ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.

R. B. Lehoucq, D. C. Sorensen, C. Yang

8 Oct 97

Contents

List of Figures

List of Tables

Preface

The development of ARPACK began as a research code written in Matlab and then in Fortran77 in 1990. Initially, the code was developed to study and verify the properties of the Implicitly Restarted Arnoldi Method described in [13]. Preliminary experience with that code showed considerable promise in performance and also seemed to provide a solid foundation for the development of serious mathematical software for large structured eigenvalue problems.

During the academic year 1991-92, Dr. Phuong Vu (at that time with Cray Research) was granted permission to work in a half time appointment to the NSF Center for Research on Parallel Computation at Rice University on the development of ARPACK. At the outset, we attempted to design the software to be efficient and portable on conventional high performance computing architectures. Of course, our design was also intended to be easily modified to effectively utilize a variety of parallel architectures (resulting in P_ARPACK). Phuong's experience with users at Cray Research suggested that a reverse communication interface was essential. We are deeply indebted to Phuong for the design and implementation of this interface. He set the coding and documentation style and developed the initial implementation of all of the basic computational routines for real (single and double precision) matrices. His fundamental design has served us well as we have improved and expanded upon the package over the past few years.

We wish to thank the numerous users with applications and also our fellow numerical analysts who worked with initial "alpha" and then "beta" versions of the code. Their feedback and patience throughout this development has been invaluable. This interaction has often given us a wonderful sense of community and the words "it solved my problem!" always managed to brighten up the drudgery of developing and maintaining the software. In particular, we would like to mention Jean-Philippe Brunet, Daniela Calvetti, Lawrence Cowsar, Olivier Daube, David Day, Stewart Edwards, Ralph T Goodwin III, Ed Hayes, Lennart Johnsson, Michiel Kooper, Kristi Maschhoff, Karl Meerbergen, Frank Milde, Seymour Parter, Phil Pendergast, George Phillips, John Red-Horse, Lothar Reichel, Tod Romo, Will Sawyer, Jennifer Scott, Rajesh Kumar Singh, Allison Smith, Allister Spence, Zdenko Tomasic, Henk Van der Vorst.

ARPACK is freely available through the world wide web and by anonymous ftp (See Chap. 1). It relies heavily upon the LAPACK software [1] and upon the BLAS [5, 3, 2]. Portability with performance, accuracy, and robustness is a

direct consequence. We are greatly indebted to the authors of that software and more generally to the larger numerical analysis community that has contributed in many ways to its development.

Finally, we would like to thank the National Science Foundation, DARPA, and the Department of Energy for their generous support of this project (See Chap. 1 for full citation).

How to use this Guide

This is a users guide. It is not a novel or a textbook and is not intended to be read sequentially. There is a great deal of repetition amongst several of the subsections in Chapter 3 and also in Appendix A. We decided that it would be better to discuss each major problem class: Real Symmetric, Real Nonsymmetric and Complex as complete individual units even though this inevitably resulted in redundancy. We expect users to turn to the section that discusses the problem class of interest and to find everything related to that class in one complete section. We think this is preferable to searching back and forth to a general description in order to understand the special case of interest.

Chapter 1 gives an overview and contains general information. Chapter 2 provides installation instructions and describes how to get started. It is recommended for those who are just beginning with eigenvalue computations and also for those who are unfamiliar with reverse communication. Chapter 3 gives a detailed description of how to use all of the capabilities of ARPACK. Those wishing to learn a little about the underlying numerical methods should turn to Chapter 4. This discussion provides a broad overview of the methods. It gives a reasonably detailed description of the Arnoldi process with implicit restarting and what to expect. It also attempts to provide some understanding of the spectral transformation. Numerous references are provided for those who desire a more detailed level of understanding. Chapter 5 discusses implementation and usage details within the main computational routines. Experienced users of large scale eigenvalue methods can probably turn directly to Appendix A and find the discussion of the driver routine that is appropriate for their problem. These drivers are intended to be used as templates that are easily modified for a particular application. Trace debugging and check-pointing are discussed in Appendix B. Finally, there are listings of the top level reverse communication interface routines `XYaupd` for reference. The source code for all of the computational routines is available with the distribution. Each of these is fully documented in the header and many users have found that documentation sufficient to get started with.

Chapter 1

Introduction to ARPACK

ARPACK is a collection of Fortran77 subroutines designed to solve large scale eigenvalue problems. ARPACK stands for ARnoldi PACKage. ARPACK software is capable of solving large scale Hermitian, non-Hermitian, standard or generalized eigenvalue problems from significant application areas. The software is designed to compute a few, say k, eigenvalues with user specified features such as those of largest real part or largest magnitude using $n \cdot \mathcal{O}(k) + \mathcal{O}(k^2)$ storage. No auxiliary storage is required. A set of Schur basis vectors for the desired k dimensional eigen-space is computed which is numerically orthogonal to working precision. Eigenvectors are also available upon request.

The Arnoldi process is a technique for approximating a few eigenvalues and corresponding eigenvectors of a general $n \times n$ matrix. It is most appropriate for large structured matrices $\mathbf{A}$ where structured means that a matrix-vector product $\mathbf{w} \leftarrow \mathbf{Av}$ requires $\mathcal{O}(n)$ rather than the usual $\mathcal{O}(n^2)$ floating point operations (Flops). This software is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix $\mathbf{A}$ is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). These variants may be viewed as a synthesis of the Arnoldi/Lanczos process with the Implicitly Shifted QR scheme that is suitable for large scale problems. For many standard problems, a matrix factorization is not required. Only the action of the matrix on a vector is needed.

In this chapter, we give an overview of the package. Chapter 2 explains how the user can quickly start using ARPACK while Chapter 3 gives a comprehensive description of how to utilize the full capabilities of ARPACK. An overview of the theory of Krylov subspace projection methods and the underlying algorithms implemented in ARPACK is the subject of Chapter 4. The final chapter discusses the implementation details of the main computational routines in ARPACK. Appendix A is a guide on how to use the example driver routines as templates. Experienced users who are already familiar with large scale eigenvalue computations may find it most productive to go directly to this appendix, locate the suitable driver and modify that for the particular application. Appendix B describes the trace debugging capability that is easily turned on in order to monitor progress and output important intermediate computed quantities. Checkpointing to guard against loss of intermediate computational results due to system or hardware failure is possible. A description of how to recover and restart in the event of a fault is provided in Appendix B.

1.1 Important Features

The important features of ARPACK are:

- A reverse communication interface.
- Ability to return k eigenvalues which satisfy a user specified criterion such as largest real part, largest absolute value, largest algebraic value (symmetric case), etc. For many standard problems, the action of the matrix on a vector $\mathbf{w} \leftarrow \mathbf{Av}$ is all that is needed.
- A fixed pre-determined storage requirement suffices throughout the computation. Usually this is $n \cdot \mathcal{O}(k) + \mathcal{O}(k^2)$ where k is the number of eigenvalues to be computed and n is the order of the matrix. No auxiliary storage or interaction with such devices is required during the course of the computation.
- Sample driver routines are included that may be used as templates to implement various spectral transformations to enhance convergence and to solve the generalized eigenvalue problem.
- Special consideration is given to the generalized problem $\mathbf{Ax} = \mathbf{Mx}\lambda$ for singular or ill-conditioned symmetric positive semi-definite $\mathbf{M}$.
- Eigenvectors and/or Schur vectors may be computed on request. A Schur basis of dimension k is always computed. The Schur basis consists of vectors which are numerically orthogonal to working accuracy. Computed eigenvectors of symmetric matrices are also numerically orthogonal.
- The numerical accuracy of the computed eigenvalues and vectors is user specified. Residual tolerances may be set to the level of working precision. At working precision, the accuracy of the computed eigenvalues and vectors is consistent with the accuracy expected of a dense method such as the implicitly shifted QR iteration.
- Multiple eigenvalues offer no theoretical difficulty. This is possible through deflation techniques similar to those used with the implicitly shifted QR algorithm for dense problems. With the current deflation rules, a fairly tight convergence tolerance and sufficiently large subspace will be required to capture all multiple instances. However, since a block method is not used, there is no need to "guess" the correct block size that would be needed to capture multiple eigenvalues.

1.2 Getting Started

Easy to use sample driver routines are available. These *simple* drivers have been constructed to illustrate the use of ARPACK in the simplest cases of finding a few eigenvalues and corresponding eigenvectors of largest magnitude. Simple drivers for all precisions and data types are provided and these may be used as templates to easily begin using ARPACK. Chapter 2 describes how to get started by using these example driver programs.

1.3 Reverse Communication Interface

The reverse communication interface is one of the most important aspects of the design of ARPACK. This interface avoids having to express a matrix-vector product through a subroutine with a fixed calling sequence. This means that the user is free to choose any convenient data structure for the matrix representation. Moreover, if the matrix is not available explicitly, the user is free to express the action of the matrix on a vector through a subroutine call or a code segment. It is not necessary to conform to a fixed format for a subroutine interface and hence there is no need to communicate data through the use of `COMMON`.

A typical usage of this interface is illustrated with the example in Figure 1.1. This shows a code segment of the routine the user must write to set up the reverse communication call to the top level ARPACK routine `snaupd` to solve a nonsymmetric eigenvalue problem. As usual, with reverse communication, control is returned to the calling program when interaction with the matrix $\mathbf{A}$ is required. The action requested of the calling program is simply to perform the task indicated by the reverse communication parameter `ido` (in this case multiply the vector held in the array `workd` beginning at location `ipntr(1)` and inserting the result into the array `workd` beginning at location `ipntr(2)`). Note that the `call` to the subroutine `matvec` in this code segment is simply meant to indicate that this matrix-vector operation is taking place. The user is free to use any available mechanism or subroutine to accomplish this task. In particular, no specific data structure is imposed and indeed, no explicit representation of the matrix is even required. One only needs to supply the action of the matrix on the specified vector.

1.4 Availability

The codes are available by anonymous ftp from

```
ftp.caam.rice.edu
```

or by connecting directly to the `URL`

```
http://www.caam.rice.edu/software/ARPACK
```

```
10  continue
    call snaupd (ido, bmat, n, which,...,workd,..., info)
    if (ido .eq. newprod) then
       call matvec ('A', n, workd(ipntr(1)), workd(ipntr(2)))
    else
       return
    endif
    go to 10
```

Figure 1.1: An example of the reverse communication interface used by ARPACK.

To get the software by anonymous ftp, connect by ftp to `ftp.caam.rice.edu` and `login` as `anonymous`. Then change directories to

```
software/ARPACK
```

or connect directly to the URL as described above and follow the instructions in the `README` file in that directory. The ARPACK software is also available from Netlib in the directory `ScaLAPACK`.

1.5 Installation

The instructions in the `README` file that will explain how to retrieve a compressed `tar file` and how to this file. A few options are available. One of these is to retrieve the file

```
dist96.tar.Z
```

Then issue the instruction

```
zcat dist96.tar.Z | tar -xvf -
```

This will automatically create a directory named ARPACK. This directory should have the following contents:

```
BLAS
DOCUMENTS
EXAMPLES
LAPACK
README
SRC
UTIL
Makefile
ARmake.inc
ARMAKES
```

Instructions on how to proceed are given in the `ARPACK/README` file. After minor modifications to the file `makefile`, issuing the command `make lib` will compile all subroutines and create the archive `libarpack_<PLATFORM>.a` in the `ARPACK` directory. Here, `<PLATFORM>` denotes the environment where the ARPACK library is built. For example, if the `make` were done on a SUN Sparc4, the library would be archived in `libarpack_SUN4.a`.

Instructions in `README` explain how to proceed. Disk storage requirements for the directory structure shown above is just under 5 Megabytes.

1.6 Documentation

In addition to this user's guide, complete documentation of usage, data requirements, error and warning conditions is provided in the header of the source code for each subroutine. There is sufficient documentation included in the `README` files, `DOCUMENTS` directory, and headers of the codes in `SRC` and the `EXAMPLES` subdirectories to begin using ARPACK. This user's guide is intended to further explain and supplement that documentation. In addition to a detailed description of the capabilities, structure, and usage of ARPACK, this document is intended to provide a cursory overview of the Implicitly Restarted Arnoldi/Lanczos Method that the software is based upon. The goal is to provide some understanding of the underlying algorithm, expected behavior, and capabilities as well as limitations of the software.

Extensive references to the literature on large scale eigenvalue methods and software are given here. Additional information and articles on the algorithmic theory and on applications of ARPACK may be found at

`http://www.caam.rice.edu/software/ARPACK`

1.7 Dependence on LAPACK and BLAS

ARPACK is dependent upon a number of subroutines from LAPACK [1] and the BLAS [3, 2, 5]. The necessary routines are distributed along with the ARPACK software. Whenever possible, BLAS routines that have been optimized for the given machine should be used in place of the ones provided with ARPACK. A list of routines required from these two sources is available in Chapter 5. If local installations of BLAS and/or LAPACK are available then the corresponding ARPACK subdirectories may be deleted and the local installations may be pointed to instead. Care should be taken to verify consistency of the version dates of the local installations with the version dates of the BLAS and LAPACK routines provided with ARPACK.

NOTE: The LAPACK library on your system MUST be the public release. The current release is version 2.0. If you are not certain if the public release has been installed, we strongly recommend that you compile and link to the subset of LAPACK included with ARPACK.

1.8 Expected Performance

ARPACK has been designed for straightforward adaptation to a variety of high performance architectures including vector, super-scalar and parallel machines. It is intended to be portable and efficient across a wide range of computing platforms. Computationally intensive kernels are all expressed through BLAS operations and if the number k remains fixed as n increases the performance will scale asymptotically to the Level 2 BLAS operation `_GEMV`. Computational rates near maximum achievable peak are possible on multi-vector processors such as CRAY-C90 and on workstation clusters such as SGI Power Challenge.

The package is written in the ANSI standard Fortran 77 language with the one exception of *include* files. These are associated solely with the trace debugging facility provided with ARPACK. Each of the ARPACK subroutines reference two include files for debugging and timing purposes (see Appendix B). These references may be easily deleted if they are incompatible with your system.

1.9 P_ARPACK

A parallel version of the ARPACK library is also available. The message passing layers currently supported are BLACS and MPI. Parallel ARPACK (P_ARPACK) is provided as an extension to the current ARPACK library. P_ARPACK has been installed on CRAY-T3D, Intel Delta and Paragon, Intel Paragon IBM-SP2, an SGI cluster and a network of Sun workstations. The package runs efficiently in each of these environments. More detailed information about Parallel ARPACK is available in the report [8] by Maschhoff and Sorensen.

1.10 Contributed Additions

A collection of useful codes related to ARPACK is available in the directory.

```
pub/software/ARPACK/CONTRIBUTED
```

The codes that are available there have been contributed by ARPACK users. These include specialized drivers for unusual applications, alternate shift strategies, and in some cases minor modifications to the ARPACK source. These codes offer extended capabilities but are not part of the official release and are not supported as part of the ARPACK distribution. However, each of the included entries has undergone some degree of testing and there is a contact person listed with each code.
Users wishing to contribute software to this collection should send e-mail to `arpack@caam.rice.edu`.

1.11 Trouble Shooting and Problems

An up to date list of known problems is available in

```
pub/software/ARPACK/Known_Problems
```

Any difficulties with using the software should be reported to

```
arpack@caam.rice.edu
```

1.12 Research Funding of ARPACK

Financial support for this work was provided in part by the National Science Foundation cooperative agreement CCR-912008, and by the ARPA contract number DAAL03-91-C-0047 (administered by the U.S. Army Research Office). R.B. Lehoucq was also supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Chapter 2

Getting Started with ARPACK

This chapter will describe the basic structure of ARPACK and how to begin using it for computing a few eigenvalues and eigenvectors of a symmetric matrix.

Difficult problems and generalized problems will require one to use a shift-invert strategy that is based upon the use of a (sparse-direct) matrix factorization. These more sophisticated modes of operation are described in the next chapter. Example driver routines have been constructed for each problem type, computational mode, data type and precision. These drivers may be used as templates to construct a code for a specific application by substituting the appropriate data structures, matrix factorizations, solvers and matrix-vector products. An explanation of how to use these drivers as templates and how to modify them for your own use is given in Appendix A (Templates and Driver Routines). Each of the various drivers has been provided to address a typical situation arising from significant applications of eigenvalue calculations.

We begin with a description of the ARPACK directory structure. We then use one of the sample drivers to illustrate the use of ARPACK in the simplest mode of operation.

2.1 Directory Structure and Contents

Once the ARPACK software has been unbundled as described in Chapter 1, a directory structure will have been created. The top level directory is named ARPACK. The directory structure is pictured in Figure 2.1.

The `ARMAKES` subdirectory contains sample files with machine specific information needed during the building of the ARPACK library. The `BLAS` and `LAPACK` subdirectories contain the necessary codes from the respective software libraries. The `DOCUMENTS` subdirectory contains files that have example templates of how to invoke the different computational modes offered by ARPACK. Example driver programs that illustrate all of the computational modes, data types and precisions may be found in the `EXAMPLES` directory. Programs for banded, complex, non-symmetric, symmetric eigenvalue problems and singular value decomposition may be found in the directories `BAND`, `COMPLEX`, `NONSYM`, `SYM`, `SVD` respectively. Look at the `README` files in each subdirectory for further information. The `SRC` subdirectory contains all the ARPACK source codes. The `UTIL` subdirectory contains the various utility routines needed for printing results and timing the execution of the ARPACK subroutines.

The archived library `libarpack_<PLATFORM>.a` is created upon completion of the installation instructions. Here, `_<PLATFORM>` denotes the environment where the ARPACK library is built. All of the subroutines other than those in the `EXAMPLES` directory are compiled and archived into `libarpack_<PLATFORM>.a`. The installer should be aware that the BLAS and LAPACK directories contain a subset of routines from these packages that will require an additional megabyte of memory once they are compiled and archived. If these packages are already available on your system, you may delete the BLAS and LAPACK directories provided here and point to the ones that are already installed on your system. This is easily done by modifying the file `ARmake.inc` as described in the `README` file in the top level directory `ARPACK`.

To get started, we recommend that the user enter the `SIMPLE` subdirectory and issue the commands

```
make dssimp ; dssimp > output
```

This will compile, link, and execute the `dssimp` program. `dssimp` is a sample driver for the reverse communication interface to the ARPACK routine `dsaupd` which finds a few eigenvalues and eigenvectors of a symmetric matrix.

This chapter discusses the use of `dssimp` for computing eigenvalues and eigenvectors of a symmetric matrix using the simplest computational mode. There are additional drivers available for all of the computational modes, data types, and precisions. These additional driver programs are in the `EXAMPLES` subdirectories. Each of them is self-contained and may be compiled and executed in a similar manner as described in the `README` files.

This `dssimp` driver should serve as template to enable a user to create a program to use `dsaupd` on a specific problem in the simplest computational mode. All of the driver programs in the various `EXAMPLES` subdirectories are intended to be used in this way. The `simple` programs have more extensive documentation to aid in the understanding and conversion but essentially the same principle and structure apply to all of the driver programs.

2.2 Getting Started

The collection of driver programs mentioned above have been constructed to illustrate how to use ARPACK in a straightforward way to solve some of the most frequently occurring eigenvalue problems. The purpose of this section

Figure 2.1: The ARPACK directory structure.

```
                        ARPACK
         ----------------------------------------
        /      /       /       |   \       \    \
  ARMAKES  BLAS  DOCUMENTS  |  LAPACK  SRC  UTIL
                               |
                           EXAMPLES
                  ---------------------------------
                 /      /      |      \      \      \
              SIMPLE  BAND   SVD    SYM  NONSYM  COMPLEX
```

Table 2.1: List of the simple drivers illustrating the use of ARPACK.

NAME	PURPOSE
`[d,s]ssimp`	Real symmetric driver
`[d,s]nsimp`	Real non-symmetric driver
`[z,c]nsimp`	Complex (Hermitian or general)

and the corresponding *simple* codes is to provide a means to get started with ARPACK as quickly as possible. These codes may be used as templates that are easily altered to solve new problems after a few straightforward changes. The available simple drivers are listed in Table 2.1. These codes may be found in the `EXAMPLES/SIMPLE` directory.

2.3 An Example for a Symmetric Eigenvalue Problem

In this section, the simple code `dssimp` is discussed. All of the other example drivers are similar in nature. This particular example program illustrates the simplest computational mode of using ARPACK in considerable detail. `dssimp` shows how to use ARPACK to find a few eigenvalues λ and corresponding eigenvectors $\mathbf{x}$ for the standard eigenvalue problem:

$$\mathbf{A}\mathbf{x} = \mathbf{x}\lambda$$

where $\mathbf{A}$ is an n by n real symmetric matrix. The main points illustrated are:

- How to declare sufficient memory to find `nev` eigenvalues. `dssimp` is set up to find `nev` eigenvalues of largest magnitude `LM`. This may be reset to any one of the additional options (`SM, LA, SA, BE`) to find other eigenvalues of interest.

- Illustration of the reverse communication interface needed to utilize the top level ARPACK routine `dsaupd`. This routine computes the quantities needed to construct the desired eigenvalues and the corresponding eigenvectors.

- How to extract the desired eigenvalues and eigenvectors from the quantities computed with `dsaupd` by using the ARPACK routine `dseupd`.

This `dssimp` program is a driver for the subroutine `dsaupd` and it is set up to solve the following problem:

- Solve $\mathbf{Ax} = \mathbf{x}\lambda$ in regular mode. Regular mode only uses matrix vector products involving $\mathbf{A}$.

- The matrix $\mathbf{A}$ for this example is derived from the central difference discretization of the 2-dimensional Laplacian on the unit square with zero Dirichlet boundary conditions.

- The goal is to compute `nev` eigenvalues of largest magnitude and corresponding eigenvectors.

The only thing that must be supplied in order to use this routine on your problem is to change the array dimensions and to supply a means to compute the matrix-vector product

$$\mathbf{w} \leftarrow \mathbf{Av}$$

on request from `dsaupd`. The selection of which eigenvalues to compute may be altered by changing the parameter `which`.

Once usage of `dsaupd` in the simplest mode is understood, you may wish to explore the other available options such as solving generalized eigenvalue problems using a shift-invert computational mode. Some of these additional modes are described in the latter sections of this chapter and also in the file `ex-sym.doc` in `DOCUMENTS` directory.

2.3.1 The Reverse Communication Interface

The easiest way to describe the reverse communication interface is through the example code segment shown in Figure 2.2. Once storage has been declared and the input parameters initialized, the reverse communication loop (Fig. 2.2) is entered and repeated calls to `dsaupd` are made. On return the parameter `ido` will indicate the action to be taken. In this simple example, the only action taken is a matrix-vector product (see `call av` in the code segment of Figure 2.2). The more sophisticated shift-invert computational modes require more complicated actions but the basic idea remains the same.

Table 2.2: Parameters for the top level ARPACK routines.

PARAMETER	DESCRIPTION
`ido`	Reverse communication flag.
`nev`	The number of requested eigenvalues to compute.
`ncv`	The number of Lanczos basis vectors to use through the course of the computation.
`bmat`	Indicates whether the problem is standard `bmat = 'I'` or generalized (`bmat = 'G'`).
`which`	Specifies which eigenvalues of **A** are to be computed.
`tol`	Specifies the relative accuracy to which eigenvalues are to be computed.
`iparam`	Specifies the computational mode, number of IRAM iterations, the implicit shift strategy, and outputs various informational parameters upon completion of IRAM.

```
c
c     %-------------------------------------------------%
c     | M A I N   L O O P (Reverse communication loop) |
c     %-------------------------------------------------%
c
 10   continue
c
c        %---------------------------------------------%
c        | Repeatedly call the routine DSAUPD and take |
c        | actions indicated by parameter IDO until    |
c        | either convergence is indicated or maxitr   |
c        | has been exceeded.                          |
c        %---------------------------------------------%
c
         call dsaupd ( ido, bmat, n, which, nev, tol, resid,
     &                 ncv, v, ldv, iparam, ipntr, workd,
     &                 workl, lworkl, info )
c
         if (ido .eq. -1 .or. ido .eq. 1) then
c
c           %--------------------------------------%
c           | Perform matrix vector multiplication |
c           |              y <--- OP*x             |
c           | The user should supply his/her own   |
c           | matrix vector multiplication routine |
c           | here that takes workd(ipntr(1)) as   |
c           | the input, and return the result to  |
c           | workd(ipntr(2)).                     |
c           %--------------------------------------%
c
            call av (nx, workd(ipntr(1)), workd(ipntr(2)))
c
c           %-----------------------------------------%
c           | L O O P   B A C K to call DSAUPD again. |
c           %-----------------------------------------%
c
            go to 10
c
         end if
```

Figure 2.2: The reverse communication interface in example program `dssimp`.

2.3.2 Post Processing for Eigenvalues and Eigenvectors

If `dsaupd` indicates that convergence has taken place, then various steps may be taken to recover the results in a useful form. This is done through the subroutine `dseupd` and is illustrated in Figure 2.3. In the simple mode described in this chapter, the computed eigenvectors returned by `dseupd` are normalized to have unit length (in the 2-norm).

2.3.3 Setting up the problem

To set up the problem, the user needs to specify the number of eigenvalues to compute, which eigenvalues are of interest, the number of basis vectors to use, and whether or not the problem is standard or generalized. These items are controlled with the parameters listed in Table 2.2.

The simple codes described in this chapter are set up to solve the standard eigenvalue problem using only matrix-vector products $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$. Generalized eigenvalue problems require selection of another mode. These are addressed in Chapter 3. The value of `ncv` must be at least `nev` + 1. The options available for `which` include `'LA'` and `'SA'` for the algebraically largest and smallest eigenvalues, `'LM'` and `'SM'` for the eigenvalues of largest or smallest magnitude, and `'BE'` for the simultaneous computation of the eigenvalues at both ends of the spectrum. For a given problem, some of these options may converge more rapidly than others due to the approximation properties of the IRLM as well as the distribution of the eigenvalues of **A**. Convergence behavior can be quite different for various settings of the `which` parameter. For example, if the matrix is indefinite then setting `which` = `'SM'` will require interior eigenvalues to be computed and the Lanczos process may require many steps before these are resolved.

For a given `ncv`, the computational work required is proportional to $\mathtt{n} \cdot \mathtt{ncv}^2$ FLOPS. Setting `nev` and `ncv` for optimal performance is very much problem dependent. If possible, it is best to avoid setting `nev` in a way that will split clusters of eigenvalues. For example, if the the five smallest eigenvalues are positive and on the order of 10^{-4} and the sixth smallest eigenvalue is on the order of 10^{-1} then it is probably better to ask for `nev` = 5 than for `nev` = 3 even if the three smallest are the only ones of interest.

Setting the optimal value of `ncv` relative to `nev` is not completely understood. As with the choice of `which`, it depends upon the underlying approximation properties of the IRLM as well as the distribution of the eigenvalues of **A**. As a rule of thumb, $\mathtt{ncv} \geq 2 \cdot \mathtt{nev}$ is reasonable. There are tradeoffs due to the cost of the user supplied matrix-vector products and the cost of the implicit restart mechanism and the cost of maintaining the orthogonality of the Lanczos vectors. If the user supplied matrix-vector product is relatively cheap, then a smaller value of `ncv` may lead to more user matrix-vector products, but an overall decrease in computation time. Chapter 4 will discuss these issues in more detail.

```
c
c        %-------------------------------------------%
c        | No fatal errors occurred.                 |
c        | Post-Process using DSEUPD.                |
c        |                                           |
c        | Computed eigenvalues may be extracted.    |
c        |                                           |
c        | Eigenvectors may be also computed now if  |
c        | desired.  (indicated by rvec = .true.)    |
c        |                                           |
c        | The routine DSEUPD is called to do this   |
c        | post processing (Other modes may require  |
c        | more complicated post processing than     |
c        | mode1.)                                   |
c        |                                           |
c        %-------------------------------------------%
c
          rvec = .true.
c
          call dseupd ( rvec, 'All', select, d, v, ldv, sigma,
     &         bmat, n, which, nev, tol, resid, ncv, v, ldv,
     &         iparam, ipntr, workd, workl, lworkl, ierr )
c
c         %----------------------------------------------%
c         | Eigenvalues are returned in the first column |
c         | of the two dimensional array D and the       |
c         | corresponding eigenvectors are returned in   |
c         | the first NCONV (=IPARAM(5)) columns of the  |
c         | two dimensional array V if requested.        |
c         | Otherwise, an orthogonal basis for the       |
c         | invariant subspace corresponding to the      |
c         | eigenvalues in D is returned in V.           |
c         %----------------------------------------------%
c
```

Figure 2.3: Post Processing for Eigenvalues and Eigenvectors using `desupd`.

```
c
c     %--------------%
c     | Local Arrays |
c     %--------------%
c
      Double precision
     &                 v(ldv,maxncv), workl(maxncv*(maxncv+8)),
     &                 workd(3*maxn), d(maxncv,2), resid(maxn),
     &                 ax(maxn)
      logical          select(1)
      integer          iparam(11), ipntr(11)
```

Figure 2.4: Storage declarations needed for ARPACK subroutine `dsaupd`

2.3.4 Storage Declarations

The program is set up so that the setting of the three parameters `maxn, maxnev, maxncv` will automatically declare all of the work space needed to run `dsaupd` on a given problem.

The declarations allow a problem size of $\texttt{N} \leq \texttt{maxn}$, computation of $\texttt{nev} \leq \texttt{maxnev}$ eigenvalues, and using at most $\texttt{ncv} \leq \texttt{maxncv}$ Lanczos basis vectors during the IRLM. The user may override the default settings used for the example problem by modifying `maxn`, `maxnev` and `maxncv` in the following parameter statement.

```
integer          maxn, maxnev, maxncv, ldv
parameter       (maxn=256, maxnev=10, maxncv=25, ldv=maxn )
```

These parameters are used in the code segment listed in Figure 2.3.4 for declaring all of the output and work arrays needed by the ARPACK subroutines `dsaupd` and `dseupd`.

2.3.5 Stopping Criterion

The stopping criterion is determined by the user through specification of the parameter `tol`. The default value for `tol` is machine precision ϵ_M. There are several things to consider when setting this parameter. In absence of all other considerations, one should expect a computed eigenvalue λ_c to satisfy

$$|\lambda_c - \lambda_t| \leq \texttt{tol} \cdot |\lambda_c|,$$

where λ_t is the eigenvalue of $\mathbf{A}$ nearest to λ_c. Typically, decreasing the value of `tol` will increase the work required to satisfy the stopping criterion. However, setting `tol` too large may cause eigenvalues to be missed when they are multiple or very tightly clustered. Typically, a fairly small setting of `tol` and a reasonably large setting of `ncv` is required to avoid missing multiple eigenvalues. However, some care must be taken. It is possible to set `tol` so small that convergence never occurs. There may be additional complications when the matrix $\mathbf{A}$ is non-normal or when the eigenvalues of interest are clustered near the origin. A detailed discussion of the stopping rules and what they imply about the computed results is given in Chapter 4 § 4.6.

2.3.6 Initial Parameter Settings

The reverse communication flag is denoted by `ido`. This parameter must be initially set to `0` before the first call to `dsaupd`. During the course of the IRLM, `ido` is used to indicate the action to be taken by the user when control is returned to the program calling `dsaupd`.

Various algorithmic modes may be selected through the settings of the entries in the integer array `iparam`. The most important of these is the value of `iparam(7)` which specifies the computational mode to use. The selection in this simple example is `iparam(7) = 1` indicating `mode = 1` is to be used and this only requires matrix-vector products. Convergence can be greatly enhanced through the the use of the shift-invert computational modes provided. These additional modes are described in Chapter 3. In addition, `iparam(1)` specifies the shift selection strategy to be used with the implicit restarting mechanism described in Chapter 4. Setting `iparam(1) = 1` as in the example will specify the so called *exact shift* strategy. Exact shifts are recommended unless the user has a very good reason based upon *a-priori* information and an expert knowledge of the underlying IRLM to specify an alternative. The maximum number of IRLM iterations allowed must be specified in `iparam(3)`. In specifying this parameter, the user should keep in mind that an IRLM iteration costs approximately `ncv - nev` user supplied matrix-vector products. In addition, $4 \cdot \mathtt{n} \cdot \mathtt{ncv} \cdot (\mathtt{ncv} - \mathtt{nev})$ FLOPS are needed for the work associated with an IRLM iteration.

The integer argument `lworkl` sets the length of the work array `workl`. Its value is set at $\mathtt{ncv} \cdot (\mathtt{ncv} + 8)$.

2.3.7 Setting the Starting Vector

The parameter `info` should be set to `0` on the initial call to `dsaupd` unless the user wants to supply the starting vector that initializes the IRLM. Normally, this default is a reasonable choice. However, if this eigenvalue calculation is one of a sequence of closely related problems then convergence may be accelerated if a suitable starting vector is specified. Typical choices in this situation might be to use the final value of the starting vector from the previous eigenvalue calculation (that vector will already be in the first column of `V`) or to construct a starting vector by taking a linear combination of the computed eigenvectors from the previously converged eigenvalue calculation. If the starting vector is

```
include 'debug.h'
ndigit = -3
logfil = 6
msgets = 0
msaitr = 0
msapps = 0
msaupd = 1
msaup2 = 0
mseigt = 0
mseupd = 0
```

Figure 2.5: How to initiate the trace debugging capability in ARPACK.

to be supplied, then it should be placed in the array `resid` and `info` should be set to `1` on entry to `dsaupd`. On completion, the parameter `info` may contain the value `0` indicating the iteration was successful or it may contain a nonzero value indicating an error or a warning condition. The meaning of a nonzero value returned in `info` may be found in the header comments of the subroutine `dsaupd`.

2.3.8 Trace Debugging Capability

ARPACK provides a means to trace the progress of the computation as it proceeds. Various levels of output may be specified from no output (`level = 0`) to voluminous (`level = 3`) . The code segment listed in Figure 2.5 gives an example of the statements that may be used within the calling program to initiate and request this output.

The include statement sets up the storage declarations that are solely associated with this trace debugging feature. The parameter `ndigit` specifies the number of decimal digits and the width of the output lines. A positive or negative value of `ndigit` specifies that `132` or `80` columns, respectively, are used during output. The parameter `logfil` specifies the logical unit number of the output file. The values of the remaining parameters indicate the output levels from the indicated routines. For example, `msaitr` indicates the level of output requested from the subroutine `dsaitr`. The above configuration will give a breakdown of the number of matrix vector products required, the total number of iterations, the number of re-orthogonalization steps and an estimate of the time spent in each routine and phase of the computation. Figure 2.6 displays the output produced by the above settings. The user is encouraged to experiment with the other settings once some familiarity has been gained with the routines. See Appendix B for a detailed discussion of the Trace Debugging Capabilities.

```
==========================================
= Symmetric implicit Arnoldi update code =
= Version Number: 2.1                    =
= Version Date:   11/15/95               =
==========================================
= Summary of timing statistics           =
==========================================


Total number update iterations             =     8
Total number of OP*x operations            =   125
Total number of B*x operations             =     0
Total number of reorthogonalization steps  =   125
Total number of iterative refinement steps =     0
Total number of restart steps              =     0
Total time in user OP*x operation          =     0.020002
Total time in user B*x operation           =     0.000000
Total time in Arnoldi update routine       =     0.210021
Total time in ssaup2 routine               =     0.190019
Total time in basic Arnoldi iteration loop =     0.110011
Total time in reorthogonalization phase    =     0.070007
Total time in (re)start vector generation  =     0.000000
Total time in trid eigenvalue subproblem   =     0.040004
Total time in getting the shifts           =     0.000000
Total time in applying the shifts          =     0.040004
Total time in convergence testing          =     0.000000
```

Figure 2.6: Output from a Debug session for `dsaupd`.

Chapter 3

General Use of ARPACK

This chapter will describe the complete structure of the reverse communication interface to the ARPACK codes. Numerous computational modes are available, including several shift-invert strategies designed to accelerate convergence. Two of the more sophisticated modes will be described in detail. The remaining ones are quite similar in principle, but require slightly different tasks to be performed with the reverse communication interface.

This chapter is structured as follows. The naming conventions used in ARPACK, and the data types and precisions available are described in § 3.1. Spectral transformations are discussed in § 3.2. Spectral transformations are usually extremely effective but there are a number of problem dependent issues that determine which one to use. In § 3.3 we describe the reverse communication interface needed to exercise the various shift-invert options. Each shift-invert option is specified as a computational mode and all of these are summarized in the remaining sections. There is a subsection for each problem type and hence these sections are quite similar and repetitive. Once the basic idea is understood, it is probably best to turn directly to the subsection that describes the problem setting that is most interesting to you.

Perhaps the easiest way to rapidly become acquainted with the modes of ARPACK is to run the example driver routines (see Appendix A) that have been supplied for each of the modes. These may be used as templates and adapted to solve specific problems.

3.1 Naming Conventions, Precisions and Types

ARPACK has two interface routines that must be invoked by the user. They are `_aupd` that implements the IRAM and `_eupd` to post process the results of `_aupd`. The user may request an orthogonal basis for a selected invariant subspace or eigenvectors corresponding to selected eigenvalues with `_eupd`. If a spectral transformation is used, `_eupd` transforms the computed eigenvalues for the problem $\mathbf{Ax} = \mathbf{Mx}\lambda$.

Both `_aupd` and `_eupd` are available for several combinations of problem

Table 3.1: Available precisions and data types for ARPACK.

FIRST LETTER	PRECISION	DATA TYPE
`s`	Single	Real
`d`	Double	Real
`c`	Single	Complex
`z`	Double	Complex

type (symmetric and non-symmetric), data type (real, complex), and precision (single, double). The first letter `(s,d,c,z)` denotes precision and data type. The second letter denotes whether the problem is symmetric (`s`) or non-symmetric (`n`). Table 3.1 lists the possibilities.

Thus, `dnaupd` is the routine to use if the problem is a double precision non-symmetric (standard or generalized) problem and `dneupd` is the post-processing routine to use in conjunction with `dnaupd` to recover eigenvalues and eigenvectors of the original problem upon convergence. For complex matrices, one should use `_naupd` and `_neupd` with the first letter either `c` or `z` regardless of whether the problem is Hermitian or non-Hermitian. Table 3.2 lists the double precision routines available.

