d$, which are the roots of the GMRES polynomial $q(\alpha) = 1 - \alpha p(\alpha)$: Solve for the eigenvalues of $H_d + h_{d+1,d}^2 f e_d^T$, where $f = H_d^{-*T} e_d$ with elementary coordinate vector $e_d = [0, \ldots, 0, 1]^T$. (See (2.20).)
   (c) Order the GMRES roots with modified Leja ordering (Alg. 4.1).
   (d) Use Algorithm 4.4 to add extra roots for stability, as needed.
   (e) To further verify stability of the polynomial, compute either ψ (4.15) or τ (4.14). If either value is more than one order of magnitude greater than the requested convergence tolerance, consider adding more roots, lowering the polynomial degree, or composing the polynomial with another preconditioner.
   (f) If damping is needed, recompute the polynomial using starting vector Av_0.
2. PP-GMRES: Apply restarted GMRES to the matrix $Ap(A) = I - \Pi_{i=1}^d (I - A/\theta_i)$ to compute an approximate solution to right-preconditioned system $Ap(A)y = b$. Use Algorithm 4.2 when applying $Ap(A)$ to a vector. To find x, compute $p(A)y$ using Algorithm 4.3.

Example 4.11.1. We consider the matrix cz20468 from the SuiteSparse CPM collection. It is a nonsymmetric PDE matrix with size $n = 20{,}468$. No right-hand side is provided, so we generate b random normal and scale it to have norm 1. With no preconditioning and with low-degree polynomials, convergence stalls. The problem requires a high-degree polynomial to converge; even a degree 90 polynomial was not enough. The degree 100 polynomial gives convergence in 1544 iterations and 91.8 seconds. With a degree of 225, the solve time is reduced to a mere 8.22 seconds with

Table 4.2: Data for polynomial preconditioning the matrix cz20468. The first column indicates the requested polynomial degree before roots are added for stability. The "Poly Dots" column indicates the number dot products that were used in creating the polynomial. (This number is included in the "Total Dots" column.) Similarly, the "Total Time" includes polynomial creation time (last column) and solve time.
**For the degree 625 polynomial, the short residual converges to 1×10^{-8}, but the true residual stops near 1.27×10^{-6}.

Degree	Add Roots	Iters	SpMVs	Dots	Poly Dots	Total Time	Poly Create Time
90	0	Stalls					
100	1	1544	156246	45966	5151	91.80	0.53
150	2	334	51223	20062	11476	30.36	1.33
225	4	42	10304	26597	25651	8.22	2.63
300	9	33	11123	46046	45451	11.06	5.22
400	16	27	12479	81007	80601	16.65	10.18
500	35	Too Slow					
625	61	25**	19207	196602	196251	42.19	32.32
750	55	26	23344	282754	282376	61.41	49.31
900	74	Stalls					

only 42 iterations, over 10 times improvement from the degree 100 polynomial. The iteration count continues to improve up to degree 400. See full statistics in Table 4.2.

All the polynomials need added roots for stability according to Algorithm 4.4; the number of roots added is given in the second column of the table. Beginning with degree 500, added roots are not sufficient to overcome the stability problems. The degree 500 problem converges far too slowly to run to the full tolerance. With a degree 625 polynomial and 61 added roots, the short residual converges, but the true residual stalls out two orders of magnitude short. The 750 degree polynomial works reasonably well with 55 added roots, but polynomials of degree 900 and above were completely useless.

Table 4.3: Data from double preconditioning for the matrix cz2048. "Poly Dots" indicates the portion of the total dot products that were used in creating the polynomial. For the degree 625 polynomial, only the short residual converged.

Degree	Iters	SpMVs	Total Dots	Poly Dots	Total Time
$10 \times 10 = 100$	Stalls				
$15 \times 10 = 150$	Stalls				
$15 \times 15 = 225$	640	154335	17045	272	83.43
$20 \times 15 = 300$	191	61130	5248	367	33.59
$20 \times 20 = 400$	43	19340	1452	462	10.35
$25 \times 20 = 500$	35	19785	1248	582	10.64
$25 \times 25 = 625$	30	21475	1198	702	11.47
$30 \times 25 = 750$	26	22505	1225	847	12.07
$30 \times 30 = 900$	29	30782	1457	992	16.21
$35 \times 30 = 1050$	25	30275	1513	1162	16.00
$35 \times 35 = 1225$	26	36575	1710	1332	19.21

As the polynomial degree increases, the total number of dot products required becomes much larger than the number of SpMVs. For instance, with the polynomial of degree 225, the number of dot products is 2.6 times the number of SpMVs. Because this increase in dot products comes primarily from creating the polynomial, the polynomial creation time begins to dominate the solve time. With the degree 400 polynomial, 99.5% of the dot products come from polynomial creation, and polynomial creation requires 61% of total solve time.

We next compare double polynomial preconditioners of the same composite degrees. See Table 4.3. We choose degrees d_1 and d_2 that are multiples of 5, keeping the same degree for the inner and outer polynomials when possible. Where the inner and outer degrees differ, we choose the larger degree for $\alpha p_1(\alpha)$ and the smaller degree for $\alpha p_2(\alpha)$. Overall, the double polynomials are less prone to stability issues than the single polynomials. None of the double polynomials needed any added roots for

stability. Unlike the single polynomials, the double polynomials do not help GMRES converge until degree 225, but after this they converge successfully for much higher degrees than with the single polynomials.

The most significant improvement comes with the reduction in dot products. With double polynomials, the total number of dot products (for polynomial creation and the linear solve) is typically less than one-tenth of the number of matrix-vector products. The minimum number of dot products used for double preconditioning is 1198, compared to 5151 with single preconditioning. Even with the polynomial of degree $35 \times 35 = 1225$, the dot products for polynomial creation are only 78% of the total number. This reduction shows in the solve times: For degrees 400 and higher, the total solve times with double preconditioning are significantly less than with single preconditioning. The best solve time still rests with the single polynomial of degree 225, but the double polynomials of degrees 400 and 500 take a close second place, converging faster than the other single polynomials. On a parallel computer, the cost savings and improvement in timing for reducing dot products could be even more substantial.

Recall that the above results use modified Gram-Schmidt orthogonalization. Even if we had performed orthogonalization as in Trilinos, with two passes of block classical Gram-Schmidt for a total of 3 dot products per GMRES basis vector, double preconditioning would still reduce dot products substantially. Of the degrees tested, the minimum number of dot products for a single polynomial (counting the cost of polynomial creation and solving) would have been 801 dot products for degree 225.

The minimum number for a double polynomial would have been 243 dot products for the polynomials of degrees 500 and 625.

In terms of iterations and sparse matrix-vector products, the double polynomials almost universally perform worse than their single polynomial counterparts (when the single polynomial problems converged). This is expected because double polynomials are not optimal for minimizing the residual over the Krylov subspace of dimension $d = d_1 \times d_2$. The exception to this is the degree $30 \times 25 = 750$ polynomial. It converges in 26 iterations, just like the single polynomial of degree 750, but since the double polynomial does not need added roots, it uses fewer matrix-vector products. One area of further study is to determine if there is a specific class of problems for which double preconditioning is generally more effective than single polynomial preconditioning in improving the iteration count.

4.12 Conclusion

We have introduced the roots implementation of the GMRES polynomial and demonstrated that is more stable than the power basis method. We discussed how the polynomial can be made more stable by added roots and how it can be adjusted through damping. The polynomial is effective for many small problems, but we also want to show it is effective at large scale. The next chapter tests the polynomial in a parallel implementation and validates its performance for several larger problems.

CHAPTER FIVE

Practical Considerations for Polynomial Preconditioning in Parallel

This chapter is adapted from the following publication: Jennifer A. Loe, Heidi K. Thornquist, and Erik G. Boman. Polynomial Preconditioned GMRES in Trilinos: Practical Considerations for High-Performance Computing. *2020 Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, in press.

Large-scale models of physical applications increasingly require solves of multiple large, sparse linear systems $Ax = b$. As computational models become more complex, efficient analysis of such models will require advanced numerical algorithms that target high-performance computing. As we work to support applications and simulations at exascale, communication has become a bottleneck for commonly-used Krylov solvers while, floating-point operations are comparatively inexpensive. As discussed in Chapter Two, the dot products required for orthogonalizing the basis vectors are especially costly as they require global all-to-all communication and serve as a synchronization point. GMRES [54] in particular suffers the effects of dot products since each new basis vector must be fully orthogonalized against previous basis vectors. While classical Gram-Schmidt allows some dot products to be combined into a single block dot product, the cost of communication from dot products can still be a bottleneck at large scale.

