University of Maryland College Park

Institute for Advanced Computer Studies TR–2006-07

Department of Computer Science TR–4783

EIGENTEST: A Test Matrix Generator for Large-Scale Eigenproblems [*]

Che Rung Lee[†]

G. W. Stewart[†]

February, 2006

ABSTRACT

Eigentest is a package that produces real test matrices with known eigensystems. A test matrix, called an eigenmat, is generated in a factored form, in which the user can specify the eigenvalues and has some control over the condition of the eigenvalues and eigenvectors. An eigenmat A of order n requires only $O(n)$ storage for its representation. Auxiliary programs permit the computation of $(A - sI)b$, $(A - sI)^{\mathrm{T}}b$, $(A - sI)^{-1}b$, and $(A - sI)^{-\mathrm{T}}b$ in $O(n)$ operations. A special routine computes specified eigenvectors of an eigenmat and the condition of its eigenvalue. Thus eigenmats are suitable for testing algorithms based on Krylov sequences, as well as others based on matrix-vector products. This paper introduces the eigenmat and describes implementations in Fortran 77, Fortran 95, C, and Matlab.

[*]This report is available by anonymous ftp from `thales.cs.umd.edu` in the directory `pub/reports` or on the web at `http://www.cs.umd.edu/~stewart/`.

[†]Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. This work was supported in part by the National Science Foundation under grant CCR0204084.

EIGENTEST: A Test Matrix Generator for Large-Scale Eigenproblems

Che Rung Lee
G. W. Stewart

ABSTRACT

Eigentest is a package that produces real test matrices with known eigensystems. A test matrix, called an eigenmat, is generated in a factored form, in which the user can specify the eigenvalues and has some control over the condition of the eigenvalues and eigenvectors. An eigenmat A of order n requires only $O(n)$ storage for its representation. Auxiliary programs permit the computation of $(A - sI)b$, $(A - sI)^{\mathrm{T}}b$, $(A - sI)^{-1}b$, and $(A - sI)^{-\mathrm{T}}b$ in $O(n)$ operations. A special routine computes specified eigenvectors of an eigenmat and the condition of its eigenvalue. Thus eigenmats are suitable for testing algorithms based on Krylov sequences, as well as others based on matrix-vector products. This paper introduces the eigenmat and describes implementations in Fortran 77, Fortran 95, C, and Matlab.

1. Introduction

A recurring problem in developing algorithms for large-scale eigenproblems is how to find test matrices with known eigensystems. Matrices from applications can be useful, but they have a number of drawbacks. First, the list of publicly available test matrices is not long. Second, such matrices come in a variety of formats, which must be mastered in order to use them. Third, the complete eigensystem of such a matrix is seldom known. The reason is that time and storage considerations make it impossible to compute the entire eigensystem, at least for a large matrix. Fourth, the properties of the eigensystem are fixed. The developer cannot change the distribution of the eigenvalues or the condition of the eigenvalues and eigenvectors. Finally, many eigensystem algorithms use shift-and-invert techniques that are expensive to implement with matrices from applications. This problem will not go away in real-life; but in developing algorithms, shift-and-invert techniques should be testable at low cost.

This paper describes a package, Eigentest, that generates real matrices, called eigenmats, with prespecified distributions of eigenvalues and with some control over the condition of the eigenvalues and eigenvectors. The matrices are generated in a factored form that permits the fast implementation of matrix-vector multiplication, with or without shifts and inversions. The storage required by an eigenmat of order n is $O(n)$, as is the work to perform the various multiplications. Thus Eigentest is specially suitable for testing algorithms based on matrix vector products — in particular Krylov sequence algorithms and their relatives. The test matrices are not sparse, unlike most matrices encountered in applications. But sparsity or lack thereof does not affect the convergence of most algorithms for large-scale eigenproblems — only the economics of using them.

We give implementations in C, Fortran 95, Matlab, and Fortran 77. We restrict ourselves to real matrices (possibly with complex conjugate eigenvalues) because the Matlab and Fortran programs can be easily adapted to the complex case. C is another story, since it does not have a complex type.