3.2 Shift and Invert Spectral Transformation Mode

The most general problem that may be solved with ARPACK is to compute a few selected eigenvalues and corresponding eigenvectors for

$$\mathbf{A}\mathbf{x} \;=\; \mathbf{M}\mathbf{x}\lambda \tag{3.2.1}$$

where $\mathbf{A}$ and $\mathbf{M}$ are real or complex $n \times n$ matrices.

The shift and invert spectral transformation is used to enhance convergence to a desired portion of the spectrum. If $(\mathbf{x}, \lambda)$ is an eigen-pair for $(\mathbf{A}, \mathbf{M})$ and $\sigma \neq \lambda$ then

$$(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}\mathbf{x} = \mathbf{x}\nu \quad \text{where} \quad \nu = \frac{1}{\lambda - \sigma}. \tag{3.2.2}$$

This transformation is effective for finding eigenvalues near σ since the `nev` eigenvalues ν_j of $\mathbf{C} \equiv (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ that are largest in magnitude correspond to the `nev` eigenvalues λ_j of the original problem that are nearest to the shift σ in absolute value. These transformed eigenvalues of largest magnitude are precisely the eigenvalues that are easy to compute with a Krylov method. Once they are found, they may be transformed back to eigenvalues of the original problem. The direct relation is

$$\lambda_j = \sigma + 1/\nu_j,$$

Table 3.2: Double Precision Top level routines in ARPACK subdirectory `SRC`.

ROUTINE	DESCRIPTION
`dsaupd`	Top level reverse communication interface to solve real double precision symmetric problems.
`dseupd`	Post processing routine used to compute eigenvectors associated with the computed eigenvalues. This requires output from a converged application of `dsaupd`.
`dnaupd`	Top level reverse communication interface to solve real double precision non-symmetric problems.
`dneupd`	Post processing routine used to compute eigenvectors and/or Schur vectors corresponding to the invariant subspace associated with the computed eigenvalues. This requires output from a converged application of `dnaupd`.
`znaupd`	Top level reverse communication interface to solve double precision complex arithmetic problems. This routine should be used for both Hermitian and Non-Hermitian problems.
`zneupd`	Post processing routine used to compute eigenvectors and/or Schur vectors corresponding to the invariant subspace associated with the computed eigenvalues in complex arithmetic. This requires output from a converged application of `znaupd`.

and the eigenvector $\mathbf{x}_j$ associated with ν_j in the transformed problem is also an (generalized) eigenvector of the original problem corresponding to λ_j. Usually, the IRAM will rapidly obtain good approximations to the eigenvalues of $\mathbf{C}$ of largest magnitude. However, to implement this transformation, one must provide a means to solve linear systems involving $\mathbf{A} - \sigma\mathbf{M}$ either with a matrix factorization or with an iterative method.

In general, $\mathbf{C}$ will be non-Hermitian even if $\mathbf{A}$ and $\mathbf{M}$ are both Hermitian. However, this is easily remedied. The assumption that $\mathbf{M}$ is Hermitian positive definite implies that the bi-linear form

$$< \mathbf{x}, \mathbf{y} > \equiv \mathbf{x}^H \mathbf{M} \mathbf{y}$$

is an inner product. If $\mathbf{M}$ is positive semi-definite and singular, then a semi-inner product results. We call this a weighted $\mathbf{M}$-inner product and vectors $\mathbf{x}, \mathbf{y}$ are called $\mathbf{M}$-orthogonal if $< \mathbf{x}, \mathbf{y} >= 0$. It is easy to show that if $\mathbf{A}$ is Hermitian (self-adjoint) then $\mathbf{C}$ is Hermitian (self-adjoint) with respect to this $\mathbf{M}$-inner product (meaning $< \mathbf{Cx}, \mathbf{y} >=< \mathbf{x}, \mathbf{Cy} >$ for all vectors $\mathbf{x}, \mathbf{y}$). Therefore, symmetry will be preserved if we force the computed basis vectors to be orthogonal in this $\mathbf{M}$-inner product. Implementing this $\mathbf{M}$-orthogonality requires the user to provide a matrix-vector product $\mathbf{w} \leftarrow \mathbf{Mv}$ on request along with each application of $\mathbf{C}$. In the following sections we shall discuss some of the more familiar transformations to the standard eigenproblem. However, when $\mathbf{M}$ is positive (semi) definite, we recommend using the shift-invert spectral transformation with $\mathbf{M}$-inner products if at all possible. This is a far more robust transformation when $\mathbf{M}$ is ill-conditioned or singular. With a little extra manipulation (provided automatically in `_eupd`) the (semi-) inner product induced by $\mathbf{M}$ prevents corruption of the computed basis vectors by roundoff-error associated with the presence of *infinite* eigenvalues. These very ill-conditioned eigenvalues are generally associated with a singular or highly ill-conditioned $\mathbf{M}$. A detailed discussion of this theory may be found in Chapter 4.

Shift-invert spectral transformations are very effective and should even be used on standard problems ($\mathbf{M} = \mathbf{I}$) whenever possible. This is particularly true when interior eigenvalues are sought or when the desired eigenvalues are clustered. Roughly speaking, a set of eigenvalues is clustered if the maximum distance between any two eigenvalues in that set is much smaller than the maximum distance between any two eigenvalues of $(\mathbf{A}, \mathbf{M})$.

If one has a generalized problem ($\mathbf{M} \neq \mathbf{I}$), then one must provide a way to solve linear systems with either $\mathbf{A}$, $\mathbf{M}$ or a linear combination of the two matrices in order to use ARPACK. In this case, a sparse direct method should be used to factor the appropriate matrix whenever possible. The resulting factorization may be used repeatedly to solve the required linear systems once it has been obtained. If an iterative method is used for the linear system solves, the accuracy of the solutions must be commensurate with the convergence tolerance used for ARPACK. A slightly more stringent tolerance is needed for the iterative linear system solves (relative to the desired accuracy of the eigenvalue calculation). See [4, 10, 9, 12] for further information and references.

The main drawback with using the shift-invert spectral transformation is that the coefficient matrix $\mathbf{A} - \sigma\mathbf{M}$ is typically indefinite in the Hermitian case and has 0 in the interior of the convex hull of the spectrum in the non-Hermitian case. These are typically the most difficult situations for iterative methods and also for sparse direct methods.

The decision to use a spectral transformation on a standard eigenvalue problem ($\mathbf{M} = \mathbf{I}$) or to use one of the simple modes described in Chapter 2 is problem dependent. The simple modes have the advantage that one only need supply a matrix vector product $\mathbf{w} \leftarrow \mathbf{Av}$. However, this approach is usually only successful for problems where extremal non-clustered eigenvalues are sought. In non-Hermitian problems, extremal means eigenvalues near the boundary of the convex hull of the spectrum of $\mathbf{A}$. For Hermitian problems, extremal means eigenvalues at the left or right end points of the spectrum of $\mathbf{A}$. The notion of non-clustered (or well separated) is difficult to define without going into considerable detail. A simplistic notion of a *well-separated* eigenvalue λ_i for a Hermitian problem would be $|\lambda_i - \lambda_j| > \tau|\lambda_n - \lambda_1|$ for all $j \neq i$ with $\tau >> \epsilon_M$. Unless a matrix vector product is quite difficult to code or extremely expensive computationally, it is probably worth trying to use the simple mode first if you are seeking extremal eigenvalues.

The remainder of this section discusses additional transformations that may be applied to convert a generalized eigenproblem to a standard eigenproblem. These are appropriate when $\mathbf{M}$ is well conditioned (Hermitian or non-Hermitian).

3.2.1 M is Hermitian Positive Definite

If $\mathbf{M}$ is Hermitian positive definite and well conditioned ($\|\mathbf{M}\| \cdot \|\mathbf{M}^{-1}\|$ is of modest size), then computing the Cholesky factorization $\mathbf{M} = \mathbf{LL}^H$ and converting equation (3.2.1) to

$$(\mathbf{L}^{-1}\mathbf{AL}^{-H})\mathbf{y} = \mathbf{y}\lambda \quad \text{where} \quad \mathbf{L}^H\mathbf{x} = \mathbf{y}$$

provides a transformation to a standard eigenvalue problem. In this case, a request for a matrix vector product would be satisfied with the following three steps:

1. Solve $\mathbf{L}^H\mathbf{z} = \mathbf{v}$ for $\mathbf{z}$,

2. Matrix-vector multiply $\mathbf{z} \leftarrow \mathbf{Az}$,

3. Solve $\mathbf{Lw} = \mathbf{z}$ for $\mathbf{w}$.

Upon convergence, a computed eigenvector $\mathbf{y}$ for $(\mathbf{L}^{-1}\mathbf{AL}^{-H})$ is converted to an eigenvector $\mathbf{x}$ of the original problem by solving the the triangular system $\mathbf{L}^H\mathbf{x} = \mathbf{y}$. This transformation is most appropriate when $\mathbf{A}$ is Hermitian, $\mathbf{M}$

is Hermitian positive definite and extremal eigenvalues are sought. This is because $\mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-H}$ will be Hermitian when $\mathbf{A}$ is.

If $\mathbf{A}$ is Hermitian positive definite and the smallest eigenvalues are sought, then it would be best to reverse the roles of $\mathbf{A}$ and $\mathbf{M}$ in the above description and ask for the largest algebraic eigenvalues or those of largest magnitude. Upon convergence, a computed eigenvalue $\hat{\lambda}$ would then be converted to an eigenvalue of the original problem by the relation $\lambda \leftarrow 1/\hat{\lambda}$.

3.2.2 M is NOT Hermitian Positive Semi–Definite

If neither $\mathbf{A}$ nor $\mathbf{M}$ is Hermitian positive semi-definite, then a direct transformation to standard form is required. One simple way to obtain a direct transformation of equation (3.2.1) to a standard eigenvalue problem $\mathbf{Cx} = \mathbf{x}\lambda$ is to multiply on the left by $\mathbf{M}^{-1}$ which results in $\mathbf{C} = \mathbf{M}^{-1}\mathbf{A}$. Of course, one should not perform this transformation explicitly since it will most likely convert a sparse problem into a dense one. If possible, one should obtain a direct factorization of $\mathbf{M}$ and when a matrix-vector product involving $\mathbf{C}$ is called for, it may be accomplished with the following two steps:

1. Matrix-vector multiply $\mathbf{z} \leftarrow \mathbf{Av}$,

2. Solve: $\mathbf{Mw} = \mathbf{z}$.

Several problem dependent issues may modify this strategy. If $\mathbf{M}$ is singular or if one is interested in eigenvalues near a point σ then a user may choose to work with $\mathbf{C} \equiv (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ but without using the $\mathbf{M}$-inner products discussed previously. In this case the user will have to transform the converged eigenvalues ν_j of $\mathbf{C}$ to eigenvalues λ_j of the original problem.

3.3 Reverse Communication Structure for Shift-Invert

The reverse communication interface routine for all problem types is `__aupd`. If the eigenvalue problem (3.2.1) is a double precision non-symmetric one, then the subroutine to use is `dnaupd`. First the reverse communication loop structure will be described and then the details and nuances of the problem set up will be discussed. We shall use the symbol `OP` for the operator that is applied to vectors in the Arnoldi/Lanczos process and $\mathbf{B}$ will stand for the matrix to use in the weighted inner product described previously. For the shift-invert spectral transformation mode, `OP` denotes $(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ and $\mathbf{B}$ denotes $\mathbf{M}$. They will stand for different matrices in each of the various modes.

The basic idea is to set up a loop that repeatedly calls `__aupd`. On each return, one must either apply `OP` or $\mathbf{B}$ to a specified vector or exit the loop depending upon the value returned in the reverse communication parameter `ido`.

```
c     %--------------------------------------------------%
c     | Call a routine FAC to factor the matrix (A-sigma*M) |
c     |   into L*U.                                      |
c     |                                                  |
c     | A routine MV is called repeatedly below to       |
c     |   form z = Mv.                                   |
c     |                                                  |
c     | A routine SOLVE is used repeatedly below to solve |
c     |   (A-sigma*M) w = z  using the single LU         |
c     |   factorization provided by FAC.                 |
c     %--------------------------------------------------%
c
      call fac( (A-sigma*M), L, U)
c
c     %------------------------------------------%
c     | M A I N   L O O P (Reverse communication) |
c     %------------------------------------------%
c
 10   continue
c
c        %---------------------------------------------%
c        | Repeatedly call the routine DNAUPD and take |
c        | actions indicated by parameter IDO .        |
c        %---------------------------------------------%
c
         call dnaupd ( ido, bmat, n, which, nev, tol, resid,
     &                       ncv, v, ldv, iparam, ipntr, workd,
     &                       workl, lworkl, info )

         if (ido .eq. -1) then
c
c           %-------------------------------------------%
c           | Perform  y <--- OP*x = inv[A-sigma*M]*M*x  |
c           | to force the starting vector into the     |
c           | range of OP.                              |
c           |               x = workd(ipntr(1))         |
c           |               y = workd(ipntr(2))         |
c           %-------------------------------------------%
c
            call mv (workd(ipntr(1)), workd(ipntr(2)))
c
```

Figure 3.1: Reverse communication interface for Shift-Invert.

```
            call solve(L,U, workd(ipntr(2)))
c
c           %---  L O O P   B A C K to call DSAUPD again. ---%
c
            go to 10
c
         else if (ido .eq. 1) then
c
c           %---------------------------------------%
c           | Perform y <-- OP*x = inv[A-sigma*M]*M*x |
c           | M*x has been saved in workd(ipntr(3)).  |
c           |      M*x = workd(ipntr(3)               |
c           |        y = workd(ipntr(2)).             |
c           %---------------------------------------%
c
            call dcopy (n, workd(ipntr(3)), 1,
     &                                    workd(ipntr(2)), 1)
            call solve(L,U, workd(ipntr(2)))
c
c           %---  L O O P   B A C K to call DSAUPD again. ---%
c
            go to 10
c
        else if (ido .eq. 2) then
c
c           %---------------------------------------%
c           |  Perform  y <--- M*x                  |
c           |              x = workd(ipntr(1))      |
c           |              y = workd(ipntr(2))      |
c           %---------------------------------------%
c
            call mv (workd(ipntr(1)), workd(ipntr(2)))
c
c           %---  L O O P   B A C K to call DSAUPD again. ---%
c
            go to 10
        end if
```

Figure 3.2: Reverse communication interface for Shift-Invert contd.

3.3.1 Shift and invert on a Generalized Eigen-problem

The above code segments shown in Figures 3.1–3.2 illustrate the reverse communication loop for `dnaupd` in shift-invert mode for a generalized non-symmetric eigenvalue problem. This loop structure will be identical for the symmetric problem. The only change needed is to replace `dnaupd` with `dsaupd`. The loop structure is also identical for the complex arithmetic subroutine `znaupd`.

In the example shown in Figures 3.1–3.2, the matrix $\mathbf{M}$ is assumed to be symmetric and positive semi-definite. In the structure above, the user will have to supply some routine such as `fac` to obtain a matrix factorization of $\mathbf{A} - \sigma\mathbf{M}$ that may repeatedly be used to solve linear systems. Moreover, a routine needs to be provided in place of `mv` to perform the matrix-vector product $\mathbf{z} = \mathbf{Mv}$ and a routine in place of `solve` is required to solve linear systems of the form $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{z}$ as needed using the previously computed factorization.

When convergence has taken place (indicated by `ido = 99`), the reverse communication loop will be exited. Then, post-processing using the ARPACK subroutine `dneupd` must be done to recover the eigenvalues and corresponding eigenvectors of the original problem. When operating in Shift-invert mode, the eigenvalue selection parameter `which` is normally set to `which = 'LM'`. The routine `dneupd` is then used to convert the converged eigenvalues of `OP` to eigenvalues of the original problem (3.2.1). Also, when $\mathbf{M}$ is singular or ill-conditioned, the routine `dneupd` takes steps to *purify* the eigenvectors and rid them of numerical corruption from eigenvectors corresponding to near-infinite eigenvalues. These procedures are done automatically by the routine `dneupd` when operating in any one of the computational modes described above and later in this chapter.

The user may wish to construct alternative computational modes using spectral transformations that are not addressed by any of the modes specified in this chapter. The reverse communication interface will easily accommodate these modifications. However, it will most likely be necessary to construct explicit transformations of the eigenvalues of `OP` to eigenvalues of the original problem in these situations.

3.4 Using the Computational Modes

The problem set up is similar for all of the available computational modes. In the previous section, a detailed description of the reverse communication loop for a specific mode (Shift-Invert for a Real Non-symmetric Generalized Problem) was given. To use this or any of the other modes listed below, the user is strongly urged to modify one of the driver routine *templates* as described in Appendix A.

The first thing to decide is whether the problem will require a spectral transformation. If the problem is generalized ($\mathbf{M} \neq \mathbf{I}$) then a spectral transformation will be required (see § 3.2). Such a transformation will most likely

```
  call dnaupd (ido, bmat, n, which, nev, tol, resid, ncv, v,
 &             ldv, iparam, ipntr, workd, workl, lworkl, info)
```

Figure 3.3: Calling the ARPACK subroutine `dnaupd`.

be needed for a standard problem if the desired eigenvalues are in the interior of the spectrum or if they are clustered at the desired part of the spectrum. Once this decision has been made and `OP` has been specified, an efficient means to implement the action of the operator `OP` on a vector must be devised. The expense of applying `OP` to a vector will of course have direct impact on performance.

Shift-Invert spectral transformations may be implemented with or without the use of a weighted **M**-inner product. The relation between the eigenvalues of `OP` and the eigenvalues of the original problem must also be understood in order to make the appropriate specification of `which` in order to recover eigenvalues of interest for the original problem. The user must specify the number of eigenvalues to compute, which eigenvalues are of interest, the number of basis vectors to use, and whether or not the problem is standard or generalized. These items are controlled with the parameters listed in Table 2.2 of Chapter 2.

Setting `nev` and `ncv` for optimal performance is very much problem dependent. If possible, it is best to avoid setting `nev` in a way that will split clusters of eigenvalues. As a rule of thumb, $\texttt{ncv} \geq 2 \cdot \texttt{nev}$ is reasonable. There are tradeoffs due to the cost of the user supplied matrix-vector products and the cost of the implicit restart mechanism. If the user supplied matrix-vector product is relatively cheap, then a smaller value of `ncv` may lead to more user matrix-vector products and IRA iterations but an overall decrease in computation time. Convergence behavior can be quite different for various settings of the `which` parameter. The Arnoldi process tends to converge most rapidly to extreme points of the convex hull of the spectrum. Implicit restarting can be effective in focusing on and isolating a selected set of eigenvalues near these extremes. In principle, implicit restarting could isolate eigenvalues in the interior, but in practice this is difficult and usually unsuccessful. If one is interested in eigenvalues near a point σ that is in the interior of the convex hull of the spectrum, a shift-invert strategy is usually required for reasonable convergence.

The call to `dnaupd` is listed in Figure 3.3. The integer `ido` is the reverse communication flag that will specify a requested action on return from `dnaupd`. The `character*1` parameter `bmat` specifies if this is a standard `bmat = 'I'` or a generalized `bmat = 'G'` problem. The integer `n` specifies the dimension of the problem. The `character*2` parameter `which` specifies the `nev` eigenvalues to be computed. The options for `which` differ depending on whether a Hermitian or non-Hermitian eigenvalue problem is to be solved. Tables 3.3–3.4 list

```
   call dsaupd (ido, bmat, n, which, nev, tol, resid, ncv, v,
 &             ldv, iparam, ipntr, workd, workl, lworkl, info)
```

Figure 3.4: Calling the ARPACK subroutine `dsaupd`.

the different selections possible. The specification of problem type will be described separately but the reverse communication interface and loop structure is the same for each type of the three basic modes `regular`, `regular-inverse`, `shift-invert` (standard or generalized). There are some additional specialized modes for symmetric problems (`Buckling` and `Cayley`) and for real nonsymmetric problems with complex shifts applied in real arithmetic. The user is encouraged to examine the sample driver routines for these modes.

The integer `nev` indicates the number of eigenvalues to compute and `tol` specifies the accuracy requested. The integer array `iparam` has eleven entries. On input, `iparam(1)` should be set to `0` if the user wishes to supply shifts for implicit restarting or to `1` if the default exact-shift strategy is requested. The entry `iparam(1)` should be set to `1` unless the user has a great deal of knowledge about the spectrum and about the IRAM and underlying theory. The entry `iparam(3)` should be set to the maximum number of implicit restarts allowed. The cost of an implicit restart step (major iteration) is on the order of $4\mathtt{n}\cdot(\mathtt{ncv}-\mathtt{nev})$ Flops for the dense matrix operations and `ncv - nev` matrix-vector products $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$ with the matrix $\mathbf{A}$. On output, `iparam(3)` will contain the number of implicit restarts taken during the computation.

With respect to shift-invert modes, entry `iparam(7)` is very important. The remaining entries of `iparam` are either no longer referenced or are output parameters. The legitimate values for `iparam(7)` differ with each problem type and will be listed below for each of them.

3.5 Computational Modes for Real Symmetric Problems

The reverse communication interface subroutine for symmetric eigenvalue problems is `dsaupd`. The subroutine is called as shown in Figure 3.4. The argument `which` may be any one of the settings listed in Table 3.3.

The following is a list of the spectral transformation options for symmetric eigenvalue problems. In the following list, the specification of `OP` and $\mathbf{B}$ are given for the various modes. Also, the `iparam(7)` and `bmat` settings are listed along with the name of the sample driver for the given mode. Sample drivers for the following modes may be found in the `EXAMPLES/SYM` subdirectory.

1. Regular mode (`iparam(7) = 1, bmat = 'I'`). Use driver `dsdrv1`.

   (a) Solve $\mathbf{A}\mathbf{x} = \mathbf{x}\lambda$ in regular mode.

Table 3.3: The various settings for the argument `which` in `_saupd`

WHICH	DESCRIPTION
`'LA'`	Largest algebraic eigenvalues.
`'SA'`	Smallest algebraic eigenvalues.
`'LM'`	Eigenvalues largest in magnitude.
`'SM'`	Eigenvalues smallest in magnitude.
`'BE'`	Compute `nev` eigenvalues, half from each end of the spectrum. When `nev` is odd, compute one more from the high end than from the low end.

(b) `OP` $= \mathbf{A}$ and $\mathbf{B} = \mathbf{I}$.

2. Shift-invert mode (`iparam(7) = 3, bmat = 'I'`). Use driver `dsdrv2`.

   (a) Solve $\mathbf{Ax} = \mathbf{x}\lambda$ in shift-invert mode.

   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{I})^{-1}$ and $\mathbf{B} = \mathbf{I}$.

3. Regular inverse mode (`iparam(7) = 2, bmat = 'G'`). Use driver `dsdrv3`.

   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in regular inverse mode.

   (b) `OP` $= \mathbf{M}^{-1}\mathbf{A}$ and $\mathbf{B} = \mathbf{M}$.

4. Shift-invert mode (`iparam(7) = 3, bmat = 'G'`). Use driver `dsdrv4`.

   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in shift-invert mode.

   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ and $\mathbf{B} = \mathbf{M}$.

5. Buckling mode (`iparam(7) = 4, bmat = 'G'`). Use driver `dsdrv5`.

   (a) Solve $\mathbf{Kx} = \mathbf{K_G}\mathbf{x}\lambda$ in Buckling mode.

   (b) `OP` $= (\mathbf{K} - \sigma\mathbf{K_G})^{-1}\mathbf{K}$ and $\mathbf{B} = \mathbf{K}$.

6. Cayley mode (`iparam(7) = 5, bmat = 'G'`). Use driver `dsdrv6`.

   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in Cayley mode.

   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}(\mathbf{A} + \sigma\mathbf{M})$ and $\mathbf{B} = \mathbf{M}$.

```
c
c        %------------------------------------------%
c        | No fatal errors occurred.                |
c        | Post-Process using DSEUPD.               |
c        |                                          |
c        | Computed eigenvalues may be extracted.   |
c        |                                          |
c        | Eigenvectors may also be computed now if |
c        | desired.  (indicated by rvec = .true.)   |
c        %------------------------------------------%
c
         rvec = .true.
c
         call dseupd ( rvec, 'All', select, d, v, ldv, sigma,
     &        bmat, n, which, nev, tol, resid, ncv, v, ldv,
     &        iparam, ipntr, workd, workl, lworkl, ierr )
c
```

Figure 3.5: Post-Processing for Eigenvectors Using `dseupd`.

3.6 Post-Processing for Eigenvectors Using `dseupd`

On the final return from `dsaupd` (indicated by `ido = 99`), the error flag `info` must be checked. If `info = 0` then no fatal errors have occurred and it is time to post-process using `dseupd` to get eigenvalues of the original problem and the corresponding eigenvectors if desired. In the case shown here (shift-invert and generalized), there are some subtleties to recovering eigenvectors when $\mathbf{M}$ is ill-conditioned. This process is called eigenvector purification. It prevents eigenvectors from being corrupted with noise due to the presence of eigenvectors corresponding to near infinite eigenvalues (See Chapter 4). These operations are completely transparent to the user. The general calling sequence for `dseupd` is shown in Figure 3.5.

The input parameters `bmat, n,` $\cdots$, `info` are precisely the same parameters that appear in the calling sequence of `dsaupd`. It is extremely IMPORTANT that none of these parameters are altered between the final return from `dsaupd` and the subsequent call to `dseupd`.

There is negligible additional cost to obtain eigenvectors. An orthonormal (Lanczos) basis is always computed. In the above example, this basis is overwritten with the eigenvectors in the array `v` . Both basis sets may be obtained if desired but there is an additional storage cost of `n` $\cdot$ `nev` if both are requested (in this case a separate `n` by `nev` array `z` must be supplied).

The approximate eigenvalues of the original problem are returned in ascending algebraic order in the array `d`. If it is desirable to retain the Lanczos

```
  call dnaupd (ido, bmat, n, which, nev, tol, resid, ncv, v,
&             ldv, iparam, ipntr, workd, workl, lworkl, info)
```

Figure 3.6: Calling sequence of subroutine `dnaupd`.

basis in `v` and storage is an issue, the user may elect to call this routine once for each desired eigenvector and store it peripherally. There is also the option of computing a selected set of these vectors with a single call.

The input parameters that must be specified are

- The logical variable `rvec` = `.true.` if eigenvectors are requested `.false.` if only eigenvalues are desired.
- The `character*1` parameter `howmny` that specifies how many eigenvectors are desired. `howmny = 'A'`: compute `nev` eigenvectors; `howmny = 'S'`: compute some of the eigenvectors, specified by the logical array `select`.
- `sigma` should contain the value of the shift used if `iparam(7) = 3,4,5`. It is not referenced if `iparam(7) = 1 or 2`.

When requested, the eigenvectors returned by `dseupd` are normalized to have unit length with respect to the **B** semi-inner product that was used. Thus, if $\mathbf{B} = \mathbf{I}$ they will have unit length in the standard 2-norm. In general, a computed eigenvector $\mathbf{x}$ will satisfy $1 = \mathbf{x}^T B\mathbf{x}$.

3.7 Computational Modes for Real Non-Symmetric Problems

The following subroutines are used to solve non-symmetric generalized eigenvalue problems in real arithmetic. These routines are appropriate when **A** is a general non-symmetric matrix and **M** is symmetric and positive semi-definite. The reverse communication interface routine for the non-symmetric double precision eigenvalue problem is `dnaupd`. The routine is called as shown in Figure 3.6. The specification of which `nev` eigenvalues is controlled by the `character*2` argument `which`. Table 3.4 lists the choices available.

There are three different shift-invert modes for non-symmetric eigenvalue problems. These modes are specified by setting the parameter entry `iparam(7) = mode` where `mode = 1,2,3, or 4`.

In the following list, the specification of `OP` and **B** are given for the various modes. Also, the `iparam(7)` and `bmat` settings are listed along with the name of the sample driver for the given mode. Sample drivers for the following modes may be found in the `EXAMPLES/NONSYM` subdirectory.

1. Regular mode (`iparam(7) = 1, bmat = 'I'`). Use driver `dndrv1`.

Table 3.4: The various settings for the argument `which` in `_naupd`

WHICH	DESCRIPTION
`'LM'`	Eigenvalues of largest magnitude.
`'SM'`	Eigenvalues of smallest magnitude.
`'LR'`	Eigenvalues of largest real part.
`'SR'`	Eigenvalues of smallest real part.
`'LI'`	Eigenvalues of largest imaginary part.
`'SI'`	Eigenvalues of smallest imaginary part.

   (a) Solve $\mathbf{Ax} = \mathbf{x}\lambda$ in regular mode.
   (b) `OP` $= \mathbf{A}$ and $\mathbf{B} = \mathbf{I}$.

2. Shift-invert mode (`iparam(7) = 3, bmat = 'I'`). Use driver `dndrv2` with `sigma` a real shift.
   (a) Solve $\mathbf{Ax} = \mathbf{x}\lambda$ in shift-invert mode.
   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{I})^{-1}$ and $\mathbf{B} = \mathbf{I}$.

3. Regular inverse mode (`iparam(7) = 2, bmat = 'G'`). Use driver `dndrv3`.
   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in regular inverse mode.
   (b) `OP` $= \mathbf{M}^{-1}\mathbf{A}$ and $\mathbf{B} = \mathbf{M}$.

4. Shift-invert mode (`iparam(7) = 3, bmat = 'G'`). Use driver `dndrv4` with `sigma` a real shift.
   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in shift-invert mode.
   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ and $\mathbf{B} = \mathbf{M}$.

5. Complex Shift-invert mode (`iparam(7) = 3, bmat = 'G'`). Use driver `dndrv5` when `sigma` is complex. $\mathbf{A} - \sigma\mathbf{M}$ must be factored in complex arithmetic.
   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ using complex shift in real arithmetic.
   (b) `OP` $= \mathrm{Real}\{(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}\}$ and $\mathbf{B} = \mathbf{M}$.

6. Complex Shift-invert mode (`iparam(7) = 4, bmat = 'G'`). Use driver `dndrv6` when `sigma` is complex. $\mathbf{A} - \sigma\mathbf{M}$ must be factored in complex arithmetic.
   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ using complex shift in real arithmetic.
   (b) `OP` $= \mathrm{Imag}\{(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}\}$ and $\mathbf{B} = \mathbf{M}$.

Note that there are two shift-invert modes with complex shifts (See `dndrv5` and `dndrv6`). Since σ is complex, these both require the factorization of the matrix $\mathbf{A} - \sigma\mathbf{M}$ in complex arithmetic even though both $\mathbf{A}$ and $\mathbf{M}$ are real. The only advantage of using this option instead of using the standard shift-invert mode in complex arithmetic with the routine `znaupd` is that all of the internal operations in the IRAM are executed in real arithmetic. This results in a factor of two savings in storage and a factor of four savings in arithmetic. There is additional post-processing that is somewhat more complicated than the other modes in order to get the eigenvalues and eigenvectors of the original problem. These modes are only recommended if storage is extremely critical.

3.8 Post-Processing for Eigenvectors Using dneupd

On the final return from `dnaupd` (indicated by `ido = 99`), the error flag `info` must be checked. If `info = 0`, then no fatal errors have occurred and it is time to post-process using `dneupd` to get eigenvalues of the original problem and the corresponding eigenvectors if desired. In the case shown here (shift-invert and generalized), there are some subtleties to recovering eigenvectors when $\mathbf{M}$ is ill-conditioned. This process is called eigenvector purification. It prevents eigenvectors from being corrupted with noise due to the presence of eigenvectors corresponding to near infinite eigenvalues (See Chapter 4). These operations are completely transparent to the user. The general calling sequence for `dseupd` is shown in Figure 3.7.

The input parameters `bmat, n` $\cdots$ `info` are precisely the same parameters that appear in the calling sequence of `dnaupd`. It is extremely IMPORTANT that none of these parameters are altered between the final return from `dsaupd` and the subsequent call to `dneupd`.

The approximate eigenvalues of the original problem are returned with real part array `dr` and imaginary part in the array `di`. Since the problem is real, complex eigenvalues must come in complex conjugate pairs. There is negligible additional cost to obtain eigenvectors. An orthonormal (Schur) basis for the invariant subspace corresponding to the converged approximate eigenvalues is always computed. In the above example, this basis is overwritten with the eigenvectors in the array `v`. When the eigenvectors corresponding to a complex conjugate pair of eigenvaues are computed, the vector corresponding to the eigenvalue with positive imaginary part is stored with real and imaginary parts in consecutive columns of `v`. The eigenvector corresponding to the conjugate eigenvalue is, of course, the conjugate of this vector. Both basis sets may be obtained if desired but there is an additional storage cost of `n`$\cdot$`nev` if both are requested (in this case a separate `n` by `nev` array `z` must be supplied). In some cases it may be desirable to have both basis sets.

In the non-Hermitian case, the eigenvector basis is potentially ill-conditioned and may not even exist. While, eigenvectors may have physical meaning, they are generally not the best basis to use. If a basis for a selected invariant sub-

```
c
c       %-----------------------------------------------%
c       | No fatal errors occurred.                     |
c       | Post-Process using DNEUPD.                    |
c       |                                               |
c       | Computed eigenvalues may be extracted.        |
c       |                                               |
c       | Eigenvectors may also be computed now if      |
c       | desired.  (indicated by rvec = .true.)        |
c       |                                               |
c       | The real part of the eigenvalue is returned   |
c       | in the first column of the two dimensional    |
c       | array D, and the IMAGINARY part is returned   |
c       | in the second column of D.  The corresponding |
c       | eigenvectors are returned in the first NEV    |
c       | columns of the two dimensional array V if     |
c       | requested.  Otherwise, an orthogonal basis    |
c       | for the invariant subspace corresponding to   |
c       | the eigenvalues in D is returned in V.        |
c       %-----------------------------------------------%
c
        rvec = .true.
        call dneupd ( rvec, 'A', select, d, d(1,2), v, ldv,
     &       sigmar, sigmai, workev, bmat, n, which, nev, tol,
     &       resid, ncv, v, ldv, iparam, ipntr, workd,
     &       workl, lworkl, ierr )
c
```

Figure 3.7: Post-Processing for Eigenvectors Using dneupd.

space is required, then it is generally better to compute a Schur basis. This will provide an orthogonal, hence well conditioned, basis for the subspace. The sensitivity of a given subspace to perturbations (such as roundoff error) is another question. See § 4.6 for a brief discussion.

If it is desirable to retain the Schur basis in `v` and storage is an issue, the user may elect to call this routine once for each desired eigenvector and store it peripherally. There is also the option of computing a selected set of these vectors with a single call.

The input parameters that must be specified are

- The logical variable `rvec` = `.true.` if eigenvectors are requested `.false.` if only eigenvalues are desired.
- The `character*1` parameter `howmny` specifies how many eigenvectors are desired. `howmny = 'A'`: compute `nev` eigenvectors; `howmny = 'P'`: Compute `nev` Schur vectors; `howmny = 'S'`: compute some of the eigenvectors, specified by the logical array `select`.
- `sigmar, sigmai` should contain the real and imaginary portions, respectively, of the shift that was used if `iparam(7) = 3 or 4`. Neither is referenced if `iparam(7) = 1 or 2`.

When requested, the eigenvectors returned by `dneupd` are normalized to have unit length with respect to the $\mathbf{B}$ semi-inner product that was used. Thus, if $\mathbf{B} = \mathbf{I}$ they will have unit length in the standard 2-norm. In general, a computed eigenvector $\mathbf{x}$ will satisfy $1 = \mathbf{x}^H \mathbf{B} \mathbf{x}$. Eigenvectors $\mathbf{x}_\pm = \mathbf{x}_R \pm i\mathbf{x}_I$ corresponding to a complex conjugate pair of eigenvalues $\lambda_\pm = \xi \pm i\eta$ with $\eta > 0$ are returned with $\mathbf{x}_R$ stored in the $j - th$ column and $\mathbf{x}_I$ stored in the $(j+1) - st$ column of the eigenvector matrix when λ_+ and λ_- are the $j - th$ and $(j+1) - st$ eigenvalues. This is the same storage convention as the one used for LAPACK. Note that $1 = \mathbf{x}^H \mathbf{B} \mathbf{x}$ implies $\mathbf{x}_R^T \mathbf{B} \mathbf{x}_R + \mathbf{x}_I^T \mathbf{B} \mathbf{x}_I = 1$.

3.9 Computational Modes for Complex Problems

This section describes the solution of eigenvalue problems in complex arithmetic. The reverse communication interface subroutine for the double precision complex eigenvalue problem is `znaupd`. This routine is to be used for both Hermitian and non-Hermitian problems. The routine is called as shown in Figure 3.8. It should be noted that the calling sequences for `znaupd` and `zneupd` differ slightly from those of `dnaupd` and `dneupd`. The main difference is that an additional work array `rwork` is required by `znaupd` that is not required by `dnaupd`.

Occasionally, when using `znaupd` on a complex Hermitian problem, eigenvalues will be returned with small but non-zero imaginary part due to unavoidable round-off errors. These should be ignored unless they are significant with respect to the eigenvalues of largest magnitude that have been computed.

```
  call znaupd ( ido, bmat, n, which, nev, tol, resid, ncv, v,
 &        ldv, iparam, ipntr, workd, workl, lworkl, rwork,info )
```

Figure 3.8: Calling the ARPACK subroutine `znaupd`.

There is little computational penalty for using the non-Hermitian routines in this case. The only additional cost is to compute eigenvalues of a Hessenberg rather than a tridiagonal matrix. For the problem configurations this software is designed to solve, the size of these matrices are small enough that the differences in computational cost are negligible compared to the major $\mathcal{O}(n)$ work that is required.

The integer `ido` is the reverse communication flag that specifies a requested action on return from `dnaupd`. The `character*1` parameter `bmat` specifies if this is a standard `bmat = 'I'` or a generalized `bmat = 'G'` problem. The integer `n` specifies the dimension of the problem. The `character*2` parameter `which` may take the same possible values listed for subroutine `dnaupd` in Table 3.4.

There are three shift-invert modes for complex problems. These modes are specified by setting the parameter entry `iparam(7) = mode` where `mode = 1,2, or 3`.

