RICE UNIVERSITY

# A Matlab Implementation of the Implicitly Restarted Arnoldi Method for Solving Large-Scale Eigenvalue Problems

by

**Richard J. Radke**

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

**Master of Arts**

Approved, Thesis Committee:

---

Danny C. Sorensen, Chairman
Professor of Computational and Applied Mathematics

---

C. Sidney Burrus
Professor of Electrical and Computer Engineering

---

Steven J. Cox
Associate Professor of Computational and Applied Mathematics

Houston, Texas

April, 1996

# A Matlab Implementation of the Implicitly Restarted Arnoldi Method for Solving Large-Scale Eigenvalue Problems

Richard J. Radke

## Abstract

This thesis describes a Matlab implementation of the Implicitly Restarted Arnoldi Method for computing a few selected eigenvalues of large structured matrices. Shift-and-invert methods allow the calculation of the eigenvalues nearest any point in the complex plane, and polynomial acceleration techniques aid in computing eigenvalues of operators which are defined by m-files instead of Matlab matrices. These new Matlab functions will be incorporated into the upcoming version 5 of Matlab and will greatly extend Matlab's capability to deal with many real-world eigenvalue problems that were intractable in version 4.

The thesis begins with a discussion of the Implicitly Restarted Arnoldi Method. The bulk of the thesis is a user's manual for the Matlab functions which implement this algorithm. The user's guide not only describes the functions' syntax and structure but also discusses some of the difficulties that were overcome during their development. The thesis concludes with several examples of the functions applied to realistic test problems which illustrate the versatility and power of this new tool.

# Acknowledgments

I would like to thank Dan Sorensen, Steve Cox, and Sid Burrus for their comments and support. Dan Sorensen did an excellent job of introducing me to the field of numerical linear algebra and helped me extensively throughout the writing of both the software and thesis. Sid Burrus did an equally excellent job of introducing me to digital signal processing and spent many hours giving me valuable advice about how to choose a graduate school. I owe much to Steve Cox for helping me through the process of becoming a graduate student at Rice. His encouragement and advice were invaluable.

I am grateful to the entire CAAM department at Rice for giving me the wonderful opportunity to earn a Master's degree as a fourth-year undergraduate student.

Very special thanks go to Chao Yang for TeX algorithm templates, test matrix help, many pointers to references, and wide-ranging linear algebra expertise. I expect he will be the next big name in numerical linear algebra.

Ivan Selesnick provided just what I was looking for in the way of FIR filter design code and was kind enough to spend time getting it to work for me.

I should thank Wanda Bussey and Steve Merrill for introducing me to eigenvalues and eigenvectors in high school; these outstanding teachers still inspire me today.

Warmest thanks to my dear friends Elizabeth Cole and Joel McFarland for their constant support and amazing patience. I probably spent as many hours complaining about this thesis as I did writing it, and neither of them slapped me even once!

This thesis is dedicated to my parents, Rita and Robert Radke, who have always encouraged me to do my best by example and not by demand. I hope that one day they will be able to explain to their friends what their son does for a living.

# Contents

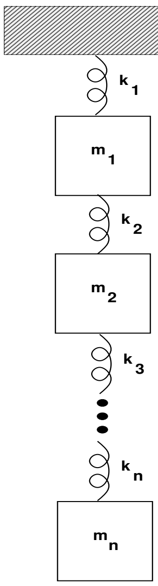
# Chapter 1

# Introduction

## 1.1 Why do we need eigenvalues?

How well will a building withstand an earthquake? What is the long-term behavior of an RLC circuit? How does heat flow through an irregularly shaped domain? The theory of eigenvalues and eigenvectors, or *spectral analysis*, can help answer these science and engineering questions, and many more.

Suppose a structural engineer must determine whether a certain building will withstand an earthquake. A natural way to model the building is as a large system of masses connected together by springs. The movement of each mass depends on the forces exerted on it by neighboring masses; masses further away in the structure have only an indirect impact. In some sense, all of the information about this mass-spring system can be expressed as a huge matrix in which the $(i,j)^{th}$ entry corresponds to the effect mass $i$ has on mass $j$ via a connecting spring.




$$\Rightarrow \begin{pmatrix} k_1+k_2 & -k_2 & 0 & 0 & \cdots & 0 \\ -k_2 & k_2+k_3 & -k_3 & 0 & \cdots & 0 \\ 0 & -k_3 & k_3+k_4 & -k_4 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & -k_{n-1} & k_{n-1}+k_n & -k_n \\ 0 & 0 & 0 & 0 & -k_n & k_n \end{pmatrix}$$

Figure 1.1: One-dimensional mass-spring system

This example illustrates an important and fortunate characteristic of many matrices which originate in real-world applications: they are very **sparse**. That is, the number of nonzero elements in the matrix is very small compared to the total number of matrix elements. In the above example, if the number of masses is a large number $m$, then the resulting matrix has at most $3m$ nonzero entries, a small number compared to the total number of entries, $m^2$.

Furthermore, this matrix is highly **structured**. That is, the nonzero entries are not scattered randomly throughout the matrix, but occur at predictable locations. Plotting the locations of the nonzero entries of the matrix produces:

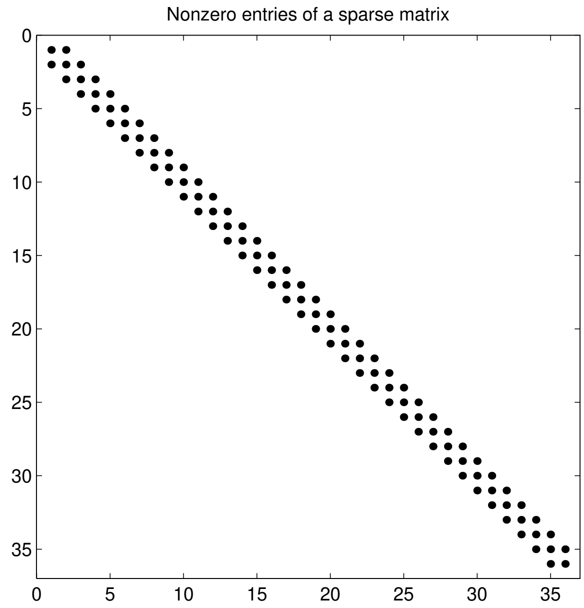



Figure 1.2: Nonzero entries of the mass-spring matrix

Many matrices which arise from engineering applications have some underlying structure. This structure allows us to optimize operations involving the matrix, such as multiplying the matrix by a vector. Efficient algorithms for matrix computations become especially important as the order of the matrix grows.

In the mass-spring system, a standard problem is to determine the behavior of the system given forces on each mass due to gravity, the springs, internal damping, and external conditions. When there is no outside force on the masses, the resulting behavior is termed *damped free vibration*. The masses will oscillate with a certain period, and the amplitude of the oscillations will decrease according to the resistance of the medium in which the system is placed. The situation becomes more interesting when there is an external force on each mass. If this force has the same period as the oscillations which would occur in the damped free vibration case, dangerous oscillations can occur whose amplitude increases without bound. A famous actual example occurred when a column of soldiers marched in step across a suspension bridge in England, causing the bridge to resonate and collapse. For this reason, soldiers now break step when crossing bridges [19, p. 309].

Many structural engineering models take the form of systems of masses and springs, and spectral analysis of these systems can provide important real-world information. For example, civil engineers need to calculate the vibrational modes (which correspond to eigenvalues) of models of skyscrapers, in order to determine if any of these modes lie in the earthquake band. Designers of luxury cars seek to develop car frames whose vibrational modes lie outside the frequencies induced by highway driving, in order to create a quieter ride. The portable compact disc player a jogger uses for running to music should be able to absorb the vibrations induced by running without affecting the quality of sound output. The mathematical formulation of these problems is to find the eigenvalues of a matrix which are closest to a point or region. In these examples, the frequencies corresponding to the earthquake band or road noise can be mapped to a region in the complex plane. To solve the problem, we need to model the building or car as a mass-spring system, construct the appropriate matrix, and determine whether or not any eigenvalues of this matrix approach the region of concern.

A different structural engineering problem, *buckling analysis*, involves determining the point at which a system will go from a stable state to an unstable one. For example, if a rod is compressed at both ends with increasing force, after a certain point it will bend or break. A certain mass placed at the top of a structure may have a stabilizing effect, while a heavier mass may cause the structure to collapse [19]. The structure seeks the state in which its internal energy is minimum. Finding the "natural" state of a system therefore involves an optimization problem. In one variable, we find candidates for minima of a function $f(x)$ by solving for the critical points where $f'(x) = 0$. If $x^*$ is a critical point and $f''(x^*) > 0$, then the critical point is a minimum. This strategy extends to the higher dimensional case; in this setting, the second derivative test takes the guise of checking to see if all of the eigenvalues of a certain matrix are positive (that is, if the matrix is *positive definite*). The structure goes from a stable state to an unstable one when one eigenvalue of this matrix becomes zero or negative. Therefore, the smallest eigenvalue is of the most interest; if this eigenvalue is negative, the structure will buckle. Spectral approaches are also important for more complicated buckling problems in which the structure may deform to one of several states. The calculations involved are known as *bifurcation analysis.*

Spectral analysis arises more generally in the study of stability of dynamical systems in the form $y' = F(y)$. In this situation, $y$ is some vector-valued function of time and $F$ could be a partial differential operator and is usually a nonlinear function of $y$. The standard technique, called *linearization* or *first-order perturbation*, is to find equilibrium points by solving $F(y) = 0$, and then to determine the stability of these points by considering the eigenvalues of a special matrix called the *Jacobian* of $F$ at the equilibria. An equilibrium is stable if all of the eigenvalues of the Jacobian at that point have negative real part. The transition from stability to instability is equivalent to an eigenvalue of the Jacobian crossing from the left half-plane to the right half-plane. In this situation, we need only concern ourselves with the eigenvalues of largest real part. If any of the rightmost eigenvalues occurs in the right half plane, the system is unstable. Only a few of the rightmost eigenvalues are needed, for if a system is unstable to a small perturbation, the effects of larger perturbations are irrelevant. In the stable case, we can approximate the long-term behavior of a system by analyzing the eigenvectors associated with these rightmost eigenvalues. This type of stability analysis arises in many fields, including computational chemistry, fluid dynamics, and the study of electric power systems.

A final common application of spectral analysis occurs when a continuous operator over a region of interest is discretized by imposing a fine mesh over the region. More accurate approximations to the infinite number of *eigenfunctions* associated with the operator and region can be obtained by further refining the mesh. The operator which is discretized varies from field to field. In quantum chemistry, the properties of electrons and other elementary particles are described by their wave functions $\psi$, which are solutions to the Schrödinger equation $H\psi = E\psi$. We shall see that the wave functions $\psi$ are exactly the eigenfunctions of the Hamiltonian operator, with corresponding eigenvalues given by the energy $E$. To solve the problem on a digital computer, the Hamiltonian $H$ is discretized; the eigenvectors of this matrix can be used to construct approximations to the eigenfunctions. These wave function approximations give information about quantal transition probabilities and on a larger scale, molecular motion.

Very **large** matrices often arise from spatial discretization problems in high dimensions. In the $n$-dimensional case, a grid size of $b$ bins per dimension produces a matrix of order $b^n$. To achieve engineering accuracy (three to four decimal places) of the results in three dimensions, it may be necessary to discretize each dimension with $\mathcal{O}(10^2)$ bins or more, giving rise to matrices with $n = \mathcal{O}(10^6)$. These matrices can be stored entirely in main memory on most powerful computer systems, but they cannot be factored.

The above examples introduce some of the many engineering contexts in which eigenvalues and eigenvectors arise. In Chapter 7, we shall return to these problems and demonstrate how to solve them.

## 1.2 The state of the art

Though we have simple algebraic expressions for the eigenvalues and eigenvectors of a matrix, the practical problem of finding eigenvalues is actually much harder than it looks. Computing the characteristic polynomial or the nullspace of a matrix is a process ill-suited to a digital computer. In fact, it turns out that it is not possible to develop *direct methods* for solving the eigenvalue problem which involve a finite number of operations to arrive at precise eigenvalues. Instead, we must rely on *iterative methods* which gradually refine the precision of approximations to the eigenvalues until the results are acceptably good.

The process is further complicated by the fact that the methods which are commonly applied to solve small, dense eigenvalue problems are not appropriate for the large, sparse, and structured problems that commonly arise from real-world engineering problems.

Numerical considerations, cases involving multiple or clustered eigenvalues, and roundoff errors produced by floating-point operations make the possibility of a "black box" algorithm for computing eigenvalues slim. In few situations does there exist an algorithm which will perform excellently no matter what problem the user supplies. Parameters may need to be set intelligently to get the best performance from an algorithm for a specific problem. Furthermore, the algorithms we will discuss converge much more quickly if the user is able to specify certain information about the matrix and perhaps guesses about the nature of the eigenvalues and eigenvectors. If the user has some intuition about what the results should be, this is a reasonable request. Several software packages to compute eigenvalues and eigenvectors are currently available for the user with a background in numerical linear algebra, notably the collection of Fortran routines ARPACK. This software package allows the user to allocate space for working vectors, adjust many algorithm parameters, and observe intermediate results of the iterations as they proceed [14].

However, we cannot expect every user to have a firm command of numerical linear algebra, a great deal of intuition about the problem, and a knowledge of many different programming languages. Typically, an engineer would prefer a tool which requires little thought and mathematical background to use and will deliver results in a reasonable amount of time without prompting or fuss.

The mathematical software Matlab is arguably the industry standard numerical analysis tool for both industrial and educational environments [15]. The software attempts to address the needs of both the research scientist and the working engineer by providing powerful, state-of-the-art algorithms whose details of implementation are transparent.

Today's computers have more memory and faster processors than ever before. It is becoming easier for users to construct problems which can be stored but cannot be solved by existing means. For example, users can store huge matrices of order 500 or more in Matlab version 4. It is intuitive to believe that if the problem can be posed within the existing Matlab framework, it can be solved. However, if a user attemps to find the eigenvalues of such a huge matrix using Matlab's `eig` command, the result will either be an `Out of memory` error message or a screen which stays blank for hours as the algorithm converges. Neither of these outcomes is desirable for or addressable by a working engineer.

The Implicitly Restarted Arnoldi Method is an algorithm well-suited to solving the types of eigenvalue problems discussed in the previous section. It is not merely a theoretical tool but the dominant method by which these real problems are solved.

The goal of my work over the previous year was to study the Implicitly Restarted Arnoldi Method and implement this algorithm in the Matlab programming language. The resulting Matlab functions are not a direct transcription of Fortran source code, but rather a translation both in programming language and philosophy. The Matlab implementation replaces the complex and lengthy function calls characteristic of Fortran routines with clean, intuitive syntax and an informative user interface. This new suite of functions greatly extends Matlab's capability to solve previously intractable eigenvalue problems.

We now turn to developing the mathematical theory needed to understand eigenvalues and eigenvectors and the Implicitly Restarted Arnoldi Method which we will use to compute them.

# Chapter 2

# Foundations of the QR Method

## 2.1 Notation

We review some basic notational conventions. Capital and lower case letters denote matrices and vectors, respectively. The identity matrix in $\mathcal{R}^{n\times n}$ is denoted by $I_n$, and the subscript is dropped when the dimension is clear. We use $e_j$ to denote the $j^{th}$ column of the identity matrix. The transpose of a vector $x$ is denoted by $x^T$ and $x^H$ denotes the complex conjugate of $x^T$. The nullspace of a matrix $A$ is denoted by $\mathcal{N}(A)$, and the determinant of the matrix is denoted $\det(A)$. The inverse of $A$ is denoted $A^{-1}$. $A(:,j)$ denotes the $j^{th}$ column of the matrix $A$; $A(i,:)$ denotes the $i^{th}$ row. $A(i, j_1 : j_2)$ denotes the vector composed of the $j_1^{st}$ through $j_2^{nd}$ entries in row $i$ of $A$; this extends in the natural way to $A(i_1 : i_2, j)$.

## 2.2 Eigenvalues and eigenvectors

Consider a matrix $A \in \mathcal{C}^{n\times n}$. We wish to find nonzero vectors $x \in \mathcal{C}^n$ such that $Ax = x\lambda$, where $\lambda \in \mathcal{C}$. Given a pair $(x, \lambda)$ satisfying this relationship, we call $x$ an *eigenvector* of the matrix $A$ and $\lambda$ the *eigenvalue* of $A$ corresponding to $x$. Algebraically, the eigenvalues are the roots of the $n^{th}$-order *characteristic polynomial* $p(\lambda) = \det(\lambda I - A)$, and any nonzero vector in the nullspace $\mathcal{N}(\lambda I - A)$ is an eigenvector corresponding to the eigenvalue $\lambda$. Even if the matrix $A$ has real valued entries, the eigenvalues and eigenvectors are not necessarily real. Only when $A$ is symmetric (or more generally, Hermitian) are the eigenvalues assured to be real.

We should also note that the eigenvalue $\lambda$ corresponding to the eigenvector $x$ can be computed using the *Rayleigh quotient*:

$$\lambda = \frac{x^T A x}{x^T x}.$$

## 2.3 Why use iterative methods?

Given a polynomial $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + x^n$, a matrix $P$ can be constructed such that the roots of the polynomial $p(x)$ are the eigenvalues of the "companion" matrix $P$. A sample companion matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{n-1} \end{pmatrix}$$

Galois theory tells us that the roots of a polynomial of degree greater than 4 cannot be expressed as combinations of radicals of rational functions of the polynomial coefficients [10].

Suppose we had a closed-form expression for the eigenvalues of a matrix that was a combination of radicals of rational functions of the matrix entries. Then this formula could be applied to the companion matrix of a polynomial to yield the roots of the polynomial. However, this contradicts the result from Galois theory, and it follows that there can exist no algorithm for finding eigenvalues (and thus, eigenvectors) of matrices of order greater than 4 that requires a finite number of additions and multiplications.

Therefore, procedures for calculating eigenvalues are all iterative in some aspect; at each iteration, we would generally like the precision of the eigenvalue-eigenvector approximations to increase. The user must specify how precise the approximation needs to be for the application at hand. At some point in each iteration, a test is applied to determine whether the precision of the current results is within the desired tolerance. Once convergence has occurred, the procedure terminates and the eigenpair approximations are returned.

We will begin with a simple iterative method used to find the eigenpair of a matrix corresponding to the eigenvalue of largest magnitude and extend this method to more powerful algorithms.

## 2.4 The power method

Suppose we have a matrix $A$ which is *simple*; that is, it has $n$ eigenvectors which form a basis for $\mathcal{C}^n$. We write the eigenvalues as $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, with associated eigenvectors $\{x_1, x_2, \ldots, x_n\}$. We suppose without loss of generality that $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$.

Since the eigenvectors of $A$ form a basis for $\mathcal{C}^n$, we can write any vector $v_0 \in \mathcal{C}^n$ as a linear combination of the eigenvectors:

$$v_0 = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n.$$

If we multiply this vector by the matrix $A$, we obtain:

$$\begin{aligned} Av_0 &= c_1 A x_1 + c_2 A x_2 + \cdots + c_n A x_n \\ &= c_1 \lambda_1 x_1 + c_2 \lambda_2 x_2 + \cdots + c_n \lambda_n x_n. \end{aligned}$$

Similarly, if we multiply $v_0$ by $A$ many times, we obtain:

$$\begin{aligned} A^k v_0 &= c_1 A^k x_1 + c_2 A^k x_2 + \cdots + c_n A^k x_n \\ &= c_1 \lambda_1^k x_1 + c_2 \lambda_2^k x_2 + \cdots + c_n \lambda_n^k x_n \\ &= \lambda_1^k (c_1 x_1 + c_2 \left(\frac{\lambda_2}{\lambda_1}\right)^k x_2 + \cdots + c_n \left(\frac{\lambda_n}{\lambda_1}\right)^k x_n). \end{aligned}$$

Suppose that $A$ has a *dominant eigenvalue*; that is, $|\lambda_1| > |\lambda_2|$. Then as $k \to \infty$, the fraction $(\frac{\lambda_i}{\lambda_1})^k \to 0$ for $i > 1$. Therefore, we have

$$\lim_{k \to \infty} \frac{A^k v_0}{\lambda_1^k} = c_1 x_1.$$

This suggests the following algorithm for computing the dominant eigenvalue and corresponding eigenvector of $A$:

**Algorithm 1**: Power method

**Input**: A simple matrix $A$ with a dominant eigenvalue
A random *starting vector* $v_0 \in \mathcal{C}^n$
A tolerance $\tau$

**Output**: An approximation $v$ to the dominant eigenvector $x_1$ of $A$.

1. **for** $i = 1, 2, 3, \ldots$ until $\|Av_i - v_i\lambda_i\| < \tau|\lambda_i|$
   1.1. Set $v_i \leftarrow (A^i v_0)/\lambda_1^i$;
   1.2. Set $\lambda_i = \frac{v_i^T A v_i}{v_i^T v_i}$;
2. **end**;

As written, this requires a priori knowledge of $\lambda_1$; however, since any scalar multiple of $x_1$ is still an eigenvector of $A$, we can choose any scaling of $v_i$ which is convenient. One commonly used scaling is to divide $v_i$ by its element of largest magnitude, so that its largest component is 1. The power method will then converge to the dominant eigenvector with largest element scaled to 1. We can also form $v_i$ by applying $A$ to $v_{i-1}$. Thus a more intelligent implementation of the power method is:

**Algorithm 2**: Smarter power method

**Input**: A simple matrix $A$ with a dominant eigenvalue
A random *starting vector* $v_0 \in \mathcal{C}^n$
A tolerance $\tau$

**Output**: An approximation $v$ to the dominant eigenvector $x_1$ of $A$.

1. **for** $i = 1, 2, 3, \ldots$ until $\|Av_i - v_i\lambda_i\| < \tau|\lambda_i|$
   1.1. Set $v_i \leftarrow Av_{i-1}$.
   1.2. Normalize $v_i$ so that the component of largest absolute value is 1.
   1.3. Set $\lambda_i = \frac{v_i^T A v_i}{v_i^T v_i}$;
2. **end**;

This algorithm will converge to the dominant eigenvector of $A$ as long as the starting vector $v_0$ has some component $c_1$ in the $x_1$ direction. In practice, even if $c_1$ is extremely small, the algorithm will converge; on a finite precision computer, $c_1$ on the order of roundoff error is often enough to guarantee convergence!