In the next section we describe the factored form of the test matrix. In Section 3 we show how to compute eigenvectors of the matrix and how to compute condition numbers of the eigenvalues and bound the condition of the eigenvectors. We also discuss the accuracy of the computed products. In Section 4 we sketch the structures used in the four different versions. Since C, Matlab, and Fortran 95 all have the ability to define structures, we use C to describe the basic structures and operations in detail, and then sketch the Matlab and Fortran 95 versions. Unfortunately, Fortran 77 has no user defined structures, and we describe in detail how the general structures for the other languages is transcribed into two Fortran arrays.

Throughout this paper $\|\cdot\|$ denotes the Euclidean vector norm and its subordinate matrix 2-norm [1].

2. The eigenmat

Our test matrix A — called an *eigenmat* — is the result applying two similarity transformations to a block diagonal matrix of order n that contains the eigenvalues of A. The matrix is not formed explicitly. Instead the block diagonal matrix and the similarity transformations are maintained and applied separately to compute products such as Ab, $(A-sI)^{-1}b$, etc. The transformations are contrived to make this process efficient — $O(n)$ in both storage and operation counts.

In describing the factorization it will be convenient to work from the inside out, beginning with the block diagonal matrix of eigenvalues.

The core of our test matrix is the block diagonal matrix

$$L = \mathrm{diag}(L_1, L_2, \ldots, L_m)$$

of order n. Each block L_i is of order one or two. The blocks $L_i = \lambda_i$ of order one comprise the real eigenvalues of A. The blocks of order two have the form

$$L_i = \begin{pmatrix} \mu_i & \nu_i \\ -\nu_i & \mu_i \end{pmatrix}.$$

This is a normal matrix, whose eigenvalues are are $\mu_i \pm \nu_i i$ with eigenvectors

$$\begin{pmatrix} 1 \\ \pm i \end{pmatrix}.$$

In Eigentest the matrix L is represented by a floating-point array `eig` and an integer array `type` both of order `n`. The elements of `eig` consist of the numbers λ_i and the pairs (μ_i, ν_i) that define the blocks L_i. The the `i`th entry of `type` is 1, 2, or 3, depending on whether the corresponding element of `eig` is a λ, a μ, or a ν. In `eig`, a μ must always be followed by its ν.

The application of $L - s{*}I$ to a vector is done blockwise. The $1{\times}1$ blocks each require 2 floating-point operations — one of them a division when the inverse matrix is to be applied. The direct application $2{\times}2$ blocks requires 8 floating-point operations. Inversion, if required, is done by Gaussian elimination with partial pivoting with a count of 10 floating-point operations. Thus the worst case — inversion with only $2{\times}2$ blocks — requires $5n$ floating-point operations.

The eigenvalues of L are perfectly conditioned in the sense that a perturbation of size bounded by ϵ in the elements of L change the eigenvalues by quantities bounded by ϵ. To produce groups of eigenvalues with varying degrees of ill-conditioning we proceed as follows.

Partition L in the form

$$L = \mathrm{diag}(B_1, B_2, \ldots, B_{\mathrm{nblocks}}), \tag{2.1}$$

where B_j is of order k_j, and apply a similarity transformation $Z_j B_j Z_j^{-1} = M_j$ to each block to give

$$M = (M_1 \ \cdots \ M_{\mathrm{nblocks}}) = ZLZ^{-1}.$$

where

$$Z = \mathrm{diag}(Z_1, \ldots, Z_{\mathrm{nblocks}}).$$

Let

$$\kappa_j = \|Z_j\|\|Z_j^{-1}\|.$$

A special case of the Bauer–Fike theorem [1, Theorem 7.2.2] then states that for any matrix E_j if λ is an eigenvalue of B_j then there is an eigenvalue $\tilde{\lambda}_j$ of $\tilde{B}_j = B_j + E_j$ such that

$$|\tilde{\lambda}_j - \lambda_j| \le \kappa_j \|E_j\|. \tag{2.2}$$

The number κ_j is called the *condition number with respect to inversion* of the matrix Z_j. The inequality (2.2) shows that these numbers are related to the sensitivity of the eigenvalues of M_j. The larger κ_j is, the more sensitive we can expect the eigenvalues of M_j to be. (However, as we will see later, κ_j is not a condition number for the individual eigenvalues of M_j.)