In the following list, the specification of `OP` and $\mathbf{B}$ are given for the various modes. Also, the `iparam(7)` and `bmat` settings are listed along with the name of the sample driver for the given mode. Sample drivers for the following modes may be found in the `EXAMPLES/COMPLEX` subdirectory.

1. Regular mode (`iparam(7) = 1, bmat = 'I'`). Use driver `zndrv1`.
   (a) Solve $\mathbf{Ax} = \mathbf{x}\lambda$ in regular mode.
   (b) `OP` $= \mathbf{A}$ and $\mathbf{B} = \mathbf{I}$.
2. Shift-invert mode (`iparam(7) = 3, bmat = 'I'`). Use driver `zndrv2`.
   (a) Solve $\mathbf{Ax} = \mathbf{x}\lambda$ in shift-invert mode.
   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{I})^{-1}$ and $\mathbf{B} = \mathbf{I}$.
3. Regular inverse mode (`iparam(7) = 2, bmat = 'G'`). Use driver `zndrv3`.
   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in regular inverse mode.
   (b) `OP` $= \mathbf{M}^{-1}\mathbf{A}$ and $\mathbf{B} = \mathbf{M}$.
4. Shift-invert mode (`iparam(7) = 3, bmat = 'G'`). Use driver `zndrv4`.
   (a) Solve $\mathbf{Ax} = \mathbf{Mx}\lambda$ in shift-invert mode.
   (b) `OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ and $\mathbf{B} = \mathbf{M}$.

```
c
c        %-------------------------------------------%
c        | No fatal errors occurred.                 |
c        | Post-Process using CNEUPD.                |
c        |                                           |
c        | Computed eigenvalues may be extracted.    |
c        |                                           |
c        | Eigenvectors may also be computed now if  |
c        | desired.  (indicated by rvec = .true.)    |
c        %-------------------------------------------%
c
         rvec = .true.
c
         call cneupd (rvec, 'A', select, d, v, ldv, sigma,
     &        workev, bmat, n, which, nev, tol, resid, ncv, v,
     &        ldv, iparam, ipntr, workd, workl, lworkl, rwork,
     &        ierr)
c
```

Figure 3.9: Post-Processing for Eigenvectors Using `cneupd`.

3.10 Post-Processing for Eigenvectors Using zneupd

On the final return from `znaupd` (indicated by `ido = 99`), the error flag `info` must be checked. If `info = 0`, then no fatal errors have occurred and it is time to post-process using `zneupd` to get eigenvalues of the original problem and the corresponding eigenvectors if desired. In the case shown here (shift-invert and generalized), there are some subtleties to recovering eigenvectors when **M** is ill-conditioned. This process is called eigenvector purification. It prevents eigenvectors from being corrupted with noise due to the presence of eigenvectors corresponding to nearly infinite eigenvalues. Details are given in Chapter 4. These operations are completely transparent to the user. The general calling sequence for `dseupd` is shown in Figure 3.9.

The input parameters `bmat, n,` $\cdots$, `info` are precisely the same parameters that appear in the calling sequence of `znaupd`. It is extremely IMPORTANT that none of these parameters are altered between the final return from `znaupd` and the subsequent call to `zneupd`.

There is negligible additional cost to obtain eigenvectors. An orthonormal (Schur) basis for the invariant subspace corresponding to the converged approximate eigenvalues is always computed. In the above example, this basis is overwritten with the eigenvectors in the array `v`. Both basis sets may be obtained if desired but there is an additional storage cost of `n`$\cdot$`nev` if both are requested (in this case a separate `n` by `nev` array `z` must be supplied). In some cases it may be desirable to have both basis sets.

In the non-Hermitian case, the eigenvector basis is potentially ill-conditioned and may not even exist. While, eigenvectors may have physical meaning, they are generally not the best basis to use. If a basis for a selected invariant subspace is required, then it is generally better to compute a Schur basis. This will provide an orthogonal, hence well conditioned, basis for the subspace. The sensitivity of a given subspace to perturbations (such as roundoff error) is another question. See § 4.6 for a brief discussion.

If it is desirable to retain the Schur basis in `v` and storage is an issue, the user may elect to call this routine once for each desired eigenvector and store it peripherally. There is also the option of computing a selected set of these vectors with a single call.

The input parameters that must be specified are

- The logical variable `rvec` = `.true.` if eigenvectors are requested `.false.` if only eigenvalues are desired.

- The Character*1 parameter `howmny` specifies how many eigenvectors are desired.

  - `howmny = 'A'`: compute `nev` eigenvectors;
  - `howmny = 'P'`: Compute `nev` Schur vectors;
  - `howmny = 'S'`: compute some of the eigenvectors, specified by the logical array `select`.

- `sigma` should contain the value of the (complex) shift that was used if `iparam(7) = 3` . It is referenced if `iparam(7) = 1 or 2`.

When requested, the eigenvectors returned by `zneupd` are normalized to have unit length with respect to the $\mathbf{B}$ semi-inner product that was used. Thus, if $\mathbf{B} = \mathbf{I}$ they will have unit length in the standard 2-norm. In general, a computed eigenvector $\mathbf{x}$ will satisfy $1 = \mathbf{x}^H\mathbf{B}\mathbf{x}$.

Chapter 4

The Implicitly Restarted Arnoldi Method

This chapter presents an overview of the theory of Krylov subspace projection and the underlying algorithms implemented in ARPACK. The basic Implicitly Restarted Arnoldi Method (IRAM) is quite simple in structure and is very closely related to the Implicitly Shifted QR-Algorithm for dense problems. This discussion is intended to give a broad overview of the theory and to develop a high level description of the algorithms. Specific implementation details concerned with efficiency and numerical stability are treated in Chapter 5.

The remainder of this chapter will develop enough background to understand the origins, motivation, and expected behavior of this algorithm. The discussion begins with a very brief review of the structure of the algebraic eigenvalue problem and some basic numerical methods that either influence or play a direct role in the IRAM. Overcoming the basic disadvantages of the simple power method motivates the introduction of Krylov subspaces along with the important projection idea and the related approximation properties. The Lanczos/Arnoldi factorization is introduced as a concrete way to construct an orthogonal basis for a Krylov subspace and provides a means to implement the projection numerically. Implicit restarting is introduced as an efficient way to overcome the often intractable storage and computational requirements in the original Lanczos/Arnoldi method. This new technique turns out to be a truncated form of the implicitly shifted QR algorithm and hence implementation issues and ultimate behavior are closely tied to that well understood method. Because of its reduced storage and computational requirements, the technique is suitable for large scale eigenvalue problems. Implicit restarting provides a means to approximate a few eigenvalues with user specified properties in space proportional to $n \cdot k$ where k is the number of eigenvalues sought.

Generalized eigenvalue problems are discussed in some detail. They arise naturally in PDE applications and they have a number of subtleties with respect to numerically stable implementation of spectral transformations. Spectral transformations are presented within the context of the generalized prob-

Figure 4.1: The Implicitly Restarted Arnoldi Method in ARPACK.

- *Start:* Build a length m Arnoldi factorization $\mathbf{AV}_m = \mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T$ with the starting vector $\mathbf{v}_1$.
- *Iteration:* Until convergence
  1. Compute the eigenvalues $\{\lambda_j : j = 1, 2, \cdots, m\}$ of $\mathbf{H}_m$. Sort these eigenvalues according to the user selection criterion into a wanted set $\{\lambda_j : j = 1, 2, \cdots, k\}$ and an unwanted set $\{\lambda_j : j = k+1, k+2, \cdots, m\}$.
  2. Perform $m - k = p$ steps of the QR iteration with the unwanted eigenvalues $\{\lambda_j : j = k+1, k+2, \cdots, m\}$ as shifts to obtain $\mathbf{H}_m\mathbf{Q}_m = \mathbf{Q}_m\mathbf{H}_m^+$.
  3. *Restart:* Postmultiply the length m Arnoldi factorization with the matrix $\mathbf{Q}_k$ consisting of the leading k columns of $\mathbf{Q}_m$ to obtain the length k Arnoldi factorization $\mathbf{AV}_m\mathbf{Q}_k = \mathbf{V}_m\mathbf{Q}_k\mathbf{H}_k^+ + \mathbf{f}_k^+\mathbf{e}_k^T$, where $\mathbf{H}_k^+$ is the leading principal submatrix of order k for $\mathbf{H}_m^+$. Set $\mathbf{V}_k \leftarrow \mathbf{V}_m\mathbf{Q}_k$.
  4. Extend the length k Arnoldi factorization to a length m factorization.

lem as a means to improve the performance of Krylov methods.

The basic iteration of the IRAM is outlined in Figure 4.1 for those familiar with Krylov subspace methods and basic dense eigenvalue methods. In the iteration shown, $\mathbf{H}_m$ is an $m \times m$ upper Hessenberg matrix, $\mathbf{V}_m^H\mathbf{V}_m = \mathbf{I}_m$, and the residual vector $\mathbf{f}_m$ is orthogonal to the columns of $\mathbf{V}_m$.

4.1 Structure of the Eigenvalue Problem

A brief discussion of the mathematical structure of the eigenvalue problem is necessary to fix notation and introduce ideas that lead to an understanding of the behavior, strengths and limitations of the algorithm. In this discussion, the real and complex number fields are denoted by $\mathbf{R}$ and $\mathbf{C}$ respectively. The standard n-dimensional real and complex vectors are denoted by $\mathbf{R}^n$ and $\mathbf{C}^n$ and the symbols $\mathbf{R}^{m\times n}$ and $\mathbf{C}^{m\times n}$ will denote the real and complex matrices with m rows and n columns. Scalars are denoted by lower case Greek letters, vectors are denoted by lower case Latin letters and matrices by capital Latin letters. The transpose of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^T$ and the conjugate-transpose by $\mathbf{A}^H$. The symbol, $\|\cdot\|$ will denote the Euclidean or 2-norm of a vector. The standard basis for $\mathbf{C}^n$ is denoted by the set $\{\mathbf{e}_j\}_{j=1}^n$.

The set of numbers $\sigma(\mathbf{A}) \equiv \{\lambda \in \mathbf{C} : \text{rank}(\mathbf{A} - \lambda I) < n)\}$ is called the *spectrum* of $\mathbf{A}$. The elements of this discrete set are the *eigenvalues* of $\mathbf{A}$

and they may be characterized as the n roots of the *characteristic polynomial* $p_A(\lambda) \equiv \det(\lambda I - \mathbf{A})$. Corresponding to each distinct eigenvalue $\lambda \in \sigma(\mathbf{A})$ is at least one nonzero vector $\mathbf{x}$ such that $\mathbf{A}\mathbf{x} = \mathbf{x}\lambda$. This vector is called a right eigenvector of $\mathbf{A}$ corresponding to the eigenvalue λ. The pair $(\mathbf{x}, \lambda)$ is called an *eigenpair*. A nonzero vector $\mathbf{y}$ such that $\mathbf{y}^H\mathbf{A} = \lambda\mathbf{y}^H$ is called a *left* eigenvector. The multiplicity $n_a(\lambda)$ of λ as a root of the characteristic polynomial is the *algebraic* multiplicity and the dimension $n_g(\lambda)$ of $Null(\lambda I - \mathbf{A})$ is the *geometric* multiplicity of λ. The matrix $\mathbf{A}$ is *defective* if $n_g(\lambda) < n_a(\lambda)$ for some eigenvalue and otherwise it is *non-defective*. The eigenvalue λ is *simple* if $n_a(\lambda) = 1$.

A subspace $\mathcal{S}$ of $\mathbf{C}^{n\times n}$ is called an *invariant subspace* of $\mathbf{A}$ if $\mathbf{A}\mathcal{S} \subset \mathcal{S}$. It is straightforward to show if $\mathbf{A} \in \mathbf{C}^{n\times n}$, $\mathbf{X} \in \mathbf{C}^{n\times k}$ and $\mathbf{G} \in \mathbf{C}^{k\times k}$ satisfy

$$\mathbf{A}\mathbf{X} = \mathbf{X}\mathbf{G}, \tag{4.1.1}$$

then $\mathcal{S} \equiv Range(\mathbf{X})$ is an invariant subspace of $\mathbf{A}$. Moreover, if $\mathbf{X}$ has full column rank k then the columns of $\mathbf{X}$ form a basis for this subspace and $\sigma(\mathbf{G}) \subset \sigma(\mathbf{A})$. If, in addition, $k = n$ then $\sigma(\mathbf{G}) = \sigma(\mathbf{A})$ and $\mathbf{A}$ is said to be *similar* to $\mathbf{G}$ under the *similarity* transformation with $\mathbf{X}$. $\mathbf{A}$ is *diagonalizable* if it is similar to a diagonal matrix and this property is equivalent to $\mathbf{A}$ being non-defective.

Invariant subspaces are central to the methods developed here. Invariant subspaces generalize the notion of eigenvectors. If $\mathbf{A}\mathbf{x} = \mathbf{x}\lambda$, then $\mathbf{X} = (\mathbf{x})$, $\mathbf{G} = (\lambda)$ and $\mathcal{S} \equiv Range(\mathbf{X}) = Span(\mathbf{x})$ is a one dimensional invariant subspace of $\mathbf{A}$. More generally, if $\mathbf{A}\mathbf{x}_j = \mathbf{x}_j\lambda_j$ for $j = 1, 2, \cdots, k$ and we put $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k)$, then $\mathcal{S} \equiv Range(\mathbf{X})$ is an invariant subspace of $\mathbf{A}$ and indeed $\mathbf{A}\mathbf{X} = \mathbf{X}\mathbf{\Lambda}$ where $\mathbf{\Lambda} = diag(\lambda_1, \lambda_2, \cdots, \lambda_k)$. If $\mathbf{X} = \mathbf{Q}\hat{\mathbf{R}}$ where $\mathbf{Q}$ is unitary and $\hat{\mathbf{R}}$ is upper triangular(the standard QR-factorization), then $\mathbf{A}\mathbf{Q} = \mathbf{Q}\mathbf{R}$ where $\mathbf{R} = \hat{\mathbf{R}}\mathbf{\Lambda}\hat{\mathbf{R}}^{-1}$ is an upper triangular matrix with the eigenvalues λ_j on its diagonal. The columns of $\mathbf{Q}$ provide an orthonormal basis for the invariant subspace $\mathcal{S}$.

A large condition number $\|\hat{\mathbf{R}}\| \cdot \|\hat{\mathbf{R}}^{-1}\|$ of $\hat{\mathbf{R}}$ and hence of $\mathbf{X}$ will indicate these eigenvalues and this invariant subspace are sensitive to perturbations in $\mathbf{A}$ (such as those introduced by roundoff error in a finite precision computation). But this is not the entire story. Separation of eigenvalues and angles between invariant subspaces also come into play. In the symmetric (Hermitian) case, invariant subspaces corresponding to distinct eigenvalues are orthogonal to each other and completely decouple the action of the matrix (as an operator on $\mathbf{C}^n$. In the non-symmetric case, such a decoupling is generally not possible. The nature of the coupling is completely described by the Jordan canonical form but this form is usually extremely sensitive to perturbations and hence unsuitable as the basis for a computational algorithm.

The Schur decomposition [?] does provide a means to express this coupling and provides a foundation for the development of stable numerical algorithms.

In particular, the implicitly shifted QR algorithm computes a Schur decomposition. Schur's result states that every square matrix is *unitarily similar* to an upper triangular matrix. In other words, for any linear operator on $\mathbf{C}^n$, there is a unitary basis in which the operator has an upper triangular matrix representation. The following result may be found in [**?**].

Theorem 4.1.1 *(Schur Decomposition). Let* $\mathbf{A} \in \mathbf{C}^{n\times n}$*. Then there is a unitary matrix* $\mathbf{Q}$ *and an upper triangular matrix* $\mathbf{R}$ *such that*

$$\mathbf{AQ} \;=\; \mathbf{QR}. \tag{4.1.2}$$

The diagonal elements of $\mathbf{R}$ *are the eigenvalues of* $\mathbf{A}$*.*

A Schur decomposition is not unique. The eigenvalues of $\mathbf{A}$ may appear on the diagonal of $\mathbf{R}$ in any specified order. Thus, for any specified set of k eigenvalues of $\mathbf{A}$, there is a Schur decompostion such that these k eigenvalues appear as diagonal elements of the leading principal submatrix $\mathbf{R}_k$ of the upper triangular matrix $\mathbf{R}$. If $\mathbf{Q}_k$ denotes the leading k columns of the corresponding unitary matrix $\mathbf{Q}$, then

$$\mathbf{AQ}_k = \mathbf{Q}_k\mathbf{R}_k \tag{4.1.3}$$

is obtained by equating the leading k columns on both sides of (4.1.2). We shall call this a *partial Shur decomposition.* Note that $\mathcal{S} = Range(\mathbf{Q}_k)$ is an invariant subspace of $\mathbf{A}$ corresponding to the k specified eigenvalues and that the columns of $\mathbf{Q}_k$ form an orthonormal basis for this space. This is called a *Schur basis* for the invariant subspace. The Implicitly Restarted Arnoldi Method is designed to compute a partial Schur form corresponding to selected eigenvalues of $\mathbf{A}$.

The fundamental structure of Hermitian and normal matrices is easily derived from the Schur decomposition.

Lemma 4.1.2 *A matrix* $\mathbf{A} \in \mathbf{C}^{n\times n}$ *is normal* $(\mathbf{AA}^H = \mathbf{A}^H\mathbf{A})$ *if and only if* $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^H$ *with* $\mathbf{Q} \in \mathbf{C}^{n\times n}$ *unitary and* $\mathbf{\Lambda} \in \mathbf{C}^{n\times n}$ *diagonal. Morevover,* $\mathbf{A}$ *is Hermitian* $(\mathbf{A} = \mathbf{A}^H)$ *if and only if* $\mathbf{\Lambda}$ *is diagonal with real entries. In either case, the diagonal entries of* $\mathbf{\Lambda}$ *are the eigenvalues of* $\mathbf{A}$ *and the columns of* $\mathbf{Q}$ *are the corresponding eigenvectors.*

For purposes of algorithmic development this structure is fundamental. In fact, the well known Implicitly Shifted QR-Algorithm [**?**, **?**] is designed to produce a sequence of unitary similarity transformations with $\mathbf{Q}_j$ that iteratively reduce $\mathbf{A}$ to upper triangular form. This algorithm begins with an initial unitary similarity transformation of $\mathbf{A}$ with $\mathbf{V}$ to the condensed form $\mathbf{AV} = \mathbf{VH}$ where $\mathbf{H}$ is upper Hessenberg matrix. An upper Hessenberg matrix has zero elements below its main subdiagonal so it is almost upper triangular. When $\mathbf{A}$ is Hermitian, $\mathbf{H}$ is a real symmetric tridiagonal matrix in which case all the elements are zero except for those on the main, sub and super diagonals.

Figure 4.2: Algorithm 1: Shifted QR-iteration.

Input: $(\mathbf{A}, \mathbf{V}, \mathbf{H})$ with $\mathbf{AV} = \mathbf{VH}, \mathbf{V}^H\mathbf{V} = \mathbf{I}$, $\mathbf{H}$ upper Hessenberg;
For $j = 1, 2, 3, ...$ until *convergence*,
  (a1.1) Select a shift $\mu \leftarrow \mu_j$
  (a1.2) Factor $[\mathbf{Q}, \mathbf{R}] = \mathrm{qr}(\mathbf{H} - \mu\mathbf{I})$;
  (a1.3) $\mathbf{H} \leftarrow \mathbf{Q}^H\mathbf{HQ}$; $\mathbf{V} \leftarrow \mathbf{VQ}$;
End_For

The QR-iteration is shown in Figure 4.2. The QR factorization of $\mathbf{H} - \mu\mathbf{I}$ is given by the unitary matrix $\mathbf{Q}$ and upper triangular $\mathbf{R}$. It is easy to see that $\mathbf{H}$ is unitarily similar to $\mathbf{A}$ throughout the course of this iteration. The iteration is continued until the subdiagonal elements of $\mathbf{H}$ converge to zero, i.e. until a Schur decomposition has been (approximately) obtained. In the standard implicitly shifted QR-iteration, the unitary matrix $\mathbf{Q}$ is never actually formed. It is computed indirectly as a product of 2×2 Givens or 3×3 Householder transformations through a "bulge chase" process. The elegant details of an efficient and stable implementation would be too much of a digression here. They may be found in [**?**, **?**, **?**]. The convergence behavior of this iteration is fascinating. The columns of $\mathbf{V}$ converge to Schur vectors at various rates. These rates are fundamentally linked to the simple power method and its rapidly convergent variant, inverse iteration. See [**?**] for further information and references.

Despite the extremely fast rate of convergence and the efficient use of storage, the implicitly shifted QR method is not suitable for large scale problems and it has proven to be extremely difficult to parallelize. Large scale problems are typically sparse or structured so that a matrix-vector product $\mathbf{w} \leftarrow \mathbf{Av}$ may be computed with time and storage proportional to n rather than n^2. A method based upon full similarity transformations quickly destroys this structure. Storage and operation counts per iteration become order n^2. Hence, there is considerable motivation for methods that only require matrix-vector products with the original $\mathbf{A}$.

The power method provides a simple means to find the dominant eigenvalue (of largest magnitude) of a matrix without performing matrix factorizations and dense similarity transformations. It has the drawback that only one eigenpair may be found and that convergence may be slow or non-existent. Deflation schemes and/or block variants may be employed to overcome the first problem and spectral transformations may be used to accelerate convergence and focus on selected eigenvalues. However, there is another very elegant way to extract additional eigenvalue information from the power method sequence.

4.2 Krylov Subspaces and Projection Methods

The methods that underly the ARPACK software are derived from a class of algorithms called Krylov subspace projection methods. These methods take full advantage of the intricate structure of the sequence of vectors naturally produced by the power method.

An examination of the behavior of the sequence of vectors produced by the power method suggests that the successive vectors may contain considerable information along eigenvector directions corresponding to eigenvalues other than the one with largest magnitude. The expansion coefficients of the vectors in the sequence evolve in a very structured way. Therefore, linear combinations of the these vectors can be constructed to enhance convergence to additional eigenvectors. A single vector power iteration simply ignores this additional information, but more sophisticated techniques may be employed to extract it.

If one hopes to obtain additional information through various linear combinations of the power sequence, it is natural to formally consider the *Krylov* subspace

$$\mathcal{K}_k(\mathbf{A}, \mathbf{v}_1) = \text{Span }\{\mathbf{v}_1, \mathbf{A}\mathbf{v}_1, \mathbf{A}^2\mathbf{v}_1, \ldots, \mathbf{A}^{k-1}\mathbf{v}_1\}$$

and to attempt to formulate the best possible approximations to eigenvectors from this subspace.

It is reasonable to construct approximate eigenpairs from this subspace by imposing a Galerkin condition: A vector $\mathbf{x} \in \mathcal{K}_k(\mathbf{A}, \mathbf{v}_1)$ is called a *Ritz vector* with corresponding *Ritz value* θ if the Galerkin condition

$$< \mathbf{w}, \mathbf{A}\mathbf{x} - \mathbf{x}\theta >= 0\ , \quad \text{for all} \quad \mathbf{w} \in \mathcal{K}_k(\mathbf{A}, \mathbf{v}_1)$$

is satisfied. There are some immediate consequences of this definition: Let $\mathbf{W}$ be a matrix whose columns form an orthonormal basis for $\mathcal{K}_k \equiv \mathcal{K}_k(\mathbf{A}, \mathbf{v}_1)$. Let $\mathcal{P} = \mathbf{W}\mathbf{W}^H$ denote the related orthogonal projector onto $\mathcal{K}_k$ and define $\hat{\mathbf{A}} \equiv \mathcal{P}\mathbf{A}\mathcal{P} = \mathbf{W}\mathbf{G}\mathbf{W}^H$ where $\mathbf{G} \equiv \mathbf{W}^H\mathbf{A}\mathbf{W}$. It can be shown that

Lemma 4.2.1 *For the quantities defined above:*

1. $(\mathbf{x}, \theta)$ *is a Ritz-pair if and only if* $\mathbf{x} = \mathbf{W}\mathbf{s}$ *with* $\mathbf{G}\mathbf{s} = \mathbf{s}\theta$ *.*
2. $\|(\mathbf{I} - \mathcal{P})\mathbf{A}\mathbf{W}\| = \|(\mathbf{A} - \hat{\mathbf{A}})\mathbf{W}\| \leq \|(\mathbf{A} - \mathbf{M})\mathbf{W}\|$
   for all $\mathbf{M} \in \mathbf{C}^{n\times n}$ *such that* $\mathbf{M}\mathcal{K}_k \subset \mathcal{K}_k$.
3. *The Ritz-pairs* $(\mathbf{x}, \theta)$ *and the minimum value* $\|(\mathbf{I} - \mathcal{P})\mathbf{A}\mathbf{W}\|$ *are independent of the choice of orthonormal basis* $\mathbf{W}$.

These facts are actually valid for any k dimensional subspace $\mathcal{S}$ in place of $\mathcal{K}_k$. Additional useful properties may be derived as consequences of the fact that every $\mathbf{w} \in \mathcal{K}_k$ is of the form $\mathbf{w} = \phi(\mathbf{A})\mathbf{v}_1$ for some polynomial ϕ of degree less than k. A thorough discussion is given by Saad in [12] and in his earlier papers. These facts have important algorithmic consequences. In particular, it may be shown that $\mathcal{K}_k$ is an invariant subspace for $\mathbf{A}$ if and only if the starting vector $\mathbf{v}_1$ is a linear combination of vectors spanning an invariant subspace of $\mathbf{A}$. An important example of this is to put $\mathbf{v}_1 = \mathbf{Q}_k\mathbf{s}$ where $\mathbf{A}\mathbf{Q}_k = \mathbf{Q}_k\mathbf{R}_k$ is a partial Schur decomposition of $\mathbf{A}$.

There is some algorithmic motivation to seek a convenient orthonormal basis $\mathbf{V} = \mathbf{W}\mathbf{Q}$ that will provide a means to successively construct these basis vectors. It is possible to construct a $k \times k$ unitary $\mathbf{Q}$ using standard Householder transformations such that $\mathbf{v}_1 = \mathbf{V}\mathbf{e}_1$ and $\mathbf{H} = \mathbf{Q}^H\mathbf{G}\mathbf{Q}$ is upper Hessenberg with non-negative subdiagonal elements. It is also possible to show that in this basis,

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{H} + \mathbf{f}\mathbf{e}_k^T, \quad \text{where} \quad \mathbf{f} = \gamma\hat{p}(\mathbf{A})\mathbf{v}_1,$$

with $\mathbf{V}^H\mathbf{f} = \mathbf{0}$ implied by the projection property and $\hat{p}(\lambda) = \det(\lambda\mathbf{I} - \mathbf{H})$.

If it is possible to obtain a $\mathbf{v}_1$ as a linear combination of k eigenvectors of $\mathbf{A}$, the first observation implies that $\mathbf{f} = \mathbf{0}$ and $\mathbf{V}$ is an orthonormal basis for an invariant subspace of $\mathbf{A}$. Hence, the Ritz values $\sigma(\mathbf{H})$ are eigenvalues and corresponding Ritz vectors are eigenvectors for $\mathbf{A}$. The second observation leads to the Lanczos/Arnoldi process [**?**, **?**].

4.3 The Arnoldi Factorization

Definition : If $\mathbf{A} \in \mathbf{C}^{n\times n}$ then a relation of the form

$$\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T$$

where $\mathbf{V}_k \in \mathbf{C}^{n\times k}$ has orthonormal columns, $\mathbf{V}_k^H\mathbf{f}_k = 0$ and $\mathbf{H}_k \in \mathbf{C}^{k\times k}$ is upper Hessenberg with non-negative subdiagonal elements is called a *k-step Arnoldi Factorization* of $\mathbf{A}$. If $\mathbf{A}$ is Hermitian then $\mathbf{H}_k$ is real, symmetric and tridiagonal and the relation is called a *k-step Lanczos Factorization* of $\mathbf{A}$. The columns of $\mathbf{V}_k$ are referred to as the *Arnoldi vectors* or *Lanczos vectors* , respectively.

The preceding development of this factorization has been purely through the consequences of the orthogonal projection imposed by the Galerkin conditions. (A more straightforward but less illuminating derivation is to equate the first k columns of the Hessenberg decomposition $\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{H}$.) An alternative way to write this factorization is

$$\mathbf{A}\mathbf{V}_k = (\mathbf{V}_k, \mathbf{v}_{k+1}) \begin{pmatrix} \mathbf{H}_k \\ \beta_k\mathbf{e}_k^T \end{pmatrix} \quad \text{where} \quad \beta_k = \|\mathbf{f}_k\| \quad \text{and} \quad \mathbf{v}_{k+1} = \frac{1}{\beta_k}\mathbf{f}_k \; .$$

This factorization may be used to obtain approximate solutions to a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ if $\mathbf{b} = \mathbf{v}_1\beta_0$ and this underlies the GMRES method [**?**]. However, the purpose here is to investigate the use of this factorization to obtain approximate eigenvalues and eigenvectors. The discussion of the previous section implies that Ritz pairs satisfying the Galerkin condition are immediately available from the eigenpairs of the small projected matrix $\mathbf{H}_k$.

If $\mathbf{H}_k\mathbf{s} = \mathbf{s}\theta$ then the vector $\mathbf{x} = \mathbf{V}_k\mathbf{s}$ satisfies

$$\|\mathbf{A}\mathbf{x} - \mathbf{x}\theta\| = \|(\mathbf{A}\mathbf{V}_k - \mathbf{V}_k\mathbf{H}_k)\mathbf{s}\| = |\beta_k \mathbf{e}_k^T \mathbf{s}|.$$

The number $|\beta_k \mathbf{e}_k^T \mathbf{s}|$ is called the *Ritz estimate* for the the Ritz pair $(\mathbf{x}, \theta)$ as an approximate eigenpair for $\mathbf{A}$. Observe that if $(\mathbf{x}, \theta)$ is a Ritz pair then

$$\theta = \mathbf{s}^H\mathbf{H}_k\mathbf{s} = (\mathbf{V}_k\mathbf{s})^H\mathbf{A}(\mathbf{V}_k\mathbf{s}) = \mathbf{x}^H\mathbf{A}\mathbf{x}$$

is a Rayleigh quotient (assuming $\|\mathbf{s}\| = 1$) and the associated Rayleigh quotient residual $\mathbf{r}(\mathbf{x}) \equiv \mathbf{A}\mathbf{x} - \mathbf{x}\theta$ satisfies

$$\|\mathbf{r}(\mathbf{x})\| = |\beta_k \mathbf{e}_k^T \mathbf{s}|.$$

When $\mathbf{A}$ is Hermitian, this relation may be used to provide computable rigorous bounds on the accuracy of the eigenvalues of $\mathbf{H}_k$ as approximations to eigenvalues [?] of $\mathbf{A}$. When $\mathbf{A}$ is non-Hermitian the possibility of non-normality precludes such bounds and one can only say that the Rayleigh Quotient residual is small if $|\beta_k \mathbf{e}_k^T \mathbf{s}|$ is small without further information. However, in either case, if $\mathbf{f}_k = 0$ these Ritz pairs become exact eigenpairs of $\mathbf{A}$.

The k-step factorization may be advanced one step at the cost of a (sparse) matrix-vector product involving $\mathbf{A}$ and two dense matrix vector products involving $\mathbf{V}_k^H$ and $\mathbf{V}_k$. The explicit steps needed to form a k-Step Arnoldi factorization are listed in Algorithm 2 shown in Figure 4.3. In exact arithmetic, the columns of $\mathbf{V}_j$ form an orthonormal basis for the Krylov subspace and $\mathbf{H}_j$ is the orthogonal projection of $\mathbf{A}$ onto this space. In finite precision arithmetic, care must be taken to assure that the computed vectors are orthogonal to working precision. The method proposed by Daniel, Gragg, Kaufman and Stewart (DGKS) in [?] provides an excellent way to construct a vector $\mathbf{f}_{j+1}$ that is numerically orthogonal to $\mathbf{V}_{j+1}$. It amounts to computing a correction

$$\mathbf{c} = \mathbf{V}_{j+1}^H\mathbf{f}_{j+1}; \quad \mathbf{f}_{j+1} \leftarrow \mathbf{f}_{j+1} - \mathbf{V}_{j+1}\mathbf{c}; \quad \mathbf{h} \leftarrow \mathbf{h} + \mathbf{c};$$

just after Step (a2.4) if necessary. A simple test is used to avoid this DGKS correction if it is not needed.

The dense matrix-vector products at Step (a2.4) and also the correction may be accomplished using Level 2 BLAS. This is important for performance on vector, and parallel-vector supercomputers. The Level 2 BLAS operation _GEMV is easily parallelized and vectorized and has a much better ratio of floating point computation to data movement [3, ?] than the Level 1 BLAS operations.

The information obtained through this process is completely determined by the choice of the starting vector. Eigen-information of interest may not appear until k gets very large. In this case it becomes intractable to maintain numerical orthogonality of the basis vectors $\mathbf{V}_k$. Moreover, extensive storage will be required and repeatedly finding the eigensystem of $\mathbf{H}_k$ will become prohibitive at a cost of $\mathcal{O}(k^3)$ flops.

Figure 4.3: Algorithm 2: The k-Step Arnoldi Factorization

Input: $(\mathbf{A}, \mathbf{v}_1)$
Put $\mathbf{v}_1 = \mathbf{v}/\|\mathbf{v}_1\|$; $\mathbf{w} = \mathbf{A}\mathbf{v}_1$; $\alpha_1 = \mathbf{v}_1^H \mathbf{w}$;
Put $\mathbf{f}_1 \leftarrow \mathbf{w} - \mathbf{v}_1\alpha_1$; $\mathbf{V}_1 \leftarrow (\mathbf{v}_1)$; $\mathbf{H}_1 \leftarrow (\alpha_1)$;
For $j = 1, 2, 3, ...$k-1,
(a2.1) $\beta_j = \|\mathbf{f}_j\|$; $\mathbf{v}_{j+1} \leftarrow \mathbf{f}_j/\beta_j$;
(a2.2) $\mathbf{V}_{j+1} \leftarrow (\mathbf{V}_j, \mathbf{v}_{j+1})$; $\hat{\mathbf{H}}_j \leftarrow \begin{pmatrix} \mathbf{H}_j \\ \beta_j \mathbf{e}_j^T \end{pmatrix}$;
(a2.3) $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}_{j+1}$;
(a2.4) $\mathbf{h} \leftarrow \mathbf{V}_{j+1}^H \mathbf{w}$; $\mathbf{f}_{j+1} \leftarrow \mathbf{w} - \mathbf{V}_{j+1}\mathbf{h}$;
(a2.5) $\mathbf{H}_{j+1} \leftarrow (\hat{\mathbf{H}}_j, \mathbf{h})$;
End_For

Failure to maintain orthogonality leads to several numerical difficulties. In a certain sense, the computation (or approximation) of the projection indicated at Step (a2.4) in a way that overcomes these difficulties has been the main source of research activity for the Lanczos method. Paige [**?**] showed that the loss of orthogonality occurs precisely when an eigenvalue of $\mathbf{H}_j$ is close to an eigenvalue of $\mathbf{A}$. In fact, the Lanczos vectors lose orthogonality in the direction of the associated approximate eigenvector. Moreover, failure to maintain orthogonality results in spurious copies of the approximate eigenvalue produced by the Lanczos method. Implementations based on selective and partial orthogonalization [4, **?**, **?**] monitor the loss of orthogonality and perform additional orthogonalization steps only when necessary. The approaches given in [**?**, **?**, **?**] use only the three term recurrence with no additional orthogonalization steps. These strategies determine whether multiple eigenvalues produced by the Lanczos method are spurious or correspond to true multiple eigenvalue of $\mathbf{A}$. However, these approaches do not generate eigenvectors directly. Additional computation is required if eigenvectors are needed and this amounts to re-generating the basis vectors with the Lanczos process.

A related numerical difficulty associated with failure to maintain orthogonality stems from the fact that $\|\mathbf{f}_k\| = 0$ if and only if the columns of $\mathbf{V}_k$ span an invariant subspace of $\mathbf{A}$. When $\mathbf{V}_k$ "nearly" spans such a subspace, $\|\mathbf{f}_k\|$ should be small. Typically, in this situation, a loss of significant digits will take place at Step (a2.4) through numerical cancellation unless special care is taken (i.e. use of the DGKS correction).

Understanding this difficulty and overcoming it is essential for the IRA iteration which is designed to drive $\|\mathbf{f}_k\|$ to 0. It is desirable for $\|\mathbf{f}_k\|$ to become small because this indicates that the eigenvalues of $\mathbf{H}_k$ are the eigenvalues of a slightly perturbed matrix that is near to $\mathbf{A}$. This is easily seen from a re-arrangement of the Arnoldi relation:

$$(\mathbf{A} - \mathbf{f}_k \mathbf{v}_k^H)\mathbf{V}_k = \mathbf{V}_k \mathbf{H}_k$$

where $\mathbf{v}_k = \mathbf{V}_k \mathbf{e}_k$. Hence, $Range(\mathbf{V}_k)$ is an invariant subspace for the perturbed matrix $\mathbf{A} + \mathbf{E}$ with $\|\mathbf{E}\| = \|\mathbf{f}_k \mathbf{v}_k^H\| = \|\mathbf{f}_k\|$. Thus, the eigenvalues of $\mathbf{H}_k$ will be accurate approximations to the eigenvalues of $\mathbf{A}$ if the approximated eigenvalues are insensitive to perturbations (i.e. well conditioned). However, this "convergence" may well lead to loss of numerical orthogonality in the updated basis $\mathbf{V}_{k+1}$ unless care is taken to avoid this. Moreover, if subsequent Arnoldi vectors are not forced to be orthogonal to the converged ones, then components along these converged directions re-enter the basis via round-off effects and quickly cause a spurious copy of the previously computed eigenvalue to appear in the spectrum of the projected matrix $\mathbf{H}_k$. Repetition of this phenomenon can cause several spurious approximations to the same eigenvalue to occur.