It should be noted that the criterion used for convergence is independent of the scaling of $A$, and that this stopping rule can actually be implemented without forming the matrix-vector product $Av_i$.

If there exists no dominant eigenvalue (for example, if $\lambda_1$ is half of a complex conjugate pair) the power method will not converge. Furthermore, if the dominant eigenvalue $\lambda_1$ is unique but $\frac{|\lambda_2|}{|\lambda_1|}$ is close to 1, then convergence of the power method will be very slow. Therefore it is desirable that the eigenvalues be *well-separated* instead of *clustered* in the spectrum.

It is important to note that the only time the matrix $A$ is used in the iteration is to form matrix-vector products. This is an operation which exploits the sparsity of $A$, for multiplying a vector by a matrix requires about $2 \cdot nnz$ operations, where $nnz$ is the number of non-zero entries in $A$.

Though the power method may seem useful only to find an eigenvector corresponding to a well-separated dominant eigenvalue, we can extend and generalize this functionality to an algorithm which finds several eigenvalues of a matrix located anywhere in the spectrum.

## 2.5 Finding several eigenvalues

The power method as written here will only converge to a single eigenvalue of the matrix $A$. There are several approaches to modifying the power method to find the $k$ dominant eigenvalues of $A$. One technique, *deflation*, is reasonably straightforward: once the eigenpair $(x_1, \lambda_1)$ is computed, a transformation is applied to the matrix $A$ to move $\lambda_1$ to the interior of the spectrum, so that the second largest eigenvalue $\lambda_2$ becomes the dominant eigenvalue of the transformed matrix. This process is repeated until the $k$ dominant eigenvalues have been found.

Of course, we can improve on this simple deflation scheme; finding one eigenpair at a time is not very efficient. We extend the power method by applying $A$ to a set of $k$ starting vectors at each iteration. When treated independently, the iterates will all converge to the dominant eigenvector. However, if we orthonormalize the $k$ vectors at each step, this set will converge to a basis for an invariant subspace of the matrix $A$ corresponding to the $k$ eigenvectors of largest magnitude. This method is called *subspace iteration.*

The Arnoldi Method is a powerful extension of subspace iteration and is the tool we will use to find $k$ eigenpairs of a matrix simultaneously. We will discuss it in Chapter 3.

## 2.6 Inverse iteration

One technique for finding the eigenvalue of smallest magnitude of $A$ is relatively straightforward. We consider a matrix $A$ with eigenpairs $\{(x_1, \lambda_1), (x_2, \lambda_2), \ldots, (x_n, \lambda_n)\}$. Then if $A^{-1}$ exists (equivalently, if no $\lambda_i = 0$),

$$A^{-1}(Ax_i) = A^{-1}(\lambda_i x_i) \qquad \Rightarrow \qquad A^{-1}x_i = \frac{1}{\lambda_i}x_i.$$

That is, the eigenpairs of $A^{-1}$ are $\{(x_1, \frac{1}{\lambda_1}), (x_2, \frac{1}{\lambda_2}), \ldots, (x_n, \frac{1}{\lambda_n})\}$.

Therefore, we can find the eigenpair corresponding to the eigenvalue of smallest magnitude of $A$ by applying the power method to $A^{-1}$. The result is the eigenpair $(x, \mu)$, from which we can recover the smallest eigenvalue by taking $\lambda = \frac{1}{\mu}$.

## 2.7 Shift-and-invert

We can extend this idea further. If we consider the matrix $A - \sigma I$, we can see that

$$\begin{aligned}(A - \sigma I)x_i &= Ax_i - \sigma x_i \\ &= \lambda_i x_i - \sigma x_i \\ &= (\lambda_i - \sigma)x_i\end{aligned}$$

which shows that the eigenpairs of $A - \sigma I$ are $\{(x_1, \lambda_1 - \sigma), (x_2, \lambda_2 - \sigma), \ldots, (x_n, \lambda_n - \sigma)\}$.

Putting this argument and inverse iteration together, we find that the eigenpairs of $(A - \sigma I)^{-1}$ are $\{(x_1, \frac{1}{\lambda_1 - \sigma}), (x_2, \frac{1}{\lambda_2 - \sigma}), \ldots, (x_n, \frac{1}{\lambda_n - \sigma})\}$.

The eigenvalue of the original matrix $A$ that is closest to $\sigma$ corresponds to the eigenvalue of largest magnitude of the shifted and inverted matrix $(A - \sigma I)^{-1}$. Thus, a standard technique for finding the eigenpair of $A$ corresponding to the eigenvalue closest to a shift $\sigma$ is to apply the power method to the shifted problem $(A - \sigma I)^{-1}$ to obtain the eigenpair $(x, \mu)$. Then we recover the eigenvalue $\lambda$ of the original problem by the easily computable transformation $\lambda = \frac{1}{\mu} + \sigma$. This method is called *shift-and-invert*.

In practice the matrix $(A - \sigma I)^{-1}$ is never formed; to calculate the product $y = (A - \sigma I)^{-1}x$ we simply solve the system of linear equations $(A - \sigma I)y = x$.

We can develop other transformations of $A$ which allow us to compute eigenvalues located near certain regions of the spectrum; however, it is important to seek transformations which exploit the sparsity of $A$. We will discuss a class of these transformations, *polynomial acceleration techniques*, in Chapter 4.

## 2.8 The QR Method

We now discuss the classical QR method which provides the basis for the algorithms used to compute eigenvalues and eigenvectors of dense problems, and proceed to develop the Arnoldi method which is well-suited to computing eigenpairs of large sparse or structured problems. We shall see that both algorithms are closely related to the power method and its variants discussed above.

We are interested in the eigenvectors and eigenvalues of a matrix $A$. Basically, we will derive a series of similarity transformations $A_{i+1} = Q^T A_i Q$ so that at each step, the subdiagonal elements of the $A_i$ are made smaller until they are negligible; at this point, the matrix is effectively triangular and we can read the eigenvalues off the diagonal.

In general, computing similarity transformations is quite expensive, so it is advantageous to first reduce the matrix $A$ to a form which makes these transformations easy to compute. It is most common to reduce $A$ to Hessenberg form, in which all elements below the first subdiagonal are zero. Therefore, in our exposition, we will suppose we can obtain the reduction to Hessenberg form $AV = VH$ by means of Householder or Givens rotations [26, pp. 328-337].

We present the following algorithm, known as the *QR method*:

**Algorithm 3**: $QR$ method
**Input**: $(A, V, H)$ with $AV = VH, V^H V = I$.
**Output**: $(V, H)$ such that $AV = VH, V^H V = I$ and $H$ is upper triangular.

1. **for** $i = 1, 2, 3, ...$ until "convergence"
   1.1. Factor $H_i = Q_i R_i$;
   1.2. $H_{i+1} \leftarrow R_i Q_i$ ; $V_{i+1} \leftarrow V_i Q_i$;
2. **end**;

The QR method is a variant of subspace iteration. As the QR method progresses, the eigenvalues of the matrix $A$ are approximated by the diagonal entries of $H$. As the subdiagonal entries of $H$ decrease in magnitude, the diagonal entries of $H$ become better approximations to the eigenvalues of $A$. The eigenvectors can be recovered later if the product of the orthogonal matrices $Q_i$ is accumulated and stored at each step.

Of greater use is the *explicitly shifted* QR method, in which a shift $\sigma_i$ is introduced at each iteration:

**Algorithm 4**: Explicitly Shifted $QR$ method

**Input**: $(A, V, H)$ with $AV = VH, V^H V = I$.
A tolerance $\tau$.
**Output**: $(V, H)$ such that $AV = VH, V^H V = I$ and $H$ is upper triangular.

1. **for** $i = 1, 2, 3, \ldots$ until each subdiagonal element of $H < \tau$.
   1.1. Select a shift $\sigma_i$;
   1.2. Factor $H_i - \sigma_i I = Q_i R_i$;
   1.3. $H_{i+1} \leftarrow R_i Q_i + \sigma_i I$; $V_{i+1} \leftarrow V_i Q_i^H$;
2. **end**;

Note that

$$
\begin{aligned}
H_{i+1} &= Q_i^H (H_i - \sigma_i) Q_i + \sigma_i I \\
&= Q_i^H H_i Q_i.
\end{aligned}
$$

Therefore, each step is a similarity transformation of the original matrix $H$, which itself is similar to $A$. It can be shown that the $QR$ method effectively performs the power method on the first column of $V$ and inverse iteration on the last column of $V$; this leads to the rapid convergence of the algorithm.

The algorithm can be implemented without explicitly adding and subtracting the shift $\sigma_i$ from the diagonal; this variant is known as the *implicitly shifted* QR method. Its main advantage is the ability to apply a pair of complex conjugate shifts while staying in real arithmetic, but there are gains in numerical stability as well.

If $\sigma_i = \sigma$ is a good estimate of an eigenvalue, then the $(n, n-1)$ entry of $H_i$ will converge to zero very quickly, and the $(n, n)$ entry of $H_i$ will converge to the eigenvalue closest to $\sigma$. Once this convergence has occured, the problem can be deflated to a smaller problem of order $n - 1$. In practice, the QR shift $\sigma_i$ can be taken as the rightmost diagonal element of $H_i$, once the corresponding subdiagonal element is somewhat small. There are several other ways to choose the QR shifts to accelerate convergence and to deflate the problem, but they are beyond the scope of this thesis. See [26] and [28] for more details.

The `eig` command in Matlab calls an efficient version of the QR method to find eigenvalues and eigenvectors of a real matrix. For a complex matrix or a generalized eigenvalue problem (see section 3.5), a more complicated algorithm called the *QZ method* is used.

# Chapter 3

# The Implicitly Restarted Arnoldi Method

## 3.1 Motivation

The QR method can be used to compute the eigenvalues and eigenvectors of a matrix, but there are several drawbacks [20, p. 82]:

1. In general, the transformations of the QR method destroy the sparsity and structure of the matrix.
2. The QR method is ill-suited to calculating some, but not all, of the eigenvalues.
3. If eigenvector information is also desired, either the original matrix or the QR-transformations have to be preserved.

Therefore, the QR method is not appropriate for finding $k$ selected eigenpairs of a large sparse matrix.

In some sense, the power method "throws away" potentially useful spectral information during the course of the iteration. At the $k^{th}$ iteration, we overwrite the vector $A^{k-1}v_0$ with $A^k v_0$, where $v_0$ is the starting vector for the algorithm. However, it turns out to be useful to keep the previous vector instead of overwriting it, and by extension to keep the entire set of previous vectors $\{v_0, Av_0, A^2v_0, \cdots, A^{k-1}v_0\}$. We call the subspace

$$\mathcal{K}_k(A, v_0) = \mathrm{Span}\{v_0, Av_0, A^2v_0, \cdots, A^{k-1}v_0\}$$

the $k^{th}$ *Krylov subspace* corresponding to $A$ and $v_0$. Methods which use linear combinations of vectors in this space to extract spectral information are called *Krylov subspace* or *projection* methods.

The basic idea is to construct approximate eigenvectors in the Krylov subspace $\mathcal{K}_k(A, v_0)$. We define a *Ritz pair* as any pair $(x_i, \lambda_i)$ that satisfies the *Galerkin condition*

$$v^T(Ax_i - \lambda_i x_i) = 0 \qquad \forall v \in \mathcal{K}_k(A, v_0).$$

That is, the Ritz pair satisfies the eigenvector-eigenvalue relationship in the projection onto a smaller space. We only hope that the component orthogonal to the space is sufficiently small to make the Ritz pair a good approximation to an eigenpair of $A$.

## 3.2 Arnoldi factorizations

We define a *k-step Arnoldi factorization* of $A \in \mathcal{C}^{n\times n}$ as a relationship of the form

$$AV = VH + fe_k{}^T$$

where $V \in \mathcal{C}^{n\times k}$ has orthonormal columns, $V^H f = 0$, and $H \in \mathcal{C}^{k\times k}$ is upper Hessenberg with a non-negative subdiagonal. If the matrix $A$ is Hermitian, then the relationship is called a *k-step Lanczos factorization*, and the upper Hessenberg matrix $H$ is actually real, symmetric, and tridiagonal.

Note that if $(y, \theta)$ is an eigenpair of $H$, then $x = Vy$ satisfies the relation

$$\begin{aligned} \|Ax - x\theta\| &= \|AVy - Vy\theta\| \\ &= \|(AV - VH)y\| \\ &= \|fe_k{}^T y\| \\ &= \beta|e_k{}^T y| \end{aligned}$$

where $\beta = \|f\|$. Since $V^T f = 0$, it easily follows that $\theta$ is a *Ritz value* and $x$ a corresponding *Ritz vector*. The central idea behind the Arnoldi factorization is to construct eigenpairs of the large matrix $A$ from the eigenpairs of the small matrix $H$. We assume that $k \ll n$ so that the eigenpairs of $H$ can be computed by conventional (dense) means. The goal is to drive $|e_k^T y| \rightarrow 0$ , so that the Ritz pair $(x, \theta)$ well-approximates an eigenpair of $A$. The term $\beta|e_k{}^T y|$ is called the *Ritz estimate* of the pair $(x, \theta)$, and describes the goodness of the eigenpair approximation. Of course, when $f = 0$, $V$ is an invariant subspace of $A$ and the Ritz values and vectors are precisely eigenvalues and eigenvectors of $A$.

We can extend a $k$-step Arnoldi factorization to a $(k+1)$-step Arnoldi factorization using the following algorithm:

**Algorithm 5**: One step extension of $k$-step Arnoldi factorization
**Input**: $(V_k, H_k, f_k)$ such that $AV_k = V_k H_k + f_k e_k^T$
**Output**: $(V_{k+1}, H_{k+1}, f_{k+1})$ such that $AV_{k+1} = V_{k+1} H_{k+1} + f_{k+1} e_{k+1}^T$

1. $\beta_k = \|f_k\|$; $\quad v \leftarrow f_k/\beta_k$;
2. $V_{k+1} \leftarrow (V_k, v)$; $\quad H_{k+1} \leftarrow \begin{pmatrix} H_k \\ \beta_k e_k^T \end{pmatrix}$;
3. $z \leftarrow Av_{k+1}$;
4. $h_{k+1} \leftarrow V_{k+1}^T z$; $\quad f_{k+1} \leftarrow z - V_{k+1} h_{k+1}$;
5. $H_{k+1} \leftarrow (H_{k+1}, h_{k+1})$;
6. **end**;

The above scheme can be extended naturally to extend a $k$-step Arnoldi factorization to a $(k+p)$-step Arnoldi factorization:

**Algorithm 6**: $p$-step extension of $k$-step Arnoldi factorization
**Input**: $(V_k, H_k, f_k)$ such that $AV_k = V_k H_k + f_k e_k^T$
**Output**: $(V_{k+p}, H_{k+p}, f_{k+p})$ such that $AV_{k+p} = V_{k+p} H_{k+p} + f_{k+p} e_{k+p}^T$

1. **for** $i = k, k+1, \ldots, k+p-1$
   1.1. $\beta_i = \|f_i\|$; $\quad v \leftarrow f_i/\beta_i$;
   1.2. $V_{i+1} \leftarrow (V_i, v)$; $\quad H_{i+1} \leftarrow \begin{pmatrix} H_i \\ \beta_i e_i^T \end{pmatrix}$;
   1.3. $z \leftarrow Av_{i+1}$;
   1.4. $h_{i+1} \leftarrow V_{i+1}^T z$; $\quad f_{i+1} \leftarrow z - V_{i+1} h_{i+1}$;
   1.5. $H_{i+1} \leftarrow (H_{i+1}, h_{i+1})$;
2. **end**;

In exact arithmetic, the columns of $V$ will form an orthonormal basis for the Krylov subspace; however, in finite precision arithmetic, explicit reorthogonalization of the columns of $V$ is necessary. This can be accomplished by the Gram-Schmidt process with $t$ steps of iterative refinement [5]. This addition to the algorithm is accomplished by deleting step 1.5. in the above algorithm and adding the steps:

1.5. **for** $j = 1, 2, \ldots t$

1.5.1. $s = V_{i+1}^T f_{i+1}$

1.5.2. $f_{i+1} = f_{i+1} - V_{i+1} s$

1.5.3. $h_{i+1} = h_{i+1} + s$

1.6. $H_{i+1} \leftarrow (H_{i+1}, h_{i+1})$

In practice, one step of iterative refinement is usually sufficient to ensure orthogonality to machine precision.

It should be clear that an Arnoldi factorization is entirely dependent on the choice of starting vector $v_0$. In fact, it turns out that the factorization is uniquely determined by the choice of $v_0$, up to the point when a subdiagonal element of $H$ is zero. At this point an invariant subspace has been computed and the factorization continues with a new choice of starting vector.

## 3.3 Implicit restarting

For the large structured eigenvalue problems which arise in engineering contexts, the desired eigenvalues generally have special properties. For example, the user may require the $k$ eigenvalues of largest real part for a stability analysis, or the $k$ eigenvalues nearest a point in the complex plane for a vibrational analysis. In general, we would like the starting vector $v_0$ used to begin the Arnoldi factorization to be rich in the subspace spanned by the desired eigenvectors with very small components in the direction of the other eigenvectors. In some sense, as we get a better idea of what the desired eigenvectors are, we would like to adaptively refine $v_0$ to be a linear combination of the approximate eigenvectors and restart the Arnoldi factorization with this new vector instead. A convenient and stable way to do this without explicitly computing a new Arnoldi factorization is given by the implicitly restarted Arnoldi method, based on the implicitly shifted QR factorization. The implicitly restarted Arnoldi method is fully developed in [22].

Consider an $m$-step Arnoldi factorization of the form:

$$AV = VH + fe_m{}^T$$

to which we want to apply the (possibly complex) shift $\sigma$. Since $H \in \mathcal{C}^{m \times m}$ is small, we can factor $H - \sigma I = QR$. We have the following equivalent statements:

1. $AV = VH + fe_m^T$
2. $(A - \sigma I)V - V(H - \sigma I) = fe_m^T$
3. $(A - \sigma I)V - VQR = fe_m^T$
4. $(A - \sigma I)VQ - VQRQ = fe_m^T Q$
5. $A(VQ) - (VQ)(RQ + \sigma I) = fe_m^T Q$
6. $AV_+ = V_+ H_+ + fe_m^T Q$

$V_+$ has orthonormal columns since it is the product of $V$ and an orthogonal matrix $Q$. It also turns out $H_+$ is upper Hessenberg [26, pp. 355-361]. Therefore, shifting by $\sigma$ does not disturb the structure of the Arnoldi factorization. The result of these operations is that the first column of $V_+$ is $(A - \sigma I)v_1$, where $v_1$ is the first column of $V$.