Due to the high communication and global synchronization requirements of GMRES, much research has been devoted to communication-avoiding variants [23, 37, 12]. Polynomial preconditioning is another approach that has potential to be effective

for high-performance computing and reduce communication costs in Krylov solvers, as discussed in Section 3.1. Applying the polynomial preconditioner requires only sparse matrix-vector products (SpMVs) and vector updates. Because A has only a few nonzeros per row, SpMVs require only local communication between neighboring processors. With polynomial preconditioning, more matrix-vector products are used to form each basis vector before orthogonalizing, giving GMRES more power and potential for convergence for roughly the same amount of communication. Unlike preconditioners that require domain decomposition, polynomial preconditioning may give more consistent performance as the amount of parallelism is increased. Because of these advantages, we propose that polynomial preconditioning should be considered as an addition to high-performance solvers software libraries.

While polynomial preconditioners have long been considered for parallel computing ([50, 31] and references therein), they are not typically included in high-performance numerical software collections. This may be due to concerns about stability [31] or due to added costs for computing spectral information to generate the polynomial [52, p. 395]. New polynomial preconditioners are being developed for parallel applications both on CPUs [29, 32] and GPUs [28, 61, 26], but they are often limited to symmetric problems or specific applications.

Trilinos [21] is one of the major software libraries that offers a collection of advanced numerical algorithms for parallel computing, including sparse linear solvers and preconditioners. In this chapter, we demonstrate a new Trilinos implementation of the more stable "roots" polynomial method from Chapter Four. We apply the polynomial to several practical problems using small-scale parallelism to show that the

GMRES polynomial is robust, reliable, and effective. Furthermore, we demonstrate that the GMRES polynomial preconditioner can reduce othogonalization steps that result in global communication. We further compare and combine the polynomial with other standard preconditioners available in Trilinos such as ILU, block Jacobi, and algebraic multigrid. Our findings suggest general guidelines for choosing an effective polynomial degree and reveal trade-offs to consider when combining the polynomial with other preconditioners. We also contrast the GMRES polynomial with a Chebyshev polynomial preconditioner. Additional experiments demonstrate preconditioner performance at a larger scale of parallelism using multiple nodes. Finally, we discuss how polynomial preconditioning can be used as a communication-avoiding method and/or be combined with existing communication-avoiding GMRES variations.

5.1 Implementation in Trilinos

Trilinos [21] is open-source scientific computing software used in many large-scale applications such as ice sheet modeling, circuit simulation, and fluid flows. It is written primarily in C++. Trilinos provides parallel iterative solvers, algebraic preconditioners, multigrid preconditioners, eigensolvers, and many other capabilities. Its Tpetra linear algebra package supports distributed memory, while Kokkos Kernels supports shared-memory (on-node) operations. Together, this allows for portable parallel linear algebra and graph operations on multiple CPUs, on GPUs, and on other advanced architectures. Additionally, communication-avoiding solvers, such as s-step GMRES and pipelined methods, are under development.

We implement the "roots" version of the minimum residual polynomial in Trilinos, using the algorithms given in Chapter Four. Previously, only Chebyshev and least-squares polynomials were available in Trilinos as polynomial preconditioners. While these polynomial implementations only work with Hermitian matrices, the new GMRES polynomial preconditioner is designed to work for a general linear system. The code for the polynomial is written directly in Belos, the next-generation iterative linear solvers package of Trilinos [8]. We choose this package so that the polynomial preconditioner is compatible with both Epetra and Tpetra linear algebra implementations and can be composed with other preconditioners. Source code for Trilinos, including the new GMRES polynomial preconditioner, is freely available for download on GitHub (www.github.com/trilinos).

While the focus of this work is on GMRES, the new polynomial preconditioner can be used with all Krylov solvers available in Trilinos. As in the previous chapter, we let d denote the degree of $Ap(A)$ so that $p(A)$ has degree $d-1$. Our implementation in Trilinos differs from that of Chapter Four (Algorithm 4.5) in one key aspect: We always use Algorithm 4.3 to apply $p(A)$ in the GMRES iteration. While our approach may require up to twice as many AXPYs, it is more intuitive and avoids stability issues that might arise from using two different formulas to compute $p(A)$. It also allows us to monitor convergence using Belos' native residual norm status tests and was more straightforward to implement than using both Algorithms 4.3 and 4.2. For stability options, our implementation defaults to adding needed roots as per Algorithm 4.4. Damping is also available but must be activated manually. We expect

that the GMRES polynomial preconditioner will become a versatile addition to the Trilinos library.

5.2 Robustness and Cost Reduction with Polynomial Preconditioning

For all examples that follow, we run GMRES with two steps of classical Gram-Schmidt orthogonalization (ICGS(2)). This requires two block dot products and one norm per iteration. We also use the Zoltan package for hyper-graph partitioning and load-balancing. This helps optimize the communication pattern for the matrix-vector product. Times for load-balancing, partitioning, and preconditioner creation are not included in total solve times. These costs would be amortized over a full-scale simulation with multiple linear solves. Throughout this chapter, we use the Epetra linear algebra interfaces, except in Section 5.2.4, which uses Tpetra linear algebra to interface with the MueLu multigrid preconditioning. Automatic root-adding for stability is enabled throughout the following experiments. Typically, the number of added roots is less than 4. Thus, we do not explicitly make note of added roots when indicating the polynomial degree. (So, for example, a polynomial of degree $d = 20$ may actually be running with a polynomial of degree 23.) Additional matrix-vector products due to added roots are included in final counts of SpMVs. Experiments in this section were run on Baylor University's Kodiak cluster. The compute nodes have dual 18-core Intel E5-2695 V4 Broadwell processors with 256 GB of RAM. All tests in this section were run using 32 MPI processes on a single compute node.

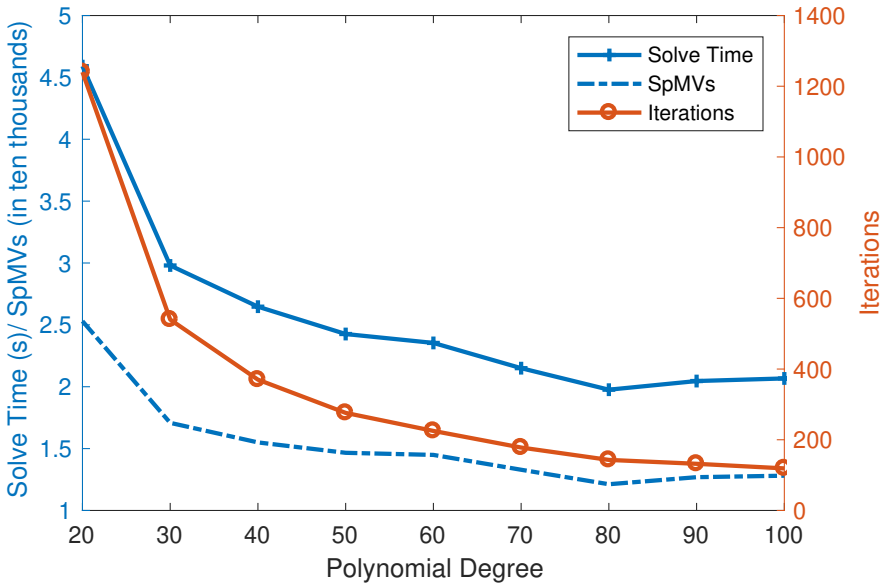


Figure 5.1: Solve time (crosses), SpMVs (dotted), and iteration count (circles) for matrix cfd2 with varying degrees of polynomial preconditioner over 32 MPI processes. The left vertical axis corresponds to solve time (in seconds) and SpMVs (in ten thousands). The right vertical axis corresponds to iterations.