4.4 Restarting the Arnoldi Method

An unfortunate aspect of the Lanczos/Arnoldi process is that one cannot know in advance how many steps will be required before eigenvalues of interest are well approximated by the Ritz values. This is particularly true when the problem has a wide range of eigenvalues but the eigenvalues of interest are clustered. Without a spectral transformation, many Lanczos steps are required to obtain the selected eigenvalues. In order to recover eigenvectors, one is obliged either to store all of the Lanczos basis vectors (usually on a peripheral device) or to re-compute them. Also, very large tridiagonal eigenvalue problems will have to be solved at each step. In the Arnoldi process that is used in the non-Hermitian case, not only do the basis vectors have to be stored, but the cost of the Hessenberg eigenvalue subproblem is $\mathcal{O}(k^3)$ at the k-th step. The obvious need to control this cost has motivated the development of restarting schemes.

4.4.1 Implicit Restarting

A restarting alternative has been proposed by Saad based upon the polynomial acceleration scheme developed by Manteuffel [?] for the iterative solution of linear systems. Saad [11] proposed to restart the factorization with a vector that has been preconditioned so that it is more nearly in a k-dimensional invariant subspace of interest. This preconditioning takes the form of a polynomial in $\mathbf{A}$ applied to the starting vector that is constructed to damp unwanted components from the eigenvector expansion. An iteration is defined by a repeatedly restarting until the current Arnoldi factorization contains the desired information. Saad's ideas are closely related to techniques developed for the Lanczos process by Paige [?], Cullum and Donath [?], and Golub and Underwood [?]. The first example of a restarted Lanczos method that we are aware of was proposed by Karush [?].

The ARPACK software is based upon another approach to restarting that offers a more efficient and numerically stable formulation. This approach called *implicit restarting* is a technique for combining the implicitly shifted QR scheme with a k-step Arnoldi or Lanczos factorization to obtain a truncated form of the implicitly shifted QR-iteration. The numerical difficulties and storage problems normally associated with Arnoldi and Lanczos processes are avoided. The algorithm is capable of computing a few (k) eigenvalues with user specified features such as largest real part or largest magnitude using $2nk + \mathcal{O}(k^2)$storage. No auxiliary storage is required. The computed Schur basis vectors for the desired k-dimensional eigen-space are numerically orthogonal to working precision. The suitability of this method for the development of mathematical software stems from this concise and automatic treatment of the primary difficulties with the Arnoldi/Lanczos process.

Implicit restarting provides a means to extract interesting information from large Krylov subspaces while avoiding the storage and numerical difficulties associated with the standard approach. It does this by continually compressing the interesting information into a fixed size k-dimensional subspace. This is accomplished through the implicitly shifted QR mechanism. An Arnoldi factorization of length $m = k + p$

$$\mathbf{A}\mathbf{V}_m = \mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T, \tag{4.4.1}$$

is compressed to a factorization of length k that retains the eigen-information of interest. This is accomplished using QR steps to apply p shifts implicitly. The first stage of this shift process results in

$$\mathbf{A}\mathbf{V}_m^+ = \mathbf{V}_m^+\mathbf{H}_m^+ + \mathbf{f}_m\mathbf{e}_m^T\mathbf{Q}, \tag{4.4.2}$$

where $\mathbf{V}_m^+ = \mathbf{V}_m\mathbf{Q}$, $\mathbf{H}_m^+ = \mathbf{Q}^H\mathbf{H}_m\mathbf{Q}$, and $\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_p$. Each $\mathbf{Q}_j$ is the orthogonal matrix associated with the shift μ_j used during the shifted QR algorithm. Because of the Hessenberg structure of the matrices $\mathbf{Q}_j$, it turns out that the first $k-1$ entries of the vector $\mathbf{e}_m^T\mathbf{Q}$ are zero (i.e. $\mathbf{e}_m^T\mathbf{Q} = (\sigma\mathbf{e}_k^T, \hat{\mathbf{q}}^H)$). This implies that the leading k columns in equation (4.4.2) remain in an Arnoldi relation. Equating the first k columns on both sides of (4.4.2) provides an updated $k-$step Arnoldi factorization

$$\mathbf{A}\mathbf{V}_k^+ = \mathbf{V}_k^+\mathbf{H}_k^+ + \mathbf{f}_k^+\mathbf{e}_k^T, \tag{4.4.3}$$

with an updated residual of the form $\mathbf{f}_k^+ = \mathbf{V}_m^+\mathbf{e}_{k+1}\hat{\beta}_k + \mathbf{f}_m\sigma$. Using this as a starting point it is possible to apply p additional steps of the Arnoldi process to return to the original m-step form.

Each of these shift cycles results in the implicit application of a polynomial in $\mathbf{A}$ of degree p to the starting vector.

$$\mathbf{v}_1 \leftarrow \psi(\mathbf{A})\mathbf{v}_1 \quad \text{with} \quad \psi(\lambda) = \prod_1^p(\lambda - \mu_j). \tag{4.4.4}$$

The roots of this polynomial are the shifts used in the QR process and these may be selected to filter unwanted information from the starting vector and hence from the Arnoldi factorization. Full details may be found in [13].

The choice of shifts and hence construction of the polynomial is motivated by the fact that if the starting vector $\mathbf{v}_1 = \sum_{j=1}^{k} \mathbf{x}_j \gamma_j$ where $\mathbf{A}\mathbf{x}_j = \mathbf{x}_j \lambda_j$, then $\mathbf{f}_k = \mathbf{0}$, $\mathbf{A}\mathbf{V}_k = \mathbf{V}_k \mathbf{H}_k$ and thus $\mathbf{V}_k$ will provide an orthonormal basis for the invariant subspace $\mathcal{S} \equiv Range(\mathbf{V}_k)$. Moreover, the spectrum of $\mathbf{H}_k$ will be the desired eigenvalues: $\sigma(\mathbf{H}_k) = \{\lambda_1, \lambda_2, \cdots, \lambda_k\}$.

Figure 4.4: Algorithm 3: An Implicitly Restarted Arnoldi Method (IRAM).

Input: $(\mathbf{A}, \mathbf{V}, \mathbf{H}, \mathbf{f})$ with $\mathbf{A}\mathbf{V}_m = \mathbf{V}_m \mathbf{H}_m + \mathbf{f}_m \mathbf{e}_m^T$,
an m-Step Arnoldi Factorization;
For $\ell = 1, 2, 3, \ldots$ until *convergence*
(a3.2) Compute $\sigma(\mathbf{H}_m)$ and select set of p shifts $\mu_1, \mu_2, \ldots \mu_p$
based upon $\sigma(\mathbf{H}_m)$ or perhaps other information;
(a3.3) $\mathbf{q}^H \leftarrow \mathbf{e}_m^T$;
(a3.4) For $j = 1, 2, \ldots, p$,
Factor $[\mathbf{Q}, \mathbf{R}] = qr(\mathbf{H}_m - \mu_j \mathbf{I})$;
$\mathbf{H}_m \leftarrow \mathbf{Q}^H \mathbf{H}_m \mathbf{Q}$; $\mathbf{V}_m \leftarrow \mathbf{V}_m \mathbf{Q}$;
$\mathbf{q} \leftarrow \mathbf{q}^H \mathbf{Q}$;
End_For
(a3.5) $\mathbf{f}_k \leftarrow \mathbf{v}_{k+1} \hat{\beta}_k + \mathbf{f}_m \sigma_k$;
(a3.6) $\mathbf{V}_k \leftarrow \mathbf{V}_m(1:n, 1:k)$; $\mathbf{H}_k \leftarrow \mathbf{H}_m(1:k, 1:k)$;
(a3.7) Beginning with the k-step Arnoldi factorization
$\mathbf{A}\mathbf{V}_k = \mathbf{V}_k \mathbf{H}_k + \mathbf{f}_k \mathbf{e}_k^T$,
apply p additional steps of the Arnoldi process
to obtain a new m-step Arnoldi factorization
$\mathbf{A}\mathbf{V}_m = \mathbf{V}_m \mathbf{H}_m + \mathbf{f}_m \mathbf{e}_m^T$.
End_For

The repeated update of the starting vector $\mathbf{v}_1$ through implicit restarting is designed to enhance the components of this vector in the directions of the wanted eigenvectors and damp its components in the unwanted directions. If $\mathbf{v}_1$ has an expansion as a linear combination of eigenvectors $\{\mathbf{x}_j\}$ of $\mathbf{A}$, the effect of this polynomial restarting is illustrated as follows:

$$\mathbf{v}_1 = \sum_{j=1}^{n} \mathbf{x}_j \gamma_j \Rightarrow \psi(\mathbf{A})\mathbf{v}_1 = \sum_{j=1}^{n} \mathbf{x}_j \psi(\lambda_j) \gamma_j.$$

If the same polynomial (i.e. the same shifts) were applied each time, then after ℓ iterations, the j-th original expansion coefficient would be essentially attenuated by a factor

$$\left(\frac{\psi(\lambda_j)}{\psi(\lambda_1)}\right)^{\ell} ,$$

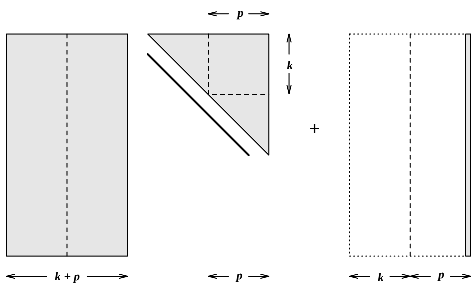

Figure 4.5: The set of rectangles represents the matrix equation $\mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T$ of an Arnoldi factorization. The unshaded region on the right is a zero matrix of $m-1$ columns.

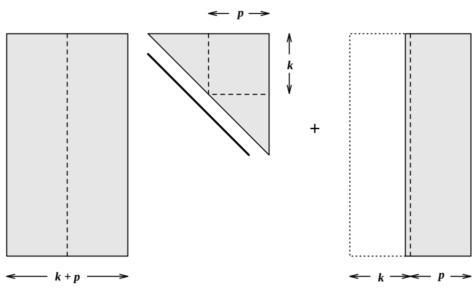

Figure 4.6: After performing $m-k$ implicitly shifted QR steps on $\mathbf{H}_m$, the middle set of pictures illustrates $\mathbf{V}_m\mathbf{Q}_m\mathbf{H}_m^+ + \mathbf{f}_m\mathbf{e}_m^T\mathbf{Q}_m$. The last p columns of $\mathbf{f}_m\mathbf{e}_m^T\mathbf{Q}_m$ are nonzero because of the QR iteration.

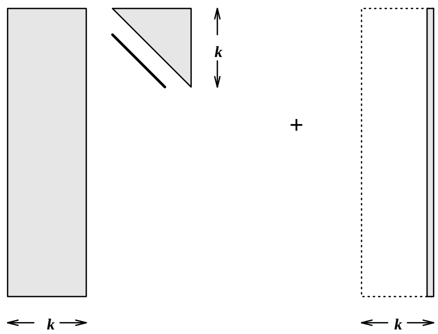

Figure 4.7: An implicitly restarted length k Arnoldi factorization results after discarding the last $m-k$ columns.

where the eigenvalues have been ordered according decreasing values of $|\psi(\lambda_j))|$. The leading k eigenvalues become dominant in this expansion and the remaining eigenvalues become less and less significant as the iteration proceeds. Adaptive choices of shifts can further enhance the isolation of the wanted components in this expansion. Hence, the wanted eigenvalues are approximated better and better as the iteration proceeds.

The basic iteration is listed in Figure 4.4 as Algorithm 3 and the diagrams in Figures 4.5—4.7 describe how this iteration proceeds schematically. In Algorithm 3 and in the discussion that follows, the notation $\mathbf{M}(1:n, 1:k)$ denotes the leading $n \times k$ submatrix of $\mathbf{M}$.

We illustrate a typical polynomial that was constructed during an iteration on a matrix $\mathbf{A}$ that is a small example of a turbine model. The goal was to compute the five eigenvalues of largest real part of this order 375 matrix. The surface shown in Figure 4.8 is the $log(|\psi(\lambda)|)$ plotted over a region containing the spectrum of $\mathbf{A}$. Here, ψ is the product of all of the filter polynomials constructed during the course of the iteration. The degree of ψ is around 600 and this would be quite challenging to apply directly. The "+" signs are the eigenvalues of $\mathbf{A}$ in the complex plane and the contours are the level curves of $|\psi(\lambda)|$. The circled plus signs are the converged eigenvalues including two complex conjugate pairs and on real root of largest real part. Observe how well the the location of the unwanted portion of the spectrum was determined and damped during the course of the iteration. The figure illustrates that this method can isolate desired eigenvalues on the "boundary" of the spectrum even though they may be quite close to other eigenvalues. However, when the clustering becomes pronounced, it will be very difficult to achieve this.

Observe that if $m = n$ then $\mathbf{f}_m = \mathbf{0}$ and this iteration is precisely the same as the Implicitly Shifted QR iteration. Even for $m < n$, the first k columns of $\mathbf{V}_m$ and the Hessenberg submatrix $\mathbf{H}_m(1:k, 1:k)$ are mathematically equivalent to the matrices that would appear in the full Implicitly Shifted QR iteration using the same shifts μ_j. In this sense, the Implicitly Restarted Arnoldi method may be viewed as a truncation of the Implicitly Shifted QR iteration. See [?] for details on a connection with subspace iteration and the QR algorithm. The fundamental difference is that the standard Implicitly Shifted QR iteration selects shifts to drive subdiagonal elements of $\mathbf{H}_n$ to zero from the bottom up while the shift selection in the Implicitly Restarted Arnoldi method is made to drive subdiagonal elements of $\mathbf{H}_m$ to zero from the top down.

Important implementation details concerning the deflation (setting to zero) of subdiagonal elements of $\mathbf{H}_m$ and the purging of unwanted but converged Ritz values are beyond the scope of this discussion. However, these details are extremely important to the success of this iteration in difficult cases. Complete details of these numerical refinements may be found in [7, 6].

The above iteration can be used to apply any known polynomial restart. If the roots of the polynomial are not known there is an alternative implementation that only requires one to compute $\mathbf{q}_1 = \psi(\mathbf{H}_m)\mathbf{e}_1$ where ψ is the desired degree p polynomial. A sequence of Householder transformations may devel-

```
%---------------------------------------------------------%
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%     Picture of Total Filter Polynomial Goes Here        %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%---------------------------------------------------------%
```

Figure 4.8: Total Filter Polynomial From an IRA Iteration.

oped to form a unitary matrix $\mathbf{Q}$ such that $\mathbf{Q}\mathbf{e}_1 = \mathbf{q}_1$ and $\mathbf{H}_m \leftarrow \mathbf{Q}^H\mathbf{H}_m\mathbf{Q}$ is upper Hessenberg. The details which follow standard developments for the Implicitly Shifted QR iteration will be omitted here.

A shift selection strategy that has proved successful in practice and is used as the default in ARPACK is called the "Exact Shift Strategy". In this strategy, one computes $\sigma(\mathbf{H}_m)$ and sorts this into two disjoint sets Ω_w and Ω_u. The k Ritz values in the set Ω_w are regarded as approximations to the "wanted" eigenvalues of $\mathbf{A}$, and the p Ritz values in the set Ω_u are taken as the shifts μ_j. An interesting consequence (in exact arithmetic) is that after Step (a3.4) above, the spectrum of $\mathbf{H}_k$ in Step (a3.6) is $\sigma(\mathbf{H}_k) = \Omega_w$ and the updated starting vector $\mathbf{v}_1$ is a particular linear combination of the k Ritz vectors associated with these Ritz values. In other words, the implicit restarting scheme with exact shifts provides a specific selection of expansion coefficients γ_j for a new starting vector as a linear combination of the current best estimates (the Ritz vectors) for wanted eigenvectors. This implicit scheme costs p rather than the $k+p$ matrix-vector products the explicit scheme would require. Thus the exact shift strategy can be viewed both as a means to damp unwanted components from the starting vector and also as directly forcing the starting vector to be a linear combination of wanted eigenvectors. See [**?**, 13] for information on the convergence of IRAM and [**?**, **?**] for other possible shift strategies for Hermitian $\mathbf{A}$. The reader is referred to [**?**, **?**] for studies comparing implicit restarting with other schemes.

4.4.2 Block Methods

Implicit restarting can also be used in conjunction with a block Arnoldi method. A block method attempts to use the block Krylov subspace

$$\mathcal{K}_k(\mathbf{A}, \mathbf{U}_1) = \text{Span}\ \{\mathbf{U}_1, \mathbf{A}\mathbf{U}_1, \mathbf{A}^2\mathbf{U}_1, \ldots, \mathbf{A}^{k-1}\mathbf{U}_1\},$$

where $\mathbf{U}_1$ has b columns, to approximate eigenvalues and eigenvectors. A block Arnoldi reduction uses a subspace drawn from a sequence of subspace iterates. The details associated with implicitly restarting a block Arnoldi reduction are laid out in [**?**, **?**].

Block methods are used for two major reasons. The first one is to aid in reliably determining multiple and/or clustered eigenvalues. Although [7] indicates that an unblocked Arnoldi method coupled with an appropriate deflation strategy may be used to compute multiple and/or clustered eigenvalues, a relatively small convergence tolerance is required to reliably compute clustered eigenvalues. Many problems do not require this much accuracy, and such a criterion can result in unnecessary computation. The second reason for using a block formulation is related to computational efficiency. Often, when a matrix-vector product with $\mathbf{A}$ is very costly, it is possible to compute the action of $\mathbf{A}$ on several vectors at once with roughly the same cost as computing a single matrix-vector product. This can happen, for example, when the matrix is so large that it must be retrieved from disk each time a matrix-vector product is performed. In this situation, a block method may have considerable advantages.

The performance tradeoffs of block methods and potential improvements to deflation techniques are under investigation.

4.5 The Generalized Eigenvalue Problem

A typical source of large scale eigenproblems is through a discrete form of a continuous problem. The resulting finite dimensional problems become large due to accuracy requirements and spatial dimensionality. Typically this takes the form

$$\mathbf{A}\mathbf{x} \;=\; \mathbf{M}\mathbf{x}\lambda,$$

where $\mathbf{A}$ is a finite dimensional approximation to the continuous operator obtained either through finite difference approximations on a spatial grid or through restriction of the continuous operator to a finite dimensional subspace. In the latter "finite element" case, the entries of $\mathbf{M}$ are inner products of the respective basis functions for the finite dimensional space and these basis functions are usually chosen so that few entries in a typical row of $\mathbf{A}$ or $\mathbf{M}$ are nonzero. In structures problems $\mathbf{A}$ is called the "stiffness" matrix and $\mathbf{M}$ is called the "mass" matrix. In chemistry and physics $\mathbf{M}$ is often referred to as the "overlap" matrix. A nice feature of finite element approach to discretization is that boundary conditions are naturally incorporated into the discrete problem. Moreover, in the self-adjoint case, the Rayleigh principle is preserved from the continuous to the discrete problem. In particular, since Ritz values are Rayleigh quotients, this assures the smallest Ritz value is greater than or equal to the smallest eigenvalue of the original problem.

Basis functions that provide sparsity are usually not orthogonal in the natural inner product and hence, $\mathbf{M}$ is usually not diagonal. Thus it is typical for large scale eigenproblems to arise as generalized rather than standard problems with $\mathbf{M}$ symmetric and positive semi-definite. The matrix $\mathbf{A}$ is generally symmetric when the underlying continuous operator is self-adjoint and nonsymmetric otherwise. There are a number of ways to convert the generalized problem to standard form. There is always motivation to preserve symmetry when it is present.

The simplest direct conversion to a standard problem is through factorization of $\mathbf{M}$. If $\mathbf{M}$ is positive definite then factor $\mathbf{M} = \mathbf{L}\mathbf{L}^H$ and the eigenvalues of $\hat{\mathbf{A}} \equiv \mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-H}$ are the eigenvalues of $(\mathbf{A}, \mathbf{M})$ and the eigenvectors are obtained by solving $\mathbf{L}^H\mathbf{x} = \hat{\mathbf{x}}$ where $\hat{\mathbf{x}}$ is an eigenvector of $\hat{\mathbf{A}}$. This standard transformation is fine if one wants the eigenvalues of largest magnitude and it preserves symmetry if $\mathbf{A}$ is symmetric. However, when $\mathbf{M}$ is ill-conditioned this can be a dangerous transformation leading to numerical difficulties. Since a matrix factorization will have to be done anyway, one may as well formulate a spectral transformation.

4.5.1 Structure of the Spectral Transformation

The use of spectral transformations is closely related to the inverse power and inverse iteration techniques. They are typically designed to enhance convergence to eigenvalues near a selected point σ in the complex plane. For example, if one were interested in computing the smallest eigenvalues of a symmetric positive definite matrix then $\sigma = 0$ would be a reasonable choice.

A convenient way to provide such a spectral transformation is to note that

$$\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{x}\lambda \iff (\mathbf{A} - \sigma\mathbf{M})\mathbf{x} = \mathbf{M}\mathbf{x}(\lambda - \sigma)$$

Thus

$$(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}\mathbf{x} = \mathbf{x}\nu, \quad \text{where} \quad \nu = \frac{1}{\lambda - \sigma}.$$

A moments reflection will reveal the advantage of such a spectral transformation. Eigenvalues λ that are near σ will be transformed to eigenvalues $\nu = \frac{1}{\lambda - \sigma}$ that are at the extremes and typically well separated from the rest of the transformed spectrum. The corresponding eigenvectors remain unchanged. Perhaps more important is the fact that eigenvalues far from the shift σ are mapped into a tight cluster in the interior of the transformed spectrum. We illustrate this by showing the transformed spectrum of the matrix $\mathbf{A}$ from Figure 4.8 with a shift $\sigma = .2$ (here $\mathbf{M} = \mathbf{I}$). Again, we show the total filter polynomial that was constructed during an IRA iteration on the transformed matrix $(\mathbf{A} - \sigma I)^{-1}$. Here we compute the six eigenvalues ν of largest magnitude. These will transform back to eigenvalues of $\mathbf{A}$ nearest to σ through the formula $\lambda = \sigma + 1/\nu$. The surface shown in Figure 4.9 is again $log(|\psi(\lambda)|)$ but plotted over a region containing the spectrum of $(\mathbf{A} - \sigma I)^{-1}$. Here, ψ is the product of all of the filter polynomials constructed during the course of the iteration. Since the extrem eigenvalues are well separated the iteration converges much faster and degree of ψ is only 45 in this case. Here, the "+" signs are the eigenvalues of $(\mathbf{A} - \sigma I)^{-1}$ in the complex plane and the contours are the level curves of $|\psi(\lambda)|$. The circled plus signs are the converged eigenvalues. The figure illustrates how much easier it is to isolate desired eigenvalues after a spectral transformation.

If $\mathbf{A}$ is symmetric then one can maintain symmetry in the Arnoldi/Lanczos process by taking the inner product to be

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^H \mathbf{M}\mathbf{y}.$$

It is easy to verify that the operator $(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ is symmetric with respect to this inner product if $\mathbf{A}$ is symmetric. In the Arnoldi/Lanczos process the matrix-vector product $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$ is replaced by $\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}\mathbf{v}$ and the step $\mathbf{h} \leftarrow \mathbf{V}^H\mathbf{f}$ is replaced by $\mathbf{h} \leftarrow \mathbf{V}^H(\mathbf{M}\mathbf{f})$. If $\mathbf{A}$ is symmetric then the matrix $\mathbf{H}$ is symmetric and tridiagonal. Moreover, this process is well defined even when $\mathbf{M}$ is singular and this can have important consequences even if $\mathbf{A}$ is non-symmetric. We shall refer to this process as the $\mathbf{M}$-Arnoldi process.

```
%---------------------------------------------------------%
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%     Picture of Total Filter Polynomial Goes Here        %
%                                                         %
%             Shift Invert                                %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%                                                         %
%---------------------------------------------------------%
```

Figure 4.9: Total Filter Polynomial with Spectral Transformation

If $\mathbf{M}$ is singular then the operator $\mathbf{S} \equiv (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ has a non-trivial null space and the bilinear function $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^H\mathbf{M}\mathbf{y}$ is a semi-inner product and $\|\mathbf{x}\|_M \equiv \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$ is a semi-norm. Since $\mathbf{A} - \sigma\mathbf{M}$ is assumed to be nonsingular, $\mathcal{N} \equiv Null(\mathbf{S}) = Null(\mathbf{M})$. Vectors in $\mathcal{N}$ are generalized eigenvectors corresponding to *infinite* eigenvalues. Typically, one is only interested in the finite eigenvalues of $(\mathbf{A}, \mathbf{M})$ and these will correspond to the non-zero eigenvalues of S. The invariant subspace corresponding to these non-zero eigenvalues is easily corrupted by components of vectors from $\mathcal{N}$ during the Arnoldi process. However, using the M-Arnoldi process with some refinements can provide a solution.

In order to better understand the situation, it is convenient to note that since $\mathbf{M}$ is positive semi-definite, there is an orthogonal matrix $\mathbf{Q}$ such that

$$\mathbf{Q}^H\mathbf{M}\mathbf{Q} = \begin{bmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $\mathbf{D}$ is a positive definite diagonal matrix of order n, say. Thus

$$\hat{\mathbf{S}} \equiv \mathbf{Q}^H\mathbf{S}\mathbf{Q} = \begin{bmatrix} \mathbf{S}_1 & \mathbf{0} \\ \mathbf{S}_2 & \mathbf{0} \end{bmatrix},$$

where $\mathbf{S}_1$ is a square matrix of order n and $\mathbf{S}_2$ is an $m \times n$ matrix with the original $\mathbf{A}, \mathbf{M}$ being of order $m + n$. Observe now that a non-zero eigenvalue ν of $\hat{\mathbf{S}}$ satisfies $\hat{\mathbf{S}}\mathbf{x} = \mathbf{x}\nu$, i.e.

$$\begin{bmatrix} \mathbf{S}_1\mathbf{x}_1 \\ \mathbf{S}_2\mathbf{x}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1\theta \\ \mathbf{x}_2\theta \end{bmatrix}$$

so that $\mathbf{x}_2 = \theta^{-1}\mathbf{S}_2\mathbf{x}_1$ must hold. Note also that for any eigenvector $\mathbf{x}^H = (\mathbf{x}_1^H, \mathbf{x}_2^H)$, the leading vector $\mathbf{x}_1$ must be an eigenvector of $\mathbf{S}_1$. Since $\hat{\mathbf{S}}$ is block triangular, $\sigma(\hat{\mathbf{S}}) = \sigma(\mathbf{S}_1) \cup \sigma(\mathbf{0}_m)$. Assuming $\mathbf{S}_2$ has full rank, it follows that if $\mathbf{S}_1$ has a zero eigenvalue then there is no corresponding eigenvector (since $\mathbf{S}_2\mathbf{x}_1 = \mathbf{0}$ would be implied). Thus, if zero is an eigenvalue of $\mathbf{S}_1$ with algebraic multiplicity m_o, then zero is an eigenvalue of $\hat{\mathbf{S}}$ of algebraic multiplicity $m+m_o$ and with geometric multiplicity m. Of course, since $\mathbf{S}$ is similar to $\hat{\mathbf{S}}$ all of these statements hold for $\mathbf{S}$ as well.

4.5.2 Eigenvector/Null-Space Purification

With these observations in hand, it is possible to see the virtue of using M-Arnoldi on $\mathbf{S}$. After k-steps of M-Arnoldi,

$$\mathbf{S}\mathbf{V} = \mathbf{V}\mathbf{H} + \mathbf{f}\mathbf{e}_k^T \quad \text{with} \quad \mathbf{V}^H\mathbf{M}\mathbf{V} = \mathbf{I}_k, \mathbf{V}^H\mathbf{M}\mathbf{f} = \mathbf{0}.$$

Introducing the similarity transformation $\mathbf{Q}$ gives

$$\hat{\mathbf{S}}\hat{\mathbf{V}} = \hat{\mathbf{V}}\mathbf{H} + \hat{\mathbf{f}}\mathbf{e}_k^T \quad \text{with} \quad (\mathbf{Q}\hat{\mathbf{V}})^H\mathbf{M}\mathbf{Q}\hat{\mathbf{V}} = \mathbf{I}_k, (\mathbf{Q}\hat{\mathbf{V}})^H\mathbf{M}\mathbf{Q}\hat{\mathbf{f}} = \mathbf{0},$$

where $\hat{\mathbf{V}} = \mathbf{Q}^H\mathbf{V}$ and $\hat{\mathbf{f}} = \mathbf{Q}^H\mathbf{f}$. Partitioning $\hat{\mathbf{V}}^H = [\ \mathbf{V}_1^H\ \mathbf{V}_2^H\ ]$ and $\hat{\mathbf{f}}^H = [\ \mathbf{f}_1^H\ \mathbf{f}_2^H\ ]$ consistent with the blocking of $\hat{\mathbf{S}}$ gives

$$\mathbf{S}_1\mathbf{V}_1 = \mathbf{V}_1\mathbf{H} + \mathbf{f}_1\mathbf{e}_k^T \quad \text{with} \quad \mathbf{V}_1^H\mathbf{D}\mathbf{V}_1 = \mathbf{I}_k, \mathbf{V}_1^H\mathbf{D}\mathbf{f}_1 = \mathbf{0}.$$

Moreover, the side condition $\mathbf{S}_2\mathbf{V}_1 = \mathbf{V}_2\mathbf{H} + \mathbf{f}_2\mathbf{e}_k^T$ holds, so that in exact arithmetic a zero eigenvalue should not appear as a converged Ritz value of $\mathbf{H}$. This argument shows that $\mathbf{M}$-Arnoldi on $\mathbf{S}$ is at the same time doing $\mathbf{D}$-Arnoldi on $\mathbf{S}_1$ while avoiding convergence to zero eigenvalues.

Round-off error due to finite precision arithmetic will cloud the situation, as usual. It is clear that the goal is to prevent components in $\mathcal{N}$ from corrupting the vectors $\mathbf{V}$. Thus to begin, the starting vector $\mathbf{v}_1$ should be of the form $\mathbf{v}_1 = \mathbf{S}\mathbf{v}$. If a final approximate eigenvector $\mathbf{x}$ has components in $\mathcal{N}$ they may be purged by replacing $\mathbf{x} \leftarrow \mathbf{S}\mathbf{x}$ and then normalizing. To see the effect of this, note that

$$\mathbf{x} = \mathbf{Q}\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \quad \text{implies} \quad \mathbf{S}\mathbf{x} = \mathbf{Q}\begin{bmatrix} \mathbf{S}_1\mathbf{x}_1 \\ \mathbf{S}_2\mathbf{x}_1 \end{bmatrix}$$

and all components in $\mathcal{N}$ which are of the form $\mathbf{Q}\begin{bmatrix} \mathbf{0} \\ \mathbf{p} \end{bmatrix}$ will have been purged. This final application of $\mathbf{S}$ may be done implicitly in two ways. One is to note that if $\mathbf{x} = \mathbf{V}\mathbf{s}$ with $\mathbf{H}\mathbf{s} = \mathbf{s}\theta$ then $\mathbf{S}\mathbf{x} = \mathbf{V}\mathbf{H}\mathbf{s} + \mathbf{f}\mathbf{e}_k^T\mathbf{s} = \mathbf{x}\theta + \mathbf{f}\mathbf{e}_k^T\mathbf{s}$ and the approximate eigenvector $\mathbf{x}$ is replaced with the improved approximation $\mathbf{x} \leftarrow (\mathbf{x}\theta + \mathbf{f}\mathbf{e}_k^T\mathbf{s})/\tau$ where $\tau = \|\mathbf{x}\theta + \mathbf{f}\mathbf{e}_k^T\mathbf{s}\|_{\mathbf{M}} = \sqrt{\theta^2 + (\beta_k\mathbf{e}_k^T\mathbf{s})^2}$. This correction was originally suggested by Ericsson and Ruhe [?] as a mean of performing a formal step of the power method with $\mathbf{S}$. The residual error of the computed Ritz vector with respect to the original problem is

$$\|\mathbf{A}\mathbf{x} - \mathbf{M}\mathbf{x}\lambda\| = \|\mathbf{M}\mathbf{f}\|\frac{|\mathbf{e}_k^T\mathbf{s}|}{|\theta|^2} \tag{4.5.1}$$

where $\lambda = \sigma + 1/\theta$. Keeping in mind that under the spectral transformation $|\theta|$ is usually quite large, this estimate indicates even greater accuracy than expected from the usual estimate. This is the purification used in ARPACK.

Another recent suggestion due to Meerbergen and Spence is to use implicit restarting with a zero shift [?]. Recall that implicit restarting with ℓ zero shifts is equivalent to starting the $\mathbf{M}$-Arnoldi process with a starting vector of $\mathbf{S}^\ell\mathbf{v}_1$ and all the resulting Ritz vectors will be multiplied by $\mathbf{S}^\ell$ as well. After applying the implicit shifts to $\mathbf{H}$, the leading submatrix of order $k - \ell$ will provide the updated Ritz values. No additional explicit matrix-vector products with $\mathbf{S}$ are required.

The ability to apply ℓ zero shifts (i.e. to multiply by $\mathbf{S}^\ell$ implicitly) is very important when $\mathbf{S}_1$ has zero eigenvalues. If $\mathbf{S}_1\mathbf{x}_1 = \mathbf{0}$ then

$$\begin{bmatrix} \mathbf{S}_1 & \mathbf{0} \\ \mathbf{S}_2 & \mathbf{0} \end{bmatrix}\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{S}_2\mathbf{x}_1 \end{bmatrix} \in \mathcal{N}.$$

Thus, in order to completely eradicate components from $\mathcal{N}$ one must multiply by $\mathbf{S}^{\ell}$ where ℓ is equal to the dimension of the largest Jordan block corresponding to a zero eigenvalue of $\mathbf{S}_1$. Eigenvector purification by implicit restarting may be incorporated into ARPACK at a future date.

Spectral transformations were studied extensively by Ericsson and Ruhe [?] and the first eigenvector purification strategy was developed in [?]. Shift and invert techniques play an essential role in the block Lanczos code developed by Grimes, Lewis, and Simon and the many nuances of this technique in practical applications are discussed thoroughly in [4]. The development presented here and the eigenvector purification through implicit restarting is due to Meerbergen and Spence [?].

4.6 Stopping Criterion

This section considers the important question of determining when a length m Arnoldi factorization has computed approximate eigenvalues of acceptable accuracy.

Let $\mathbf{H}_m\mathbf{s} = \mathbf{s}\theta$ where $\|\mathbf{s}\| = 1$ and $\hat{\mathbf{x}} \equiv \mathbf{V}_m\mathbf{s}$. Then

$$\|\mathbf{A}\hat{\mathbf{x}} - \hat{\mathbf{x}}\theta\| = \|\mathbf{A}\mathbf{V}_m\mathbf{s} - \mathbf{V}_m\mathbf{H}_m\mathbf{s}\| = \|\mathbf{f}_m\|\,|\mathbf{e}_m^T\mathbf{s}|, \tag{4.6.1}$$

suggests that the Ritz pair $(\hat{\mathbf{x}}, \theta)$ is a good approximation to an eigenpair of $\mathbf{A}$ if the last component of an eigenvector for $\mathbf{H}_m$ is small. If the upper Hessenberg matrix $\mathbf{H}$ is unreduced (has no zero subdiagonal elements) then standard results imply that $|\mathbf{e}_m^T\mathbf{s}| \neq 0$. However, this quantity can be quite small even if all of the subdiagonal element of $\mathbf{H}$ are far from zero. Usually, this is how convergence takes place, but it is also possible for $\mathbf{f}_m$ to become small. If the quantity $\|\mathbf{f}_m\|$ is small enough, then all m eigenvalues of $\mathbf{H}_m$ are likely to be good approximations to m eigenvalues of $\mathbf{A}$. In the Hermitian case, this estimate on the residual can be turned into a precise statement about the accuracy of the Ritz value θ as an approximation to the eigenvalue of A that is nearest to θ. However, an analogous statement in the non-Hermitian case is not possible without further information concerning non-normality and defectiveness.

We shall develop a crude but effective assessment of accuracy based upon this estimate. Far more sophisticated analysis is available for the symmetric problem in [?] and in [12] for the non-symmetric case. It is easily shown that

$$(\mathbf{A} + \mathbf{E})\hat{\mathbf{x}} = \hat{\mathbf{x}}\theta, \quad \text{with} \quad \mathbf{E} \equiv -(\mathbf{e}_m^T\mathbf{s})\mathbf{f}_m\hat{\mathbf{x}}^H. \tag{4.6.2}$$

Thus, the Ritz pair $(\hat{\mathbf{x}}, \theta)$ is exact eigenpair for a nearby problem whenever $\|\mathbf{f}_m\|$ or $|\mathbf{e}_m^T\mathbf{s}|$ is small (relative to $\|\mathbf{A}\|$). The advantage of using the Ritz estimate $\|\mathbf{f}_m\|\,|\mathbf{e}_m^T\mathbf{s}|$ is to avoid explicit formation of the direct residual $\mathbf{A}\hat{\mathbf{x}} - \hat{\mathbf{x}}\theta$ when accessing the numerical accuracy of an approximate eigenpair.

In the Hermitian case a small residual implies an accurate answer. However, in the non-Hermitian case, a small $\|\mathbf{E}\|$ does not necessarily imply that the

Ritz pair $(\hat{\mathbf{x}}, \theta)$ is an accurate approximation to an eigenpair $(\mathbf{x}, \lambda)$ of $\mathbf{A}$. The following theorem indicates what accuracy might be expected of a Ritz value as an approximation to an eigenvalue of $\mathbf{A}$.

Theorem 4.6.1 *Suppose that λ is a simple eigenvalue of $\mathbf{A}$ nearest the eigenvalue θ of $\mathbf{A}+\mathbf{E}$. Denote the left and right eigenvectors for λ by $\mathbf{y}$ and $\mathbf{x}$, respectively, each of unit length. Then*

$$|\lambda - \theta| \;\; \leq \;\; \frac{\|\mathbf{E}\|}{|\mathbf{y}^H\mathbf{x}|} + O(\|\mathbf{E}\|^2)$$

Proof: See Wilkinson [**?**, p. 68].