The idea of the method is to extend a $k$-step Arnoldi factorization

$$AV_k = V_k H_k + f_k e_k^T$$

to a $(k+p)$-step Arnoldi factorization

$$AV_{k+p} = V_{k+p} H_{k+p} + f_{k+p} e_{k+p}^T.$$

Then $p$ implicit shifts are applied to the factorization, resulting in the new factorization

$$AV_+ = V_+ H_+ + f_{k+p} e_{k+p}^T Q$$

where $V_+ = V_{k+p}Q$, $H_+ = Q^H H_{k+p} Q$, and $Q = Q_1 Q_2 \cdots Q_p$, where $Q_i$ is associated with factoring $(H - \sigma_i I) = Q_i R_i$. It turns out that the first $k-1$ entries of $e_{k+p}Q$ are zero, so that a new $k$-step Arnoldi factorization can be obtained by equating the first $k$ columns on each side:

$$AV_k^+ = V_k^+ H_k^+ + f_k^+ e_k^T$$

We can iterate the process of extending this new $k$-step factorization to a $(k+p)$-step factorization, applying shifts, and condensing. The payoff is that every iteration implicitly applies a $p^{th}$ degree polynomial in $A$ to the initial vector $v_0$. The roots of the polynomial are the $p$ shifts that were applied to the factorization. Therefore, if we choose as the shifts $\sigma_i$ eigenvalues that are "unwanted", we can effectively filter the starting vector $v_0$ so that it is rich in the direction of the "wanted" eigenvectors.

There are several strategies for selecting the shifts $\sigma_i$. A useful method known as the Exact Shift Strategy takes the shifts as the $p$ eigenvalues of $H_{k+p}$ that are furthest away from the wanted eigenvalues. For example, if the desired eigenvalues are the $k$ eigenvalues of largest magnitude, the eigenvalues of $H_{k+p}$ are sorted with respect to magnitude and the $p$ eigenvalues of smallest magnitude are used as shifts. Or, if the desired eigenvalues are the $k$ eigenvalues closest to 5, the $p$ eigenvalues of $H_{k+p}$ furthest away from 5 are used as shifts.

We are now ready to present the full implicitly restarted Arnoldi method:

**Algorithm 7**: Implicitly restarted Arnoldi method

**Input**: The matrix $A$ whose eigenpairs are to be computed
$k$, the number of eigenpairs to be computed
$p$, the number of implicit shifts to apply to the factorization at each iteration
$S$, a sort criterion which determines which are the "wanted" eigenvalues
A starting vector $v_0$
A tolerance $\tau$

**Output**: $\{(x_1, \lambda_1), (x_2, \lambda_2), \ldots, (x_k, \lambda_k)\}$,
approximations to the $k$ wanted eigenvalues of $A$.

1. Using $v_0$ as a starting vector, generate a $k$-step Arnoldi factorization
$$AV = VH + fe_k^T.$$
2. **for** $i = 1, 2, \ldots$ until $\|Ax_i - \lambda_i x_i\| < \tau \;\; \forall i = 1 \ldots k$

 2.1. Extend the $k$-step Arnoldi factorization to a $k+p$ step Arnoldi factorization
$$AV = VH + fe_{k+p}^T.$$
 2.2. Let $q = e_{k+p}$.
 2.3. Sort the eigenvalues of $H$ from best to worst according to the sort criterion $S$ and take $\{\sigma_1, \ldots, \sigma_p\}$ to be the $p$ worst eigenvalues.
 2.4. **for** $j = 1, 2, \ldots p$
  2.4.1. Factor $H - \sigma_j I = QR$.
  2.4.2. $H \leftarrow Q^H H Q$.
  2.4.3. $V \leftarrow VQ$.
  2.4.4. $q \leftarrow q^H Q$.
 2.5. $f \leftarrow V(:, k+1) \cdot H(k+1, k) + f \cdot q(k)$.
 2.6. Take the first $k$ columns on each side of the factorization to get $V = V(:, 1:k)$, $H = H(1:k, 1:k)$.
 2.7. Take as eigenpair approximations $(x_i, \lambda_i)$ the Ritz pairs of the problem.
3. **end**;

We should remark that step 2.4.2 is accomplished by what is called a *bulge chase* [23], and that the convergence criterion is evaluated using the Ritz estimates instead of actually forming the Ritz pairs and computing matrix-vector products.

## 3.4 Shift-and-invert

The Arnoldi method as written converges to the $k$ eigenvalues of the largest magnitude. Therefore, if we are trying to find eigenvalues closest to a shift in the interior of the spectrum, convergence may be quite slow. The eigenvectors corresponding to the eigenvalues of largest magnitude will continue to resurface despite our attempts (via implicit restarting) to eliminate them from the Arnoldi basis. In fact, the accumulated roundoff error during one iteration in the direction of the eigenvector of largest magnitude is enough to cause this eigenvector to resurface in the next iteration! Therefore, we use the same shift-and-invert techniques discussed in Section 2.7 in our implementation of the implicitly restarted Arnoldi method. That is, if we wish to compute the $k$ eigenvalues closest to a shift $\sigma$, we instead compute the $k$ eigenvalues $\{\mu_1, \mu_2, \ldots, \mu_k\}$ of largest magnitude of $(A - \sigma I)^{-1}$ and recover the desired eigenvalues using the transformation $\lambda_i = \frac{1}{\mu_i} + \sigma$.

## 3.5 The generalized eigenvalue problem

The Arnoldi method can also be used to solve the *generalized eigenvalue problem* $Ax = \lambda Bx$. Structural engineering design problems are often posed in this form; in this context, $A$ is called the *stiffness matrix* and $B$ is called the *mass matrix*. The pair $(A, B)$ is often referred to as a *matrix pencil*.

### 3.5.1 Complexities

The generalized eigenvalue problem is substantially more complex than the standard eigenvalue problem. Consider the following examples:

- $A = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$

  Here, the generalized eigenpairs are $\{(\begin{pmatrix} 1 \\ i \end{pmatrix}, i), (\begin{pmatrix} 1 \\ -i \end{pmatrix}, -i)\}$

  Thus we see that even if $A$ and $B$ are symmetric (or Hermitian) , the generalized eigenvalues of the pencil may not be real [19, p. 284].

- $A = \begin{pmatrix} -1 & 1 \\ 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 & 0 \\ -1 & 1 \end{pmatrix}$.

  In this case, $A$ and $B$ share a common null vector $x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Therefore $Ax = Bx = 0$ and any scalar $\lambda$ is a generalized eigenvalue corresponding to $x$. Note that this case occurs if and only if $A$ and $B$ are both singular with $\mathcal{N}(A) \cap \mathcal{N}(B) \neq \emptyset$.

- $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$.

  While $x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is an eigenvector of the pencil with eigenvalue 1, the second eigenvector corresponds to what is sometimes called an "infinite eigenvalue". This terminology is used since we require the pencil to have $n$ eigenvalues, counting multiplicity, when the pencil is of order $n$. This case can arise when $A$ or $B$ is singular.

### 3.5.2 Reductions to standard form

However, the situation is not always so bad. For the remainder of the exposition, we suppose $B$ is a symmetric matrix. We can easily transform the generalized eigenvalue problem into a standard eigenvalue problem if $B$ is nonsingular; in this case, we have:

$$Ax = \lambda Bx \qquad \Rightarrow \qquad B^{-1}Ax = \lambda x.$$

We can then apply the Arnoldi method to $B^{-1}A$. In practice, this matrix is never formed since typically symmetry, structure, and sparsity are lost [18, p. 309]. Instead, to compute $y = B^{-1}Ax$, we follow the steps:

1. Compute $u = Ax$

2. Solve $By = u$

If $B$ is positive definite, we can compute the Cholesky factorization $B = LL^T$, where $L$ is a lower triangular matrix. We can then transform the problem into the standard eigenvalue problem $L^{-1}AL^{-T}y = \lambda y$, where $y = L^T x$.

It should be noted that in the generalized eigenvalue problem, the eigenvectors are no longer orthogonal. However, they are orthogonal with respect to the $B$ *inner* *product* $(x, y)_B = y^H Bx$; that is, if $x$ and $y$ are eigenvectors of the matrix pencil $(A, B)$, then $y^H Bx = 0$. This is true because the matrix $B^{-1}A$ (whose eigenpairs are the eigenpairs of the pencil) is self-adjoint with respect to the $B$ inner product [18, p. 316].

### 3.5.3 Shift-and-invert techniques

Suppose we wish to find the generalized eigenvalues of a pencil closest to a shift $\sigma$. Then we have:

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

Thus a standard method of finding the $k$ generalized eigenvalues of the pencil closest to $\sigma$ is to compute the $k$ eigenvalues of largest magnitude of the matrix $C = (A - \sigma B)^{-1}B$ using the Arnoldi method, and recover the eigenvalues of the original problem using the standard transformation. Again, the matrix $C$ is never explicitly formed; instead we use the following procedure to compute the matrix-vector product $y = Cx$:

1. Factor $(A - \sigma B) = LU$
2. Compute $u = Bx$
3. Solve $Ld = u$
4. Solve $Uy = d$

Since $L$ and $U$ are logically triangular, steps 3 and 4 can be done quickly by backwards substitution. The $LU$ factorization in step 1 only needs to be done once and can be stored for use in successive iterations. Furthermore, we can compute the $LU$ factorization in a way that takes advantage of the sparsity of $A$ and $B$.

# Chapter 4

# Polynomial acceleration

In Section 2.7, we proved that if the matrix $A$ has eigenvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and corresponding eigenvectors $\{x_1, x_2, \ldots, x_n\}$, then the eigenvalues of $(A - \sigma I)$ are $\{\lambda_1 - \sigma, \lambda_2 - \sigma, \ldots, \lambda_n - \sigma\}$, and the eigenvectors are the same as the eigenvectors of $A$. It is also important to note that the eigenvalues of $A^k$ are $\{\lambda_1^k, \lambda_2^k, \ldots, \lambda_n^k\}$ and the eigenvectors are the same as the eigenvectors of $A$; the result follows by a straightforward induction on $k$.

Now we prove a more powerful result. Consider the following matrix polynomial in A:

$$P = p(A) = a_n A^n + a_{n-1} A^{n-1} + \cdots + a_1 A + a_0 I.$$

If we apply $P$ to an eigenvector of $A$, we find:

$$\begin{aligned} Px_i &= p(A)x_i \\ &= a_n A^n x_i + a_{n-1} A^{n-1} x_i + \cdots + a_1 A x_i + a_0 x_i \\ &= a_n \lambda_i^n x_i + a_{n-1} \lambda_i^{n-1} x_i + \cdots + a_1 \lambda_i x_i + a_0 x_i \\ &= p(\lambda_i) x_i \end{aligned}$$

That is, the eigenvalues of $p(A)$ are $\{p(\lambda_1), p(\lambda_2), \ldots, p(\lambda_n)\}$, and the eigenvectors are the original $\{x_1, x_2, \ldots, x_n\}$.

Worthwhile gains in eigenvalue computation can be made by applying the Arnoldi method to polynomials in the input matrix $A$. These techniques are collectively known as *polynomial acceleration* methods. For the moment, we will confine our attention to a symmetric matrix $A$, whose eigenvalues are real. Suppose the eigenvalues of interest are contained within some set $\mathcal{S} \in \mathcal{R}$. We let $p$ be a polynomial whose value is large over the set $\mathcal{S}$ but small over the rest of the spectrum of $A$. Then if we let $P = p(A)$ and search for the eigenvalues of largest magnitude of $P$, we will recover the eigenvectors of $A$ corresponding to eigenvalues in $\mathcal{S}$ first, and the "unwanted" eigenvalues outside of $\mathcal{S}$ will be damped out of the space spanned by the Arnoldi vectors. The normalized eigenvectors of $P$ will be eigenvectors of $A$, and we can recover the eigenvalues by using Rayleigh quotients.

## 4.1 Chebyshev polynomials

Chebyshev polynomials are a class of functions known in numerical analysis for solving the optimization problem

$$\min_{p \in \mathcal{P}_m, p(1)=1} \max_{t \in [-1,1]} |p(t)|$$

where $\mathcal{P}_m$ is the set of all polynomials of degree $m$ [19, p. 143].

The $m^{th}$ Chebyshev polynomial has the form $C_m(x) = \cos(m \cos^{-1}(x))$. Although it may not be obvious that this cosine formula produces a polynomial in $x$, it can be shown that the Chebyshev polynomials obey the three-term recursion:

$$\begin{aligned} C_0(x) &= 1 \\ C_1(x) &= x \\ C_{m+1}(x) &= 2xC_m(x) - C_{m-1}(x). \end{aligned}$$

In this form, the polynomial character of $C_m(x)$ is clear.

The normalized Chebyshev polynomial of degree $m$ oscillates $m-1$ times between -1 and 1 in the interval [-1,1], and then increases without bound outside of [-1,1]. The rate of increase of the Chebyshev polynomial outside the interval is quite large [11, p. 285].

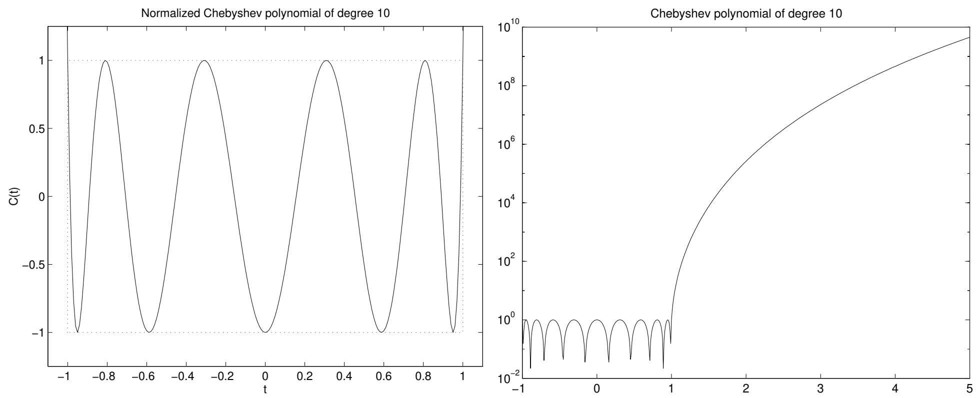



Figure 4.1: a. Equiripple behavior of the Chebyshev polynomial in [-1,1]
b. Rapid increase of the Chebyshev polynomial outside of [-1,1]

### 4.1.1 Computing eigenvalues of largest real part

Because of their rapid increase outside of [-1,1] and equiripple behavior within [-1,1], the Chebyshev polynomials are natural candidates for polynomial accelerants when we seek the eigenvalues of largest real part of a symmetric matrix $A$. The goal is to map the unwanted left side of the spectrum of $A$ into the equiripple region of a Chebyshev polynomial. Then the wanted eigenvalues on the right side of the spectrum will be mapped into the region of high increase outside the equiripple region. This process not only separates the wanted eigenvalues from the unwanted eigenvalues, but also well-separates the eigenvalues within the wanted region.

The complete procedure is:

**Algorithm 8**: Chebyshev polynomial acceleration

**Input**: A matrix $A$ for which to compute the $k$ eigenvalues of largest real part
and corresponding eigenvectors
An estimate $[lbd, ubd]$ of an interval containing the spectrum of $A$
A parameter $\rho$ which is the fraction of the spectrum
to map into the interval [-1,1]
The degree $m$ for the polynomial accelerant
$C_m$, the normalized Chebyshev polynomial of degree $m$

**Output**: $\{(x_1, \lambda_1), (x_2, \lambda_2), \ldots, (x_k, \lambda_k)\}$,
approximations to the $k$ wanted eigenvalues of $A$.

1. Determine the linear mapping $M$ which takes $lbd \rightarrow -1$, $lbd + \rho(ubd - lbd) \rightarrow 1$.
   Specifically, the mapping is $M(x) = ax + b$,
   where $a = \frac{2}{(1-\rho)(ubd-lbd)}$ and $b = -1 - a(lbd)$.
2. Define $P = C_m(M(A))$.
3. Compute the $k$ largest eigenvalues and corresponding eigenvectors of $P$,
   $\{(x_1, \mu_1), (x_2, \mu_2), \ldots, (x_k, \mu_k)\}$.
4. Let $\lambda_i = \frac{x_i^T A x_i}{x_i^T x_i}$ for $i = 1, \ldots, k$.
5. **end**;

The matter of computing and applying the Chebyshev polynomial to the matrix $M(A)$ is accomplished by means of the three-term recurrence mentioned above.

### 4.1.2 Computing eigenvalues of smallest real part

We can use the left-hand side of an even-order Chebyshev polynomial to acclerate the computation of the eigenvalues of a symmetric matrix of smallest real part. The procedure outlined in Algorithm 8 is unchanged except for one step:

1. Determine the linear mapping $M$ which takes $ubd - \rho(ubd - lbd) \to -1$, $ubd \to 1$. Specifically, the mapping is $M(x) = ax + b$, where $a = \frac{2}{(1-p)(ubd-lbd)}$ and $b = 1 - a(ubd)$.

## 4.2 Applications of FIR filter design to polynomial acceleration

The use of polynomial acceleration is not limited to calculating the leftmost or rightmost eigenvalues of a matrix. By choosing the correct polynomial, one can also enhance the eigenvalues around a shift in the middle of the spectrum. It is instructuve to pose this problem in the context of *FIR filter design*. Here we use the standard engineering notation $j = \sqrt{-1}$.

### 4.2.1 Overview of FIR filters

A *digital filter* is a system that takes a discrete input vector $x(n), n = 1, 2, \ldots$, sometimes called the *signal*, and returns a discrete output vector $y(n), n = 1, 2, \ldots$.


Figure 4.2: Discrete-time digital filter

Suppose the results of applying the system to the inputs $x_1(n)$ and $x_2(n)$ are the outputs $y_1(n)$ and $y_2(n)$ respectively. We call the system *linear* when the result of applying the system to the input $c_1x_1(n) + c_2x_2(n)$ is the output $c_1y_1(n) + c_2y_2(n)$. Linearity will allow us to express any input as a combination of simpler components whose responses to the system are known. The response of the system to the new input can then be constructed as a combination (or *superposition*) of the simpler components' responses.

A system is called *time-invariant* if when the input is shifted by a time step $M$, the output is shifted by the same time step $M$. That is, the response of the system to $x(n-M)$ is $y(n-M)$. Time-invariance is important because it guarantees that a system will respond the same way to a specific input no matter when the input is applied.

We define the *digital impulse function*

$$\delta(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{elsewhere.} \end{cases}$$

The *impulse response h(n)* is simply defined as the result of applying the system to $\delta(n)$.

It is then a straightforward matter to define the response of a linear, time-invariant system to any signal $x(n)$ in terms of the impulse response $h(n)$ by the following procedure:

1. Express the signal as a sum of weighted and shifted impulses:
   $x(n) = \sum_m x(m)\delta(n-m)$

2. By linearity and time invariance, the filtered signal is simply the same sum of weighted and shifted impulse responses:
   $y(n) = \sum_m x(m)h(n-m) = \sum_m h(m)x(n-m)$.

The digital filter is called a *finite-duration impulse response* or *FIR* filter if $h(n)$ is identically zero outside of a finite set of values $0 < n < N-1$. Then the output $y(n)$ is a linear combination of the input $x(n)$ and a finite number of previous input values:

$$y(n) = \sum_{m=0}^{N-1} h(m)x(n-m).$$

We can define further properties of an FIR filter by considering its response to a complex exponential signal $x(n) = e^{j\omega n}$:

$$y(n) = \sum_{m=0}^{N-1} h(m)e^{j\omega(n-m)} = \left[\sum_{m=0}^{N-1} h(m)e^{-j\omega m}\right] e^{j\omega n} = H(\omega)x(n).$$

We call $H(\omega)$ the *frequency response* of the system; it is fundamentally a polynomial in the complex exponential $e^{-j\omega}$.

The above introduction to the theory of digital filters is adapted from and developed further in [17].

### 4.2.2 The filter design problem

The general FIR filter design problem is to approximate a desired frequency response over the region $\omega \in [-\pi, \pi]$ by the frequency response of a length $N$ FIR filter. The filter designer needs to specify the measure of goodness by which an approximation should be judged; typical measures of error are the least-squared ($\ell_2$) error or the Chebyshev ($\ell_\infty$) error. Depending on the error measure, an algorithm is chosen or designed to find the optimal filter coefficients $h(n)$.

Here we consider the specific problem of designing a *bandpass* filter; that is, a filter whose frequency response approximates the ideal frequency response given by:

$$H(\omega) = \begin{cases} 1 & \text{if } \omega \in [\omega_{p_1}, \omega_{p_2}] \\ 0 & \text{otherwise} \end{cases}$$

The interval $[\omega_{p_1}, \omega_{p_2}]$ in which the ideal frequency response is 1 is called the *passband*; the part of the frequency band in which the ideal response is 0 is called the *stopband*.

We will use the Chebyshev error criterion, which minimizes the maximum deviation from the ideal frequency response over a specific set of frequencies.