5.2.1 A CFD example

We first demonstrate the potential cost savings from polynomial preconditioning with a small computational fluid dynamics example. The matrix is cfd2, from the Rothberg group in the SuiteSparse Matrix Collection [11]. The size is $n = 123{,}440$, and the matrix is symmetric positive definite. We run GMRES(50) with a random right-hand side b up to a tolerance of 1×10^{-8}. Without preconditioning, GMRES needs 113.8 seconds and 171,541 iterations to converge. Even a degree 10 polynomial preconditioner gives almost ten times improvement with a solve time of 11.43 seconds, using only 5436 iterations and 55,460 SpMVs. The degree 20 polynomial gives another two times speedup with a solve time of 4.6 seconds and 1240 iterations. Costs for higher degree polynomials are shown in Figure 5.1.

Observe that solve time, SpMVs and iteration count continue to drop with increasing polynomial degrees. The drop is sharp from degree 20 to $d = 30$ and then slower as the degree continues to increase. The best solve time occurs with $d = 80$ at 1.98 seconds with 12,120 SpMVs and 143 iterations. Recall that the iteration count corresponds to the communication-intensive orthogonalization cost of norms and dot products. The 143 iterations for a degree 80 polynomial represent more than thirty-eight times reduction of dot products and norms over the degree 10 polynomial.

After degree 80, we reach a point of diminishing returns; while iterations and orthogonalization continue to decrease, this does not give improved performance. Beginning with degree 90, SpMVs begin to increase slightly, and so does solve time. For example, the degree 100 polynomial converges in only 119 iterations, but its 12,810 SpMVs cause it to converge in 2.07 seconds. This behavior is typical of examples over a fixed number of processors: high degree polynomials often continue to improve the iteration count even when they are no longer reducing SpMVs and solve time. When applied in a large-scale computing situation, the trade-offs for solve time become less clear. The additional SpMVs of a high-degree polynomial could be justified for the sake of avoiding global communication.

Figure 5.2 shows the proportion of time spent in three different kernels: SpMVs, inner products/norms, and vector updates (AXPYs). Notice that with no preconditioning, 52% of the solve time is spent performing communication-intensive inner products and norms. With a polynomial of degree 10, about 71% of solve time is spent in SpMVs with only 17% of solve time spent in inner products and norms. As the degree increases, this balance continues to shift further towards SpMVs, which take

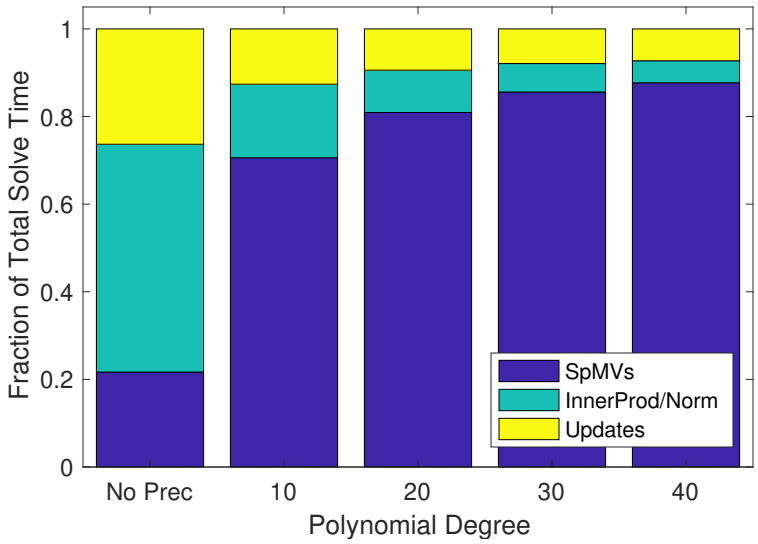


Figure 5.2: Distribution of total solve time for matrix cfd2 over 32 MPI processes using no preconditioning and polynomials of increasing degree. 1 on the vertical axis represents total solve time. The bottom (purple) section of each bar represents fraction of total solve time spent in SpMVs, the middle (teal) section time spent in inner products and norms, and the top (yellow) section time spent for vector updates (AXPYs).

88% of solve time with degree 40. Since this experiment was run on one compute node, actual communication costs are minimized. However, this distribution foreshadows the improvement that polynomial preconditioning can bring at large-scale when global reductions and synchronizations become far more costly. Additionally, the local communication costs of the SpMVs can be further reduced by using a Matrix Powers Kernel, which will be discussed in section 5.4.

For comparison to other preconditioners, solve time for cfd2 over 32 MPI processes with a KLU (block Jacobi) preconditioner is 8.19 seconds with 951 iterations. This is faster than the degree 10 polynomial, but slower than degree 20. The iteration count is less than for degree 20, but higher than with degree 30. Combining KLU with a degree 5 polynomial reduces solve time to 2.78 seconds and 68 iterations, about

2.5 seconds of which consists of applying the KLU preconditioner. This solve time is higher than with the degree 40 polynomial, but the iteration count is lower than that of even the degree 100 polynomial. Higher degrees of polynomial combined with KLU continued to reduce the iteration count but did not provide any further decrease in solve time. Although block Jacobi alone is effective for this problem, polynomial preconditioning makes it even more worthwhile.

5.2.2 Comparing to Chebyshev Polynomials

Using the same cfd2 matrix, we compare the GMRES polynomial preconditioner with a Chebyshev polynomial from the Ifpack package. (GMRES polynomials in this subsection were generated with a different starting vector than above, so timings and convergence differ.) We used the Ifpack Chebyshev preconditioner as a "black box." Internally, it diagonally scales the problem matrix and runs 10 iterations of the power method to estimate the largest eigenvalue λ_{max}.

The Chebyshev polynomials, regardless of degree, were cheaper to set up: Chebyshev polynomial creation always required less than 0.01 seconds, while GMRES polynomials needed from 0.012 seconds (degree 5) to 0.09 seconds (degree 60). More important is the difference in solve times, plotted in Figure 5.3. For degrees 5 and 10, the Chebyshev polynomials win by a wide margin, using roughly half the iterations and solve time of the GMRES polynomials. However, GMRES polynomials give the best timings overall. For degree 30 and above, the GMRES polynomials win, solving in under 3 seconds while Chebyshev polynomials need over 7 seconds. The Chebyshev

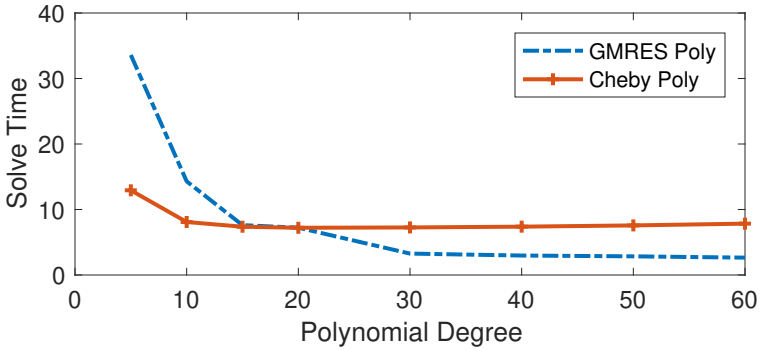


Figure 5.3: Solve times comparing GMRES polynomial and Chebyshev polynomial with different degrees for matrix cfd2.

polynomial of degree 60 needs 696 iterations while the GMRES polynomial of degree 60 needs only 225 iterations.

It is interesting that a Chebyshev polynomial works best at low degrees and the GMRES polynomial is better at higher degrees. We observed similar patterns with a 3D Laplacian. The diagonal scaling with the Chebyshev polynomial may have skewed the results. Future investigation is needed to study which polynomial creates the best preconditioned spectrum for GMRES and whether this determines the best timings. It is also possible that the Chebyshev polynomial creates a good spectrum but is simply less stable for high degrees. On a matrix which needed damping for the GMRES polynomial (s1rmq4m1), the Chebyshev polynomial performed better for all degrees.

5.2.3 A Large Test Set

We now demonstrate polynomial preconditioning with a set of seven matrices from the SuiteSparse Matrix Collection [11]. Information about the size and structure of each matrix is in Table 5.1. If provided, we used the given right-hand side vector

Table 5.1: Size and format of matrices tested from SuiteSparse

Matrix	n	NNZ	Symm
cfd2	123,440	3,085,406	SPD
Transport	1,602,111	23,487,281	none
FEM_3D_thermal2	147,900	3,489,300	none
thermal2	1,228,045	8,580,313	SPD
xenon2	157,464	3,866,688	none
ML_Geer	1,504,002	110,686,677	none
G3_circuit	1,585,478	7,660,826	SPD

from SuiteSparse. Otherwise, we generated a random right-hand side vector b. We run GMRES(100) to a tolerance of 1×10^{-8}.