Thus the problem of controlling the condition of the eigenvalues of a block B_j amounts to determining a Z_j with known κ_j. In addition, it must be easy to multiply Z_j and Z_j^{-1} into a vector. With these conditions in mind, we will take Z_j in the form

$$Z_j = (I - u_j u_j^{\mathrm{T}})\Sigma_j(I - v_j v_j^{\mathrm{T}}) \equiv U_j \Sigma_j V_j \tag{2.3}$$

where

$$\|u_j\| = \|v_j\| = \sqrt{2}$$

and

$$\Sigma_j = \mathrm{diag}(\sigma_1^{(j)}, \ldots, \sigma_{k_j}^{(j)}), \qquad \sigma_i^{(j)} > 0 \ (i = 1, \ldots, k_j).$$

The matrices U_j and V_j are orthogonal matrices called *Householder transformations.* The scalars $\sigma_i^{(j)}$ are called the *singular values* of Z_j. Thus we will call the a matrix of the form (2.3) a *Householder SVD matrix* or, for short, an *hsvdmat.*

The condition number of Z_j is

$$\kappa(Z_j) = \frac{\max_i \sigma_i^{(j)}}{\min_i \sigma_i^{(j)}}.$$

Thus by setting adjusting the largest and smallest values of the $\sigma_i^{(j)}$ we can control the condition of Z_j.

The matrix Z_j requires $3k_j$ floating-point words to store the vectors u_j and v_j and the diagonal of Σ_j. The product $c = Z_j b$ can be formed by the following algorithm.

```
1. $c = b$
2. $s = v_j^{\mathrm{T}} c$
3. $c = c - s{*}v_j$
4. $c = \Sigma_j c$
5. $s = u_j^{\mathrm{T}} c$
6. $c = c - s{*}u_j$
```

This algorithm requires about $9k_j$ floating-point operations. Since $k_1 + \cdots + k_{\mathrm{nblocks}} = n$, the matrix Z requires $3n$ floating-point words to store, and the computation of $c = Zb$ costs $9n$ floating-point operations. The computation of $Z^{\mathrm{T}}b$, $Z^{-1}b$ and $Z^{-\mathrm{T}}b$ can be done by analogous algorithms with the same operation counts.

The matrix Z is stored as follows. The vectors u_j are packed in a floating-point array `u` of length n in their natural order. Likewise, the vectors v_j are packed in a floating-point array `v`, and the singular values σ_i are stored in a floating-point array `sig`. These arrays are accompanied by an integer array `bs` (for block start) of length `nblocks+1`. The absolute value of `j`th entry of `bx` contains the starting index for the `j`th block. The absolute value of the last entry is n or $n+1$, depending on whether the base index is zero (C) or one (Fortran and Matlab). Thus the statement

```
for i=abs(bs(j): abs(bs[j+1])-1 \\
   u(i) \\
end
```

traverses the vector u_j.

As suggested by the absolute values in the preceding paragraph, the values in the block-start array can be negative. If `bs(i+1)` is negative, the `i`th block is assumed to be an identity matrix, and its application is skipped in forming products.[1]

The blocks M_j in M are uncoupled — that is they represent unrelated eigenvalue problems. To couple them we perform a final similarity transformation, to get our matrix A. Specifically,

$$A = YMY^{-1} = YZLZ^{-1}Y^{-1},$$

where Y is an hsvdmat of order n having only one block. The singular values of Y can be chosen to increase the condition number of Y, thus providing a second source of sensitivity in the eigendecomposition. The matrix Y can be stored in the same way as Z, but with `nblocks` equal one.

Finally, the worst-case operation count for applying A is $41n$ floating-point operations.

3. Eigenvectors, condition numbers, and accuracy

An eigenmat has known eigenvalues. Its eigenvectors are not represented explicitly. In fact, for very large n storage limitations would prohibit any explicit representation of all the eigenvectors, since the eigenvectors are not in general sparse. Nonetheless, individual eigenvectors can be calculated.

Specifically, let

$$A = YZLZ^{-1}Y^{-1} \equiv XLX^{-1} \tag{3.1}$$

Now suppose that `type(j)` is one; i.e., the jth eigenvalue is real. Then the jth column x_j of X is the corresponding eigenvector. On the other hand, if `type(j)` is two, so that we have a complex conjugate pair of eigenvalues, then $x_j \pm ix_{j+1}$ are the corresponding eigenvectors.

Thus to compute eigenvectors, we must to be able to compute columns of X. This can be done as follows. Let e_j be the vector whose jth component is 1 and whose other components are 0. Then the jth column of X is

$$Xe_j = YZe_j.$$

[1]The reason that `bs(i+1)`, rather than `bs(i)` determines the status of block `i`, is that in C `bs[0]` is always zero and cannot be negative.