The number $|\mathbf{y}^H\mathbf{x}|$ is the cosine of the angle between $\mathbf{y}$ and $\mathbf{x}$ and it determines the conditioning of λ. Thus, if $\mathbf{y}$ and $\mathbf{x}$ are nearly orthogonal, the eigenvalue λ is highly sensitive to perturbations in $\mathbf{A}$ but if they are nearly parallel then λ is insensitive to perturbations. For Hermitian matrices $\mathbf{y}=\mathbf{x}$ so that $|\mathbf{y}^H\mathbf{x}| = 1$ and θ will be an excellent approximation to λ. However, if the left and right eigenvectors are nearly orthogonal, then even if $\|\mathbf{E}\| \approx \epsilon_M\|\mathbf{A}\|$, where ϵ_M is machine precision, θ may contain few digits, if any, of accuracy. Roughly, as a rule of thumb, if $|\mathbf{y}^H\mathbf{x}| \approx 10^{-d}$ and $\epsilon_M \approx 10^{-t}$ then the leading $t-d$ decimal digits of θ will agree with those of λ.

The question of how close the Ritz vector $\hat{\mathbf{x}}$ is to $\mathbf{x}$ is complicated by the fact that an eigenvector is not a unique quantity. Any scaling of an eigenvector by a nonzero complex number is also an eigenvector. For this reason, it is better to estimate the positive angle φ between an eigenvector and its approximation.

Theorem 4.6.2 *Suppose that* $\mathbf{AQ} = \mathbf{Q}\begin{bmatrix} \lambda & \mathbf{r}_{12}^H \\ \mathbf{0} & \mathbf{R}_{22} \end{bmatrix}$ *is a Schur form for $\mathbf{A}$ and let λ be a simple eigenvalue of $\mathbf{A}$ nearest the eigenvalue θ of $\mathbf{A}+\mathbf{E}$ with corresponding eigenvectors $\mathbf{x}$ and $\hat{\mathbf{x}}$ respectively. If φ is the positive angle between $\mathbf{x}$ and $\hat{\mathbf{x}}$ then*

$$\varphi \;\; \leq \;\; \frac{2\|\mathbf{E}\|_F}{sep(\lambda, \mathbf{R}_{22})} + O(\|\mathbf{E}\|_F^2).$$

Proof: See § 5 in [**?**].

The definition of the quantity *sep* in this theorem is

$$\mathrm{sep}(\lambda, \mathbf{R}_{22}) \equiv \min_{\mathbf{z}\neq 0} \frac{\|\lambda\mathbf{z}^H - \mathbf{z}^H\mathbf{R}_{22}\|_F}{\|\mathbf{z}\|_F},$$

and the norm $\|\mathbf{E}\|_F = (trace\ \mathbf{E}^H\mathbf{E})^{\frac{1}{2}}$ is the Frobenius norm.

This is a more refined indicator of eigenvector sensitivity that accounts for non-normality as well as clustering of eigenvalues. Varah [**?**] shows that

$$\mathrm{sep}(\lambda, \mathbf{R}_{22}) \;\; \leq \;\; \min_{\lambda_i \neq \lambda, \lambda_i \in \sigma(\mathbf{R}_{22})} |\lambda - \lambda_i|,$$

$$\mathrm{sep}(\lambda, \mathbf{R}_{22}) \;\; \leq \;\; \|\mathbf{r}_{12}\| \frac{|\mathbf{y}^H\mathbf{x}|}{\sqrt{1 - |\mathbf{y}^H\mathbf{x}|^2}}, \tag{4.6.3}$$

where the latter bound is only defined for nonzero $\mathbf{r}_{12}$. Thus, the conditioning of the eigenvector problem depends upon both the distance to the other eigenvalues of $\mathbf{A}$ and the sensitivity of λ. Varah also notes that both upper bounds may be significant over estimates. Note that when $\mathbf{A}$ is symmetric, $\mathbf{r}_{12} = \mathbf{0}$ and it may be shown that the first bound is an equality. Multiple eigenvalues or clusters of eigenvalues cause further complications. The above estimates may be extended to these cases through the conditioning of invariant subspaces and angles between invariant subspaces.

The conclusion we must draw is that the eigenvalues of a non-symmetric matrix may be very sensitive to perturbations such as those introduced by round-off error. This sensitivity is intricately tied to the departure from normality of the given matrix. The classic example of a matrix with an extremely ill-conditioned eigensystem is $\mathbf{J}_m(\lambda) + \epsilon \mathbf{e}_m \mathbf{e}_1^T$ where $\mathbf{J}_m(\lambda)$ is bi-diagonal with the number λ on the diagonal and all ones on the super-diagonal. The eigenvalues θ_j of this perturbed Jordan matrix satisfy $|\theta_j - \lambda| = \epsilon^{\frac{1}{m}}$. This is quite contrary to the behavior of eigenvalues of normal matrices. If the matrix $\mathbf{A}$ is near to a matrix with such an ill-conditioned eigensystem, then we can say little about the accuracy of the computed eigenvalues and eigenvectors. The best we can say is that we have computed the exact eigenvalues and eigenvectors of a nearby matrix (or matrix pencil).

We must be content with a stopping criterion that assures small backward error. This strategy is used in ARPACK, where the Ritz pair $(\hat{\mathbf{x}}, \theta)$ is considered converged when

$$\|\mathbf{f}_m\| \, |\mathbf{e}_m^T \mathbf{s}| \leq \max(\epsilon_M \|\mathbf{H}_m\|, \texttt{tol} \cdot |\theta|)$$

is satisfied where ϵ_M is machine precision. d. Since $|\theta| \leq \|\mathbf{H}_m\| \leq \|\mathbf{A}\|$, this implies that (4.6.2) is satisfied with $\|E\| \leq \texttt{tol}\|A\|$ and this test is invariant to scaling of $\mathbf{A}$ through multiplication by a nonzero scalar.

The backward error is defined as the smallest, in norm, perturbation $\Delta\mathbf{A}$ such that the Ritz pair is an eigenpair for $\mathbf{A} + \Delta\mathbf{A}$. The recent study [**?**] presents a thorough discussion of the many issues involved in determining stopping criteria for the non-symmetric eigenvalue problem. In ARPACK we are more stringent than just asking for a small backward error relative to $\|\mathbf{A}\|$. We instead ask for small backward error relative to the projected matrix $\mathbf{H}_m$.

Chapter 5

Computational Routines

This chapter will discuss the implementation details of the main computational routines of ARPACK. We first give an outline of the code structure. This shows how the Implicitly Restarted Arnoldi/Lanczos Method that was described in Algorithm 3 of Chapter 4 is modularized. Each of the basic steps described in the algorithm defines a subroutine module, i.e. a computational routine. The basic tasks and salient implementation details are presented here for each of these computational routines.

ARPACK relies heavily upon a number of basic operations and algorithms provided by the BLAS and LAPACK. These have contributed greatly to the robustness, accuracy, and computational performance of ARPACK. The most important of these with respect to performance is the BLAS subroutine `_gemv`. For a fixed number `nev` of requested eigenvalues and a fixed length `ncv` Arnoldi basis, the computational cost scales linearly with `n`, the order of the matrix. The rate of execution (in FLOPS) for the IRA iteration is asymptotic to the rate of execution of `_gemv`.

In the outline of the implementation described in Figure 5.1, $\mathbf{H}_j$ is a $j \times j$ upper Hessenberg matrix, $\mathbf{V}_j^H \mathbf{B} \mathbf{V}_j = \mathbf{I}_j$, and the residual vector $\mathbf{f}_j$ is $\mathbf{B}$-orthogonal to the columns of $\mathbf{V}_j$. For each j,

$$\mathtt{OP}\mathbf{V}_j = \mathbf{V}_j \mathbf{H}_j + \mathbf{f}_j \mathbf{e}_j^T$$

where `OP` and $\mathbf{B}$ are defined with respect to one of the computational modes described in Chapter 3.

The integer k denotes the desired number of eigenvalue approximations and this may, at times, be greater than the users request for `nev` approximations. The integer $m =$ `ncv` will be the largest size factorization constructed. An eigenvalue θ of $\mathbf{H}_j$ is a Ritz value of `OP` and $\mathbf{x} = \mathbf{V}_j \mathbf{s}$ is the corresponding Ritz vector when $\mathbf{H}_j \mathbf{s} = \mathbf{s}\theta$. (A formal definition of Ritz value and vector is given in Chap. 4). The normalization $\|\mathbf{s}\| = 1$ is assumed throughout.

Figure 5.1: `XYaupd` – Implementation of the IRAM/IRLM in ARPACK

- Perform basic error checks, partition the internal workspace. Set $k =$ `nev` and $m =$ `ncv`.
- Enter `XYaup2`. Generate a random initial vector $\mathbf{V}_m\mathbf{e}_1 = \mathbf{v}_1$ by calling `Xgetv0`, unless the initial vector is provided by the caller.
- Call `XYaitr` to compute the initial Arnoldi/Lanczos factorization `OP`$\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T$ of length $k =$ `nev`.
- For `iter` $= 1, \ldots,$ `maxiter` $+1$,
  1. Call `XYaitr` to extend the length k Arnoldi/Lanczos factorization to a length m factorization. Reverse communication is performed to compute matrix vector products with `OP` and possibly $\mathbf{B}$.
  2. Compute the eigenvalues of $\mathbf{H}_m$ and the associated error bounds. Call `Xseigt` for the symmetric eigenvalue problem or `Xneigh` otherwise.
  3. Call `XYgets` to partition the eigenvalues into two sets Ω_w and Ω_u. The k eigenvalues in the set Ω_w are the desired approximations while the remaining eigenvalues in the set Ω_u are to be used as shifts.
  4. Call `XYconv` to determine how many of the wanted Ritz values satisfy the convergence tolerance.
  5. Exit the loop if all k eigenvalues in Ω_w satisfy the convergence criterion, or if `iter`$>$ `maxiter`.
  6. Possibly increment k. Determine $p = m - k$ shifts $\{\mu_j\}_{j=1}^p$. If the exact shift strategy is used, the eigenvalues of Ω_u are used as shifts otherwise the p shifts are provided through a reverse communication interface. If $p = 0$, exit the loop.
  7. Implicitly restart the Arnoldi/Lanczos factorization by calling `XYapps`.
     (a) Perform p steps of the implicitly shifted QR algorithm on $\mathbf{H}_m$ with shifts $\{\mu_j\}_{j=1}^p$ to get $\mathbf{H}_m\mathbf{Q}_m = \mathbf{Q}_m\mathbf{H}_m^+$ where $\mathbf{H}_m^+$ and $\mathbf{Q}_m$ are upper Hessenberg and orthogonal matrices respectively.
     (b) Let $\mathbf{Q}_k$ denote the matrix consisting of the first k columns of $\mathbf{Q}_m$ and $\mathbf{H}_k^+$ the leading principal submatrix of $\mathbf{H}_m^+$ of order k. Update $\mathbf{V}_k^+ = \mathbf{V}_m\mathbf{Q}_k$ to get the new length k Arnoldi factorization `OP`$\mathbf{V}_m^+ = \mathbf{V}_m^+\mathbf{H}_k^+ + \mathbf{f}_k^+\mathbf{e}_k^T$, (See Algorithm 3, Chap. 4).
- End For
- Call `XYeupd` for computing Ritz and/or Schur vectors and for transforming the Ritz values of `OP` if a spectral transformation was used.

5.1 ARPACK subroutines

Table 5.1 lists all the auxiliary subroutines of ARPACK. The naming convention is `XY<mnemonic>` The first letter `X` is one of the letter `s,d,c,z` and denotes the data type as follows:

`s`	single precision real arithmetic,
`d`	double precision real arithmetic,
`c`	single precision complex arithmetic,
`z`	double precision complex arithmetic.

If the second letter `Y` occurs, it is one of the letters `s, n` that denote the type of eigensystem as follows:

`n`	non-symmetric,
`s`	symmetric.

For the subroutines listed in Table 5.1 that start with "`[X1,X2]`", only those data types are available.

5.1.1 XYaupd

The top level subroutine `XYaupd` provides the reverse communication interface to IRAM or IRLM. The user directly calls subroutine `XYaupd` in one of the reverse communication modes to compute the eigenvalues of the linear operator defined by `OP` required for the computational mode selected by the user. Every time an operation involving `OP` and/or **B** is needed, `XYaupd` prompts the user to provide the action of `OP` and/or **B** on a vector and then re-enter `XYaupd`. This is the reverse communication interface.

During the initial call to subroutine `XYaupd`, error checking is performed on many of the input arguments. The workspace provided to `XYaupd` is also partitioned and various counters and pointers are initialized.

5.1.2 XYaup2

Subroutine `XYaup2` implements Algorithm 2 (see Figure 4.3) in Chapter 4. The decision to terminate the IRAM/IRLM is made in `XYaup2`. The iteration is terminated either when the requested number of wanted Ritz values satisfy the convergence criterion or when the specified maximum number of iterations has been exceeded. For both situations `XYaup2` is exited with a complete length m Arnoldi/Lanczos factorization and the number of Ritz values that satisfy the convergence criterion is stored. This allows the user to call `XYeupd` and compute the eigenvalues and eigenvectors or Schur vectors that have converged to within the requested tolerance.

At step 6, k may be increased for one of three reasons. The primary reason occurs when the exact shift strategy is used. In this case, the number of shifts to apply is decreased by 1 for every wanted Ritz value in Ω_w that satisfies the convergence criterion. Since $p = m - k$ this is most easily achieved

Table 5.1: Description of the auxiliary subroutines of ARPACK.

ROUTINE	DESCRIPTION
`XYaupd`	Top level subroutine that implements the IRAM.
`XYeupd`	This routine computes eigenvectors and/or Schur vectors for the computed eigenvalues.
`XYaup2`	Intermediate level interface called by `XYaupd` that performs the iteration.
`Xgetv0`	Initial vector generation subroutine.
`XYaitr`	Arnoldi factorization subroutine.
`Xneigh`	Compute Ritz values and error bounds subroutine for the non-symmetric and non-Hermitian eigenvalue problems.
`[s,d]seigt`	Compute Ritz values and error bounds subroutine for the symmetric eigenvalue problem.
`XYgets`	Sort the Ritz values and corresponding error bounds.
`[s,d]Yconv`	Determines which Ritz values satisfy the convergence criterion.
`XYapps`	Application of implicit shifts routine.
`Xortc`	Sorting routines for complex vectors.
`[s,d]ortr`	Sorting routine for real vectors.
`[s,d]laqrb`	Compute the eigenvalues and the last components of the Schur vectors of an upper Hessenberg matrix.
`[s,d]stqrb`	Compute the eigenvalues and the last components of the eigenvectors of a symmetric tridiagonal matrix.

by increasing the value of k. This scheme helps prevent stagnation of the IRAM/IRLM. If k is held fixed at its original value of `nev`, the polynomial restarting becomes less and less effective as as the number of converged wanted Ritz values approaches `nev`. The linear rate of convergence is related directly to the ratios $|\psi(\lambda_j)|/|\psi(\lambda_i)|$ of the restart (or filter) polynomial ψ evaluated at wanted eigenvalues λ_i and unwanted eigenvalues λ_j (See (4.4.4) in Chapter 4). These ratios become unfavorable for wanted eigenvalues λ_i that are too close to the convex hull of the zero set of the filter polynomial ψ. Increasing k artificially increases the distance of the wanted eigenvalues to the zero set of the filter polynomial and as a consequence improves (i.e. decreases) these ratios and decreases the linear convergence factor. A check is performed so that k never exceeds `(ncv-nev)/2`.

A second reason to increase k is when an unwanted Ritz value in Ω_u has converged to the point that it has a computed zero error estimate. The value of k is incremented for every such instance. This prevents attempting to implicitly shift with a converged unwanted Ritz value that is located in a leading submatrix of $\mathbf{H}_m$ that has split from active portion of the iteration.

The third way k may be increased can only occur in `[s,d]naup2`. Complex conjugate pairs of Ritz values must be classified together either as members of the unwanted set Ω_u or of the wanted set Ω_w. If such a pair would have to be split between the two sets then the value of k is increased by 1 so that both are included in the wanted set Ω_w.

If the user has decided to provide the shifts for implicit restarting via reverse communication, the only manner in which p may be decreased from $m-k$ is for second and third reasons given above. One example of shifts that the user may wish to provide is the set of roots of a Chebyshev polynomial of degree p that has been constructed to be small in magnitude over an elliptical region that (approximately) encloses the unwanted eigenvalues. Typically, an ellipse that encloses the set Ω_u but excludes the set Ω_w is constructed and the Chebyshev polynomial of degree p that is small on the ellipse is then specified [11].

5.1.3 XYaitr

Subroutine `XYaitr` is responsible for all the work associated with building the needed factorization. It implements Algorithm 2 of Chapter 4 using the classical Gram-Schmidt procedure with possible re-orthogonalization by the DGKS scheme. At each step j, a residual vector $\mathbf{f}_j$ is computed that is numerically orthogonal to the columns of $\mathbf{V}_j$. If $\|\mathbf{f}_j\| \leq \sin(\pi/4)\|\mathbf{A}\mathbf{V}_j\mathbf{e}_j\|$, a step of re-orthogonalization is performed in order to produce an updated residual $\hat{\mathbf{f}}_j$. If the angle between the successive residual vectors $\hat{\mathbf{f}}_j$ and $\mathbf{f}_j$ is greater than $\pi/4$ then the orthogonalization has been successful and $\mathbf{f}_j$ is replaced by $\hat{\mathbf{f}}_j$ and the last column of $\mathbf{H}_j$ is updated. Otherwise, another re-orthogonalization step is required. This is repeated at most one more time. If a third re-orthogonalization is necessary, then the original $\mathbf{A}\mathbf{V}_j\mathbf{e}_j$ lies in the numerical span of the columns of $\mathbf{V}_j$.

In the event that a third re-orthogonalization is necessary, special action must be taken. If this occurs, then it means that the component of the original $\mathbf{AV}_j\mathbf{e}_j$ that is orthogonal to the $Range(\mathbf{V}_j)$ is indistiguishable from roundoff error. Numerically, the columns of $\mathbf{V}_j$ form a basis for an invariant subspace of $\mathbf{A}$ and consequently the corresponding subdiagonal element β_j is set to zero. In order to continue building the Arnoldi factorization to the desired length, an arbitrary nonzero vector $\mathbf{v}_{j+1}$ must be generated that is orthogonal to the existing columns of $\mathbf{V}_j$. This is accomplished by generating a random vector and orthogonalizing it against the columns of $\mathbf{V}_j$ to get the new basis vector $\mathbf{v}_{j+1}$.

5.1.4 `Xgetv0`

The subroutine `Xgetv0` is responsible for generating starting vectors. It is called upon to construct the initial Arnoldi/Lanczos vector when this option is requested. `Xgetv0` is also called upon to generate a new basis vector when the re-orthogonalization scheme of `XYaitr` calls for this, i.e. when an invariant subspace is encountered early during the construction of the Arnoldi factorization. In the latter case, the vector that is returned by `Xgetv0` is already orthogonalized against the existing set of basis vectors. In either case, if the (shift-invert) computational mode calls for it, the vector is also forced to be in the range of `OP`. The LAPACK subroutine `Xlarnv` is used to generate the random vectors. Subroutine `Xgetv0` is called by `XYaup2` and `XYaitr`.

5.1.5 `Xneigh`

Subroutine `[s,d]neigh` calls the ARPACK subroutine `[s,d]laqrb`. This routine is a modified version of the LAPACK subroutine `[s,d]lahqr` for computing a real Schur form for an upper Hessenberg matrix. Subroutine `[s,d]laqrb` computes the upper quasi-triangular Schur matrix just as `[s,d]lahqr` does but it only computes the last components of the associated Schur vectors since these are all that is needed for the error estimates. The complex arithmetic subroutine `[c,z]neigh` computes the complete Schur decomposition of the projected matrix $\mathbf{H}_m$ using LAPACK subroutine `[c,z]lahqr`.

The subroutine `[c,z]neigh` then calls the LAPACK subroutine `[c,z]trevc` to compute the eigenvectors of the upper triangular Schur matrix. However, just as in real arithmetic case, only the last components of these eigenvectors are needed for computing error estimates.

5.1.6 `[s,d]seigt`

Subroutine `[s,d]seigt` calls the ARPACK subroutine `[s,d]stqrb` that computes the eigenvalues and last components of the corresponding eigenvectors of the real symmetric tridiagonal matrix. This is a modification of the LAPACK subroutine `[s,d]stqr`. `[s,d]seigt` performs the task equivalent to `[s,d]neigh` but takes advantage of symmetry.

5.1.7 [s,d]Yconv

Subroutine `[s,d]Yconv` declares that a Ritz value θ is an acceptable approximation to an eigenvalue of `OP` if $\|\mathbf{f}_m\| \, |\mathbf{e}_m^T\mathbf{s}| \leq \max(\epsilon_M\|\mathbf{H}_m\|, \mathtt{tol}\cdot|\theta|)$ is satisfied where ϵ_M is machine precision. Subroutine `[c,z]naup2` performs the checks directly within the code. The value of `tol` is defined by the user and has a default of machine precision ϵ_M. Since $\|\mathtt{OP}\mathbf{x} - \mathbf{x}\theta\| = \|\mathbf{f}_m\| \, |\mathbf{e}_m^T\mathbf{s}|$, ARPACK avoids computation of the direct residual $\mathtt{OP}\mathbf{x} - \mathbf{x}\theta$ when assessing the numerical quality of the Ritz pair.
CAUTION: Only for symmetric (Hermitian) eigenproblems may the user assume that a computed Ritz value is an accurate approximation to an eigenvalue of `OP`. As § 4.6 of Chapter 4 explained, determining whether an IRAM computes accurate eigenvalues for non-symmetric (non Hermitian) eigenproblems depends on the sensitivity of the eigenproblem. If `OP` arises from a shift-invert spectral transformation, then the eigenvector purification step performed by `XYeupd` will most likely result in smaller residuals for the original problem than those obtained for `OP` (See (4.5.1) in Chapter 4).

5.1.8 XYapps

Subroutine `XYapps` applies the shifts using an implicitly shifted QR mechanism on the projected matrix $\mathbf{H}_m$. If a shift is complex, then `[s,d]napps` employs a double implicit shift application so that the complex conjugate of the shift is applied simultaneously. This allows the implicit shift application to take place in real arithmetic. Finally, `XYapps` updates the current Arnoldi/Lanczos basis to obtain the new length k factorization and $\mathbf{f}_k^+$. Subdiagonal elements of the Hessenberg matrices are set to zero if they satisfy the same criterion as used in a standard implementation of the QR algorithm. (This amounts to checking if any subdiagonal of $\mathbf{H}_m$ is less than the product of machine precision and the sum of the two adjacent diagonal elements of $\mathbf{H}_m$.)

5.1.9 XYeupd

The purpose of `XYeupd` is to obtain the requested eigenvalues and eigenvectors (or Schur basis vectors) for the original problem $\mathbf{Ax} = \mathbf{Mx}\lambda$ from the information computed by `XYaupd` for the linear operator `OP`. Regardless of whether a spectral transformation is used, the eigenvectors will remain unchanged on transforming back to the original problem. If a spectral transformation is used, then subroutine `XYaupd` will compute eigenvalues of `OP`. Subroutine `XYeupd` maps them to those of $\mathbf{Ax} = \mathbf{Mx}\lambda$ except in two cases. The exceptions occur when using `[s,d]naupd` with a complex shift σ with either of $\mathtt{OP} = \mathrm{Real}((\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M})$ or $\mathtt{OP} = \mathrm{Imag}((\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M})$. Note that if σ is a real shift, `[s,d]neupd` can recover the eigenvalues since then $\mathtt{OP}_{\mathrm{real}} = \bar{\mathtt{OP}}_{\mathrm{real}}$. Otherwise, the eigenvalues must be recovered by the user, preferably by using the converged Ritz vectors and computing Rayleigh quotients with them for the original problem. We hope to automate this step in a future release.

Figure 5.2: Outline of algorithm used by subroutine `XYeupd` to compute Schur vectors and possibly eigenvectors.

1. Compute the partial Schur form $\mathbf{H}_m\mathbf{Q}_k = \mathbf{Q}_k\mathbf{R}_k$ where the k converged wanted Ritz values computed by `XYaupd` are located on the diagonal of the upper triangular matrix $\mathbf{R}_k$ of order k.

2. Compute the approximate Schur vectors of $\mathbf{A}$ by forming $\mathbf{V}_m\mathbf{Q}_k$ and placing in the first k columns of $\mathbf{V}_m$. Denote the matrix consisting of these first k columns by $\mathbf{V}_k$.

3. If eigenvectors are desired, then

   (a) Compute the eigendecomposition $\mathbf{R}_k\mathbf{S}_k = \mathbf{S}_k\mathbf{D}_k$.

   (b) Compute the Ritz vectors by forming $\mathbf{V}_k\mathbf{S}_k$.

If eigenvectors are desired, an orthonormal basis for the invariant subspace corresponding to the converged Ritz values is first computed. The vectors of this orthonormal basis are called approximate Schur vectors for $\mathbf{A}$. Figure 5.2 outlines our strategy. Refer to Figure 5.2 for definitions of the quantities discussed in the remainder of this section.

For symmetric eigenvalue problems `[s,d]seupd` does not need Step 3 of Figure 5.2 since Schur vectors are also eigenvectors. Moreover, a special routine is not required to re-order the Schur form since $\mathbf{R}_k$ is a diagonal matrix of real eigenvalues.

For real non-symmetric eigenvalue problems, `[s,d]neupd` uses the real Schur form. That is, $\mathbf{R}_k$ is an upper quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign. Associated with each 2-by-2 diagonal block is a complex conjugate pair of eigenvalues. The real eigenvalues are stored on the diagonal of $\mathbf{R}_k$. Similarly, $\mathbf{D}_k$ is a block diagonal matrix. When the eigenvalue is complex, the complex eigenvector associated with the eigenvalue with positive imaginary part is stored in two consecutive columns of $\mathbf{S}_k$. The first column holds the real part of the eigenvector and the second column holds the imaginary part. The eigenvector associated with the eigenvalue with negative imaginary part is simply the complex conjugate of the eigenvector associated with the positive imaginary part. The computed Ritz vectors are stored in the same manner.

The computation of the partial Schur form needed at Step 1 is performed by first calling the appropriate LAPACK subroutine that computes the full Schur decomposition of $\mathbf{H}_m$. Another LAPACK subroutine, `Xtrsen`, re-orders the computed Schur form to obtain $\mathbf{Q}_k$ and $\mathbf{R}_k$. The approximate Schur vectors are formed by computing the QR factorization of $\mathbf{Q}_k$ and then postmultiplying $\mathbf{V}_m$ with the factored form. This avoids the need for the additional storage that would be necessary if $\mathbf{V}_m\mathbf{Q}_k$ were computed directly. The appropriate LAPACK subroutines are used to compute and apply the QR factorization of $\mathbf{Q}_k$. The factored approach described above is extremely stable and efficient since $\mathbf{Q}_k$ is a numerically orthogonal matrix.

In exact arithmetic, there would be no need to perform the reordering (or the sorting for the symmetric eigenvalue problem). In theory, the implicit restarting mechanism would obviate the need for this. However, computing in finite precision arithmetic (as usual) complicates the issue and make these final reorderings mandatory. See Chapter 5 in [6] and [7] for further information.

When Ritz vectors are required, the LAPACK subroutine `Xtrevc` is called to compute the decomposition $\mathbf{R}_k\mathbf{S}_k = \mathbf{S}_k\mathbf{D}_k$. Since $\mathbf{S}_k$ is an upper quasi triangular matrix, the product $\mathbf{V}_k\mathbf{S}_k$ is easily formed using the level 3 BLAS subroutine `Xtrmm`.

The computed eigenvectors (Ritz vectors) returned by `XYeupd` are normalized to have unit length with respect to the $\mathbf{B}$ semi-inner product that was used. Thus, if $\mathbf{B} = \mathbf{I}$ they will have unit length in the standard 2-norm. In general, a computed eigenvector $\mathbf{x}$ will satisfy $1 = \mathbf{x}^H\mathbf{B}\mathbf{x}$ with respect to the $\mathbf{B}$ matrix that was specified.

5.2 LAPACK routines used by ARPACK

ARPACK uses a variety of LAPACK auxiliary and computational subroutines. An auxiliary routine is one that performs some basic computation and/or an unblocked form of an algorithm. On the other hand, a computational routine typically implements the block version of the algorithm. For example, the *computational* subroutine `Xhseqr` determines the eigenvalues and Schur decomposition of an upper Hessenberg matrix using a multishift QR algorithm. The *auxiliary* routine `Xlahqr` implements the standard double shift form of the QR algorithm for determining the eigenvalues and Schur decomposition. For further details and information, see Chapter 2 and Appendices A and B in [1].

Tables 5.2 and 5.3 list all the LAPACK routines used by ARPACK. The current release of LAPACK used is version `2.0`.

5.3 BLAS routines used by ARPACK

Tables 5.4–5.6 list all the BLAS subroutines called by ARPACK directly. We remark that there are other BLAS subroutines needed by ARPACK that are called by the LAPACK routines.

Table 5.2: Description of the LAPACK computational routines used by ARPACK.

ROUTINE	DESCRIPTION
`Xtrsen`	Re-orders the Schur form of a matrix.
`[s,d]steqr`	Diagonalize a symmetric tridiagonal matrix.
`ctrevc`	Computes the eigenvectors of a matrix in upper triangular form.
`strevc`	Computes the eigenvectors of a matrix in upper quasi-triangular form.

Table 5.3: Description of the LAPACK auxiliary routines used by ARPACK.

ROUTINE	DESCRIPTION
`Xlahqr`	Computes the Schur decomposition of an upper Hessenberg matrix.
`Xgeqr2`	Computes the QR factorization of a matrix.
`sorm2r`	Applies a real orthogonal matrix in factored form.
`cunm2r`	Applies a complex orthogonal matrix in factored form.
`Xlascl`	Scales a matrix stably.
`Xlanhs`	Compute various matrix norms of a Hessenberg matrix.
`Xlacpy`	Perform a matrix copy.
`Xlamch`	Determine various machine parameters.
`[s,d]labad`	Determines over- and underflow limits.
`[s,d]lapy2`	Compute $\sqrt{x^2+y^2}$ stably.
`Xlartg`	Generates a plane rotation.
`[s,d]larfg`	Generates a real elementary reflector.
`[s,d]larf`	Applies a real elementary reflector H to a real matrix.
`Xlaset`	Initialize a matrix.

Table 5.4: Description of the Level three BLAS used by ARPACK.

ROUTINE	DESCRIPTION
`Xtrmm`	Matrix times an upper triangular matrix.

Table 5.5: Description of the Level two BLAS used by ARPACK.

ROUTINE	DESCRIPTION
`Xgemv`	Matrix vector product.
`[s,d]ger`	Rank one update to a real matrix.
`[c,z]geru`	Rank one update to a complex matrix.

Table 5.6: Description of the Level one BLAS used by ARPACK.

ROUTINE	DESCRIPTION
`Xaxpy`	Compute a vector triad.
`Xscal`	Scale a vector.
`[s,d]dot`	Compute a real inner product.
`[c,z]dotc`	Compute a complex inner product.
`[cs,zd]scal`	Scale a complex vector with a real constant.
`[s,d]nrm2`	Computes the Euclidean norm of a real vector.
`[sc,dz]nrm2`	Computes the Euclidean norm of a complex vector.
`Xcopy`	Copy one vector to another.
`Xswap`	Swap two vectors

Appendix A

Templates and Driver Routines

A collection of simple driver routines was described in Chapter 2. This appendix describes a more sophisticated set of example drivers that illustrate all of the available computational modes and shift-invert strategies available within ARPACK. All aspects of solving standard or generalized eigenvalue problems, as well as computing a partial SVD are covered here. As with the simple drivers, these have been designed to use as templates that may be modified for a specific user application. These drivers can be found in the following subdirectories in the `EXAMPLES` directory. The contents of this `EXAMPLES` directory are:

`SYM`	Real symmetric eigenvalue problems,
`NONSYM`	Real non-symmetric eigenvalue problems,
`COMPLEX`	Complex eigenvalue problems,
`BAND`	Eigenvalue problems in which matrices are stored in LAPACK band form,
`SVD`	Singular value decomposition (SVD) problems.

Each driver illustrates how variables are declared and used, how the reverse communication communication interface is used within a particular computational mode, and how to check the accuracy of the computed results.

This Appendix provides some guidance on deciding which driver to select and how to use it effectively. In order to aid the process of building a user's application code from these drivers, we discuss the necessary steps to be followed to use these drivers as templates and modify them appropriately. These steps may be summarized as follows:

- Selecting an appropriate driver.
- Identifying and constructing the linear operator `OP` and the matrix `B` used in a driver.
- Substituting the constructed linear operator `OP` and the matrix `B` in the reverse communication interface.

- Modifying the problem dependent variables.
- Checking the accuracy of the computed results.

This procedure is discussed in §§ A.1–A.3 for the solution of symmetric, non-symmetric and complex arithmetic eigenvalue problems with ARPACK. Each case is discussed independently, so there is considerable repetition. The user is encouraged to select the section that discusses the specific problem type of interest.

A.1 Symmetric Drivers

There are six drivers for the symmetric eigenvalue problem ($\mathbf{A} = \mathbf{A}^T$ and $\mathbf{M} = \mathbf{M}^T$). They are named in the form of `XsdrvY`, where the first character `X` specifies the precision used as follows:

`s`	single precision
`d`	double precision.

The last character `Y` is a number between `1` and `6` indicating the type of the problem to be solved and the mode to be used. Each number is associated with a unique combination of the `bmat` and `iparam(7)` parameters. The parameter `which` used to select the eigenvalues of interest is controlled by the user, but recommended settings are given in the discussion that follows. Table A.1 lists the problem solved by each double precision driver. The first four drivers are the most commonly used. The last two drivers use two special spectral transformations that may accelerate the convergence for particular problems.

A.1.1 Selecting a Symmetric Driver

Several drivers may be used to solve the same problem. However, one driver may be more appropriate or may be easier to modify than the others. This decision typically depends upon the nature of the application and the portion of the spectrum to be computed. See § 3.2 of Chapter 3 for more discussion on the issues that should be considered when deciding to use a spectral transformation.

Standard Mode

Driver `dsdrv1` solves the standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \mathbf{x}\lambda.$$

This mode only requires matrix vector products with $\mathbf{A}$. It is most appropriate for computing extremal, non-clustered eigenvalues. This mode can be successful for clustered eigenvalues, but convergence is often much slower in this case. Since this mode is generally the easiest to try, it is recommended to use it

Table A.1: The functionality of the symmetric drivers.

DRIVER	PROBLEM SOLVED
`dsdrv1`	Standard eigenvalue problem (`bmat = 'I'`) in the regular mode (`iparam(7) = 1`).
`dsdrv2`	Standard eigenvalue problem (`bmat = 'I'`) in a shift-invert mode (`iparam(7) = 3`).
`dsdrv3`	Generalized eigenvalue problem (`bmat = 'G'`) in the regular inverse mode (`iparam(7) = 2`).
`dsdrv4`	Generalized eigenvalue problem (`bmat = 'G'`) in a shift-invert mode (`iparam(7) = 3`).
`dsdrv5`	Generalized eigenvalue problem (`bmat = 'G'`) in the Buckling mode (`iparam(7) = 4`).
`dsdrv6`	Generalized eigenvalue problem (`bmat = 'G'`) in the Cayley mode (`iparam(7) = 5`).

initially and convert to more appropriate modes after some information about the distribution of the spectrum has been obtained. It can be particularly useful to run initially with `which = 'BE'` with a loose tolerance just to get an idea of the largest and smallest eigenvalues.

Shift-Invert Mode

Driver `dsdrv2` uses the shift-invert mode to find eigenvalues closest to a shift σ. This is often used to compute interior eigenvalues or clustered extremal eigenvalues. For a discussion of shift-invert mode, see § 3.2 in Chapter 3. To use `dsdrv2`, the user is required to supply the action of

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{I})^{-1}\mathbf{v}.$$

This is typically accomplished by factoring the matrix $\mathbf{A} - \sigma\mathbf{I}$ once before the iteration begins and then using this factorization repeatedly to solve the sequence of linear systems that arise during the calculation. The IRL iteration will find selected eigenvalues of $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ depending on the setting of `which`. To compute the eigenvalues of $\mathbf{A}$ just to the right of σ, one should set `which = 'LA'` and for those just to the left set `which = 'SA'`. Eigenvalues closest to σ may be obtained by setting `which = 'LM'`. Eigenvalues to either side of σ may be obtained by setting `which = 'BE'`.

Generalized Eigenvalue Problem

If the generalized eigenvalue problem

$$\mathbf{Ax} = \mathbf{Mx}\lambda,$$

is to be solved, then one of the drivers `dsdrv3, dsdrv4, dsdrv5, dsdrv6` should be used. If either $\mathbf{A}$ or $\mathbf{M}$ may be factored, then the generalized eigenvalue problem may be converted to a standard eigenvalue problem as described in § 3.2 of Chapter 3,

Regular Inverse Mode

Driver `dsdrv3` uses the regular inverse mode to solve the generalized eigenvalue problem. This mode is appropriate when $\mathbf{M}$ is symmetric and positive definite but it is not feasible to compute a sparse direct Cholesky factorization $\mathbf{M} = \mathbf{LL}^T$. It might also be appropriate if $\mathbf{M}$ can be factored but there is reason to think that $\mathbf{M}$ is ill-conditioned. To use `dsdrv3` the user must supply the action of

$$\mathbf{w} \leftarrow \mathbf{M}^{-1}\mathbf{Av} \quad \text{and} \quad \mathbf{w} \leftarrow \mathbf{Mv}$$

The action of $\mathbf{M}^{-1}$ is typically done with an iterative solver such as preconditioned conjugate gradient. The use of $\mathbf{M}$-inner products restores symmetry. If $\mathbf{M}$ can be factored and is reasonably well conditioned, then direct conversion to a standard problem is recommended. Also, note that if A is positive definite and the smallest eigenvalues are sought, then it is best to reverse the roles of $\mathbf{A}$ and $\mathbf{M}$ and compute the largest eigevalues of $\mathbf{A}^{-1}\mathbf{M}$. The reciprocals of these will then be the eigenvalues of interest.