Various preconditioning scenarios were considered in this study. We constructed polynomials of degrees $20, 40$, and 60 with no additional preconditioning, and we also tested an ILU(k) preconditioner from Ifpack by itself and composed with polynomials of degrees $5, 10$, and 20. The ILU preconditioner had fill level 1 and used Additive Schwarz decomposition with an overlap of 1. Results for the polynomial with no ILU preconditioning are in Table 5.2. Results for the problems with ILU are listed in Table 5.3. Recall that the iteration count is directly proportional to communication-intensive orthogonalization steps. For matrices ML_Geer, thermal2, and G3_Circuit, GMRES(100) with no preconditioning did not converge within 100,000 iterations, so "xxx" indicates the failure of the solver. We now highlight key insights from each of the matrix experiments:

cfd2. This is the same matrix from Example 5.2.1, but GMRES uses a subspace size of 100 instead of 50. Across the board, solve times are faster with the large subspace, using less iterations and SpMVs. However, with polynomial preconditioning,

Table 5.2: Solve times and statistics for several SuiteSparse matrices

	No Prec			Degree 20			Degree 40			Degree 60		
Matrix	SPMVs	Time	Iters	SPMVs	Time	Iters	SPMVs	Time	Iters	SPMVs	Time	Iters
cfd2	85970	102	85116	15740	3.82	779	11850	2.52	286	12600	2.533	198
Transport	40670	1042	40268	6048	26.16	285	3948	15.19	93	4032	14.51	63
FEM_3D_thermal2	485	0.73	480	630	0.151	29	748	0.16	16	804	0.193	11
thermal2	xxx	xxx	xxx	26200	78.98	1297	14280	37.01	353	6466	17.1	104
xenon2	3338	5.73	3304	2080	0.626	102	2200	0.521	54	2340	0.606	38
ML_Geer	xxx	xxx	xxx	260500	3214	12897	61580	731.5	1487	29570	346.7	472
G3_circuit	xxx	xxx	xxx	xxx	xxx	xxx	20080	35.88	473	xxx	xxx	xxx
G3_circuit DAMPED	N/A	N/A	N/A	47380	108.7	2233	32100	57.74	775	28730	45.82	451

Table 5.3: Solve times and statistics for ILU and polynomial preconditioning

	ILU Only			ILU + Deg 5			ILU + Deg 10			ILU + Deg 20		
Matrix	SPMVs	Time	Iters	SPMVs	Time	Iters	SPMVs	Time	Iters	SPMVs	Time	Iters
Transport	1898	55.9	1879	1595	22.59	315	920	10.81	91	960	10.22	47
FEM_3D_thermal2	21	0.06	20	30	0.099	5	40	0.058	3	45	0.064	2
thermal2	13030	301	12902	7310	64.73	1447	4710	34.58	466	2180	16.52	107
G3_circuit	992	23.9	982	1120	9.474	221	1160	7.502	114	1080	5.484	53

the point of diminishing returns comes sooner. SpMVs and solve time go up starting with degree 60 instead of degree 90. This suggests that high-degree polynomials may be more effective with smaller subspaces where more power is required to overcome the information lost with restarting.

Transport. For this problem, solve time decreases from degree 40 to degree 60 even though the number of SpMVs has slightly increased. ILU(1) preconditioning alone is less effective than the polynomial preconditioners alone, but the best solve time occurs when combining polynomials with ILU(1). You can see the breakdown of relative solve times with ILU plus PP-GMRES in Figure 5.4. With ILU(1) and no polynomial, applying ILU takes 21% of solve time, while inner products and norms need 31% of solve time. When composed with a degree 20 polynomial, ILU dominates solve time at 63% while inner products and norms are reduced to 3% of solve time. Multiplications with A consume less than 30% of solve time, even when we increase the polynomial degree to 40. Unlike the cfd2 solve times in Figure 5.2 where SpMVs dominated, combining with an ILU preconditioner places most of the work in the ILU kernel.

Polynomial preconditioner creation times (not included in the solve times) seem reasonable compared to ILU. For this matrix, the ILU preconditioner alone required about 0.32 seconds to create. To compose the ILU with a degree 5 polynomial requires an additional 0.06 seconds, and to compose with a degree 40 polynomial requires an additional 0.73 seconds. By comparison, constructing a polynomial of degree 20

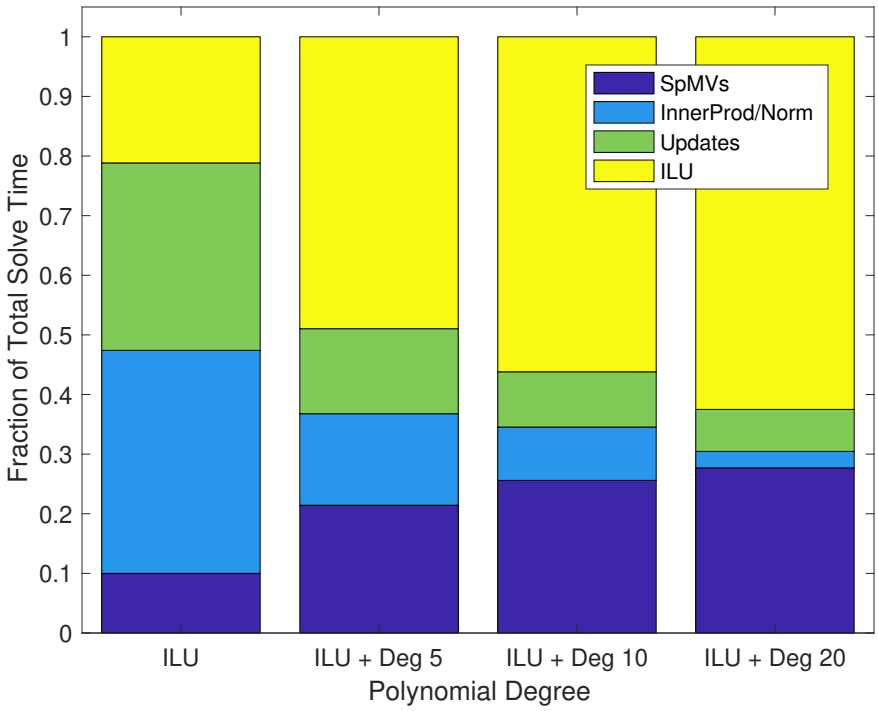


Figure 5.4: Relative solve times (scaled) for matrix Transport for ILU and ILU combined with polynomials of varying degrees.

without ILU requires 0.18 seconds, degree 40 needs 0.49 seconds, and degree 80 uses 1.66 seconds.

FEM_3D_thermal2. For this matrix, polynomial preconditioning alone works well, but ILU(1) is significantly better. The ILU preconditioner is so effective that adding a polynomial does not give noticeable speedup. For polynomials without ILU, this matrix used the most added roots for stability. It needed 1 added root for degree 20, 4 roots with degree 40, and 7 added roots for degree 60. No other matrices in the test set required a degree 60 polynomial with more than 3 added roots or a degree 40 polynomial with more than 2 added roots. It is interesting to note that, for all matrices, none of the polynomials combined with ILU needed added roots. While this

will not be true universally, it suggests that starting with a preconditioned spectrum and using low polynomial degrees helps polynomial behavior.

thermal2. In this instance, ILU(1) applied alone is the clear loser. It needs almost ten times as many SpMVs and over three times the solve time of the degree 20 polynomial. However, high-degree polynomials applied alone give comparable results to ILU(1) plus low-degree polynomials. In terms of solve time and iterations, the degree 60 polynomial is almost equivalent to ILU(1) plus degree 20. To balance the trade-offs at large scale, one would need to consider whether it is cheaper to do more SpMVs with the matrix A alone versus fewer SpMVs with AM, where M is the ILU preconditioner.

xenon2 and ML_Geer. For both of these matrices, polynomial preconditioning gives good improvement, while ILU(1) makes convergence worse. The ML_Geer problem is especially difficult. While low-degree polynomials work well, degree 80 gives good improvement over degree 60, with 16,970 SpMVs, 200 iterations, and 197 seconds solve time. For this matrix, the point of diminishing returns occurs after degree 120, which gives a solve time of 141.5 seconds. This suggests that very high-degree polynomials can be useful for challenging problems. Notice also the improvement in iterations and dot products: Doubling the polynomial degree from 20 to 40 gives over 8.6 times reduction in dot products, and doubling from degree 40 to degree 80 gives another 7.4 times reduction.