We state without proof the important *Alternation Theorem*, [17, pp. 87-88]:

If $A(f)$ is a linear combination of $r$ cosine functions

$$A(f) = \sum_{k=0}^{r-1} c_k \cos(2\pi k f)$$

then the following statements are equivalent:

1. $A(f)$ is the unique optimal Chebyshev approximation to a given continuous function $D(f)$ for $f \in$ a given set of frequencies $\mathcal{F}$
2. The error function $E(f) = D(f) - A(f)$ has at least $r+1$ *extremal frequencies* $f_1 < f_2 < \cdots < f_{r+1} \in \mathcal{F}$ such that $E(f_m) = -E(f_{m+1})$ for $m = 1, 2, \ldots, r$ and $|E(f_i)| = \max_{f \in \mathcal{F}} E(f)$.

An important feature of the alternation theorem is that it guarantees the optimal Chebyshev approximation to a desired function to have an *equiripple* character. In the context of filter design, $D(f)$ is the desired frequency response, and $A(f)$ is the frequency response of a length $N$ FIR filter. $A(f)$ can be expressed as a sum of cosines when the filter coefficients are symmetric; that is, when $h(m) = h(N-m)$. In filter design, we apply a tool called the *Remez exchange algorithm* to iteratively compute an optimal set of extremal frequencies; from this we obtain the unique optimal Chebyshev approximation to $D(f)$.

A simple transformation allows us to convert an optimal Chebyshev bandpass FIR filter into a polynomial in $x$ over the interval [-1,1] which is equiripple outside of a peak at a specific ordinate $x = \sigma$. We can then use this polynomial to accelerate the computation of the eigenvalues of a symmetric matrix $A$ nearest this shift $\sigma$ (after $A$ has been suitably scaled so its spectrum lies in [-1,1]).

The complete procedure is:

**Algorithm 9**: FIR filter polynomial acceleration

**Input**: A matrix $A$ for which to compute the $k$ eigenvalues nearest a shift $\sigma$
An estimate $[lbd, ubd]$ of an interval containing the spectrum of $A$
The degree $N$ of the polynomial accelerant

**Output**: $\{(x_1, \lambda_1), (x_2, \lambda_2), \ldots, (x_k, \lambda_k)\}$,
approximations to the $k$ wanted eigenvalues of $A$.

1. Determine the linear mapping $M$ which takes $lbd \to -1$, $ubd \to 1$.
   Specifically, the mapping is $M(x) = ax + b$,
   where $a = \frac{2}{(ubd-lbd)}$ and $b = \frac{-(lbd+ubd)}{(ubd-lbd)}$.
2. Compute $\mu = M(\sigma)$.
3. Design $F_m$, an order $m$ linear phase bandpass FIR filter with
   - a monotonically decreasing passband,
   - a single peak in the frequency response at $\cos^{-1}(\mu)$, and
   - an equiripple stopband.
4. Let $H_m$ be the polynomial in $x$ obtained by transforming $x = \cos(\omega)$
   in the frequency response generated by $F_m$.
5. Define $P = H_m(M(A))$.
6. Compute the $k$ largest eigenvalues and corresponding eigenvectors of $P$,
   $\{(x_1, \mu_1), (x_2, \mu_2), \ldots, (x_k, \mu_k)\}$.
7. Let $\lambda_i = \frac{x_i^T A x_i}{x_i^T x_i}$ for $i = 1, \ldots, k$.
8. **end**;

The filter design can be accomplished using an iterative method based on the Remez exchange algorithm.

The *Zolotarev polynomials* are a class of functions similar to the Chebyshev polynomials which have a similar character to the polynomials which result from the filter design scheme discussed above. These polynomials involve elliptic functions in the same way that Chebyshev polynomials involve cosine functions. Sadly, there is no closed form expression for Zolotarev polynomials of degree greater than 3, though these polynomials can be computed iteratively. A reference on the application of Zolotarev polynomials to FIR filter design is given in [2].

### 4.2.3 Further issues to consider

The peak of the filter polynomial generally will not be exactly symmetric about the shift $\sigma$. Therefore, eigenvalues that are an equal distance away from $\sigma$ may not get mapped to the same value by the polynomial. The worst case result is that the eigenvalues the algorithm returns as the $k$ eigenvalues "closest to $\sigma$" may not actually be the $k$ closest eigenvalues. Since the peak of the filter polynomial is approximately symmetric about the shift, this problem should rarely arise, especially if the user understands the nature of the polynomial accelerant and knows roughly how the spectrum is distributed.

The problem could be avoided entirely by using an accelerant polynomial that is symmetric about $\sigma$; perhaps a polynomial interpolant to a sharply peaked Gaussian function would be appropriate. Then the spectrum could simply be shifted into the domain of the peak.

However, polynomial interpolants to smooth functions are dangerous to use as accelerants because of problems with monotonicity over the region of interest. An $m^{th}$ order polynomial must oscillate $m-1$ times; controlling where these oscillations occur is central to designing an interpolant whose largest eigenvalues correctly match the desired eigenvalues of the problem. It could be the case that the size and location of the oscillations badly damage the monotonicity of the polynomial, and thus the accuracy of the eigenvalue approximations.

In the above examples, the regions of oscillation of the Chebyshev and filter polynomials are known and mapped to regions in which the oscillation is harmless. In the general interpolation problem, there is no control over the location and magnitude of oscillations.

In the special case of interpolation of a Gaussian function using a polynomial of high degree we encounter the *Runge phenomenon*. Essentially the agreement between the interpolant and the Gaussian function is fairly good in the center of the interpolation interval but is terrible near the ends of the interval [4].

## 4.3 Extensions to the non-symmetric case

To this point we have only applied polynomial acceleration to symmetric matrices, whose eigenvalues are real. It is possible to generalize the method to the case when eigenvalues lie off of the real axis. Saad [19] outlines in some detail the scheme for designing a Chebyshev polynomial variant that takes on small values within an ellipse containing the unwanted eigenvalues.

Unfortunately, we cannot construct a polynomial accelerant in the spirit of FIR filter design which can be used to accelerate the computation of the eigenvalues of a nonsymmetric matrix about any shift in the complex plane. The maximum modulus theorem from complex analysis tells us that any analytic function defined on a closed and bounded region must take its maximum and minimum value on the region's boundary. Since polynomials are analytic, the construction of a polynomial in two variables which peaks in the center of a disk is therefore impossible.

## 4.4 When should polynomial acceleration be used?

Suppose acceleration using a polynomial $p$ is applied to a symmetric matrix $A$ whose eigenvalues are $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$. Then the transformed eigenvalues are given by $\{p(\lambda_1), p(\lambda_2), \ldots, p(\lambda_n)\}$.

Suppose $\lambda_1$ is the largest eigenvalue, and $p$ has been selected so that $p(\lambda_1) > p(\lambda_i), i = 2 \ldots n$. Define $\kappa_i = |p(\lambda_i)/p(\lambda_1)|, i = 2 \ldots n$. Then by our previous discussion of the convergence of the power method, we know that convergence to $\lambda_1$ is proportional to $\max_{i=2 \ldots n} \kappa_i$.

Thus, polynomial acceleration is only worthwhile when this convergence ratio is quite small. It may happen that a shift-and-invert technique centered around a shift $\sigma$ may have a higher convergence ratio than a polynomial acceleration technique based on Algorithm 9 centered around $\sigma$. However, without a priori knowledge of the eigenvalues, there is no way to determine which method will converge more quickly.

Since solution of linear systems is a fast operation in Matlab, shift-and-invert techniques generally converge more quickly than polynomial acceleration techniques when the operator is stored as a matrix.

The primary use of polynomial acceleration is when shift-and-invert techniques are not an option; for example, when the operator of interest is not explicitly stored as a matrix, but instead defined by its action on a vector (see Section 5.7). In this situation, there may be no other way to compute eigenvalues in the interior of the spectrum. If the order of the accelerant polynomial is $m$, then it will take only $m$ times as much work to find the eigenvalues of this polynomial applied to the operator. A nested multiplication form based on Horner's rule will allow us to apply a polynomial in the operator to a vector without requiring anything more than matrix-vector products.

# Chapter 5

# User's guide for `speig`

Matlab is a technical computing environment for high-performance numeric computation and visualization. It is used extensively both in academia and in industry and is becoming the standard language which applied mathematicians are expected to speak. In the following sections, a working knowledge of Matlab syntax and notation is assumed. For more information, see [15].

## 5.1 What is `speig`?

In Matlab version 4, the `eig` command could be used to find all the eigenvectors and eigenvalues of a matrix. However, the algorithms in `eig` were designed with full matrices in mind and are not well-suited to calculating a few selected eigenvalues of large structured or sparse matrices.

The functionality of the `eig` command is unchanged in Matlab version 5 for matrices in the full storage class. However, when `eig` in Matlab version 5 is called with a sparse matrix or a string in the first argument, the input arguments are passed to the m-file `speig.m` contained in the `sparfun` directory. The `speig` function (for sparse eigenvalue) incorporates the implicitly restarted Arnoldi method for eigenvalue computation discussed in Chapter 3, and not the QR or QZ algorithms normally used by `eig` to solve dense eigenvalue problems. When control is passed from `eig` to `speig` a warning appears to inform the user of the change and remind him or her that `speig` can and should be called directly for large sparse problems.

## 5.2 When to use `speig`

Using `speig` instead of `eig` is most appropriate when:

1. The order of the input matrix is large;

2. Computing factorizations (for example, $LU$ or $QR$) of the input matrix is expensive;

3. Computing the product of the input matrix and a vector is inexpensive (for example, $O(n)$, where $n$ is the order of the matrix);

4. Only a small fraction of the eigenpairs are desired;

5. The eigenvalues of interest are either concentrated in the spectrum extremally (i.e. the 5 eigenvalues of largest magnitude or smallest real part) or centered around a point in the complex plane (i.e. the 5 eigenvalues closest to $2 + 2i$.)

`speig` can be used to calculate eigenpairs of any matrix, full or sparse. However, it is best suited to problems for which the above conditions are satified. Usually, matrices with these characteristics are highly structured and could profitably be stored as sparse. `speig` is also viable for problems which are large enough to fit into memory but for which `eig` does not converge. If only a few eigenpairs of these dense problems are desired, `speig` is a useful tool.

## 5.3 Sparsity

When deciding between `eig` and `speig`, it is important to know the matrix class and structure of the input argument. Matrices that are structurally dense but are stored as sparse may be created as the result of operations involving sparse matrices. For example, the command

```
>> rand(5).*spones(5)
```

produces a fully dense matrix with random entries that is classed as sparse. The `whos` command displays the size and density (if sparse) of all matrices in the workspace.

The matrix $A$ is sparse if

```
>> A
```

produces a list of ordered entries

```
A =

   (1,1)        0.5717
   (2,2)        0.8024
   (3,3)        0.0331
   (4,4)        0.5344
   (5,5)        0.4985
```

instead of a rectangular array

```
A =

    0.5717         0         0         0         0
         0    0.8024         0         0         0
         0         0    0.0331         0         0
         0         0         0    0.5344         0
         0         0         0         0    0.4985
```

To convert a full matrix $A$ to sparse storage mode, use

```
>> A = sparse(A);
```

To convert a sparse matrix $B$ to full storage mode, use

```
>> B = full(B);
```

If the input matrix is stored as sparse but any of the conditions in the previous section are not met (for example, the order of the matrix is low, the matrix is actually dense, or all of the eigenvalues are desired) then it may be more appropriate to use

```
>> [V,D] = eig(full(A))
```

instead of

```
>> [V,D] = speig(A).
```

Be aware that invoking

```
>> full(A)
```

when $A$ is a sparse matrix whose order is very large (in the hundreds, for example) will consume a large amount of memory and may result in an `Out of memory` error. Sparsity should be exploited whenever possible by converting matrices into the sparse storage mode.

`speig` will issue a warning if the input matrix is classed as sparse but has a "low" order. When the input matrix order is less than $c$, `eig` is faster than `speig`. The number $c$ which determines the crossover point will vary according to the amount of memory, clock speed, and architecture of the host machine, as well as the type of eigenvalue problem being solved. On a 40MHz 486DX PC with 8MB of RAM, $c$ is about 150. On a Sparc-20 workstation with 32MB of RAM, $c$ may be higher than 400. It is very important to choose the eigenvalue code that best suits the problem at hand.

## 5.4 Syntax of `eig` in Matlab version 4

In Matlab version 4, the `eig` command is used to obtain a column vector of the eigenvalues of a matrix simply by typing

```
>> d = eig(A);
```

When the eigenvectors of the matrix are also desired,

```
>> [V,D] = eig(A);
```

produces a diagonal matrix $D$ of eigenvalues and a full matrix $V$ whose columns are the corresponding eigenvectors so that $AV = VD$.

`eig` works similarly for generalized eigenvalues and eigenvectors, so that

```
>> d = eig(A,B)
```

returns a vector containing the generalized eigenvalues of square matrices $A$ and $B$, and

```
>> [V,D] = eig(A,B)
```

produces a diagonal matrix $D$ of generalized eigenvalues and a full matrix $V$ whose columns are the corresponding eigenvectors so that $AV = BVD$.

## 5.5 Basic syntax of `speig`

The syntax of `speig` is more complicated than that of `eig` because it is not possible (or at least unwise) to compute all of the eigenvalues of a large structured matrix at once using the implicitly restarted Arnoldi method. In many practical applications in which large matrices arise, only a small fraction of the spectral information is of interest. Therefore, the user needs to be able to specify how many eigenpairs of the matrix are desired and in what part of the spectrum the eigenvalues reside.

`speig` was designed to be as consistent with `eig` as possible. The output arguments of `speig` are the same as the output arguments of `eig`; that is, providing one output argument as in

```
>> d = speig(A,...);
```

returns a vector $d$ of eigenvalues of $A$, while providing two output arguments as in

```
>> [V,D] = speig(A,...);
```

produces a diagonal matrix $D$ of eigenvalues and a full matrix $V$ of corresponding eigenvectors so that $AV = VD$. The eigenvectors are scaled so that the norm of each is 1.0. Even if the input matrix is sparse, the output from `speig` is always full.

The general calling sequence of `speig` is

```
>> [V,D] = speig(A,k,sigma);
```

This computes the $k$ eigenvalues closest to *sigma* in the following sense:

| *sigma* | "find the k eigenvalues..." |
|---|---|
| A real or complex scalar | closest to *sigma* in the sense of absolute value |
| `'LM'` | of Largest Magnitude |
| `'SM'` | of Smallest Magnitude |
| `'LR'` | of Largest Real Part |
| `'SR'` | of Smallest Real Part |
| `'BE'` | on Both Ends |

Using *sigma* =`'SM'` is equivalent to using $sigma = 0$ in most situations.

*sigma* =`'BE'` computes $k/2$ eigenvalues from each end of the spectrum with respect to real part; that is, the $k/2$ eigenvalues of smallest real part and the $k/2$ eigenvalues of largest real part. One additional eigenvalue is computed from the high end if $k$ is odd. The `'BE'` feature is useful in getting a rough idea of the spectrum of a symmetric matrix.

`'BE'` is the default *sigma* used when the input matrix $A$ is symmetric, and `'LM'` is the default *sigma* used when $A$ is nonsymmetric. The default $k$ is 5. Thus, by default,

```
>> d = speig(A);
```

returns a vector containing the 5 eigenvalues of the largest magnitude of a nonsymmetric input matrix $A$, and

```
>> d = speig(A,k);
```

returns the $k$ eigenvalues of largest magnitude. However, for readable m-files and code, it is best to call `speig` with $k$ and *sigma* explicitly defined instead of relying on the default values.

While the user can supply a value for $k$ without supplying a value for *sigma*, the reverse is not true. If the user supplies a *sigma*, a $k$ must also be given. Of course, as in most Matlab commands, the order of the input arguments is important, and errors will result from badly phrased commands such as

```
>> [V,D] = speig(A,'LM',5);
```

The generalized eigenvalue problem $Ax = \lambda Bx$ can be solved using `speig` only when $B$ is a positive semidefinite matrix the same size as $A$. The syntax described above extends in the natural way to the command structure

```
>> [V,D] = speig(A,B,k,sigma);
```

as well as the structures

```
>> [V,D] = speig(A,B);
>> [V,D] = speig(A,B,k);
```

which rely on the default values of $k$ and $sigma$.

## 5.6 The `speig` options structure

### 5.6.1 Motivation

The implictly restarted Arnoldi method is somewhat complicated. While the `speig` interface is easy to use, the user may obtain better performance by changing some of the parameters used during the Arnoldi iteration from their defaults. Allowing the user to alter these parameters without cluttering the calling sequence was an interesting design problem. A parameter-setting mechanism with the following attributes was desired:

1. An experienced user might want to change the parameters to `speig` fairly often. The mechanism should be easy to use and not rely on remembering the order of arguments to a function.

2. The user should be able to store a set of parameter settings as some type of variable, for easy use and reuse.

3. The parameter-setting mechanism should not create variables in the global workspace.

4. The parameter-setting mechanism should be flexible enough to accomodate the addition of more tunable parameters.

Several structures for the parameter-setting mechanism were considered:

1. Add an additional function similar to the existing Matlab function `spparms` to set `speig` algorithm parameters. `spparms` is a relatively obscure function which allows the user to tune Matlab internal parameters which affect the sparse matrix ordering algorithms and linear equation solvers. The parameters are set internally and do not reside as local or global variables in the workspace.

2. Create a set of optional input arguments to `speig`. These arguments would either have to be ordered correctly on input, or would be sorted by a parser inside the function. For example, the first line of `speig.m` might look like:

   ```
   function [v,d] = speig(A,B,k,sigma,p,tol,maxit,other options...)
   ```

3. Create a pair of parameter-setting functions similar to the `set` and `get` functions used to tune parameters of Matlab's Handle Graphics.

Adding an `spparms`-like command was attractive, but precluded holding different sets of parameters in memory simultaneously. Every parameter change would require that an `eigparms` command be issued. Furthermore, it was preferable to keep all of the `speig` commands at the m-file level instead of mixing m-files and Matlab-internal C code.

Providing `speig` with several additional input arguments seemed to be a bad idea, since the user might have to supply in full all the parameters to be changed every time the function was called. The user would either have to supply the input arguments in the correct order, which might give rise to messy commands such as

```
>> speig(A,k,sigma,,,,,,,50)
```

or add the input arguments in a manner such as

```
>> speig(A,k,sigma,'tol',1e-6,'maxit',300)
```

The second option would require even more input argument parsing inside the main function. The current version of `speig` already includes several hundred lines of code to determine which input argument should be assigned to $A$, which to $k$, which to $sigma$, and so on. This is a consequence of the flexibility of the calling sequence.

Analogs to `set` and `get` were the most attractive and tenable option, provided some method of storing the parameter structure as a Matlab variable. Ultimately the commands `speigset` and `speigget` were implemented in this way. These commands were modeled after the `odeset` and `odeget` commands from the Matlab Ordinary Differential Equations Suite.

### 5.6.2 Syntax of speigset and speigget

The command

```
>> opt = speigset('name1', 'value1', 'name2', 'value2', ...)
```

creates a variable *opt*, called an *options structure*, in which the value of the parameter *name1* is set to *value1*, *name2* to *value2*, etc. The current valid parameter names, descriptions, and default values appear in the following subsection. When `speigset` is called with no arguments, it prints all parameter names and permissible data types:

```
>> speigset

        n: [positive integer]
        p: [positive integer]
        tol: [positive scalar]
        maxit: [positive integer]
        issym: [non-negative scalar]
        dopoly: [non-negative scalar]
        gui: [scalar]
```

The `speigget` command extracts parameter values from an options structure created with `speigset`.

```
>> v = speigget(opt, 'name')
```

extracts the value of the parameter specified by *name* from the options structure *opt*, returning 0 if the parameter value is not specified in *opt*.

```
>> speigget(opt)
```

displays all the parameter names and their current values for the options structure *opt*. 0 is displayed for parameter values not specified in *opt*.

Here is a typical example of the usage of `speigset` and `speigget`:

```
>> opt1 = speigset('tol',1e-6,'maxit',200,'gui',-1);

>> speigget(opt1);

     n       : 0
     p       : 0
     tol     : 1e-06
     maxit   : 200
     issym   : 0
     dopoly  : 0
     gui     : -1

>> p1 = speigget(opt1,'gui')

p1 =

    -1

>> p2 = speigget(opt1,'tolerance')

??? Error using ==> speigget
Unknown parameter name tolerance.

>> p2 = speigget(opt1,'tol')

p2 =

   1.0000e-06
```

The complete parameter name need not be supplied to enter or extract a parameter from an options structure; the first few characters which uniquely identify a parameter will suffice. Currently, the parameters all begin with different letters, so simply specifying the first letter of the parameter to set will work. For example:

```
>> opt1 = speigset('t',1e-6,'m',200,'g',-1);

>> speigget(opt1);

     n       : 0
     p       : 0
     tol     : 1e-06
     maxit   : 200
     issym   : 0
     dopoly  : 0
     gui     : -1

>> p = speigget(opt1,'g')

p =

    -1
```

Of course, for clarity and for readable code, it is always best to use the complete parameter name.