Shift-Invert Mode

Driver `dsdrv4` uses the shift-inverse mode to solve the generalized eigenvalue problem. Eigenvalues closest to a shift σ can be obtained by computing selected eigenvalues ν for

$$(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{Mx} = \mathbf{x}\nu.$$

To compute the eigenvalues of $(\mathbf{A}, \mathbf{M})$ just to the right of σ, one should set `which = 'LA'` and for those just to the left set `which = 'SA'`. Eigenvalues closest to σ may be obtained by setting `which = 'BE'`.

The eigenvalue λ of the original problem and ν are related by

$$\lambda = \sigma + \frac{1}{\nu}$$

To use `dsdrv4`, the user is required to supply the two matrix vector operations

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{v}, \quad \text{and} \quad \mathbf{w} \leftarrow \mathbf{Mv}$$

where σ is the shift defined by the user. Typically, the above matrix operation is performed by factoring $\mathbf{A} - \sigma\mathbf{M}$ once at the beginning and and then using this factorization repeatedly to solve the resulting linear system. Note, this will require a symmetric *indefinite* factorization whenever σ is a point in the interior of the spectrum of $(\mathbf{A}, \mathbf{M})$. A general (sparse) LU factorization may also be used, but there will be a storage penalty for ignoring symmetry.

Buckling Mode

Driver `dsdrv5` implements the Buckling transformation. Eigenvalues closest to a shift σ can be obtained by computing the selected eigenvalues ν of

$$\mathtt{OP}\mathbf{x} \equiv (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{A}\mathbf{x} = \mathbf{x}\nu.$$

Settings of `which` are the same as those of the shift-invert modes. This mode assumes that $\mathbf{A}$ is a symmetric postive semi definite matrix. Note that the operator `OP` is symmetric with respect to the semi-inner product defined by $\mathbf{A}$. The eigenvalue λ of the original problem and ν are related by

$$\lambda = \frac{\sigma\nu}{\nu - 1}$$

Note that $\sigma = 0$ should not be used for this mode. The two operations

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{y} \quad \text{and} \quad \mathbf{y} \leftarrow \mathbf{A}\mathbf{v}$$

are required in order to use this driver.

Cayley Transformation Mode

Driver `dsdrv6` implements the Cayley spectral transformation. Eigenvalues closest to a shift σ can be obtained by computing the selected eigenvalues ν of

$$\mathtt{OP}\mathbf{x} \equiv (\mathbf{A} - \sigma\mathbf{M})^{-1}(\mathbf{A} + \sigma\mathbf{M})\mathbf{x} = \mathbf{x}\nu.$$

Settings of `which` are the same as those of the shift-invert modes. It is easy to verify that `OP` is symmetric with respect to the (semi) inner product defined by $\mathbf{M}$ so it is best to use $\mathbf{M}$-inner products with this transformation. The eigenvalue λ of the original problem and ν are related by

$$\lambda = \sigma\left(\frac{1+\nu}{1-\nu}\right).$$

Note that the transformation becomes ill-defined as $\sigma \to 0$ since $\nu \to 1$. The expression does have a limit as $\sigma \to 0$ but it would be better to use regular inverse mode or to use shift-invert mode with $\sigma = 0$ when eigenvalues near the origin are sought.

The Cayley transformation has slightly different properties than the standard shift and invert spectral transformation used by drivers `dsdrv3` and `dsdrv4`. To use this driver, the operations

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{y}, \quad \mathbf{y} \leftarrow (\mathbf{A} + \sigma\mathbf{M})\mathbf{v} \quad \text{and} \quad \mathbf{w} \leftarrow \mathbf{M}\mathbf{v}$$

are required.

Table A.2: The operators `OP` and `B` for `dsaupd`.

DRIVER	`OP`	`B`
`dsdrv1`	$\mathbf{A}$	$\mathbf{I}$
`dsdrv2`	$(\mathbf{A} - \sigma\mathbf{I})^{-1}$	$\mathbf{I}$
`dsdrv3`	$\mathbf{M}^{-1}\mathbf{A}$	$\mathbf{M}$
`dsdrv4`	$(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$	$\mathbf{M}$
`dsdrv5`	$(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{A}$	$\mathbf{A}$
`dsdrv6`	$(\mathbf{A} - \sigma\mathbf{M})^{-1}(\mathbf{A} + \sigma\mathbf{M})$	$\mathbf{M}$

A.1.2 Identify `OP` and `B` for the Driver

Once an appropriate driver has been selected, it is necessary to construct the action of the linear operator `OP` and matrix `B` associated with that driver. Eigenvalues of `OP` are computed by the computational routine `dsaupd`. These eigenvalues are converted to those of $\mathbf{A}$ or $(\mathbf{A}, \mathbf{M})$ by the post-processing routine `dseupd`. The Lanczos vectors generated by `dsaupd` are orthogonal with respect to the (semi-) inner product defined by `B`. It is imperative that the operations `OPv` and `Bv` be computed as prescribed for the selected driver. Table A.2 summarizes the operators `OP` and `B` required for each driver.

Because of the reverse communication interface in ARPACK, the construction of

$$\mathbf{w} \leftarrow \mathtt{OP}\mathbf{v} \quad \text{and} \quad \mathbf{w} \leftarrow \mathtt{B}\mathbf{v}$$

is left completely to the user. This means that the user is free to choose any convenient data structure for the matrix representation. If the matrix is not available, the user is free to express the action of the matrix on a vector through a subroutine call or a code segment.

A.1.3 The Reverse Communication Interface

The use of the reverse communication interface for the regular and shift-invert modes was illustrated in Chapter 2 and 3, respectively. The most general structure of the reverse communication loop is presented in Figure A.1.

The specific actions to be taken within the reverse communication loop vary with each of the different drivers. Each action in Figure A.1 corresponds to an `ido` value returned from the call to `dsaupd`. These actions involve matrix vector operations such as $\mathbf{w} \leftarrow \mathtt{OP}\mathbf{v}$ and $\mathbf{w} \leftarrow \mathtt{B}\mathbf{v}$. The matrix vector operation must be performed on the correct portion of the work array `workd`. Below, we list the the specific matrix-vector operations that must be performed in each driver for a given returned value of the reverse communication parameter `ido`.

```
c      %-----------------------------------------------%
c      | M A I N   L O O P (Reverse communication loop) |
c      %-----------------------------------------------%
c
 10    continue
c
c         %---------------------------------------------%
c         | Repeatedly call the routine DSAUPD and take |
c         | actions indicated by parameter IDO until    |
c         | either convergence is indicated or maxitr   |
c         | has been exceeded.                          |
c         %---------------------------------------------%
c
          call dsaupd ( ido, bmat, n, which, nev, tol, resid,
     &                  ncv, v, ldv, iparam, ipntr, workd, workl,
     &                  lworkl, info )
c
          if ( ido .eq.  -1 ) then
c
             w ← OP*v
c
c            % ----- L O O P   B A C K to call DSAUPD again.  ----- %
c
             go to 10
c
          else if ( ido .eq.  1 ) then
c
             w ← OP*v
c
c            % ----- L O O P   B A C K to call DSAUPD again.  ----- %
c
             go to 10
c
          else if ( ido .eq.  2 ) then
c
             w ← B*v
c
c            % ----- L O O P   B A C K to call DSAUPD again.  ----- %
c
             go to 10
c
          end if
```

Figure A.1: Reverse communication structure

Driver `dsdrv1`

`OP` $= \mathbf{A}$ and `B` $= \mathbf{I}$
The action $\mathbf{w} \leftarrow \mathbf{Av}$ is required in this driver.

- `ido=1`
  Action Required:

  Matrix vector multiplication $\mathbf{w} \leftarrow \mathbf{Av}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dsdrv2`

`OP` $= (\mathbf{A} - \sigma \mathbf{I})^{-1}$ and `B` $= \mathbf{I}$
The action $\mathbf{w} \leftarrow (\mathbf{A} - \sigma \mathbf{I})^{-1}\mathbf{v}$ is required in this driver.

- `ido=-1` or `ido=1`
  Action Required:

  Solve $(\mathbf{A} - \sigma \mathbf{I})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  The righthand side $\mathbf{v}$ is in the array `workd(ipntr(1))`.
  The solution $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dsdrv3`

`OP` $= \mathbf{M}^{-1}\mathbf{A}$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow \mathbf{Av}$, $\mathbf{w} \leftarrow \mathbf{M}^{-1}\mathbf{v}$ and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required.

- `ido=-1` or `ido=1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Av}$. Solve $\mathbf{Mw} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The vector $\mathbf{y}$ must returned in (overwrite) `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{Mv}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dsdrv4`

`OP` $= (\mathbf{A} - \sigma \mathbf{M})^{-1}\mathbf{M}$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow (\mathbf{A} - \sigma \mathbf{M})^{-1}\mathbf{v}$. and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Mv}$. Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

  Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(3))` (Action by $\mathbf{M}$ has already been made).
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{Mv}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dsdrv5`

`OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{A}$ and `B` $= \mathbf{A}$
The actions $\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{v}$. and $\mathbf{w} \leftarrow \mathbf{Av}$ are required.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Av}$. Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

  Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(3))` (Action by $\mathbf{A}$ has already been made).
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{Av}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dsdrv6`

`OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}(\mathbf{A} + \sigma\mathbf{M})$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{v}$, $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$, and $\mathbf{w} \leftarrow \mathbf{M}\mathbf{v}$ are required.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow (\mathbf{A} + \sigma\mathbf{M})\mathbf{v}$. Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow (\mathbf{A} + \sigma\mathbf{M})\mathbf{v}$. Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(3))` .
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{M}\mathbf{v}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

A.1.4 Modify the Problem Dependent Variables

To set up the proper storage and to arrange for effective use of the IRLM various parameters will have to be set. These variables include:

`n` The dimension of the problem.

`nev` The number of eigenvalues needed.

`ncv` The length of the Lanczos factorization. This represents the maximum number of Lanczos vectors used.

`which` The eigenvalues of interest.

`info` Set to `0` for a randomly generated starting vector. If the user decides to use another starting vector, this value should be set to 1, and the starting vector should be provided in the array `resid`.

`sigma` The shift used if a spectral transformation is employed.

The variable `nev` may be set to be a value larger than the number of eigenvalues desired to avoid splitting a eigenvalue cluster. The only restriction is that `nev` must be less than `ncv`. The recommended initial choice of `ncv` is to set `ncv =`

Table A.3: The eigenvalues of interest for symmetric eigenvalue problems.

which	EIGENVALUES
'LA'	Largest (algebraic) eigenvalues.
'SA'	Smallest (algebraic) eigenvalues.
'LM'	Largest eigenvalues in magnitude.
'SM'	Smallest eigenvalues in magnitude.
'BE'	Eigenvalue at both ends. When `nev` is odd, compute one more from the high end than from the low end.

$2 \cdot$ `nev`. The user is encouraged to experiment with both `nev` and `ncv`. The possible choices for the input variable `which` are listed in Table A.3. When a spectral transformation is employed, only the selections `'BE'`, `'LA'` or `'SA'` should be used.

Once the above variables are modified, the storage declarations

```
integer          maxn, maxnev, maxncv, ldv
parameter       (maxn=256, maxnev=10, maxncv=25, ldv=maxn )
```

should be adjusted so that the conditions

$$\begin{array}{lcl} \texttt{n} & \leq & \texttt{maxn}, \\ \texttt{nev} & \leq & \texttt{maxnev}, \\ \texttt{ncv} & \leq & \texttt{maxncv}, \\ \texttt{nev + 1} & \leq & \texttt{ncv} \end{array}$$

are satisfied.

Other Variables

The following variables are also set in all drivers. Their usage is described in Chapter 2. In most cases, they do not need to be changed.

`lworkl` The size of the work array `workl` used by `dsaupd`. Must be set to at least `ncv*(ncv+8)`.

`tol` The convergence criterion. The default setting is machine precision. However, the value of `tol` should be set to control the desired accuracy. Typically, the smaller this value the more work is required to satisfy the stopping criteria. However, setting this value too large may cause eigenvalues to be missed when there are multiple or clustered eigenvalues.

`ido` The reverse communication flag. Must be set to `0` before entering dsaupd.

`bmat` Designates whether a standard (`bmat = 'I'`) or generalized eigenvalue (`bmat = 'G'`) problem

`iparam(1)` The shifting strategy used during the implicitly restarted portion of an IRLM. Unless the user has an expert understanding of IRLM, an exact shifting strategy selected by setting `iparam(1) = 1` should be used.

`iparam(3)` Maximum number of IRLM iterations allowed.

`iparam(7)` Indicates the algorithmic mode used with ARPACK.

`rvec` Indicates whether eigenvectors are desired. If the eigenvectors are of to be computed, then one must set `rvec = .true.` and otherwise set it to `.false.` .

A.1.5 Postprocessing and Accuracy Checking

Once the eigenvalues and eigenvectors have been extracted from the postprocessing routine `dseupd`, the user may check the accuracy of the result by computing the direct residuals $\|\mathbf{Ax} - \mathbf{x}\lambda\|$ or $\|\mathbf{Ax} - \mathbf{Mx}\lambda\|$ for standard or generalized eigenvalue problems, respectively. In order to compute the above quantities, the matrix vector product routines for $\mathbf{Ax}$ and $\mathbf{Mx}$ must be supplied, even if they are not used in the reverse communication loop. Residual checking is provided in all drivers.

A.2 Real Nonsymmetric Drivers

There are six drivers for nonsymmetric eigenvalues problem. They are named in the form of `XndrvY`, where the first character `X` specifies the precision used,

`s` single precision
`d` double precision

and the last character `Y` is a number between `1` and `6` indicating the mode to be used. Each number is associated with a combination of `bmat` and `iparam(7)` parameters used in that driver and also on whether the desired shift σ is real or complex. The parameter `which` used to select the eigenvalues of interest is controlled by the user, but recommended settings are given in the discussion that follows. Table A.4 summarizes the features of the double precision drivers. The first four drivers are the ones most commonly used. The last two drivers are used when the complex shift used in the shift-invert mode has a nonzero imaginary part. Either `dndrv5` or `dndrv6` may be modified to solve a standard eigenvalue problem in shift-invert mode with a complex shift. If the amount of storage used by complex arithmetic is not prohibitive, then the complex drivers

Table A.4: The functionality of the non-symmetric drivers.

DRIVER	PROBLEM SOLVED
`dndrv1`	Standard eigenvalue problem (`bmat = 'I'`) in the regular mode (`iparam(7) = 1`) No shift is needed in this driver.
`dndrv2`	Standard eigenvalue problem (`bmat = 'I'`) in a shift-invert mode (`iparam(7) = 3`) The shift is real (`sigmai = 0.0`).
`dndrv3`	Generalized eigenvalue problem (`bmat = 'G'`) in the regular inverse mode (`iparam(7) = 2`) No shift is needed in this driver.
`dndrv4`	Generalized eigenvalue problem (`bmat = 'G'`) in a shift-invert mode (`iparam(7) = 3`) with a real shift (`sigmai = 0.0`).
`dndrv5`	Generalized eigenvalue problem (`bmat = 'G'`) in a shift invert mode (`iparam(7) = 3`) The shift has a nonzero imaginary part (`sigmai` $\neq$ `0.`)
`dndrv6`	Solve a generalized eigenvalue problem (`bmat = 'G'`) in a shift invert mode (`iparam(7) = 4`) The shift has a nonzero imaginary part (`sigmai` $\neq$ `0`).

of § A.3 should be used instead. A procedure for modifying a nonsymmetric driver is outlined below. It is similar to the one used for the symmetric drivers.

A.2.1 Selecting a Non-symmetric Driver

Several drivers may be used to solve the same problem. However, one driver may converge faster or may be easier to modify than the other depending on the nature of the application. The decision of what to use should be based on the type of problem to be solved and the part of the spectrum that is of interest. See § 3.2 of Chapter 3 for more discussion on the issues that should be considered when deciding to use a spectral transformation.

Standard Mode

Driver `dndrv1` solves the standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \mathbf{x}\lambda.$$

This mode only requires matrix vector products with $\mathbf{A}$. This driver will compute the eigenvalues of largest or smallest magnitude, largest or smallest real part, largest or smallest imaginary part depending on the setting of `which`. It is most appropriate for computing extremal, non-clustered eigenvalues (here extremal means extreme points of the convex hull of the spectrum). In particular, if the operation $\mathbf{A}^{-1}\mathbf{x}$ can be easily formed, then using the shift-invert driver `dndrv2` with zero shift is certainly a far more effective way to compute eigenvalues of the smallest magnitude.

Shift-Invert Mode

Driver `dndrv2` uses the shift-invert mode to find eigenvalues closest to a real shift σ. This is often used to compute interior eigenvalues. For a discussion of shift-invert mode, see § 3.2 in Chapter 3. To use `dndrv2`, the user is required to supply the action of

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{I})^{-1}\mathbf{v},$$

This is typically accomplished by factoring the matrix $\mathbf{A} - \sigma\mathbf{I}$ once before the iteration begins and then using this factorization repeatedly to solve the sequence of linear systems that arise during the calculation. The IRAM will find selected eigenvalues of $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ depending on the setting of `which`. The recommended setting for computing the eigenvalues of A nearest to σ is `which = 'LM'`. If the desired shift has a nonzero imaginary part, then `dndrv5` or `dndrv6` should be modified to solve the problem. For eigenvalue problems where the additional storage needed is not prohibitive, the complex shift-invert driver `zndrv2` should be used.

Generalized Nonsymmetric Eigenvalue Problem

If the generalized eigenvalue problem

$$\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{x}\lambda$$

is to be solved, one can either convert it into a standard eigenvalue problem as described in § 3.2 of Chapter 3 or use `dndrv3, dndrv4, dndrv5, dndrv6` that are designed specifically for the generalized problem.

Regular Inverse Mode

Driver `dndrv3` uses the regular inverse mode to solve the generalized eigenvalue problem. This mode should be used if $\mathbf{M}$ is symmetric and positive definite but it is not possible to factor into a Cholesky factorization $\mathbf{M} = \mathbf{L}\mathbf{L}^T$ or if there is reason to think that $\mathbf{M}$ is ill-conditioned. To use `dndrv3` the user must supply the action of

$$\mathbf{w} \leftarrow \mathbf{M}^{-1}\mathbf{A}\mathbf{v} \quad \text{and} \quad \mathbf{w} \leftarrow \mathbf{M}\mathbf{v}$$

The action of $\mathbf{M}^{-1}$ is typically done with an iterative solver such as preconditioned conjugate gradient. If $\mathbf{M}$ can be factored then direct conversion to a standard problem is recommended. This driver is appropriate for `which`

`= 'LM','LR', 'SR','LI','SI'` settings. If interior eigenvalues are sought then driver `dndrv4` is probably more appropriate. If convergence is slower than desired then one of the shift-invert modes described below should be used.

Spectral Transformations for Non-symmetric Eigenvalue Problems

If eigenvalues near a point σ are sought, then one of the shift-invert drivers `dndrv4`, `dndrv5` or `dndrv6` should be used. Driver `dndrv4` can be used when the shift σ has a zero imaginary part. Otherwise, either `dndrv5` or `dndrv6` should be used. To use these drivers, one is required to supply the action of

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}\mathbf{v}.$$

When σ has a nonzero imaginary part, a complex factorization routine or a complex iterative solver will be required. In that case, the vector $\mathbf{w}$ will in general be a complex vector. However, only the real or imaginary part will be passed into `dnaupd` and all arithmetic and data types within the ARPACK routines will be real. In any case, the IRAM will find selected eigenvalues of $(\mathbf{A} - \sigma\mathbf{M})^{-1}$ depending on the setting of `which`. The recommended setting for computing eigenvalues of the pair $(\mathbf{A}, \mathbf{M})$ nearest to σ is `which = 'LM'`. However, other settings are not precluded.

A.2.2 Identify `OP` and `B` for the Driver

Once a driver is chosen, the next step is to identify `OP` and `B` associated with that driver. Eigenvalues of `OP` are computed by the computational routine `dnaupd`. These eigenvalues are converted to those of $\mathbf{A}$ or $(\mathbf{A}, \mathbf{M})$ in the post-processing routine `dseupd`. The Arnoldi vectors generated by `dnaupd` are `B`-orthonormal. It is very important to construct the matrix vector operations

$$\mathbf{w} \leftarrow \mathtt{OP}\mathbf{v} \quad \text{and} \quad \mathbf{w} \leftarrow \mathtt{B}\mathbf{v}$$

correctly. Table A.5 summarizes the operators `OP` and `B` used by the drivers. The notation Real($\mathbf{A}$) and Real($\mathbf{M}$) is used to denote the real and imaginary parts of a complex matrix, respectively.

Because of the reverse communication interface of ARPACK, the construction of

$$\mathbf{w} \leftarrow \mathtt{OP}\mathbf{v} \quad \text{and} \quad \mathbf{w} \leftarrow \mathtt{B}\mathbf{v}$$

is left completely to the user. This means that the user is free to choose any convenient data structure for the matrix representation. If the matrix is not available, the user is free to express the action of the matrix on a vector through a subroutine call or a code segment.

A.2.3 The Reverse Communication Interface

The basic structure of the reverse communication loop for the nonsymmetric drivers is similar to that for the symmetric driver with `dsaupd` replaced with

Table A.5: The operators `OP` and `B` for `dnaupd`.

DRIVER	`OP`	`B`
`dndrv1`	$\mathbf{A}$	$\mathbf{I}$
`dndrv2`	$(\mathbf{A}-\sigma\mathbf{I})^{-1}$	$\mathbf{I}$
`dndrv3`	$\mathbf{M}^{-1}\mathbf{A}$	$\mathbf{M}$
`dndrv4`	$(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{M}$	$\mathbf{M}$
`dndrv5`	$\text{Real}(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{M}$	$\mathbf{M}$
`dndrv6`	$\text{Imag}(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{M}$	$\mathbf{M}$

`dnaupd`. See Figure A.1. The actions taken in `dndrv1`, `dndrv2`, `dndrv3` and `dndrv4` are exactly the same as those in `dsdrv1` `dsdrv2`, `dsdrv3` and `dsdrv4`. The matrix-vector operations required of each driver are listed in detail below. The operator `OP` defined in `dndrv5` and `dndrv6` is designed to accomplish spectral transformations for real nonsymmetric problems while computing exclusively in real arithmetic, the actions taken in these two drivers are completely different from `dsdrv5` and `dsdrv6`.

Driver `dndrv1`

`OP` $= \mathbf{A}$ and `B` $= \mathbf{I}$
The action $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$ is required in this driver.

- `ido=1`
  Action Required:

  > Matrix vector multiplication $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}$.
  > The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  > The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dndrv2`

`OP` $= (\mathbf{A}-\sigma\mathbf{I})^{-1}$ and `B` $= \mathbf{I}$
The action $\mathbf{w} \leftarrow (\mathbf{A}-\sigma\mathbf{I})^{-1}\mathbf{v}$ is required in this driver.

- `ido=-1` or `ido=1`
  Action Required:

  > Solve $(\mathbf{A}-\sigma\mathbf{I})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  > The righthand side $\mathbf{v}$ is in the array `workd(ipntr(1))`.
  > The solution $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dndrv3`

`OP` $= \mathbf{M}^{-1}\mathbf{A}$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow \mathbf{Av}$, $\mathbf{w} \leftarrow \mathbf{M}^{-1}\mathbf{v}$ and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required.

- `ido=-1` or `ido=1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Av}$. Solve $\mathbf{Mw} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The vector $\mathbf{y}$ must returned in (overwrite) `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{Mv}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dndrv4`

`OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{v}$ and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Mv}$. Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

  Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(3))` (Action by $\mathbf{M}$ has already been made).
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{Mv}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dndrv5`

`OP` $= \text{Real}\left[(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{M}\right]$ and `B` $= M$.
The actions $\mathbf{w} \leftarrow \text{Real}\left[(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{v}\right]$ and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required. Note, the factorization of $(\mathbf{A}-\sigma\mathbf{M})$ will generally need to be computed in complex arithmetic and stored as complex data.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Mv}$. Solve $(\mathbf{A}-\sigma\mathbf{M})\mathbf{z} = \mathbf{y}$ for $\mathbf{z}$.
  Set $\mathbf{w} = \text{Real}\left[\mathbf{z}\right]$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

  Solve $(\mathbf{A}-\sigma\mathbf{M})\mathbf{z} = \mathbf{v}$ for $\mathbf{z}$.
  Set $\mathbf{w} = \text{Real}\left[\mathbf{z}\right]$.
  The vector $\mathbf{v}$ is in `workd(ipntr(3))` (Action by $\mathbf{M}$ has already been made).
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{Mv}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `dndrv6`

`OP` $= \text{Imag}\left[(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{M}\right]$ and `B` $= M$.
The actions $\mathbf{w} \leftarrow \text{Imag}\left[(\mathbf{A}-\sigma\mathbf{M})^{-1}\mathbf{v}\right]$. and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required. Note, the factorization of $(\mathbf{A}-\sigma\mathbf{M})$ will generally need to be computed in complex arithmetic and stored as complex data.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{Mv}$. Solve $(\mathbf{A}-\sigma\mathbf{M})\mathbf{z} = \mathbf{y}$ for $\mathbf{z}$.
  Set $\mathbf{w} = \text{Imag}\left[\mathbf{z}\right]$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{z} = \mathbf{v}$ for $\mathbf{z}$.
Set $\mathbf{w} = \mathrm{Imag}\,[\mathbf{z}]$.
The vector $\mathbf{v}$ is in `workd(ipntr(3))` (Action by $\mathbf{M}$ has already been made).
The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{M}\mathbf{v}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

A.2.4 Modify the Problem Dependent Variables

To set up the proper storage and to arrange for effective use of the IRAM various parameters will have to be set. These variables include:

`n`	The dimension of the problem.
`nev`	The number of eigenvalues needed.
`ncv`	The length of the Arnoldi factorization. This represents the maximum number of Arnoldi vectors used.
`which`	The eigenvalues of interest.
`info`	Set to `0` for a randomly generated starting vector. If the user decides to use another starting vector, this value should be set to 1, and the starting vector should be provided in the array `resid`.
`sigmar`	The real part of the shift used if a spectral transformation is employed.
`sigmai`	The imaginary part of the shift used if a spectral transformation is employed.

The variable `nev` may be set to be a value larger than the number of eigenvalues desired to avoid splitting an eigenvalue cluster. The only restriction is that `nev` must be less than `ncv`. The recommended choice of `ncv` is to set `ncv = 2` $\cdot$ `nev`. The user is encouraged to experiment with both `nev` and `ncv`. The possible choices for the input variable `which` are listed in Table A.6. When using a spectral transformation, the selection of `which = 'SM'` should be avoided.

Once the above variables are modified, the storage declarations

```
integer          maxn, maxnev, maxncv, ldv
parameter       (maxn=256, maxnev=10, maxncv=25, ldv=maxn )
```

Table A.6: The eigenvalues of interest for non-symmetric eigenvalue problems.

`which`	EIGENVALUES
`'LM'`	Largest magnitude
`'SM'`	Smallest magnitude
`'LR'`	Largest real parts
`'SR'`	Smallest real parts
`'LI'`	Largest imaginary parts
`'SI'`	Smallest imaginary parts

should be adjusted so that the conditions

$$\begin{array}{lcl} \texttt{n} & \leq & \texttt{maxn}, \\ \texttt{nev} & \leq & \texttt{maxnev}, \\ \texttt{ncv} & \leq & \texttt{maxncv}, \\ \texttt{nev+2} & \leq & \texttt{ncv} \end{array}$$

are satisfied. The last condition on `nev` is needed to assure that complex conjugate pairs of eigenvalues are kept together.

Other Variables

The following variables are also set in all drivers. Their usage is described in Chapter 2. In most cases, they do not need to be changed.

`lworkl` The size of the work array `workl` used by `dnaupd`. Must be set to at least `3*ncv*(ncv+6)`.

`tol` The convergence criterion. The default setting is machine precision. However, the value of `tol` should be set to control the desired accuracy. Typically, the smaller this value the more work is required to satisfy the stopping criteria. However, setting this value too large may cause eigenvalues to be missed when there are multiple or clustered eigenvalues.

`ido` The reverse communication flag. Must be set to `0` before entering dsaupd.

`bmat` Designates whether a standard (`bmat = 'I'`) or generalized eigenvalue (`bmat = 'G'`) problem

`iparam(1)` The shifting strategy used during the implicitly restarted portion of an IRAM. Unless the user has an expert understanding of

IRAM, an exact shifting strategy selected by setting `iparam(1) = 1` should be used.

`iparam(3)` Maximum number of IRAM iterations allowed.

`iparam(7)` Indicates the algorithmic mode used with ARPACK.

`rvec` Indicates whether eigenvectors are needed. If the eigenvectors are of interest, then `rvec = .true.` and set to `.false.` otherwise.

A.2.5 Postprocessing and Accuracy Checking

The eigenvalues and eigenvectors of $\mathbf{A}$ and $(\mathbf{A}, \mathbf{M})$ can be extracted with a call to `dneupd` when using drivers `dndrv1 - dndrv4`. However, since the eigenvalues ν computed for `OP` by the drivers `dndrv5` and `dndrv6` are related to the eigenvalues λ of $\mathbf{Ax} = \mathbf{Mx}\lambda$ by

$$\nu = \frac{1}{2}(\frac{1}{\lambda - \sigma} + \frac{1}{\lambda + \sigma}), \quad \text{and} \quad \nu = \frac{1}{2i}(\frac{1}{\lambda - \sigma} + \frac{1}{\lambda + \sigma}),$$

respectively. These equations do not have a unique solution λ and it appears to be difficult to match the correct solution with a given eigenvector. Thus, the Rayleigh Quotient $\lambda = x^H\mathbf{Ax}/(x^H\mathbf{Mx})$ must be formed by the user to obtain the eigenvalue corresponding to the eigenvector $\mathbf{x}$. This will be done automatically in `dneupd` in a later release of ARPACK.

Once the converged eigenvalues and eigenvectors have been obtained the user may check the accuracy of the results by computing the direct residuals $||\mathbf{Ax} - \mathbf{x}\lambda||$ and $||\mathbf{Ax} - \mathbf{Mx}\lambda||$ for a standard or generalized eigenvalue problems. Residual checking is provided in all drivers.

A.3 Complex Drivers

There are four drivers for complex date type problems. They are named in the form of `XndrvY`, where the first character `X` specifies the precision used as follows:

`c` single precision complex,
`z` double precision complex,

and the last character `Y` is a number between `1` and `4` indicating the type of the problem to be solved and the mode to be used. Each number is associated with a unique combination `bmat` and `iparam(7)` value. The parameter `which` used to select the eigenvalues of interest is controlled by the user, but recommended settings are given in the discussion that follows. Table A.7 summarizes the features of each double precision complex arithmetic driver. The procedure for modifying a complex driver is similar to that is used for the first four symmetric drivers.

Table A.7: The functionality of the complex arithmetic drivers.

DRIVER	PROBLEM SOLVED
`zndrv1`	Standard eigenvalue problem (`bmat = 'I'`) in the regular mode (`iparam(7) = 1`).
`zndrv2`	Standard eigenvalue problem (`bmat = 'I'`) in a shift-invert mode (`iparam(7) = 3`).
`zndrv3`	Generalized eigenvalue problem (`bmat = 'G'`) in the regular inverse mode (`iparam(7) = 2`).
`zndrv4`	Generalized eigenvalue problem (`bmat = 'G'`) in a shift-invert mode (`iparam(7) = 3`).

A.3.1 Selecting a Complex Arithmetic Driver

Several drivers may be used to solve the same problem. However, one driver may work better or may be easier to modify than the other depending on the nature of the application. The decision of what to use should be based on the type of problem to be solved and the part of the spectrum that is of interest. See § 3.2 of Chapter 3 for more discussion on the issues that should be considered when deciding to use a spectral transformation.

Standard Mode

Driver `zndrv1` solves the standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \mathbf{x}\lambda.$$

This mode only requires matrix vector products with $\mathbf{A}$. This driver will compute the eigenvalues of largest or smallest magnitude, largest or smallest real part, largest or smallest imaginary part depending on the setting of `which`. It is most appropriate for computing extremal, non-clustered eigenvalues (here extremal means extreme points of the convex hull of the spectrum). In particular, if the operation $\mathbf{A}^{-1}\mathbf{x}$ can be easily formed, then using the shift-invert driver `zndrv2` with zero shift is certainly a far more effective way to compute eigenvalues of the smallest magnitude.

Shift and Invert Spectral Transformation

Driver `zndrv2` uses the shift-invert mode to find eigenvalues closest to a real shift σ. This is often used to compute interior eigenvalues. For a discussion of shift-invert mode, see § 3.2 in Chapter 3. To use `zndrv2`, the user is required to supply the action of

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{I})^{-1}\mathbf{v},$$

This is typically accomplished by factoring the matrix $\mathbf{A} - \sigma\mathbf{I}$ once before the iteration begins and then using this factorization repeatedly to solve the sequence of linear systems that arise during the calculation. The IRAM will find selected eigenvalues of $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ depending on the setting of `which`. The recommended setting for computing the eigenvalues of A nearest to σ is `which = 'LM'`.

Generalized Eigenvalue Problems

If the generalized eigenvalue problem

$$\mathbf{Ax} = \mathbf{Mx}\lambda$$

is to be solved, one can either convert it into a standard eigenvalue problem as described in § 3.2 of Chapter 3, or employ `zndrv3` or `zndrv4` that are designed for the generalized problem.

Regular Inverse Mode

Driver `zndrv3` uses the regular inverse mode to solve the generalized eigenvalue problem. This mode should be used if $\mathbf{M}$ is Hermitian and positive definite but it is not possible to factor into a Cholesky factorization $\mathbf{M} = \mathbf{LL}^H$ or if there is reason to think that $\mathbf{M}$ is ill-conditioned. To use `zndrv3` the user must supply the action of

$$\mathbf{w} \leftarrow \mathbf{M}^{-1}\mathbf{Av} \quad \text{and} \quad \mathbf{w} \leftarrow \mathbf{Mv}$$

The action of $\mathbf{M}^{-1}$ is typically done with an iterative solver such as preconditioned conjugate gradient. If $\mathbf{M}$ can be factored then direct conversion to a standard problem is recommended. This driver is appropriate for `which = 'LM','LR', 'SR','LI','SI'` settings. If interior eigenvalues are sought then driver `zndrv4` is probably more appropriate.

General Shift-Invert Spectral Transformation

If eigenvalues near a point σ are sought, then the shift-invert driver `zndrv4` should be used. To use this driver, one is required to supply the action of

$$\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{Mv}.$$

The IRAM will find selected eigenvalues of $(\mathbf{A} - \sigma\mathbf{M})^{-1}$ depending on the setting of `which`. The recommended setting for computing eigenvalues of the pair $(\mathbf{A}, \mathbf{M})$ nearest to σ is `which = 'LM'`. However, other settings are not precluded.

Table A.8: The operators `OP` and `B` for `znaupd`.

DRIVER	OP	B
`zndrv1`	$\mathbf{A}$	$\mathbf{I}$
`zndrv2`	$(\mathbf{A} - \sigma\mathbf{I})^{-1}$	$\mathbf{I}$
`zndrv3`	$\mathbf{M}^{-1}\mathbf{A}$	$\mathbf{M}$
`zndrv4`	$(\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$	$\mathbf{M}$

A.3.2 Identify `OP` and `B` for the Driver to be Modified

Once a driver is chosen. The next step is to identify `OP` and `B` associated with that driver. Eigenvalues of `OP` are computed by the computational routine `znaupd`. These eigenvalues are converted to those of $\mathbf{A}$ or $(\mathbf{A}, \mathbf{M})$ in the post-processing routine `zneupd`. The Arnoldi vectors generated by `znaupd` are `B`-orthonormal. It is very important to construct the operation `OPv` and `Bv` correctly. The following list summarize the operator `OP` and `B` defined in each driver.

Because of the reverse communication interface of ARPACK, the construction of

$$\mathbf{w} \leftarrow \mathtt{OPv} \quad \text{and} \quad \mathbf{w} \leftarrow \mathtt{Bv}$$

is left completely to the user. This means that the user is free to choose any convenient data structure for the matrix representation. If the matrix is not available, the user is free to express the action of the matrix on a vector through a subroutine call or a code segment.

A.3.3 The Reverse Communication Interface

The basic reverse communication loop for the complex driver is exactly the same as those used for the symmetric driver except that `dsaupd` is replaced by `znaupd`. (Figure A.1.) Actions to be taken within the loop vary from one driver to the other. Some drivers may take only one or two actions listed in Figure A.1. Each action corresponds to an `ido` value returned from the call to `znaupd`. These actions involve matrix vector operations such as $\mathbf{w} \leftarrow$ `OPv` and $\mathbf{w} \leftarrow$ `Bv`. The matrix vector operation must be performed correctly on the correct portion of the work array `workd` (either `workd(ipntr(1))` or `workd(ipntr(3))`.) The output should also be returned in the correct portion of `workd` (either `workd(ipntr(2))` or `workd(ipntr(1))`) before the loop goes back to the next call to `znaupd`. The matrix-vector operations required of each driver are listed in detail below.

Driver `zndrv1`

`OP` $= \mathbf{A}$ and `B` $= \mathbf{I}$
The action $\mathbf{w} \leftarrow \mathbf{Av}$ is required in this driver.

- `ido=1`
  Action Required:

  > Matrix vector multiplication $\mathbf{w} \leftarrow \mathbf{Av}$.
  > The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  > The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `zndrv2`

`OP` $= (\mathbf{A} - \sigma\mathbf{I})^{-1}$ and `B` $= \mathbf{I}$
The action $\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{I})^{-1}\mathbf{v}$ is required in this driver.

- `ido=-1` or `ido=1`
  Action Required:

  > Solve $(\mathbf{A} - \sigma\mathbf{I})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  > The righthand side $\mathbf{v}$ is in the array `workd(ipntr(1))`.
  > The solution $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `zndrv3`

`OP` $= \mathbf{M}^{-1}\mathbf{A}$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow \mathbf{Av}$, $\mathbf{w} \leftarrow \mathbf{M}^{-1}\mathbf{v}$ and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required.