G3_Circuit. Of particular interest is the matrix G3_Circuit. With stand-alone polynomial preconditioning, degree 40 gives convergence while degrees 20 and 60 do not. With damping, (see last row in Table 5.2), polynomials of degrees $20, 40$, and 60 all give convergence, but degree 40 converges more slowly than without damping. This is expected because a damped polynomial usually gives a more difficult spectrum than an effective un-damped polynomial. Ultimately, the best solution for this problem was to combine with ILU(1) preconditioning. The ILU preconditioned problem proves to be faster than any of the polynomials working alone. Composing it with a degree 20 polynomial further improves the solve time by four times and the iteration/orthogonalization count by 18.5 times.

5.2.4 Combining with a Simple Multigrid

Factorization-based preconditioners are not the only preconditioners that can be composed with polynomials. As long as algebraic multigrid (AMG) is applied consistently, it, too, can be combined with a polynomial. (Polynomials should not be combined with any preconditioner that requires FGMRES; to do so would require re-creating the polynomial each time the inner preconditioner changes.) We use the MueLu package of Trilinos to combine algebraic multigrid with a polynomial preconditioner. The problem is a 3D Laplacian generated with the Galeri package. It has a unit cube mesh with 250 grid points in each direction and size $n = 15{,}625{,}000$. The right-hand side is randomly generated. The algebraic multigrid uses smoothed aggregation with Chebyshev smoothing over 5 levels on 32 MPI processes. We run GMRES(50) to a tolerance of 1×10^{-8}.

Table 5.4: Solve time, iterations, and polynomial creation time for AMG with polynomial preconditioning.

Preconditioning	Iters	Solve Time	Poly Create	Solve + Poly Create
AMG only	42	13.95		13.95
AMG + Deg 2	26	9.71	0.29	10.00
AMG + Deg 3	19	8.55	0.44	8.99
AMG + Deg 5	12	7.78	0.75	8.53
AMG + Deg 7	9	7.90	1.10	9.00
AMG + Deg 10	6	7.62	1.69	9.31
AMG + Deg 12	4	7.75	2.14	9.89

With multigrid preconditioning, the problem converges in 42 iterations and 13.95 seconds. Setup for the multigrid preconditioner takes about 0.95 seconds. Composing with even a degree 2 polynomial gives noticeable improvement. Table 5.4 gives timings and iteration counts for several degrees. The polynomial of degree 10 gives the best solve time of 7.62 seconds with 6 iterations. The point of diminishing returns comes quickly compared to previous examples; degrees 12 and above give slightly slower convergence than degree 10. These results are not surprising since multigrid is more costly to apply than ILU. With the degree 10 polynomial, applying multigrid uses about 73% of the solve time. Using higher degree polynomials means spending more time in multigrid with fewer basis vectors saved. Polynomial preconditioners combined with multigrid are more expensive to create than those combined with ILU. When we add the polynomial creation and solve times, the degree 5 polynomial with AMG is fastest at 8.53 seconds. While this problem shows only minimal benefit from polynomial preconditioning, it foreshadows the potential improvement for problems such as convection-diffusion that are more difficult for AMG.

Aggregating results of the previous examples suggests a strategy for choosing polynomial degrees: When polynomial preconditioning is deployed alone, degrees of 40 and higher are often appropriate. When combined with a simple factorization-based preconditioner such as ILU or block Jacobi, degrees from 5 to 30 seem to be most effective. Combining a polynomial with multigrid makes each polynomial application very expensive, so small degrees from 2 to 15 are likely to be best. Polynomial preconditioners are versatile and adaptable with many preconditioning combinations.

5.2.5 Comparing Polynomial Implementations

At this point in time, Trilinos has three different implementations of the minimum residual polynomial preconditioner: the power basis implementation (as in Chapter Three), the new "Roots" implementation, and an Arnoldi basis implementation based upon Algorithm 3.2. We use the matrix thermal2 from SuiteSparse and run GMRES(50) to compare the polynomial implementations. Instead of using the right-hand side provided on SuiteSparse (used in the previous test set), we generate a random vector for b and read it in from file. We also use this vector as the starting vector for the polynomial, so polynomials of the same degree are mathematically equivalent across implementations.

Without preconditioning, the solver needs more than 500,000 iterations and 4500 seconds to converge. Solve times for polynomials with degree divisible by 5 are plotted in Figure 5.5. At degree 10, the power basis method is slightly faster than the other two methods, but then it falls behind. The least number of iterations for the power basis method is 16,276 for degree 15, and the iteration count does not improve

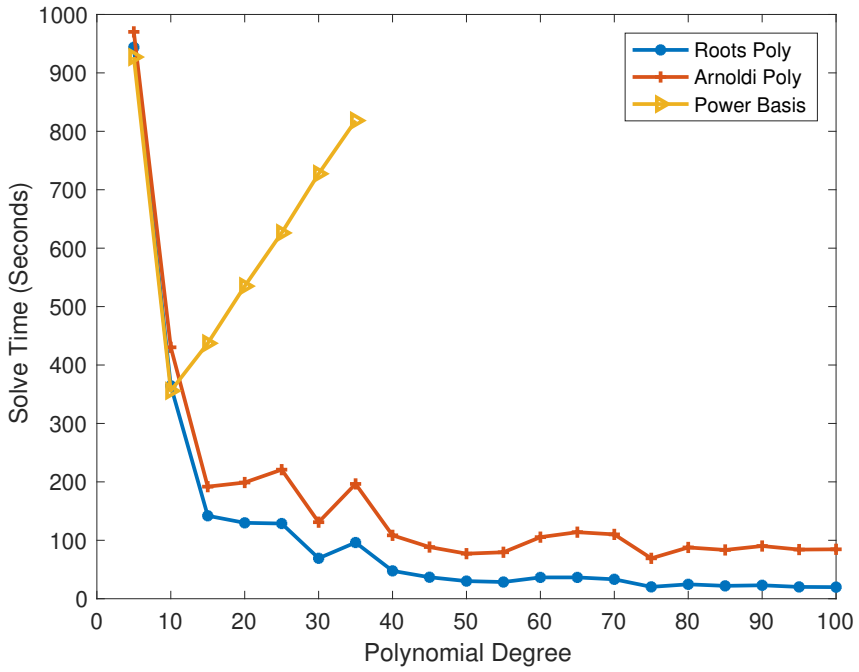


Figure 5.5: Solve times for matrix thermal2 using three different polynomial implementations: the new "roots" polynomial method, the Arnoldi basis method, and the power basis method. Polynomial degree indicates the degree of $Ap(A)$. No extra roots are added for stability.

any further. Beginning with degree 40, convergence becomes extremely erratic and unpredictable; no problems converge in less than 100,000 iterations, and many don't converge at all.

Starting with degree 10, the Arnoldi and Roots polynomial implementations converge in the same number of iterations. This problem does not need additional roots for stability, so the polynomials are the same at each degree. Since thermal2 has all real eigenvalues, we are using the maximum number of AXPYs for the Roots method to apply $p(A)$ with Algorithm 4.3. Even so, the timings for the Roots method polynomials are consistently smaller than for the Arnoldi polynomials. The margin between the timings widens as the polynomial degree increases. For degree 30, the Roots

polynomial requires about half the solve time of the Arnoldi polynomial (69 seconds versus 131). With degree 100, the Roots polynomial uses less than one-fourth the solve time of the Arnoldi polynomial (19.8 seconds versus 84.6). Applying the polynomial via the Arnoldi basis method is significantly more expensive, even in parallel. By degree 100, the iteration count has decreased to 156. Polynomial creation times remained comparable between the two methods.