Once an options structure *opt* has been created, it can be passed as the last input argument to `speig`, giving rise to the constructions

```
>> [V,D] = speig(A,opt);
>> [V,D] = speig(A,k,opt);
>> [V,D] = speig(A,k,sigma,opt);
>> [V,D] = speig(A,B,opt);
>> [V,D] = speig(A,B,k,opt);
>> [V,D] = speig(A,B,k,sigma,opt);
```

The nonzero parameters in *opt* are then used in `speig` instead of the defaults. Since options structures are stored just like normal variables, it is easy to create several parameter settings and compare which gives the best results for a certain problem. For example,

```
>> tol8 = speigset('tol',1e-8);
>> tol12gui = speigset('tol',1e-12,'gui',1);
>> d1 = speig(A,5,'SM',tol8);
>> d2 = speig(A,5,'SM',tol12gui);
```

With correct access, the user can change the default parameter settings within the file `speig.m` itself.

### 5.6.3 Description of parameters and defaults

The following parameters for the implicitly restarted Arnoldi method implemented in `speig` can be set using an options structure created by `speigset`:

| Name | Description | Default Value |
|---|---|---|
| $n$ | Dimension of the problem: the order of the square input matrix $A$ | none |
| $p$ | Dimension of the Arnoldi basis: the number of Arnoldi vectors generated at each iteration | $2k$ |
| *tol* | Tolerance for convergence of $\frac{\lVert AV-VD\rVert}{\lVert A\rVert}$ | $10^{-10}$ (symmetric $A$,$B$)<br>$10^{-6}$ (nonsymmetric $A$,$B$) |
| *maxit* | Maximum number of Arnoldi iterations | 300 |
| *issym* | Positive if $A$ is symmetric, 0 otherwise | 0 |
| *dopoly* | Positive if<br>• $A$ specifies a matrix-vector product<br>• *sigma* is `'LR'`, `'SR'` or numeric, and<br>• Polynomial interpolation is to be used to accelerate convergence. | 0 |
| *gui* | $\begin{cases} \text{positive} & \text{if the Progress Report window is to be shown} \\ \text{negative} & \text{if the Stop Button window is to be shown} \\ 0 & \text{otherwise} \end{cases}$ | 0 |
| *v0* | Starting vector (in version 5 implementation only) | `rand(n,1) - 0.5` |

### 5.6.4 Matlab version 4 implementation

In Matlab version 4, options structures are simply stored as row vectors, with zero entries to denote default values:

```
>> opt1=speigset('tol',1e-6,'maxit',200,'gui',-1)
```

```
opt1 =

         0         0    0.0000  200.0000         0         0   -1.0000
```

The parameters are stored in *opt* in the order:

| Position in *opt* | Parameter |
|---|---|
| 1 | $n$ |
| 2 | $p$ |
| 3 | $tol$ |
| 4 | $maxit$ |
| 5 | $issym$ |
| 6 | $dopoly$ |
| 7 | $gui$ |

The above list is stored as a variable $names$ in both `speigset.m` and `speigget.m`. If new parameters are added, $names$ must be updated in both places. In addition, the m-file `validopt.m`, which is used to determine if a Matlab variable is a valid options structure, must be updated.

Parsing within `speigset` ensures that the parameters are stored in the correct slots, regardless of the order in which they are supplied in the calling sequence. Input arguments are compared against $names$ to ensure that valid parameters are being set.

Input arguments to `speigget` are compared against $names$ in order to determine the parameter that is being queried and the corresponding value. If there is no match between the first characters of the input argument and the first characters of any entry in $names$, an error message is printed.

### 5.6.5 Matlab version 5 implementation

Though their syntax remains the same, `speigset` and `speigget` are much more efficient in Matlab version 5 with the introduction of structures as Matlab data types. A structure naturally accomodates the name/value pairs that make up an options structure and makes error checking and parsing much easier. Furthermore, parameters that are non-scalar (for example, the starting vector `v0`) can be incorporated into the options structure, a feature not implemented in the Matlab version 4 code.

## 5.7 Finding the eigenvalues of an operator

### 5.7.1 Motivation

The implicitly restarted Arnoldi method for eigenpair computation, without using shift-and-invert methods, only requires the matrix $A$ in order to form matrix-vector products $Ax$. Therefore, it suffices to know the action of $A$ on a vector in order to compute the eigenpairs of $A$. This presents the possibility of calculating the eigenvalues of an operator that is not actually stored as a Matlab variable. All that is required is a Matlab function *mvprod* such that

```
>> w = mvprod(v)
```

returns the same answer as $w = Av$.

### 5.7.2 A simple example

Consider the $n \times n$ one-dimensional discrete Laplacian matrix:

$$\begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \ddots & 0 \\ 0 & 0 & 1 & -2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & 0 & 0 & \cdots & 1 & -2 \end{pmatrix}$$

This operator could be created as a Matlab sparse matrix:

```
>> e = ones(n,1);
>> A = spdiags([e -2*e e], -1:1, n, n);
```

Now consider this Matlab m-file, `lap.m`:

```
function w = lap(v);

n = size(v,1);
w(2:(n-1),:) = -2*v(2:(n-1),:) + v(1:(n-2),:) + v(3:n,:);
w([1,n],:) = -2*v([1,n],:) + v([2,n-1],:);
```

It is easy to verify that

```
>> w = lap(v);
```

gives the same result as

```
>> w = A*v .
```

The m-file implementation requires substantially fewer floating point operations to perform the matrix-vector product. Furthermore, the m-file is independent of the problem size, $n$.

There are many real-world applications in which all that is known about the operator $A$ is its action on a vector, and not its structure as a matrix. These situations arise in several fields including control systems and the discretization of large partial differential equations.

### 5.7.3 Warning: `mvprod` must accept a matrix as input

When creating a function `mvprod` that specifies a matrix-vector product for the purpose of using `speig`, it is extremely important that `mvprod` be able to return a matrix of column vectors corresponding to the results of `mvprod` applied to each column of the input matrix. In the above example, `lap` takes advantage of Matlab vectorization to generate all of the matrix-vector products at once. An improper implementation of `lap` would be:

```
function w = lapbad(v);

n = size(v);
w(2:(n-1)) = -2*v(2:(n-1)) + v(1:(n-2)) + v(3:n);
w([1,n]) = -2*v([1,n]) + v([2,n-1]);
```

While `lapbad` would act correctly on a single vector,

```
>> lapbad(rand(10))
```

results in an error since no provision is made for a matrix as input.

Loops should be avoided in matrix-vector product m-files. Often, thoughtful coding and use of vectorization can improve the speed of the function significantly. In addition, it is better to make use of global variables to define constants and matrices used in the function than to explicitly define these in the m-file itself. Execution will be very slow if large matrices are defined each time the m-file is called.

### 5.7.4 Syntax

To use `speig` to find the eigenpairs of an operator defined by a matrix-vector product in the m-file `mvprod.m`, use the syntax

```
>> [V,D] = speig('mvprod',n);
```

Here, $n$ is the order of the operator $A$; that is, the dimension of the domain of $A$. For matrices, this is simply the number of columns, or `size(A,2)`.

The operator dimension $n$ must be supplied in the last argument of the call to `speig`. Otherwise, there is no way to determine the dimension of the problem. There are two ways to supply $n$:

1. If no options structure is to be supplied, then give $n$ as the last input argument to `speig`. For example,

   $$\texttt{>> [V,D] = speig('lap',5,'LR',}\underbrace{\texttt{40}}_{\mathrm{n}}\texttt{);}$$

2. If parameters are to be set via an options structure, then include $n$ in the options structure along with the other parameters and use the normal syntax, such as:

   ```
   >> opt1 = speigset('tol',1e-6,'n',40);
   >> [V,D] = speig('lap',5,'LR',opt1);
   ```

Again, it is very important that $n$ be supplied as the last argument in the call; if it is supplied elsewhere, it may be misinterpreted as $k$ or *sigma*.

Either the first or second method above must be used; a mixture of the two as in

```
>> opt1 = speigset('tol',1e-6);
>> [V,D] = speig('lap',5,'LR',opt1,40);
```

will result in an error.

## 5.8 Polynomial acceleration

When $A$ is a string, and *sigma* is `'SR'`,`'LR'`, or a numeric shift, polynomial acceleration can be activated by passing `speig` an options structure with the parameter *dopoly* set to 1 (or a positive number). Polynomial acceleration often speeds up convergence by an appreciable factor. The mathematics of polynomial acceleration was discussed in Chapter 4. The acceleration scheme as implemented in `speig` is discussed in the following subsections.

### 5.8.1 sigma = 'LR'

We consider the Matlab commands:

```
>> opt = speigset('n',625,'issym',1,'dopoly',1,'gui',1);
>> [V,D] = speig('lap',5,'LR',opt);
```

When the `dopoly` parameter is parsed, `speig` converts `sigma = 'LR'` into an internal setting `sigma = 'LO'`. The interpretation of `'LO'` is to compute the $k-1$ eigenvalues of largest real part and 1 eigenvalue of smallest real part. This allows `speig` to get a good start on computing the wanted eigenvectors while also getting rough bounds on the spectrum. When the Ritz estimates of the smallest and largest Ritz values are less than 0.1, the global variables `lbd` and `ubd` are assigned to be these Ritz values, respectively. The Arnoldi basis is then explicitly restarted with the Ritz vector corresponding to the largest eigenvalue. The name of the m-file specifying the matrix-vector product, in this case `'lap'`, held in the input argument `A`, is assigned to the global variable `op`. The global variable `sig` holds the value of the original `sigma` requested, in this case `'LR'`. The argument `A` is then reassigned to the m-file `'accpoly'`, the polynomial acceleration subfunction. The argument `sigma` is assigned to the value `'LR'`, since we will be searching for the largest eigenvalues of the matrix polynomial in `op`.

The part of the function `accpoly` that applies to this case is:

```
function w = accpoly(v)

global op lbd ubd sig

m = 10;
p = .25;

slope = 2/(ubd-p*(ubd-lbd)-lbd);
intercept = -1 - (lbd*slope);

w0 = v;
w1 = slope*feval(op,v)+intercept*v;

for jj = 2:m
```

```
        w = 2*(slope*feval(op,w1)+intercept*w1)-w0;
        w0 = w1;
        w1 = w;
end
```

`w=accpoly(v)` has the following effect:

1. Maps the left hand side of the spectrum of `op` into [-1,1] by the transformation `op2 = slope*op + intercept`. The parameter `p` in `accpoly` controls the fraction of the spectrum that is mapped into the equiripple region of the polynomial.

2. Applies the Chebyshev polynomial of degree 10 in the scaled operator `op2` to the input argument `v`.

Then `speig` sets the iteration count to 1 and restarts the entire main loop as if the command

```
>> [v,d] = speig('accpoly',k,'LR');
```

was originally issued. If `'gui'` is enabled, the normal graphics display will switch over to a graph of the accelerant polynomial plotted over the Ritz values of the original problem.

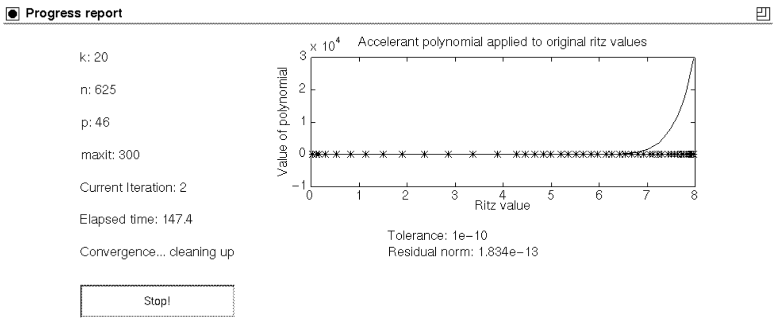



Figure 5.1: Polynomial accelerant for `sigma = 'LR'`

It is important to note that the Ritz values which are now displayed in the command window are no longer estimates of the eigenvalues of the original operator `op`, but rather estimates of the eigenvalues of the polynomial accelerant operator `accpoly`. After convergence, eigenvalue estimates for the original operator `op` are recovered by forming the Rayleigh quotient `D = V'*feval(op,V)`, where V is the matrix of Ritz vectors obtained from applying `speig` to `accpoly`.

### 5.8.2 sigma = 'SR'

If `sigma = 'SR'` instead of `'LR'` in the above example, the only change in the algorithm is to map the right hand side of the spectrum of `op` into [-1,1] instead by using the different values for `slope` and `intercept` discussed in section 4.1.2.

### 5.8.3 sigma = a numeric shift

When `sigma` is a numeric shift, the polynomial that is used to accelerate convergence is obtained by the FIR filter design methods discussed in section 4.2. The program flow is still shunted to the subfunction `accpoly` as discussed above; however, the code which is executed in this case is:

```
function w = accpoly(v)

global op lbd ubd sig filtpoly iter

m = 10;

slope = 2/(ubd-lbd);
intercept = 1 - (ubd*slope);

if iter == 1
        wp = acos(slope*sig+intercept);
        m = 10;
        N = 2*m+1;
        L = 4;
        del = 0.01;
        h = bp_fer(N,L,wp,del,-del,2^7);
        filtpoly = h2x(h);
        iter = 2;
end

w = filtpoly(1)*v;

for i=2:(m+1)
```

```
        w = (slope*feval(op,w)+intercept*w) + filtpoly(i)*v;
end
```

The accelerant polynomial `filtpoly` is calculated using the FIR filter design m-file `bp_fer` and subfunctions written by Ivan Selesnick of the Electrical and Computer Engineering Department, Rice University [21]. The polynomial is only computed once, at the first polynomial iteration; it remains in global memory for the remaining iterations and need not be recalculated.

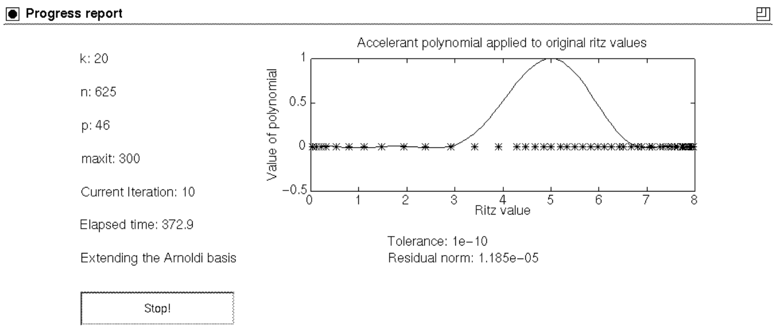



Figure 5.2: Polynomial accelerant for `sigma` = a numeric shift

The polynomial in the scaled operator is applied to the input argument `v` by nested multiplication.

It should be noted that if the eigenvalues are irregularly distributed in the spectrum, the user may not exactly receive the $k$ eigenvalues closest to the shift `sigma` due to asymmetries in the accelerant polynomial about `sigma`. In spite of this shortcoming, this method still allows us to compute interior eigenvalues and eigenvectors which were not easily computed by previous means.

## 5.9 The graphical user interface

One limitation of Fortran eigenvalue codes is their text-only interface. I felt it was important to create a graphical display for `speig` so that the user could monitor the eigenvalue approximations as they progressed, and stop the process at an intermediate stage if desired.

### 5.9.1 Stop Button

Inside Matlab, the escape sequence **CNTL-C** halts execution of code and returns to the workspace prompt. All intermediate results from the interrupted process are lost. In an application such as `speig`, valuable information about the eigenpairs can be gained from intermediate variables. Since each successive iteration improves the eigenpair approximations, intermediate results from `speig` will give "looser" estimates of the eigenpairs. If the algorithm is converging too slowly, or the user does not want to wait for the Ritz pairs to converge to full tolerance, there should be some way to interrupt the algorithm and extract these looser estimates. I envisioned a "stop button" that the user could press to halt the algorithm and return the current results.

Unfortunately, Matlab has no capacity to tell whether a key is pressed while a command is executing unless

1. The key is an escape sequence (**CNTL-C**), in which case intermediate information is lost, or

2. A keypress is solicited within the m-file (for example, using the `input` function). However, `input` will stop the program flow and wait indefinitely for a keypress, an undesirable attribute.

The ideal Matlab function for this purpose would be some type of keyscan function that would return True if the user pressed a key on the keyboard. Then at selected points in the algorithm, the keyboard would be scanned for the stop character, and the algorithm would terminate at the current iteration. However, no such Matlab function exists in version 4.

Therefore, I had to implement the "stop button" using Matlab's graphical user interface tools. When the user passes `speig` an options structure with *gui* set to a negative number, the following Matlab window appears.

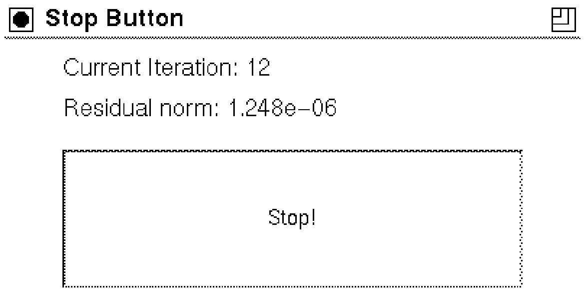



Figure 5.3: SPEIG Stop Button

The window displays the current iteration number and a residual which indicates the goodness of approximation of the current Ritz pairs. The residual is an approximation to $\frac{||AV-VD||}{||A||}$, where $D$ is a $k \times k$ diagonal matrix containing the current Ritz values , and $V$ is an $n \times k$ matrix containing the corresponding Ritz vectors. However, until the very end of the algorithm, these matrices are not actually formed. The displayed residual is really the norm of the vector of Ritz estimates corresponding to the approximate eigenpairs, because it is inexpensive to calculate the Ritz estimates during the iteration. However, as the algorithm converges, the Ritz estimates become poor estimates of the actual error in each Ritz pair (See the convergence history in Figure 7.2). When the Ritz estimates become too small, the matrix of errors $\frac{||AV-VD||}{||A||}$ is actually formed. This is expensive computationally but ensures that the user receives the desired eigenpairs to the requested tolerance.

The large Stop Button can be pressed at any time during the execution of `speig`; however, the program flow is interrupted only at certain points to check whether the button has been pushed. The command which defines the stop button is:

```
b1 = uicontrol('style','pushbutton',
                    'units','normalized',
                    'string','Stop!',
                    'callback','set(gcf,''userdata'',1);',
                    'position',[.1 .1 .8 .5]);
```

The *callback* portion of this command is the important feature; when the button is pushed, the *userdata* attribute of the Stop Button window is set from its default value 0 to 1. Figure window attributes are handy places to store variables that are visible to all functions but are not declared as global. At certain points in the iteration, the conditional `if(get(gcf,'userdata') == 1)` is evaluated to determine whether the stop button was pushed. If the conditional evaluates True, then the algorithm finishes the current iteration, calculates the final Ritz pairs, and terminates. Care is taken to make sure the algorithm does not exit in the middle of an iteration, corrupting the information.

In Matlab for Windows, it is necessary to select "Enable Background Process" from the Options menu in order to make the Stop Button work.

### 5.9.2 The Progress Report window

The Stop Button window is intended mostly as a convenience for the user who may want to halt the `speig` algorithm before it naturally terminates. The extra commands used to run the stop button interface are minimal and do not appreciably slow execution time.

The Progress Report window gives a more complete picture of the algorithm as it proceeds. The display includes

- The stop button
- Values of the parameters $k$, $n$, $p$, $maxit$
- The current iteration number
- The elapsed time since the function call
- The current task within the iteration
- The tolerance
- The residual norm as described in the previous section (when the residual norm drops below the tolerance, the algorithm terminates)
- A large graph with the current eigenvalue estimates plotted as dots in the complex plane. The title of the graph indicates which eigenvalues were requested (i.e. "The 5 eigenvalues of largest magnitude"). The dots change position and color as the algorithm proceeds. The color of the dot corresponds to the value $log(Av - v\theta)$, where $(\theta, v)$ is the Ritz pair represented by the dot. The initial color of the dots is green (corresponding to an error greater than 1) and progresses through yellow and orange to red (corresponding to a tolerance of machine precision). In this way, the user can tell exactly which eigenvalues are

converging more quickly or slowly and how good the estimate of a specific value is.