Thus the component can be computed by multiplying e_j by the two hsvdmats Z and Y — an $O(n)$ process. The left eigenvectors are the columns of $X^{-\mathrm{T}} = Y^{-\mathrm{T}}Z^{-\mathrm{T}}$, and can be computed similarly.

A function is ill-conditioned if it is very sensitive to perturbations in its arguments. The degree of ill-conditioning is usually quantified by a condition number. For example, let A have a simple eigenvalue λ with right and left eigenvectors r and s. Then for sufficiently small E, there is a unique eigenvalue $\tilde{\lambda}$ of $A + E$, satisfying [1, §7.2.2]

$$|\tilde{\lambda} - \lambda| \leq \sec\angle(r,s)\|E\| + O(\|E\|^2).$$

Thus

$$\sec\angle(r,s) = \frac{\|r\|\|s\|}{|r^{\mathrm{T}}s|} \tag{3.2}$$

is a condition number for the eigenvalue.

In the preceding section we have argued that we can use the matrices Y and Z to increase the ill-conditioning of the eigenvalues of A. But the argument does not give the condition numbers of the individual eigenvalues. However, if we have computed the left and right eigenvectors corresponding to an eigenvalue, we can compute its condition number from (3.2). Eigentest provides procedures for computing left and right eigenvectors and the condition numbers of the corresponding eigenvalues.

The condition number for an eigenvector is expensive to compute. However we have the following useful result. Let A be as in (3.1) and let λ be a simple eigenvalue of A. Let δ be the absolute distance between λ and its nearest neighbor. Then if $\gamma(x)$ denote the condition number of the eigenvector x corresponding to λ,

$$\frac{1}{\delta} \leq \gamma(x) \leq \frac{\kappa(X)}{\delta}. \tag{3.3}$$

In our application, the condition number of X can be bounded by $\kappa(Y)\kappa(Z)$. Moreover, $\kappa(Y) = \sigma_{\max}(Y)/\sigma_{\min}(Y)$, where $\sigma_{\max}(Y)$ is the largest singular value of Y and $\sigma_{\min(Y)}$ is the smallest singular value. This shows that we can make an eigenvector ill-conditioned by making its eigenvalue poorly separated from the others. It also suggests (but does not prove) that we can add ill-conditioning by making X ill-conditioned (for more see [3]).

Symmetric and normal eigenmats can be generated by setting the singular values of the hsvdmats `Y` and `Z` to one. In this case the condition number for an eigenvector is δ^{-1}, where δ is as in (3.3).

As we have seen, Eigentest evaluates the product Ab by successively multiplying by the factors of A. This is not the same as traditional matrix multiplication (or solution of linear systems in the case of shift-and-invert). It is therefore necessary to inquire into the accuracy of the computed products.

The basic result that we shall use in our inquiry is the following. Let $S = PQR$, and suppose that $y = Sx$ is computed in floating-point arithmetic with rounding unit ϵ_{M} by successive multiplication of the factors — i.e., $\tilde{y} = P(Q(Rx))$. Then $\tilde{y} = (S + E)x$, where

$$\|E\| \le \tilde{\gamma}\|P\|\|Q\|\|R\|\epsilon_{\mathrm{M}}.$$

Here $\tilde{\gamma}$ is a constant that depends on the dimensions of P, Q, and R. This result is easily established by an elementary rounding error analysis (for the basic step in the analysis, see [2, §3.5]).

This result is a backward rounding-error result. From it we can derive the bound

$$\|\tilde{y} - y\| \le \tilde{\gamma}\|P\|\|Q\|\|R\|\|x\|\epsilon_{\mathrm{M}}. \tag{3.4}$$

Now if we know S and compute $\hat{y} = Sx$ in the usual way, the bound is

$$\|\hat{y} - y\| \le \hat{\gamma}\|S\|\|x\|\epsilon_{\mathrm{M}}. \tag{3.5}$$

Since

$$\|S\| \le \|P\|\|Q\|\|R\|,$$

the bound (3.4) is potentially greater than (3.5). The bounds become more nearly equal in proportion as $\|P\|\|Q\|\|R\|$ is near $\|S\|$; i.e., in proportion as the product of the norms of the factors is near the norm of the product.