5.3 Larger-Scale Experiments

Experiments in this section were run on a bigger cluster. The compute nodes each have 2.6 GHz Intel E5-2670 Sandy Bridge processors with 16 cores and 64 GB of RAM. These experiments were run with GMRES(50) to a tolerance of 1×10^{-8}. Problem matrices were generated with the finite difference stencils in the Galeri package. All ILU preconditioners have a fill level of 1 and Additive Schwarz overlap of 1.

We read in a randomly generated vector from file to use as the right-hand side for both problems. We also use this vector as the starting vector to generate the polynomial. This prevents inconsistencies that may arise when random number generators create different vectors as the number of MPI processes increases. Note that even with the same generating vector, a degree d polynomial may have minor differences over different numbers of MPI processes. In exact arithmetic, two degree d polynomials generated from the same starting vector will be the same. This is unlike an ILU preconditioner with domain decomposition, which is guaranteed to change with increasing MPI processes and subdomains. However, varying dot-product reduction trees over more processes can cause small differences in the polynomial, just as they

can cause unpreconditioned GMRES to have somewhat different convergence when varying the number of processors.

5.3.1 Convection-Diffusion

Our first example is the convection-diffusion "UniFlow2D" problem. We use a square mesh with 1000 grid points, resulting in a matrix of size $n = 100$ million with 449,960,000 nonzeros. While polynomial preconditioning alone did help to improve convergence, the iteration count and solve times were high. For instance, over 128 cores, a degree 40 polynomial needed 728 iterations and 943 seconds to converge. Polynomials worked much better when combined with an ILU(1) preconditioner.

We tested ILU(1) alone, and then combined it with polynomials of degree $5, 10, 20, 40$ and 80. Tests used from 2 to 32 nodes, each with 16 cores, for a total of 32 to 512 MPI processes. Adding a degree 5 polynomial gave anywhere from 2.8 to 4.9 times improvement in solve time over ILU alone. Over 2 nodes, ILU(1) alone converges in 252.9 seconds (208 iterations), and it uses 35 seconds (327 iterations) over 32 nodes. ILU(1) plus a degree 10 polynomial gives the fastest convergence with 64.24 seconds (17 iterations) over 2 nodes and 5.41 seconds (24 iterations) over 32 nodes. ILU with polynomials of degrees $20, 40$, and 80 does run slower. This supports the assertion that from section 5.2.4 that ILU is best combined with not-too-large of a polynomial degree. It is notable, however, that the ILU with a degree 80 polynomial converges in only 3 to 5 iterations.

The polynomial preconditioned problems in this example do show better strong scaling than ILU alone (Figure 5.6). This is likely because the polynomials help to

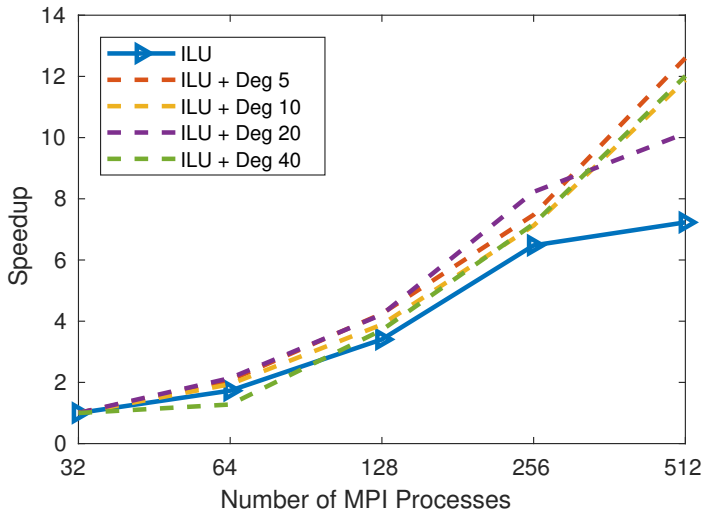


Figure 5.6: Strong scaling for total solve time of a 2D convection-diffusion problem, computed as solve time over N cores divided by solve time over 32 cores. Scaling is shown for ILU(1) preconditioning alone versus ILU combined with various polynomials.

mitigate the effects of domain decomposition on the ILU preconditioner and maintain somewhat more consistent iteration counts than ILU alone. The one exception is the degree 40 polynomial plus ILU over 64 cores. This solver required 11 iterations to converge, while other solves with this polynomial required 6 to 8 iterations. We do not believe that global communication played a significant role in these scaling results. However, it is notable that polynomial preconditioning may help to overcome the scaling difficulties of ILU.

5.3.2 3D Laplacian

We also test a standard finite difference 3D Laplacian from the Galeri package. We use a cube mesh with 550 grid points. This gives a matrix of size $n = 166{,}375{,}000$ with over 1.1 billion nonzeros. This problem was run on 16 nodes up to 256 nodes (256 to 4096 cores). All polynomial preconditioners were more effective than ILU(1)

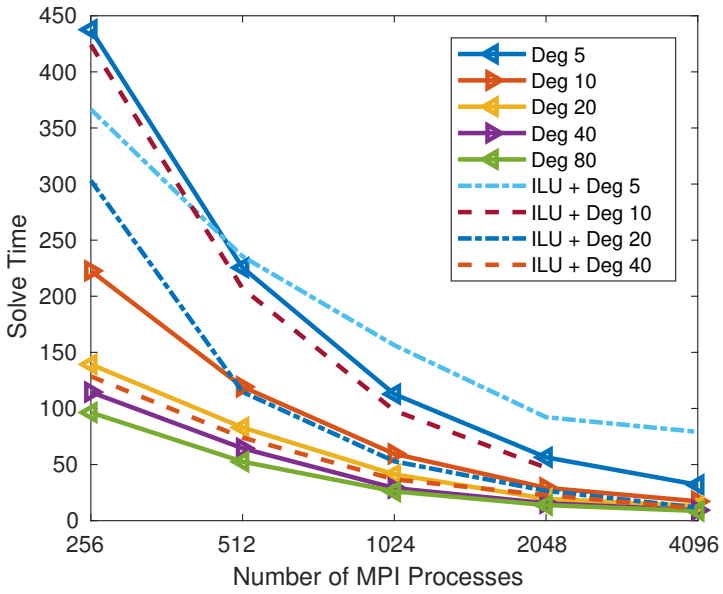


Figure 5.7: Total solve time for polynomial preconditioners of varying degrees (some with ILU) applied to a 3D Laplacian over increasing numbers of MPI processes.

with no polynomial. Combining the polynomials with ILU gave improvement, but solve times were interspersed with those of polynomials working alone, similar to the thermal2 problem. Several solve times are shown in Figure 5.7. We observe that high-degree polynomials seem to make less of an impact to solve time at high core counts than at low core counts. The degree 80 polynomial clearly converges fastest over 256 cores, but beginning with 2048 cores, it is slightly overtaken by the degree 60 polynomial [not plotted] and is very close in run time to the degree 40 polynomial. Even so, the degree 40 polynomial still represents a three times improvement in solve time over the degree 5 polynomial. Further study is required at larger scale to analyze the trade-offs between a more expensive matrix-vector product versus reducing dot products and global synchronizations.

Strong scaling for the polynomial preconditioned problems was near perfect and roughly equivalent to the strong scaling of non-preconditioned GMRES. This indicates that global communication is not a bottleneck at this scale on this computing cluster. Strong scaling for ILU with polynomials was mixed. ILU combined with degree 5 did not maintain consistent iteration counts as well as ILU combined with higher degree polynomials. In fact, over 256 nodes, ILU alone converges slightly faster than ILU plus a degree 5 polynomial. ILU plus a degree 10 polynomial performs well for up to 2048 cores, but then doesn't converge for 4096 cores. Further investigation is needed to understand this failure. The ILU preconditioner alone scaled worse than the problems with only polynomial preconditioning.

5.4 Relation to other Communication-Avoiding Methods

There are several variants of GMRES which avoid communication inherent in the standard implementation. These methods reduce the number of global communication steps at the cost of more memory and flops. The savings in global reduction and synchronization is important on extreme-scale parallel systems.