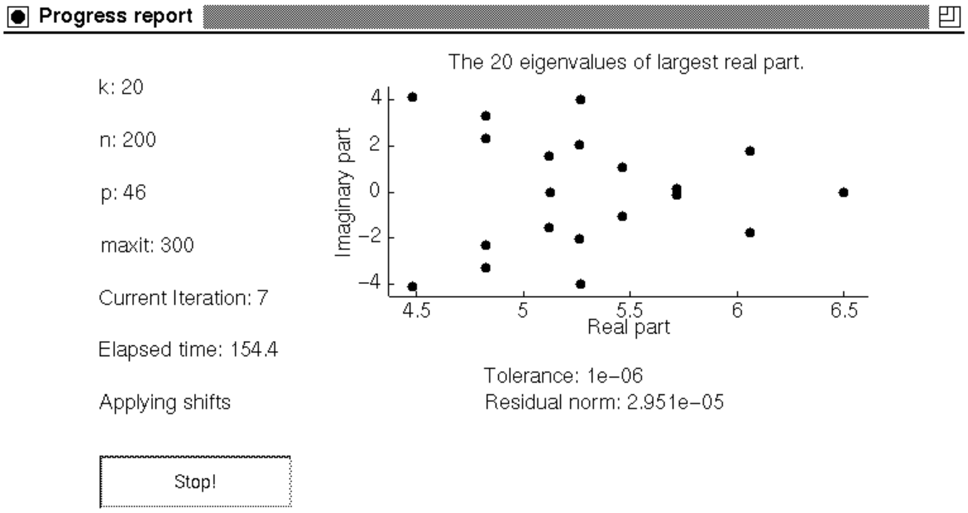



Figure 5.4: SPEIG Progress Report Window

Continually updating the progress report window may take an appreciable amount of time during each iteration. The progress report window is provided mainly for the user who wishes to monitor more closely the evolution of the eigenpair estimates. The progress report window also serves as a good educational tool for understanding how the Arnoldi method proceeds.

## 5.10 speig and its subfunctions

### 5.10.1 Directory listing

Once the directory containing `speig` and its subfunctions has been added to Matlab's search path, typing

```
>> help spdir
```

where **spdir** is the name of this directory will produce the following Table of Contents file:

```
% Sparse eigenvalue functions.
%
% Primary sparse eigenvalue functions
```

```
%   speig       - Sparse matrix generalized eigenvalues and eigenvectors.
%   speigset    - Create a SPEIG options structure.
%   speigget    - View a SPEIG options structure.
%   ssvd        - Sparse singular value decomposition.
%
% Secondary sparse eigenvalue functions (called from SPEIG)
%   apshft1     - Apply shifts to update an Arnoldi factorization.
%   apshft2     - Apply shifts to update an Arnoldi factorization.
%   arnold      - Compute or extend an Arnoldi factorization.
%   arnold2     - ARNOLD for factored matrices.
%   accpoly     - Applies an accelerant polynomial to a set of vectors.
%   accpoly1    - Applies an accelerant polynomial to a set of scalars.
%   isscalar    - True for scalar.
%   shftit      - Calculate shifts to update an Arnoldi factorization.
%   strmatch    - Find possible matches for a string.
%   strvcat     - Concatenation of strings into a matrix.
%   validopt    - True for a valid SPEIG options structure.
%
% Accelerant polynomial functions (called from ACCPOLY)
%   These files were all authored by Ivan Selesnick, ECE Dept, Rice U
%
%   add_poly    - Add two polynomials
%   bp_fer      - Design linear-phase bandpass FIR Chebyshev filter
%   chebpoly    - Chebshev polynomial
%   cos2x       - Converts a polynomial in cos x to a polynomial in x
%   h2cos       - Converts filter coefficients to cosine polynomial
%   h2x         - Converts filter coeffiecients to polynomial in x
%   localmax    - Finds location of local maxima of a vector
%   rlz         - Removes leading zeros of a polynomial
```

speig requires all of the functions in the first two sections in order to execute. The accelerant polynomial functions in the third section are used only when polynomial acceleration is required for an operator whose desired eigenvalues lie close to a shift in the complex plane. Ivan Selesnick of the Electrical and Computer Engineering department at Rice University wrote these functions for a separate filter design application discussed in [21].

In the Matlab version 5 implementation of `speig`, all of the subfunctions will be incorporated into the single m-file `speig.m`. The subfunctions are really not designed to be called separately from `speig`, although documentation is provided in the headers of these files. The user need only deal with the `speig` function to compute the eigenvalues and eigenvectors of a matrix.

### 5.10.2 Program flow

The following flowchart illustrates how `speig` and its subfunctions interact during the iterative Arnoldi method.

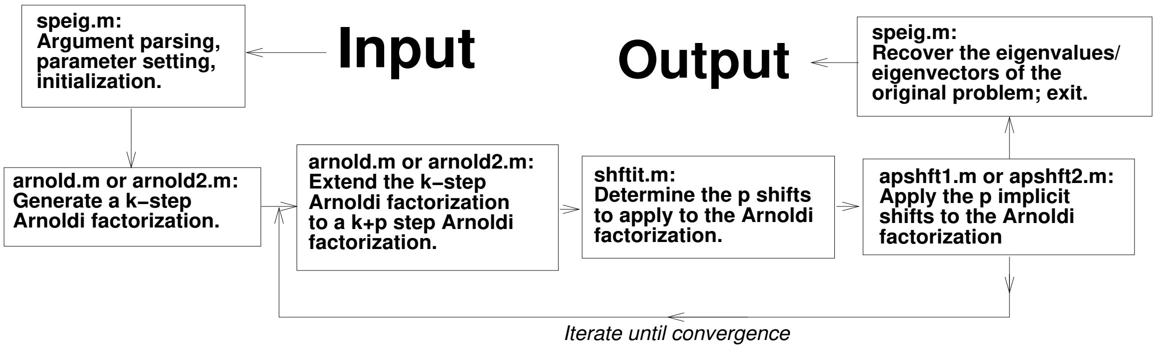



Figure 5.5: Flowchart of `speig` and subfunctions

If polynomial acceleration is activated, the changes in the input arguments described in section 5.8 are made and the entire process is restarted at the top of the diagram.

# Chapter 6

# Sparse singular value decomposition (ssvd)

## 6.1 The singular value decomposition

The *singular value decomposition* (SVD) is one of the most powerful tools of linear algebra. Any $m$ by $n$ matrix $A$ can be factored into

$$A = U\Sigma V^T$$

where $U \in \mathcal{R}^{m\times m}$ and $V \in \mathcal{R}^{n\times n}$ are orthogonal matrices and $\Sigma \in \mathcal{R}^{m\times n}$ is diagonal. The columns of $U$ and $V$ are called left and right *singular vectors* respectively. The diagonal entries of $\Sigma$ are called *singular values*.

The singular values and vectors of $A$ are closely related to the eigenvalues and eigenvectors of $AA^T$ and $A^TA$. Note:

$$AA^T = (U\Sigma V^T)(V\Sigma^T U^T) = U\Sigma\Sigma^T U^T$$
$$A^TA = (V\Sigma^T U^T)(U\Sigma V^T) = V\Sigma^T\Sigma V^T$$

From this is should be clear that the columns of $U$ are the eigenvectors of $AA^T$, and the columns of $V$ are the eigenvectors of $A^TA$. Furthermore, the $r$ nonzero singular values on the diagonal of $\Sigma$ are the square roots of the nonzero eigenvalues of both $AA^T$ and $A^TA$.

When the singular values of $A$ are ranked from largest to smallest magnitude, it is a well-known result that the matrix

$$A_k = U_k\Sigma_k V_k^T$$

is the best rank-$k$ approximation to $A$, where $\Sigma_k = diag(\sigma_1^2, \sigma_2^2, \cdots, \sigma_k^2)$, and the columns of $U_k$ and $V_k$ are the corresponding left and right singular vectors. This truncation of the SVD is useful in contexts such as image processing, in which a picture containing most of the "information" from an original picture can be constructed from the largest singular values and corresponding singular vectors.

## 6.2 Relationship to speig

The connection between the singular value decomposition of $A$ and the eigenvalues and eigenvectors of $AA^T$ and $A^TA$ suggests several methods of using `speig` to compute the size-$k$ truncated singular value decomposition of large structured matrices.

**Method 1**

Form $B = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$, and calculate

```
>> [Q,D] = speig(B,k,'LR');
```

Then if $d_{ii}$ is a positive eigenvalue of $B$, and $Q(:,i)$ is a corresponding eigenvector of norm $\sqrt{2}$, $d_{ii}$ is a singular value of $A$ and $u = Q(1:m,i)$, $v = Q(m+(1:n),i)$ are the corresponding left and right singular vectors.

**Proof:**

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \lambda u \\ \lambda v \end{pmatrix} \Leftrightarrow \begin{array}{c} Av = \lambda u \\ A^T u = \lambda v \end{array} \Leftrightarrow \begin{array}{c} A(\frac{1}{\lambda}A^T u) = \lambda u \\ A^T(\frac{1}{\lambda}Av) = \lambda v \end{array} \Leftrightarrow \begin{array}{c} AA^T u = \lambda^2 u \\ A^T Av = \lambda^2 v \end{array}$$

The last equivalence is simply the relationship of the singular values and vectors to eigenvalues and eigenvectors of $AA^T$ and $A^TA$ discussed in the previous section.

If $B$ has $t$ zero eigenvalues and a complete set of corresponding orthogonal eigenvectors $\{\begin{pmatrix} u_j \\ v_j \end{pmatrix}\}$, then 0 is a singular value of $A$ with corresponding left and right singular vectors obtained by orthogonalizing the $u_j$ and $v_j$, respectively.

**Method 2**

Compute

```
>> [Q,D] = speig(A'*A,k,'LR').
```

Then if $d_{ii}$ is positive, $\sqrt{d_{ii}}$ is a singular value of $A$ with corresponding right singular vector $v = Q(:,i)$ and corresponding left singular vector $u = \frac{Av}{\sqrt{d_{ii}}}$.

**Proof:**

If $v$ is a unit-length eigenvector of $A^TA$ with associated eigenvalue $\lambda$, then $Av$ is an eigenvector of $AA^T$ associated with the same eigenvalue:

$$AA^T(Av) = A(A^TA)v = A(\lambda v) = \lambda(Av).$$

The normalization factor follows from the fact:

$$\|Av\|_2 = \sqrt{v^TA^TAv} = \sqrt{v^T\lambda v} = \sqrt{\lambda}.$$

**Method 3**

Compute

```
>> [Q,D] = speig(A*A',k,'LR').
```

Then if $d_{ii}$ is positive, $\sqrt{d_{ii}}$ is a singular value of $A$ with corresponding left singular vector $u_i = Q(:,i)$, and corresponding right singular vector $v_i = \frac{A^Tu_i}{\sqrt{d_{ii}}}$.

The proof follows the same outline as above.

**Method 4**

Form $B = \begin{pmatrix} gI & A \\ A^T & gI \end{pmatrix}$, where $g = \|A\|_1$. Compute

```
>> [Q,D] = speig(B,k,'LR').
```

This is a shifted version of method 1 which assures the positive definiteness of $B$. The singular values are recovered by subtracting the shift $g$ from the computed eigenvalues of $B$. The proof follows from the proof of method 1 and the result about the eigenpairs of $A - \sigma I$ proved in section 2.7.

## 6.3 Syntax of `ssvd`

The sparse singular value decomposition function `ssvd` calculates the truncated singular value decomposition of a matrix $A$.

```
>> [U,S,V] = ssvd(A);
```

produces orthogonal matrices $U$ and $V$ and a diagonal matrix $S$ containing the 5 largest singular values of $A$, so that $B = USV^T$ is the best rank-5 approximation to $A$.

```
>> [U,S,V] = ssvd(A,k)
```

produces orthogonal matrices $U$ and $V$ and a diagonal matrix $S$ containing the $k$ largest singular values of $A$, so that $B = USV^T$ is the best rank-$k$ approximation to $A$.

```
>> [U,S,V] = ssvd(A,k,m)
```

computes the singular value decomposition according to method $m$, as described in the above section. The default method is method 1.

```
>> [U,S,V] = ssvd(A,k,method,gui)
```

displays the `speig` graphical user interface as the singular values are computed. $gui > 0$ displays the large "Progress Report" window; $gui < 0$ displays the small "Stop Button" window. Note that depending on the method selected, the converging numbers in the display window may not be the actual singular values.

```
>> d = ssvd(A,...)
```

returns a vector containing diag(S).

## 6.4 Which method should be chosen?

The choice of whether to use Method 2 or Method 3 should be made based on the dimensions of $A \in \mathcal{R}^{m\times n}$. Method 2 applies the matrix $A^TA \in \mathcal{R}^{n\times n}$, while Method 3 applies the matrix $AA^T \in \mathcal{R}^{m\times m}$. Therefore, it is reasonable to work with the method that uses the smaller matrix, since intermediate operations will require less storage and fewer floating-point operations.

However, information about the singular values can be lost through this "squaring". Suppose $\epsilon$ is the roundoff error on a certain machine, and the matrix $A$ has singular values $\sigma_1 = 1, \sigma_2 = \mathcal{O}(\sqrt{\epsilon})$. Even though the elements of $A^TA$ and $AA^T$ will have the same order of accuracy as the elements of $A$, their eigenvalues will be $\mu_1 = 1$ and $\mu_2 = \mathcal{O}(\epsilon)$. Therefore, the eigenvalue $\mu_2$ will not be computed accurately, and consequently, neither will the singular value $\sigma_2$ [28, p. 398].

However, since in many applications of the SVD, it is the largest singular values which are of interest, this squaring of the eigenvalues may not be a problem. The largest singular values can be determined safely and accurately. Alternate methods should be considered if the smaller singular values are required.

Another serious problem with methods 2 and 3 is the amount of fill-in that could occur in the formation of $AA^T$ and $A^TA$. Consider the case below, sketched using Matlab's `spy` command:

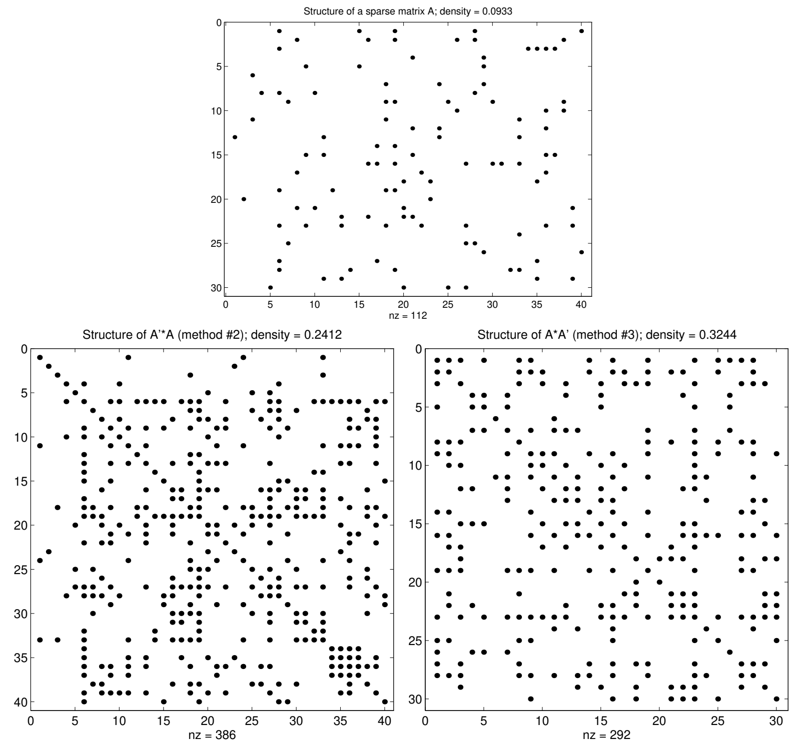



Figure 6.1: a. Structure of a sparse matrix $A$,
b. Structure of $A^TA$, c. Structure of $AA^T$

The fill-in would cause matrix-vector products in the iteration to take more time, and may slow the computations appreciably. Consequently, methods 2 and 3 are implemented in `ssvd` using helper functions which do not explicitly form the matrices $A^TA$ and $AA^T$. Instead, $A$ and $A^T$ are successively applied to the input vector in the appropriate order.

Methods 1 and 4 have the advantage that the eigenvalues computed by `speig` are actually the singular values of $A$. Furthermore, the matrix $B$ is proportionally sparser than $A$, and matrix-vector products are only twice as difficult. Although the matrix $B$ which is formed is of order $m + n$, this should not be a problem since the sparse storage class is used to form $B$. Therefore, method 1 is used as the default, with methods 2, 3, and 4 provided for comparison and choice.

The possibility of using polynomial acceleration to speed convergence of the largest singular values is being considered.

# Chapter 7

# Test cases

We now present the results of `speig` applied to several test cases which arise from real-world applications.

## 7.1 Modes of vibration of an L-shaped membrane

The L-shaped membrane is a familiar sight to any Matlab user; it is the MathWorks' company logo and appears on the cover of the Matlab user's manuals. This membrane is actually the first eigenfunction of the two-dimensional Laplacian operator with Dirichlet boundary conditions on an L-shaped domain. That is, it is the "first" solution of the partial differential equation:

$$\begin{aligned} \Delta u = \lambda u &\qquad \text{on } L \\ u = 0 &\qquad \text{on } \partial L \\ \text{where } L = \{[-1,1]\times[-1,1]\} &- \{[-1,0]\times[0,1]\} \end{aligned}$$

The problem has a rich classical history and the eigenfunctions can be obtained using Bessel functions [7, 15].

There are an infinite number of eigenfunctions for this problem; however, we can discretize the Laplacian over the L-shaped domain and approximate its eigenfunctions by computing the eigenpairs corresponding to the eigenvalues of smallest real part. Moler derived the 12 eigenvalues of smallest real part for the continuous operator [7]; they are given by:

$$\begin{bmatrix} 9.6397238445 \\ 15.19725192 \\ 2\pi^2 \approx 19.73921 \\ 29.5214811 \\ 31.9126360 \\ 41.4745099 \\ 44.948488 \\ 5\pi^2 \approx 49.348022 \\ 5\pi^2 \approx 49.348022 \\ 56.709610 \\ 65.376535 \\ 71.057755 \end{bmatrix}$$

We will attempt to obtain this result and the MathWorks' logo using `speig`.

First we generate the L-shaped domain using a suitably large grid size, 64 points on a side.

```
>> n = 64;
>> h = 2/(n-1);
>> G = numgrid('L',64);
```

The `numgrid` command generates the domain and numbers the grid points in an "intuitive" way.

Now we form the two-dimensional discrete Laplacian using the `delsq` command:

```
>> A = delsq(G);
```

The matrix $A$ is of order 2883, but is very sparse, with at most five entries per column. The density of $A$ is .0017.

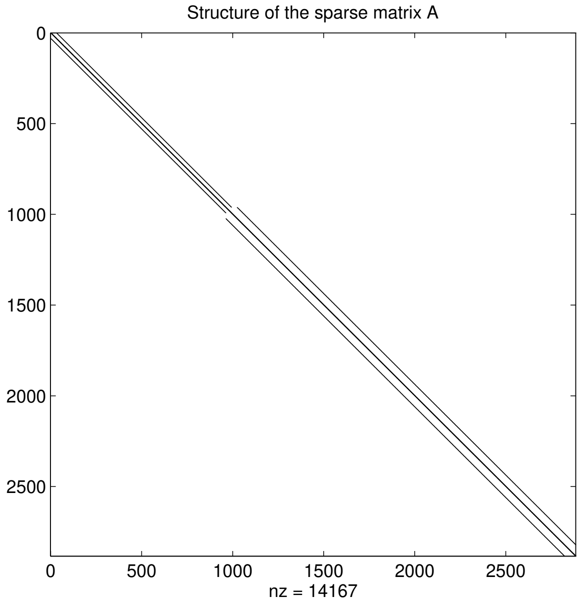



Figure 7.1: Structure of the discretized two-dimensional Laplacian over the L-shaped membrane

Each column corresponds to a grid point; we have reordered the grid as a single long vector instead of a two-dimensional array. Once we compute eigenvectors of $A$, we can use the numbering scheme in $G$ to recover a two-dimensional eigenfunction approximation over the L-shaped domain.