In applying this result, we first note that a hsvdmat is the product of two orthogonal matrices, whose norms are one, and a diagonal matrix. Hence the norm of the product is the product of the norms, which means that the products of hsvdmats in the factorization of A are computed to their limiting accuracy. Thus we can consider the factorization $A = XDX^{-1}$, where D is Λ, possibly shifted and inverted.

Unfortunately, for this factorization the product of the norms can be considerably greater than the norm of the products, as the following 2×2 example shows. Let

$$A = \begin{pmatrix} 1+\epsilon & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & -\epsilon \end{pmatrix} \begin{pmatrix} 1+\epsilon & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \epsilon^{-1} \\ 0 & -\epsilon^{-1} \end{pmatrix} \equiv XDX^{-1}. \tag{3.6}$$

For this matrix we have $\|A\| = \mathrm{O}(1)$, while $\|X\|\|D\|\|X^{-1}\| = O(\epsilon^{-1})$. As $\epsilon \to 0$, we can expect the factored product to become increasingly inaccurate. In fact, when $\epsilon \le \epsilon_{\mathrm{M}}$, we can expect no accuracy at all in the first component of the product.

The matrix A defined by (3.6) is a perturbation of a Jordan block of order two. Jordan blocks have only one eigenvector and cannot be written in the form $X\Lambda X^{-1}$. It should therefore come as no surprise that problems would emerge when we attempt to work with the such a decomposition when the matrix is near a Jordan block.

On should not make too much of this example. Without special contrivance the product of the norms will be near the norm of the product. The problem only occurs when we apply an ill-conditioned transformation to a block of poorly separated eigenvalues. The cure is to work directly with a perturbation of a Jordan block. This is a strong reason for including perturbed Jordan blocks in a later version of Eigentest.

4. Structures and operations

In this section we will define the data structures informally described in the previous section along with the subprograms that perform operations with eigenmats. Since C, Fortran 95, and Matlab all have structured types, we will describe the C implementation in detail, and then sketch the implementations for Matlab and Fortran 95. We will then describe how the basic data structures are transcribed into arrays in Fortran 77, which does not have structure types.

In all the variants of Eigentest, an error causes the run to be terminated by an error message. The rationale is that the errors in Eigentest are all fatal, from which the only recovery is to change the input. In fact, the errors are either a misuse of the `type` array or an illegal operation.

4.1. Eigentest: C

The structure representing the matrix A is the following.

```
struct eigenmat{
   int n;            /* The order of the matrix */
   double *eig;      /* Array containing the eigenvalues of A */
   int    *type;     /* type[i] is
                           1 if eig[i] is a real eigenvalue
                           2 if eig[i] is the real part of an
                             eigenvalue
                           3 if eig[i] is the imaginary part of an
                             eigenvalue */
   struct hsvd Y, Z; /* The outer and inner hsvdmats */
};
```

The entries are self-explanatory, with the exception of the structure `hsvd`, which is shown below.

```
struct hsvd{
   int n;           /* The order of the matrix */
   int nblocks;     /* The number of blocks in Lambda */
   int *bs;         /* abs(bs[i]) is the index of the start of the
                       i-th block.  abs(bs[nblock])=n.
                       if bs(i+1)<0, the i-th block
                       is an identity. */
   double *u;       /* The vector generating the left Householder
                       transformations. */
   double *v;       /* The vector generating the right Householder
                       transformations. */
   double *sig;     /* The singular values */
};
```

To initialize these structures Eigentest provides the following routine.

```
void EigenmatAlloc(struct eigenmat *A, int n, int nblocks,
                   int yident, int zident)
```

This routine allocates storage for the arrays of the eigenmat `A`. It is the user's responsibility to fill in values in these arrays. If `yindent` is nonzero, the hsvdmat `Y` is set up as an identity matrix with `nblocks`=1 and `bs[1]=-1`. In this case, no storage is allocated for `u`, `v`, or `sig`. The parameter `zident` performs the same function for `Z`.

The function

```
void EigenmatFree(struct eigenmat *A)
```

deallocates the storage. Note that for both `Y` and `Z` if `nblocks=1` and `bs[1]<0`, no storage is deallocated for `u`, `v`, or `sig`.