- `ido=-1` or `ido=1`
  Actions Required:

  > Compute $\mathbf{y} \leftarrow \mathbf{Av}$. Solve $\mathbf{Mw} = \mathbf{y}$ for $\mathbf{w}$.
  > The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  > The vector $\mathbf{y}$ must returned in (overwrite) `workd(ipntr(1))`.
  > The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  > Compute $\mathbf{w} \leftarrow \mathbf{Mv}$.
  > The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  > The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

Driver `zndrv4`

`OP` $= (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{M}$ and `B` $= \mathbf{M}$
The actions $\mathbf{w} \leftarrow (\mathbf{A} - \sigma\mathbf{M})^{-1}\mathbf{v}$. and $\mathbf{w} \leftarrow \mathbf{Mv}$ are required.

- `ido=-1`
  Actions Required:

  Compute $\mathbf{y} \leftarrow \mathbf{M}\mathbf{v}$. Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{y}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=1`
  Actions Required:

  Solve $(\mathbf{A} - \sigma\mathbf{M})\mathbf{w} = \mathbf{v}$ for $\mathbf{w}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(3))` (Action by $\mathbf{M}$ has already been made).
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

- `ido=2`
  Action Required:

  Compute $\mathbf{w} \leftarrow \mathbf{M}\mathbf{v}$.
  The vector $\mathbf{v}$ is in `workd(ipntr(1))`.
  The result vector $\mathbf{w}$ must be returned in the array `workd(ipntr(2))`.

A.3.4 Modify the Problem Dependent Variables

To set up the proper storage and to arrange for effective use of the IRAM various parameters will have to be set. These variables include:

`n`	The dimension of the problem.
`nev`	The number of eigenvalues needed.
`ncv`	The length of the Arnoldi factorization. This represents the maximum number of Arnoldi vectors used.
`which`	The eigenvalues of interest.
`info`	Set to `0` for a randomly generated starting vector. If the user decides to use another starting vector, this value should be set to 1, and the starting vector should be provided in the array `resid`.
`sigma`	The shift used if a spectral transformation is employed.

The variable `nev` may be set to be a value larger than the number of eigenvalues desired to avoid splitting a eigenvalue cluster. The only restriction is that `nev` must be less than `ncv`. The recommended choice of `ncv` is to set `ncv = 2` $\cdot$ `nev`. The user is encouraged to experiment with both `nev` and `ncv`. The possible choices for the input variable `which` are listed in Table A.9. When using a spectral transformation, the selection of `which = 'SM'` should be avoided.

Once the above variables are modified, the storage declarations

Table A.9: The eigenvalues of interest for complex arithmetic eigenvalue problems.

`which`	EIGENVALUES
'LM'	Largest magnitude
'SM'	Smallest magnitude
'LR'	Largest real parts
'SR'	Smallest real parts
'LI'	Largest imaginary parts
'SI'	Smallest imaginary parts

```
integer         maxn, maxnev, maxncv, ldv
parameter      (maxn=256, maxnev=10, maxncv=25, ldv=maxn )
```

should be adjusted so that the conditions

$$\begin{array}{lcl} \texttt{n} & \leq & \texttt{maxn}, \\ \texttt{nev} & \leq & \texttt{maxnev}, \\ \texttt{ncv} & \leq & \texttt{maxncv}, \\ \texttt{nev+1} & \leq & \texttt{ncv} \end{array}$$

are satisfied.

Other Variables

The following variables are also set in all drivers. Their usage is described in Chapter 2. In most cases, they do not need to be changed.

`lworkl` The size of the work array `workl` used by `znaupd`. Must be set to at least `3*ncv*(ncv+5)`.

`tol` The convergence criterion. The default setting is machine precision. However, the value of `tol` should be set to control the desired accuracy. Typically, the smaller this value the more work is required to satisfy the stopping criteria. However, setting this value too large may cause eigenvalues to be missed when there are multiple or clustered eigenvalues.

`ido` The reverse communication flag. Must be set to `0` before entering dsaupd.

`bmat` Designates whether a standard (`bmat = 'I'`) or generalized eigenvalue (`bmat = 'G'`) problem `iparam(1)` The shifting strategy used during the implicitly restarted portion of an IRAM. Unless the user has an expert understanding of IRAM, an exact shifting strategy selected by setting `iparam(1) = 1` should be used.

`iparam(3)` Maximum number of IRAM iterations allowed.

`iparam(7)` Indicates the algorithmic mode used with ARPACK.

`rvec` Indicates whether eigenvectors are needed. If the eigenvectors are of interest, then `rvec = .true.` and set to `.false.` otherwise.

A.3.5 Post-processing and Accuracy Checking

Once the eigenvalues and eigenvectors have been extracted from the post-processing routine `zneupd`, the user may check the accuracy of the result by computing the direct residuals $||\mathbf{Ax} - \mathbf{x}\lambda||$ and $||\mathbf{Ax} - \mathbf{Mx}\lambda||$ for standard or generalized eigenvalue problems, respectively.

A.4 Band Drivers

If the matrix **A** and **M** are stored in LAPACK band form, then one of the band drivers may be used. Band drivers are named in the form of `XYbdrZ`, where the first character `X` specifies the precision and data type,

`s`	single precision
`d`	double precision
`c`	single precision complex
`z`	double precision complex

the second character `Y` indicates the symmetry property of the problem,

`s`	symmetric problem
`n`	nonsymmetric problem

and the third character `Z` is a number between 1 and 6 indicating the type of the problem to be solved and the mode to be used. Each number is associated with a combination of `bmat` and `iparam(7)` settings used in that driver. Tables A.10—A.12 list the double precision band storage drivers.

There are no special drivers for complex Hermitian problem. Complex Hermitian problems can be solved by using `[c,z]nbdrZ`. These drivers call the band eigenvalue computation routine `XYband`, where the first character `X` (`s`,`d`) specifies the precision and data type as listed above, and the second character `Y` indicates the symmetry property of the problem that can be solved with this routine. Since the reverse communication interface has already been implemented in these computational routines, users only need to provide the matrix and modify a few variables in these drivers to solve their own problem. A procedure for modifying these drivers is presented below.

Table A.10: Band storage drivers for symmetric eigenvalue problems

BAND DRIVER	PROBLEM SOLVED
`dsbdr1`	Standard eigenvalue problem (`bmat = 'I'`) in the regular mode (`iparam(7) = 1`).
`dsbdr2`	Standard eigenvalue problem (`bmat = 'I'`) in a shift-invert mode (`iparam(7) = 3`).
`dsbdr3`	Generalized eigenvalue problem (`bmat = 'G'`) in the regular inverse mode (`iparam(7) = 2`).
`dsbdr4`	Generalized eigenvalue problem (`bmat = 'G'`) in a shift-invert mode (`iparam(7) = 3`).
`dsbdr5`	Generalized eigenvalue problem (`bmat = 'G'`) in the Buckling mode (`iparam(7) = 4`).
`dsbdr6`	Generalized eigenvalue problem (`bmat = 'G'`) in the Cayley mode (`iparam(7) = 5`).

Table A.11: Band storage drivers for non-symmetric eigenvalue problems

BAND DRIVER	PROBLEM SOLVED
`dnbdr1`	Standard eigenvalue problem (`bmat = 'I'`) in the regular mode (`iparam(7) = 1`).
`dnbdr2`	Standard eigenvalue problem (`bmat = 'I'`) in a shift-invert mode (`iparam(7) = 3`).
`dnbdr3`	Generalized eigenvalue problem (`bmat = 'G'`) in the regular inverse mode (`iparam(7) = 2`).
`dnbdr4`	Generalized eigenvalue problem (`bmat = 'G'`) in a real shift-invert mode (`iparam(7) = 3`).
`dnbdr5`	Standard eigenvalue problem (`bmat = 'I'`) in a complex shift invert mode (`iparam(7) = 4`).
`dnbdr6`	Generalized eigenvalue problem (`bmat = 'G'`) in a complex shift invert mode (`iparam(7) = 4`).

Table A.12: Band storage drivers for Complex arithmetic eigenvalue problems.

BAND DRIVER	PROBLEM SOLVED
`znbdr1`	Standard eigenvalue problem (`bmat = 'I'`) in the regular mode (`iparam(7) = 1`).
`znbdr2`	Standard eigenvalue problem (`bmat = 'I'`) in a shift-invert mode (`iparam(7) = 3`).
`znbdr3`	Generalized eigenvalue problem (`bmat = 'G'`) in the regular inverse mode (`iparam(7) = 2`).
`znbdr4`	Generalized eigenvalue problem (`bmat = 'G'`) in a shift-invert mode (`iparam(7) = 3`).

A.4.1 Selecting a Band Storage Driver

Several drivers may be used to solve the same problem. However, one driver may work better or may be easier to modify than the other depending on the nature of the application. The decision of what to use should be based on the type of problem to be solved and the part of the spectrum that is of interest. Typically, regular mode drivers can be used to find extremal eigenvalues and shift-invert drivers are used to find interior eigenvalues or extremal eigenvalues that are clustered.

All of the drivers discussed in the previous sections are available in band form. For more detail on a particular mode, select the driver from the appropriate section above and then use its banded counterpart. For example, if the appropriate general driver is `dndrv4` then the corresponding band driver would be `dnbdr4`. The primary difference between these two is that `dnbdr4` has been derived from `dndrv4` by modifying it to use band storage and LAPACK band factorization routines.

A.4.2 Store the matrix correctly

The band routines assume the matrix **A** and **M** are stored in LAPACK band form. In the following we used AB and MB to denote **A** and `B` stored in band form. If the matrix **A** has kl subdiagonals and ku superdiagonals, then the ijth element of A a_{ij} is stored in AB$(kl+ku+1+i-j,j)$ for $max(1,j-ku) \leq i \leq min(m,j+kl)$. An example of a band matrix **A** with $kl=2$, $ku=1$ is illustrated below.

$$A = \begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix} \to AB = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{pmatrix}$$

The elements marked * in the matrix AB need not be set.

A.4.3 Modify problem dependent variables

These variables include the following

`n` The dimension of the problem.

`nev` The number of eigenvalues needed.

`ncv` The length of the Arnoldi factorization.

`which` Part of the spectrum that is of interest.

`sigma` Real shift used in `dsbdr2` and `dsbdr4`, or complex shift used in `znbdr2` and `znbdr4`.

`sigmar` The real part of the shift used in a real nonsymmetric shift invert mode driver (`dnbdr2, dnbdr4, dnbdr5, dnbdr6`.)

`sigmai` The imaginary part of the shift used in a real nonsymmetric shift invert mode driver (`dnbdr5, dnbdr6`). It should be set to zero in `dnbdr2` and `dndrv4`.

A.4.4 Modify other variables if necessary

The following variables are also set in all drivers. Their usage is described in Chapters 2 and 3. In most cases, they do not need to be changed.

`lworkl` Must be set to at least `ncv*ncv+3*ncv`.

`tol` Usually set to zero. It can be changed depending on the accuracy desired. Typically, the smaller this value the more work is required to satisfy the stopping criteria. However, setting this value too large may cause eigenvalues to be missed when they are multiple or very tightly clustered.

`ido` Must be set to 0 before entering dnaupd.

`info` Usually set to 0. In this case, a random starting vector is generated to start the Arnoldi iteration. If the user decides to use other starting vector, this value should be set to 1, and the starting vector should be provided in the array `resid`.

`bmat`	Either 'I' or 'G' depending on the problem to be solved. This variable has been set appropriately in each driver. The user should not change its value.
`iparam(1)`	Usually set to 1. This indicates that *exact shift* strategy is used in the computation. For a discussion on shift strategy see § 4.4.1 of Chapter 4.
`iparam(3)`	Maximum iterations allow. It is set to 300 in all drivers. But it can be reset to any reasonable value.
`iparam(7)`	Indicate algorithmic mode. This variable has been set appropriately in each driver. The user should not change its value.
`rvec`	Indicate whether eigenvector is needed. It is set to `.true.` in all drivers. If no eigenvector is needed, this may be set to `.false.`

A.4.5 Accuracy checking

Once the eigenvalues and eigenvectors have been obtained, the user may check the accuracy of the result by computing the direct residuals $\|\mathbf{Ax} - \lambda\mathbf{x}\|$, and $\|\mathbf{Ax} - \lambda\mathbf{Mx}\|$, for a standard and generalized eigenvalue problem, respectively.

A.5 The Singular Value Decomposition

Every rectangular matrix $\mathbf{A} \in \mathbf{R}^{m\times n}$ with $m \geq n$ may be factored into the form

$$\mathbf{A} = \mathbf{USV}^T \tag{A.5.1}$$

where $\mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{I}_n$ are matrices with orthonormal columns and the diagnoal matrix $\mathbf{S} = diag(\sigma_1, \sigma_2, \cdots, \sigma_n)$. The numbers $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$ are called the *singular values* of $\mathbf{A}$. The columns of $\mathbf{U}$ are the *left singular vectors* and the columns of $\mathbf{V}$ are the *right singular vectors* of $\mathbf{A}$. This is the so-called "short form" of the Singular Value Decompositon (SVD) of $\mathbf{A}$.

The relationship (A.5.1) may be manipulated using orthogonality to reveal that

$$\mathbf{A}^T\mathbf{A} = \mathbf{VS}^2\mathbf{V}^T \quad \text{with} \quad \mathbf{U} = \mathbf{AVS}^{-1}, \tag{A.5.2}$$

if $\sigma_n > 0$. Thus selected singular values and the corresponding right singular vectors may be computed by finding eigenvalues and vectors for the $n \times n$ matrix $\mathbf{A}^T\mathbf{A}$.

In many applications, one is iterested in computing a few (say k) of the largest singular values and corresponding vectors. If $\mathbf{U}_k$, $\mathbf{V}_k$ denote the leading k-columns of $\mathbf{U}$ and $\mathbf{V}$ respectively, and if $\mathbf{S}_k$ denotes the leading principal submatrix of $\mathbf{S}$ then

$$\mathbf{A}_k \equiv \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$$

Figure A.2: Compute $\mathbf{w} \leftarrow \mathbf{A}^T\mathbf{A}\mathbf{v}$ by Blocks

Initialize $\mathbf{w} \leftarrow 0$;
For $j = 1, 2, 3, ...\ell$,
  $\mathbf{C} \leftarrow \mathbf{A}_j$; % Read next block of $\mathbf{A}$%
  $\mathbf{z} \leftarrow \mathbf{C}\mathbf{v}$;
  $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{C}^T\mathbf{z}$;
End_For

is the best rank-k approximation to $\mathbf{A}$ in both the 2-norm and the Frobenius norm. Often a very small k will suffice to approximate important features of the original $\mathbf{A}$ or to approximately solve least squares problems involving $\mathbf{A}$.

This partial SVD may be computed efficiently using ARPACK subroutine `_saupd` in `mode = 1` with `which = 'LA'` and taking

$$\mathtt{OP} = \mathbf{A}^T\mathbf{A} \quad \text{and} \quad \mathtt{B} = \mathbf{I}.$$

Of course, the action of $\mathbf{w} \leftarrow \mathtt{OP}\mathbf{v}$ should be computed with the steps

1. Matrix-vector multiply $\mathbf{z} \leftarrow \mathbf{A}\mathbf{v}$.
2. Matrix-vector multiply $\mathbf{w} \leftarrow \mathbf{A}^T\mathbf{z}$.

Also, note that if the matrix $\mathbf{A}$ is huge and must be stored on a peripheral device, then $\mathbf{A}$ may be read in by blocks to achieve the action of $\mathbf{w} \leftarrow \mathtt{OP}\mathbf{v}$ using the fact that

$$\mathtt{OP}\mathbf{v} = \sum_{j=1}^{\ell} \mathbf{A}_j^T\mathbf{A}_j\mathbf{v},$$

where $\mathbf{A}^T = (\mathbf{A}_1^T, \mathbf{A}_2^T, \cdots, \mathbf{A}_\ell^T)$ to obtain the loop shown in Figure A.2.

The drivers illustrate how to compute the leading k terms of the SVD as just described. The left singular vectors are recovered from the right singular vectors. As long as the largest singular values are not multiple or tightly clusterd, there should be no problem in obtaining numerically orthogonal left singular vectors from the computed right singular vectors. However, there is a way to get the both the left and right singular vectors directly. This is to define

$$\mathtt{OP} = \begin{pmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0} \end{pmatrix}$$

and utilize the fact that

$$\begin{pmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{U} \\ \mathbf{V} \end{pmatrix} = \begin{pmatrix} \mathbf{U} \\ \mathbf{V} \end{pmatrix} \mathbf{S}$$

to extract the columns of $\mathbf{U}_k$ from the first m components of the converged eigenvectors of `OP` and the columns of $\mathbf{V}_k$ from the remaining n components. If this scheme is used, it is important to set `which = 'LA'` because the blocked matrix `OP` will have both σ_j and $-\sigma_j$ as eigenvalues for $j = 1, 2, \cdots, n$.

We only provide the first approach in the ARPACK drivers. We also should mention that in case you have a matrix $\mathbf{A}$ with $m < n$ then replace $\mathbf{A}$ with $\mathbf{A}^T$ in the above discussion and reverse the roles of $\mathbf{U}$ and $\mathbf{V}$.

A.5.1 The SVD Drivers

The drivers for computing the singular value decomposition are of the form `Xsvd` where `X` denotes the precision and data type,

`s`	single precision
`d`	double precision.

Of course, the SVD is defined for complex matrices as well and it is a straightforward matter to convert the real arithmetic driver to a corresponding complex arithmetic driver.

These drivers may be easily modified to estimate the 2-norm condition number $\kappa_2(\mathbf{A}) = \frac{\sigma_1}{\sigma_n}$ by setting `which = 'BE'`. This will ask for a few of the smallest and a few of the largest singular values simultaneously. The condition number could then be estimated by taking the ratio of the largest and smallest singular values.

Since these drivers are simply special cases of `dsdrv1`, the parameter settings will not be described further. The only cautionary note is that the parameter `which` may be set to `'SA'` if desired but this is not recommended if it is expected that $\mathbf{A}$ will be nearly rank deficient.

Appendix B

Tracking the progress of ARPACK

This appendix describes two mechanisms that are helpful in debugging and also for understanding performance issues and convergence behavior. We shall discuss the trace facility that is included as part of the ARPACK codes. We also describe how to include a check pointing capability for longer runs that may require intermittent interruptions or recovery from system crashes.

B.1 Obtaining Trace Output

ARPACK provides a means to trace the progress of the computation as it proceeds. Various levels of output may be specified from no output, level = 0, to voluminous, level = 3. The following statements may be used within the calling program to initiate and request this output.

```
include 'debug.h'
ndigit = -3
logfil = 6
msgets = 0
msaitr = 0
msapps = 0
msaupd = 1
msaup2 = 0
mseigt = 0
mseupd = 0
```

The parameter `logfil` specifies the logical unit number of the output file. The parameter `ndigit` specifies the number of decimal digits and the width of the output lines. A positive value of `ndigit` specifies that 132 columns are used during output and a negative value specifies eighty columns are to be used. The values of the remaining parameters indicate the output levels from the indicated routines.

For the above example, `msaitr` indicates the level of output requested for the subroutine `ssaitr` or `dsaitr`. The above configuration will give a

Figure B.1: Sample output produced by `dsaupd`.

```
==========================================
= Symmetric implicit Arnoldi update code =
= Version Number: 2.1                    =
= Version Date:   11/15/95               =
==========================================
= Summary of timing statistics           =
==========================================


Total number update iterations             =     8
Total number of OP*x operations            =   125
Total number of B*x operations             =     0
Total number of reorthogonalization steps  =   125
Total number of iterative refinement steps =     0
Total number of restart steps              =     0
Total time in user OP*x operation          =     0.020002
Total time in user B*x operation           =     0.000000
Total time in Arnoldi update routine       =     0.210021
Total time in ssaup2 routine               =     0.190019
Total time in basic Arnoldi iteration loop =     0.110011
Total time in reorthogonalization phase    =     0.070007
Total time in (re)start vector generation  =     0.000000
Total time in trid eigenvalue subproblem   =     0.040004
Total time in getting the shifts           =     0.000000
Total time in applying the shifts          =     0.040004
Total time in convergence testing          =     0.000000
```

breakdown of the number of matrix vector products required, the total number of iterations, the number of re-orthogonalization steps and an estimate of the time spent in each routine and phase of the computation. The output displayed by Figure B.1 is produced.

The user is encouraged to experiment with the other settings once some familiarity has been gained with the routines. The sample drivers discussed in Chapter 2 use the trace debugging capability.

The include statement sets up the storage declarations that are solely associated with this trace debugging feature. The structure of `debug.h` is displayed in Figure B.2. The parameters on the line starting with `msaupd` are for the symmetric codes, while the next two lines are for the nonsymmetric and complex arithmetic codes, respectively. A comprehensive break down of each parameter is listed in Table B.1.

Table B.1: Description of the message level settings for ARPACK.

Routine	Level	Description
`mYaupd`	1	Print the total number of iterations taken, the number of converged Ritz values, the Ritz values and corresponding Ritz estimates, and various timing statistics.
`mYaup2`	1	Print the current iteration and the number of converged Ritz values. Upon exit, print the number of converged Ritz values, the Ritz values and estimates.
	2	Print the length of the Arnoldi extended factorization, the $\mathbf{B}-$norm of its residual vector. Print `NEV` and `NP`, the Ritz values and estimates at each iteration.
	3	Print the real and imaginary parts of all the Ritz values and associated Ritz estimates, `NEV, NP, NUMCNV, NCONV`. Print the shifts. If the exact shift strategy is used, also print the associated Ritz estimates of the shifts. Print the $\mathbf{B}-$norm of the residual of the compressed factorization and the compressed upper Hessenberg matrix.
`mYaitr`	1	Notification of a restart.
	2	Print the number of Arnoldi vector being generated and the $\mathbf{B}-$norm of the current residual.
	3	Print the columns of the Hessenberg matrix as they are generated, reorthogonalization and iterative refinement information, the final upper Hessenberg matrix of order `K+NEV`, and $\mathbf{V}_j^T\mathbf{B}\mathbf{f}_j/\|\mathbf{f}_j\|_\mathbf{B}$.
`mYeigh`	2	Print the last row of the Schur matrix for $\mathbf{H}$, and the last row of the eigenvector matrix for $\mathbf{H}$.
	3	Print the initial upper Hessenberg matrix, the computed eigenvalues associated Ritz estimates.
`mYapps`	1	Print information about where deflation occurred.
	2	Print `sigmak, betak`, order of the final Hessenberg matrix, and the final compressed upper Hessenberg matrix.
	3	Print implicit application of shift number, real and imaginary part of the shift, and the indices of the submatrix that the shift is applied.
`mYeupd`	2	Print the `NCV` eigenvalues. Print the final set of converged Ritz values.
	3	Print the reordered eigenvalues.

Figure B.2: The include file debug.h.

```
c
c\SCCS Information: @(#)
c FILE: debug.h  SID: 2.3  DATE OF SID: 11/16/95   RELEASE: 2
c
c     %---------------------------------%
c     | See debug.doc for documentation |
c     %---------------------------------%
      integer  logfil, ndigit, mgetv0,
     &         msaupd, msaup2, msaitr, mseigt, msapps, msgets,
     &         mseupd, mnaupd, mnaup2, mnaitr, mneigh, mnapps,
     &         mngets, mneupd, mcaupd, mcaup2, mcaitr, mceigh,
     &         mcapps, mcgets, mceupd
      common /debug/
     &         logfil, ndigit, mgetv0,
     &         msaupd, msaup2, msaitr, mseigt, msapps, msgets,
     &         mseupd, mnaupd, mnaup2, mnaitr, mneigh, mnapps,
     &         mngets, mneupd, mcaupd, mcaup2, mcaitr, mceigh,
     &         mcapps, mcgets, mceupd
```

B.2 Check Pointing ARPACK

There are several situations where it would be desirable to have a mechanism to recover from an unexpected interruption in a computation. One way to accomplish this is to save the *state* of the computation every so often at regular intervals or *check points*. In case of an interruption, the computation may be resumed from the last point before the fault occurred. This section explains how to implement this *check pointing* with the ARPACK codes. Familiarity with the `ARPACK` codes and with the reverse communication protocol is assumed.

There are two major reasons why check pointing might be done. The first, is that the computation is sufficiently time consuming (say at least a day) that the user might want to save the state of the computation in case of hardware/system failures. The second is that the user exceeds the maximum number of iterations initially set and the user may desire to continue the computation to convergence without starting over completely. For example, the user initially set `NEV=6` and after `IPARAM(3) = MXITER` only five Ritz values satisfy the convergence requirement specified by `TOL`. The user would then increase the value of `MXITER` and resume the computation.

We briefly explain the procedure for check pointing the *ARPACK* codes using the double precision symmetric code `dsaupd` as an example. The example shows how to save the state of the computation every `10` iterations. This will require a minor modification to the source codes `dsaupd.f` and `dsaup2.f` found

Figure B.3: Reading in a previous state with the example program dssave.

```
c
c     %---------------------------------------%
c     | A file state exists, so read it in.   |
c     %---------------------------------------%
c
 99   open(12,err=199,file='arpack_state',status='old')
      print*,'dssave: input existing state'
      iounit = 12
      read(iounit,8000) ido, bmat, n, which, nev, tol, ncv,
     &                  iparam, ipntr, lworkl, info, np,
     &                  rnorm, nconv, nev2
      valfmt = '(3e22.16)'
      read(iounit,valfmt) (resid(i), i = 1, n)
      ntmp = 3*n
      read(iounit,valfmt) (workd(i), i = 1, ntmp)
      read(iounit,valfmt) (workl(i), i = 1, lworkl)
     do 7002 j=1,ncv
         read(iounit,valfmt) (v(i,j), i = 1, n)
 7002 continue
```

in the ARPACK subdirectory SRC (see Chapter 1). These modified codes are already available at the ftp site in the directory pub/software/ARPACK/CONTRIBUTED.

The driver routine is called dssave. When executed, control first seeks the file arapck_state that is located in the current working directory. This is done by the following

```
c
c     %--------------------------------------------%
c     | Open the data file that will save the state. |
c     %--------------------------------------------%
c
      open(12,err= 99,file='arpack_state',status='new')
```

This statement attempts to open the file arpack_state. Since the open statement contains the status='new' flag, an error is encountered if a file named arpack_state exists and a jump to the statement labeled 99 is taken. The open statement successfully occurs only if there is no file named arpack_state. The statements immediately following the open statement are executed.

The section of code listed in Figure B.3 is executed when the file arpack_state exists. The code reads in a previous state of computation.

The section of code listed in Figure B.4 writes the state of the computation when ido=-2. This occurs when the number of iterations equals 10. This is all accomplished within the modified subroutine dsaup2. The do 100 restrt = 1,mxstrt will allow up to mxstrt writes of the state of the computation to the file arpack_state. Note that before each save of the computation,

Figure B.4: Writing a state with the example program `dssave`.

```
c
c     %--------------------------------%
c     | Start of the checkpointing loop |
c     %--------------------------------%
c
      mxstrt = 3
      do 100 restrt = 1,mxstrt
c
c        %---------------------------------------------%
c        | M A I N   L O O P (Reverse communication loop) |
c        %---------------------------------------------%
c
 10      continue
c
c        %---------------------------------------------%
c        | Repeatedly call the routine DSAUPD and take |
c        | actions indicated by parameter IDO until    |
c        | either convergence is indicated or maxitr   |
c        | has been exceeded.                          |
c        %---------------------------------------------%
c
         call dsaupd ( ido, bmat, n, which, nev, tol, resid,
     &        ncv, v, ldv, iparam, ipntr, workd, workl,
     &        lworkl, info, np, rnorm, nconv, nev2 )
c
```

the file `arpack_state` is rewound resulting in an overwrite of the contents of `arpack_state`.

Figure B.5: Writing a state with the example program `dssave` contd.

```
         if (ido .eq. -2) then
c
c        %--------------------------------------------------%
c        | After maxitr iterations without convergence,     |
c        | output the computed quantities to the file state. |
c        %--------------------------------------------------%
c
            rewind(iounit,err=399)
            write(iounit,8000) ido, bmat, n, which, nev, tol,
     &           ncv, iparam,
     &           ipntr, lworkl, info,
     &           np, rnorm, nconv, nev2
 8000       format(i2,a1,i14,a2,i14,d23.16,16x,/,
     &           12i5,12x,/,
     &           13i5,7x,/,
     &           i5,d23.16,i5,i5)
            ifmt = 16
            len  = ifmt + 6
            nperli = 3
            write(valfmt,8001) nperli,len,ifmt
 8001       format(1h(,i1,1he,i2,1h.,i2,1h))
            write(iounit,valfmt) (resid(i), i = 1, n)
            ntmp = 3*n
            write(iounit,valfmt) (workd(i), i = 1, ntmp)
            write(iounit,valfmt) (workl(i), i = 1, lworkl)
            do 8002 j=1,ncv
               write(iounit,valfmt) (v(i,j), i = 1, n)
 8002       continue
            go to 100
         endif
```

Appendix C

The XYaupd ARPACK Routines

In this appendix we exhibit the headers of the three main computational routines `dsaupd`, `dnaupd`, and `znaupd`. Although these codes are nearly identical in structure and usage, there are a number of differences that are problem dependent. Therefore, each is listed separately.

Information on the calling sequence, input and output parameters, storage and data types may be found here. Also, error flags and warnings are listed.

C.1 DSAUPD

```
c-----------------------------------------------------------------------
c\BeginDoc
c
c\Name: dsaupd
c
c\Description:
c
c  Reverse communication interface for the Implicitly Restarted Arnoldi
c  Iteration.  For symmetric problems this reduces to a variant of the Lanczos
c  method.  This method has been designed to compute approximations to a
c  few eigenpairs of a linear operator OP that is real and symmetric
c  with respect to a real positive semi-definite symmetric matrix B,
c  i.e.
c
c       B*OP = (OP')*B.
c
c  Another way to express this condition is
c
c       < x,OPy > = < OPx,y >  where < z,w > = z'Bw  .
c
c  In the standard eigenproblem B is the identity matrix.
c  ( A' denotes transpose of A)
c
c  The computed approximate eigenvalues are called Ritz values and
c  the corresponding approximate eigenvectors are called Ritz vectors.
c
c  dsaupd is usually called iteratively to solve one of the
c  following problems:
c
c  Mode 1:  A*x = lambda*x, A symmetric
c           ===> OP = A  and  B = I.
c
c  Mode 2:  A*x = lambda*M*x, A symmetric, M symmetric positive definite
c           ===> OP = inv[M]*A  and  B = M.
c           ===> (If M can be factored see remark 3 below)
c
c  Mode 3:  K*x = lambda*M*x, K symmetric, M symmetric positive semi-definite
c           ===> OP = (inv[K - sigma*M])*M  and  B = M.
c           ===> Shift-and-Invert mode
c
c  Mode 4:  K*x = lambda*KG*x, K symmetric positive semi-definite,
c           KG symmetric indefinite
c           ===> OP = (inv[K - sigma*KG])*K  and  B = K.
c           ===> Buckling mode
c
c  Mode 5:  A*x = lambda*M*x, A symmetric, M symmetric positive semi-definite
c           ===> OP = inv[A - sigma*M]*[A + sigma*M]  and  B = M.
c           ===> Cayley transformed mode
c
c  NOTE: The action of w <- inv[A - sigma*M]*v or w <- inv[M]*v
c        should be accomplished either by a direct method
c        using a sparse matrix factorization and solving
c
c           [A - sigma*M]*w = v  or M*w = v,
c
c        or through an iterative method for solving these
c        systems.  If an iterative method is used, the
c        convergence test must be more stringent than
c        the accuracy requirements for the eigenvalue
c        approximations.
c
c\Usage:
c  call dsaupd
c     ( IDO, BMAT, N, WHICH, NEV, TOL, RESID, NCV, V, LDV, IPARAM,
c       IPNTR, WORKD, WORKL, LWORKL, INFO )
c
c\Arguments
c  IDO     Integer.  (INPUT/OUTPUT)
c          Reverse communication flag.  IDO must be zero on the first
c          call to dsaupd.  IDO will be set internally to
c          indicate the type of operation to be performed.  Control is
c          then given back to the calling routine which has the
c          responsibility to carry out the requested operation and call
c          dsaupd with the result.  The operand is given in
c          WORKD(IPNTR(1)), the result must be put in WORKD(IPNTR(2)).
c          (If Mode = 2 see remark 5 below)
c          -------------------------------------------------------------
c          IDO =  0: first call to the reverse communication interface
c          IDO = -1: compute  Y = OP * X  where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y.
c                    This is for the initialization phase to force the
c                    starting vector into the range of OP.
c          IDO =  1: compute  Y = OP * Z  and Z = B * X where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y,
c                    IPNTR(3) is the pointer into WORKD for Z.
c          IDO =  2: compute  Y = B * X  where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y.
c          IDO =  3: compute the IPARAM(8) shifts where
c                    IPNTR(11) is the pointer into WORKL for
c                    placing the shifts. See remark 6 below.
c          IDO = 99: done
c          -------------------------------------------------------------
c          After the initialization phase, when the routine is used in
c          either the "shift-and-invert" mode or the Cayley transform
c          mode, the vector B * X is already available and does not
c          need to be recomputed in forming OP*X.
c
```

```
c  BMAT    Character*1.  (INPUT)
c          BMAT specifies the type of the matrix B that defines the
c          semi-inner product for the operator OP.
c          B = 'I' -> standard eigenvalue problem A*x = lambda*x
c          B = 'G' -> generalized eigenvalue problem A*x = lambda*B*x
c
c  N       Integer.  (INPUT)
c          Dimension of the eigenproblem.
c
c  WHICH   Character*2.  (INPUT)
c          Specify which of the Ritz values of OP to compute.
c
c          'LA' - compute the NEV largest (algebraic) eigenvalues.
c          'SA' - compute the NEV smallest (algebraic) eigenvalues.
c          'LM' - compute the NEV largest (in magnitude) eigenvalues.
c          'SM' - compute the NEV smallest (in magnitude) eigenvalues.
c          'BE' - compute NEV eigenvalues, half from each end of the
c                 spectrum.  When NEV is odd, compute one more from the
c                 high end than from the low end.
c           (see remark 1 below)
c
c  NEV     Integer.  (INPUT)
c          Number of eigenvalues of OP to be computed. 0 < NEV < N.
c
c  TOL     Double precision scalar.  (INPUT)
c          Stopping criterion: the relative accuracy of the Ritz value
c          is considered acceptable if BOUNDS(I) .LE. TOL*ABS(RITZ(I)).
c          If TOL .LE. 0. is passed a default is set:
c          DEFAULT = DLAMCH('EPS')  (machine precision as computed
c                    by the LAPACK auxiliary subroutine DLAMCH).
c
c  RESID   Double precision array of length N.  (INPUT/OUTPUT)
c          On INPUT:
c          If INFO .EQ. 0, a random initial residual vector is used.
c          If INFO .NE. 0, RESID contains the initial residual vector,
c                          possibly from a previous run.
c          On OUTPUT:
c          RESID contains the final residual vector.
c
c  NCV     Integer.  (INPUT)
c          Number of columns of the matrix V (less than or equal to N).
c          This will indicate how many Lanczos vectors are generated
c          at each iteration.  After the startup phase in which NEV
c          Lanczos vectors are generated, the algorithm generates
c          NCV-NEV Lanczos vectors at each subsequent update iteration.
c          Most of the cost in generating each Lanczos vector is in the
c          matrix-vector product OP*x. (See remark 4 below).
c
c  V       Double precision N by NCV array.  (OUTPUT)
c          The NCV columns of V contain the Lanczos basis vectors.
c
```

```
c  LDV     Integer.  (INPUT)
c          Leading dimension of V exactly as declared in the calling
c          program.
c
c  IPARAM  Integer array of length 11.  (INPUT/OUTPUT)
c          IPARAM(1) = ISHIFT: method for selecting the implicit shifts.
c          The shifts selected at each iteration are used to restart
c          the Arnoldi iteration in an implicit fashion.
c          -------------------------------------------------------------
c          ISHIFT = 0: the shifts are provided by the user via
c                      reverse communication.  The NCV eigenvalues of
c                      the current tridiagonal matrix T are returned in
c                      the part of WORKL array corresponding to RITZ.
c                      See remark 6 below.
c          ISHIFT = 1: exact shifts with respect to the reduced
c                      tridiagonal matrix T.  This is equivalent to
c                      restarting the iteration with a starting vector
c                      that is a linear combination of Ritz vectors
c                      associated with the "wanted" Ritz values.
c          -------------------------------------------------------------
c
c          IPARAM(2) = LEVEC
c          No longer referenced. See remark 2 below.
c
c          IPARAM(3) = MXITER
c          On INPUT:  maximum number of Arnoldi update iterations allowed.
c          On OUTPUT: actual number of Arnoldi update iterations taken.
c
c          IPARAM(4) = NB: blocksize to be used in the recurrence.
c          The code currently works only for NB = 1.
c
c          IPARAM(5) = NCONV: number of "converged" Ritz values.
c          This represents the number of Ritz values that satisfy
c          the convergence criterion.
c
c          IPARAM(6) = IUPD
c          No longer referenced. Implicit restarting is ALWAYS used.
c
c          IPARAM(7) = MODE
c          On INPUT determines what type of eigenproblem is being solved.
c          Must be 1,2,3,4,5; See under \Description of dsaupd for the
c          five modes available.
c
c          IPARAM(8) = NP
c          When ido = 3 and the user provides shifts through reverse
c          communication (IPARAM(1)=0), dsaupd returns NP, the number
c          of shifts the user is to provide. 0 < NP <=NCV-NEV. See Remark
c          6 below.
c
c          IPARAM(9) = NUMOP, IPARAM(10) = NUMOPB, IPARAM(11) = NUMREO,
c          OUTPUT: NUMOP  = total number of OP*x operations,
```

```
c                 NUMOPB = total number of B*x operations if BMAT='G',
c                 NUMREO = total number of steps of re-orthogonalization.
c
c  IPNTR   Integer array of length 11.  (OUTPUT)
c          Pointer to mark the starting locations in the WORKD and WORKL
c          arrays for matrices/vectors used by the Lanczos iteration.
c          -------------------------------------------------------------
c          IPNTR(1): pointer to the current operand vector X in WORKD.
c          IPNTR(2): pointer to the current result vector Y in WORKD.
c          IPNTR(3): pointer to the vector B * X in WORKD when used in
c                    the shift-and-invert mode.
c          IPNTR(4): pointer to the next available location in WORKL
c                    that is untouched by the program.
c          IPNTR(5): pointer to the NCV by 2 tridiagonal matrix T in WORKL.
c          IPNTR(6): pointer to the NCV RITZ values array in WORKL.
c          IPNTR(7): pointer to the Ritz estimates in array WORKL associated
c                    with the Ritz values located in RITZ in WORKL.
c          Note: IPNTR(8:10) is only referenced by dseupd. See Remark 2.
c          IPNTR(8): pointer to the NCV RITZ values of the original system.
c          IPNTR(9): pointer to the NCV corresponding error bounds.
c          IPNTR(10): pointer to the NCV by NCV matrix of eigenvectors
c                     of the tridiagonal matrix T. Only referenced by
c                     dseupd if RVEC = .TRUE. See Remarks.
c          Note: IPNTR(8:10) is only referenced by dseupd. See Remark 2.
c          IPNTR(11): pointer to the NP shifts in WORKL. See Remark 6 below.
c          -------------------------------------------------------------
c
c  WORKD   Double precision work array of length 3*N.  (REVERSE COMMUNICATION)
c          Distributed array to be used in the basic Arnoldi iteration
c          for reverse communication.  The user should not use WORKD
c          as temporary workspace during the iteration. Upon termination
c          WORKD(1:N) contains B*RESID(1:N). If the Ritz vectors are desired
c          subroutine dseupd uses this output.
c          See Data Distribution Note below.
c
c  WORKL   Double precision work array of length LWORKL.  (OUTPUT/WORKSPACE)
c          Private (replicated) array on each PE or array allocated on
c          the front end.  See Data Distribution Note below.
c
c  LWORKL  Integer.  (INPUT)
c          LWORKL must be at least NCV**2 + 8*NCV .
c
c  INFO    Integer.  (INPUT/OUTPUT)
c          If INFO .EQ. 0, a randomly initial residual vector is used.
c          If INFO .NE. 0, RESID contains the initial residual vector,
c                          possibly from a previous run.
c          Error flag on output.
c          =  0: Normal exit.
c          =  1: Maximum number of iterations taken.
c                All possible eigenvalues of OP has been found. IPARAM(5)
c                returns the number of wanted converged Ritz values.
c          =  2: No longer an informational error. Deprecated starting
c                with release 2 of ARPACK.
c          =  3: No shifts could be applied during a cycle of the
c                Implicitly restarted Arnoldi iteration. One possibility
c                is to increase the size of NCV relative to NEV.
c                See remark 4 below.
c          = -1: N must be positive.
c          = -2: NEV must be positive.
c          = -3: NCV must be greater than NEV and less than or equal to N.
c          = -4: The maximum number of Arnoldi update iterations allowed
c                must be greater than zero.
c          = -5: WHICH must be one of 'LM', 'SM', 'LA', 'SA' or 'BE'.
c          = -6: BMAT must be one of 'I' or 'G'.
c          = -7: Length of private work array WORKL is not sufficient.
c          = -8: Error return from trid. eigenvalue calculation;
c                Informational error from LAPACK routine dsteqr.
c          = -9: Starting vector is zero.
c          = -10: IPARAM(7) must be 1,2,3,4,5.
c          = -11: IPARAM(7) = 1 and BMAT = 'G' are incompatable.
c          = -12: IPARAM(1) must be equal to 0 or 1.
c          = -13: NEV and WHICH = 'BE' are incompatable.
c          = -9999: Could not build an Arnoldi factorization.
c                   IPARAM(5) returns the size of the current Arnoldi
c                   factorization. The user is advised to check that
c                   enough workspace and array storage has been allocated.
c
c
c\Remarks
c  1. The converged Ritz values are always returned in ascending
c     algebraic order.  The computed Ritz values are approximate
c     eigenvalues of OP.  The selection of WHICH should be made
c     with this in mind when Mode = 3,4,5.  After convergence,
c     approximate eigenvalues of the original problem may be obtained
c     with the ARPACK subroutine dseupd.
c
c  2. If the Ritz vectors corresponding to the converged Ritz values
c     are needed, the user must call dseupd immediately following completion
c     of dsaupd. This is new starting with version 2.1 of ARPACK.
c
c  3. If M can be factored into a Cholesky factorization M = LL'
c     then Mode = 2 should not be selected.  Instead one should use
c     Mode = 1 with  OP = inv(L)*A*inv(L').  Appropriate triangular
c     linear systems should be solved with L and L' rather
c     than computing inverses.  After convergence, an approximate
c     eigenvector z of the original problem is recovered by solving
c     L'z = x  where x is a Ritz vector of OP.
c
c  4. At present there is no a-priori analysis to guide the selection
c     of NCV relative to NEV.  The only formal requirement is that NCV > NEV.
c     However, it is recommended that NCV .ge. 2*NEV.  If many problems of
c     the same type are to be solved, one should experiment with increasing
```

```
c     NCV while keeping NEV fixed for a given test problem.  This will
c     usually decrease the required number of OP*x operations but it
c     also increases the work and storage required to maintain the orthogonal
c     basis vectors.   The optimal "cross-over" with respect to CPU time
c     is problem dependent and must be determined empirically.
c
c  5. If IPARAM(7) = 2 then in the Reverse communication interface the user
c     must do the following. When IDO = 1, Y = OP * X is to be computed.
c     When IPARAM(7) = 2 OP = inv(B)*A. After computing A*X the user
c     must overwrite X with A*X. Y is then the solution to the linear set
c     of equations B*Y = A*X.
c
c  6. When IPARAM(1) = 0, and IDO = 3, the user needs to provide the
c     NP = IPARAM(8) shifts in locations:
c     1   WORKL(IPNTR(11))
c     2   WORKL(IPNTR(11)+1)
c                        .
c                        .
c                        .
c     NP  WORKL(IPNTR(11)+NP-1).
c
c     The eigenvalues of the current tridiagonal matrix are located in
c     WORKL(IPNTR(6)) through WORKL(IPNTR(6)+NCV-1). They are in the
c     order defined by WHICH. The associated Ritz estimates are located in
c     WORKL(IPNTR(8)), WORKL(IPNTR(8)+1), ... , WORKL(IPNTR(8)+NCV-1).
c
c-----------------------------------------------------------------------------
```