Polynomial preconditioned GMRES (PP-GMRES) itself avoids communication. Polynomial preconditioners help to mitigate all-to-all communication from dot products and norms by performing more work (mostly SpMVs) between each orthogonalization step. The SpMVs are "communication-avoiding" compared to dot products because the dependency pattern is sparse and it is simple to determine the required replication/ghosting of data [23]. Unlike other communication-avoiding GMRES variants, PP-GMRES is not mathematically equivalent to standard GMRES,

nor does it give the same convergence in terms of SpMVs. For instance, we do not expect GMRES(50) with a degree 6 polynomial to converge in as few SpMVs as GMRES(300). In this case, PP-GMRES minimizes the problem residual over only a subset of the subspace that GMRES(300) uses. Nevertheless, polynomial preconditioning can be very effective in reducing orthogonalization and overcoming the limitations imposed by restarting GMRES.

We discuss polynomial preconditioning in comparison to and in combination with three other communication-reducing strategies. Two are variants of GMRES: pipelined GMRES and s-step GMRES. Pipelined versions of GMRES have communication patterns which may be well-suited for polynomial preconditioners, and s-step GMRES methods share several similarities with PP-GMRES. The third strategy we discuss is the Matrix Powers Kernel (MPK) [23] which can minimize local communication in the many SpMVs required to apply the polynomial.

5.4.1 Pipelined Methods

Pipelined methods hide global communication by overlapping it with other calculations, typically to include SpMVs. A one-level pipelined GMRES method in [19] combines the block dot product and norm for one orthogonalization step and overlaps this with the SpMV for creating the next Krylov basis vector. However, sometimes dot product latency is longer than the time to compute an SpMV, leaving processors idle as the synchronization completes. To compensate for this, one can extend the "pipeline length"; that is, one can combine the dot products of ℓ iterations into one step, and overlap this communication step with ℓ SpMVs. This is the idea behind

the $p(\ell)$-GMRES algorithm in [19]. While this strategy is advantageous in terms of keeping processors running, longer pipelines require more calculations, more vector storage, and introduce more sources of instability. In addition to the extra computations required to use overlapping dot products, one must compute shifts to maintain linear independence of basis vectors and avoid breakdown of the method.

Adding preconditioning to pipelined GMRES can be advantageous because the preconditioned matrix-vector product will be more expensive than an SpMV alone. This means that shorter pipeline lengths will be needed to hide communication latency and the method will have better stability. Polynomial preconditioning may be better suited than other preconditioners to pipelined GMRES for two reasons: 1) Some preconditioners, such as algebraic multigrid, can require global communication to apply. This defeats the intention of pipelined methods; one cannot hide global synchronizations with even more global communication. Since polynomial preconditioning requires only local communications, it can be easily overlapped with the dot products. 2) Polynomial degree selection may give flexibility to obtain a preconditioner fit to the desired pipeline length. One could select a polynomial degree based upon the number of SpMVs required to hide the latency of one dot product.

5.4.2 S-Step Methods

S-step GMRES (see [23] and citations therein) takes a different approach to avoiding communication. Like pipelined GMRES, it consecutively performs several SpMVs for the next s Krylov basis vectors. But instead of trying to overlap orthogonalization with SpMVs, all the new basis vectors are orthogonalized at once using the

Tall-Skinny QR (TSQR) algorithm. Thus global communication (inner products, orthogonalization) happens only once every s iterations. Like the pipelined methods, s-step GMRES is also prone to numerical instability as the step size s increases. The vector sequence $b, Ab, A^2b, .., A^sb$ will quickly lose linear independence without orthogonalization. To maintain stability and linear independence, extra computations are performed to create a Newton Basis [7].

Polynomial preconditioning is similar to s-step GMRES in that orthogonalization is only performed once every d SpMVs after applying the polynomial (of degree d). However, s-step GMRES is mathematically equivalent to standard GMRES, while polynomial preconditioned GMRES is not. Even so, we could use polynomial preconditioning as an alternative to s-step GMRES. We could also use polynomial preconditioning within s-step GMRES. In this case, orthogonalization is only needed once every $d * s$ SpMVs. This PP-s-Step GMRES may give the best balance between minimizing SpMVs, delaying orthogonalization, and maintaining stability.

5.4.3 Matrix Powers Kernel and CA-GMRES

The Matrix Powers Kernel is designed to perform several matrix-vector products consecutively while minimizing reads from memory. With the Matrix Powers Kernel (MPK), local communication can be reduced at a cost in increased storage requirement. Communication-Avoiding (CA)-GMRES [23], is another GMRES variant which combines the MPK with s-Step GMRES. We could use the Matrix Powers Kernel with polynomial preconditioning to evaluate $Ap(A)$. This could be combined with either of s-Step GMRES or pipelined GMRES. When the MPK is used to eval-

uate a polynomial, rather than to take s steps as in CA-GMRES, cache reuse is improved because intermediate vectors do not need to be stored. Thus, PP-GMRES provides an avenue for taking advantage of the Matrix Powers Kernel while avoiding the numerical pitfalls of delayed orthogonalization that occur in s-step and pipelined methods.

5.5 Conclusions and Future Work

We described an implementation of the GMRES polynomial preconditioner in Trilinos and showed parallel results. We demonstrated that polynomial preconditioning is effective and stable for a variety of practical problems. Furthermore, it greatly reduces the number of dot products and norms that require costly global communication and synchronization. Our experiments show that the polynomials are versatile and adaptable with many preconditioning combinations.

We showed parallel results on a moderate size cluster. Future work include experiments very large-scale applications on highly parallel supercomputers, where communication is more expensive. We also plan to test the method on GPUs. In particular, we would like to analyze the effectiveness of polynomial preconditioning combined with s-step GMRES in avoiding global communication. We believe polynomial preconditioning is under-appreciated and is a good alternative (or complement) to recent communication-avoiding methods such as CA-GMRES. It should be made available in high-performance software libraries to help enable exascale computing. We expect that the GMRES polynomial preconditioner will become a useful addition to Trilinos.

CHAPTER SIX

Conclusions and Future Work

The minimum residual polynomial preconditioner has advanced to be stable and effective for many practical linear systems. We have developed new insight into the power basis method and provided the more stable roots implementation. Our additional modifications with added roots and damping enable the polynomial to improve convergence for a wider variety of problems. Finally we provided an implementation in Trilinos and tested the preconditioner for larger problems. Our future objectives are three-fold: 1) to refine our understanding of polynomial behavior so we can more accurately refine it to be stable and effective, 2) to validate the communication-avoiding properties of the polynomial preconditioner and understand its relation to other communication-avoiding methods, and 3) to enable modern software implementations of the polynomial to promote use in real-life applications.

As we explored the new polynomial preconditioner, several opportunities arose to fine-tune our understanding of its behavior and parameters. We would like to refine the root-adding criteria so that unneeded roots are less likely to be added while still appending all necessary roots. More study may also allow us to automate polynomial damping and quickly identify problems where this strategy will be effective. Furthermore, we would like to identify classes of problems, such as matrices with indefinite spectra, which often prove difficult for polynomial preconditioning. We hope to present new strategies to apply the polynomial for these special cases. We also plan

to expand our study of the polynomial preconditioner for solvers other than GMRES, such as BiCGStab and IDR, so that we can understand its performance for non-restarted Krylov methods. Finally, it would be helpful to find a method to estimate the best polynomial degree ahead of time without testing all possible combinations.

In our future work, we plan to validate the communication-avoiding properties of the polynomial preconditioner by testing it on highly parallel computers and on alternative architectures such as those with multiple GPUs. We want to directly compare the polynomial to current communication-avoiding GMRES variants and study the improvement when we use the minimum residual polynomial to precondition these methods. More study will investigate both the strong and weak scaling properties of the polynomial. We hope to see that the polynomial gives significant improvements for strong scaling. We do not anticipate that polynomial preconditioning alone will have very good weak scaling, but we expect it could be useful for weak scaling when combined with another preconditioner such as algebraic multigrid.

Our Trilinos polynomial implementation has made our preconditioner available to use in several applications. We plan to maintain this polynomial preconditioning code and continue conversations with developers in applications software to encourage its use. Furthermore, we would like to code an implementation in PETSC4Py so that the preconditioner will be available to a broader audience of users. We would also like to study the minimum residual polynomial as a smoother for algebraic multigrid preconditioning. In the future, we may also expand both software implementations to apply polynomial preconditioning to eigenvalue problems.

Ultimately, our work overcomes the common hurdles of polynomial preconditioning by providing a polynomial that is straightforward to compute and stable for very high degrees. We hope that this endeavor will spark renewed interest in polynomial preconditioning and that our methods can be helpful in problems of practical interest.