Eigentest provides a routine to compute the product of an eigenmat A and a matrix B with an optional shift. Its calling sequence is

```
void EigenmatProd(struct eigenmat *A, int ncols,
                  double *B, int tdb,
                  double *C, int tdc,                    (4.1)
                  double shift, char *job)
```

Here `B` points to a two dimensional array containing and `n`×`m` matrix to be multiplied by the eigenmat `A`. The result is returned in the array `C`. The integers `tdb` and `tdc` are the trailing dimensions of the arrays `B` and `C`. The string `job` specifies the operation to be performed:

job	operation
ab	$C = (A - \mathtt{shift}*I)B$
atb	$C = (A - \mathtt{shift}*I)^{T}B$
aib	$C = (A - \mathtt{shift}*I)^{-1}B$
aitb	$C = (A - \mathtt{shift}*I)^{-T}B$

(4.2)

This function makes use of the following function to manipulate `hsvd` matrices.

```
void HsvdProd(struct hsvd *X, int ncols,
              double *B, int tdb, char *job)
```
(4.3)

There are two differences between this function and `EigenmatProd`. First, there is no shift; otherwise the operations are as in the table (4.2). Second, the routine overwrites `B` with the product, so there is no matrix `C`.

Finally, Eigentest includes a routine to compute eigenvectors. Its calling sequence is

```
void EigenmatVecs(struct eigenmat *A, int eignum,
                  double *eigr, double *eigi,
                  double *xr, double *xi,
                  double *yr, double *yi,
                  double *cond, char job)
```

`EigenmatVecs` computes an eigenvalue `eig[eignum]` of the eigenmat `A`, and the corresponding left or right eigenvectors If both are computed, EigenmatVecs also returns the condition number (`cond`) of the eigenvalue. The eigenvectors are scaled to have Euclidean norm 1. The string `job` specifies what is to be computed as follows.

`job`	returned
`r`	right eigenvector `(xr, xi)`
`l`	left eigenvector `(yr, yi)`
`b`	both eigenvectors and `cond`

Since C has no complex type, the real and imaginary parts of the eigenvectors must be returned in separate arrays as indicated above. Similarly, the eigenvalue is returned in `eigr` and `eigi`.

4.2. Eigentest: Fortran 95

The defined types and procedures for the Fortran 95 version of Eigentest differ minimally from those for the C version. The components of the Fortran 95 types have the same names as the members of the C structures. The main difference is that array indexing begins at 1 in Fortran 95 instead of 0 at C. In particular, for any hsvdmat, `abs(bs[1])` is `1` and `abs(bs[nblocks+1])` is `n+1`. The precision of the computations is controlled by the parameter `wp` (for working precision), which is defined as

```
integer, parameter :: wp = kind(0.0d0)
```

The subroutines to allocate and free storage for a Fortran 95 eigenmat are

```
subroutine EigenmatAlloc(A, n, nblocks, yident, zident)
   type(eigenmat), intent(inout) :: A
   integer, intent(in)           :: n
   integer, intent(in)           :: nblocks
   logical, intent(in)           :: yident, zident
```

and

```
subroutine EigenmatFree(A)
   type(Eigenmat), intent(inout) :: A
```

The following subroutine computes the product of an eigenmat with a matrix.

```
subroutine EigenmatProd(A, ncols, B, C, shift, job)
   type(Eigenmat), intent(in) :: A
   integer, intent(in)        :: ncols
   real(wp), intent(in)       :: B(:,:)
   real(wp), intent(inout)    :: C(:,:)
   real(wp), intent(in)       :: shift
   character(*), intent(in)   :: job
```

The subroutine `eigenmatprod` uses `HsvdProd` to compute a the product of an hsvdmat and a matrix.

```
subroutine HsvdProd(X, B, ncols, job)
   type(hsvdmat), intent(in) :: X
   real(wp), intent(inout)   :: B(:,:)
   integer, intent(in)       :: ncols
   character(*), intent(in)  :: job
```

These four subroutines are analogous to their C counterparts. The main difference is that the trailing dimensions `tdb` and `tdc`, which Fortran 95 does not need, have disappeared from the calling sequence.