C.2 DNAUPD

```
c\BeginDoc
c
c\Name: dnaupd
c
c\Description:
c  Reverse communication interface for the Implicitly Restarted Arnoldi
c  iteration. This subroutine computes approximations to a few eigenpairs
c  of a linear operator "OP" with respect to a semi-inner product defined by
c  a symmetric positive semi-definite real matrix B. B may be the identity
c  matrix. NOTE: If the linear operator "OP" is real and symmetric
c  with respect to the real positive semi-definite symmetric matrix B,
c  i.e. B*OP = (OP')*B, then subroutine ssaupd should be used instead.
c
c  The computed approximate eigenvalues are called Ritz values and
c  the corresponding approximate eigenvectors are called Ritz vectors.
c
c  dnaupd is usually called iteratively to solve one of the
c  following problems:
c
c  Mode 1:  A*x = lambda*x.
c           ===> OP = A  and  B = I.
c
c  Mode 2:  A*x = lambda*M*x, M symmetric positive definite
c           ===> OP = inv[M]*A  and  B = M.
c           ===> (If M can be factored see remark 3 below)
c
c  Mode 3:  A*x = lambda*M*x, M symmetric semi-definite
c           ===> OP = Real_Part{ inv[A - sigma*M]*M }  and  B = M.
c           ===> shift-and-invert mode (in real arithmetic)
c           If OP*x = amu*x, then
c           amu = 1/2 * [ 1/(lambda-sigma) + 1/(lambda-conjg(sigma)) ].
c           Note: If sigma is real, i.e. imaginary part of sigma is zero;
c                 Real_Part{ inv[A - sigma*M]*M } == inv[A - sigma*M]*M
c                 amu == 1/(lambda-sigma).
c
c  Mode 4:  A*x = lambda*M*x, M symmetric semi-definite
c           ===> OP = Imaginary_Part{ inv[A - sigma*M]*M }  and  B = M.
c           ===> shift-and-invert mode (in real arithmetic)
c           If OP*x = amu*x, then
c           amu = 1/2i * [ 1/(lambda-sigma) - 1/(lambda-conjg(sigma)) ].
c
c  Both mode 3 and 4 give the same enhancement to eigenvalues close to
c  the (complex) shift sigma.  However, as lambda goes to infinity,
c  the operator OP in mode 4 dampens the eigenvalues more strongly than
c  does OP defined in mode 3.
c
c  NOTE: The action of w <- inv[A - sigma*M]*v or w <- inv[M]*v
c        should be accomplished either by a direct method
c        using a sparse matrix factorization and solving
c
c           [A - sigma*M]*w = v  or M*w = v,
c
c        or through an iterative method for solving these
c        systems.  If an iterative method is used, the
c        convergence test must be more stringent than
c        the accuracy requirements for the eigenvalue
c        approximations.
c
c\Usage:
c  call dnaupd
c     ( IDO, BMAT, N, WHICH, NEV, TOL, RESID, NCV, V, LDV, IPARAM,
c       IPNTR, WORKD, WORKL, LWORKL, INFO )
c
c\Arguments
c  IDO     Integer.  (INPUT/OUTPUT)
c          Reverse communication flag.  IDO must be zero on the first
c          call to dnaupd.  IDO will be set internally to
c          indicate the type of operation to be performed.  Control is
c          then given back to the calling routine which has the
```

```
c          responsibility to carry out the requested operation and call
c          dnaupd with the result.  The operand is given in
c          WORKD(IPNTR(1)), the result must be put in WORKD(IPNTR(2)).
c          -------------------------------------------------------------
c          IDO =  0: first call to the reverse communication interface
c          IDO = -1: compute  Y = OP * X  where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y.
c                    This is for the initialization phase to force the
c                    starting vector into the range of OP.
c          IDO =  1: compute  Y = OP * Z  and Z = B * X where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y,
c                    IPNTR(3) is the pointer into WORKD for Z.
c          IDO =  2: compute  Y = B * X  where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y.
c          IDO =  3: compute the IPARAM(8) real and imaginary parts
c                    of the shifts where INPTR(14) is the pointer
c                    into WORKL for placing the shifts. See Remark
c                    5 below.
c          IDO =  4: compute Z = OP * X
c          IDO = 99: done
c          -------------------------------------------------------------
c          After the initialization phase, when the routine is used in
c          the "shift-and-invert" mode, the vector B * X is already
c          available and does not need to be recomputed in forming OP*X.
c
c  BMAT    Character*1.  (INPUT)
c          BMAT specifies the type of the matrix B that defines the
c          semi-inner product for the operator OP.
c          BMAT = 'I' -> standard eigenvalue problem A*x = lambda*x
c          BMAT = 'G' -> generalized eigenvalue problem A*x = lambda*B*x
c
c  N       Integer.  (INPUT)
c          Dimension of the eigenproblem.
c
c  WHICH   Character*2.  (INPUT)
c          'LM' -> want the NEV eigenvalues of largest magnitude.
c          'SM' -> want the NEV eigenvalues of smallest magnitude.
c          'LR' -> want the NEV eigenvalues of largest real part.
c          'SR' -> want the NEV eigenvalues of smallest real part.
c          'LI' -> want the NEV eigenvalues of largest imaginary part.
c          'SI' -> want the NEV eigenvalues of smallest imaginary part.
c
c  NEV     Integer.  (INPUT)
c          Number of eigenvalues of OP to be computed. 0 < NEV < N-1.
c
c  TOL     Double precision scalar.  (INPUT)
c          Stopping criterion: the relative accuracy of the Ritz value
c          is considered acceptable if BOUNDS(I) .LE. TOL*ABS(RITZ(I))
c          where ABS(RITZ(I)) is the magnitude when RITZ(I) is complex.
c          DEFAULT = DLAMCH('EPS')  (machine precision as computed
c                    by the LAPACK auxiliary subroutine DLAMCH).
c
c  RESID   Double precision array of length N.  (INPUT/OUTPUT)
c          On INPUT:
c          If INFO .EQ. 0, a random initial residual vector is used.
c          If INFO .NE. 0, RESID contains the initial residual vector,
c                          possibly from a previous run.
c          On OUTPUT:
c          RESID contains the final residual vector.
c
c  NCV     Integer.  (INPUT)
c          Number of columns of the matrix V. NCV must satisfy the two
c          inequalities 2 <= NCV-NEV and NCV <= N.
c          This will indicate how many Arnoldi vectors are generated
c          at each iteration.  After the startup phase in which NEV
c          Arnoldi vectors are generated, the algorithm generates
c          approximately NCV-NEV Arnoldi vectors at each subsequent update
c          iteration. Most of the cost in generating each Arnoldi vector is
c          in the matrix-vector operation OP*x.
c          NOTE: 2 <= NCV-NEV in order that complex conjugate pairs of Ritz
c          values are kept together. (See remark 4 below)
c
c  V       Double precision array N by NCV.  (OUTPUT)
c          Contains the final set of Arnoldi basis vectors.
c
c  LDV     Integer.  (INPUT)
c          Leading dimension of V exactly as declared in the calling program.
c
c  IPARAM  Integer array of length 11.  (INPUT/OUTPUT)
c          IPARAM(1) = ISHIFT: method for selecting the implicit shifts.
c          The shifts selected at each iteration are used to restart
c          the Arnoldi iteration in an implicit fashion.
c          -------------------------------------------------------------
c          ISHIFT = 0: the shifts are provided by the user via
c                      reverse communication.  The real and imaginary
c                      parts of the NCV eigenvalues of the Hessenberg
c                      matrix H are returned in the part of the WORKL
c                      array corresponding to RITZR and RITZI. See remark
c                      5 below.
c          ISHIFT = 1: exact shifts with respect to the current
c                      Hessenberg matrix H.  This is equivalent to
c                      restarting the iteration with a starting vector
c                      that is a linear combination of approximate Schur
c                      vectors associated with the "wanted" Ritz values.
c          -------------------------------------------------------------
c
c          IPARAM(2) = No longer referenced.
c
c          IPARAM(3) = MXITER
```

```
c          On INPUT:  maximum number of Arnoldi update iterations allowed.
c          On OUTPUT: actual number of Arnoldi update iterations taken.
c
c          IPARAM(4) = NB: blocksize to be used in the recurrence.
c          The code currently works only for NB = 1.
c
c          IPARAM(5) = NCONV: number of "converged" Ritz values.
c          This represents the number of Ritz values that satisfy
c          the convergence criterion.
c
c          IPARAM(6) = IUPD
c          No longer referenced. Implicit restarting is ALWAYS used.
c
c          IPARAM(7) = MODE
c          On INPUT determines what type of eigenproblem is being solved.
c          Must be 1,2,3,4; See under \Description of dnaupd for the
c          four modes available.
c
c          IPARAM(8) = NP
c          When ido = 3 and the user provides shifts through reverse
c          communication (IPARAM(1)=0), dnaupd returns NP, the number
c          of shifts the user is to provide. 0 < NP <=NCV-NEV. See Remark
c          5 below.
c
c          IPARAM(9) = NUMOP, IPARAM(10) = NUMOPB, IPARAM(11) = NUMREO,
c          OUTPUT: NUMOP  = total number of OP*x operations,
c                  NUMOPB = total number of B*x operations if BMAT='G',
c                  NUMREO = total number of steps of re-orthogonalization.
c
c  IPNTR   Integer array of length 14.  (OUTPUT)
c          Pointer to mark the starting locations in the WORKD and WORKL
c          arrays for matrices/vectors used by the Arnoldi iteration.
c          -------------------------------------------------------------
c          IPNTR(1): pointer to the current operand vector X in WORKD.
c          IPNTR(2): pointer to the current result vector Y in WORKD.
c          IPNTR(3): pointer to the vector B * X in WORKD when used in
c                    the shift-and-invert mode.
c          IPNTR(4): pointer to the next available location in WORKL
c                    that is untouched by the program.
c          IPNTR(5): pointer to the NCV by NCV upper Hessenberg matrix
c                    H in WORKL.
c          IPNTR(6): pointer to the real part of the ritz value array
c                    RITZR in WORKL.
c          IPNTR(7): pointer to the imaginary part of the ritz value array
c                    RITZI in WORKL.
c          IPNTR(8): pointer to the Ritz estimates in array WORKL associated
c                    with the Ritz values located in RITZR and RITZI in WORKL.
c
c          Note: IPNTR(9:13) is only referenced by dneupd. See Remark 2 below.
c
c          IPNTR(9): pointer to the real part of the NCV RITZ values of the
c                    original system.
c          IPNTR(10): pointer to the imaginary part of the NCV RITZ values of
c                     the original system.
c          IPNTR(11): pointer to the NCV corresponding error bounds.
c          IPNTR(12): pointer to the NCV by NCV upper quasi-triangular
c                     Schur matrix for H.
c          IPNTR(13): pointer to the NCV by NCV matrix of eigenvectors
c                     of the upper Hessenberg matrix H. Only referenced by
c                     dneupd if RVEC = .TRUE. See Remark 2 below.
c          Note: IPNTR(9:13) is only referenced by dneupd. See Remark 2 below.
c          IPNTR(14): pointer to the NP shifts in WORKL. See Remark 5 below.
c          -------------------------------------------------------------
c
c  WORKD   Double precision work array of length 3*N.  (REVERSE COMMUNICATION)
c          Distributed array to be used in the basic Arnoldi iteration
c          for reverse communication.  The user should not use WORKD
c          as temporary workspace during the iteration. Upon termination
c          WORKD(1:N) contains B*RESID(1:N). If an invariant subspace
c          associated with the converged Ritz values is desired, see remark
c          2 below, subroutine dneupd uses this output.
c          See Data Distribution Note below.
c
c  WORKL   Double precision work array of length LWORKL.  (OUTPUT/WORKSPACE)
c          Private (replicated) array on each PE or array allocated on
c          the front end.  See Data Distribution Note below.
c
c  LWORKL  Integer.  (INPUT)
c          LWORKL must be at least 3*NCV**2 + 6*NCV.
c
c  INFO    Integer.  (INPUT/OUTPUT)
c          If INFO .EQ. 0, a randomly initial residual vector is used.
c          If INFO .NE. 0, RESID contains the initial residual vector,
c                          possibly from a previous run.
c          Error flag on output.
c          =  0: Normal exit.
c          =  1: Maximum number of iterations taken.
c                All possible eigenvalues of OP has been found. IPARAM(5)
c                returns the number of wanted converged Ritz values.
c          =  2: No longer an informational error. Deprecated starting
c                with release 2 of ARPACK.
c          =  3: No shifts could be applied during a cycle of the
c                Implicitly restarted Arnoldi iteration. One possibility
c                is to increase the size of NCV relative to NEV.
c                See remark 4 below.
c          = -1: N must be positive.
c          = -2: NEV must be positive.
c          = -3: NCV-NEV >= 2 and less than or equal to N.
c          = -4: The maximum number of Arnoldi update iteration
c                must be greater than zero.
c          = -5: WHICH must be one of 'LM', 'SM', 'LR', 'SR', 'LI', 'SI'
c          = -6: BMAT must be one of 'I' or 'G'.
```

```
c          = -7: Length of private work array is not sufficient.
c          = -8: Error return from LAPACK eigenvalue calculation;
c          = -9: Starting vector is zero.
c          = -10: IPARAM(7) must be 1,2,3,4.
c          = -11: IPARAM(7) = 1 and BMAT = 'G' are incompatable.
c          = -12: IPARAM(1) must be equal to 0 or 1.
c          = -9999: Could not build an Arnoldi factorization.
c                   IPARAM(5) returns the size of the current Arnoldi
c                   factorization.
c
c\Remarks
c  1. The computed Ritz values are approximate eigenvalues of OP. The
c     selection of WHICH should be made with this in mind when
c     Mode = 3 and 4.  After convergence, approximate eigenvalues of the
c     original problem may be obtained with the ARPACK subroutine dneupd.
c
c  2. If a basis for the invariant subspace corresponding to the converged Ritz
c     values is needed, the user must call dneupd immediately following
c     completion of dnaupd. This is new starting with release 2 of ARPACK.
c
c  3. If M can be factored into a Cholesky factorization M = LL'
c     then Mode = 2 should not be selected.  Instead one should use
c     Mode = 1 with  OP = inv(L)*A*inv(L').  Appropriate triangular
c     linear systems should be solved with L and L' rather
c     than computing inverses.  After convergence, an approximate
c     eigenvector z of the original problem is recovered by solving
c     L'z = x  where x is a Ritz vector of OP.
c
c  4. At present there is no a-priori analysis to guide the selection of NCV
c     relative to NEV.  The only formal requirement is that NCV > NEV + 2.
c     However, it is recommended that NCV .ge. 2*NEV+1.  If many problems of
c     the same type are to be solved, one should experiment with increasing
c     NCV while keeping NEV fixed for a given test problem.  This will
c     usually decrease the required number of OP*x operations but it
c     also increases the work and storage required to maintain the orthogonal
c     basis vectors.  The optimal "cross-over" with respect to CPU time
c     is problem dependent and must be determined empirically.
c     See Chapter 8 of Reference 2 for further information.
c
c  5. When IPARAM(1) = 0, and IDO = 3, the user needs to provide the
c     NP = IPARAM(8) real and imaginary parts of the shifts in locations
c         real part                  imaginary part
c         -----------------------    --------------
c     1   WORKL(IPNTR(14))           WORKL(IPNTR(14)+NP)
c     2   WORKL(IPNTR(14)+1)         WORKL(IPNTR(14)+NP+1)
c                        .                          .
c                        .                          .
c                        .                          .
c     NP  WORKL(IPNTR(14)+NP-1)      WORKL(IPNTR(14)+2*NP-1).
c
c     Only complex conjugate pairs of shifts may be applied and the pairs
c     must be placed in consecutive locations. The real part of the
c     eigenvalues of the current upper Hessenberg matrix are located in
c     WORKL(IPNTR(6)) through WORKL(IPNTR(6)+NCV-1) and the imaginary part
c     in WORKL(IPNTR(7)) through WORKL(IPNTR(7)+NCV-1). They are ordered
c     according to the order defined by WHICH. The complex conjugate
c     pairs are kept together and the associated Ritz estimates are located in
c     WORKL(IPNTR(8)), WORKL(IPNTR(8)+1), ... , WORKL(IPNTR(8)+NCV-1).
c
c-----------------------------------------------------------------------
```

C.3 ZNAUPD

```
c\BeginDoc
c
c\Name: znaupd
c
c\Description:
c  Reverse communication interface for the Implicitly Restarted Arnoldi
c  iteration. This is intended to be used to find a few eigenpairs of a
c  complex linear operator OP with respect to a semi-inner product defined
c  by a hermitian positive semi-definite real matrix B. B may be the identity
c  matrix.  NOTE: if both OP and B are real, then dsaupd or dnaupd should
c  be used.
c
c
c  The computed approximate eigenvalues are called Ritz values and
c  the corresponding approximate eigenvectors are called Ritz vectors.
c
c  znaupd is usually called iteratively to solve one of the
c  following problems:
c
c  Mode 1:  A*x = lambda*x.
c           ===> OP = A  and  B = I.
c
c  Mode 2:  A*x = lambda*M*x, M symmetric positive definite
c           ===> OP = inv[M]*A  and  B = M.
c           ===> (If M can be factored see remark 3 below)
c
c  Mode 3:  A*x = lambda*M*x, M symmetric semi-definite
c           ===> OP =  inv[A - sigma*M]*M   and  B = M.
c           ===> shift-and-invert mode
c           If OP*x = amu*x, then lambda = sigma + 1/amu.
c
c
c  NOTE: The action of w <- inv[A - sigma*M]*v or w <- inv[M]*v
c        should be accomplished either by a direct method
c        using a sparse matrix factorization and solving
c
```

```
c            [A - sigma*M]*w = v  or M*w = v,
c
c         or through an iterative method for solving these
c         systems.  If an iterative method is used, the
c         convergence test must be more stringent than
c         the accuracy requirements for the eigenvalue
c         approximations.
c
c\Usage:
c  call znaupd
c     ( IDO, BMAT, N, WHICH, NEV, TOL, RESID, NCV, V, LDV, IPARAM,
c       IPNTR, WORKD, WORKL, LWORKL, RWORK, INFO )
c
c\Arguments
c  IDO     Integer.  (INPUT/OUTPUT)
c          Reverse communication flag.  IDO must be zero on the first
c          call to znaupd.  IDO will be set internally to
c          indicate the type of operation to be performed.  Control is
c          then given back to the calling routine which has the
c          responsibility to carry out the requested operation and call
c          znaupd with the result.  The operand is given in
c          WORKD(IPNTR(1)), the result must be put in WORKD(IPNTR(2)).
c          -------------------------------------------------------------
c          IDO =  0: first call to the reverse communication interface
c          IDO = -1: compute  Y = OP * X  where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y.
c                    This is for the initialization phase to force the
c                    starting vector into the range of OP.
c          IDO =  1: compute  Y = OP * Z  and Z = B * X where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y,
c                    IPNTR(3) is the pointer into WORKD for Z.
c          IDO =  2: compute  Y = M * X  where
c                    IPNTR(1) is the pointer into WORKD for X,
c                    IPNTR(2) is the pointer into WORKD for Y.
c          IDO =  3: compute and return the shifts in the first
c                    NP locations of WORKL.
c          IDO =  4: compute Z = OP * X
c          IDO = 99: done
c          -------------------------------------------------------------
c          After the initialization phase, when the routine is used in
c          the "shift-and-invert" mode, the vector M * X is already
c          available and does not need to be recomputed in forming OP*X.
c
c  BMAT    Character*1.  (INPUT)
c          BMAT specifies the type of the matrix B that defines the
c          semi-inner product for the operator OP.
c          BMAT = 'I' -> standard eigenvalue problem A*x = lambda*x
c          BMAT = 'G' -> generalized eigenvalue problem A*x = lambda*M*x
c
c  N       Integer.  (INPUT)
c          Dimension of the eigenproblem.
c
c  WHICH   Character*2.  (INPUT)
c          'LM' -> want the NEV eigenvalues of largest magnitude.
c          'SM' -> want the NEV eigenvalues of smallest magnitude.
c          'LR' -> want the NEV eigenvalues of largest real part.
c          'SR' -> want the NEV eigenvalues of smallest real part.
c          'LI' -> want the NEV eigenvalues of largest imaginary part.
c          'SI' -> want the NEV eigenvalues of smallest imaginary part.
c
c  NEV     Integer.  (INPUT)
c          Number of eigenvalues of OP to be computed. 0 < NEV < N-1.
c
c  TOL     Double precision  scalar.  (INPUT)
c          Stopping criteria: the relative accuracy of the Ritz value
c          is considered acceptable if BOUNDS(I) .LE. TOL*ABS(RITZ(I))
c          where ABS(RITZ(I)) is the magnitude when RITZ(I) is complex.
c          DEFAULT = dlamch('EPS')  (machine precision as computed
c                    by the LAPACK auxiliary subroutine dlamch).
c
c  RESID   Complex*16 array of length N.  (INPUT/OUTPUT)
c          On INPUT:
c          If INFO .EQ. 0, a random initial residual vector is used.
c          If INFO .NE. 0, RESID contains the initial residual vector,
c                          possibly from a previous run.
c          On OUTPUT:
c          RESID contains the final residual vector.
c
c  NCV     Integer.  (INPUT)
c          Number of columns of the matrix V. NCV must satisfy the two
c          inequalities 2 <= NCV-NEV and NCV <= N.
c          This will indicate how many Arnoldi vectors are generated
c          at each iteration.  After the startup phase in which NEV
c          Arnoldi vectors are generated, the algorithm generates
c          approximately NCV-NEV Arnoldi vectors at each subsequent update
c          iteration. Most of the cost in generating each Arnoldi vector is
c          in the matrix-vector operation OP*x.
c          NOTE: 2 <= NCV-NEV in order that complex conjugate pairs of Ritz
c          values are kept together. (See remark 4 below)
c
c  V       Complex*16 array N by NCV.  (OUTPUT)
c          Contains the final set of Arnoldi basis vectors.
c
c  LDV     Integer.  (INPUT)
c          Leading dimension of V exactly as declared in the calling program.
c
c  IPARAM  Integer array of length 11.  (INPUT/OUTPUT)
c          IPARAM(1) = ISHIFT: method for selecting the implicit shifts.
c          The shifts selected at each iteration are used to filter out
c          the components of the unwanted eigenvector.
```

```
c          -------------------------------------------------------------
c          ISHIFT = 0: the shifts are to be provided by the user via
c                      reverse communication.  The NCV eigenvalues of
c                      the Hessenberg matrix H are returned in the part
c                      of WORKL array corresponding to RITZ.
c          ISHIFT = 1: exact shifts with respect to the current
c                      Hessenberg matrix H.  This is equivalent to
c                      restarting the iteration from the beginning
c                      after updating the starting vector with a linear
c                      combination of Ritz vectors associated with the
c                      "wanted" eigenvalues.
c          ISHIFT = 2: other choice of internal shift to be defined.
c          -------------------------------------------------------------
c
c          IPARAM(2) = No longer referenced
c
c          IPARAM(3) = MXITER
c          On INPUT:  maximum number of Arnoldi update iterations allowed.
c          On OUTPUT: actual number of Arnoldi update iterations taken.
c
c          IPARAM(4) = NB: blocksize to be used in the recurrence.
c          The code currently works only for NB = 1.
c
c          IPARAM(5) = NCONV: number of "converged" Ritz values.
c          This represents the number of Ritz values that satisfy
c          the convergence criterion.
c
c          IPARAM(6) = IUPD
c          No longer referenced. Implicit restarting is ALWAYS used.
c
c          IPARAM(7) = MODE
c          On INPUT determines what type of eigenproblem is being solved.
c          Must be 1,2,3,4; See under \Description of znaupd for the
c          four modes available.
c
c          IPARAM(8) = NP
c          When ido = 3 and the user provides shifts through reverse
c          communication (IPARAM(1)=0), _naupd returns NP, the number
c          of shifts the user is to provide. 0 < NP < NCV-NEV.
c
c          IPARAM(9) = NUMOP, IPARAM(10) = NUMOPB, IPARAM(11) = NUMREO,
c          OUTPUT: NUMOP  = total number of OP*x operations,
c                  NUMOPB = total number of B*x operations if BMAT='G',
c                  NUMREO = total number of steps of re-orthogonalization.
c
c  IPNTR   Integer array of length 14.  (OUTPUT)
c          Pointer to mark the starting locations in the WORKD and WORKL
c          arrays for matrices/vectors used by the Arnoldi iteration.
c          -------------------------------------------------------------
c          IPNTR(1): pointer to the current operand vector X in WORKD.
c          IPNTR(2): pointer to the current result vector Y in WORKD.
c          IPNTR(3): pointer to the vector B * X in WORKD when used in
c                    the shift-and-invert mode.
c          IPNTR(4): pointer to the next available location in WORKL
c                    that is untouched by the program.
c          IPNTR(5): pointer to the NCV by NCV upper Hessenberg
c                    matrix H in WORKL.
c          IPNTR(6): pointer to the  ritz value array  RITZ
c          IPNTR(7): pointer to the (projected) ritz vector array Q
c          IPNTR(8): pointer to the error BOUNDS array in WORKL.
c          Note: IPNTR(9:13) is only referenced by zneupd. See Remark 2 below.
c          IPNTR(9): pointer to the NCV RITZ values of the
c                    original system.
c          IPNTR(10): Not Used
c          IPNTR(11): pointer to the NCV corresponding error bounds.
c          IPNTR(14): pointer to the NP shifts in WORKL. See Remark 5 below.
c          -------------------------------------------------------------
c
c  WORKD   Complex*16 work array of length 3*N.  (REVERSE COMMUNICATION)
c          Distributed array to be used in the basic Arnoldi iteration
c          for reverse communication.  The user should not use WORKD
c          as temporary workspace during the iteration !!!!!!!!!!
c          See Data Distribution Note below.
c
c  WORKL   Complex*16 work array of length LWORKL.  (OUTPUT/WORKSPACE)
c          Private (replicated) array on each PE or array allocated on
c          the front end.  See Data Distribution Note below.
c
c  LWORKL  Integer.  (INPUT)
c          LWORKL must be at least 3*NCV**2 + 5*NCV.
c
c  RWORK   Double precision  work array of length NCV (WORKSPACE)
c          Private (replicated) array on each PE or array allocated on
c          the front end.
c
c
c  INFO    Integer.  (INPUT/OUTPUT)
c          If INFO .EQ. 0, a randomly initial residual vector is used.
c          If INFO .NE. 0, RESID contains the initial residual vector,
c                          possibly from a previous run.
c          Error flag on output.
c          =  0: Normal exit.
c          =  1: Maximum number of iterations taken.
c                All possible eigenvalues of OP has been found. IPARAM(5)
c                returns the number of wanted converged Ritz values.
c          =  2: No longer an informational error. Deprecated starting
c                with release 2 of ARPACK.
c          =  3: No shifts could be applied during a cycle of the
c                Implicitly restarted Arnoldi iteration. One possibility
c                is to increase the size of NCV relative to NEV.
c                See remark 4 below.
c          = -1: N must be positive.
```

```
c          = -2: NEV must be positive.
c          = -3: NCV-NEV >= 2 and less than or equal to N.
c          = -4: The maximum number of Arnoldi update iteration
c                must be greater than zero.
c          = -5: WHICH must be one of 'LM', 'SM', 'LR', 'SR', 'LI', 'SI'
c          = -6: BMAT must be one of 'I' or 'G'.
c          = -7: Length of private work array is not sufficient.
c          = -8: Error return from LAPACK eigenvalue calculation;
c          = -9: Starting vector is zero.
c          = -10: IPARAM(7) must be 1,2,3.
c          = -11: IPARAM(7) = 1 and BMAT = 'G' are incompatable.
c          = -12: IPARAM(1) must be equal to 0 or 1.
c          = -9999: Could not build an Arnoldi factorization.
c                   User input error highly likely.  Please
c                   check actual array dimensions and layout.
c                   IPARAM(5) returns the size of the current Arnoldi
c                   factorization.
c
c\Remarks
c  1. The computed Ritz values are approximate eigenvalues of OP. The
c     selection of WHICH should be made with this in mind when using
c     Mode = 3.  When operating in Mode = 3 setting WHICH = 'LM' will
c     compute the NEV eigenvalues of the original problem that are
c     closest to the shift SIGMA . After convergence, approximate eigenvalues
c     of the original problem may be obtained with the ARPACK subroutine zneupd.
c
c  2. If a basis for the invariant subspace corresponding to the converged Ritz
c     values is needed, the user must call zneupd immediately following
c     completion of znaupd. This is new starting with release 2 of ARPACK.
c
c  3. If M can be factored into a Cholesky factorization M = LL'
c     then Mode = 2 should not be selected.  Instead one should use
c     Mode = 1 with  OP = inv(L)*A*inv(L').  Appropriate triangular
c     linear systems should be solved with L and L' rather
c     than computing inverses.  After convergence, an approximate
c     eigenvector z of the original problem is recovered by solving
c     L'z = x  where x is a Ritz vector of OP.
c
c  4. At present there is no a-priori analysis to guide the selection of NCV
c     relative to NEV.  The only formal requirement is that NCV > NEV + 2.
c     However, it is recommended that NCV .ge. 2*NEV+1.  If many problems of
c     the same type are to be solved, one should experiment with increasing
c     NCV while keeping NEV fixed for a given test problem.  This will
c     usually decrease the required number of OP*x operations but it
c     also increases the work and storage required to maintain the orthogonal
c     basis vectors.  The optimal "cross-over" with respect to CPU time
c     is problem dependent and must be determined empirically.
c     See Chapter 8 of Reference 2 for further information.
c
c  5. When IPARAM(1) = 0, and IDO = 3, the user needs to provide the
c     NP = IPARAM(8) complex shifts in locations
c     WORKL(IPNTR(14)), WORKL(IPNTR(14)+1), ... , WORKL(IPNTR(14)+NP).
c     Eigenvalues of the current upper Hessenberg matrix are located in
c     WORKL(IPNTR(6)) through WORKL(IPNTR(6)+NCV-1). They are ordered
c     according to the order defined by WHICH.  The associated Ritz estimates
c     are located in WORKL(IPNTR(8)), WORKL(IPNTR(8)+1), ... ,
c     WORKL(IPNTR(8)+NCV-1).
c
c-----------------------------------------------------------------------
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[2] J. J. Dongarra, J. DuCroz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[3] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson. An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. on Math. Software*, 14(1):1–17, 1988.

[4] R. G. Grimes, J. G. Lewis, and H. D. Simon. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Analysis and Applications*, 15(1):228–272, January 1994.

[5] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[6] R. B. Lehoucq. *Analysis and Implementation of an Implicitly Restarted Iteration*. PhD thesis, Rice University, Houston, Texas, May 1995. Also available as Technical Report TR95-13, Dept. of Computational and Applied Mathematics.

[7] R. B. Lehoucq and D. C. Sorensen. Deflation techniques for an implicitly restarted Arnoldi iteration. *SIAM J. Matrix Analysis and Applications*, 17(4):789–821, October 1996.

[8] K. J. Maschhoff and D. C. Sorensen. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In Jerzy Wasniewski, Jack Dongarra, Kaj Madsen, and Dorte Olesen, editors, *Applied Parallel Computing in Industrial Problems and Optimization*, volume 1184 of *Lecture Notes in Computer Science*, Berlin, 1996. Springer–Verlag.

[9] K. Meerbergen and D. Roose. Matrix transformations for computing rightmost eigenvalues of large sparse non-symmetric eigenvalue problems, 1996.

[10] K. Meerbergen, A. Spence, and D. Roose. Shift-invert and Cayley transforms for the detection of rightmost eigenvalues of nonsymmetric matrices. *BIT*, 34:409–423, 1994.

[11] Y. Saad. Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems. *Mathematics of Computation*, 42:567–588, 1984.

[12] Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. Halsted Press, 1992.

[13] D. C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Analysis and Applications*, 13(1):357–385, January 1992.

Index