BIBLIOGRAPHY

[1] LAPACK POTRF documentation. http://www.netlib.org/lapack/explore-3.1.1-html/dpotrf.f.html.

[2] Matrix Market. https://math.nist.gov/MatrixMarket/.

[3] A. M. Abdel-Rehim, R. B. Morgan, and W. Wilcox. Improved seed methods for symmetric positive definite linear equations with multiple right-hand sides. *Numer. Linear Algebra Appl.*, 21(3):453–471, 2014.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[5] S. F. Ashby. *Polynomial preconditioning for conjugate gradient methods*. ProQuest LLC, Ann Arbor, MI, 1988. Thesis (Ph.D.)–University of Illinois at Urbana-Champaign.

[6] S. F. Ashby, T. A. Manteuffel, and J. S. Otto. A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems. *SIAM J. Sci. Statist. Comput.*, 13(1):1–29, 1992.

[7] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA J. Numer. Anal.*, 14(4):563–581, 1994.

[8] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.

[9] D. Calvetti, B. Lewis, and L. Reichel. GMRES-type methods for inconsistent systems. *Linear Algebra Appl.*, 316(1-3):157–169, 2000. Conference Celebrating the 60th Birthday of Robert J. Plemmons (Winston-Salem, NC, 1999).

[10] D. Calvetti and L. Reichel. On the evaluation of polynomial coefficients. *Numer. Algorithms*, 33(1-4):153–161, 2003. International Conference on Numerical Algorithms, Vol. I (Marrakesh, 2001).

[11] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Software*, 38(1):Art. 1, 25, 2011.

[12] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008.

[13] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, second edition, 2014.

[14] M. Embree, J. A. Loe, and R. B. Morgan. Polynomial preconditioned Arnoldi. Preprint, https://arxiv.org/abs/1806.08020.

[15] V. Faber, W. Joubert, E. Knill, and T. Manteuffel. Minimal residual method stronger than polynomial preconditioning. *SIAM J. Matrix Anal. Appl.*, 17(4):707–729, 1996.

[16] R. Fletcher. Conjugate gradient methods for indefinite systems. In *Numerical analysis (Proc 6th Biennial Dundee Conf., Univ. Dundee, Dundee, 1975)*, pages 73–89. Lecture Notes in Math., Vol. 506. 1976.

[17] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.*, 14(2):470–482, 1993.

[18] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60(3):315–339, 1991.

[19] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM J. Sci. Comput.*, 35(1):C48–C71, 2013.

[20] S. Goossens and D. Roose. Ritz and harmonic Ritz values and the convergence of FOM and GMRES. *Numer. Linear Algebra Appl.*, 6(4):281–293, 1999.

[21] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005.

[22] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards*, 49:409–436 (1953), 1952.

[23] M. F. Hoemmen. Communication-avoiding Krylov subspace methods. Technical Report UCB/EECS-2010-37, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, April 2010.

[24] O. G. Johnson, C. A. Micchelli, and G. Paul. Polynomial preconditioners for conjugate gradient calculations. *SIAM J. Numer. Anal.*, 20(2):362–376, 1983.

[25] W. Joubert. A robust GMRES-based adaptive polynomial preconditioning algorithm for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 15(2):427–439, 1994. Iterative methods in numerical linear algebra (Copper Mountain Resort, CO, 1992).

[26] A. A. Kamiabad and J. E. Tate. *Polynomial Preconditioning of Power System Matrices with Graphics Processing Units*, pages 229–246. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[27] C. Lanczos. Solution of systems of linear equations by minimized-iterations. *J. Research Nat. Bur. Standards*, 49:33–53, 1952.

[28] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, Feb 2013.

[29] R. Li, Y. Xi, E. Vecharynski, C. Yang, and Y. Saad. A thick-restart Lanczos algorithm with polynomial filtering for Hermitian eigenvalue problems. *SIAM J. Sci. Comput.*, 38(4):A2512–A2534, 2016.

[30] Y. Liang. Stability of polynomial preconditioning. In A. Handlovicova, M. Komornikova, K. Mikula, and D. Sevcovic, editors, *Proceedings of Algoritmy 2000, Conference on Scientific Computing*, pages 264–273. Comenius Univ. Press, 2001.

[31] Y. Liang. *The use of parallel polynomial preconditioners in the solution of systems of linear equations.* PhD thesis, University of Ulster, 2005.

[32] Y. Liang, M. Szularz, and L. T. Yang. Finite-element-wise domain decomposition iterative solvers with polynomial preconditioning. *Math. Comput. Modelling*, 58(1-2):421–437, 2013.

[33] Y. Liang, J. Weston, and M. Szularz. Generalized least-squares polynomial preconditioners for symmetric indefinite linear equations. volume 28, pages 323–341. 2002. Parallel matrix algorithms and applications (Neuchâtel, 2000).

[34] Q. Liu, R. B. Morgan, and W. Wilcox. Polynomial preconditioned GMRES and GMRES-DR. *SIAM J. Sci. Comput.*, 37(5):S407–S428, 2015.

[35] J. A. Loe and R. B. Morgan. Polynomial preconditioned BICGStab and IDR. Preprint, https://sites.baylor.edu/ronald_morgan/files/2015/05/PPNSymmLanLinEqs-1828ts3.pdf.

[36] C. Meyer. *Matrix Analysis and Applied Linear Algebra.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.

[37] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12, New York, NY, USA, 2009. ACM.

[38] R. B. Morgan. Computing interior eigenvalues of large matrices. *Linear Algebra Appl.*, 154/156:289–309, 1991.

[39] R. B. Morgan. GMRES with deflated restarting. *SIAM J. Sci. Comput.*, 24(1):20–37, 2002.

[40] R. B. Morgan and M. Zeng. A harmonic restarted Arnoldi algorithm for calculating eigenvalues and determining multiplicity. *Linear Algebra Appl.*, 415(1):96–113, 2006.

[41] N. M. Nachtigal, L. Reichel, and L. N. Trefethen. A hybrid GMRES algorithm for nonsymmetric linear systems. volume 13, pages 796–825. 1992. Iterative methods in numerical linear algebra (Copper Mountain, CO, 1990).

[42] D. P. O'Leary. Yet another polynomial preconditioner for the conjugate gradient algorithm. *Linear Algebra Appl.*, 154/156:377–388, 1991.

[43] C. C. Paige, B. N. Parlett, and H. A. van der Vorst. Approximate solutions and eigenvalue bounds from Krylov subspaces. *Numer. Linear Algebra Appl.*, 2(2):115–133, 1995.

[44] C. C. Paige and M. A. Saunders. Solutions of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975.

[45] B. N. Parlett. *The symmetric eigenvalue problem*, volume 20 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998. Corrected reprint of the 1980 original.

[46] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, 2016.

[47] L. Reichel. Newton interpolation at Leja points. *BIT*, 30(2):332–346, 1990.

[48] H. Rutishauser. Theory of gradient methods. In M. Engeli, T. Ginsburg, H. Rutishauser, and E. Stiefel, editors, *Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems*, pages 24–49. Birkhäuser, Basel-Stuttgart, 1959.

[49] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Math. Comp.*, 37(155):105–126, 1981.

[50] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985.

[51] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.

[52] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[53] Y. Saad. *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2011.

[54] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7(3):856–869, 1986.

[55] P. Sonneveld and M. B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM J. Sci. Comput.*, 31(2):1035–1062, 2008/09.

[56] H. K. Thornquist. Fixed-polynomial approximate spectral transformations for preconditioning the eigenvalue problem. PhD Thesis, Rice University, TR06-05, 2006.

[57] H. A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13(2):631–644, 1992.

[58] H. A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. *Numer. Linear Algebra Appl.*, 1(4):369–386, 1994.

[59] M. B. van Gijzen. A polynomial preconditioner for the GMRES algorithm. *J. Comput. Appl. Math.*, 59(1):91–107, 1995.

[60] H. F. Walker and L. Zhou. A simpler GMRES. *Numer. Linear Algebra Appl.*, 1(6):571–581, 1994.

[61] J. Zhang and L. Zhang. Efficient CUDA polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems. *Math. Probl. Eng.*, pages Art. ID 398438, 12, 2013.