Finally, the routine to compute eigenvectors has essentially the same calling sequence as the corresponding C routine.

```
subroutine EigenmatVecs(A, eignum, eig, x, y, cond, job)
   type(Eigenmat), intent(in) :: A
   integer, intent(in)        :: eignum
   complex(wp), intent(out)   :: eig
   complex(wp), intent(inout) :: x(:)
   complex(wp), intent(inout) :: y(:)
   real(wp), intent(out)      :: cond
   character, intent(in)      :: job
```

The chief difference is that the eigenvalue and the eigenvectors are returned as type complex.

4.3. Eigentest: Matlab

The structures for the Matlab eigenmat and hsvdmat are analogous to the ones for C and Fortran 95, and there is no need to elaborate further. The raw structure for the Eigenmat is

```
struct('n', [], 'eig', [], 'type', [], 'Z', Z, 'Y', Y)
```

and for the hsvdmat

```
struct('n', [], 'nblocks', [], 'bs', [], ...
                 'sig', [], 'u', [], 'v', []);
```

The routine to initialize an eigenmat is called `EigenmatGen` (not `EigenmatAlloc`) because it actually generates the structure for the eigenmat:

```
function A = EigenmatGen(n, yident, zident)
```

It performs the same minimal initialization as `EigenmatAlloc`; but it allocates no storage, since Matlab itself handles storage management.

The functions

```
function C = EigenmatProd(A, B, shift, job)
```

and

```
function B = HsvdProd(X, B, job)
```

are the same as their counterparts for C and Fortran 95, except they include no information on the dimensions of matrices and arrays, since these are not needed in Matlab.

Finally, the routine for computing vectors has the form

```
function [eig, x, y, cond] = EigenmatVecs(A, eignum, job)
```

The input and output arguments are the same as in the Fortran 95 version.

```
*       iem(1)        nmax  The maximum size of the matrix.
*       iem(2)        n     The order of the matrix.
*       iem(3)        type  The beginning of an array
*                           containing the types of the
*                           eigenvalues in eig.  If type(i)
*                           is 1, the corresponding entry of
*                           eig is a real eigenvalue.  If
*                           type(i) is 2 and type(i+1) is 3
*                           then the corresponding entries of
*                           eig contain mu and nu as above.
*       iem(nmax+3)         The begining of the integer
*                           array for the hsvdmat Y.
*       iem(nmax+9)         The beginning of the integer array
*                           for the hsvdmat Z.
*
*       fem(1)        eig   The array containing the eigenvalues
*                           of the matrix.
*       fem(nmax+1)         The beginning of the double array
*                           for the hsvdmat Y.
*       fem(4*nmax+1)       The beginning of the double array
*                           for the hsvdmat Z.
```

Figure 4.1: Fortran 77 representation of an eigenmat

4.4. Eigentest: Fortran 77

The principal difficulty in implementing Eigentest in Fortran 77 is the fact that the language has no equivalent of structures or derived types. Of the possible solutions to this problem, we have chosen to represent the entire eigenmat and its hsvdmats in two arrays — `iem`, an integer array, and `fem`, a floating-point array. A special subroutine `eminit.f` is provided to help the user initialize these arrays. (Because Fortran 77 limits identifiers to six characters, the names in Eigentest F77 are necessarily less informative than those in the other parts of the package. For example, `em` represents `eigenmat`.)

Figure 4.1 contains the arrangement of the components of an eigenmat in the `iem` and `fem` arrays. For the most part, it represents a straightforward transcription of the `eigenmat` structure described above. However, two points are to be noted.

First, since Fortran 77 does not have dynamic storage allocation, the maximum order of the eigenmats in question must be specified at compile time. This maximum is represented by `nmax` in the figure. Note that the length of the vector components, e.g.,

```
*       iem(1)          nmax   maximum value of n
*       iem(2)          n      The order of the hsvd
*       iem(3)          nblk   The number of blocks in the hsvd
*       iem(4)          bs     Beginning of the block-start array.
*                              bs(5)=1 and bs(i+1)-bs(i) is the size
*                              of the ith block.
*
*       fem(1)          u      start of u
*       fem(1+nmax)     v      start of v
*       fem(1+2*nmax)   sig    start of the singular values
```

Figure 4.2: Fortran77 representation of an hsvdmat

`type` in `iem` is `nmax` not `n`, so that when `n` is smaller that `nmax`, there is internal fragmentation in the eigenmat structure. This represents no inefficiency, since the storage must already be allocated; and it is simpler, since the positions of the components are independent of the size of the matrix.