Now we compute the 12 eigenpairs of $A$ of smallest real part using `speig`:

```
>> [v,d]=speig(A,12,'SR');
```

We recover the eigenvalues of the Laplacian over the L-shaped domain using the following scaling:

```
>> e = diag(d)/(h^2);
```

```
>> e

e =

    9.3914
   14.9531
   19.5196
   29.3112
   31.3711
   40.5923
   43.9461
   48.7238
   48.7257
   55.6510
   64.6118
   70.2767
```

We can see that the computed eigenvalues are quite close to Moler's analytic eigenvalues. Of course, since the matrix is a discretization of a continuous operator, refining the mesh will increase the accuracy of the eigenvalues. However, the eigenvalues in $e$ are within default tolerance to the actual eigenvalues of $A$. The convergence history of the eigenvalues is illustrated by the graph below:

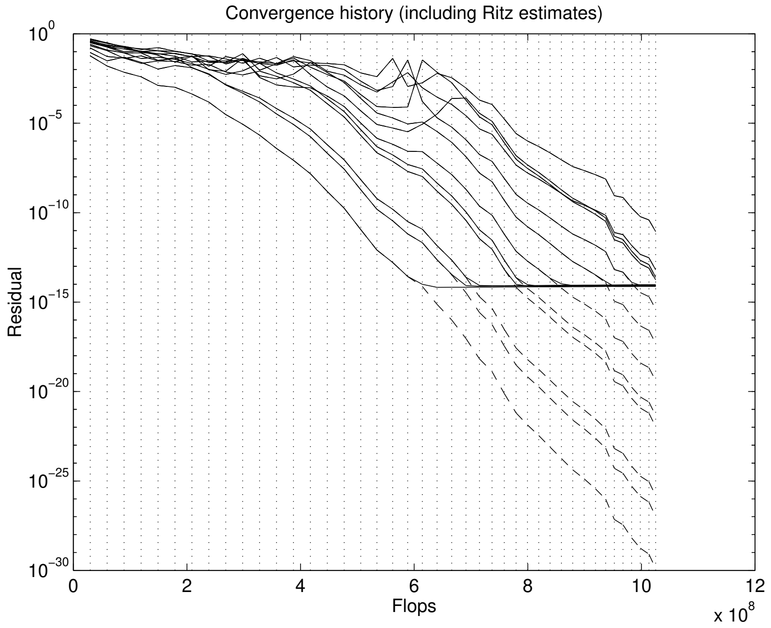



Figure 7.2: Convergence history for the L-shaped membrane

The x-axis corresponds to the cumulative number of floating-point iterations. A vertical dotted line indicates the beginning of a new iteration. Notice that later iterations require fewer floating point operations. This phenonmenon is due to the "moving boundary" incorporated into `speig`; when the residual for an eigenvalue is suitably low, we subtract one from $k$, the number of eigenvalues to calculate, and add 1 to $p$, the number of shifts to apply. Therefore the algorithm actually speeds up as eigenvalues converge.

The y-axis corresponds to the residual of each Ritz pair. The solid lines are the actual residuals $\|Ax - x\theta\|$, where $(x, \theta)$ is the Ritz pair. The dotted lines are the Ritz estimates for these Ritz pairs. We can see that the Ritz estimates are good approximations to the actual residuals for several iterations, but eventually become inaccurate due to machine roundoff.

This convergence history is characteristic of what is observed during a typical application of `speig`. The dominant Ritz value (represented by the lowest line) converges quickly, while the others take several more iterations to catch up. The series of peaks around $6 \cdot 10^8$ flops corresponds to the discovery of the double eigenvalue at $5\pi^2$. Once the double eigenvalue has been found, the residuals quickly decrease. In general, one must be careful not to set the tolerance for convergence too low; otherwise the results may be inaccurate. For example, if we had set the tolerance to $10^{-1}$ in this problem, we never would have detected the double eigenvalue.

Now we will recover the L-shaped membrane itself. We scale the eigenvectors so that the largest component of each is 1:

```
>> v = v*diag(1./max(abs(v)));
```

It takes some tricky Matlab indexing to transform the first eigenvector into an eigenfunction approximation over the L-shaped domain:

```
>> w = v(:,1);
>> m = zeros(size(G));
>> m(G>0) = w(G(G>0));
>> mesh(m)
```

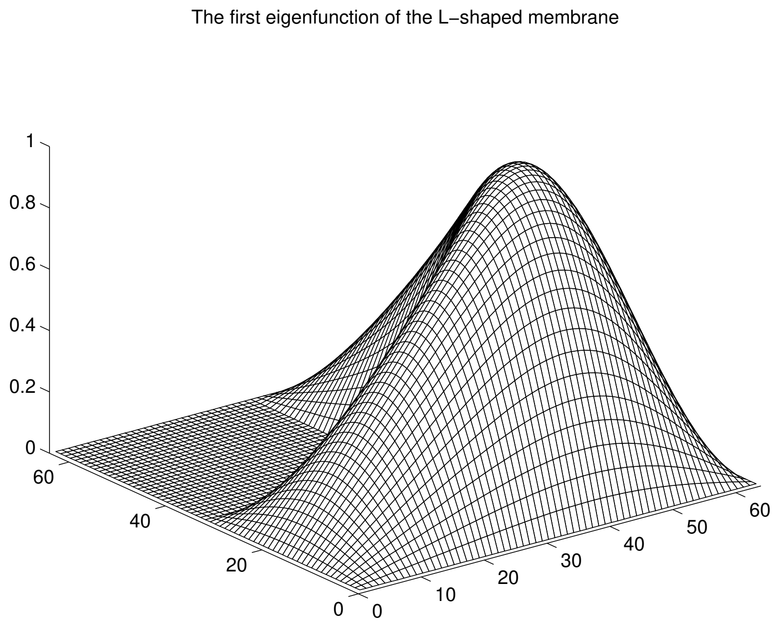



Figure 7.3: The MathWorks' logo.

We can see that we indeed recover the famous MathWorks logo (with minor differences because of the boundary conditions). The other eigenfunctions have similar patterns of peaks distributed symmetrically over the L-shaped domain.

## 7.2 A Tolosa matrix

The Tolosa matrices take their name from the city of Toulouse, where the Aerospatiale Aircraft Division and CERFACS are located [9]. This family of matrices arises in the stability analysis of a model of an airplane in flight. [25]. Tolosa matrices are very sparse and have a $5 \times 5$ block structure; their order $n$ is always at least 90 and divisible by 5. For more information about the structure of Tolosa matrices, see the brief sketch included with the Harwell-Boeing collection [9] and the references cited therein.

We will consider a Tolosa matrix of order 340, whose structure is illustrated below:

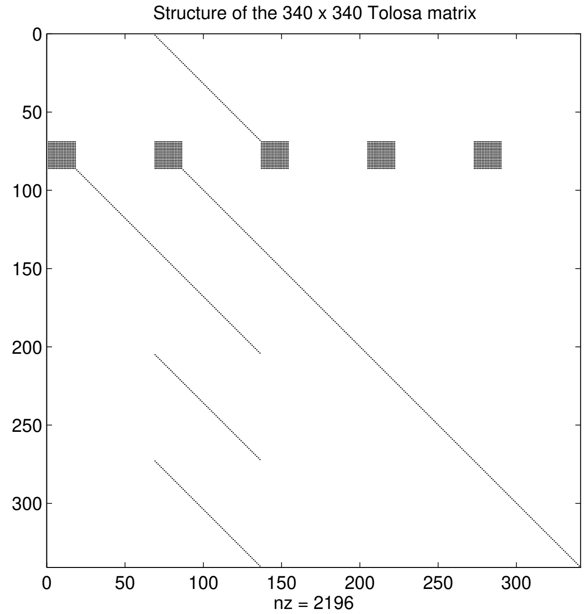



Figure 7.4: Structure of the $340 \times 340$ Tolosa matrix.

We generate this matrix using Matlab functions provided by Chao Yang of the CAAM department, Rice University:

```
>> tolosax;
>> A = atolosa(xs,340/5);
```

Since this is a relatively small problem, we can compute all the eigenvalues using normal full `eig`:

```
>> [V,D] = eig(full(A));
```

The spectrum looks like:

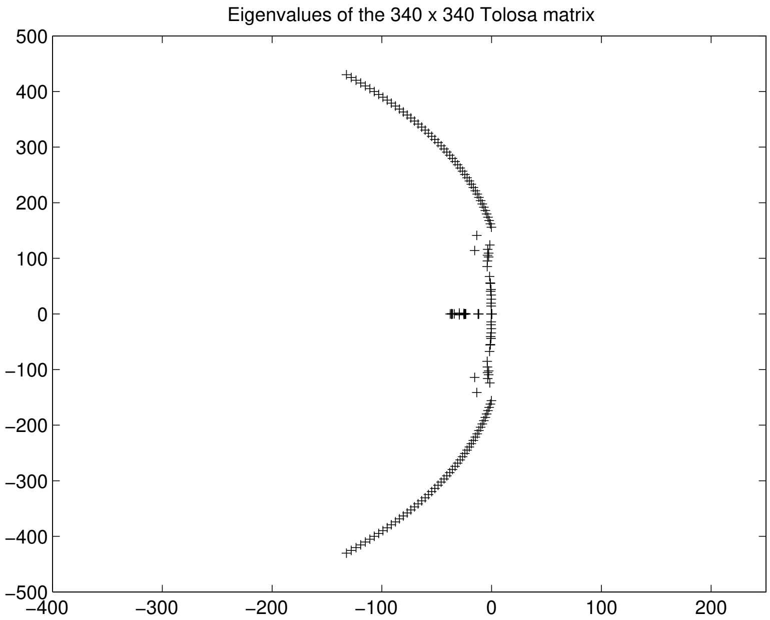



Figure 7.5: Spectrum of the Tolosa matrix

We are interested in the eigenvalues of largest imaginary part. While it would be easy to implement a *sigma* = 'LI' feature in `speig`, we choose instead to find the eigenvalues closest to a suitable shift in the complex plane. In this example we will use $\sigma = -150 + 410j$, and find 30 eigenvalues. The call to `speig` looks like:

```
>> [v,d]=speig(A,30,-150+410*j));

residnorm =

   2.770144723987897e-07
```

The eigenvalues converge impressively quickly, in only 2 iterations.

We try a more ambitious task of finding the eigenvalues of largest imaginary part when the shift we give is far away from the eigenvalues; for example, we will use $\sigma = -100 + 9000j$. Furthermore, we will use an options structure to specify we want the tolerance to be $10^{-10}$.

```
>> [v,d]=speig(A,30,-100+9000*j,speigset('tol',1e-10));
```

Despite the inaccurate shift, the algorithm still converges after only 6 iterations.

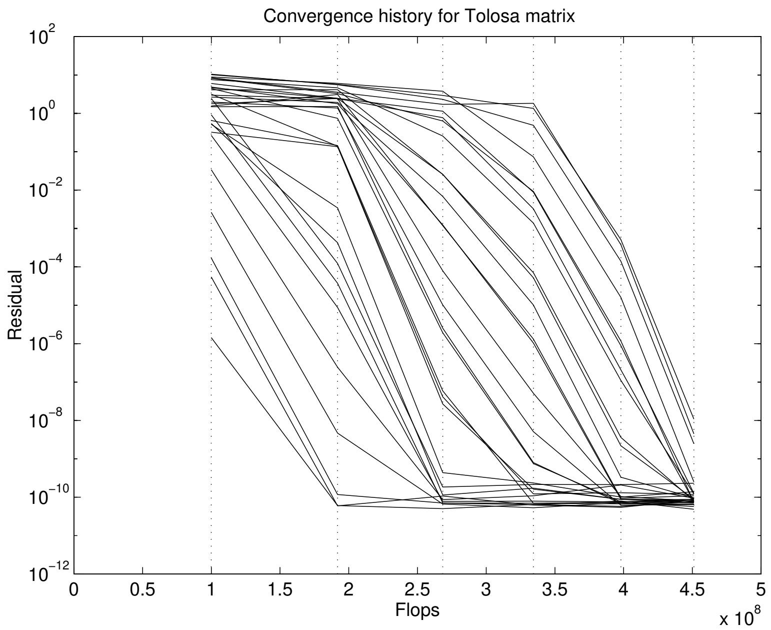



Figure 7.6: Convergence history of the Tolosa matrix

We recover the correct eigenvalues:

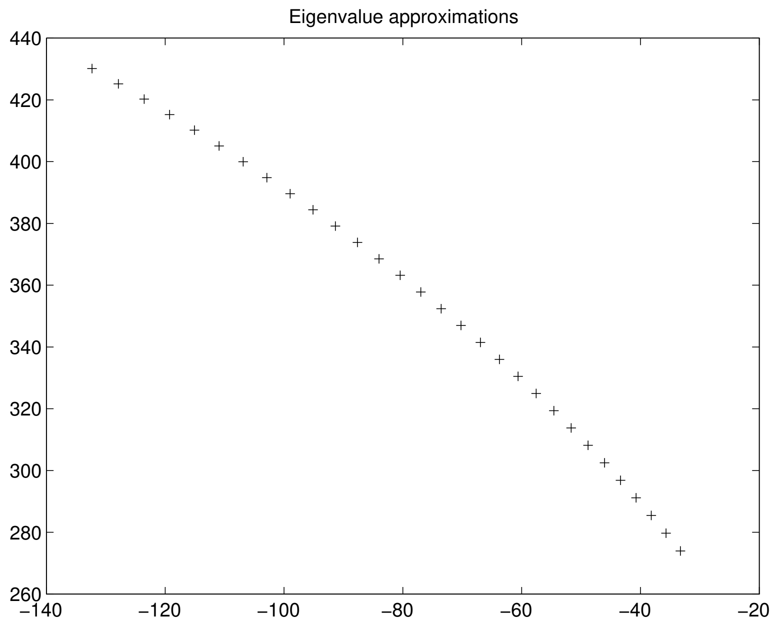



Figure 7.7: Eigenvalues of the Tolosa matrix

## 7.3 The Brusselator matrix: finding the eigenvalues of an operator

We consider a well-known model of the Belousov-Zhabotinski chemical reaction known as the Brusselator model [8, p. 312]. The concentrations of two chemical components are modeled by a pair of coupled partial differential equations. We seek to determine the stability of the reaction at a certain equilibrium; therefore, we need to consider the eigenvalues of the Jacobian at this equilibrium. We call the Jacobian the Brusselator matrix. This matrix can be built from the identity matrix and the one-dimensional Laplacian $T$:

$$B = \begin{pmatrix} \tau_1 T + \alpha I & \beta I \\ \gamma I & \tau_2 T + \delta I \end{pmatrix}$$

As in the L-shaped membrane problem, we will approximate some of the infinite number of eigenvalues of the Brusselator matrix by computing eigenvalues of its discrete counterpart. This matrix is relatively easy to construct; we use the following m-file:

```
function Br = bruss(n)

%  Brusselator matrix, p. 52 Saad
%  The Brusselator matrix is a 2 x 2 block matrix involving T and I.

e = ones(n,1);
T = spdiags([e -2*e e], -1:1, n, n);   % 1-D Laplacian
I = speye(n);                          % Identity matrix

Dx = 0.008;                            % Parameters from Saad
Dy = 0.004;
A  = 2;
B  = 5.45;
L  = 0.51302;                          % Bifurcation occurs here

h = 1/(n+1);
t1 = Dx/(h*L)^2;
```

```
t2 = Dy/(h*L)^2;

x = A;                                      % The equilibrium
y = B/A;

Br = [t1*T + (2*x*y-(B+1))*I,  x^2*I ; (B-2*x*y)*I , t2*T - x^2*I];
```

Here is an illustration of the Brusselator matrix's structure:

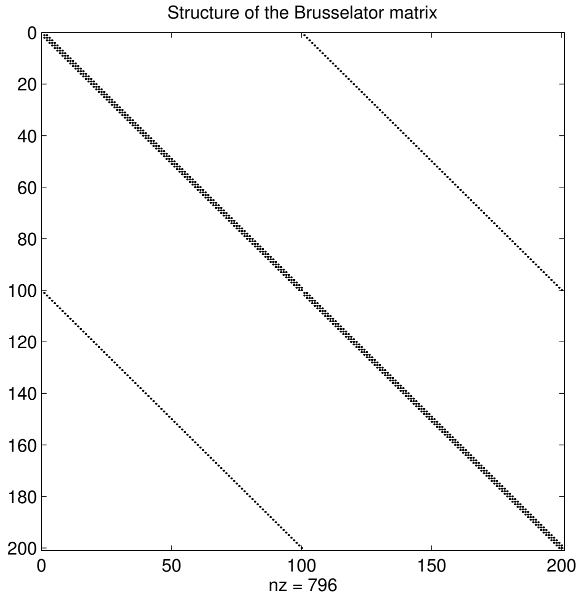



Figure 7.8: Structure of the 200 x 200 Brusselator matrix

Because of the underlying structure of the Brusselator matrix, it is possible to compute a closed form expression for its eigenvalues; this formula entails solving quadratic equations whose coefficients involve the parameters and the known eigenvalues of the one-dimensional discrete Laplacian $T$. (These eigenvalues can be found in [13, p. 99].) However, we will again use `speig` to perform the numerical computation. The eigenvalues of largest real part are of interest, since they determine the stability of the equilibrium.

Here is a graph of the right side of the spectrum of the Brusselator matrix. The left side continues along the imaginary axis to the left.

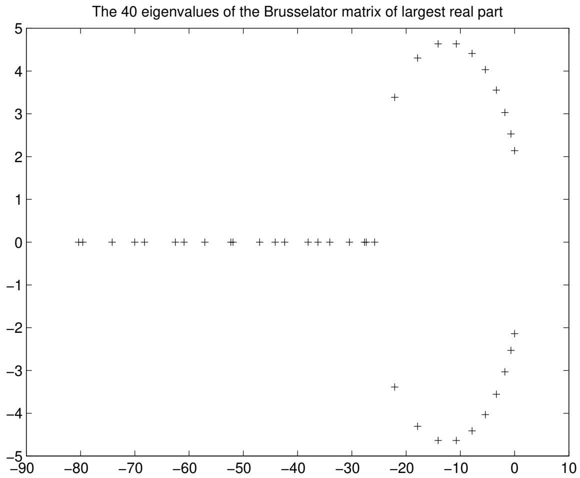



Figure 7.9: Rightmost eigenvalues of the Brusselator matrix

The Brusselator matrix is highly structured. Therefore it is reasonable to believe some gains could be both in speed and accuracy by creating an m-file which encapsulates the action of the Brusselator operator. If we wish to compute $w = Bv$, we can write $v$ in the block form $v = \begin{pmatrix} x \\ y \end{pmatrix}$ so that

$$\begin{aligned} w &= Bv \\ &= B\begin{pmatrix} x \\ y \end{pmatrix} \\ &= \begin{pmatrix} \tau_1 T + \alpha I & \beta I \\ \gamma I & \tau_2 T + \delta I \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} \\ &= \begin{pmatrix} \tau_1 Tx + \alpha x + \beta y \\ \tau_2 Ty + \gamma x + \delta y \end{pmatrix} \end{aligned}$$

Recall that in Section 5.6.2, we created an m-file `lap.m` which produces the action of T on a set of input vectors:

```
function w = lap(v);

n = size(v,1);
w(2:(n-1),:) = -2*v(2:(n-1),:) + v(1:(n-2),:) + v(3:n,:);
w([1,n],:) = -2*v([1,n],:) + v([2,n-1],:);
```

Therefore, we can write a Brusselator m-file as follows:

```
function w = bruss2(v)

global tau1 tau2 alpha beta gamma delta n

x = v(1:n,:);
y = v(n+1:2*n,:);

w = [ tau1 * lap(x) + alpha * x + beta * y ; ...
      tau2 * lap(y) + gamma * x + delta * y ];
```

The global variables are calculated only once in the base workspace to avoid unnecessary computation. It turns out that the operations `w = B*v` and `w = bruss2(v)` require roughly the same number of floating point operations so the gain in speed will probably be negligible.

Now we use `speig` to attack the problem. We let $n = 200$ and suppose we are interested in the 10 eigenvalues of largest real part. We use the syntax:

```
>> [v,d]=speig('bruss2',10,'LR',speigset('n',200));
```

It takes several iterations for the algorithm to converge, but eventually we reach a stage where the Ritz values are good approximations of the rightmost eigenvalues.

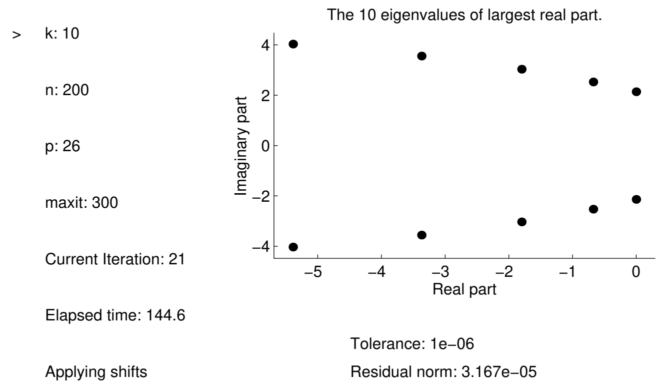



Figure 7.10: An intermediate stage

We finally obtain the 10 rightmost eigenvalues:

```
>> diag(d)

ans =

  1.8200e-05 - 2.1395e+00i
  1.8200e-05 + 2.1395e+00i
 -6.7471e-01 - 2.5286e+00i
 -6.7471e-01 + 2.5286e+00i
 -1.7985e+00 - 3.0322e+00i
 -1.7985e+00 + 3.0322e+00i
 -3.3704e+00 - 3.5553e+00i
 -3.3704e+00 + 3.5553e+00i
 -5.3887e+00 - 4.0323e+00i
 -5.3887e+00 + 4.0323e+00i
```

We can see that since there are eigenvalues in the right half plane, the equilibrium will be unstable. This example was chosen for its closeness to the bifurcation between stability and instability. Here is the convergence history:

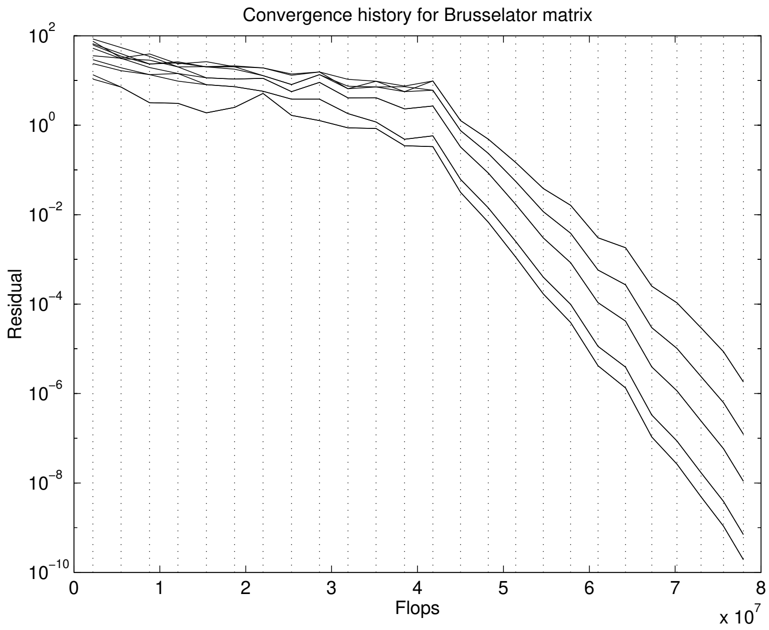



Figure 7.11: Convergence history for the Brusselator problem

## 7.4 The 1-D Laplacian with polynomial acceleration

We now compute 6 leftmost eigenvalues of the one-dimensional discrete Laplacian $T$ of order 625 in two different ways.

First, we explicitly create the matrix in Matlab:

```
>> n = 625;
>> e = ones(n,1);
>> T = spdiags([e -2*e e], -1:1, n, n);
```

Now we use `speig` to compute the eigenvalues:

```
>> o = speigset('tol',1000*eps);
>> [V,D] = speig(T,6,'SR',o);
```

The following eigenvalues were obtained:

```
-3.999974814523785e+00
-3.999899258729349e+00
-3.999773334519658e+00
-3.999597045066184e+00
-3.999370394808843e+00
-3.999093389455986e+00
```

We see that the residual $\|TV - VD\|$ is within the requested tolerance:

```
residnorm =

    4.596505711663322e-14
```

Furthermore, the computed eigenvectors are orthogonal to machine precision:

```
>> norm(v'*v - eye(size(d)))

ans =

    8.810505531885305e-15
```

The total elapsed time for the process was 803 seconds, and 88 Arnoldi iterations were taken. The total process required around $6.5 \cdot 10^8$ floating point operations. The convergence history is shown below:

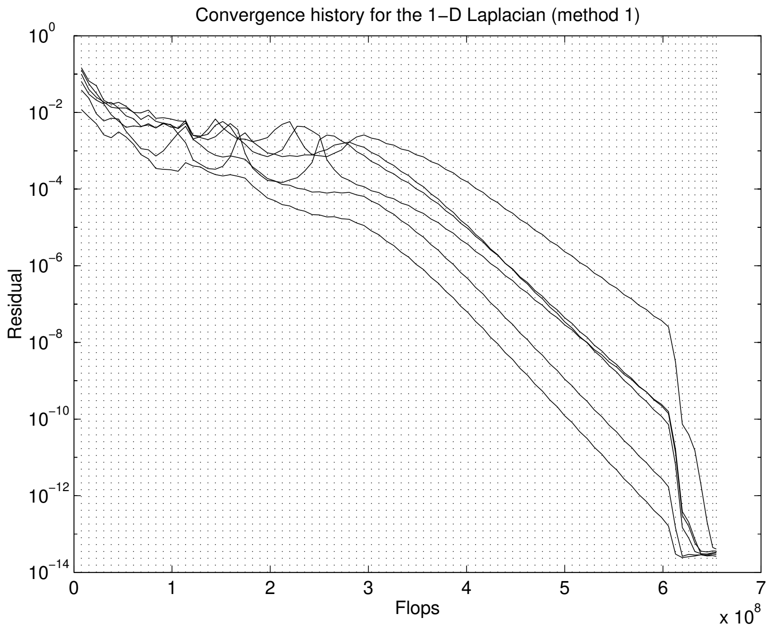



Figure 7.12: Convergence history, method 1.

The algorithm requires many iterations since the eigenvalues of interest are poorly separated. The process takes several minutes since the matrix $T$ is very large and matrix-vector products are time-consuming.

To demonstrate the advantage of using polynomial acceleration, we now use the m-file `lap.m` to represent the matrix $A$ and use a different calling sequence:

```
>> o = speigset('n',625,'tol',1000*eps,'dopoly',1, 'issym',1);
>> [V,D] = speig('lap',6,'SR',o);
```

The following eigenvalues were obtained:

```
-3.999974814523766e+00
-3.999899258729350e+00
-3.999773334519678e+00
-3.999597045066205e+00
-3.999370394808856e+00
-3.999093389455942e+00
```

These eigenvalues agree to 13 decimal places with the eigenvalues obtained by the first method.

The residual $\|TV - VD\|$ is again within the requested tolerance:

```
residnorm =

    8.640398339932052e-14
```

The computed eigenvectors are also orthogonal to machine precision:

```
>> norm(v'*v - eye(size(d)))

ans =

    5.077352722636109e-15
```

The total elapsed time for the second method was only 214 seconds, and only 29 iterations were taken. The total process required around $1.6 \cdot 10^8$ floating point operations. The convergence history is shown below:

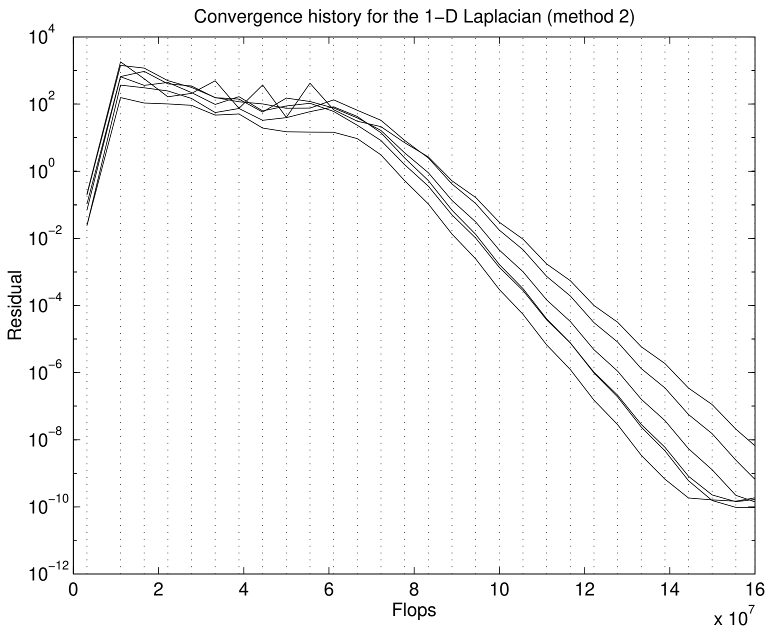



Figure 7.13: Convergence history, method 2.

Polynomial acceleration begins at the second iteration; the residuals are high at first but dampen quickly. It should be noted that the residuals in this convergence history are the residuals of the transformed problem and thus are not strictly comparable to the residuals in the previous graph.

We now demonstrate an example of the use of polynomial acceleration when *sigma* is a numeric shift. We will find the 6 eigenvalues of the order 625 discrete Laplacian closest to -2.5.

The calling sequence is:

```
>> opt = speigset('n',625,'issym',1,'dopoly',1);
>> [v,d] = speig('lap',6,-2.5,opt);
```

We obtain the results:

```
>> diag(d)

ans =

   -2.496601997844897e+00
   -2.506318410062859e+00
   -2.486873078469110e+00
   -2.516022070410546e+00
   -2.477131896962976e+00
   -2.525712734496627e+00

residnorm =

    5.556372395256302e-11
```

The process took 39 iterations and required around $2.1 \cdot 10^8$ floating point operations. The convergence history is shown below:

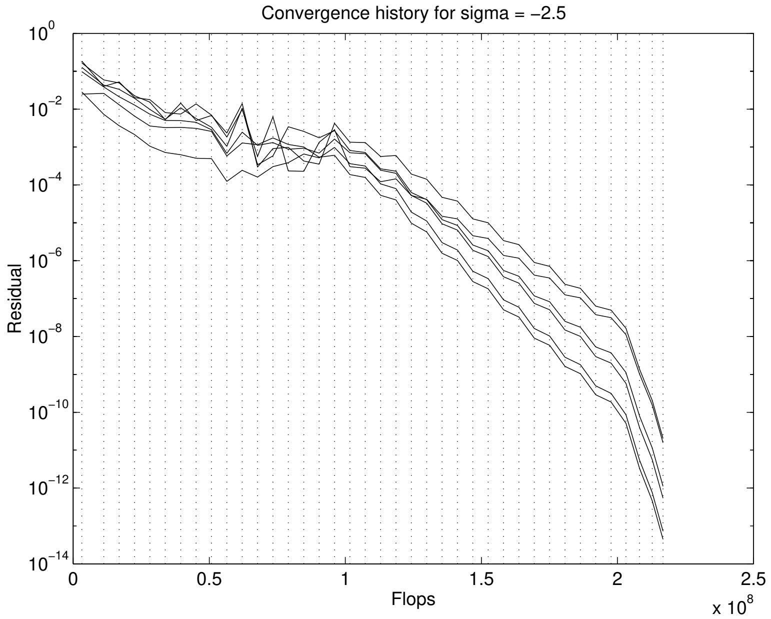



Figure 7.14: Convergence history, $sigma$ = -2.5

In these examples, polynomial acceleration allows us to make substantial gains in both speed and number of iterations. Expressing the operator as an m-file may take some thought, but allows us to enjoy the benefits of the powerful polynomial acceleration method.

## 7.5 A generalized example

We now consider a generalized eigenvalue problem taken from the Harwell-Boeing collection of sparse matrices. The problem is found in the library BCSSTRUC1, which includes matrices representing dynamic analyses in structural engineering [6]. Specifically, the problem is pair number 5 in this library and arises from a model of a transmission tower. The order of the problem is 153. The mass matrix $M$ is diagonal with positive entries, and the symmetric stiffness matrix $K$ has the structure illustrated below:

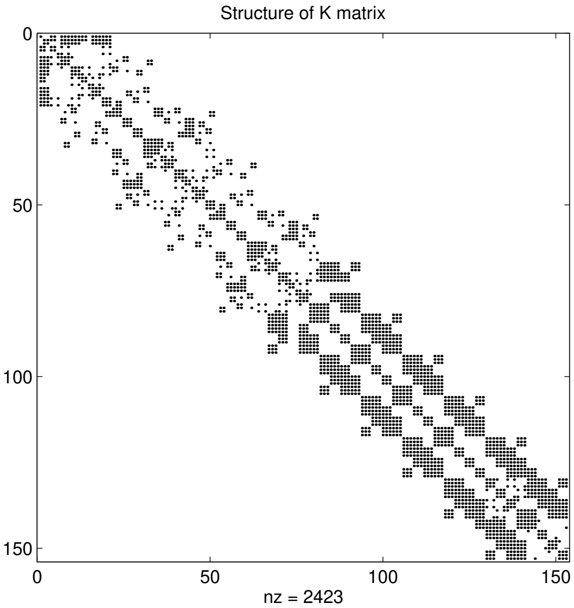



Figure 7.15: Structure of the stiffness matrix $K$.

The spectrum for the problem is spread between $2.5 \cdot 10^3$ to $3.0 \cdot 10^7$, so that the condition number of the problem is $1.3 \cdot 10^4$. We expect this spread to hinder convergence of the eigenpair estimates. We will attempt to compute the 10 eigenvalues of smallest real part using the following command syntax:

```
>> [v,d]=speig(k5,m5,10,'SR');
```

The process takes 79 iterations to converge; the convergence history is shown below:

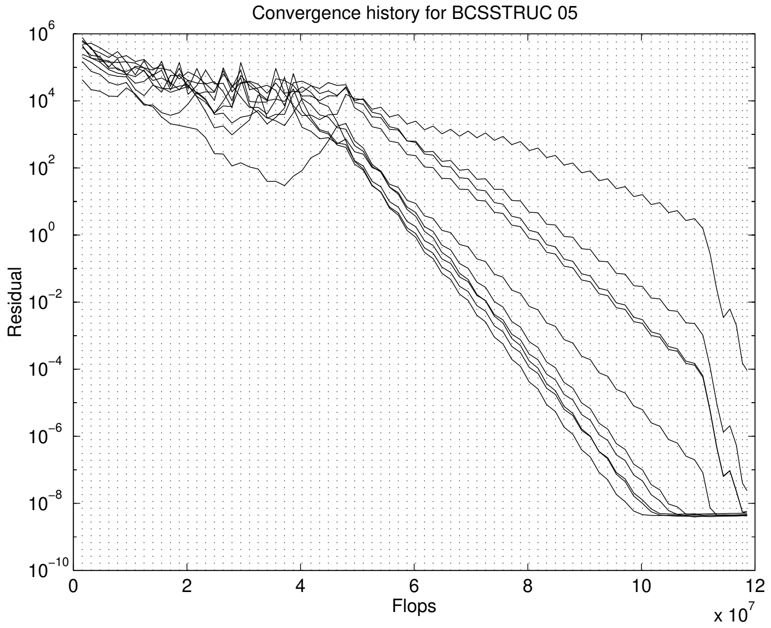



Figure 7.16: Convergence history for BCSSTRUC 05

The residual norm in this case is only $5.5 \cdot 10^{-5}$ although the default tolerance for a symmetric problem is $10^{-10}$; however, recall that the tolerance is actually defined as $\frac{\|AV-VD\|}{\|A\|}$, and in this case, $\|M^{-1}K\| = 3.1 \cdot 10^{7}$. This accounts for the seeming discrepancy. It should be noted that even relatively small generalized eigenvalue problems give rise to problems with high condition numbers and norms, which makes the solution of these problems difficult.

# Chapter 8

# Conclusions

We can use the Implictly Restarted Arnoldi Method to obtain approximate solutions to many eigenvalue problems that are posed in engineering contexts. This thesis does not discuss several refinements to the method which avoid the pitfalls to which a straightforward implementation such as `speig` may fall prey. Improvements such as locking and purging to protect already converged Ritz pairs [12, 22, 13], the use of Meerbergen shifts for the generalized problem, and more sophisticated deflation techniques to speed convergence [12] are not included in the `speig` code. The ARPACK collection of Fortran functions [14] is a more powerful tool than `speig` which incorporates these refinements. However, we conclude by noting that `speig` has several advantages over other implementations of the implicitly restarted Arnoldi method that are available today:

- Since `speig` is written in Matlab, the code is portable and it is highly readable. The power of the BLAS and LAPACK are already part of Matlab and the machine dependent details of their implementation are transparent to the user. The `speig` collection is written in high-level vectorized Matlab code which should be understandable to a user with moderate familiarity with Matlab.

- The `speig` code is extremely easy to use. The basic syntax is uncomplicated but highly flexible. The options structure allows lower-level parameters to be tuned without cluttering the main calling sequence. The input and output from `speig` can be immediately used by other Matlab functions.

- Polynomial accleration centered around a numeric shift works very well and, to the knowledge of the author, had not been previously implemented in any publicly available eigenvalue code.

- Distribution as part of Matlab version 5 makes `speig` immediately accessible to a wide range of users from students to design engineers. The graphical user interface provides both novice and experienced users with intuition about the mechanics of the process, making `speig` a powerful tool for both engineering and educational environments.

# Bibliography


[1] James R. Bunch and Donald J. Rose. *Sparse Matrix Computations*. Academic Press, 1975.

[2] X. Chen and Thomas W. Parks. Analytic Design of Optimal FIR Smooth Passband Filters Using Zolotarev Polynomials. *IEEE Trans. Circuits Systems*, CAS–33:1065–1071, November 1986.

[3] Jane Cullum and Ralph A. Willoughby, editors. *Large Scale Eigenvalue Problems*. Elsevier Science Publishers B.V., 1986.

[4] Germund Dahlquist and Åke Björck. *Numerical Methods*. Prentice-Hall, 1974.

[5] J. Daniel, W.B. Gragg, L. Kaufman, and G.W. Stewart. Reorthogonalization and Stable Algorithms for Updating the Gram-Schmidt QR Factorization. *Math. Comp.*, 30:772–795, 1976.

[6] Iain S. Duff, Roger G. Grimes, and John G. Lewis. *Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I).*

[7] L. Fox, P. Henrici, and C. Moler. *SIAM J. Numer. Anal.*, 4:89–102, 1967.

[8] Martin Golubitsky and David G. Schaeffer. *Singularities and Groups in Bifurcation Theory*, volume 1. Springer-Verlag, 1985.

[9] Harwell-Boeing sparse matrix library. *Documentation for Tolosa matrix library.*

[10] I. N. Herstein. *Topics in Algebra*. John Wiley and Sons, 1975.

[11] David Kincaid and Ward Cheney. *Numerical Analysis*. Brooks/Cole, 1991.

[12] R.B. Lehoucq and D.C. Sorensen. Deflation Techniques for an Implicitly Restarted Arnoldi Iteration *(CAAM-TR 94-13)*. Technical report, Rice University, 1994, (to appear *SIAM J. Matrix Anal. Appl.*).

[13] Rich B. Lehoucq. *Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration*. PhD thesis, Rice University, 1995.

[14] Rich B. Lehoucq, Danny C. Sorensen, and Chao Yang. ARPACK User's Guide: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods. Technical report, Rice University, Department of Computational and Applied Mathematics, 1996.

[15] The MathWorks, Inc. *Matlab 4.0 Reference Guide*.

[16] Karl Meerbergen and Dirk Roose. Matrix Transformtions for Computing Rightmost Eigenvalues of Large Sparse Nonsymmetric Matrices. Technical report, Katholieke Universiteit Leuven, Department of Computer Science, 1994.

[17] Thomas W. Parks and C. Sidney Burrus. *Digital Filter Design*. John Wiley and Sons, 1987.

[18] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, 1980.

[19] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. Manchester University Press, 1992.

[20] Udo Schendel. *Sparse Matrices: Numerical Aspects with Applications for Scientists and Engineers*. Ellis Horwood Limited, 1989.

[21] Ivan W. Selesnick and C. Sidney Burrus. Exchange Algorithms for the Design of Linear Phase FIR Filters and Differentiators Having Flat Monotonic Passbands and Equiripple Stopbands. *IEEE Transactions on Circuits and Systems II*, to appear 1996.

[22] Danny C. Sorensen. Implicit Application of Polynomial Filters in a k-step Arnoldi Method. *SIAM J. Matrix Anal. Appl.*, 13:357–385, 1992.

[23] Danny C. Sorensen. CAAM 551 Lecture Notes: Iterative Methods for Large Scale Eigenvalue Problems and Linear Systems. Technical report, Rice University, Department of Computational and Applied Mathematics, 1995.

[24] Danny C. Sorensen. Implicitly Restarted Arnoldi / Lanzcos Methods for Large Scale Eigenvalue Calculations. In D. E. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms: Proceedings of an ICASE/LaRC Workshop, May 23-25, 1994, Hampton, VA*. Kluwer, 1995 (to appear).

[25] D.C. Sorensen and Chao Yang. A Truncated RQ-Iteration for Large Scale Eigenvalue Calculations *(CAAM-TR 96-06)*. Technical report, Rice University, Department of Computational and Applied Mathematics, 1996.

[26] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973.

[27] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, 1988.

[28] David S. Watkins. *Fundamentals of Matrix Computations*. John Wiley and Sons, 1991.

[29] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, 1965.