Second, the structures of `Y` and `Z` are not contained in separate arrays, but are embedded in the arrays `iem` and `fem`. If they were represented separately, then `emprod` (the counterpart of `eigenmatprod`) would have six arguments (two extra arrays each for `Y` and `Z`).

Figure 4.2 contains the arrangement of the components of an hsvdmat. Once again, the components of this structure should be clear. The indexing of the components in the arrays `iem` and `fem` are the ones used in the subroutine `hsvdpr` to compute products. We will see how these indices link up with those in Figure 4.1 in a moment.

The routine for computing the product of an eigenmat and a matrix has the following calling sequence.

```
      subroutine emprod(iem, fem, ncols, B, ldb,
     &                  C, ldc, shift, op)
```

The last seven arguments are as in (4.1). The arrays `iem` and `fem` take the place of `eigenmat *A`.

The routine for computing the product of an hsvdmat and a matrix has the calling sequence

```
      call hsvdpr(iem, fem, ncols, B, ldb, op)
```

The last four arguments are the same as in (4.3). The arrays `iem` and `fem` take the place of `*X`. If one wants to work with the hsvdmat `Y` in the structure in Figure 4.1, one simply writes

```
          call hsvdpr(iem(nmax+3), fem(nmax+1), ... )
```

To work with Z, write

```
          call hsvdpr(iem(nmax+9), fem(4*nmax+1), ... )
```

The initialization of the arrays `iem` and `fem` is likely to be an error-prone procedure. To aid the programmer, Eigentest provides an initialization routine to pack these arrays. To start with, the programmer must declare the arrays `iem` and `fem`:

```
          integer iem(<2*nmax+13>)
          double precision fem(<7*nmax>)
```

The angular brackets indicate that these are minimal dimensions. The subroutine `eminit` has the form

```
          subroutine eminit(nmax, n, iem, fem,  nblk, idy, idz,
         $                  ia, fa, job)
          integer nmax, n, iem(*), nblk, idy, idz, ia(<nmax+1>)
          double precision fem(*), fa(<nmax>)
          character job*(*)
```

where again the angular brackets represent minimal quantities. (The mysterious `1` in `nmax+1` comes from the fact that the block-start array in an hsvdmat may be as large as `nmax+1`.

The calling sequence of `eminit` has two arrays `ia` and `fa` which are transferred to the appropriate positions in `iem` and `fem` under control of the parameter `job`. The following table describes options.

`job`	Action
`setup`	load `nmax`, `n` into their several positions in `iem`. Set up Y as having one block. Set up Y and Z as identity matrices if `idy` and `idz` are nonzero. `ia`.
`yu`	load `u` for `y` from `fa`.
`yv`	load `v` for `y` from `fa`.
`ysig`	load `sig` for `y` from `fa`.
`zu`	load `u` for `z` from `fa`.
`zv`	load `v` for `z` from `fa`.
`zsig`	load `sig` for `z` from `fa`.
`eig`	load `eig` and `type` from `fa` and `ia`.

These jobs can be performed in any order provided `nmax` and `n` remain the same throughout.

The routine for computing eigenvectors is

```
subroutine emvecs(iem, fem, eignum, eig, x, y, cond, job)
integer          iem(*), eignum
double precision fem(*), cond
complex*16       eig, x(*), y(*)
character        job
```

The parameters are as in the Fortran 95 version.

5. Miscellany

The several versions of Eigentest are not large programs, and with the exception of the Matlab version they each come in a single file ready for compilation.[2] The distribution also includes a source file `testeigentest` that runs 32 test cases probing various aspects of the package. The numbers in the output should be within two or so orders of magnitude of the rounding unit. The code of `testeigentest` can also serve as a template for setting up and eigenmat in its particular version.

We have also used Eigentest in the development of RKPACK, a Krylov based eigensolver that is insensitive to errors in the underlying Krylov sequence. Space considerations preclude the inclusion of examples from this application.

6. Acknowledgement

One of the authors (Stewart) would like to thank the Mathematical and Computational Sciences Division of the National Institute of Standards and Technology for the use of their facilities during the development of this project.

References

[1] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

[2] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, 1996.

[3] G. W. Stewart. Error and perturbation bounds for subspaces associated with certain eigenvalue problems. *SIAM Review*, 15:727–764, 1973.

[2]Matlab is the exception because each Matlab function must reside in its own m-file.